Effiziente Algorithmen

Martin Hoefer

Winter 2024/25

Inhaltsverzeichnis

Kapitel 1

Flüsse

1.1 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:
Sei $c(u, v) = c((u, v))$ für wenn $(u, v) \in E$ und $c(u, v) = 0$ sonst.
Ein **Fluss** $f : V \times V \to \mathbb{R}$ erfüllt
1. $f(u, v) \leq c(u, v) \quad \forall u, v \in V$ (Kapazitätsbeschränkung)
2. $f(u, v) = -f(v, u)$ (Symmetrie)
3. $\forall u \in V \setminus \{s, t\}$:

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

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 $x \notin \{s,t\}$. 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 $x \notin \{s,t\}$ mit Flusserhaltung.
Daher haben also alle s-t-Schnitte den Wert $f(S,T) = f(\{s\}, V \setminus \{s\}) = f(\{s\}, V) = |f|$. □

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).

1.1.1 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 **Restnetzwerk** (oder Residualnetzwerk) G_f die **Restkapazitäten** (oder 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)
Für eine Beispielkonstruktion siehe Figure 1.1.

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 Restnetzwerk G_f
- Identifiziere kleinste Restkapazität μ auf P
- Sende zusätzlichen Fluss von μ entlang P

Solch ein s-t-Pfad $P = (s = v_1, \ldots, v_\ell, v_{\ell+1} = t)$ mit $v_i \in V : 2 \leq i \leq \ell$ ist ein **augmentierender Pfad**. Die Flussänderung f_P auf P ist gegeben durch

$$|f_P| \quad = \quad c_f(P) \quad = \quad \min\{c_f(v_i, v_{i+1}) \mid 1 \leq i \leq \ell\}$$

und $f_P(v_i, v_{i+1}) = c_f(P)$, $f_P(v_{i+1}, v_i) = -c_f(P)$ für $i = 1, \ldots, \ell$, sowie $f_P(u,v) = 0$ sonst.
Ein Beispiel für einen augmentierenden Pfad ist in Figure 1.1 der Pfad $P = (s, v_2, v_3, t)$ mit Flussänderung $|f_P| = 4$.

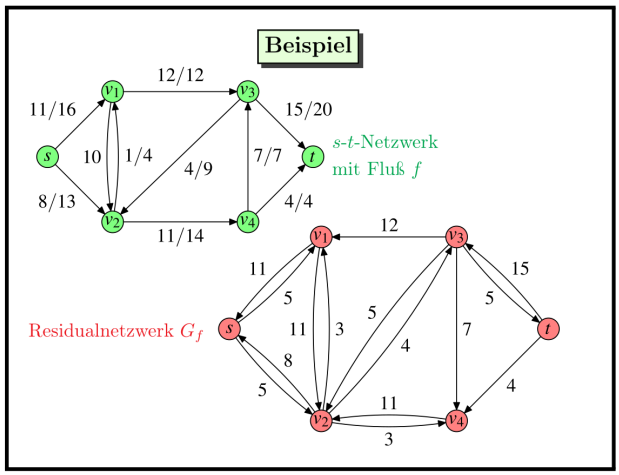

Abbildung 1.1: Beispiel für die Konstruktion eines Restnetzwerks G_f

Algorithm 1: Ford-Fulkerson Algorithmus

```
1 Anfangs f(u,v) = 0 für alle u,v ∈ V.
2 while es gibt s-t-Pfad P in G_f do
3 |  Augmentiere f:   f(u,v) ← f(u,v) + f_P(u,v) für alle u,v ∈ V.
4 return f
```

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 integralen (d.h. ganzzahligen) Kapazitäten JA, jeder Schnitt ergibt obere Schranke auf $|f|$, wir erhöhen in Einheitswerten. Ähnliches Argument wenn die Kapazitäten rationale Zahlen sind: Sei h der Hauptnenner aller Kapazitätswerte. Der Fluss wird erhöht in ganzzahligen Vielfachen von $1/h$.

Zu B: Für integrale Kapazitäten höchstens $|f^*|$ viele Schritte, wobei f^* 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

Algorithm 2: Ford-Fulkerson mit Skalierung

```
1 Anfangs f(u,v) = 0 für alle u,v ∈ V.
2 K ← 2^⌈log2(max{c(u,v)|(u,v)∈E}⌉
3 while K ≥ 1 do
4 |   while es gibt s-t-Pfad P in G_f mit c_f(P) ≥ K do
5 |   |   Augmentiere f:   f(u,v) ← f(u,v) + f_P(u,v) für alle u,v ∈ V.
6 |   K ← K/2
7 return f
```

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: Für integrale Kapazitäten erreicht man eine polynomielle Laufzeit (in n, m, und der logarithmischen Kodierungslänge aller Kapazitäten) durch eine leichte Abwandlung mit **Skalierung** und einen Fokus auf Pfade, die eine möglichst große Flussänderung erzeugen.

Der **Edmonds-Karp Algorithmus** ist eine andere Variante von Ford-Fulkerson. Er wählt immer einen **kürzesten** s-t-Pfad in G_f, d.h., mit der **kleinsten Anzahl Kanten**. Er terminiert sogar in stark polynomieller Zeit (unabhängig von der Kodierungslänge der Zahlen).

Wir betrachten zuerst die Variante mit Skalierung.

Lemma 2. *Der Ford-Fulkerson Algorithmus mit Skalierung berechnet einen maximalen Fluss für integrale Kapazitäten in Zeit $O(m^2 \log C)$ wobei $C = \max\{c(u, v) \mid (u, v) \in E\}$.*

Beweis: Die äußere Schleife erzeugt höchstens $O(\log C)$ verschiedene Werte für K.

Betrachte einen minimalen Schnitt. Die Restkapazität des Schnittes sinkt durchgängig. Wenn kein augmentierender Pfad mit Wert K mehr existiert, dann ist die Restkapazität des Schnittes höchstens $K \cdot m$. Daher ist die Restkapazität immer höchstens $2K \cdot m$. Daraus folgt, dass es für jedes K nur m Augmentierungen geben kann.

In der inneren Schleife braucht jede Augmentierung nur Zeit $O(m)$, wir beschränken uns im Restnetzwerk auf Kanten mit Restkapazität mindestens K und suchen einen s-t-Pfad mit Breitensuche. □

Nun betrachten wir die Edmonds-Karp Variante.

Lemma 3. *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: Durch Widerspruch:

- Sei G_i das Restnetzwerk und δ_i die Distanzen darin zu Beginn von Runde i im Algorithmus.
- Annahme: v ist ein Knoten mit $\delta_i(s, v) > \delta_{i+1}(s, v)$.
- O.B.d.A. wähle v als "nahesten" solchen Knoten in Iteration $i + 1$ mit kleinstem $\delta_{i+1}(s, v)$.

- Es ändern sich nur Kanten entlang des augmentierenden Pfades, und ist v der naheste Knoten, dessen Distanz zu s sinkt.

$\Rightarrow$ v liegt auf dem kürzesten augmentierenden Pfad in G_i.

Betrachten wir nun den kürzesten s-v-Pfad in G_{i+1}. Sei u der Vorgänger von v auf diesem Pfad, d.h. insbesondere $(u, v) \in E[G_{i+1}]$. Ausserdem

$$\delta_{i+1}(s, u) \geq \delta_i(s, u),$$

weil v der naheste Knoten ist, für den sich die Distanz strikt verringert.

Was ist mit (u, v) in G_i?

Fall 1: Sei $(u, v) \in E[G_i]$. Dann gilt $\delta_{i+1}(s, v) = \delta_{i+1}(s, u) + 1 \geq \delta_i(s, u) + 1$. Ausserdem $\delta_i(s, v) \leq \delta_i(s, u) + 1$ da $(u, v) \in E[G_i]$. Daraus folgt $\delta_{i+1}(s, v) \geq \delta_i(s, v)$, ein Widerspruch.

Fall 2: Sei $(u, v) \notin E[G_i]$. Dann geht der augmentierende Pfad von s zu v, zu u, dann zu t – denn die Kante $(u, v) \in E[G_{i+1}]$, sie muss also durch den Pfad entstehen. Dann gilt $\delta_i(s, u) = \delta_i(s, v)+1$ und $\delta_{i+1}(s, u) = \delta_{i+1}(s, v) - 1$. Da $\delta_{i+1}(s, u) \geq \delta_i(s, u)$ gilt $\delta_{i+1}(s, v) \geq \delta(s, v) + 2$, ein Widerspruch. □

[Pic: Anschauung Fall 1 und Fall 2]

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

Beweis: Mit BFS benötigt jede Augmentierung nur $O(m)$ Schritte. Wir zeigen, dass nur $O(nm)$ Augmentierungen nötig sind.

- Sei f_i der Fluss zu Beginn von Runde i.
- Eine Kante (u, v) heißt *kritisch* wenn $f_i(u, v) < c(u, v)$ und $f_{i+1}(u, v) = c(u, v)$.
- Der augmentierende Pfad hat mindenstens eine kritische Kante. Diese Kante wird in Runde i entfernt. Kann sie nochmal zurückkommen?
- Sei $j + 1$ die Runde, in der (u, v) wieder erscheint. $(v, u) \in E[G_j]$ ist Teil des kürzesten augmentierenden Pfades in G_j. Also gilt $\delta_j(s, u) = \delta_j(s, v) + 1$ und $\delta_i(s, v) = \delta_i(s, u) + 1$, sowie $\delta_j(s, v) \geq \delta_i(s, v)$ wegen Lemma 3.

$\Rightarrow$ $\delta_j(s, u) \geq \delta_i(s, u) + 2$.

Da die Distanz $1 \leq \delta_j(s, u) \leq n - 1$, kann jede Kante kann nur höchstens n-Mal kritisch werden (eigentlich sogar nur höchstens $\lceil \frac{n-1}{2} \rceil$ Mal). Es gibt höchstens nm Augmentierungen. □[Pic: Erhöhung des Abstandes]

1.1.2 PreflowPush Algorithmus

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 $v \in V$. [Pic: Intuition, anfangs "zu viel" Fluss im Netzwerk, versuche ihn "bergab" loszuwerden]

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

Algorithm 3: PreflowPush Algorithmus

```
1 Anfangs h(v) = 0 ∀v ≠ s, h(s) = n
2 f(s,v) = c(s,v), f(v,s) = −c(s,v) für alle (s,v) ∈ E und f(u,v) = 0 sonst.
3 while es gibt v ≠ t mit Δ_f(v) > 0 do
4 |   Wähle v mit Δ_f(v) > 0
5 |   if es gibt w mit h(w) < h(v) und (v,w) ∈ E_f then Push(f,h,v,w)
6 |   else Relabel(f,h,v)
```

Algorithm 4: Routine Push(f,h,v,w)

```
1 Sei δ = min{Δ_f(v), c(v,w) − f(v,w)}
2 Setze f(v,w) ← f(v,w) + δ und f(w,v) ← f(w,v) − δ.
```

Algorithm 5: Routine Relabel(f,h,v)

```
1 Höhenanstieg: h(v) ← h(v) + 1.
```

Höhen

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.

Eine **Höhenfunktion** $h : V \to \{0,1,2,\ldots\}$ 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 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 Höhe, erreiche Fluss.

[Pic: Beispieldurchlauf Algorithmus]

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 $h(w) \geq h(v)$.

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

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

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 $2n(n-2)$.*

Beweis: $h(s) = n$, $h(t) = 0$ immer. Betrachte $v \in V \setminus \{s, t\}$, 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 $h(v) - h(s) \leq n - 1$. Damit gibt es höchstens $2n$ Relabel-Operationen für jeden Knoten in $V \setminus \{s, t\}$. □

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. Nach mind. 2 weiteren Erhöhungen von $h(v)$ ist wieder $h(v) = h(w) + 1$, und ein erneuter Push auf (v,w) wird möglich. Mit vorherigem Lemma wird ein saturierender Push höchstens n 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: $2nm(2n-1) + 2n(n-2) \leq 4n^2m$. □

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

1.2 Flüsse mit minimalen Kosten

Wir betrachten wieder ein Flussnetzwerk $G = (V, E, c, s, t)$. Daneben haben wir nun auch einen **vorgegebenen Flusswert** $b \geq 0$ sowie nicht-negative **Kantenkosten** $\ell : E \to \mathbb{R}_{\geq 0}$.

Der Einfachheit halber nehmen wir an, dass es für jedes Paar von Knoten $u, v \in V$ **höchstens eine der beiden Kanten** $(u,v), (v,u)$ **in** E gibt, also $(u,v) \in E \Rightarrow (v,u) \notin E$.

Diese Annahme kann leicht erreicht werden: Wenn $(u, v) \in E$ *und* $(v, u) \in E$ sind, dann ersetze (u, v) durch Kanten $(u, w), (w, v)$ für einen neuen (Hilfs-)Knoten w. Die Kapazität $c(u, w) = c(w, v) = c(u, v)$. Für die Kosten gilt $\ell(u, w) = \ell(u, v)$ und $\ell(w, v) = 0$. Das neue Netzwerk ist offensichtlich äquivalent im Sinne des Flusses und der erzeugten Kosten.

Dann können wir die Kostenfunktion wieder zwischen Knotenpaaren definieren $\ell : V \times V \to \mathbb{R}$ mit
- $\ell(u, v) \geq 0$ und $\ell(v, u) = -\ell(u, v)$ wenn $(u, v) \in E$
- $\ell(u, v) = 0$ sonst.

Die **Kosten eines Flusses** f sind

$$\ell(f) = \sum_{(u,v)\in E} \ell(u, v) \cdot f(u, v) \ .$$

Wir suchen einen **optimalen Fluss** f mit Wert $|f| = b$ und minimalen Kosten.

Zur Beschreibung eines optimalen Flusses betrachten wir folgende Charakterisierung.

Lemma 10. *Ein Fluss f hat minimale Kosten genau dann wenn es in G_f keinen Kreis mit negativen Kosten gibt.*

Beweis: Beide Richtungen durch Widerspruch.
"⇒": Sei K ist ein Kreis in G_f mit negativen Kosten. Annahme: f hat minimale Kosten.

Sei $c_f(K) = \min\{c_f(u, v) \mid (u, v) \in K\}$ die kleinste Restkapazität.
Erhöhen wir nun f entlang K um $c_f(K)$, dann
- bleibt f ein Fluss und hat weiterhin Flusswert $|f| = b$.
- sinken die Kosten von f strikt, denn

$$c_f(K) \cdot \sum_{(u,v)\in K} \ell(u, v) < 0,$$

da K ein Kreis mit negativen Kosten.
Also hatte f nicht minimale Kosten, Widerspruch.

"⇐": Sei f ein suboptimaler Fluss. Dann gibt es f^* mit $\ell(f^*) < \ell(f)$. Annahme: G_f hat keinen Kreis mit negativen Kosten.

Betrachte $\Delta : V \times V \to \mathbb{R}$ mit $\Delta = f^* - f$. Es ist leicht einzusehen, dass
- $\ell(\Delta) = \ell(f^*) - \ell(f) < 0$
- $\Delta(s, V) = \Delta(V, t) = 0$, da $|f| = |f^*| = b$
- $\Delta(v, V) = 0$, da $f(v, V) = 0$ und $f^*(v, V) = 0$.
- Δ ist eine gültige Flussveränderung in G_f, die aus Kreisen besteht.
- Mindestens einer der Kreise hat negative Kosten, da $\ell_f(\Delta) < 0$.

Also gibt es in G_f einen Kreis mit negativen Kosten, Widerspruch. □

Der **Cycle-Canceling Algorithmus** nutzt diese Einsicht algorithmisch aus. Durch eine evtl. schlechte Wahl des Kreises kann im worst-case keine polynomielle Laufzeit garantiert werden. Der Algorithmus gehört auch in der Praxis nicht zu den schnellsten Verfahren.

Hier eine weitere Charakterisierung, die optimale Flüsse beschreibt.

Algorithm 6: Successive Shortest Path Algorithm

Anfangs $f(u,v) = 0$ für alle $u, v \in V$.
Bellman-Ford Algorithmus berechnet kürzeste s-v-Distanzen $\mu(s,v)$ in G_f, für alle $v \in V$
Setze $h_0(v) = \mu(s,v)$, für alle $v \in V$
Setze $i = 0$ und $\ell_0(u,v) = \ell(u,v) + h_0(u) - h_0(v)$, für alle $(u,v) \in E$
while *es gibt s-t-Pfad P in G_f und $|f| < b$* **do**
    `// Berechne kürzeste Wege mit den nicht-negativen Kosten in` G_f
    Dijkstra's Algorithmus berechnet kürzeste s-v-Pfade bzgl. ℓ_i in G_f, für jedes $v \in V$
    Sei $\mu_i(s,v)$ die s-v-Distanz bzgl. ℓ_i in G_f, für jedes $v \in V$
    Sei P ein kürzester s-t-Pfad
    `// Augmentiere Fluss entlang eines kürzesten` s-t-`Weges`
    Setze $f_P(u,v) = 0$ für alle $u, v \in V$
    Setze $f_P(u,v) = \min\{c_f(P), b - |f|\}$, $f_P(v,u) = -f_P(u,v)$ für alle $(u,v) \in P$
    Augmentiere f: $f(u,v) \leftarrow f(u,v) + f_P(u,v)$ für alle $u, v \in V$.
    `// Neue Knotenpotenziale und nicht-negative Kosten in Iteration` $i+1$
    Setze $h_{i+1}(v) = h_i(v) + \mu_i(s,v)$, für jedes $v \in V$
    Setze $\ell_{i+1}(u,v) = \ell(u,v) + h_{i+1}(u) - h_{i+1}(v)$, für alle $(u,v) \in E$
    Setze $i \leftarrow i + 1$
return f

Lemma 11. *Sei f ein Fluss mit minimalen Kosten ist und P ein augmentierender Pfad mit minimalen Kosten. Dann ist $f' = f + f_P$ wieder ein Fluss mit minimalen Kosten.*

Beweis: Durch Widerspruch.
Annahme: P hat minimale Kosten und f' hat keine minimalen Kosten.

