Vorlesungsskript

Graphalgorithmen

Sommersemester 2021

Prof. Dr. Johannes Köbler

Humboldt-Universität zu Berlin
Lehrstuhl Komplexität und Kryptografie

27. Juni 2021

Inhaltsverzeichnis

1 Graphentheoretische Grundlagen

Definition 1.1. *Ein **(ungerichteter) Graph** ist ein Paar $G = (V, E)$, wobei*

V *- eine endliche Menge von **Knoten/Ecken** und*

E *- die Menge der **Kanten** ist.*

Hierbei gilt

$$E \subseteq \binom{V}{2} = \Big\{\{u, v\} \subseteq V \mid u \neq v\Big\}.$$

Sei $v \in V$ ein Knoten.

*a) Die **Nachbarschaft von v** ist $\boldsymbol{N_G(v)} = \{u \in V \mid \{u, v\} \in E\}$.*

*b) Der **Grad von v** ist $\boldsymbol{\deg_G(v)} = |N_G(v)|$.*

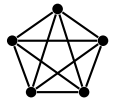
*c) Der **Minimalgrad von G** ist $\boldsymbol{\delta(G)} = \min_{v \in V} \deg_G(v)$ und der **Maximalgrad von G** ist $\boldsymbol{\Delta(G)} = \max_{v \in V} \deg_G(v)$.*

*d) Jeder Knoten $u \in V$ vom Grad ≤ 1 heißt **Blatt** und die übrigen Knoten (vom Grad ≥ 2) heißen **innere Knoten** von G.*

Falls G aus dem Kontext ersichtlich ist, schreiben wir auch einfach $N(v)$, $\deg(v)$, δ usw.

Beispiel 1.2.

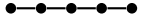
- *Der **vollständige Graph** (V, E) auf n Knoten, d.h. $|V| = n$ und $E = \binom{V}{2}$, wird mit $\boldsymbol{K_n}$ und der **leere Graph** $(V, \emptyset)$ auf n Knoten wird mit $\boldsymbol{E_n}$ bezeichnet.*

K_1:  K_2: K_3:  K_4:  K_5: 

- *Der **vollständige bipartite Graph** (A, B, E) auf $a + b$ Knoten, d.h. $A \cap B = \emptyset$, $|A| = a$, $|B| = b$ und $E = \{\{u, v\} \mid u \in A, v \in B\}$ wird mit $\boldsymbol{K_{a,b}}$ bezeichnet.*

$K_{1,1}$:  $K_{1,2}$:  $K_{2,2}$:  $K_{2,3}$:  $K_{3,3}$: 

- *Der **Pfad** mit n Knoten wird mit $\boldsymbol{P_n}$ bezeichnet.*

P_2:  P_3:  P_4:  P_5: 

- *Der **Kreis** mit n Knoten wird mit $\boldsymbol{C_n}$ bezeichnet.*

C_3:  C_4:  C_5:  C_6: 

Definition 1.3. *Sei $G = (V, E)$ ein Graph.*

*a) Eine Knotenmenge $U \subseteq V$ heißt **unabhängig** oder **stabil**, wenn es keine Kante von G mit beiden Endpunkten in U gibt, d.h. es gilt $E \cap \binom{U}{2} = \emptyset$. Die **Stabilitätszahl** ist*

$$\boldsymbol{\alpha(G)} = \max\{|U| \mid U \text{ ist stabile Menge in } G\}.$$

*b) Eine Knotenmenge $U \subseteq V$ heißt **Clique**, wenn jede Kante mit beiden Endpunkten in U in E ist, d.h. es gilt $\binom{U}{2} \subseteq E$. Die **Cliquenzahl** ist*

$$\boldsymbol{\omega(G)} = \max\{|U| \mid U \text{ ist Clique in } G\}.$$

*c) Ein Graph $G' = (V', E')$ heißt **Sub-/Teil-/Untergraph** von G, falls $V' \subseteq V$ und $E' \subseteq E$ ist. Im Fall $V' = V$ wird G' auch ein **(auf)spannender Teilgraph** von G genannt und wir schreiben für G' auch $\boldsymbol{G - E''}$ (bzw. $G = \boldsymbol{G' \cup E''}$), wobei $E'' = E - E'$ die Menge der aus G entfernten Kanten ist. Im Fall $E'' = \{e\}$ schreiben wir für G' auch einfach $\boldsymbol{G - e}$ (bzw. $G = \boldsymbol{G' \cup e}$).*

d) *Ein k-regulärer spannender Teilgraph von G wird auch als* ***k-Faktor*** *von G bezeichnet. Ein d-regulärer Graph G heißt* ***k-faktorisierbar****, wenn sich G in $l = d/k$ kantendisjunkte k-Faktoren $G_1, \ldots, G_l$ zerlegen lässt.*

e) *Ein Subgraph $G' = (V', E')$ heißt (****durch V'****) **induziert**, falls $E' = E \cap \binom{V'}{2}$ ist. Für G' schreiben wir dann auch $G[V']$ oder $G - V''$, wobei $V'' = V - V'$ die Menge der aus G entfernten Knoten ist. Ist $V'' = \{v\}$, so schreiben wir für G' auch einfach $G - v$ und im Fall $V' = \{v_1, \ldots, v_k\}$ auch $G[v_1, \ldots, v_k]$.*

f) *Ein* ***Weg*** *ist eine Folge von (nicht notwendig verschiedenen) Knoten $v_0, \ldots, v_\ell$ mit $\{v_i, v_{i+1}\} \in E$ für $i = 0, \ldots, \ell - 1$. Die* ***Länge*** *des Weges ist die Anzahl der durchlaufenen Kanten, also ℓ. Im Fall $\ell = 0$ heißt der Weg* ***trivial****. Ein Weg $(v_0, \ldots, v_\ell)$ heißt auch* ***v_0-v_ℓ-Weg****.*

g) *G heißt* ***zusammenhängend****, falls es für alle Paare $\{u, v\} \in \binom{V}{2}$ einen u-v-Weg gibt.*

h) *Die durch die Äquivalenzklassen $V_i \subseteq V$ der Relation*

$$Z = \{(u, v) \in V \times V \mid \text{ es gibt in } G \text{ einen } u\text{-}v\text{-Weg}\}$$

induzierten Teilgraphen $G[V_i]$ heißen ***Zusammenhangskomponenten*** *(engl.* ***connected components****) oder einfach* ***Komponenten*** *von G.*

i) *Ein u-v-Weg heißt* ***einfach*** *oder* ***u-v-Pfad****, falls alle durchlaufenen Knoten verschieden sind.*

j) *Ein* ***Zyklus*** *ist ein u-v-Weg mit $u = v$.*

k) *Eine Menge von Pfaden heißt* ***disjunkt****, wenn je zwei Pfade in der Menge keine gemeinsamen Knoten haben,* ***kantendisjunkt****, wenn je zwei Pfade in der Menge keine gemeinsamen Kanten haben, und* ***knotendisjunkt****, wenn je zwei Pfade in der Menge höchstens gemeinsame Endpunkte haben.*

l) *Ein* ***Kreis*** *ist ein Zyklus $(v_1 \ldots, v_\ell, v_1)$ der Länge $\ell \geq 3$, für den $v_1, \ldots, v_\ell$ paarweise verschieden sind.*

m) *Ein Graph heißt* ***kreisfrei****,* ***azyklisch*** *oder* ***Wald****, falls er keinen Kreis enthält. Ein* ***Baum*** *ist ein zusammenhängender Wald.*

Definition 1.4. *Ein* ***gerichteter Graph*** *oder* ***Digraph*** *ist ein Paar $G = (V, E)$, wobei*

V - eine endliche Menge von ***Knoten/Ecken*** *und*

E - die Menge der ***Kanten*** *ist.*

Hierbei gilt

$$E \subseteq V \times V = \Big\{(u, v) \mid u, v \in V\Big\},$$

wobei E auch Schlingen (u, u) enthalten kann. Sei $v \in V$ ein Knoten.

a) *Die* ***Nachfolgermenge von v*** *ist $N^+(v) = \{u \in V \mid (v, u) \in E\}$.*

b) *Die* ***Vorgängermenge von v*** *ist $N^-(v) = \{u \in V \mid (u, v) \in E\}$.*

c) *Die* ***Nachbarmenge von v*** *ist $N(v) = N^+(v) \cup N^-(v)$.*

d) *Der* ***Ausgangsgrad von v*** *ist $\mathbf{deg^+(v)} = |N^+(v)|$ und der* ***Eingangsgrad von v*** *ist $\mathbf{deg^-(v)} = |N^-(v)|$. Der* ***Grad von v*** *ist $\mathbf{deg(v)} = \deg^+(v) + \deg^-(v)$.*

e) *Ein* ***(gerichteter) v_0-v_ℓ-Weg*** *ist eine Folge von Knoten $v_0, \ldots, v_\ell$ mit $(v_i, v_{i+1}) \in E$ für $i = 0, \ldots, \ell - 1$.*

f) *Ein* ***(gerichteter) Zyklus*** *ist ein gerichteter u-v-Weg mit $u = v$.*

g) *Ein gerichteter Weg heißt* ***einfach*** *oder* ***(gerichteter) Pfad****, falls alle durchlaufenen Knoten verschieden sind.*

h) *Ein* ***(gerichteter) Kreis*** *in G ist ein gerichteter Zyklus $(v_1 \ldots, v_\ell, v_1)$ der Länge $\ell \geq 1$, für den $v_1, \ldots, v_\ell$ paarweise verschieden sind.*

i) *G heißt* ***kreisfrei*** *oder* ***azyklisch****, wenn es in G keinen gerichteten Kreis gibt.*

j) *G heißt* ***stark zusammenhängend****, wenn es in G für jedes Knotenpaar $u \neq v \in V$ sowohl einen u-v-Pfad als auch einen v-u-Pfad gibt.*

k) *G heißt* ***gerichteter Wald****, wenn G kreisfrei ist und jeder Knoten $v \in V$ Eingangsgrad $\deg^-(v) \leq 1$ hat.*

*l) Ein Knoten $w \in V$ vom Eingangsgrad $\deg^-(w) = 0$ heißt **Wurzel** von G, und ein Knoten $u \in V$ vom Ausgangsgrad $\deg^+(u) = 0$ heißt **Blatt** von G.*

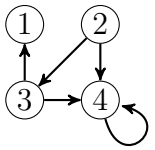
Die ***Adjazenzmatrix*** eines Graphen bzw. Digraphen $G = (V, E)$ mit (geordneter) Knotenmenge $V = \{v_1, \ldots, v_n\}$ ist die $(n \times n)$-Matrix $A = (a_{ij})$ mit den Einträgen

$$a_{ij} = \begin{cases} 1, & \{v_i, v_j\} \in E \\ 0, & \text{sonst} \end{cases} \qquad \text{bzw.} \qquad a_{ij} = \begin{cases} 1, & (v_i, v_j) \in E \\ 0, & \text{sonst.} \end{cases}$$

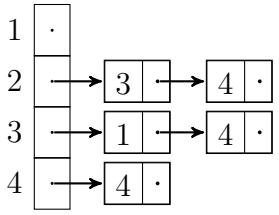
Für ungerichtete Graphen ist die Adjazenzmatrix symmetrisch mit $a_{ii} = 0$ für $i = 1, \ldots, n$.

Bei der ***Adjazenzlisten-Darstellung*** wird für jeden Knoten v_i eine Liste mit seinen Nachbarn verwaltet. Im gerichteten Fall verwaltet man entweder nur die Liste der Nachfolger oder zusätzlich eine weitere für die Vorgänger. Falls die Anzahl der Knoten statisch ist, organisiert man die Adjazenzlisten in einem Feld, d.h. das Feldelement mit Index i verweist auf die Adjazenzliste von Knoten v_i. Falls sich die Anzahl der Knoten dynamisch ändert, so werden die Adjazenzlisten typischerweise ebenfalls in einer doppelt verketteten Liste verwaltet.

Beispiel 1.5.
Betrachte den gerichteten Graphen $G = (V, E)$ mit $V = \{1, 2, 3, 4\}$ und $E = \{(2, 3), (2, 4), (3, 1), (3, 4), (4, 4)\}$. Dieser hat folgende Adjazenzmatrix- und Adjazenzlisten-Darstellung:



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



◁

2 Färben von Graphen

Definition 2.1. *Sei $G = (V, E)$ ein Graph und sei $k \in \mathbb{N}$.*

*a) Eine Abbildung $f\colon V \to \mathbb{N}$ heißt **Färbung** von G, wenn $f(u) \neq f(v)$ für alle $\{u, v\} \in E$ gilt.*

*b) G heißt **k-färbbar**, falls eine Färbung $f\colon V \to \{1, \ldots, k\}$ existiert. Eine solche Färbung bezeichnen wir als **k-Färbung**.*

*c) Die **chromatische Zahl** ist*

$$\boldsymbol{\chi(G)} = \min\{k \in \mathbb{N} \mid G \text{ ist } k\text{-färbbar}\}.$$

Beispiel 2.2.

$$\chi(E_n) = 1,\ \chi(K_{n,m}) = 2,\ \chi(K_n) = n,$$

$$\chi(C_n) = \begin{cases} 2, & n \text{ gerade} \\ 3, & \text{sonst.} \end{cases}$$

Ein wichtiges Entscheidungsproblem ist, ob ein gegebener Graph k-färbbar ist. Dieses Problem ist für jedes feste $k \geq 3$ schwierig.

k-Färbbarkeit (k-Coloring):

Gegeben: Ein Graph G.
Gefragt: Ist G k-färbbar?

Satz 2.3. *k-Coloring ist für $k \geq 3$ NP-vollständig.*

Das folgende Lemma setzt die chromatische Zahl $\chi(G)$ in Beziehung zur Stabilitätszahl $\alpha(G)$.

Lemma 2.4. $n/\alpha(G) \leq \chi(G) \leq n - \alpha(G) + 1$.

Beweis. Sei G ein Graph und sei c eine $\chi(G)$-Färbung von G. Da dann die Mengen $S_i = \{u \in V \mid c(u) = i\}$, $i = 1, \ldots, \chi(G)$, stabil sind, folgt $|S_i| \leq \alpha(G)$ und somit gilt

$$n \;=\; \sum_{i=1}^{\chi(G)} |S_i| \;\leq\; \chi(G)\alpha(G).$$

Für den Beweis von $\chi(G) \leq n - \alpha(G) + 1$ sei S eine stabile Menge in G mit $|S| = \alpha(G)$. Dann ist $G - S$ k-färbbar für ein $k \leq n - |S|$. Da wir alle Knoten in S mit der Farbe $k + 1$ färben können, folgt $\chi(G) \leq k + 1 \leq n - \alpha(G) + 1$. ■

Beide Abschätzungen sind scharf, können andererseits aber auch beliebig schlecht werden.

Lemma 2.5. $\binom{\chi(G)}{2} \leq m$ *und somit* $\chi(G) \leq \frac{1}{2} + \sqrt{2m + \frac{1}{4}}$.

Beweis. Zwischen je zwei Farbklassen einer optimalen Färbung muss es mindestens eine Kante geben. ■

Die chromatische Zahl steht auch in Beziehung zur Cliquenzahl $\omega(G)$ und zum Maximalgrad $\Delta(G)$:

Lemma 2.6. $\omega(G) \leq \chi(G) \leq \Delta(G) + 1$.

Beweis. Die erste Ungleichung folgt daraus, dass die Knoten einer maximal großen Clique unterschiedliche Farben erhalten müssen.

Um die zweite Ungleichung zu erhalten, betrachten wir folgenden Färbungsalgorithmus:

Algorithmus `greedy-color`

```
1  input ein Graph G = (V, E) mit V = {v_1, ..., v_n}
2  c(v_1) := 1
3  for i := 2 to n do
4     F_i := {c(v_j) | j < i, v_j ∈ N(v_i)}
5     c(v_i) := min{k ≥ 1 | k ∉ F_i}
```


Da für die Farbe $c(v_i)$ von v_i nur $|F_i| \leq \Delta(G)$ Farben verboten sind, gilt $c(v_i) \leq \Delta(G) + 1$. ■

2.1 Färben von planaren Graphen

Ein Graph G heißt ***planar***, wenn er so in die Ebene einbettbar ist, dass sich zwei verschiedene Kanten höchstens in ihren Endpunkten berühren. Dabei werden die Knoten von G als Punkte und die Kanten von G als Verbindungslinien (genauer: Jordankurven) zwischen den zugehörigen Endpunkten dargestellt.

Bereits im 19. Jahrhundert wurde die Frage aufgeworfen, wie viele Farben höchstens benötigt werden, um eine Landkarte so zu färben, dass aneinander grenzende Länder unterschiedliche Farben erhalten. Offensichtlich lässt sich eine Landkarte in einen planaren Graphen transformieren, indem man für jedes Land einen Knoten zeichnet und benachbarte Länder durch eine Kante verbindet. Länder, die sich nur in einem Punkt berühren, gelten dabei nicht als benachbart.

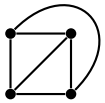
Die Vermutung, dass 4 Farben ausreichen, wurde 1878 von Kempe „bewiesen“ und erst 1890 entdeckte Heawood einen Fehler in Kempes „Beweis“. Übrig blieb der ***5-Farben-Satz***. Der ***4-Farben-Satz*** wurde erst 1976 von Appel und Haken bewiesen. Hierbei handelt es sich jedoch nicht um einen Beweis im klassischen Sinne, da zur Überprüfung der vielen auftretenden Spezialfälle Computer benötigt werden.

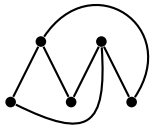
Satz 2.7 (Appel, Haken 1976)**.**
Jeder planare Graph ist 4-färbbar.

Aus dem Beweis des 4-Farben-Satzes von Appel und Haken lässt sich ein 4-Färbungsalgorithmus für planare Graphen mit einer Laufzeit von $\mathcal{O}(n^4)$ gewinnen.

In 1997 fanden Robertson, Sanders, Seymour und Thomas einen einfacheren Beweis für den 4-Farben-Satz, welcher zwar einen deutlich schnelleren $\mathcal{O}(n^2)$ Algorithmus liefert, aber ebenfalls nur mit Computer-Unterstützung verifizierbar ist.

Beispiel 2.8. *Wie die folgenden Einbettungen von K_4 und $K_{2,3}$ in die Ebene zeigen, sind K_4 und $K_{2,3}$ planar.*

K_4: 

$K_{2,3}$: 

◁

Zur Beantwortung der Frage, ob auch K_5 und $K_{3,3}$ planar sind, betrachten wir die ***Gebiete***, die bei der Einbettung von (zusammenhängenden) Graphen in die Ebene entstehen. Dabei gehören 2 Punkte zum selben Gebiet, falls es zwischen ihnen eine Verbindungslinie gibt, die keine Kante des eingebetten Graphen kreuzt oder berührt. Nur eines dieser Gebiete ist unbeschränkt und dieses wird als ***äußeres Gebiet*** bezeichnet. Die Anzahl der Gebiete von G bezeichnen wir mit $\boldsymbol{r(G)}$ oder kurz mit r.

Die begrenzenden Kanten eines Gebietes g bilden seinen ***Rand*** $\boldsymbol{rand(g)}$. Ihre Anzahl bezeichnen wir mit $\boldsymbol{d(g)}$, wobei Kanten $\{u, v\}$, an die g von beiden Seiten grenzt, doppelt gezählt werden. Wir ordnen die Kanten auf dem Rand R von g, indem wir sie der Reihe nach so durchlaufen, dass g „in Fahrtrichtung links“ liegt. Anders gesagt, verlassen wir jeden Knoten u, den wir über eine Kante e erreichen, über die im Uhrzeigersinn nächste mit u inzidente Kante e' wieder. Auf diese Weise erhält jede Kante auf dem Rand von g eine Richtung (oder Orientierung). Grenzt eine Kante e von beiden Seiten an g, so kommt sie in der zirkulären Ordnung $rand(g)$ mit beiden Orientierungen vor (siehe Beispiel 2.9).

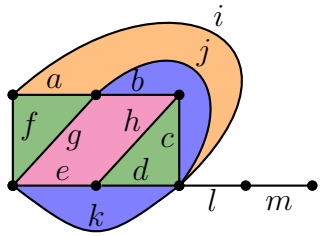
Da jede Kante zur Gesamtlänge $\sum_g d(g)$ aller Ränder den Wert 2 beiträgt (sie wird genau einmal in jeder Richtung durchlaufen), folgt

$$\sum_g d(g) = 2m(G).$$

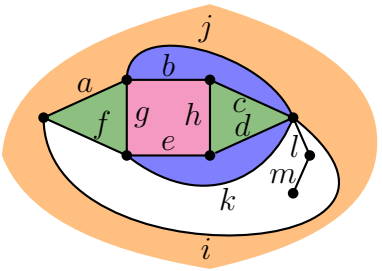
Wir nennen das Tripel $G' = (V, E, R)$ eine ***ebene Realisierung*** des Graphen $G = (V, E)$, falls es eine Einbettung von G in die Ebene gibt, deren Gebiete die Ränder in R haben. In diesem Fall nennen wir $G' = (V, E, R)$ auch einen ***ebenen Graphen***. Ist G nicht zusammenhängend, so betten wir die Komponenten von G in die Ebene ein und fassen alle Ränder, die bei diesen Einbettungen entstehen, zu einer Randmenge R zusammen.

Führen zwei Einbettungen von G in die Ebene auf dieselbe Randmenge R, so werden sie als ***äquivalent*** angesehen. Man kann auch Einbettungen kombinatorisch bis auf Äquivalenz beschreiben, indem man für jeden Knoten u die zirkuläre Ordnung π_u aller mit u inzidenten Kanten angibt. Man nennt $\pi = \{\pi_u \mid u \in V\}$ ein ***Rotationssystem*** für G, falls es eine entsprechende Einbettung gibt. Rotationssysteme haben den Vorteil, dass sie bei Verwendung der Adjazenzlistendarstellung ohne zusätzlichen Platzaufwand gespeichert werden können, indem man die zu u adjazenten Knoten gemäß π_u anordnet.

Beispiel 2.9. *Die beiden nebenstehenden Einbettungen eines Graphen $G = (V, E)$ in die Ebene haben jeweils 7 Gebiete und führen beide auf den ebenen Graphen $G' = (V, E, R)$ mit den 7 Rändern*



$$R = \{(a, f, g), (a, j, i), (b, g, e, h), (b, c, j), (c, h, d), (d, e, k), (f, i, l, m, m, l, k)\}.$$



Das zugehörige Rotationssystem ist

$$\pi = \{(a, f, i), (a, j, b, g), (b, c, h), (e, k, f, g), (d, e, h), (c, j, i, l, k, d), (l, m), (m)\}.$$

Man beachte, dass sowohl in R als auch in π jede Kante genau zweimal vorkommt. Anstelle von (zirkulären) Kantenfolgen kann man die Elemente von R und π natürlich auch durch entsprechende Knotenfolgen beschreiben. ◁

Satz 2.10 (Polyederformel von Euler, 1750)**.**
Für einen zusammenhängenden ebenen Graphen $G = (V, E, R)$ gilt

$$n(G) - m(G) + r(G) = 2. \qquad (*)$$

Beweis. Wir führen den Beweis durch Induktion über die Kantenzahl $m(G) = m$.

$m = 0$: Da G zusammenhängend ist, muss dann $n = 1$ sein.

Somit ist auch $r = 1$, also $(*)$ erfüllt.

$m - 1 \rightsquigarrow m$: Sei G ein zusammenhängender ebener Graph mit m Kanten.

Ist G ein Baum, so entfernen wir ein Blatt und erhalten einen zusammenhängenden ebenen Graphen G' mit $n' = n - 1$ Knoten, $m' = m - 1$ Kanten und $r' = r$ Gebieten. Nach IV folgt $n - m + r = (n - 1) - (m - 1) + r = n' - m' + r' = 2$.

Falls G kein Baum ist, entfernen wir eine Kante auf einem Kreis in G und erhalten einen zusammenhängenden ebenen Graphen G' mit $n' = n$ Knoten, $m' = m - 1$ Kanten und $r' = r - 1$ Gebieten. Nach IV folgt $n - m + r = n - (m - 1) + (r - 1) = n' - m' + r' = 2$. ■

Korollar 2.11. *Sei $G = (V, E)$ ein planarer Graph mit $n \geq 3$ Knoten. Dann ist $m \leq 3n - 6$. Falls G dreiecksfrei ist, gilt sogar $m \leq 2n - 4$.*

Beweis. O.B.d.A. sei G zusammenhängend. Wir betrachten eine beliebige planare Einbettung von G. Da $n \geq 3$ ist, ist jedes Gebiet g von $d(g) \geq 3$ Kanten umgeben. Daher ist $2m = i = \sum_g d(g) \geq 3r$ bzw. $r \leq 2m/3$. Eulers Formel liefert

$$m = n + r - 2 \leq n + 2m/3 - 2,$$

was $(1 - 2/3)m \leq n - 2$ und somit $m \leq 3n - 6$ impliziert.

Wenn G dreiecksfrei ist, ist jedes Gebiet von $d(g) \geq 4$ Kanten umgeben. Daher ist $2m = i = \sum_g d(g) \geq 4r$ bzw. $r \leq m/2$. Eulers Formel liefert daher $m = n + r - 2 \leq n + m/2 - 2$, was $m/2 \leq n - 2$ und somit $m \leq 2n - 4$ impliziert. ■

Korollar 2.12. *Die Graphen K_5 und $K_{3,3}$ sind nicht planar.*

Beweis. Wegen $n(K_5) = 5$, also $3n(K_5) - 6 = 9$, und wegen $m(K_5) = \binom{5}{2} = 10$ gilt $m(K_5) \not\leq 3n(K_5) - 6$.

Wegen $n(K_{3,3}) = 6$, also $2n(K_{3,3}) - 4 = 8$, und wegen $m(K_{3,3}) = 3 \cdot 3 = 9$ gilt $m(K_{3,3}) \not\leq 2n(K_{3,3}) - 4$. ■

Als weitere interessante Folgerung aus der Polyederformel können wir zeigen, dass jeder planare Graph einen Knoten v vom Grad $\deg(v) \leq 5$ hat.

