Vorlesungsskript

Graphalgorithmen

Sommersemester 2015

Prof. Dr. Johannes Köbler
Sebastian Kuhnert
Humboldt-Universität zu Berlin
Lehrstuhl Komplexität und Kryptografie

24. Juni 2015

Inhaltsverzeichnis

1 Grundlagen

Der Begriff *Algorithmus* geht auf den persischen Gelehrten Muhammed Al Chwarizmi (8./9. Jhd.) zurück. Der älteste bekannte nichttriviale Algorithmus ist der nach *Euklid* benannte Algorithmus zur Berechnung des größten gemeinsamen Teilers zweier natürlicher Zahlen (300 v. Chr.). Von einem Algorithmus wird erwartet, dass er jede *Problemeingabe* nach endlich vielen Rechenschritten löst (etwa durch Produktion einer Ausgabe). Ein Algorithmus ist ein „Verfahren" zur Lösung eines Entscheidungs- oder Berechnungsproblems, das sich prinzipiell auf einer Turingmaschine (TM) implementieren lässt (Church-Turing-These).

Die Registermaschine

Bei Aussagen zur Laufzeit von Algorithmen beziehen wir uns auf die Registermaschine (engl. random access machine; RAM). Dieses Modell ist etwas flexibler als die Turingmaschine, da es den unmittelbaren Lese- und Schreibzugriff (random access) auf eine beliebige Speichereinheit (Register) erlaubt. Als Speicher stehen beliebig viele Register zur Verfügung, die jeweils eine beliebig große natürliche Zahl speichern können. Auf den Registerinhalten sind folgende arithmetische Operationen in einem Rechenschritt ausführbar: Addition, Subtraktion, abgerundetes Halbieren und Verdoppeln. Unabhängig davon geben wir die Algorithmen in Pseudocode an. Das RAM-Modell benutzen wir nur zur Komplexitätsabschätzung.

Die Laufzeit von RAM-Programmen wird wie bei TMs in der Länge der Eingabe gemessen. Man beachte, dass bei arithmetischen Problemen (wie etwa Multiplikation, Division, Primzahltests, etc.) die Länge einer Zahleingabe n durch die Anzahl $\lceil \log n \rceil$ der für die Binärkodierung von n benötigten Bits gemessen wird. Dagegen bestimmt bei nicht-arithmetischen Problemen (z.B. Graphalgorithmen oder Sortierproblemen) die Anzahl der gegebenen Zahlen, Knoten oder Kanten die Länge der Eingabe.

Asymptotische Laufzeit und Landau-Notation

Definition 1.1. *Seien f und g Funktionen von $\mathbb{N}$ nach $\mathbb{R}^+$. Wir schreiben $f(n) = \mathcal{O}(g(n))$, falls es Zahlen n_0 und c gibt mit*

$$\forall n \geq n_0 : f(n) \leq c \cdot g(n).$$

Die Bedeutung der Aussage $f(n) = \mathcal{O}(g(n))$ ist, dass f „nicht wesentlich schneller" als g wächst. Formal bezeichnet der Term $\mathcal{O}(g(n))$ die Klasse aller Funktionen f, die obige Bedingung erfüllen. Die Gleichung $f(n) = \mathcal{O}(g(n))$ drückt also in Wahrheit eine Element-Beziehung $f \in \mathcal{O}(g(n))$ aus. O-Terme können auch auf der linken Seite vorkommen. In diesem Fall wird eine Inklusionsbeziehung ausgedrückt. So steht $n^2 + \mathcal{O}(n) = \mathcal{O}(n^2)$ für die Aussage $\{n^2 + f \mid f \in \mathcal{O}(n)\} \subseteq \mathcal{O}(n^2)$.

Beispiel 1.2.

- $7\log(n) + n^3 = \mathcal{O}(n^3)$ *ist richtig.*
- $7\log(n)n^3 = \mathcal{O}(n^3)$ *ist falsch.*
- $2^{n+\mathcal{O}(1)} = \mathcal{O}(2^n)$ *ist richtig.*
- $2^{\mathcal{O}(n)} = \mathcal{O}(2^n)$ *ist falsch (siehe Übungen).* ◁

Es gibt noch eine Reihe weiterer nützlicher Größenvergleiche von Funktionen.

Definition 1.3. *Wir schreiben $f(n) = o(g(n))$, falls es für jedes $c > 0$ eine Zahl n_0 gibt mit*

$$\forall n \geq n_0 : f(n) \leq c \cdot g(n).$$

Damit wird ausgedrückt, dass f „wesentlich langsamer“ als g wächst. Außerdem schreiben wir

- $f(n) = \Omega(g(n))$ *für* $g(n) = \mathcal{O}(f(n))$, *d.h. f wächst mindestens so schnell wie g)*
- $f(n) = \omega(g(n))$ *für* $g(n) = o(f(n))$, *d.h. f wächst wesentlich schneller als g, und*
- $f(n) = \Theta(g(n))$ *für* $f(n) = \mathcal{O}(g(n)) \wedge f(n) = \Omega(g(n))$, *d.h. f und g wachsen ungefähr gleich schnell.*

1.1 Graphentheoretische Grundlagen

Definition 1.4. *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 $N_G(v) = \{u \in V \mid \{u, v\} \in E\}$.*

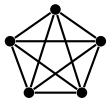
b) Der ***Grad*** *von v ist $\deg_G(v) = \|N_G(v)\|$.*

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

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

Beispiel 1.5.

- *Der* ***vollständige Graph*** *(V, E) auf n Knoten, d.h. $\|V\| = n$ und $E = \binom{V}{2}$, wird mit K_n und der* ***leere Graph*** *$(V, \emptyset)$ auf n Knoten wird mit 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 der Länge $n - 1$*** *wird mit P_n bezeichnet.*

P_2:  P_3:  P_4:  P_5: 

- *Der* ***Kreis der Länge n*** *wird mit C_n bezeichnet.*

C_3:  C_4:  C_5:  C_6: 

Definition 1.6. *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*

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

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*

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

c) 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. G heißt* ***k-färbbar****, falls eine Färbung $f\colon V \to \{1, \ldots, k\}$ existiert. Die* ***chromatische Zahl*** *ist*

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

d) Ein Graph heißt ***bipartit****, wenn $\chi(G) \leq 2$ ist.*

e) *Ein Graph $G' = (V', E')$ heißt* ***Sub-/Teil-/Untergraph*** *von G, falls $V' \subseteq V$ und $E' \subseteq E$ ist. Ein Subgraph $G' = (V', E')$ heißt (**durch V'**) **induziert**, falls $E' = E \cap \binom{V'}{2}$ ist. Hierfür schreiben wir auch $H = G[V']$.*

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 $i = 0, \ldots, \ell - 1$, der jede Kante $e \in E$ höchstens einmal durchläuft. 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) *Ein Graph $G = (V, E)$ heißt* ***zusammenhängend***, *falls es für alle Paare $\{u, v\} \in \binom{V}{2}$ einen u-v-Weg gibt. G heißt* ***k-fach zusammenhängend***, *$1 < k < n$, falls G nach Entfernen von beliebigen $l \leq \min\{n - 1, k - 1\}$ Knoten immer noch zusammenhängend ist.*

h) *Ein* ***Zyklus*** *ist ein u-v-Weg der Länge $\ell \geq 2$ mit $u = v$.*

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

j) *Ein* ***Kreis*** *ist ein Zyklus $v_0, v_1 \ldots, v_{\ell-1}, v_0$ der Länge $\ell \geq 3$, für den $v_0, v_1, \ldots, v_{\ell-1}$ paarweise verschieden sind.*

k) *Ein Graph $G = (V, E)$ heißt* ***kreisfrei***, ***azyklisch*** *oder* ***Wald***, *falls er keinen Kreis enthält.*

l) *Ein* ***Baum*** *ist ein zusammenhängender Wald.*

m) *Jeder Knoten $u \in V$ vom Grad $\deg(u) \leq 1$ heißt* ***Blatt*** *und die übrigen Knoten (vom Grad ≥ 2) heißen* ***innere Knoten***.

Es ist leicht zu sehen, dass die Relation

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

eine Äquivalenzrelation ist. Die durch die Äquivalenzklassen von Z induzierten Teilgraphen heißen die ***Zusammenhangskomponenten*** (engl. *connected components*) von G.

Definition 1.7. *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 $\deg^+(v) = \|N^+(v)\|$ und der* ***Eingangsgrad*** *von v ist $\deg^-(v) = \|N^-(v)\|$. Der* ***Grad*** *von v ist $\deg(v) = \deg^+(v) + \deg^-(v)$.*

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 $i = 0, \ldots, \ell - 1$, der jede Kante $e \in E$ höchstens einmal durchläuft.*

f) *Ein* ***(gerichteter) Zyklus*** *ist ein gerichteter u-v-Weg der Länge $\ell \geq 1$ 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 $v_0, v_1 \ldots, v_{\ell-1}, v_0$ der Länge $\ell \geq 1$, für den $v_0, v_1, \ldots, v_{\ell-1}$ paarweise verschieden sind.*

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

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

k) *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.*

1.2 Datenstrukturen für Graphen

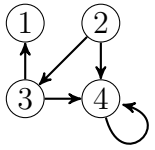
Sei $G = (V, E)$ ein Graph bzw. Digraph und sei $V = \{v_1, \ldots, v_n\}$. Dann ist die $(n \times n)$-Matrix $A = (a_{ij})$ mit den Einträgen

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

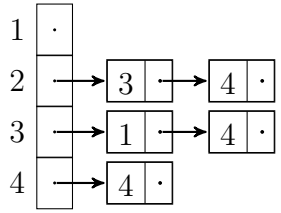
die ***Adjazenzmatrix*** von G. Für ungerichtete Graphen ist die Adjazenzmatrix symmetrisch mit $a_{ii} = 0$ für $i = 1, \ldots, n$.

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 gleichbleibt, 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 1.8.
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



◁

Folgende Tabelle gibt den Aufwand der wichtigsten elementaren Operationen auf Graphen in Abhängigkeit von der benutzten Datenstruktur an. Hierbei nehmen wir an, dass sich die Knotenmenge V nicht ändert.

	Adjazenzmatrix		Adjazenzlisten	
	einfach	clever	einfach	clever
Speicherbedarf	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n+m)$	$\mathcal{O}(n+m)$
Initialisieren	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Kante einfügen	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Kante entfernen	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Test auf Kante	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$

Bemerkung 1.9.

- *Der Aufwand für die Initialisierung des leeren Graphen in der Adjazenzmatrixdarstellung lässt sich auf $\mathcal{O}(1)$ drücken, indem man mithilfe eines zusätzlichen Feldes B die Gültigkeit der Matrixeinträge verwaltet (siehe Übungen).*
- *Die Verbesserung beim Löschen einer Kante in der Adjazenzlistendarstellung erhält man, indem man die Adjazenzlisten doppelt verkettet und im ungerichteten Fall die beiden Vorkommen jeder Kante in den Adjazenzlisten der beiden Endknoten gegenseitig verlinkt (siehe die Prozeduren* `Insert(Di)Edge` *und* `Remove(Di)Edge` *auf den nächsten Seiten).*
- *Bei der Adjazenzlistendarstellung können die Knoten auch in einer doppelt verketteten Liste organisiert werden. In diesem Fall können dann auch Knoten in konstanter Zeit hinzugefügt und in Zeit $\mathcal{O}(n)$ wieder entfernt werden (unter Beibehaltung der übrigen Speicher- und Laufzeitschranken).*

Es folgen die Prozeduren für die in obiger Tabelle aufgeführten elementaren Graphoperationen, falls G als ein Feld $G[1, \ldots, n]$ von (Zeigern auf) doppelt verkettete Adjazenzlisten repräsentiert wird. Wir behandeln zuerst den Fall eines Digraphen.

Prozedur `Init`

```
1 for i := 1 to n do
2   G[i] := ⊥
```

Prozedur `InsertDiEdge`(u, v)

```
1 erzeuge Listeneintrag e
2 source(e) := u
3 target(e) := v
4 prev(e) := ⊥
5 next(e) := G[u]
6 if G[u] ≠ ⊥ then
7   prev(G[u]) := e
8 G[u] := e
9 return e
```

Prozedur `RemoveDiEdge`(e)

```
1 if next(e) ≠ ⊥ then
2   prev(next(e)) := prev(e)
3 if prev(e) ≠ ⊥ then
4   next(prev(e)) := next(e)
5 else
6   G[source(e)] := next(e)
```

Prozedur `Edge`(u, v)

```
1 e := G[u]
2 while e ≠ ⊥ do
3   if target(e) = v then
4     return 1
5   e := next(e)
6 return 0
```


Falls G ungerichtet ist, können diese Operationen wie folgt implementiert werden (die Prozeduren `Init` und `Edge` bleiben unverändert).

Prozedur `InsertEdge`(u, v)

```
1  erzeuge Listeneinträge e, e'
2  opposite(e) := e'
3  opposite(e') := e
4  next(e) := G[u]
5  next(e') := G[v]
6  if G[u] ≠ ⊥ then
7    prev(G[u]) := e
8  if G[v] ≠ ⊥ then
9    prev(G[v]) := e'
10 G[u] := e; G[v] := e'
11 source(e) := target(e') := u
12 target(e) := source(e') := v
13 prev(e) := ⊥
14 prev(e') := ⊥
15 return e
```

Prozedur `RemoveEdge`(e)

```
1 RemoveDiEdge(e)
2 RemoveDiEdge(opposite(e))
```

1.3 Keller und Warteschlange

Für das Durchsuchen eines Graphen ist es vorteilhaft, die bereits besuchten (aber noch nicht abgearbeiteten) Knoten in einer Menge B zu speichern. Damit die Suche effizient ist, sollte die Datenstruktur für B folgende Operationen effizient implementieren.

`Init`(B):	Initialisiert B als leere Menge.
`Empty`(B):	Testet B auf Leerheit.
`Insert`(B, u):	Fügt u in B ein.
`Element`(B):	Gibt ein Element aus B zurück.
`Remove`(B):	Gibt ebenfalls `Element`(B) zurück und entfernt es aus B.

Andere Operationen wie z.B. `Remove`(B, u) werden nicht benötigt.

Die gewünschten Operationen lassen sich leicht durch einen ***Keller*** (auch ***Stapel*** genannt) (engl. *stack*) oder eine ***Warteschlange*** (engl. *queue*) implementieren. Falls maximal n Datensätze gespeichert werden müssen, kann ein Feld zur Speicherung der Elemente benutzt werden. Andernfalls können sie auch in einer einfach verketteten Liste gespeichert werden.

Stack S – Last-In-First-Out

`Top`(S):	Gibt das oberste Element von S zurück.
`Push`(S, x):	Fügt x als oberstes Element zum Keller hinzu.
`Pop`(S):	Gibt das oberste Element von S zurück und entfernt es.

Queue Q – Last-In-Last-Out

`Enqueue`(Q, x):	Fügt x am Ende der Schlange hinzu.
`Head`(Q):	Gibt das erste Element von Q zurück.
`Dequeue`(Q):	Gibt das erste Element von Q zurück und entfernt es.

Die Kelleroperationen lassen sich wie folgt auf einem Feld $S[1 \ldots n]$ implementieren. Die Variable `size`(S) enthält die Anzahl der im Keller gespeicherten Elemente.

Prozedur `StackInit`(S)

```
1  size(S) := 0
```

Prozedur `StackEmpty`(S)

```
1  return(size(S) = 0)
```

Prozedur `Top`(S)

```
1  if size(S) > 0 then
2    return(S[size(S)])
3  else
4    return(⊥)
```

Prozedur `Push`(S, x)

```
1  if size(S) < n then
2    size(S) := size(S) + 1
3    S[size(S)] := x
4  else
5    return(⊥)
```

Prozedur `Pop`(S)

```
1  if size(S) > 0 then
2    size(S) := size(S) − 1
3    return(S[size(S) + 1])
4  else
5    return(⊥)
```

Es folgen die Warteschlangenoperationen für die Speicherung in einem Feld $Q[1 \ldots n]$. Die Elemente werden der Reihe nach am Ende der Schlange Q (zyklisch) eingefügt und am Anfang entnommen. Die Variable `head`(Q) enthält den Index des ersten Elements der Schlange und `tail`(Q) den Index des hinter dem letzten Element von Q befindlichen Eintrags.

Prozedur `QueueInit`(Q)

```
1 head(Q) := 1
2 tail(Q) := 1
3 size(Q) := 0
```

Prozedur `QueueEmpty`(Q)

```
1 return(size(Q) = 0)
```

Prozedur `Head`(Q)

```
1 if QueueEmpty(Q) then
2   return(⊥)
3 else
4   returnQ[head(Q)]
```

Prozedur `Enqueue`(Q, x)

```
1 if size(Q) = n then
2   return(⊥)
3 size(Q) := size(Q) + 1
4 Q[tail(Q)] := x
5 if tail(Q) = n then
6   tail(Q) := 1
7 else
8   tail(Q) := tail(Q) + 1
```

Prozedur `Dequeue`(Q)

```
1 if QueueEmpty(Q) then
2   return(⊥)
3 size(Q) := size(Q) − 1
4 x := Q[head(Q)]
5 if head(Q) = n then
6   head(Q) := 1
7 else
8   head(Q) := head(Q) + 1
9 return(x)
```

Satz 1.10. *Sämtliche Operationen für einen Keller S und eine Warteschlange Q sind in konstanter Zeit $\mathcal{O}(1)$ ausführbar.*

Bemerkung 1.11. *Mit Hilfe von einfach verketteten Listen sind Keller und Warteschlangen auch für eine unbeschränkte Anzahl von Datensätzen mit denselben Laufzeitbeschränkungen implementierbar.*

Die für das Durchsuchen von Graphen benötigte Datenstruktur B lässt sich nun mittels Keller bzw. Schlange wie folgt realisieren.

Operation	Keller S	Schlange Q
`Init`(B)	`StackInit`(S)	`QueueInit`(Q)
`Empty`(B)	`StackEmpty`(S)	`QueueEmpty`(Q)
`Insert`(B, u)	`Push`(S, u)	`Enqueue`(Q, u)
`Element`(B)	`Top`(S)	`Head`(Q)
`Remove`(B)	`Pop`(S)	`Dequeue`(Q)

1.4 Durchsuchen von Graphen

Wir geben nun für die Suche in einem Graphen bzw. Digraphen $G = (V, E)$ einen Algorithmus `GraphSearch` mit folgenden Eigenschaften an:

`GraphSearch` benutzt eine Prozedur `Explore`, um alle Knoten und Kanten von G zu besuchen.

`Explore`(w) findet Pfade zu allen von w aus erreichbaren Knoten.

Hierzu speichert `Explore`(w) für jeden über eine Kante $\{u, v\}$ bzw. (u, v) neu entdeckten Knoten $v \neq w$ den Knoten u in `parent`(v). Wir nennen die bei der Entdeckung eines neuen Knotens v durchlaufenen Kanten $(\texttt{parent}(v), v)$ **parent-*Kanten***.

Im Folgenden verwenden wir die Schreibweise $e = uv$ sowohl für gerichtete als auch für ungerichtete Kanten $e = (u, v)$ bzw. $e = \{u, v\}$.

Algorithmus `GraphSearch`(V, E)

```
1  for all v ∈ V, e ∈ E do
2    vis(v) := false
3    parent(v) := ⊥
4    vis(e) := false
5  for all w ∈ V do
6    if vis(w) = false then Explore(w)
```

Prozedur `Explore`(w)

```
1  vis(w) := true
2  Init(B)
3  Insert(B, w)
4  while ¬Empty(B) do
5    u := Element(B)
6    if ∃ e = uv ∈ E : vis(e) = false then
7      vis(e) := true
8      if vis(v) = false then
9        vis(v) := true
10       parent(v) := u
11       Insert(B, v)
12   else
13     Remove(B)
```

Um die nächste von u ausgehende Kante uv, die noch nicht besucht wurde, in konstanter Zeit bestimmen zu können, kann man bei der Adjazenzlistendarstellung für jeden Knoten u neben dem Zeiger auf die erste Kante in der Adjazenzliste von u einen zweiten Zeiger bereithalten, der auf die aktuelle Kante in der Liste verweist.

Suchwälder

Definition 1.12. *Sei $G = (V, E)$ ein Digraph.*

- *Ein Knoten $w \in V$ heißt **Wurzel** von G, falls alle Knoten $v \in V$ von w aus erreichbar sind (d.h. es gibt einen gerichteten w-v-Weg in G).*
- *G heißt **gerichteter Wald**, wenn G kreisfrei ist und jeder Knoten $v \in V$ Eingangsgrad $\deg^-(v) \leq 1$ hat.*
- *Ein Knoten $u \in V$ vom Ausgangsgrad $\deg^+(u) = 0$ heißt **Blatt**.*
- *Ein **gerichteter Wald**, der eine Wurzel hat, heißt **gerichteter Baum**.*

In einem gerichteten Baum liegen die Kantenrichtungen durch die Wahl der Wurzel bereits eindeutig fest. Daher kann bei bekannter Wurzel auf die Angabe der Kantenrichtungen auch verzichtet werden. Man spricht dann von einem ***Wurzelbaum***.

Betrachte den durch `SearchGraph`(V, E) erzeugten Digraphen $W = (V, E_{\texttt{parent}})$ mit

$$E_{\texttt{parent}} = \left\{(\texttt{parent}(v), v) \mid v \in V \text{ und } \texttt{parent}(v) \neq \bot\right\}.$$

Da `parent`(v) vor v markiert wird, ist klar, dass W kreisfrei ist. Zudem hat jeder Knoten v höchstens einen Vorgänger `parent`(v). Dies zeigt, dass W tatsächlich ein gerichteter Wald ist. W heißt ***Suchwald*** von G und die Kanten $(\texttt{parent}(v), v)$ von W werden auch als ***Baumkanten*** bezeichnet.

W hängt zum einen davon ab, wie die Datenstruktur B implementiert ist (z.B. als Keller oder als Warteschlange). Zum anderen hängt W aber auch von der Reihenfolge der Knoten in den Adjazenzlisten ab.

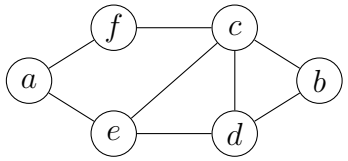
Klassifikation der Kanten eines (Di-)Graphen

Die Kanten eines Graphen $G = (V, E)$ werden durch den Suchwald $W = (V, E_{\texttt{parent}})$ in vier Klassen eingeteilt. Dabei erhält jede Kante

die Richtung, in der sie bei ihrem ersten Besuch durchlaufen wird.

Neben den Baumkanten $(\texttt{parent}(v), v) \in E_{\texttt{parent}}$ gibt es noch Rückwärts-, Vorwärts- und Querkanten. ***Rückwärtskanten*** (u, v) verbinden einen Knoten u mit einem Knoten v, der auf dem `parent`-Pfad $P(u)$ von u liegt. Liegt dagegen u auf $P(v)$, so wird (u, v) als ***Vorwärtskante*** bezeichnet. Alle übrigen Kanten heißen ***Querkanten***. Diese verbinden zwei Knoten, von denen keiner auf dem `parent`-Pfad des anderen liegt.

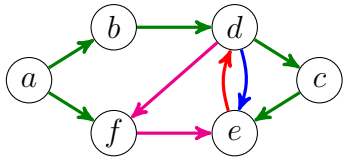
Beispiel 1.13. *Bei Aufruf mit dem Startknoten a könnte die Prozedur* `Explore` *den nebenstehendem Graphen beispielsweise wie folgt durchsuchen.*



Menge B	Knoten	Kante	Typ
$\{a\}$	a	(a,b)	B
$\{a,b\}$	a	(a,f)	B
$\{a,b,f\}$	a	-	-
$\{b,f\}$	b	(b,d)	B
$\{b,d,f\}$	b	-	-
$\{d,f\}$	d	(d,c)	B
$\{c,d,f\}$	c	(c,e)	B
$\{c,d,e,f\}$	c	-	-

B	Knoten	Kante	Typ
$\{d,e,f\}$	d	(d,e)	V
$\{d,e,f\}$	d	(d,f)	Q
$\{d,e,f\}$	d	-	-
$\{e,f\}$	e	(e,d)	R
$\{e,f\}$	e	-	-
$\{f\}$	f	(f,e)	Q
$\{f\}$	f	-	-
$\emptyset$			

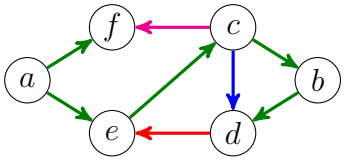
Dabei entsteht nebenstehender Suchwald.



Die Klassifikation der Kanten eines Digraphen G erfolgt analog, wobei die Richtungen jedoch bereits durch G vorgegeben sind (dabei werden Schlingen der Kategorie der Vorwärtskanten zugeordnet). Tatsächlich

durchläuft `Explore` bei einem Graphen die Knoten und Kanten in der gleichen Reihenfolge wie bei dem Digraphen, der für jede ungerichtete Kante $\{u, v\}$ die beiden gerichteten Kanten (u, v) und (v, u) enthält.

Beispiel 1.14. *Bei Aufruf mit dem Startknoten a könnte die Prozedur* `Explore` *beispielsweise nebenstehenden Suchwald generieren.*



Menge B	Knoten	Kante	
$\{a\}$	a	$\{a,e\}$	B
$\{a,e\}$	a	$\{a,f\}$	B
$\{a,e,f\}$	a	-	-
$\{e,f\}$	e	$\{e,a\}$	-
$\{e,f\}$	e	$\{e,c\}$	B
$\{c,e,f\}$	c	$\{c,b\}$	B
$\{b,c,e,f\}$	b	$\{b,c\}$	-
$\{b,c,e,f\}$	b	$\{b,d\}$	B
$\{b,c,d,e,f\}$	b	-	-
$\{c,d,e,f\}$	c	$\{c,d\}$	V
$\{c,d,e,f\}$	c	$\{c,e\}$	-

B	Knoten	Kante	
$\{c,d,e,f\}$	c	$\{c,f\}$	Q
$\{c,d,e,f\}$	c	-	-
$\{d,e,f\}$	d	$\{d,b\}$	-
$\{d,e,f\}$	d	$\{d,c\}$	-
$\{d,e,f\}$	d	$\{d,e\}$	R
$\{d,e,f\}$	d	-	-
$\{e,f\}$	e	$\{e,d\}$	-
$\{e,f\}$	e	-	-
$\{f\}$	f	$\{f,a\}$	-
$\{f\}$	f	$\{f,c\}$	-
$\{f\}$	f	-	-

Satz 1.15. *Falls der (un)gerichtete Graph G in Adjazenzlisten-Darstellung gegeben ist, durchläuft* `GraphSearch` *alle Knoten und Kanten von G in Zeit $\mathcal{O}(n+m)$.*

Beweis. Offensichtlich wird jeder Knoten u genau einmal zu B hinzugefügt. Dies geschieht zu dem Zeitpunkt, wenn u zum ersten Mal „besucht“ und das Feld `visited` für u auf `true` gesetzt wird. Außerdem werden in Zeile 6 von `Explore` alle von u ausgehenden Kanten durchlaufen, bevor u wieder aus B entfernt wird. Folglich werden tatsächlich alle Knoten und Kanten von G besucht.

Wir bestimmen nun die Laufzeit des Algorithmus `GraphSearch`. Innerhalb von `Explore` wird die while-Schleife für jeden Knoten u genau $(\deg(u)+1)$-mal bzw. $(\deg^+(u)+1)$-mal durchlaufen:

- einmal für jeden Nachbarn v von u und
- dann noch einmal, um u aus B zu entfernen.

Insgesamt sind das $n+2m$ im ungerichteten bzw. $n+m$ Durchläufe im gerichteten Fall. Bei Verwendung von Adjazenzlisten kann die nächste von einem Knoten v aus noch nicht besuchte Kante e in konstanter Zeit ermittelt werden, falls man für jeden Knoten v einen Zeiger auf e in der Adjazenzliste von v vorsieht. Die Gesamtlaufzeit des Algorithmus `GraphSearch` beträgt somit $\mathcal{O}(n+m)$. ■

Als nächstes zeigen wir, dass `Explore`(w) zu allen von w aus erreichbaren Knoten v einen (gerichteten) w-v-Pfad liefert. Dieser lässt sich mittels `parent` wie folgt zurückverfolgen. Sei

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

und sei $\ell = \min\{i \geq 0 \mid u_{i+1} = \bot\}$. Dann ist $u_\ell = w$ und $p = (u_\ell, \ldots, u_0)$ ein w-v-Pfad. Wir nennen P den **`parent`-*Pfad*** von v und bezeichnen ihn mit $\boldsymbol{P(v)}$.

Satz 1.16. *Falls beim Aufruf von* `Explore` *alle Knoten und Kanten als unbesucht markiert sind, berechnet* `Explore`(w) *zu allen erreichbaren Knoten v einen (gerichteten) w-v-Pfad $P(v)$.*

Beweis. Wir zeigen zuerst, dass `Explore`(w) alle von w aus erreichbaren Knoten besucht. Hierzu führen wir Induktion über die Länge ℓ eines kürzesten w-v-Weges.

$\ell = 0$**:** In diesem Fall ist $v = w$ und w wird in Zeile 1 besucht.

$\ell \rightsquigarrow \ell + 1$**:** Sei v ein Knoten mit Abstand $l+1$ von w. Dann hat ein Nachbarknoten $u \in N(v)$ den Abstand ℓ von w. Folglich wird u nach IV besucht. Da u erst dann aus B entfernt wird, wenn alle seine Nachbarn (bzw. Nachfolger) besucht wurden, wird auch v besucht.

Es bleibt zu zeigen, dass `parent` einen Pfad $P(v)$ von w zu jedem besuchten Knoten v liefert. Hierzu führen wir Induktion über die Anzahl k der vor v besuchten Knoten.

$k = 0$**:** In diesem Fall ist $v = w$. Da `parent`$(w) = \bot$ ist, liefert `parent` einen w-v-Pfad (der Länge 0).

$k - 1 \rightsquigarrow k$**:** Sei $u =$ `parent`(v). Da u vor v besucht wird, liefert `parent` nach IV einen w-u-Pfad $P(u)$. Wegen $u =$ `parent`(v) ist u der Entdecker von v und daher mit v durch eine Kante verbunden. Somit liefert `parent` auch für v einen w-v-Pfad $P(v)$. ■

1.5 Spannbäume und Spannwälder

In diesem Abschnitt zeigen wir, dass der Algorithmus `GraphSearch` für jede Zusammenhangskomponente eines (ungerichteten) Graphen G einen Spannbaum berechnet.

Definition 1.17. *Sei $G = (V, E)$ ein Graph und $H = (U, F)$ ein Untergraph.*

- *H heißt* ***spannend****, falls $U = V$ ist.*
- *H ist ein* ***spannender Baum*** *(oder* ***Spannbaum****) von G, falls $U = V$ und H ein Baum ist.*
- *H ist ein* ***spannender Wald*** *(oder* ***Spannwald****) von G, falls $U = V$ und H ein Wald ist.*

Es ist leicht zu sehen, dass für G genau dann ein Spannbaum existiert, wenn G zusammenhängend ist. Allgemeiner gilt, dass die Spannbäume für die Zusammenhangskomponenten von G einen Spannwald bilden. Dieser ist bzgl. der Subgraph-Relation maximal, da er in keinem größeren Spannwald enthalten ist. Ignorieren wir die Richtungen der Kanten im Suchwald W, so ist der resultierende Wald W' ein maximaler Spannwald für G.

Da $\texttt{Explore}(w)$ alle von w aus erreichbaren Knoten findet, spannt jeder Baum des (ungerichteten) Suchwaldes $W' = (V, E'_{\texttt{parent}})$ mit

$$E'_{\texttt{parent}} = \Big\{\{\texttt{parent}(v), v\} \mid v \in V \text{ und } \texttt{parent}(v) \neq \perp\Big\}$$

eine Zusammenhangskomponente von G.

Korollar 1.18. *Sei G ein (ungerichteter) Graph.*

- *Der Algorithmus* $\texttt{GraphSearch}(V, E)$ *berechnet in Linearzeit einen Spannwald W', dessen Bäume die Zusammenhangskomponenten von G spannen.*
- *Falls G zusammenhängend ist, ist W' ein Spannbaum für G.*

1.6 Berechnung der Zusammenhangskomponenten

Folgende Variante von `GraphSearch` bestimmt die Zusammenhangskomponenten eines (ungerichteten) Eingabegraphen G.

Algorithmus $\texttt{CC}(V, E)$

```
1  k := 0
2  for all v ∈ V, e ∈ E do
3     cc(v) := 0
4     cc(e) := 0
5  for all w ∈ V do
6     if cc(w) = 0 then
7        k := k + 1
8        ComputeCC(k, w)
```

Prozedur $\texttt{ComputeCC}(k, w)$

```
1  cc(w) := k
2  Init(B)
3  Insert(B, w)
4  while ¬Empty(B) do
5     u := Element(B)
6     if ∃ e = {u, v} ∈ E : cc(e) = 0 then
7        cc(e) := k
8        if cc(v) = 0 then
9           cc(v) := k
10          Insert(B, v)
11    else
12       Remove(B)
```

Korollar 1.19. *Der Algorithmus* $\texttt{CC}(V, E)$ *bestimmt für einen Graphen $G = (V, E)$ in Linearzeit $\mathcal{O}(n + m)$ sämtliche Zusammenhangskomponenten $G_k = (V_k, E_k)$ von G, wobei $V_k = \{v \in V \mid \texttt{cc}(v) = k\}$ und $E_k = \{e \in E \mid \texttt{cc}(e) = k\}$ ist.*

1.7 Breiten- und Tiefensuche

Wie wir gesehen haben, findet $\texttt{Explore}(w)$ sowohl in Graphen als auch in Digraphen alle von w aus erreichbaren Knoten. Als nächstes zeigen wir, dass $\texttt{Explore}(w)$ zu allen von w aus erreichbaren Knoten sogar einen kürzesten Weg findet, falls wir die Datenstruktur B als Warteschlange Q implementieren.