Obiges Lemma: G_f hat keinen Kreis mit negativen Kosten, $G_{f'}$ aber schon.

- Sei K ein Kreis K in $G_{f'}$ mit negativen Kosten.
- K ist ein neuer Kreis, der durch die Augmentierung entlang P entstanden ist.
- Dafür muss P durch mindestens eine Kante $(u,v) \in K$ **in entgegengesetzer Richtung** gelaufen sein, d.h. $(v,u) \in P$.
- Wir nehmen einen Pfad P' mit **Umweg**:
  Von s entlang P zu v, dann entlang K zu u, danach weiter entlang P zu t.
- Die Kosten des Umwegs sind kleiner als $\ell(v,u)$, da K negative Kosten hat:

$$\sum_{(x,y)\in K} \ell(x,y) < 0 \quad \Rightarrow \sum_{(x,y)\in K\setminus\{(u,v)\}} \ell(x,y) < -\ell(u,v) = \ell(v,u)$$

Also hat P' strikt geringere Kosten als P, Widerspruch. □

Der **Successive-Shortest-Path Algorithmus** nutzt diese Einsicht algorithmisch aus. Zur Berechnung des kürzesten Weges P kann der Bellman-Ford Algorithmus mit Laufzeit $O(nm)$ genutzt werden. Damit ergibt sich eine Laufzeit von $O(b \cdot nm)$. Mit etwas Preprocessing geht es schneller:

Theorem 4. *Für integrale Kapazitäten kann der Successive-Shortest-Path Algorithmus mit einer Laufzeit von $O(nm + b \cdot (m + n \log n))$ implementiert werden.*

Dafür kann man **Preise** oder **Knotenpotenziale** $h : V \to \mathbb{R}$ nutzen (ähnlich wie die Höhe im PreflowPush Algorithmus):

- G_f enthält anfangs keine negativen Kreise (sonst $f = 0$ kein optimaler Fluss mit Wert $|f| = 0$).
- Nutze Bellman-Ford Algorithmus und berechne kürzeste Wege von s zu jedem anderen Knoten $v \in V$. Wir definieren $h_0(v) = \mu(s, v)$ als Distanz von s nach v. (Preprocessing in Zeit $O(nm)$)
- Da h aus kürzesten Wegen resultiert, gilt $h_0(v) \leq h_0(u) + \ell(u, v)$.
- Damit kann man neue, **nicht-negative Kosten** definieren

$$\ell_0(u, v) = \ell(u, v) + h_0(u) - h_0(v) \geq 0.$$

- Für jeden Knoten $w \in V$ und einen s-w-Pfad P gilt

$$\ell_0(P) = \sum_{(u,v)\in P} \ell(u, v) + h_0(u) - h_0(v) = h_0(s) - h_0(w) + \sum_{(u,v)\in P} \ell(u, v) = h_0(s) - h_0(w) + \ell(P)$$

- Also gilt für s-w-Pfade P und P':

$$\ell(P') \geq \ell(P) \quad \Longleftrightarrow \quad \ell_0(P) \geq \ell_0(P'),$$

  d.h. jeder kürzeste Pfad von s zu einem anderen Knoten bleibt ein kürztester Pfad.
- Negative Kantenkosten in G_f werden vermieden, kürzeste Wege von s bleiben erhalten.
- Wir können **Dijkstra's Algorithmus** für kürzteste Wege benutzen, Laufzeit $O(m + n \log n)$.

Um diese Idee nun in jeder Iteration anzuwenden, müssen wir h anpassen wenn G_f sich ändert.

- Annahme: In der i-ten Iteration sei $\ell_i(u, v) = \ell(u, v) - h_i(u) + h_i(v) \geq 0$ für alle $(u, v) \in E_f$
- Dijkstra berechnet neben den kürztesten Weg auch die Distanzen kürzester Wege $\mu_i(s, v)$ von s zu $V \setminus \{s\}$ in G_f bzgl. ℓ_i.
- Augmentiere entlang P. Dadurch ändert sich G_f. Um weiterhin nicht-negative Kantenkosten zu behalten setzen wir

$$h_{i+1}(v) \leftarrow h_i(v) + \mu_i(s, v).$$

- Dann gilt für jede Kante $(u, v) \in P$

$$\ell_{i+1}(u, v) = \ell(u, v) + h_i(u) - h_i(v) = \ell_i(u, v) + \mu_i(s, u) - \mu_i(s, v) = 0,$$

  jede Kante (v, u) für $(u, v) \in P$

$$\ell_{i+1}(v, u) = \ell(v, u) + h_i(v) - h_i(u) = -\ell_0(u, v) + \mu_0(s, v) - \mu_0(s, u) = 0,$$

  und jede sonstige Kante (u, v)

$$\ell_{i+1}(u, v) = \ell(u, v) + h_i(u) - h_1(v) = \ell_i(u, v) + \mu_i(s, u) - \mu_i(s, v) \geq 0$$

  da $\mu_i(s, v) \leq \mu_i(s, u) + \ell_i(u, v)$.
- Wie oben gilt: Kürzeste s-w-Pfade unter ℓ_{i+1} sind genau die kürzesten s-w-Pfade unter ℓ, für jedes $w \in V$.
- Dies zeigt insbesondere nochmals, dass durch die Augmentierung entlang des kürzesten Pfades P keine negativen Kreise entstehen!

Es gibt Algorithmen mit stark polynomieller Laufzeit wie z.B. den **Min-Mean Cycle-Canceling Algorithmus**. Der Successive-Shortest-Path Algorithmus ist in der Praxis sehr viel schneller.

Kapitel 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.

Sei M ein Matching. Zwei zentrale Definitionen:
- **Alternierender Pfad** für M: Ein Pfad in G mit Kanten abwechselnd $\in M$ und $\notin M$.
- **Augmentierender Pfad** für M: 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.

2.1 Bipartite Graphen

2.1.1 Maximum Matching in bipartiten Graphen

Berechnung von maximum Matchings

Wir nutzen Flüsse, Residualnetzwerke, augmentierende Pfade und den Ford-Fulkerson Algorithmus.
Sei M^* ein maximum Matching in G und f^* ein maximaler Fluss in G'.

Theorem 5. *Es gilt $|M^*| = |f^*|$. Die Kanten in M^* sind genau die Kanten mit Fluss 1 in f^*.*

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.

Algorithm 7: 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** $M = \{\{u, v\} \in A \times B \mid f(u, v) = 1\}$

"$\Leftarrow$": 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 12. *Der Ford-Fulkerson Algorithmus berechnet ein maximum Matching in einem bipartiten Graph in Zeit $O(nm)$.*

- Ford-Fulkerson höchstens $|f^*| = |M^*| \leq n/2$ Iterationen
- Jede Iteration in $O(m)$ Zeit: Erstellung Residualnetzkwerk, BFS für augmentierenden Pfad
- Augmentierender s-t-Pfad für f in G' entspricht augmentierendem Pfad für M in G
- Konstruktion von G' am Anfang und M^* am Ende in Zeit $O(n + m)$.

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

Perfekte 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.
- $|A| = |B|$ (nur dann kann es ein perfektes Matching geben).

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

Theorem 6 (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 $k = |A| = |B|$. Dann gilt: G ein perfektes Matching $\Leftrightarrow$ Max-Fluss in G' hat Wert $|f^*| = k$.

Sei also $|f^*| < k$. Wir zeigen, dass es dann $X \subseteq A$ gibt mit $|X| > |\Gamma(X)|$. 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

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

Das zeigt die Behauptung und das Theorem. □

2.1.2 Bipartite Maximum Matchings mit minimalen Kosten

Wir betrachten eine Variante mit Kantenkosten $\ell : E \to \mathbb{R}_{\geq 0}$. Wir suchen nun wieder ein maximum Matching in G. Unter den maximum Matchings möchten wir ein Matching M^*, das minimale Gesamtkosten $\sum_{e \in M^*} \ell(e)$ aufweist.

Sei $k = \max\{|A|, |B|\}$. Wir erweitern G zu einem **vollständigen bipartiten Graphen** $K_{k,k}$. Darin suchen wir ein optimales **perfektes Matching**:

- Sei A die Partition mit weniger Knoten. Füge $|B| - |A|$ viele Hilfsknoten zu A hinzu. Das ändert die Matchings nicht. Der Graph erüfllt nun $|A| = |B|$.
- Für jede "bipartite Nicht-Kante", also jedes $\{u, v\} A \times B \setminus E$, füge Kante $\{u, v\}$ ein mit extrem hohen Kosten $\ell(\{u, v\}) > \sum_{e' \in E} \ell(e')$.
- Der Graph ist nun vollständig bipartit und hat $|A| = |B|$, d.h. es gibt perfekte Matchings.
- Im optimalen perfekten Matching werden möglichst viele der original vorhandenen Kanten gewählt, denn die sind viel billiger.

$\Rightarrow$ Optimales perfektes Matching in $K_{k,k} \equiv$ Optimales maximum Matching in G

Wir konstruieren wieder das Flussnetzwerk G' wie oben. Für die Optimierung nutzen wir diesmal Flüsse mit minimalen Kosten. Die Kostenfunktion in G' wird dafür wie folgt definiert:

- $\ell(s, u) = \ell(u, s) = 0$ für alle $u \in A$
- $\ell(v, t) = \ell(t, v) = 0$ für alle $v \in B$

- $\ell(u,v) = \ell(\{u,v\}) = -\ell(v,u)$ für alle $\{u,v\} \in A \times B$
- $\ell(x,y) = 0$ sonst.

Daraus ist sofort ersichtlich:

Lemma 13. *Ein Fluss mit Wert $|f| = |A|$ und minimalen Kosten in G' entspricht einem perfekten Matching in $K_{k,k,}$ (und damit einem maximum Matching in G) mit minimalen Kosten.*

Die Optimaltätskriterien für optimale Flüsse in Lemmas 10 und 11 sind auf perfekte Matchings anwendbar. Für ein Matching M betrachten wir alternierende Kreise K mit Kanten abwechselnd $\in M$ und $\notin M$. Die Kosten des Kreises K sind definiert als

$$\ell(K) = \sum_{e \in K \setminus M} \ell(e) - \sum_{e \in K \cap M} \ell(e).$$

Für alternierende Pfade P seien die Kosten analog definiert.

Lemma 14. *M ist ein perfektes Matching mit minimalen Kosten genau dann wenn es für M keinen alternierenden Kreis mit negativen Kosten gibt.*

Beweis: Ähnlich zu Lemma 10:
"⇒": Sei K ein alternierender Kreis mit negativen Kosten. Beachte: K hat gerade Länge, da G bipartit. Betrachte

$$M' = M \oplus K := (M \cup K) \setminus (M \cap K)$$

M' ist ein perfektes Matching - alle Knoten in K werden alternierend "umgematcht". Es gilt $\ell(M') = \ell(M) + \ell(K) < \ell(M)$. M ist also nicht optimal.

"⇐": Betrachte M und ein optimales perfektes Matching M^*. Sei nun

$$C = M \cup M^*$$

Wenn $e \in M \cap M^*$, dann haben die inzidenten Knoten Grad 1 in K. Ansonsten hat jeder Knoten genau Grad 2. C ist eine Vereinigung von einzelnen Kanten und alternierenden Kreisen für M. Da $\ell(M) > \ell(M^*)$ muss es in C mindestens einen Kreis mit negativen Kosten geben. □

Lemma 15. *Sei M ein Matching mit k Kanten und minimalen Kosten. Sei P ein augmentierender Pfad für M mit minimalen Kosten. Dann ist $M \oplus P$ ein Matching mit $k+1$ Kanten und minimalen Kosten.*

Beweis: Ein Matching mit k Kanten und minimalen Kosten entspricht einem Fluss f mit Wert $|f| = k$ und minimalen Kosten. Ein augmentierender Pfad im Restnetzwerk G'_f enspricht einem augmentierenen Pfad für M in G mit den gleichen Kosten. Daher ist das Resultat eine direkte Konsequenz von Lemma 11. □

Theorem 7. *Der Successive-Shortest-Path Algorithmus berechnet ein maximum Matching mit minimalen Kosten für bipartite Graphen in Zeit $O(n^3)$.*

Die Laufzeit ist hier sogar stark polynomiell, da $b = k \leq n$. Da G' aus $K_{k,k}$ gewonnen wird, gilt $m = \Omega(n^2)$ und es folgt eine Laufzeit von $O(n^3)$.

Matching mit maximalem Gewicht

Wir können den Ansatz auch nutzen, um ein Matching mit *maximalem Gewicht* zu berechnen. Sei $G = (A \cup B, E)$ ein bipartiter Graph mit Kantengewichten $w : E \to \mathbb{R}_{\geq 0}$. Wir suchen nun ein Matching M^* mit maximalem Gesamtgewicht $\sum_{e \in M^*} w(e)$.

Eine Reduktion auf perfekte Matchings mit minimalen Kosten:

- Wir verwandeln G wieder in den vollständig bipartiten Graphen $K_{k,k}$ ähnlich wie oben. Füge Dummy-Knoten hinzu damit $|A| = |B|$
- Füge jede "bipartite Nicht-Kante", also jedes Paar $\{u, v\} \in (A \times B) \setminus E$, zu E hinzu mit Gewicht $w(\{u, v\}) = 0$.

$\Rightarrow$ Optimales Matching in $G \equiv$ Optimales perfektes Matching in $K_{k,k}$

- Sei nun $w^* = \sum_{e \in E} w(e)$. Betrachte die Kantenkosten $\ell(e) = w^* - w(e)$. Dann gilt für jedes perfekte Matching M

$$\ell(M) = \sum_{e \in M} (w^* - w(e)) = k \cdot w^* - w(M),$$

- M perfektes Matching mit minimalen Kosten
  $\iff$ M perfektes Matching mit maximalem Gewicht.

Korollar 2. *Der Successive-Shortest-Path Algorithmus berechnet ein Matching mit maximalem Gewicht für bipartite Graphen in Zeit $O(n^3)$.*

2.2 Allgemeine Graphen

Lemma 16 (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 $M' = M \oplus P = (M \cup P) \setminus (M \cap P)$. $|M'| = |M| + 1$, 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 $\in M^*$

[Pics für alle Alternativen]

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. □

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.