Korollar 2.13. *Jeder planare Graph hat einen Minimalgrad $\delta \leq 5$.*

Beweis. Für $n \leq 6$ ist die Behauptung klar. Für $n > 6$ impliziert die Annahme $\delta \geq 6$ die Ungleichung

$$m = \tfrac{1}{2} \textstyle\sum_{u \in V} \deg(u) \geq \tfrac{1}{2} \sum_{u \in V} 6 = 3n,$$

was im Widerspruch zu $m \leq 3n - 6$ steht. ■

Definition 2.14. *Seien $G = (V, E)$ und H Graphen und seien $u, v \in V$.*

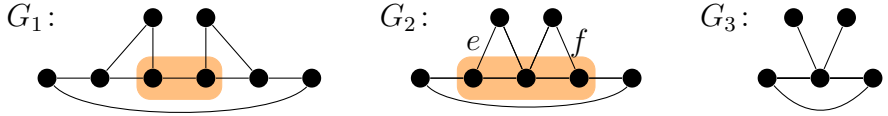
- *Durch **Fusion** von u und v entsteht aus G der Graph $G_{uv} = (V - \{v\}, E')$ mit*

$$E' = \{e \in E \mid v \notin e\} \cup \{\{u, v'\} \mid \{v, v'\} \in E - \{u, v\}\}.$$

- *Ist $e = \{u, v\}$ eine Kante von G (also $e \in E$), so sagen wir auch, G_{uv} entsteht aus G durch **Kontraktion** der Kante e.*

- *Hat zudem v den Grad 2 mit $N_G(v) = \{u, w\}$, so sagen wir auch, G_{uv} entsteht aus G durch **Überbrückung** des Knotens v bzw. G aus G_{uv} durch **Unterteilung** der Kante $\{u, w\}$.*
- *G heißt zu H **kontrahierbar**, falls H aus einer isomorphen Kopie von G durch wiederholte Kontraktionen gewonnen werden kann. In diesem Fall nennen wir H auch eine **Kontraktion** von G bzw. G eine **Expansion** von H.*
- *H heißt zu G **unterteilbar**, falls G aus einer isomorphen Kopie von H durch wiederholte Unterteilungen von Kanten gewonnen werden kann. In diesem Fall nennen wir G auch eine **Unterteilung** von H bzw. H eine **Überbrückung** von G.*
- *H heißt **Minor** von G, wenn ein Teilgraph von G zu H kontrahierbar ist. H heißt **topologischer Minor** von G, wenn ein Teilgraph von G eine Unterteilung von H ist.*
- *G heißt **H-frei**, falls H kein Minor von G ist. Für eine Menge $\mathcal{H}$ von Graphen heißt G **$\mathcal{H}$-frei**, falls kein $H \in \mathcal{H}$ ein Minor von G ist.*

Beispiel 2.15. *Betrachte folgende Graphen:*



- *G_2 ist ein Minor von G_1, da G_2 durch Kontraktion der in G_1 umrandeten Kante entsteht; entsprechend ist G_3 ein Minor von G_2 und auch von G_1.*
- *G_2 ist keine Unterteilung von G_3, da G_2 im Gegensatz zu G_3 Knoten vom Grad 3 hat. Falls wir jedoch die beiden Kanten e und f aus G_2 entfernen, so ist der resultierende Teilgraph G_2' eine Unterteilung von G_3. Somit ist G_3 ein topologischer Minor von G_2.*
- *G_2 und G_3 sind aber keine topologischen Minoren von G_1, da G_2 und G_3 einen Knoten vom Grad 4 haben, aber G_1 nur Knoten vom Grad ≤ 3.* ◁

Es ist klar, dass die Klasse $\mathcal{K}$ der planaren Graphen zwar unter Subgraphbildung, Kontraktion, Unterteilung und Überbrückung abgeschlossen ist, aber nicht unter Fusion. Folglich ist jeder (topologische) Minor und jede Unterteilung eines planaren Graphen ebenfalls planar.

Nach Definition lässt sich jeder (topologische) Minor H von G aus einem zu G isomorphen Graphen durch wiederholte Anwendung folgender Operationen gewinnen:

- Entfernen einer Kante oder eines Knotens,
- Kontraktion einer Kante (bzw. Überbrückung eines Knotens).

Da die Kontraktionen (bzw. Überbrückungen) o.B.d.A. auch zuletzt ausgeführt werden können, gilt hiervon auch die Umkehrung. Zudem ist leicht zu sehen, dass zwei Graphen G und H genau dann (topologische) Minoren voneinander sind, wenn sie isomorph sind.

Satz 2.16 (Kempe 1878, Heawood 1890)**.**
Jeder planare Graph ist 5-färbbar.

Beweis. Wir beweisen den Satz durch Induktion über n.

$n = 1$: Klar.

$n - 1 \rightsquigarrow n$: Sei G ein planarer Graph mit $n(G) = n$ Knoten. Da G planar ist, existiert ein Knoten u mit $\deg(u) \leq 5$. Nun konstruieren wir zu G wie folgt einen Minor G':

- Im Fall $\deg(u) \leq 4$ sei $G' = G - u$, d.h. wir entfernen u aus G.
- Andernfalls hat u zwei Nachbarn v und w, die nicht durch eine Kante verbunden sind (andernfalls wäre K_5 ein Teilgraph von G). In diesem Fall sei $G' = (G_{vu})_{vw}$, d.h. wir kontrahieren die beiden Kanten $\{u, v\}$ und $\{u, w\}$ zum Knoten v.

Da G' ein Minor von G ist, ist G' planar. Da G' zudem höchstens $n - 1$ Knoten hat, hat G' nach IV eine 5-Färbung c'. Wir erweitern c' wie folgt zu einer 5-Färbung c von G:

- Im 2. Fall geben wir dem Knoten w die Farbe $c(w) = c'(v)$.

- Da nun in beiden Fällen die Nachbarn von u in G höchstens 4 verschiedene Farben haben, können wir auch u eine Farbe $c(u) \leq 5$ geben. ■

Kuratowski konnte 1930 beweisen, dass jeder nichtplanare Graph G den $K_{3,3}$ oder den K_5 als topologischen Minor enthält. Für den Beweis benötigen wir noch folgende Notationen.

Definition 2.17. *Sei $G = (V, E)$ ein Graph und $S \subseteq V$.*

- *Die Menge S heißt **Separator** in G, wenn es zwei Knoten $u, v \in V \setminus S$ gibt, zwischen denen in $G - S$ kein u-v-Weg existiert. Ist $|S| = k$, so nennen wir S auch einen **k-Separator** zwischen u und v oder auch einen **u-v-Separator** der Größe k. Ein 1-Separator wird auch **Artikulation** oder **Schnittknoten** von G genannt.*
- *Ein Graph G heißt **k-zusammenhängend**, $0 \leq k \leq n-1$, falls G keinen $(k-1)$-Separator hat. Die größte Zahl k, für die G k-zusammenhängend ist, heißt die **Zusammenhangszahl** von G und wird mit $\kappa(G)$ bezeichnet.*

Ein Graph G mit $n \geq 2$ Knoten ist also genau dann zusammenhängend, wenn $\kappa(G) \geq 1$ ist.

Lemma 2.18. *Ist ein Graph $G = (V, E)$ nicht planar, so hat er einen*

- *2-zusammenhängenden Untergraphen $U = (V', E')$ und einen*
- *3-zusammenhängenden topologischen Minor $M = (V'', E'')$,*

*die **minimal nicht planar** sind, d.h. U und M sind nicht planar und für alle $e' \in E'$ und $e'' \in E''$ sind die Graphen $U - e'$ und $M - e''$ planar.*

Beweis. Wir entfernen zuerst solange Kanten und Knoten aus G, bis wir aus dem verbliebenen Teilgraphen $U = (V', E')$ keine weiteren Kanten oder Knoten entfernen können, ohne dass U planar wird.

U ist zusammenhängend, da andernfalls mindestens eine Komponente von U nicht planar ist und wir alle übrigen Komponenten entfernen könnten, ohne dass U planar wird.

U ist sogar 2-zusammenhängend, da U sonst einen Schnittknoten s enthält und $U - s$ in $k \geq 2$ Komponenten $U[V_1], \ldots, U[V_k]$ zerfällt. Dann ist aber mindestens ein Teilgraph $T_i = U[V_i \cup \{s\}]$ nicht planar und wir können alle Knoten außerhalb von T_i entfernen, ohne dass U planar wird.

Um einen topologischen Minor M von G mit den behaupteten Eigenschaften zu erhalten, konstruieren wir zu U einen topologischen Minor U', der minimal nicht planar ist und zudem 3-zusammenhängend ist oder weniger Knoten als U hat. Indem wir diese Konstruktion wiederholen, erhalten wir schließlich M.

Falls U 3-zusammenhängend ist, ist $U' = U$. Andernfalls gibt es in U einen 2-Separator $S = \{u, v\}$, d.h. $U - S$ zerfällt in $k \geq 2$ Komponenten $U[V_1], \ldots, U[V_k]$. Betrachte die (2-zusammenhängenden) Graphen $G_i = U[V_i \cup \{u, v\}] \cup \{u, v\}$. Dann ist mindestens ein G_i nicht planar (z.B. G_1), da sonst auch U planar wäre. Da $k \geq 2$ ist, erhalten wir einen zu G_1 isomorphen Graphen U' als topologischen Minor von $H = U[V_1 \cup V_2 \cup \{u, v\}]$ (und damit von U), indem wir in $U[V_2 \cup \{u, v\}]$ einen beliebigen u-v-Pfad P wählen und aus H alle Knoten und Kanten entfernen, die nicht auf P liegen und danach P überbrücken. Dann hat U' weniger Knoten als U und ist wie U minimal nicht planar. ■

Definition 2.19. *Sei G ein Graph und sei K ein Kreis in G. Ein Teilgraph B von G heißt **Brücke** von K in G, falls*

- *B nur aus einer Kante besteht, die zwei Knoten von K verbindet, aber nicht auf K liegt (solche Brücken werden auch als **Sehnen** von K bezeichnet), oder*
- *$B - K$ eine Komponente von $G - K$ ist und B aus $B - K$ durch Hinzufügen aller Kanten zwischen $B - K$ und K (und der zugehörigen Endpunkte auf K) entsteht.*

*Die Knoten von B, die auf K liegen, heißen **Kontaktpunkte** von B. Zwei Brücken B und B' von K heißen **inkompatibel**, falls*

- *B Kontaktpunkte u, v und B' Kontaktpunkte u', v' hat, so dass diese vier Punkte in der Reihenfolge u, u', v, v' auf K liegen, oder*
- *B und B' mindestens 3 gemeinsame Kontaktpunkte haben.*

Es ist leicht zu sehen, dass in einem planaren Graphen kein Kreis mehr als zwei inkompatible Brücken haben kann.

Satz 2.20 (Kuratowski 1930)**.**
Für einen Graphen G sind folgende Aussagen äquivalent:

(i) G ist planar.

(ii) G enthält weder den $K_{3,3}$ noch den K_5 als topologischen Minor.

Beweis. Die Implikation von *i*) nach *ii*) folgt aus der Abgeschlossenheit der planaren Graphen unter (topologischer) Minorenbildung.

Die Implikation von *ii*) nach *i*) zeigen wir durch Kontraposition. Sei also $G = (V, E)$ nicht planar. Dann hat G nach Lemma 2.18 einen 3-zusammenhängenden nicht planaren topologischen Minor $M = (V', E')$, so dass $M - e'$ für jede Kante $e' \in E'$ planar ist. Wir entfernen eine beliebige Kante $e_0 = \{a_0, b_0\}$ aus M. Dann ist $M - e_0$ planar. Da $M - e_0$ 2-zusammenhängend ist, gibt es in $M - e_0$ einen Kreis K durch die beiden Knoten a_0 und b_0 (siehe Übungen). Wir wählen K zusammen mit einer ebenen Realisierung H' von $M - e_0$ so, dass K möglichst viele Gebiete in H' einschließt.

Für zwei Knoten a, b auf K bezeichnen wir mit $K[a, b]$ die Menge aller Knoten, die auf dem Bogen von a nach b (im Uhrzeigersinn) auf K liegen. Zudem sei $K[a, b) = K[a, b] \setminus \{b\}$. Die Mengen $K(a, b)$ und $K(a, b]$ sind analog definiert.

Die Kanten jeder Brücke B von K in $M - e_0$ verlaufen in H' entweder alle innerhalb oder alle außerhalb von K. Im ersten Fall nennen wir B eine ***innere Brücke*** und im zweiten eine ***äußere Brücke***.

Es ist klar, dass K in H' mindestens eine innere und mindestens eine äußere Brücke haben muss (sonst könnten wir e' zu H' hinzufügen). Zudem muss jede äußere Brücke B genau zwei Kontaktpunkte haben:

einen Knoten $u \in K(a_0, b_0)$ und einen Knoten $v \in K(b_0, a_0)$. Andernfalls hätte B nämlich mindestens 2 Kontaktpunkte auf $K[a_0, b_0]$ oder auf $K[b_0, a_0]$. Daher könnte K zu einem Kreis K' erweitert werden, der in H' mehr Gebiete einschließt (bzw. ausschließt) als K, was der Wahl von K und H' widerspricht. Da M 3-zusammenhängend ist, muss B zudem eine Sehne $\{u, v\}$ sein.

K hat in M außer den Brücken in $M - e_0$ noch zusätzlich die Brücke e_0. Wir wählen nun eine innere Brücke B, die sowohl zu e_0 als auch zu mindestens einer äußeren Brücke $e_1 = \{a_1, b_1\}$ inkompatibel ist. Eine solche Brücke muss es geben, da wir sonst alle mit e_0 inkompatiblen inneren Brücken nach außen klappen und e_0 als innere Brücke hinzunehmen könnten, ohne die Planarität zu verletzen.

Wir benutzen K und die drei Brücken e_0, e_1 und B, um eine Unterteilung des $K_{3,3}$ oder des K_5 in M zu finden. Hierzu geben wir entweder zwei disjunkte Mengen $A_1, A_2 \subseteq V'$ mit jeweils 3 Knoten an, so dass 9 knotendisjunkte Pfade zwischen allen Knoten $a \in A_1$ und $b \in A_2$ existieren. Oder wir geben eine Menge $A \subseteq V'$ mit fünf Knoten an, so dass 10 knotendisjunkte Pfade zwischen je zwei Knoten $a, b \in A$ existieren. Da e_0 und e_1 inkompatibel sind, können wir annehmen, dass die vier Knoten a_0, a_1, b_0, b_1 in dieser Reihenfolge auf K liegen.

Fall 1: B hat einen Kontaktpunkt $k_1 \notin \{a_0, a_1, b_0, b_1\}$. Aus Symmetriegründen können wir $k_1 \in K(a_0, a_1)$ annehmen. Da B weder zu e_0 noch zu e_1 kompatibel ist, hat B weitere Kontaktpunkte $k_2 \in K(b_0, a_0)$ und $k_3 \in K(a_1, b_1)$, wobei $k_2 = k_3$ sein kann.

Fall 1a: Ein Knoten $k_i \in \{k_2, k_3\}$ liegt auf dem Bogen $K(b_0, b_1)$. In diesem Fall existieren 9 knotendisjunkte Pfade zwischen $\{a_0, a_1, k_i\}$ und $\{b_0, b_1, k_1\}$.

Fall 1b: $K(b_0, b_1) \cap \{k_2, k_3\} = \emptyset$. In diesem Fall ist $k_2 \in K[b_1, a_0)$ und $k_3 \in K(a_1, b_0]$. Dann gibt es in B einen Knoten u, von dem aus 3 knotendisjunkte Pfade zu $\{k_1, k_2, k_3\}$ existieren. Folglich gibt es 9 knotendisjunkte Pfade zwischen $\{a_0, a_1, u\}$ und $\{k_1, k_2, k_3\}$.

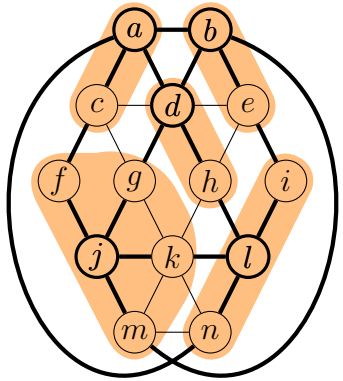
Fall 2: Alle Kontaktpunkte von B liegen in der Menge $\{a_0, a_1, b_0, b_1\}$. Da B inkompatibel zu e_0 und e_1 ist, müssen in diesem Fall alle vier Punkte zu B gehören. Sei P_0 ein a_0-b_0-Pfad in B und sei P_1 ein a_1-b_1-Pfad in B. Sei u der erste Knoten auf P_0, der auch auf P_1 liegt und sei v der letzte solche Knoten.

Fall 2a: $u = v$. Dann gibt es in B vier knotendisjunkte Pfade von u zu $\{a_0, a_1, b_0, b_1\}$ und somit existieren in M 10 knotendisjunkte Pfade zwischen den Knoten u, a_0, a_1, b_0, b_1.

Fall 2b: $u \neq v$. Durch u und v wird der Pfad P_1 in drei Teilpfade P_{xu}, P_{uv} und P_{vy} unterteilt, wobei die Indizes die Endpunkte bezeichnen und $\{x, y\} = \{a_1, b_1\}$ ist.

Somit gibt es in B drei Pfade zwischen u und jedem Knoten in $\{a_0, v, x\}$ und zwei Pfade zwischen v und jedem Knoten in $\{b_0, y\}$, die alle 5 knotendisjunkt sind. Folglich gibt es in M 9 knotendisjunkte Pfade zwischen $\{a_0, v, x\}$ und $\{b_0, y, u\}$. ■

Beispiel 2.21. *Der nebenstehende Graph ist nicht planar, da wir den K_5 durch Kontraktion der farblich unterlegten Teilgraphen als Minor von G erhalten.*
Alternativ lässt sich der K_5 auch als ein topologischer Minor von G erhalten, indem wir die dünnen Kanten entfernen und in dem resultierenden Teilgraphen alle Knoten vom Grad 2 überbrücken. ◁



Eine unmittelbare Folgerung aus dem Satz von Kuratowski ist folgende Charakterisierung der Klasse der planaren Graphen.

Korollar 2.22 (Wagner 1937)**.** *Ein Graph ist genau dann planar, wenn er $\{K_{3,3}, K_5\}$-frei ist.*

Satz 2.23 (Satz von Robertson und Seymour, 1983-2004)**.** *Sei $\mathcal{K}$ eine Graphklasse, die unter Minorenbildung abgeschlossen ist. Dann gibt es eine endliche Menge $\mathcal{H}$ von Graphen mit*

$$\mathcal{K} = \{G \mid G \text{ ist } \mathcal{H}\text{-frei}\}.$$

*Die Graphen in $\mathcal{H}$ sind bis auf Isomorphie eindeutig bestimmt und heißen **verbotene Minoren** für die Klasse $\mathcal{K}$.*

Eine interessante Folgerung aus diesem Satz ist, dass jede unendliche Graphklasse zwei Graphen G und H enthält, so dass H ein Minor von G ist. Das Problem, für zwei gegebene Graphen G und H zu entscheiden, ob H ein Minor von G ist, ist zwar NP-vollständig (da sich das Hamiltonkreisproblem darauf reduzieren lässt). Für einen festen Graphen H ist das Problem dagegen effizient entscheidbar.

Satz 2.24 (Robertson und Seymour, 1995)**.** *Für jeden Graphen H gibt es einen $O(n^3)$-zeitbeschränkten Algorithmus, der für einen gegebenen Graphen G entscheidet, ob er H-frei ist.*

Korollar 2.25. *Die Zugehörigkeit zu jeder unter Minorenbildung abgeschlossenen Graphklasse $\mathcal{K}$ ist in* P *entscheidbar.*

Der Entscheidungsalgorithmus für $\mathcal{K}$ lässt sich allerdings nur angeben, wenn wir die verbotenen Minoren für $\mathcal{K}$ kennen. Leider ist der Beweis von Satz 2.23 in dieser Hinsicht nicht konstruktiv, so dass der Nachweis, dass $\mathcal{K}$ unter Minorenbildung abgeschlossen ist, nicht automatisch zu einem effizienten Erkennungsalgorithmus für $\mathcal{K}$ führt.

2.2 Färben von chordalen Graphen

Chordale Graphen treten in vielen Anwendungen auf, z.B. sind alle Intervall- und alle Splitgraphen chordal. Wir werden sehen, dass sich für chordale Graphen effizient eine optimale Knotenfärbung berechnen lässt.

Definition 2.26. *Ein Graph $G = (V, E)$ heißt **chordal** oder **trianguliert**, wenn jeder Kreis $K = (u_1, \ldots, u_l, u_1)$ der Länge $l \geq 4$ in G mindestens eine Sehne hat.*

G ist also genau dann chordal, wenn er keinen induzierten Kreis der Länge $l \geq 4$ enthält (ein induzierter Kreis ist ein induzierter Teilgraph $G[V']$, $V' \subseteq V$, der ein Kreis ist). Dies zeigt, dass die Klasse der chordalen Graphen unter induzierter Teilgraphbildung abgeschlossen ist (aber nicht unter Teilgraphbildung). Jede solche Graphklasse $\mathcal{G}$ ist durch eine Familie von minimalen ***verbotenen induzierten Teilgraphen*** H_i charakterisiert, die bis auf Isomorphie eindeutig bestimmt sind. Die Graphen H_i gehören also nicht zu $\mathcal{G}$, aber sobald wir einen Knoten daraus entfernen, erhalten wir einen Graphen in $\mathcal{G}$. Die Klasse der chordalen Graphen hat die Familie der Kreise C_n der Länge $n \geq 4$ als verbotene induzierte Teilgraphen.

Lemma 2.27. *Für einen Graphen G sind folgende Aussagen äquivalent.*

(i) G ist chordal.

(ii) Jeder inklusionsminimale x-y-Separator S in G ist eine Clique.

(iii) Jedes Paar von nicht adjazenten Knoten x und y in G hat einen x-y-Separator S, der eine Clique ist.

Beweis. Um zu zeigen, dass die zweite Aussage aus der ersten folgt, nehmen wir an, dass G einen minimalen x-y-Separator S hat (d.h. $S \setminus \{s\}$ ist für jedes $s \in S$ kein x-y-Separator), der zwei nicht adjazente Knoten u und v enthält. Seien $G[V_1]$ und $G[V_2]$ die beiden Komponenten in $G - S$ mit $x \in V_1$ und $y \in V_2$. Da S minimal ist, haben die beiden Knoten u und v sowohl einen Nachbarn in V_1 als auch in V_2. Betrachte die beiden Teilgraphen $G_i = G[V_i \cup \{u, v\}]$ $(i = 1, 2)$ und wähle jeweils einen kürzesten u-v-Pfad P_i in G_i. Da deren Länge ≥ 2 ist, ist $K = P_1 \cup P_2$ ein Kreis der Länge ≥ 4. Aufgrund der Konstruktion ist zudem klar, dass K keine Sehnen in G hat.

Dass die zweite Aussage die dritte impliziert, ist klar, da jedes Paar von nicht adjazenten Knoten x und y einen x-y-Separator S hat, und S eine Clique sein muss, wenn wir S inklusionsminimal wählen.

Um zu zeigen, dass die erste Aussage aus der dritten folgt, nehmen wir an, dass G nicht chordal ist. Dann gibt es in G einen induzierten Kreis K der Länge ≥ 4. Seien x und y zwei beliebige nicht adjazente Knoten auf K und sei S ein x-y-Separator in G. Dann muss S mindestens zwei nicht adjazente Knoten aus K enthalten. ■

Definition 2.28. *Sei $G = (V, E)$ ein Graph und sei $k \geq 0$. Ein Knoten $u \in V$ vom Grad k heißt **k-simplizial**, wenn alle Nachbarn von u paarweise adjazent sind. Jeder k-simpliziale Knoten wird auch als **simplizial** bezeichnet.*

Zusammenhängende chordale Graphen können als eine Verallgemeinerung von Bäumen aufgefasst werden. Ein Graph G ist ein Baum, wenn er aus K_1 durch sukzessives Hinzufügen von 1-simplizialen Knoten erzeugt werden kann. Entsprechend heißt G ***k-Baum***, wenn G aus K_k durch sukzessives Hinzufügen von k-simplizialen Knoten erzeugt werden kann. Wir werden sehen, dass ein zusammenhängender Graph G genau dann chordal ist, wenn er aus einem isolierten Knoten (also aus einer 1-Clique) durch sukzessives Hinzufügen von simplizialen Knoten erzeugt werden kann. Äquivalent hierzu ist, dass G durch sukzessives Entfernen von simplizialen Knoten auf einen isolierten Knoten reduziert werden kann.

Definition 2.29. *Sei $G = (V, E)$ ein Graph. Eine lineare Ordnung $(u_1, \ldots, u_n)$ auf V heißt **perfekte Eliminationsordnung (PEO)** von G, wenn u_i simplizial in $G[u_1, \ldots, u_i]$ für $i = 2, \ldots, n$ ist.*

Wir eliminieren die Knoten von G also in der Reihenfolge $u_n, \ldots, u_2, u_1$. Es ist klar dass der K_n alle $n!$ lineare Ordnungen auf V als PEO hat. Das folgende Lemma verallgemeinert die bekannte Tatsache, dass jeder nicht vollständige Baum T (also $T \notin \{K_1, K_2\}$) mindestens zwei nicht adjazente Blätter hat.

Lemma 2.30. *Jeder nicht vollständige chordale Graph G besitzt mindestens zwei simpliziale Knoten, die nicht adjazent sind.*

Beweis. Wir führen Induktion über n. Für $n \leq 2$ ist die Behauptung klar. Sei $G = (V, E)$ ein Graph mit $n \geq 3$ Knoten. Da G nicht vollständig ist, enthält G zwei nichtadjazente Knoten x_1 und x_2. Sei S ein minimaler x_1-x_2-Separator der Größe $k \geq 0$. Im Fall $k > 0$ ist S nach Lemma 2.27 eine Clique in G. Seien $G[V_1]$ und $G[V_2]$ die beiden Komponenten von $G - S$ mit $x_i \in V_i$. Wir zeigen die Existenz zweier simplizialer Knoten $s_i \in V_i$, $i = 1, 2$.

Betrachte die Teilgraphen $G_i = G[V_i \cup S]$. Da G_i chordal ist und weniger als n Knoten hat, ist G_i nach IV entweder eine Clique oder G_i enthält mindestens zwei nicht adjazente simpliziale Knoten y_i, z_i.

Falls G_i eine Clique ist, ist $s_i = x_i$ simplizial in G_i, und da x_i keine Nachbarn außerhalb von $V_i \cup S$ hat, ist s_i dann auch simplizial in G.

Ist G_i keine Clique, kann höchstens einer der beiden Knoten y_i, z_i zu S gehören (da S im Fall $S \neq \emptyset$ eine Clique und $\{y_i, z_i\} \notin E$ ist). O.B.d.A. sei $y_i \in V_i$. Dann hat $s_i = y_i$ keine Nachbarn außerhalb von $V_i \cup S$ und somit ist s_i auch simplizial in G. ■

Satz 2.31. *Ein Graph ist genau dann chordal, wenn er eine PEO hat.*

Beweis. Falls G chordal ist, lässt sich eine PEO gemäß Lemma 2.30 bestimmen, indem wir für $i = n, \ldots, 2$ sukzessive einen simplizialen Knoten u_i in $G - \{u_{i+1}, \ldots, u_n\}$ wählen.

