Effiziente Algorithmen

Martin Hoefer

Sommer 2026

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

Sei f ein s-t-Fluss.
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)

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

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 Abbildung 1.1.

Beobachtung: f Fluss für G, f' Fluss für G_f $\Rightarrow$ $f+f'$ 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

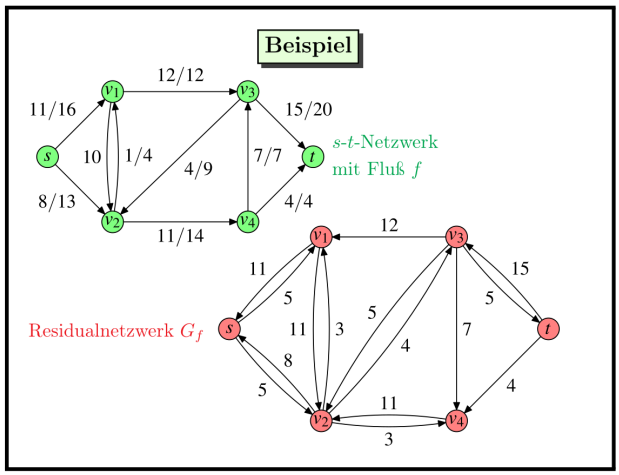

Abbildung 1.1: Beispiel für die Konstruktion eines Restnetzwerks G_f

Algorithmus 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
```

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 alle $i = 1, \ldots, \ell$, sowie $f_P(u, u') = 0$ mit $u, u' \in V \backslash V(P)$.
Ein Beispiel für einen augmentierenden Pfad ist in Abbildung 1.1 der Pfad $P = (s, v_2, v_3, t)$ mit Flussänderung $|f_P| = 4$.

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 $1/h$.

Algorithmus 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
```

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 2 (Max-Flow-Min-Cut)**.** *Die folgenden Aussagen sind gleichbedeutend:*

1. *f ist ein maximaler Fluss.*
2. *f erzeugt keinen augmentierenden Pfad in G_f.*
3. *Es gibt s-t-Schnitt (S,T) und Fluss f mit $c(S,T) = |f| = f(S,T)$.*

Beweis:
3. $\Rightarrow$ 1.: klar, denn $|f| \leq c(S,T)$ für alle s-t-Schnitte (S,T)
1. $\Rightarrow$ 2.: klar, mit Ford-Fulkerson Algorithmus
2. $\Rightarrow$ 3.: Kein augmentierender Pfad in G_f. Also s nicht zu t verbunden in G_f. Betrachte nun G. Für jeden s-t-Pfad in G, sei $e = (u,v)$ die erste Kante des Pfades, die nicht in G_f ist, also für die gilt $c_f(u,v) = 0 = c(u,v) - f(u,v)$. Dann sei

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

die Menge der von s aus erreichbaren Knoten 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 3. *Der Ford-Fulkerson Algorithmus mit Skalierung (Algorithm 2) 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. Wir bezeichnen eine Iteration der äußeren Schleife als **Phase** (also die Menge der Augmentierungen mit gleichem Wert für K).

Betrachte einen minimalen Schnitt und seine Restkapazität.

- Zu Beginn der ersten Phase ist die Restkapazität höchstens $K \cdot m$.
- Sie sinkt durchgängig um den Wert des augmentierenden Pfades.
- Am Ende einer Phase ist sie also höchstens $K \cdot m$.
- Zu Beginn der nächsten Phase (mit halbiertem K) also höchstens $2K \cdot m$.

In jeder Phase also nur höchstens $2m$ Augmentierungen.

In einer Phase braucht jede Augmentierung nur Zeit $O(m)$, denn 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 4. *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 es 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 gilt

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

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 $(u, v) \in E[G_{i+1}]$ muss durch den Pfad entstanden sein. 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 $\delta_{i+1}(s, v) \geq \delta_i(s, v) + 2$, ein Widerspruch. □

[Pic: Anschauung Fall 1 und Fall 2]

Theorem 5. *Der Edmonds-Karp Algorithmus berechnet einen maximalen Fluss in Zeit* $O(n \cdot m^2)$.

Beweis: Mit BFS (Breitensuche) 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 $f_{i+1}(u, v) = c(u, v)$ gilt.
- Der augmentierende Pfad hat mindenstens eine kritische Kante. Diese Kante wird in Runde i entfernt. Kann sie nochmal zurückkommen?
- Sei $j + 1 \geq i + 1$ die Runde, in der (u, v) wieder erscheint. Dann ist $(v, u) \in E[G_j]$ 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 Theorem 4.

$\Rightarrow$ $\delta_j(s, u) \geq \delta_i(s, u) + 2$, insbesondere ist $j > i$ (und nicht nur $j \geq i$)

Da die Distanz $1 \leq \delta_j(s, u) \leq n-1$ ist, kann jede Kante nur höchstens n-Mal kritisch werden (eigentlich sogar nur höchstens $\lceil \frac{n-1}{2} \rceil$ Mal). Also gibt es 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".

$\Delta_f(v) = \sum_{x \in V} f(x, v)$ 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 $v \in V \setminus \{s, t\}$. Dann gilt für den Wert des Flusses $|f| = \Delta_f(t) = -\Delta_f(s)$.

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

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 s.

Eine **Höhenfunktion** $h : V \to \mathbb{N}_0$ weist jedem Knoten $v \in V$ eine **Höhe** $h(v)$ zu. Die Höhe 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).

Algorithmus 3: PreflowPush

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

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

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

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

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

Mit der Abstiegsbedingung geht keine Kante in G_f zu steil nach unten. Bei jeder Kante (v,w) im Restnetzwerk geht es von v zu w immer nur mäßig bergab (höchstens um 1), aber beliebig steil bergauf.

Lemma 6. *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 7. *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 8. *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 $v \in V \setminus \{s,t\}$). 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 es 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 9. *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 10. *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 11. *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 (also relabeled werden), 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.
Wegen Theorem 10 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 $2n$ saturierende Pushs für jedes $(v, w) \in E$ und somit höchstens $2nm$ saturierende Push Operationen insgesamt. □

Lemma 12. *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 $h(w) \leq 2n-1$ dank Theorem 10. Davon gibt es höchstens $2nm$ Operationen laut Theorem 11, weshalb der Gesamtzuwachs in Φ durch alle saturierenden Push Operationen höchstens $2nm \cdot (2n-1)$ sein kann.

Nicht-saturierende Push Operationen senken Φ um mind. 1 ab. Anzahl beschränkt mit Gesamtzuwachs in Φ durch die anderen Operationen:

$$2nm(2n-1) + 2n(n-2) \leq 4n^2m.$$

Für die Ungleichung nehmen wir oBdA an, dass n die Anzahl der *von s aus erreichbaren Knoten* ist, denn dann gilt $m \geq n-1$. □

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 13. *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 14. *Ein Fluss f hat minimale Kosten genau dann, wenn es in G_f keinen Kreis mit negativen Kosten gibt.*

Beweis: Beide Richtungen durch Kontraposition.
"⇒": Sei K ist ein Kreis in G_f mit negativen 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.

"⇐": Sei f ein suboptimaler Fluss. Dann gibt es f^* mit $\ell(f^*) < \ell(f)$.

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 $\ell_f(\Delta) < 0$ gilt.

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

Der **Cycle-Canceling Algorithmus** nutzt diese Einsicht algorithmisch aus – er startet mit einem beliebigen Fluss mit Wert b und verbessert die Kosten Schritt für Schritt durch Verschiebung des Flusses in Kreisen mit negativen Kosten, so lange solch ein Kreis noch

existiert. Durch eine evtl. schlechte Wahl der Kreise 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.

Lemma 15. *Sei f ein Fluss mit minimalen Kosten 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.

Laut Theorem 14: G_f hat keinen Kreis mit negativen Kosten, $G_{f'}$ aber schon.

- Sei K ein Kreis 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** (siehe Algorithmus 6) 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 $O(b \cdot nm)$. Es geht auch deutlich schneller:

Theorem 16. *Für integrale Kapazitäten kann der Successive-Shortest-Path Algorithmus mit einer Laufzeit von $O(b \cdot (m + n \log n))$ implementiert werden.*

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

- G_f enthält anfangs keine negativen Kosten – mit $f = 0$ sind nur Kanten (u, v) in G_f, die auch in E sind. Alle diese Kanten haben $\ell(u, v) \geq 0$ per Definition.
- Für den ersten augmentierenden Pfad können wir also **Dijkstra's Algorithmus** nutzen. Laufzeit $O(m + n \log n)$

Wir wollen auch in den weiteren Iterationen die Kosten der Kanten in E_f nicht-negativ halten. Dazu definieren wir zu Anfang von Iteration i sog. **reduced costs**

$$\ell_i(u, v) = \ell(u, v) + h_i(u) - h_i(v)$$

für eine passende Wahl von h_i. Der Vollständigkeit halber beginnen wir für $i = 1$ mit Werten $h_1(u) = h_1(v) = 0$, und somit anfangs $\ell_1(u, v) = \ell(u, v)$.

In jeder Iteration i liefern die reduced costs immer die gleichen kürzesten Wege von s zu jedem Knoten w wie die originalen Kosten ℓ:

Algorithmus 6: Successive Shortest Path

Anfangs $f(u,v) = 0$ für alle $u, v \in V$.
`// Jede Kante` (u,v) `in` G_f `ist in` E `und hat daher` $\ell(u,v) \geq 0$
Setze $i = 1$ und $\ell_1(u,v) = \ell(u,v)$, für alle $(u,v) \in V \times V$
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 alle $v \in V$
    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`
    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

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

$$\begin{aligned}\ell_i(P) &= \sum_{(u,v)\in P} \ell(u,v) + h_i(u) - h_i(v) \\ &= h_i(s) - h_i(w) + \sum_{(u,v)\in P} \ell(u,v) \\ &= h_i(s) - h_i(w) + \ell(P)\end{aligned}$$

- Also gilt für s-w-Pfade P und P':

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

  d.h. jeder kürzeste Pfad von s zu einem anderen Knoten bleibt ein kürztester Pfad.

