# Algorithms and Probability

## Week 12

G09 - mkilic

21.V.2026

# Overview

# Roadmap

1. Graphentheorie
   - Zusammenhang
   - Kreise
   - Matchings
   - Färbungen
2. W'keitstheorie
   - Bedingte W'keit
   - Unabhängigkeit
   - (mehrere) Zufallsvariablen
   - Diskrete Verteilungen
   - Abschätzen von W'keiten
   - Randomisierte Algorithmen
3. Algorithmen
   - Lange-Bunte Pfade
   - MaxFlow
   - MinCut
   - Kleinster umschliessender Kreis
   - Konvexe Hülle

# Minitest

**Passwort:** `cut`

# Flüsse - Einführung & Definitionen

**Netzwerk**

Ein *Netzwerk* ist ein Tupel $N = (V, A, c, s, t)$, wobei gilt:

- $(V, A)$ ist ein **gerichteter** Graph ohne Schleifen
- $s \in V$, die *Quelle / source*
- $t \in V \setminus \{s\}$, die *Senke / target, sink*
- $c : A \to \mathbb{R}_0^+$, die *Kapazitätsfunktion / capacity function*

# Flüsse - Einführung & Definitionen

## Der Fluss $f$

Gegeben sei ein Netzwerk $N = (V, A, c, s, t)$. Ein Fluss in $N$ ist eine Funktion $f : A \to \mathbb{R}$ mit den Bedingungen

- **Zulässigleit**: $0 \leq f(e) \leq c(e)$ für alle $e \in A$
- **Flusserhaltung**: für alle $v \in V \setminus \{s, t\}$ gilt

$$\sum_{\substack{u \in V \\ (u,v) \in A}} f(u, v) = \sum_{\substack{u \in V \\ (v,u) \in A}} f(v, u)$$

[0] *engl. Zulässigkeit: capacity constraint, Flusserhaltung: flow conservation constraint*

# Flüsse - Einführung & Definitionen

## Der Wert eines Flusses

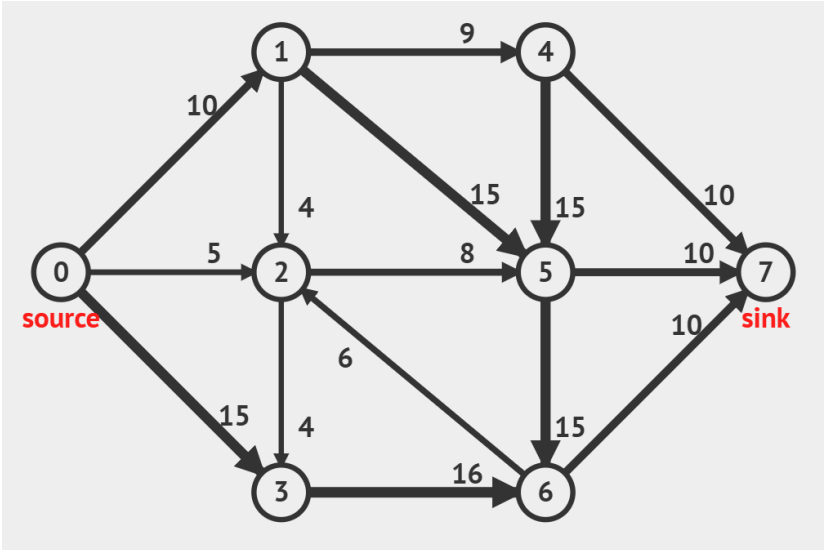
Der **Wert** eines Flusses $f$ ist durch

$$\text{val}(f) := \text{netoutflow}(s) := \sum_{\substack{u \in V \\ (s,u) \in A}} f(s,u) - \sum_{\substack{u \in V \\ (u,s) \in A}} f(u,s)$$

definiert. Wir nennen $f$ **ganzzahlig**, wenn $f(e) \in \mathbb{Z} \ \ \forall e \in A$.

[0] *engl. Wert: value, ganzzahlig: integer-valued*

# Flussnetzwerk - Beispiel



# Nettozufluss der Senke

**Lemma 3.6**

Der Nettozufluss der Senke gleicht dem Wert des Flusses, d.h.

$$\mathit{netinflow}(t) := \sum_{u \in V:(u,t) \in A} f(u,t) - \sum_{u \in V:(t,u) \in A} f(t,u) = \mathit{val}(f)$$

Also gilt $\mathit{netoutflow}(s) = \mathit{val}(f) = \mathit{netinflow}(t)$.

# Netinflow(t) = Netoutflow(s) - Beweis

Beweis:

$$0 = \sum_{(v,u)\in A} f(v,u) - \sum_{(u,v)\in A} f(u,v)$$

$$= \sum_{v\in V} \underbrace{\left( \sum_{u\in V:(v,u)\in A} f(v,u) - \sum_{u\in V:(u,v)\in A} f(u,v) \right)}_{=0 \text{ für } v\notin\{s,t\},\ \text{Flusserhaltung}}$$

$$= \underbrace{\left( \sum_{u\in V:(s,u)\in A} f(s,u) - \sum_{u\in V:(u,s)\in A} f(u,s) \right)}_{=\mathrm{val}(f)} + \underbrace{\left( \sum_{u\in V:(t,u)\in A} f(t,u) - \sum_{u\in V:(u,t)\in A} f(u,t) \right)}_{=-\mathrm{netinflow}(t)}.$$