Für die umgekehrte Richtung sei $(u_1, \ldots, u_n)$ eine PEO von G. Wir zeigen induktiv, dass $G_i = G[u_1, \ldots, u_i]$ chordal ist. Da u_{i+1} simplizial in G_{i+1} ist, enthält jeder Kreis K der Länge ≥ 4 in G_{i+1}, auf dem u_{i+1} liegt, eine Sehne zwischen den beiden Kreisnachbarn von u_{i+1}. Daher ist mit G_i auch G_{i+1} chordal. ■

Korollar 2.32. *Es gibt einen Polynomialzeitalgorithmus A, der für einen gegebenen Graphen G eine PEO berechnet, falls G chordal ist, und andernfalls einen induzierten Kreis der Länge ≥ 4 ausgibt.*

Beweis. A versucht wie im Beweis von Theorem 2.31 beschrieben, eine PEO zu bestimmen. Stellt sich heraus, dass $G_i = G - \{u_{i+1}, \ldots, u_n\}$ keinen simplizialen Knoten u_i hat, so ist G_i wegen Lemma 2.30 nicht chordal. Daher gibt es in G_i nach Lemma 2.27 *(iii)* ein Knotenpaar x, y, so dass kein x-y-Separator eine Clique ist. Berechnen wir für dieses Paar einen beliebigen minimalen x-y-Separator S, so ist S keine Clique und wir können wie im Beweis von $(i) \implies (ii)$ einen induzierten Kreis K der Länge ≥ 4 in G_i konstruieren. Da G_i ein induzierter Teilgraph von G ist, ist K auch ein induzierter Kreis in G. ■

Eine PEO kann verwendet werden, um einen chordalen Graphen zu färben:

Algorithmus `chordal-color`(V, E)

1 berechne eine PEO $(u_1, \ldots, u_n)$ für $G = (V, E)$
2 starte `greedy-color` mit der Knotenfolge $(u_1, \ldots, u_n)$

Satz 2.33. *Für einen gegebenen chordalen Graphen $G = (V, E)$ berechnet der Algorithmus* `chordal-color` *eine k-Färbung c von G mit $k = \chi(G) = \omega(G)$.*

Beweis. Sei u_i ein beliebiger Knoten mit $c(u_i) = k$. Da $(u_1, \ldots, u_n)$ eine PEO von G ist, ist u_i simplizial in $G[u_1, \ldots, u_i]$. Somit bilden die Nachbarn u_j von u_i mit $j < i$ eine Clique und wegen $c(u_i) = k$ bilden sie zusammen mit u_i eine k-Clique. Daher gilt $\chi(G) \leq k \leq \omega(G)$, woraus wegen $\omega(G) \leq \chi(G)$ die Behauptung folgt. ■

Um `chordal-color` in Linearzeit zu implementieren, benötigen wir einen Linearzeit-Algorithmus zur Bestimmung einer PEO. Rose, Tarjan und Lueker haben 1976 einen solchen Algorithmus angegeben, der auf ***lexikographischer Breitensuche*** (kurz ***LexBFS*** oder ***LBFS***, engl. ***lexicographic breadth-first search***) basiert. Bevor wir diese

Variante der Breitensuche vorstellen, gehen wir kurz auf verschiedene Ansätze zum Durchsuchen von Graphen ein.

Der folgende Algorithmus `GraphSearch(`V, E`)` startet eine Suche in einem beliebigen Knoten u und findet zunächst alle von u aus erreichbaren Knoten. Danach wird solange von einem noch nicht erreichten Knoten eine neue Suche gestartet, bis alle Knoten erreicht wurden. Die Menge der aktuellen Knoten wird dabei in einer Datenstruktur A gespeichert. Genauer enthält A alle bereits entdeckten Knoten, die noch nicht abgearbeitet sind.

Algorithmus `GraphSearch(`V, E`)`

```
1  R := ∅    // Menge der erreichten Knoten
2  L := ()   // Ausgabeliste
3  repeat
4    wähle u ∈ V\R    // u wurde neu entdeckt
5    append(L, u)
6    parent(u) := ⊥
7    A := {u}    // Menge der aktuellen Knoten
8    R := R ∪ {u}
9    while A ≠ ∅ do
10     wähle u aus A
11     if ∃v ∈ N(u)\R then
12       A := A ∪ {v}    // v wurde neu entdeckt
13       R := R ∪ {v}
14       append(L, v)
15       parent(v) := u
16     else entferne u aus A    // u wurde abgearbeitet
17 until R = V
18 return(L)
```

Der Algorithmus `GraphSearch(`V, E`)` findet in jedem Durchlauf der repeat-Schleife eine neue Komponente des Eingabegraphen $G = (V, E)$. Dies bedeutet, dass alle Knoten, die zu einer Komponente gehören,

konsekutiv in der Ausgabeliste $L = (u_1, \ldots, u_n)$ auftreten, wobei abgesehen vom ersten Knoten jeder Komponente jeder Knoten u_k einen Nachbarn u_i mit $i < k$ hat.

Die folgende Definition fasst diese Eigenschaften der Ausgabeliste zusammen.

Definition 2.34. *Sei $G = (V, E)$ ein Graph. Eine lineare Ordnung $(u_1, \ldots, u_n)$ auf V heißt* ***Suchordnung (SO)*** *von G, wenn für jedes Tripel $j < k < l$ gilt:*

$$u_j \in N(u_l) \setminus N(u_k) \Rightarrow \exists i < k : i \neq j \wedge u_i \in N(u_k).$$

Satz 2.35. *Für jeden Graphen $G = (V, E)$ gibt der Algorithmus* `GraphSearch(`V, E`)` *eine SO von G aus.*

Beweis. Ein Knoten u_k erhält nur dann den Wert $\texttt{parent}(u_k) = \bot$, wenn alle Knoten u_j mit $j < k$ bereits abgearbeitet sind und diese nur Nachbarn u_l mit $l < k$ hatten. Falls also ein Vorgänger u_j von u_k mit einem Nachfolger u_l von u_k verbunden ist, liefert die `parent`-Funktion einen Nachbarn $u_i = \texttt{parent}(u_k)$ von u_k mit $i < k$. Da $u_j \notin N(u_k)$ ist, gilt zusätzlich $i \neq j$. ■

Die `parent`-Funktion induziert einen gerichteten Wald $W = (V, E_{\texttt{parent}})$, dessen Kantenmenge aus allen Kanten der Form $(\texttt{parent}(v), v)$ mit $\texttt{parent}(v) \neq \bot$ besteht. Die Kanten von W werden auch als ***Baumkanten*** (kurz ***B-Kanten***) und W wird auch als ***Suchwald*** von $G = (V, E)$ bezeichnet. Für jeden Knoten $v \in V$ gibt es genau eine Wurzel w in W, von der aus v in W erreichbar ist. Der eindeutig bestimmte w-v-Pfad $P = (u_0, \ldots, u_l)$ in W mit $u_0 = w$ und $u_l = v$ lässt sich ausgehend von $u_l = v$ unter Verwendung der `parent`-Funktion mittels $u_{i-1} = \texttt{parent}(u_i)$ für $i = l, \ldots, 1$ berechnen. P wird auch als ***parent-Pfad*** von v bezeichnet. Es ist klar, dass 2 Knoten v und v' genau dann in einer Komponente von G liegen, wenn sie die gleiche Wurzel haben.

Realisieren wir die Menge der aktuellen Knoten als einen Keller S, so erhalten wir eine Suchstrategie, die als ***Tiefensuche*** (kurz ***DFS***, engl. ***depth first search***) bezeichnet wird. Die Benutzung eines Kellers bewirkt, dass nach der Entdeckung eines neuen Knotens v unter den Nachbarn des aktuellen Knotens u die Suche zuerst bei den Nachbarn von v fortgesetzt wird, bevor die anderen Nachbarn von u getestet werden.

Algorithmus `DFS(`V, E`)`

```
1  R := ∅    // Menge der erreichten Knoten
2  L := ()   // Ausgabeliste
3  repeat
4    wähle u ∈ V\R    // u wurde neu entdeckt
5    R := R ∪ {u}
6    append(L, u)
7    parent(u) := ⊥
8    S := (u)    // Keller der aktuellen Knoten
9    while S ≠ () do
10     u := top(S)
11     if ∃v ∈ N(u)\R then
12       push(S, v)    // v wurde neu entdeckt
13       R := R ∪ {v}
14       append(L, v)
15       parent(v) := u
16     else pop(S)    // u wurde abgearbeitet
17 until R = V
18 return(L)
```

Definition 2.36. *Sei $G = (V, E)$ ein Graph. Eine lineare Ordnung $(u_1, \ldots, u_n)$ auf V heißt **DFS-Ordnung (DO)** von G, wenn für jedes Tripel $j < k < l$ gilt:*

$$u_j \in N(u_l) \setminus N(u_k) \Rightarrow \exists i : j < i < k \wedge u_i \in N(u_k).$$

Satz 2.37. *Für jeden Graphen $G = (V, E)$ gibt der Algorithmus* `DFS(`V, E`)` *eine DO von G aus.*

Beweis. Siehe Übungen. ■

Realisieren wir die Menge der abzuarbeitenden Knoten als eine Warteschlange Q, so findet der resultierende Algorithmus `BFS(`V, E`)` einen kürzesten Weg vom Startknoten u zu allen von u aus erreichbaren Knoten. Diese Suchstrategie wird als ***Breitensuche*** (kurz ***BFS***, engl. ***breadth first search***) bezeichnet. Die Benutzung einer Warteschlange Q zur Speicherung der noch abzuarbeitenden Knoten bewirkt, dass alle Nachbarknoten v des aktuellen Knotens u vor den bisher noch nicht erreichten Nachbarn von v ausgegeben werden.

Algorithmus `BFS(`V, E`)`

```
1  R := ∅    // Menge der erreichten Knoten
2  L := ()   // Ausgabeliste
3  repeat
4    wähle u ∈ V\R    // u wurde neu entdeckt
5    R := R ∪ {u}
6    parent(u) := ⊥
7    Q := (u)    // Warteschlange der aktuellen Knoten
8    while Q ≠ () do
9      u := dequeue(Q)    // u wird komplett abgearbeitet
10     append(L, u)
11     for all v ∈ N(u)\R do
12       enqueue(Q, v)    // v wurde neu entdeckt
13       parent(v) := u
14     R := R ∪ N(u)
15 until R = V
16 return(L)
```

Definition 2.38. *Sei $G = (V, E)$ ein Graph. Eine lineare Ordnung $(u_1, \ldots, u_n)$ auf V heißt **BFS-Ordnung (BO)** von G, wenn für jedes*

Tripel $j < k < l$ gilt:

$$u_j \in N(u_l) \setminus N(u_k) \Rightarrow \exists i < j : u_i \in N(u_k).$$

Satz 2.39. *Für jeden Graphen $G = (V, E)$ gibt der Algorithmus* `BFS(`V, E`)` *eine BO von G aus.*

Beweis. Existiert im Fall $k < l$ eine Position $j < k$ mit $u_j \in N(u_l) \setminus N(u_k)$, so muss es einen Knoten $u_i \in N(u_k)$ mit $i < j$ geben, der dafür gesorgt hat, dass der Knoten u_k vor dem Knoten u_l in die Warteschlange aufgenommen wurde. ■

BFS-Ordnungen lassen sich anschaulich anhand der Adjazenzmatrix charakterisieren. Sei $(u_1, \ldots, u_n)$ eine BO für $G = (V, E)$ und sei $A = (a_{ij})$ die Adjazenzmatrix von G mit $a_{ij} = 1 \Leftrightarrow \{u_i, u_j\} \in E$. Weiter seien $z_i = a_{i1} \ldots a_{i,i-1}$ die Präfixe der Zeilen von A, die unterhalb der Diagonale verlaufen. Sind nun die ersten j Einträge $a_{k1} \ldots a_{kj}$ einer Zeile s_k Null, so muss dies auch für jede Zeile s_l mit $l > k$ so sein, da im Fall $a_{lj} = 1$ der Knoten $u_j \in N(u_l) \setminus N(u_k)$ wäre und somit ein $i < j$ mit $a_{ki} = 1$ existieren müsste. Dies bedeutet, dass s_l mindestens so viele Nullen als Präfix hat wie z_k. Es ist aber möglich, dass z_k bspw. mit $00010\ldots$ beginnt und z_l mit $00011\ldots$.

Alternativ können wir Q auch als eine Warteschlange von Knotenmengen realisieren (siehe Algorithmus `BFS'`), um einen Überblick über alle möglichen Fortsetzungen der aktuellen Liste L zu einer BO zu erhalten. Die Prozedur `Dequeue`(Q) liefert ein beliebiges Element aus der ersten Menge in Q zurück und entfernt dieses aus Q.

Algorithmus `BFS'(`V, E`)`

```
1  R := ∅    // Menge der erreichten Knoten
2  L := ()   // Ausgabeliste
3  repeat
4    wähle u ∈ V\R
5    R := R ∪ {u}
6    Q := ({u})    // Warteschlange von Knotenmengen
7    while Q ≠ () do
8      u := Dequeue(Q)    // u wird komplett abgearbeitet
9      append(L, u)
10     if N(u) ⊈ R then enqueue(Q, N(u)\R)
11     R := R ∪ N(u)
12 until R = V
13 return(L)
```

Prozedur `Dequeue`(Q)

```
1  entferne u aus first(Q)
2  if first(Q) = ∅ then dequeue(Q)
3  return(u)
```

Fassen wir die Menge $V \setminus R$ der noch nicht erreichten Knoten als Nachfolgemenge der letzten Menge in Q auf, so wird von dieser Restmenge in jedem Durchlauf der `while`-Schleife von `BFS'` die Teilmenge $N(u) \setminus R$ abgetrennt und im Fall $N(u) \setminus R \neq \emptyset$ der Schlange Q hinzugefügt.

Der Unterschied von LexBFS zur normalen Breitensuche besteht darin, dass die zulässigen Ausgabefolgen gegenüber der BFS weiter eingeschränkt werden. Der Name von LexBFS rührt daher, dass die Knoten in einer Reihenfolge ausgegeben werden, die eine lexikographische Sortierung der Zeilenpräfixe z_i bewirkt, sofern man sie durch Anhängen von Einsen auf die gleiche Länge bringt. Eine solche Sortierung kann auch bei einer gewöhnlichen Breitensuche auftreten, ist bei dieser aber nicht garantiert. Bei einer Breitensuche werden die noch nicht besuchten Nachbarn des aktuellen Knotens in beliebiger Reihenfolge zur Warteschlange hinzugefügt und auch wieder in dieser Reihenfolge entfernt. Dagegen werden bei einer LexBFS die Knoten in der Warteschlange nachträglich umsortiert, falls dies notwendig ist, um eine LexBFS-Ordnung der Knoten zu erhalten (siehe Definition 2.40). Ähnlich wie bei `BFS'` wird hierzu die Menge der noch

nicht abgearbeiteten Knoten in eine Folge von Knotenmengen zerlegt. Im Gegensatz zu `BFS'` kann `LexBFS` aber nicht nur die letzte Menge $V \setminus R$ splitten, sondern alle Mengen der Folge.

Algorithmus `LexBFS`(V, E, u)

```
1 L := ()    // Ausgabeliste
2 Q := (V)    // Warteschlange von Knotenmengen
3 while Q ≠ () do
4   u := Dequeue(Q)    // u wird komplett abgearbeitet
5   append(L, u)
6   Splitqueue(Q, N(u))
7 return(L)
```

Prozedur `Splitqueue`(Q, S)

```
1 for T in Q with T ∩ S ∉ {∅, T} do
2   ersetze die Teilfolge (T) in Q durch (T ∩ S, T \ S)
```

Für eine effiziente Implementierung sollte die Schlange $Q = (T_1, \ldots, T_k)$ von Knotenmengen $T_i \subseteq V$ als doppelt verkettete Liste realisiert werden und für jeden Knoten v in der Adjazenzliste ein Zeiger auf die Menge T_i, die v enthält und auf seinen Eintrag in T_i gespeichert werden. Zudem sollte die for-Schleife in der Prozedur `Splitqueue` durch eine Schleife über die Knoten v in der Adjazenzliste $S = N(u)$ ersetzt werden.

Definition 2.40. *Sei $G = (V, E)$ ein Graph. Eine lineare Ordnung $(u_1, \ldots, u_n)$ auf V heißt **LexBFS-Ordnung (LBO)** von G, wenn für jedes Tripel $j < k < l$ gilt:*

$$u_j \in N(u_l) \setminus N(u_k) \Rightarrow \exists i < j : u_i \in N(u_k) \setminus N(u_l).$$

Ob eine Ordnung $(u_1, \ldots, u_n)$ eine LBO ist, lässt sich wie folgt an der gemäß $(u_1, \ldots, u_n)$ geordneten Adjazenzmatrix A ablesen: die verkürzten Zeilen $z_1, \ldots, z_n$ unter der Diagonalen müssen im folgenden

Sinne lexikalisch sortiert sein: entweder ist z_i ein Präfix von z_{i+1} oder z_i hat an der ersten Position, wo sich die beiden Strings unterscheiden, eine Eins. Bringen wir also die verkürzten Zeilen durch Anhängen von Einsen auf dieselbe Länge, so sind sie lexikographisch sortiert. Man erhält sogar eine lexikographische Ordnung auf den *kompletten* Zeilen von A, falls man die Diagonale auf 1 setzt und die Knoten in jeder Menge von Q nach absteigendem Knotengrad in G sortiert.

Satz 2.41. *Für jeden Graphen $G = (V, E)$ gibt der Algorithmus* `LexBFS`(V, E) *eine LBO $(u_1, \ldots, u_n)$ von G aus.*

Beweis. Sei $A = (a_{ij})$ die Adjazenzmatrix von G mit $a_{ij} = 1 \Leftrightarrow \{u_i, u_j\} \in E$. Wir zeigen, dass $(u_1, \ldots, u_n)$ eine LBO ist. Existiert nämlich im Fall $k < l$ eine Position $j < k$ mit $a_{kj} = 0$ und $a_{lj} = 1$, so muss es eine Position $i < j$ mit $a_{ki} = 1$ und $a_{li} = 0$ geben. Ansonsten wäre der Knoten u_l spätestens beim Abarbeiten von u_j in eine Menge vor dem Knoten u_k sortiert worden und könnte daher nicht nach dem Knoten u_k ausgegeben werden. ■

Satz 2.42. *Jede LBO für einen chordalen Graphen G ist eine PEO für G.*

Beweis. Sei $(u_1, \ldots, u_n)$ eine LBO für $G = (V, E)$ und sei $A = (a_{ij})$ die Adjazenzmatrix von G mit $a_{ij} = 1 \Leftrightarrow \{u_i, u_j\} \in E$, wobei wir für a_{ij} auch $A[i, j]$ schreiben. Wir zeigen, dass G nicht chordal ist, wenn u_i nicht simplizial in $G_i = G[u_1, \ldots, u_i]$ ist.

Falls u_i nicht simplizial in G_i ist, müssen Indizes $i_2 < i_1 < i =: i_0$ mit $A[i_0, i_1] = A[i_0, i_2] = 1$ und $A[i_1, i_2] = 0$ existieren. Wegen $A[i_1, i_2] = 0$ und $A[i_0, i_2] = 1$ muss es einen Index $i_3 < i_2$ geben mit $A[i_1, i_3] = 1$ und $A[i_0, i_3] = 0$, wobei wir i_3 möglichst klein wählen.

Falls nun $A[i_2, i_3] = 1$ ist, haben wir einen induzierten Kreis $G[u_{i_0}, u_{i_1}, u_{i_2}, u_{i_3}] = (u_{i_0}, u_{i_1}, u_{i_3}, u_{i_2})$ der Länge 4 in G gefunden. Andernfalls muss es wegen $A[i_2, i_3] = 0$ und $A[i_1, i_3] = 1$ einen Index

$i_4 < i_3$ geben mit $A[i_2, i_4] = 1$ und $A[i_1, i_4] = 0$, wobei wir i_4 wieder möglichst klein wählen. Da spätestens für $i_k = 1$ kein Index $i_{k+1} < i_k$ existiert, also $A[i_{k-1}, i_k] = 1$ sein muss, erhalten wir eine Indexfolge $1 \leq i_k < \cdots < i_1 < i_0$ mit

(a) $A[i_0, i_1] = A[i_j, i_{j+2}] = A[i_{k-1}, i_k] = 1$ für $j = 0, \ldots, k-2$ und

(b) $A[i_0, i_3] = A[i_j, i_{j+1}] = A[i_j, i_{j+3}] = A[i_{k-2}, i_{k-1}] = 0$ für $j = 1, \ldots, k-3$ und

(c) $A[i_j, l] = A[i_{j-1}, l]$ für $j = 1, \ldots, k-3$ und $l < i_{j+2}$.

Die Eigenschaften (a) und (b) ergeben sich direkt aus der Konstruktion der Folge. Eigenschaft (c) folgt aus der minimalen Wahl der Indizes $i_3, \ldots, i_k$ und impliziert für $r = 4, \ldots, k$ die Gleichungen $A[i_0, i_r] = A[i_1, i_r] = \cdots = A[i_{r-3}, i_r]$, indem wir $j = 1, \ldots, r-3$ und $l = i_r$ setzen. Da zudem $A[i_{r-3}, i_r]$ gemäß Eigenschaft (b) für $r = 3, \ldots, k$ den Wert 0 hat, folgt für alle Paare $0 \leq j < r \leq k$ die Äquivalenz

$$A[i_j, i_r] = 1 \Leftrightarrow r = j + 2 \text{ oder } j = 0 \wedge r = 1 \text{ oder } j = k - 1 \wedge r = k.$$

Folglich ist $G[u_{i_0}, \ldots, u_{i_k}]$ ein Kreis der Länge $k + 1 \geq 4$. ■

Damit haben wir einen Linearzeitalgorithmus, der für chordale Graphen eine PEO berechnet. Da auch `greedy-color` linear zeitbeschränkt ist, können wir den Algorithmus `chordal-color` in Linearzeit implementieren. Diesen Algorithmus können wir leicht noch so modifizieren, dass er zusammen mit der gefundenen k-Färbung entweder eine Clique C der Größe k (als Zertifikat, dass $\chi(G) = k = \omega(G)$ ist) oder einen induzierten Kreis der Länge ≥ 4 (als Zertifikat, dass G nicht chordal ist) ausgibt.

2.3 Der Satz von Brooks

Satz 2.43 (Brooks 1941)**.** *Für einen zusammenhängenden Graphen G gilt $\chi(G) = \Delta(G) + 1$ genau dann, wenn $G = K_n$ für ein $n \geq 1$ oder $G = C_n$ für ein ungerades $n \geq 3$ ist.*

Beweis. Es ist klar, dass die Graphen $G = K_n$ für $n \geq 1$ und $G = C_n$ für ungerades $n \geq 3$ die chromatische Zahl $\Delta(G) + 1$ haben.

Um zu zeigen, dass dies die einzigen zusammenhängenden Graphen mit $\chi(G) = \Delta(G) + 1$ sind, betrachten wir verschiedene Fälle.

Falls G nicht regulär ist, können wir ausgehend von einem Knoten u_1 vom Grad $\deg_G(u_1) < \Delta(G)$ eine Suchordnung $(u_1, \ldots, u_n)$ berechnen und G greedy in der umgekehrten Reihenfolge $(u_n, \ldots, u_1)$ $\Delta(G)$-färben. Dies ist möglich, da jeder Knoten u_i mit $i \geq 2$ zum Zeitpunkt der Berechnung von $c(u_i)$ noch einen ungefärbten Nachbar `parent`(u_i) und u_1 einen Grad $\deg_G(u_1) < \Delta(G)$ hat.

Falls G regulär, aber nicht 2-zusammenhängend ist, berechnen wir $\Delta(G)$-Färbungen c_i für die einzelnen Blöcke B_i von G. Dies ist möglich, da jeder Block B_i mindestens einen Schnittknoten enthält und daher höchstens für ein $k < \Delta(G)$ k-regulär ist. Die Färbungen c_i lassen sich ausgehend von einem beliebigen Wurzelblock des BC-Baums hin zu den Blattblöcken in eine $\Delta(G)$-Färbung c für G transformieren. Hierzu müssen wir lediglich die Farben im aktuellen Block B_i so umbenennen, dass der Schnittknoten, der B_i mit seinem Elternblock verbindet, die vorgegebene Farbe erhält.

Es bleibt also der Fall, dass G d-regulär und $\kappa(G) \geq 2$ ist. Der Fall $d = 2$ ist klar, da G ein Kreis sein muss. Für den Fall $d \geq 3$ benutzen wir folgende Behauptung.

Behauptung 2.44. *Sei $d \geq 3$ und sei $G \neq K_n$ ein d-regulär Graph mit $\kappa(G) \geq 2$. Dann gibt es in G einen Knoten u_1, der zwei nichtadjazente Nachbarn a und b hat, so dass $G - \{a, b\}$ zusammenhängend ist.*

Da $G \neq K_n$ ist, gibt es einen Knoten x, der zwei Nachbarn $y, z \in N(x)$ mit $\{y, z\} \notin E$ hat. Wir betrachten folgende zwei Fälle.

- Falls $G - y$ 2-zusammenhängend ist, ist $G - \{y, z\}$ zusammenhängend und die Behauptung folgt für $u_1 = x$.
- Ist $G - y$ nicht 2-zusammenhängend, d.h. $G - y$ hat mindestens zwei Blöcke, dann hat der BC-Baum T von $G - y$ mindestens zwei Blätter. Da $\kappa(G) \geq 2$ ist, ist y in G zu mindestens einem Knoten in jedem Blatt von T benachbart, der kein Schnittknoten ist. Wählen wir für a und b zwei dieser Knoten in verschiedenen Blättern, so ist $G - \{a, b\}$ zusammenhängend und somit die Behauptung für $u_1 = y$ bewiesen.

