Graphalgorithmen

Johannes Köbler

Institut für Informatik
Humboldt-Universität zu Berlin

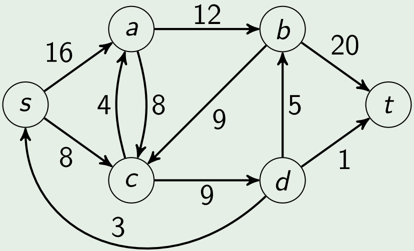
SS 2021

Flüsse in Netzwerken

Definition

- 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 $c : V \times V \to \mathbb{N}$
- Dabei muss jede Kante $(u, v) \in E$ eine Kapazität $c(u, v) > 0$ und jede Nichtkante $(u, v) \notin E$ muss die Kapazität $c(u, v) = 0$ haben

Beispiel. Die folgende Abbildung zeigt ein Netzwerk N.



Flüsse in Netzwerken

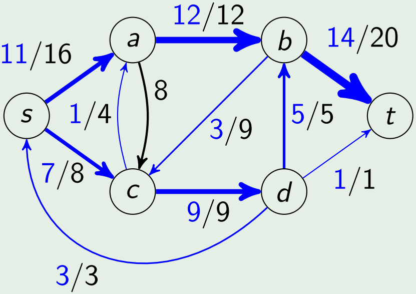
Definition. Sei $N = (V, E, s, t, c)$ ein Netzwerk.

- 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)
- Die Größe von f ist $|f| = \sum_{v \in V} f(s, v)$
- Der Fluss in den Knoten u ist $f^-(u) = \sum_{v \in V} \max\{0, f(v, u)\}$
- Der Fluss aus u ist $f^+(u) = \sum_{v \in V} \max\{0, f(u, v)\}$

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

Beispiel (Fortsetzung).

- Die Abbildung zeigt einen Fluss f in N:



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

- Dieser hat die Größe $|f| = \sum_{v \in V} f(s, v) = 11 + 7 - 3 = 15$

Der Ford-Fulkerson-Algorithmus

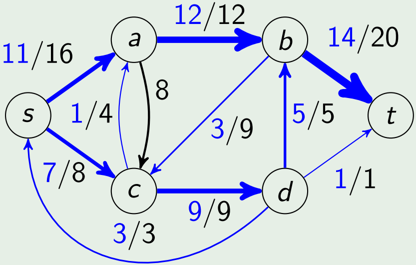
- Wie kann man für einen Fluss f in einem Netzwerk N entscheiden, ob er vergrößert werden kann?
- Diese Frage ist leicht zu beantworten, falls f auf $V \times V$ den Wert 0 hat
- In diesem Fall genügt es, in $G = (V, E)$ einen s-t-Pfad zu finden
- Andernfalls können wir ein Netzwerk N_f konstruieren,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

Definition

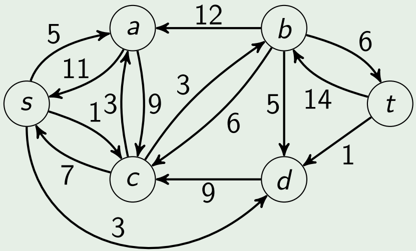
- Sei $N = (V, E, s, t, c)$ ein Netzwerk und sei f ein Fluss in N
- Das zugeordnete Restnetzwerk ist $N_f = (V, E_f, s, t, c_f)$ mit
  - den Kapazitäten $c_f(u, v) = c(u, v) - f(u, v)$ und
  - der Kantenmenge $E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\}$

Beispiel (Fortsetzung). Der Fluss f führt auf das folgende Restnetzwerk

Fluss f:



Restnetzwerk N_f:



Flüsse in Netzwerken

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

- Dann heißt jeder s-t-Pfad P in (V, E) Zunahmepfad in N
- Die Kapazität von P in N ist

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

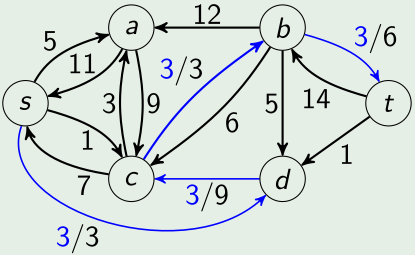
- und der Fluss durch P in N ist

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

- Es ist leicht zu sehen, dass f_P tatsächlich ein Fluss in N ist
- Ein Pfad $P = (u_0, \ldots, u_k)$ ist also ein Zunahmepfad in N, falls
  - $u_0 = s$ und $u_k = t$ ist
  - die Knoten $u_0, \ldots, u_k$ paarweise verschieden sind, und
  - $c(u_i, u_{i+1}) > 0$ für $i = 0, \ldots, k - 1$ gilt

Beispiel (Fortsetzung).

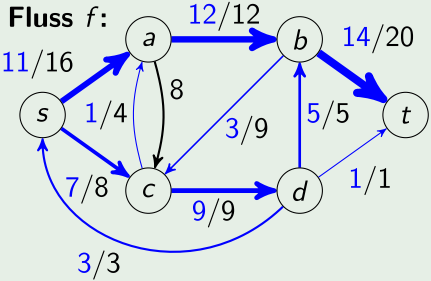
Die folgende Abbildung zeigt den Zunahmepfad $P = (s, d, c, b, t)$ in N_f mit der Kapazität $c_f(P) = 3$ und den zugehörigen Fluss f_P in N_f



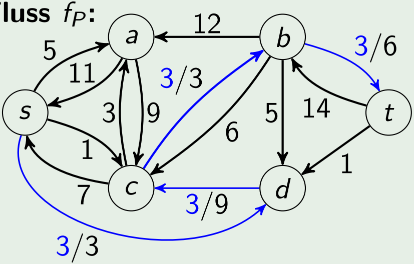
Durch Addition der beiden Flüsse f und f_P erhalten wir einen Fluss $f' = f + f_P$ in N der Größe $|f'| = |f| + |f_P| = |f| + c_f(P) > |f|$

Beispiel (Fortsetzung).

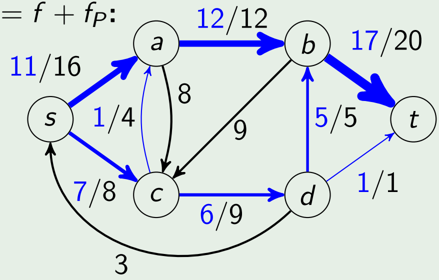
Fluss f**:**



Fluss f_P**:**



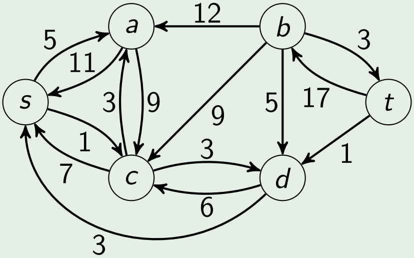
Fluss $f' = f + f_P$**:**



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

for all $(u, v) \in E \cup E^R$ **do**
  $f(u, v) := 0$
while es gibt einen Zunahmepfad P in N_f **do**
  $f := f + f_P$

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



In diesem existiert kein Zunahmepfad mehr ◁

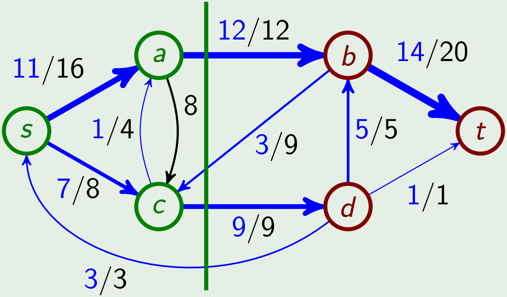
Um zu zeigen, dass der Algorithmus von Ford-Fulkerson tatsächlich einen Maximalfluss berechnet, weisen wir nach, dass f maximale Größe in N hat, wenn im Restnetzwerk N_f kein Zunahmepfad existiert

Definition. Sei $N = (V, E, s, t, c)$ ein Netzwerk und sei f ein Fluss in N.

- Eine Menge S mit $\emptyset \subsetneq S \subsetneq V$ heißt Schnitt durch N
- Der zugehörige Kantenschnitt ist $E(S) = \{(u, v) \in E \mid u \in S, v \notin S\}$ (dieser wird oft auch einfach als Schnitt bezeichnet)
- Die Kapazität eines Schnittes S ist $c(S) = \sum_{e \in E(S)} c(e)$
- Der Fluss durch den Schnitt S ist $f(S) = \sum_{e \in E(S)} f(e)$
- Ist $u \in S$ und $v \notin S$, so wird S auch als u-v-Schnitt bezeichnet

Beispiel

- Betrachte folgenden s-t-Schnitt $S = \{s, a, c\}$ durch das Netzwerk N mit dem Fluss f:

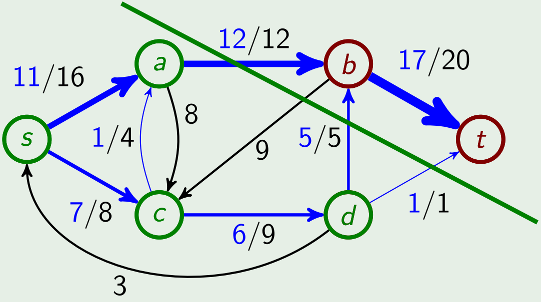


- Die Kapazität von S ist $c(S) = c(a, b) + c(c, d) = 12 + 9 = 21$ und die Größe des Flusses f durch den Schnitt S ist

$$f(S) = f(a, b) + f(c, b) + f(c, d) + f(s, d) = 12 - 3 + 9 - 3 = 15$$

Beispiel (Fortsetzung).

- Dagegen hat der s-t-Schnitt $S' = \{s, a, c, d\}$ durch das Netzwerk N mit dem Fluss f'



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

- Diese stimmt mit der Größe des Flusses f' durch den Schnitt S' überein:

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

◁

Lemma

Für jeden Fluss f in einem Netzwerk N und jeden s-t-Schnitt S durch N gilt $|f| = f(S) \leq c(S)$

Beweis.

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

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

- Die Gleichheit $|f| = f(S)$ zeigen wir durch Induktion über $k = |S|$
  - Im Fall $k = 1$ (IA) ist $S = \{s\}$ und wegen $f(s,s) = 0$ folgt

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

Beweis (Fortsetzung).