Die Benutzung einer Warteschlange Q zur Speicherung der bereits entdeckten, aber noch nicht abgearbeiteten Knoten bewirkt, dass

zuerst alle Nachbarknoten $u_1, \ldots, u_k$ des aktuellen Knotens u besucht werden, bevor ein anderer Knoten aktueller Knoten wird. Da die Suche also zuerst in die Breite geht, spricht man von einer ***Breitensuche*** (kurz *BFS*, engl. *breadth first search*). Den hierbei berechneten Suchwald bezeichnen wir als ***Breitensuchwald***.

Bei Benutzung eines Kellers wird dagegen u_1 aktueller Knoten, bevor die übrigen Nachbarknoten von u besucht werden. Daher führt die Benutzung eines Kellers zu einer ***Tiefensuche*** (kurz *DFS*, engl. *depth first search*). Der berechnete Suchwald heißt dann ***Tiefensuchwald***.

Die Breitensuche eignet sich eher für Distanzprobleme wie z.B. das Finden

- kürzester Wege in Graphen und Digraphen,
- längster Wege in Bäumen (siehe Übungen) oder
- kürzester Wege in Distanzgraphen (Dijkstra-Algorithmus).

Dagegen liefert die Tiefensuche interessante Strukturinformationen wie z.B.

- die zweifachen Zusammenhangskomponenten in Graphen,
- die starken Zusammenhangskomponenten in Digraphen oder
- eine topologische Sortierung bei azyklischen Digraphen (s. Übungen).

Wir betrachten zuerst den Breitensuchalgorithmus.

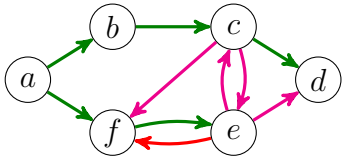
Algorithmus `BFS`(V, E)

```
1  for all v ∈ V, e ∈ E do
2     vis(v) := false
3     parent(v) := ⊥
4     vis(e) := false
5  for all w ∈ V do
6     if vis(w) = false then BFS-Explore(w)
```

Prozedur `BFS-Explore`(w)

```
1  vis(w) := true
2  QueueInit(Q)
3  Enqueue(Q, w)
4  while ¬QueueEmpty(Q) do
5     u := Head(Q)
6     if ∃ e = uv ∈ E : vis(e) = false then
7        vis(e) := true
8        if vis(v) = false then
9           vis(v) := true
10          parent(v) := u
11          Enqueue(Q, v)
12    else
13       Dequeue(Q)
```

Beispiel 1.20. `BFS-Explore` *generiert bei Aufruf mit dem Startknoten a nebenstehenden Breitensuchwald.*



Schlange Q	bes. Knoten	bes. Kante	Typ
$\leftarrow a \leftarrow$	a	(a,b)	B
a, b	a	(a,f)	B
a, b, f	a	-	-
b, f	b	(b,c)	B
b, f, c	b	-	-
f, c	f	(f,e)	B
f, c, e	f	-	-
c, e	c	(c,d)	B

Q	bes. Knoten	bes. Kante	Typ
c, e, d	c	(c,e)	Q
c, e, d	c	(c,f)	Q
c, e, d	c	-	-
e, d	e	(e,c)	Q
e, d	e	(e,d)	Q
e, d	e	(e,f)	R
e, d	e	-	-
d	d	-	-

◁

Satz 1.21. *Sei G ein Graph oder Digraph und sei w Wurzel des von* `BFS-Explore`(w) *berechneten Suchbaumes T. Dann liefert* `parent` *für jeden Knoten v in T einen kürzesten w-v-Weg $P(v)$.*

Beweis. Wir führen Induktion über die kürzeste Weglänge ℓ von w nach v in G.

$\ell = 0$**:** Dann ist $v = w$ und `parent` liefert einen Weg der Länge 0.

$\ell \rightsquigarrow \ell + 1$**:** Sei v ein Knoten, der den Abstand $\ell + 1$ von w in G hat. Dann existiert ein Knoten $u \in N^-(v)$ (bzw. $u \in N(v)$) mit Abstand ℓ von w in G hat. Nach IV liefert also `parent` einen w-u-Weg $P(u)$ der Länge ℓ. Da u erst aus Q entfernt wird, nachdem alle Nachfolger von u entdeckt sind, wird v von u oder einem bereits zuvor in Q eingefügten Knoten z entdeckt. Da Q als Schlange organisiert ist, ist $P(u)$ nicht kürzer als $P(z)$. Daher folgt in beiden Fällen, dass $P(v)$ die Länge $\ell + 1$ hat. ■

Wir werden später noch eine Modifikation der Breitensuche kennen lernen, die kürzeste Wege in Graphen mit nichtnegativen Kantenlängen findet (Algorithmus von Dijkstra).

Als nächstes betrachten wir den Tiefensuchalgorithmus.

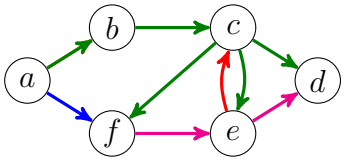
Algorithmus `DFS`(V, E)

```
1  for all v ∈ V, e ∈ E do
2     vis(v) := false
3     parent(v) := ⊥
4     vis(e) := false
5  for all w ∈ V do
6     if vis(w) = false then DFS-Explore(w)
```

Prozedur `DFS-Explore`(w)

```
1  vis(w) := true
2  StackInit(S)
3  Push(S, w)
4  while ¬StackEmpty(S) do
5     u := Top(S)
6     if ∃ e = uv ∈ E : vis(e) = false then
7        vis(e) := true
8        if vis(v) = false then
9           vis(v) := true
10          parent(v) := u
11          Push(S, v)
12    else
13       Pop(S)
```

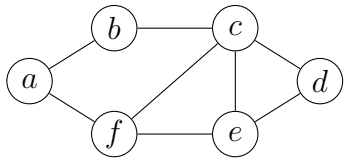
Beispiel 1.22. *Bei Aufruf mit dem Startknoten a generiert die Prozedur* `DFS-Explore` *nebenstehenden Tiefensuchwald.*

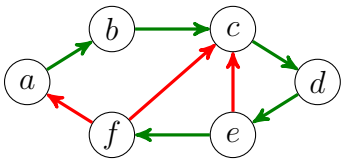


Keller S	bes. Knoten	bes. Kante	Typ
$a \leftrightarrow$	a	(a,b)	B
a, b	b	(b,c)	B
a, b, c	c	(c,d)	B
a, b, c, d	d	-	-
a, b, c	c	(c,e)	B
a, b, c, e	e	(e,c)	R
a, b, c, e	e	(e,d)	Q
a, b, c, e	e	-	-

S	bes. Knoten	bes. Kante	Typ
a, b, c	c	(c,f)	B
a, b, c, f	f	(f,e)	Q
a, b, c, f	f	-	-
a, b, c	c	-	-
a, b	b	-	-
a	a	(a,f)	V
a	a	-	-

Die Tiefensuche auf nebenstehendem Graphen führt auf folgende Klassifikation der Kanten (wobei wir annehmen,





dass die Nachbarknoten in den Adjazenzlisten alphabetisch angeordnet sind):

Keller S	Kante	Typ
$a \leftrightarrow$	$\{a,b\}$	B
a,b	$\{b,a\}$	-
a,b	$\{b,c\}$	B
a,b,c	$\{c,b\}$	-
a,b,c	$\{c,d\}$	B
a,b,c,d	$\{d,c\}$	-
a,b,c,d	$\{d,e\}$	B
a,b,c,d,e	$\{e,c\}$	R
a,b,c,d,e	$\{e,d\}$	-
a,b,c,d,e	$\{e,f\}$	B
a,b,c,d,e,f	$\{f,a\}$	R

Keller S	Kante	Typ
a,b,c,d,e,f	$\{f,c\}$	R
a,b,c,d,e,f	$\{f,e\}$	-
a,b,c,d,e,f	-	-
a,b,c,d,e	-	-
a,b,c,d	-	-
a,b,c	$\{c,e\}$	-
a,b,c	$\{c,f\}$	-
a,b,c	-	-
a,b	-	-
a	$\{a,f\}$	-
a	-	-

◁

Die Tiefensuche lässt sich auch rekursiv implementieren. Dies hat den Vorteil, dass kein (expliziter) Keller benötigt wird.

Prozedur `DFS-Explore-rec`(w)

```
1  vis(w) := true
2  while ∃ e = uv ∈ E : vis(e) = false do
3    vis(e) := true
4    if vis(v) = false then
5      parent(v) := w
6      DFS-Explore-rec(v)
```

Da `DFS-Explore-rec`(w) zu `parent`(w) zurückspringt, kann auch das Feld `parent`(w) als Keller fungieren. Daher lässt sich die Prozedur auch nicht-rekursiv ohne zusätzlichen Keller implementieren, indem die Rücksprünge explizit innerhalb einer Schleife ausgeführt werden (siehe Übungen).

Bei der Tiefensuche lässt sich der Typ jeder Kante algorithmisch leicht bestimmen, wenn wir noch folgende Zusatzinformationen speichern.

- Ein neu entdeckter Knoten wird bei seinem ersten Besuch grau gefärbt. Sobald er abgearbeitet ist, also bei seinem letzten Besuch, wird er schwarz. Zu Beginn sind alle Knoten weiß.
- Zudem merken wir uns die Reihenfolge, in der die Knoten entdeckt werden, in einem Feld `r`.

Dann lässt sich der Typ jeder Kante $e = (u,v)$ bei ihrem ersten Besuch wie folgt bestimmen:

Baumkante: `farbe`(v) = weiß,
Vorwärtskante: `farbe`$(v) \neq$ weiß und `r`$(v) \geq$ `r`(u),
Rückwärtskante: `farbe`(v) = grau und `r`$(v) <$ `r`(u),
Querkante: `farbe`(v) = schwarz und `r`$(v) <$ `r`(u).

Die folgende Variante von `DFS` berechnet diese Informationen.

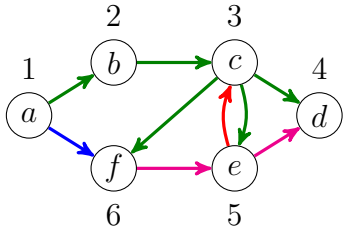
Algorithmus `DFS`(V,E)

```
1  r := 0
2  for all v ∈ V,  e ∈ E  do
3    farbe(v) := weiß
4    vis(e) := false
5  for all u ∈ V  do
6    if farbe(u) = weiß then DFS-Explore(u)
```

Prozedur `DFS-Explore`(u)

```
1  farbe(u) := grau
2  r := r + 1
3  r(u) := r
4  while ∃ e = (u, v) ∈ E : vis(e) = false do
5    vis(e) := true
6    if farbe(v) = weiß then
7      DFS-Explore(v)
8  farbe(u) := schwarz
```

Beispiel 1.23. *Bei Aufruf mit dem Startknoten a werden die Knoten im nebenstehenden Digraphen von der Prozedur* `DFS-Explore` *wie folgt gefärbt (die Knoten sind mit ihren* `r`*-Werten markiert).*



Keller	Farbe	Kante	Typ
a	a: grau	(a,b)	B
a,b	b: grau	(b,c)	B
a,b,c	c: grau	(c,d)	B
a,b,c,d	d: grau	-	-
	d: schwarz		
a,b,c	-	(c,e)	B
a,b,c,e	e: grau	(e,c)	R
a,b,c,e	-	(e,d)	Q

Keller	Farbe	Kante	Typ
a,b,c,e	e: schwarz	-	-
a,b,c	-	(c,f)	B
a,b,c,f	f: grau	(f,e)	Q
a,b,c,f	f: schwarz	-	-
a,b,c	c: schwarz	-	-
a,b	b: schwarz	-	-
a	-	(a,f)	V
a	a: schwarz	-	-

◁

Bei der Tiefensuche in ungerichteten Graphen können weder Quer- noch Vorwärtskanten auftreten. Da v beim ersten Besuch einer solchen Kante (u,v) nicht weiß ist und alle grauen Knoten auf dem `parent`-Pfad $P(u)$ liegen, müsste v nämlich bereits schwarz sein. Dies ist aber nicht möglich, da die Kante $\{u,v\}$ in v-u-Richtung noch gar nicht durchlaufen wurde. Folglich sind alle Kanten, die nicht zu einem neuen Knoten führen, Rückwärtskanten. Das Fehlen von Quer- und Vorwärtskanten spielt bei manchen Anwendungen eine wichtige Rolle, etwa bei der Zerlegung eines Graphen G in seine ***zweifachen Zusammenhangskomponenten***.

2 Berechnung kürzester Wege

In vielen Anwendungen tritt das Problem auf, einen kürzesten Weg von einem Startknoten s zu einem Zielknoten t in einem Digraphen zu finden, dessen Kanten (u, v) vorgegebene ***Längen*** $l(u, v)$ haben. Die Länge eines Weges $W = (v_0, \ldots, v_\ell)$ ist

$$l(W) = \sum_{i=0}^{\ell-1} l(v_i, v_{i+1}).$$

Die kürzeste Weglänge von s nach t wird als ***Distanz*** $dist(s, t)$ zwischen s und t bezeichnet,

$$dist(s, t) = \min\{l(W) \mid W \text{ ist ein } s\text{-}t\text{-Weg}\}.$$

Falls kein s-t-Weg existiert, setzen wir $dist(s, t) = \infty$. Man beachte, dass die Distanz auch dann nicht beliebig klein werden kann, wenn Kreise mit negativer Länge existieren, da ein Weg jede Kante höchstens einmal durchlaufen kann. In vielen Fällen haben jedoch alle Kanten in E eine nichtnegative Länge $l(u, v) \geq 0$. In diesem Fall nennen wir $D = (V, E, l)$ einen ***Distanzgraphen***.

2.1 Der Dijkstra-Algorithmus

Der Dijkstra-Algorithmus findet einen kürzesten Weg $P(u)$ von s zu allen erreichbaren Knoten u *(single-source shortest-path problem)*. Hierzu führt der Algorithmus eine modifizierte Breitensuche aus. Dabei werden die in Bearbeitung befindlichen Knoten in einer Prioritätswarteschlange U verwaltet. Genauer werden alle Knoten u, zu denen bereits ein s-u-Weg $P(u)$ bekannt ist, zusammen mit der Weglänge g solange in U gespeichert bis $P(u)$ optimal ist. Auf der Datenstruktur U sollten folgende Operationen (möglichst effizient) ausführbar sein.

`Init`(U): Initialisiert U als leere Menge.

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

`RemoveMin`(U): Gibt ein Element aus U mit dem kleinsten Wert zurück und entfernt es aus U (ist U leer, wird der Wert $\perp$ (nil) zurückgegeben).

Voraussetzung für die Korrektheit des Algorithmus ist, dass alle Kanten eine nichtnegative Länge haben. Während der Suche werden bestimmte Kanten $e = (u, v)$ daraufhin getestet, ob $g(u) + \ell(u, v) < g(v)$ ist. Da in diesem Fall die Kante e auf eine Herabsetzung von $g(v)$ auf den Wert $g(u) + \ell(u, v)$ „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 unter allen noch nicht fertigen Knoten mit minimalem g-Wert und teste alle Kanten (u, v), für die v nicht schon fertig ist.

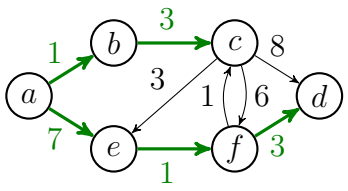
Algorithmus `Dijkstra`(V, E, l, s)

```
1  for all v ∈ V do
2     g(v) := ∞
3     parent(v) := ⊥
4     done(v) := false
5  g(s) := 0
6  Init(P)
7  Update(P, s, 0)
8  while u := RemoveMin(P) ≠ ⊥ do
9     done(u) := true
```

```
10    for all v ∈ N⁺(u) do
11      if done(v) = false ∧ g(u) + l(u,v) < g(v) then
12        g(v) := g(u) + l(u,v)
13        Update(P, v, g(v))
14        parent(v) := u
```

Der Algorithmus speichert die aktuelle Länge des Pfades $P(u)$ in $g(u)$. Knoten außerhalb des aktuellen Breitensuchbaums T haben den Wert $g(u) = \infty$. In jedem Schleifendurchlauf wird in Zeile 8 ein Knoten u mit minimalem g-Wert aus U entfernt und als fertig markiert. Anschließend werden alle von u wegführenden Kanten $e = (u, v)$ auf Relaxation getestet sowie g, U und T gegebenenfalls aktualisiert.

Beispiel 2.1. *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 2.2. *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 8 aus U entfernt wird, sämtliche Nachfolger von u zu U hinzugefügt werden.

Zudem ist klar, dass $g(u) \geq dist(s, u)$ ist, da $P(u)$ im Fall $g(u) < \infty$ ein s-u-Weg der Länge $g(u)$ ist. Es bleibt also nur noch zu zeigen, dass $P(u)$ für jeden aus U entfernten Knoten u ein kürzester s-u-Weg ist, d.h. es gilt $g(u) \leq dist(s, u)$.

Hierzu zeigen wir induktiv über die Anzahl k der vor u aus U entfernten Knoten, dass $g(u) \leq dist(s, u)$ ist.

$k = 0$**:** In diesem Fall ist $u = s$ und $P(u)$ hat die Länge $g(u) = 0$.

$k - 1 \rightsquigarrow k$**:** Sei $W = v_0, \ldots, v_\ell = u$ ein *kürzester* s-u-Weg in G und sei v_i der Knoten mit maximalem Index i auf diesem Weg, der vor u aus P entfernt wird.

Nach IV gilt dann

$$g(v_i) = dist(s, v_i). \tag{2.1}$$

Zudem ist

$$g(v_{i+1}) \leq g(v_i) + l(v_i, v_{i+1}). \tag{2.2}$$

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

$$g(u) \leq g(v_{i+1}). \tag{2.3}$$

Daher folgt

$$\begin{aligned} g(u) \overset{(2.3)}{\leq} g(v_{i+1}) &\overset{(2.2)}{\leq} g(v_i) + l(v_i, v_{i+1}) \\ &\overset{(2.1)}{=} dist(s, v_i) + l(v_i, v_{i+1}) \\ &= dist(s, v_{i+1}) \leq dist(s, u). \end{aligned}$$

■

Um die Laufzeit des Dijkstra-Algorithmus abzuschätzen, überlegen wir uns zuerst, wie oft die einzelnen Operationen auf der Datenstruktur P 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 `RemoveMin`-Operation höchstens $\min\{n, m\}$-mal ausgeführt.
- Wie die Prozedur `BFS-Explore` besucht der Dijkstra-Algorithmus jede Kante maximal einmal. Daher wird die `Update`-Operation höchstens m-mal ausgeführt.

Beobachtung 2.3. *Bezeichne $Init(n)$, $RemoveMin(n)$ und $Update(n)$ den Aufwand zum Ausführen der Operationen* `Init`, `RemoveMin` *und* `Update` *für den Fall, dass P nicht mehr als n Elemente aufzunehmen hat. Dann ist die Laufzeit des Dijkstra-Algorithmus durch*

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

beschränkt.

Die Laufzeit 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 (ersten) Besuchs. D.h. wir können U als Warteschlange implementieren. Dies führt wie bei der Prozedur `BFS-Explore` auf eine Laufzeit von $\mathcal{O}(n + m)$.

Für den allgemeinen Fall, dass die Kanten unterschiedliche Längen haben, betrachten wir folgende drei Möglichkeiten.

1. Da die Felder g und `done` bereits alle zur Verwaltung von U benötigten Informationen enthalten, kann man auf die (explizite) Implementierung von U auch verzichten. In diesem Fall kostet die `RemoveMin`-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 `RemoveMin` 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 `RemoveMin` immer noch in Zeit $\mathcal{O}(\log n^2) = \mathcal{O}(\log n)$ ausführbar. Insgesamt erhalten wir somit eine Laufzeit von $\mathcal{O}(n + m \log n)$, da sowohl `Insert` als auch `RemoveMin` maximal m-mal ausgeführt werden.

   Die Laufzeit von $\mathcal{O}(n+m \log n)$ bei Benutzung eines Heaps ist zwar für dünne Graphen sehr gut, aber für dichte Graphen schlechter als die implizite Implementierung von U mithilfe der Felder g 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 `RemoveMin`-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.

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

Die Tabelle fasst die Laufzeiten des Dijkstra-Algorithmus für die verschiedenen Möglichkeiten zur Implementation der Datenstruktur U zusammen. Eine offene Frage ist, ob es auch einen Algorithmus mit linearer Laufzeit $\mathcal{O}(n+m)$ zur Bestimmung kürzester Wege in Distanzgraphen gibt.

2.2 Der Bellman-Ford-Algorithmus

In manchen Anwendungen treten negative Kantengewichte auf. Geben die Kantengewichte beispielsweise die mit einer Kante verbundenen Kosten wider, so kann ein Gewinn durch negative Kosten modelliert werden. Auf diese Weise lassen sich auch längste Wege in Distanzgraphen berechnen, indem man alle Kantenlängen $l(u,v)$ mit -1 multipliziert und in dem resultierenden Graphen einen kürzesten Weg bestimmt.

Die Komplexität des Problems hängt wesentlich davon ab, ob man (gerichtete) Kreise mit negativer Länge zulässt oder nicht. Falls negative Kreise zugelassen werden, 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). Diese Algorithmen lösen das single-source shortest-path Problem mit einer Laufzeit von $\mathcal{O}(nm)$ im schlechtesten Fall.

Der Ford-Algorithmus arbeitet ganz ähnlich wie der Dijkstra-Algorithmus, betrachtet aber jede Kante nicht wie dieser nur einmal, sondern eventuell mehrmals. In seiner einfachsten Form sucht der Algorithmus wiederholt eine Kante $e=(u,v)$ mit

$$g(u)+\ell(u,v) < g(v)$$

und aktualisiert den Wert von $g(v)$ auf $g(u)+\ell(u,v)$ (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 der zugrunde liegende Digraph azyklisch ist). Bei der Bellman-Ford-Variante wird in $\mathcal{O}(nm)$ Schritten ein kürzester Weg von s zu allen erreichbaren Knoten gefunden (sofern keine negativen Kreise existieren).

Wir zeigen induktiv über die Anzahl k der Kanten eines kürzesten s-u-Weges, dass $g(u) = \mathit{dist}(s,u)$ gilt, falls g für alle Kanten (u,v) die Dreiecksungleichung $g(v) \le g(u)+\ell(u,v)$ erfüllt (also keine Relaxationen mehr möglich sind).

Im Fall $k=0$ ist nämlich $u=s$ und somit $g(s)=0=\mathit{dist}(s,s)$. Im Fall $k>0$ sei v ein Knoten, dessen kürzester s-v-Weg W aus k Kanten besteht. Dann gilt nach IV für den Vorgänger u von v auf W $g(u)=\mathit{dist}(s,u)$. Aufgrund der Dreiecksungleichung folgt dann

$$g(v) \le g(u)+\ell(u,v) = \mathit{dist}(s,u)+\ell(u,v) = \mathit{dist}(s,v).$$

Aus dem Beweis folgt zudem, dass nach Relaxation aller Kanten eines kürzesten s-v-Weges W (in der Reihenfolge, in der die Kanten in W durchlaufen werden) den Wert $\mathit{dist}(s,v)$ hat. Dies gilt auch für den Fall, dass zwischendurch noch weitere Kantenrelaxationen stattfinden.

Der Bellman-Ford-Algorithmus prüft in $n-1$ Iterationen jeweils alle Kanten auf Relaxation. Sind in der n-ten Runde noch weitere Relaxationen möglich, muss ein negativer Kreis existieren. Die Laufzeit ist offensichtlich $\mathcal{O}(nm)$ und die Korrektheit folgt leicht durch Induktion über die minimale Anzahl von Kanten eines kürzesten s-t-Weges. Zudem wird bei jeder Relaxation einer Kante (u,v) der Vorgänger u im Feld $\mathtt{parent}(v)$ vermerkt, so dass sich ein kürzester Weg von s zu allen erreichbaren Knoten (bzw. ein negativer Kreis) rekonstruieren lässt.

Algorithmus $\mathtt{BF}(V,E,l,s)$

```
1  for all v ∈ V do
2      g(v) := ∞
3      parent(v) := ⊥
4  g(s) := 0
5  for i := 1 to n − 1 do
```

```
6     for all (u,v) ∈ E do
7       if g(u) + l(u,v) < g(v) then
8         g(v) := g(u) + l(u,v)
9         parent(v) := u
10  for all (u,v) ∈ E do
11    if g(u) + l(u,v) < g(v) then
12      error(es gibt einen negativen Kreis)
```

2.3 Der Bellman-Ford-Moore-Algorithmus

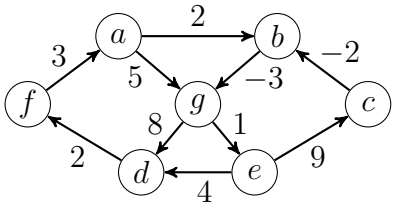
Die BFM-Variante prüft in jeder Runde nur diejenigen Kanten (u, v) auf Relaxation, für die $g(u)$ in der vorigen Runde erniedrigt wurde. Dies führt auf eine deutliche Verbesserung der durchschnittlichen Laufzeit. Wurde nämlich $g(u)$ in der $(i-1)$-ten Runde nicht verringert, dann steht in der i-ten Runde sicher keine Relaxation der Kante (u, v) an. Es liegt nahe, die in der nächsten Runde zu prüfenden Knoten u in einer Schlange Q zu speichern. Dabei kann mit u auch die aktuelle Rundenzahl i in Q gespeichert werden. In Runde 0 wird der Startknoten s in Q eingefügt. Können in Runde n immer noch Kanten relaxiert werden, so bricht der Algorithmus mit der Fehlermeldung ab, dass negative Kreise existieren. Da die BFM-Variante die Kanten in derselben Reihenfolge relaxiert wie der BF-Algorithmus, führt sie auf dasselbe Ergebnis.

Algorithmus `BFM`(V, E, l, s)

```
1   for all v ∈ V do
2     g(v) := ∞, parent(v) := ⊥, inQueue(v) := false
3   g(s) := 0, Init(Q), Enqueue(Q,(0,s)), inQueue(s) := true
4   while (i,u) := Dequeue(Q) ≠ ⊥ and i < n do
5     inQueue(u) := false
6     for all v ∈ N+(u) do
7       if g(u) + l(u,v) < g(v) then
8         g(v) := g(u) + l(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    error(es gibt einen negativen Kreis)
```

Für kreisfreie Graphen lässt sich eine lineare Laufzeit $\mathcal{O}(n+m)$ erzielen, indem die Nachfolger in Zeile 6 in topologischer Sortierung gewählt werden. Dies bewirkt, dass jeder Knoten höchstens einmal in die Schlange eingefügt wird.

Beispiel 2.4. *Betrachte untenstehenden kantenbewerteten Digraphen mit dem Startknoten a.*

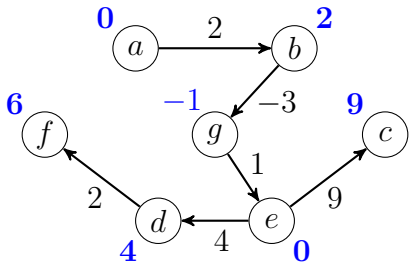


Die folgende Tabelle zeigt jeweils den Inhalt der Schlange Q, bevor der BFM-Algorithmus das nächste Paar (i, u) von Q entfernt. Dabei enthält jeder Eintrag (i, u, v, g) neben der Rundenzahl i und dem Knoten u auch noch den `parent`*-Knoten v und den g-Wert von u, obwohl diese nicht in Q gespeichert werden.*

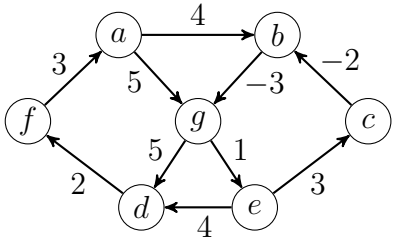
$$
\begin{array}{l}
\Uparrow \\
\left|(0,a,\perp,0)\right| \\
\Uparrow
\end{array}
\left|\begin{array}{l}(1,b,a,2)\\(1,g,a,5)\end{array}\right|
\left|(1,g,b,-1)\right|
\left|\begin{array}{l}(2,d,g,7)\\(2,e,g,0)\end{array}\right|
$$

$$
\left|\begin{array}{l}(2,e,g,0)\\(3,f,d,9)\end{array}\right|
\left|\begin{array}{l}(3,f,d,9)\\(3,c,d,9)\\(3,d,e,4)\end{array}\right|
\left|\begin{array}{l}(3,c,d,9)\\(3,d,e,4)\end{array}\right|
\left|(3,d,e,4)\right|
\left|(4,f,d,6)\right|
$$

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



Als nächstes betrachten wir den folgenden Digraphen:

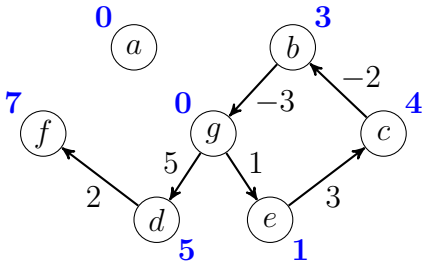


Da dieser einen negativen Kreis enthält, der vom Startknoten aus erreichbar ist, lassen sich die Entfernungen zu allen Knoten, die von diesem Kreis aus erreichbar sind, beliebig verkleinern.

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

$$
\left|(3,c,e,5)\right|
\left|(4,b,c,3)\right|
\left|(5,g,b,0)\right|
\left|\begin{array}{l}(6,d,g,5)\\(6,e,g,1)\end{array}\right|
\left|\begin{array}{l}(2,e,g,1)\\(7,f,d,7)\end{array}\right|
\left|\begin{array}{l}(7,f,d,7)\\(7,c,e,4)\end{array}\right|
$$

Da nun der Knoten f mit der Rundenzahl $i = n = 7$ 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.*



◁

2.4 Der Floyd-Warshall-Algorithmus

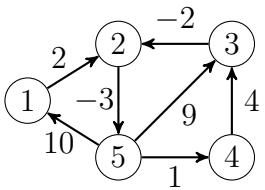
Der Algorithmus von Floyd-Warshall berechnet die Distanzen zwischen allen Knoten unter der Voraussetzung, dass keine negativen Kreise existieren.

Algorithmus `Floyd-Warshall`(V, E, l)

```
1  for i := 1 to n do
2    for j := 1 to n do
3      if (i,j) ∈ E then d_0(i,j) := l(i,j) else d_0(i,j) := ∞
4  for k := 1 to n do
5    for i := 1 to n do
6      for j := 1 to n do
7        d_k(i,j) = min{d_{k-1}(i,j), d_{k-1}(i,k) + d_{k-1}(k,j)}
```

Hierzu speichert der Algorithmus in $\mathsf{d}_k(i,j)$ die Länge eines kürzesten Weges von i nach j, der außer i und j nur Knoten $\leq k$ besucht. Die Laufzeit ist offenbar $\mathcal{O}(n^3)$. Da die d_k-Werte nur von den d_{k-1}-Werten abhängen, ist der Speicherplatzbedarf $\mathcal{O}(n^2)$. Die Existenz negativer Kreise lässt sich daran erkennen, dass mindestens ein Diagonalelement $\mathsf{d}_k(i,i)$ einen negativen Wert erhält.