Sei also u_1 ein Knoten, der zwei Nachbarn a und b mit $\{a, b\} \notin E$ hat, so dass $G - \{a, b\}$ zusammenhängend ist. Durchsuchen wir den Graphen $G - \{a, b\}$ ausgehend vom Startknoten u_1, so erhalten wir eine Suchordnung $(u_1, \ldots, u_{n-2})$. Starten wir nun `greedy-color` mit der Reihenfolge $(a, b, u_{n-2}, \ldots, u_1)$, so erhalten wir eine d-Färbung c für G mit $c(a) = c(b) = 1$. Zudem hat Knoten u_i, $i > 1$, einen Nachbarn u_j mit $j < i$, weshalb $c(u_i) \leq \deg(u_i) \leq d$ ist. Zuletzt erhält auch u_1 eine Farbe $c(u_1) \leq d$, da die Nachbarn a und b von u_1 dieselbe Farbe haben. ■

2.4 Kantenfärbungen

Neben der Frage, mit wievielen Farben die Knoten eines Graphen gefärbt werden können, muss bei vielen Anwendungen auch eine Kantenfärbung mit möglichst wenigen Farben gefunden werden.

Definition 2.45. *Sei $G = (V, E)$ ein Graph und sei $k \in \mathbb{N}$.*

*a) Eine Abbildung $c\colon E \to \mathbb{N}$ heißt **Kantenfärbung** von G, wenn $c(e) \neq c(e')$ für alle $e \neq e' \in E$ mit $e \cap e' \neq \emptyset$ gilt.*

*b) G heißt **k-kantenfärbbar**, falls eine Kantenfärbung $c\colon E \to \{1, \ldots, k\}$ existiert.*

*c) Die **kantenchromatische Zahl** oder der **chromatische Index** von G ist*

$$\chi'(G) = \min\{k \in \mathbb{N} \mid G \text{ ist } k\text{-kantenfärbbar}\}.$$

Eine k-Kantenfärbung $c\colon E \to \mathbb{N}$ muss also zwei Kanten, die einen Knoten gemeinsam haben, verschiedene Farben zuweisen. Daher bildet jede ***Farbklasse*** $E_i = \{e \in E \mid f(e) = i\}$ ein Matching von G, d.h. c zerlegt E in k disjunkte Matchings. Umgekehrt liefert jede Zerlegung von E in k disjunkte Matchings eine k-Kantenfärbung von G.

Neben Graphen treten in manchen Anwendungen auch ***Multigraphen*** $G = (V, E)$ auf. Diese können mehr als eine Kante zwischen zwei Knoten haben, d.h. E ist eine Multimenge auf $\binom{V}{2}$.

Eine ***Multimenge*** A auf einer Grundmenge M lässt sich durch eine Funktion $v_A\colon M \to \mathbb{N}$ beschreiben, wobei $v_A(a)$ die Anzahl der Vorkommen des Elements a in A angibt. Die Mächtigkeit von A ist $|A| = \sum_{a \in A} v_A(a)$.

Wie bei Graphen gehen wir davon aus, dass jede Kante $e = \{u, v\}$ eines Multigraphen $G = (V, E)$ zwei verschiedene Endpunkte $u \neq v$ hat, d.h. G ist schlingenfrei. In G gibt es genau $v_E(e) = v_E(u, v)$ Kanten zwischen den beiden Knoten u und v. Die Zahl $v_E(e)$ wird auch als ***(Kanten-)Vielfachheit*** von e bezeichnet. Ein wichtiger Parameter von Multigraphen ist die maximale Kantenvielfachheit

$$v(G) = \max_{e \in E} v_E(e),$$

die auch als ***(Graph-)Vielfachheit*** von G bezeichnet wird. Der ***Grad*** eines Knotens $u \in V$ ist $\deg_G(u) = \sum_{v \in N(u)} v_E(u, v)$ und der ***Maximalgrad*** von G ist wie üblich $\Delta(G) = \max_{u \in V} \deg_G(u)$.

Eine k-Kantenfärbung für einen Multigraphen $G = (V, E)$ lässt sich durch eine Funktion c beschreiben, die jeder Kante $e \in \binom{V}{2}$ eine Menge $c(e) \subseteq \{1, \ldots, k\}$ von $|c(e)| = v_E(e)$ Farben zuordnet, so dass $c(e) \cap c(e') = \emptyset$ für alle $e \neq e' \in \binom{V}{2}$ mit $e \cap e' \neq \emptyset$ gilt.

Beispiel 2.46.

$$\chi'(C_n) = \begin{cases} 2, & n \text{ gerade,} \\ 3, & \text{sonst,} \end{cases}$$

$$\chi'(K_n) = 2\lceil n/2 \rceil - 1 = \begin{cases} n-1, & n \text{ gerade,} \\ n, & \text{sonst.} \end{cases}$$

Das Kantenfärbungsproblem für einen Graphen G lässt sich leicht auf das Knotenfärbungsproblem für einen Graphen G' reduzieren.

Definition 2.47. *Sei $G' = (V, E')$ ein Graph mit $m \geq 1$ Kanten. Dann heißt der Graph $G = L(G') = (E', E)$ mit*

$$E = \left\{ \{e, e'\} \in \binom{E'}{2} \;\middle|\; e \cap e' \neq \emptyset \right\}$$

*der **Kantengraph** oder **Line-Graph** von G'.*

Ist G' ein Multigraph, so verwenden wir als Knotenmenge von $L(G')$ eine Menge $V_{E'}$ mit der Mächtigkeit $|V_{E'}| = |E'|$, die $v_{E'}(e)$ verschiedene Kopien $e^1, \ldots, e^{v_E(e)}$ jeder Kante $e \in E'$ enthält. Die folgenden Beziehungen zwischen einem (Multi-)Graphen G' und dem zugehörigen Line-Graphen lassen sich leicht verifizieren.

Proposition 2.48. *Für den Line-Graphen $G' = (V', E') = L(G)$ eines Multigraphen $G = (V, E)$ gilt:*

(i) $n(G') = m(G)$,

(ii) $\chi(G') = \chi'(G)$,

(iii) $\alpha(G') = \mu(G)$,

(iv) $\omega(G') \geq \Delta(G)$,

(v) Für jede Kante $e = \{u, v\} \in E$ gilt $\deg_{G'}(e) = \deg_G(u) + \deg_G(v) - v_E(e) - 1$ und somit ist $\Delta(G') \leq 2\Delta(G) - 2$.

Damit erhalten wir aus den Abschätzungen $\omega(G') \leq \chi(G') \leq \Delta(G')+1$ und $n/\alpha(G') \leq \chi(G') \leq n - \alpha(G') + 1$ die folgenden Abschätzungen für $\chi'(G)$.

Lemma 2.49. *Für jeden Multigraphen G mit $m \geq 1$ Kanten gilt $\Delta \leq \chi' \leq 2\Delta - 1$ und $m/\mu \leq \chi' \leq m - \mu + 1$.*

Korollar 2.50. *Für jeden regulären Multigraphen mit einer ungeraden Knotenzahl und $m \geq 1$ Kanten gilt $\chi' \geq \Delta + 1 \geq 3$.*

Beweis. Wegen $\mu \leq (n-1)/2$ und $m = n\Delta/2$ folgt $\chi' \geq m/\mu \geq n\Delta/(n-1) > \Delta$. Da n ungerade und $m \geq 1$ ist, folgt $\Delta \geq 2$. ∎

Als nächstes geben wir einen effizienten Algorithmus an, der für jeden Graphen eine $(\Delta + 1)$-Kantenfärbung berechnet. Hierfür benötigen wir folgende Begriffe.

Definition 2.51. *Sei $G = (V, E)$ ein Graph und sei $c\colon E \to \{1, \ldots, k\}$ eine k-Kantenfärbung von G. Weiter sei $F \subseteq \{1, \ldots, k\}$ und es gelte $1 \leq i \neq j \leq k$.*

*a) Ein Nachbar v von u heißt **F-Nachbar** von u, wenn $c(u, v) \in F$ ist. Im Fall $F = \{i\}$ nennen wir v auch den **i-Nachbarn** von u.*

*b) Die Farbe i ist **frei** an einem Knoten u (kurz **$i \in$ frei(u)**), falls u keinen i-Nachbarn hat.*

*c) Der Graph $G_{ij} = (V, E_{ij})$ mit $E_{ij} = \left\{ e \in E \;\middle|\; c(e) \in \{i, j\} \right\}$ heißt **(i, j)-Subgraph** von G.*

*d) Jede Komponente K von G_{ij} heißt **(i, j)-Komponente** von G. Je nachdem ob K ein Pfad oder ein Kreis ist, nennen wir K auch einen **(i, j)-Pfad** bzw. **(i, j)-Kreis** in G (bzgl. c).*

Man sieht leicht, dass jede (i, j)-Komponente K von G entweder ein Pfad der Länge $l \geq 0$ oder ein Kreis gerader Länge ist. Zudem können wir aus c eine weitere k-Kantenfärbung c' von G gewinnen, indem wir die beiden Farben i und j entlang der Kanten von K vertauschen.

Satz 2.52 (Vizing 1964). *Für jeden Graphen G gilt* $\chi'(G) \leq \min_{e\in E} \Delta(G-e)+1 \leq \Delta(G)+1$.

Beweis. Wir führen Induktion über m. Der IA $m = 0$ ist klar. Für den IS sei $G' = (V, E')$ ein Graph mit $m+1$ Kanten und sei $k = \min_{e\in E'} \Delta(G'-e)+1$. Wir wählen eine beliebige Kante $e_1 = \{y_0, y_1\} \in E'$, so dass der Graph $G = G' - e_1$ den Maximalgrad $\Delta(G) = k-1$ hat. Dann hat G nach IV eine k-Kantenfärbung $c\colon E \to \{1,\dots,k\}$. Da zudem unter c an jedem Knoten u mindestens $k - \deg_G(u) \geq 1$ Farben frei sind, folgt $\mathit{free}(u) \neq \emptyset$ für alle $u \in V$. Betrachte nun folgende Prozeduren.

Prozedur `expand`(G, c, y_0, y_1)

```
1  ℓ := 1
2  wähle α_1 ∈ free(y_1)
3  while α_ℓ ∉ free(y_0) ∪ {α_1, ..., α_{ℓ-1}} do
4    sei y_{ℓ+1} der α_ℓ-Nachbar von y_0
5    wähle α_{ℓ+1} ∈ free(y_{ℓ+1})
6    ℓ := ℓ + 1
7  wähle 0 ≤ i < ℓ minimal mit α_ℓ ∈ free(y_0) ∪ {α_1, ..., α_i}
8  if i = 0 then // α_ℓ ∈ free(y_0)
9    recolor(ℓ, α_ℓ)
10 else // α_ℓ = α_i
11   wähle eine Farbe α_0 ∈ free(y_0)
12   berechne den (α_0, α_i)-Pfad P mit Startknoten y_ℓ und
13   vertausche dabei die Farben α_0 und α_i entlang P
14   sei z der Endknoten von P // z = y_ℓ ist möglich
15   case
16     z = y_0 : recolor(i, α_i)
17     z = y_i : recolor(i, α_0)
18   else recolor(ℓ, α_0)
19 return c
```

Prozedur `recolor`(i, α)

```
1  for j := 1 to i − 1 do c(y_0, y_j) := α_j
2  c(y_0, y_i) := α
```

Wir verifizieren, dass die Abbildung c eine Kantenfärbung von G' ist.

Fall 1 $i = 0$: Da die Farbe α_ℓ an y_0 und die Farbe α_j für $j = 1,\dots,\ell$ an y_j frei ist, kann `recolor`(ℓ, α_ℓ) die Kanten $\{y_0, y_j\}$ mit α_j färben.

Fall 2 $i > 0 \wedge z = y_0$: In diesem Fall erreicht P den Knoten $z = y_0$ über die Kante $\{y_0, y_{i+1}\}$. Nach dem Vertauschen von α_0 und α_i entlang P hat diese Kante dann die Farbe α_0, weshalb `recolor`(i, α_i) die Kanten $\{y_0, y_j\}$ für $j = 1,\dots,i$ mit α_j färben kann.

Fall 3 $i > 0 \wedge z = y_i$: Da $\alpha_i \in \mathit{free}(y_i) \cap \mathit{free}(y_\ell)$ ist, müssen die Endkanten von P mit α_0 gefärbt sein. Nach Vertauschen von α_0 und α_i entlang P ist daher die Farbe α_0 an y_0 und y_i frei, weshalb `recolor`(i, α_0) die Kanten $\{y_0, y_j\}$ für $j = 1,\dots,i-1$ mit α_j und die Kante $\{y_0, y_i\}$ mit α_0 färben kann.

Fall 4 $i > 0 \wedge z \notin \{y_0, y_i\}$: Da die Farbe α_0 durch Vertauschen von α_0 und α_i entlang P an y_ℓ frei wird und zudem die Farbe α_j für $j = 0,\dots,\ell-1$ wegen $z \notin \{y_0, y_i\}$ und $\alpha_j \notin \{\alpha_0, \alpha_i\}$ für $j \notin \{0, i\}$ an y_j frei bleibt, kann `recolor`(ℓ, α_0) die Kanten $\{y_0, y_j\}$ für $j = 1,\dots,\ell-1$ mit α_j und die Kante $\{y_0, y_\ell\}$ mit α_0 färben. ■

Da die Prozedur `expand` mit Hilfe geeigneter Datenstrukturen so implementiert werden kann, dass jeder Aufruf Zeit $\mathcal{O}(n)$ erfordert, und diese Prozedur m-mal aufgerufen wird, um alle m Kanten eines gegebenen Graphen G zu färben, ergibt sich eine Gesamtlaufzeit von $\mathcal{O}(nm)$.

Für einen Graphen G kann $\chi'(G)$ nur einen der beiden Werte $\Delta(G)$ oder $\Delta(G)+1$ annehmen. Graphen G mit $\chi'(G) = \Delta(G)$ heißen ***Klasse 1*** und Graphen G mit $\chi'(G) = \Delta(G)+1$ heißen ***Klasse 2***.

Neben den vollständigen Graphen K_n mit gerader Knotenzahl n sind alle Graphen G mit $\Delta(G) \leq 1$ und alle bipartiten Graphen Klasse 1.

Zudem sind alle planaren Graphen G mit $\Delta(G) \geq 7$ Klasse 1. Für $2 \leq d \leq 5$ existieren planare Graphen G mit $\Delta(G) = d$, die Klasse 2 sind. Für $d = 6$ ist dies offen.

Das Problem, für einen gegebenen Graphen G zu entscheiden, ob er Klasse 1 ist (also $\chi'(G) \leq \Delta(G)$ gilt), ist NP-vollständig.

Der Satz von Vizing lässt sich wie folgt auf Multigraphen verallgemeinern.

Satz 2.53 (Vizing 1964)**.** *Für jeden Multigraphen $G = (V, E)$ gilt $\chi'(G) \leq \max_{u,v \in V}(\deg(u) + v_E(u, v)) \leq \Delta(G) + v(G)$.*

Beweis. Der Beweis folgt derselben Argumentation wie bei einfachen Graphen. Wir müssen nur die Prozeduren `expand` und `recolor` entsprechend anpassen.

Prozedur `multiexpand`(G, c, y_0, y_1)

```
1  ℓ := 1; Y := {y_1}
2  wähle α_1 ∈ free(y_1); used(y_1) := {α_1}
3  while α_ℓ ∉ free(y_0) ∪ {α_1, ..., α_{ℓ-1}} do
4    sei y_{ℓ+1} der α_ℓ-Nachbar von y_0;
5    if y_{ℓ+1} ∉ Y then Y := Y ∪ {y_{ℓ+1}}; used(y_{ℓ+1}) := ∅
6    wähle α_{ℓ+1} ∈ free(y_{ℓ+1}) \ used(y_{ℓ+1})
7    used(y_{ℓ+1}) := used(y_{ℓ+1}) ∪ {α_{ℓ+1}}
8    ℓ := ℓ + 1
9  wähle 0 ≤ i < ℓ minimal mit α_ℓ ∈ free(y_0) ∪ {α_1, ..., α_i}
10 if i = 0 then      // α_ℓ ∈ free(y_0)
11   multirecolor(ℓ, α_ℓ)
12 else               // α_ℓ = α_i für ein i ≥ 1
13   wähle eine Farbe α_0 ∈ free(y_0)
14   berechne den (α_0, α_i)-Pfad P mit Startknoten y_ℓ und
15   vertausche dabei die Farben α_0 und α_i entlang P
16   sei z der Endknoten von P // z = y_ℓ ist möglich
17   case
18     z = y_0: multirecolor(i, α_i)
19     z = y_i: multirecolor(i, α_0)
20   else multirecolor(ℓ, α_0)
```

Prozedur `multirecolor`(i, α)

```
1  for j := 1 to i − 1 do c(y_0, y_j) := (c(y_0, y_j) \ {α_{j-1}}) ∪ {α_j}
2  c(y_0, y_i) := (c(y_0, y_i) \ {α_{i-1}}) ∪ {α}
```

■

In den Übungen zeigen wir noch folgende Schranken.

Korollar 2.54. *Sei G ein Multigraph. Dann gilt*

(i) $\chi'(G) \leq 3\Delta(G)/2$ *und*

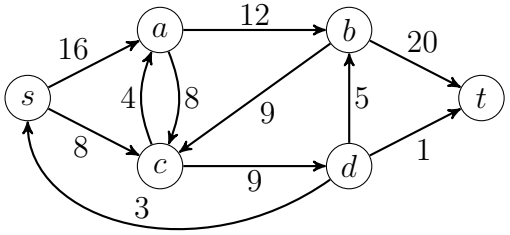
(ii) $\chi'(G) = \Delta(G)$*, falls G bipartit (also $\chi(G) \leq 2$) ist.*

Man beachte, dass für jeden Multigraphen die Ungleichungen $\Delta(G) \leq \Delta(G) + v(G) \leq 2\Delta(G)$ gelten.

3 Flüsse in Netzwerken

Definition 3.1. *Ein **Netzwerk** $N = (V, E, s, t, c)$ besteht aus einem gerichteten Graphen $G = (V, E)$ mit einer **Quelle** $s \in V$ und einer **Senke** $t \in V$ sowie einer **Kapazitätsfunktion** $c : V \times V \to \mathbb{N}$. Zudem muss jede Kante $(u, v) \in E$ positive Kapazität $c(u, v) > 0$ und jede Nichtkante $(u, v) \notin E$ muss die Kapazität $c(u, v) = 0$ haben.*

Beispiel 3.2. *Die Abbildung zeigt ein Netzwerk N:*



◁

Definition 3.3.

*a) Ein **Fluss in N** ist eine Funktion $f : V \times V \to \mathbb{Z}$ mit*

$f(u, v) \leq c(u, v)$, *(Kapazitätsbedingung)*

$f(u, v) = -f(v, u)$ *und* *(Antisymmetrie)*

$\sum_{v \in V} f(u, v) = 0$ *für alle* $u \in V \setminus \{s, t\}$. *(Kontinuität)*

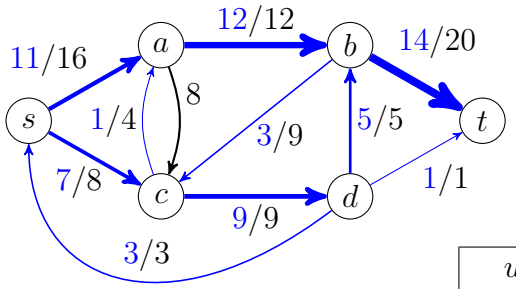
*b) Die **Größe von f** ist* $|f| = \sum_{v \in V} f(s, v)$.

*c) Der **Fluss in den Knoten u** ist* $f^-(u) = \sum_{v \in V} \max\{0, f(v, u)\}$.

*d) Der **Fluss aus u** ist* $f^+(u) = \sum_{v \in V} \max\{0, f(u, v)\}$.

Die Antisymmetrie impliziert, dass $f(u, u) = 0$ für alle $u \in V$ gilt und $|f| = f^+(s) - f^-(s)$ ist. Wir können also annehmen, dass $c(u, u) = 0$ für alle Knoten $u \in V$ gilt und somit G schlingenfrei ist. Die Kontinuität besagt, dass $f^+(u) = f^-(u)$ für alle Knoten $u \in V \setminus \{s, t\}$ gilt.

Beispiel 3.4 (Fortsetzung)**.** *Die Abbildung zeigt einen Fluss f der Größe $|f| = \sum_{v \in V} f(u, v) = 11 + 7 - 3 = 15$ in N:*



u	s	a	b	c	d	t
$f^+(u)$	18	12	17	10	9	0
$f^-(u)$	3	12	17	10	9	15

◁

3.1 Der Ford-Fulkerson-Algorithmus

Wie kann man für einen Fluss f in einem Netzwerk N entscheiden, ob er vergrößert werden kann? Diese Frage ist leicht zu beantworten, falls f auf $V \times V$ den Wert 0 hat: In diesem Fall genügt es, in $G = (V, E)$ einen Pfad von s nach t zu finden. Andernfalls können wir zu N und f ein Netzwerk N_f konstruieren, so dass f genau dann vergrößert werden kann, wenn sich in N_f der Nullfluss vergrößern lässt.

Definition 3.5. *Sei $N = (V, E, s, t, c)$ ein Netzwerk und sei f ein Fluss in N. Das zugeordnete **Restnetzwerk** ist $N_f = (V, E_f, s, t, c_f)$ mit den Kapazitäten*

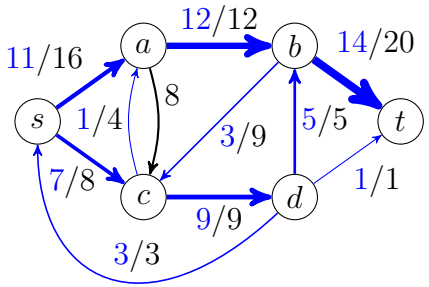
$$c_f(e) = c(e) - f(e)$$

für alle $e \in V \times V$ und der Kantenmenge

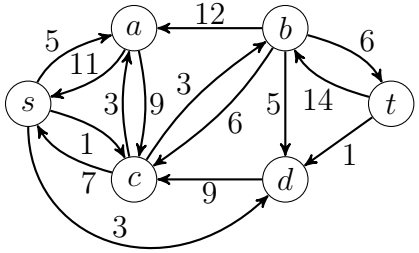
$$E_f = \{e \in V \times V \mid c_f(e) > 0\}.$$

Beispiel 3.6 (Fortsetzung). *Der Fluss f führt auf folgendes Restnetzwerk N_f für N:*

Fluss f in N:



Restnetzwerk N_f:



◁

Definition 3.7. *Sei $N = (V, E, s, t, c)$ ein Netzwerk. Dann heißt jeder s-t-Pfad P in (V, E)* ***Zunahmepfad in N****. Die* ***Kapazität von P in N*** *ist*

$$c(P) = \min\{c(u, v) \mid (u, v) \text{ liegt auf } P\}$$

und der ***Fluss durch P in N*** *ist*

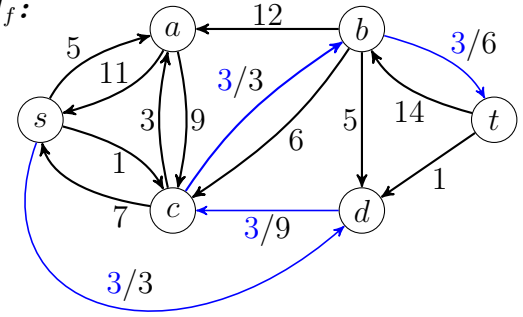
$$f_P(u, v) = \begin{cases} c(P), & (u, v) \text{ liegt auf } P, \\ -c(P), & (v, u) \text{ liegt auf } P, \\ 0, & \text{sonst.} \end{cases}$$

Es ist leicht zu sehen, dass f_P tatsächlich ein Fluss in N ist. $P = (u_0, \ldots, u_k)$ ist also genau dann ein Zunahmepfad in N_f, falls

- $u_0 = s$ und $u_k = t$ ist,
- die Knoten $u_0, \ldots, u_k$ paarweise verschieden sind
- und $c(u_i, u_{i+1}) > 0$ für $i = 0, \ldots, k-1$ gilt.

Beispiel 3.8 (Fortsetzung). *Die Abbildung zeigt den zum Zunahmepfad $P = (s, d, c, b, t)$ gehörigen Fluss f_P im Restnetzwerk N_f:*

Fluss f_P in N_f:

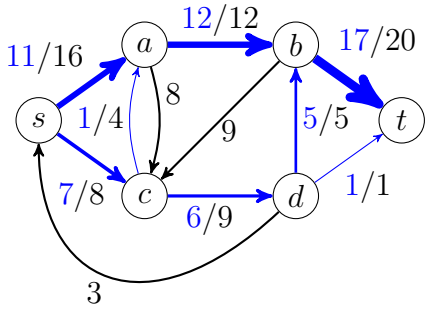


Die Kapazität von P in N_f ist $c_f(P) = 3$. ◁

Durch Addition der beiden Flüsse f und f_P erhalten wir einen Fluss $f' = f + f_P$ in N der Größe $|f'| = |f| + |f_P| = |f| + c_f(P) > |f|$.

Beispiel 3.9 (Fortsetzung). *Durch Addition von f und f_P erhalten wir folgenden Fluss f' in N:*

Fluss $f' = f + f_P$ in N:

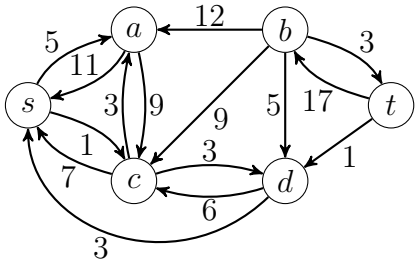


Nun können wir den **Ford-Fulkerson-Algorithmus** angeben.

Algorithmus `Ford-Fulkerson`(V, E, s, t, c)

```
1  for all (u, v) ∈ E ∪ E^R do
2    f(u, v) := 0
3  while es gibt einen Zunahmepfad P in N_f do
4    f := f + f_P
```

Beispiel 3.10 (Fortsetzung). *Für den neuen Fluss f' erhalten wir nun folgendes Restnetzwerk $N_{f'}$:*



In diesem existiert kein Zunahmepfad mehr. ◁

Um zu zeigen, dass der Algorithmus von Ford-Fulkerson tatsächlich einen Maximalfluss berechnet, weisen wir nach, dass f ein Fluss maximaler Größe in N ist, wenn im Restnetzwerk N_f kein Zunahmepfad existiert. Hierzu benötigen wir den Begriff des Schnitts.