- Für den IS sei S ein s-t-Schnitt mit $|S| = k \geq 2$ und sei S' der Schnitt $S - \{w\}$, wobei wir $w \in S - \{s\}$ beliebig wählen
- Nach IV gilt dann $|f| = f(S')$ und wegen $S = S' \cup \{w\}$ folgt

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

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

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

- Daher erhalten wir

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

- Also gilt $f(S) = f(S') = |f|$

$\square$

Der Ford-Fulkerson-Algorithmus

Satz (Max-Flow-Min-Cut-Theorem)

Für einen Fluss f in einem Netzwerk $N = (V, E, s, t, c)$ sind folgende Aussagen äquivalent:

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

Beweis.

- Die Implikation 1 $\Rightarrow$ 2 ist klar, da die Existenz eines Zunahmepfads in N_f zu einer Vergrößerung von f führen würde
- Für die Implikation 2 $\Rightarrow$ 3 betrachten wir den Schnitt

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

  - Dann gilt $t \notin S$ (da in N_f kein Zunahmepfad existiert)

Der Ford-Fulkerson-Algorithmus

Beweis (Fortsetzung).

- Für die Implikation ❷ $\Rightarrow$ ❸ betrachten wir den Schnitt

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

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

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

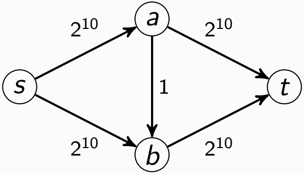
- Die Implikation ❸ $\Rightarrow$ ❶ folgt direkt aus obigem Lemma, da jeder Fluss f' in N im Fall $c(S) = |f|$ einen Wert $|f'| = f'(S) \leq c(S) = |f|$ hat □

Das Max-Flow-Min-Cut-Theorem gilt auch für Netzwerke mit beliebigen reellen Kapazitäten $c(e) \geq 0$

Der Ford-Fulkerson-Algorithmus

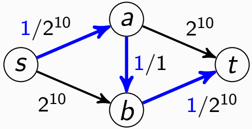
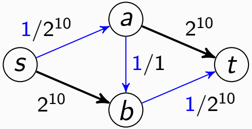
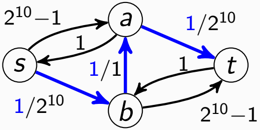
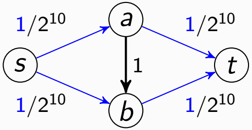
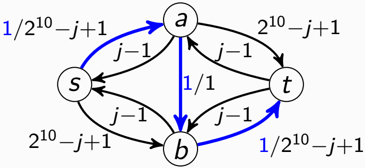
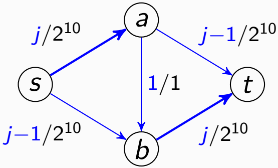
- Ist $c_0 = c(S)$ die Kapazität des Schnittes $S = \{s\}$, so durchläuft der Ford-Fulkerson-Algorithmus die while-Schleife höchstens c_0-mal, da sich der aktuelle Fluss in jedem Durchlauf um mindestens 1 erhöht
- Bei jedem Durchlauf ist zuerst das Restnetzwerk N_f und danach ein Zunahmepfad in N_f zu berechnen
  - Die Berechnung des Zunahmepfads P kann durch Breitensuche in Zeit $\mathcal{O}(n + m)$ erfolgen
  - Da sich das Restnetzwerk nur entlang von P ändert, kann es in Zeit $\mathcal{O}(n)$ aktualisiert werden
- Jeder Durchlauf benötigt also Zeit $\mathcal{O}(n + m)$, was auf eine Gesamtlaufzeit von $\mathcal{O}(c_0(n + m))$ führt
- Da der Wert von c_0 jedoch exponentiell in der Länge der Eingabe (also der Beschreibung des Netzwerkes N) sein kann, ergibt dies keine polynomiell beschränkte Laufzeit
- Bei Netzwerken mit reellen Kapazitäten kann Ford-Fulkerson sogar unendlich lange laufen (siehe Übungen)

Der Ford-Fulkerson-Algorithmus

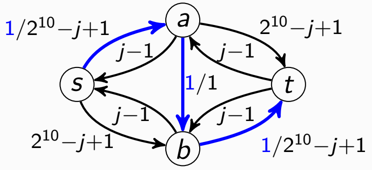
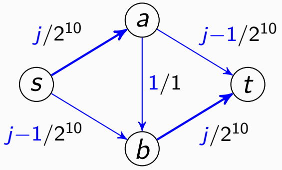
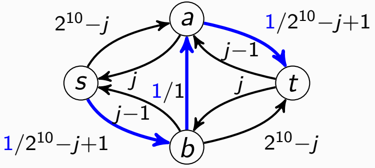
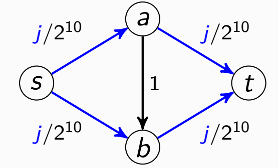
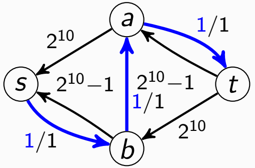
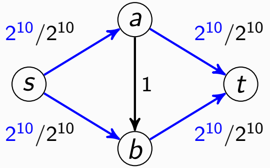


- Bei nebenstehendem Netzwerk benötigt Ford-Fulkerson zur Bestimmung des Maximalflusses abhängig von der Wahl der Zunahmepfade zwischen 2 und 2^{11} Schleifendurchläufe
- Im günstigsten Fall wird nämlich ausgehend vom Nullfluss f_0 zuerst der Zunahmepfad $P_1 = (s, a, t)$ mit der Kapazität 2^{10} und dann im Restnetzwerk N_{f_1} der Pfad $P_2 = (s, b, t)$ mit der Kapazität 2^{10} gewählt
- Im ungünstigsten Fall werden abwechselnd die beiden Zunahmepfade $P_1 = (s, a, b, t)$ und $P_2 = (s, b, a, t)$ (also $P_i = P_1$ für ungerades i und $P_i = P_2$ für gerades i) mit der Kapazität 1 gewählt. Dies führt auf insgesamt 2^{11} Schleifendurchläufe (siehe folgende Tabelle)

Der Ford-Fulkerson-Algorithmus

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

Der Ford-Fulkerson-Algorithmus

<table>
<tr>
<td>$2j-1$,
$1 < j \le 2^{10}$</td>
<td>
</td>
<td>
</td>
</tr>
<tr>
<td>$2j$,
$1 < j < 2^{10}$</td>
<td>
</td>
<td>
</td>
</tr>
<tr>
<td>$\vdots$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2^{11}</td>
<td>
</td>
<td>
</td>
</tr>
</table>

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 den Kapazitäten)
  - Diesen werden wir als nächstes betrachten

Der Ford-Fulkerson-Algorithmus

- Dinitz hatte die Idee, in jeder Iteration einen blockierenden Fluss g in N_f zu berechnen
  - Dieser benutzt nur Kanten, die auf einem kürzesten s-t-Pfad in N_f liegen, und sättigt auf jedem kürzesten s-t-Pfad mindestens eine Kante e (d.h. $g(e) = c_f(e)$), die dann in der nächsten Iteration fehlt
  - Da die Länge der kürzesten s-t-Pfade im Restnetzwerk in jeder Iteration um mindestens eins zunimmt, liegt nach spätestens $n - 1$ Iterationen ein maximaler Fluss vor
  - Dinitz hat gezeigt, dass der Fluss g in Zeit $\mathcal{O}(mn)$ bestimmt werden kann, was auf eine Laufzeit von $\mathcal{O}(n^2 m)$ führt
- Malhotra, Kumar und Maheswari fanden später einen Ansatz, den Fluss g in Zeit $\mathcal{O}(n^2)$ zu berechnen, wodurch sich die Laufzeit auf $\mathcal{O}(n^3)$ verbessern ließ

Der Edmonds-Karp-Algorithmus

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

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

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

Prozedur `addierepfad`(f, P)

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

Der Edmonds-Karp-Algorithmus

Prozedur `zunahmepfad`(f)

```
for all v ∈ V do
  parent(v) := ⊥
Q := (s)
while Q ≠ () ∧ parent(t) = ⊥ do
  u := dequeue(Q)
  for all e = (u, v) ∈ E ∪ E^R do
    if c(e) − f(e) > 0 ∧ parent(v) = ⊥ then
      c′(e) := c(e) − f(e)
      parent(v) := u
      enqueue(Q, v)
if parent(t) = ⊥ then
  P := ⊥
else
  P := parent-Pfad von s nach t
  c_f(P) := min{c′(e) | e ∈ P}
return P
```

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

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

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

Der Edmonds-Karp-Algorithmus

Satz

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

Beweis.

- Sei k die Anzahl der Schleifendurchläufe und seien $P_1, \ldots, P_k$ die Zunahmepfade, die der Algorithmus bei Eingabe N berechnet
- Es gilt also $f_{i+1} = f_i + f_{P_{i+1}}$, wobei f_0 der triviale Nullfluss und P_{i+1} der im Restnetzwerk N_{f_i} berechnete Zunahmepfad ist
- Eine Kante e auf P_{i+1} heißt kritisch für P_{i+1}, falls der Fluss $f_{P_{i+1}}$ in N_{f_i} die Kante e sättigt, d.h. $f_{P_{i+1}}(e) = c_{f_i}(e)$
- Eine kritische Kante e für P_{i+1} ist wegen

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

nicht in $N_{f_{i+1}}$ enthalten, wohl aber die Kante e^R:

$$c_{f_{i+1}}(e^R) = c(e^R) - f_{i+1}(e^R) = c(e^R) + f_{i+1}(e) = c(e^R) + c(e) > 0$$

Der Edmonds-Karp-Algorithmus

Beweis (Fortsetzung).

- Sei $d_i(u, v)$ die minimale Länge eines Pfades von u nach v im Restnetzwerk N_{f_i} und sei $\ell_{i+1} = d_i(s, t)$ die Länge von P_{i+1}
- Für den Rest des Beweises benötigen wir folgende drei Behauptungen

Behauptung 1.

Für jeden Knoten $u \in V$ gilt $d_i(s, u) \leq d_{i+1}(s, u)$ und $d_i(u, t) \leq d_{i+1}(u, t)$

Behauptung 2. Sei $e = (u, v)$ eine Kante auf dem Pfad P_{i+1}.