□

# Schnitte

**Schnitte**

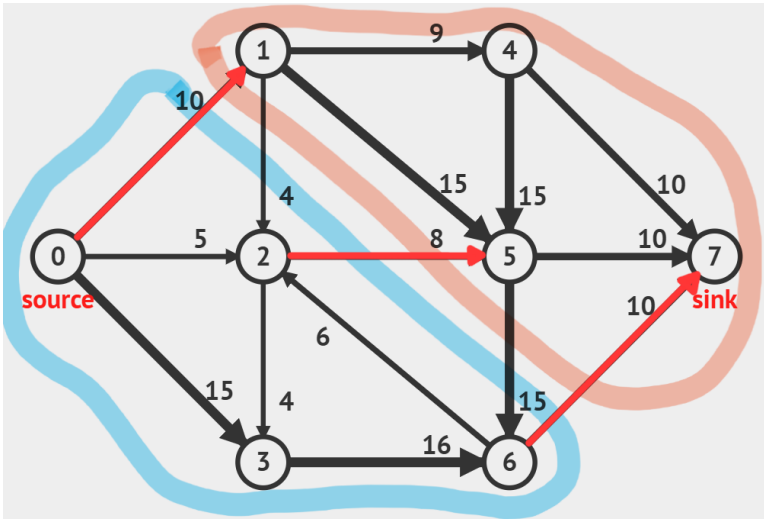
Ein *s-t-Schnitt* für ein Netzwerk $(V, A, c, s, t)$ ist eine Partition $(S, T)$ von $V$ (d. h. $S \cup T = V$ und $S \cap T = \emptyset$) mit $s \in S$ und $t \in T$. Die *Kapazität* eines $s$-$t$-Schnitts $(S, T)$ ist durch

$$\text{cap}(S, T) := \sum_{(u,w) \in (S \times T) \cap A} c(u, w)$$

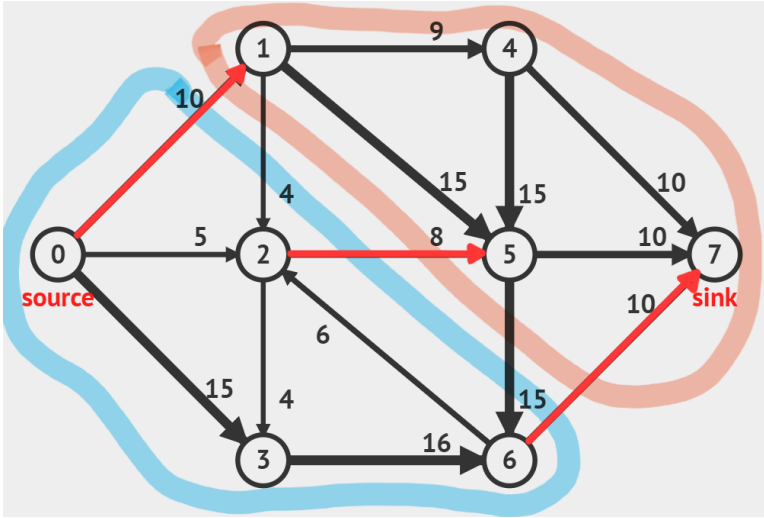
definiert.

- Die Kapazität eines Schnittes $(S, T)$ ignoriert die Kanten von $T$ nach $S$!
  Nur Einweg!

# Schnitte - Beispiel



# Schnitte - Beispiel




$cap(S, T) = 28$

# Schnitte und Flüsse

**$val(f)$ vs. $cap(S, T)$**

Ist $f$ ein Fluss und $(S, T)$ ein $s$-$t$-Schnitt in einem Netzwerk $(V, A, c, s, t)$, so gilt

$$val(f) \leq cap(S, T)$$

# $val(f) \leq cap(S, T)$ - **Beweis**

Sei $(U, W)$ eine Partition:

$$f(U, W) = \sum_{(u,w) \in (U \times W) \cap A} f(u, w)$$

Jetzt behaupten wir

$$val(f) \overset{(i)}{=} f(S, T) - f(T, S) \overset{(ii)}{\leq} f(S, T) \overset{(iii)}{\leq} cap(S, T)$$

(i) Wenn man nur die Flüsse für die Knoten in $S$ berücksichtigt, dann bleiben am Ende nur die Kanten die nur ihren Anfangspunkt in $S$ haben, oder nur ihren Endpunkt in $S$ haben. In anderen Worten bleiben nur die Kanten die über den $S$-$T$-Schnitt verlaufen.

(ii) $f$ ist nichtnegativ

(iii) $f(u, v) \leq c(u, v)$ für alle $(u, v) \in A$

[0] Formaler Beweis im Skript s. 176-177

# Maxflow-Mincut Theorem

**Maxflow-Mincut Theorem**

Jedes Netzwerk $N = (V, A, c, s, t)$ erfüllt:

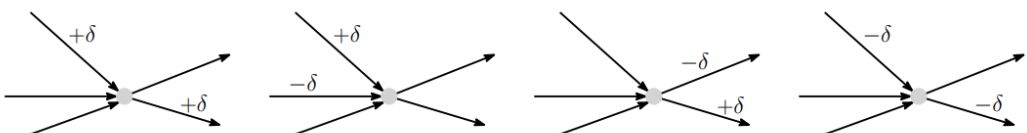
$$\max_{f \text{ Fluss in } N} val(f) = \min_{(S,T) \text{ s-t-Schnitt in } N} cap(S, T)$$

Besser weil:

- Es gibt nur endlich viele $s$-$t$-Schnitte.
- Ein minimaler Schnitt existiert immer

# Das Restnetzwerk und Augmentierende Pfade

Grundidee: beginne mit einem beliebigen Fluss, dann verbessere den Fluss.
Wir haben 4 Tricks um lokale Veränderungen zu machen, die die Flusserhaltung erhalten.



# Das Restnetzwerk und Augmentierende Pfade

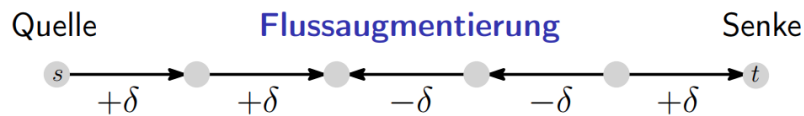
Wir betrachten Pfade von Quelle zu Senke. Lass $P$ ein solcher Pfad sein:

Nehmen wir an, dass wir auf jeder vorwärts gerichteten Kante den Fluss um $\delta$ erhöhen können, ohne die Zulässigkeit zu verletzen, und auf jeder rückwärts gerichteten Kante den Fluss um $\delta$ verkleinern können, ohne dass dieser negativ wird.