Definition 3.11. *Sei $N = (V, E, s, t, c)$ ein Netzwerk. Eine Menge S mit $\emptyset \subsetneq S \subsetneq V$ heißt **Schnitt** durch N. Der zugehörige **Kantenschnitt** ist $E(S) = \{(u, v) \in E \mid u \in S, v \notin S\}$ (dieser wird oft auch einfach als **Schnitt** bezeichnet). Die **Kapazität eines Schnittes** S ist*

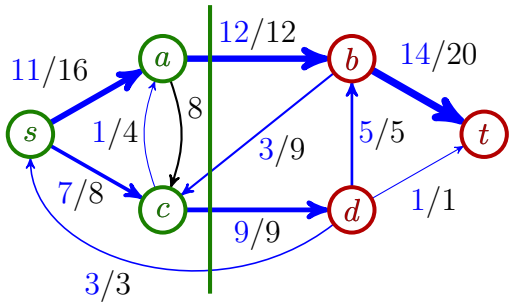
$$c(S) = \sum_{e \in E(S)} c(e).$$

Ist f ein Fluss in N, so heißt

$$f(S) = \sum_{e \in E(S)} f(e)$$

*der **Nettofluss** (oder einfach **Fluss**) **durch den Schnitt** S. Ist $u \in S$ und $v \notin S$, so wird S auch als **u-v-Schnitt** bezeichnet.*

Beispiel 3.12. *Betrachte folgenden Schnitt $S = \{s, a, c\}$ durch das Netzwerk N mit dem Fluss f:*



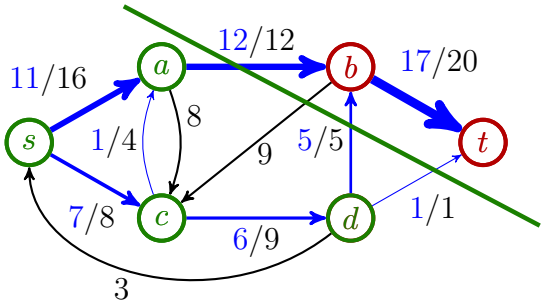
Dieser Schnitt hat die Kapazität

$$c(S) = c(a, b) + c(c, d) = 12 + 9 = 21$$

und der Fluss f durch ihn ist

$$\begin{aligned} f(S) &= f(a, b) + f(c, b) + f(c, d) + f(s, d) \\ &= 12 - 3 + 9 - 3 \\ &= 15. \end{aligned}$$

Dagegen hat der Schnitt $S' = \{s, a, c, d\}$



die Kapazität

$$c(S') = c(a, b) \ + \ c(d, b) \ + \ c(d, t) = 12 + 5 + 1 = 18$$

die mit dem Fluss

$$f(S') = f(a, b) + f(d, b) + f(d, t) = 12 + 5 + 1 = 18$$

durch ihn übereinstimmt. ◁

Lemma 3.13. *Für jeden Fluss f in einem Netzwerk N und jeden s-t-Schnitt S durch N gilt*

$$|f| = f(S) \leq c(S).$$

Beweis. Wir zeigen zuerst die Ungleichung $f(S) \leq c(S)$. Wegen $f(e) \leq c(e)$ für alle $e \in V \times V$ gilt

$$f(S) = \sum_{e \in E(S)} f(e) \leq \sum_{e \in E(S)} c(e) = c(S).$$

Die Gleichheit $|f| = f(S)$ zeigen wir durch Induktion über $k = |S|$.
$k = 1$: In diesem Fall ist $S = \{s\}$ und wegen $f(s, s) = 0$ folgt

$$|f| = \sum_{v} f(s, v) = \sum_{v \neq s} f(s, v) = f(S).$$

$k \rightsquigarrow k + 1$: Sei S ein s-t-Schnitt mit $|S| = k > 1$ und sei $w \in S - \{s\}$. Für den Schnitt $S' = S - \{w\}$ gilt dann nach IV $|f| = f(S')$. Wegen $S = S' \cup \{w\}$ folgt dann

$$f(S) = \sum_{u \in S, v \notin S} f(u, v) = \sum_{u \in S', v \notin S} f(u, v) + \sum_{v \notin S} f(w, v)$$

und wegen $V \setminus S' = (V \setminus S) \cup \{w\}$ folgt

$$f(S') = \sum_{u \in S', v \notin S'} f(u, v) = \sum_{u \in S', v \notin S} f(u, v) + \sum_{u \in S'} f(u, w).$$

Daher erhalten wir

$$f(S) - f(S') = \sum_{v \notin S} f(w, v) - \sum_{u \in S'} \underbrace{f(u, w)}_{= -f(w,u)} = \sum_{v \neq w} f(w, v) = 0$$

und somit $f(S) = f(S') = |f|$. ∎

Satz 3.14 (Max-Flow-Min-Cut-Theorem)**.** *Für einen Fluss f in einem Netzwerk $N = (V, E, s, t, c)$ sind folgende Aussagen äquivalent:*

1. *f ist maximal, d.h. für jeden Fluss f' in N gilt $|f'| \leq |f|$.*
2. *Im Restnetzwerk N_f existiert kein Zunahmepfad.*
3. *Es gibt einen s-t-Schnitt S durch N mit $c(S) = |f|$.*

Beweis. Die Implikation ***1*** ⇒ ***2*** ist klar, da die Existenz eines Zunahmepfads in N_f zu einer Vergrößerung von f führen würde.
Für die Implikation ***2*** ⇒ ***3*** betrachten wir den Schnitt

$$S = \{u \in V \mid u \text{ ist in } N_f \text{ von } s \text{ aus erreichbar}\}.$$

Dann gilt $t \notin S$ (da in N_f kein Zunahmepfad existiert) und für alle Kanten $e = (u, v) \in E(S)$ ist $c_f(e) = c(e) - f(e) = 0$ (sonst wäre mit u auch v in S enthalten). Daher folgt

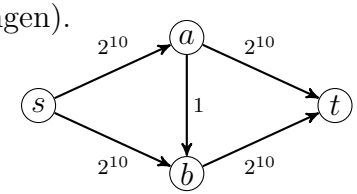
$$|f| = f(S) = \sum_{e \in E(S)} f(e) = \sum_{e \in E(S)} c(e) = c(S).$$

Die Implikation ***3*** ⇒ ***1*** folgt direkt aus obigem Lemma, da jeder Fluss f' in N im Fall $c(S) = |f|$ einen Wert $|f'| = f'(S) \leq c(S) = |f|$ hat. ∎

Das Max-Flow-Min-Cut-Theorem gilt auch für Netzwerke mit beliebigen reellen Kapazitäten $c(e) \geq 0$.
Sei $c_0 = c(S)$ die Kapazität des Schnittes $S = \{s\}$. Dann durchläuft der Ford-Fulkerson-Algorithmus die while-Schleife höchstens c_0-mal, da sich der aktuelle Fluss in jedem Durchlauf um mindestens 1 erhöht. Bei jedem Durchlauf ist zuerst das Restnetzwerk N_f und danach ein Zunahmepfad in N_f zu berechnen.

- Die Berechnung des Zunahmepfads P kann durch Breitensuche in Zeit $\mathcal{O}(n + m)$ erfolgen.
- Da sich das Restnetzwerk nur entlang von P ändert, kann es in Zeit $\mathcal{O}(n)$ aktualisiert werden.

Jeder Durchlauf benötigt also Zeit $\mathcal{O}(n+m)$, was auf eine Gesamtlaufzeit von $\mathcal{O}(c_0(n+m))$ führt. Da der Wert von c_0 jedoch exponentiell in der Länge der Eingabe (also der Beschreibung des Netzwerkes N) sein kann, ergibt dies keine polynomiell beschränkte Laufzeit. Bei Netzwerken mit reellen Kapazitäten kann der Ford-Fulkerson-Algorithmus sogar unendlich lange laufen (siehe Übungen).

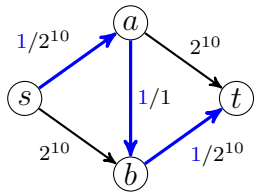
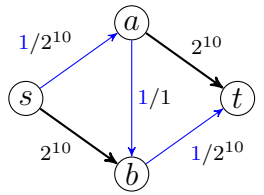
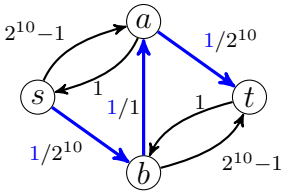
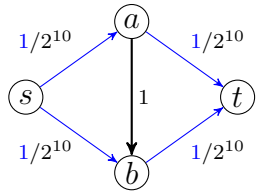
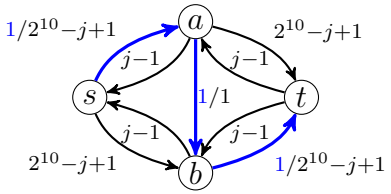
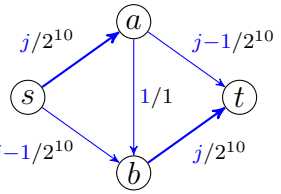
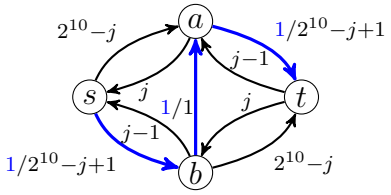
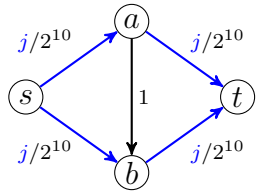
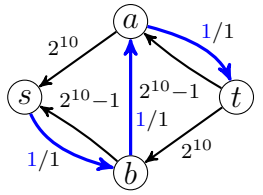
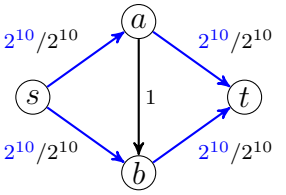


Bei nebenstehendem Netzwerk benötigt Ford-Fulkerson zur Bestimmung des Maximalflusses abhängig von der Wahl der Zunahmepfade zwischen 2 und 2^{11} Schleifendurchläufe.

- Im günstigsten Fall wird nämlich ausgehend vom Nullfluss f_0 zuerst der Zunahmepfad $P_1 = (s, a, t)$ mit der Kapazität 2^{10} und dann im Restnetzwerk N_{f_1} der Pfad $P_2 = (s, b, t)$ mit der Kapazität 2^{10} gewählt.
- Im ungünstigsten Fall werden abwechselnd die beiden Zunahmepfade $P_1 = (s, a, b, t)$ und $P_2 = (s, b, a, t)$ (also $P_i = P_1$ für ungerades i und $P_i = P_2$ für gerades i) mit der Kapazität 1 gewählt. Dies führt auf insgesamt 2^{11} Schleifendurchläufe (siehe nebenstehende Tabelle).

Nicht nur in diesem Beispiel lässt sich die exponentielle Laufzeit wie folgt vermeiden:

- Man betrachtet nur Zunahmepfade mit einer geeignet gewählten Mindestkapazität. Dies führt auf eine Laufzeit, die polynomiell in n, m und $\log c_0$ ist.
- Man bestimmt in jeder Iteration einen kürzesten Zunahmepfad im Restnetzwerk mittels Breitensuche in Zeit $\mathcal{O}(n+m)$. Diese Vorgehensweise führt auf den ***Edmonds-Karp-Algorithmus***, der eine Laufzeit von $\mathcal{O}(nm^2)$ hat (unabhängig von der Kapazitätsfunktion).
- Man bestimmt in jeder Iteration einen Fluss g im Restnetzwerk N_f, der nur Kanten benutzt, die auf einem kürzesten s-t-Pfad in N_f liegen. Zudem hat g die Eigenschaft, dass g auf jedem kürzesten

i	Fluss f_{P_i} in N_{f_i}	neuer Fluss f_{i+1} in N
1		
2		
⋮		
$2j-1$, $1 < j \leq 2^{10}$		
$2j$, $1 < j < 2^{10}$		
⋮		
2^{11}		

s-t-Pfad P mindestens eine Kante $e \in P$ ***sättigt*** (d.h. der Fluss $g(e)$ durch e schöpft die Restkapazität $c_f(e)$ von e vollkommen aus), weshalb diese Kante in der nächsten Iteration fehlt. Dies führt auf den ***Algorithmus von Dinitz***. Da die Länge der kürzesten s-t-Pfade im Restnetzwerk in jeder Iteration um mindestens eins zunimmt, liegt nach spätestens $n-1$ Iterationen ein maximaler Fluss vor. Dinitz hat gezeigt, dass der Fluss g in Zeit $\mathcal{O}(nm)$ bestimmt werden kann. Folglich hat der Algorithmus von Dinitz eine Laufzeit von $\mathcal{O}(n^2m)$.

- Malhotra, Kumar und Maheswari fanden später einen $\mathcal{O}(n^2)$-Algorithmus zur Bestimmung von g. Damit kann die Gesamtlaufzeit auf $\mathcal{O}(n^3)$ verbessert werden.

3.2 Der Edmonds-Karp-Algorithmus

Der Edmonds-Karp-Algorithmus ist eine spezielle Form von Ford-Fulkerson, die möglichst kurze Zunahmepfade benutzt. Diese können mittels Breitensuche bestimmt werden.

Algorithmus `Edmonds-Karp`(V, E, s, t, c)

```
1  for all (u,v) ∈ E ∪ E^R do
2    f(u,v) := 0
3  while P := zunahmepfad(f) ≠ ⊥ do
4    addierepfad(f, P)
```

Prozedur `zunahmepfad`(f)

```
1  for all v ∈ V \ {s} do
2    parent(v) := ⊥
3  parent(s) := s
4  Q := ()
5  while Q ≠ () ∧ parent(t) = ⊥ do
6    u := dequeue(Q)
7    for all e = (u,v) ∈ E ∪ E^R do
8      if c(e) − f(e) > 0 ∧ parent(v) = ⊥ then
9        c'(e) := c(e) − f(e)
10       parent(v) := u
11       enqueue(Q, v)
12 if parent(t) = ⊥ then
13   P := ⊥
14 else
15   P := parent-Pfad von s nach t
16   c_f(P) := min{c'(e) | e ∈ P}
17 return P
```

Prozedur `addierepfad`(f, P)

```
1  for all e ∈ P do
2    f(e) := f(e) + c_f(P)
3    f(e^R) := f(e^R) − c_f(P)
```

Die Prozedur `zunahmepfad(`f`)` berechnet im Restnetzwerk N_f einen (gerichteten) s-t-Pfad P, sofern ein solcher existiert. Dies ist genau dann der Fall, wenn die while-Schleife mit `parent`$(t) \neq \bot$ abbricht. Der gefundene Zunahmepfad $P = (u_\ell, \ldots, u_0)$ lässt sich dann ausgehend von $u_0 = t$ mittels `parent` zurückverfolgen:

$$u_i = \begin{cases} t, & i = 0, \\ \texttt{parent}(u_{i-1}), & i > 0 \text{ und } u_{i-1} \neq s \end{cases}$$

wobei $\ell = \min\{i \geq 1 \mid u_i = s\}$ ist. Dann ist $u_\ell = s$ und P ein s-t-Pfad, den wir als den ***parent-Pfad*** von s nach t bezeichnen.

Satz 3.15. *Der Edmonds-Karp-Algorithmus durchläuft die while-Schleife höchstens $(nm/2)$-mal und hat somit eine Laufzeit von $O(nm^2)$.*

Beweis. Sei k die Anzahl der Schleifendurchläufe und seien $P_1, \ldots, P_k$ die Zunahmepfade, die der Algorithmus bei Eingabe N berechnet,

d.h. $f_{i+1} = f_i + f_{P_{i+1}}$, wobei f_0 der triviale Nullfluss und P_{i+1} der im Restnetzwerk N_{f_i} berechnete Zunahmepfad ist. Eine Kante e auf P_{i+1} heißt ***kritisch*** für P_{i+1}, falls der Fluss $f_{P_{i+1}}$ durch P_{i+1} in N_{f_i} die Kante e ***sättigt***, d.h. $f_{P_{i+1}}(e) = c_{f_i}(e)$. Eine kritische Kante e für P_{i+1} ist wegen

$$c_{f_{i+1}}(e) = c(e) - f_{i+1}(e) = c(e) - (f_i + f_{P_{i+1}}) = c_{f_i}(e) - f_{P_{i+1}}(e) = 0$$

nicht in $N_{f_{i+1}}$ enthalten, wohl aber die Kante e^R, da $c_{f_{i+1}}(e^R) = c(e^R) - f_{i+1}(e^R) = c(e^R) + f_{i+1}(e) = c(e^R) + c(e) > 0$ ist.
Sei $d_i(u,v)$ die minimale Länge eines Pfades von u nach v im Restnetzwerk N_{f_i} und sei $\ell_{i+1} = d_i(s,t)$ die Länge von P_{i+1}. Wir zeigen zuerst, dass die Abstände jedes Knotens $u \in V$ von s und von t beim Übergang von N_{f_i} zu $N_{f_{i+1}}$ höchstens zu- aber nicht abnehmen. Hierzu beweisen wir für jeden kürzesten Pfad $P = (u_0, \ldots, u_\ell)$ von $u_0 = s$ nach $u_\ell = u$ in $N_{f_{i+1}}$ (d.h. $d_{i+1}(s,u) = \ell$) die Ungleichungen

$$d_i(s,u_h) \le d_i(s,u_{h-1}) + 1 \text{ für } h = 1, \ldots, \ell,$$

die $d_i(s,u) \le \ell$ implizieren. Falls die Kante $e = (u_{h-1}, u_h)$ auch in N_{f_i} enthalten ist, ist nichts zu zeigen. Andernfalls muss $f_{i+1}(e) \neq f_i(e)$ sein, d.h. e oder e^R müssen auf P_{i+1} liegen. Da e nicht in N_{f_i} ist, muss $e^R = (u_h, u_{h-1})$ auf P_{i+1} liegen. Da P_{i+1} ein kürzester Pfad von s nach t in N_{f_i} ist, folgt $d_i(s,u_{h-1}) = d_i(s,u_h) + 1$, was $d_i(s,u_h) = d_i(s,u_{h-1}) - 1 \le d_i(s,u_{h-1}) + 1$ impliziert. Vollkommen analog lässt sich $d_i(u,t) \le d_{i+1}(u,t)$ zeigen, womit wir folgende Behauptung bewiesen haben.

Behauptung 3.16. *Für jeden Knoten $u \in V$ gilt $d_i(s,u) \le d_{i+1}(s,u)$ und $d_i(u,t) \le d_{i+1}(u,t)$.*

Daraus ergibt sich nun folgende Behauptung.

Behauptung 3.17. *Falls $e = (u,v)$ in P_{i+1} und $e^R = (v,u)$ in P_{j+1} für $0 \le i < j < k$ enthalten ist, so gilt $\ell_{j+1} = d_j(s,t) \ge d_i(s,t) + 2 = \ell_{i+1} + 2$.*

Da P_{i+1} und P_{j+1} kürzeste s-t-Pfade in N_{f_i} bzw. N_{f_j} sind, folgt dies direkt aus obiger Behauptung:

$$d_j(s,t) = \underbrace{d_j(s,v)}_{\ge d_i(s,v)} + \underbrace{d_j(u,t)}_{\ge d_i(u,t)} + 1 \ge \underbrace{d_i(s,v)}_{d_i(s,u)+1} + \underbrace{d_i(u,t)}_{d_i(v,t)+1} + 1 = d_i(s,t) + 2.$$

Da jeder Zunahmepfad P_i mindestens eine kritische Kante enthält und $E \cup E^R$ höchstens m Kantenpaare der Form $\{e, e^R\}$ enthält, impliziert schließlich folgende Behauptung, dass $k \le mn/2$ ist.

Behauptung 3.18. *Zwei Kanten e und e^R sind zusammen höchstens $n/2$-mal kritisch.*

Seien $P_{i_1}, \ldots, P_{i_h}$ mit $1 \le i_1 < \cdots < i_h \le k$ die Pfade, für die eine der Kanten in $\{e, e^R\}$ kritisch ist. Falls $e' \in \{e, e^R\}$ kritisch für P_{i_j} mit $1 \le j < h$ ist, dann fehlt e' im Restnetzwerk $N_{f_{i_j}}$. Daher kann e' nur dann eine kritische Kante auf dem Pfad $P_{i_{j+1}}$ sein, wenn e'^R auf einem Pfad P_i mit $i_j < i \le i_{j+1}$ liegt. Dies gilt natürlich erst recht, wenn die Kante e'^R auf $P_{i_{j+1}}$ kritisch ist. Mit Behauptung 1 und Behauptung 2 folgt also $\ell_{i_{j+1}} \ge \ell_i \ge \ell_{i_j} + 2$. Daher ist

$$n - 1 \ge \ell_{i_h} \ge \ell_{i_1} + 2(h-1) \ge 1 + 2(h-1) = 2h - 1,$$

was $h \le n/2$ impliziert. ■

Man beachte, dass der Beweis auch bei Netzwerken mit reellen Kapazitäten seine Gültigkeit behält.

3.3 Der Algorithmus von Dinitz

Man kann zeigen, dass sich in jedem Netzwerk ein maximaler Fluss durch Addition von höchstens m Zunahmepfaden konstruieren lässt (siehe Übungen). Es ist nicht bekannt, ob sich solche Pfade in Zeit $O(m)$ bestimmen lassen. Wenn ja, würde dies auf eine Gesamtlaufzeit von $O(m^2)$ führen. Für dichte Netzwerke (d.h. $m = \Theta(n^2)$) hat der

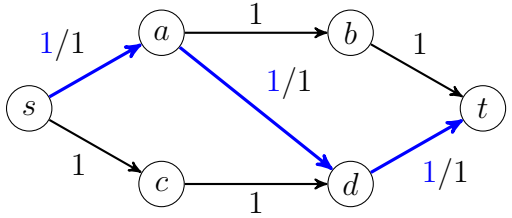
Algorithmus von Dinitz die gleiche Laufzeit $O(n^2m) = O(n^4)$ und die verbesserte Version ist mit $O(n^3)$ in diesem Fall sogar noch schneller.

Die Analyse der Laufzeit des Edmonds-Karp-Algorithmus beruht auf der Tatsache, dass der Fluss $f_{P_{i+1}}$ durch den Zunahmepfad P_{i+1}, der in jedem Schleifendurchlauf auf den aktuellen Fluss f_i addiert wird, eine Kante auf *mindestens einem* kürzesten Pfad im Restnetzwerk N_{f_i} sättigt. Dies hat zur Folge, dass nicht mehr als $nm/2$ Zunahmepfade P_i benötigt werden, um einen maximalen Fluss zu erhalten.

Dagegen addiert der Algorithmus von Dinitz in jedem Schleifendurchlauf auf den aktuellen Fluss f einen Fluss g, der auf *jedem* kürzesten Pfad im Restnetzwerk N_f mindestens eine Kante sättigt. Wir werden sehen, dass maximal $n-1$ solche Flüsse g_i benötigt werden.

Definition 3.19. *Ein Fluss g in einem Netzwerk $N = (V, E, s, t, c)$ **sättigt** eine Kante $e \in E$, falls $g(e) = c(e)$ ist. g heißt **blockierend**, falls g mindestens eine Kante auf jedem Pfad P von s nach t sättigt.*

Nach dem Max-Flow-Min-Cut-Theorem gibt es zu jedem maximalen Fluss f einen s-t-Schnitt S, so dass alle Kanten in $E(S)$ gesättigt sind. Da jeder Pfad von s nach t mindestens eine Kante in $E(S)$ enthalten muss, ist jeder maximale Fluss auch blockierend. Für die Umkehrung gibt es jedoch einfache Gegenbeispiele, wie etwa



Ein blockierender Fluss muss also nicht unbedingt maximal sein. Tatsächlich ist g genau dann ein blockierender Fluss in N, wenn es im Restnetzwerk N_g keinen Zunahmepfad gibt, der nur aus *Vorwärtskanten* $e \in E$ mit $g(e) < c(e)$ besteht.

Der Algorithmus von Dinitz berechnet anstelle eines kürzesten Zunahmepfades P im aktuellen Restnetzwerk N_f einen blockierenden Fluss g im Schichtnetzwerk N'_f. Dieses enthält nur diejenigen Kanten von N_f, die auf einem kürzesten Pfad mit Startknoten s liegen. Zudem werden aus N'_f alle Knoten $u \neq t$ entfernt, die einen Abstand $d(s,u) \geq d(s,t)$ in N_f haben. Der Name rührt daher, dass jeder Knoten in N'_f einer Schicht S_j zugeordnet wird.

Definition 3.20. *Sei $N = (V, E, s, t, c)$ ein Netzwerk und bezeichne $d(x, y)$ die Länge eines kürzesten Pfades von x nach y in N. Das zugeordnete **Schichtnetzwerk** ist $N' = (V', E', s, t, c')$ mit der Knotenmenge $V' = S_0 \cup \cdots \cup S_\ell$ und der Kantenmenge*

$$E' = \bigcup_{j=1}^{\ell} \{(u,v) \in E \mid u \in S_{j-1} \wedge v \in S_j\}$$

sowie der Kapazitätsfunktion

$$c'(e) = \begin{cases} c(e), & e \in E', \\ 0, & \text{sonst}, \end{cases}$$

wobei $\ell = 1 + \max\{d(s,u) < d(s,t) \mid u \in V\}$ und

$$S_j = \begin{cases} \{u \in V \mid d(s,u) = j\}, & 0 \leq j \leq \ell - 1, \\ \{t\}, & j = \ell \end{cases}$$

ist.

Der Algorithmus von Dinitz arbeitet wie folgt.

Algorithmus `Diniz`(N), $N = (V, E, s, t, c)$

```
1  for all (u,v) ∈ E ∪ E^R do
2    f(u,v) := 0
3  while S := schichtnetzwerk(N, f) ≠ ⊥ do
4    f := f + blockfluss(S)
```

Das zum Restnetzwerk $N_f = (V, E_f, s, t, c_f)$ gehörige Schichtnetzwerk $S = N'_f = (V', E'_f, s, t, c'_f)$ wird von der Prozedur `schichtnetzwerk`(N, f) in Zeit $O(n + m)$ berechnet. Für die Berechnung eines blockierenden Flusses g in einem Schichtnetzwerk S werden wir zwei Algorithmen angeben: Eine Prozedur `blockfluss1`, deren Laufzeit durch $O(nm)$ und eine Prozedur `blockfluss2`, deren Laufzeit durch $O(n^2)$ beschränkt ist.

Wir beschreiben zuerst die Prozedur `schichtnetzwerk`. Diese Prozedur führt in N_f eine modifizierte Breitensuche mit Startknoten s durch und speichert dabei in der Menge E' nicht nur alle Baumkanten, sondern zusätzlich alle Querkanten (u, v) (d.h. u und v liegen nicht auf einem gemeinsamen parent-Pfad), die auf einem kürzesten Weg von s zu v liegen. Die Suche bricht ab, sobald t am Kopf der Schlange erscheint oder alle von s aus erreichbaren Knoten abgearbeitet sind.

