Vorlesungsskript

Graphalgorithmen

Sommersemester 2013

Prof. Dr. Johannes Köbler

Humboldt-Universität zu Berlin
Lehrstuhl Komplexität und Kryptografie

31. Mai 2013

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

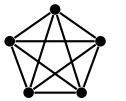
b) Der ***Grad*** *von v ist $\deg_G(v) = \|N_G(v)\|$.*

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 5.

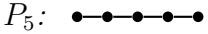
- *Der* ***vollständige Graph*** *(V, E) auf n Knoten, d.h. $\|V\| = n$ und $E = \binom{V}{2}$, wird mit K_n und der* ***leere Graph*** *$(V, \emptyset)$ auf n Knoten wird mit E_n bezeichnet.*

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.*

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

- *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 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 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

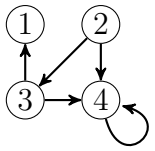
Sei $G = (V, E)$ ein Graph bzw. Digraph und sei $V = \{v_1, \ldots, v_n\}$. Dann ist die $(n \times n)$-Matrix $A = (a_{ij})$ mit den Einträgen

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

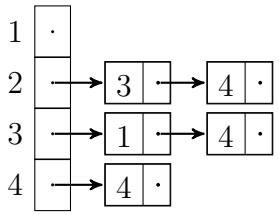
die ***Adjazenzmatrix*** von G. Für ungerichtete Graphen ist die Adjazenzmatrix symmetrisch mit $a_{ii} = 0$ für $i = 1, \ldots, n$.

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

Bemerkung 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 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

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

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.

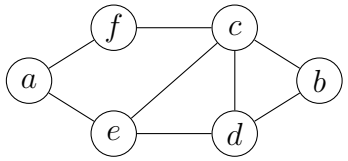
Klassifikation der Kanten eines (Di-)Graphen

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.

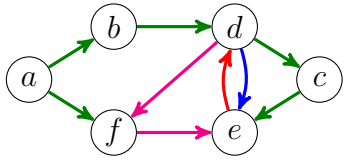
Beispiel 13. *Bei Aufruf mit dem Startknoten a könnte die Prozedur* `Explore` *den nebenstehendem Graphen beispielsweise wie folgt durchsuchen.*



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	-	-
$\emptyset$			

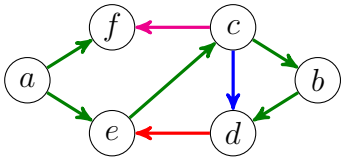
Dabei entsteht nebenstehender Suchwald.



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 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 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

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

und sei $\ell = \min\{i \geq 0 \mid u_{i+1} = \perp\}$. 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 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 $\texttt{parent}(w) = \perp$ ist, liefert `parent` einen w-v-Pfad (der Länge 0).

$k - 1 \rightsquigarrow k$**:** Sei $u = \texttt{parent}(v)$. Da u vor v besucht wird, liefert `parent` nach IV einen w-u-Pfad $P(u)$. Wegen $u = \texttt{parent}(v)$ 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 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 `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 18. *Sei G ein (ungerichteter) Graph.*

- *Der Algorithmus* `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 `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 `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 19. *Der Algorithmus* `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 `Explore`(w) sowohl in Graphen als auch in Digraphen alle von w aus erreichbaren Knoten. Als nächstes zeigen wir, dass `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.