Wir versuchen nun induktiv, durch die Wahl geeigneter h_i sicherzustellen, dass ℓ_i negative Kantenkosten in G_f vermeidet. Dann bleiben kürzeste Wege von s erhalten, und wir können Dijkstra's Algorithmus zur Berechnung des kürztesten augmentierenden Pfades nutzen.

Das ist auf jeden Fall zu Anfang für $i = 1$ der Fall. Damit das auch in weiteren Iterationen so bleibt, arbeitet der Algorithmus wie folgt.

- Annahme: In der i-ten Iteration sei $\ell_i(u,v) = \ell(u,v) + h_i(u) - h_i(v) \geq 0$ für alle $(u,v) \in E_f$
- Nun nutze Dijkstra und berechne kostengünstigsten augmentierenden s-t-Pfad P. Der Algorithmus berechnet auch alle $\mu_i(s,v)$, die kürzeste Distanz von s zu $v \in V \setminus \{s\}$ in G_f bzgl. ℓ_i.

- Augmentiere entlang von P. Dadurch ändert sich G_f. Um weiterhin nicht-negative Kantenkosten zu behalten, nutzen wir

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

- G_f ändert sich nur entlang von P. Es könnten Kanten entlang von P verschwinden (in Richtung von P) oder entstehen (mit entgegengesetzter Richtung von P).
- Es gilt aber für jede Kante $(u, v) \in P$

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

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

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

  und jede sonstige Kante (u, v) in G_f

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

  da $\mu_i(s, v) \leq \mu_i(s, u) + \ell_i(u, v)$.
- Also gilt in Iteration $i+1$ wieder: Alle Kanten in E_f haben nicht-negative Kosten ℓ_{i+1}.
- Es gilt natürlich weiterhin: 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! □

Ähnlich wie bei Ford-Fulkerson kann man in den Successive-Shortest-Path Algorithmus eine Skalierung einbauen und erhält dadurch eine polynomielle Laufzeit. Es gibt sogar 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** $M \subseteq E$ 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)
vollständig im bipartiten Graphen $G = (A \cup B, E)$: Es gilt $|M| = \min(|A|, |B|)$. (M matcht alle Knoten der kleineren Partition)
maximum: Für jedes Matching M' gilt $|M'| \leq |M|$. (M hat größte Anzahl Kanten)
maximal: Für jede Kante $e \in E \backslash M$ gilt $M \cup \{e\}$ ist kein Matching. (M ist nicht erweiterbar)

Für jeden Graphen G gilt: perfekte Matchings $\subseteq$ maximum Matchings $\subseteq$ maximale Matchings.
Für bipartite Graphen G ausserdem: $|A| = |B| \iff$ Jedes vollständige Matching in G ist perfekt.

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. Der Anfangs- und Endknoten des Pfades ist ungematched in M.

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

In Algorithmus 7 nutzen wir Flüsse, Residualnetzwerke, augmentierende Pfade und den Ford-Fulkerson Algorithmus, um ein maximum Matching zu berechnen. Sei M^* ein maximum

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

Matching in G und f^* ein maximaler Fluss in G', der vom Algorithmus berechnet wird.

Theorem 17. *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.

"$\Leftarrow$": 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 18. *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 $O(m\sqrt{n})$.

Vollständige und perfekte Matchings

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

- Bipartiter Graph $G = (A \cup B, E)$ mit $|A| \leq |B|$ 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.

Wenn M vollständig 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 19 (Hall, König, etc.)**.** *Sei $G = (A \cup B, E)$ ein bipartiter Graph mit $|A| \leq |B|$. Dann enthält G **entweder** ein vollständiges Matching **oder** eine Menge $X \subseteq A$ mit $|X| > |\Gamma(X)|$. Ein vollständiges Matching oder die Menge X können in Zeit $O(nm)$ berechnet werden.*

Beweis: Wir nutzen die gleiche Konstruktion des Flussnetzwerkes G' wie oben. Sei $k = |A| \leq |B|$. Dann gilt: G hat ein vollständiges 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 $\{u, v\} \in (A \times B) \setminus E$, 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 $|A| = |B|$. Es gibt perfekte Matchings.
- Im optimalen perfekten Matching werden möglichst viele der original vorhandenen Kanten gewählt, denn die sind viel billiger.
- Wenn in zwei perfekten Matchings gleich viele Originalkanten gewählt werden, machen die Kosten der Originalkanten den Unterschied.

$\Rightarrow$ 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 20. *Ein binärer Fluss mit $f(u, v) \in \{-1, 0, 1\}$, Wert $|f| = k$ 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 Theorem 14 und Theorem 15 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 21. *M ist ein perfektes Matching mit minimalen Kosten genau dann wenn es für M keinen alternierenden Kreis mit negativen Kosten gibt.*

Beweis: Ähnlich zu Theorem 14:
"$\Rightarrow$": 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.

"$\Leftarrow$": 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 22. *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 binären Fluss f mit Wert $|f| = k$ und minimalen Kosten. Ein augmentierender Pfad im Restnetzwerk G'_f hat Restkapazität 1 auf allen Kanten und enspricht einem augmentierenen Pfad für M in G mit den gleichen Kosten. Daher ist das Resultat eine direkte Konsequenz von Theorem 15. □

Theorem 23. *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, dominieren in der Laufzeit die Terme $bm \leq nm = O(n^3)$.