Beispiel 2.5. *Betrachte folgenden kantenbewerteten Digraphen:*



d_0	1	2	3	4	5
1	∞	2	∞	∞	∞
2	∞	∞	∞	∞	−3
3	∞	−2	∞	∞	∞
4	∞	∞	4	∞	∞
5	10	∞	9	1	∞

d_1	1	2	3	4	5
1	∞	2	∞	∞	∞
2	∞	∞	∞	∞	−3
3	∞	−2	∞	∞	∞
4	∞	∞	4	∞	∞
5	10	12	9	1	∞

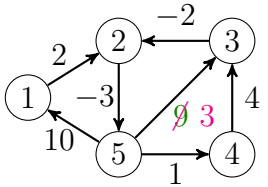
d_2	1	2	3	4	5
1	∞	2	∞	∞	−1
2	∞	∞	∞	∞	−3
3	∞	−2	∞	∞	−5
4	∞	∞	4	∞	∞
5	10	12	9	1	9

d_3	1	2	3	4	5
1	∞	2	∞	∞	−1
2	∞	∞	∞	∞	−3
3	∞	−2	∞	∞	−5
4	∞	2	4	∞	−1
5	10	7	9	1	4

d_4	1	2	3	4	5
1	∞	2	∞	∞	−1
2	∞	∞	∞	∞	−3
3	∞	−2	∞	∞	−5
4	∞	2	4	∞	−1
5	10	3	5	1	0

d_5	1	2	3	4	5
1	9	2	4	0	−1
2	7	0	2	−2	−3
3	5	−2	0	−4	−5
4	9	2	4	0	−1
5	10	3	5	1	0

Als nächstes betrachten wir folgenden Digraphen:



d_0	1	2	3	4	5
1	∞	2	∞	∞	∞
2	∞	∞	∞	∞	−3
3	∞	−2	∞	∞	∞
4	∞	∞	4	∞	∞
5	10	∞	3	1	∞

d_1	1	2	3	4	5
1	∞	2	∞	∞	∞
2	∞	∞	∞	∞	−3
3	∞	−2	∞	∞	∞
4	∞	∞	4	∞	∞
5	10	12	3	1	∞

d_2	1	2	3	4	5
1	∞	2	∞	∞	-1
2	∞	∞	∞	∞	-3
3	∞	-2	∞	∞	-5
4	∞	∞	4	∞	∞
5	10	12	3	1	9

d_3	1	2	3	4	5
1	∞	2	∞	∞	-1
2	∞	∞	∞	∞	-3
3	∞	-2	∞	∞	-5
4	∞	2	4	∞	-1
5	10	1	3	1	-2

d_4	1	2	3	4	5
1	∞	2	∞	∞	-1
2	∞	∞	∞	∞	-3
3	∞	-2	∞	∞	-5
4	∞	2	4	∞	-1
5	10	1	3	1	-2

d_5	1	2	3	4	5
1	9	0	2	0	-3
2	7	-2	0	-2	-5
3	5	-4	-2	-4	-7
4	9	0	2	0	-3
5	8	-1	1	-1	-4

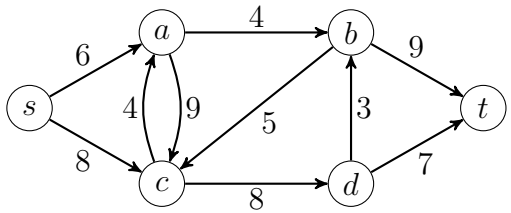
Wegen $d_3(5,5) = -2$ liegt der Knoten 5 auf einem negativen Kreis. Folglich ist die Weglänge nicht für alle Knotenpaare nach unten beschränkt. ◁

Ohne großen Mehraufwand lassen sich auch die kürzesten Wege selbst berechnen, indem man in einem Feld `parent`$[i,j]$ den Vorgänger von j auf einem kürzesten Weg von i nach j speichert (falls ein Weg von i nach j existiert). Eine elegantere Möglichkeit besteht jedoch darin, die Kantenfunktion l in eine äquivalente Distanzfunktion l' zu transformieren, die keine negativen Werte annimmt, aber dieselben kürzesten Wege in G wie l hat. Da wir für diese Transformation nur alle kürzesten Wege von einem festen Knoten s zu allen anderen Knoten berechnen müssen, ist sie in Zeit $O(nm)$ durchführbar.

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.*

Die folgende Abbildung zeigt ein Netzwerk N.



Definition 3.2.

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

$$f(u,v) \le c(u,v), \qquad \text{(Kapazitätsbedingung)}$$
$$f(u,v) = -f(v,u), \qquad \text{(Antisymmetrie)}$$
$$\textstyle\sum_{v \in V} f(u,v) = 0 \text{ für alle } u \in V \setminus \{s,t\} \qquad \text{(Kontinuität)}$$

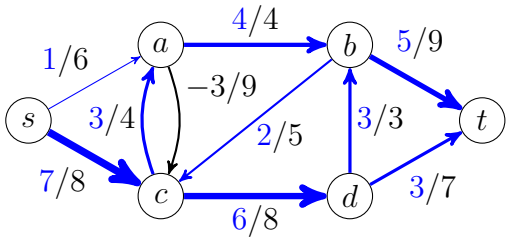
*b) Der **Fluss in den Knoten u** ist $f^-(u) = \sum_{v \in V} \max\{0, f(v,u)\}$.*

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

*d) Der **Nettofluss durch u** ist $f^+(u) - f^-(u) = \sum_{v \in V} f(u,v)$.*

*e) Die **Größe von f** ist $|f| = f^+(s) - f^-(s)$.*

Die Antisymmetrie impliziert, dass $f(u,u) = 0$ für alle $u \in V$ ist, d.h. wir können annehmen, dass G schlingenfrei ist. Die folgende Abbildung zeigt einen Fluss f in N.



u	s	a	b	c	d	t
$f^+(u)$	8	4	7	9	6	0
$f^-(u)$	0	4	7	9	6	8

3.1 Der Ford-Fulkerson-Algorithmus

Wie lässt sich für einen Fluss f in einem Netzwerk N entscheiden, ob er vergrößert werden kann? Diese Frage lässt sich leicht beantworten, falls f der konstante Nullfluss $f = 0$ ist: 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.

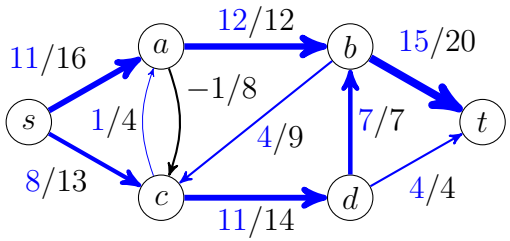
Definition 3.3. *Sei $N = (V, E, s, t, c)$ ein Netzwerk und sei f ein Fluss in N. Das zugeordnete* ***Restnetzwerk*** *ist $N_f = (V, E_f, s, t, c_f)$ mit der Kapazität*

$$c_f(u, v) = c(u, v) - f(u, v)$$

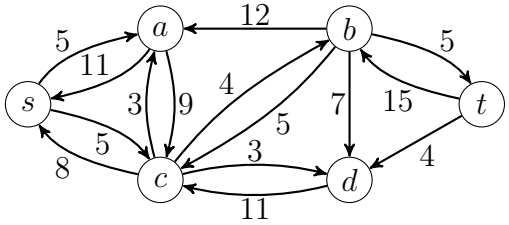
und der Kantenmenge

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

Zum Beispiel führt der Fluss



auf das folgende Restnetzwerk N_f:



Definition 3.4. *Sei $N_f = (V, E_f, s, t, c_f)$ ein Restnetzwerk. Dann heißt jeder s-t-Pfad P in (V, E_f)* ***Zunahmepfad*** *in N_f. Die* ***Kapazität von P in N_f*** *ist*

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

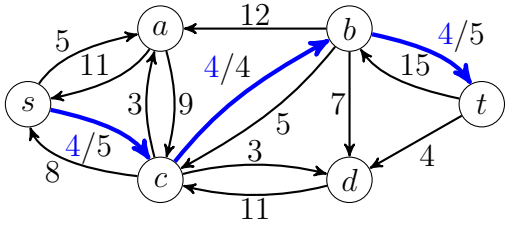
und der ***zu P gehörige Fluss in N_f*** *ist*

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

$P = (u_0, \ldots, u_k)$ ist also genau dann ein Zunahmepfad in N_f, falls

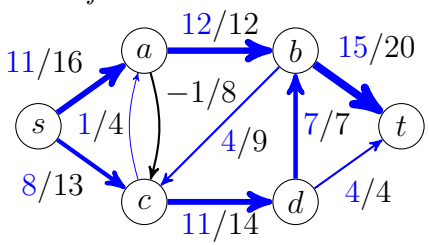
- $u_0 = s$ und $u_k = t$ ist,
- die Knoten $u_0, \ldots, u_k$ paarweise verschieden sind
- und $c_f(u_i, u_{i+1}) > 0$ für $i = 0, \ldots, k - 1$ ist.

Die folgende Abbildung zeigt den zum Zunahmepfad $P = s, c, b, t$ gehörigen Fluss f_P in N_f. Die Kapazität von P ist $c_f(P) = 4$.

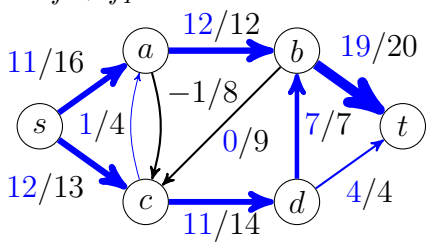


Es ist leicht zu sehen, dass f_P tatsächlich ein Fluss in N_f ist. Durch Addition der beiden Flüsse f und f_P erhalten wir einen Fluss $f' = f + f_P$ in N der Größe $|f'| = |f| + |f_P| > |f|$.

Fluss f:



Fluss $f + f_P$:

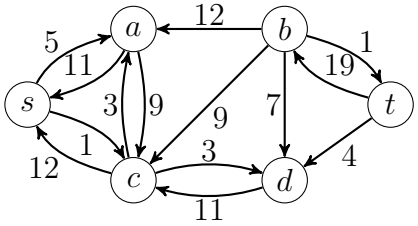


Nun können wir den **Ford-Fulkerson-Algorithmus** angeben.

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

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

Beispiel 3.5. *Für den neuen Fluss erhalten wir nun folgendes Restnetzwerk:*



In diesem existiert kein Zunahmepfad mehr. ◁

Um zu beweisen, dass der Algorithmus von Ford-Fulkerson tatsächlich einen Maximalfluss berechnet, zeigen wir, dass es nur dann im Restnetzwerk N_f keinen Zunahmepfad mehr gibt, wenn der Fluss f maximal ist. Hierzu benötigen wir den Begriff des Schnitts.

Definition 3.6. *Sei $N = (V, E, s, t, c)$ ein Netzwerk und sei $\emptyset \subsetneq S \subsetneq V$. Dann heißt die Menge $E(S) = \{(u, v) \in E \mid u \in S, v \notin S\}$* ***Kantenschnitt*** *(oder einfach* ***Schnitt****; oft wird auch einfach S als Schnitt bezeichnet). Die* ***Kapazität eines Schnittes S*** *ist*

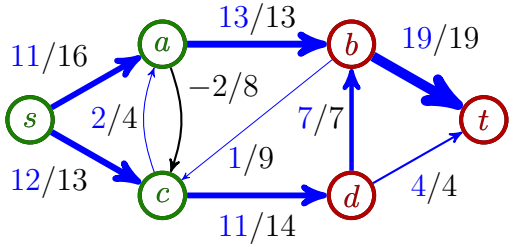
$$c(S) = \sum_{u \in S, v \notin S} c(u, v).$$

Ist f ein Fluss in N, so heißt

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

der ***Nettofluss*** *(oder einfach* ***Fluss****)* ***durch den Schnitt S****.*

Beispiel 3.7. *Betrachte den Schnitt $S = \{s, a, c\}$ in folgendem Netzwerk N mit dem Fluss f:*



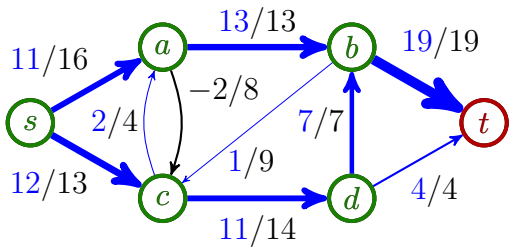
Dieser Schnitt hat die Kapazität

$$c(S) = c(a, b) + c(c, d) = 13 + 14 = 27$$

und der Fluss $f(S)$ durch diesen Schnitt ist

$$f(S) = f(a,b) + f(c,b) + f(c,d) = 13 - 1 + 11 = 23.$$

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



die Kapazität

$$c(S') = c(b,t)+c(d,t) = 19+4 = f(b,t)+f(d,t) = f(S'),$$

die mit dem Fluss durch diesen Schnitt übereinstimmt. ◁

Lemma 3.8. *Für jeden Schnitt S mit $s \in S$, $t \notin S$ und jeden Fluss f gilt*

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

Beweis. Die Gleichheit $|f| = f(S)$ zeigen wir durch Induktion über $k = \|S\|$.

$k = 1$**:** In diesem Fall ist $S = \{s\}$ und somit

$$|f| = f^+(s) - f^-(s) = \sum_{v \in V} f(s,v) = \underbrace{f(s,s)}_{=0} + \sum_{v \neq s} f(s,v) = f(S).$$

$k-1 \rightsquigarrow k$**:** Sei S ein Schnitt mit $\|S\| = k > 1$ und sei $w \in S - \{s\}$. Betrachte den Schnitt $S' = S - \{w\}$. Dann gilt

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

und

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

Wegen $f(w,w) = 0$ ist $\sum_{u \in S'} f(u,w) = \sum_{u \in S} f(u,w)$ und daher

$$f(S) - f(S') = \sum_{v \notin S} f(w,v) - \sum_{u \in S} f(u,w) \quad = \sum_{v \in V} f(w,v) = 0.$$

Nach Induktionsvoraussetzung folgt somit $f(S) = f(S') = |f|$. Schließlich folgt wegen $f(u,v) \leq c(u,v)$ die Ungleichung

$$f(S) = \sum_{(u,v) \in E(S)} f(u,v) \leq \sum_{(u,v) \in E(S)} c(u,v) = c(S). \qquad \blacksquare$$

Satz 3.9 (Min-Cut-Max-Flow-Theorem)**.** *Sei f ein Fluss in einem Netzwerk $N = (V,E,s,t,c)$. Dann sind folgende Aussagen äquivalent:*

1. *f ist maximal.*
2. *In N_f existiert kein Zunahmepfad.*
3. *Es gibt einen Schnitt S in N mit $s \in S$, $t \notin S$ und $c(S) = |f|$.*

Beweis. Die Implikation „*1* $\Rightarrow$ *2*“ ist klar, da die Existenz eines Zunahmepfads zu einer Vergrößerung von f führen würde.

Für die Implikation „*2* $\Rightarrow$ *3*“ betrachten wir den Schnitt

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

Da in N_f kein Zunahmepfad existiert, gilt dann

- $s \in S$, $t \notin S$ und
- $c_f(u,v) = 0$ für alle $u \in S$ und $v \notin S$.

Wegen $c_f(u,v) = c(u,v) - f(u,v)$ folgt somit

$$|f| = f(S) = \sum_{u \in S, v \notin S} f(u,v) = \sum_{u \in S, v \notin S} c(u,v) = c(S).$$

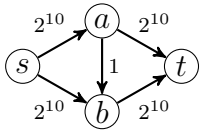
Die Implikation „*3* $\Rightarrow$ *1*“ ist wiederum klar, da im Fall $c(S) = |f|$ für jeden Fluss f' die Abschätzung $|f'| = f'(S) \leq c(S) = |f|$ gilt. $\blacksquare$

Der obige Satz gilt auch für Netzwerke mit Kapazitäten in $\mathbb{R}^+$.

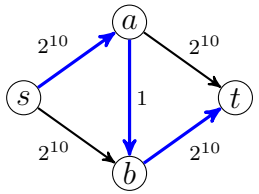
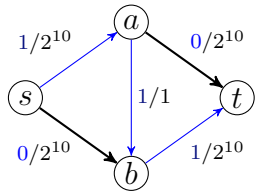
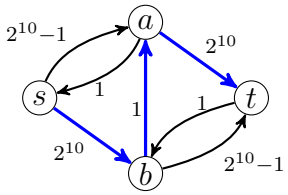
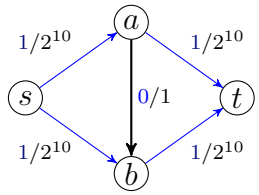
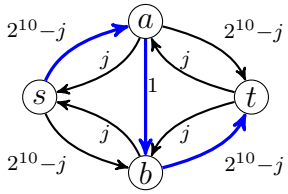
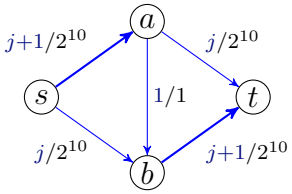
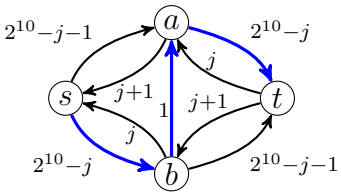
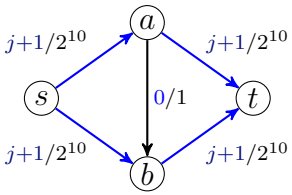
Sei $c_0 = c(S)$ die Kapazität des Schnittes $S = \{s\}$. Dann durchläuft der Ford-Fulkerson-Algorithmus die while-Schleife höchstens c_0-mal. Bei jedem Durchlauf ist zuerst das Restnetzwerk N_f und danach ein Zunahmepfad in N_f zu berechnen.

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 polynomielle Zeitschranke. Bei Netzwerken mit Kapazitäten in $\mathbb{R}^+$ 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 zuerst der Zunahmepfad (s,a,t) und dann der Pfad (s,b,t) gewählt. Im ungünstigsten Fall werden abwechselnd die beiden Zunahmepfade (s,a,b,t) und (s,b,a,t) gewählt:

i	Zunahmepfad P_i in $N_{f_{i-1}}$	neuer Fluss f_i in N
1		
2		
$2j+1$		
$2j+2$		

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 s-t-Pfad P mindestens eine Kante $e \in P$ *blockiert* (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 Dinic*. Da die Länge der kürzesten s-t-Pfade im Restnetzwerk in jeder Iteration um mindestens 1 zunimmt, liegt nach spätestens $n-1$ Iterationen ein maximaler Fluss vor. Dinic hat gezeigt, dass ein blockierender Fluss g in Zeit $\mathcal{O}(nm)$ bestimmt werden kann. Folglich hat der Algorithmus von Dinic eine Laufzeit von $\mathcal{O}(n^2m)$. Malhotra, Kumar und Maheswari fanden später einen $\mathcal{O}(n^2)$-Algorithmus zur Bestimmung eines blockierenden Flusses. Damit lässt sich die Gesamtlaufzeit auf $\mathcal{O}(n^3)$ verbessern.

3.2 Der Edmonds-Karp-Algorithmus

Der Edmonds-Karp-Algorithmus ist eine spezielle Form von Ford-Fulkerson, die nur Zunahmepfade mit möglichst wenigen Kanten benutzt, welche mittels Breitensuche bestimmt werden.

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

```
1  for all (u,v) ∈ V × V do
2    f(u,v) := 0
3  repeat
4    P ← zunahmepfad(f)
5    if P ≠ ⊥ then add(f,P)
6  until P = ⊥
```

Prozedur `zunahmepfad`(f)

```
1  for all v ∈ V do
2    vis(v) := false
3    parent(v) := ⊥
4  vis(s) := true
5  QueueInit(Q)
6  Enqueue(Q, s)
7  while ¬QueueEmpty(Q) ∧ Head(Q) ≠ t do
8    u := Head(Q)
9    Dequeue(Q)
10   for all v ∈ N⁻(v) ∪ N⁺(v) do
11     e := (u,v)
12     if c(e) − f(e) > 0 ∧ vis(v) = false then
13       c'(e) := c(e) − f(e)
14       vis(v) := true
15       parent(v) := u
16       Enqueue(Q, v)
17 if Head(Q) = t then
18   P := parent-Pfad von s nach t
19   c_f(P) := min{c'(e) | e ∈ P}
20 else
21   P := ⊥
22 return P
```

Prozedur `add`(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)
```

Satz 3.10. *Der Edmonds-Karp-Algorithmus durchläuft die repeat-Schleife höchstens $nm/2$-mal und hat somit eine Laufzeit von $O(nm^2)$.*

Beweis. Sei f_0 der triviale Fluss und seien $P_1, \ldots, P_k$ die Zunahmepfade, die der Edmonds-Karp-Algorithmus der Reihe nach berechnet,

d.h. $f_i = f_{i-1} + f_{P_i}$. Eine Kante e heißt ***kritisch*** in P_i, falls der Fluss f_{P_i} die Kante e sättigt, d.h. $c_{f_{i-1}}(e) = f_{P_i}(e) = c_{f_{i-1}}(P_i)$. Man beachte, dass eine kritische Kante e in P_i wegen $c_{f_i}(e) = c_{f_{i-1}}(e) - f_{P_i}(e) = 0$ nicht in N_{f_i} enthalten ist, wohl aber e^R.

Wir überlegen uns zunächst, dass die Längen ℓ_i von P_i (schwach) monoton wachsen. Hierzu beweisen wir die stärkere Behauptung, dass sich die Abstände jedes Knotens $u \in V$ von s und von t beim Übergang von $N_{f_{i-1}}$ zu N_{f_i} nicht verringern können. Sei $d_i(u,v)$ die minimale Länge eines Pfades von u nach v im Restnetzwerk $N_{f_{i-1}}$.

Behauptung 3.11. *Für jeden Knoten $u \in V$ gilt $d_{i+1}(s,u) \geq d_i(s,u)$ und $d_{i+1}(u,t) \geq d_i(u,t)$.*

Hierzu zeigen wir folgende Behauptung.

Behauptung 3.12. *Falls die Kante $e = (u_j, u_{j+1})$ auf einem kürzesten Pfad $P = (u_0, \ldots, u_h)$ von $s = u_0$ nach $u = u_h$ in N_{f_i} liegt (d.h. $d_{i+1}(s,u_{j+1}) = d_{i+1}(s,u_j) + 1$), dann gilt $d_i(s,u_{j+1}) \leq d_i(s,u_j) + 1$.*

Die Behauptung ist klar, wenn die Kante $e = (u_j, u_{j+1})$ auch in $N_{f_{i-1}}$ enthalten ist. Ist dies nicht der Fall, muss $f_{i-1}(e) \neq f_i(e)$ sein, d.h. e oder e^R müssen in P_i vorkommen. Da e nicht in $N_{f_{i-1}}$ ist, muss $e^R = (u_{j+1}, u_j)$ auf P_i liegen. Da P_i ein kürzester Pfad von s nach t in $N_{f_{i-1}}$ ist, folgt $d_i(s,u_j) = d_i(s,u_{j+1}) + 1$, was $d_i(s,u_{j+1}) = d_i(s,u_j) - 1 \leq d_i(s,u_j) + 1$ impliziert.

Damit ist Behauptung 3.12 bewiesen und es folgt

$$d_i(s,u) \leq d_i(s,u_{h-1}) + 1 \leq \cdots \leq d_i(s,s) + h = h = d_{i+1}(s,u).$$

Die Ungleichung $d_{i+1}(u,t) \geq d_i(u,t)$ folgt analog, womit auch Behauptung 3.11 bewiesen ist. Als nächstes zeigen wir folgende Behauptung.

Behauptung 3.13. *Für $1 \leq i < j \leq k$ gilt: Falls $e = (u,v)$ in P_i und $e^R = (v,u)$ in P_j enthalten ist, so ist $l_j \geq l_i + 2$.*

Dies folgt direkt aus Behauptung 3.11:

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

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 $k \leq mn/2$ ist.

Behauptung 3.14. *Zwei Kanten e und e^R sind zusammen höchstens $n/2$-mal kritisch.*

Seien $P_{i_1}, \ldots, P_{i_h}$ die Pfade, in denen e oder e^R kritisch ist. Falls $k \in \{e, e^R\}$ kritisch in P_{i_j} ist, dann fällt k aus $N_{f_{i_j+1}}$ heraus. Damit also e oder e^R kritisch in $P_{i_{j+1}}$ sein können, muss ein Pfad $P_{j'}$ mit $i_j < j' \leq i_{j+1}$ existieren, der k^R enthält. Wegen Behauptung 3.11 und Behauptung 3.13 ist $\ell_{i_{j+1}} \geq \ell_{j'} \geq \ell_{i_j} + 2$. Daher ist

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

was $h \leq n/2$ impliziert. ■

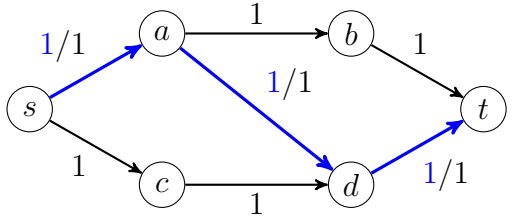
Man beachte, dass der Beweis auch bei Netzwerken mit reellen Kapazitäten seine Gültigkeit behält.

3.3 Der Algorithmus von Dinic

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 $O(n+m)$ bestimmen lassen. Wenn ja, würde dies auf eine Gesamtlaufzeit von $O(n+m^2)$ führen. Für dichte Netzwerke (d.h. $m = \Theta(n^2)$) hat der Algorithmus von Dinic die gleiche Laufzeit $O(n^2 m) = O(n^4)$ und die verbesserte Version ist mit $O(n^3)$ in diesem Fall sogar noch schneller.

Definition 3.15. *Sei $N = (V, E, s, t, c)$ ein Netzwerk und sei g ein Fluss in N. Der Fluss g* ***sättigt*** *eine Kante $e \in E$, falls $g(e) = c(e)$ ist. g heißt* ***blockierend****, falls g auf jedem Pfad P von s nach t mindestens eine Kante $e \in E$ sättigt.*

Nach dem Min-Cut-Max-Flow-Theorem gibt es zu jedem maximalen Fluss f einen 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. Wir werden sehen, dass sich ein blockierender Fluss in Zeit $O(n^2)$ berechnen lässt.

Der Algorithmus von Dinic arbeitet wie folgt.

Algorithmus `Dinic`(V, E, s, t, c)

```
1  for all (u, v) ∈ V × V do
2      f(u, v) := 0
3  while schichtnetzwerk(f) do
4      g := blockfluss(f)
5      f := f + g
```

Die Prozedur `blockfluss`(f) berechnet einen blockierenden Fluss im Restnetzwerk N_f, der für alle Kanten den Wert 0 hat, die nicht auf einem kürzesten Pfad P von s nach t in N_f liegen. Hierzu werden aus N_f alle Knoten $u \neq t$ entfernt, die einen Abstand $d(s, u) \geq d(s, t)$ in N_f haben. Falls in N_f kein Pfad von s nach t existiert (d.h. $d(s, t) = \infty$), wird auch t entfernt.

Das resultierende Netzwerk N'_f wird als ***Schichtnetzwerk*** bezeichnet, da jeder Knoten in N'_f einer Schicht S_j zugeordnet werden kann: Für $0 \leq j < d(s, t)$ ist $S_j = \{u \in V \mid d(s, u) = j\}$. Im Fall $d(s, t) < \infty$ kommt für $j = d(s, t)$ noch die Schicht $S_j = \{t\}$ hinzu. Zudem werden alle Kanten aus N_f entfernt, die nicht auf einem kürzesten Pfad von s zu einem Knoten in N'_f liegen, d.h. jede Kante (u, v) in N'_f verbindet einen Knoten u in Schicht S_j mit einem Knoten v in Schicht S_{j+1} von N'_f.

Das Schichtnetzwerk N'_f wird von der Prozedur `schichtnetzwerk` durch eine modifizierte Breitensuche in Zeit $O(n + m)$ berechnet. Diese Prozedur gibt den Wert `true` zurück, falls t im berechneten Schichtnetzwerk N'_f enthalten (und somit der aktuelle Fluss f noch nicht maximal) ist, und sonst den Wert `false`.

Satz 3.16. *Der Algorithmus von Dinic durchläuft die while-Schleife höchstens n-mal.*

Beweis. Sei k die Anzahl der Schleifendurchläufe und für $i = 1, \ldots, k$ sei g_i der blockierende Fluss, den der Dinic-Algorithmus im Schichtnetzwerk $N'_{f_{i-1}}$ berechnet, d.h. $f_i = f_{i-1} + g_i$. Zudem sei $d_i(u, v)$ wieder die minimale Länge eines Pfades von u nach v im Restnetzwerk $N_{f_{i-1}}$. Wir zeigen, dass $d_{i+1}(s, t) > d_i(s, t)$ ist. Da $d_1(s, t) \geq 1$ und $d_k(s, t) \leq n - 1$ ist, folgt $k \leq n - 1$.

Behauptung 3.17. *Für jeden Knoten $u \in V$ gilt $d_{i+1}(s, u) \geq d_i(s, u)$.*

Hierzu zeigen wir folgende Behauptung.

Behauptung 3.18. *Falls die Kante $e = (u_j, u_{j+1})$ auf einem kürzesten Pfad $P = (u_0, \ldots, u_h)$ von $s = u_0$ nach $u = u_h$ in N_{f_i} liegt (d.h. $d_{i+1}(s, u_{j+1}) = d_{i+1}(s, u_j) + 1$), dann gilt $d_i(s, u_{j+1}) \leq d_i(s, u_j) + 1$.*

Die Behauptung ist klar, wenn die Kante $e = (u_j, u_{j+1})$ auch in $N_{f_{i-1}}$ enthalten ist. Ist dies nicht der Fall, muss $f_{i-1}(e) \neq f_i(e)$ sein, d.h. $g_i(e)$ muss ungleich 0 sein. Da e nicht in $N_{f_{i-1}}$ und somit auch nicht in $N'_{f_{i-1}}$ ist, muss $e^R = (u_{j+1}, u_j)$ in $N'_{f_{i-1}}$ sein. Da $N'_{f_{i-1}}$ nur Kanten auf kürzesten Pfaden von s zu einem Knoten in $N'_{f_{i-1}}$ enthält, folgt $d_i(s, u_j) = d_i(s, u_{j+1}) + 1$, was $d_i(s, u_{j+1}) = d_i(s, u_j) - 1 \leq d_i(s, u_j) + 1$ impliziert.

Damit ist Behauptung 3.18 bewiesen und Behauptung 3.17 folgt wie im Beweis von Satz 3.10. Als nächstes zeigen wir folgende Behauptung.

Behauptung 3.19. *Für $i = 1, \ldots, k-1$ gilt $d_{i+1}(s,t) > d_i(s,t)$.*

Sei $P = (u_0, u_1, \ldots, u_h)$ ein kürzester Pfad von $s = u_0$ nach $t = u_h$ in N_{f_i}. Dann gilt wegen Behauptung 3.17, dass $d_i(s, u_j) \leq d_{i+1}(s, u_j) = j$ für $j = 0, \ldots, h$ ist.

Wir betrachten zwei Fälle. Wenn alle Knoten u_j in $N'_{f_{i-1}}$ enthalten sind, führen wir die Annahme $d_i(s,t) = d_{i+1}(s,t)$ auf einen Widerspruch. Wegen Behauptung 3.18 folgt aus dieser Annahme nämlich die Gleichheit $d_i(s, u_{j+1}) = d_i(s, u_j) + 1$, da sonst $d_i(s,t) < h$ wäre. Folglich ist P auch ein kürzester Pfad von s nach t in $N_{f_{i-1}}$ und somit g_i kein blockierender Fluss in $N_{f_{i-1}}$.

Es bleibt der Fall, dass mindestens ein Knoten u_j nicht in $N'_{f_{i-1}}$ enthalten ist. Sei u_{j+1} der erste Knoten auf P, der nicht in $N'_{f_{i-1}}$ enthalten ist. Dann ist $u_{j+1} \neq t$ und daher $d_{i+1}(s,t) > d_{i+1}(s, u_{j+1})$. Zudem liegt die Kante $e = (u_j, u_{j+1})$ nicht nur in N_{f_i}, sondern wegen $f_i(e) = f_{i-1}(e)$ (da weder e noch e^R zu $N'_{f_{i-1}}$ gehören) auch in $N_{f_{i-1}}$. Da somit u_j in $N'_{f_{i-1}}$ und e in $N_{f_{i-1}}$ ist, kann u_{j+1} nur aus dem Grund nicht zu $N'_{f_{i-1}}$ gehören, dass $d_i(s, u_{j+1}) = d_i(s,t)$ ist. Daher folgt wegen $d_{i+1}(s, u_j) \geq d_i(s, u_j)$ (Behauptung 3.17) und $d_i(s, u_j) + 1 \geq d_i(s, u_{j+1})$ (Behauptung 3.18)

$$d_{i+1}(s,t) > d_{i+1}(s, u_{j+1}) = d_{i+1}(s, u_j) + 1 \geq d_i(s, u_{j+1}) = d_i(s,t). \quad \blacksquare$$

Die Prozedur `schichtnetzwerk` führt eine Breitensuche mit Startknoten s im Restnetzwerk N_f aus und speichert dabei in der Menge E' nicht nur alle Baumkanten, sondern zusätzlich alle Querkanten (u, v), die auf einem kürzesten Weg von s zu v liegen. Sobald alle von s aus erreichbaren Knoten besucht (und in V' gespeichert) wurden oder t am Kopf der Warteschlange Q erscheint, bricht die Suche ab. Falls der Kopf von Q gleich t ist, werden alle Knoten $v \neq t$, die die gleiche Entfernung von s wie t haben, sowie alle Kanten, die in diesen Knoten enden, wieder aus N'_f entfernt.

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

Prozedur `schichtnetzwerk`(f)

```
1  for all v ∈ V do
2    niv(v) := n
3  niv(s) := 0
4  V' := {s}
5  E' := ∅
6  QueueInit(Q)
7  Enqueue(Q, s)
8  while ¬QueueEmpty(Q) ∧ Head(Q) ≠ t do
9    u := Head(Q)
10   Dequeue(Q)
11   for all v ∈ N⁺(u) ∪ N⁻(u) do
12     e := (u, v)
13     if c(e) − f(e) > 0 ∧ niv(v) > niv(u) then
14       V' := V' ∪ {v}
15       E' := E' ∪ {e}
16       c'(e) := c(e) − f(e)
17       if niv(v) > niv(u) + 1 then
18         niv(v) := niv(u) + 1
19         Enqueue(Q, v)
```