Algorithm 8: Maximum Matching in allgmeinen Graphen

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

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 9: High-Level Algorithmus zum Finden eines augmentierenden Pfades

```
1 Beginne bei ungematchtem Knoten
2 Durchsuche Graphen nach alternierendem Pfad mit alternierender Breitensuche
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 8. *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. [Pic: Augmentierender Pfad in H wird in der Blüte erweitert zu augmentierendem Pfad in G] "$\Leftarrow$":

It's complicated :) □

Laufzeit:

- Implementation startet alternierende Breitensuche (BFS) *parallel von jedem isolierten Knoten*
- Augmentierender Pfad: Zwei 0-er Knoten von verschiedenen BFS-Bäumen über eine Nicht-Matching-Kante verbunden.
- Blüte: Zwei 0-er Knoten aus demselben BFS-Baum über Nicht-Matching-Kante verbunden
- Zeit $O(n+m)$ bis (1) Blüte gefunden oder (2) augmentierender Pfad gefunden oder (3) Ende des Algorithmus
- (1) Kontrahierung einer Blüte: $O(n+m)$, Anzahl Kontrahierungen pro Suche höchstens n bis augmentierender Pfad gefunden
- (2) Zeit $O(n+m)$ um den Pfad im Originalgraphen zu erstellen, Matching wächst um eine Kante, maximal $n/2$ Wiederholungen

$\rightarrow$ $O(n \cdot (n+m+n \cdot (n+m))$, also maximal $O(n^2 m)$

Mit fancy Datenstrukturen: $O(n^3)$, schnellste bekannte Algos: $O(m\sqrt{n})$ (z.B. Micali/Vazirani 1980).

Kapitel 3

Lineare Optimierung

3.1 Kanonische Form, Polytope und Ecken

Lineare Optimierung (oder **lineare Programmierung**) hat sehr viele praktische Anwendungen. Wir optimieren eine lineare Zielfunktion in den Variablen $x_1, \ldots, x_n$ bzgl. linearer Nebenbedingungen (sog. **Constraints**). Hier ein einfaches Beispiel mit $n = 4$ Variablen und $m = 5$ Constraints:

$$\begin{array}{lrcl} \text{Minimiere} & 2x_1 - x_2 + 4x_4 & & \\ & & & \\ \text{so dass} & x_1 - 2x_2 + x_3 & \geq & 5 \\ & 2x_1 + 3x_2 - 4x_4 & \geq & -10 \\ & 3x_2 - 2x_3 & \geq & 2 \\ & x_2 & \geq & 0 \\ & x_4 & \geq & 0 \end{array} \tag{3.1}$$

Zur Übersicht nutzen wir Matrix-Vektor-Notation. Mit

$$\boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix}, \quad \boldsymbol{c} = \begin{pmatrix} 2 \\ -1 \\ 0 \\ 4 \end{pmatrix}, \quad \boldsymbol{A} = \begin{pmatrix} 1 & -2 & 1 & 0 \\ 2 & 3 & 0 & -4 \\ 0 & 3 & -2 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \text{ und } \boldsymbol{b} = \begin{pmatrix} 5 \\ -10 \\ 2 \\ 0 \\ 0 \end{pmatrix}$$

können wir das Problem kompakter beschreiben

$$\begin{array}{lrcl} \text{Minimiere} & \displaystyle\sum_{i=1}^{4} c_j x_j & & \\ & & & \\ \text{so dass} & \displaystyle\sum_{i=1}^{4} a_{ij} x_j & \geq & b_i \quad \text{für jedes } i = 1, \ldots, 5 \end{array} \qquad \text{oder einfach} \qquad \begin{array}{ll} \text{Min.} & \boldsymbol{c}^T \boldsymbol{x} \\ \text{s.d.} & \boldsymbol{A}\boldsymbol{x} \geq \boldsymbol{b}. \end{array}$$

Wir nutzen durchgängig diese Formulierung. Alle Einträge in $\boldsymbol{c}$, $\boldsymbol{A}$ und $\boldsymbol{b}$ sind rationale Zahlen.

Definition **lineares Programm (LP) in kanonischer Form**:

- n Variablen und m Constraints
- Vektor $\boldsymbol{x}^T = (x_1, \ldots, x_n)$ der Variablen
- Vektor $\boldsymbol{c} \in \mathbb{Q}^n$ mit Kosten, Matrix $\boldsymbol{A} \in \mathbb{Q}^{m \times n}$, Vektor $\boldsymbol{b} \in \mathbb{Q}^m$

- Das Ziel ist

$$\begin{aligned} \text{Min.} \quad & \boldsymbol{c}^T\boldsymbol{x} \\ \text{s.d.} \quad & \boldsymbol{A}\boldsymbol{x} \geq \boldsymbol{b}. \end{aligned}$$

Jedes LP kann man in kanonische Form bringen:

- Für die **Maximierung** von $\boldsymbol{c}^T\boldsymbol{x}$ nutzen wir **Minimierung** von $-\boldsymbol{c}^T\boldsymbol{x}$.
- Ein **=-Constraint** ist äquivalent zu zwei Constraints:

$$\sum_{j=1}^{n} a_{ij}x_j = b_i \qquad \Longleftrightarrow \qquad \sum_{j=1}^{n} a_{ij}x_j \geq b_i \text{ und } \sum_{j=1}^{n} a_{ij}x_j \leq b_i.$$

- Ein **≤-Constraint** ist äquivalent zu einem **≥-Constraint**:

$$\sum_{j=1}^{n} a_{ij}x_j \leq b_i \qquad \Longleftrightarrow \qquad \sum_{j=1}^{n} -a_{ij}x_j \geq -b_i.$$

Beispiel: Produktionsplanung

- n Produkte, m Ressourcen
- Produkt j erzielt Gewinn $p_j \geq 0$ pro Einheit.
- Um einen Einheit von Produkt j zu produzieren, braucht man a_{ij} Einheiten von Ressource i
- Ein Vorrat von b_i Einheiten von Ressource i ist verfügbar.
- **Ziel:** Bestimme die Einheiten x_j jedes Produktes j um den Gewinn zu maximieren.

Lineares Programm (in kanonischer Form):

- Gesamtgewinn $\sum_{j=1}^{n} p_j x_j$.
- Ressourcen beschränkt verfügbar: $\sum_{j=1}^{n} a_{ij}x_j \leq b_i$
- Wir produzieren keine negativen Einheiten von Produkt j: $x_j \geq 0$.

$$\begin{aligned} \text{Min.} \quad & \sum_{j=1}^{n} -p_j x_j & & & \\ \text{s.d.} \quad & \sum_{j=1}^{n} -a_{ij}x_j & \geq & -b_i & \text{für jedes } i = 1, \ldots, m \\ & x_j & \geq & 0 & \text{für jedes } j = 1, \ldots, n \end{aligned}$$

Beispiel: Min-Cost Flow

- x_e Variable für Fluss über Kante e
- Jeder Kantenfluss ist nicht-negativ und hält Kapazität ein: $0 \leq x_e \leq c_e$ für jedes $e \in E$
- Fluss aus s ist b: $\sum_{(s,v)\in E} x_{(s,v)} = b$
- Flusserhaltung bei den Knoten $v \in V \setminus \{s,t\}$: $\sum_{(u,v)\in E} x_{(u,v)} - \sum_{(v,u)\in E} x_{(v,u)} = 0$
- Minimiere die Kosten $\sum_{e\in E} \ell_e x_e$

Lineares Programm (nicht in kanonischer Form):

$$\begin{aligned} \text{Min.} \quad & \sum_{e\in E} \ell_e x_e & & & \\ \text{s.d.} \quad & \sum_{(s,v)\in E} x_{(s,v)} & = & b & \\ & \sum_{(u,v)\in E} x_{(u,v)} - \sum_{(v,u)\in E} x_{(v,u)} & = & 0 & \text{für jedes } v \in V \setminus \{s,t\} \\ & x_e & \geq & 0 & \text{für jedes } e \in E \\ & x_e & \leq & c_e & \text{für jedes } e \in E \end{aligned}$$

Beispiel: Gewichtetes Vertex Cover

- Verlangt *binäre* Entscheidung für jeden Knoten (Problem wird dadurch NP-hart)
- Sei $G = (V, E)$ mit (der Einfachheit halber $V = \{1, 2, \ldots, n\}$
- $x_v \in \{0, 1\}$ zeigt an, ob Knoten v in der Überdeckung ($x_v = 1$) oder nicht ($x_v = 0$).
- Knoten v hat Kosten $w_v \geq 0$ wenn Teil der Überdeckung.
- Jede Kante soll überdeckt sein. Wie formuliert man das als lineares Constraint?
- $x_u + x_v \geq 1$ für jedes $e = \{u, v\} \in E$.

Integrales lineares Programm (ILP) für gewichtetes Vertex Cover:

$$\begin{array}{lrcll} \text{Min.} & \sum_{v \in V} w_v x_v & & & \\ \text{s.d.} & x_u + x_v & \geq & 1 & \text{for each } \{u, v\} \in E \\ & x_v & \in & \{0, 1\} & \text{für jedes } v \in V \end{array} \tag{3.2}$$

Wenn wir $x_v \in \{0, 1\}$ durch $0 \leq x_v \leq 1$ ersetzen...

- ... erhalten wir ein LP (kein ILP mehr). Es beschreibt das **fraktionale** Vertex-Cover-Problem, in dem wir Knoten in beliebig kleine Teile brechen und diese in der Überdeckung nutzen können.
- Das LP ist eine **lineare Relaxierung** von (3.2) (hat eine Obermenge von gültigen Lösungen). Die optimale fraktionale Lösung kann kleinere Gesamtkosten erreichen.

Betrachte ein LP in kanonischer Form. Definitionen/Beobachtungen:

- Ein LP ist **lösbar** wenn es mind. einen Vektor $\boldsymbol{x}$ gibt mit $\boldsymbol{Ax} \geq \boldsymbol{b}$. Der Vektor heißt **Lösung**.
- Lösungsraum $P = \{\boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{Ax} \geq \boldsymbol{b}\}$ des LP ist *unabhängig von* $\boldsymbol{c}$.
- Für ein Constraint $\sum_{j=1}^n a_{ij} x_j \geq b_i$ ist der Lösungsraum $P_i = \{\boldsymbol{x} \in \mathbb{R}^n \mid \sum_{j=1}^n a_{ij} x_j \geq b_i\}$ ein (geschlossener) **Halbraum**.
- Der Halbraum ist **konvex**: Für jedes Paar $\boldsymbol{x}, \boldsymbol{y} \in P_i$ gilt $\{\lambda \boldsymbol{x} + (1 - \lambda)\boldsymbol{y} \mid \lambda \in [0, 1]\} \subseteq P_i$.
- $P = \bigcap_{i=1}^m P_i$ ist der *Schnitt der Halbräume* aller Constraints. Es ist ein **Polyeder** (oder **Polytop** wenn beschränkt).
- P ist konvex (denn alle Halbräume sind konvex).

Beispiel: Betrachte das LP mit Lösungen aus $\mathbb{R}^2$.

$$\begin{array}{lrcl} \text{Max.} & x_1 + x_2 & & \\ \text{s.d.} & x_1 + 2x_2 & \leq & 3 \\ & 2x_1 + x_2 & \leq & 3 \\ & x_1 & \geq & 0 \\ & x_2 & \geq & 0 \end{array} \tag{3.3}$$

[Pic: Halbräume der Constraints, Lösungen, Hyperebenen, optimale Ecken-Lösungen]

Optimierung eines LP:

- Betrachte den Vektor $\boldsymbol{c}$. Alle Lösungen $\boldsymbol{x}$ mit $\boldsymbol{c}^T \boldsymbol{x} = 0$ haben den gleichen Wert 0.
- $\{\boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{c}^T \boldsymbol{x} = 0\}$ ist eine Hyperebene senkrecht zu $\boldsymbol{c}$
- Allgemein ist $\{\boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{c}^T \boldsymbol{x} = \alpha\}$ die Hyperebene mit allen Lösungen gleichen Wertes $\alpha \in \mathbb{R}$.
- Alle diese Hyperebenen sind parallel und orthogonal zu $\boldsymbol{c}$.
- Maximierung: Wähle die Hyperebene am weitesten in der Richtung von $\boldsymbol{c}$, die noch eine Lösung zu $\boldsymbol{Ax} \geq \boldsymbol{b}$ enthält (Minimierung: in Richtung von $-\boldsymbol{c}$)

- Drei Möglichkeiten für LPs:
  (a) **Ungültiges** LP, leerer Lösungeraum
  (b) **Unbeschränktes** LP, Lösungsraum unbeschränkt (in der Optimierungsrichtung)
  (c) **Beschränktes** LP, Lösungsraum beschränkt (in der Optimierungsrichtung)

[Pic: Ungültig, unbeschränkt, beschränkt, Abhängigkeit von der Richtung von $\boldsymbol{c}$]

Eigenschaften ungültig oder unbeschränkt werden in den Algorithmen "nebenbei" entdeckt.
Wir konzentrieren uns auf beschränkte LPs.

- Betrachte das Lösungspolytop P.
- Eine Seite (in $\mathbb{R}^n$) von P nennt man *Facette*. Sie grenzt P ab vom Rest von $\mathbb{R}^n$.
- Ein *Grat* ist der Schnitt zweier Facetten.
- ...
- Wie viele Facetten müssen sich schneiden für eine **Ecke** von P?
- $\mathbb{R}^n$ hat Dimension n, also brauchen wir mindestens n Facetten (bzw. den Schnitt ihrer Habräume), um eine Ecke zu erzeugen.
- Diese Facetten müssen *linear unabhängig* sein. Genauer gesagt, die Zeilenvektoren $\boldsymbol{a}_i$ der Constraint-Matrix müssen linear unabhängig sein.
- Warum sind Ecken interessant? Es gibt **immer mindestens eine optimale Lösung** in einer der **Ecken** von P.

[Pics: Facetten und Ecken in $\mathbb{R}^2$ und $\mathbb{R}^3$, Beispiele für linear abhängige Constraints]

Formal wird eine **Ecke** $\boldsymbol{x}$ eines Polytops P durch die folgenden zwei äquivalenten Kriterien definiert:

1. $\boldsymbol{x}$ ist ein Punkt, an dem n linear unabhängige Constraints exakt erfüllt sind.
2. Es gibt keinen Vektor $\boldsymbol{y} \in \mathbb{R}^n, \boldsymbol{y} \neq \boldsymbol{0}$, so dass $\boldsymbol{x} + \boldsymbol{y} \in P$ and $\boldsymbol{x} - \boldsymbol{y} \in P$.

Lemma 17. *Die Kriterien für eine Ecke sind gleichbedeutend.*

Beweis: Sei $B \subseteq \{1, \ldots, m\}$ die Menge der Constraints, die bei $\boldsymbol{x}$ exakt erfüllt sind. Sei $\boldsymbol{A}_B = (\boldsymbol{a}_i)_{i \in B}$ die Untermatrix von $\boldsymbol{A}$ mit den Zeilen der Constraints. Sei $\boldsymbol{b}_B$ der Teilvektor der rechten Seiten der Constraints in B. Es gilt also $\boldsymbol{A}_B\boldsymbol{x} = \boldsymbol{b}_B$.

"$\Rightarrow$": Sei $\boldsymbol{x}$ ein Punkt, der 1. erfüllt.

- $\boldsymbol{A}_B$ ist eine $n \times n$-Matrix mit linear unabhängigen Zeilen, also regulär.
- Wenn $\boldsymbol{x} + \boldsymbol{y} \in P$ dann $\boldsymbol{A}_B(\boldsymbol{x} + \boldsymbol{y}) = \boldsymbol{b}_B + \boldsymbol{A}_B\boldsymbol{y} \leq \boldsymbol{b}_B$, also $\boldsymbol{A}_B\boldsymbol{y} \leq 0$.
- Aus $\boldsymbol{x} - \boldsymbol{y} \in P$ folgt genauso $\boldsymbol{A}_B\boldsymbol{y} \geq 0$. Also $\boldsymbol{A}_B\boldsymbol{y} = 0$ und $\boldsymbol{y} = \boldsymbol{0}$, da $\boldsymbol{A}_B$ regulär. Widerspruch.

"$\Leftarrow$": Sei $\boldsymbol{x}$ ein Punkt, der 2. erfüllt.

- Sei $N = \{1, \ldots, m\} \setminus B$, und $\boldsymbol{A}_N$, $\boldsymbol{b}_N$ definiert analog zu oben.
- Wenn $\boldsymbol{A}_B$ regulär, dann gilt 1. – fertig. Sonst hat $\boldsymbol{A}_B$ linear abhängige Zeilen (und Spalten).
- Es gibt also mindestens eine Gerade mit Richtungsvektor $\boldsymbol{z} \neq \boldsymbol{0}$ und $\boldsymbol{A}_B\boldsymbol{z} = \boldsymbol{0}$.
- Also $\boldsymbol{A}_B(\boldsymbol{x} \pm \alpha \cdot \boldsymbol{z}) = \boldsymbol{b}_B$ für jedes $\alpha \in \mathbb{R}$
- Wähle $\boldsymbol{y} = \varepsilon \cdot \boldsymbol{z}$ mit $\epsilon > 0$ klein genug damit $\boldsymbol{A}_N(\boldsymbol{x} + \boldsymbol{y}) \leq \boldsymbol{b}_N$ und $\boldsymbol{A}_N(\boldsymbol{x} - \boldsymbol{y}) \leq \boldsymbol{b}_N$
- Damit gilt $\boldsymbol{A}(\boldsymbol{x} \pm \boldsymbol{y}) \leq \boldsymbol{b}$ und $\boldsymbol{x} \pm \boldsymbol{y} \in P$. □

Es gibt auch **entartete** oder **degenerierte** Ecken, an denen mehr als n linear unabhängige Constraints exakt erfüllt sind. Dann liegt formal eine "Menge von Ecken" an einem Punkt vor. Diese Punkte benötigen eine spezielle Behandlung in den folgenden Algorithmen und der Analyse. Diese Argumente dafür sind eher technisch (und nicht sehr intuitiv). Zum Großteil werden wir entartete Ecken in unserer Diskussion hier ignorieren.