# Das Restnetzwerk und Augmentierende Pfade

Wir betrachten Pfade von Quelle zu Senke. Lass $P$ ein solcher Pfad sein:

Nehmen wir an, dass wir auf jeder vorwärts gerichteten Kante den Fluss um $\delta$ erhöhen können, ohne die Zulässigkeit zu verletzen, und auf jeder rückwärts gerichteten Kante den Fluss um $\delta$ verkleinern können, ohne dass dieser negativ wird.




Wir können den Fluss verbessern ohne die Flusserhaltung zu verletzen.

# Das Restnetzwerk und Augmentierende Pfade

Wenn wir wieder und wieder augmentieren dann erwarten wir dass wir in maximalen Flüssen stecken bleiben. Das muss man beweisen!

Man erreicht einen maximalen Fluss nicht immer in endlich vielen Schritten. Das gilt nur wenn die Kapazitäten **rational** sind!

# Das Restnetzwerk und Augmentierende Pfade

Für Vereinfachung nehmen wir an dass wir in $N$ keine entgegen gerichtete Kanten haben.

## Restnetzwerk

Sei $N = (V, A, c, s, t)$ ein Netzwerk ohne entgegen gerichtete Kanten und sei $f$ ein Fluss in $N$. Das Restnetzwerk $N_f := (V, A_f, r_f, s, t)$ ist wie folgt definiert:

1. Ist $e \in A$ mit $f(e) < c(e)$, dann ist $e$ auch eine Kante in $A_f$, mit

$$r_f(e) := c(e) - f(e).$$

2. Ist $e \in A$ mit $f(e) > 0$, dann ist $e^{\text{opp}}$ in $A_f$, mit

$$r_f(e^{\text{opp}}) = f(e).$$

3. Nur Kanten wie in (1) und (2) beschrieben finden sich in $A_f$.

$r_f(e)$, $e \in A_f$, nennen wir die *Restkapazität* der Kante $e$. (Alle strikt positiv)

# Charakterisierung Maximaler Fluss

**Charakterisierung Maximaler Fluss Satz 3.11**

Sei $N$ ein Netzwerk ohne entgegen gerichtete Kanten.

Ein Fluss $f$ ist maximal

$\Longleftrightarrow$

Es gibt keinen gerichteten s-t-Pfad im Restnetzwerk $N_f$

Für jeden maximalen Fluss $f$ gibt es einen $s$-$t$-Schnitt $(S, T)$ mit

$$val(f) = cap(S, T)$$

# Beweis: Satz 3.11

Beweisstruktur:

f maximal

(per contraposition, falls ein s-tPfad in $N_f$ könnten wir $f$ verbssern)

$\Rightarrow$ es gibt im $N_f$ keinen gerichteten s-t-Pfad

(Schnitt mit $S =$ von $s$ erreichbare Knoten in $(V, A_f)$ und $T = V \setminus S$)

$\Rightarrow$ es gibt einen s-t-Schnitt$(S, T)$ mit $val(f) = cap(S, T)$

(wir wissen $val(f) \leq cap(S, T)$. Wenn $val(f) = cap(S, T)$ ist dies maximal)

$\Rightarrow$ f maximal

□

# Ford-Fulkerson

FORD-FULKERSON$(V, A, c, s, t)$

1: $f \leftarrow 0$ ▷ Fluss konstant 0
2: **while** $\exists$ $s$-$t$-Pfad $P$ in $(V, A_f)$ **do**
3: Erhöhe den Fluss entlang $P$ ▷ wie unten
4: **end while**
5: **return** $f$ ▷ maximaler Fluss $=0$

▷ Erhöhung von $f$ zu einem Fluss $f'$ ist wie folgt:

$$f'(e) := \begin{cases} f(e) + \varepsilon & \text{falls } e \text{ auf } P, \\ f(e) - \varepsilon & \text{falls } e^{\text{opp}} \text{ auf } P, \text{ und} \\ f(e) & \text{sonst.} \end{cases}$$

# Ford-Fulkerson Laufzeitanalyse

- Es ist möglich, dass der Algorithmus für irrationalen Kapazitäten nicht terminiert.
- Der Algorithmus terminiert immer für ganzzahligen Kapazitäten.
- Sei $U \in \mathbb{N}$ eine obere Schranke für die aufteretende Kapazitäten, sei $n$ die Anzahl der Knoten und $m$ die Anzahl der Kanten.
- Dann kann kein Fluss Wert grösser als $nU$ haben wegen $val(f) \leq cap(\{s\}, V \setminus \{s\}) \leq nU$
- Also terminiert Ford-Fulkerson in höchstens $nU$ Runden. Wir müssen in jeder Runde das Restnetzwerk aktualisieren und einen neuen gerichteten $s$-$t$-Pfad in $N_f$ suchen. Das geht in $\mathcal{O}(n)$
- Insgesamt haben wir eine Laufzeit von $\mathcal{O}(mnU)$ falls alle Kapazitäten ganzzahlig ist.

# Weitere Algorithmen

- **Capacity-Scaling** [Dinitz-Gabow '73]
  Sind in einem Netzwerk alle Kapazitäten ganzzahlig und höchstens $U$, so kann ein ganzzahliger maximaler Fluss in Zeit

  $$O(mn(1 + \log U))$$

  berechnet werden.
- **Dynamic Trees** [Sleator-Tarjan '83]
  Der maximale Fluss eines Netzwerks kann in Zeit

  $$O(mn \log n)$$

  berechnet werden.

# Bipartites Matching als Flussproblem

**Lemma 3.15**

Die maximale Grösse eines Matchings im bipartiten Graph $G$ ist gleich dem Wert eines maximalen Flusses im Netzwerk $N$. *i.e.*

$$\max_{M \text{ Matching in } G} |M| = \max_{f \text{ Fluss in } N_G} val(f)$$

Wir können aus einem ganzzahligen Fluss das Matching M herausfinden und aus ein Matching M den Fluss in $N_G$ mit $val(f_M) = |M|$.