```
20  if Head(Q) = t then
21     V'' := {v ∈ V' | v ≠ t, niv(v) = niv(t)}
22     V' := V' \ V''
23     E' := E' \ (V' × V'')
24     return true
25  else
26     return false
```

Die Prozedur `blockfluss1` berechnet einen blockierenden Fluss g im Schichtnetzwerk N'_f in der Zeit $O(nm)$. Hierzu bestimmt sie in der repeat-Schleife mittels Tiefensuche einen Zunahmepfad P in N'_{f+g}, addiert den Fluss $(f+g)_P$ zum aktuellen Fluss g hinzu, und entfernt die gesättigten Kanten $e \in P$ aus E'. Falls die Tiefensuche in einer Sackgasse endet (weil E' keine weiterführenden Kanten enthält), wird die zuletzt besuchte Kante (u', u) ebenfalls aus E' entfernt und die Tiefensuche vom Startpunkt u' dieser Kante fortgesetzt (back tracking). Die Prozedur `blockfluss1` bricht ab, falls keine weiteren Pfade von s nach t existieren. Folglich ist der berechnete Fluss g tatsächlich blockierend.

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

Prozedur `blockfluss1`(f)

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

Die Prozedur `blockfluss2` benötigt nur Zeit $O(n^2)$, um einen blockierenden Fluss g im Schichtnetzwerk N'_f zu berechnen. Zu ihrer Beschreibung benötigen wir folgende Notation.

Definition 3.20. *Sei $N = (V, E, s, t, c)$ ein Netzwerk und sei u ein Knoten in N. Die* ***Ausgangskapazität*** *von u ist*

$$c^+(u) = \sum_{(u,v)\in E} c(u, v)$$

und die ***Eingangskapazität*** *von u ist*

$$c^-(u) = \sum_{(v,u)\in E} c(v, u).$$

Der ***Durchsatz*** *von u ist*

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

Ein Fluss g in N ***sättigt*** *einen Knoten $u \in V$, falls $d(u) = \max\{f^+(u), f^-(u)\}$ ist.*

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

Proposition 3.21. *Sei $N = (V, E, s, t, c)$ ein Netzwerk und sei g ein Fluss in N. g ist blockierend, falls jeder s-t-Pfad in N mindestens einen Knoten enthält, der durch g gesättigt wird.*

Beweis. Dies folgt aus der Tatsache, dass ein Fluss g in N, der auf jedem s-t-Pfad P mindestens einen Knoten u sättigt, auch mindestens eine Kante in P sättigt. ■

Beginnend mit dem trivialen Fluss $g = 0$ berechnet die Prozedur `blockfluss2` für jeden Knoten u den Durchsatz $D(u)$ im Schichtnetzwerk N'_f und wählt in jedem Durchlauf der repeat-Schleife einen Knoten u mit minimalem Durchsatz $D(u)$, um den aktuellen Fluss g um den Wert $D(u)$ zu erhöhen. Hierzu benutzt sie die Prozeduren `propagierevor` und `propagiererück`, die dafür Sorge tragen, dass der zusätzliche Fluss tatsächlich durch den Knoten u fließt und die Durchsatzwerte $D(v)$ von allen Knoten aktualisiert werden, die von der Flusserhöhung betroffen sind. Aus diesem Grund wird u durch den zusätzlichen Fluss gesättigt und kann aus dem Netzwerk entfernt werden.

In der Menge B werden alle Knoten gespeichert, deren Durchsatz durch die Erhöhungen des Flusses g oder durch die Entfernung von

Kanten aus E' auf 0 gesunken ist. Diese Knoten und die mit ihnen verbundenen Kanten werden in der while-Schleife der Prozedur `blockfluss2` aus dem Schichtnetzwerk N'_f entfernt.

Prozedur `blockfluss2`(f)

```
1  for all e ∈ E' ∪ E'^R do g(e) := 0
2  for all u ∈ V' do
3     D+(u) := Σ_{uv∈E'} c'(u,v)
4     D-(u) := Σ_{vu∈E'} c'(v,u)
5  repeat
6     for all u ∈ V' \ {s,t} do
7        D(u) := min{D-(u), D+(u)}
8     D(s) := D+(s)
9     D(t) := D-(t)
10    wähle u ∈ V' mit D(u) minimal
11    Init(B); Insert(B,u)
12    propagierevor(u)
13    propagiererück(u)
14    while u := Remove(B) ∉ {s,t} do
15       V' := V' \ {u}
16       for all e = uv ∈ E' do
17          D-(v) := D-(v) - c'(u,v)
18          if D-(v) = 0 then Insert(B,v)
19          E' := E' \ {e}
20       for all e = vu ∈ E' do
21          D+(v) := D+(v) - c'(v,u)
22          if D+(v) = 0 then Insert(B,v)
23          E' := E' \ {e}
24 until u ∈ {s,t}
25 return g
```

Da in jedem Durchlauf der repeat-Schleife mindestens ein Knoten u gesättigt und aus V' entfernt wird, wird nach höchstens $n - 1$ Itera-

tionen 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 3.21 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 Knoten 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  QueueInit(Q); Enqueue(Q, u)
4  while v := Dequeue(Q) ≠ ⊥ do
5     while z(v) ≠ 0 ∧ ∃e = vu ∈ E' do
6        m := min{z(v), c'(e)}
7        z(v) := z(v) − m; z(u) := z(u) + m
8        aktualisierekante(e, m)
9        Enqueue(Q, u)
```

Prozedur `aktualisierekante`$(e = vu, m)$

```
1        g(e) := g(e) + m
2        c'(e) := c'(e) − m
3        if c'(e) = 0 then E' := E' \ {e}
4        D⁺(v) := D⁺(v) − m
5        if D⁺(v) = 0 then Insert(B, v)
6        D⁻(u) := D⁻(u) − m
7        if D⁻(u) = 0 then Insert(B, u)
```

Die Prozedur `propagiererück` unterscheidet sich von der Prozedur `propagierevor` nur dadurch, dass in Zeile 5 die Bedingung $\exists e = vu \in E'$ durch die Bedingung $\exists e = uv \in E'$ ersetzt wird.

Da die repeat-Schleife von `blockfluss2` maximal $(n-1)$-mal durchlaufen 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 $m = c'(u,v)$ 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 3.22. *Der Algorithmus von Dinic 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 Dinic-Algorithmus nachweisen.

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

(i) Falls jeder Knoten $u \in V \setminus \{s,t\}$ einen Durchsatz $d(u) \leq C$ hat, so durchläuft der Algorithmus von Dinic die while-Schleife höchstens $2(Cn)^{1/2}$ mal.

(ii) Falls jede Kante $e \in E$ eine Kapazität $c(e) \leq C$ hat, so durchläuft der Algorithmus von Dinic die while-Schleife höchstens $(2^5 Cn^2)^{1/3}$ mal.

Beweis. Sei $M = |f|$ die Größe eines maximalen Flusses f in N.

(i) Da die Anzahl a der Schleifendurchläufe durch M beschränkt ist, können wir $M > (Cn)^{1/2}$ annehmen. Betrachte den i-ten Schleifendurchlauf, in dem ein blockierender Fluss g_i im Schichtnetzwerk $N'_{f_{i-1}}$ mit den Schichten $S_0 = \{s\}, S_1, \ldots, S_{d_i-1}, S_{d_i} = \{t\}$ berechnet wird. Da ein maximaler Fluss in $N_{f_{i-1}}$ (in $N'_{f_{i-1}}$ kann er kleiner sein) die Größe $r_i = M - |f_{i-1}|$ hat und dieser durch die Knoten jeder einzelnen Schicht S_j, $1 \le j \le d_i - 1$, fließt, muss

$$r_i \le C\|S_j\| \text{ bzw. } r_i/C \le \|S_j\|,$$

sein, woraus

$$(d_i-1)r_i/C \le \|S_1\|+\cdots+\|S_{d_i-1}\| \le n-2 \le n \text{ bzw. } d_i \le 1+nC/r_i$$

folgt. Damit ist die Anzahl a der Schleifendurchläufe durch

$$a \le i + r_{i+1} \le d_i + r_{i+1} \le r_{i+1} + 1 + nC/r_i$$

beschränkt. Nun wählen wir i so, dass $r_i > (Cn)^{1/2}$ und $r_{i+1} \le (Cn)^{1/2}$ ist. Dann folgt

$$a - 1 < r_{i+1} + nC/r_i \le (Cn)^{1/2} + nC/(Cn)^{1/2} = 2(Cn)^{1/2}.$$

(ii) Da die Anzahl a der Schleifendurchläufe durch M beschränkt ist, können wir $M > (2n\sqrt{C})^{2/3}$ annehmen. Betrachte den i-ten Schleifendurchlauf, in dem ein blockierender Fluss g_i im Schichtnetzwerk $N'_{f_{i-1}}$ mit den Schichten $S_0 = \{s\}, S_1, \ldots, S_{d_i-1}, S_{d_i}$ berechnet wird. Hierbei nehmen wir zu S_{d_i} alle Knoten hinzu, die nicht in $N'_{f_{i-1}}$ liegen. Sei k_j die Anzahl der Kanten von S_j nach S_{j+1}. Da ein maximaler Fluss in $N_{f_{i-1}}$ (in $N'_{f_{i-1}}$ kann er wieder kleiner sein) die Größe $r_i = M - |f_{i-1}|$ hat und dieser für $j = 0, \ldots, d_i - 1$ durch die k_j Kanten von S_j nach S_{j+1} fließt, muss

$$r_i \le Ck_j \le C\|S_j\|\|S_{j+1}\| \text{ bzw. } r_i/C \le \|S_j\|\|S_{j+1}\|$$

sein. Somit enthält mindestens eine von zwei benachbarten Schichten S_j und S_{j+1} mindestens $\sqrt{r_i/C}$ Knoten, woraus

$$(d_i/2)\sqrt{r_i/C} \le \|S_0\| + \cdots + \|S_{d_i}\| \le n \text{ bzw. } d_i \le 2n\sqrt{C/r_i}$$

folgt. Damit ist die Anzahl a der Schleifendurchläufe durch

$$a \le i + r_{i+1} \le d_i + r_{i+1} \le r_{i+1} + 2n\sqrt{C/r_i}$$

beschränkt. Nun wählen wir i so, dass $r_i > (2n\sqrt{C})^{2/3}$ und $r_{i+1} \le (2n\sqrt{C})^{2/3}$ ist. Dann folgt

$$a \le (2n\sqrt{C})^{2/3} + 2n\sqrt{C}/(2n\sqrt{C})^{1/3} = (2^5Cn^2)^{1/3}. \qquad \blacksquare$$

Korollar 3.24. *Sei $N = (V, E, s, t, c)$ ein Netzwerk.*

(i) Falls jeder Knoten $u \in V \setminus \{s, t\}$ einen Durchsatz $d(u) \le C$ hat, so berechnet der Algorithmus von Dinic bei Verwendung der Prozedur `blockfluss1` *einen maximalen Fluss in Zeit $O((nC + m)\sqrt{Cn})$.*

(ii) Falls jede Kante $e \in E$ eine Kapazität $c(e) \le C$ hat, so berechnet der Algorithmus von Dinic bei Verwendung der Prozedur `blockfluss1` *einen maximalen Fluss in Zeit $O(C^{4/3}n^{2/3}m)$.*

Beweis. Zunächst ist leicht zu sehen, dass die Kapazitätschranke auf den Kanten oder Knoten auch für jedes Schichtnetzwerk N'_{f_i} gilt.

(i) Jedesmal wenn `blockfluss1` einen s-t-Pfad P im Schichtnetzwerk findet, verringert sich der Durchsatz $c''(u)$ der auf P liegenden Knoten u um den Wert $c'_g(P) \ge 1$, 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 $c'_g(P) \geq 1$. 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 $O(Cm + m) = O(Cm)$. ■

3.4 Kostenoptimale Flüsse

In manchen Anwendungen fallen für die Benutzung jeder Kante e eines Netzwerkes Kosten an, die proportional zur Höhe des Flusses $f(e)$ durch diese Kante sind. Falls die Kosten für die einzelnen Kanten differieren, ist es möglich, dass zwei Flüsse unterschiedliche Kosten verursachen, obwohl sie die gleiche Größe haben. Man möchte also einen maximalen Fluss f berechnen, der minimale Kosten hat.

Die Kosten eines Flusses f werden auf der Basis einer ***Kostenfunktion*** $k : E \to \mathbb{Z}$ berechnet, wobei für jede Kante $e \in E$ mit $f(e) \geq 0$ Kosten in Höhe von $f(e)k(e)$ anfallen.

Die Gesamtkosten von f im Netzwerk berechnen sich also zu

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

Ein negativer Kostenwert $k(e) < 0$ bedeutet, dass eine Erhöhung des Flusses durch die Kante e um 1 mit einem Gewinn in Höhe von $-k(e)$ verbunden ist. Ist zu einer Kante $e \in E$ auch die gegenläufige Kante e^R in E enthalten, so muss k die Bedingung $k(e^R) = -k(e)$ erfüllen.* Der Grund hierfür ist, dass die Erniedrigung von $f(e) > 0$ um einen bestimmten Wert $w \leq f(e)$ gleichbedeutend mit einer Erhöhung von $f(e^R)$ um diesen Wert im Restnetzwerk N_f ist und die Kostenfunktion auch für N_f gelten soll. Daher können wir k mittels $k(e) = -k(e^R)$, falls $e^R \in E$ und $k(e) = 0$ für alle $e \in (V \times V) \setminus (E \cup E^R)$ auf die Menge $V \times V$ erweitern. Zudem definieren wir für beliebige Multimengen $F \subseteq V \times V$ die Kosten von F als $k(F) = \sum_{e \in F} k(e)$ (d.h. jede Kante $e \in F$ wird bei der Berechnung von $k(F)$ entsprechend der Häufigkeit ihres Vorkommens in F berücksichtigt). Wir nennen F ***negativ***, falls F negative Kosten $k(F) < 0$ hat.

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

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

das Minimum der Kosten aller Flüsse der Größe $|f|$.

Lemma 3.25. *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 Kreis K mit Kosten $k(K) < 0$ gibt, dann können wir den Fluss durch alle Kanten $e \in K$ um 1 erhöhen. Dies führt auf einen Fluss g mit $|g| = |f|$ und $k(g) = k(f) + k(K) < k(f)$.

Sei umgekehrt g ein Fluss in N mit $|g| = |f|$ und $k(g) < k(f)$. Dann ist $g - f$ wegen $g(e) - f(e) \leq c(e) - f(e)$ ein Fluss in N_f. Da $g - f$ die Größe $|g - f| = 0$ hat, können wir $g - f$ als Summe von Flüssen $h_1, \ldots, h_k$ in N_f darstellen, wobei h_i nur für Kanten e auf einem Kreis K_i in N_f einen positiven Wert $h_i(e) = w_i > 0$ annimmt (siehe nächsten Abschnitt). Da $k(h_1) + \cdots + k(h_k) = k(g - f) = k(g) - k(f) < 0$ ist, muss wegen $k(h_i) = \sum_{e \in K_i} h_i(e)k(e) = w_i k(K_i)$ mindestens ein Kreis K_i negativ sein.

Um h_i und die zugehörigen Kreise K_i für $i = 1, \ldots, k$ zu finden, wählen wir eine beliebige Kante $e_{i,1}$ aus E_f, für die der Fluss $h'_{i-1} = g - f - h_1 - \cdots - h_{i-1}$ einen minimalen positiven Wert $w = h'_{i-1}(e_{i,1}) > 0$ annimmt (falls es keine solche Kante $e_{i,1}$ gibt, sind

*Natürlich kann man diese Einschränkung bspw. dadurch umgehen, dass man die Kante $e = (u, v)$ durch einen Pfad (u, w, v) über einen neuen Knoten w ersetzt.

wir fertig, weil dann h'_{i-1} der triviale Fluss ist). Da h'_{i-1} den Wert 0 hat und somit die Kontinuitätsbedingung für alle Knoten (inklusive s und t) erfüllt, lässt sich nun zu jeder Kante $e_{i,j} = (a, b) \in E_f$ solange eine Fortsetzung $e_{i,j+1} = (b, c) \in E_f$ mit $h'_{i-1}(e_{i,j+1}) > 0$ (und damit $h'_{i-1}(e_{i,j+1}) \geq w$) finden bis sich ein Kreis K_i schließt. Nun setzen wir $h_i(e_{i,j}) = w_i$ für alle Kanten $e_{i,j} \in K_i$, wobei $w_i = \min\{h'_{i-1}(e) \mid e \in K_i\}$ ist.

Da sich die Anzahl der Kanten in E_f, die unter dem verbleibenden Fluss $h'_i = g - f - h_1 - \cdots - h_i$ einen Wert ungleich 0 haben, gegenüber h'_{i-1} mindestens um 1 verringert, ist die Anzahl der Kreise K_i durch $\|E_f\| \leq 2m$ beschränkt. ■

Mithilfe von Lemma 3.25 lässt sich ein maximaler Fluss mit minimalen Kosten wie folgt berechnen. Wir berechnen zuerst einen maximalen Fluss f. Dann suchen wir beginnend mit $i = 1$ und $f_0 = f$ einen negativen Kreis K_i in $N_{f_{i-1}}$. Hierzu kann der Bellman-Ford-Moore Algorithmus benutzt werden, wenn wir zu $N_{f_{i-1}}$ einen neuen Knoten s' hinzufügen und diesen mit allen Knoten u durch eine neue Kante (s', u) verbinden.

Falls kein negativer Kreis existiert, ist f_{i-1} ein maximaler Fluss mit minimalen Kosten. Andernfalls bilden wir den Fluss f_i, indem wir zu f_{i-1} den Fluss f_{K_i} addieren, der auf jeder Kante $e \in K_i$ den Wert $f_{K_i}(e) = c_{f_{i-1}}(K_i) = \min\{c_{f_{i-1}}(e) \mid e \in K_i\}$ hat. Da sich die Kosten $k(f_i) = k(f_{i-1}) + k(f_{K_i}) = k(f_{i-1}) + c_{f_{i-1}}(K_i)k(K_i)$ von f_i wegen $k(K_i) \leq -1$ bei jeder Iteration um mindestens 1 verringern und die Kostendifferenz zwischen zwei beliebigen Flüssen durch $D = \sum_{u \in V} |k(s, u)|(c(s, u) + c(u, s))$ beschränkt ist, liegt nach $k \leq D$ Iterationen ein kostenminimaler Fluss f_k vor.

Der nächste Satz bereitet den Weg für einen Algorithmus zur Bestimmung eines kostenminimalen Flusses, dessen Laufzeit nicht von D, sondern von der Größe $M = |f|$ eines maximalen Flusses f in N abhängt. Voraussetzung hierfür ist jedoch, dass es in N keine negativen Kreise gibt.

Lemma 3.26. *Ist f_{i-1} ein Fluss in N mit $k(f_{i-1}) = k_{\min}(f_{i-1})$ und ist P_i ein Zunahmepfad in $N_{f_{i-1}}$ mit*

$$k(P_i) = \min\{k(P') \mid P' \text{ ist ein Zunahmepfad in } N_{f_{i-1}}\},$$

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

Beweis. Angenommen, es gibt einen Fluss g in N mit $|g| = |f_i|$ und $k(g) < k(f_i)$. Dann gibt es in N_{f_i} einen negativen Kreis K mit $k(K) < 0$. Wir benutzen K, um einen Zunahmepfad P' mit $k(f_{P'}) < k(f_{P_i})$ zu konstruieren.

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

Da jede Kante $e \in \hat{F} = K \setminus E_{f_{i-1}}$ wegen $f_{i-1}(e) = c(e)$ zwar von f_{i-1} aber wegen $e \in K \subseteq E_{f_i}$ nicht von f_i gesättigt wird, muss $f_{i-1}(e) \neq f_i(e)$ und somit $e^R \in P_i$ sein, was $\hat{F} \subseteq P_i^R$ impliziert. Somit ist jede Kante $e \in \hat{F}$ und mit ihr auch e^R genau einmal in F enthalten. Entfernen wir nun für jede Kante $e \in \hat{F}$ die beiden Kanten e und e^R aus F, so erhalten wir die Multimenge $F' = F \setminus (\hat{F} \cup \hat{F}^R)$, die wegen $k(e) + k(e^R) = 0$ dieselben Kosten $k(F') = k(F) < k(P_i)$ wie F hat.

Zudem gilt $F' \subseteq E_{f_{i-1}}$. Da F' aus F durch Entfernen von Kreisen (der Länge 2) entsteht, ist auch F' ein Multigraph, der sich in einen s-t-Pfad P' und eine gewisse Anzahl von Kreisen $K_1, \ldots, K_\ell$ in $N_{f_{i-1}}$ zerlegen lässt. Da nach Voraussetzung keine negativen Kreise in $N_{f_{i-1}}$ existieren, folgt

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

Basierend auf Lemma 3.26 können wir nun leicht einen Algorithmus zur Bestimmung eines maximalen Flusses mit minimalen Kosten in einem Netzwerk N angeben, falls es in N keine negativen Kreise gibt.

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

```
1 for all (u,v) ∈ V × V do
2     f(u,v) := 0
3 repeat
4     P ← min-zunahmepfad(f)
5     if P ≠ ⊥ then add(f,P)
6 until P = ⊥
```

Hierbei berechnet die Prozedur `min-zunahmepfad`(f) einen Zunahmepfad in N_f, der minimale Kosten unter allen Zunahmepfaden in N_f hat. Da es in N_f keine negativen Kreise gibt, kann hierzu bspw. der Bellman-Ford-Moore Algorithmus benutzt 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.

Satz 3.27. *In einem Netzwerk N kann ein maximaler Fluss f mit minimalen Kosten in Zeit $O(|f|mn)$ bestimmt werden, falls es in N keine negativen Kreise bzgl. der Kostenfunktion k gibt.*

Tatsächlich lässt sich für Netzwerke ohne negative Kreise die Laufzeit unter Verwendung des Dijkstra-Algorithmus in Kombination mit einer Preisfunktion auf $O(Mm \log n)$ verbessern.

Definition 3.28. *Sei $G = (V, E)$ ein Digraph mit Kostenfunktion $k : E \to \mathbb{Z}$. Eine Funktion $p : V \to \mathbb{Z}$ heißt* ***Preisfunktion*** *für (G, k), falls für jede Kante $e = (x, y)$ in E die Ungleichung*

$$k(x, y) + p(x) - p(y) \geq 0$$

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

$$k^p(x, y) = k(x, y) + p(x) - p(y).$$

Lemma 3.29. *Ein Digraph $G = (V, E)$ mit Kostenfunktion $k : E \to \mathbb{Z}$ hat genau dann keine negativen Kreise, wenn es eine Preisfunktion p für (G, k) 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 $k(K) = k^p(K) \geq 0$.

Sei nun G ein Digraph und sei $k : E \to \mathbb{Z}$ eine Kostenfunktion ohne negativen Kreise. Betrachte den Digraphen G', der aus G durch Hinzunahme eines neuen Knotens s und Kanten (s, x) für alle $x \in V$ entsteht. Zudem erweitern wir k mittels $k'(s, x) = 0$ zu einer Kostenfunktion k' auf G'. Da es auch in (G', k') keine negativen Kreise gibt, existiert in G' für jeden Knoten $x \in V$ ein bzgl. k' kürzester Pfad von s nach x, dessen Länge wir mit $d^{k'}(s, x)$ bezeichnen. Da nun für jede Kante $e = (x, y) \in E$ die Ungleichung

$$d^{k'}(s, x) + k(x, y) \geq d^{k'}(s, y)$$

gilt, ist $p(x) = d^{k'}(s, x)$ die gesuchte Preisfunktion. Diese lässt sich mit BFM in Zeit $O(nm)$ finden. ■

Sobald wir eine Preisfunktion p für das Restnetzwerk N_f haben, können wir Dijkstra zur Berechnung eines bzgl. k^p kürzesten Zunahmepfades P in N_f benutzen. P ist dann auch ein kürzester Pfad bzgl. k, da für jeden s-t-Pfad P die Beziehung $k^p(P) = k(P) + p(s) - p(t)$ gilt und $p(s) - p(t)$ eine von P unabhängige Konstante ist.

Falls N keine negativen Kreise hat, können wir für $N = N_{f_0}$ eine Preisfunktion $p_0(x) = \min\{k(P) \mid P \text{ ist ein } s\text{-}x\text{-Pfad}\}$ mit dem BFM-Algorithmus in Zeit $O(nm)$ berechnen. Angenommen, wir haben für ein $i \geq 1$ einen Fluss f_{i-1} mit minimalen Kosten $k(f_{i-1}) = k_{min}(f_{i-1})$ und eine Preisfunktion p_{i-1} für $(N_{f_{i-1}}, k)$. Sofern in $N_{f_{i-1}}$ ein Zunahmepfad existiert, können wir mit dem Dijkstra-Algorithmus in Zeit

$O(m \log n)$ einen bzgl. $k^{p_{i-1}}$ kürzesten Zunahmepfad P_i berechnen und erhalten einen größeren Fluss $f_i = f_{i-1} + f_{P_i}$ mit minimalen Kosten $k(f_i) = k_{min}(f_i)$. Andernfalls ist f_{i-1} ein maximaler Fluss.

Es bleibt die Frage, wie wir im Fall, dass P_i existiert, eine Preisfunktion p_i für N_{f_i} finden können, ohne erneut BFM zu benutzen.

Lemma 3.30. *Sei $d_i(s,x)$ die minimale Pfadlänge von s nach x in $N_{f_{i-1}}$ bzgl. $k^{p_{i-1}}$, wobei $p_{i-1} : V \to \mathbb{Z}$ eine beliebige Funktion ist. Dann ist $p_i(x) = p_{i-1}(x) + d_i(s,x)$ eine Preisfunktion für k in $N_{f_{i-1}}$ und in N_{f_i}.*

Beweis. Wir zeigen zuerst, dass p_i eine Preisfunktion für $(N_{f_{i-1}}, k)$ ist. Für jede Kante $e = (x,y) \in E_{f_{i-1}}$ gilt nämlich $d_i(y) \leq d_i(x) + k^{p_{i-1}}(e)$ und $k^{p_{i-1}}(e) = k(e) + p_{i-1}(x) - p_{i-1}(y)$. Somit ist

$$\begin{aligned} k^{p_i}(e) &= k(e) + p_i(x) - p_i(y) \\ &= k(e) + p_{i-1}(x) + d_i(s,x) - p_{i-1}(y) - d_i(s,y) \\ &= k^{p_{i-1}}(e) + d_i(s,x) - d_i(s,y) \geq 0. \end{aligned}$$

Falls e auf P_i liegt, gilt sogar $k^{p_i}(e) = 0$, da P_i ein bzgl. $k^{p_{i-1}}$ kürzester s-t-Pfad in $N_{f_{i-1}}$ und daher $d_i(s,y) = d_i(s,x) + k^{p_{i-1}}(e)$ ist.

Da zudem für jede Kante e in N_{f_i}, die nicht zu $N_{f_{i-1}}$ gehört, die gespiegelte Kante e^R auf dem Pfad P_i liegt, folgt $k^{p_i}(e^R) = 0$ und somit $k^{p_i}(e) = k(e) + p_i(x) - p_i(y) = -k(e^R) - p_i(y) + p_i(x) = -k^{p_i}(e^R) = 0$. Dies zeigt, dass p_i eine Preisfunktion für (N_{f_i}, k) ist. ■

Satz 3.31. *In einem Netzwerk N kann ein maximaler Fluss f mit minimalen Kosten in Zeit $O(mn + |f| m \log n)$ bestimmt werden, falls es in N keine negativen Kreise bzgl. der Kostenfunktion k gibt.*

Beweis. Wir berechnen zuerst mit BFM in Zeit $O(nm)$ eine Preisfunktion p_0 für die Kostenfunktion k im Netzwerk $N = N_{f_0}$. Dann bestimmen wir in $\leq |f|$ Iterationen eine Folge von kostenminimalen Flüssen f_i, indem wir mit dem Dijkstra-Algorithmus in Zeit $O(m \log n)$ einen bzgl. $k^{p_{i-1}}$ kürzesten Zunahmepfad P_i in $N_{f_{i-1}}$ berechnen. Da hierbei bereits die Distanzen $d_i(x)$ für alle Knoten x berechnet werden können, erfordert die Bestimmung von p_i in jeder Iteration nur $O(n)$ Zeit. ■

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.*
- *Ein Knoten $v \in V$ heißt* ***gebunden****, falls v Endpunkt einer Matchingkante (also $v \in \bigcup M$) ist und sonst* ***frei****.*
- *M heißt* ***perfekt****, falls alle Knoten von G gebunden sind (also $V = \bigcup M$ ist).*
- *Die Matchingzahl von G ist*

$$\mu(G) = \max\{\|M\| \mid \ M \text{ ist ein Matching in } G\}$$

- *Ein Matching M heißt* ***maximal****, falls $\|M\| = \mu(G)$ ist. M heißt* ***gesättigt****, 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 G zu finden.

Durch eine einfache Reduktion des bipartiten Matchingproblems auf ein Flussproblem erhält man aus Korollar 3.24 das folgende Resultat (siehe Übungen).

Satz 4.2. *In einem bipartiten Graphen lässt sich ein maximales Matching in Zeit $O(m\sqrt{n})$ bestimmen.*

Beweis. Sei $G = (U, V, E)$ der gegebene bipartite Graph. Konstruiere das Netzwerk $N = (V', E', s, t, c)$ mit den Knoten $V' = U \cup V \cup \{s, t\}$ und den Kanten

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

die alle Kapazität 1 haben. Es ist leicht zu sehen, dass sich aus jedem Matching M in G ein Fluss f in N konstruieren lässt mit $\|M\| = |f|$ und umgekehrt. Es genügt also, einen maximalen Fluss in N zu finden.

Nach Korollar 3.24 ist dies mit dem Algorithmus von Dinic unter Einsatz von `blockfluss1` in $O(m\sqrt{n})$ Zeit möglich, da der Durchsatz aller Knoten außer s und t durch 1 beschränkt ist. ■

In den Übungen wird gezeigt, dass die Laufzeit durch eine verbesserte Analyse sogar durch $O(m\sqrt{\mu(G)})$ abgeschätzt werden kann.

Die Konstruktion aus Satz 4.2 lässt sich nicht ohne Weiteres auf allgemeine, nicht-bipartite Graphen verallgemeinern. Wir werden jedoch sehen, dass sich manche bei den Flussalgorithmen verwendete Ideen auch für Matchingalgorithmen einsetzen lassen.

Beispiel 4.3. *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 $M' = \{\{v, x\}, \{u, w\}\}$ größer 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.

Es gibt jedoch eine Methode, mit der sich jedes Matching, das nicht maximal ist, vergrößern lässt.

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

1. *Ein Pfad $P = (u_1, \ldots, u_k)$ heißt* ***alternierend****, falls für $i = 1, \ldots, k-1$ gilt:*

$$e_i = \{u_i, u_{i+1}\} \in M \Leftrightarrow e_{i+1} = \{u_{i+1}, u_{i+2}\} \in E \setminus M.$$

2. *Ein Kreis $C = (u_1, \ldots, u_k)$ heißt* ***alternierend****, falls der Pfad $P = (u_1, \ldots, u_{k-1})$ alternierend ist und zusätzlich gilt:*

$$e_1 \in M \Leftrightarrow e_{k-1} \in E \setminus M.$$

3. *Ein alternierender Pfad P heißt* ***vergrößernd****, falls weder e_1 noch e_{k-1} zu M gehören.*

Satz 4.5. *Ein Matching M in G ist genau dann maximal, wenn es keinen vergrößernden Pfad in G bzgl. M gibt.*

Beweis. Ist P ein vergrößernder Pfad, so liefert $M' = M \Delta P$ ein Matching der Größe $\|M'\| = \|M\| + 1$ in G. Hierbei identifizieren wir P mit der Menge $\{e_i \mid i = 1, \ldots, k-1\}$ der auf $P = (u_1, \ldots, u_k)$ liegenden Kanten $e_i = \{u_i, u_{i+1}\}$.

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

Damit haben wir das Problem, ein maximales Matching in einem Graphen G zu finden, auf das Problem reduziert, zu einem Matching M in G einen vergrößernden Pfad zu finden, sofern ein solcher existiert.

4.1 Der Algorithmus von Edmonds

Der Algorithmus von Edmonds bestimmt einen vergrößernden Pfad wie folgt. Jeder Knoten v hat einen von 3 Zuständen, welcher entweder mit gerade (falls v frei ist) oder unerreicht (falls v gebunden ist) initialisiert wird. Dann wird ausgehend von den freien Knoten als Wurzeln ein Suchwald W aufgebaut, indem für einen beliebigen geraden Knoten v eine Kante zu einem Knoten v' besucht wird, der entweder ebenfalls gerade oder unerreicht ist.

Ist v' unerreicht, so wird der aktuelle Suchwald W um die beiden Kanten (v, v') und $(v', M(v'))$ erweitert, wobei $M(v')$ der Matchingpartner von v' ist (d.h. $\{v', M(v')\} \in M$). Zudem wechselt der Zustand von v' von unerreicht zu ungerade und der von $M(v')$ von unerreicht zu gerade. Damit wird erreicht, dass jeder Knoten in W genau dann gerade (bzw. ungerade) ist, wenn der Abstand zu seiner Wurzel in W gerade (bzw. ungerade) ist.

Ist v' dagegen gerade, so gibt es zwei Unterfälle. Sind die beiden Wurzeln von v und v' verschieden, so wurde ein vergrößernder Pfad gefunden, der von der Wurzel von v zu v über v' zur Wurzel von v' verläuft.

