Flüsse und Matchings

Effiziente Algorithmen – Sommer 2019

Martin Hoefer

1 Maximale Flüsse

Ein **Flussnetzwerk** ist ein gerichteter Graph $G = (V, E, c, s, t)$ mit
- V Knotenmenge, E Kantenmenge, $s, t \in V$, s Quelle, t Senke
- c Kantenkapazitäten $c : E \to \mathbb{R}_{\geq 0}$
- Notation: $n = |V|$, $m = |E|$

Ein **Fluss** $f : E \to \mathbb{R}$ erfüllt
1. $0 \leq f(e) \leq c(e) \quad \forall e \in E$ (Kapazitätsbeschränkung)
2. für alle $v \in V \setminus \{s, t\}$:

$$\sum_{(x,v)\in E} f((x, v)) = \sum_{(v,y)\in E} f((v, y)) \ . \quad \text{(Flusserhaltung)}$$

Andere Sichtweise:
Ein **Fluss** $f : V \times V \to \mathbb{R}$ erfüllt
1. $f(u, v) \leq c((u, v)) \quad \forall (u, v) \in E \quad$ und $\quad f(u, v) \leq 0$ wenn $(u, v) \notin E$
2. $f(u, v) = -f(v, u)$
3. $\forall u \in V \setminus \{s, t\}$:

$$\sum_{v\in V} f(u, v) = 0 \ .$$

Erweiterung auf Knotenmengen:
Seien $U, W \subseteq V$, dann ist

$$f(U, W) = \sum_{u\in U} \sum_{w\in W} f(u, w) \ .$$

Der **Wert** eines Flusses f ist

$$|f| = f(\{s\}, V) = f(V, \{t\}) \ .$$

Ein **s-t-Schnitt** in G ist eine Partition von V in Mengen S und T so dass

$$S \cap T = \emptyset \quad S \cup T = V \quad s \in S \quad t \in T \ .$$

Lemma 1. *Wenn (S, T) ein s-t-Schnitt in G ist, dann gilt $f(S, T) = |f|$.*

Beweis:
Betrachte zwei s-t-Schnitte $(X\cup\{x\}, Y)$ und $(X, \{x\}\cup Y)$, wobei $x \neq s, t$. Dann gilt $f(X\cup\{x\}, Y) = f(X, \{x\} \cup Y)$, denn

$$\begin{aligned} & f(X \cup \{x\}, Y) = f(X, \{x\} \cup Y) \\ \Leftrightarrow \quad & f(X, Y) + f(\{x\}, Y) = f(X, \{x\}) + f(X, Y) \\ \Leftrightarrow \quad & f(\{x\}, Y) = f(X, \{x\}) = -f(\{x\}, X) \\ \Leftrightarrow \quad & f(\{x\}, Y) + f(\{x\}, X) = 0 \\ \Leftrightarrow \quad & f(\{x\}, X \cup Y) = 0, \end{aligned}$$

und die letzte Aussage gilt für $x \neq s, t$ mit Flusserhaltung.
Daher haben also alle s-t-Schnitte den Wert $f(S, T) = f(\{s\}, V \setminus \{s\}) = |f|$. □

Beachte: Für jeden s-t-Schnitt (S, T) ist der Wert jedes Flusses f beschränkt durch

$$|f| \quad = \quad f(S,T) \quad = \quad \sum_{u\in S}\sum_{v\in T} f(u,v) \quad \leq \quad \sum_{u\in S}\sum_{v\in T} c((u,v)) \quad = \quad c(S,T),$$

der **Kapazität des Schnittes** (S, T).

1.1 Ford-Fulkerson und Max-Flow-Min-Cut

Aussage: Es gibt Fluss f mit $|f| = \min\{c(S, T) \mid (S, T)$ ist ein s-t-Schnitt $\}$

Für Fluss f in G hat das **Residualnetzwerk** G_f die **residualen Kapazitäten**

$$c_f(u, v) = c(u, v) - f(u, v) \geq 0 \qquad \forall (u, v) \in V \times V \ .$$

Beachte: $c(u, v) = 0$ wenn $(u, v) \notin E$.
G_f enthält alle Kanten (u, v) mit $c_f(u, v) > 0$. (hat "umgekehrte" Kanten wo f fliesst)

Beobachtung: f Fluss für G, f' Fluss für G_f $\quad \Rightarrow \quad$ $f + f'$ Fluss für G und $|f + f'| = |f| + |f'|$.