Theorem 9. *Wenn ein beschränktes LP mit n Variablen mindestens n linear unabhängige Constraints hat, dann ist mindestens eine optimale Lösung $\boldsymbol{x}^*$ eine Ecke des Lösungspolytops P.*

Beweis: Sei $\boldsymbol{x}$ eine optimale Lösung, die keine Ecke ist.

- Es gibt $\boldsymbol{y} \neq \boldsymbol{0}$ mit $\boldsymbol{x} + \boldsymbol{y} \in P$ und $\boldsymbol{x} - \boldsymbol{y} \in P$.
- Wenn $\boldsymbol{c}^T\boldsymbol{y} > 0$, dann $\boldsymbol{c}^T(\boldsymbol{x} - \boldsymbol{y}) < \boldsymbol{c}^T\boldsymbol{x}$, d.h. $\boldsymbol{x}$ ist echt schlechter als $\boldsymbol{x} - \boldsymbol{y}$ (Widerspruch).
- Gleiches Argument für $\boldsymbol{c}^T\boldsymbol{y} < 0$ und $\boldsymbol{x} + \boldsymbol{y}$.
- Also muss $\boldsymbol{c}^T\boldsymbol{y} = 0$, dann $\boldsymbol{c}^T\boldsymbol{x} = \boldsymbol{c}(\boldsymbol{x} + \boldsymbol{y}) = \boldsymbol{c}^T(\boldsymbol{x} - \boldsymbol{y})$.
- Auf der Geraden $G = \{\boldsymbol{x} + \lambda\boldsymbol{y} \mid \lambda \in \mathbb{R}\}$ sind alle Punkte optimal.
- P enthält keine Gerade – verschiebe $\boldsymbol{x}$ entlang G zum Rand von P.
- Eine Facette wird getroffen, d.h. mind. ein weiteres Constraint wird erfüllt.
- Falls $\boldsymbol{x}$ (noch) keine Ecke ist, gibt wieder ein $\boldsymbol{y}$ mit $\boldsymbol{x}+\boldsymbol{y}, \boldsymbol{x}-\boldsymbol{y} \in P$. Dann müssen $\boldsymbol{x}+\boldsymbol{y}, \boldsymbol{x}-\boldsymbol{y}$ und alle zukünftigen Verschiebungen von x innerhalb der Facette liegen.

Wiederholt zeigt das Argument, dass wir eine optimale Ecke erreichen. □

Beide Bedingungen im Theorem sind notwendig und hinreichend:

- Wenn P unbeschränkt in der Optimierungsrichtung ist, dann gibt es keine optimale Lösung.
- Wenn das LP beschränkt ist aber keine n linear unabhängigen Constraints hat, dann beschreiben die Constraints einen Subraum, der mindestens eine unendliche Gerade enthält. Die Menge der optimalen Lösungen ist dann unendlich groß.

3.2 Standardform und Simplex Algorithmus

Seit Ende der 1970er gibt es **effiziente Algorithmen zur optimalen Lösung von LPs**. Sie liefern einen optimalen Vektor $\boldsymbol{x}^*$ in Zeit polynomiell in n, m und der Länge der (binären) Codierung[1] aller Parameter a_{ij}, c_i und b_j.

In der Praxis sind sie jedoch oftmals langsamer als ein einfaches Verfahren der lokalen Suche schneller als die Algorithmen mit garantiert polynomieller Laufzeit. Wir geben eine High-Level Beschreibung des **Simplex Algorithmus**. Der Algorithmus ist eine lokale Suche, die sich von einer Ecke zur nächsten bewegt, um die Zielfunktion zu optimieren. Dadurch findet der Algorithmus eine **global optimale Lösung**! Um die lokale Suche anzuwenden, benötigen wir ein Konzept von Nachbarschaft.

Definition 1. *Zwei Ecken $\boldsymbol{x}$ und $\boldsymbol{x}'$ eines konvexen Polytops P sind **Nachbarn** falls es $n-1$ linear unabhängige Constraints gibt, die bei $\boldsymbol{x}$ und $\boldsymbol{x}'$ exakt erfüllt sind.*

Der Simplex Algorithmus bewegt sich zu benachbarten Ecken, die die Zielfunktion verbessern. Er beinhaltet auch konsistente Tie-Breaking-Funktionen, um "Plateus" zu überwinden, in denen die Zielfunktion nicht strikt steigt (z.B. an entarteten Ecken) ohne vorher besuchte Ecken nochmal anzulaufen. Der Algorithmus findet immer eine Möglichkeit weiterzulaufen. Eine Intuition dafür ist folgende Einsicht, dass in P jedes lokale Optimum auch global optimal sein muss.

Lemma 18. *Sei $\boldsymbol{x} \in P$ ein suboptimaler Punkt und $\boldsymbol{x}^* \in P$ die optimale Ecke. Für jedes $\varepsilon > 0$ gibt es ein $\boldsymbol{y} \in P$ mit $||\boldsymbol{x} - \boldsymbol{y}|| \leq \varepsilon$ und $\boldsymbol{c}^T\boldsymbol{y} < \boldsymbol{c}^T\boldsymbol{x}$.*

Beweis: Betrachte den Richtungsvektor $\boldsymbol{z} = \boldsymbol{x}^* - \boldsymbol{x}$.

- P ist konvex, d.h. $\boldsymbol{x} + \lambda\boldsymbol{z} \in P$ für jedes $\lambda \in [0, 1]$. Die Strecke von $\boldsymbol{x}$ nach $\boldsymbol{x}^*$ ist in P.

[1] Ein riesiges offenes Problem ist die Existenz von *stark* polynomiellen Algorithmen, deren Laufzeit nur von n und m abhängt, aber nicht von der Größe der Zahlen (unter der Annahme, dass beliebig lange Zahlen in konstanter Zeit addiert oder multipliziert werden können).

- Entlang der Strecke nimmt der Kostenwert ab: $\boldsymbol{c}^T(\boldsymbol{x} + \lambda \boldsymbol{z}) = \boldsymbol{c}^T\boldsymbol{x} + \lambda \boldsymbol{c}^T\boldsymbol{z}$, und $\boldsymbol{c}^T\boldsymbol{z} = \boldsymbol{c}^T(\boldsymbol{x}^* - \boldsymbol{x}) = \boldsymbol{c}^T\boldsymbol{x}^* - \boldsymbol{c}^T\boldsymbol{x} < 0$.
- Wähle $\boldsymbol{y} \neq \boldsymbol{x}$ auf der Strecke mit Distanz höchstens ε zu $\boldsymbol{x}$. □

Die kanonische Form ist intuitiv, aber der Algorithmus nutzt als Eingabe LPs in **Standardform**. Diese Formulierung ist nützlich für die technische Analyse.

Definition **LP in Standardform**:

- n Variablen und m Constraints
- Vektor $\boldsymbol{x}^T = (x_1, \ldots, x_n)$ mit Variablen
- Vektor $\boldsymbol{c} \in \mathbb{Q}^n$ mit Kosten, Matrix $\boldsymbol{A} \in \mathbb{Q}^{m \times n}$, Vektor $\boldsymbol{b} \in \mathbb{Q}^m$
- Das Ziel ist

$$\begin{aligned} \text{Min.} \quad & \boldsymbol{c}^T\boldsymbol{x} \\ \text{s.d.} \quad & \boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} \\ & \boldsymbol{x} \geq \boldsymbol{0} \end{aligned}$$

Jedes LP in kanonischer Form hat eine äquivalente Formulierung in Standardform:

- Für jede Ungleichung $\sum_{j=1}^{n} a_{ij}x_j \geq b_i$ fügen wir eine **Schlupfvariable** s_i ein und erhalten zwei neue Constraints

$$\sum_{j=1}^{n} a_{ij}x_j - s_i = b_i \qquad \text{and} \qquad s_i \geq 0$$

- Die Standardform verlangt $x_j \geq 0$ für jedes $i = 1, \ldots, n$:
  Wenn $x_j \geq 0$ in kanonischer Form, keine Anpassung.
  Wenn $x_j \leq 0$ in kanonischer Form, ersetze x_j überall durch $-x_j^-$, und $x_j^- \geq 0$.
  Wenn x_j unbeschränkt in kanonischer Form, ersetze x_j überall durch $x_j^+ - x_j^-$, und $x_j^+ \geq 0$, $x_j^- \geq 0$.

Beispiel: Betrachte ein LP und die Transformation in Standardform:

$$\begin{aligned} \text{Min.} \quad & 2x_1 + 4x_2 \\ \text{s.d.} \quad & x_1 + x_2 \geq 3 \\ & 2x_1 + x_2 \geq 14 \\ & x_1 \geq 0 \end{aligned}$$

$$\begin{aligned} \text{Min.} \quad & 2x_1 + 4x_2^+ - 4x_2^- \\ \text{s.d.} \quad & x_1 + x_2^+ - x_2^- - s_1 = 3 \\ & 2x_1 + x_2^+ - x_2^- - s_2 = 14 \\ & x_1 \geq 0 \\ & x_2^+, x_2^- \geq 0 \\ & s_1, s_2 \geq 0 \end{aligned}$$

Beobachtungen:

- Wir haben drei weitere Variablen eingeführt. Anstatt ein 2-dimensionales $(x_1, x_2) \in \mathbb{R}^2$ suchen wir nun eine 5-dimensionale Lösung $(x_1, x_2^+, x_2^-, s_1, s_2) \in \mathbb{R}^5$.
- Durch s_1 und s_2 haben wir auch zwei Gleichheitsconstraints erhalten. Dies erhält die Dimension des Lösungsraumes und auch seine Struktur.
- Die Transformation $x_j = x_j^+ - x_j^-$ kann die Dimension des Lösungsraumes echt verändern und evtl. auch neue Ecken erzeugen!

Beispiel:

$$\begin{aligned} \text{Min.} \quad & c_1x_1 + c_2x_2 \\ \text{s.d.} \quad & x_1 + x_2 \leq 1 \\ & x_1 \geq 0 \\ & x_2 \geq 0 \end{aligned}$$

$$\begin{aligned} \text{Min.} \quad & c_1x_1 + c_2x_2 \\ \text{s.d.} \quad & x_1 + x_2 - s_1 = 1 \\ & x_1 \geq 0 \\ & x_2 \geq 0 \\ & s_1 \geq 0 \end{aligned}$$

[Pic: Lösungssraum = Dreieck, Einbettung in 3-dimensionalen Raum]

Ecken in der Standardform:

- Von den m Gleichheitsconstraints sind $k \leq n$ linear unabhängig
- Wir können annehmen, dass $k = m$, da sonst redundante Gleichungen existieren.
- Jedes Gleichheitsconstraint reduziert die Dimension von P um 1 (da linear unabhängig)
- Wenn $P \neq \emptyset$, dann hat $P \subseteq \mathbb{R}^n$ genau $n - m$ Dimensionen.
- Betrachte eine Ecke $\boldsymbol{x}$ von P. Da $\boldsymbol{x} \in P$, sind m Gleichheitsconstraints exakt erfüllt.
- $\boldsymbol{x}$ erfüllt $n - m$ weitere Constraints (der Form $x_j \geq 0$) exakt, also $x_j = 0$ für $n - m$ Variablen x_j. Es gibt also eine Menge $N \subseteq \{1, \ldots, n\}$ mit $n - m$ Variablen und $x_j = 0$, für jedes $j \in N$.
- (Wenn $x_j = 0$ für mehr als $n - m$ Variablen, dann ist $\boldsymbol{x}$ eine entartete Ecke!)

Für eine gegebene Ecke $\boldsymbol{x}$ definieren wir Basis- und Nichtbasisvariablen:

- $j \in N$ heißt eine **Nichtbasiskomponente**, x_j eine **Nichtbasisvariable**
- Die restlichen $B = \{1, \ldots, n\} \setminus N$ heißen eine **Basis**
- $j \in B$ ist eine **Basiskomponente**, x_j eine **Basisvariable**.
- Falls $\boldsymbol{x}$ nicht entartet, dann gilt $x_j > 0$ für jedes $j \in B$.
- Wir nennen den Spaltenvektor $\boldsymbol{a}^j$ einer Basisvariablen x_j einen **Basisvektor**.

Mehr zu Basisvariablen:

- $\boldsymbol{A}_B$ ist die $(m \times m)$-Untermatrix von Basisvektoren aus $\boldsymbol{A}$
- $\boldsymbol{A}_N$ ist die Untermatrix der verbleibenden (Nichtbasis-)Spaltenvektoren von $\boldsymbol{A}$
- $\boldsymbol{A}_B$ hat k linear unabhängige Zeilen $\Rightarrow$ $\boldsymbol{A}_B$ regulär.
- $\boldsymbol{x}_B$ bezeichne den Subvektor von $\boldsymbol{x}$ für Basisvariablen, $\boldsymbol{x}_N$ den für Nichtbasisvariablen
- Für $\boldsymbol{c}$ machen wir die Einteilung in $\boldsymbol{c}_B$ und $\boldsymbol{c}_N$ entsprechend
- Beachte, dass $\boldsymbol{b} = \boldsymbol{A} \cdot \boldsymbol{x} = \boldsymbol{A}_B\boldsymbol{x}_B + \boldsymbol{A}_N\boldsymbol{x}_N = \boldsymbol{A}_B\boldsymbol{x}_B$, da $\boldsymbol{x}_N = \boldsymbol{0}$ an der Ecke $\boldsymbol{x}$
- Also gilt: $\boldsymbol{x}_B = \boldsymbol{A}_B^{-1}\boldsymbol{b} \geq \boldsymbol{0}$ an der Ecke $\boldsymbol{x}$

Allgemeiner gilt für jede Basis B und jede Lösung $\boldsymbol{x} \in P$, dass

$$\boldsymbol{x}_B = \boldsymbol{A}_B^{-1}\boldsymbol{b} + \boldsymbol{A}_B^{-1}\boldsymbol{A}_N\boldsymbol{x}_N \tag{3.4}$$

Für die Kosten gilt

$$\boldsymbol{c}^T\boldsymbol{x} = \boldsymbol{c}_B^T\boldsymbol{x}_B + \boldsymbol{c}_N^T\boldsymbol{x}_N = \boldsymbol{c}_B^T\boldsymbol{A}_B^{-1}\boldsymbol{b} + \underbrace{(\boldsymbol{c}_N^T - \boldsymbol{c}_B^T\boldsymbol{A}_B^{-1}\boldsymbol{A}_N)}_{:= \bar{\boldsymbol{c}}^T}\boldsymbol{x}_N$$

Hauptroutine des Simplex Algorithmus (sehr High-Level und ohne entartete Ecken):

- Benachbarte Ecken $\boldsymbol{x}$ und $\boldsymbol{x}'$ unterscheiden sich in exakt einem erfüllten Constraint
- Für die Mengen N und N' gilt $N' = (N \setminus \{k\}) \cup \{j\}$ für $j, k \in \{1, \ldots, n\}, j \neq k$
- Die Basen B und B' ändern sich entsprechend. Wie bekommt man einen Basiswechsel?
- Falls $\bar{c}_j \geq 0$, für jedes $j \in N$, dann ist $\boldsymbol{x}$ schon eine optimale Ecke.
- Sonst wähle ein j mit $c_j \leq 0$. In Richtung einer Erhöhung von x_j sinkt die Zielfunktion.
- Bewege die Lösung so weit wie möglich in diese Richtung: Erhöhe x_j so weit, dass für $\boldsymbol{x}_B$ (gegeben durch (3.4)) noch gilt $\boldsymbol{x}_B \geq 0$
- Wir können x_j beliebig weiter erhöhen $\rightarrow$ LP ist unbeschränkt.
- Ansonsten wird (mindestens) eine Basisvariable $x_k \geq 0$ verletzen. Dann wird k aus B entfernt, j zu B hinzugefügt, und der Basiswechsel von B zu B' ist abgeschlossen.

Laufzeiten:

- Jede Iteration im Simplex benötigt polynomielle Zeit (eher aufwändig: Matrix-Inversion)
- Es gibt Polytope in $\mathbb{R}^n$ und initiale Lösungen, so dass der Algorithmus $2^{\Omega(n)}$ Ecken bensuchen muss. Simplex hat **keine polynomielle Laufzeit** im worst case.
- Dennoch sehr oft sehr schnell in der Praxis (polynomielle Smoothed-Komplexität!)
- Simplex wurde in den 1940ern entwickelt. 1979 wurde der erste Algorithmus mit garantierter Polynomzeit entwickelt: **Ellipsoid** (allerdings nicht sehr praktikabel)
- Seit der Mitte der 1980er: Polynomielle **Interior-Point-Verfahren**, siehe unten. Gut für große LPs, insbesondere wenn approximativ-optimale Lösungen ausreichen

Unbeschränktheit wird im Algorithmus gefunden. Was ist mit Ungültigkeit?

- Finden einer gültigen Lösung für ein LP in Standardform
- Wenn $b_i < 0$, dann setze $a_{ij} \leftarrow -a_{ij}$ und $b_i \leftarrow -b_i$. Dann gilt $\boldsymbol{b} \geq \boldsymbol{0}$.
- Neues LP mit Variablen $\boldsymbol{x}$ und neuen Variablen $\boldsymbol{y} = (y_1, \ldots, y_m)^T$:

$$\begin{aligned} \text{Min.} \quad & \sum_{i=1}^{m} y_i \\ \text{s.d.} \quad & \boldsymbol{A}\boldsymbol{x} + \boldsymbol{I}\boldsymbol{y} = \boldsymbol{b} \\ & \boldsymbol{x} \geq \boldsymbol{0} \\ & \boldsymbol{y} \geq \boldsymbol{0} \end{aligned}$$

- Gültige Anfangslösung für dieses LP ist $\boldsymbol{y} = \boldsymbol{b}$ und $\boldsymbol{x} = \boldsymbol{0}$. Starte Simplex auf diesem LP.
- Optimum mit $\boldsymbol{y}^* = \boldsymbol{0}$: $\boldsymbol{x}^*$ ist gültig für originales LP. Starte Simplex auf originalem LP.
- Sonst: Originales LP ist ungültig.

3.3 Seidels Algorithmus

Wir betrachten einfaches Verfahren für LPs mit m Constraints und konstant vielen Variablen (z.B. $n = 2$ oder $n = 3$). Das Verfahren hat dann sogar **lineare Laufzeit** $O(m)$.

Algorithm 10: Seidels Algorithmus