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

2.2 Allgemeine Graphen

Lemma 25 (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. □

Algorithmus 8: Maximum Matching in allgemeinen Graphen

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

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

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

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

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

Algorithmus 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
```

- 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 26. *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] "⇐": 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

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

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. O.B.d.A. im Optimum kein Fluss, der nach s zurückfließt, machts nur unnötig teurer! Wir **löschen jede Kante** (v, s) **aus** E. Danach muss gelten: $\sum_{(s,v) \in E} x_{(s,v)} = b$

- Flusserhaltung bei allen 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:

$$\begin{aligned}
\text{Min.} \quad & \sum_{v\in V} w_v x_v & & & \\
\text{s.d.} \quad & x_u + x_v & \geq & 1 & \text{für jede } \{u,v\} \in E \\
& x_v & \in & \{0,1\} & \text{für jeden } v \in V
\end{aligned} \tag{3.2}$$

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 $\mathbb{R}^2$.

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

[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 $\boldsymbol{A}\boldsymbol{x} \geq \boldsymbol{b}$ enthält (Minimierung: in Richtung von $-\boldsymbol{c}$)
- Drei Möglichkeiten für LPs:
  (a) **Ungültiges** LP, leerer Lösungsraum
  (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 im Rest dieses Abschnitts 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 Hablräume), 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** $\boldsymbol{x} \in P$ 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 27. *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 $\boldsymbol{A}_B \boldsymbol{x} = \boldsymbol{b}_B$.

"⇒": 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 $\boldsymbol{A}_B(\boldsymbol{x} + \boldsymbol{y}) = \boldsymbol{b}_B + \boldsymbol{A}_B \boldsymbol{y} \geq \boldsymbol{b}_B$, also $\boldsymbol{A}_B \boldsymbol{y} \geq 0$.
- Aus $\boldsymbol{x} - \boldsymbol{y} \in P$ folgt genauso $\boldsymbol{A}_B \boldsymbol{y} \leq 0$. Also $\boldsymbol{A}_B \boldsymbol{y} = 0$ und $\boldsymbol{y} = \boldsymbol{0}$, da $\boldsymbol{A}_B$ regulär. Widerspruch.

"⇐": 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 $\boldsymbol{A}_N(\boldsymbol{x} + \boldsymbol{y}) \geq \boldsymbol{b}_N$ und $\boldsymbol{A}_N(\boldsymbol{x} - \boldsymbol{y}) \geq \boldsymbol{b}_N$
- Damit gilt $\boldsymbol{A}(\boldsymbol{x} \pm \boldsymbol{y}) \geq \boldsymbol{b}$ 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 28. *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. Sei das Ziel der Verschiebung $\boldsymbol{x}'$.
- Falls $\boldsymbol{x}'$ (noch) keine Ecke ist, gibt es 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 $\boldsymbol{x}'$ 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 Normalform 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. 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 29. *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 30. *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.
- 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 **Normalform**. Diese Formulierung ist nützlich für die technische Analyse.

Definition **LP in Normalform**:

- 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

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

[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).

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

- 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 Normalform 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 Normalform:

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

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:

$$\begin{array}{lrcl} \text{Min.} & c_1x_1 + c_2x_2 & & \\ \text{s.d.} & x_1 + x_2 & \leq & 1 \\ & x_1 & \geq & 0 \\ & x_2 & \geq & 0 \end{array} \qquad \begin{array}{lrcl} \text{Min.} & c_1x_1 + c_2x_2 & & \\ \text{s.d.} & x_1 + x_2 - s_1 & = & 1 \\ & x_1 & \geq & 0 \\ & x_2 & \geq & 0 \\ & s_1 & \geq & 0 \end{array}$$

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

Ecken in der Normalform:

- 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 (originales) LP in Normalform
- 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 Hilfs-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 Hilfs-LP ist $\boldsymbol{y} = \boldsymbol{b}$ und $\boldsymbol{x} = \boldsymbol{0}$. Starte Simplex auf Hilfs-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 LP-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** $O(m)$.

Algorithmus 10: Seidels LP-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 31. *Seidels LP-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 32. *Die Wahrscheinlichkeit, dass der Algorithmus in 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 33. *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:

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

3.4 Dualität

LPs haben eine überraschende Dualität – sie ergeben sich in Paaren. Das *duale LP* entsteht 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, *duales* LP formuliert werden. Primale und duale LPs sind eng verbunden. Dualität liefert elegante Strukturen, die z.B. beim Entwurf von Approximationsalgorithmen sehr nützlich sind.

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 → 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:

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

Die Transformation **geht in beide Richtungen**.

Beispiele: LP in Normalform

$$\begin{array}{rl} \text{Min.} & 13x_1 + 10x_2 + 6x_3 \\ \text{s.d.} & 5x_1 + x_2 + 3x_3 = 8 \\ & 3x_1 + x_2 = 3 \\ & x_1, x_2, x_3 \geq 0 \end{array}$$

$$\text{Min. } \begin{bmatrix} 13 & 10 & 6 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

$$\text{s.d. } \begin{bmatrix} 5 & 1 & 3 \\ 3 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 8 \\ 3 \end{bmatrix}$$

$$\boldsymbol{x} \geq \boldsymbol{0}$$

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.

$$\begin{array}{rlcll} \text{Min.} & \sum_{v \in V} x_v & & & \\ \text{s.d.} & x_u + x_v & \geq & 1 & \text{für jedes } e \in E \\ & x_v & \leq & 0 & \text{für jedes } v \in V \end{array} \tag{3.9}$$

und

$$\begin{array}{rlcll} \text{Max.} & \sum_{e \in E} y_e & & & \\ \text{s.d.} & \sum_{\{u,v\} \in E} y_{\{u,v\}} & \leq & 1 & \text{für jedes } v \in V \\ & y_e & \geq & 0 & \text{für jedes } e \in E \end{array} \tag{3.10}$$

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 34 (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 Normalform), 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 Theorem 34, dass beide **optimale Lösungen** für ihre jeweiligen LPs sein müssen.
- Theorem 34 zeigt auch:
  Primales unbeschränkt ⇒ Duales ungültig.
  Dual unbeschränkt ⇒ Primales ungültig.

Lemma 34 kann zu einem der zentralen Resultate der linearen Optimierung erweitert werden:

Theorem 35 (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 Normalform

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

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

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

- 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 Normalform.

- 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:

$$(x_B)_j = \frac{\det(\boldsymbol{a}_B^1, \ldots \boldsymbol{a}_B^{j-1}, \boldsymbol{b}, \boldsymbol{a}_B^{j+1}, \ldots, \boldsymbol{a}_B^m)}{\det(\boldsymbol{A}_B)} \tag{3.11}$$

- Die Ecke können wir also schreiben als

$$\boldsymbol{x} = \left(\frac{p_1}{q}, \frac{p_2}{q}, \ldots, \frac{p_n}{q}\right)^T \tag{3.12}$$

  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 36. *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

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

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 Normalform: $\boldsymbol{c}^T\boldsymbol{x} - \boldsymbol{y}^T\boldsymbol{b}$
- 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 Theorem 36 alle Ecken in der Nähe optimal sein müssen.

Interior-Point-Verfahren (sehr High-Level):

- Nutze folgende **Potenzialfunktion**, um den Fortschritt des Algorithmus sicherzustellen:
$$G(\boldsymbol{x}, \boldsymbol{s}) = (n + \sqrt{n}) \cdot \ln(\boldsymbol{x}^T\boldsymbol{s}) - \sum_{j=1}^{n} \ln(x_j s_j)$$
- 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:

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

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

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 37. *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 (3.11) und (3.12).
- 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 38. *Sei $\boldsymbol{A} \in \{-1, 0, 1\}^{m \times n}$ eine Matrix, so dass ...*
1. *jede Spalte $1 \leq j \leq n$ höchstens zwei Einträge $a_{ij} \neq 0$ beinhaltet.*
2. *die Zeilen von $\boldsymbol{A}$ in zwei disjunkte Mengen $I_1, I_2 \subseteq \{1, \ldots, m\}$ partitioniert werden können, sodass für alle Spalten $1 \leq j \leq n$ mit zwei Einträgen $a_{i_1 j}, a_{i_2, j} \neq 0$ folgendes gilt:*
   (a) *Falls $a_{i_1 j} = a_{i_2, j}$ erfüllt ist, so sind i_1 und i_2 in unterschiedlichen Mengen.*
   (b) *Falls $a_{i_1 j} = -a_{i_2, j}$ erfüllt ist, so sind i_1 und i_2 in der gleichen Mengen.*

Dann ist $\boldsymbol{A}$ total unimodular.

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 39. *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 Grundlagen

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 $\ell_j = \sum_{i \text{ auf } j} p_i$
- 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 40. *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 40: 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 41. *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. □

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 $|S|$.

Steinerbaum:
- Gegeben sei eine Partition von V in Terminalknoten $S \subseteq V$ und Nicht-Terminals $V \setminus S$
- Eine Menge der Kanten $T \subseteq E$ heißt *Steinerbaum* wenn sie kreisfrei ist und (mindestens) alle Terminalknoten in einen Baum verbindet.
- **Ziel:** Finde einen Steinerbaum T mit minimaler Größe $|T|$

Stellen wir uns (z.B. für das Clique-Problem) einen Algorithmus vor, der immer eine optimale Lösung berechnet. Der Algorithmus kann auch für die *Entscheidungsvariante des Problems* genutzt werden (z.B. entscheide, ob die größte Clique mindestens k Knoten enthält, für gegebenes k). Wenn das Entscheidungsproblem NP-vollständig ist, dann können wir nicht erwarten, dass der Algorithmus gleichzeitig
1. eine optimale Lösung berechnet,
2. in polynomieller Zeit terminiert, und
3. dies für jede Instanz des Problems tut.

Wir suchen einen *guten (suboptimalen) Algorithmus* für ein Optimierungsproblem, der *immer in Polynomialzeit terminiert*, 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 42. 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 40). 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. Theorem 41).

Können wir den Faktor $2 - 1/m$ noch verbessern? Ja, z.B. mit einer besseren Permutation, *longest processing time first (LPT).*

Theorem 43. *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 $4/3 \cdot OPT(I)$ überschreiten.
- 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 $i = 1, \ldots, m$ mit $m \leq n/2 - 1$ jeweils auf Maschine i, Tasks $i = m+1, \ldots, 2m$ jeweils auf Maschine $2m+1-i$.
- Das wird von LPT berechnet! Also $LPT(I) = OPT(I) \not> 4/3 \cdot OPT(I)$ - Widerspruch

□

Algorithmus 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

4.1.2 PTAS und FPTAS

Geht es noch besser als 4/3? Ja, aber dafür betrachten wir 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.

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

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

$\Rightarrow$ Nutze Analyse von ListScheduling aus Theorem 40 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 44. *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 $O(n \log n + m^{m/\varepsilon + m} + n \log m)$.

Theorem 45. *BruteForceList ist ein PTAS für Makespan-Scheduling mit konstanter Anzahl identischer 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 46. *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

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

- 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 47. *Betrachte ein* NP*-hartes Optimierungsproblem, bei dem für jede Instanz I*

Algorithmus 12: Matching Heuristik für Vertex Cover

```
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]
```

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 47 greift hier nicht!

4.2 Greedy Algorithmen

4.2.1 Vertex Cover, Clique, Independent Set

Theorem 47 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 48. *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 (3.9) und (3.10) 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 49. *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 49 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.

Theorem 50. *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:

Algorithmus 13: Spannbaum-Heuristik für 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

- Hamilton-Kreis Problem: 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.

Definition 51. *Ein* metrischer Raum (V, d) *besteht aus einer Menge V und einer Funktion $d : V \times V \to \mathbb{R}$, die die folgenden Eigenschaften erfüllt, für jede $u, v, w \in V$:*

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

[Pic: Dreiecksungleichung]

Korollar 52. *Δ-TSP ist NP-hart.*

Beweisidee: Nutze die Reduktion aus Theorem 50 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 jede Kante genau einmal durchläuft.

Theorem 53. *Sei G ein ungerichteter verbundener Graph. Dann gilt:*
G hat einen Eulerkreis $\iff$ Jeder Knoten in G hat geraden Grad.

[Pic: Beispiel Eulerkreis]

Die **Spannbaum-Heuristik** (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 54. *Algorithm 13 berechnet eine 2-Approximation für Δ-TSP in polynomieller Zeit.*

Beweis: Sei C^* eine optimale TSP-Tour, $d(C^*) = \sum_{(u,v)\in C^*} d(u, v)$ ihre Gesamtdistanz und $d(T) = \sum_{(u,v)\in T} d(u, v)$ die Gesamtdistanz von T. Wir beobachten, dass $d(C^*) \geq d(T)$:

- Entferne aus C^* die Kante (u, v) mit kleinster Distanz
- Sei $C_{u,v}$ die Menge der verbleibenden Kanten.
- $C_{u,v}$ ist ein Spannbaum von K_n. T ist minimaler Spannbaum von K_n.
- Also $d(C^*) \geq d(C_{u,v}) \geq d(T)$

Durch Duplizieren der Kanten gilt $2d(T) = d(\hat{T}) = d(C_{\hat{T}})$. Mit der Dreiecksungleichung gilt für das Abkürzen von $C_{\hat{T}}$, dass $d(C_{\hat{T}}) \geq d(C)$. Insgesamt:

$$d(C) \leq d(C_{\hat{T}}) = 2 \cdot d(T) \leq 2 \cdot d(C^*). \quad \square$$

Der Faktor 2 ergibt sich insbesondere durch das Duplizieren aller Kanten von T. Der **Christofides-Serdyukov Algorithmus** (Algorithmus 14) ist sorgfältiger bei der Konstruktion eines Eulerkreises. Er fügt nur Kanten für die Knoten $U \subseteq V$ hinzu, die ungeraden Grad in T haben. Dies verbessert den Faktor deutlich.

Theorem 55. *Algorithmus 14 berechnet eine 3/2-Approximation für Δ-TSP in polynomieller Zeit.*

Beweis: Der Beweis ist ähnlich zum Beweis von Theorem 54. Der Hauptunterschied ist, dass M die Gesamtdistanz von $d(\hat{T})$ um höchstens $d(C^*)/2$ erhöht:

- Sei C^*_U die Tour, die durch "Abkürzen" von C^* zwischen den Knoten von U entsteht
- Sume der Knotengrade in T ist gerade.
  $\Rightarrow T$ muss eine gerade Anzahl Knoten mit ungeradem Grad enthalten
  $\Rightarrow |U|$ ist gerade.
- C^*_U hat eine gerade Anzahl Kanten
- Zerteile C^*_U in zwei perfekte Matchings von U, sei M_U das Matching mit kleinerer Gesamtdistanz
- M_U hat Gesamtdistanz $d(M_U) \leq 0.5 \cdot d(C^*_U) \leq 0.5 \cdot d(C^*)$
- M ist Min-Distanz-Matching von U, also $d(M) \leq d(M_U)$.

Algorithmus 14: Christofides-Serdyukov Algorithmus für Δ-TSP