Falls e^R für ein $j > i$ auf dem Pfad P_{j+1} liegt, dann ist $\ell_{j+1} \geq \ell_{i+1} + 2$

Behauptung 3.

Seien $P_{i_1}, \ldots, P_{i_h}$ mit $1 \leq i_1 < \cdots < i_h \leq k$ die Pfade, für die e oder e^R kritisch sind. Dann ist $h \leq n/2$.

Der Edmonds-Karp-Algorithmus

Behauptung 1.

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

Beweis von Behauptung 1.

- Hierzu beweisen wir für jeden kürzesten Pfad $P = (u_0, \ldots, u_\ell)$ von $u_0 = s$ nach $u_\ell = u$ in $N_{f_{i+1}}$ (d.h. $d_{i+1}(s,u) = \ell$) die Ungleichungen

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

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

Behauptung 2. Sei $e = (u, v)$ eine Kante auf dem Pfad P_{i+1}.

Falls e^R für ein $j > i$ auf dem Pfad P_{j+1} liegt, dann ist $\ell_{j+1} \geq \ell_{i+1} + 2$

Beweis von Behauptung 2.

- Da P_{i+1} ein kürzester s-t-Pfad der Länge $\ell_{i+1} = d_i(s,t)$ in N_{f_i} und P_{j+1} ein kürzester s-t-Pfad der Länge $\ell_{j+1} = d_j(s,t)$ in N_{f_j} ist, folgt dies direkt aus Behauptung 1:

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

□

Der Edmonds-Karp-Algorithmus

Behauptung 3.

Seien $P_{i_1}, \ldots, P_{i_h}$ mit $1 \leq i_1 < \cdots < i_h \leq k$ die Pfade, für die e oder e^R kritisch sind. Dann ist $h \leq n/2$.

Beweis von Behauptung 3.

- Wir zeigen zuerst, dass $\ell_{i_{j+1}} \geq \ell_{i_j} + 2$ für $j = 1, \ldots, h-1$ gilt
- Falls $e' \in \{e, e^R\}$ für ein $j \in \{1, \ldots, h-1\}$ kritisch für den Pfad P_{i_j} ist, dann fehlt e' im Restnetzwerk $N_{f_{i_j}}$
- Daher kann e' nur dann eine kritische Kante für den Pfad $P_{i_{j+1}}$ sein, wenn e'^R auf einem Pfad P_i mit $i_j < i \leq i_{j+1}$ liegt
- Dies gilt natürlich erst recht, wenn die Kante e'^R für $P_{i_{j+1}}$ kritisch ist
- Mit Behauptung 1 und Behauptung 2 folgt also $\ell_{i_{j+1}} \geq \ell_i \geq \ell_{i_j} + 2$
- 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 $\square$

Der Edmonds-Karp-Algorithmus

Beweis des Satzes (Schluss)

- Nach Behauptung 3 sind e und e^R für jede Kante $e \in E$ zusammen höchstens auf $n/2$ vielen Pfaden P_i kritisch
- Da $E \cup E^R$ höchstens m Kantenpaare der Form $\{e, e^R\}$ enthält, können die k Pfade $P_1, \ldots, P_k$ köchstens $mn/2$ kritische Kanten enthalten
- Da aber auf jedem Pfad P_i mindestens eine Kante kritisch ist, muss $k \leq mn/2$ sein $\square$

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

Der Algorithmus von Dinitz

- In den Übungen wird gezeigt, dass sich in jedem Netzwerk N ein maximaler Fluss durch Addition von höchstens m Zunahmepfaden konstruieren lässt
- Es ist aber nicht bekannt, ob sich solche Pfade in Zeit $O(m)$ bestimmen lassen
- Wenn ja, würde dies auf eine Gesamtlaufzeit von $O(m^2)$ führen
- Für dichte Netzwerke (d.h. $m = \Theta(n^2)$) hat der Algorithmus von Dinitz die gleiche Laufzeit $O(n^2 m) = O(n^4)$ und die verbesserte Version ist mit $O(n^3)$ in diesem Fall sogar noch schneller

Der Algorithmus von Dinitz

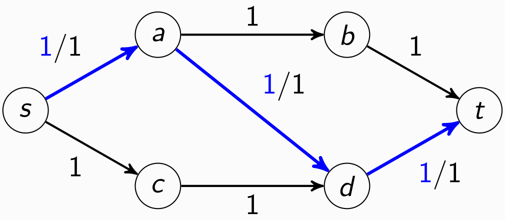
- Die Analyse der Laufzeit des Edmonds-Karp-Algorithmus benutzt die Tatsache, dass der Fluss durch den Zunahmepfad P_{i+1}, der in jedem Schleifendurchlauf auf den aktuellen Fluss f_i addiert wird, eine Kante auf *mindestens einem* kürzesten Pfad im Restnetzwerk N_{f_i} sättigt
- Dies hat zur Folge, dass nicht mehr als $mn/2$ Zunahmepfade P_i benötigt werden, um einen maximalen Fluss zu erhalten
- Dagegen addiert der Algorithmus von Dinitz in jedem Schleifendurchlauf auf den aktuellen Fluss f einen Fluss g, der auf *jedem* kürzesten Pfad im Restnetzwerk N_f mindestens eine Kante sättigt
- Wir werden sehen, dass maximal $n-1$ solche Flüsse g_i benötigt werden

Definition

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

Der Algorithmus von Dinitz

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



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

Der Algorithmus von Dinitz

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

Der Algorithmus von Dinitz

Definition

- Sei $N = (V, E, s, t, c)$ ein Netzwerk und bezeichne $d(x, y)$ die Länge eines kürzesten Pfades von x nach y in N
- Das zugeordnete Schichtnetzwerk ist $N' = (V', E', s, t, c')$ mit
  - der Knotenmenge $V' = S_0 \cup \cdots \cup S_\ell$
  - der Kantenmenge $E' = \bigcup_{j=1}^{\ell} \{(u, v) \in E \mid u \in S_{j-1} \land v \in S_j\}$ und
  - der Kapazitätsfunktion

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

wobei

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

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

Der Algorithmus von Dinitz

Algorithmus $\texttt{Dinitz}(N)$, $N = (V, E, s, t, c)$

for all $(u, v) \in E \cup E^R$ **do**
  $f(u, v) := 0$
while $S := \texttt{schichtnetzwerk}(N, f) \neq \perp$ **do**
  $f := f + \texttt{blockfluss}(S)$

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

Der Algorithmus von Dinitz

- Diese führt in N_f eine modifizierte Breitensuche mit Startknoten s durch und speichert dabei in der Menge E' nicht nur alle Baumkanten, sondern zusätzlich alle Querkanten (u, v) (d.h. u und v liegen nicht auf einem gemeinsamen parent-Pfad), die auf einem kürzesten Weg von s zu v liegen
- Die Suche bricht ab, sobald t am Kopf der Schlange erscheint oder alle von s aus erreichbaren Knoten abgearbeitet sind
- Falls t erreicht wird, werden außer der Senke t alle Knoten u, die in N_f einen Abstand $d(s, u) < d(s, t)$ von der Quelle s haben, in der Menge V' zusammengefasst
- Zudem werden alle Kanten aus E' wieder entfernt, die nicht zwischen zwei Knoten aus V' verlaufen
- Wird t dagegen nicht erreicht, so existiert in N_f (und damit in N'_f) kein (blockierender) Fluss g mit $|g| > 0$ und somit auch kein Zunahmepfad in N_f, d.h. f ist bereits maximal

Prozedur $\texttt{schichtnetzwerk}(N, f)$

```
E' := ∅
for all v ∈ V do niv(v) := n
niv(s) := 0; Q := (s)
while Q ≠ () ∧ head(Q) ≠ t do
  u := dequeue(Q)
  for all e = (u, v) ∈ E ∪ E^R do
    if c(e) − f(e) > 0 ∧ niv(v) > niv(u) then
      E' := E' ∪ {e}
      c'(e) := c(e) − f(e)
      if niv(v) > niv(u) + 1 then
        niv(v) := niv(u) + 1
        enqueue(Q, v)
if head(Q) = t then
  V' := {v ∈ V | niv(v) < niv(t)} ∪ {t}
  E' := E' ∩ (V' × V')
  return (V', E', s, t, c')
else return ⊥
```

Der Algorithmus von Dinitz

- Die Laufzeitschranke $O(n+m)$ für die Prozedur `schichtnetzwerk` folgt aus der Tatsache, dass jede Kante in $E \cup E^R$ höchstens einmal besucht wird und jeder Besuch mit einem konstanten Zeitaufwand verbunden ist
- Nun kommen wir zur Prozedur `blockfluss1`, die einen blockierenden Fluss g in einem Schichtnetzwerk $S = (V, E, s, t, c)$ berechnet
- Beginnend mit dem Nullfluss g bestimmt diese in der repeat-Schleife
  - mittels Tiefensuche einen s-t-Pfad P in S
  - addiert den Fluss f_P durch P in S zum aktuellen Fluss g hinzu
  - aktualisiert die Kapazitäten aller Kanten e auf dem Pfad P und
  - entfernt aus S die von g gesättigten Kanten
- Der gefundene Pfad P lässt sich hierbei direkt aus dem Inhalt des Kellers K rekonstruieren, weshalb wir ihn als K-Pfad bezeichnen
- Man beachte, dass die Kapazitäten der auf P liegenden Kanten nur in Vorwärtsrichtung und nicht wie bei Ford-Fulkerson und Edmonds-Karp auch in Rückwärtsrichtung angepasst werden

- Falls die Tiefensuche in einem Knoten $u \neq s$ in einer Sackgasse endet (weil E keine von u aus weiterführenden Kanten enthält), wird die zuletzt besuchte Kante (u', u) ebenfalls aus E entfernt und die Tiefensuche vom Startpunkt u' dieser Kante fortgesetzt (backtracking)
- Die Prozedur `blockfluss1` bricht ab, sobald alle Kanten mit Startknoten s aus E entfernt wurden und somit in (V, E) keine Pfade mehr von s nach t existieren (d.h. g ist ein blockierender Fluss in S)
- Die Laufzeitschranke $O(mn)$ für `blockfluss1` folgt aus der Tatsache, dass sich die Anzahl der aus E entfernten Kanten nach spätestens n Schleifendurchläufen um 1 erhöht

Prozedur $\texttt{blockfluss1}(S)$, $S = (V, E, s, t, c)$

for all $e \in E \cup E^R$ **do** $g(e) := 0$

$u := s$; $K := (s)$; $\texttt{done} := \texttt{false}$

repeat

  if $\exists e = (u, v) \in E$ **then**

    $\text{push}(K, v)$; $u := v$

  elsif $u = t$ **then**

    $P :=$ K-Pfad von s nach t

    $c(P) := \min\{c(e) \mid e \in P\}$

    for all $e \in P$ **do**

      $g(e) := g(e) + c(P)$; $g(e^R) := -g(e)$; $c(e) := c(e) - c(P)$

      if $c(e) = 0$ **then** $E := E \setminus \{e\}$

    $K := (s)$; $u := s$

  elsif $u \neq s$ **then** \\ backtracking

    $\text{pop}(K)$; $u' := \text{top}(K)$; $E := E \setminus \{(u', u)\}$; $u := u'$

  else $\texttt{done} := \texttt{true}$

until $\texttt{done}$

return g

Der Algorithmus von Dinitz

Satz

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

Beweis.