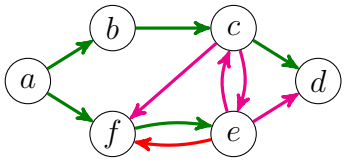
Algorithmus `BFS`(V, E)

```
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 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 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.

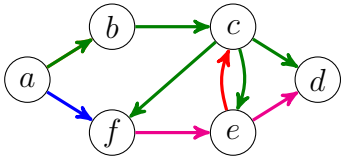
Algorithmus `DFS`(V, E)

```
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)
```

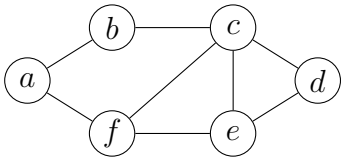

Beispiel 22. *Bei Aufruf mit dem Startknoten a generiert die Prozedur* `DFS-Explore` *nebenstehenden Tiefensuchwald.*

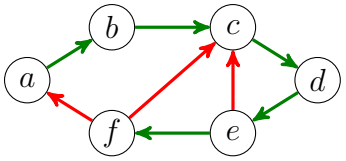


Keller S	bes. Knoten	bes. Kante	Typ
$a \leftrightarrow$	a	(a,b)	B
a, b	b	(b,c)	B
a, b, c	c	(c,d)	B
a, b, c, d	d	-	-
a, b, c	c	(c,e)	B
a, b, c, e	e	(e,c)	R
a, b, c, e	e	(e,d)	Q
a, b, c, e	e	-	-

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)

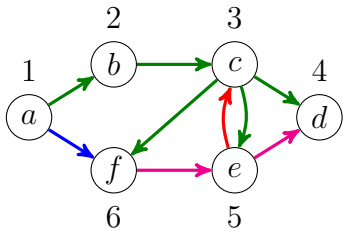
```
1  r := 0
2  for all v ∈ V,  e ∈ E  do
3    farbe(v) := weiß
4    vis(e) := false
5  for all u ∈ V  do
6    if farbe(u) = weiß then DFS-Explore(u)
```

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 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).

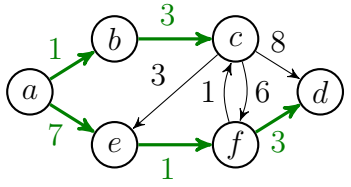
Voraussetzung für die Korrektheit des Algorithmus ist, dass alle Kanten eine nichtnegative Länge haben. Während der Suche werden bestimmte Kanten $e = (u, v)$ daraufhin getestet, ob $g(u) + \ell(u, v) < g(v)$ ist. Da in diesem Fall die Kante e auf eine Herabsetzung von $g(v)$ auf den Wert $g(u) + \ell(u, v)$ „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 24. *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 25. *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 $g(u) \geq \mathit{dist}(s,u)$ 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 $g(u) \leq \mathit{dist}(s,u)$.

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

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

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

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

■

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 26. *Bezeichne $\mathit{Init}(n)$, $\mathit{RemoveMin}(n)$ und $\mathit{Update}(n)$ 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*

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

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

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

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 $\texttt{parent}(v)$ vermerkt, so dass sich ein kürzester Weg von s zu allen erreichbaren Knoten (bzw. ein negativer Kreis) rekonstruieren lässt.

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

```
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.