Andernfalls befindet sich v' im gleichen Suchbaum wie v, d.h. es gibt einen gemeinsamen Vorfahren v'', so dass durch Verbinden der beiden Pfade von v'' nach v und von v'' nach v' zusammen mit der Kante $\{v, v'\}$ ein Kreis C entsteht. Da v und v' beide gerade sind, hat C eine ungerade Länge. Zudem muss auch v'' gerade sein, da jeder ungerade Knoten in W genau ein Kind hat. Der Pfad von der Wurzel von v'' zu v'' zusammen mit dem Kreis C wird als ***Blume*** mit der ***Blüte*** C bezeichnet. Der Knoten v'' heißt ***Basis*** der Blüte C.

Zwar führt das Auffinden einer Blüte C nicht direkt zu einem vergrößernden Pfad, sie bedeutet aber dennoch einen Fortschritt, da sich der Graph wie folgt vereinfachen lässt. Wir ***kontrahieren*** C zu einem einzelnen geraden Knoten b, der die Nachbarschaften aller Knoten in C zu Knoten außerhalb von C erbt, und setzen die Suche nach einem vergrößernden Pfad fort. Bezeichnen wir den aus G durch Kontraktion von C entstandenen Graphen mit G_C und das aus M durch Kontraktion von C entstandene Matching in G_C mit M_C, so stellt folgendes Lemma die Korrektheit dieser Vorgehensweise sicher.

Lemma 4.6. *In G lässt sich ausgehend von M genau dann ein vergrößernder Pfad finden, wenn dies in G_C ausgehend von M_C möglich ist. Zudem kann jeder vergrößernde Pfad in G_C zu einem vergrößernden Pfad in G expandiert werden.*

Beweis. Sei P ein vergrößernder Pfad in G_C. Falls P nicht den Knoten b besucht, zu dem die Blüte C kontrahiert wurde, so ist P auch ein vergrößernder Pfad in G. Besucht P dagegen den Knoten b, so betrachten wir die beiden Nachbarn a und c von b in P (o.B.d.A sei $\{a, b\}$ in M_C). Dann existiert in M eine Kante zwischen a und der Basis v'' von C. Zudem gibt es in C mindestens einen Nachbarn v_c von c. Im Fall $v'' = v_c$ genügt es, b durch v'' zu ersetzen. Andernfalls ersetzen wir b durch denjenigen der beiden Pfade P_1 und P_2 von v'' nach v_c auf C, der v_c über eine Matchingkante erreicht. Falls b Endknoten von P ist, also nur einen Nachbarn c in P hat, ersetzen wir b durch den gleichen Pfad.

Der Beweis der Rückrichtung ist komplizierter, da viele verschiedene Fälle möglich sind. Alternativ ergibt sich die Rückrichtung aber auch als Folgerung aus der Korrektheit des Edmonds-Algorithmus (siehe Satz 4.9). ■

Die folgende Prozedur `VergrößernderPfad` berechnet einen vergrößernden Pfad für G, falls das aktuelle Matching M nicht maximal ist. Da M nicht mehr als $n/2$ Kanten enthalten kann, wird diese Prozedur höchstens $(n/2 + 1)$-mal aufgerufen.

Prozedur `VergrößernderPfad`(G, M)

```
1  Q ← ∅
2  for v ∈ V(G) do
3    if ∃e ∈ M : v ∈ e then zustand(v) ← unerreicht
4    else
5      zustand(v) ← gerade
6      root(v) ← v
7      depth(v) ← 0
8      for u ∈ N(v) do Q ← Q ∪ {(v, u)}
9  while Q ≠ ∅ do
10   entferne eine Kante (v, v') aus Q
11   if inblüte(v) = inblüte(v') ≠ ⊥ then // tue nichts
12   else if zustand(v') = unerreicht then
13     parent(v') ← v
14     root(v') ← root(v)
15     depth(v') ← depth(v) + 1
16     if zustand(v) = gerade then
17       zustand(v') ← ungerade
18       Q ← Q ∪ {v',partner(v')}
19     else
20       zustand(v') ← gerade
21       for u ∈ N(v') \ {v} do Q ← Q ∪ {(v', u)}
22   else if zustand(v') = zustand(v) or inblüte(v) or
           inblüte(v') then
23     if root(v) =root(v') then // v und v' sind im
           gleichen Baum: kontrahiere Blüte
24       v'' ← tiefster gemeinsamer Vorfahr von v und v'
             // verwende depth(v) und depth(v')
25       b ← neuer Knoten
26       blüte(b) ← (v'', . . . , v, v', . . . , v'') // setze die
           beiden Pfade entlang der Baum-Kanten zu
           einem ungeraden Kreis zusammen
27       parent(b) ← v''
28       root(b) ← root(v'')
29       depth(b) ← depth(v'')+1
30       for u ∈ blüte(b)\{v''} do
31         inblüte(u) ← b
32         if zustand(u) = ungerade then
33           for w ∈ N(u) do Q ← Q ∪ {(u, w)}
34     else // vergrößernder Pfad gefunden, muss noch
           expandiert werden
```

```
35      P ← leere doppelt verkettete Liste
36      u ← v
37      while u ≠ ⊥ do
38        while inblüte(u) ≠ ⊥ do u ← inblüte(u)
39        hänge u vorne an P an
40        u ← parent(u)
41      u ← v'
42      while u ≠ ⊥ do
43        while inblüte(u) do u ← inblüte(u)
44        hänge u hinten an P an
45        u ← parent(u)
46      u ← der erste Knoten auf P
47      while u ≠ ⊥ do
48        if blüte(u) = ⊥ then
49          u ← succ_P(u)
50        else // blüte(u) = (v_0, ..., v_k) mit v_0 = v_k
51          ersetze u in P durch den alternierenden
          Pfad in blüte(u), der pred_P(u) und
          succ_P(u) verbindet und auf der Nicht-
          Basis-Seite mit einer Kante aus M endet
52          u ← der erste Knoten dieses Pfads
53      return P
```

Für den Beweis der Korrektheit des Edmonds-Algorithmus benötigen wir den Begriff des OSC.

Definition 4.7. *Sei $G = (V, E)$ ein Graph. Eine Menge $S = \{v_1, \ldots, v_k, V_1, \ldots, V_\ell\}$ von Knoten $v_1, \ldots, v_k \in V$ und Teilmengen $V_1, \ldots, V_\ell \subseteq V$ heißt* ***OSC*** *(engl. odd set cover) in G, falls*

1. $\forall e \in E : e \cap V_0 \neq \emptyset \vee \exists i \geq 1 : e \subseteq V_i$, wobei $V_0 = \{v_1, \ldots, v_k\}$.

2. $\forall i \geq 1 : n_i \equiv_2 1$, wobei $n_i = \|V_i\|$.

Das ***Gewicht*** *von S ist weight$(S) = k + \sum_{i=1}^{\ell}(n_i - 1)/2$. Im Fall $\ell = 0$ nennen wir V_0 auch* ***Knotenüberdeckung*** *(oder kurz* ***VC*** *für engl. vertex cover) in G.*

Lemma 4.8. *Für jedes Matching M in einem Graphen $G = (V, E)$ und jedes OSC S in G gilt $\|M\| \leq$ weight(S).*

Beweis. M kann für jeden Knoten $v_j \in S$ höchstens eine Kante und von den Kanten in V_i, $i \geq 1$, höchstens $(n_i - 1)/2$ Kanten enthalten. ■

Satz 4.9. *Der Algorithmus von Edmonds berechnet ein maximales Matching M für G.*

Beweis. Es ist klar, dass der Algorithmus von Edmonds terminiert. Wir analysieren die Struktur des Suchwalds zu diesem Zeitpunkt. Jede Kante $e \in E$ lässt sich in genau eine von drei Kategorien einteilen:

1. e hat mindestens einen ungeraden Endpunkt,
2. beide Endpunkte von e sind unerreicht,
3. e liegt komplett innerhalb einer Blüte.

Würde nämlich e keine dieser 3 Bedingungen erfüllen, so würde der Algorithmus nicht terminieren, da alle Kanten $e = (v, v')$, die mindestens einen geraden Endpunkt v haben, von dem Algorithmus betrachtet werden und somit v' nicht gerade oder unerreicht sein kann, da

1. im Fall, dass auch v' gerade ist, e entweder zur Kontraktion einer weiteren Blüte oder zu einem vergrößernden Pfad führen würde, und
2. im Fall, dass v' unerreicht ist, v' in einen ungeraden Knoten verwandelt würde.

Folglich können wir ein OSC S wie folgt konstruieren. Sei U die Menge der unerreichten Knoten. Jede Blüte bildet eine Menge V_i in S und jeder ungerade Knoten wird als Einzelknoten zu S hinzugefügt. Falls U nicht leer ist, fügen wir einen beliebigen unerreichten Knoten $u_0 \in U$ als Einzelknoten zu S hinzu. Falls U mindestens 4 Knoten enthält, fügen wir auch die Menge $U \setminus \{u_0\}$ zu S hinzu.

Nun ist leicht zu sehen, dass S alle Kanten überdeckt und jeder Einzelknoten in S mit einer Matchingkante inzident. Da zudem jede Blüte V_i der Größe n_i genau $(n_i - 1)/2$ (und auch die Menge $U \setminus \{u_0\}$ im Fall $\|U\| \geq 4$ genau $(\|U\| - 2)/2$) Matchingkanten enthält, folgt $\mathit{weight}(S) = \|M\|$. ■

Korollar 4.10. *Für jeden Graphen G gilt*

$$\mu(G) = \min\{\mathit{weight}(S) \mid S \text{ ist ein OSC in } G\}.$$

Ein Spezialfall hiervon ist der Satz von König für bipartite Graphen (siehe Übungen).

Der Algorithmus von Edmonds lässt sich leicht dahingehend modifizieren, dass er nicht nur ein maximales Matching M, sondern auch ein OSC S ausgibt, das die Optimalität von M beweist. In den Übungen werden wir noch eine weitere Möglichkeit zur „Zertifizierung“ der Optimalität von M kennenlernen.

Lemma 4.11. *Die Prozedur* `VergrößernderPfad` *benötigt $O(m)$ Zeit; der Algorithmus von Edmonds hat damit eine Gesamtlaufzeit von $O(nm)$.*

Beweis. Wir können annehmen, dass G keinen isolierten Knoten hat, da sich diese in Zeit $O(n)$ entfernen lassen. Wir zeigen, dass die Prozedur `VergrößernderPfad` in Zeit $O(m)$ läuft. Da die Prozedur höchstens $n/2$-mal aufgerufen wird und die Laufzeit außerhalb von `VergrößernderPfad` durch $O(n^2)$ beschränkt ist, ergibt sich somit eine Gesamtlaufzeit von $O(nm)$ (genauer $O(nm+n)$). Dass jeder Aufruf von `VergrößernderPfad` nach $O(m)$ Schritten terminiert, liegt daran, dass die Initialisierung $O(n+m) = O(m)$ Schritte benötigt und danach für jede Kante $e \in E$ nur $O(1)$ Schritte ausgeführt werden:

1. Für jede Kante $e = \{u, v\} \in E$ wird jede der beiden Orientierungen (u, v) und (v, u) von e maximal einmal zu Q hinzugefügt.
2. Außerdem ist jede Kante maximal einmal an der Kontraktion einer Blüte beteiligt, und folglich auch höchstens einmal an der Expansion einer Blüte. ■

4.2 Effiziente Implementierung von Edmonds' Algorithmus

Micali und Vazirani haben gezeigt, dass eine Variante von Edmonds' Algorithmus sogar mit $O(m\sqrt{\mu})$ Zeit auskommt. Der Ansatz ist ähnlich wie beim Algorithmus von Dinic: Pro Runde wird nicht nur ein einzelner vergrößernder Pfad zum Matching hinzugefügt, sondern eine maximale Menge knotendisjunkter vergrößernder Pfade, die minimale Länge (unter allen vergrößernden Pfaden) haben.

Hopcroft und Karp haben mit den folgenden Lemmata gezeigt, dass $O(\sqrt{\mu})$ solcher Runden ausreichen.

Lemma 4.12. *Sei M ein Matching in einem Graphen G, sei P ein kürzester vergrößernder Pfad bezüglich M, und sei P' ein vergrößernder Pfad bezüglich $M \Delta P$. Dann ist $\|P'\| \geq \|P\| + \|P \cap P'\|$, wobei die Kardinalität sich hier auf die Anzahl der Kanten bezieht.*

Beweis. Sei $M' = (M \Delta P) \Delta P'$ das resultierende Matching. Betrachte den Graphen $H = M \Delta M' = P \Delta P'$. In H hat jeder Knoten höchstens den Grad 2: Sofern P' einen Knoten v aus P enthält, einer der P-Nachbarn von v auch ein P'-Nachbar von v sein muss, da P' ein alternierender Pfad bezüglich $M \Delta P$ ist. Die Zusammenhangskomponenten von H sind also in Pfade und Kreise. Wegen $M' = M \Delta H$ müssen diese Pfade und Kreise alternierend bezüglich M sein. Wegen $\|M'\| = \|M\| + 2$ enthält H mindestens zwei disjunkte vergrößernde Pfade P_1 und P_2 für M.

Nun gilt: $\|H\| \geq \|P_1\| + \|P_2\| \geq 2\|P\|$, da P ein kürzester vergrößernder Pfad bezüglich M ist. Zusammen mit $\|H\| = \|P \Delta P'\| =$

$\|P\| + \|P'\| - \|P \cap P'\|$ ergibt dies die Behauptung. ■

Wenn im Algorithmus von Edmonds nun ausgehend vom leeren Matching $M_0 = \emptyset$ vergrößernde Pfade P_i minimaler Länge bezüglich M_{i-1} gefunden werden um die Matchings $M_i = M_{i-1}\Delta P_i$ zu erhalten, gilt folglich $\|P_i\| \leq \|P_{i+1}\|$.

Lemma 4.13. *Wenn $\|P_i\| = \|P_j\|$ für $i < j$ gilt, so sind P_i und P_j knotendisjunkt.*

Beweis. Angenommen P_i und P_j wären nicht knotendisjunkt. Wir können o.B.d.A. annehmen, dass kein k existiert mit $i < k < j$ (also $\|P_k\| = \|P_i\|$), für das P_k nicht knotendisjunkt zu P_i ist. Dann ist P_j ein vergrößernder Pfad bezüglich $M_i = M_{i-1}\Delta P_i$. Mit Lemma 4.12 folgt $\|P_j\| \geq \|P_i\| + \|P_i \cap P_j\|$. Wegen $\|P_i\| = \|P_j\|$ sind P_i und P_j damit kantendisjunkt. Wenn P_j nun einen Knoten v aus P_i enthalten würde, müsste P_j auch den M_i-Nachbarn von v enthalten. Da dieser aber auch ein Nachbar von v auf P_i ist, ergibt sich ein Widerspruch dazu, dass P_i und P_j kantendisjunkt sind. ■

Satz 4.14. *Sei G ein Graph mit Matchingzahl μ und sei $P_1, \ldots, P_\mu$ die Folge der kürzesten Zunahmepfade. Dann gilt*

$$\left\|\left\{\|P_i\| \;\middle|\; 1 \leq i \leq \mu\right\}\right\| \leq 2\lfloor\sqrt{\mu}\rfloor + 1.$$

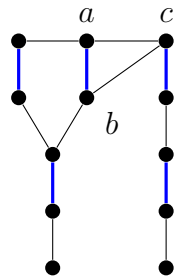
Beweis. Sei $r = \mu - \left\lfloor\sqrt{\mu}\right\rfloor$. Dann gilt $\|M_r\| = r$. Der Graph $H = M_r\Delta M_\mu$ hat als Zusammenhangskomponenten (unter anderen) $\mu - r = \left\lfloor\sqrt{\mu}\right\rfloor$ knotendisjunkte vergrößernde Pfade bezüglich M_r. Da M_r nur r Kanten enthält, enthält mindestens einer dieser vergrößernden Pfade höchstens $\left\lfloor r / \left\lfloor\sqrt{\mu}\right\rfloor\right\rfloor \leq \left\lfloor\sqrt{\mu}\right\rfloor$ Kanten aus M_r; der Pfad P_r ist damit höchstens $2\left\lfloor\sqrt{\mu}\right\rfloor + 1$ lang. Da die Länge eines vergrößernden Pfads immer eine ungerade natürliche Zahl ist, folgt

$$\left\|\left\{\|P_i\| \;\middle|\; 1 \leq i \leq r\right\}\right\| \leq \lfloor\sqrt{\mu}\rfloor + 1.$$

Da $P_{r+1}, \ldots, P_\mu$ höchstens $\mu - r = \left\lfloor\sqrt{\mu}\right\rfloor$ weitere Längen beisteuern, folgt die Behauptung. ■

Es bleibt also zu zeigen, wie eine *maximale Menge* vergrößernden Pfade *minimaler Länge* in $O(m)$ Zeit gefunden werden kann. Gegenüber der Prozedur `VergrößernderPfad` sind dazu eine Reihe von Änderungen nötig:

- Um nur vergrößernde Pfade minimaler Länge zu finden, muss die Reihenfolge in der die Kanten betrachtet werden angepasst werden. Dass dies notwendig ist ergibt sich aus folgendem Beispiel:



  Hier kann es passieren, dass der vergrößernde Pfad über die Kante $\{a, c\}$ mit Länge 11 vor dem über die Kante $\{b, c\}$ der Länge 9 gefunden wird.

  Abhilfe schafft das folgende Vorgehen: Es werden nicht alle Kanten gleichberechtigt zur Menge Q hinzugefügt und in einer beliebigen Reihenfolge entnommen. Vielmehr wird eine Breitensuche durchgeführt (d.h. Q wird als Warteschlange implementiert), damit kürzere Pfade zuerst gefunden werden. Das alleine genügt aber noch nicht, um das Problem aus dem vorhergehenden Beispiel zuverlässig zu vermeiden. Deshalb werden für jeden Knoten die folgenden Werte gespeichert und aktualisiert:

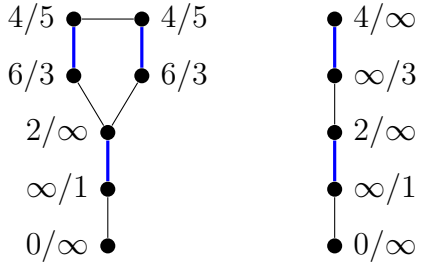
  `evenlevel`(u): Die Länge des kürzesten alternierenden Pfades gerader Länge von u zu einem freien Knoten. Initial 0 für freie Knoten und ∞ für alle übrigen.

  `oddlevel`(u): Die Länge des kürzesten alternierenden Pfades un-

gerader Länge von u zu einem freien Knoten. Initial ∞.

`level`(u): Das Minimum von `evenlevel`(u) und `oddlevel`(u).

Während der Breitensuche erhalten die als `gerade` markierten Knoten endliches `evenlevel` und die als `ungerade` markierten Knoten endliches `oddlevel`. Beim Kontrahieren einer Blüte erhalten die in der Blüte enthaltenen als `gerade` markierten Knoten endliches `oddlevel` und die in der Blüte enthaltenen als `ungerade` markierten Knoten endliches `evenlevel`. Im folgenden Beispiel sind die Knoten jeweils mit `evenlevel`/`oddlevel` beschriftet.



Eine Kante e wird *Brücke* genannt, wenn e eine Matchingkante ist, deren Endpunkte beide endliches `evenlevel` haben, oder wenn e eine Nicht-Matchingkante ist, deren Endpunkte beide endliches `oddlevel` haben. Damit sind Brücken gerade die Kanten, die im Algorithmus von Edmonds zur Erkennung von Blüten oder vergrößernden Pfaden führen. Einer Brücke $\{u, v\}$ wird ihre *Zähigkeit* zugeordnet:

$$\texttt{tenacity}(\{u,v\}) = \begin{cases} \texttt{oddlevel}(u) + \texttt{oddlevel}(v) + 1 & \text{falls } \{u,v\} \in M \\ \texttt{evenlevel}(u) + \texttt{evenlevel}(v) + 1 & \text{sonst} \end{cases}$$

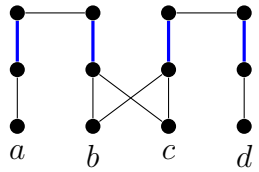
Wenn eine Brücke dazu führt, dass ein vergrößernder Pfad gefunden wird, entspricht ihre Zähigkeit damit gerade der Länge des vergrößernden Pfades.

Wenn der Algorithmus eine Brücke findet, reiht er diese nicht einfach in die Warteschlange Q ein, sondern sammelt sie getrennt nach Zähigkeit. Brücken mit Zähigkeit $2i + 1$ (die Zähigkeit ist immer ungerade) werden zu dem Zeitpunkt behandelt (d.h. der zugehörige vergrößernde Pfad ermittelt beziehungsweise die zugehörige Blüte kontrahiert), an dem die Breitensuche alle Knoten der Schicht i gefunden hat. Im obigen Beispiel hat die Kante $\{a, c\}$ die Zähigkeit 9 und wird damit nach Schicht 4 behandelt, während die Kante $\{b, c\}$ Zähigkeit 11 hat und damit erst nach Schicht 5 an die Reihe kommt.

Damit ist sichergestellt, dass der Algorithmus kürzere vergrößernde Pfade zuerst findet.

- Um die Zeitschranke $O(m)$ für das finden einer maximalen Menge von kürzesten vergrößernden Pfaden einzuhalten, kann die Suche nicht nach jedem gefundenen Pfad neu gestartet werden. Stattdessen werden alle Knoten (und die zu ihnen inzidenten Kanten) gelöscht, die auf dem gefundenen vergrößernder Pfad liegen.

  Das folgende Beispiel zeigt, dass weitere Änderungen nötig sind, damit der Algorithmus tatsächlich eine maximale Menge kürzester vergrößernder Pfade des ursprünglichen Graphen findet.



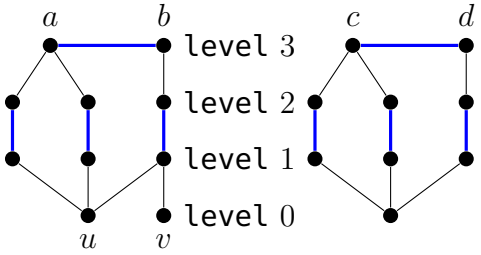
In diesem Graphen gibt es zwei disjunkte vergrößernde Pfade (von a nach b und von c nach d). Allerdings wird von der Breitensuche entweder der Knoten b oder der Knoten c zuerst betrachtet, und die zum anderen Knoten inzidenten Kanten tauchen nicht im Breitensuchbaum auf. Nach dem Löschen des ersten vergrößernden Pfads kann deshalb kein weiterer vergrößernder Pfad gefunden werden, der nur Brücken- und Suchbaumkanten verwendet.

Um dieses Problem zu umgehen, wird für jeden Knoten nicht nur sein Elternknoten im Suchbaum gespeichert, sondern eine Menge von *Vorgängern*:

$$\texttt{predecessors}(u) = \begin{cases} \left\{ v \;\middle|\; \{u,v\} \in M \right\} & \text{falls } u \texttt{ gerade} \\ \left\{ v \;\middle|\; \begin{array}{l} \{u,v\} \in E \setminus M \wedge \\ \texttt{evenlevel}(v) + 1 = \texttt{oddlevel}(u) \end{array} \right\} & \text{sonst} \end{cases}$$

Anstelle des Suchwalds mit seinen `parent`-Kanten erhalten wir so einen geschichteten Graphen mit (gerichteten) `predecessor`-Kanten. Ein Knoten w wird *Vorfahr* von u genannt, wenn er von u aus entlang solcher `predecessor`-Kanten erreichbar ist.

Nach dieser Änderung ist es natürlich nicht mehr möglich für jeden von der Breitensuche erreichten Knoten zu speichern, was die Wurzel seines Suchbaums ist. Einer Brücke kann der Algorithmus deshalb nicht mehr ohne Weiteres anzusehen, ob sie zu einem vergrößernden Pfad führt. Im folgenden Beispiel führt die Brücke $\{a, b\}$ zu einem vergrößernden Pfad von u nach v, während es keinen vergrößernden Pfad durch die Brücke $\{c, d\}$ gibt.



Da der Algorithmus weiterhin genau dann eine Blüte finden soll, wenn er ausgehend von einer Brücke keinen vergrößernden Pfad finden kann, ergibt sich die folgende verallgemeinerte Blütendefinition: Eine Brücke $\{u, v\}$ schließt eine Blüte, wenn es einen Knoten w gibt der sowohl für u als auch für v der einzige Vorfahr auf der Schicht $\texttt{level}(w)$ ist; wir können annehmen, dass w unter allen solchen Knoten das größte `level` hat. Die Blüte besteht aus u und v sowie allen ihren Vorgängern, die keine Vorgänger von w sind. Der Knoten w heißt *Basis* der Blüte.

Mit dieser Definition ist leicht zu sehen, dass jede Brücke $\{u, v\}$ entweder eine Blüte schließt oder zu einem vergrößernden Pfad führt. Es bleibt die Frage, wie der Algorithmus diese Blüte beziehungsweise diesen Pfad anhand der `predecessor`-Kanten effizient finden kann. Dies gelingt mit einer simultanen Tiefensuche. Hierbei werden zwei disjunkte Tiefensuchbäume aufgebaut, einer von u aus und einer von v aus, die nur `predecessor`-Kanten verwenden. Der nächste Tiefensuchschritt wird immer in dem Suchbaum durchgeführt, dessen aktueller Knoten die größere Schicht hat; liegen beide in der gleichen Schicht, wird der erste Suchbaum bevorzugt. Erreichen beide Suchbäume die Schicht 0, ist ein vergrößernder Pfad gefunden. Stößt der zweite Suchbaum auf einen Knoten w, der bereits zum ersten Suchbaum gehört, versucht er (durch Backtracking) einen weiteren Vorgänger von v auf dieser Schicht zu finden. Ist dies nicht möglich, wird der Knoten w an den Suchbaum von v übergeben und (mit Backtracking) versucht, einen weiteren Vorfahren von u auf dieser Schicht zu finden. Gelingt auch das nicht, ist w Basis einer Blüte, die alle in den beiden Suchbäumen enthaltenen Knoten umfasst.

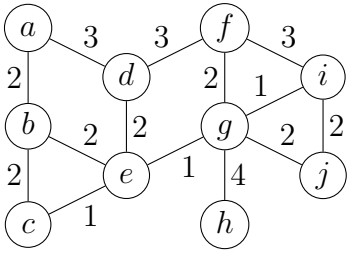
Um sicherzustellen, dass die simultane Tiefensuche nie steckenbleibt ohne einen vergrößernden Pfad oder eine Blüte zu finden, können alle gebundenen Knoten (zusammen mit ihren inzidenten Kanten) gelöscht werden, deren Vorgängermenge durch Löschen von Knoten leer werden. Später gefundene vergrößernde Pfade gleicher Länge können die gelöschten Knoten nicht verwenden, da sie außer `predecessor`-Kanten nur eine Brückenkante enthalten können, die noch nicht Teil einer Blüte ist, und nach Lemma 4.13 knotendisjunkt zum gerade gelöschten Pfad sind.

Um den Aufwand aller durchgeführten simultanen Tiefensuchen durch $O(m)$ abzuschätzen, sind folgende Beobachtungen nötig. Wird eine Blüte gefunden, sind alle während der Suche besuchten Kanten in der Blüte enthalten; da jede Kante nur einmal Teil einer Blüte ist, ergibt sich konstanter Aufwand pro Kante. Und wenn ein vergrößernder Pfad gefunden, entsteht nur konstanter Aufwand pro gelöschter Kante, weil die Tiefensuchen nur dann in Sackgassen geraten, wenn die (eigentlich vorhandenen) `predecessor`-Kanten durch den jeweils anderen Suchbaum blockiert sind.

4.3 Gewichtete Matchings

Beim *Gewichteten Matchingproblem* wird für einen gegebenen Graphen $G = (V, E)$ mit Kantengewichten $w\colon E \to \mathbb{Z}$ ein maximales Matching M von G mit minimalem Gewicht $w(M) = \sum_{e\in M} w(e)$ gesucht.

Eine Anwendung ist das *Chinese Postman Problem*, bei dem in einem Graphen $G = (V, E)$ mit Kostenfunktion $c\colon E \to \mathbb{N}$ eine Tour $T = (v_0, \ldots, v_k)$ gesucht wird, die jede Kante mindestens einmal durchläuft und minimale Kosten $c(T) = \sum_{i=1}^{k} c(\{v_{i-1}, v_i\})$ hat.



Der folgende Algorithmus reduziert das Chinese Postman Problem auf das Gewichtete Matchingproblem:

Prozedur `ChinesePostman`(V, E, c)

1 $U := \{v \in V \mid \deg(v) \equiv_2 1\}$ `// Knoten mit ungeradem Grad`
2 $H := (U, \binom{U}{2})$ `// vollständiger Graph auf` U
3 `Definiere` $w\colon \binom{U}{2} \to \mathbb{Z}$ `durch` $w(\{u, v\}) := d_{G,c}(u, v)$ `//` $d_{G,c}$`: Entfernung in` G `bezüglich` c
4 $M :=$ `GewichtetesMatching`(H, w)
5 **for** $\{u, v\} \in M$ **do**
6 `Finde einen kürzesten` u`-`v`-Pfad in` G `und füge eine neue Kopie aller seiner Kanten ein`
7 $T :=$ `Euler-Tour im so entstandenen Multigraphen`
8 **return** T

Um die Korrektheit der Reduktion zu zeigen, nehmen wir an, dass es eine Tour T' gäbe, die ebenfalls alle Kanten von G mindestens einmal durchläuft und die günstiger als T ist, d.h. $c(T') < c(T)$. Bezeichne die Multimenge der durch T (beziehungsweise T') wiederholt durchlaufenen Kanten mit E_T (beziehungsweise $E_{T'}$). Es gilt $c(E_T) - c(E_{T'}) = c(T) - c(T') > 0$. Die Kanten in $E_{T'}$ lassen sich in Pfade zerlegen, die jeden Knoten in U einmal als Endknoten haben (zusätzlich könnte es noch Kreise geben). Diese Pfade definieren damit ein perfektes Matching M' in H, das Gewicht $w(M') \leq c(E_{T'}) < -c(E_T) = w(M)$ hat, was im Widerspruch dazu steht, dass M minimales Gewicht hat.

Das gewichtete Matchingproblem im allgemeinen Fall kann mit Techniken der *Linearen Programmierung* gelöst werden, die aber den Rahmen dieser Vorlesung sprengen würden. Für bipartite Graphen können wir es jedoch auf die Berechnung eines kostenoptimalen Flusses reduzieren.

Um das gewichtete Matchingproblem in einem bipartiten Graphen $G = (U, W, E)$ auf die Berechnung eines kostenminimalen maximalen Flusses in einem azyklischen Netzwerk $N(G)$ zu reduzieren, fügen wir zwei neue Knoten s und t hinzu und verbinden s mit allen Knoten

$u \in U$ durch eine neue Kante (s, u) sowie alle Knoten $w \in W$ durch eine neue Kante (w, t) mit t. Alle Kanten in E werden von U nach W gerichtet und haben die vorgegebenen Kosten/Gewichte. Alle neue Kanten e haben die Kosten $k(e) = 0$ und alle Kanten e in $N(G)$ haben die Kapazität $c(e) = 1$. Dann entspricht jedem Fluss f in $N(G)$ genau ein Matching M von G mit $M = \{\{u, w\} \in U \times W \mid f(u, w) = 1\}$ (und umgekehrt entspricht jedem Matching M genau ein Fluss f mit dieser Eigenschaft).

Da die maximale Flussgröße M in $N(G)$ durch $n/2$ beschränkt ist, erhalten wir einen $O(mn \log n)$ Algorithmus für das gewichtete Matchingproblem in bipartiten Graphen. Da $N(G)$ kreisfrei ist, können wir hierbei beliebige Kantengewichte zulassen.

Korollar 4.15. *In einem bipartiten Graphen $G = (V, E)$ lässt sich ein maximales Matching mit minimalen Kosten in Zeit $O(\mu(G)m \log n)$ berechnen.*

Beweis. Wir transformieren G in das zugehörige Netzwerk $N = N(G)$. Da N eine sehr spezielle Form hat, lässt sich eine Preisfunktion p_0 für (N, k) in Linearzeit bestimmen. Dann berechnen wir in höchstens $\mu(G)$ Iterationen, die jeweils Zeit $O(m \log n)$ beanspruchen, einen kostenminimalen maximalen Fluss f in N. Aus diesem lässt sich ein Matching M_f in G gewinnen, das wegen $\|M_f\| = |f|$ maximal und wegen $k(M_f) = k(f)$ kostenminimal ist. Die beiden Transformationen von G in N und von f in M_f benötigen nur Linearzeit. ■

Tatsächlich leistet der Algorithmus von Korollar 4.15 noch mehr. Er berechnet für jede Zahl i mit $1 \leq i \leq \mu(G)$ ein Matching M_i der Größe i, das minimale Kosten unter allen Matchings dieser Größe hat, und eine zu M_i kompatible Preisfunktion p_{i-1} (siehe Übungen). Dabei heißt eine Preisfunktion p ***kompatibel*** zu einem Matching M in G, falls die reduzierten Kosten von allen Kanten $e = (u, w) \in U \times W$ mit $\{u, w\} \in E$ einen nichtnegativen Wert $k^p(e) \geq 0$ und alle Kanten $e = (u, w) \in U \times W$ mit $\{u, w\} \in M$ den Wert $k^p(e) = 0$ haben.

5 Färben von Graphen

Definition 5.1. *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.*

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

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

Beispiel 5.2.