# Formale Darstellung

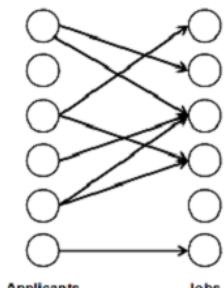
Wir haben: $G = (V, E)$ ein bipartiter Graph. D.h. Wir haben eine Partition von $(U, W)$ for $V$, sodass $E \subseteq \{\{u, w\} \mid u \in U, w \in W\}$.

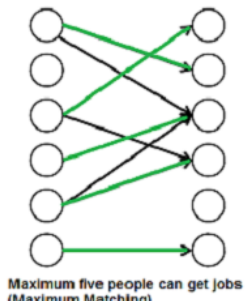
Wir definieren ein Netzwerk $N = (V \uplus \{s, t\}, A, c, s, t)$

- Wir haben zwei neue Knoten die die Rolle von der Quelle und der Senke SPIELEN.
- Die Kapazitätsfunktion $c$ ist konstant 1.
- $s$ hat Kanten zu allen Knoten in $U$.
- $t$ hat eingehende Kanten aus allen Knoten in $W$

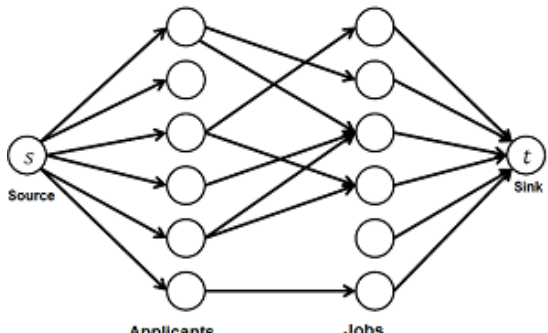
$$A := (\{s\} \times U) \cup \{(u, w) \in U \times W \mid \{u, w\} \in E\} \cup (W \times \{t\})$$

# Beispiel: Matching Problem

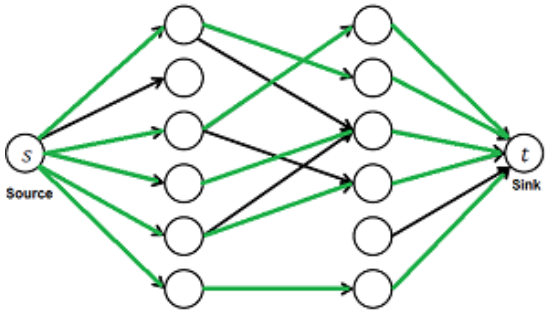


# Beispiel: Graph Konstruktion



# Beispiel: Max Flow




The maximum flow from source to sink is five units. Therefore, maximum five people can get jobs.

# Kanten- und Knotendisjunkte Pfade

# Schritt 1 – Ungerichteten Graph in Netzwerk umwandeln

- Ausgangspunkt: Ungerichteter Graph $G = (V, E)$
- Für jede Kante $\{x, y\} \in E$:
  - Ersetze sie durch zwei gerichtete Kanten: $(x, y)$ und $(y, x)$
  - Gib beiden Kanten Kapazität 1
- So entsteht ein gerichteter Graph mit Kapazitäten, der als Netzwerk genutzt werden kann.

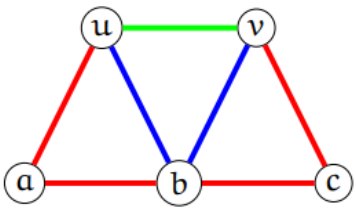
# Schritt 2 – Quellen- und Senkenknoten einführen

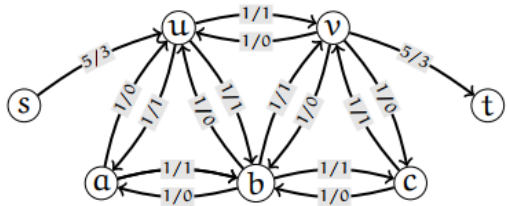
- Füge zwei neue Knoten $s$ und $t$ hinzu
- Füge Kanten hinzu:
  - $(s, u)$ mit Kapazität $n = |V|$
  - $(v, t)$ mit Kapazität $n = |V|$
- Ziel: Ermögliche beliebig viele Wege von $u$ nach $v$, beschränkt nur durch Kapazitäten in $G$

# Warum funktioniert das?

- Alle Kanten im Netzwerk haben Kapazität 1
- Maximaler Fluss von $s$ nach $t$ entspricht der maximalen Anzahl **kanten-disjunkter** $u$-$v$-Pfade
- Warum?
  - Jeder Pfad nutzt disjunkte Kanten (wegen Kapazität 1)
  - Rückfluss (z.B. $f(x, y) = f(y, x) = 1$) kann durch Zyklusreduktion entfernt werden

# Wie sieht es aus? - Beispiel Graph



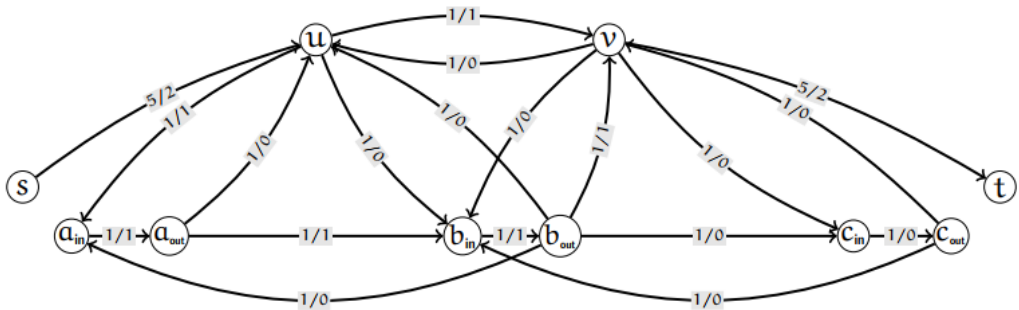
# Knoten-disjunkte Pfade

- Etwas trickreicher: Um **knoten-disjunkte** Pfade zu zählen, wandle den Graphen wie folgt um:
  - Ersetze jeden Knoten $v \in V \setminus \{u, v\}$ durch zwei Knoten $v_{\text{in}}$ und $v_{\text{out}}$
  - Verbinde $v_{\text{in}} \to v_{\text{out}}$ mit Kapazität 1
  - Ersetze jede Kante $\{x, y\} \in E$ durch zwei gerichtete Kanten:

$$x_{\text{out}} \to y_{\text{in}}, \quad y_{\text{out}} \to x_{\text{in}}$$

- So erzwingst du, dass kein Pfad denselben Knoten mehrfach benutzt.

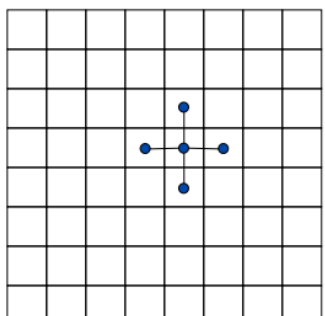
# Knotendisjunkte Pfade - Graphkonstruktion



# Bildesgmentierung mit Flüssen

Wir wollen für jedes Pixel entscheiden ob es im Hintergrund oder im Vordergrund steht. Wir haben ein Gitter und jedes Pixel hat ungerichtete Kanten zu den benachbarten Pixeln. (darüber, darunter, links und rechts). Wir haben also einen Graph $(P, E)$.

- $\alpha_p \geq 0$ ist die 'likelihood' dass $p \in A$
- $\beta_p \geq 0$ ist die 'likelihood' dass $p \in B$
- $p_{\{ij\}} \geq 0, \{i, j\} \in E$ ist Trennungsstrafe, also Strafe wenn einer der zwei Pixeln $i, j$ als Vordergrund und das Andere als Hintergrund klassifiziert wird.



Wir wollen den Qualitätsfuntion maximieren:

$$q(A, B) := \sum_{p \in A} \alpha_p + \sum_{p \in B} \beta_p - \sum_{e \in E, |e \cap A| = 1} \gamma_e$$

# Qualitätsfuntkion Umformulieren

$$
\begin{aligned}
\arg\max_{A,B} q(A,B) &= \arg\max_{A,B} \sum_{p\in A} \alpha_p + \sum_{p\in B} \beta_p - \sum_{e\in E, |e\cap A|=1} \gamma_e \\
&= \arg\max_{A,B} \sum_{p\in P} (\alpha_p + \beta_p) - \sum_{p\in A} \beta_p - \sum_{p\in B} \alpha_p - \sum_{e\in E, |e\cap A|=1} \gamma_e \\
= \arg\min_{A,B} q'(A,B) &= \arg\min_{A,B} \sum_{p\in A} \beta_p + \sum_{p\in B} \alpha_p + \sum_{e\in E, |e\cap A|=1} \gamma_e
\end{aligned}
$$

Wir wollen den letzten Ausdruck minimieren. Dazu konstruieren wir einen Graphen.

# Netzwerkaufbau für Bildsegmentierung

Gegeben: Menge von Pixeln $P$

Wir erstellen ein gerichtetes Netzwerk $N = (V, A)$ mit:

- Knotenmenge $V = P \cup \{s, t\}$, wobei $s$ Quelle und $t$ Senke ist
- Für jedes Pixel $p \in P$:
  - Kante $(s, p)$ mit Kapazität $\alpha_p$
  - Kante $(p, t)$ mit Kapazität $\beta_p$
- Für jede ungerichtete Kante $\{p, p'\} \in E$:
  - Zwei gerichtete Kanten $(p, p')$ und $(p', p)$ mit Kapazität $\gamma_e$

# Kapazität eines $s$-$t$-Schnitts

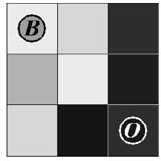
Ein Schnitt $(S, T)$ im Netzwerk $N$ trennt $s$ von $t$. Sei:

$$A := S \setminus \{s\}, \quad B := T \setminus \{t\}$$

Die Schnittkapazität $cap(S, T)$ ergibt sich durch:

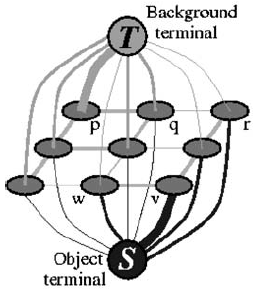
- **Quelle zu Pixeln:** $\sum_{p \in B} \alpha_p$
- **Pixel zu Senke:** $\sum_{p \in A} \beta_p$
- **Pixel-Pixel-Kanten:** $\sum_{(p,p') \in A \times B} \gamma_{(p,p')}$

# Bildsegmentierung mit Flüssen - Beispielgraph




(a) Image with seeds.

⇓



# Bedeutung für Segmentierung

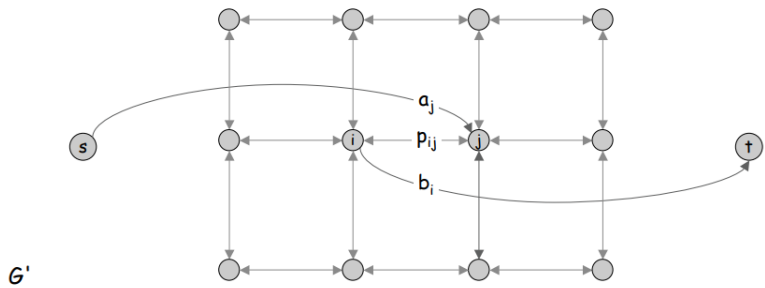
Die aus $(S, T)$ abgeleitete Partition $(A, B)$ von $P$ hat Kosten