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

Beweis (Fortsetzung).

- Sei $P = (u_0, \dots, u_\ell)$ ein kürzester Pfad von $u_0 = s$ nach $u_\ell = u$ in $N_{f_{i+1}}$ (d.h. es gilt $d_{i+1}(s, u_h) = h$ für $h = 1, \dots, \ell$)
- Wir beweisen für $h = 1, \dots, \ell$ die folgenden (Un)gleichungen:

$$d_i(s, u_h) \le d_i(s, u_{h-1}) + 1, \text{ falls } (u_{h-1}, u_h) \in E_{f_i} \tag{1}$$
$$d_i(s, u_h) = d_i(s, u_{h-1}) - 1, \text{ falls } (u_{h-1}, u_h) \notin E_{f_i} \tag{2}$$

- Es ist klar, dass (1) gilt
- Falls die Kante $e = (u_{h-1}, u_h)$ nicht in N_{f_i} (aber in $N_{f_{i+1}}$) enthalten ist, muss $f_{i+1}(e) \ne f_i(e)$ und somit $g_{i+1}(e) \ne 0$ sein
- Da e dann auch nicht in N'_{f_i} ist, muss $e^R = (u_h, u_{h-1})$ in N'_{f_i} sein
- Da N'_{f_i} nur Kanten auf kürzesten Pfaden mit Startknoten s enthält, folgt $d_i(s, u_{h-1}) = d_i(s, u_h) + 1$, was (2) impliziert

Der Algorithmus von Dinitz

Beweis (Fortsetzung).

- Aus $(1+2)$ folgt

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

- Somit haben wir folgende Ungleichung bewiesen:

$$\text{Für jeden Knoten } u \in V \text{ gilt } d_i(s, u) \leq d_{i+1}(s, u) \tag{3}$$

- Nun zeigen wir $\delta_i < \delta_{i+1}$ für $i = 1, \ldots, k-1$
- Sei $P = (u_0, u_1, \ldots, u_{\delta_{i+1}})$ ein kürzester Pfad von $s = u_0$ nach $t = u_{\delta_{i+1}}$ in $N_{f_{i+1}}$ (und somit auch in $N'_{f_{i+1}}$)
- Mit Ungleichung 3 folgt, dass $d_i(s, u_h) \leq d_{i+1}(s, u_h) = h$ für $h = 0, \ldots, \delta_{i+1}$ ist
- Wir unterscheiden nun zwei Fälle:
  - Alle Knoten u_h sind in N'_{f_i} enthalten
  - Mindestens ein Knoten u_h ist nicht in N'_{f_i} enthalten

Beweis (Fortsetzung).

- Alle Knoten u_h sind in N'_{f_i} enthalten
  - In diesem Fall muss ein h mit $d_i(s, u_h) \leq d_i(s, u_{h-1})$ existieren
  - Würde nämlich $d_i(s, u_h) > d_i(s, u_{h-1})$ für $h = 1, \dots, \delta_{i+1} - 1$ gelten, so wären die Kanten (u_{h-1}, u_h) für $h = 1, \dots, \delta_{i+1} - 1$ wegen (2) in N_{f_i} enthalten und somit würde wegen (1) $d_i(s, u_h) = d_i(s, u_{h-1}) + 1$ für $h = 1, \dots, \delta_{i+1} - 1$ folgen
  - Dies hätte wiederum zur Folge, dass P ein kürzester Pfad von s nach t in N_{f_i} und somit ein s-t-Pfad in N'_{f_i} wäre, der von g_i nicht blockiert wird, da er auch in $N_{f_{i+1}}$ existiert
  - Da aber g_i jeden s-t-Pfad in N'_{f_i} blockiert, muss also ein h mit $d_i(s, u_h) \leq d_i(s, u_{h-1})$ existieren und es folgt mit $(1+2)$:

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

Der Algorithmus von Dinitz

Beweis (Schluss)

- Mindestens ein Knoten u_h ist nicht in N'_{f_i} enthalten
  - Sei u_h der erste solche Knoten auf P und sei $e = (u_{h-1}, u_h)$
  - Da $u_h \neq t = u_{\delta_{i+1}}$ ist, folgt $d_{i+1}(s, u_h) < d_{i+1}(s, t) = \delta_{i+1}$
  - Zudem liegt die Kante e nicht nur in $N_{f_{i+1}}$, sondern wegen $f_{i+1}(e) = f_i(e)$ (da weder e noch e^R zu N'_{f_i} gehören) auch in N_{f_i}
  - Da somit u_{h-1} in N'_{f_i} und e in N_{f_i} ist, kann u_h nur aus dem Grund nicht zu N'_{f_i} gehören, dass $d_i(s, u_h) = d_i(s, t)$ ist
  - Daher folgt unter Verwendung von $(1 + 2 + 3)$ auch in diesem Fall die Ungleichung $\delta_i < \delta_{i+1}$:

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

□

Korollar

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

Der Algorithmus von Dinitz

- Wir betrachten nun die Prozedur `blockfluss2`
- Zu ihrer Beschreibung benötigen wir folgende Notation

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

- Der Durchsatz eines Knotens $u \in V$ in N ist

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

  wobei $c^+(u) = \sum_{v \in V} c(u, v)$ die Ausgangskapazität und $c^-(u) = \sum_{v \in V} c(v, u)$ die Eingangskapazität von u in N ist
- Ein Fluss g in N sättigt einen Knoten $u \in V$ in N, falls
  - $u = s$ ist und g alle Kanten $(s, v) \in E$ mit Startknoten s sättigt
  - oder $u = t$ ist und g alle Kanten $(v, t) \in E$ mit Zielknoten t sättigt
  - oder $u \in V - \{s, t\}$ ist und g alle Kanten $(u, v) \in E$ mit Startknoten u oder alle Kanten $(v, u) \in E$ mit Zielknoten u sättigt

Der Algorithmus von Dinitz

Die Korrektheit von `blockfluss2` basiert auf folgender Implikation

Proposition

Ein Fluss g in einem Netzwerk N ist g blockierend, wenn er auf jedem s-t-Pfad mindestens einen Knoten u sättigt

Beweis.

Falls g mindestens einen Knoten u auf einem s-t-Pfad P sättigt, dann sättigt g auch mindestens eine Kante auf dem Pfad P □

Der Algorithmus von Dinitz

- Beginnend mit dem trivialen Fluss $g = 0$ berechnet die Prozedur `blockfluss2` für jeden Knoten u den Durchsatz $D(u)$ im Schichtnetzwerk $S = (V, E, s, t, c)$ und wählt in jedem Durchlauf der repeat-Schleife einen Knoten u mit minimalem Durchsatz
- Dann benutzt sie die Prozeduren `propagierevor` und `propagiererück`, um den aktuellen Fluss g um den Wert $D(u)$ zu erhöhen und die Restkapazitäten der betroffenen Kanten sowie die Durchsatzwerte $D(v)$ der betroffenen Knoten zu aktualisieren
- Anschließend werden alle gesättigten Knoten aus V und alle gesättigten Kanten aus E entfernt
- Hierzu werden in der Menge B alle Knoten gespeichert, sobald deren Durchsatz durch die vorgenommenen Erhöhungen von g auf 0 sinkt

Prozedur $\texttt{blockfluss2}(S)$, $S = (V, E, s, t, c)$

for all $e \in E \cup E^R$ **do** $g(e) := 0$
for all $u \in V$ **do**
  $c^+(u) := \sum_{(u,v)\in E} c(u,v)$; $c^-(u) := \sum_{(v,u)\in E} c(v,u)$
repeat
  wähle $u \in V$ mit $D(u)$ minimal
  $B := \{u\}$; $\texttt{propagierevor}(u)$; $\texttt{propagiererück}(u)$
  while $\exists v \in B \setminus \{s,t\}$ **do**
    $B := B \setminus \{v\}$; $V := V \setminus \{v\}$
    for all $e = (v,w) \in E$ **do**
      $E := E \setminus \{e\}$; $c^-(w) := c^-(w) - c(v,w)$
      if $c^-(w) = 0$ **then** $B := B \cup \{w\}$
    for all $e = (w,v) \in E$ **do**
      $E := E \setminus \{e\}$; $c^+(w) := c^+(w) - c(w,v)$
      if $c^+(w) = 0$ **then** $B := B \cup \{w\}$
until $u \in \{s,t\}$
return g

Der Algorithmus von Dinitz

- Die Prozedur `blockfluss2` sättigt in jedem Durchlauf der repeat-Schleife mindestens einen Knoten u und entfernt ihn aus V
- Daher 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 der letzten Iteration der Durchsatz von s oder von t gleich Null ist, wird einer dieser beiden Knoten zu diesem Zeitpunkt von g gesättigt, d.h. g ist nach obiger Proposition ein blockierender Fluss
- Die Prozeduren `propagierevor` und `propagiererück` propagieren den Fluss durch u in Vorwärtsrichtung hin zu t bzw. in Rückwärtsrichtung hin zu s
- Dies geschieht in Form einer Breitensuche mit Startknoten u unter Benutzung der Kanten in E bzw. E^R
- Da der Durchsatz $D(u)$ von u unter allen Knoten minimal ist, ist sichergestellt, dass der Durchsatz $D(v)$ jedes Knotens v ausreicht, um den für ihn ermittelten Zusatzfluss in Höhe von $z(v)$ weiterzuleiten