```
1  Sei C die Menge der Constraints (ohne Box-Constraints)
2  if |C| = 1 oder n = 1 then berechne optimale Lösung x* und return x*
3  Wähle ein Constraint i ∈ C uniform zufällig
4  Rekursiver Aufruf für LP mit gleicher Zielfunktion und Constraints C \ {i}
5  Sei x*_{-i} die Ausgabe des rekursiven Aufrufs
6  if x*_{-i} ist auch gültige Lösung für C then return x* = x*_{-i}
7  Berechne die n − 1-dimensionale Facette F_i von i in P
8  Rekursiver Aufruf für das LP mit gleicher Zielfunktion und Lösungsraum F_i
9  Sei x*_{F_i} die Ausgabe des rekursiven Aufrufs
10 return x*_{F_i}
```

Annahmen:

- Das LP ist lösbar (und damit alle LPs in den rekursiven Aufrufen.
- Keines der LPs in den rekursiven Aufrufen ist **unbeschränkt**.
- Füge **Box-Constraints** hinzu $t \leq x_i \leq t$ für hinreichend großes t, so dass $P \subset [-t, t]^n$.
- So ein t kann in polynomieller Zeit berechnet werden.

- t kann auch symbolisch "mitgeschleppt" und adaptiv angepasst werden

[Pic: Beispiel Durchlauf des Algorithmus]

Theorem 10. *Seidels Algorithmus berechnet die optimale Lösung eines LPs in Zeit $O(n! \cdot m)$.*

Die Korrektheit des Algorithmus ist einfach einzusehen. Wir konzentrieren uns auf die Laufzeit der verschiedenen Schritte.
Laufzeit für Basisfälle und Tests:

- Nur noch 1 Variable und m Constraints: Bestimme Optimum in Zeit $O(m)$
- Falls $x_i^* \in (-t, t)$ ist das LP beschränkt, sonst optimale Lösung $x_i^* \in \{-t, t\}$
- Nur noch 1 Constraint: Fraktionales Rucksackproblem
- Greedy-Algorithmus in Zeit $O(n^2)$ (genauer $O(n \log n)$, aber quadratisch reicht uns)
- Test ob $\boldsymbol{x}_{-i}^*$ weiter gültig: Einsetzen von $\boldsymbol{x}_{-i}^*$ in Constraint i, Laufzeit $O(n)$

Laufzeit für Berechnung von F_i:

- F_i erfüllt die Gleichung $\boldsymbol{a}_i \boldsymbol{x} = b_i$
- Löse zu einer Variablen x_j auf und ersetze x_j durch den Ausdruck in Zielfunktion und allen Constraints (auch den Box-Constraints)
- Füge die angepassten Box-Constraints für x_j als "normale" Constraints hinzu.
- Laufzeit zur Berechnung $O(nm)$

Laufzeitbestimmung über Rekursionsgleichung:

- Sei $T(n, m)$ die Gesamtlaufzeit für das LP. Laufzeiten für die Abschnitte:
- Schritt 1-3: $O(1)$
- Rekursion in Schritt 4: $T(n, m-1)$
- Schritt 5+6: $O(n)$
- Schritt 7: $O(nm)$
- Schritt 8: $T(n-1, m+1)$

Aber die Rekursion mit F_i wird nicht immer ausgeführt!

Lemma 19. *Die Wahrscheinlichkeit, dass der Algorithmus einem rekursiven Aufruf Schritt 7 erreicht, ist höchstens n/m.*

Beweis: Die optimale Lösung $\boldsymbol{x}^*$ ist eine Ecke, die aus n Constraints gebildet wird.

- Sei D die Menge dieser n Constraints. D enthält evtl. Box-Constaints.
- Schritt 7-9 werden nur ausgeführt wenn $i \in D$, sonst führt die Herausnahme von i zu keiner anderen Optimallösung

Die Wahrscheinlichkeit, dass $i \in D$ ist höchstens $|D|/|C| = n/m$ – evtl. kleiner, denn Box-Constratins in D werden nie ausgewählt. □

Für die erwartete Laufzeit $T(n, m)$ gilt (konstante Faktoren werden vernachlässigt!):

$$T(n, m) = T(n, m-1) + n^2 + \frac{n}{m} \cdot T(n-1, m-1)$$

Lemma 20. *Die Laufzeit der Rekursion ist $T(n, m) \in O(n! \cdot m)$*

Beweis: Sei

$$f(n) = n \cdot f(n-1) + 3n^3 \qquad \text{mit } f(1) = 1.$$

Dann ist

$$f(n) = n! + \sum_{k=2}^{n} 3k^3 \cdot \frac{n!}{(k-1)!} = O(n!) \qquad \text{denn} \quad \sum_{k=2}^{n} \frac{3n^3}{(k-1)!} = O(1).$$

Wir nutzen $f(n)$ für die Abschätzung von $T(n,m)$ und zeigen induktiv

$$T(n,m) \leq (m-1)f(n) + 2n^2$$

Basisfälle oben:
$n = 1$: $T(1,m) \leq m \leq (m-1)f(n) + 2n^2$
$m = 1$: $T(n,1) \leq n^2 \leq (m-1)f(n) + 2n^2$

Schritt:
Sei nun $n, m \geq 2$. Sei $k = 2n+m$. Wir nehmen an, die Annahme ist gezeigt für alle Paare $n', m' \geq 2$ so dass $2n' + m' < k$ – also insbesondere für $n, m-1$ und $n-1, m+1$.
Aussage für n, m folgt durch Einsetzen und Umformen:

$$\begin{aligned}
T(n,m) &= T(n-1,m) + n^2 + \frac{n}{m} \cdot T(n-1,m+1) \\
&\leq (m-2)f(n) + 2n^2 + n^2 + \frac{n}{m} \cdot (mf(n-1) + 2(n-1)^2) \\
&\leq (m-2)f(n) + 3n^2 + \frac{n}{m} \cdot (mf(n-1)) + 2n^2 \\
&= (m-2)f(n) + 3n^2 + n \cdot \frac{f(n) - 3n^2}{n} + 2n^2 \\
&= (m-2)f(n) + f(n) + 2n^2 \\
&= (m-1)f(n) + 2n^2
\end{aligned}$$

□

3.4 Dualität

LPs haben eine überraschende Dualität – sie ergeben sich in Paaren. Das *duale LP* ergibt sich aus der Aufgabe, eine gute Schranke auf den Zielfunktionswert der optimalen Lösung eines (hier genannt *primalen*) LPs zu finden. Die Optimierung der Schranke auf den besten Wert des primalen LPs kann als ein anderes, das *duale* LP formuliert werden. Primale und duale LPs sind eng verbunden. Dualität liefert elegante Strukturen, die sehr nützlich ist, z.B. beim Entwurf von Approximationsalgorithmen.

Wir möchten eine gute *untere Schranke* auf den Optimalwert dieses (primalen) LPs finden:

$$\begin{array}{lrcl}
\text{Min.} & 7x_1 + x_2 + 5x_3 & & \\
\text{s.d.} & x_1 - x_2 + 3x_3 & \geq & 10 \\
& 5x_1 + 2x_2 - x_3 & \geq & 6 \\
& x_1, x_2, x_3 & \geq & 0
\end{array} \tag{3.5}$$

Wir nutzen die Constraints:

- Das erste Constraint liefert eine einfache untere Schranke von 10 auf jede gültige Lösung:

  $$10 \leq x_1 - x_2 + 3x_3 \leq 7x_1 + x_2 + 5x_3$$

  denn x_1, x_2, x_3 sind nicht-negativ.
- Durch Kombination von zwei Constraints erhöht sich die Schranke auf 16:

  $$10 + 6 \leq (x_1 - x_2 + 3x_3) + (5x_1 + 2x_2 - x_3) = 6x_1 + x_2 + 2x_3 \leq 7x_1 + x_2 + 5x_3$$

- Schauen wir die Koeffizienten genauer an, verstärken wir die Schranke auf 26:

$$2 \cdot 10 + 1 \cdot 6 \leq 2 \cdot (x_1 - x_2 + 3x_3) + 1 \cdot (5x_1 + 2x_2 - x_3) = 7x_1 + 5x_3 \leq 7x_1 + x_2 + 5x_3$$

Welches sind die optimalen Werte y_1, y_2 so dass

$$y_1 \cdot 10 + y_2 \cdot 6 \leq y_1 \cdot (x_1 - x_2 + 3x_3) + y_2 \cdot (5x_1 + 2x_2 - x_3) \leq 7x_1 + x_2 + 5x_3?$$

- $y_1, y_2 \geq 0$, denn sonst drehen sich die Ungleichungen.
- $y_1 + 5y_2 \leq 7$: die untere Schranke soll weniger von $x_1 \geq 0$ nutzen als die Zielfunktion.
- Gleiches gilt analog für x_2 und x_3.
- Dies ergibt ein neues Optimierungsproblem.

Finde die besten Werte, um die untere Schranke für den optimalen Wert der Zielfunktion von LP (3.5) zu maximieren:

$$\begin{array}{lrcl} \text{Max.} & 10y_1 + 6y_2 & & \\ \text{s.d.} & y_1 + 5y_2 & \leq & 7 \\ & -y_1 + 2y_2 & \leq & 1 \\ & 3y_1 - 2y_2 & \leq & 5 \\ & y_1, y_2 & \geq & 0 \end{array} \tag{3.6}$$

Dies ist das **duale LP** des (primalen) LP (3.5). Bemerkungen:

- Koeffizienten der Zielfunktion des Dualen = Koeffizienten der rechten Seite des Primalen.
- Koeffizienten der rechten Seite des Dualen = Koeffizienten der Zielfunktion des Primalen.
- Reihen der primalen Constraintmatrix = Spalten der dualen Constraintmatrix.
- Minimierung im Primalen $\rightarrow$ Maximierung im Dualen
- Wenden wir die gleichen Argumente auf das duale LP (3.6) an, erzeugen wir das primale LP (3.5). **Das Duale des Dualen ist das Primale**!

Betrachte ein allgemeines LP in kanonischer Form mit $\boldsymbol{c} \in \mathbb{Q}^n$, $\boldsymbol{b} \in \mathbb{Q}^m$, $\boldsymbol{A} \in \mathbb{Q}^{m \times n}$, in dem auch $\boldsymbol{x} \geq \boldsymbol{0}$ gilt. Dann folgt mit den Argumenten oben, dass primales und duales LP gegeben sind durch

$$\begin{array}{lrcl} \text{Min.} & \boldsymbol{c}^T\boldsymbol{x} & & \\ \text{s.d.} & \boldsymbol{A}\boldsymbol{x} & \geq & \boldsymbol{b} \\ & \boldsymbol{x} & \geq & \boldsymbol{0} \end{array} \qquad \begin{array}{lrcl} \text{Max.} & \boldsymbol{y}^T\boldsymbol{b} & & \\ \text{s.d.} & \boldsymbol{y}^T\boldsymbol{A} & \leq & \boldsymbol{c}^T \\ & \boldsymbol{y} & \geq & \boldsymbol{0} \end{array} \tag{3.7}$$

Die dualen Constraints sind $(\boldsymbol{y}^T\boldsymbol{A})^T \leq (\boldsymbol{c}^T)^T$, also $\boldsymbol{A}^T\boldsymbol{y} \leq \boldsymbol{c}$ wie in LP (3.6).

Durch die Konstruktion ergibt sich direkt: Der Wert *jeder dualen Lösung* ist eine untere Schranke auf den Wert *jeder primalen Lösung*!

[Pic: primale und duale Lösungswerte]

Maximierung im primalen LP in (3.7)?

- Suche eine *obere Schranke* auf den opimalen Wert.
- Nutze eine Linearkombination der Constraints für so eine obere Schranke.
- Duales LP: Finde eine optimale Kombination der Constraints für die kleinste Schranke.
- Gleiche Schritte wie oben: Duales ist das Minimierungs-LP in (3.7).
- Das Duale des Dualen ist das Primale!

Hier ein generisches "Rezept" zur Erstellung des Dualen. Minimierung wird zu Maximierung und umgekehrt. Tausche die Rollen von Parametern in Zielfunktion $\boldsymbol{c}$ und rechter Seite $\boldsymbol{b}$. Variablen in einem LP entsprechen Constraints im anderen LP:

- $y_i \geq 0$ entspricht Constraint $\boldsymbol{a}_i\boldsymbol{x} \geq b_i$,
- $y_i \leq 0$ entspricht Constraint $\boldsymbol{a}_i\boldsymbol{x} \leq b_i$ und
- y_i frei entspricht Constraint $\boldsymbol{a}_i\boldsymbol{x} = b_i$.

und

- $x_j \geq 0$ entspricht Constraint $\boldsymbol{y}^T\boldsymbol{a}^j \leq c_j$,
- $x_j \leq 0$ entspricht Constraint $\boldsymbol{y}^T\boldsymbol{a}^j \geq c_j$ und
- x_j frei entspricht Constraint $\boldsymbol{y}^T\boldsymbol{a}^j = c_j$.

Hier ist $\boldsymbol{a}^j$ der Spaltenvektor von $\boldsymbol{A}$ für Variable x_j.

Schematisch sind primales und duales LP wie folgt in Beziehung:

$$\begin{array}{rrcl} \text{Min.} & \boldsymbol{c}^T\boldsymbol{x} & & \\ \text{s.d.} & \boldsymbol{a}_i\boldsymbol{x} & \geq & b_i \\ & \boldsymbol{a}_i\boldsymbol{x} & \leq & b_i \\ & \boldsymbol{a}_i\boldsymbol{x} & = & b_i \\ & x_j & \geq & 0 \\ & x_j & \leq & 0 \\ & x_j & & \text{frei} \end{array} \quad \longleftrightarrow \quad \begin{array}{rrcl} \text{Max.} & \boldsymbol{y}^T\boldsymbol{b} & & \\ \text{s.d.} & y_i & \geq & 0 \\ & y_i & \leq & 0 \\ & y_i & & \text{frei} \\ & \boldsymbol{y}^T\boldsymbol{a}^j & \leq & c_j \\ & \boldsymbol{y}^T\boldsymbol{a}^j & \geq & c_j \\ & \boldsymbol{y}^T\boldsymbol{a}^j & = & c_j \end{array} \tag{3.8}$$

Die Transformation **geht in beide Richtungen**.

Beispiele: LP in Standardform

$$\begin{array}{rrcl} \text{Min.} & 13x_1 + 10x_2 + 6x_3 & & \\ \text{s.d.} & 5x_1 + x_2 + 3x_3 & = & 8 \\ & 3x_1 + x_2 & = & 3 \\ & x_1, x_2, x_3 & \geq & 0 \end{array} \qquad \begin{array}{l} \text{Min.} \begin{bmatrix} 13 & 10 & 6 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \\ \\ \text{s.d.} \begin{bmatrix} 5 & 1 & 3 \\ 3 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 8 \\ 3 \end{bmatrix} \\ \boldsymbol{x} \geq \boldsymbol{0} \end{array}$$

Das duale LP ist

$$\begin{array}{rrcl} \text{Max.} & 8y_1 + 3y_2 & & \\ \text{s.d.} & 5y_1 + 3y_2 & \leq & 13 \\ & y_1 + y_2 & \leq & 10 \\ & 3y_1 & \leq & 6 \\ & y_1, y_2 & \in & \mathbb{R} \end{array} \qquad \begin{array}{l} \text{Max.} \begin{bmatrix} y_1 & y_2 \end{bmatrix} \cdot \begin{bmatrix} 8 \\ 3 \end{bmatrix} \\ \\ \text{s.d.} \begin{bmatrix} y_1 & y_2 \end{bmatrix} \cdot \begin{bmatrix} 5 & 1 & 3 \\ 3 & 1 & 0 \end{bmatrix} \leq \begin{bmatrix} 13 & 10 & 6 \end{bmatrix} \end{array}$$

Ein weiteres Paar primaler und dualer LPs – sei $G = (V, E)$ ein ungerichteter Graph.

$$\begin{array}{rlcll} \text{Min.} & \displaystyle\sum_{v \in V} x_v & & & \\ \text{s.d.} & x_u + x_v & \geq & 1 & \text{für jedes } e \in E \\ & x_v & \leq & 0 & \text{für jedes } v \in V \end{array} \qquad \begin{array}{rlcll} \text{Max.} & \displaystyle\sum_{e \in E} y_e & & & \\ \text{s.d.} & \displaystyle\sum_{\{u,v\} \in E} y_{\{u,v\}} & \leq & 1 & \text{für jedes } v \in V \\ & y_e & \geq & 0 & \text{für jedes } e \in E \end{array}$$

Dies sind LPs für **fraktionales Vertex Cover und fraktionales Matching**! Vertex Cover und Matching sind duale Probleme.

Wir verallgemeinern die Argumente von oben auf allgemeine LPs. Betrachte eine Lösung $\boldsymbol{x}$ für ein Minimierungs-LP und Lösung $\boldsymbol{y}$ des dualen Maximierungs-LPs.

Lemma 21 (Schwache Dualität)**.** *Für jede Lösung $\boldsymbol{x}$ eines primalen LPs und jede Lösung $\boldsymbol{y}$ des entsprechenden dualen LPs gilt*

$$\boldsymbol{c}^T\boldsymbol{x} \geq \boldsymbol{y}^T\boldsymbol{b}.$$

Der Wert jeder primalen Lösung ist größer als der Wert jeder dualen Lösung.

Beweis:

1. Betrachte die Kombination aus Constraints $i = 1, \ldots, m$ mit $\boldsymbol{a}_i\boldsymbol{x}$ und b_i sowie die jeweilige Variable y_i. Die Richtung der Constraint-Ungleichung und (Nicht-)Negativität von y_i stellen sicher, dass
$$y_i \cdot (\boldsymbol{a}_i\boldsymbol{x} - b_i) \geq 0.$$
Summierung über diese Ungleichungen ergibt
$$\sum_{i=1}^{m} y_i \cdot (\boldsymbol{a}_i\boldsymbol{x} - b_i) = \boldsymbol{y}^T\boldsymbol{A}\boldsymbol{x} - \boldsymbol{y}^T\boldsymbol{b} \geq 0,$$
also $\boldsymbol{y}^T\boldsymbol{b} \leq \boldsymbol{y}^T\boldsymbol{A}\boldsymbol{x}$.
2. Betrachte die Kombination aus Variablen x_j mit $j = 1, \ldots, n$ sowie dem jeweiligen Constraint mit $\boldsymbol{y}^T\boldsymbol{a}^j$ und c_j. Die Richtung der Constraint-Ungleichung und (Nicht-)Negativität von y_i stellen sicher, dass
$$(c_j - \boldsymbol{y}^T\boldsymbol{a}^j) \cdot x_j \geq 0.$$
Summierung über diese Ungleichungen ergibt
$$\sum_{j=1}^{n} (c_j - \boldsymbol{y}^T\boldsymbol{a}^j)x_j = \boldsymbol{c}^T\boldsymbol{x} - \boldsymbol{y}^T\boldsymbol{A}\boldsymbol{x} \geq 0,$$
also $\boldsymbol{y}^T\boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{c}^T\boldsymbol{x}$. □

Bemerkungen und Konsequenzen:

- Das Duale des Dualen ist das Primale.
- Wenn wir das primale LP äquivalent umformen (z.B. in Standardform), dann ist das duale LP des transformierten primalen LP äquivalent zum dualen LP des originalen primalen LP.
- Wenn primale und duale Lösungen $\boldsymbol{x}$ und $\boldsymbol{y}$ den **gleichen Zielfunktionswert** $\boldsymbol{y}^T\boldsymbol{b} = \boldsymbol{c}^T\boldsymbol{x}$ haben, dann folgt aus Lemma 21, dass beide **optimale Lösungen** für ihre jeweiligen LPs sein müssen.
- Lemma 21 zeigt auch:
  Primales unbeschränkt ⇒ Duales ungültig.
  Dual unbeschränkt ⇒ Primales ungültig.

Lemma 21 kann zu einem der zentralen Resultate der linearen Optimierung erweitert werden:

Theorem 11 (Starke Dualität)**.** *Wenn ein primales LP beschränkt ist und eine optimale Lösung $\boldsymbol{x}^*$ besitzt, dann ist das duale LP beschränkt und hat eine optimale Lösung $\boldsymbol{y}^*$. Die* optimalen Lösungen haben den gleichen Zielfunktionswert

$$(\boldsymbol{y}^*)^T\boldsymbol{b} = \boldsymbol{c}^T\boldsymbol{x}^*.$$