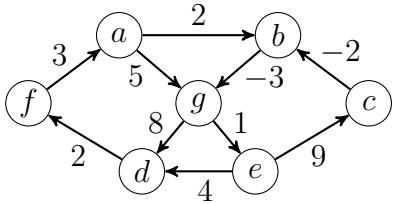
Algorithmus $\texttt{BFM}(V,E,l,s)$

```
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.

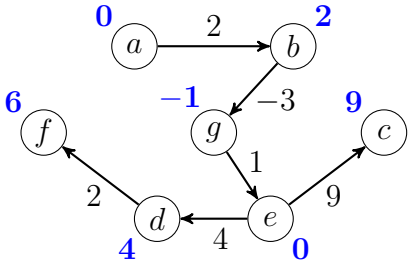
Beispiel 27. *Betrachte untenstehenden kantenbewerteten Digraphen mit dem Startknoten a.*



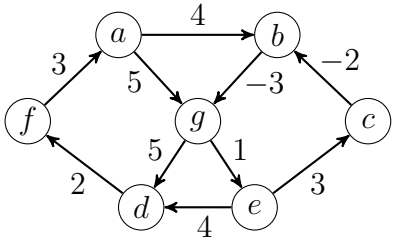
Die folgende Tabelle zeigt jeweils den Inhalt der Schlange Q, bevor der BFM-Algorithmus das nächste Paar (i,u) von Q entfernt. Dabei enthält jeder Eintrag (i,u,v,g) neben der Rundenzahl i und dem Knoten u auch noch den `parent`*-Knoten v und den g-Wert von u, obwohl diese nicht in Q gespeichert werden.*

⇑			
$(0,a,\perp,0)$			
⇑	$(1,b,a,2)$		
	$(1,g,a,5)$	$(1,g,b,-1)$	
			$(2,d,g,7)$
$(2,e,g,0)$			$(2,e,g,0)$
$(3,f,d,9)$	$(3,f,d,9)$		
	$(3,c,d,9)$	$(3,c,d,9)$	
	$(3,d,e,4)$	$(3,d,e,4)$	$(3,d,e,4)$
			$(4,f,d,6)$

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



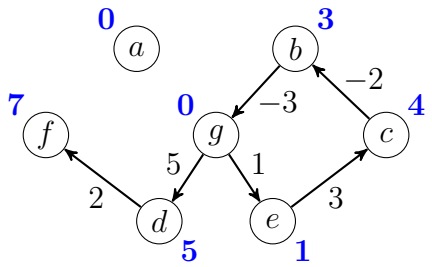
Als nächstes betrachten wir den folgenden Digraphen:



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

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

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

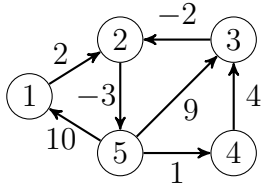
Der Algorithmus von Floyd-Warshall berechnet die Distanzen zwischen allen Knoten unter der Voraussetzung, dass keine negativen Kreise existieren.

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 $\mathtt{d}_k(i,j)$ 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 $\mathtt{d}_k$-Werte nur von den $\mathtt{d}_{k-1}$-Werten abhängen, ist der Speicherplatzbedarf $\mathcal{O}(n^2)$. Die Existenz negativer Kreise lässt sich daran erkennen, dass mindestens ein Diagonalelement $\mathtt{d}_k(i,i)$ einen negativen Wert erhält.

Beispiel 28. *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	∞

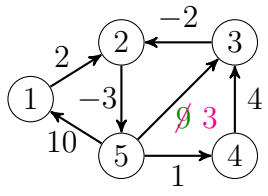
d_2	1	2	3	4	5
1	∞	2	∞	∞	-1
2	∞	∞	∞	∞	-3
3	∞	-2	∞	∞	-5
4	∞	∞	4	∞	∞
5	10	12	9	1	9

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 Matchings

Definition 29. *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.

Beispiel 30. *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 31. *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 32. *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 einer dieser Pfade zunehmend 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.

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 2 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 33. *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 36). ■

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. In den Übungen wird gezeigt, dass die Prozedur die Laufzeit $O(m)$ hat, woraus sich eine Gesamtlaufzeit von $O(nm)$ für den Edmonds-Algorithmus ergibt.

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 zustand(v′) = ungerade or
           inblüte(v) = inblüte(v′) ≠ ⊥ then // tue
           nichts
12   else if zustand(v′) = unerreicht then
13     zustand(v′) ← ungerade
14     parent(v′) ← v
15     root(v′) ← root(v)
16     depth(v′) ← depth(v) + 1
17     v″ ← partner(v′)
18     zustand(v″) ← gerade
19     parent(v″) ← v′
20     root(v″) ← root(v′)
21     depth(v″) ← depth(v′) + 1
22     for u ∈ N(v″) \ {v′} do Q ← Q ∪ {(v″, u)}
23   else // zustand(v′) = gerade
24     if root(v) =root(v′) then // v und v′ sind im
           gleichen Baum: kontrahiere Blüte
25       v″ ← tiefster gemeinsamer Vorfahr von v und v′
             // verwende depth(v) und depth(v′)
26       b ← neuer Knoten
27       blüte(b) ← (v″, . . . , v, v′, . . . , v″) // setze die
           beiden Pfade entlang der Baum-Kanten zu
           einem ungeraden Kreis zusammen
28       parent(b) ← parent(v″)
29       root(b) ← root(v″)
30       depth(b) ← depth(v″)
31       for u ∈ blüte(b) do
32         inblüte(u) ← b
33         if zustand(u) = ungerade then
34           zustand(u) ← gerade
35           for w ∈ N(u) do Q ← Q ∪ {(u, w)}
36 else // vergrößernder Pfad gefunden, muss noch
       expandiert werden
37   P ← leere doppelt verkettete Liste
38   u ← v
39   while u ≠ ⊥ do
40     while inblüte(u) ≠ ⊥ do u ← inblüte(u)
41     hänge u vorne an P an
42     u ← parent(u)
43   u ← v′
44   while u ≠ ⊥ do
45     while inblüte(u) do u ← inblüte(u)
46     hänge u hinten an P an
47     u ← parent(u)
48   u ← der erste Knoten auf P
49   while u ≠ ⊥ do
50     if blüte(u) = ⊥ then
51       u ← succ_P(u)
52     else // blüte(u) = (v_0, . . . , v_k) mit v_0 = v_k
53       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
54       u ← der erste Knoten dieses Pfads
55   return P
```