Mit G_f können wir Fluss erhöhen und einfach erkennen, ob die Änderung zulässig ist:

- Finde einen s-t-Pfad P im Residualnetzwerk G_f
- Identifiziere kleinste Residualkapazität μ auf P
- Sende zusätzlichen Fluss von μ entlang P

Solch ein s-t-Pfad $P = (e_1, \ldots, e_\ell)$ ist ein **augmentierender Pfad**. Sei $e_i = (v_i, v_{i+1})$, mit $v_1 = s$ und $v_{\ell+1} = t$. Die Flussänderung f_P auf P ist gegeben durch

$$|f_P| \quad = \quad \mu \quad = \quad \min\{c_f(e_i) \mid 1 \leq i \leq \ell\}$$

und $f_P(v_i, v_{i+1}) = \mu$, $f_P(v_{i+1}, v_i) = -\mu$ für $i = 1, \ldots, \ell$, sowie $f_P(u, v) = 0$ sonst.

Algorithm 1: Ford-Fulkerson Algorithmus

1 Anfangs $f(u,v) = 0$ für alle $u, v \in V$.
2 **while** *es gibt s-t-Pfad P in* G_f **do**
3 Augmentiere f: $f(u,v) \leftarrow f(u,v) + f_P(u,v)$ für alle $u, v \in V$.

Ford-Fulkerson Algorithmus für maximale Flüsse

Fragen:
A) Terminiert dieser Algorithmus?
B) Anzahl Iterationen?
C) Wie wählt man die s-t-Pfade?

Zu A: Allgemein: NEIN (reellwertige Kapazitäten)
Mit ganzzahligen Kapazitäten JA, jeder Schnitt ergibt obere Schranke auf $|f|$, wir erhöhen in Einheitswerten

Zu B: Höchstens $|f_{\max}|$ viele Schritte, wobei $f_{\max}$ ein maximaler Fluss. Es gibt Netzwerke, in denen der Algorithmus tatsächlich so viele Iterationen durchlaufen kann.

Theorem 1 (Max-Flow-Min-Cut)**.** *Die folgenden Aussagen sind gleichbedeutend:*
1. *f ist ein maximaler Fluss.*
2. *f erzeugt keinen augmentierenden Pfad in G_f.*
3. *Es gibt s-t-Schnitt (S,T) und Fluss f mit $c(S,T) = |f| = f(S,T)$.*

Beweis:
3. $\Rightarrow$ 1.: klar, denn $|f| \leq c(S,T)$ für alle s-t-Schnitte (S,T)
1. $\Rightarrow$ 2.: klar, mit Ford-Fulkerson Algorithmus
2. $\Rightarrow$ 3.: Kein augmentierender Pfad in G_f. Also s nicht zu t verbunden in G_f. Für jeden s-t-Pfad betrachte erste Kante $e = (u,v)$ mit $c_f(u,v) = 0 = c(u,v) - f(u,v)$. Sei

$$S = \{v \in V \mid \text{ es gibt } s\text{-}v\text{-Pfad in } G_f\}$$

Mit $T = V \setminus S$ erhalten wir (S,T), einen s-t-Schnitt mit $|f| = f(S,T) = c(S,T)$. □

Zu C: Der **Edmonds-Karp Algorithmus** ist eine Variante von Ford-Fulkerson. Er wählt immer einen **kürzesten** s-t-Pfad in G_f, d.h., mit der **kleinsten Anzahl Kanten**.

Lemma 2. *Sei $\delta_f(s,v)$ die kürzeste-Pfad-Distanz von s zu v in G_f. Im Edmonds-Karp Algorithmus ist $\delta_f(s,v)$ monoton steigend, für alle Knoten $v \in V$.*

Beweis: Betrachte einen BFS-Baum von s und den kürzesten augmentierenden Pfad P. P durchläuft seine Knoten in streng monoton steigendem Level des BFS-Baumes. Bei der Flusserhöhung werden eine oder mehrere Kanten von P aus G_f gelöscht (wenn $f(u,v) = c(u,v)$ eintritt). Diese Kanten werden evtl. in invertierter Richtung in G_f eingefügt. Solche Kanten können aber in G_f keine Distanz zu s verringern. □