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

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

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 5.3. *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 5.4. $n/\alpha(G) \leq \chi(G) \leq n - \alpha(G) + 1$.

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 5.5. $\binom{\chi(G)}{2} \leq m$.

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

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

Um die zweite Ungleichung zu erhalten, betrachte 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}
```

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

Satz 5.7 (Brooks 1941 (vereinfachter Beweis von Lovász, 1975))**.** *Sei G ein Graph mit $\Delta(G) \geq 3$. Dann gilt $\chi(G) = \Delta(G) + 1$ nur dann, wenn $K_{\Delta(G)+1}$ ein Teilgraph von G ist.*

Beweis. Wir führen Induktion über n. Für $n \leq 4$ gibt es genau 3 Graphen G mit $\Delta(G) \geq 3$. Diese erfüllen die Behauptung.

Sein nun G ein Graph mit $n > 4$ Knoten und Maximalgrad $d = \Delta(G) \geq 3$, der K_{d+1} nicht als Teilgraph enthält. Wir können annehmen, dass G zusammenhängend ist.

Falls es in G einen Knoten u mit $\deg(u) < d$ gibt, dann ist $G - u$ nach IV d-färbbar und somit auch G.

Es bleibt der Fall, dass alle Knoten u den Grad d haben. Da $G \neq K_{d+1}$ ist, folgt $n \geq d + 2$. Falls G einen Schnittknoten s hat, d.h. in $G - s$ gibt es $k \geq 2$ Komponenten $G_1, \ldots G_k$, folgt nach IV $\chi(G_i) \leq d$ und somit auch $\chi(G) \leq d$.

Behauptung 5.8. *In G gibt es einen Knoten u, der zwei Nachbarn a und b mit $\{a, b\} \notin E$ hat, so dass $G - \{a, b\}$ zusammenhängend ist.*

Da G den K_{d+1} nicht als Teilgraph enthält, hat jeder Knoten u zwei Nachbarn $v, w \in N(u)$ mit $\{v, w\} \notin E$. Falls $G - v$ 2-fach zusammenhängend ist, ist $G - \{v, w\}$ zusammenhängend und die Behauptung folgt.

Falls $G - v$ nicht 2-fach zusammenhängend ist, hat $G - v$ mindestens zwei 2-fach-Zusammenhangskomponenten (Blöcke) $B_1, \ldots, B_\ell$ der Blockbaum T hat mindestens zwei Blätter B_i, B_j. Da $\kappa(G) \geq 2$ ist, ist v in G zu mindestens einem Knoten in jedem Blatt B von T benachbart, der kein Schnittknoten ist. Wählen wir für a und b zwei dieser Knoten, so ist $G - \{a, b\}$ zusammenhängend und somit die Behauptung bewiesen.

Sei also u ein Knoten, der zwei Nachbarn a und b mit $\{a,b\} \notin E$ hat, so dass $G - \{a,b\}$ zusammenhängend ist. Wir wenden auf den Graphen $G - \{a,b\}$ eine Tiefensuche an mit Startknoten $u_1 = u$. Sei $(u_1, \ldots, u_{n-2})$ die Reihenfolge, in der die Knoten besucht werden. Nun lassen wir `greedy-color` mit der Reihenfolge $(a, b, u_{n-2}, \ldots, u_1)$ laufen.

Behauptung 5.9. `greedy-color` *benutzt* $\leq d$ *Farben.*

Die Knoten a und b erhalten die Farbe $c(a) = c(b) = 1$. Jeder Knoten u_i, $i > 1$, ist mit einem Knoten u_j mit $j < i$ verbunden. Daher ist seine Farbe $c(u_i) \leq \deg(u_i) \leq d$. Da $u = u_1$ bereits zwei Nachbarn a und b mit derselben Farbe hat, folgt auch $c(u) \leq d$. ■

In den Übungen wird folgendes Korollar gezeigt:

Korollar 5.10. *Es gibt einen Linearzeitalgorithmus, der alle Graphen G mit $\Delta(G) \leq 3$ mit $\chi(G)$ Farben färbt.*

5.1 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 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.

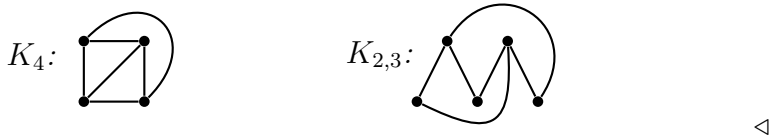
Die Vermutung, dass 4 Farben ausreichen, wurde 1878 von Kempe „bewiesen“ und erst 1890 entdeckte Heawood einen Fehler in Kempes „Beweis“. Übrig blieb der *5-Farben-Satz*. Der *4-Farben-Satz* wurde erst 1976 von Appel und Haken bewiesen. Hierbei handelt es sich jedoch nicht um einen Beweis im klassischen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

Satz 5.11 (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 auch nicht ohne Computer-Unterstützung verifizierbar ist.

Beispiel 5.12. *Wie die folgenden Einbettungen von K_4 und $K_{2,3}$ in die Ebene zeigen, sind K_4 und $K_{2,3}$ planar.*



◁

Um eine Antwort auf die Frage zu finden, ob auch K_5 und $K_{3,3}$ planar sind, betrachten wir die Gebiete von in die Ebene eingebetteten Graphen.

Durch die Kanten eines eingebetteten Graphen wird die Ebene in so genannte ***Gebiete*** unterteilt. Nur eines dieser Gebiete ist unbeschränkt und dieses wird als ***äußeres Gebiet*** bezeichnet. Die Anzahl der Gebiete von G bezeichnen wir mit $r(G)$ oder kurz mit r. Der ***Rand*** $rand(g)$ eines Gebiets g ist die (zirkuläre) Folge aller Kanten,

die an g grenzen, wobei jede Kante so durchlaufen wird, dass g „in Fahrtrichtung links“ liegt bzw. bei Erreichen eines Knotens über eine Kante e, u über die im Uhrzeigersinn nächste Kante e' wieder verlassen wird. Die Anzahl der an ein Gebiet g grenzenden Kanten bezeichnen wir mit $d(g)$, wobei Kanten, die nur an g und an kein anderes Gebiet grenzen, doppelt gezählt werden.

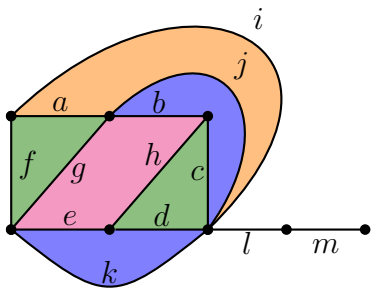
Die Gesamtzahl $\sum_g d(g)$ aller Inzidenzen von Gebieten und Kanten bezeichnen wir mit $i(G)$. Da jede Kante genau 2 Inzidenzen zu dieser Summe beiträgt, folgt

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

Ein ***ebener Graph*** wird durch das Tripel $G = (V, E, R)$ beschrieben, wobei R aus den Rändern aller Gebiete von G besteht. Wir nennen G auch ***ebene Realisierung*** des Graphen (V, E). Durch R ist für jeden Knoten u die (zirkuläre) Ordnung π auf allen mit u inzidenten Kanten eindeutig festgelegt (und umgekehrt). Man nennt π das zu G gehörige ***Rotationssystem***. Dieses kann bei Verwendung der Adjazenzlistendarstellung ohne zusätzlichen Platzaufwand gespeichert werden, indem man die zu u adjazenten Knoten gemäß π anordnet.

Beispiel 5.13. *Nebenstehender ebener Graph hat 13 Kanten $a, \ldots, m$ und 7 Gebiete mit den 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. ◁

Satz 5.14 (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 $n-1$ Knoten, $m-1$ Kanten und r Gebieten. Nach IV folgt $(n-1)-(m-1)+r = 2$, d.h. $(*)$ ist erfüllt.

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

Korollar 5.15. *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 5.16. *K_5 ist nicht planar.*

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

Korollar 5.17. *$K_{3,3}$ ist nicht planar.*

Beweis. Wegen $n = 6$, also $2n - 4 = 8$, und wegen $m = 3 \cdot 3 = 9$ gilt $m \not\leq 2n - 4$. ■

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

Lemma 5.18. *Jeder planare Graph hat einen Minimalgrad $\delta(G) \leq 5$.*

Beweis. Für $n \leq 6$ ist die Behauptung klar. Für $n > 6$ impliziert die Annahme $\delta(G) \geq 6$ 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 5.19. *Sei $G = (V, E)$ ein Graph und seien $u, v \in V$. Dann entsteht der Graph $G_{uv} = (V - \{v\}, E')$ mit*

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

durch ***Fusion*** *von u und 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. G heißt zu H* ***kontrahierbar****, falls H aus einer isomorphen Kopie von G durch eine Folge von Kontraktionen gewonnen werden kann.*

Satz 5.20 (Kempe 1878, Heawood 1890)**.**
Jeder planare Graph ist 5-färbbar.

Beweis. Wir beweisen den Satz durch Induktion über n.

$n = 1$**:** Klar.

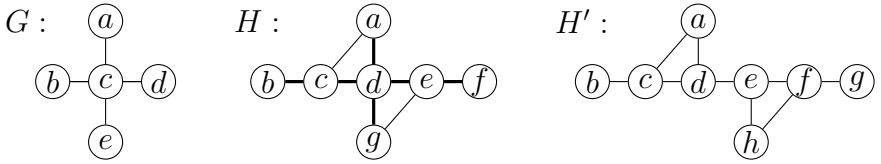
$n - 1 \rightsquigarrow n$**:** Da G planar ist, existiert ein Knoten u mit $\deg(u) \leq 5$. Zunächst entfernen wir u aus G. Falls u fünf Nachbarn hat, existieren zwei Nachbarn v und w, die nicht durch eine Kante verbunden sind, und wir fusionieren diese zu v.

Der resultierende Graph G' ist planar und hat $n' \leq n - 1$ Knoten. Daher existiert nach IV eine 5-Färbung c' für G'. Da wir nun w mit $c'(v)$ färben können und somit die Nachbarn von u höchstens 4 verschiedene Farben haben, ist G 5-färbbar. ■

Definition 5.21. *Seien $G = (V, E)$ ein Graph, $v \in V$ und $e \in \binom{V}{2}$.*

- *Durch Entfernen des Knotens v entsteht der Graph $G[V - \{v\}]$ aus G, den wir mit $\boldsymbol{G - v}$ bezeichnen.*
- *Den Graphen $(V, E - \{e\})$ bezeichnen wir mit $\boldsymbol{G - e}$ und den Graphen $(V, E \cup \{e\})$ mit $\boldsymbol{G \cup e}$.*
- *Hat v den Grad 2 und sind u und w die beiden Nachbarn von v, so entsteht der Graph $G' = (G - v) \cup \{u, w\}$ durch* ***Überbrückung*** *von v aus G.*
- *H heißt* ***Unterteilung*** *von G, wenn G durch sukzessive Überbrückungen aus einer isomorphen Kopie von H entsteht.*

Beispiel 5.22. *Betrachte folgende Graphen.*



Offensichtlich ist H keine Unterteilung von G. Entfernen wir jedoch die beiden dünnen Kanten aus H, so ist der resultierende Teilgraph eine Unterteilung von G. Dagegen ist kein Teilgraph von H' eine Unterteilung von G. ◁

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

Definition 5.23. *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, oder*
- *$B - K$ eine Zusammenhangskomponente 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 ein Graph G genau dann planar ist, wenn sich die Brücken jedes Kreises K von G in höchstens zwei Mengen partitionieren lassen, so dass jede Menge nur kompatible Brücken enthält.

Satz 5.24 (Kuratowski 1930)**.**
Für einen Graphen G sind folgende Aussagen äquivalent:

- *G ist planar.*
- *Keine Unterteilung des $K_{3,3}$ oder des K_5 ist ein Teilgraph von G.*

Beweis. Wenn eine Unterteilung G' des $K_{3,3}$ oder des K_5 ein Teilgraph von G ist, so ist G' und folglich auch G nicht planar.

Sei nun $G = (V, E)$ nicht planar. Durch Entfernen von Knoten und Kanten erhalten wir einen 3-zusammenhängenden nicht planaren Teilgraphen $G' = (V', E')$, so dass $G' - e'$ für jede Kante $e' \in E'$ planar ist (siehe Übungen). Wir entfernen eine beliebige Kante $e_0 = \{a_0, b_0\}$ aus G'. Da $G' - e_0$ 2-zusammenhängend ist, gibt es einen Kreis durch die beiden Knoten a_0 und b_0 in $G' - e_0$. Sei H' eine ebene Realisierung von $G' - e_0$ und sei K ein Kreis durch die beiden Knoten a_0 und b_0. Dabei wählen wir H' und K so, dass es keine ebene Realisierung H'' von $G' - e_0$ gibt, in der ein Kreis durch a_0 und b_0 existiert, der in H'' mehr Gebiete als K in H' einschließt.

Dann ist e_0 eine Brücke von K in G'. Die übrigen Brücken von K in G' sind auch Brücken von K in H'. Die Kanten jeder solchen Brücke B verlaufen entweder alle innerhalb oder alle außerhalb von K in H'. Im ersten Fall nennen wir B eine ***innere Brücke*** und im zweiten eine ***äußere Brücke***.

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.

Behauptung 5.25. *Jede äußere Brücke B besteht aus einer Kante, die einen Knoten in $K(a_0, b_0)$ mit einem Knoten in $K(b_0, a_0)$ verbindet.*

Zum Beweis der Behauptung nehmen wir an, dass B mindestens 3 Kontaktpunkte oder mindestens einen Kontaktpunkt in $\{a_0, b_0\}$ hat. Dann liegen mindestens zwei dieser Punkte auf $K[a_0, b_0]$ oder auf $K[b_0, a_0]$. Folglich kann K zu einem Kreis K' erweitert werden, der

mehr Gebiete einschließt (bzw. ausschließt) als K, was der Wahl von K und H' widerspricht.

Nun wählen wir eine innere Brücke B^*, die sowohl zu e_0 als auch zu einer äußeren Brücke $\hat{B}$ 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.

Sei $\hat{B} = \{a_1, b_1\}$. Da e_0 und $\hat{B}$ inkompatibel sind, können wir annehmen, dass diese vier Knoten in der Reihenfolge a_0, a_1, b_0, b_1 auf K liegen. Wir zeigen nun, dass G' eine Unterteilung des $K_{3,3}$ oder des K_5 als Teilgraph enthält. Hierzu geben wir entweder zwei disjunkte Mengen $A, B \subseteq V'$ mit jeweils 3 Knoten an, so dass 9 knotendisjunkte Pfade zwischen allen Knoten $a \in A$ und $b \in B$ existieren. Oder wir geben fünf Knoten an, zwischen denen 10 knotendisjunkte Pfade existieren.

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 $\hat{B}$ 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: $\exists k \in \{k_2, k_3\} \cap K(b_0, b_1)$. In diesem Fall existieren 9 knotendisjunkte Pfade zwischen $\{a_0, a_1, k\}$ und $\{b_0, b_1, k_1\}$.

Fall 1b: $\{k_2, k_3\} \cap K(b_0, b_1) = \emptyset$. 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: B^* hat nur Kontaktpunkte $k \in \{a_0, a_1, b_0, b_1\}$. In diesem Fall müssen alle vier Punkte zu B^* gehören (denn B^* ist inkompatibel zu B und $\hat{B}$) und es gibt in B^* einen a_0-b_0-Pfad P_0 sowie einen a_1-b_1-Pfad P_1.

Fall 2a: P_0 und P_1 haben nur einen Knoten u gemeinsam. Dann gibt es in B^* vier knotendisjunkte Pfade von u zu $\{a_0, a_1, b_0, b_1\}$ und somit 10 knotendisjunkte Pfade zwischen den Knoten u, a_0, a_1, b_0, b_1.

Fall 2b: P_0 und P_1 haben mindestens zwei Knoten gemeinsam. Seien u der erste und v der letzte Knoten auf P_0, die auch auf P_1 liegen. Dann gibt es in B^* drei knotendisjunkte Pfade zwischen u und allen Knoten in $\{v, a_0, a_1\}$ und zwei zwischen v und allen Knoten in $\{b_0, b_1\}$. Folglich gibt es 9 knotendisjunkte Pfade zwischen $\{a_0, a_1, v\}$ und $\{b_0, b_1, u\}$. ■

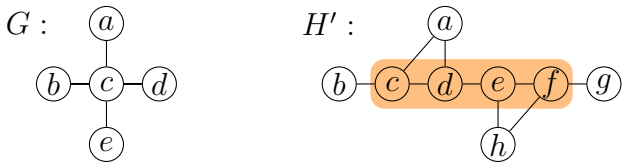
Definition 5.26. *Seien G, H Graphen. H heißt **Minor** von G, wenn sich H aus einem zu G isomorphen Graphen durch wiederholte Anwendung folgender Operationen gewinnen lässt:*

- *Entfernen von Kanten,*
- *Entfernen von isolierten Knoten und*
- *Kontraktion von Kanten.*

*G heißt **H-frei**, falls H kein Minor von G ist. Für eine Menge $\mathcal{H}$ von Graphen heißt G **$\mathcal{H}$-frei**, falls G für alle $H \in \mathcal{H}$ H-frei ist.*

Da die Kantenkontraktionen zuletzt ausgeführt werden können, ist H genau dann ein Minor von G, wenn ein Teilgraph von G zu H kontrahierbar ist. Zudem ist leicht zu sehen, dass G und H genau dann Minoren voneinander sind, wenn sie isomorph sind.

Beispiel 5.27. *Wir betrachten nochmals die Graphen G und H'.*

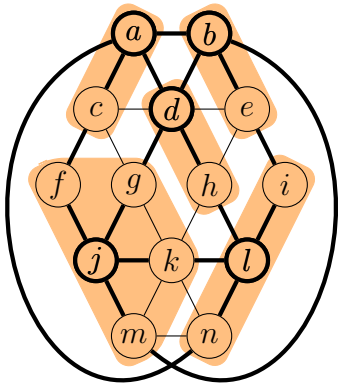


G ist ein Minor von H', da durch Fusion der Knoten c, d, e, f ein zu G isomorpher Graph aus H' entsteht. ◁

Wagner beobachtete, dass sich aus dem Satz von Kuratowski folgende Charakterisierung der Klasse der planaren Graphen ableiten lässt (siehe Übungen).

Korollar 5.28 (Wagner 1937)**.** *Ein Graph ist genau dann planar, wenn er $\{K_{3,3}, K_5\}$-frei ist.*

Beispiel 5.29. *Betrachte folgenden Graphen.*



Durch Entfernen der dünnen Kanten entsteht eine Unterteilung des K_5. Aus dieser erhalten wir den K_5, indem wir alle dünn umrandeten Knoten (also alle Knoten vom Grad 2) überbrücken.

Alternativ lässt sich der K_5 auch durch Fusion aller Knoten in den farblich unterlegten Umgebungen der dick umrandeten Knoten gewinnen. ◁

Definition 5.30. *Sei $\leq$ eine binäre Relation auf einer Menge A.*

a) $(A, \leq)$ *heißt* ***Quasiordnung****, wenn $\leq$ reflexiv und transitiv auf A ist.*

b) $(A, \leq)$ *heißt* ***Wohlquasiordnung****, wenn es zudem zu jeder Folge $a_1, a_2, \ldots$ von Elementen aus A Indizes $i < j$ mit $a_i \leq a_j$ gibt.*

Beispiele für Quasiordnungen sind der Betrag auf komplexen Zahlen und die Erreichbarkeit in gerichteten Graphen.

$(\mathbb{N}, \leq)$ ist eine Wohlquasiordnung. Dagegen ist $(\mathbb{Z}, \leq)$ keine Wohlquasiordnung, da unendliche absteigende Ketten $a_1 > a_2 > \cdots$ existieren, z.B. $a_i = -i$. Auch die Teilbarkeitsrelation auf den natürlichen Zahlen ist keine Wohlquasiordnung, da mit der Folge der Primzahlen eine unendliche Antikette existiert (d.h. die Glieder der Folge sind paarweise unvergleichbar).

Proposition 5.31. *Eine Quasiordnung $(A, \leq)$ ist genau dann eine Wohlquasiordnung, wenn es in $(A, \leq)$ weder unendliche absteigende Ketten noch unendliche Antiketten gibt.*

Beweis. Es ist klar, dass $(A, \leq)$ keine Wohlquasiordnung ist, wenn es eine unendliche absteigende Kette oder eine unendliche Antikette gibt.

Wenn umgekehrt weder unendliche absteigende Ketten noch unendliche Antikette existieren, so können wir in jeder Folge $a_1, a_2, \ldots$ alle Elemente a_j streichen, für die ein $i < j$ existiert, so dass $a_i > a_j$ ist. Da hierbei von jeder absteigenden Kette ein Element in der Folge verbleibt und alle diese Ketten endlich sind, enthält die verbleibende Folge immer noch unendlich viele Elemente.

Als nächstes streichen wir alle Elemente a_j, für die ein $i < j$ existiert, so dass a_i und a_j unvergleichbar sind. Die verbleibende Folge ist dann immer noch unendlich und sogar monoton, d.h. es gilt $a_i \leq a_{i+1}$ für alle i. ■

Proposition 5.32. *In einer Wohlquasiordnung $(A, \leq)$ hat jede Teilmenge $B \subseteq A$ bis auf Äquivalenz nur endlich viele minimale Elemente. Dabei heißen $a, b \in A$* ***äquivalent****, falls $a \leq b$ und $b \leq a$ gilt.*

Satz 5.33 (Satz von Robertson und Seymour, 1983-2004)**.** *Die Minorenrelation bildet auf der Menge aller endlichen ungerichteten Graphen eine Wohlquasiordnung.*

Korollar 5.34. *Sei $\mathcal{K}$ eine Graphklasse, die unter Minorenbildung abgeschlossen ist (d.h. wenn $G \in \mathcal{K}$ und H ein Minor von G ist, dann folgt $H \in \mathcal{K}$). 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 $\mathcal{K}$. Für den Beweis des Korollars betrachten wir die komplementäre Klasse $\overline{\mathcal{K}}$ aller endlichen Graphen, die nicht zu $\mathcal{K}$ gehören. Nach Satz 5.33 in Kombination mit Proposition 5.32 hat $\overline{\mathcal{K}}$ bis auf Isomorphie nur endlich viele minimale Elemente. Da mit H auch jeder Graph G, der H als Minor enthält, zu $\overline{\mathcal{K}}$ gehört, gibt es demnach eine endliche Menge $\mathcal{H}$ von Graphen mit

$$\overline{\mathcal{K}} = \{G \mid \exists H \in \mathcal{H} : H \text{ ist ein Minor von } G\},$$

womit Korollar 5.34 bewiesen ist.

Das Problem, für zwei gegebene Graphen G und H zu entscheiden, ob H ein Minor von G ist, ist zwar NP-vollständig. Für einen festen Graphen H ist das Problem dagegen effizient entscheidbar.

Satz 5.35 (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 5.36. *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 Theorem 5.33 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.

5.2 Färben von chordalen Graphen

Definition 5.37. *Ein Graph $G = (V, E)$ heißt **chordal**, wenn er keinen induzierten Kreis der Länge ≥ 4 enthält.*

Ein induzierter Kreis $G[\{u_1, \ldots, u_k\}]$ enthält also nur die Kreiskanten $\{u_1, u_2\}, \ldots, \{u_{k-1}, u_k\}, \{u_k, u_1\}$, aber keine ***Sehnen*** $\{u_i, u_j\}$ mit $i - j \not\equiv_k \pm 1$.

Definition 5.38. *Sei G ein Graph. Eine Menge $S \subseteq V$ heißt **Separator** von G, wenn $G - S$ mehr Komponenten als G hat.*

Lemma 5.39. *Für einen Graphen G sind folgende Aussagen äquivalent.*

(i) G ist chordal.

(ii) Jeder inklusionsminimale Separator von G ist eine Clique.

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

Beweis. Sei G chordal und sei S ein minimaler Separator von G. Dann hat $G - S$ mindestens zwei Komponenten $G[V_1]$ und $G[V_2]$. Angenommen, S enthält zwei nicht adjazente Knoten x und y. Da S minimal ist, sind beide Knoten sowohl mit $G[V_1]$ als auch mit $G[V_2]$ verbunden. Betrachte die beiden Teilgraphen $G_i = G[V_i \cup \{x, y\}]$ und wähle jeweils einen kürzesten x-y-Pfad P_i in G_i. Da diese eine Länge ≥ 2 haben, bilden sie zusammen einen Kreis $K = P_1 \cup P_2$ der Länge ≥ 4. Aufgrund der Konstruktion von K ist klar, dass K keine Sehne in G hat. Dies zeigt, dass die erste Aussage die zweite impliziert.

Dass die zweite die dritte impliziert, ist klar. Um zu zeigen, dass die erste 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 minimaler x-y-Separator in G. Dann muss S mindestens zwei nicht adjazente Knoten aus K enthalten. ■

Definition 5.40. *Sei $G = (V, E)$ ein Graph und sei $k \geq 0$. Ein Knoten $u \in V$ heißt **k-simplizial** in G, wenn die Nachbarschaft $N(u)$ eine Clique der Größe k in G bildet. 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 5.41. *Sei $G = (V, E)$ ein Graph. Eine lineare Ordnung $(v_1, \ldots, v_n)$ auf V heißt **perfekte Eliminationsordnung** von G, wenn v_i simplizial in $G[\{v_1, \ldots, v_i\}]$ für $i = 1, \ldots, n$ ist.*

Lemma 5.42. *Jeder nicht vollständige chordale Graph $G = (V, E)$ besitzt mindestens zwei simpliziale Knoten, die nicht durch eine Kante verbunden sind.*

Beweis. Wir führen Induktion über n. Für $n \leq 2$ ist die Behauptung klar. Sei G ein zusammenhängender Graph mit $n \geq 3$ Knoten. Falls G nicht vollständig ist, enthält G zwei nichtadjazente Knoten x_1 und x_2. Sei S ein minimaler x_1-x_2-Separator und seien $G[V_1]$ und $G[V_2]$ die beiden Komponenten von $G - S$ mit $x_i \in V_i$. Nach Lemma 5.39 ist S eine Clique in G. Betrachte die Teilgraphen $G_i = G[V_i \cup S]$. Da G_i chordal ist und weniger als n Knoten hat, ist $V_i \cup S$ entweder eine Clique oder G_i enthält mindestens zwei nicht adjazente simpliziale Knoten y_i, z_i, wovon höchstens einer zu S gehört. Da im zweiten Fall y_i oder z_i in V_i ist, ist mindestens einer der drei Knoten x_i, y_i und z_i ohne Nachbarn in $G[V_{3-i}]$ und somit auch simplizial in G. ■

Satz 5.43. *Ein Graph ist genau dann chordal, wenn er eine perfekte Eliminationsordnung hat.*

Beweis. Falls G chordal ist, lässt sich eine perfekte Eliminationsordnung gemäß Lemma 5.42 bestimmen, indem wir beginnend mit $i = n$ sukzessive einen simplizialen Knoten v_i in $G[V - \{v_{i+1}, \ldots, v_n\}]$ wählen.

Für die umgekehrte Richtung sei $(v_1, \ldots, v_n)$ eine perfekte Eliminationsordnung von G. Wir zeigen induktiv, dass $G_i = G[\{v_1, \ldots, v_i\}]$ chordal ist. Da v_{i+1} simplizial in G_{i+1} ist, enthält jeder Kreis K der Länge ≥ 4 in G_{i+1}, auf dem v_{i+1} liegt, eine Sehne zwischen den beiden Kreisnachbarn von v_{i+1}. Daher ist mit G_i auch G_{i+1} chordal. ■

Korollar 5.44. *Es gibt einen Polynomialzeitalgorithmus A, der für einen gegebenen Graphen eine perfekte Eliminationsordnung berechnet falls G chordal ist und andernfalls einen induzierten Kreis der Länge ≥ 4 findet.*

Beweis. A versucht wie im Beweis von Theorem 5.43 beschrieben, eine perfekte Eliminationsordnung zu bestimmen. Stellt sich heraus, dass $G_i = G[V - \{v_{i+1}, \ldots, v_n\}]$ keinen simplizialen Knoten v_i hat, so ist G_i wegen Lemma 5.42 nicht chordal. Folglich gibt es wegen Lemma 5.39 in G_i zwei nicht adjazente Knoten x und y, so dass kein minimaler x-y-Separator S eine Clique ist. Wie im Beweis von Lemma 5.39 beschrieben, lässt sich mithilfe von S ein induzierter 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 perfekte Eliminationsordnung kann verwendet werden, um einen chordalen Graphen zu färben:

Algorithmus `chordal-color`(V, E)

```
1 berechne eine PEO (v_1, ..., v_n) für G = (V, E)
2 greedy-color(v_n, ..., v_1)
```

Lemma 5.45. *Für einen gegebenen chordalen Graphen $G = (V, E)$ berechnet der Algorithmus* `chordal-color` *eine k-Färbung von G mit $k = \chi(G) = \omega(G)$.*

Beweis. Sei $(v_1, \ldots, v_n)$ eine perfekten Eliminationsordnung von G und sei v_i ein beliebiger Knoten mit $f(v_i) = k$. Die Nachbarn von v_i in der Menge $\{v_{i+1}, \ldots, v_n\}$ bilden eine Clique, da v_i simplizial in $G[\{v_{i+1}, \ldots, v_n\}]$ ist. Wegen $f(v_i) = k$ bilden sie zusammen mit v_i eine k-Clique. Es folgt $k = \chi(G) = \omega(G)$. ■

Um `chordal-color` effizient zu implementieren, benötigen wir einen möglichst effizienten Algorithmus zur Bestimmung einer perfekten Eliminationsordnung. Rose, Tarjan und Lueker haben hierfür 1976 einen Linearzeitalgorithmus angegeben, der auf *lexikographischer Breitensuche* (kurz LexBFS oder LBFS) basiert. Der Unterschied zur normalen Breitensuche besteht darin, dass die Warteschlange Q nicht einzelne Knoten, sondern Knotenmengen enthält, welche die Menge der noch nicht besuchten Knoten partitionieren. Diese Partition wird vom Algorithmus wiederholt verfeinert. Der Name von LexBFS rührt daher, dass die Knoten in einer Reihenfolge besucht werden, die auch bei einer gewöhnlichen Breitensuche auftreten kann, bei dieser aber nicht garantiert ist, weil die Nachbarn eines Knoten in beliebiger Reihenfolge zur Warteschlange hinzugefügt werden. Die Zeilen der Adjazenzmatrix sind lexikographisch sortiert, wenn die Einträge auf der Hauptdiagonalen auf 1 gesetzt und die Zeilen und Spalten in der Reihenfolge angeordnet werden, in der `LexBFS` sie findet, wenn jeweils unter allen Kandidatenknoten einer mit maximalem Grad gewählt wird (siehe Übungen).

Algorithmus `LexBFS`(V, E)

```
1  Q ← (V) // doppelt verkettete Liste von Mengen
2  result ← () // leere Liste
3  while L ≠ ∅ do
4    wähle v ∈ first(L)
5    first(L) ← first(L)\{v}
6    if first(L)= ∅ then entferne den ersten Eintrag
         von L
7    append(result, v)
8    for S in L with N(v) ∩ S ≠ ∅ do
9      ersetze (S) in L durch (S ∩ N(v), S \ N(v))
10 return result
```

Um diesen Algorithmus effizient zu implementieren, kann die innere `for`-Schleife durch eine Schleife über die Nachbarn von v ersetzt werden, wenn die Knotenmengen in Q durch verkettete Listen realisiert werden und für jeden Knoten ein Zeiger auf die Menge die ihn enthält und auf seinen Eintrag in dieser Menge gespeichert wird.

Lemma 5.46. *Sei $G = (V, E)$ ein chordaler Graph und sei $(v_n, \ldots, v_1)$ die durch* `LexBFS`(V, E) *gefundene Knotenreihenfolge. Dann ist $(v_1, \ldots, v_n)$ eine perfekte Eliminationsordnung für G.*

Beweis. Wir führen den Beweis mittels Induktion über n. Für $n = 1$ ist die Aussage trivialerweise erfüllt.

Für $n > 1$ werden wir zeigen, dass der zuletzt von `LexBFS` besuchte Knoten v_1 simplizial in G ist. Dies ist ausreichend, da $G - v_1$ wieder chordal ist und `LexBFS` bei Eingabe $G - v_1$ die Folge $(v_n, \ldots, v_2)$ berechnet, deren Umkehrung nach Induktionsvoraussetzung eine perfekte Eliminationsordnung für $G - v_1$ ist. Wenn v_1 simplizial in G ist, folgt daraus, dass $(v_1, v_2, \ldots, v_n)$ eine perfekte Eliminationsordnung für G ist.

Wir werden unter der Annahme, dass v_1 nicht simplizial ist, eine

unendliche Folge von aufsteigenden Knotenindizes $i_0 < i_1 < i_2 < \cdots$ konstruieren, sodass

(a) für $j < k$ gilt: $\{v_{i_j}, v_{i_k}\} \in E \Leftrightarrow j + 2 = k \vee (j,k) = (0,1)$ und

(b) für $j \geq 2$ gilt: $\nexists i'_j > i_j : v_{i'_j} \in N(v_{i_{j-2}}) \setminus N(v_{i_{j-1}})$.

Dies ergibt einen Widerspruch zur Endlichkeit von G. Zunächst setzen wir $i_0 = 1$. Da $v_{i_0} = v_1$ nicht simplizial ist, gibt es zwei nichtbenachbarte Knoten $v_{i_1}, v_{i_2} \in N(v_{i_0})$; wir können $i_1 < i_2$ annehmen und dass die Bedingung (b) gilt.

Es bleibt, i_j für $j \geq 3$ zu finden. Da $v_{i_{j-1}} \in N(v_{i_{j-3}}) - N(v_{i_{j-2}})$ gilt und `LexBFS` den Knoten $v_{i_{j-2}}$ vor $v_{i_{j-3}}$ ausgewählt hat, können die Knoten $v_{i_{j-2}}$ und $v_{i_{j-3}}$ zu dem Zeitpunkt als `LexBFS` den Knoten $v_{i_{j-1}}$ ausgewählt hat nicht mehr in der gleichen Menge sein. Dies ist nur möglich, wenn es ein $i_j > i_{j-1}$ gibt mit $v_{i_j} \in N(v_{i_{j-2}}) - N(v_{i_{j-3}})$. Um zu zeigen, dass i_j der Bedingung (a) genügt, müssen wir nachweisen, dass $v_{i_{j-2}}$ der einzige Nachbar von v_{i_j} unter den bisher ausgewählten Knoten ist. Durch Induktion über $k \geq 3$ lässt sich zeigen, dass $v_{i_{j-k}}$ kein Nachbar von v_{i_j} ist: Für $k = 3$ ist dies bereits durch die Wahl von i_j sichergestellt. Und wenn $v_{i_j} \in N(v_{i_{j-k}}) \setminus N(v_{i_{j-k+1}})$ für $k > 3$ wäre, so ergäbe sich wegen (b) ein Widerspruch zur Wahl von i_{j-k+2}. Schließlich folgt aus der Chordalität von G, dass $\{v_{i_j}, v_{i_{j-1}}\} \notin E$, womit Bedingung (a) gezeigt ist. Außerdem können wir wieder annehmen, dass i_j der Bedingung (b) genügt. ■

Damit haben wir einen Linearzeitalgorithmus, der für chordale Graphen eine perfekte Eliminationsordnung berechnet. Um zu entscheiden, ob ein gegebener Graph chordal ist, genügt es nach Satz 5.43, ob die Umkehrung der durch `LexBFS` gefundenen Knotenreihenfolge tatsächlich eine perfekte Eliminationsordnung ist. Der folgende Algorithmus realisiert diese Überprüfung in linearer Zeit:

Algorithmus `PEO`(V, E)