Mögliche Szenarien für primale und duale LPs:

		Dual		
		beschränkt	unbeschränkt	ungültig
Primal	beschränkt	√		
	unbeschränkt			√
	ungültig		√	√

Zur Illustration ein Beispiel, in dem sowohl primales als auch duales LP ungültig sind:

$$\begin{array}{lrcl} \text{Min.} & x_1 + 2x_2 & & \\ \text{s.d.} & x_1 + x_2 & = & 1 \\ & 2x_1 + 2x_2 & = & 3 \end{array} \qquad \begin{array}{lrcl} \text{Max.} & y_1 + 3y_2 & & \\ \text{s.d.} & y_1 + 2y_2 & = & 1 \\ & y_1 + 2y_2 & = & 2 \end{array}$$

3.5 Interior-Point Verfahren

3.5.1 Größe der Darstellung

Für polynomielle Laufzeitschranken müssen wir klären, wie groß (asymptotisch) die Eingabelänge eines LPs ist. Dazu ist insbesondere entscheidend, wie wir die Einträge von $\boldsymbol{A}$, $\boldsymbol{b}$ und $\boldsymbol{c}$ darstellen wollen. Daneben müssen wir untersuchen, wie groß die Darstellungsgröße der Ausgabe, also der berechneten Ecke des Polyeders ist.

Die Lage ist deutlich klarer, wenn die Parameter alle ganzzahlig sind.

- Die Darstellungslänge einer ganzen Zahl $k \in \mathbb{Z}$ sei gegeben durch $size(k) = 1 + \lceil \log_2(|k|+1) \rceil$.
- Für $\boldsymbol{A} \in \mathbb{Z}^{m \times n}$, $\boldsymbol{b} \in \mathbb{Z}^m$ und $\boldsymbol{c} \in \mathbb{Z}^n$ ist die Größe des zugehörigen LPs in Standardform

$$size(\boldsymbol{A}) = \sum_{i=1}^{m} \sum_{j=1}^{n} size(a_{ij})$$

$$size(\boldsymbol{b}) = \sum_{i=1}^{m} size(b_i) \qquad size(\boldsymbol{c}) = \sum_{j=1}^{n} size(c_j)$$

$$size(LP) = size(\boldsymbol{A}) + size(\boldsymbol{b}) + size(\boldsymbol{c})$$

- Durch Muliplikation mit dem kleinsten gemeinsamen Vielfachen aller Zahlen können wir das LP skalieren und ganze Zahlen als Koeffizienten garantieren.

Statt mit der präziseren Größe $size(LP)$ rechnet es sich oft leichter mit einem Parameter L, der wie folgt gegeben ist:

$$L = size(\det{}_{\max}) + size(b_{\max}) + size(c_{\max}) + m + n$$

wobei

$$\det{}_{\max} = \max_{A'} |\det(A')| \qquad (A' \text{ Untermatrix von } A)$$

$$b_{\max} = \max_{i=1,\ldots,m} |b_i| \qquad \text{und} \qquad c_{\max} = \max_{j=1,\ldots,n} |c_j|$$

Behauptung: $L = O(size(LP))$, also polynomielle Laufzeit in L liefert einen effizienten Algorithmus.

Damit haben wir eine Beschreibung der Eingabegröße. Wie groß ist die Darstellung der Ausgabe, d.h. der optimalen Ecke $\boldsymbol{x}^*$?

Sei $\boldsymbol{x}$ eine beliebige Ecke eines LPs in Standardform.

- Die Ecke entspricht einer Basis B mit $\boldsymbol{x}_B \geq \boldsymbol{0}$ und $\boldsymbol{x}_N = \boldsymbol{0}$.
- Das zu B gehörende Gleichungssystem wird durch eine quadratische Untermatrix $\boldsymbol{A}_B$ von $\boldsymbol{A}$ beschrieben.
- Die Lösung von $\boldsymbol{A}_B\boldsymbol{x}_B = \boldsymbol{b}$ folgt mit der Cramerschen Regel:

$$(x_B)_j = \frac{\det(\boldsymbol{a}_B^1, \ldots \boldsymbol{a}_B^{j-1}, \boldsymbol{b}, \boldsymbol{a}_B^{j+1}, \ldots, \boldsymbol{a}_B^m)}{\det(\boldsymbol{A}_B)} \tag{3.9}$$

- Die Ecke können wir also schreiben als

$$\boldsymbol{x} = \left(\frac{p_1}{q}, \frac{p_2}{q}, \ldots, \frac{p_n}{q}\right)^T \tag{3.10}$$

  mit Koeffizienten $p_i, q \in \mathbb{N}$ sowie $0 \leq p_i < 2^L$ und $1 \leq q < 2^L$.
- Die Ecken liegen auf einem n-dimensionalen "Grid" mit Genauigkeit höchstens 2^L.
- Die Ausgabe ist polynomiell in n und L repräsentierbar.

3.5.2 Interior-Point Verfahren

Die Interior-Point Methode verschiebt einen Punkt innerhalb des Lösungspolyeders in Richtung der optimalen Ecke. Wenn der Punkt nahe genug an einer Ecke ist, kann man das Verfahren beenden und die Lösung auf die Ecke "runden". Damit erzielt man polynomielle Laufzeit.

Dieser Ansatz wird ermöglicht, da **suboptimale Ecken nicht beliebig nahe an optimalen Ecken** liegen können, weder als Punkte Raum noch im Wertebereich der Zielfunktion.

Lemma 22. *Seien $\boldsymbol{x} \neq \boldsymbol{x}'$ zwei verschiedene Ecken eines LPs. Wenn $\boldsymbol{c}^T\boldsymbol{x} \neq \boldsymbol{c}^T\boldsymbol{x}'$ dann gilt*

$$|\boldsymbol{c}^T\boldsymbol{x} - \boldsymbol{c}^T\boldsymbol{x}'| > 2^{-2L}$$

Daraus folgt:

- Sei $z = \boldsymbol{c}^T\boldsymbol{x}^*$ der optimale Wert eines LP
- $\boldsymbol{x}$ sei eine Lösung mit $\boldsymbol{c}^T\boldsymbol{x} \leq z + 2^{-2L}$
- Jede Ecke $\boldsymbol{x}'$ mit $\boldsymbol{c}^T\boldsymbol{x}' \leq \boldsymbol{c}^T\boldsymbol{x}$ ist optimal.

Das Interior-Point-Verfahren löst gleichzeitig primales und duales LP:

$$\begin{array}{lrcl} \text{Min.} & \boldsymbol{c}^T\boldsymbol{x} & & \\ \text{s.d.} & \boldsymbol{A}\boldsymbol{x} & = & \boldsymbol{b} \\ & \boldsymbol{x} & \geq & \boldsymbol{0} \end{array} \qquad \begin{array}{lrcl} \text{Max.} & \boldsymbol{y}^T\boldsymbol{b} & & \\ \text{s.d.} & \boldsymbol{y}^T\boldsymbol{A} + \boldsymbol{s}^T & = & \boldsymbol{c}^T \\ & \boldsymbol{s} & \geq & \boldsymbol{0} \end{array}$$

Der Algorithmus verwaltet Lösungen $\bar{\boldsymbol{x}}, \bar{\boldsymbol{s}} > \boldsymbol{0}$, so dass ein zugehöriges $\boldsymbol{y}$ existiert mit $\boldsymbol{y}^T\boldsymbol{A} + \bar{\boldsymbol{s}}^T\boldsymbol{I} = \boldsymbol{c}^T$, und keine Komponente von $\bar{\boldsymbol{x}}, \bar{\boldsymbol{s}}$ zu klein wird (wir also im Inneren des Polyeders bleiben).

Um die Verschiebung am Punkt $\bar{\boldsymbol{x}}$ auszurichten, nutzt der Algorithmus eine *Skalierung*:

- $\boldsymbol{x}$ wird abgebildet auf $\boldsymbol{x}' = (x_1/\bar{x}_1, \ldots, x_n/\bar{x}_n)^T$
- Die Abbildung kann durch Multiplikation mit einer Diagonalmatrix erreicht werden:

$$\boldsymbol{x}' = \bar{\boldsymbol{X}}^{-1}\boldsymbol{x} \qquad \text{mit } \bar{\boldsymbol{X}} = \begin{pmatrix} \bar{x}_1 & 0 & 0 & \ldots & 0 \\ 0 & \bar{x}_2 & 0 & \ldots & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & 0 & \ldots & \bar{x}_{n-1} & 0 \\ 0 & 0 & \ldots & 0 & \bar{x}_n \end{pmatrix}$$

- Damit wird $\bar{\boldsymbol{x}}$ abgebildet auf $\bar{\boldsymbol{X}}^{-1}\bar{\boldsymbol{x}} = \mathbf{1} = (1, \ldots, 1)^T$.

Durch Skalierung entsprechen die LPs nun

$$\begin{array}{lrcl} \text{Min.} & \boldsymbol{c}^T\bar{\boldsymbol{X}}\boldsymbol{x}' & & \\ \text{s.d.} & \boldsymbol{A}\bar{\boldsymbol{X}}\boldsymbol{x}' & = & \boldsymbol{b} \\ & \boldsymbol{x} & \geq & \mathbf{0} \end{array} \qquad \begin{array}{lrcl} \text{Max.} & \boldsymbol{y}^T\boldsymbol{b} & & \\ \text{s.d.} & \boldsymbol{y}^T\boldsymbol{A}\bar{\boldsymbol{X}} + (\boldsymbol{s}')^T & = & \boldsymbol{c}^T\bar{\boldsymbol{X}} \\ & \boldsymbol{s}' & \geq & \mathbf{0} \end{array}$$

Man kann nachweisen, dass $\boldsymbol{s}' = \bar{\boldsymbol{X}}\boldsymbol{s} = (s_1\bar{x}_1, \ldots, s_n\bar{x}_n)^T$.

Der Algorithmus nutzt die **Dualitätslücke**:

- Dualitätslücke für gültige primale und duale Lösungen der LPs in Standardform: $\boldsymbol{c}^T\boldsymbol{x} - \boldsymbol{y}^T\boldsymbol{b}$
- Für Lösungen des Algorithmus gilt $\boldsymbol{c}^T\bar{\boldsymbol{x}} - \bar{\boldsymbol{y}}^T\boldsymbol{b} = \bar{\boldsymbol{s}}^T\bar{\boldsymbol{x}} = (\bar{\boldsymbol{s}}')^T\bar{\boldsymbol{x}}'$
- Skalierung ändert die Dualitätslücke nicht!
- Wenn die Dualitätslücke klein ist, kann der Algorithmus terminieren, da mit Lemma 22 alle Ecken in der Nähe optimal sein müssen.

Innere-Punkt-Verfahren (sehr High-Level):

- Nutze folgende **Potenzialfunktion**, um den Fortschritt des Algorithmus sicherzustellen:

$$G(\boldsymbol{x}, \boldsymbol{s}) = (n + \sqrt{n}) \ln(\boldsymbol{x}^T\boldsymbol{s}) - \sum_{j=1}^{n} (x_j s_j)$$

- G ist invariant unter Skalierung
- Es ist möglich, Startpunkte $\bar{\boldsymbol{x}} > \mathbf{0}, \bar{\boldsymbol{s}} > \mathbf{0}$ zu finden, so dass $G(\bar{\boldsymbol{x}}, \bar{\boldsymbol{s}}) = O(\sqrt{n}L)$
- Wenn $G(\bar{\boldsymbol{x}}, \bar{\boldsymbol{s}}) < -2\sqrt{n}L$, dann sind wir nah genug am Optimum und können stoppen.
- In jeder Iteration können wir sicherstellen, dass wir G um mindestens $7/120$ verringern. Nur $O(\sqrt{n}L)$ Iterationen notwendig

In jeder Iteration machen wir einen **primalen** oder **dualen** Schritt.
Primaler Schritt:

- Betrachte den Bildraum (also $(\bar{\boldsymbol{x}}, \bar{\boldsymbol{s}}) \to (\mathbf{1}, \bar{\boldsymbol{s}}')$). Im primalen Schritt ändern wir nur $\bar{\boldsymbol{x}}$.
- Betrachte den Gradienten von G am Punkt $(\mathbf{1}, \bar{\boldsymbol{s}}')$:

$$\boldsymbol{g} = \nabla_{\boldsymbol{x}} G(\boldsymbol{x}, \boldsymbol{s})|_{(\mathbf{1}, \bar{\boldsymbol{s}}')} = \frac{n + \sqrt{n}}{\mathbf{1}^T\bar{\boldsymbol{s}}'} \cdot \bar{\boldsymbol{s}}' - \mathbf{1}$$

- Ein Schritt in Richtung $-\boldsymbol{g}$ könnte den zulässigen Raum verlassen.
- Sei $\boldsymbol{d}$ die Projektion von $\boldsymbol{g}$ auf $\{\boldsymbol{x} \mid \boldsymbol{A}\bar{\boldsymbol{X}}\boldsymbol{x} = \mathbf{0}\}$. Dann ist $\boldsymbol{d} = (\boldsymbol{I} - \boldsymbol{A}\bar{\boldsymbol{X}}(\boldsymbol{A}\bar{\boldsymbol{X}}(\boldsymbol{A}\bar{\boldsymbol{X}})^T)^{-1}\boldsymbol{A}\bar{\boldsymbol{X}})\boldsymbol{g}$
- Wir versuchen einen Schritt in Richtung $-\boldsymbol{d}$. Wir erhalten $(\tilde{\boldsymbol{x}}, \tilde{\boldsymbol{s}})$ mit

$$\tilde{\boldsymbol{x}} = \mathbf{1} - \frac{1}{4|\boldsymbol{d}|}\boldsymbol{d} \quad \text{und} \quad \tilde{\boldsymbol{s}} = \bar{\boldsymbol{s}}' \ .$$

- Wenn $|\boldsymbol{d}| = \sqrt{\boldsymbol{d}^T\boldsymbol{d}} \geq 0.4$, dann erzeugt der Schritt den benötigten Fortschritt in G. Danach wird die Lösung $(\tilde{\boldsymbol{x}}, \tilde{\boldsymbol{s}})$ in den Urbildraum zurückskaliert und eine neue Iteration startet.
- Sonst machen wir keinen primalen sondern stattdessen einen dualen Schritt.

Dualer Schritt:

- Wenn $|\boldsymbol{d}| = \sqrt{\boldsymbol{d}^T\boldsymbol{d}} < 0.4$ machen wir einen dualen Schritt, der nur $\boldsymbol{s}'$ verändert. Wir betrachten wieder den Gradienten von G im Bildraum am Punkt $(\mathbf{1}, \bar{\boldsymbol{s}}')$:

$$\boldsymbol{h} = \nabla_{\boldsymbol{s}} G(\boldsymbol{x}, \boldsymbol{s})|_{(\mathbf{1},\bar{\boldsymbol{s}}')} = \frac{n+\sqrt{n}}{\mathbf{1}^T\bar{\boldsymbol{s}}'} \cdot \mathbf{1} - \begin{pmatrix} 1/s'_1 \\ \vdots \\ 1/s'_n \end{pmatrix}$$

- Für den neuen Vektor $\tilde{\boldsymbol{s}}$ müssen wir sicherstellen, dass es ein $\boldsymbol{y}$ gibt mit $\boldsymbol{y}^T\boldsymbol{A}\bar{\boldsymbol{X}} + (\tilde{\boldsymbol{s}})^T = \boldsymbol{c}^T\bar{\boldsymbol{X}}$.
- Dafür kann man $\tilde{\boldsymbol{s}}$ wie folgt wählen:

$$\tilde{\boldsymbol{s}} = \frac{\mathbf{1}^T\bar{\boldsymbol{s}}'}{n+\sqrt{n}}(\boldsymbol{d}+\mathbf{1}) \quad \text{und} \quad \tilde{\boldsymbol{x}} = \mathbf{1}.$$

3.6 Ganzzahlige Lineare Programme (ILP)

ILPs haben lineare Constraints und eine lineare Zielfunktion, aber die Variablen müssen eine Teilmenge der *ganzen* Zahlen sein.

[Pic: Ganzzahliger Lösungsraum im $\mathbb{R}^2$]

Mit ILPs kann man viele diskrete Optimierungs- und Entscheidungsprobleme modellieren, z.B. Vertex Cover und Maximum Matching:

$$\begin{array}{lllll} \text{Min.} & \displaystyle\sum_{v\in V} x_v & & & \\ \text{s.d.} & x_u + x_v & \geq & 1 & \text{für alle } e \in E \\ & x_v & \in & \{0,1\} & \text{für alle } v \in V \end{array} \qquad \begin{array}{lllll} \text{Max.} & \displaystyle\sum_{e\in E} y_e & & & \\ \text{s.d.} & \displaystyle\sum_{\{u,v\}\in E} y_{\{u,v\}} & \leq & 1 & \text{für alle } v \in V \\ & y_e & \in & \{0,1\} & \text{für alle } e \in E \end{array}$$

Die Formulierung von Vertex Cover zeigt, dass das Lösen von ILPs allgemein NP-hart ist und viele andere schwere Probleme (SAT, Clique, Maximum Independent Set, usw.) ebenfalls als ILPs formuliert werden können. Für Maximum Matching kennen wir dagegen schon (kombinatorische) effiziente Algorithmen. Wir untersuchen ein Kriterium, dass eine optimale Lösung von ILPs mit LP-Algorithmen erlaubt.

Stellen wir uns eine **LP-Relaxierung** des ILPs zu einem (normalen) LP vor (z.B. durch Ersetzung von $x_i \in \{0,1\}$ durch $x_i \in [0,1]$). Eine wünschenswerte Eigenschaft ist, dass durch die Relaxierung **keine besseren Optimallösungen** entstehen. Das ist insbesondere dann der Fall, wenn **alle Ecken der LP-Relaxierung ganzzahlig sind**.

[Pic: Ganzzahliger Lösungsraum im $\mathbb{R}^2$, LP-Relaxierung, ganzzahlige Ecken]

Unimodularität:

- Eine ganzzahlige quadratische Matrix $\boldsymbol{A}$ heißt **unimodular** wenn $\det \boldsymbol{A} \in \{-1, 1\}$.
- Eine ganzzahlige Matrix $\boldsymbol{A}$ heißt **total unimodular** wenn jede quadratische *reguläre* Teilmatrix A' unimodular ist.

Theorem 12. *Sei $\boldsymbol{A}$ eine total unimodulare Matrix. Dann gilt:*
1. *Jede Basislösung von $\boldsymbol{Ax} = \boldsymbol{b}$ ist ganzzahlig.*
2. *Jede Ecke eines LPs in kanonischer Form mit Matrix $\boldsymbol{A}$ ist ganzzahlig.*

Beweis: Betrachte eine Basislösung $\boldsymbol{x}_B$ mit Basis B.
- $\boldsymbol{A}_B\boldsymbol{x}_B = \boldsymbol{b}$ für eine quadratische, reguläre Matrix $\boldsymbol{A}_B$.
- Die Lösung $\boldsymbol{x}_B$ ist gegeben durch die Cramersche Regel, siehe (3.9) und (3.10).
- Alle Einträge von $\boldsymbol{x}_B$ haben Nenner $q = \det \boldsymbol{A}_B \in \{-1, 1\}$, sind also ganze Zahlen $p_i \in \mathbb{Z}$.

Für die kanonische Form:
- Sei m die Anzahl der Zeilen von $\boldsymbol{A}$. Für $\boldsymbol{Ax} \geq \boldsymbol{b}$ füge Schlupfvariablen ein.
- Dann erhalten wir die neue Constraintmatrix $(\boldsymbol{A} \mid \boldsymbol{I}_m)$ und Gleichheitsconstraints.
- Wir zeigen: Die neue Constraintmatrix ist auch total unimodular.
- Ecken entsprechen Basislösungen, und damit folgt das Resultat aus dem Beweis oben.
- Sei $\boldsymbol{C}$ eine quadratische, reguläre Teilmatrix. Durch Umformung erhält sie die Form

$$\boldsymbol{C}' = \begin{pmatrix} \boldsymbol{A}' & 0 \\ * & \boldsymbol{I}_k \end{pmatrix}$$

- $\boldsymbol{I}_k$ ist die $k \times k$-Einheitsmatrix. $\boldsymbol{A}'$ ist eine quadratische, reguläre Teilmatrix von $\boldsymbol{A}$
- Es gilt $|\det(C)| = |\det(C')| = |\det(A')| = 1$ □

Theorem 13. *Eine Matrix $\boldsymbol{A}$ mit Einträgen aus $\{-1, 0, 1\}$ ist total unimodular, wenn*
- *in jeder Spalte j höchstens zwei Einträge (in Zeilen $i_{j,1}$ und $i_{j,2}$) aus $\{-1, 1\}$ sind und*