Nach höchstens m Runden erhöht sich die Distanz $\delta_f(s,t)$ um mindestens 1. Sie kann aber nur bis auf $n-1$ anwachsen. Daher gibt es höchstens $n \cdot m$ Iterationen im Algorithmus. Jeder BFS-Aufruf zum Finden eines kürzesten s-t-Pfades braucht Zeit $O(m)$. Daher

Lemma 3. *Der Edmonds-Karp Algorithmus berechnet einen maximalen Fluss in Zeit $O(n \cdot m^2)$.*

1.2 PreflowPush Algorithmus

Algorithm 2: PreflowPush Algorithmus

1 Anfangs $h(v) = 0\ \forall v \neq s$, $h(s) = n$, $f(s,v) = c(s,v)$, $f(v,s) = -c(s,v)$ für alle $(s,v) \in E$ und $f(u,v) = 0$ sonst.
2 **while** *es gibt* $v \neq t$ *mit* $\Delta_f(v) > 0$ **do**
3 Wähle v mit $\Delta_f(v) > 0$
4 **if** *es gibt* w *mit* $h(w) < h(v)$ *und* $(v,w) \in E_f$ **then** Push(f,h,v,w)
5 **else** Relabel(f,h,v)

Algorithm 3: Routine Push(f,h,v,w)

1 Sei $\delta = \min\{\Delta_f(v), c(v,w) - f(v,w)\}$
2 Setze $f(v,w) \leftarrow f(v,w) + \delta$ und $f(w,v) \leftarrow f(w,v) - \delta$.

Algorithm 4: Routine Relabel(f,h,v)

1 Höhenanstieg: $h(v) \leftarrow h(v) + 1$.

Ein anderer Ansatz für Flüsse:
Ein **Preflow** $f : E \to \mathbb{R}$ erfüllt

1. $f(u,v) = -f(v,u)$
2. $f(u,v) \leq c(u,v)$ $\quad \to$ Kapazität bei $(u,v) \in E$, sonst 0
3. Für alle $u \neq s$ ist eingehender Fluss mindestens ausgehender Fluss:

$$\sum_{x \in V} f(x,u) \geq 0 \qquad \forall u \in V \setminus \{s\}\ .$$

Normale Knoten können "überschwemmt werden" aber selbst keinen Fluss "erzeugen".

$\Delta_f(v) = \sum_{x \in V} f(x,u)$ ist der **Überschuss** von Preflow f in $v \in V$.

Beachte: Preflow ist Fluss wenn $\Delta_f(v) = 0$ für alle $v \in V \setminus \{s,t\}$ und $|f| = \Delta_f(t) = -\Delta_f(s)$.

Das **Residualnetzwerk** $G_f = (V, E_f)$ für Preflow f ist genauso definiert wie oben.

Labellings

Der PreflowPush Algorithmus schiebt anfangs mehr Fluss ins Netzwerk als möglich. Der Fluss findet dann seinen Weg "bergab", entweder zur Senke t oder zurück zur Quelle.

Ein **Labelling** $h : V \to \{0,1,2,\ldots\}$ weist jedem Knoten $v \in V$ eine **Höhe** $h(v)$ zu. h ist **kompatibel** mit Preflow f wenn

1. $h(s) = n$, $h(t) = 0$.
2. für alle $(v,w) \in E_f$ gilt $h(v) \leq h(w) + 1$. (Abstiegsbedingung).

Mit der Abstiegsbedingung geht keine Kante in G_f zu steil nach unten. Bei jeder Kante (v, w) ist Höhenunterschied von v zu w immer höchstens um 1 bergab (aber beliebig steil bergauf!)

Lemma 4. *Wenn Preflow f kompatibel mit Labelling h ist, dann gibt es keinen s-t-Pfad in G_f.*

Beweis: Betrachte kreisfreien s-t-Pfad in G_f. Jede Kante reduziert die Höhe um höchstens 1, höchstens $n-1$ Kanten, aber gesamter Höhenunterschied n. □

Mit Max-Flow-Min-Cut Theorem folgt:

Korollar 1. *Fluss f kompatibel mit h $\Rightarrow$ f ist maximaler Fluss!*

Unterschiedliche Ansätze zum Berechnen von maximalen Flüssen:

Ford-Fulkerson: Erfülle Fluss, erreiche Optimalität (= Kompatibilität)

PreflowPush: Erfülle Kompatibilität von Preflow und Labelling, erreiche Fluss.