Falls t erreicht wurde, werden außer der Senke t alle Knoten u, die in N_f einen Abstand $d(s, u) < d(s, t)$ von der Quelle s haben, in der Menge V' zusammengefasst. Zudem werden alle Kanten aus E' wieder entfernt, die nicht zwischen zwei Knoten aus V' verlaufen.

Wurde dagegen t nicht erreicht, so existiert in N_f (und damit in N'_f) kein (blockierender) Fluss g mit $|g| > 0$ und somit auch kein Zunahmepfad in N_f, d.h. f ist maximal.

Prozedur `schichtnetzwerk`(N, f)

```
1  E' := ∅
2  for all v ∈ V do niv(v) := n
3  niv(s) := 0; Q := (s)
4  while Q ≠ () ∧ head(Q) ≠ t do
5    u := dequeue(Q)
6    for all e = (u, v) ∈ E ∪ E^R do
7      if c(e) − f(e) > 0 ∧ niv(v) > niv(u) then
8        E' := E' ∪ {e}
9        c'(e) := c(e) − f(e)
10       if niv(v) > niv(u) + 1 then
11         niv(v) := niv(u) + 1
12         enqueue(Q, v)
13 if head(Q) = t then
14   V' := {v ∈ V | niv(v) < niv(t)} ∪ {t}
15   E' := E' ∩ (V' × V')
16   return (V', E', s, t, c')
17 else return ⊥
```

Die Laufzeitschranke $O(n + m)$ für die Prozedur `schichtnetzwerk` folgt aus der Tatsache, dass jede Kante in $E \cup E^R$ höchstens einmal besucht wird und jeder Besuch mit einem konstanten Zeitaufwand verbunden ist.

Nun kommen wir zur Beschreibung der Prozedur `blockfluss1`, die einen blockierenden Fluss g in einem gegebenen Schichtnetzwerk $S = (V, E, s, t, c)$ berechnet. Beginnend mit dem Nullfluss g bestimmt diese in der repeat-Schleife mittels Tiefensuche

- einen s-t-Pfad P in S,
- addiert den Fluss f_P durch P in S zum aktuellen Fluss g hinzu,
- aktualisiert die Kapazitäten aller Kanten auf dem Pfad P und
- entfernt aus S die von g gesättigten Kanten.

Der Pfad P lässt sich hierbei direkt aus dem Inhalt des Kellers K rekonstruieren, weshalb er ***K-Pfad*** genannt wird. Man beachte, dass die Kapazitäten der auf dem gefundenen Pfad P liegenden Kanten nur in Vorwärtsrichtung, aber anders als bei Ford-Fulkerson und Edmonds-Karp nicht auch in Rückwärtsrichtung angepasst werden.

Falls die Tiefensuche in einem Knoten $u \neq s$ in einer Sackgasse endet (weil E keine von u aus weiterführenden Kanten enthält), wird die zuletzt besuchte Kante (u', u) ebenfalls aus E entfernt und die Tiefensuche vom Startpunkt u' dieser Kante fortgesetzt (backtracking). Die Prozedur `blockfluss1` bricht ab, sobald alle Kanten mit Startknoten s aus E entfernt wurden und somit in (V, E) keine Pfade mehr von s nach t existieren (d.h. g ist ein blockierender Fluss in S).

Prozedur `blockfluss1`(S), $S = (V, E, s, t, c)$

```
1  for all e ∈ E ∪ E^R do g(e) := 0
2  u := s
3  K := (s)
4  done := false
5  repeat
6    if ∃e = (u, v) ∈ E then
7      push(K, v)
8      u := v
9    elsif u = t then
10     P := K-Pfad von s nach t
11     c(P) := min{c(e) | e ∈ P}
12     for all e ∈ P do
13       g(e) := g(e) + c(P)
14       g(e^R) := −g(e)
15       c(e) := c(e) − c(P)
16       if c(e) = 0 then E := E \ {e}
17     K := (s)
18     u := s
19   elsif u ≠ s then \\ backtracking
20     pop(K)
21     u' := top(K)
22     E := E \ {(u', u)}
23     u := u'
24   else done := true
25 until done
26 return g
```

Die Laufzeitschranke $O(nm)$ folgt aus der Tatsache, dass sich die Anzahl der aus E entfernten Kanten nach spätestens n Schleifendurchläufen um 1 erhöht.

Satz 3.21. *Der Algorithmus von Dinitz durchläuft die while-Schleife höchstens $(n-1)$-mal.*

Beweis. Sei f_0 der Nullfluss in N und seien $g_1, \ldots, g_k$ die blockierenden Flüsse, die der Dinitz-Algorithmus der Reihe nach berechnet, d.h. $f_{i+1} = f_i + g_{i+1}$. Zudem sei $d_i(u, v)$ die minimale Länge eines Pfades von u nach v im Restnetzwerk N_{f_i} und sei $\delta_i = d_i(s, t)$. Wir zeigen, dass $\delta_i < \delta_{i+1}$ für $i = 1, \ldots, k-1$ gilt. Da $\delta_1 \geq 1$ und $\delta_k \leq n-1$ ist, folgt $k \leq n-1$.

Hierzu beweisen wir zunächst, dass für jeden kürzesten Pfad $P = (u_0, \ldots, u_l)$ von $u_0 = s$ nach $u_l = u$ in $N_{f_{i+1}}$ (d.h. $d_{i+1}(s, u_h) = h$) für $h = 1, \ldots, l$ folgende (Un)gleichungen gelten:

$$d_i(s, u_h) \leq d_i(s, u_{h-1}) + 1, \text{ falls } (u_{h-1}, u_h) \in E_{f_i} \tag{3.1}$$

$$d_i(s, u_h) = d_i(s, u_{h-1}) - 1, \text{ falls } (u_{h-1}, u_h) \notin E_{f_i} \tag{3.2}$$

Es ist klar, dass (3.1) gilt, falls die Kante $e = (u_{h-1}, u_h)$ auch in N_{f_i} enthalten ist. Andernfalls ist $f_{i+1}(e) \neq f_i(e)$, d.h. $g_{i+1}(e) \neq 0$. Da e nicht in N_{f_i} und somit auch nicht in N'_{f_i} enthalten ist, muss $e^R = (u_h, u_{h-1})$ in N'_{f_i} sein. Da N'_{f_i} nur Kanten auf kürzesten Pfaden mit Startknoten s enthält, folgt $d_i(s, u_{h-1}) = d_i(s, u_h) + 1$, was (3.2) impliziert. Aus (3.1 + 3.2) folgt

$$d_i(s, u_l) \leq d_i(s, u_{l-1}) + 1 \leq \cdots \leq d_i(s, s) + l = l = d_{i+1}(s, u_l)$$

und wir haben für jeden Knoten $u \in V$ folgende Ungleichung bewiesen:

$$d_i(s, u) \leq d_{i+1}(s, u). \tag{3.3}$$

Um nun zu zeigen, dass $\delta_i < \delta_{i+1}$ für $i = 1, \ldots, k-1$ gilt, sei $P = (u_0, u_1, \ldots, u_{\delta_{i+1}})$ ein kürzester Pfad von $s = u_0$ nach $t = u_{\delta_{i+1}}$ in $N_{f_{i+1}}$ (und somit auch in $N'_{f_{i+1}}$). Mit Ungleichung 3.3 folgt, dass $d_i(s, u_h) \leq d_{i+1}(s, u_h) = h$ für $h = 0, \ldots, \delta_{i+1}$ ist. Wir unterscheiden zwei Fälle.

- Wenn alle Knoten u_h in N'_{f_i} enthalten sind, muss ein h mit $d_i(s, u_h) \leq d_i(s, u_{h-1})$ existieren. Würde nämlich $d_i(s, u_h) > d_i(s, u_{h-1})$ für $h = 1, \ldots, \delta_{i+1} - 1$ gelten, so wären die Kanten (u_{h-1}, u_h) für $h = 1, \ldots, \delta_{i+1} - 1$ wegen (3.2) in N_{f_i} enthalten und somit würde wegen (3.1) $d_i(s, u_h) = d_i(s, u_{h-1}) + 1$ für

$h = 1, \ldots, \delta_{i+1} - 1$ folgen. Dies hätte wiederum zur Folge, dass P ein kürzester Pfad von s nach t in N_{f_i} und somit ein s-t-Pfad in N'_{f_i} wäre, der von g_i nicht blockiert wird, da er auch in $N_{f_{i+1}}$ existiert. Da aber g_i blockierend ist, muss also ein h mit $d_i(s, u_h) \leq d_i(s, u_{h-1})$ existieren und es folgt unter Verwendung von (3.1 + 3.2):

$$\delta_i = d_i(s,t) \leq d_i(s,u_h) + \underbrace{d_i(u_h,t)}_{\leq \delta_{i+1}-h} \leq \underbrace{d_i(s,u_{h-1})}_{\leq d_{i+1}(s,u_{h-1})=h-1} + \delta_{i+1} - h < \delta_{i+1}$$

- Falls mindestens ein Knoten u_h nicht in N'_{f_i} enthalten ist, sei u_h der erste solche Knoten auf P. Da $u_h \neq t$ ist, folgt $d_{i+1}(s, u_h) < d_{i+1}(s,t) = \delta_{i+1}$. Zudem liegt die Kante $e = (u_{h-1}, u_h)$ nicht nur in $N_{f_{i+1}}$, sondern wegen $f_{i+1}(e) = f_i(e)$ (da weder e noch e^R zu N'_{f_i} gehören) auch in N_{f_i}. Da somit u_{h-1} in N'_{f_i} und e in N_{f_i} ist, kann u_h nur aus dem Grund nicht zu N'_{f_i} gehören, dass $d_i(s,u_h) = d_i(s,t)$ ist. Daher folgt unter Verwendung von (3.1 + 3.2 + 3.3) auch in diesem Fall die Ungleichung $\delta_i < \delta_{i+1}$:

$$\delta_i = d_i(s,t) = d_i(s,u_h) \leq \underbrace{d_i(s,u_{h-1})}_{\leq d_{i+1}(s,u_{h-1})} + 1 \leq d_{i+1}(s,u_h) < \delta_{i+1}$$

■

Korollar 3.22. *Der Algorithmus von Dinitz berechnet bei Verwendung der Prozedur* `blockfluss1` *einen maximalen Fluss in Zeit $O(n^2 m)$.*

Die Prozedur `blockfluss2` benötigt nur Zeit $O(n^2)$, um einen blockierenden Fluss g im Schichtnetzwerk N'_f zu berechnen, was auf eine Gesamtlaufzeit des Algorithmus von Dinitz von $O(n^3)$ führt. Zu ihrer Beschreibung benötigen wir folgende Notation.

Definition 3.23. *Sei $N = (V, E, s, t, c)$ ein Netzwerk.*

*a) Der **Durchsatz eines Knotens u** in V ist*

$$D(u) = \begin{cases} c^+(u), & u = s, \\ c^-(u), & u = t, \\ \min\{c^+(u), c^-(u)\}, & \text{sonst}, \end{cases}$$

*wobei $c^+(u) = \sum_{v \in V} c(u,v)$ die **Ausgangskapazität** und $c^-(u) = \sum_{v \in V} c(v,u)$ die **Eingangskapazität von u** ist.*

*b) Ein Fluss g in N **sättigt einen Knoten u** in V, falls*

- *$u = s$ ist und g alle Kanten $(s,v) \in E$ mit Startknoten s sättigt, oder*
- *$u = t$ ist und g alle Kanten $(v,t) \in E$ mit Zielknoten t sättigt, oder*
- *$u \in V - \{s,t\}$ ist und g alle Kanten $(u,v) \in E$ mit Startknoten u oder alle Kanten $(v,u) \in E$ mit Zielknoten u sättigt.*

Die Korrektheit der Prozedur `blockfluss2` basiert auf folgender Proposition.

Proposition 3.24. *Wenn ein Fluss g in einem Netzwerk N auf jedem s-t-Pfad P mindestens einen Knoten u sättigt, dann ist g blockierend.*

Beweis. Falls g mindestens einen Knoten u auf dem s-t-Pfad P sättigt, dann sättigt g auch mindestens eine Kante auf dem Pfad P. ■

Beginnend mit dem trivialen Fluss $g = 0$ berechnet die Prozedur `blockfluss2` für jeden Knoten u den Durchsatz $D(u)$ im Schichtnetzwerk $S = (V, E, s, t, c)$ und wählt in jedem Durchlauf der repeat-Schleife einen Knoten u mit minimalem Durchsatz $D(u)$. Dann benutzt sie die Prozeduren `propagierevor` und `propagiererück`, um den aktuellen Fluss g um den Wert $D(u)$ zu erhöhen und die Restkapazitäten der betroffenen Kanten sowie die Durchsatzwerte $D(v)$ der betroffenen Knoten entsprechend zu aktualisieren.

Anschließend werden alle gesättigten Knoten aus V und alle gesättigten Kanten aus E entfernt. Hierzu werden in der Menge B alle Knoten gespeichert, deren Durchsatz durch die Erhöhungen des Flusses g auf 0 gesunken ist.

Prozedur `blockfluss2`(S), $S = (V, E, s, t, c)$

```
1  for all e ∈ E ∪ E^R do g(e) := 0
2  for all u ∈ V do
3    c⁺(u) := Σ_{(u,v)∈E} c(u, v)
4    c⁻(u) := Σ_{(v,u)∈E} c(v, u)
5  repeat
6    for all u ∈ V \ {s, t} do D(u) := min{c⁻(u), c⁺(u)}
7    D(s) := c⁺(s)
8    D(t) := c⁻(t)
9    wähle u ∈ V mit D(u) minimal
10   B := {u}
11   propagierevor(u)
12   propagiererück(u)
13   while ∃v ∈ B \ {s, t} do
14     B := B \ {v}; V := V \ {v}
15     for all e = (v, w) ∈ E do
16       c⁻(w) := c⁻(w) − c(v, w)
17       if c⁻(w) = 0 then B := B ∪ {w}
18       E := E \ {e}
19     for all e = (w, v) ∈ E do
20       c⁺(w) := c⁺(w) − c(w, v)
21       if c⁺(w) = 0 then B := B ∪ {w}
22       E := E \ {e}
23 until u ∈ {s, t}
24 return g
```

Da in jedem Durchlauf der repeat-Schleife mindestens ein Knoten u gesättigt und aus V entfernt wird, wird nach höchstens $n-1$ Iterationen einer der beiden Knoten s oder t als Knoten u mit minimalem Durchsatz $D(u)$ gewählt und die repeat-Schleife verlassen. Da nach Beendigung des letzten Durchlaufs der Durchsatz von s oder von t gleich 0 ist, wird einer dieser beiden Knoten zu diesem Zeitpunkt von g gesättigt. Nach Proposition 3.24 ist somit g ein blockierender Fluss.

Die Prozeduren `propagierevor` und `propagiererück` propagieren den Fluss durch u in Vorwärtsrichtung hin zu t bzw. in Rückwärtsrichtung hin zu s. Dies geschieht in Form einer Breitensuche mit Startknoten u unter Benutzung der Kanten in E bzw. E^R. Da der Durchsatz $D(u)$ von u unter allen Knoten minimal ist, ist sichergestellt, dass der Durchsatz $D(v)$ jedes Knotens v ausreicht, um den für ihn ermittelten Zusatzfluss in Höhe von $z(v)$ weiterzuleiten.

Prozedur `propagierevor`(u)

```
1  for all v ∈ V do z(v) := 0
2  z(u) := D(u)
3  Q := (u); R := {u}
4  while Q ≠ () do
5    v := dequeue(Q)
6    while z(v) ≠ 0 ∧ ∃e = (v, w) ∈ E do
7      if w ∉ R then enqueue(Q, w)
8      R := R ∪ {w}
9      m := min{z(v), c(e)}; z(v) := z(v) − m; z(w) := z(w) + m
10     aktualisierekante(e, m)
```

Prozedur `aktualisierekante`(e, m), $e = (v, w)$

```
1  g(e) := g(e) + m
2  c(e) := c(e) − m
3  if c(e) = 0 then E := E \ {e}
4  c⁺(v) := c⁺(v) − m
5  if c⁺(v) = 0 then B := B ∪ {v}
6  c⁻(w) := c⁻(w) − m
7  if c⁻(w) = 0 then B := B ∪ {w}
```

Die Prozedur `propagiererück` unterscheidet sich von der Prozedur `propagierevor` nur dadurch, dass in Zeile 6 die Bedingung $\exists e = (v, w) \in E$ durch die Bedingung $\exists e = (w, v) \in E$ ersetzt wird.

Da die repeat-Schleife von `blockfluss2` maximal $(n-1)$-mal durch-

laufen wird, werden die Prozeduren `propagierevor` und `propagiererück` höchstens $(n-1)$-mal aufgerufen. Sei a die Gesamtzahl der Durchläufe der inneren while-Schleife von `propagierevor`, summiert über alle Aufrufe. Da in jedem Durchlauf eine Kante aus E entfernt wird (falls $m = c(v,u)$ ist) oder der zu propagierende Fluss $z(v)$ durch einen Knoten v auf 0 sinkt (falls $m = z(v)$ ist), was pro Knoten und pro Aufruf höchstens einmal vorkommt, ist $a \leq n^2 + m$. Der gesamte Zeitaufwand ist daher $O(n^2 + m)$ innerhalb der beiden while-Schleifen und $O(n^2)$ außerhalb. Die gleichen Schranken gelten für `propagiererück`.

Eine ähnliche Überlegung zeigt, dass die while-Schleife von `blockfluss2` einen Gesamtaufwand von $O(n+m)$ hat. Folglich ist die Laufzeit von `blockfluss2` $O(n^2)$.

Korollar 3.25. *Der Algorithmus von Dinitz berechnet bei Verwendung der Prozedur* `blockfluss2` *einen maximalen Fluss in Zeit* $O(n^3)$.

Auf Netzwerken, deren Flüsse durch jede Kante oder durch jeden Knoten durch eine relativ kleine Zahl C beschränkt sind, lassen sich noch bessere Laufzeitschranken für den Dinitz-Algorithmus nachweisen. Hierzu benötigen wir folgende Beziehungen zwischen einem Netzwerk und den zugehörigen Restnetzwerken.

Lemma 3.26. *Sei $N = (V, E, s, t, c)$ ein Netzwerk, f ein Fluss in N und N_f das zugehörige Restnetzwerk. Zudem sei $h : V \times V \to \mathbb{Z}$.*

(i) Die Funktion h ist genau dann ein Fluss (bzw. maximaler Fluss) in N_f, wenn $f + h$ ein Fluss (bzw. maximaler Fluss) in N ist.

(ii) Für jede Kante $e \in E \cup E^R$ gilt $c_f(e) + c_f(e^R) = c(e) + c(e^R)$.

(iii) Für jeden Knoten $u \in V \setminus \{s,t\}$ gilt $c_f^+(u) = c^+(u)$ und $c_f^-(u) = c^-(u)$ und somit $D_f(u) = D(u)$.

Beweis.

(i) Da f die Antisymmetrie und die Kontinuität erfüllt, übertragen sich diese Eigenschaften von h auf $f+h$ und umgekehrt. Weiter gilt

$$h(u,v) \leq \underbrace{c_f(u,v)}_{c(u,v)-f(u,v)} \Leftrightarrow f(u,v) + h(u,v) \leq c(u,v),$$

d.h. h erfüllt genau dann die Kapazitätsbedingung in N_f, wenn $f+h$ sie in N erfüllt. Zudem ist $f+h$ genau dann ein maximaler Fluss in N, wenn h ein maximaler Fluss in N_f ist, da jeder Fluss h' in N_f mit $|h'| > |h|$ einen Fluss $f + h'$ der Größe $|f+h'| > |f+h|$ in N und jeder Fluss g in N mit $|g| > |f+h|$ einen Fluss $g-f$ der Größe $|g-f| = |g|-|f| > |f+h|-|f| = |h|$ in N_f liefern würde.

(ii) Es gilt $\underbrace{c_f(e)}_{c(e)-f(e)} + \underbrace{c_f(e^R)}_{c(e^R)-f(e^R)} = c(e) + c(e^R) - (\underbrace{f(e) + f(e^R)}_{=0})$.

(iii) Für jeden Knoten $u \in V \setminus \{s,t\}$ gilt

$$c_f^+(u) = \sum_{v \in V} \underbrace{c_f(u,v)}_{c(u,v)-f(u,v)} = \underbrace{\sum_{v \in V} c(u,v)}_{c^+(u)} - \underbrace{\sum_{v \in V} f(u,v)}_{=0}.$$

Die Gleichheit $c_f^-(u) = c^-(u)$ folgt analog. ∎

Lemma 3.27. *Sei $F > 0$ die maximale Flussgröße in einem Netzwerk N und sei ℓ die Länge des zu N gehörigen Schichtnetzwerks N'.*

(i) Falls jeder Knoten $u \in V \setminus \{s,t\}$ einen Durchsatz $D(u) \leq C$ in N hat, gilt $\ell \leq 1 + (n-2)C/F$.

(ii) Falls jede Kante $e \in E$ eine Kapazität $c(e) \leq C$ hat, gilt $\ell \leq \min\{mC/F, 2n\sqrt{C/F}\}$.

Beweis. Sei f ein Fluss der Größe F in N.

(i) Da f für $j = 1, \ldots, \ell - 1$ durch die n_j Knoten der Schicht S_j von N' fließt, von denen jeder einen Durchsatz $\leq C$ hat, muss

$$F \leq n_j C \text{ bzw. } F/C \leq n_j$$

sein, woraus $n-2 \geq \sum_{j=1}^{\ell-1} n_j \geq (\ell-1)F/C$ bzw. $\ell \leq 1+(n-2)C/F$ folgt.

(ii) Für $j=1,\dots,\ell-1$ sei E_j die Menge der Kanten von Schicht S_{j-1} nach Schicht S_j und sei E_ℓ die Menge der Kanten von $S_{\ell-1}$ nach $S_\ell := V - \bigcup_{j=0}^{\ell-1} S_j$ in N. Da der Fluss f für $j=1,\dots,\ell$ durch die m_j Kanten in E_j fließt, die alle eine Kapazität $\leq C$ haben, muss

$$F \leq m_j C \leq C|S_{j-1}||S_j| \text{ bzw. } F/C \leq m_j \leq |S_{j-1}||S_j|$$

sein, woraus sofort $m \geq \sum_{j=1}^{\ell} m_j \geq \ell F/C$ bzw. $\ell \leq mC/F$ folgt. Wegen $F/C \leq |S_{j-1}||S_j|$ muss zudem S_{j-1} oder S_j mindestens $\sqrt{F/C}$ Knoten enthalten und es folgt

$$(\ell/2)\sqrt{F/C} \leq |S_0| + \dots + |S_\ell| = n \text{ bzw. } \ell \leq 2n\sqrt{C/F} \quad \blacksquare$$

Satz 3.28. *Sei k die Anzahl der Schleifendurchläufe des Algorithmus von Dinitz bei Eingabe eines Netzwerks $N=(V,E,s,t,c)$.*

(i) Falls jeder Knoten $u \in V \setminus \{s,t\}$ einen Durchsatz $D(u) \leq C$ hat, so gilt $k \leq 1+2\sqrt{nC}$.

(ii) Falls jede Kante $e \in E$ eine Kapazität $c(e) \leq C$ hat, so gilt $k \leq \min\{\sqrt{8mC}, 4(n^2C)^{1/3}\}$.

Beweis. Sei $F=|f|$ die Größe eines maximalen Flusses f in N und seien $g_1,\dots,g_k$ die blockierenden Flüsse, die der Dinitz-Algorithmus der Reihe nach im Schichtnetzwerk N'_{f_i} berechnet, d.h. f_1 ist der Nullfluss in N und $f_{i+1}=f_i+g_i$.

(i) Da die Anzahl k der Schleifendurchläufe durch F beschränkt ist, können wir $F > \sqrt{nC}$ annehmen. Betrachte den i-ten Schleifendurchlauf. Da $f-f_i$ nach Lemma 3.26(i) ein maximaler Fluss in N_{f_i} der Größe $R_i = F-|f_i|$ ist und nach Lemma 3.26(iii) jeder Knoten $u \in V \setminus \{s,t\}$ in N_{f_i} den gleichen Durchsatz wie in N hat, folgt nach Lemma 3.27(i), dass N'_{f_i} eine Länge $\ell_i \leq 1+nC/R_i$ hat. Wegen $\ell_i \geq i+1$ folgt daher

$$k \leq i+1+R_{i+1} \leq \ell_i + R_{i+1} \leq R_{i+1}+1+nC/R_i.$$

Nun wählen wir i so, dass $R_i > \sqrt{nC}$ und $R_{i+1} \leq \sqrt{nC}$ ist. Dann folgt

$$k-1 \leq R_{i+1} + nC/R_i < \sqrt{nC} + nC/\sqrt{nC} = 2\sqrt{nC}.$$

(ii) Zum Nachweis von $k \leq \sqrt{8mC}$ können wir $F > \sqrt{2mC}$ annehmen. Wir betrachten wieder den i-ten Schleifendurchlauf. Da jede Kante $e \in E_{f_i}$ nach Lemma 3.26(ii) eine Kapazität $c_{f_i}(e) \leq 2C$ hat und $f-f_i$ nach Lemma 3.26(i) ein maximaler Fluss in N_{f_i} ist und die Größe $R_i = F-|f_i|$ hat, folgt nach Lemma 3.27(ii), dass N'_{f_i} eine Länge $\ell_i \leq 2mC/R_i$ hat. Wegen $\ell_i \geq i+1$ folgt daher

$$k \leq i+1+R_{i+1} \leq \ell_i + R_{i+1} \leq R_{i+1} + 2mC/R_i.$$

Wählen wir nun i so, dass $R_i > \sqrt{2mC}$ und $R_{i+1} \leq \sqrt{2mC}$ ist, so erhalten wir

$$k \leq \sqrt{2mC} + \sqrt{2mC} = \sqrt{8mC}.$$

Zum Nachweis von $k \leq 4(n^2C)^{1/3}$ können wir $F > (2n\sqrt{2C})^{2/3} = 2(n^2C)^{1/3}$ annehmen. Zudem folgt nach Lemma 3.27(ii), dass N'_{f_i} eine Länge $\ell_i \leq 2n\sqrt{2C/R_i}$ hat. Damit ist k für jedes $i=0,\dots,k-1$ durch

$$k \leq i+1+R_{i+1} \leq \ell_i + R_{i+1} \leq R_{i+1} + 2n\sqrt{2C/R_i}$$

beschränkt. Wählen wir nun i so, dass $R_i > (2n\sqrt{2C})^{2/3}$ und $R_{i+1} \leq (2n\sqrt{2C})^{2/3}$ ist, so erhalten wir

$$k \leq (2n\sqrt{2C})^{2/3} + 2n\sqrt{2C}/(2n\sqrt{2C})^{1/3} = 4(n^2C)^{1/3} \quad \blacksquare$$

Korollar 3.29. *Sei T die Laufzeit des Algorithmus von Dinitz unter Verwendung von* `blockfluss1` *bei Eingabe von $N = (V, E, s, t, c)$.*

(i) Falls jeder Knoten $u \in V \setminus \{s, t\}$ einen Durchsatz $D(u) \leq C$ hat, so gilt $T = O((nC + m)\sqrt{nC})$.