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

Definition 34. *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 35. *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 36. *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 im Fall,

1. dass auch v' gerade ist, e entweder zur Kontraktion einer weiteren Blüte oder zu einem vergrößernden Pfad führen
2. 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 $weight(S) = \|M\|$. ■

Korollar 37. *Für jeden Graphen G gilt*

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

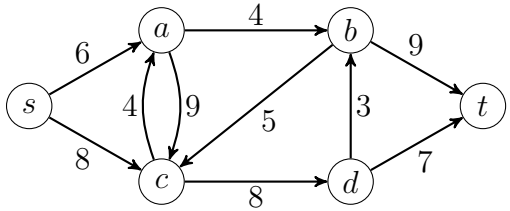
Ein Spezialfall hiervon ist der Satz von König für bipartite Graphen (siehe Übungen).

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.

4 Flüsse in Netzwerken

Definition 38. *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 39.

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

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

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

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

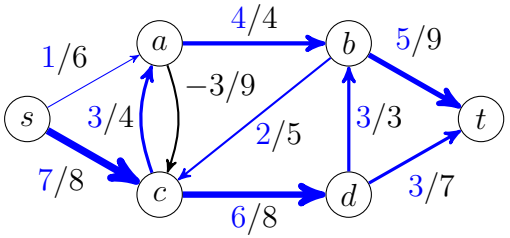
d) Der ***Fluss durch u*** *ist* $f(u) = \max\{f^+(u), f^-(u)\}$.

e) Der ***Nettofluss in u*** *ist* $f^-(u) - f^+(u)$.

f) Der ***Nettofluss aus u*** *ist* $f^+(u) - f^-(u)$.

g) 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

4.1 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.

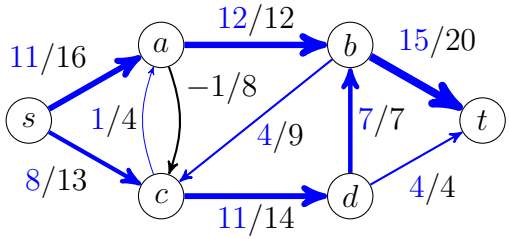
Definition 40. *Sei* $N = (V, E, s, t, c)$ *ein Netzwerk und sei* f *ein Fluss in* N. *Das zugeordnete* ***Restnetzwerk*** *ist* $N_f = (V, E_f, s, t, c_f)$ *mit der Kapazität*