Lemma 5. *Im PreflowPush-Algorithmus*
1. *sind die Labels nicht-negative ganze Zahlen*
2. *ist f Preflow. Wenn Kapazitäten ganzzahlig, dann ist f ganzzahlig.*
3. *sind f und h kompatibel.*

Wenn der Algorithmus terminiert, dann liefert er einen Fluss f (mit $\Delta_f(v) = 0$ für alle $v \neq s, t$) und mit 3. ist dies ein maximaler Fluss.

Beweis: Wir zeigen nur 3., der Rest ist klar.
Push kann Kanten aus G_f entfernen, aber nur eine Kante (w, v) zu G_f hinzufügen. (w, v) erfüllt Abstiegsbedigung denn $h(w) < h(v)$. Relabel erhöht $h(v)$, die Steigung von allen (v, w) wird erhöht. Aber Relabel wird nur aufgerufen wenn für alle $(v, w) \in E_f$ gilt $h(w) < h(v)$.
Daher gilt: Alle Push und Relabel Operationen erhalten Kompatibilität. □

In wievielen Schritten terminiert der Algorithmus (wenn überhaupt)?

Anzahl Relabel Operationen

Lemma 6. *Sei f ein Preflow. Wenn $\Delta_f(v) > 0$, dann gibt es einen v-s-Pfad in G_f.*

Beweis: Sei $A = \{v \in V \mid \text{es gibt } v\text{-}s\text{-Pfad in } G_f\}$ und $B = V \setminus A$.
Zu zeigen: Wenn $\Delta_f(v) > 0$, dann $v \in A$.

Es ist $s \in A$. Daneben gilt für jedes $e = (x, y) \in E$ mit $x \in A$, $y \in B$, dass $f(x, y) = 0$, sonst wäre $(y, x) \in E_f$ und $y \in A$. Es gilt $\Delta_f(v) \geq 0$, denn $s \notin B$, und daher

$$
\begin{aligned}
0 \leq \sum_{v \in B} \Delta_f(v) &= \sum_{v \in B} \sum_{x \in V} f(x, v) \\
&= \underbrace{\sum_{v \in B} \sum_{x \in B} f(x, v)}_{= 0, \text{ denn } f(x,v) = -f(v,x)} + \underbrace{\sum_{\substack{v \in B, x \in A \\ (x,v) \in E}} f(x, v)}_{= 0 \text{ siehe oben}} + \underbrace{\sum_{\substack{v \in B, x \in A \\ (x,v) \notin E}} f(x, v)}_{\leq 0, \text{ da Nicht-Kanten}} \\
&\leq 0
\end{aligned}
$$

also $\Delta_f(v) = 0$ für jedes $v \in B$. □

Lemma 7. *Im PreflowPush-Algorithmus gilt $h(v) \leq 2n-1$ für jedes $v \in V$. Die Anzahl der Relabel Operationen ist kleiner als $2n^2$.*

Beweis: $h(s) = n$, $h(t) = 0$ immer. Betrachte $v \in V$, sowie f und h nach Relabel(f, h, v). Da $\Delta_f(v) > 0$ gibt es einen v-s-Pfad in G_f mit Länge höchstens $n-1$. Mit Abstiegsbedingung (beachte: f, h immer kompatibel) wissen wir $h(v) - h(s) \leq n - 1$. □

Anzahl Push Operationen

Ein Push ist **saturierend** wenn $\delta = c(u, v) - f(u, v)$.

Lemma 8. *Die Anzahl der saturierenden Push Operationen ist höchstens $2nm$.*

Beweis: Betrachte $(v, w) \in E_f$. Nach saturierendem Push ist $h(v) = h(w) + 1$ und $(v, w) \notin E_f$. Nun muss $h(w)$ um mind. 2 ansteigen, bevor ein Push in die Gegenrichtung möglich ist. Nur nach solch einem Push in die Gegenrichtung wird (v, w) wieder in E_f erscheinen und ein erneuter Push auf (v, w) wird möglich.
Mit vorherigem Lemma wird ein saturierender Push höchstens $n - 1$ Mal ausgeführt, und zwar jeweils für $(v, w) \in E$ und (w, v) mit $(v, w) \in E$. Damit gibt es höchstens $2nm$ saturierende Push Operationen. □

Lemma 9. *Die Anzahl der nicht-saturierenden Push Operationen ist höchstens $4n^2m$.*

Beweis: Wir betrachten eine **Potenzialfunktion**

$$\Phi(f, h) = \sum_{v:\Delta_f(v)>0} h(v)$$

und machen eine amortisierte Analyse. Anfangs ist $\Phi(f, h) = 0$. Es gilt immer $\Phi(f, h) \geq 0$. Wie ändert sich Φ im Algorithmus?