```
1 Consider the undirected, complete graph K_n with edge weights given by d
2 T ← minimaler Spannbaum in (K_n, d)
3 U ← Menge von Knoten in T mit ungeradem Grad
4 M ← Min-Distanz-Matching der Knoten in U
5 T̂ ← T ∪ M
6 C_T̂ ← Eulerkreis von T̂
7 Kürze die Kanten von C_T̂ ab und erhalte TSP-Tour C
8 return C
```

[Pic: Tour C_U^* durch Abkürzen von C^*, Partition in zwei Matchings]

Kombination dieser Einsicht mit den Argumenten oben ergibt

$$d(C) \leq d(C_{\hat{T}}) = d(T) + d(M) \leq d(C^*) + d(M_U) \leq 1.5 \cdot d(C^*) \quad \square$$

Die Laufzeit des Algorithmus wird dominiert durch die Suche nach dem Min-Distanz-Matching M der Knoten in U und liegt in $O(n^3)$. Der Algorithmus wurde unabhängig von Christofides und Serdyukov in 1976 entwickelt und war über 40 Jahre lang der Algorithmus mit dem kleinsten bekannten Approximationsfaktor für Δ-TSP. Von Kurzem haben Karlin, Klein, und Oveis Gharan (2021) einen Algorithmus mit kleinerem Faktor vorgestellt – er liegt bei ca. $3/2 - 10^{-36}$...

4.2.3 Gewichtetes k-Center

Das (gewichtete) k-Center-Problem fragt nach einem Partitionierung von n Datenpunkten, so dass die Cluster möglichst kleinen (gewichteten) Radius haben. Formal errichten wir k *Zentren* (z.B. Polizeistationen, Feuerwachen, etc.) an einer Auswahl der n Punkte, so dass die höchsten (gewichteten) Distanzkosten von jedem Punkt zu seinem nächstgelegenen Zentrum möglichst klein werden.

- Ein metrischer Raum (V, d) mit n Punkten (siehe Definition 51)
- Positionsgewichte $w(v) \geq 0$ für jeden Punkt $v \in V$
- Maximalanzahl $k \in \{1, \ldots, n\}$ von möglichen Zentren
- Sei $Z \subseteq V$ die Auswahl von Zentren, und sei $d(v, Z) = \min_{z \in Z} d(v, z)$ die Distanz zum nächstgelegenen Zentrum, für jedes $v \in V$
- **Ziel:** Wähle $Z \subseteq V$ mit $|Z| \leq k$, so dass $\max_{v \in V} w(v) \cdot d(v, Z)$ minimiert wird.

Wir untersuchen zuerst, welche Approximationsgüte wir bestenfalls erwarten können.

Theorem 56. *Für jedes $\varepsilon > 0$ gibt es keinen effizienten Algorithmus mit Approximationsfaktor $(2 - \varepsilon)$ für k-Center wenn $\mathsf{P} \neq \mathsf{NP}$.*

Beweis: Wir nutzen das NP-vollständige Entscheidungsproblem Dominating Set:

- Gegeben ein ungerichteter Graph $G = (V, E)$ und eine ganze Zahl k.

- $D \subseteq V$ heißt *Dominating Set* wenn, für jedes $v \in V$, mindestens ein Nachbar von v oder v selbst in D ist
- **Ziel:** Entscheide, ob es für G ein Dominating Set mit Kardinalität höchstens k gibt.

Dominating Set ist eine Variante von Vertex Cover, in der wir *jeden Knoten* durch (ihn selbst oder) einen Nachbarn überdecken wollen. Bei Vertex Cover überdecken wir *jede Kante* durch einen inzidenten Knoten.

[Pic: Dominating Set, Beispiel]

Einen $(2-\varepsilon)$-Approximationsalgorithmus für k-Center könnte man benutzen, um Dominating Set in polynomieller Zeit zu entscheiden. Für jede Instanz von Dominating Set konstruieren wir in polynomieller Zeit eine Instanz von k-Center:

- Wir übernehmen die Menge V, d.h. Knoten in G entsprechen Punkten im Raum.
- Gewichte $w(v) = 1$ für alle $v \in V$ ($\rightarrow$ Resultat gilt sogar für *ungewichtetes* k-Center-Problem!)
- Distanzen sind $d(u, v) = 1$ wenn $\{u, v\} \in E$ und $d(u, v) = 2$ sonst.
- Die Zahl k wird übernommen.
- Daher liefert jede Lösung von k-Center maximale Distanzkosten von 0, 1, oder 2.

Nehmen wir nun an, unser Algorithmus erzielt auf den resultierenden Instanzen für k-Center immer eine $(2 - \varepsilon)$-Approximation.

- Dominating-Set-Instanz ist Ja-Instanz
  $\iff$ Menge $D \subseteq V$ mit $|D| = k$ enthält v oder Nachbarn von v, für jedes $v \in V$
  $\iff$ Menge $D \subseteq V$ mit $|D| = k$ hat für jedes $v \in V$ ein Zentrum in Distanz höchstens 1
  $\iff$ k-Center-Instanz hat Lösung mit Kosten 0 oder 1.
- Also muss der Algorithmus auf der k-Center-Instanz, die sich aus einer Ja-Instanz von Dominating Set ergibt, eine Lösung mit Kosten höchstens $1 \cdot (2 - \varepsilon) < 2$ errechnen, also eine Lösung mit Kosten 0 oder 1 (also ein gültiges Dominating Set!).
- Bei Nein-Instanzen hat die k-Center-Instanz nur Lösungen mit Kosten 2.
- Algorithmus liefert ein Dominating Set mit Größe k wann immer es existiert. □

[Pic: Approximationsschranke]

Ein einfacher **Greedy-Algorithmus** (Algorithmus 15) erzielt eine 2-Approximation.

Theorem 57. *Algorithmus 15 berechnet eine 2-Approximation für gewichtetes k-Center in polynomieller Zeit.*

Beweis: Sei Z^* eine optimale Lösung. Wir definieren den dazugehörigen (gewichteten) **Radius**

$$r^* = \max_{v \in V} w(v) \cdot d(v, Z^*) = \max_{v \in V} \min_{z \in Z^*} w(v) \cdot d(v, z).$$

Wir kennen r^* nicht, aber es gibt insgesamt nur höchstens n^2 Möglichkeiten für r^*:

- r^* wird definiert durch einen Knoten v und sein nächstgelegenes Zentrum $z \in Z^*$.
- Wir müssen nur das Knotenpaar (v, z) aus Z^* kennen, so dass $r_{v,z} = r^* = w(v) \cdot d(v, z)$.
- Es gibt nur n^2 Möglichkeiten für so ein Paar (v, z), also nur n^2 Möglichkeiten für r^*.

Algorithmus 15: Greedy Algorithmus für gewichtetes k-Center

```
1  R ← ∅
2  for jedes Paar v, z ∈ V do R ← R ∪ {w(v)d(v, z)}
3  for alle r ∈ R aufsteigend sortiert do
4      i ← 0, U ← V, Z ← ∅
5      while i < k und U ≠ ∅ do
6          wähle z = arg max_{u∈U} w(u)
7          Z ← Z ∪ {z}
8          U ← U \ {u ∈ U | w(u) · d(u, z) ≤ 2 · r}
9          i ← i + 1
10     if U = ∅ then return Z
```

Wir generieren alle Möglichkeiten für den Radius in der Menge R. In der äußeren Schleife von Algorithmus 15 testen wir alle Möglichkeiten in aufsteigender Reihenfolge:

- In einem Durchlauf der For-Schleife versucht der Algorithmus, alle Punkte in V durch k Cluster mit **Radius** $2 \cdot r$ abzudecken.
- Dazu wird in der While-Schleife (bis zu) k-Mal greedy ein Punkt u aus den bisher nicht überdeckten Punkten U gewählt, der das größte Gewicht hat.
- u überdeckt (neben bereits durch andere Cluster überdeckten Knoten) weitere Punkte aus U mit gewichtetem Abstand höchstens $2r$ zu u.
- Der Algorithmus bricht ab beim kleinsten Radius r, der dazu führt, dass alle Punkte überdeckt werden (also $U = \emptyset$ nach k Runden). Dies liefert die Menge Z der Ausgabe.
- Sei $r_A = \max_{v \in V} \min_{z \in Z} w(v) \cdot d(v, z)$ der (echte) Radius der Ausgabe Z. Es gilt $r_A \leq 2 \cdot r$.

Wir zeigen, dass der Algorithmus spätestens bei $r = r^*$ abbricht. Dadurch gilt, dass am Ende des Algorithmus $r \leq r^*$, und daher $r_A \leq 2 \cdot r \leq 2 \cdot r^*$. Das Theorem ist bewiesen.

Bleibt zu zeigen, dass der Algorithmus spätestens bei $r = r^*$ abbricht, also $U = \emptyset$ am Ende der While-Schleife. Betrachte einen Radius $r \geq r^*$.

- Sei $z_i \in Z^*$ und $V_i = \{v \in V \mid w(v)d(v, z_i) \leq r^*\}$ die von z_i überdeckte Menge an Punkten
- Sei $z \in V_i \cap Z$ das erste gewählte Zentrum aus V_i.
- Dann gilt für alle aktuell noch unüberdeckten Punkte $v \in V_i \cap U$:

$$\begin{aligned} w(v) \cdot d(v, z) &\leq w(v) \cdot (d(v, z_i) + d(z_i, z)) && \text{(Dreiecksungleichung)} \\ &= w(v) \cdot d(v, z_i) + w(v) \cdot d(z, z_i) && \text{(Symmetrie)} \\ &\leq w(v) \cdot d(v, z_i) + w(z) \cdot d(z, z_i) && (z \text{ hat größtes Gewicht in } U) \\ &\leq 2r^* \\ &\leq 2r \end{aligned}$$

- Durch Wahl von z wird jeder Knoten von V_i aus U entfernt.
- Aus jeder Menge V_i wird im Algorithmus höchstens ein Zentrum gewählt.
- Es werden höchstens k Zentren gewählt, am Ende der While-Schleife gilt $U = \emptyset$. □

Algorithmus 16: FPTAS für Rucksack