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

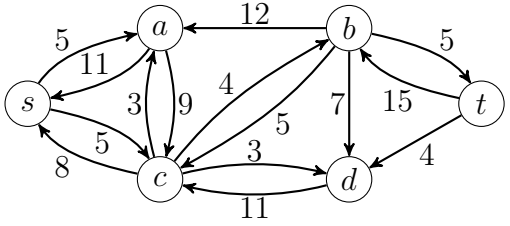
und der Kantenmenge

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

Zum Beispiel führt der Fluss



auf das folgende Restnetzwerk N_f:



Definition 41. *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*

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

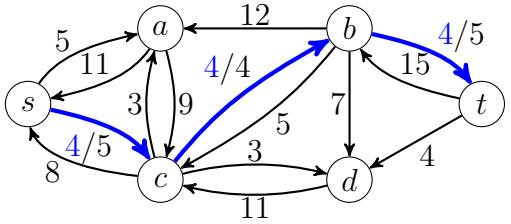
und der ***zu P gehörige Fluss in N_f*** *ist*

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

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

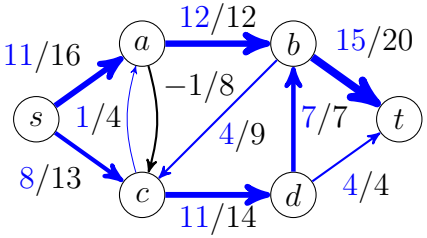
- $u_0 = s$ und $u_k = t$ ist,
- die Knoten $u_0, \ldots, u_k$ paarweise verschieden sind
- und $c_f(u_i, u_{i+1}) > 0$ für $i = 0, \ldots, k-1$ ist.

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

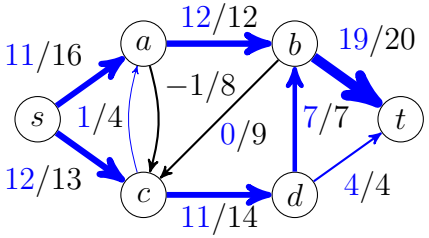


Es ist leicht zu sehen, dass f_P tatsächlich ein Fluss in N_f ist. Durch Addition der beiden Flüsse f und f_P erhalten wir einen Fluss $f' = f + f_P$ in N der Größe $|f'| = |f| + |f_P| > |f|$.

Fluss f:



Fluss $f + f_P$:

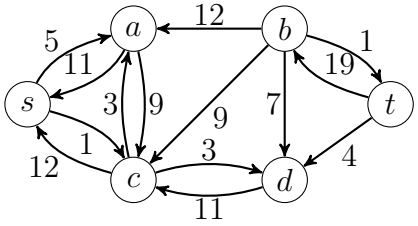


Nun können wir den **Ford-Fulkerson-Algorithmus** angeben.

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 42. *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 43. *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* ***Schnitt****; oft wird auch einfach S als Schnitt bezeichnet). Die* ***Kapazität eines Schnittes S*** *ist*

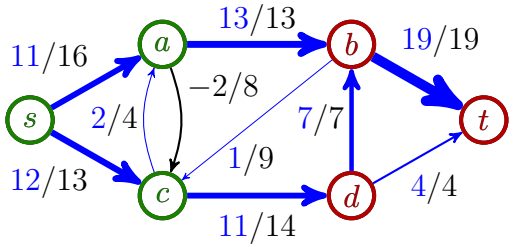
$$c^+(S) = \sum_{(u,v)\in E^+(S)} c(u, v).$$

Ist f ein Fluss in N, so heißt

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

der ***Fluss durch den Schnitt S****.*

Beispiel 44. *Betrachte den Schnitt $S = \{s, a, c\}$ in folgendem Netzwerk N mit dem Fluss f:*



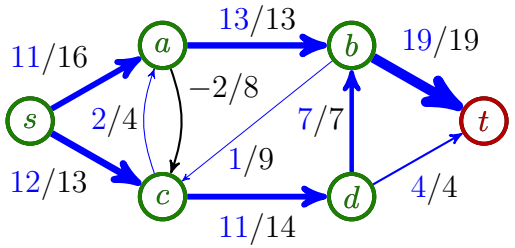
Dieser Schnitt hat die Kapazität

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

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 45. *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 $f^+(s) = f^+(S)$ zeigen wir durch Induktion über $k = \|S\|$.