```
1 (v_1, ..., v_n) ← (LexBFS(V, E))^R
2 for i := 1 to n do
3   if N(v_i) ∩ {v_{i+1}, ..., v_n} = ∅ then
4     j ← min{k > i | v_k ∈ N(v_i)}
5     if (N(v_i) ∩ {v_{j+1}, ..., v_n}) \ N(v_j) ≠ ∅ then
6       return "nicht chordal"
7 return (v_1, ..., v_n)
```

Wegen Lemma 5.46 ist klar, dass der Eingabegraph G nicht chordal ist, wenn der Algorithmus `PEO nicht chordal` ausgibt. Umgekehrt gilt: Wenn G nicht chordal ist, ist Umkehrung $(v_1, \ldots, v_n)$ der durch *LexBFS* berechneten Knotenfolge nach Satz 5.43 keine perfekte Eliminationsordnung, d.h. es gibt einen Knoten v_i, sodass $N(v_i) \cap \{v_{i+1}, \ldots, v_n\}$ zwei nicht-benachbarte Knoten v_j und v_k enthält. Wir können annehmen, dass j der kleinste solche Index ist. Sei i' der größte Index mit $i \leq i' < j$ und $v_j, v_k \in N(v_{i'})$. Dann gibt der Algorithmus `PEO` bei dem Durchlauf der `for`-Schleife, bei dem i' betrachtet wird, `nicht chordal` aus.

Der Algorithmus `PEO` kann sogar noch so erweitert werden, dass er für nicht-chordale Graphen in Linearzeit einen induzierten Kreis der Länge ≥ 4 berechnet: Hierzu wird Zeile 6 durch die im Beweis von Lemma 5.46 angegebene Konstruktion der Knotenfolge ersetzt. Diese wird mit $i_0 = i$ gestartet. Da der Eingabegraph endlich ist, muss die Folge irgendwann abbrechen – und dies ist nur dann möglich, wenn die Kante $\{v_{i_j}, v_{i_{j-1}}\}$ existiert und damit einen induzierten Kreis schließt.

5.3 Kantenfärbungen

Definition 5.47. *Sei $G = (V, E)$ ein Graph und sei $k \in \mathbb{N}$.*

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

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

c) Die ***kantenchromatische Zahl*** *oder der* ***chromatische Index***

von G ist

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

Ist f eine k-Kantenfärbung von G, so bildet jede ***Farbklasse*** $M_i = \{e \in E \mid f(e) = i\}$ ein Matching in G, d.h. f zerlegt E in k disjunkte Matchings $M_1, \ldots, M_k$. Umgekehrt liefert jede Zerlegung von E in k disjunkte Matchings eine k-Kantenfärbung von G.

Beispiel 5.48.

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

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

(siehe Übungen).

Lemma 5.49. *Für jeden nicht leeren Graphen gilt $\Delta \leq \chi' \leq 2\Delta - 1$ und $m/\mu \leq \chi' \leq 2\lceil n/2 \rceil - 1$.*

Beweis. Siehe Übungen. ■

Korollar 5.50. *Für jeden nicht leeren k-regulären Graphen mit einer ungeraden Knotenzahl gilt $\chi'(G) > k$.*

Beweis. Wegen $\mu \leq (n-1)/2$ und $2m = n\Delta$ folgt $\chi' \geq m/\mu \geq n\Delta/(n-1) > \Delta = k$. ■

Lemma 5.51. *Für jeden bipartiten Graphen gilt $\chi' = \Delta$.*

Beweis. Siehe Übungen. Dort wird die Aussage sogar für bipartite *Multigraphen* (d.h. zwei Knoten können durch mehrere Kanten verbunden sein) bewiesen. ■

Als nächstes geben wir einen Algorithmus an, der für jeden Graphen G eine k-Kantenfärbung mit $k \leq \Delta(G) + 1$ berechnet. Für den Beweis benötigen wir folgende Begriffe.

Definition 5.52. *Sei $G = (V, E)$ ein Graph.*

*a) Ein Knoten $u \in V$ heißt **d-gradig**, wenn folgende Bedingungen erfüllt sind:*
- $\deg(u) \leq d$,
- *alle Nachbarn $v \in N(u)$ haben einen Grad $\deg(v) \leq d$ und*
- $\|\{v \in N(u) \mid \deg(v) = d\}\| \leq 1$.

*b) u heißt **stark d-gradig**, wenn folgende Bedingungen erfüllt sind:*
- $\deg(u) = d$,
- *für alle $v \in N(u)$ gilt $d - 1 \leq \deg(v) \leq d$ und*
- $\|\{v \in N(u) \mid \deg(v) = d\}\| = 1$.

Sei u ein Knoten in einem Graphen G und sei f eine k-Kantenfärbung von $G - u$ mit zugehöriger Partition $M_1, \ldots, M_k$. Dann bezeichnet $N_i(u) = N(u) \cap \mathit{free}(M_i)$ die Menge der Nachbarn v von u, für die die Farbe i noch frei ist (d.h. es ist möglich, die Kante $\{u, v\}$ mit i zu färben). Wir sagen f ***blockiert*** die Farbe i, falls $N_i(u) = \emptyset$ ist.

Das nächste Lemma ist eine direkte Folgerung aus obiger Definition.

Lemma 5.53. *Sei u ein stark k-gradiger Knoten in G und sei f eine k-Kantenfärbung von $G - u$. Dann erfüllen die Anzahlen $a_i = \|N_i(u)\|$ folgende Bedingungen:*

(i) $\sum_{i=1}^{k} a_i = 2k - 1$,

(ii) falls f eine Farbe blockiert, dann gibt es eine Farbe j mit $a_j \geq 3$,

(iii) falls f keine Farbe blockiert, dann gibt es eine Farbe j mit $a_j = 1$.

Lemma 5.54. *Sei u ein stark k-gradiger Knoten in $G = (V, E)$ und sei $G - u$ k-kantenfärbbar. Dann hat $G - u$ eine k-Kantenfärbung g, die keine Farbe blockiert.*

Beweis. Sei f eine k-Kantenfärbung für $G-u$. Falls f eine Farbe i blockiert, gibt es nach Lemma 5.53 eine Farbe j mit $a_j \geq 3$.

Betrachte den Graphen $H = (V, M_i \cup M_j)$. Sei v ein beliebiger Knoten in $N_j(u)$. Da $v \notin \mathit{free}(M_i)$ und somit Endpunkt einer Kante in M_i ist, folgt $\deg_H(v) = 1$. Sei P ein bzgl. Inklusion maximaler Pfad in H mit Startknoten v. Da $\Delta(H) \leq 2$ ist, ist P eine Zusammenhangskomponente von H. Vertauschen wir daher die Farben i und j von allen Kanten auf P, so erhalten wir wieder eine k-Kantenfärbung f' für $G-u$. Für diese sind a'_i und a'_j größer 0, da $v \in \mathit{free}(M'_i)$ ist und $\mathit{free}(M_j)$ höchstens 2 Knoten verliert (nämlich v und evtl. den anderen Endpunkt von P).

Folglich blockiert f' eine Farbe weniger als f und wir können diesen Prozess fortsetzen, bis keine Farben mehr blockiert sind. ■

Lemma 5.55. *Sei u ein k-gradiger Knoten in $G = (V, E)$ und sei g eine k-Kantenfärbung für $G-u$. Dann lässt sich aus g eine k-Kantenfärbung f für G konstruieren.*

Beweis. Wir führen Induktion über k. Im Fall $k = 1$ ist u höchstens mit einem Knoten in G verbunden und daher lässt sich g leicht zu einer 1-Färbung für G erweitern.

Ist $k \geq 2$, so modifizieren G zuerst zu einem Graphen G', so dass u stark k-gradig in G' ist. Hierzu erweitern wir die Nachbarschaft von u durch Hinzunahme von Blattknoten auf die Größe k. Anschließend vergrößern wir die Nachbarschaft $N(v)$ jedes Knotens $v \in N(u)$ auf dieselbe Weise, wobei wir sicherstellen, dass genau ein Nachbar von u den Grad k und alle anderen den Grad $k-1$ haben. Zudem erweitern wir g zu einer k-Kantenfärbung g' für $G'-u$.

Nun benutzen wir Lemma 5.54, um g' in eine k-Kantenfärbung f' für $G'-u$ zu transformieren, die keine Farbe blockiert. Nach Lemma 5.53 gibt es eine Farbe j mit $a'_j = 1$. Sei v der einzige Nachbar von u in G', dessen Kanten nicht mit j gefärbt sind. Nun entfernen wir die Kante $\{u, v\}$ sowie alle mit j gefärbten Kanten aus G' und färben alle mit k gefärbten Kanten mit j. Dann haben in dem resultierenden Graphen G'' sowohl u als auch alle seine Nachbarn einen um 1 kleineren Grad, d.h. u ist $(k-1)$-gradig (aber evtl. nicht stark $(k-1)$-gradig) in G''. Zudem liefert die Einschränkung f'' von f' auf die Kanten von $G''-u$ eine $(k-1)$-Kantenfärbung für diesen Graphen.

Nach IV lässt sich aus f'' eine $(k-1)$-Kantenfärbung g'' für G'' konstruieren. Färben wir nun die aus G' entfernten Kanten mit k, so erhalten wir eine k-Kantenfärbung für G' und daraus eine für G. ■

Satz 5.56 (Vizing)**.** *Für jeden Graphen gilt $\chi' \leq \Delta + 1$.*

Beweis. Wir führen Induktion über n. Der Fall $n = 1$ ist trivial.

Im Fall $n \geq 2$ wählen wir einen beliebigen Knoten u in G. Dann ist u d-gradig für ein $d \leq \Delta(G) + 1$. Sei $k = \max\{\Delta(G-u)+1, d\}$. Nach IV hat $G-u$ eine k-Kantenfärbung, aus der wir nach Lemma 5.55 eine k-Kantenfärbung für G konstruieren können. ■

Da der Beweis von Satz 5.56 konstruktiv ist, können wir daraus leicht einen Algorithmus ableiten.

Algorithmus `Vizing`

```
1  input Graph G = (V,E) mit V = {u_1, ..., u_n}
2  f := ∅
3  for ℓ := 2 to n do
4     f := erweitere-faerbung(f, u_ℓ, G[{u_1, ..., u_ℓ}])
```

Prozedur `erweitere-faerbung`(f, u, G)

```
1  k := Δ(G) + 1
2  while deg(u) < k do
3     füge einen neuen Blattnachbarn von u hinzu
4  for all v ∈ N(u) do
5     if deg(v) < k − 1 then
```

füge $k-1-\deg(v)$ neue Blattnachbarn von v
hinzu und erweitere f auf die neuen Kanten
$S := N(u)$; $C := \{1, \ldots, k\}$
while $S \neq \emptyset$ **do**
  if $\forall v \in S : \|\{i \in C \mid v \in \mathit{free}(M_i)\}\| = 2$ **then**
    wähle einen beliebigen Knoten $v \in S$
    füge einen neuen Blattnachbarn von v hinzu
    und erweitere f auf die neue Kante
  while $\exists i \in C : N_i(u) = \emptyset$ **do**
    wähle $j \in C$ mit $\|N_j(u)\| \geq 3$
    berechne einen inklusionsmaximalen Pfad P
    in $G[M_i \cup M_j]$ mit Startpunkt in $N_j(u)$
    vertausche die Farben i und j auf P
  wähle $i \in C$ mit $\|N_i(u)\| = 1$ und sei $N_i(u) = \{v\}$
  $f(\{u, v\}) := i$; $C := C \setminus \{i\}$; $S := S \setminus \{v\}$
$f :=$ Einschränkung von f auf die Kanten in G
return f

Die Prozedur erweitere-faerbung(f, u, G) erweitert eine k-Kantenfärbung f von $G - u$ zu einer k-Kantenfärbung f von G, wobei $k = \Delta(G) + 1$ ist. Hierzu wird G durch Hinzufügen von Blattknoten so zu einem Graphen G' erweitert, dass u stark k-gradig in G' ist. Anschließend wird f auf die mit u inzidenten Kanten in G' fortgesetzt. Schließlich wird die Einschränkung von f auf die Kanten in G zurückgegeben.

Die innere while-Schleife wird für jede blockierte Farbe einmal durchlaufen. Da zu Beginn höchstens k Farben blockiert sind und bei jedem der $\|S\| = k$ Durchläufe der äußeren while-Schleife maximal eine Farbe blockiert wird, wird die innere while-Schleife insgesamt höchstens $2k$-mal durchlaufen.

Damit der Pfad P in Zeit $O(n)$ gefunden werden kann, speichern wir für jeden Knoten v und jede Farbe j den Matchingpartner $M_j(v)$ von v in einer $(k \times n)$-Matrix $M[i, v]$. Dabei setzen wir $M[i, v] = \perp$, falls $v \in \mathit{free}(M_i)$ ist. Nun ist leicht zu sehen, dass die Prozedur erweitere-faerbung eine Laufzeit von $O(kn)$ und somit der Algorithmus Vizing eine Laufzeit von $O(n^2\Delta(G))$ hat.

6 Baum- und Pfadweite

Definition 6.1. *Sei $G = (V, E)$ ein Graph.*

a) Eine ***Baumzerlegung*** *von G ist ein Paar (T, X), wobei $T = (V_T, E_T)$ ein Baum und $X = (X_t)_{t \in V_T}$ eine Familie von Untermengen von V ist, so dass gilt:*

- $\bigcup_{t \in V_T} X_t = V$,
- *für jede Kante $\{u, v\} \in E$ gibt es ein $t \in V_T$ mit $\{u, v\} \subseteq X_t$, und*
- *für jeden Knoten $u \in V$ ist der durch $X^{-1}(u) := \{t \in V_T \mid u \in X_t\}$ induzierte Untergraph $T[X^{-1}(u)]$ von T zusammenhängend.*

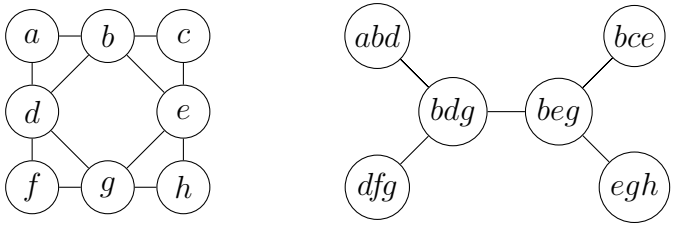
b) Die ***Weite*** *von (T, X) ist $w(T, X) = \max_{t \in V_T} \|X_t\| - 1$.*

c) Die ***Baumweite*** *$tw(G)$ von G ist die kleinste Weite aller möglichen Baumzerlegungen von G.*

d) Eine Baumzerlegung (T, X) von G heißt ***Pfadzerlegung****, wenn T ein Pfad ist. Die* ***Pfadweite*** *$pw(G)$ von G ist die kleinste Weite aller möglichen Pfadzerlegungen von G.*

Die Mengen X_t werden als ***Taschen*** (engl. bags) bezeichnet.

Beispiel 6.2. *(i) Betrachte folgenden Graphen G:*

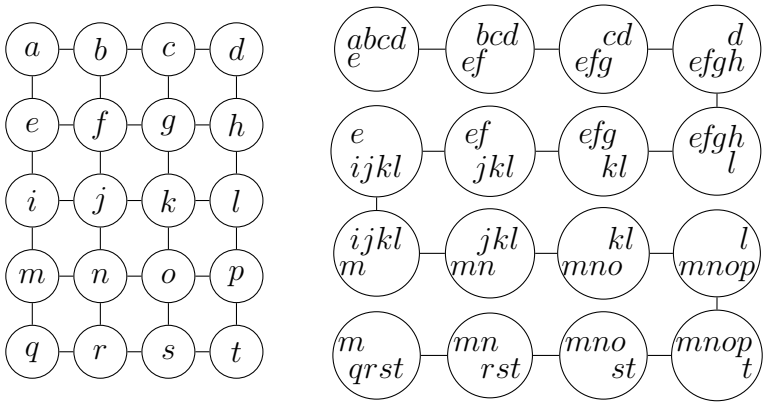


G hat eine Baumzerlegung mit den Taschen $X_1 = \{a, b, d\}$, $X_2 = \{b, d, g\}$, $X_3 = \{b, e, g\}$, $X_4 = \{b, e, c\}$, $X_5 = \{e, g, h\}$, $X_6 = \{d, f, g\}$ und dem Baum $T = (\{1, \ldots, 6\}, E_T)$, wobei E_T folgende Kanten enthält: $\{1, 2\}$, $\{2, 3\}$, $\{3, 4\}$, $\{3, 5\}$, $\{2, 6\}$.

(ii) Sei $G_{k \times \ell} = P_k \times P_\ell$ der $(k \times \ell)$-Gittergraph mit $k\ell$ Knoten. Dann gilt

$$tw(G_{k \times \ell}) \leq pw(G_{k \times \ell}) \leq \min\{k, \ell\}.$$

Der Graph $G_{5 \times 4}$ hat bspw. folgende Pfadzerlegung der Weite 4:



◁

Proposition 6.3. *Sei $G = (V, E)$ ein Graph. Dann gilt:*

- *$tw(G) = 0 \Leftrightarrow pw(G) = 0 \Leftrightarrow E = \emptyset$*
- *Wenn G ein nicht-leerer Wald ist, gilt $tw(G) = 1$.*

Beweisidee. Für den leeren Graphen kann eine Baumzerlegung der Weite 0 konstruiert werden, indem für jeden Knoten eine Tasche erzeugt wird, die nur diesen Knoten enthält. Diese Taschen können dann in beliebiger Reihenfolge zu einem Pfad verbunden werden. Wenn G umgekehrt eine Baumdekomposition der Weite 0 hat, kommen nie zwei Knoten gemeinsam in einer Tasche vor, woraus $E = \emptyset$ folgt.

Für die zweite Aussage genügt es zu zeigen, dass jeder nicht-leere Baum Baumweite 1 hat, da sich die Baumdekompositionen für die

Zusammenhangskomponenten durch Verbinden mit zusätzlichen Kanten leicht zu einer für den gesamten Graphen kombinieren lassen. Sei $G = (V, E)$ ein Baum. Dann ist (T, X) mit $V_T = E \cup \left\{\{u\} \;\middle|\; u \in V\right\}$, $X_t = t$ für $t \in V_T$ und $E_T = \left\{\{s, t\} \;\middle|\; s \subset t\right\}$ eine Baumdekomposition der Weite 1 von G. ■

Definition 6.4. *Eine Baumzerlegung (T, X) heißt **kompakt**, wenn alle Taschen paarweise unvergleichbar sind, d.h. für alle $s \neq t \in V_T$ gilt $X_s \not\subseteq X_t$ und $X_t \not\subseteq X_s$.*

Proposition 6.5. *Sei $G = (V, E)$ ein Graph.*

(i) Jede Baumzerlegung (T, X) von G lässt sich in Polynomialzeit in eine kompakte Baumzerlegung (T', X') von G transformieren, die nur Taschen aus X enthält.

(ii) Für jede kompakte Baumzerlegung (T, X) von G gilt $n(T) \leq n(G)$.

(iii) Für jeden Untergraphen H von G gilt $tw(H) \leq tw(G)$ und $pw(H) \leq pw(G)$.

(iv) Für jede Kante $\{u, v\} \in E$ gilt $tw(G_{uv}) \leq tw(G)$ und $pw(G_{uv}) \leq pw(G)$.

Beweisidee.

(i) Der Algorithmus traversiert T von einem beliebigen Blatt aus. Für jede Kante $\{s, t\}$ (wobei s näher am Startknoten liege als t) wird geprüft, ob $X_s \subseteq X_t$ oder $X_s \supseteq X_t$ gilt. In diesem Fall wird die Kante $\{s, t\}$ zu s kontrahiert und X_s auf $X_s \cup X_t \in \{X_s, X_t\}$ gesetzt. Anschließend wird die Graphsuche im so erhaltenen Graphen T_{st} fortgesetzt.

Dieser Algorithmus benötigt $O(w(T, X) \cdot n(T))$ Zeit. Es ist leicht zu sehen, dass das Ergebnis eine kompakte Baumrepräsentation von G ist. Angewandt auf eine Pfadkomposition findet der Algorithmus eine kompakte Pfaddekomposition.

(ii) Sei (T, X) eine kompakte Baumdekomposition von $G = (V, E)$, und sei s ein Blatt von T und s' sein Nachbar. Da (T, X) kompakt ist, muss es einen Knoten $v_s \in X_s \setminus X_{s'}$. Betrachte nun einen beliebigen Baumknoten $t \in V_T \setminus \{s\}$, und sei $t' \in V_T$ der erste Knoten auf dem Pfad von t nach s. Dann muss es wieder einen Knoten $v_t \in X_t \setminus X_{t'}$ geben. Da die Baumknoten, deren Taschen diese Knoten v_s und v_t enthalten jeweils einen Teilbaum bilden, gilt $t_1 \neq t_2 \Rightarrow v_{t_1} \neq v_{t_2}$ – und damit $n(T) \leq n(G)$.

(iii) Sei (T, X) eine Baumdekomposition von $G = (V, E)$ und sei $H = (V', E')$ ein Untergraph von G. Dann ist (T, X') mit $X'_t = X_t \cap V'$ eine Baumdekomposition von H mit $w(T, X') \leq w(T, X)$.

(iv) Sei (T, X) eine Baumdekomposition von $G = (V, E)$. Dann ist (T, X') mit

$$X'_t = \begin{cases} X_t & \text{wenn } v \notin X_t \\ (X_t \setminus \{v\}) \cup \{u\} & \text{wenn } v \in X_t \end{cases}$$

eine Baumdekomposition von G_{uv} mit $w(T, X') \leq w(T, X)$. ■

Korollar 6.6. *Für jeden Minor H eines Graphen G gilt $tw(H) \leq tw(G)$ und $pw(H) \leq pw(G)$.*

Als nächstes zeigen wir, dass $tw(G) \geq \omega(G) - 1$ ist. Für 2 Teilbäume T' und T'' eines Baumes T bezeichne $d_T(T', T'')$ die minimale Länge eines Pfades P im Baum T, der einen Knoten $u \in T'$ mit einem Knoten $v \in T''$ verbindet. Man beachte, dass der kürzeste Pfad P zwischen T' und T'' im Fall $d_T(T', T'') > 0$ eindeutig bestimmt ist und der einzige Pfad zwischen T' und T'' ist, dessen innere Knoten weder zu T' noch zu T'' gehören.

Lemma 6.7. *Seien $T_1, \ldots, T_k, S$ Teilbäume von T, so dass $T_1 \cap \cdots \cap T_k \neq \emptyset$ ist. Dann gilt*

$$d_T(T_1 \cap \cdots \cap T_k, S) \leq \max\{d_T(T_i, S) \mid i = 1, \ldots, k\}.$$

Beweis. Wir führen Induktion über k. Für $k = 1$ ist nichts zu zeigen. Im Fall $k \geq 2$ gilt für $S' = T_1 \cap \cdots \cap T_{k-1}$ nach IV, dass $d_T(S', S) \leq \max\{d_T(T_i, S) \mid i = 1, \ldots, k-1\}$ ist. Daher reicht es zu zeigen, dass $d_T(S' \cap T_k, S) \leq \max\{d_T(S', S), d_T(T_k, S)\}$ ist. Sei $d = d_T(S' \cap T_k, S) > 0$ und sei $P = (u_1, \ldots, u_d)$ der kürzeste Pfad zwischen $S' \cap T_k$ und S. Dann gehören $u_2, \ldots, u_d$ alle nicht zu S' oder alle nicht zu T_k. Somit ist P kürzester S'-S Pfad oder kürzester T_k-S Pfad. ■

Definition 6.8. *Ein Mengensystem $X = (X_i)_{i \in I}$ hat die **Helly-Eigenschaft**, wenn für jede Teilmenge $J \subseteq I$ gilt:*

$$(\forall i, j \in J : X_i \cap X_j \neq \emptyset) \Rightarrow \bigcap_{j \in J} X_j \neq \emptyset.$$

Lemma 6.9. *Die Knotenmengen aller Unterbäume eines Baumes T haben die Helly-Eigenschaft.*

Beweis. Sei $Z = \{T_1, \ldots, T_k\}$ eine beliebige Menge von Unterbäumen von T, so dass je zwei Unterbäume T_i und T_j aus Z einen nicht leeren Schnitt haben. Wir zeigen induktiv über $k = \|Z\|$, dass dann auch $\bigcap Z$ nicht leer ist.

Für $k \leq 2$ ist nichts zu zeigen. Im Fall $k \geq 3$ gilt nach IV, dass $T' = \bigcap_{i=1}^{k-1} T_i \neq \emptyset$ ist. Zudem folgt mit obigem Lemma, dass $d_T(T', T_k) \leq \max\{d_T(T_i, T_k) \mid i = 1, \ldots, k-1\} = 0$ und somit $T' \cap T_k \neq \emptyset$ ist. ■

Satz 6.10. *Sei (T, X) eine Baumzerlegung eines Graphen G und sei C eine Clique in G. Dann gibt es eine Tasche X_t, in der C enthalten ist.*

Beweis. Da je 2 Knoten $u, v \in C$ durch eine Kante verbunden sind, gibt es eine Tasche X_t mit $\{u, v\} \in X_t$. Folglich haben die zugehörigen Unterbäume $X^{-1}(u)$ und $X^{-1}(v)$ mindestens einen Knoten t gemeinsam. Nach Lemma 6.9 gibt es daher einen Knoten, der in allen Unterbäumen $X^{-1}(u)$ mit $u \in C$ enthalten ist. ■

Korollar 6.11.

(i) $\omega(G) \leq tw(G) + 1$,

(ii) $tw(K_n) = n - 1$.

Definition 6.12. *Eine Baumzerlegung (T, X) von G heißt **Baumzerlegung in Cliquen**, wenn alle Taschen Cliquen in G sind.*

Lemma 6.13. *Ein Graph G ist genau dann chordal, wenn er eine Baumzerlegung in Cliquen hat.*

Beweis. Sei G chordal. Wir zeigen mittels Induktion über die Knotenanzahl n, dass G eine Baumzerlegung in Cliquen hat. Für $n = 1$ ist dies klar. Für $n \geq 2$ sei v ein simplizialer Knoten von G. Nach Induktionsvoraussetzung hat $G - v$ eine Baumzerlegung (T', X') in Cliquen. Da $N_G(v)$ eine Clique in $G - v$ ist, gibt es nach Satz 6.10 $t \in V_{T'}$ mit $N_G(v) \subseteq X_t$. Um eine Baumzerlegung (T, X) von G zu erhalten, konstruieren wir T aus T' durch Anfügen eines neuen Blattnachbarn s von t und erweitern X' zu X durch $X_s = N_G(v) \cup \{v\}$. Es ist leicht zu sehen, dass (T, X) eine Baumzerlegung in Cliquen ist.

Sei nun umgekehrt (T, X) eine Baumzerlegung in Cliquen von G. Für jeden Kreis K der Länge ≥ 4 kann wie im Beweis von Proposition 6.5(iii) eine Baumdekomposition (T', X') für den von K induzierten Teilgraphen konstruieren; auch hier sind alle Taschen Cliquen. Wegen $tw(G[K]) \geq 2$ gibt es $t \in V_{T'}$ mit $\|X'_t\| \geq 3$. Da X'_t eine Clique induziert, hat K eine Sehne. ■

Frage: Welche Graphen haben eine Pfadzerlegung in Cliquen?

Definition 6.14. *Sei $G = (V, E)$ ein Graph. Dann heißt ein Graph $H = (V, E')$ eine **chordale Erweiterung** von G, wenn H chordal und G ein Teilgraph von H ist.*

Lemma 6.15. *$tw(G) = \min\{\omega(H) - 1 \mid H$ ist eine chordale Erweiterung von $G\}$.*

Beweis. $\leq$: Sei H ein beliebiger chordaler Graph, der G als Subgraph enthält. Dann hat H eine Baumzerlegung (T, X) in Cliquen und es gilt $tw(H) = \omega(H) - 1$. Da (T, X) auch eine Baumzerlegung von G ist, folgt $tw(G) \leq \omega(H) - 1$.

$\geq$: Sei (T, X) eine Baumzerlegung von G der Weite $w = tw(G)$. Fügen wir alle Kanten $\{u, v\}$ zu G hinzu, für die u und v in einer gemeinsamen Tasche liegen, so ist der resultierende Graph H chordal und es gilt $\omega(H) \leq w - 1$. ■

Definition 6.16. *Sei $G = (V, E)$ ein Graph und seien $U, W \subseteq V$ zwei nichtleere disjunkte Knotenmengen in G. Dann heißt $X \subseteq V$ ein U-W-Separator in G, wenn $U \setminus X$ und $W \setminus X$ nichtleer sind und es in $G - X$ keinen Pfad von einem Knoten $u \in U$ zu einem Knoten $w \in W$ gibt.*

Es ist leicht zu sehen, dass ein Graph $G = (V, E)$ genau dann k-fach zusammenhängend ist, wenn es keine Mengen $U, W, X \subseteq V$ gibt sodass X ein U-W-Separator ist mit $\|X\| < k$.

Lemma 6.17. *Sei (T, X) eine Baumzerlegung von $G = (V, E)$ und sei $e = \{u, w\}$ eine Kante in T. Seien T_u und T_w die beiden Komponenten von $T - e$ und seien $X(T_u) = \bigcup_{t \in T_u} X_t$ und $X(T_w) = \bigcup_{t \in T_w} X_t$. Falls $U = X(T_u) \setminus X(T_w)$ und $W = X(T_w) \setminus X(T_u)$ beide nicht leer sind, ist $X = X_u \cap X_w$ ein U-W-Separator in G.*

Beweis. Wegen $X_u \subseteq X(T_u)$ und $X_w \subseteq X(T_w)$ folgt $X = X_u \cap X_w \subseteq X(T_u) \cap X(T_w)$. Da (T, X) eine Baumzerlegung ist, gilt auch die umgekehrte Inklusion, d.h. $X, U = X(T_u) \setminus X(T_w)$ und $W = X(T_w) \setminus X(T_u)$ zerlegen V in 3 Mengen. Daher reicht es zu zeigen, dass in G keine Kante zwischen U und W existiert.

Sei $\{a, b\} \in E$. Dann gibt es eine Tasche X_t mit $\{a, b\} \subseteq X_t$. Falls t zu T_u gehört, folgt $\{a, b\} \subseteq X(T_u)$ und somit $\{a, b\} \cap W = \emptyset$. Falls t zu T_w gehört, folgt $\{a, b\} \subseteq X(T_w)$ und somit $\{a, b\} \cap U = \emptyset$. ■

Satz 6.18. *Für jeden Graphen G gilt $\kappa(G) \leq tw(G)$.*

Beweis. Sei $k = tw(G)$. Dann hat G mindestens $k + 1$ Knoten. Falls G genau $k + 1$ Knoten hat, ist G höchstens k-fach zusammenhängend. Falls G $n > k + 1$ Knoten hat, sei (T, X) eine kompakte Baumzerlegung von G und sei $e = \{u, w\}$ eine Kante in T. Da (T, X) kompakt ist, gilt $X_u \setminus X_w \neq \emptyset$ und $X_w \setminus X_u \neq \emptyset$ und somit $\|X_u \cap X_w\| \leq k$. Nach obigem Lemma ist $X = X_u \cap X_w$ ein Separator in G und somit G höchstens k-fach zusammenhängend. ■

Korollar 6.19. *Ein Graph $G = (V, E)$ hat genau dann die Baumweite $tw(G) = 1$, wenn G ein Wald und $E \neq \emptyset$ ist.*

Frage: Welche Graphen G haben Pfadweite $pw(G) = 1$?

Satz 6.20. *Für jeden Graphen G gilt $m(G) \leq tw(G) \cdot n(G)$.*

Beweis. Wir führen den Beweis mittels Induktion über die Knotenzahl n.

$n \leq tw(G) + 1$: Jeder Knoten hat maximal $tw(G)$ Nachbarn, folglich gilt sogar $m(G) \leq \frac{1}{2} \cdot tw(G) \cdot n(G)$.

$n > tw(G) + 1$: Sei (T, X) eine Baumdekomposition von G mit minimaler Weite. Nach Proposition 6.5 können wir annehmen, dass (T, X) kompakt ist. Sei $t \in V_T$ ein Blatt von T, und sei s der Nachbar von t. Wegen der Kompaktheit existiert ein Knoten $u \in X_t \setminus X_s$; dieser taucht damit nur in der Tasche X_t auf. Insbesondere müssen alle Nachbarn von u in X_t enthalten sein, es gilt also $\deg_G(u) \leq \|X_t\| - 1 \leq tw(G)$. Nach Induktionsvoraussetzung gilt außerdem $m(G - u) \leq$

$tw(G-u) \cdot n(G-u) \leq tw(G) \cdot \Big(n(G)-1\Big)$. Zusammen ergibt sich $m(G) = \deg_G(u) + m(G-u) \leq tw(G) \cdot n(G)$. ■

6.1 Dynamische Programmierung über Baumdekompositionenen

Definition 6.21. *Sei $P \subseteq \Sigma^*$ ein Entscheidungsproblem und $k\colon \Sigma^* \to \mathbb{N}$ ein Parameter (beispielsweise die Baumweite des Eingabegraphen). Dann heißt P* fixed parameter tractable (FPT) *bezüglich k (kurz: $(P,k) \in \mathsf{FPT}$), wenn es eine Funktion $f\colon \mathbb{N} \to \mathbb{N}$ und einen Algorithmus gibt, der für jedes $x \in \Sigma^*$ in $f(k(x)) \cdot |x|^{O(1)}$ Zeit entscheidet, ob $x \in P$ gilt.*

Die Funktion f ist hierbei beliebig, insbesondere kann sie exponentiell wachsen. Für NP-schwere Probleme sind FPT-Algorithmen dennoch interessant, weil sie das exponentielle Laufzeitverhalten auf den Parameter beschränken. Wird der Parameter als konstant angenommen, ergibt sich für wachsende Eingabegrößen polynomielle Laufzeit. Dabei ist die Aussage »P ist FPT bezüglich der Baumweite« stärker als »für jede Baumweite w ist das Problem P für Graphen mit durch w beschränkter Baumweite in Polynomialzeit lösbar«, weil letzteres auch bei einer Laufzeit von $n^{O(w)}$ zutrifft.

Viele Probleme, die für allgemeine Graphen NP-schwer sind, sind FPT bezüglich der Baumweite. Häufig lassen sich sogar Laufzeiten der Form $f(w) \cdot n$ erreichen; die Größe der Eingabe geht also nur linear ein.

Um nachzuweisen, dass ein Problem FPT bezüglich der Baumweite ist, bietet es sich an, den Dekompositiosbaum an einem beliebigen Knoten zu wurzeln und dann von den Blättern aufwärts Teillösungen zu berechnen, bis eine Lösung für den gesamten Graphen vorliegt.

Definition 6.22. *Sei (T,X) eine Baumdekomposition für einen Graphen $G=(V,E)$ und sei $r \in V_T$. Dann heißt (T,r,X)* gewurzelte Baumdekomposition von *G, wobei alle kannten von der Wurzel r weggerichtet werden. Zudem definieren wir für $t \in V_T$:*

$$T(t) = \{s \in V_T \mid s \text{ ist von } u \text{ aus erreichbar}\}$$
$$V(t) = \bigcup_{s \in T(t)} X_t$$
$$G(v) = G[V(t)]$$

Die Idee ist nun, für jeden Baumknoten $t \in V_T$ und jede lokale Lösung L_t auf $G[X_t]$ in einer Tabelle zu speichern, ob (und ggf. wie) sich L_t zu einer Lösung $\hat{L}_t$ auf $G(t)$ erweitern lässt. Dabei werden wir T post-order traversieren, um bei der Behandlung eines inneren Knotens bereits auf die Information über seine Kinder zugreifen können. Um zu prüfen, ob zu einer lokalen Lösung L_t auf $G[X_t]$ eine Erweiterung zu einer Lösung $\hat{L}_t$ auf $G(t)$ existiert, werden wir prüfen, ob zu L_t kompatible Teillösungen $L_{s_1}, \dots, L_{s_k}$ für die Kinder $s_1, \dots, s_k$ von t existieren, die sich zu Lösungen auf $G(s_1), \dots, G(s_k)$ erweitern lassen.

Damit ergibt sich folgender Meta-Algorithmus:

```
1 for each t ∈ V_T (bottom-up) do
2   for each Teillösung L_t auf G[X_t] do
3     Speichere, ob L_t mit erweiterbaren Teillösungen
        L_{s_1}, ..., L_{s_k} für die Kinder s_1, ..., s_k von t zu
        einer Lösung L̂_t auf G(t) kombiniert werden
        kann
4 Prüfe, ob eine passende "Teil-"Lösung für die
        Wurzel r existiert
```

Um diesen Meta-Algorithmus für ein konkretes Problem anzupassen, muss jeweils geklärt werden, was eine *lokale Teillösung* ist, wann Teillösungen *kompatibel* sind und wie kompatible Teillösungen *kombiniert* werden können.

Färbbarkeit

Als erstes Beispiel für ein NP-schweres Problem, dass FPT in der Baumweite ist, werden wir k-Färbbarkeit betrachten – also die Frage, ob für einen gegebenen Graphen G eine k-Färbung existiert.

Als lokale Teillösung an einem Baumknoten $t \in V_T$ verwenden wir Funktionen $f_t \colon X_t \to [k]$, wobei $[k] = \{1, 2, \ldots, k\}$. Um den Algorithmus zu vereinfachen, betrachten wir auch solche f_t, die keine k-Färbung von $G[X_t]$ sind.

Für zwei Teilgraphen $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ eines Graphen $G = (V, E)$ nennen wir k-Färbungen $f_1 \colon V_1 \to [k]$ von G_1 und $f_2 \colon V_2 \to [k]$ von G_2 *kompatibel*, wenn $f_1(u) = f_2(u)$ für alle $u \in V_1 \cap V_2$ gilt. Falls zusätzlich $V_1 \subset V_2$ gilt, so heißt f_2 *Erweiterung* von f_1.

Für $t \in V_T$ und $f_t \colon X_t \to [k]$ definieren wir das Prädikat

$$P_t(f_t) = \begin{cases} 1 & \text{falls es eine } k\text{-Färbung } \hat{f}_t \text{ von } G(t) \text{ gibt, die } f_t \text{ erweitert} \\ 0 & \text{sonst.} \end{cases}$$

Das dynamische Programm wird dieses Prädikat von den Blättern aus für jeden Knoten von T berechnen. Für Blätter $t \in V_T$ genügt es zu prüfen, ob f_t eine k-Färbung von $G[X_t]$ ist. Für innere Knoten muss zusätzlich die Erweiterbarkeit geprüft werden. Dies gelingt mit dem folgenden Lemma.

Lemma 6.23. *Sei $t \in V_T$ und $f_t \colon X_t \to [k]$ eine k-Färbung von $G[X_t]$. Dann gilt $P_t(f_t) = 1$ genau dann, wenn für alle Kinder s von t eine lokale Lösung $f_s \colon X_s \to [k]$ mit $P_s(f_s) = 1$ existiert, die zu f_t kompatibel ist.*

Beweis. »⇒«: Sei $\hat{f}_t$ eine k-Färbung von $G(t)$, die f_t erweitert. Wir definieren für jedes Kind s von t die k-Färbung $f_s = \hat{f}_t|_{X_s}$. Da sich diese auf die k-Färbung $\hat{f}_s = \hat{f}_t|_{V(s)}$ von $G(s)$ erweitern lässt, gilt $P_s(f_s) = 1$. Außerdem ist f_s nach Konstruktion zu f_t kompatibel.

»⇐«: Seien $s_1, \ldots, s_d$ die Kinder von t. Für $i \in [d]$ sei $f_{s_i} \colon X_{s_i} \to [k]$ eine lokale Lösung, die zu f_t kompatibel ist und die sich zu einer k-Färbung $\hat{f}_s$ von $G(s_i)$ erweitern lässt. Wegen $i \neq j \Rightarrow V(s_i) \cap V(s_j) \subseteq X_t$ sind die Funktionen in $F = \{f_t, f_{s_1}, \ldots, f_{s_d}\}$ paarweise kompatibel. Ihre Vereinigung $\hat{f}_t = \bigcup F$ ist eine k-Färbung von $G(t)$, die f_t erweitert. ■

Zusammen ergibt sich der folgende Algorithmus:

Prozedur `treedec-color`(V, E, T, r, X, k)

```
1  for each t ∈ V_T (bottom-up) do
2    for each f_t : X_t → [k] do
3      P_t(f_t) := 1
4      if ∃u, v ∈ X_t : {u, v} ∈ E ∧ f_t(u) = f_t(v) then
5        P_t(f_t) := 0
6      else
7        for each child s of t do
8          if ∄f_s : X_s → [k] : P_s(f_s) = 1 und f_s, f_t
           kompatibel then
9            P_t(f_t) := 0; break
10 return ∃f_r : X_r → [k] : P_r(f_r) = 1
```

Die Korrektheit folgt aus Lemma 6.23. Um die Laufzeit abzuschätzen, nehmen wir an, dass die gegebene Baumdekomposition kompakt ist; damit hat T nach Proposition 6.5 höchstens n Knoten und die äußere Schleife über t wird höchstens n mal durchlaufen. Für f_t stehen dabei jeweils höchstens k^{w+1} Möglichkeiten zur Verfügung. In den Übungen wird $\chi(G) \leq tw(G) + 1$ gezeigt, daher können wir $k^{w+1} \leq (w+1)^{w+1}$ abschätzen. Der Test in Zeile 4 kann in $O(w\Delta)$ implementiert werden, indem für jeden Knoten $u \in X_t$ seine Adjazenzliste durchgegangen wird. Da jeder Baumknoten s höchstens einen Elternknoten t hat, wird die innere `for`-Schleife höchstens $\big((w+1)^{w+1} \cdot n\big)$-mal durchlaufen. In ihr müssen jeweils höchstens $(w+1)^{w+1}$ Funktionen f_s getestet werden. Zusammen ergibt sich eine Laufzeit von $O\big((w+1)^{2w+2} w\Delta n\big)$.

Da der Maximalgrad Δ nicht durch w beschränkt ist (der $K_{1,\Delta}$ hat Baumweite 1 und Maximalgrad Δ), ist diese Laufzeit noch nicht linear in n. Um dies zu erreichen, kann für den Adjazenztest in Zeile 4 eine Lazy-Adjazenzmatrix verwendet werden, in der nur die 1-Einträge initialisiert werden, und die Gültigkeit eines Eintrags $a_{i,j}$ beim Zugriff noch dadurch geprüft wird, ob er auf einen Eintrag einer zusätzlichen Liste zeigt, in der an dieser Stelle das Tupel (i, j) steht. Hierbei entsteht ein Initialisierungsaufwand von $O(m) = O(wn)$ und es kann für ein Paar von Knoten in konstanter Zeit getestet werden, ob sie benachbart sind. Allerdings wird quadratisch viel Platz in n benötigt.

Um nicht nur die Laufzeit, sondern auch den Platzbedarf linear in n zu halten, hilft das folgende Lemma. Für $S \subseteq V_T$ definieren wir $top(S)$ als den Knoten in S, der den kürzesten Abstand zur Wurzel von T hat. Für $u \in V$ sei $top(u) = top(X^{-1}(u))$ und $N_{top}(u) = N(u) \cap X_{top(u)}$.

Lemma 6.24. $\{u, v\} \in E \Leftrightarrow u \in N_{top}(v) \vee v \in N_{top}(u)$

Beweis. »⇐« ist nach Definition klar.
»⇒«: Sei $S = X^{-1}(u) \cap X^{-1}(v)$. Dann gilt $top(S) \in \{top(u), top(v)\}$, da sonst auch der Elternknoten von $top(S)$ in S enthalten wäre. ■

Die Mengen $N_{top}(u)$ können für alle Knoten $u \in V$ in $O(n + m) = O(wn)$ Zeit berechnet werden, indem T traversiert wird und an jeder Tasche t für alle Knoten in X_t, die nicht auch in der Elterntasche enthalten sind, die Adjazenzliste durchgegangen wird (was jede Kante nur zweimal besucht). Wegen $\|N_{top}(u)\| \leq w + 1$ sind dann Adjazenztests in $O(w)$ Zeit möglich. Dies ergibt für `treedec-color` eine Gesamtlaufzeit von $O\Big((w+1)^{2w+2}w^3n\Big)$.

Stabile Mengen

Als zweites Beispiel für ein NP-schweres Problem, das FPT in der Baumweite des Eingabegraphen ist, betrachten wir das Stabilitätsproblem. Hier wird für einen gegebenen Graphen $G = (V, E)$ und

eine Zahl $k \in \mathbb{N}$ gefragt, ob es eine stabile Menge $U \subseteq V$ der Größe $\|U\| = k$ gibt.

Der Algorithmus wird wieder eine Variante des zu Beginn der Abschnitts vorgestellten Meta-Algorithmus sein. Eine *lokale Teillösung* für einen Baumknoten $t \in V_T$ ist eine Menge $I_t \subseteq X_t$. Dabei interessieren wir uns eigentlich nur für solche I_t die stabil in $G[X_t]$ sind, nehmen der Einfachheit halber aber alle Teilmengen von X_t in die Tabelle des dynamischen Programms auf.

Seien $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ Teilgraphen eines Graphen $G = (V, E)$. Zwei stabile Mengen I_1 von G_1 und I_2 von G_2 heißen *kompatibel*, wenn $I_1 \cap V_2 = I_2 \cap V_1$ gilt. In diesem Fall ist $I_1 \cup I_2$ stabil in $(V_1 \cup V_2, E_1 \cup E_2)$.

Das dynamische Programm berechnet für $t \in V_T$ und $I_t \subseteq X_t$ die Werte

$$P_t(I_t) = \begin{cases} \max\left\{ \|\hat{I}_t\| \;\middle|\; \begin{array}{l} \hat{I}_t \text{ ist stabil in } G(T) \\ \text{und kompatibel zu } I_t \end{array} \right\} & \text{falls } I_t \text{ stabil in } G[X_t] \\ -\infty & \text{sonst.} \end{cases}$$

Um die rekursive Berechnungsvorschrift für P_t möglichst einfach zu halten, werden wir Baumdekompositionen mit einer besonderen Struktur verwenden.

Definition 6.25. *Deine gewurzelte Baumdekomposition (T, r, X) heißt **einfach**, wenn alle Baumknoten $t \in V_T$ eine der folgenden Bedingungen erfüllen:*

*(a) t hat keine Kinder und $\|X_t\| = 1$. In diesem Fall heißt t **Blatt**.*

*(b) t hat ein Kind s mit $X_t \subset X_s$ und $\|X_t\| = \|X_s\| - 1$. In diesem Fall heißt t **Weglass-Knoten**.*

*(c) t hat ein Kind s mit $X_t \supset X_s$ und $\|X_t\| = \|X_s\| + 1$. In diesem Fall heißt t **Hinzufüge-Knoten**.*

*(d) t hat zwei Kinder r und s mit $X_r = X_s = X_t$. In diesem Fall heißt t **Kombinationsknoten**.*

Lemma 6.26. *Es gibt einen Algorithmus, der aus einer kompakten Baumdekomposition (T, X) von $G = (V, E)$ in $O(wn)$ Zeit eine einfache Baumdekomposition von G mit gleicher Weite und $O(wn)$ Knoten berechnet.*

Beweis. Der Algorithmus arbeitet wie folgt:

Prozedur `simple-treedec`(T, X)

```
1  for t ∈ V_T do
2    if deg_T(t) = 1 ∧ ‖X_t‖ > 1 then
3      füge ein neues Blatt t' als Nachbarn von t an
4      wähle u ∈ X_t und setze X_t' = {u}
5  for {s,t} ∈ E_T do
6    if X_s ⊈ X_t ∧ X_s ⊉ X_t then
7      unterteile {s,t} durch einen neuen Knoten r
8      setze X_r = X_s ∩ X_t
9  for {s,t} ∈ E_T do
10   sei {v_1,...,v_p} = X_s und {v_1,...,v_q} = X_t und p < q
11   unterteile {s,t} durch q − p − 1 Knoten t_1,...,t_{q−p−1}
12   setze X_{t_i} = {v_1,...,v_{p+i}}
13 wähle r ∈ V_T und betrachte T als in r gewurzelt
14 for t ∈ V_T do
15   seien t_1,...,t_d die Kinder von t
16   if d > 1 then
17     füge einen Binärbaum mit d Blättern zwischen t
         und seinen Kindern ein, alle neuen Knoten
         erhalten die gleiche Tasche wie t
```

Die erste Schleife stellt sicher, dass alle Blätter einelementige Taschen haben. Die zweite und dritte Schleife stellen sicher, dass sich die Taschen von benachbarten Baumknoten um genau einen Graphknoten unterscheiden. Die vierte Schleife stellt schließlich sicher, dass interne Knoten entweder nur ein oder zwei Kinder haben und in letzterem Fall beide die gleiche Tasche haben.

Nach Proposition 6.5 gilt zu Beginn des Algorithmus $\|V_T\| \leq n$. Nach der ersten Schleife gilt $\|V_T\| \leq 2n$. Nach der zweiten Schleife gilt $\|V_T\| \leq 3n$, da nur für die höchstens $n-1$ Kanten des ursprünglichen Baums neue Knoten eingefügt werden können. Nach der dritten Schleife gilt $\|V_T\| \leq 3wn$, da jede Kante durch höchstens $w = w(T, X)$ Knoten unterteilt wird. Schließlich gilt nach der vierten Schleife $\|V_T\| \leq 9wn$, da die eingefügten Binärbäume höchstens so viele Blätter wie zuvor vorhandene Knoten haben und höchstens so viele innere Knoten wie Blätter.

Es ist leicht zu sehen, dass die Laufzeit linear durch die Ausgabegröße und damit durch $O(wn)$ beschränkt ist. ■

Das folgende Lemma zeigt, wie P_t für einfache Baumdekompositionen rekursiv berechnet werden kann.

Lemma 6.27. *Sei (T, r, X) eine einfache Baumdekomposition und $t \in V_T$. Dann gilt:*

(a) Wenn t ein Blatt ist mit $X_t = \{u\}$, so ist $P_t(\{u\}) = 1$ und $P_t(\emptyset) = 0$.

(b) Wenn t ein Weglass-Knoten mit Kind s und $\{u\} = X_s \setminus X_t$ ist, so gilt $P_t(I) = \max\{P_s(I), P_s(I \cup \{u\})\}$ für alle $I \subseteq X_t$.

(c) Wenn t ein Hinzufüge-Knoten mit Kind s und $\{u\} = X_t \setminus X_s$ ist, so gilt für alle $I \subseteq X_t$:

$$P_t(I) = \begin{cases} -\infty & \text{wenn } I \text{ nicht stabil in } G[X_t] \\ P_s(I) & \text{wenn } I \text{ stabil in } G[X_t] \text{ und } u \notin I \\ P_s(I \setminus \{u\}) + 1 & \text{wenn } I \text{ stabil in } G[X_t] \text{ und } u \in I \end{cases}$$

(d) Wenn t ein Kombinationsknoten mit Kindern r und s ist, so gilt $P_t(I) = P_r(I) + P_s(I) - \|I\|$ für alle $I \subseteq X_t$.

Satz 6.28. *Es gibt einen Algorithmus, der für eine gegebene Baumdekomposition (T, X) eines Graphen $G = (V, E)$ der Weite w in $O(2^w w^4 n)$ Zeit die Stabilitätszahl $\alpha(G)$ berechnet.*

Beweis. Der Algorithmus berechnet zunächst eine einfache Baumdekomposition (T', r, X) von G der Weite w (in $O(wn)$ Zeit nach Lemma 6.26) und die Mengen $N_{top}(u)$ für schnelle Adjazenztests nach Lemma 6.24 (ebenfalls $O(wn)$ Zeit). Dann verwendet er die Berechnungsvorschriften aus Lemma 6.27, um bottom-up für alle Baumknoten $t \in V_{T'}$ und Mengen $I_t \subseteq X_t$ die Werte $P_t(I_t)$ auszurechnen. Schließlich gibt er $\max_{I_r \subseteq X_r} P_r(I_r)$ zurück.

Es gibt $2^w wn$ Paare von t und I_t, für die $P_t(I_t)$ berechnet wird. Je nachdem, welcher Fall von Lemma 6.27 jeweils zum tragen kommt, fällt dabei folgender Aufwand an:

- Fall (a) kann in $O(1)$ Zeit implementiert werden.
- Fall (b) kann in $O(w)$ Zeit implementiert werden, wenn die Indizes $I_t \subseteq X_t$ als Bitfolgen der Länge $\|X_t\|$ repräsentiert werden.
- Fall (c) kann in $O(w^3)$ Zeit implementiert werden, wenn zur Überprüfung ob I_t stabil in $G[X_t]$ ist der $O(w)$-Adjazenztest nach Lemma 6.24 verwendet wird.
- Fall (d) kann wieder in $O(w)$ Zeit implementiert werden.

Zusammen ergibt sich die angegebene Laufzeit. ■

Courcelle hat sogar gezeigt, dass für einen gegebenen Graphen G und eine als MSO-Formel ϕ (*monadic second order logic*) gegebene Grapheigenschaft in $f(tw, \phi)n$ Zeit entschieden werden kann, ob ϕ durch G erfüllt wird. Gegenüber der Prädikatenlogik erster Stufe kann in MSO zusätzlich über Mengen von Knoten oder Kanten quantifiziert werden.

6.2 Berechnen von Baumdekompositionen geringer Weite

Es ist NP-hart zu prüfen, ob ein gegebener Graph eine gegebene Baumweite hat. Der folgende Satz ermöglicht es, dennoch FPT-Algorithmen bezüglich der Baumweite eines gegebenen Graphen anzugeben, ohne dass eine Baumzerlegung als Teil der Eingabe übergeben werden muss.

Satz 6.29. *Es gibt einen Algorithmus, der bei Eingabe eines Graphen $G = (V, E)$ und einer Zahl $w \in \mathbb{N}$ in $f(w) \cdot m \cdot n$ Zeit entweder eine Baumdekomposition (T, X) von G mit $w(T, X) < 4w$ berechnet oder ein Zertifikat für $tw(G) \geq w$ ausgibt.*

Um ein geeignetes Zertifikat für große Baumweite zu finden, werden wir Trennbarkeitseigenschaften betrachten. Für den Rest dieses Abschnitts werden wir voraussetzen, dass G zusammenhängend ist. Dies ist keine Einschränkung, da sich Baumzerlegungen der Zusammenhangskomponenten von G leicht zu einer Baumzerlegung von G zusammensetzen lassen.

Definition 6.30. *Sei $G = (V, E)$ ein Graph.*

- *Zwei Mengen $Y, Z \subseteq V$ mit $\|Y\| = \|Z\|$ sind **trennbar**, wenn eine Menge $S \subseteq V$ mit $\|S\| < \|Y\|$ existiert, sodass es in $G - S$ keinen Pfad von einem Knoten in $Y \setminus S$ zu einem Knoten in $Z \setminus S$ gibt.*
- *Für $w \in \mathbb{N}$ heißt eine Menge $U \subseteq V$ w-verbunden, wenn $\|U\| \geq w$ und es keine trennbaren Mengen $Y, Z \subseteq U$ mit $\|Y\| = \|Z\| \leq w$ gibt.*

Lemma 6.31. *Es gibt einen Algorithmus, der für einen gegebenen Graphen $G = (V, E)$, $U \subseteq V$ und $w \leq \|U\|$ in $f(\|U\|) \cdot m$ Zeit entscheidet, ob U w-verbunden ist. Wenn nicht, findet der Algorithmus Mengen $Y, Z \subseteq U$ und $S \subseteq V$ mit $\|S\| < \|Y\| = \|Z\| \leq w$, sodass es in $G - S$ keinen $(Y \setminus S)$-$(Z \setminus S)$-Pfad gibt.*

Beweis. Der Algorithmus probiert alle $Y, Z \subseteq U$ mit $\|Y\| = \|Z\| \leq w$ durch, davon gibt es höchstens $4^{\|U\|}$ viele.

Y und Z sind genau dann trennbar, wenn es weniger als $\|Y\|$ knotendisjunkte Pfade zwischen Y und Z in G gibt. Um dies zu prüfen, wird im Netzwerk $N = (V', E', s, t, c)$ mit

$$\begin{aligned} V' &= \{v, v' \mid v \in V\} \cup \{s, t\} \\ E' &= \{\{s, v\} \mid v \in Y\} \cup \{\{v', t\} \mid v \in Z\} \\ &\quad \cup \{\{v, v'\} \mid v \in V\} \cup \{\{u', v\} \mid \{u, v\} \in E\} \\ c(e) &= 1 \end{aligned}$$

geprüft, ob der maximale Fluss den Wert $\|Y\|$ hat. Wenn nicht, gibt es einen minimalen Schnitt $S' \subseteq V'$ mit $\|S'\| < \|Y\|$. Aus diesem lässt sich die gesuchte Menge S gewinnen durch $S = \{v \in V \mid v \in S' \vee v' \in S'\}$, die $\|S\| \leq \|S'\| < \|Y\|$ erfüllt. Der maximale Fluss in N kann in $O(\|Y\| \cdot m) = O(\|U\| \cdot m)$ Zeit gefunden werden. ■

Wir wissen bereits von Lemma 6.17, dass es in jedem Graphen mit Baumweite $w - 1$ und $n > w$ einen Separator S mit $\|S\| \leq w - 1$ gibt. Das folgende Lemma ermöglicht es uns zu verlangen, dass sogar jede Teilmenge $U \subseteq V$ ausreichender Größe durch einen Separator S getrennt wird.

Lemma 6.32. *Wenn ein Graph $G = (V, E)$ eine $(w+1)$-verbundene Menge U mit $\|U\| \geq 3w$ hat, gilt $tw(G) \geq w$.*

Beweis. Angenommen G hat auch eine Baumzerlegung (T, X) mit $w(T, X) < w$. Nach Lemma 6.5 können wir annehmen, dass (T, X) kompakt ist. Die Idee ist, eine Tasche X_t zu finden, die »mittig« in U liegt, und diese dann als Separator zu nutzen.

Zunächst wählen wir eine Wurzel $r \in V_T$. Sei $t \in V_T$ ein Knoten mit $\|V(t) \cap U\| > 2w$, der unter allen Baumknoten mit dieser Eigenschaft maximale Entfernung zu r hat. Seien $t_1, \ldots, t_d$ die Kinder von t. Es gilt $d \geq 1$, da sonst $\|V(t)\| = \|X_t\| \leq w$ gelten würde.

1. Fall ($\exists i \in [d] : \|V(t_i) \cap U\| \geq w$): Wir wählen $Y \subseteq V(t_i) \cap U$ und $Z \subseteq U \setminus V(t_i)$ mit $\|Y\| = \|Z\| = w$. Letzteres ist wegen $\|V(t_i) \cap U\| \leq 2w$ möglich. Dann ist $S = X_t \cap X_{t_i}$ ein $(Y \setminus S)$-$(Z \setminus S)$-Separator. Da (T, X) kompakt ist, gilt außerdem $\|S\| < w$.

2. Fall ($\forall i \in [d] : \|V(t_i) \cap U\| < w$): Für $i \in [d]$ sei $W_i = \bigcup_{j=1}^{i}(V(t_j) \cap U)$. Wir wählen $i_0 = \min\{i \in [d] \mid \|W_i\| > w\}$. Nun gilt $w < \|W_{i_0}\| < 2w$, da $\|V(t_{i_0}) \cap U\| < w$. Wir wählen $Y \subseteq W_{i_0}$ und $Z \subseteq U \setminus W_{i_0}$ mit $\|Y\| = \|Z\| = w + 1$. Dann ist $S = X_t$ ein $(Y \setminus S)$-$(Z \setminus S)$-Separator mit $\|S\| \leq w$.

Damit haben wir in beiden Fällen einen Widerspruch dazu hergeleitet, dass U $(w+1)$-verbunden ist. ■

Nun haben wir das Handwerkszeug, um den Algorithmus aus Satz 6.29 anzugeben. Die Idee dazu ist, schrittweise eine Teilmenge $U \subseteq V$ zu vergrößern und dabei eine Baumdekomposition (T, X) von $G[U]$ mit $w(T, X) < 4w$ zu berechnen. Um zu verhindern, dass der Algorithmus dabei in eine Sackgasse gerät, stellen wir zusätzliche Anforderungen. Für eine Komponente C von $G - U$ bezeichne $N_U(C)$ die Menge der Nachbarn von C in U, d.h. $N_U(C) = \{u \in U \mid \exists v \in C : \{u, v\} \in E\}$. Der Algorithmus wird für jede Komponente C von $G - U$ folgende Invariante erfüllen:

$$\|N_U(C)\| \leq 3w \text{ und } \exists t \in V_T : N_U(C) \subseteq X_t \qquad (*)$$

Prozedur `find-treedec`(V, E, w)

```
1 wähle U ⊆ V mit ‖U‖ = 3w
2 setze T = ({t}, ∅) und X_t = U
3 while U ≠ V do
4   finde eine Komponente C von G − U
5   // wegen (*) gilt N_U(C) ≤ 3w
6   finde t ∈ V_T mit N_U(C) ⊆ X_t // existiert wegen (*)
7   if ‖N_U(C)‖ < 3w then
8     wähle W ⊆ C mit ‖W‖ = 3w − ‖N_U(C)‖
```

```
    setze U = U ∪ W
    füge ein neues Blatt s neben t in T ein
    setze X_s = N_U(C) ∪ W
  else if N_U(C) ist nicht (w+1)-verbunden then
    seien Y, Z ⊆ N_U(C) und S ⊆ V die Mengen, die
      dies bezeugen, d.h. ‖S‖ < ‖Y‖ = ‖Z‖ ≤ w+1
      und S ist (Y \ S)-(Z \ S)-Separator
    sei W = S ∩ C
    setze U = U ∪ W
    füge ein neues Blatt s neben t in T ein
    setze X_s = N_U(C) ∪ W
  else
    return N_U(C) // Zertifikat für Weite > w
return (T, X)
```

Lemma 6.33. *Der Algorithmus* `find-treedec` *erfüllt die Invariante* (*).

Beweis. Es ist klar, dass (T, X) zu jedem Zeitpunkt eine Baumdekomposition von $G[U]$ ist. Während der Initialisierung und im ersten Fall werden nur Taschen der Größe $3w$ erzeugt. Für die im zweiten Fall erzeugten Taschen gilt wegen $\|N_U(C)\| = 3w$ und $\|W\| \leq \|S\| \leq w$, dass $\|N_U(C) \cup W\| \leq 4w$. Damit gilt $w(T, X) \leq 4w - 1$.

Für Komponenten, aus denen während eines Schleifendurchlaufs keine Knoten zu U hinzugefügt werden, gilt die Invariante danach unverändert. Sei C die während eines Schleifendurchlaufs gewählte Komponente von $G - U$, und sei C' eine Komponente von $G - (U \cup W)$. Im beiden Fällen gilt $N_{U \cup W}(C') \subseteq N_U(C) \cup W = X_s$. Im ersten Fall ist außerdem $\|X_s\| = 3w$, sodass (*) folgt. Im zweiten Fall gilt $N_{U \cup W}(C') \cap (Y \setminus S) = \emptyset$ oder $N_{U \cup W}(C') \cap (Z \setminus S) = \emptyset$, da es sonst einen Pfad von $Y \setminus S$ nach $Z \setminus S$ durch C' und damit in $G - S$ gäbe. Mit $\|S\| < \|Y\| = \|Z\|$ folgt daraus $\|N_{U \cup W}(C')\| < \|N_U(C)\| \leq 3w$. ■

Die Korrektheit von `find-treedec` folgt aus den Lemmas 6.32 und 6.33.

Die Laufzeitschranke ergibt sich daraus, dass die `while`-Schleife höchstens n mal durchlaufen wird (U wird jedes mal um mindestens einen Knoten vergrößert), eine Zusammenhangskomponente von $G - U$ in $O(m)$ Zeit gefunden werden kann, die Tasche X_t durch Traversieren von T in $O(wn)$ Zeit gefunden werden kann, und der Test auf $(w+1)$-Verbundenheit nach Lemma 6.31 in $f(w)m$ Zeit möglich ist. Zusammen ergibt sich (da wir voraussetzen dass G zusammenhängend ist) eine Laufzeit von $O(f(w) \cdot m \cdot n)$. Damit ist Satz 6.29 bewiesen.