```
1  w_max ← max_{i∈[n]} w_i
2  s ← ε · w_max/n
3  ŵ_i = ⌊w_i/s⌋                                          // Hilfsprofite
   // Dynamisches Programm mit Hilfsprofiten
4  Ŵ = Σ_{i∈[n]} ŵ_i
5  G(0,0) ← 0 und G(0,x) ← ∞ für 1 ≤ x ≤ Ŵ
6  for i = 1,...,n do
7      for W' = 1,...,W do
8          G(i,W') ← min(G(i−1,W'), g_i + G(i−1,W' − ŵ_i))
   // Umgekehrte Reihenfolge für die eingepackten Gegenstände
9  W* ← max{W' | G(n,W') ≤ G}
10 S ← ∅
11 for i = n,...,1 do
12     if G(i,W*) = g_i + G(i−1,W* − ŵ_i)) then
13         S ← (S ∪ {i})
14         W* ← W* − ŵ_i
15 return S
```

[Pic: Cluster, optimale Zentren, höchstens ein Zentrum aus Z in jedem Cluster vom Optimum]

4.3 Dynamische Programmierung

4.3.1 Rucksack

Das Rucksackproblem ist ein zentrales Packproblem:

- Menge $[n]$ von n Gegenständen, i hat Gewicht $g_i \geq 0$ und Profit $w_i \geq 0$
- Rucksack hat Größenschranke G. O.B.d.A. $g_i \leq G$ für alle $i \in [n]$
- Teilmenge $S \subseteq [n]$ ist *gültig* wenn sie in den Rucksack passt: $\sum_{i \in S} g_i \leq G$
- **Ziel:** Finde eine gültige Teilmenge, die den Gesamtprofit maximiert.

Wir nehmen wieder an, dass wir alle Gewichte und Profite hochskalieren, so dass die Parameter **nicht-negative ganze Zahlen** sind. Das folgende Argument funktioniert auch für reellwertige Gewichte, aber wir nutzen aus, dass die Profite ganzzahlig sind.

Theorem 58. *Algorithm 16 ist ein FPTAS für Rucksack mit Laufzeit in $O(n^3/\varepsilon)$*

Sei $W = \sum_{i=1}^{n} w_i$ die Summe aller Profite. Wir beschreiben zuerst das dynamische Programm für Rucksack. Betrachte die Teilprobleme:

- Teilproblem: $G(i, W')$ ist die kleinste Gewichtssumme einer Teilmenge $S \subseteq \{1, \ldots, i\}$, die Gesamtprofit $\sum_{j \in S} w_j = W'$ hat.

- Wir verlangen Gesamtprofit W' und betrachten alle Teilmengen der ersten i Gegenstände, die exakt diesen Gesamtprofit haben. Was ist das kleinste Gesamtgewicht einer solchen Menge?
- Wenn es keine solche Teilmenge von $\{1, \ldots, i\}$ gibt, dann sei $G(i, W') = \infty$.
- Wir betrachten die Teilprobleme $G(i, W')$ für alle $1 \leq i \leq n$ und $0 \leq W' \leq W$.

Basisfälle und Rekursion:

- Basisfälle: $G(1, w_1) = g_1$ und $G(1, x) = \infty$ für alle $0 \leq x \leq W', x \neq g_1$.
- Rekursion: Gibt es für Teilproblem $G(i, W')$ eine Teilmenge mit kleinstem Gesamtgewicht, die i beinhaltet?
- (1) Ja: Dann $G(i, W^*) = g_i + G(i-1, W' - w_i)$, d.h. wir kombinieren Gegenstand i mit Teilmenge von $\{1, \ldots, i-1\}$ mit kleinstem Gewicht, die Gesamtprofit genau $W' - w_i$ hat.
- Wenn $W' - w_i < 0$, dann ist das unmöglich, und dann gilt $G(i-1, W' - w_i) = \infty$.
- (2) Nein: Keine Teilmenge mit kleinstem Gesamtgewicht beinhaltet i. Also ist die beste Teilmenge eine Teilmenge von $\{1, \ldots, i-1\}$ und $G(i, W') = G(i-1, W')$.
- Insgesamt wählen wir die bessere der beiden Alternativen:

$$G(i, W') = \min\{G(i-1, W'), g_i + G(i-1, W' - w_i)\}$$

[Pic: Tabelle der Teilprobleme $G(i, W')$, Rekursionsschema]

Das Ausfüllen der Tabelle beginnt mit den o.g. Basisfällen für $i = 1$. Danach berechnen wir nacheinander für $i = 2, 3, 4 \ldots$ jeweils alle Einträge von $G(i, W')$.

Lemma 59. *Das dynamische Programm zur Lösung des Rucksack-Problems hat Laufzeit $O(nW)$.*

Dies ist *keine polynomielle Laufzeit*, da die Zahlen in der Eingabe logarithmisch codiert sind. Die Zahlen $w_1, \ldots, w_n$ brauchen nur $O\left(\sum_{i=1}^{n} \log_2(w_i)\right)$ Bits in der Eingabe. W ist also i.A. nur exponenziell in der Eingabegröße. Die Laufzeit des Algorithmus ist nur **pseudopolynomiell**, d.h. sie wäre polynomiell wenn jede Zahl w_i in unärer Codierung gegeben wäre.

Definition 60. *In der* unären Eingabe *einer Instanz wird jeder numerische Wert unär codiert. Die Laufzeit eines Algorithmus wird* pseudopolynomiell *genannt, wenn sie polynomiell beschränkt ist in der Länge der unären Eingabe.*

Statt einer exakten Lösung in pseudopolynomieller Zeit berechnen wir eine approximative Lösung in polynomieller Zeit. Dazu "machen wir die Profite klein".

[Pic: Hilfsprofite]

Beweis von Theorem 58 Gegeben eine Zahl $\varepsilon > 0$ runden wir jeden Wert w_i zum nächstkleineren Vielfachen einer geeignet gewählten Zahl $s > 1$. Dadurch erhalten wir Hilfsprofite $\hat{w}_i = \lfloor w_i / s \rfloor$. Wir wenden das dynamische Progamm mit diesen Profiten an.

Wir wählen $s = \varepsilon w_{\max}/n$. Was ist der Einfluss auf die Laufzeit?

- Summe aller Hilfsprofite sinkt auf höchstens $\lfloor W/s \rfloor$.

- Mit Theorem 59 ergibt sich die Laufzeit $O(nW/s)$.
- Die Abschätzung
$$\frac{n \cdot W}{s} = \frac{n^2 \cdot W}{\varepsilon w_{\max}} \leq \frac{n^2 \cdot n \cdot w_{\max}}{\varepsilon w_{\max}} = \frac{n^3}{\varepsilon}$$
zeigt eine Laufzeitschranke von $O(n^3/\varepsilon)$.

Und die Approximationsgarantie? Sei S die Ausgabe des Algorithmus und S^* eine optimale Lösung. Dann gilt

$$\begin{aligned}
\sum_{i \in S^*} w_i &= \sum_{i \in S^*} s \cdot \frac{w_i}{s} \leq \sum_{i \in S^*} s \cdot \left(\lfloor \frac{w_i}{s} \rfloor + 1 \right) \leq ns + s \cdot \sum_{i \in S^*} \hat{w}_i \\
&\leq ns + s \cdot \sum_{i \in S} \hat{w}_i && \text{(da } S \text{ optimal für } \hat{w}_i\text{)} \\
&\leq ns + s \cdot \sum_{i \in S} \frac{w_i}{s} = \quad \varepsilon w_{\max} + \sum_{i \in S} w_i && \text{(Definition von } s\text{)}
\end{aligned}$$

Mit $\sum_{i \in S^*} w_i \geq w_{\max}$ ergibt sich

$$\sum_{i \in S} w_i \geq \sum_{i \in S^*} w_i - \varepsilon w_{\max} \geq (1 - \varepsilon) \sum_{i \in S^*} w_i = \frac{\sum_{i \in S^*} w_i}{1/(1-\varepsilon)}$$

Der Approximationsfaktor ist höchstens $1/(1-\varepsilon) \leq 1 + 2\varepsilon$. □

4.3.2 Vertex Cover auf Bäumen

Wir diskutieren einen Ansatz zur Lösung von Optimierungsproblemen auf Bäumen mit dynamischer Progammierung. Der Ansatz kann für viele Probleme angepasst werden, die auf allgemeinen Graphen NP-hart sind. Die Idee wird hier am gewichteten Vertex Cover Problem vorgestellt.

- $G = (V, E)$ Baum, d.h. schlichter, ungerichteter, zusammenhängender Graph ohne Kreise
- Jeder Knoten $v \in V$ hat Kosten/Gewichte $w_v \geq 0$.
- **Ziel:** Finde ein Vertex Cover C mit kleinsten Gesamtkosten $\sum_{v \in C} w_v$

Wir konstruieren ein dynamisches Programm.

- Wurzele Baum in beliebigem Knoten r. Jeder Knoten (außer r) hat genau einen Elternknoten
- Knoten ohne Kinder sind *Blätter*
- Sei T_v der Teilbaum von v und allen Nachkommen von v
- Ein Teilproblem für jeden Knoten/Teilbaum. Konstruiere die Lösung "bottom-up":
- Starte bei den Blättern. Gegeben Lösungen für alle Kinderknoten konstruiere Lösung für den Elternknoten

An jedem Knoten berechnen wir zwei Werte:

$$W^0(v) = \text{ Kosten des günstigsten Vertex Cover von } T_v\text{, das } v \text{ nicht enthält}$$

$$W^1(v) = \text{ Kosten des günstigsten Vertex Cover von } T_v, \text{ das } v \text{ enthält}$$

Rekursion ergibt sich wie folgt:

- Sei v ein Blatt, dann $W^0(v) = 0$ und $W^1(v) = w_v$
- Andernfalls, seien $v_1, \ldots, v_k$ die Kinder von v.
- Wenn v nicht im günstigsten Cover ist, dann müssen alle Kanten zu den Kindern von den Kindern überdeckt werden. Also müssen sie alle im Cover sein:

$$W^0(v) = \sum_{i=1}^{k} W^1(v_i)$$

- Wenn v im günstigsten Cover ist, müssen die Kinder nicht unbedingt dabei sein. Wir können für jedes Kind die Option wählen, die für den jeweiligen Teilbaum des Kindes günstiger ist:

$$W^1(v) = w_v + \sum_{i=1}^{k} \min(W^0(v_i), W^1(v_i))$$

[Pic: v raus $\rightarrow$ jedes Kind muss rein. v drin $\rightarrow$ Kinder können drin sein oder nicht (was besser ist bzgl. Kosten)]

Die besten Kosten eines Vertex Cover sind gegeben durch $\min(W^0(r), W^1(r))$. Zur Konstruktion der optimalen Menge C^* gehen wir "top-down" vor:

- Minimum bei $W^0(r)$: r bleibt draussen und alle Kinder müssen in C^* sein. Die Kinderknoten werden für C^* "angemeldet".
- Minimum bei $W^1(r)$: r kommt in C^*. Die Kinderknoten werden nicht angemeldet.
- Wende die Regeln rekursiv an: Jeder angemeldete Knoten muss in C^* aufgenommen werden.
- Jeder Knoten, der nicht von seinem Elternknoten angemeldet wurde, kann rein oder raus, je nachdem was günstigere Kosten liefert.
- Jeder Knoten, der draussen bleibt, meldet seine Kinderknoten für C^* an.
- Auf diese Weise errechnen wir eine optimale Lösung.

Theorem 61. *Das gewichtete Vertex Cover Problem auf Bäumen kann in polynomieller Zeit optimal gelöst werden.*

4.4 LP-basierte Algorithmen

4.4.1 Makespan-Scheduling mit allgemeinen Maschinen

Wir verallgemeinern Makespan-Scheduling auf allgemeine (also evtl. nicht identische) Maschinen:

- m (nicht notwendigerweise identische) Maschinen, n Tasks
- Task $i \in [n]$ hat Bearbeitungszeit $p_{ij} \in \mathbb{N}$ auf Maschine $j \in [m]$
- Weise jeden Task i (vollständig) zu auf eine Maschine $j \in [m]$
- Last von Maschine j ist $\ell_j = \sum_{i \text{ auf } j} p_{ij}$
- Maximale Last $\max_{j \in [m]} \ell_j$ ist der Makespan der Zuweisung

- **Ziel:** Verteile alle Tasks auf die Maschinen und minimiere den Makespan.

Representation: Vollständiger bipartiter Graph G, Knotenmengen sind Tasks und Maschinen, Kantengewichte sind Bearbeitungszeiten. Zuweisung ist ein "one-to-many"-Matching in G.

[Pic: Vollständiger, bipartiter, kantengewichteter Graph. Zuweisung als "one-to-many"-Matching]

Die Annahme, dass alle $p_{ij} \in \mathbb{N}$ ist oBdA: Wenn wir $p_{ij} \in \mathbb{Q}$ erlauben für alle $i \in [n], j \in [m]$, erhalten wir nach Multiplikation aller Zahlen mit dem Hauptnenner eine äquivalente Instanz mit integralen Bearbeitunszeiten. Diese Multiplikation vergrößert die Eingabelänge nur um einen polynomiellen Faktor (in der Repräsentation der orignalen rationalen Zahlen).

Für eine Formulierung als ILP nutzen wir $x_{ij} = 1$ wenn Task i auf Maschine j zugewiesen wird, und 0 sonst. Das ILP ergibt sich als

$$\begin{aligned} \text{Min.} \quad & t & & & \\ \text{s.d.} \quad & \sum_{i=1}^{n} x_{ij} p_{ij} & \leq & t & \text{für jede } j \in [m] \\ & \sum_{j=1}^{m} x_{ij} & = & 1 & \text{für jeden } i \in [n] \\ & x_{ij} & \in & \{0,1\} & \text{for jede } i \in [n], j \in [m] \end{aligned} \tag{4.2}$$

Bemerkungen:

- Erste Constraints: t ist mindestens der Makespan (Minimierung ergibt t = Makespan)
- Zweite Constraints: Jeder Task i auf genau einer Maschine.
- LP-Relaxierung: Ersetze $x_{ij} \in \{0,1\}$ durch $x_{ij} \geq 0$ (zweite Constraints implizieren $x_{ij} \leq 1$)
- Die LP-Relaxierung hat eine **riesige Integralitätslücke**. Die optimale LP-Lösung kann bis zu einem Faktor m besser sein als die beste ganzzahlige Lösung[1].

Wir nutzen das LP auf eine andere Weise. Wir **raten eine obere Schranke** T für den Makespan und **entfernen (engl. pruning)** "Zuweisungskanten" (i, j) mit $p_{ij} > T$. Keine dieser Kanten kann in einer ganzzahligen Lösung mit Makespan T genutzt werden.

Gegeben eine Schranke T für den Makespan, sei $E_T = \{\{i, j\} \in [n] \times [m] \mid p_{ij} \leq T\}$ die Menge verbleibender Zuweisungskanten. Wir suchen eine gültige *fraktionale* Zuweisung im verbleibenen Graphen $G_T = ([n] \cup [m], E_T)$. Die folgenden Ungleichungen beschreiben das Polytop der gültigen fraktionalen Zuweisungen mit Makespan höchstens T in der verbleibenden Instanz. Wir nennen dies $LP(T)$:

$$\begin{aligned} \sum_{i:\{i,j\} \in E_T} x_{ij} p_{ij} & \leq T & \text{for jede } j \in [m] \\ \sum_{j:\{i,j\} \in E_T} x_{ij} & = 1 & \text{for jeden } i \in [n] \\ x_{ij} & \geq 0 & \text{for jede } \{i, j\} \in E_T \end{aligned} \tag{4.3}$$

Algorithmus 17: Parametrisiertes Pruning