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

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

$k - 1 \rightsquigarrow k$**:** 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

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

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

$$f^+(S) \;=\; \sum_{(u,v) \in E^+(S)} f(u,v) \;\leq\; \sum_{(u,v) \in E^+(S)} c(u,v) \;=\; c^+(S).$$

■

Satz 46 (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 mit $c^+(S) = |f|$.*

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

Für die Implikation „*2* ⇒ *3*“ betrachten wir den Schnitt

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

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

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

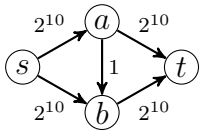
Die Implikation „*3* ⇒ *1*“ ist wiederum klar, da im Fall $c^+(S) = |f|$ für jeden Fluss f' die Abschätzung $|f'| = f'^+(S) \leq c^+(S) = |f|$ gilt. ■

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

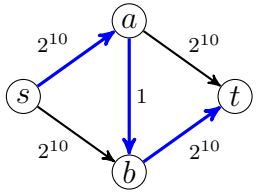
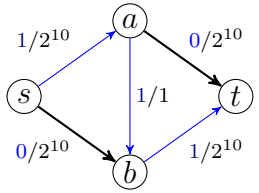
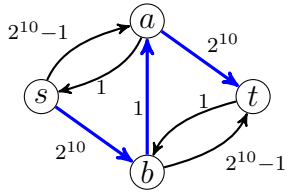
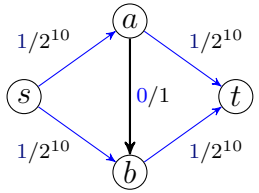
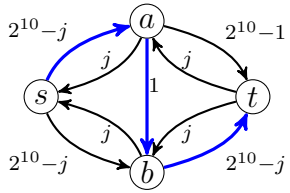
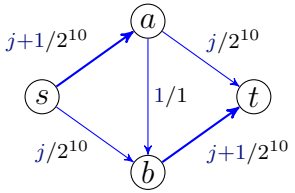
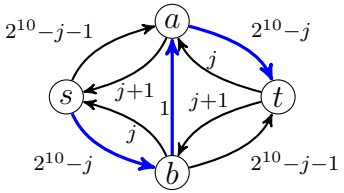
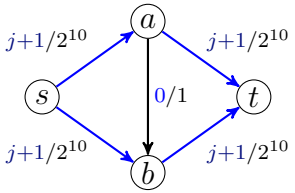
Sei $c_0 = c^+(s)$ die Kapazität des Schnittes $S = \{s\}$. Dann durchläuft der Ford-Fulkerson-Algorithmus die while-Schleife höchstens c_0-mal. Bei jedem Durchlauf ist zuerst das Restnetzwerk N_f und danach ein Zunahmepfad in N_f zu berechnen.

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.

4.2 Der Edmonds-Karp-Algorithmus

Der Edmonds-Karp-Algorithmus beschränkt die Suche nach P auf kürzeste Zunahmepfade. Ansonsten ist er mit dem Ford-Fulkerson-Algorithmus identisch.

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,  e ∈ E ∪ E^R do
2     vis(v) := vis(e) := 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     if ∃ e = uv ∈ E ∪ E^R : vis(e) = false then
10       vis(e) := true
11       if c(e) − f(e) > 0 ∧ vis(v) = false then
12          c'(e) := c(e) − f(e)
13          vis(v) := true
14          parent(v) := u
15          Enqueue(Q, v)
16    else Dequeue(Q)
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 47. *Der Edmonds-Karp-Algorithmus durchläuft die repeat-Schleife höchstens $nm/2$-mal.*

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 veringern können. Sei $d_i(u, v)$ die minimale Länge eines Pfades von u nach v im Restnetzwerk $N_{f_{i-1}}$.