Nicht-saturierender Push: Verringert $\Phi(f, h)$ um (mind.) 1. Nach Push über (v, w) hat v nun $\Delta_f(v) = 0$ (und $h(v)$ verlässt Φ) und w hat $h(w) \leq h(v) - 1$ (und $h(w)$ tritt evtl. Φ nun bei).

Relabel: Erhöht $h(v)$ und damit $\Phi(f, h)$ um genau 1. Davon gibt es höchstens $2n^2$ Operationen, Gesamtzuwachs in Φ durch alle Relabel Operationen höchstens $2n^2$.

Saturierender Push: h bleibt gleich, aber f ändert sich. Nach Push über (v, w) gilt bei w dann $\Delta_f(w) > 0$, also ist der Zuwachs in Φ durch die Hinzunahme von w höchstens $h(w) \leq 2n - 1$. Davon gibt es höchstens $2nm$ Operationen, Gesamtzuwachs in Φ durch alle saturierenden Push Operationen ist höchstens $2nm \cdot (2n - 1)$.

Nicht-saturierende Push Operationen senken Φ um mind. 1 ab. Anzahl beschränkt mit Gesamtzuwachs in Φ durch die anderen Operationen: $2n^2m(2n - 1) + 2n^2 \leq 4n^2m$. □

Wenn wir in jedem Schritt einen Knoten mit Überschuss in maximaler Höhe wählen, dann reduziert sich die Anzahl nicht-saturierender Pushs auf höchstens $4n^3$. Mit passenden Datenstrukturen kann der Algorithmus in Laufzeit $O(n^3)$ implementiert werden. Es gilt folgendes Resultat.

Theorem 2. *Es gibt eine Implementation des PreflowPush-Algorithmus, die einen maximalen Fluss in Zeit $O(n^3)$ berechnet.*

2 Matchings

Gegeben: Ungerichteter, schlichter Graph (keine Schleifen, keine Multi-Kanten)
Gesucht: Ein **Matching** $M \subset E$ so dass für jedes $u \in V$ **höchstens eine** Kante $\{u, v\} \in M$.

Notation:

- Kante $e \in E$ ist **gematched** oder **im Matching** wenn $e \in M$; sonst **ungematched**
- Knoten $v \in V$ ist **gematched** wenn $\exists u \in V$ und $\{u, v\} \in M$; sonst **ungematched**

Matching M ist ...
perfekt: Für jedes $u \in V$ gibt es $v \in V$ mit $\{u, v\} \in M$. (In M sind alle Knoten gematched)
maximum: Für jedes Matching M' gilt $|M'| \leq |M|$. (M hat größte Anzahl Kanten)
maximal: Für jede Kante $e \in E \setminus M$ gilt $M \cup \{e\}$ ist kein Matching. (M ist nicht erweiterbar)

Für jeden Graphen G gilt: perfekte Matchings $\subseteq$ maximum Matchings $\subseteq$ maximale Matchings.

2.1 Maximum Matching in bipartiten Graphen

Berechnung von maximum Matchings

Wir nutzen Flüsse, Residualnetzwerke, augmentierende Pfade und den Ford-Fulkerson Algorithmus.

Algorithm 5: Maximum Matching in bipartiten Graphen

1 Sei $G = (A \cup B, E)$ der bipartite Graph. Erstelle Flussnetzwerk $G' = (V', E')$:
2 $V' \leftarrow A \cup B$, $E' \leftarrow E$, richte alle Kanten von A nach B
3 Füge Knoten s und t zu V' hinzu.
4 Füge gerichtete Kanten $\{(s, u) \mid u \in A\}$ und $\{(v, t) \mid v \in B\}$ zu E' hinzu
5 Alle Kanten erhalten Kapazität $c(e) = 1$.
6 Berechne maximalen s-t-Fluss f in G'
7 **return** $M = \{\{u, v\} \in A \times B \mid f(u, v) = 1\}$

Sei M^* ein maximum Matching in G und $f_{\max}$ ein maximaler Fluss in G'.