Der Algorithmus von Dinitz

Prozedur `propagierevor`(u)

1 **for all** $v \in V$ **do** $z(v) := 0$
2 $z(u) := D(u)$
3 $Q := (u)$; $R := \{u\}$
4 **while** $Q \neq ()$ **do**
5    $v :=$ `dequeue`(Q)
6   **while** $z(v) \neq 0 \wedge \exists e = (v, w) \in E$ **do**
7   **if** $w \notin R$ **then** `enqueue`(Q, w)
8       $R := R \cup \{w\}$
9       $m := \min\{z(v), c(e)\}$; $z(v) := z(v) - m$; $z(w) := z(w) + m$
10   `aktualisierekante`(e, m)

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

Der Algorithmus von Dinitz

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

$g(e) := g(e) + m$
$c(e) := c(e) - m$
if $c(e) = 0$ **then** $E := E \setminus \{e\}$
$c^+(v) := c^+(v) - m$
if $c^+(v) = 0$ **then** $B := B \cup \{v\}$
$c^-(w) := c^-(w) - m$
if $c^-(w) = 0$ **then** $B := B \cup \{w\}$

- Da die repeat-Schleife von `blockfluss2` maximal $(n-1)$-mal durchlaufen wird, werden die Prozeduren `propagierevor` und `propagiererück` höchstens $(n-1)$-mal aufgerufen
- Sei a die Gesamtzahl der Durchläufe der inneren while-Schleife von `propagierevor`, summiert über alle Aufrufe

Der Algorithmus von Dinitz

- Dann gilt $a \leq m + n^2$, da in jedem Durchlauf der inneren while-Schleife
  - eine Kante aus E entfernt wird (falls in Zeile 9 $m = c(v, u)$ ist), was pro Kante höchstens einmal vorkommt
  - oder der zu propagierende Fluss $z(v)$ durch einen Knoten v auf Null sinkt (falls in Zeile 9 $m = z(v)$ ist), was pro Aufruf und pro Knoten höchstens einmal vorkommt
- Der gesamte Zeitaufwand ist daher $O(n^2 + m)$ innerhalb der beiden while-Schleifen und $O(n^2)$ außerhalb
- Die gleichen Schranken gelten für `propagiererück` und 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

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

Der Algorithmus von Dinitz

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

Lemma

Sei $N = (V, E, s, t, c)$ ein Netzwerk, f ein Fluss in N und $h : V \times V \to \mathbb{Z}$

- (A) Die Funktion h ist genau dann ein Fluss (bzw. maximaler Fluss) in N_f, wenn $f + h$ ein Fluss (bzw. maximaler Fluss) in N ist
- (B) Für jede Kante $e \in E \cup E^R$ gilt $c_f(e) + c_f(e^R) = c(e) + c(e^R)$
- (C) Für alle Knoten $u \in V \setminus \{s, t\}$ gilt $c_f^+(u) = c^+(u)$ und $c_f^-(u) = c^-(u)$ und somit $D_f(u) = D(u)$, wobei $D_f(u)$ der Durchsatz von u in N_f ist

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

Beweis.

- Da f die Antisymmetrie und die Kontinuität erfüllt, übertragen sich diese Eigenschaften von h auf $f + h$ und umgekehrt
- Weiter erfüllt h wegen

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

  genau dann die Kapazitätsbedingung in N_f, wenn $f + h$ sie in N erfüllt
- h ist genau dann maximal in N_f, wenn $g = f + h$ maximal in N ist, da
  - jeder Fluss h' in N_f mit $|h'| > |h|$ einen Fluss $g' = f + h'$ der Größe $|g'| = |f + h'| > |f + h| = |g|$ in N
  - und jeder Fluss g' in N mit $|g'| > |g|$ einen Fluss $h' = g' - f$ der Größe $|h'| = |g' - f| > |g| - |f| = |f + h| - |f| = |h|$ in N_f

  liefern würde

Der Algorithmus von Dinitz

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

Beweis.

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

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

Beweis.

- Für alle Knoten $u \in V \setminus \{s, t\}$ gilt

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

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

□

Der Algorithmus von Dinitz

Lemma

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

- **D** Falls jeder Knoten $u \in V \setminus \{s, t\}$ einen Durchsatz $D(u) \leq C$ in N hat, gilt $\ell \leq 1 + (n-2)C/F$
- **E** Falls jede Kante $e \in E$ eine Kapazität $c(e) \leq C$ hat, gilt $\ell \leq \min\left\{mC/F, 2n\sqrt{C/F}\right\}$

Beweis. Sei f ein Fluss der Größe F in N

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

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

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

Der Algorithmus von Dinitz

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

Beweis.

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

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

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

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

□

Der Algorithmus von Dinitz

Satz

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

1. Falls jeder Knoten $u \in V \setminus \{s, t\}$ einen Durchsatz $D(u) \leq C$ hat, so gilt $k \leq 1 + 2\sqrt{nC}$
2. Falls jede Kante $e \in E$ eine Kapazität $c(e) \leq C$ hat, so gilt $k \leq \min\left\{8(mC)^{1/2}, 4(n^2C)^{1/3}\right\}$

Beweis.

- Sei $F = |f|$ die Größe eines maximalen Flusses f in N und seien $g_1, \ldots, g_k$ die blockierenden Flüsse, die der Dinitz-Algorithmus der Reihe nach in den Schichtnetzwerken $N'_{f_0}, \ldots, N'_{f_{k-1}}$ berechnet
- f_0 ist also der Nullfluss in N und $f_{i+1} = f_i + g_{i+1}$ für $i = 0, \ldots, k-1$

Der Algorithmus von Dinitz

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

Beweis.

- Da die Anzahl k der Schleifendurchläufe $\leq F$ ist, müssen wir nur den Fall $F > \sqrt{nC}$ betrachten
- Wir schätzen zuerst die Längen ℓ_i der Schichtnetzwerke N'_{f_i} ab
- Da $f - f_i$ nach A ein maximaler Fluss in N_{f_i} der Größe $R_i = F - |f_i|$ ist und nach C jeder Knoten $u \in V \setminus \{s, t\}$ in N_{f_i} den gleichen Durchsatz wie in N hat, folgt nach D dass $\ell_i \leq 1 + nC/R_i$ ist
- Wegen $\ell_i \geq i + 1$ folgt daher

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

- Nun wählen wir i so, dass $R_{i+1} \leq \sqrt{nC} < R_i$ ist und erhalten

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

□

Der Algorithmus von Dinitz

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

Beweis.

- Zum Nachweis von $k \leq \sqrt{8mC}$ können wir $F > \sqrt{2mC}$ annehmen
- Wir schätzen wieder die Längen ℓ_i der Schichtnetzwerke N'_{f_i} ab
- Da jede Kante $e \in E_{f_i}$ nach B eine Kapazität $c_{f_i}(e) \leq 2C$ hat und $f - f_i$ nach A ein maximaler Fluss in N_{f_i} der Größe $R_i = F - |f_i|$ ist, folgt mit E dass $\ell_i \leq 2mC/R_i$ ist
- Wegen $\ell_i \geq i + 1$ folgt daher

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

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

$$k \leq R_{i+1} + 2mC/R_i \leq \sqrt{2mC} + \sqrt{2mC} = \sqrt{8mC}$$

Der Algorithmus von Dinitz

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

Beweis (Schluss)

- Zum Nachweis von $k \leq 4(n^2C)^{1/3}$ können wir annehmen, dass $F > 2(n^2C)^{1/3} = (2n\sqrt{2C})^{2/3}$ ist
- Zudem folgt mit E dass N'_{f_i} eine Länge $\ell_i \leq 2n\sqrt{2C/R_i}$ hat
- Wegen $\ell_i \geq i+1$ folgt daher

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

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

$$\begin{aligned} k &\leq R_{i+1} + 2n\sqrt{2C/R_i} \\ &\leq (2n\sqrt{2C})^{2/3} + 2n\sqrt{2C}/(2n\sqrt{2C})^{1/3} \\ &= 2(2n\sqrt{2C})^{2/3} = 4(n^2C)^{1/3} \end{aligned}$$

$\square$

Der Algorithmus von Dinitz

Korollar

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

1. Falls jeder Knoten $u \in V \setminus \{s, t\}$ einen Durchsatz $D(u) \leq C$ hat, so gilt $T = O((nC + m)\sqrt{nC}$
2. Falls jede Kante $e \in E$ eine Kapazität $c(e) \leq C$ hat, so gilt $T = O(\min\{(mC)^{3/2}, C^{4/3}n^{2/3}m\})$

Beweis.

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

Prozedur $\texttt{blockfluss1}(S)$, $S = (V, E, s, t, c)$

for all $e \in E \cup E^R$ **do** $g(e) := 0$

$u := s$; $K := (s)$; $\texttt{done} := \texttt{false}$

repeat

$\quad$ **if** $\exists e = (u, v) \in E$ **then**

$\quad\quad \mathrm{push}(K, v)$; $u := v$

$\quad$ **elsif** $u = t$ **then**

$\quad\quad P := K$-Pfad von s nach t

$\quad\quad c(P) := \min\{c(e) \mid e \in P\}$

$\quad\quad$ **for all** $e \in P$ **do**

$\quad\quad\quad g(e) := g(e) + c(P)$; $g(e^R) := -g(e)$; $c(e) := c(e) - c(P)$

$\quad\quad\quad$ **if** $c(e) = 0$ **then** $E := E \setminus \{e\}$

$\quad\quad K := (s)$; $u := s$

$\quad$ **elsif** $u \neq s$ **then** \\\\ backtracking

$\quad\quad \mathrm{pop}(K)$; $u' := \mathrm{top}(K)$; $E := E \setminus \{(u', u)\}$; $u := u'$

$\quad$ **else** $\texttt{done} := \texttt{true}$

until $\texttt{done}$

return g

Der Algorithmus von Dinitz

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

Beweis.

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

Der Algorithmus von Dinitz

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

Beweis.