Behauptung 48. *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 49. *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 49 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 48 bewiesen ist. Als nächstes zeigen wir folgende Behauptung.

Behauptung 50. *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 48:

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

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 51. *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 48 und Behauptung 50 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. ■

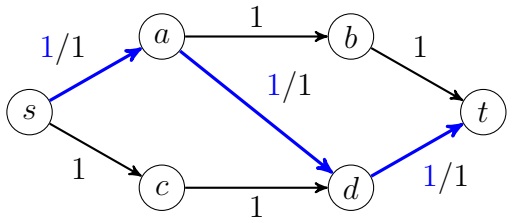
Man beachte, dass der Beweis auch bei Netzwerken mit reellen Kapazitäten seine Gültigkeit behält.

4.3 Der Algorithmus von Dinic

Man kann zeigen, dass sich in jedem Netzwerk ein maximaler Fluss durch Addition von höchstens m Zunahmepfaden P_i konstruieren lässt. Es ist nicht bekannt, ob sich jeder solche Pfad P_i in Zeit $O(n+m)$ bestimmen lässt. 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 $O(n^3)$ sogar noch schneller.

Definition 52. *Sei $N = (V, E, s, t, c)$ ein Netzwerk und sei g ein Fluss in N. 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 $j = 0, \ldots, \max\{d(s,u) \mid d(s,u) < d(s,t)\}$ 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 53. *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 54. *Für jeden Knoten $u \in V$ gilt $d_{i+1}(s,u) \geq d_i(s,u)$.*

Hierzu zeigen wir folgende Behauptung.

Behauptung 55. *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 55 bewiesen und Behauptung 54 folgt wie im Beweis von Satz 47. Als nächstes zeigen wir folgende Behauptung.

Behauptung 56. *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 54, dass $d_i(s,u_j) \leq d_{i+1}(s,u_j) = j$ für $j = 0, \ldots, h$ ist.

Wir betrachten 2 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 55 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 54) und $d_i(s,u_j)+1 \geq d_i(s,u_{j+1})$ (Behauptung 55)

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

■

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, e ∈ E ∪ E^R do
2     niv(v) := n
3     vis(e) := false
4  niv(s) := 0
5  V' := {s}
6  E' := ∅
7  QueueInit(Q)
8  Enqueue(Q, s)
```



```
9  while ¬QueueEmpty(Q) ∧ Head(Q) ≠ t do
10    u := Head(Q)
11    if ∃ e = uv ∈ E ∪ E^R : vis(e) = false then
12       vis(e) := true
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          niv(v) := niv(u) + 1
18          Enqueue(Q, v)
19    else Dequeue(Q)
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 g(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 57. *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*

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

Die ***Kapazität*** *(auch* ***Durchsatz*** *genannt) von u ist*

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

Ein Fluss g in N ***sättigt*** *einen Knoten $u \in V$, falls $g(u) = c(u)$ ist.*

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

Proposition 58. *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 $g = 0$ und dem Durchsatz $D(u) = c'(u)$ für jeden Knoten u im Schichtnetzwerk N'_f wählt die Prozedur `blockfluss2` in jedem Durchlauf der repeat-Schleife einen Knoten u mit minimalem Durchsatz $D(u)$ und erhöht den aktuellen Fluss g um den Wert $D(u)$. 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 ∈ V × V 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 $n-1$ Iterationen einer der beiden Knoten s oder t als Knoten u mit minimalem Durchsatz $D(u)$ gewählt und die repeat-Schleife verlassen. Da nach Beendigung des letzten Durchlaufs der Durchsatz von s oder von t gleich 0 ist, wird einer dieser beiden Knoten zu diesem Zeitpunkt von g gesättigt. Nach Proposition 58 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 die Kapazität $D(v)$ jedes Knoten v ausreicht, um den für ihn ermittelten Zusatzfluss in Höhe von $d(v)$ weiterzuleiten.