Theorem 3. *Es gilt $|M^*| = |f_{\max}|$. Die Kanten in M^* sind genau die Kanten mit Fluss 1 in $f_{\max}$.*

Beweis: Wir zeigen: Für jedes Matching M gibt es einen integralen Fluss f mit $|M| = |f|$ und umgekehrt. Daraus folgt das Theorem, denn integrale Kapazitäten $\Rightarrow$ es gibt integralen Max-Fluss.

"$\Rightarrow$": Sei M Matching in G. Für alle $\{u, v\} \in M$ setze $f(s, u) = f(u, v) = f(v, t) = 1$ (und -1 in die Gegenrichtung). f ist ein s-t-Fluss, $|f| = |M|$, nur Kanten aus M tragen Fluss von A nach B.

"$\Leftarrow$": Sei f ein integraler s-t-Fluss in G'. Hier $c(e) \in \{0, 1\}$, also nehmen wir an $f(u, v) \in \{0, 1\}$ für alle $(u, v) \in E'$. Sei $M' = \{(u, v) \in A \times B \mid f(u, v) = 1\}$. Dann gilt:

$|M'| = |f|$**:** Schnitt (S, T) in G' mit $S = \{s\} \cup A$. $f(S, T) = |f| = |M'|$, denn es gibt kein $(u, v) \in E'$ mit $u \in B$ und $v \in A$.

$\forall u \in A$ **gibt es höchstens eine** $(u, v') \in M'$**:**
Fluss ≤ 1 kommt bei u an, integraler Fluss, Flusserhaltung.

$\forall v \in B$ **gibt es höchstens eine** $(u', v) \in M'$**:**
Fluss ≤ 1 kommt aus v heraus, integraler Fluss, Flusserhaltung.

Also gilt: M' ist ein Matching mit Größe $|f|$. □

Lemma 10. *Der Ford-Fulkerson Algorithmus berechnet ein maximum Matching in einem bipartiten Graph in Zeit $O(nm)$.*

- Ford-Fulkerson höchstens $|f_{\max}| = |M^*| \leq n/2$ Iterationen
- Jede Iteration in $O(m)$ Zeit: Erstellung Residualnetzkwerk, BFS für augmentierenden Pfad
- Konstruktion von G' am Anfang und M^* am Ende in Zeit $O(n+m)$.

Die besten bekannten Algorithmen lösen das Problem in Zeit $O(m\sqrt{n})$.

Bipartite Graphen mit perfekten Matchings

Nicht jeder bipartite Graph hat ein perfektes Matching. Wie sieht ein Graph ohne perfektes Matching aus? Gibt es ein kurzes "Zertifikat", ob ein bipartiter Graph kein perfektes Matching erlaubt?

- Bipartiter Graph $G = (A \cup B, E)$ und $X \subseteq A$
- $\Gamma(X) = \{v \in B \mid \exists u \in X \text{ mit } \{u, v\} \in E\}$, alle Nachbarn von X in B.

M ist perfekt wenn $|X| \leq \Gamma(X)$ für alle Teilmengen $X \subseteq A$. Wenn $|A| = |B|$, dann gilt tatsächlich auch die **umgekehrte Richtung**. Damit ist ein $X \subseteq A$ mit $|X| > |\Gamma(X)|$ das gewünschte Zertifikat.

Theorem 4 (Hall, König, etc.)**.** *Sei $G = (A \cup B, E)$ ein bipartiter Graph mit $|A| = |B|$. Dann enthält G **entweder** ein perfektes Matching **oder** eine Menge $X \subseteq A$ mit $|X| > |\Gamma(X)|$. Ein perfektes Matching oder die Menge X können in Zeit $O(nm)$ berechnet werden.*

Beweis: Wir nutzen die gleiche Konstruktion des Flussnetzwerkes G' wie oben. Sei $k = |A| = |B|$.

Wir wissen: G hat perfektes Matching $\Leftrightarrow$ Max-Fluss in G' hat Wert $|f_{\max}| = k$.

Wir zeigen: $|f_{\max}| < k \Rightarrow$ Es gibt $X \subseteq A$ mit $|X| > |\Gamma(X)|$. Damit wäre das Theorem bewiesen.