- *die Zeilen sich in Mengen I_1 und I_2 aufteilen lassen, so dass für jede Spalte j gilt: Wenn $a_{i_{j,1},j} \neq a_{i_{j,2},j}$ dann sind i_{j_1} und i_{j_2} in der gleichen Menge I_i; sonst sind i_{j_1} und i_{j_2} in unterschiedlichen Mengen.*

Das Theorem kann man relativ direkt per Induktion beweisen.

Die **Inzidenzmatrix** von Graphen hat oft die im Theorem geforderte Form.
- Inzidenzmatrix $\boldsymbol{A}$ hat eine Zeile für jeden Knoten und eine Spalte für jede Kante
- Gerichtet: Für Spalte $e = (u, v)$ ist Eintrag $a_{u,e} = -1$ und $a_{v,e} = 1$ und sonst 0
- Ungerichtet: Für Spalte $e = \{u, v\}$ sind Einträge $a_{v,e} = a_{u,e} = 1$ und sonst 0.
- Gerichteter Graph $G = (V, E)$: Alle Zeilen in eine Menge I_1.
- *Bipartiter* ungerichteter Graph $G = (A \cup B, E)$:
  Alle Zeilen von A gehören zu I_1, alle von B zu I_2.

Korollar 3. *Wenn die Constraintmatrix eines LPs in kanonischer Form die*
- *Inzidenzmatrix eines gerichteten Graphen oder die*
- *Inzidenzmatrix eines bipartiten ungerichteten Graphen*

ist, dann hat das LP nur gannzahlige Ecken.

Als Folge daraus können viele ganzzahlige Optimerungsprobleme effizient durch LPs gelöst werden:
- maximale Flüsse
- kürzeste Wege
- bipartites Maximum Matching (nur bipartit, siehe K_3)
- bipartites Vertex Cover (nur bipartit, siehe K_3)

Kapitel 4

Approximationsalgorithmen

4.1 Makespan Scheduling und grundlegende Definitionen

4.1.1 Approximationsfaktoren

Ein Beispiel zum Anfang: Makespan Scheduling auf identischen Maschinen

- m identische Maschinen, n Tasks
- Task $i \in [n]$ hat Bearbeitungszeit $p_i > 0$
- Weise jeden Task i (komplett) einer Maschine $j \in [m]$ zu.
- Last der Maschine j ist $\ell_j = \sum_{i \text{ on } j} p_i$
- Maximale Last $\max_{j \in [m]} \ell_j$ wird **Makespan** der Zuweisung genannt

Ziel: Weise alle Tasks zu und minimiere den Makespan

[Pic: Beispielinstanz, Zuweisung, Makespan]

Alle Tasks und Bearbeitungszeiten sind bekannt (in dem Sinne ist es ein *Offline*-Problem). Das Problem ist NP-hart (warum?), also gibt es Instanzen, die (sehr wahrscheinlich) nicht in polynomieller Zeit optimal gelöst werden können. Gibt es trotzdem **gute Algorithmen**, obwohl sie das Problem nicht immer optimal lösen? Wie sollte man die Güte solch suboptimaler Algorithmen beschreiben?

Betrachte den Algorithmus **ListScheduling**:
Wähle eine Permutation π der Tasks. Für jedes $i = 1, \ldots, n$: Weise Task $\pi(i)$ einer Maschine j zu, die aktuell die kleinste Last hat.

ListScheduling berechnet nicht immer eine optimale Zuweisung. Der Makespan ist aber auch nicht beliebig schlecht:

Theorem 14. *Sei I eine Instanz von Makespan Scheduling, $OPT(I)$ der kleinste Makespan einer Zuweisung und $LIST(I)$ der Makespan einer Zuweisung des ListScheduling Algorithmus. Es gilt*

$$LIST(I) \leq \left(2 - \frac{1}{m}\right) \cdot OPT(I) \ .$$

Mit unserer Terminologie unten sagen wir, ListScheduling *erzielt eine* $(2 - 1/m)$*-Approximation* bzw. *der Approximationsfaktor von ListScheduling ist höchstens* $2 - 1/m$. Für jede Instanz I ist der Makespan der von ListScheduling berechneten Zuweisung höchstens *doppelt so groß* ist wie dier optimale Makespan der Instanz.

Beweis von Theorem 14: Wir zeigen **untere Schranken auf** $OPT(I)$ mit Bezug zur Ausgabe des Algorithmus.

- Sei Task k der letzte Task auf einer Machine, die am Ende Makespan-Last $LIST(I)$ hat.
- k wird einer Maschine mit kleinster Last zugewiesen.
- Also $\ell_j \geq LIST(I) - p_k$ für jede Maschine $j \in [m]$
- Es gilt $OPT(I) \geq LIST(I) - p_k$, denn bis zu dieser Zeit sind alle Maschinen ausgelastet.
- Also: $OPT(I) \geq p_k$, denn wir müssen Task k komplett auf die Maschine zuweisen

[Pic: Schedule, p_k, untere Schranken]

Die zwei unteren Schranken an $OPT(I)$ beziehen sich auf $LIST(I)$. Zusammensetzen ergibt

$$LIST(I) \leq OPT(I) + p_k \leq 2 \cdot OPT(I) \ . \tag{4.1}$$

Schauen wir etwas genauer hin:

- $OPT(I)$ muss mindestens die durchschnittliche Last aller Maschinen sein.
- Zum Zeitpunkt $LIST(I) - p_k$ sind alle Maschinen belegt, daher ist die Gesamtlast mindestens

$$\sum_{i=1}^{n} p_i \geq m(LIST(I) - p_k) + p_k \ .$$

- Dies ergibt

$$OPT(I) \quad \geq \frac{1}{m} \sum_{i=1}^{n} p_i \geq \frac{1}{m} \cdot ((m \cdot (LIST(I) - p_k) + p_k) = \quad LIST(I) - \frac{m-1}{m} \cdot p_k$$

Analog zu (4.1) ergibt sich

$$LIST(I) \leq OPT(I) + \frac{m-1}{m} \cdot p_k \leq \left(1 + \frac{m-1}{m}\right) \cdot OPT(I) = \left(2 - \frac{1}{m}\right) \cdot OPT(I) \ . \quad \square$$

Ist das die beste Garantie? Ist der Algorithmus evtl. deutlich besser? Gibt es eine Instanz, bei der Algorithmus eine Zuweisung errechnet mit nahezu dem doppelten optimalen Makespan?

Lemma 23. *Es gibt eine Instanz I und eine Permutation der Tasks, so dass*

$$LIST(I) = \left(2 - \frac{1}{m}\right) \cdot OPT(I) \ .$$

Beweisidee: Die Instanz ergibt sich wie folgt: Es gibt einen Task mit $p_1 = m$ und $m(m-1)$ Tasks mit $p_i = 1$. Wenn ListScheduling erst die kleinen Tasks zuweist, dann erzeugt der große Task 1 am Ende den Makespan von $(m-1) + m$. Es gibt aber eine Zuweisung mit Makespan m. $\square$

Ähnlich zum letzten Kapitel betrachten wir (hier: diskrete) **Optimierungsprobleme**, in denen wir eine Lösung mit minimalem oder maximalem Zielfunktionswert finden wollen. Einige Beispiele in ungerichteten Graphen $G = (V, E)$:

Vertex Cover:

- Knotenteilmenge $C \subseteq V$ ist ein *Vertex Cover* wenn für jede Kante $e \in E$ mindestens ein inzidenter Knoten in C ist
- **Ziel:** Finde ein Vertex-Cover C mit minimaler Größe $|C|$.

Clique:
- Knotenteilmenge $C \subseteq V$ ist eine *Clique* wenn für jedes Paar $u, v \in C$ die Kante $\{u, v\} \in E$
- **Ziel:** Finde eine Clique C mit maximaler Größe $|C|$.

Independent Set:
- Knotenteilmenge $S \subseteq V$ ist ein *Independent Set* wenn für jedes Paar $u, v \in S$ die Kante $\{u, v\} \notin E$
- **Ziel:** Finde ein Independent Set S mit maximaler Größe $|S|$.

Wir stellen uns einen Algorithmus für, sagen wir, das Clique-Problem vor, der immer eine optimale Lösung berechnet. Der Algorithmus kann auch für das *Entscheidungsproblem* genutzt werden (d.h. entscheide, ob die größte Clique mindestens k Knoten enthält, für gegebenes k). Dieses Entscheidungsproblem ist allerdings NP-vollständig. Daher können wir nicht erwarten, dass der Algorithmus gleichzeitig
1. eine optimale Lösung berechnet,
2. in polynomieller Zeit läuft, und
3. dies für jede Instanz des Problems tut.

Wir suchen einen *guten (suboptimalen) Algorithmus* für ein Optimierungsproblem, der *immer in Polynomialzeit läuft*, d.h. wir relaxieren Bedingung (1). Es gibt auch viel Forschung zur Relaxierung von Bedingung (2) (z.B. exakte Exponenzialzeit-Algorithmen) oder (3) (z.B. Algorithmen für Spezialfälle).

Wir messen den **Worst-Case Faktor**, um den die Kosten der Lösung des Algorithmus die minimalen Kosten einer optimalen Lösung übersteigen. Analog definieren wir den Faktor für Maximierungsprobleme.

Definition 2. Betrachte eine Instanz I eines Minimierungsproblems. Sei $OPT(I)$ der Zielfunktionswert der optimalen Lösung und $\alpha \geq 1$ eine Zahl.

- Eine Lösung ist **α-approximativ** wenn sie Wert höchstens $\alpha \cdot OPT(I)$ hat.
- Für einen Algorithmus ALG sei $ALG(I)$ der Zielfunktionswert der berechneten Lösung. ALG **berechnet eine α-Approximation** wenn für jede Instanz I des Problems
$$ALG(I) \leq \alpha \cdot OPT(I) \qquad \text{oder} \qquad \frac{ALG(I)}{OPT(I)} \leq \alpha.$$
- Analog ist eine Lösung für ein Maximierungsproblem α-approximativ wenn sie einen Wert mindestens $OPT(I)/\alpha$ hat. ALG erzielt eine α-Approximation wenn für jede Instanz I des Problems
$$ALG(I) \geq OPT(I)/\alpha \qquad \text{oder} \qquad \frac{OPT(I)}{ALG(I)} \leq \alpha.$$

Wir sagen der **Approximationsfaktor** des Algorithmus ist **(höchstens)** α.

Schranken an den Approximationsfaktor sollen für *jede Instanz I* des Problems gelten. Daher muss eine obere Schranke (die zeigt, dass der Algorithmus immer "gut" ist) *in jeder einzelnen Instanz des Problems* gelten (siehe z.B. Theorem 14). Für eine untere Schranke (die zeigt, dass der Algorithmus manchmal "schlecht" ist) reicht es aus, *eine einzelne Instanz I anzugeben*, bei der der Faktor $\frac{ALG(I)}{OPT(I)}$ (oder $\frac{OPT(I)}{ALG(I)}$ für Maximierung) die Schranke erreicht (z.B. Lemma 23).

Algorithm 11: BruteForceList

1 Berechne Menge T der $\min(n, m \cdot \lceil 1/\varepsilon \rceil)$ Tasks mit längster Bearbeitungszeit, für einen gegebenen Parameter $\varepsilon > 0$.
2 Berechne eine optimale Zuweisung der Tasks in T.
3 Rufe ListScheduling auf, um die Zuweisung für die restlichen Tasks zu berechnen.
4 **return** Zuweisung der Tasks an die Maschinen

Können wir den Faktor $2 - 1/m$ noch verbessern? Wir betrachten eine schlauere Permutation, *longest processing time first (LPT).*

Theorem 15. *ListScheduling mit fallender Bearbeitungszeit (LPT) hat einen Approximationsfaktor von* 4/3*.*

Beweis: Durch Widerspruch. Sei $LPT(I)$ der berechnete Makespan. Wir nehmen an, dass es eine Instanz I gibt mit $LPT(I) > 4/3 \cdot OPT(I)$. Sei I eine solche Instanz mit **den wenigsten Tasks**.

- Der Einfachheit halber: Tasks nummeriert so dass $p_1 \geq p_2 \geq \ldots \geq p_n$
- Betrachte wieder Task k, der (als erster Task) den Makespan erreicht
- Da I am wenigsten Tasks hat, ist $k = n$.
- Task n wurde auf eine Maschine mit kleinster Last gelegt. Es gilt wie oben:

$$\frac{1}{m} \sum_{i=1}^{n-1} p_i \leq OPT(I)$$

- Also muss $p_n > 1/3 \cdot OPT(I)$ gelten, damit wir $4/3 \cdot OPT(I)$ erreichen.
- Damit gilt für alle $p_1 \geq \ldots \geq p_n > 1/3 \cdot OPT(I)$.
- Es gibt nur maximal 2 Tasks pro Maschine, und $n \leq 2m$ Tasks insgesamt
- Unter der Bedingung ist folgende Plazierung optimal: Tasks $i = 1, \ldots, n$ jeweils auf Maschine i, Tasks $i = n+1, \ldots, 2n$ jeweils auf Maschine $2n + 1 - i$.
- Das wird von LPT berechnet! Also $LPT(I) = OPT(I) \not> 4/3 \cdot OPT(I)$ - Widerspruch □

4.1.2 PTAS und FPTAS

Geht es noch besser als 4/3? Ja, aber dafür betrachten wir zuerst nur Instanzen mit **konstanter Anzahl** m **von Maschinen**. Sei m klein, z.B., $m = 5$.

- Im ListScheduling könnte der Task k, der den Makespan erreicht, ein großer Task sein (oben haben wir gesehen, er könnte so groß sein wie $OPT(I)$).
- Wir teilen die Tasks in **große** und **kleine** Tasks. Sei $B > 0$ eine Konstante.
- Die $B \cdot m$ größten Tasks sind große Tasks, der Rest ist klein.
- Algorithmus: Berechne ein optimales Assignment (per brute-force) für die großen Tasks. Danach nutze ListScheduling für die restlichen Tasks.

Analyse des Approximationsfaktors:

- Fall 1: Maschine j hat *nur große Tasks* und ℓ_j ist der Makespan.

⇒ Schedule war optimal für große Tasks. Hat noch den gleichen Makespan am Ende
⇒ Schedule ist optimal für alle Tasks, Approximationsfaktor 1!

- Fall 2: Nur Maschinen j mit *kleinen Tasks* haben ℓ_j gleich Makespan.

⇒ Nutze Analyse von ListScheduling aus Theorem 14 oben. Da $p_k < OPT(I)/B$ wird der Faktor höchstens $LIST(I) \leq OPT(I) + p_k < (1 + 1/B)OPT(I)$.

[Pic: Approximationsfaktor, großer Task erzielt Makespan]

Je größer B desto besser ist die Approximation - beliebig gute Faktoren sind möglich!
Aber **was ist mit der Laufzeit**?

- Optimale Zuweisung der großen Tasks braucht Zeit $O(m^{Bm})$.
- Das muss polynomiell in n sein! Also sollte B eine Konstante unabhängig von n sein.
- Genauer gesagt, $B = c \cdot \log_m n$ für konstantes c wäre ok, da dann $O(m^{Bm}) = O(n^{cm})$ und c und m sind beides Konstanten.

Wir definieren formal ein **polynomielles Approximationsschema (PTAS)**.

Definition 3. *Ein PTAS für ein Optimierungsproblem ist eine Familie A_ε von Algorithmen wie folgt. Für jedes $\varepsilon > 0$ erzielt Algorithmus A_ε einen Approximationsfaktor von $(1+\varepsilon)$. Die Laufzeit von A_ε ist polynomiell beschränkt in der Eingabegröße* (aber nicht unbedingt in $1/\varepsilon$).

Der Algorithmus **BruteForceList** (Algorithm 11) beschreibt nochmal die Vorgehensweise. Er nutzt den Parameter $\varepsilon > 0$, und mit $B = \lceil 1/\varepsilon \rceil \leq 1/\varepsilon + 1$ ergibt sich unsere Analyse. Der Approximationsfaktor wird höchstens $1 + 1/B \leq 1 + \varepsilon$. Die Laufzeit ist beschränkt durch $O(nm/\varepsilon + m^{m/\varepsilon+m})$.

Theorem 16. *BrueForceList ist ein PTAS für Makespan Scheduling mit konstanter Anzahl Maschinen.*

Bemerkungen

- Laufzeit ist riesig! Sogar für kleine Werte, z.B. $m = 10$ und $\varepsilon = 0.01$, wird sie astronomisch und total unpraktikabel! Also was soll das alles?
- PTAS'e dienen oft als Orientierung für die Komplexität von Problemen – gibt es etwas Grundlegendes in der Struktur, das beliebig gute Approximationen in polynomieller Zeit verhindert?
- Unsere Laufzeit ist exponenziell in $1/\varepsilon$ (ok) und m (weniger ok). Es ist *kein PTAS* für das normale Makespan Scheduling ohne die Annahme einer konstanten Anzahl m von Maschinen.

Können wir noch mehr erreichen? Der Faktor ist schon bestmöglich, aber die Laufzeit kann verbessert werden. Ein **volles** polynomielles Approximationsschema **(FPTAS)** erlaubt eine bessere Abhängigkeit von $1/\varepsilon$.

Definition 4. *Ein FPTAS für ein Optimierungsproblem ist eine Familie A_ε von Algorithmen wie folgt. Für jedes $\varepsilon > 0$ erzielt Algorithmus A_ε einen Approximationsfaktor von $(1+\varepsilon)$. Die Laufzeit von A_ε ist polynomiell beschränkt in der Eingabegröße* und in $1/\varepsilon$.

Können wir das erreichen?

- Das PTAS für Makespan Scheduling mit konstanter Anzahl Maschinen ist kein FPTAS.
- Es exisitert tatsächlich ein FPTAS für konstant viele Maschinen.
- Es gibt auch ein PTAS für "reines" Makespan Scheduling mit beliebig vielen Maschinen *aber kein FPTAS*!

Wieso erlauben gewisse Probleme kein FPTAS? Betrachte das Clique-Problem.

- Wenn Clique ein FPTAS hätte, dann berechnet es für jedes $\varepsilon > 0$ und jeden Graphen mit $n = |V|$ Knoten eine $(1+\varepsilon)$-Approximation in Zeit polynomiell in n und $1/\varepsilon$.
- Wir setzen $\varepsilon = 1/n$, dann ist die Laufzeit polynomiell in n und $1/\varepsilon = n$.
- Sei $OPT(I)$ die Größe der größten Clique im Graphen. Das FPTAS mit $\varepsilon = 1/n$ errechnet eine Clique der Größe

$$FPTAS(I) \quad \geq \quad \frac{OPT(I)}{1+1/n} \quad = \quad \frac{n}{n+1} \cdot OPT(I)$$

Algorithm 12: Matching Heuristik