(ii) Falls jede Kante $e \in E$ eine Kapazität $c(e) \leq C$ hat, so gilt $T = O(\min\{(mC)^{3/2}, C^{4/3}n^{2/3}m\})$.

Beweis. Nach obigem Satz ist die Anzahl k der Schleifendurchläufe des Algorithmus von Dinitz im Fall *(i)* durch $k \leq 1 + 2\sqrt{nC}$ und im Fall *(ii)* durch $k \leq \min\{\sqrt{8mC}, 4(n^2C)^{1/3}\}$ beschränkt. Zudem folgt mit Lemma 3.26 dass jede Kante e und jeder Knoten u (außer s und t) in jedem Restnetzwerk N_{f_i} (und somit auch in jedem Schichtnetzwerk N'_{f_i}) eine Kapazität $c(e) \leq 2C$ bzw. einen Durchsatz $D(u) \leq C$ haben.

(i) Jedesmal wenn `blockfluss1` einen s-t-Pfad P im aktuellen Schichtnetzwerk findet, verringert sich der Durchsatz $c(u)$ der auf P liegenden Knoten u um den Wert $c(P) \geq 1$, da der Fluss g durch diese Knoten um diesen Wert steigt. Daher kann jeder Knoten an maximal C Flusserhöhungen beteiligt sein, bevor sein Durchsatz auf 0 sinkt. Da somit pro Knoten ein Zeitaufwand von $O(C)$ für alle erfolgreichen Tiefensuchschritte, die zu einem s-t-Pfad führen, und zusätzlich pro Kante ein Zeitaufwand von $O(1)$ für alle nicht erfolgreichen Tiefensuchschritte anfällt, läuft `blockfluss1` in Zeit $O(nC + m)$.

(ii) Jedesmal wenn `blockfluss1` einen s-t-Pfad P im Schichtnetzwerk findet, verringert sich die Kapazität $c(e)$ der auf P liegenden Kanten e um den Wert $c(P) \geq 1$. Da somit pro Kante ein Zeitaufwand von $O(C)$ für alle erfolgreichen Tiefensuchschritte und $O(1)$ für alle nicht erfolgreichen Tiefensuchschritte anfällt, läuft `blockfluss1` in Zeit $O(mC + m) = O(mC)$. ■

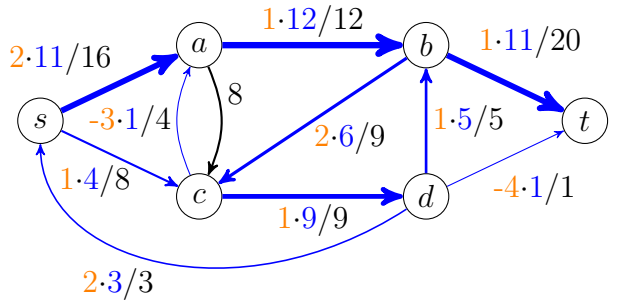
3.4 Kostenoptimale Flüsse

In bestimmten Anwendungen fallen für die Benutzung jeder Kante Kosten an, deren Höhe proportional zum Fluss durch die Kante ist. Falls zwei Flüsse f und g die einzelnen Kanten eines Netzwerks unterschiedlich beanspruchen, ist es möglich, dass f und g unterschiedliche Kosten verursachen, obwohl sie die gleiche Größe $|f| = |g|$ haben. Gesucht ist dann ein maximaler Fluss f mit minimalen Kosten, wobei die Kosten von f in N wie folgt auf der Basis einer ***Kostenfunktion k*** bestimmt werden:

- Jede Kante $e \in E$ mit $f(e) \geq 0$ verursacht Kosten in Höhe von $f(e)k(e)$.
- Die ***Gesamtkosten von f*** in $N = (V, E, s, t, c, k)$ betragen daher

$$k(f) = \sum_{f(e)>0} f(e)k(e).$$

Beispiel 3.30. *Die Abbildung zeigt einen Fluss f in N, wobei alle Kanten $e \in E$ mit $f(e) > 0$ mit $k(e)\cdot f(e)/c(e)$ beschriftet sind:*



Seine Größe ist $|f| = f(\{s\}) = 11 + 4 - 3 = 12$ und seine Kosten sind

$$k(f) = \sum_{f(e)>0} k(e)f(e) = (12+11+5+4+9)+2(11+6+3)\,\text{-}\,3\,\text{-}\,4 = 90.$$

Ist $k(e) < 0$ so bedeuten Kosten in Höhe von $k(e)f(e)$ einen Gewinn in Höhe von $-k(e)f(e)$ und umgekehrt. Erhöhen wir den Fluss $f(e)$ durch eine Kante e um a, so fallen dafür Kosten in Höhe von $k(e)\cdot a$ an. Entsprechend verursacht eine Erniedrigung von $f(e)$ um a einen Gewinn von $k(e)\cdot a$ (bzw. Kosten in Höhe von $-k(e)\cdot a$). Da eine Erhöhung von $f(e)$ um a den Fluss $f(e^R) = -f(e)$ durch e^R um a erniedrigt, muss also $k(e^R) = -k(e)$ sein. Möchten wir für eine Kante $e = (u, v)$ einen anderen Kostenfaktor b als $-k(e^R)$ haben, so können wir einen neuen Knoten w zu V hinzufügen und die Kante e durch den Pfad $P = (u, w, v)$ ersetzen. Nun kann $k(P)$ unabhängig vom Wert $k(e^R)$ mittels $k(u, w) = b$ und $k(w, v) = 0$ auf den gewünschten Wert b gesetzt werden.

Wir betrachten in diesem Abschnitt also Netzwerke der Form $N = (V, E, s, t, c, k)$, wobei k eine antisymmetrische Funktion $k : V \times V \to \mathbb{Z}$ mit $k(e) = -k(e^R)$ für alle $e \in V \times V$ ist. Zudem definieren wir für beliebige Multimengen F von Kanten $e \in V \times V$ den ***Kostenfaktor von F*** als $\boldsymbol{k(F)} = \sum_{e\in F} v_F(e)k(e)$. Jede Kante $e \in F$ wird bei der Berechnung von $k(F)$ also entsprechend der Häufigkeit $v_F(e)$ ihres Vorkommens in F berücksichtigt. Wir benutzen diese Notation auch für Pfade P (oder Kreise K) und schreiben $k(P)$, wobei wir P als Menge der Kanten auf P auffassen. Wir nennen F ***negativ***, falls $k(F) < 0$ ist. Für einen Fluss f sei

$$\boldsymbol{k_{\min}(f)} = \min\{k(g) \mid g \text{ ist ein Fluss in } N \text{ mit } |g| = |f|\}$$

das Minimum der Kosten $k(g)$ aller Flüsse g in N der Größe $|g| = |f|$.

Das nächste Lemma liefert einen Algorithmus, mit dem sich überprüfen lässt, ob ein Fluss minimale Kosten unter allen Flüssen seiner Größe hat.

Lemma 3.31. *Ein Fluss f in N hat genau dann minimale Kosten $k(f) = k_{\min}(f)$, wenn es im Restnetzwerk N_f keinen negativen Kreis K mit $k(K) < 0$ gibt.*

Beweis. Falls es in N_f einen negativen Kreis K gibt, können wir den Fluss durch alle Kanten $e \in K$ um eins erhöhen, um einen Fluss g der Größe $|g| = |f|$ mit $k(g) = k(f) + k(K) < k(f)$ zu erhalten.

Sei nun umgekehrt g ein Fluss in N mit $|g| = |f|$ und $k(g) < k(f)$. Wegen $g(e) - f(e) \le c(e) - f(e)$ ist dann $h = g - f$ ein Fluss im Restnetzwerk $N_f = (V, E_f, s, t, c_f, k)$. Da h die Größe $|h| = |g| - |f| = 0$ hat, können wir h als Summe von Flüssen $f_{K_1}, \dots, f_{K_\ell}$ in N_f darstellen, wobei jeder Fluss f_{K_i} nur für Kanten e auf einem Kreis K_i in (V, E_f) einen positiven Wert $f_{K_i}(e) = w_i > 0$ annimmt (siehe nächsten Abschnitt). Wegen $k(f_{K_1}) + \dots + k(f_{K_\ell}) = k(g - f) = k(g) - k(f) < 0$ und $k(f_{K_i}) = \sum_{e\in K_i} f_{K_i}(e)k(e) = w_i k(K_i)$ muss mindestens ein Kreis K_i negativ sein.

Um für $i = 0, \dots, \ell - 1$ den Fluss $f_{K_{i+1}}$ und den zugehörigen Kreis K_{i+1} zu finden, wählen wir eine beliebige Kante $e_{i,1}$ aus E_f, für die der Fluss $r_i = h - f_{K_1} - \dots - f_{K_i}$ einen minimalen positiven Wert $w_i = r_i(e_{i,1}) > 0$ annimmt. Falls es keine Kante $e \in E_f$ mit $r_i(e) > 0$ gibt, sind wir fertig, weil dann r_i der Nullfluss mit $r_i(e) = 0$ für alle $e \in V \times V$ und somit $\ell = i$ ist.

Andernfalls benutzen wir die Tatsache, dass r_i wie h und die bereits gefundenen Flüsse $f_{K_1}, \dots, f_{K_i}$ den Wert 0 hat, um einen negativen Kreis K_{i+1} zu finden. Wegen $|r_i| = 0$ erfüllt r_i nämlich die Kontinuitätsbedingung auch für die Knoten s und t. Daher können wir K_{i+1} wie folgt konstruieren.

- Beginnend mit $j = 1$ wählen wir zu jeder Kante $e_{i,j} = (u, v)$ solange eine Fortsetzung $e_{i,j+1} = (v, w) \in E_f$ mit $r_i(e_{i,j+1}) > 0$, bis sich ein Kreis K_{i+1} schließt.

Nun setzen wir $f_{K_{i+1}}(e_{i,j}) = w_i$ für alle Kanten $e_{i,j}$ auf dem Kreis K_{i+1} und $f_{K_{i+1}}(e_{i,j}) = 0$ für alle Kanten außerhalb von K_{i+1}. Da die Anzahl der Kanten in E_f, die unter dem Fluss r_{i+1} den Wert 0 haben, gegenüber r_i mindestens um eins zunimmt, ist die Anzahl ℓ der gefundenen Kreise durch $\ell \le |E_f| \le 2m$ beschränkt. ∎

Mithilfe von Lemma 3.31 lässt sich nun ein maximaler Fluss mit minimalen Kosten wie folgt berechnen. Wir berechnen zuerst einen maximalen Fluss f und setzen $f_0 = f$. Dann berechnen wir für $i = 0, 1, \ldots$ einen negativen Kreis K_{i+1} in N_{f_i}. Hierzu fügen wir dem Digraphen (V, E_{f_i}, k) einen neuen Knoten s' hinzu und verbinden s' mit allen Knoten $u \in V$ durch eine neue Kante (s', u) mit $k(s', u) = 0$. Dann suchen wir mit dem Bellman-Ford-Moore (BFM) Algorithmus in dem resultierenden Digraphen $G_i = (V \cup \{s'\}, E_{f_i} \cup \{(s', u) \mid u \in V\}, k)$ nach einem negativen Kreis K_{i+1}.

Falls es in G' keinen negativen Kreis gibt, ist f_i ein maximaler Fluss in N mit minimalen Kosten. Andernfalls benutzen wir den Fluss $f_{K_{i+1}}$, der auf jeder Kante e auf K_{i+1} den Wert $f_{K_{i+1}}(e) = c_{f_i}(K_{i+1}) = \min\{c_{f_i}(e) \mid e \in K_{i+1}\}$ und außerhalb von K_{i+1} den Wert 0 hat, um den Fluss $f_{i+1} = f_i + f_{K_{i+1}}$ zu erhalten.

Da sich die Kosten $k(f_{i+1}) = k(f_i) + k(f_{K_i}) = k(f_i) + c_{f_i}(K_i)k(K_i)$ von f_{i+1} wegen $k(K_i) \leq -1$ bei jeder Iteration um mindestens 1 verringern und die Kostendifferenz zwischen zwei beliebigen Flüssen in N_f durch $D = \sum_{e \in E} |k(e)|(c(e) + c(e^R))$ beschränkt ist, ist f_ℓ nach $\ell \leq D$ Iterationen ein kostenminimaler Fluss.

Da der BFM-Algorithmus in Zeit $O(mn)$ läuft, führt dies auf eine Laufzeit von $O(Dmn)$, um die Kosten von f_0 zu minimieren. Berechnen wir den maximalen Fluss f_0 in N mit Dinitz in Zeit $O(n^3)$, so erhalten wir folgenden Satz.

Satz 3.32. *Sei $N = (V, E, s, t, c, k)$ ein Netzwerk mit Kostenfunktion k. Dann kann in N ein maximaler Fluss mit minimalen Kosten in Zeit $O(n^3 + Dmn)$ bestimmt werden, wobei $D = \sum_{e \in E} |k(e)|(c(e) + c(e^R))$ ist.*

Das nächste Lemma zeigt einen Weg, wie sich in einem Netzwerk ohne negative Kreise ein maximaler Fluss f mit minimalen Kosten in Zeit $O(|f|mn)$ berechnen lässt.

Lemma 3.33. *Sei f_i ein Fluss in N mit $k(f_i) = k_{\min}(f_i)$. Dann ist $f_{i+1} = f_i + f_{P_{i+1}}$ für jeden Zunahmepfad P_{i+1} in N_{f_i} mit*

$$k(P_{i+1}) = \min\{k(P') \mid P' \text{ ist ein Zunahmepfad in } N_{f_i}\}$$

ein Fluss in N mit $k(f_{i+1}) = k_{\min}(f_{i+1})$.

Beweis. Unter der Annahme, dass $k(f_{i+1}) > k_{\min}(f_{i+1})$ ist, gibt es nach Lemma 3.31 einen negativen Kreis K in $N_{f_{i+1}}$. Wir benutzen K, um P_{i+1} in einen Zunahmepfad P' in N_{f_i} mit $k(P') < k(P_{i+1})$ zu transformieren.

Sei $F = K + P_{i+1}$ die Multimenge aller Kanten, die auf K oder P_{i+1} liegen, d.h. jede Kante in $K \Delta P_{i+1} = (K \setminus P_{i+1}) \cup (P_{i+1} \setminus K)$ kommt genau einmal und jede Kante in $K \cap P_{i+1}$ kommt genau zweimal in F vor. F ist also ein Multigraph bestehend aus dem s-t-Pfad P_{i+1} in N_{f_i} und dem Kreis K in $N_{f_{i+1}}$ und es gilt $k(F) = k(P_{i+1}) + k(K) < k(P_{i+1})$.

Um in F einen s-t-Pfad P' von N_{f_i} mit $k(P') < k(P_{i+1})$ zu finden, entfernen wir wie folgt alle Kanten in $\hat{F} = F \setminus E_{f_i} = K \setminus E_{f_i}$ aus F. Wegen $\hat{F} \subseteq K \setminus P_{i+1}$ kommt jede Kante $e \in \hat{F}$ genau einmal in F vor. Zudem wird jede Kante $e \in \hat{F}$ wegen

- $f_i(e) = c(e)$ (da $e \notin E_{f_i}$) zwar von f_i, aber wegen
- $e \in K \subseteq E_{f_{i+1}}$ nicht von f_{i+1} gesättigt.

Daher muss $f_i(e) \neq f_{i+1}(e)$ und somit $e^R \in P_{i+1}$ sein (da $e \notin P_{i+1}$). Wegen $e^R \in P_{i+1} \setminus K$ (da $e \in K$) kommt also für jede Kante $e \in \hat{F}$ auch e^R genau einmal in F vor.

Entfernen wir nun alle solchen Kantenpaare e, e^R aus F, so erhalten wir die Multimenge $F' = F \setminus (\hat{F} \cup \hat{F}^R) \subseteq E_{f_i}$, die wegen $k(e) + k(e^R) = 0$ dieselben Kosten $k(F') = k(F) < k(P_{i+1})$ wie F hat. Da F' zudem aus F durch Entfernen von Kreisen (der Länge 2) entsteht, ist F' wie F ein Multigraph, der sich in einen s-t-Pfad P' und eine gewisse Anzahl $\ell \geq 0$ von Kreisen $K_1, \ldots, K_\ell$ in N_{f_i} zerlegen lässt. Da nach

Voraussetzung keine negativen Kreise in N_{f_i} existieren, folgt

$$k(P') = k(F') - \sum_{i=1}^{\ell} k(K_i) \leq k(F') = k(F) < k(P_{i+1}). \qquad \blacksquare$$

Basierend auf Lemma 3.33 können wir nun folgenden Algorithmus zur Bestimmung eines maximalen Flusses mit minimalen Kosten in einem Netzwerk N angeben, das keine negativen Kreise enthält.

Algorithmus `Min-Cost-Flow`(V, E, s, t, c, k)

```
1  for all (u, v) ∈ V × V do
2    f(u, v) := 0
3  while P := min-zunahmepfad(f) ≠ ⊥ do
4    addierepfad(f, P)
```

Hierbei berechnet die Prozedur `min-zunahmepfad`(f) einen Zunahmepfad P in N_f mit minimalen Kosten. Da dann der Fluss $f' = f + f_P$ nach obigem Lemma minimale Kosten $k(f') = k_{\min}(f')$ in N hat, hat auch $N_{f'}$ keine negativen Kreise. Daher kann P bspw. mit dem BFM-Algorithmus berechnet werden, der in Zeit $O(mn)$ läuft. Dies führt auf eine Gesamtlaufzeit von $O(Mmn)$, wobei $M = |f|$ die Größe eines maximalen Flusses f in N ist.

Unter Verwendung einer Preisfunktion p können wir die Laufzeit für Netzwerke ohne negative Kreise mit Dijkstra auf $O(mn + |f|m \log n)$ (bzw. auf $O(|f|m \log n)$, falls p bereits bekannt ist) verbessern.

Satz 3.34. *In einem Netzwerk N kann ein maximaler Fluss f mit minimalen Kosten in Zeit $O(mn + |f|m \log n)$ bestimmt werden, falls N keine negativen Kreise hat.*

Definition 3.35. *Sei $G = (V, E, k)$ ein Digraph mit Kostenfunktion $k : E \to \mathbb{Z}$. Eine Funktion $p : V \to \mathbb{Z}$ heißt **Preisfunktion** für G, falls für jede Kante $e = (u, v)$ in E die Ungleichung*

$$k(u, v) \geq p(v) - p(u)$$

*gilt. Die bzgl. p **reduzierte Kostenfunktion** $k^p : E \to \mathbb{N}_0$ ist*

$$k^p(u, v) = k(u, v) + p(u) - p(v).$$

Wie man leicht sieht, überträgt sich die Antisymmetrie von k auf k^p:

$$k^p(u, v) = k(u, v) + p(u) - p(v) = -k(v, u) - p(v) + p(u) = -k^p(v, u).$$

Lemma 3.36. *Ein Digraph $G = (V, E, k)$ mit Kostenfunktion $k : E \to \mathbb{Z}$ hat genau dann keine negativen Kreise, wenn es eine Preisfunktion p für G gibt. Zudem lässt sich eine geeignete Preisfunktion p in Zeit $O(nm)$ finden.*

Beweis. Wir zeigen zuerst die Rückwärtsrichtung. Sei also p eine Preisfunktion mit $k^p(e) \geq 0$ für alle $e \in E$. Dann gilt für jede Kantenmenge $F \subseteq E$ die Ungleichung $k^p(F) \geq 0$. Da zudem für jeden Kreis K in G die Gleichheit $k(K) = k^p(K)$ gilt, folgt sofort $k(K) = k^p(K) \geq 0$.

Für die Vorwärtsrichtung sei nun $G = (V, E, k)$ ein Digraph ohne negative Kreise. Betrachte den Digraphen $G' = (V', E', k')$, der aus G durch Hinzunahme eines Knotens s' und von Kanten (s', x) mit $k'(s', x) = 0$ für alle $x \in V$ entsteht. Dann gibt es auch in G' keine negativen Kreise und wir können mit BFM für jeden Knoten $x \in V$ einen bzgl. k' kürzesten Pfad P_x von s nach x in Zeit $O(mn)$ berechnen. Setzen wir $p(x)$ gleich der Länge von P_x, so gilt für jede Kante $e = (u, v) \in E$ die Ungleichung

$$p(v) \leq p(u) + k(u, v),$$

d.h. $p(x)$ ist die gesuchte Preisfunktion für G. $\blacksquare$

Um einen maximalen Fluss mit minimalen Kosten in einem Netzwerk $N = (V, E, s, t, c, k)$ ohne negative Kreise in Zeit $O(mn + |f|m \log n)$ zu berechnen, rufen wir zuerst BFM mit Startknoten s' auf, um nach einem negativen Kreis K im erweiterten Digraphen $G' = (V', E', k')$ zu suchen. Wird K nicht gefunden, liefert BFM hierbei eine Preisfunktion p_0 für das Netzwerk $N_0 = N_{f_0} = N$, wobei f_0 der Nullfluss in N ist. Nun können wir mit Dijkstra für $i = 0, 1, \ldots$ in Zeit $O(m \log n)$

- einen bzgl. k^{p_i} kürzesten Zunahmepfad P_{i+1} in $N_{f_i}^{p_i}$ und
- einen Fluss $f_{i+1} = f_i + f_{P_{i+1}}$ der Größe $|f_{i+1}| > |f_i|$ mit minimalen Kosten $k(f_{i+1}) = k_{min}(f_{i+1})$
- sowie eine Preisfunktion p_{i+1} für $N_{f_{i+1}}$ berechnen.

Ein bzgl. k^{p_i} kürzester s-t-Pfad P ist nämlich auch bzgl. k ein kürzester s-t-Pfad, da $k^{p_i}(P) = k(P) + p_i(s) - p_i(t)$ und $p_i(s) - p_i(t)$ eine von P unabhängige Konstante ist. Zudem lässt sich aus p_i nach folgendem Lemma eine Preisfunktion $p_{i+1} = p_i + \ell_i$ für k in $N_{f_{i+1}}$ finden, wobei $\ell_i(v)$ die minimale Länge $k^{p_i}(P)$ eines s-v-Pfades P in $N_{f_i}^{p_i}$ ist. Dabei kann ℓ_i und somit auch p_{i+1} von Dijkstra zusammen mit P_{i+1} in Zeit $O(m \log n)$ gleich mitberechnet werden.

Lemma 3.37. *Sei $\ell_i(v)$ die minimale Pfadlänge von s nach v in $N_{f_i}^{p_i}$ bzgl. k^{p_i}, wobei $p_i : V \to \mathbb{Z}$ eine beliebige Funktion ist.*

- *Dann ist $p_{i+1} = p_i + \ell_i$ eine Preisfunktion für k in N_{f_i} und in $N_{f_{i+1}}$.*
- *Zudem gilt $k^{p_{i+1}}(e) = 0$ für alle Kanten e, die auf einem kürzesten s-v-Pfad in $N_{f_i}^{p_i}$ liegen.*

Beweis. Wir zeigen zuerst, dass p_{i+1} eine Preisfunktion für k in N_{f_i} ist. Da $\ell_i(v) \leq \ell_i(u) + k^{p_i}(e)$ und $k^{p_i}(e) = k(e) + p_i(u) - p_i(v)$ für jede Kante $e = (u, v) \in E_{f_i}$ gilt, folgt

$$\begin{aligned} k^{p_{i+1}}(e) &= k(e) + p_{i+1}(u) - p_{i+1}(v) \\ &= k(e) + p_i(u) + \ell_i(u) - p_i(v) - \ell_i(v) \\ &= k^{p_i}(e) + \ell_i(u) - \ell_i(v) \geq 0. \end{aligned}$$

Also ist p_{i+1} eine Preisfunktion für k in N_{f_i}. Falls $e = (u, v)$ auf einem bzgl. k^{p_i} kürzesten Pfad in N_{f_i} liegt, gilt sogar $k^{p_{i+1}}(u, v) = 0$, da dann $\ell_i(v) = \ell_i(u) + k^{p_i}(u, v)$ ist.

Da schließlich für jede Kante e in $N_{f_{i+1}}$, die nicht zu N_{f_i} gehört, die Kante e^R auf dem kürzesten s-t-Pfad P_{i+1} liegt, folgt wegen der Antisymmetrie $k^{p_{i+1}}(e) = -k^{p_{i+1}}(e^R) = 0$. Also ist p_{i+1} auch in $N_{f_{i+1}}$ eine Preisfunktion für k. ■

3.5 Kürzeste Pfade

In vielen Anwendungen tritt das Problem auf, einen kürzesten Pfad von einem Startknoten s zu einem Zielknoten t in einem Digraphen zu finden, dessen Kanten (u, v) vorgegebene ***Längen*** $\ell(u, v) \geq 0$ haben. Die Länge eines Pfades $P = (v_0, \ldots, v_j)$ ist $\ell(P) = \sum_{i=0}^{j-1} \ell(v_i, v_{i+1})$. Die kürzeste Pfadlänge von s nach t wird als ***Distanz*** $d(s, t)$ zwischen s und t bezeichnet,

$$d(s, t) = \min\{\ell(P) \mid P \text{ ist ein } s\text{-}t\text{-Pfad}\}$$

Falls kein s-t-Pfad existiert, setzen wir $d(s, t) = \infty$.

3.5.1 Der Dijkstra-Algorithmus

Der Dijkstra-Algorithmus findet einen kürzesten Pfad P_u von s zu allen erreichbaren Knoten u (single-source shortest-path problem). Dabei werden alle Knoten u, für die bereits ein s-u-Pfad P_u bekannt ist, zusammen mit der Pfadlänge $d(u) = \ell(P_u)$ in einer Menge U gespeichert und erst dann wieder aus U entfernt, wenn $d(u) = d(s, u)$ ist. Für eine effiziente Implementierung benötigen wir für U eine Datenstruktur, auf der sich folgende Operationen effizient ausführen lassen.

Init(U): Initialisiert U als leere Menge.

Update(U, u, d): Erniedrigt den Wert von u auf d (nur wenn der aktuelle Wert größer als d ist), ist u noch nicht in U enthalten, wird u mit dem Wert d zu U hinzugefügt.