Sei also $|f_{\max}| < k$. Mit Max-Flow-Min-Cut folgt: Es gibt s-t-Schnitt (S, T) mit $c(S, T) < k$, wobei $s \in S$ und $S \subseteq A \cup B \cup \{s\}$.

Behauptung: $X = S \cap A$ hat $|X| > |\Gamma(X)|$.

Beweis der Behauptung: Verändere (S, T) um sicherzustellen, dass $\Gamma(X) \subseteq S$ mit $X = S \cap A$.

- Betrachte $y \in \Gamma(X) \cap T$ und Schnitt (S', T') mit $S' = S \cup \{y\}$.
- Kante (y, t) kreuzt den Schnitt (S', T'), aber mindestens ein $(u, y) \in S \times T$ läuft nun innerhalb S' (denn $y \in \Gamma(X)$).
- Daher gilt:
$$c(S', T') \leq c(S, T),$$
denn Kanten (u, v) mit $u \in A \cap T$ und $v \in B \cap S$ tragen nicht zu $c(S, T)$ bei.

Verschiebe nun die Knoten von $\Gamma(X)$ alle iterativ nach S', dann gilt $c(S', T') \leq c(S, T)$.

Nun betrachte $c(S', T')$ mit $\Gamma(X) \subseteq S'$.

- Kanten in diesem Schnitt gehen entweder bei s aus oder bei t ein
- Also gilt
$$c(S', T') = |A \cap T'| + |B \cap S'|$$

- Beachte: $|A \cap T'| \geq k - |X|$ und $|B \cap S'| \geq \Gamma(X)$.
- Mit der Annahme $c(S', T') \leq c(S, T) < k$ erhalten wir

$$\begin{aligned} & k > c(S', T') = |A \cap T'| + |B \cap S'| \geq k - |X| + |\Gamma(X)| \\ \Rightarrow \quad & k + |X| > k + |\Gamma(X)| \\ \Rightarrow \quad & |X| > |\Gamma(X)| \end{aligned}$$

Die Behauptung ist gezeigt, das Theorem folgt. □

2.2 Maximum Matching in allgemeinen Graphen

Sei M ein Matching. Ein **alternierender Pfad** für M ist ein Pfad, der abwechselnd aus Kanten $\in M$ und $\notin M$ besteht. Ein **augmentierender Pfad** für M ist ein alternierender Pfad, der mit einer Kante $\notin M$ anfängt und mit einer Kanten $\notin M$ endet. Einen augmentierenden Pfad kann man nutzen, um ein Matching um 1 zu vergößern.

Lemma 11 (Berge)**.** *M ist ein maximum Matching $\Leftrightarrow$ M hat keinen augmentierenden Pfad.*

Beweis: Wir zeigen: M hat augmentierenden Pfad $\Leftrightarrow$ M nicht maximum.

"$\Rightarrow$": Klar. Sei P augmentierender Pfad. Betrachte $M' = M \oplus P = (M \cup P) \backslash (M \cap P)$. $|M'| = |M| + 1$, also M nicht maximum.

"$\Leftarrow$": Sei M nicht maximum. Betrachte maximum Matching $M^* \neq M$ und $M^* \oplus M$. Jeder Knoten ist inzident zu höchstens einer Kante in M^* und höchstens einer Kante in M. Jede Komponente in $M \oplus M^*$ hat einen der folgenden Typen:

1. isolierter Knoten
2. alternierender Pfad, erste Kante $\in M$, letzte Kante $\in M^*$
3. alternierender Pfad, erste Kante $\in M^*$, letzte Kante $\in M$
4. alternierender Kreis aus Kanten $\in M^*$ und $\in M$
5. alternierender Pfad, erste Kante $\in M$, letzte Kante $\in M$
6. alternierender Pfad, erste Kante $\in M^*$, letzte Kante $\in M^*$

Nur Typ 6 hat mehr Kanten aus M^* als aus M. Aber $|M^*| > |M|$, also muss es eine Komponente vom Typ 6 geben. Dies ist ein augmentierender Pfad für M. □

Algorithm 6: Maximum Matching in allgmeinen Graphen

```
1 M ← ∅
2 while es gibt augmentierenden Pfad P für M do
3 |   M ← M ⊕ P
4 return M
```

Wie finden wir augmentierende Pfade?
In bipartiten Graphen mit Greedy-Ansatz (vgl. Tiefensuche): Starte bei ungematchtem Knoten v, folge allen inzidenten Kanten. Wenn nächster Knoten ungematched, augmentierender Pfad gefunden. Sonst folge (eindeutiger) Matchingkante zum Nachbarn.