```
1  for jeden Task i do
2      Weise i einer Maschine j_i = arg min_{j∈[m]} p_ij zu mit kleinster Bearbeitungszeit für i
3  Sei T_g der Makespan der Greedy-Zuweisung
4  Binäre Suche im Interval [T_g/m, T_g] nach der kleinsten ganzen Zahl T*, so dass LP(T) (4.3) eine gültige Lösung hat
5  x ← Ecke von LP(T*)
6  for jeden Eintrag x_ij = 1 do weise i auf Maschine j zu
7  Konstruiere Graph H(x) der fraktional zugewiesenen Tasks und ihrer Maschinen
8  M ← vollständiges Matching in H(x)
9  for jedes Paar {i, j} ∈ M do weise Task i auf Maschine j zu
10 return Zuweisung von Tasks zu Maschinen
```

Betrachte nun Algorithmus 17. Bemerkungen:

- Zuerst wird eine greedy Zuweisung erstellt, um eine einfache obere Schranke T_g für den optimalen Makespan zu berechnen.
- T_g/m muss eine untere Schranke für den optimalen Makespan sein (Übung)
- Suche den kleinsten Integer T^*, für den $LP(T)$ (4.3) gültige (franktionale) Lösungen hat.
- Wähle eine Ecke $\boldsymbol{x}$ des Polytops $LP(T^*)$. Weise Tasks zu, die in $\boldsymbol{x}$ integral zugewiesen sind.
- Die verbleibenden Tasks werden gerundet: Nutze einen Graphen $H(\boldsymbol{x})$ mit Kanten zwischen diesen Tasks und den Maschinen, auf die diese Tasks fraktional zugewiesen sind.
- Mit einem vollständigen Matching M der Tasks und Maschinen in $H(\boldsymbol{x})$ ergibt sich die Zuweisung.

Wir zeigen nun das Hauptresultat dieses Abschnitts.

Theorem 62. *Algorithm 17 berechnet eine 2-Approximation für Makespan-Scheduling mit allgemeinen Maschinen in polynomieller Zeit.*

Eine grobe obere Schanke auf die Laufzeit der Binärsuche ist $O\left(\log \sum_{i,j} p_{ij}\right)$, da $[T_g/m, T_g] \subset [0, \sum_{i,j} p_{ij}]$. Dies ist höchstens Linearzeit in der Länge der Eingabe – die Eingabelänge ist mindestens $\sum_{i,j} \log p_{ij}$. Der Test auf Gültigkeit von $LP(T)$ und die anderen Schritte des Algorithmus benötigen auch nur polynomielle Zeit.

Für den Beweis des Approximationsfaktors zeigen wir zuerst nützliche Eigenschaften der Ecken $\boldsymbol{x}$ des Polytops (4.3). Danach diskutieren wir die Definition von $H(\boldsymbol{x})$ und zeigen, dass der Graph immer ein vollständiges Matching enthält, in dem jeder Task gematcht wird. Als letzten Schritt beweisen wir den Faktor 2.

[1] Instanz mit einem Task und $p_{1j} = 1$ für jede Maschine $j \in [m]$. Fraktionales Optimum bricht Task in m Teile mit $x_{1j} = 1/m$, Makespan ist $1/m$. Integralitätslücke mindestens m.

Lemma 63. *Jede Ecke $\boldsymbol{x}$ von $LP(T)$ hat höchstens $n+m$ strikt positive Variablen.*

Beweis: Sei $r = |E_T|$ die Anzahl der Variablen in $LP(T)$.
- Definition einer Ecke: $\boldsymbol{x}$ erfüllt r linear unabhängige Constraints exakt
- $n+m$ Constraints der ersten zwei Klassen in (4.3)
- Mindestens $r-(n+m)$ Constraints "$x_{ij} \geq 0$" exakt erfüllt
- Mindestens $r-(n+m)$ viele $x_{ij} = 0$, also höchstens $n+m$ Variablen strikt größer 0. □

Korollar 64. *In jeder Ecke $\boldsymbol{x}$ sind mindestens $n-m$ Tasks integral zugewiesen.*

Beweis: Sei $k = |\{\ \{i,j\} \in E_T \mid x_{ij} > 0\}|$ die Anzahl der strikt positiven Variablen.
- Vorheriges Lemma: $k \leq n+m$
- Task i fraktional $\rightarrow$ mindestens zwei $x_{ij} > 0$. Task i integral $\rightarrow$ ein $x_{ij} > 0$.
- Wenn wir α fraktionale und $n-\alpha$ integrale Tasks haben, dann $2\alpha+n-\alpha \leq k \leq n+m$.
- Also $\alpha \leq m$. Anzahl integraler Tasks ist $n-\alpha \geq n-m$. □

Wir definieren einen **Pseudo-Wald**:
- Ein Pseudo-Baum ist ein zusammenhängender Graph mit k Knoten und höchstens k Kanten.
  $\rightarrow$ Es ist entweder ein Baum oder ein Baum mit einer weiteren Kante.
- Ein Pseudo-Wald ist ein Graph, in dem jede Komponente ein Pseudo-Baum ist.

[Pic: Beispiele Pseudo-Baum, Pseudo-Wald]

Wir definiern zwei Graphen. In $G(\boldsymbol{x}) = ([n] \cup [m], E(\boldsymbol{x}))$ gibt es Kanten $E(\boldsymbol{x}) = \{\{i,j\} \in E_T \mid x_{ij} > 0\}$, die den positiven Zuweiseungen von $\boldsymbol{x}$ entsprechen. In $H(\boldsymbol{x})$ beschränken wir $G(\boldsymbol{x})$ auf Tasks i, die in $\boldsymbol{x}$ *fraktional* sind (die also mindestens zwei Einträge $0 < x_{ij} < 1$ haben). $H(\boldsymbol{x})$ enthält alle fraktionalen Tasks, ihre inzidenten Kanten und ihre adjazenten Maschinen aus $G(\boldsymbol{x})$.

Betrachten wir zuerst $G(\boldsymbol{x})$.

Lemma 65. *Für jede Ecke $\boldsymbol{x}$ ist der Graph $G(\boldsymbol{x})$ ein Pseudo-Wald.*

Beweis: Betrachte eine Zusammenhangskomponente C von $G(\boldsymbol{x})$.
- Beschränke $\boldsymbol{x}$ auf $\boldsymbol{x}_C$ mit Einträgen (nur) für Maschinen und Tasks in C
- Sei $\boldsymbol{x}_{-C}$ der Rest von $\boldsymbol{x}$, d.h. $\boldsymbol{x} = (\boldsymbol{x}_C, \boldsymbol{x}_{-C})$
- Wir definieren $LP_C(T)$ mit Einschränkung von (4.3) auf C. In $LP_C(T)$ gibt es nur Constraints für Tasks und Maschinen von C. In jedem Constraint summieren wir nur über Variablen x_{ij}, für die i und j beide aus C.
- $\boldsymbol{x}_C$ ist gültige Lösung für $LP_C(T)$.
- Wir behaupten: $\boldsymbol{x}_C$ ist auch eine *Ecke* von $LP_C(T)$.
- Beweis durch Widerspruch – wenn nicht, dann ist $\boldsymbol{x}_C$ eine innere Lösung von $LP_C(T)$. Dann $\boldsymbol{x}_C = \lambda \boldsymbol{x}_C^{(1)} + (1-\lambda)\boldsymbol{x}_C^{(2)}$ mit zwei gültigen Lösungen $\boldsymbol{x}_C^{(1)}$ und $\boldsymbol{x}_C^{(2)}$ für $LP_C(T)$
- Die Erweiterungen $\boldsymbol{x}^{(1)} = (\boldsymbol{x}_C^{(1)}, \boldsymbol{x}_{-C})$ und $\boldsymbol{x}^{(2)} = (\boldsymbol{x}_C^{(2)}, \boldsymbol{x}_{-C})$ sind gültig für $LP(T)$
- Also gilt $\boldsymbol{x} = \lambda \boldsymbol{x}^{(1)} + (1-\lambda)\boldsymbol{x}^{(2)}$ und $\boldsymbol{x}$ ist innere Lösung für $LP(T)$ und *keine Ecke* – Widerspruch!

Wie haben gezeigt, dass für jede Zusammenhangskomponente C der Vektor $\boldsymbol{x}_C$ eine Ecke von $LP_C(T)$ darstellt.

- Sei n_C die Anzahl Tasks in C und m_C die Anzahl Maschinen, also $n_C + m_C$ Knoten in C.
- $\boldsymbol{x}_C$ ist Ecke, also mit Theorem 63 hat $\boldsymbol{x}_C$ maximal $n_C + m_C$ strikt positive Variablen.
- Kanten in C sind strikt positive Variablen $\rightarrow$ C hat höchstens $n_C + m_C$ Kanten und ist zusammenhängend
- C ist ein Pseudo-Baum!

Jede Komponente von $G(\boldsymbol{x})$ ist ein Pseudo-Baum. □

Betrachten wir nun $H(\boldsymbol{x})$.

Lemma 66. *Für jede Ecke $\boldsymbol{x}$ ist $H(\boldsymbol{x})$ ein Pseudo-Wald und hat ein vollständiges Matching M.*

Beweis: $H(\boldsymbol{x})$ ist ein Pseudo-Wald:

- Jeder integrale Task in $\boldsymbol{x}$ hat genau eine inzidente Kante in $G(\boldsymbol{x})$.
- Integrale Tasks sind Blätter im Pseudo-Wald $G(\boldsymbol{x})$.
- Entferne diese Tasks mit ihren Kanten. Entferne isolierte (Maschinen-)Knoten.
- Das ergibt $H(\boldsymbol{x})$. Jede verbleibende Komponente bleibt ein Pseudo-Baum.

Jetzt betrachte eine Komponente von $H(\boldsymbol{x})$.

- Jeder Task hat Grad mindestens 2 in $H(\boldsymbol{x})$. Alle Blätter sind Maschinen.
- Konstruiere Matching M "bottom-up":
  Matche eine Blatt-Maschine j zum einzigen benachbarten Task i.
  Entferne i und j (sowie andere Blatt-Maschinen inzident zu i).
- Dadurch bleiben die Blätter immer Maschinen.
- Am Ende sind alle Tasks eindeutig zu Maschinen gematched, oder wir entdecken einen Kreis.