```
1 M ← ∅
2 Sei V[M] die Menge der Knoten inzident zu Kanten in M
3 while es gibt e ∈ E mit e ∩ V[M] = ∅ do M ← M ∪ {e}
4 return V[M]
```

$$\begin{aligned} &= OPT(I) - \frac{OPT(I)}{n+1} \\ &> OPT(I) - 1 \end{aligned}$$

- Cliquegrößen sind ganze Zahlen, also $FPTAS(I) = OPT(I)$.
- Das FPTAS mit $\varepsilon \leq 1/n$ errechnet eine optimale Lösung in polynomieller Zeit!

[Pic: Granularität, Approximationsgüte]

Das Problem ist die **Granularität der Lösungswerte**. Es gibt nur $n+1$ ganzzahlige Werte für die Größe der Clique. Die Approximationsgarantie zwingt die Lösung so nahe an den optimalen Wert, dass nur noch ein Wert übrig bleibt. Allgemein gilt:

Theorem 17. *Betrachte ein* NP*-hartes Optimierungsproblem, bei dem für jede Instanz I*
1. *die Zielfunktion nur Werte aus* $\mathbb{N}$ *annimmt und*
2. *es ein Polynom q gibt, so dass jede Lösung nur Wert höchstens* $q(|I|)$ *hat.*

Wenn $\mathsf{P} \neq \mathsf{NP}$*, dann gibt es kein FPTAS für das Problem.*

Der Beweis ist analog zu unseren Argumenten oben mit $\varepsilon = 1/q(|I|)$.

Makespan Scheduling mit konstanter Anzahl Maschinen erlaubt ein FTPAS. Es fällt nicht unter die Klasse von Problemen im obigen Theorem:

- Seien alle Bearbeitungszeiten ganze Zahlen aus $\mathbb{N}$. Dann ist die Eingabegröße in $O\left(\sum_{i\in[n]} \log p_i\right)$.
- Der Makespan liegt im Interval $\max_{i\in[n]} p_i, \ldots, \sum_{i\in[n]} p_i$. Das Intervall kann Zahlen enthalten, die **exponentiell in der Eingabegröße** sind. Theorem 17 greift hier nicht!

4.2 Greedy Algorithmen

4.2.1 Vertex Cover, Clique, Independent Set

Theorem 17 zeigt, dass viele Probleme kein FPTAS haben (wenn $\mathsf{P} \neq \mathsf{NP}$). Vertex Cover, Clique und IndependentSet sind alles Probleme dieser Art. Für einige dieser Probleme gibt es zumindest konstante Faktoren, bei denen der Faktor nicht von der Eingabegröße abhängt. Dies ist das beste Ergebnis wenn kein (F)PTAS erreicht werden kann.

Betrachte die **Matching-Heuristik** (Algorithm 12) für Vertex Cover.

Theorem 18. *Die Matching-Heuristik berechnet eine* 2*-Approximation für Vertex Cover in polynomieller Zeit.*

Beweis: Der Algorithmus berechnet zuerst ein **nicht-erweiterbares Matching** M.

- Keine zwei Matching-Kanten $e, e' \in M$ haben einen gemeinsamen Endknoten.

- Anfangs wahr, wird induktiv in der While-Schleife garantiert
- Sei C^* ein optimales Vertex Cover. Jede Matchingkante $e \in M$ erzwingt (mindestens) einen **eigenen Knoten** in C^*. Also gilt $|C^*| \geq |M|$.
- Die Ausgabe $C = V[M]$ des Algorithmus hat Größe $|C| = 2 \cdot |M|$. Thus,

$$|C| = 2 \cdot |M| \leq 2 \cdot |C^*|.$$

Aber warum ist C ein **gültiges Vertex Cover**?

- Beim Ende der While-Schleife gibt es keine weitere überschneidungsfreie Kante für M.
- Jede Kante $e' \in E \setminus M$ teilt einen Endknoten mit irgendeiner Kante $e \in M$
- Dieser Endknoten ist in $C \Rightarrow C$ überdeckt alle Kanten. □

[Pic: Matchingkanten, separate Knoten in C^*]

Der Algorithmus zeigt insbesondere, dass sich die Größen eines optimalen Vertex Cover und eines optimalen Matchings nur um einen Faktor 2 unterscheiden.

- Matching und Vertex Cover sind duale Probleme im Sinne ihrer (I)LP-Formulierung
- Fraktionale LP-Relaxierungen erfüllen starke Dualität, also sind die Größen von optimalem *fraktionalen* Vertex Cover und optimalem *fraktionalen* Matching gleich.
- Für ILPs gilt auch schwache (also $\boldsymbol{c}^T\boldsymbol{x}^* \geq (\boldsymbol{y}^*)^T\boldsymbol{b}$) aber nicht unbedingt starke Dualität
- Der Algorithmus zeigt, dass $\boldsymbol{c}^T\boldsymbol{x}^* \leq 2 \cdot (\boldsymbol{y}^*)^T\boldsymbol{b}$.
- Der Faktor 2 für diesen Unterschied ist optimal.

Können wir Approximationsfaktoren für Vertex Cover kleiner als 2 erzielen?
Untere Schranken (ohne Beweise):

- $10\sqrt{5} - 21 \approx 1.36$ unter Standardannahmen ($\mathsf{P} \neq \mathsf{NP}$)
- $2 - \varepsilon$, für jede Konstante ε, mit stärkeren Annahmen ("unique games conjecture")
- Bessere Ergebnisse für spezielle Graphklassen (z.B. bipartite Graphen, total unimodular!)

Im Gegensatz dazu gibt es Probleme, die keine Approximation mit konstantem Faktor in polynomieller Zeit erlauben, z.B. das Clique Problem.

Theorem 19. *wenn* $\mathsf{P} \neq \mathsf{NP}$*, dann existert kein effizienter Algorithmus für das Clique Problem mit Approximationsfaktor* $n^{1-\delta}$*, für jede Konstante* $\delta > 0$*.*

Wir beobachten kurz, dass es einen trivialen Algorithmus **mit Faktor** n^1 gibt: Wähle einen einzelnen Knoten, er ist eine Clique der Größe $|C| = 1$. Die optimale Lösung hat Größe $|C^*| \leq n$. Daher: $|C| \geq |C^*|/n$. Eine Clique im Graphen G ist ein Independent Set im Komplementgraphen mit $\tilde{G} = (V, (V \times V) \setminus E)$. Daher gilt Theorem 19 genauso für das Independent Set Problem.

4.2.2 TSP und Δ-TSP

Travelling-Salesman-Problem (TSP):

- Eine Menge V mit n Städten
- Distanzkosten $d(u, v) \geq 0$ für jedes Paar von Städten $u, v \in V$
- **TSP-Tour**: Ein Kreis, der *jede Stadt genau einmal besucht*
- **Ziel:** Finde eine TSP-Tour, die die Summe der Distanzkosten minimiert.

- TSP ist eng verwandt zum **Hamilton-Kreis Problem**: Gegeben einen ungerichteten Graphen $G = (V, E)$, existiert ein einfacher Kreis durch alle Knoten in G?
- Das Hamilton-Kreis Problem ist NP-schwer.

Algorithm 13: MST-Tour for Δ-TSP

1 Interpretiere (v, d) als ungerichteten vollständigen Graphen K_n mit Kantengewichten d
2 $T \leftarrow$ minimaler Spannbaum in (K_n, d)
3 Dupliziere alle Kanten von T und erhalte $\hat{T}$
4 $C_{\hat{T}} \leftarrow$ Eulerkreis von $\hat{T}$
5 Kürze die Kanten von $C_{\hat{T}}$ ab und erhalte TSP tour C
6 **return** C

Theorem 20. *Für jede polynomiell berechenbare Funktion $\alpha(n)$ gibt es keinen effizienten Algorithmus mit Approximationsfaktor $\alpha(n)$ wenn $\mathsf{P} \neq \mathsf{NP}$.*

Beweis:

- Reduktion mit Hamilton-Kreis: Gegeben einen Graphen G konstruiere eine Instanz für TSP wie folgt. Knoten sind Städte. Für jedes Paar $i, j \in V$ setze $d(u, v) = 1$ wenn $\{u, v\} \in E$ und $d(u, v) = n \cdot \alpha(n)$ sonst.
- Die Reduktion kann in polynomieller Zeit berechnet werden.
- Dann gilt: Optimale TSP-Tour hat Distanzkosten $n \iff G$ hat Hamiltonkreis
- Also ist TSP (in der Entscheidungsvariante NP-schwer und damit) NP-hart.
- Sei ALG ein Algorithmus für TSP mit Approximationsfaktor $\alpha(n)$.
- Lasse ALG auf den TSP-Instanzen laufen, die sich in der Reduktion ergeben. ALG erzielt Kosten höchstens $\alpha(n) \cdot n$ auf Instanzen mit Hamilton-Kreis und Kosten mindestens $(n-1) \cdot 1 + n \cdot \alpha(n)$ sonst.
- ALG kann man nutzen, um Hamilton-Kreis zu entscheiden. ALG kann nicht effizient sein (wenn $\mathsf{P} \neq \mathsf{NP}$) □

Wir beschränken uns im Folgenden auf **Δ-TSP**. Dabei ist (V, d) ein metrischer Raum. Er erfüllt die folgenden Eigenschaften, für jede $u, v, w \in V$:

1. Positive Definitheit: $d(u, v) \geq 0 \quad$ und $\quad d(u, v) = 0 \iff u = v$,
2. Symmetrie: $d(u, v) = d(v, u)$
3. Dreiecksungleichung: $d(u, w) \leq d(u, v) + d(v, w)$.

[Pic: Dreiecksungleichung]

Korollar 4. *Δ-TSP ist NP-hart.*

Beweisidee: Nutze die Reduktion aus Theorem 20 mit Distanzen 1 und 2 anstatt 1 und $n\alpha(n)$.

Unsere Algorithmen nutzen *Eulerkreise* und Abkürzungen, um eine TSP-Tour zu erstellen. Ein Eulerkreis eines Graphen G ist eine Traversierung aller Kanten im Graphen, d.h. ein Pfad der mit dem gleichen Knoten startet und endet und durch jede Kante genau einmal durchläuft. Es gilt folgende Eigenschaft:

Theorem 21. *Ein ungerichteter verbundener Graph G hat einen Eulerkreis $\iff$ Jeder Knoten in G hat geraden Grad.*

[Pic: Beispiel Eulerkreis]

Der **MST-Tour-Algorithmus** (Algorithmus 13) nutzt einen minimalen Spannbaum T im Graphen K_n mit Distanzen als Kantengewichten.

- Symmetrische Distanzen, daher K_n als *ungerichteter* Graph.
- Duplizieren aller Kanten ergibt einen Graphen $\hat{T}$ mit geradem Grad für jeden Knoten.
- Der Eulerkreis $C_{\hat{T}}$ von $\hat{T}$ kann mit einem einfachen Greedy-Ansatz berechnet werden.
- Wenn eine Kante (u, v) in $C_{\hat{T}}$ zu einem Knoten v führt, der im Kreis schon besucht wurde, dann kürzen wir diese und die nächste Kante (v, w) im Kreis zusammen zu einer Kante (u, w). Die Gesamtdistanz der Tour kann nur sinken, da $d(u, w) \leq d(u, v) + d(v, w)$ aufgrund der Dreiecksungleichung.
- Wenn wir dieses Argument wiederholt anwenden, erhalten wir eine Tour C, die jeden *Knoten* einmal besucht (anstatt jeder Kante). Die Gesamtdistanz ist höchstens die des Eulerkreises.

[Pic: Beispielablauf des Algorithmus, Abkürzen von Kanten]

Theorem 22. *Algorithm 13 hat einen Approximationsfaktor von 2 for Δ-TSP.*