Ein augmentierender Pfad muss eine ungerade Anzahl Knoten haben. Wenn er also in v beginnt, kann er nicht in v enden. Jeder Teilpfad, der in v beginnt und endet kann ignoriert werden. Daher findet der Algorithmus einen Pfad wenn er existiert.

In allgemeinen Graphen laufen wir in ungerade Kreise. Der Greedy-Algorithmus denkt, er hat einen augmentierenden Pfad gefunden, besucht aber einen Knoten evtl. mehrmals und benutzt dabei unterschiedliche Kanten $\{v,u\},\{v,w\} \notin M$. So ein Pfad P kann nicht augmentierend sein, denn $M \oplus P$ ist kein Matching.

Edmonds Idee: Kontrahiere ungerade Kreise zu einem Super-Knoten ("Blüte", engl. blossom).

Algorithm 7: High-Level Algorithmus zum Finden eines augmentierenden Pfades

```
1 Beginne bei ungematchtem Knoten
2 Durchsuche Graphen nach alternierendem Pfad mit Greedy-Ansatz
3   Labels alternieren zwischen 0/1
4   Beim 0-Knoten: Suche nach ungematchten Kanten zu besuchtem 0-Knoten
5     → Blüte gefunden, kontrahiere Blüte, suche weiter
6   Sobald wir einen ungematchten 1-Knoten finden, existiert ein augmentierender Pfad
7 Erstelle Pfade aus den durchlaufenen Kanten im umgekehrter Reihenfolge.
8 Expandiere Blüten in umgekehrter Reihenfolge der Kontrahierung
9 In jeder expandierten Blüte wähle passenden alternierenden Pfad im ungeraden Kreis
```

Knotenklassifizierung:

- Ungematchter 1-Knoten: Augmentierender Pfad gefunden!
- Von einem 0-Knoten, wenn wir einen besuchten...
  - 0-Knoten finden: Ungerader Kreis gefunden, Blüte! → Kontraktion
  - 1-Knoten finden: Gerader Kreis gefunden, Kante kann ignoriert werden.

Theorem 5. *Sei H der Graph, der durch Kontraktion einer Blüte in G entsteht.*

$$H \text{ hat augmentierenden Pfad.} \iff G \text{ hat augmentierenden Pfad.}$$

Beweis:
"⇒": Die Blüte (= ungerader Kreis) hat genau einen Knoten, der zwei nicht-gematchte Kanten hat. Wir nennen diesen Knoten die *Basis in G*. Dagegen ist die *Basis in H* einfach der Super-Knoten der Blüte.

Betrachte einen augmentierenden Pfad P_H in H. Wenn er die Basis nicht enthält, sind wir fertig, denn P existiert auch in G. Sei also die Basis (= Blütenknoten) Teil von P_H in H. Es gibt maximal eine Matchingkante von P_H, die an der Basis in H hängt. Den Teil von P, der mit dieser Matchingkante beginnt, nennen wir den *Stamm*. Der Stamm existiert auch in G, und er hängt auch dort an der Basis – dieser Knoten in der Blüte ist der einzige, der eine Matchingkante ausserhalb der Blüte haben kann.

Betrachte nun P_H in G. Er kommt vom Stamm an der Basis der Blüte an und geht dann bei einem anderen Knoten v der Blüte weiter. Vervollständige P_H innerhalb der Blüte: Wähle den (eindeutigen) alternierenden Pfad zu v. Damit existiert auch ein augmentierender Pfad P_G in G.

"⇒": It's complicated :) □

Laufzeit:

- Kontrahierung von Blüten: $O(n+m)$, Anzahl Kontrahierungen höchstens n
- Danach wird Matching um 1 vergrößert. Anzahl Vergrößerungen höchstens $n/2$.

→ $O(n^2(n+m))$, also maximal $O(n^4)$

Mit fancy Datenstrukturen: $O(n^3)$, schnellster bekannter Algo: $O(m\sqrt{n})$ (Vazirani).