$$q'(A, B) = cap(S, T)$$

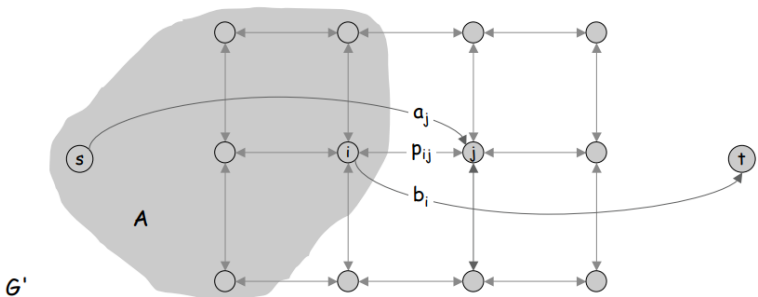
**Schlussfolgerung:** Eine optimale Segmentierung entspricht einem minimalen $s$-$t$-Schnitt im Netzwerk $N$.

- Kann mit einem Fluss- oder Schnittalgorithmus effizient bestimmt werden
- Praktisch: konkrete Gewichtungen $\alpha_p, \beta_p, \gamma_e$ hängen von Bildinhalten ab

# Eine weitere Visualization- Bildsegmentierung

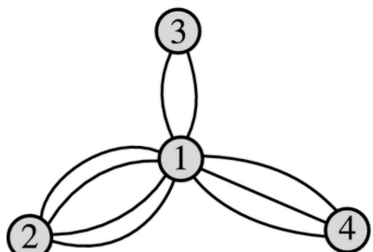


# Eine weitere Visualization- Bildsegmentierung



# Minimale Schnitte - Einführung

`MIN-CUT Problem`: Gegeben ein Multigraph $G$, bestimme $\mu(G)$: Die Kardinalität eines minimalen Schnitts.




Wir handlen ungerichteten ungewichteten Multigraphen $G = (V, E)$ ohne Schleifen.

# Minimale Schnitte - Einführung

## Kantenschnitt

In einem Multigraph $G = (V, E)$, nennen wir eine Menge $C \subseteq E$ für die $(V, E \setminus C)$ unzusammenhängend ist, ein Kantenschnitt.

Mit $\mu(G)$ bezeichnen wie die Kardinalität eines kleinstmöglichen Kantenschnitts in $G$

$$\mu(G) := \min_{C \subseteq E, (V, E \setminus C) \text{ unzusammenhängend}} |C|$$

# Erste Lösung

Wir können unsere Strategie aus maxFlow-minCut benutzen.

0. Wir haben einen Multigraph $G = (V, E)$

# Erste Lösung

Wir können unsere Strategie aus maxFlow-minCut benutzen.

0. Wir haben einen Multigraph $G = (V, E)$
1. Statt Mehrfachkanten können wir ganzzählige Kantengewichte verwenden.

# Erste Lösung

Wir können unsere Strategie aus maxFlow-minCut benutzen.

0. Wir haben einen Multigraph $G = (V, E)$
1. Statt Mehrfachkanten können wir ganzzählige Kantengewichte verwenden.
2. Ungerichtete Kanten $\{u, v\}$ können wir duch Paare gerichtete Kanten $(u, v), (v, u)$ ersetzen. Jetzt haben wir einen gerichteten Graph $(V, A)$.

# Erste Lösung

Wir können unsere Strategie aus maxFlow-minCut benutzen.

0. Wir haben einen Multigraph $G = (V, E)$
1. Statt Mehrfachkanten können wir ganzzählige Kantengewichte verwenden.
2. Ungerichtete Kanten $\{u, v\}$ können wir duch Paare gerichtete Kanten $(u, v), (v, u)$ ersetzen. Jetzt haben wir einen gerichteten Graph $(V, A)$.
3. Wir fixieren einen Knoten $s \in V$ und berechnen minimale $s$-$t$-Schnitte, für alle $t \in V \setminus \{s\}$ im Netzwerk $(V, A, w, s, t)$.

# Erste Lösung

Wir können unsere Strategie aus maxFlow-minCut benutzen.

0. Wir haben einen Multigraph $G = (V, E)$
1. Statt Mehrfachkanten können wir ganzzählige Kantengewichte verwenden.
2. Ungerichtete Kanten $\{u, v\}$ können wir duch Paare gerichtete Kanten $(u, v), (v, u)$ ersetzen. Jetzt haben wir einen gerichteten Graph $(V, A)$.
3. Wir fixieren einen Knoten $s \in V$ und berechnen minimale $s$-$t$-Schnitte, für alle $t \in V \setminus \{s\}$ im Netzwerk $(V, A, w, s, t)$.
4. Der Kleinste dieser $s$-$t$-Schnitte zeigt uns den minimalen Schnitt in unserem originalen Graph $G$.

# Erste Lösung

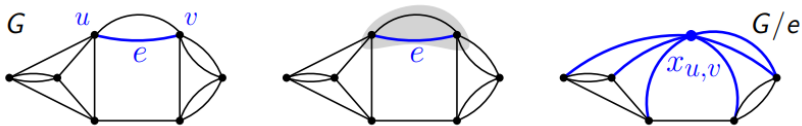
Wir können unsere Strategie aus maxFlow-minCut benutzen.

0. Wir haben einen Multigraph $G = (V, E)$
1. Statt Mehrfachkanten können wir ganzzählige Kantengewichte verwenden.
2. Ungerichtete Kanten $\{u, v\}$ können wir duch Paare gerichtete Kanten $(u, v), (v, u)$ ersetzen. Jetzt haben wir einen gerichteten Graph $(V, A)$.
3. Wir fixieren einen Knoten $s \in V$ und berechnen minimale $s$-$t$-Schnitte, für alle $t \in V \setminus \{s\}$ im Netzwerk $(V, A, w, s, t)$.
4. Der Kleinste dieser $s$-$t$-Schnitte zeigt uns den minimalen Schnitt in unserem originalen Graph $G$.