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

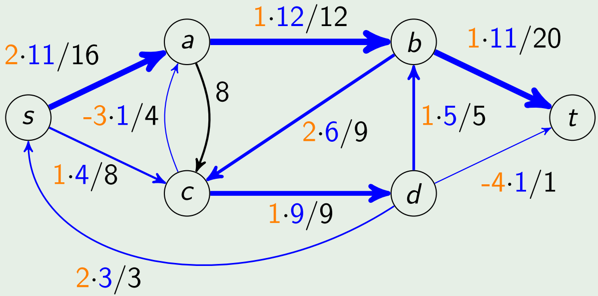
Kostenoptimale Flüsse

- In bestimmten Anwendungen fallen für die Benutzung jeder Kante Kosten an, deren Höhe proportional zum Fluss durch die Kante ist
- Falls zwei Flüsse f und g die einzelnen Kanten eines Netzwerks unterschiedlich beanspruchen, ist es möglich, dass f und g unterschiedliche Kosten verursachen, obwohl sie die gleiche Größe $|f| = |g|$ haben
- Gesucht ist dann ein maximaler Fluss f mit minimalen Kosten, wobei die Kosten von f in N wie folgt auf der Basis einer Kostenfunktion k bestimmt werden
- Jede Kante $e \in E$ mit $f(e) \geq 0$ verursacht Kosten in Höhe von $k(e)f(e)$
- Die Gesamtkosten von f in $N = (V, E, s, t, c, k)$ betragen daher

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

Beispiel

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



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

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

Kostenoptimale Flüsse

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

Kostenoptimale Flüsse

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

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

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

Kostenoptimale Flüsse

Lemma

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

Beweis.

- Falls es in N_f einen negativen Kreis K gibt, können wir den Fluss durch alle Kanten $e \in K$ um eins erhöhen, um einen Fluss g der Größe $|g| = |f|$ mit $k(g) = k(f) + k(K) < k(f)$ zu erhalten
- Sei nun umgekehrt g ein Fluss in N mit $|g| = |f|$ und $k(g) < k(f)$
- Wegen $g(e) - f(e) \leq c(e) - f(e)$ ist dann $h = g - f$ ein Fluss im Restnetzwerk $N_f = (V, E_f, s, t, c_f, k)$
- Da h die Größe $|h| = |g| - |f| = 0$ hat, können wir h als Summe von Flüssen $f_{K_1}, \ldots, f_{K_\ell}$ in N_f darstellen, wobei jeder Fluss f_{K_i} nur für Kanten e auf einem Kreis K_i in (V, E_f) einen positiven Wert $f_{K_i}(e) = w_i > 0$ annimmt (siehe nächste Folie)

Beweis (Fortsetzung).

- Wegen $k(f_{K_1}) + \cdots + k(f_{K_\ell}) = k(g - f) = k(g) - k(f) < 0$ und $k(f_{K_i}) = \sum_{e \in K_i} f_{K_i}(e)k(e) = w_i k(K_i)$ muss mindestens ein Kreis K_i negativ sein
- Um für $i = 0, \ldots, \ell - 1$ den Fluss $f_{K_{i+1}}$ und den zugehörigen Kreis K_{i+1} zu finden, wählen wir eine beliebige Kante $e_{i,1}$ aus E_f, für die der Fluss $r_i = h - f_{K_1} - \cdots - f_{K_i}$ einen minimalen positiven Wert $w_i = r_i(e_{i,1}) > 0$ annimmt
- Falls es keine Kante $e \in E_f$ mit $r_i(e) > 0$ gibt, sind wir fertig, weil dann r_i der Nullfluss mit $r_i(e) = 0$ für alle $e \in V \times V$ und somit $\ell = i$ ist
- Andernfalls benutzen wir die Tatsache, dass r_i wie h und die bereits gefundenen Flüsse $f_{K_1}, \ldots, f_{K_i}$ den Wert 0 hat, um einen negativen Kreis K_{i+1} zu finden
- Wegen $|r_i| = 0$ erfüllt r_i nämlich die Kontinuitätsbedingung auch für die Knoten s und t

Kostenoptimale Flüsse

Beweis (Schluss)

- Daher können wir K_{i+1} wie folgt konstruieren
  - Beginnend mit $j = 1$ wählen wir zu jeder Kante $e_{i,j} = (u, v)$ solange eine Fortsetzung $e_{i,j+1} = (v, w) \in E_f$ mit $r_i(e_{i,j+1}) > 0$, bis sich ein Kreis K_{i+1} schließt
- Nun setzen wir $f_{K_{i+1}}(e_{i,j}) = w_i$ für alle Kanten $e_{i,j}$ auf dem Kreis K_{i+1} und $f_{K_{i+1}}(e_{i,j}) = 0$ für alle Kanten außerhalb von K_{i+1}
- Da die Anzahl der Kanten in E_f, die unter dem Fluss r_{i+1} den Wert 0 haben, gegenüber r_i mindestens um eins zunimmt, ist die Anzahl ℓ der gefundenen Kreise durch $\ell \leq |E_f| \leq 2m$ beschränkt □

Mithilfe von obigem Lemma lässt sich nun ein maximaler Fluss mit minimalen Kosten wie folgt berechnen

Kostenoptimale Flüsse

- Wir berechnen zuerst einen maximalen Fluss f und setzen $f_0 = f$
- Dann berechnen wir für $i = 0, 1, \ldots$ einen negativen Kreis K_{i+1} in N_{f_i}
- Hierzu fügen wir dem Digraphen (V, E_{f_i}, k) einen neuen Knoten s' hinzu und verbinden s' mit allen Knoten $u \in V$ durch eine neue Kante (s', u) mit $k(s', u) = 0$
- Dann suchen wir mit dem Bellman-Ford-Moore (BFM) Algorithmus in dem resultierenden Digraphen $G_i = (V \cup \{s'\}, E_{f_i} \cup \{(s', u) \mid u \in V\}, k)$ nach einem negativen Kreis K_{i+1}
- Falls es in G_i keinen negativen Kreis gibt, ist f_i ein maximaler Fluss in N mit minimalen Kosten
- Andernfalls benutzen wir den Fluss $f_{K_{i+1}}$, der auf jeder Kante e auf K_{i+1} den Wert $f_{K_{i+1}}(e) = c_{f_i}(K_{i+1}) = \min\{c_{f_i}(e) \mid e \in K_{i+1}\}$ und außerhalb von K_{i+1} den Wert 0 hat, um den Fluss $f_{i+1} = f_i + f_{K_{i+1}}$ zu erhalten

Kostenoptimale Flüsse

- Da sich die Kosten von f_i wegen

$$k(f_{i+1}) - k(f_i) = k(f_{K_{i+1}}) = c_{f_i}(K_{i+1})k(K_{i+1}) \leq -1$$

  bei jeder Iteration um mindestens eins verringern und die Kostendifferenz zwischen zwei beliebigen Flüssen in N_f durch

$$D = \sum_{e \in E} |k(e)|(c(e) + c(e^R))$$

  beschränkt ist, ist f_ℓ nach $\ell \leq D$ Iterationen ein kostenminimaler Fluss
- Da der BFM-Algorithmus in Zeit $O(mn)$ läuft, führt dies auf eine Laufzeit von $O(Dmn)$, um die Kosten von f_0 zu minimieren
- Berechnen wir f_0 mit Dinitz in Zeit $O(n^3)$, so erhalten wir folgenden Satz

Satz

In einem Netzwerk N kann ein maximaler Fluss f mit minimalen Kosten in Zeit $O(n^3 + Dmn)$ bestimmt werden

Kostenoptimale Flüsse

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

Lemma. Sei f_i ein Fluss in N mit $k(f_i) = k_{\min}(f_i)$

Dann ist $f_{i+1} = f_i + f_{P_{i+1}}$ für jeden Zunahmepfad P_{i+1} in N_{f_i} mit

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

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

Beweis.

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

Beweis (Fortsetzung).

- Sei $F = K + P_{i+1}$ die Multimenge aller Kanten, die auf K oder P_{i+1} liegen, d.h. jede Kante in $K \Delta P_{i+1} = (K \setminus P_{i+1}) \cup (P_{i+1} \setminus K)$ kommt genau einmal und jede Kante in $K \cap P_{i+1}$ kommt genau 2-mal in F vor
- F ist also ein Multigraph bestehend aus dem s-t-Pfad P_{i+1} in N_{f_i} und dem Kreis K in $N_{f_{i+1}}$ und es gilt $k(F) = k(P_{i+1}) + k(K) < k(P_{i+1})$
- Um in F einen s-t-Pfad P' von N_{f_i} mit $k(P') < k(P_{i+1})$ zu finden, entfernen wir wie folgt alle Kanten in $\hat{F} = F \setminus E_{f_i} = K \setminus E_{f_i}$ aus F
- Wegen $\hat{F} \subseteq K \setminus P_{i+1}$ kommt jede Kante $e \in \hat{F}$ genau einmal in F vor
- Zudem wird jede Kante $e \in \hat{F}$ wegen
  - $f_i(e) = c(e)$ (da $e \notin E_{f_i}$) zwar von f_i, aber wegen
  - $e \in K \subseteq E_{f_{i+1}}$ nicht von f_{i+1} gesättigt
- Daher muss $f_i(e) \neq f_{i+1}(e)$ und somit $e^R \in P_{i+1}$ sein (da $e \notin P_{i+1}$)
- Wegen $e^R \in P_{i+1} \setminus K$ (da $e \in K$) kommt also für jede Kante $e \in \hat{F}$ auch e^R genau einmal in F vor

Beweis (Schluss)

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

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

□

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

Kostenoptimale Flüsse

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

for all $(u, v) \in V \times V$ **do**
  $f(u, v) := 0$
while $P :=$ `min-zunahmepfad`$(f) \neq \perp$ **do**
  `addierepfad`(f, P)

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

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

Satz. Sei $N = (V, E, s, t, c, k)$ ein Netzwerk ohne negative Kreise

Dann lässt sich in N ein maximaler Fluss f mit minimalen Kosten in Zeit $O(mn + |f|m \log n)$ berechnen

Definition. Sei $G = (V, E, k)$ ein Digraph mit Kostenfunktion $k : E \to \mathbb{Z}$

- Eine Funktion $p : V \to \mathbb{Z}$ heißt Preisfunktion für G, falls für jede Kante $e = (u, v)$ in E gilt:

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

- Die p-reduzierte Kostenfunktion $k^p : E \to \mathbb{N}$ ist

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

Lemma

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

Beweis.