GetMin(U): Gibt ein Element aus U mit dem kleinsten d-Wert zurück und entfernt es aus U (ist U leer, wird der Wert `nil` zurückgegeben).

Voraussetzung für die Korrektheit des Algorithmus' ist, dass alle Kanten eine nichtnegative Länge $\ell(u,v) \geq 0$ haben. In diesem Fall wird $D = (V, E, \ell)$ auch ***Distanzgraph*** genannt. Während der Suche werden bestimmte Kanten $e = (u,v)$ daraufhin getestet, ob $d(u) + \ell(u,v) < d(v)$ ist. Da in diesem Fall die Kante e auf eine Herabsetzung von $d(v)$ auf den Wert $d(u) + \ell(u,v)$ „drängt", wird diese Wertzuweisung als ***Relaxation*** von e bezeichnet. Welche Kanten (u,v) auf Relaxation getestet werden, wird beim Dijkstra-Algorithmus durch eine einfache ***Greedystrategie*** bestimmt:

- Wähle $u \in U$ mit minimalem d-Wert und teste alle Kanten (u,v), für die v nicht schon abgearbeitet ist.

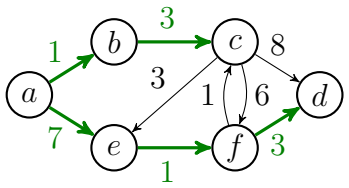
Der Algorithmus führt also eine modifizierte ***Breitensuche*** aus, bei der die in Bearbeitung befindlichen Knoten in einer ***Prioritätswarteschlange*** U verwaltet werden.

Algorithmus `Dijkstra`(V, E, ℓ, s)

```
1  for all v ∈ V do
2    d(v) := ∞
3    parent(v) := nil
4    done(v) := false
5  d(s) := 0
6  Init(U)
7  Update(U, s, 0)
8  while u := GetMin(U) ≠ nil do
9    done(u) := true
10   for all v ∈ N⁺(u) do
11     if done(v) = false ∧ d(u) + l(u,v) < d(v) then
12       d(v) := d(u) + l(u,v)
13       Update(U, v, d(v))
14       parent(v) := u
```

Der Algorithmus speichert die aktuelle Länge des Pfades P_u in $d(u)$. Knoten außerhalb des aktuellen Breitensuchbaums T haben den d-Wert ∞. In jedem Schleifendurchlauf wird in Zeile 8 ein Knoten u mit minimalem d-Wert aus U entfernt und als abgearbeitet markiert. Anschließend wird für alle von u wegführenden Kanten $e = (u,v)$ geprüft, ob eine Relaxation ansteht. Wenn ja, wird U aktualisiert und e wird neue Baumkante in T.

Beispiel 3.38. *Betrachte den nebenstehenden Distanzgraphen G. Bei Ausführung des Dijkstra-Algorithmus' mit dem Startknoten a werden die folgenden kürzesten Wege berechnet.*



Inhalt von P	entfernt	besuchte Kanten	Update-Op.
$(a,0)$	$(a,0)$	$(a,b),(a,e)$	$(b,1),(e,7)$
$(b,1),(e,7)$	$(b,1)$	(b,c)	$(c,4)$
$(c,4),(e,7)$	$(c,4)$	$(c,d),(c,e),(c,f)$	$(d,12),(f,10)$
$(e,7),(f,10),(d,12)$	$(e,7)$	(e,f)	$(f,8)$
$(f,8),(d,12)$	$(f,8)$	$(f,c),(f,d)$	$(d,11)$
$(d,11)$	$(d,11)$	–	–

◁

Als nächstes beweisen wir die Korrektheit des Dijkstra-Algorithmus'.

Satz 3.39. *Sei $D = (V, E, l)$ ein Distanzgraph und sei $s \in V$. Dann berechnet* `Dijkstra`(V, E, l, s) *im Feld* `parent` *für alle von s aus erreichbaren Knoten $t \in V$ einen kürzesten s-t-Weg $P(t)$.*

Beweis. Wir zeigen zuerst, dass alle von s aus erreichbaren Knoten $t \in V$ zu U hinzugefügt werden. Dies folgt aus der Tatsache, dass s zu U hinzugefügt wird, und spätestens dann, wenn ein Knoten u in Zeile 4 aus U entfernt wird, sämtliche Nachfolger von u zu U hinzugefügt werden.

Zudem ist klar, dass $d(t) \geq d(s,t)$ ist, da P_t im Fall $d(t) < \infty$ ein s-t-Pfad der Länge $d(t)$ ist. Es bleibt noch zu zeigen, dass P_u ein kürzester s-u-Pfad ist, sobald u aus U entfernt wird, d.h. es gilt $d(u) = d(s,u)$.

Wir zeigen induktiv über die Anzahl k der vor u aus U entfernten Knoten, dass $d(u) \leq d(s,u)$ ist.

- Im Fall $k = 0$ ist $u = s$ und P_s hat die Länge $d(s) = 0 = d(s,s)$.
- Im Fall $k \geq 1$ sei $P = v_0, \ldots, v_j = u$ ein kürzester s-u-Pfad in G und sei v_i der Knoten mit maximalem Index i, der vor u aus U entfernt wird. Nach IV gilt dann

$$d(v_i) = d(s, v_i). \tag{3.4}$$

Zudem ist

$$d(v_{i+1}) \leq d(v_i) + l(v_i, v_{i+1}). \tag{3.5}$$

Da u im Fall $u \neq v_{i+1}$ vor v_{i+1} aus P entfernt wird, ist

$$d(u) \leq d(v_{i+1}) \tag{3.6}$$

und es folgt

$$\begin{aligned} d(u) &\overset{(3.6)}{\leq} d(v_{i+1}) \overset{(3.5)}{\leq} d(v_i) + l(v_i, v_{i+1}) \\ &\overset{(3.4)}{=} d(s, v_i) + l(v_i, v_{i+1}) = d(s, v_{i+1}) \leq d(s,u). \end{aligned}$$ ■

Um die Laufzeit des Dijkstra-Algorithmus' abzuschätzen, überlegen wir uns zuerst, wie oft die einzelnen Operationen `Init`, `GetMin` und `Update` auf der Datenstruktur U ausgeführt werden. Sei $n = \|V\|$ die Anzahl der Knoten und $m = \|E\|$ die Anzahl der Kanten des Eingabegraphen.

- Die `Init`-Operation wird nur einmal ausgeführt.
- Da die while-Schleife für jeden von s aus erreichbaren Knoten genau einmal durchlaufen wird, wird die `GetMin`-Operation höchstens $\min(n,m)$-mal ausgeführt.
- Da der Dijkstra-Algorithmus jede Kante höchstens einmal besucht, wird die `Update`-Operation höchstens m-mal ausgeführt

Beobachtung 3.40. *Bezeichne Init(n), GetMin(n) und Update(n) den Aufwand zum Ausführen der Operationen* `Init`, `GetMin` *und* `Update` *für den Fall, dass U nicht mehr als n Elemente aufzunehmen hat. Dann ist die Laufzeit des Dijkstra-Algorithmus' durch*

$$\mathcal{O}(n + m + \mathit{Init}(n) + \min(n,m) \cdot \mathit{GetMin}(n) + m \cdot \mathit{Update}(n))$$

beschränkt.

Die Laufzeit des Dijkstra-Algorithmus' hängt also wesentlich davon ab, wie wir die Datenstruktur U implementieren. Falls alle Kanten die gleiche Länge haben, wachsen die Distanzwerte der Knoten monoton in der Reihenfolge ihres Besuchs. In diesem Fall können wir U als Warteschlange implementieren, d.h. Dijkstra vereinfacht sich zur Prozedur `BFS` und läuft in Zeit $\mathcal{O}(n+m)$.

Falls die Kanten unterschiedliche Längen haben, betrachten wir folgende drei Möglichkeiten.

1. Da die Felder d und `done` bereits alle nötigen Informationen enthalten, kann man auf die (explizite) Implementierung von U auch verzichten. In diesem Fall kostet die `GetMin`-Operation allerdings Zeit $\mathcal{O}(n)$, was auf eine Gesamtlaufzeit von $\mathcal{O}(n^2)$ führt.

   Dies ist asymptotisch optimal, wenn G relativ dicht ist, also $m = \Omega(n^2)$ Kanten enthält. Ist G dagegen relativ dünn, d.h. $m = o(n^2)$, so empfiehlt es sich, U als Prioritätswarteschlange zu implementieren.
2. Es ist naheliegend, U in Form eines Heaps H zu implementieren. In diesem Fall lässt sich die Operation `GetMin` in Zeit $\mathcal{O}(\log n)$ implementieren. Da die Prozedur `Update` einen linearen Zeitaufwand erfordert, ist es effizienter, sie durch eine `Insert`-Operation zu simulieren. Dies führt zwar dazu, dass derselbe Knoten evtl. mehrmals mit unterschiedlichen Werten in H gespeichert wird. Die Korrektheit bleibt aber dennoch erhalten, wenn wir nur die erste Entnahme eines Knotens aus H beachten und die übrigen ignorieren.

Da für jede Kante höchstens ein Knoten in H eingefügt wird, erreicht H maximal die Größe n^2 und daher sind die Heap-Operationen `Insert` und `GetMin` immer noch in Zeit $\mathcal{O}(\log n^2) = \mathcal{O}(\log n)$ ausführbar. Insgesamt erhalten wir somit eine Laufzeit von $\mathcal{O}(n+m\log n)$. Dies ist zwar für dünne Graphen sehr gut, aber für dichte Graphen schlechter als als die implizite Implementierung von U mithilfe der Felder d und `done`.

3. Als weitere Möglichkeit kann U auch in Form eines so genannten ***Fibonacci-Heaps*** F implementiert werden. Dieser benötigt nur eine konstante amortisierte Laufzeit $\mathcal{O}(1)$ für die `Update`-Operation und $\mathcal{O}(\log n)$ für die `GetMin`-Operation. Insgesamt führt dies auf eine Laufzeit von $\mathcal{O}(m + n\log n)$. Allerdings sind Fibonacci-Heaps erst bei sehr großen Graphen mit mittlerer Dichte schneller.

Eine offene Frage ist, ob es auch einen Algorithmus mit linearer Laufzeit $\mathcal{O}(n+m)$ gibt. Wir fassen die verschiedenen Möglichkeiten zur Implementation der Datenstruktur U in folgender Tabelle zusammen.

	implizit	Heap	Fibonacci-Heap
`Init`	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`Update`	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
`GetMin`	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
Gesamtlaufzeit	$\mathcal{O}(n^2)$	$\mathcal{O}(n+m\log n)$	$\mathcal{O}(m+n\log n)$

3.5.2 Der Ford-Algorithmus

In manchen Anwendungen treten negative Kantengewichte $k(e)$ auf. Geben die Kantengewichte $k(e)$ beispielsweise die mit einer Kante e verbundenen Kosten wider, so kann ein Gewinn durch negative Kosten modelliert werden. Auf diese Weise lassen sich auch längste Pfade in Distanzgraphen (V, E, ℓ) berechnen, indem man alle Kantenlängen $k(u,v) = -\ell(u,v)$ setzt und in dem resultierenden Kostengraphen (V, E, k) einen kürzesten Pfad bestimmt.

Die Komplexität des Problems hängt wesentlich davon ab, ob gerichtete Kreise mit negativer Länge (bzw. Kosten) vorkommen oder nicht. Falls negative Kreise vorkommen können, ist das Problem NP-hart. Andernfalls existieren effiziente Algorithmen wie z.B.

- der Bellman-Ford-Algorithmus (BF-Algorithmus) oder
- der Bellman-Ford-Moore-Algorithmus (BFM-Algorithmus).

Ausgangspunkt für diese beiden Algorithmen ist der Ford-Algorithmus. Dieser arbeitet ganz ähnlich wie der Dijkstra-Algorithmus, betrachtet aber jede Kante nicht nur einmal, sondern eventuell mehrmals. In seiner einfachsten Form sucht der Algorithmus wiederholt nach einer Kante $e = (u,v)$ mit

$$d(u) + k(u,v) < d(v)$$

und aktualisiert den Wert von $d(v)$ auf $d(u) + k(u,v)$ (Relaxation). Die Laufzeit hängt dann wesentlich davon ab, in welcher Reihenfolge die Kanten auf Relaxation getestet werden. Im besten Fall lässt sich eine lineare Laufzeit erreichen (z.B. wenn überhaupt keine Kreise existieren).

Bei der Bellman-Ford-Variante wird in $\mathcal{O}(mn)$ Schritten ein kürzester Pfad von s zu allen erreichbaren Knoten gefunden (sofern keine negativen Kreise existieren).

Wir zeigen induktiv über die Anzahl k der Kanten eines kürzesten s-t-Pfades, dass $d(t) \le d(s,t)$ gilt, falls d für alle Kanten $(u,v) \in E$ die Dreiecksungleichung $d(v) \le d(u) + k(u,v)$ erfüllt (also d keine weiteren Relaxationen erlaubt)

- Im Fall $k = 0$ ist $t = s$ und somit $d(s) = 0 = d(s,s)$.
- Im Fall $k > 0$ sei t ein Knoten, dessen kürzester s-t-Pfad P aus k Kanten besteht. Nach IV gilt dann $d(u) \le d(s,u)$ für den Vorgänger u von t auf P. Da d die Dreiecksungleichung für (u,t) erfüllt, folgt daher

$$d(t) \le d(u) + k(u,t) \le d(s,u) + k(u,t) = d(s,t).$$

Aus dem Beweis folgt zudem, dass $d(t)$ nach Relaxation aller Kanten eines kürzesten s-t-Pfades P (in der Reihenfolge, in der die Kanten auf P angeordnet sind) einen Wert $d(t) \leq d(s,t)$ hat. Dies gilt auch, wenn dazwischen noch weitere Kanten relaxiert werden.

3.5.3 Der Bellman-Ford-Algorithmus

Die Bellman-Ford-Variante testet in den ersten $n-1$ Runden jeweils alle Kanten auf Relaxation und speichert dabei für jede relaxierte Kante (u,v) den Knoten u in $\texttt{parent}(v)$. Da nach diesen $n-1$ Runden auf allen kürzesten Pfaden alle Kanten in der richtigen Reihenfolge relaxiert wurden, gilt nun $d(u) \leq d(s,u)$ für alle Knoten $u \in V$.

Kann daher in der n-ten Runde immer noch eine Kante $e = (u,v)$ relaxiert werden, so lässt sich wie folgt ein negativer Kreis K finden:

- Wir gehen von u aus mittels `parent` solange zurück, bis wir zum zweiten Mal am gleichen Knoten ankommen

Algorithmus $\texttt{BF}(V,E,k,s)$

```
1  for all v ∈ V do
2    d(v) := ∞
3    parent(v) := nil
4  d(s) := 0
5  for i := 1 to n − 1 do
6    for all (u,v) ∈ E do
7      if d(u) + k(u,v) < d(v) then
8        d(v) := d(u) + k(u,v)
9        parent(v) := u
10 for all (u,v) ∈ E do
11   if d(u) + k(u,v) < d(v) then
12     print (es gibt einen negativen Kreis)
```

Die Laufzeit ist offensichtlich $\mathcal{O}(mn)$.

3.5.4 Der BFM-Algorithmus

Die BFM-Variante versucht eine Kante (u,v) nur dann zu relaxieren, wenn $d(u)$ in der Runde davor erniedrigt wurde.

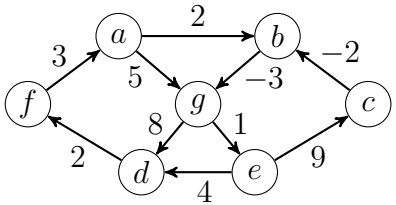
Um in Runde $i+1$ alle Knoten parat zu haben, deren d-Wert in Runde i verringert wurde, wird bei jeder Relaxation einer Kante (u,v) der Knoten v in einer Schlange Q gespeichert (falls er nicht schon in Q ist). Dabei kann in Q zusammen mit v auch die Rundenzahl $i+1$ der anstehenden Tests aller von v ausgehenden Kanten gespeichert werden.

Sobald aus Q ein Knoten mit Rundenzahl $n+1$ entfernt wird, bricht der Algorithmus mit der Meldung ab, dass negative Kreise existieren. Da gegenüber der BF-Variante nur vergebliche Relaxationsversuche vermieden werden, liefern BF und BFM dasselbe Ergebnis.

Algorithmus $\texttt{BFM}(V,E,k,s)$

```
1  for all v ∈ V do
2    d(v) := ∞, parent(v) := nil, inQueue(v) := false
3  d(s) := 0, Init(Q), Enqueue(Q,(1,s)), inQueue(s) := true
4  while (i,u) := Dequeue(Q) ≠ nil and i ≤ n do
5    inQueue(u) := false
6    for all v ∈ N+(u) do
7      if d(u) + k(u,v) < d(v) then
8        d(v) := d(u) + k(u,v)
9        parent(v) := u
10       if inQueue(v) = false then
11         Enqueue(Q,(i+1,v))
12         inQueue(v) := true
13 if i > n then
14   print (es gibt einen negativen Kreis)
```

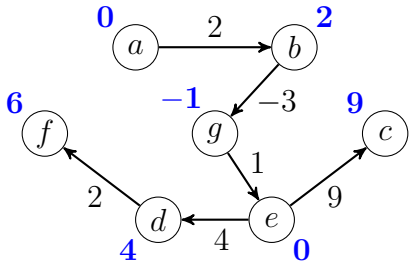
Beispiel 3.41. *Betrachte untenstehenden kantenbewerteten Digraphen mit dem Startknoten a.*



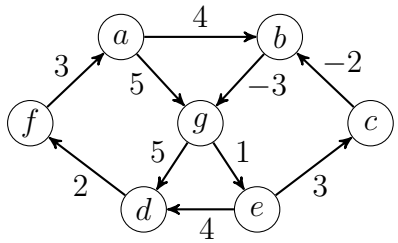
Die folgende Tabelle zeigt jeweils den Inhalt der Schlange Q, bevor der BFM-Algorithmus das nächste Paar (i, u) von Q entfernt. Dabei enthält jeder Eintrag $(i, u, \texttt{parent}(u), d(u))$ neben der Rundenzahl i und dem Knoten u auch noch den parent-Knoten und den aktuellen d-Wert von u, obwohl der BFM-Algorithmus diese Information nicht in Q speichert.

$$
\begin{array}{l|l|l|l|l}
\Uparrow & & & & \\
(1,a,\texttt{nil},0) & (2,b,a,2) & (2,g,b,-1) & (3,d,g,7) & \\
\Uparrow & (2,g,a,5) & & (3,e,g,0) & \\
(3,e,g,0) & (4,f,d,9) & & & \\
(4,f,d,9) & (4,c,d,9) & (4,c,d,9) & (4,d,e,4) & \\
 & (4,d,e,4) & (4,d,e,4) & & (5,f,d,6)
\end{array}
$$

Die berechneten Entfernungen mit den zugehörigen `parent`*-Pfaden sind in folgendem Suchbaum widergegeben:*



Als nächstes betrachten wir den folgenden Digraphen:

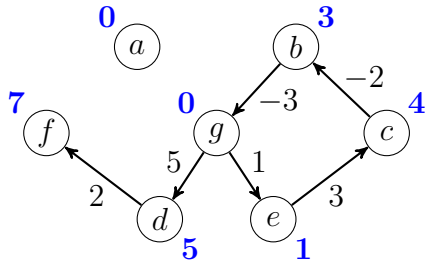


Da dieser einen negativen Kreis enthält, der vom Startknoten aus erreichbar ist, lassen sich die Entfernungen zu allen Knoten, die von diesem Kreis aus erreichbar sind, beliebig verkleinern.

$$
\begin{array}{l|l|l|l|l|l}
\Uparrow & & & & & \\
(1,a,\texttt{nil},0) & (2,b,a,4) & (2,g,b,1) & (3,d,g,6) & & \\
\Uparrow & (2,g,a,5) & & (3,e,g,2) & (3,e,g,2) & \\
 & & & & (4,f,d,8) & (4,f,d,8) \\
(4,c,e,5) & & & & & (4,c,e,5) \\
 & (5,b,c,3) & & & & \\
 & & (6,g,b,0) & (7,d,g,5) & & \\
 & & & (7,e,g,1) & (7,e,g,1) & \\
 & & & & (8,f,d,7) & (8,f,d,7) \\
 & & & & & (8,c,e,4)
\end{array}
$$

Da nun der Knoten f mit der Rundenzahl $i = n = 8$ aus der Schlange entnommen wird, bricht der Algorithmus an dieser Stelle mit der Meldung ab, dass negative Kreise existieren. Ein solcher Kreis (im Beispiel: g, e, c, b, g) lässt sich bei Bedarf anhand der `parent`*-Funktion*

aufspüren, indem wir den `parent`*-Weg zu f zurückverfolgen: f, d, g, b, c, e, g.*



◁

4 Matchings

Definition 4.1. *Sei $G = (V, E)$ ein Graph.*

- *Zwei Kanten $e, e' \in E$ heißen **unabhängig**, falls $e \cap e' = \emptyset$ ist.*
- *Eine Kantenmenge $M \subseteq E$ heißt **Matching** in G, falls alle Kanten in M paarweise unabhängig sind.*
- *Sei $M \subseteq E$. Ein Knoten $v \in V$ heißt **M-gebunden**, falls v Endpunkt einer Kante $e \in M$ (also $v \in \bigcup M$) ist und sonst **M-frei**.*
- *Wir sagen auch, **M bindet v** bzw. **M lässt v frei**.*
- *Ein Matching M heißt **perfekt**, falls alle Knoten in G M-gebunden sind (also $V = \bigcup M$ ist).*
- *Die **Matchingzahl** von G ist $\boldsymbol{\mu(G)} = \max\{|M| \mid M \text{ ist ein Matching in } G\}$.*
- *Ein Matching M heißt **maximal** (engl. maximum), falls $|M| = \mu(G)$ ist. M heißt **gesättigt** (engl. maximal), falls es in keinem größeren Matching enthalten ist.*

Offensichtlich ist $M \subseteq E$ genau dann ein Matching, wenn $|\bigcup M| = 2|M|$ ist. Das Ziel besteht nun darin, ein maximales Matching M in einem gegebenen Graphen G zu finden.

Beispiel 4.2. *Ein gesättigtes Matching muss nicht maximal sein:*

$M = \{\{v, w\}\}$ ist gesättigt, da es sich nicht erweitern lässt. M ist jedoch kein maximales Matching, da $M' = \{\{v, x\}, \{u, w\}\}$ ein größeres Matching ist. Die Greedy-Methode, ausgehend von $M = \emptyset$ solange

Kanten zu M hinzuzufügen, bis sich M nicht mehr zu einem größeren Matching erweitern lässt, funktioniert also nicht.

Durch eine einfache Reduktion des bipartiten Matchingproblems auf ein Flussproblem erhält man aus Korollar 3.29 das folgende Resultat.

Satz 4.3. *In einem bipartiten Graphen $G = (A, B, E)$ lässt sich ein maximales Matching in Zeit $O(m\sqrt{n})$ bestimmen.*

Beweis. Wir konstruieren zu G das Netzwerk $N = (V, E', s, t, c)$ mit der Knotenmenge $V = A \cup B \cup \{s, t\}$ und der Kantenmenge

$$E' = (\{s\} \times A) \cup \Big\{(u, v) \in A \times B \;\Big|\; \{u, v\} \in E\Big\} \cup (B \times \{t\}),$$

die alle Kapazität 1 haben. Da alle Knoten $u \in A \cup B$ den Durchsatz $D(u) = 1$ in N haben, liefert jeder Fluss f in N ein Matching $M = \Big\{\{u, w\} \in E \mid f(u, w) = 1\Big\}$ in G mit $|M| = |f|$ und umgekehrt. Es genügt also, einen maximalen Fluss in N zu finden.

Nach Korollar 3.29 ist dies mit dem Algorithmus von Dinitz unter Verwendung von `blockfluss1` in Zeit $O(m\sqrt{n})$ möglich, da der Durchsatz aller Knoten (außer s und t) durch 1 beschränkt ist. ■

In den Übungen wird gezeigt, dass sich die Laufzeit durch eine verbesserte Analyse sogar durch $O(m\sqrt{\mu})$ begrenzen lässt.

Die Konstruktion im Beweis von Satz 4.3 lässt sich nicht ohne Weiteres auf Graphen verallgemeinern, die nicht bipartit sind. Wir werden jedoch sehen, dass sich manche bei den Flussalgorithmen verwendete Ideen auch für Matchingalgorithmen einsetzen lassen. So lassen sich Matchings, die nicht maximal sind, ähnlich vergrößern wie dies bei nicht maximalen Flüssen durch einen Zunahmepfad möglich ist.

Definition 4.4. *Sei $G = (V, E)$ ein Graph und sei M ein Matching in G.*

1. *Ein Pfad $P = (u_0, \ldots, u_l)$ in G der Länge $\ell \geq 1$ heißt **M-alternierend**, falls für $i = 1, \ldots, \ell - 1$ gilt:*

$$e_i = \{u_{i-1}, u_i\} \in M \Leftrightarrow e_{i+1} = \{u_i, u_{i+1}\} \notin M.$$

2. *Ein Kreis $C = (u_1, \ldots, u_\ell, u_1)$ in G heißt **M-alternierend**, falls der Pfad $P = (u_1, \ldots, u_l)$ M-alternierend ist und zudem gilt:*

$$\{u_1, u_2\} \in M \Leftrightarrow \{u_1, u_\ell\} \notin M.$$

3. *Ein M-alternierender Pfad $P = (u_0, \ldots, u_l)$ heißt **M-vergrößernder Pfad** (oder einfach **M-Pfad**), falls beide Endpunkte von P M-frei sind.*

Satz 4.5 (Lemma von Berge)**.** *Ein Matching M in G ist genau dann maximal, wenn es keinen M-Pfad in G gibt.*

Beweis. Ist $P = (u_0, \ldots, u_l)$ ein M-Pfad, so liefert $M' = M \triangle P$ ein Matching der Größe $|M'| = |M| + 1$ in G. Hierbei fassen wir P als Menge $\{\{u_{i-1}, u_i\} \mid i = 1, \ldots, l\}$ seiner Kanten auf.

Ist dagegen M nicht maximal und M' ein größeres Matching, so betrachten wir die Kantenmenge $M \triangle M'$. Da jeder Knoten in dem Graphen $G' = (V, M \triangle M')$ höchstens den Grad 2 hat, lässt sich G' in disjunkte Kreise und Pfade zerlegen. Da diese Kreise und Pfade M-alternierend sind, und M' größer als M ist, muss mindestens einer dieser Pfade ein M-Pfad sein. ■