Laufzeit: $(n-1)$ mal $O(mn(1 + \log U))$ oder $O(mn^2 \log n) = O(n^4 \log n)$
Wir nehmen $m = O(n^2)$ weil im schlimmsten Fall hat jeder Knoten eine gerichtete Kante zu allen anderen Knoten.

# Kantenkontraktionen

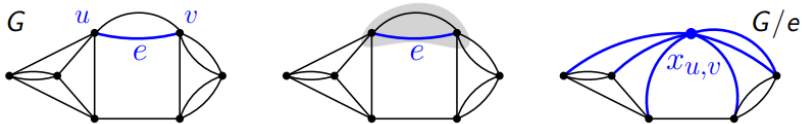
Nochmal sei $G = (V, E)$ ein Multigraph und $\{u, v\} \in E$. Die *Kontraktion* von $e$ macht aus die beiden Knoten $u$ und $v$ einen neuen Knoten $x_{u,v}$, der nun zu allen Kanten inzident ist, zu denen u oder v inzident war, ausser den Kanten zwischen u und v – diese Kanten verschwinden.




[0]Achtung! Es ist möglich, dass $deg(v) \neq |N(v)|$

# Kantenkontraktionen

Nochmal sei $G = (V, E)$ ein Multigraph und $\{u, v\} \in E$. Die *Kontraktion* von $e$ macht aus die beiden Knoten $u$ und $v$ einen neuen Knoten $x_{u,v}$, der nun zu allen Kanten inzident ist, zu denen u oder v inzident war, ausser den Kanten zwischen u und v – diese Kanten verschwinden.




Notation: $G/e$. Ist $k$ die Anzahl der Kanten zwischen $u$ und $v$, dann gilt

$$\deg_{G/e}(x_{u,v}) = deg_G(u) + deg_G(v) - 2k$$

und wir haben $|E(G/e)| = |E(G)| - k$.

[0]Achtung! Es ist möglich, dass $deg(v) \neq |N(v)|$

# Minimaler Schnitte in $G$ und $G/e$

**Lemma 3.20**

Sei $G$ ein Graph und $e$ eine Kante in $G$. Dann gilt $\mu(G/e) \geq \mu(G)$ und falls es in $G$ einen minimalen Schnitt $C$ mit $e \notin C$ gibt, dann gilt $\mu(G/e) = \mu(G)$.

- $\mu$ kann durch Kontraktion einer Kante $e$ **nie fallen**.
- $\mu$ bleibt gleich, falls es einen minimalen Schnitt ohne $e$ gibt.

Man spricht von einer natürlichen Bijektion. Entweder bleibt eine Kante in G bei der Kontraktion gleich, oder $\{w, u\}$ oder $\{w, v\}$ wird zu einer Kante $\{w, x_{u,v}\}$.

# Erster randomisierter Algorithmus

---

$\textsc{Cut}(G)$ *$G$ zusammenhängender Multigraph*

---

0: **while** $|V(G)| > 2$ **do**
0: $\quad e \leftarrow$ gleichverteilt zufällige Kante in $G$
0: $\quad G \leftarrow G/e$
**return** Grösse des eindeutigen Schnitts in $G$

---

Sei $|V| = n$, dann machen wir zwei Annahmen:

- Eine Katnenkontraktion kann in $O(n)$ Zeit durchgeführt werden.
- Eine gleichverteilte zufällige Kante in $G$ kann in $O(n)$ gewählt werden.[1]

Insgesamt können wir $\texttt{Cut}(G)$ mit Laufzeit $O(n^2)$ implementieren.

[1] Das erfordert Darstellung der Mehrfachkanten durch Kantengewichte.

# Erhalt von $\mu$

**Lemma 3.21.**

Sei $G = (V, E)$ ein Multigraph mit $n$ Knoten. Falls $e$ gleichverteilt zufällig unter den Kanten in $G$ gewählt wird, dann gilt

$$Pr[\mu(G) = \mu(G/e)] \geq 1 - \frac{2}{n}$$

- Ergebnis des Algorithmus ist nie kleiner als $\mu(G)$.
- Falls es einen minimalen Schnitt $C$ gibt, aus dem nie eine Kante kontrahiert wird, so gibt der Algorithmus $\mu(G)$ aus.

# Beweis: Lemma 3.21

- Sei $C$ ein minimaler Schnitt in $G$ und $k := |C| = \mu(G)$

□

# Beweis: Lemma 3.21

- Sei $C$ ein minimaler Schnitt in $G$ und $k := |C| = \mu(G)$
- Sicherlich ist der Grad jedes Knotens in $G$ mindestens $k$, da zu einem Knoten inzidenten Kanten immer einen Schnitt bilden: $\forall v \in V : \deg_G(v) \geq k$

□

# Beweis: Lemma 3.21

- Sei $C$ ein minimaler Schnitt in $G$ und $k := |C| = \mu(G)$
- Sicherlich ist der Grad jedes Knotens in $G$ mindestens $k$, da zu einem Knoten inzidenten Kanten immer einen Schnitt bilden: $\forall v \in V : \deg_G(v) \geq k$
- Es gilt daher: $|E| := \frac{1}{2} \sum_{v \in V} \deg(v) \geq \frac{kn}{2}$

□

# Beweis: Lemma 3.21

- Sei $C$ ein minimaler Schnitt in $G$ und $k := |C| = \mu(G)$
- Sicherlich ist der Grad jedes Knotens in $G$ mindestens $k$, da zu einem Knoten inzidenten Kanten immer einen Schnitt bilden: $\forall v \in V : \deg_G(v) \geq k$
- Es gilt daher: $|E| := \frac{1}{2} \sum_{v \in V} \deg(v) \geq \frac{kn}{2}$
- Wir wissen $e \notin C \Rightarrow \mu(G/e) = \mu(G)$ und somit

$$Pr[\mu(G) = \mu(G/e)] \geq Pr[e \notin C] = 1 - \frac{|C|}{|E|} \geq 1 - \frac{k}{kn/2} = 1 - \frac{2}{n}$$