- Der Kreis hat gerade Länge (denn $H(\boldsymbol{x})$ ist bipartit). Matche Tasks und Maschinen entsprechend.

Also gilt: $H(\boldsymbol{x})$ erlaubt ein vollständiges Matching, in dem jeder Task gematcht wird. □

[Pic: Beispielgraph, Matching Konstruktion "bottom-up" in jedem Pseudo-Baum]

Nun können wir den Beweis von Theorem 62 beenden.

Beweis von Theorem 62: Betrachte den optimalen Makespan $T_I^* \in \mathbb{N}$ einer integralen Zuweisung.

- Pruning für $LP(T)$ entfernt nur Optionen, die für integrale Zuweisungen mit Makespan T unmöglich sind.
- Jede optimale integrale Zuweisung mit Makespan T_I ist eine gültige Lösung für $LP(T_I)$.
- T^* ist die kleinste ganze Zahl, so dass $LP(T)$ gültig ist $\rightarrow$ es gilt $T^* \leq T_I^*$.

Sei $T(\boldsymbol{x})$ der Makespan einer Ecke $\boldsymbol{x}$, der vom Algorithmus berechnet wird.

- $T(\boldsymbol{x}) \leq T^*$, da $\boldsymbol{x}$ gültig für $LP(T^*)$.
- Einschränkung der Zuweisung auf integrale Tasks von $\boldsymbol{x}$ ergibt $\ell_j \leq T(\boldsymbol{x})$ für jede Maschine

- Durch das Matching M erhält jede Maschine j höchstens einen weiteren Task i
- Bearbeitungszeit dieses Task ist $p_{ij} \leq T^*$, da Kante $\{i,j\} \in E_{T^*}$ nicht entfernt wurde

Am Ende hat jede Maschine j eine Last von $\ell_j \leq T(\boldsymbol{x}) + T^* \leq 2 \cdot T^* \leq 2 \cdot T_I^*$. □

4.4.2 Primal-duale Algorithmen

Primal-duale Algorithmen nutzt mant zur Approximation von Problemen, die als ILP formuliert sind. Man arbeitet mit der LP-Relaxierung und verwaltet primale und duale Vektoren. Dies ist ähnlich wie beim Interior-Point Verfahren, allerdings sind die Vektoren sind nicht durchgängig gültige Lösungen und daher nicht im Inneren des Lösungspolytops der LP-Relaxierung. Ausserdem sucht man nur eine approximative integrale Lösung und keine optimale fraktionale Lösung.

- Konstruiere iterativ $\boldsymbol{x}$ und $\boldsymbol{y}$ und verringere die Dualitätlücke $\boldsymbol{c}^T\boldsymbol{x} - \boldsymbol{y}^T\boldsymbol{b}$
- Finale Vektoren $\boldsymbol{x}$ und $\boldsymbol{y}$ sind gültige primale und duale Lösungen
- Finales $\boldsymbol{x}$ sollte eine *integrale* Lösung sein!
- Wenn $\boldsymbol{c}^T\boldsymbol{x} \leq \alpha \cdot \boldsymbol{y}^T\boldsymbol{b}$, für ein $\alpha > 0$, dann ist $\boldsymbol{x}$ eine α-approximative Lösung für das (originale, integrale) Problem, denn mit starker Dualität gilt

$$\boldsymbol{c}^T\boldsymbol{x} \leq \alpha \cdot \boldsymbol{y}^T\boldsymbol{b} = \alpha \cdot \boldsymbol{c}^T\boldsymbol{x}^*_{LP} \leq \alpha \cdot \boldsymbol{c}^T\boldsymbol{x}^*_{ILP}$$

Eine nützliche Eigenschaft ist die **Komplementaritätsbedindung (engl. complementary slackness)** von optimalen Lösungen der primalen und dualen LPs. Eine (primale/duale) Variable kann nur ungleich Null sein wenn das entsprechende (duale/primale) Constraint exakt erfüllt (oder **tight**) ist. Formal gilt für optimale Lösungen $\boldsymbol{x}^*$ und $\boldsymbol{y}^*$:

Primal: Für jedes j gilt $x_j^* = 0$ oder $(\boldsymbol{y}^*)^T\boldsymbol{a}^j = c_j$ oder beides. Wenn $x_j^* \neq 0$, dann muss das duale Constraint tight sein. Wenn das Constraint nicht tight ist, dann gilt $x_j^* = 0$.

Dual: Für jedes i gilt $y_i^* = 0$ oder $\boldsymbol{a}_i\boldsymbol{x}^* = b_i$ oder beides. Wenn $y_i^* \neq 0$, dann muss das primale Constraint tight sein. Wenn das Constraint nicht tight ist, dann gilt $y_i^* = 0$.

Bemerkungen:

- Viele primal-duale Algorithmen halten primale Komplementaritätsbedindungen ein, sie setzen $x_j \neq 0$ nur wenn das duale Constraint $\boldsymbol{y}^T\boldsymbol{a}^j = c_j$ tight ist. Die duale Bedingung ist oft weniger relevant.
- Die LP-Relaxierung wird meist nicht optimal gelöst. Wir suchen nur *gültige* Paare primaler und dualer Lösungen mit kleiner Dualitätslücke.
- Oft haben duale Variablen y_i eine kombinatorische Interpretation. Damit kann man manchmal kombinatorische Algorithmen entwickeln ohne Nutzung von LPs.

Die Algorithmen in den folgenden Abschnitten ergeben sich über folgendens High-Level-Schema.

- Primal: $\min \boldsymbol{c}^T\boldsymbol{x}$ s.d. $\boldsymbol{A}\boldsymbol{x} \geq \boldsymbol{b}$ und $\boldsymbol{x} \geq \boldsymbol{0}$.
- Dual: $\max \boldsymbol{y}^T\boldsymbol{b}$ s.d. $\boldsymbol{y}^T\boldsymbol{A} \leq \boldsymbol{c}^T$ und $\boldsymbol{y} \geq \boldsymbol{0}$.
- Wir nehmen an, dass $\boldsymbol{A} \geq \boldsymbol{0}$, $\boldsymbol{b} \geq \boldsymbol{0}$, $\boldsymbol{c} \geq \boldsymbol{0}$.

Struktur der Algorithmen:

1. Starte mit $\boldsymbol{x} = \boldsymbol{y} = \mathbf{0}$. ($\boldsymbol{x}$ ist nicht primal gültig, aber $\boldsymbol{y}$ ist gültige duale Lösung)
2. Wir halten primale Komplementarität ein: $(c_j - \boldsymbol{y}^T \boldsymbol{a}^j) \cdot x_j = 0$.
3. Erhöhe duale Variablen $\boldsymbol{y}$ bis ein (weiteres) Constraint $\boldsymbol{y}^T \boldsymbol{a}^k \leq c_k$ exakt erfüllt.
4. Erhöhe x_k, um (weitere) primale Constraints zu erfüllen
5. Wiederhole Schritte 3. und 4. bis $\boldsymbol{x}$ gültige primale Lösung ist.

4.4.3 Set Cover

Gewichtetes Set Cover Problem:

- Menge $E = \{1, \ldots, m\}$ von Elementen, Mengensystem $\mathcal{S} = \{S_1, \ldots, S_n\} \subseteq 2^E$, d.h. eine Menge von n Teilmengen von E
- Jedes $S_j \subseteq E$ hat Gewicht/Kosten $w_j \geq 0$.
- Sei $C \subseteq \{1, \ldots, n\}$ eine Menge von Teilmengenindizes für die Kollektion $\{S_j \mid j \in C\} \subseteq \mathcal{S}$.
- C ist ein *Set Cover* wenn es für jedes Element $e \in E$ mindestens ein $j \in C$ gibt mit $e \in S_j$.
- Anders gesagt: $\bigcup_{j \in C} S_j = E$, d.h. C *überdeckt* E.
- **Ziel:** Finde ein Set Cover C mit kleinsten Gesamtkosten $w(C) = \sum_{j \in C} w_j$.

Beispiel: Elemente $E = \{1, \ldots, 10\}$, Mengen

- $S_1 = \{3, 4, 5, 8\}$
- $S_2 = \{1\}$
- $S_3 = \{1, 3, 5, 7, 9\}$
- $S_4 = \{2, 4, 10\}$
- $S_5 = \{2, 3, 4, 5, 6\}$
- $S_6 = \{2, 6, 10\}$

$\{S_1, S_3, S_4, S_5\}$ ist ein Set Cover. $\{S_1, S_3, S_6\}$ ist ein anderes Set Cover.

Vertex Cover ist ein Spezialfall von Set Cover:

- Kante $e \longleftrightarrow$ Element e
- Knoten $v \longleftrightarrow$ Menge $S_v = \{e \in E \mid e = \{u, v\}\}$ der inzidenten Kanten/Elemente
- Vertex Cover $C = \{v \mid S_v \in C'\} \subseteq V \longleftrightarrow$ Set Cover $C' = \{S_v \mid v \in C\} \subseteq \mathcal{S}$.

Wir nutzen eine binäre Variable x_j um auszudrücken, ob S_j im Set Cover sein soll oder nicht. Die LP-Relaxierung (primal) und das duale LP ergeben sich als

$$\begin{aligned}
&\text{Min.} \quad \sum_{j=1}^{n} w_j x_j \\
&\text{s.d.} \quad \sum_{j : e \in S_j} x_j \geq 1 \quad \text{für jedes } e \in E \\
&\qquad\quad x_j \geq 0 \quad \text{für jedes } j \in [n]
\end{aligned}
\qquad\qquad
\begin{aligned}
&\text{Max.} \quad \sum_{e \in E} y_e \\
&\text{s.d.} \quad \sum_{e \in S_j} y_e \leq w_j \quad \text{für jedes } j \in [n] \\
&\qquad\quad y_i \geq 0 \quad \text{für jedes } e \in E
\end{aligned}
\tag{4.4}$$

Primal-dualer Algorithmus

Der primal-duale Algorithmus für gewichtetes Set Cover ist gegeben in Algorithmus 18. Die Laufzeit ist offensichtlich polynomiell beschränkt.

Algorithmus 18: Primal-dualer Algorithmus für gewichtetes Set Cover