- Wir zeigen zuerst die Rückwärtsrichtung
- Sei also p eine Preisfunktion mit $k^p(e) \geq 0$ für alle $e \in E$
- Dann gilt für jede Kantenmenge $F \subseteq E$ die Ungleichung $k^p(F) \geq 0$
- Da zudem für jeden Kreis K in G die Gleichheit $k(K) = k^p(K)$ gilt, folgt $k(K) = k^p(K) \geq 0$
- Für die Vorwärtsrichtung sei nun $G = (V, E, k)$ ein Digraph ohne negative Kreise

Beweis (Fortsetzung).

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

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

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

Kostenoptimale Flüsse

- Um einen maximalen Fluss mit minimalen Kosten in einem Netzwerk $N = (V, E, s, t, c, k)$ in Zeit $O(mn + |f| m \log n)$ zu berechnen, rufen wir zuerst BFM mit Startknoten s' auf, um nach einem negativen Kreis K im erweiterten Digraphen $G' = (V', E', k')$ zu suchen
- Wird K nicht gefunden, berechnet BFM hierbei eine Preisfunktion p_0 für das Netzwerk $N_0 = N_{f_0} = N$, wobei f_0 der Nullfluss in N ist
- Nun können wir mit Dijkstra für $i = 0, 1, \ldots$ in Zeit $O(m \log n)$
  - einen bzgl. k^{p_i} kürzesten Zunahmepfad P_{i+1} in $N_{f_i}^{p_i}$ und
  - einen Fluss $f_{i+1} = f_i + f_{P_{i+1}}$ der Größe $|f_{i+1}| > |f_i|$ mit minimalen Kosten $k(f_{i+1}) = k_{min}(f_{i+1})$
  - sowie eine Preisfunktion p_{i+1} für $N_{f_{i+1}}$ berechnen
- Ein bzgl. k^{p_i} kürzester s-t-Pfad P ist nämlich auch bzgl. k ein kürzester s-t-Pfad, da $k^{p_i}(P) = k(P) + p_i(s) - p_i(t)$ und $p_i(s) - p_i(t)$ eine von P unabhängige Konstante ist

Kostenoptimale Flüsse

- Zudem lässt sich aus p_i nach folgendem Lemma eine Preisfunktion $p_{i+1} = p_i + \ell_i$ für k in $N_{f_{i+1}}$ finden, wobei $\ell_i(x)$ die minimale Pfadlänge von s nach x in $N_{f_i}^{p_i}$ ist
- Dabei kann ℓ_i und somit auch p_{i+1} von Dijkstra zusammen mit P_{i+1} in Zeit $O(m \log n)$ gleich mitberechnet werden

Lemma

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

Beweis.

- Wir zeigen zuerst, dass p_{i+1} eine Preisfunktion für k in N_{f_i} ist

Beweis (Fortsetzung).

- Da $\ell_i(v) \le \ell_i(u) + k^{p_i}(e)$ und $k^{p_i}(e) = k(e) + p_i(u) - p_i(v)$ für jede Kante $e = (u, v) \in E_{f_i}$ gilt, folgt

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

  d.h. p_{i+1} ist eine Preisfunktion für k in N_{f_i}

- Falls e auf P_{i+1} liegt, gilt sogar $k^{p_{i+1}}(e) = 0$, da P_{i+1} ein bzgl. k^{p_i} kürzester s-t-Pfad in N_{f_i} und daher $\ell_i(v) = \ell_i(u) + k^{p_i}(e)$ ist

- Da zudem für jede Kante e in $N_{f_{i+1}}$, die nicht zu N_{f_i} gehört, die Kante e^R auf dem Pfad P_{i+1} liegt, folgt wegen $k^{p_{i+1}}(e^R) = 0$,

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

  d.h. p_{i+1} ist auch eine Preisfunktion für k in $N_{f_{i+1}}$ □

Kürzeste Pfade in Distanzgraphen

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

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

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

Der Dijkstra-Algorithmus

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

Benötigte Operationen

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

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

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

Der Dijkstra-Algorithmus

- Voraussetzung für die Korrektheit des Algorithmus' ist, dass alle Kanten eine nichtnegative Länge $\ell(u, v) \geq 0$ haben
- In diesem Fall wird $D = (V, E, \ell)$ auch Distanzgraph genannt
- Während der Suche werden bestimmte Kanten $e = (u, v)$ daraufhin getestet, ob $d(u) + \ell(u, v) < d(v)$ ist
- Da in diesem Fall die Kante e auf eine Herabsetzung von $d(v)$ auf den Wert $d(u) + \ell(u, v)$ „drängt", wird diese Wertzuweisung als Relaxation von e bezeichnet
- Welche Kanten (u, v) auf Relaxation getestet werden, wird beim Dijkstra-Algorithmus durch eine einfache Greedystrategie bestimmt:
  - Wähle $u \in U$ mit minimalem d-Wert und teste alle Kanten (u, v), für die v nicht schon abgearbeitet ist
- Der Algorithmus führt also eine modifizierte Breitensuche aus, bei der die in Bearbeitung befindlichen Knoten in einer Prioritätswarteschlange U verwaltet werden

Der Dijkstra-Algorithmus

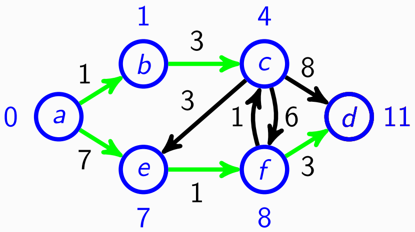
Algorithmus $\texttt{Dijkstra}(V, E, \ell, s)$

1 **for all** $v \in V$ **do**
2 $\quad d(v) := \infty$, $\texttt{parent}(v) := \texttt{nil}$, $\texttt{done}(v) := \texttt{false}$
3 $\texttt{Init}(U)$, $\texttt{Update}(U, s, 0)$, $d(s) := 0$
4 **while** $u := \texttt{GetMin}(U) \neq \texttt{nil}$ **do**
5 $\quad \texttt{done}(u) := \texttt{true}$
6 $\quad$ **for all** $v \in N^+(u)$ **do**
7 $\quad\quad$ **if** $\texttt{done}(v) = \texttt{false} \wedge d(u) + \ell(u, v) < d(v)$ **then**
8 $\quad\quad\quad d(v) := d(u) + \ell(u, v)$, $\texttt{Update}(U, v, d(v))$, $\texttt{parent}(v) := u$

- $d(u)$ speichert die aktuelle Länge des Pfades P_u; Knoten außerhalb des aktuellen Breitensuchbaums T haben den d-Wert ∞
- In jedem Schleifendurchlauf wird in Zeile 4 ein Knoten u mit minimalem d-Wert aus U entfernt und als abgearbeitet markiert
- Anschließend wird für alle von u wegführenden Kanten $e = (u, v)$ geprüft, ob eine Relaxation ansteht
- Wenn ja, wird U aktualisiert und e wird neue Baumkante in T

Der Dijkstra-Algorithmus

Beispiel. Betrachte folgenden Distanzgraphen mit dem Startknoten a:



Inhalt von U	GetMin(U)	besuchte Kanten	Updates
$(a,0)$	$(a,0)$	$(a,b),(a,e)$	$(b,1),(e,7)$
$(b,1),(e,7)$	$(b,1)$	(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)	$(f,8)$
$(f,8),(d,12)$	$(f,8)$	$(f,c),(f,d)$	$(d,11)$
$(d,11)$	$(d,11)$	–	–

Korrektheit des Dijkstra-Algorithmus'

Satz. Sei $D = (V, E, \ell)$ ein Distanzgraph und sei $s \in V$.

Dann berechnet $\texttt{Dijkstra}(V, E, \ell, s)$ für alle von s aus erreichbaren Knoten $t \in V$ einen kürzesten s-t-Pfad P_t

Beweis.

- Wir zeigen zuerst, dass alle von s aus erreichbaren Knoten $t \in V$ zu U hinzugefügt werden
- Dies folgt aus der Tatsache, dass s zu U hinzugefügt wird, und spätestens dann, wenn ein Knoten u in Zeile 4 aus U entfernt wird, sämtliche Nachfolger von u zu U hinzugefügt werden
- Zudem ist klar, dass $d(t) \geq d(s, t)$ ist, da P_t im Fall $d(t) < \infty$ ein s-t-Pfad der Länge $d(t)$ ist
- Es bleibt noch zu zeigen, dass P_u ein kürzester s-u-Pfad ist, sobald u aus U entfernt wird, d.h. es gilt $d(u) = d(s, u)$

Korrektheit des Dijkstra-Algorithmus'

Beweis.