□

# Erfolgswarscheinlichkeit von `Cut`

Wir interessieren uns für die Erfolgswahrscheinlichkeit

$$\hat{p}(G) := \text{ Waharscheinlichkeit, dass } \texttt{Cut}(G) \text{ den Wert } \mu(G) \text{ ausgibt}$$

Wir definieren diese W'keit für unsere Analyse über Anzahl Knoten in G

$$\hat{p}(n) := \inf_{G=(V,E),|V|=n} \hat{p}(G)$$

[1] z.Bsp. $\hat{p}(2) = 1$

# Erfolgswarscheinlichkeit von `Cut`

Wir interessieren uns für die Erfolgswahrscheinlichkeit

$$\hat{p}(G) := \text{ Waharscheinlichkeit, dass } \texttt{Cut}(G) \text{ den Wert } \mu(G) \text{ ausgibt}$$

Wir definieren diese W'keit für unsere Analyse über Anzahl Knoten in G

$$\hat{p}(n) := \inf_{G=(V,E),|V|=n} \hat{p}(G)$$

**Lemma 3.22.**

Es gilt für alle $n \geq 3$

$$\hat{p}(n) \geq (1 - \frac{2}{n}) \cdot \hat{p}(n-1)$$

[1] z.Bsp. $\hat{p}(2) = 1$

# Erfolgswharscheinlichkeit von`Cut`

Wenn wir jetzt dieses Lemma benutzen können wir $\hat{p}(n)$ für beliebiges $n$ berechnen:

$$\hat{p} \geq \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdots \frac{3}{5} \cdot \frac{2}{4} \cdot \frac{1}{3} \cdot \hat{p}(2) = \frac{2}{n(n-1)}$$

# Erfolgswharscheinlichkeit von`Cut`

Wenn wir jetzt dieses Lemma benutzen können wir $\hat{p}(n)$ für beliebiges $n$ berechnen:

$$\hat{p} \geq \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdots \frac{3}{5} \cdot \frac{2}{4} \cdot \frac{1}{3} \cdot \hat{p}(2) = \frac{2}{n(n-1)}$$

**Lemma 3.23.**

Es gilt $\hat{p}(n) \geq \frac{2}{n(n-1)} = 1/\binom{n}{2}$ für alle $n \geq 2$.

Wir wiederholen unseren Algorithmus um die Fehlerw'keit zu reduzieren.

# Analyse von `Cut`

**Satz 3.24**

Für den Algorithmus der $\lambda\binom{n}{2}$-maligen Wiederholung von `Cut`$(G)$ gilt:

1. Der Algorithmus hat eine Laufzeit von $O(\lambda n^4)$
2. Der kleinste angetroffene Wert ist mit einer Wahrscheinlichkeit von mindestens $1 - e^{-\lambda}$ gleich $\mu(G)$

Das ist für $\lambda = \ln n$ gleiche Laufzeit wie der deterministischer Algorithmus. Geht es besser?

# Bootstrapping

Wir mahcen die meisten Fehler in den letzten Schritten. Der Schritt von 3 auf 2 Knoten ist mit W'keit 66% ein Fehler. Deswegen brechen wir unseren Algorithmus für eine fixe Grösse des Graphen $|V| = t$ ab und benutzen einne anderen Algorithmus $z(t)$ für den Graphen $G'$ der jetzt $t$ Knoten hat. Konkret heisst das:

---

CUT1(G) — G zusammenhängender Multigraph

---

0: **while** $|V(G)| > t$ **do**
0: $e \leftarrow$ gleichverteilt zufällige Kante in $G$
0: $G \leftarrow G/e$
0: **return** Grösse des eindeutigen Schnitts in $G$ ▷ in Zeit $O(z(t))$

---

# Analyse von `Cut1`

`Cut1` läuft in Zeit $O(n(n-t)+z(t))$. Wenn wir annehmen dass $z(t)$ eine Erfolgsw;keit von $p^*(t)$ hat dann hat `Cut1` insgesamt eine Erfolgsw'keit von

$$p_{\texttt{Cut1}} \geq \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdots \frac{t+1}{t+3} \cdot \frac{t}{t+2} \cdot \frac{t-1}{t+1} \cdot p^*(t) = \frac{t(t-1)}{n(n-1)} \cdot p^*(t)$$

Wenn wir den zweiten Algorithmus $z(t)$ als unseren Algorithmus aus Satz 3.24 wählen und ihn nur 1-mal wiederholen, dann haben wir $z(t) = O(t^4)$ und $p^*(t) = 1 - e^{-1}$. Also haben wir

$$p_{\texttt{Cut1}} = \frac{t(t-1)}{n(n-1)} \cdot \frac{e-1}{e}$$

# Analyse von `Cut1`

Jetzt wiederholen wir unseren Algorithmus $\lambda \cdot \frac{1}{1-p_{\texttt{Cut1}}}$-mal um die Erfolgsw'keit auf $1 - e^{-\lambda}$ zu erhöhen. Insgesamt haben wir also eine Laufzeit von

$$\lambda \frac{n(n-1)}{t(t-1)} \frac{e}{e-1} \cdot O(n(n-t) + t^4) = O\left(\lambda\left(\frac{n^4}{t^2} + n^2 t^2\right)\right)$$

Wir können $t = \sqrt{n}$ wählen um eine Laufzeit von $O(\lambda n^3)$ zu erhalten.

Jetzt können wir aber diesen schnelleren Algorithmus als eine Subroutine benutzen. Dann können wir den gleichen Prozess immer wieder durchgehen. Das nennt man *Bootstrapping*.

Es konvergiert zu einem Verfahren mit einer Laufzeit von $O(n^2 poly(\log n))$

# Quellen - Links

- Image Segmentation with Code: https://julie-jiang.github.io/image-segmentation/
- Network Flow Applications: https://www.cs.princeton.edu/ wayne/kleinberg-tardos/pearson/07MaximumFlowApplications.pdf

The End

♠