Prozedur `propagierevor`(u)

```
1  for all v ∈ V' do d(v) := 0
2  d(u) := D(u)
3  QueueInit(Q); Enqueue(Q,u)
4  while v := Dequeue(Q) ≠ ⊥ do
5     while d(v) ≠ 0 ∧ ∃e = vu ∈ E' do
6        m := min{d(v), c'(e)}
7        d(v) := d(v) − m; d(u) := d(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 `propagierevorück` 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 `propagierevorück` 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 $d(v)$ durch einen Knoten v auf 0 sinkt (falls $m = d(v)$ 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 `propagierevorück`.

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

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

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

sein, woraus

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

folgt. Damit ist die Anzahl a der Schleifendurchläufe durch

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

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

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

(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. 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 durch

die k_j Kanten in $E^+_{f_{i-1}}(S_j) \cap E^-_{f_{i-1}}(S_{j+1})$ für $j = 0, \ldots, d_i - 1$, fließt, muss

$$r_i/C \leq k_j \leq \|S_j\| \|S_{j+1}\|$$

sein. Somit enthält mindestens eine von 2 benachbarten Schichten S_j und S_{j+1} mindestens $\sqrt{r_i/C}$ Knoten, woraus

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

folgt. Damit ist die Anzahl a der Schleifendurchläufe durch

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

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

$$a \leq (2n\sqrt{C})^{2/3} + 2n\sqrt{C}/(2n\sqrt{C})^{1/3} = (2^5 C n^2)^{1/3}.$$

■

Korollar 61. *Sei $N = (V, E, s, t, c)$ ein Netzwerk.*

(i) Falls jeder Knoten $u \in V \setminus \{s, t\}$ einen Durchsatz $c(u) \leq C$ 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 $c(e) \leq C$ 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 $c'_g(P) \geq 1$, da der Fluss g durch diese Knoten um diesen Wert steigt. Daher kann jeder Knoten an maximal C Flusserhöhungen beteiligt sein, bevor sein Durchsatz auf 0 sinkt. Da somit pro Knoten ein Zeitaufwand von $O(C)$ für alle erfolgreichen Tiefensuchschritte, die zu einem s-t-Pfad führen, und zusätzlich pro Kante ein Zeitaufwand von $O(1)$ für alle nicht erfolgreichen Tiefensuchschritte anfällt, läuft `blockfluss1` in Zeit $O(nC + m)$.

(ii) Jedesmal wenn `blockfluss1` einen s-t-Pfad P im Schichtnetzwerk findet, verringert sich die Kapazität $c''(e)$ der auf P liegenden Kanten e um den Wert $c'_g(P) \geq 1$. 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 $O(Cm + m) = O(Cm)$.

■

Durch eine einfache Reduktion des bipartiten Matchingproblems auf ein Flussproblem erhält man folgendes Resultat (siehe Übungen).

Korollar 62. *In einem bipartiten Graphen lässt sich ein maximales Matching in Zeit $O(\sqrt{\mu(G)}m)$ bestimmen.*

4.4 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 2 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).

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

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 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 63 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 Kreis K_i in $N_{f_{i-1}}$ mit negativen Kosten $k(K_i) < 0$. 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.

*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.

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 Kreise K mit negativen Kosten $k(K) < 0$ gibt.

Satz 64. *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 alle Kreise in $N_{f_{i-1}}$ nichtnegative Kosten haben, folgt

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

■

Basierend auf Satz 64 können wir nun leicht einen Algorithmus zur Bestimmung eines maximalen Flusses mit minimalen Kosten in einem Netzwerk N angeben, falls es in N keine Kreise K mit negativen Kosten $k(K) < 0$ 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 Kreise mit negativen Kosten 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.