- Wir zeigen induktiv über die Anzahl k der vor u aus U entfernten Knoten, dass $d(u) \leq d(s,u)$ ist
- Im Fall $k = 0$ ist $u = s$ und P_s hat die Länge $d(s) = 0 = d(s,s)$
- Im Fall $k \geq 1$ sei $P = v_0, \ldots, v_j = u$ ein kürzester s-u-Pfad in G und sei v_i der Knoten mit maximalem Index i, der vor u aus U entfernt wird
  - Nach IV gilt dann $d(v_i) = d(s, v_i)$ (1)
  - Zudem ist $d(v_{i+1}) \leq d(v_i) + \ell(v_i, v_{i+1})$ (2)
  - Weiter gilt $d(u) \leq d(v_{i+1})$ (3),
    da u im Fall $u \neq v_{i+1}$ vor v_{i+1} aus U entfernt wird
  - Nun folgt

$$\begin{aligned} d(u) &\overset{(3)}{\leq} d(v_{i+1}) \overset{(2)}{\leq} d(v_i) + \ell(v_i, v_{i+1}) \\ &\overset{(1)}{=} d(s, v_i) + \ell(v_i, v_{i+1}) = d(s, v_{i+1}) \leq d(s,u) \end{aligned}$$

□

Laufzeit des Dijkstra-Algorithmus'

- Wir überlegen uns zuerst, wie oft die einzelnen Operationen `Init`, `GetMin` und `Update` auf der Datenstruktur U ausgeführt werden
- Die `Init`-Operation wird nur einmal ausgeführt
- Da die while-Schleife für jeden von s aus erreichbaren Knoten genau einmal durchlaufen wird, wird die `GetMin`-Operation höchstens $\min(n, m)$-mal ausgeführt
- Da der Dijkstra-Algorithmus jede Kante höchstens einmal besucht, wird die `Update`-Operation höchstens m-mal ausgeführt
- Bezeichne $\mathit{Init}(n)$, $\mathit{GetMin}(n)$ und $\mathit{Update}(n)$ den Aufwand zum Ausführen der Operationen `Init`, `GetMin` und `Update` für den Fall, dass U nicht mehr als n Elemente aufzunehmen hat
- Dann ist die Laufzeit des Dijkstra-Algorithmus' durch

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

beschränkt

Laufzeit des Dijkstra-Algorithmus'

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

1. Da die Felder d und `done` bereits alle nötigen Informationen enthalten, kann man auf die (explizite) Implementierung von U auch verzichten
   - In diesem Fall kostet die `GetMin`-Operation allerdings Zeit $\mathcal{O}(n)$, was auf eine Gesamtlaufzeit von $\mathcal{O}(n^2)$ führt
   - Dies ist asymptotisch optimal, wenn G relativ dicht ist, also $m = \Omega(n^2)$ Kanten enthält
   - Ist G dagegen relativ dünn, d.h. $m = o(n^2)$, so empfiehlt es sich, U als Prioritätswarteschlange zu implementieren

Laufzeit des Dijkstra-Algorithmus'

2. Es ist naheliegend, U in Form eines Heaps H zu implementieren
   - In diesem Fall lässt sich die Operation `GetMin` in Zeit $\mathcal{O}(\log n)$ implementieren
   - Da die Prozedur `Update` einen linearen Zeitaufwand erfordert, ist es effizienter, sie durch eine `Insert`-Operation zu simulieren
   - Dies führt zwar dazu, dass derselbe Knoten evtl. mehrmals mit unterschiedlichen Werten in H gespeichert wird
   - Die Korrektheit bleibt aber dennoch erhalten, wenn wir nur die erste Entnahme eines Knotens aus H beachten und die übrigen ignorieren
   - Da für jede Kante höchstens ein Knoten in H eingefügt wird, erreicht H maximal die Größe n^2 und daher sind die Heap-Operationen `Insert` und `GetMin` immer noch in Zeit $\mathcal{O}(\log n^2) = \mathcal{O}(\log n)$ ausführbar
   - Insgesamt erhalten wir somit eine Laufzeit von $\mathcal{O}(n + m \log n)$
   - Dies ist zwar für dünne Graphen sehr gut, aber für dichte Graphen schlechter als die in 1 betrachtete implizite Implementierung von U

Laufzeit des Dijkstra-Algorithmus'

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

- Wir erhalten also folgende Laufzeiten:

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

- Offen ist, ob eine Verbesserung auf $\mathcal{O}(n + m)$ möglich ist

Kürzeste Pfade in Graphen mit negativen Kantengewichten

- In manchen Anwendungen treten negative Kantengewichte $k(e)$ auf
- Geben die Kantengewichte $k(e)$ beispielsweise die mit einer Kante e verbundenen Kosten wider, so kann ein Gewinn durch negative Kosten modelliert werden
- Auf diese Weise lassen sich auch längste Pfade in Distanzgraphen (V, E, ℓ) berechnen, indem man alle Kantenlängen $k(u, v) = -\ell(u, v)$ setzt und in dem resultierenden Kostengraphen (V, E, k) einen kürzesten Pfad bestimmt
- Die Komplexität des Problems hängt wesentlich davon ab, ob gerichtete Kreise mit negativer Länge (bzw. Kosten) vorkommen oder nicht
- Falls negative Kreise vorkommen können, ist das Problem NP-hart
- Andernfalls existieren effiziente Algorithmen wie z.B.
  - der Bellman-Ford-Algorithmus (BF-Algorithmus) oder
  - der Bellman-Ford-Moore-Algorithmus (BFM-Algorithmus)

Der Ford-Algorithmus

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

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

  und aktualisiert den Wert von $d(v)$ auf $d(u) + k(u, v)$ (Relaxation)
- Die Laufzeit hängt dann wesentlich davon ab, in welcher Reihenfolge die Kanten auf Relaxation getestet werden
- Im besten Fall lässt sich eine lineare Laufzeit erreichen (z.B. wenn überhaupt keine Kreise existieren)
- Bei der Bellman-Ford-Variante wird in $\mathcal{O}(mn)$ Schritten ein kürzester Pfad von s zu allen erreichbaren Knoten gefunden (sofern keine negativen Kreise existieren)

Der Ford-Algorithmus

- Wir zeigen induktiv über die Anzahl k der Kanten eines kürzesten s-t-Pfades, dass $d(t) \leq d(s,t)$ gilt, falls d für alle Kanten $(u,v) \in E$ die Dreiecksungleichung $d(v) \leq d(u) + k(u,v)$ erfüllt (also d keine weiteren Relaxationen erlaubt)
- Im Fall $k = 0$ ist $t = s$ und somit $d(s) = 0 = d(s,s)$
- Im Fall $k > 0$ sei t ein Knoten, dessen kürzester s-t-Pfad P aus k Kanten besteht
  - Nach IV gilt dann $d(u) \leq d(s,u)$ für den Vorgänger u von t auf P
  - Da d die Dreiecksungleichung für (u,t) erfüllt, folgt daher

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

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

Der Bellman-Ford-Algorithmus

- Die Bellman-Ford-Variante testet in den ersten $n - 1$ Runden jeweils alle Kanten auf Relaxation und speichert dabei für jede relaxierte Kante (u, v) den Knoten u in $\texttt{parent}(v)$
- Da jetzt alle Kanten auf allen kürzesten Pfaden in der richtigen Reihenfolge relaxiert wurden, gilt nun $d(u) \leq d(s, u)$ für alle Knoten $u \in V$
- Kann daher in der n-ten Runde immer noch eine Kante $e = (u, v)$ relaxiert werden, so lässt sich wie folgt ein negativer Kreis K finden:
  - Wir gehen von u aus mittels `parent` solange zurück, bis wir zum zweiten Mal am gleichen Knoten ankommen
- Die Laufzeit ist offensichtlich $\mathcal{O}(mn)$

Der Bellman-Ford-Algorithmus

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

```
for all v ∈ V do
    d(v) := ∞
    parent(v) := nil
d(s) := 0
for i := 1 to n − 1 do
    for all (u, v) ∈ E do
        if d(u) + k(u, v) < d(v) then
            d(v) := d(u) + k(u, v)
            parent(v) := u
for all (u, v) ∈ E do
    if d(u) + k(u, v) < d(v) then
        print (es gibt einen negativen Kreis)
```

Der Bellman-Ford-Moore-Algorithmus

- Die BFM-Variante versucht eine Kante (u, v) nur dann zu relaxieren, wenn $d(u)$ in der Runde davor erniedrigt wurde
- Um in Runde $i + 1$ alle Knoten parat zu haben, deren d-Wert in Runde i verringert wurde, wird bei jeder Relaxation einer Kante (u, v) der Knoten v in einer Schlange Q gespeichert (falls er nicht schon in Q ist)
- Dabei kann in Q zusammen mit v auch die Rundenzahl $i + 1$ der anstehenden Tests aller von v ausgehenden Kanten gespeichert werden
- Sobald aus Q ein Knoten mit Rundenzahl $n + 1$ entfernt wird, bricht der Algorithmus mit der Meldung ab, dass negative Kreise existieren
- Da gegenüber der BF-Variante nur vergebliche Relaxationsversuche vermieden werden, liefern BF und BFM dasselbe Ergebnis

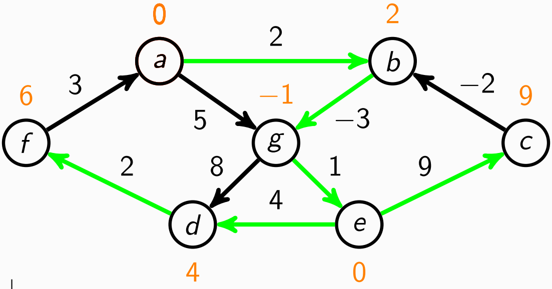
Der Bellman-Ford-Moore-Algorithmus

Algorithmus BFM(V, E, k, s)

```
for all v ∈ V do
    d(v) := ∞, parent(v) := nil, active(v) := false
d(s) := 0, Init(Q), Enqueue(Q, (1, s)), active(s) := true
while (i, u) := Dequeue(Q) ≠ nil and i ≤ n do
    active(u) := false
    for all v ∈ N⁺(u) do
        if d(u) + k(u, v) < d(v) then
            d(v) := d(u) + k(u, v)
            parent(v) := u
            if active(v) = false then
                Enqueue(Q, (i + 1, v))
                active(v) := true
if i > n then
    print (es gibt einen negativen Kreis)
```

Der Bellman-Ford-Moore-Algorithmus

Beispiel. Betrachte folgenden Kostengraphen mit dem Startknoten a:



⇑

Q: | 1: a |

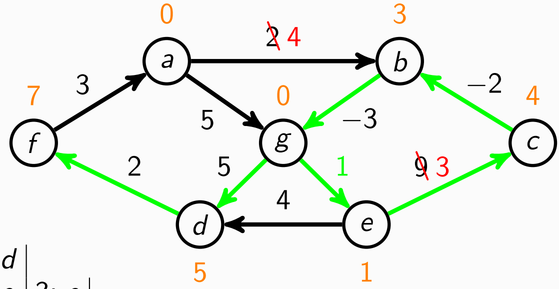
⇑

| 2: b, 2: g | 2: g | 3: d, 3: e | 3: e, 4: f | 4: f, 4: c, 4: d | 4: c, 4: d | 4: d | 5: f | |

Eintrag ≙ Runde: Knoten

Der Bellman-Ford-Moore-Algorithmus

Beispiel (Fortsetzung).



⇑

Q: | 1: a | 2: b, 2: g | 2: g | 3: d, 3: e | 3: e, 4: f | 4: f, 4: c | 4: c | 5: b | 6: g | 7: d, 7: e | 7: e, 8: f | 8: f, 8: c | 8: c |

⇑

Eintrag ≙ Runde: Knoten