```
1 Setze C ← ∅, x ← 0, und y ← 0
2 while es gibt e ∈ E noch unüberdeckt durch C do
3     Erhöhe y_e auf den höchsten Wert, der die dualen Constraints einhält:
      y_e ← min_{j:e∈S_j} ( w_j − Σ_{e'∈S_j, e'≠e} y_{e'} )
4     Sei S_k eine Menge, für die das duale Constraint durch die Erhöhung tight wird
      // Primale Komplementarität: Tightes Constraint erlaubt x_k > 0.
5     Setze x_k ← 1 und füge S_k hinzu: C ← C ∪ {k}
6 return C
```

- Interpretiere duale Variablen y_e als "Zahlungen". Element e zahlt, um Sets zu "kaufen", die e enthalten. Anfangs zahlt kein Element und das Cover C ist leer.
- Wir erhöhen iterativ die Zahlung eines unüberdeckten Elementes bis die Zahlungen ausreichen, um die Kosten w_j einer Menge S_j zu bezahlen, die e enthält. Dieses Menge wird gekauft und ins Cover C übernommen.
- Beachte, dass e seine Zahlung y_e *allen* Mengen anbietet, die e enthalten. Formal wird y_e in jedem dualen Constraint mitgerechnet, für alle Mengen S_j mit $e \in S_j$.
- Im Verlauf des Algorithmus werden evtl. weitere Mengen S_k, die e enthalten, gekauft, um andere Elemente zu überdecken. e muss dabei y_e für jede diese Mengen S_k bezahlen – die anderen Elemente zahlen nur die verbleibenden Summen, um die Gesamtzahlung $\sum_{e' \in S_k} y_{e'}$ auf w_k anzuheben.
- Eine einzelne Zahlung y_e ist eine untere Schranke auf die Kosten, um e zu überdecken. Aber um wieviel mehr müssen die Elemente zahlen, um die Kosten der Lösung C des Algorithmus zu decken?

[Pic: Beispiele für die Ausgabe des Algorithmus]

Wir definieren $f = \max_{e \in E} |\{j \mid e \in S_j\}|$ als die maximale Anzahl von Mengen, in denen ein einzelnes Element vorkommt. Der Approximationsfaktor des Algorithmus ist f.

Theorem 67. *Algorithmus 18 berechnet eine f-Approximation für gewichtetes Set Cover in polynomieller Zeit.*

Beweis: Betrachte die gesamten Zahlungen aller Elemente.

- Sei ergeben genau die Kosten des Set Cover, denn die Elemente zahlen exakt die Kosten der gekauften Sets
$$\sum_{j=1}^{n} x_j w_j = \sum_{j \in C} \sum_{e \in S_j} y_e \ .$$
- Mit Definition von f sind die Gesamtzahlungen höchstens
$$\sum_{j \in C} \sum_{e \in S_j} y_e \leq f \cdot \sum_{e \in E} y_e \ ,$$

denn jede Zahlung y_e kann nur bei maximal f Mengen geleistet werden.

Es ergibt sich mit Dualität

$$\sum_{j=1}^{n} x_j w_j = \sum_{j \in C} \sum_{e \in S_j} y_e \leq f \cdot \sum_{e \in E} y_e \leq f \cdot \sum_{e \in E} y_e^* = f \cdot \sum_j x_j^* w_j \ ,$$

d.h. die Gesamtkosten des berechneten Set Cover C sind höchstens f-Mal die Kosten einer optimalen (primalen) Lösung $\boldsymbol{x}^*$ der LP-Relaxierung. □

Der Algorithmus hält *primale* Komplementarität ein. Was ist mit den *dualen* Bedingungen?

- Für die finalen Lösungen $\boldsymbol{x}$ und $\boldsymbol{y}$ können die dualen Bedingungen nicht auch noch gelten – dann wären die Lösungen optimal für beide LPs, was oft unmöglich ist für ganzzahliges $\boldsymbol{x}$.
- Anstatt der exakten Bedingungen

  $$y_e > 0 \quad \Rightarrow \quad \boldsymbol{a}_e \cdot \boldsymbol{x} = b_e$$

  gelten *approximative Versionen*

  $$y_e > 0 \quad \Rightarrow \quad \boldsymbol{a}_e \cdot \boldsymbol{x} \leq f \cdot b_e$$

  denn $b_e = 1$ und $\boldsymbol{a}_e \cdot \boldsymbol{x} = \sum_{j: e \in S_j} x_j$ zählt die Anzahl Mengen, in denen Element e enthalten ist (maximal f per Definition).

Der Faktor der Verletzung der dualen Komplementarität und der Approximationsfaktor sind gleich. Das ist kein Zufall und eine allgemeine Eigenschaft!

Theorem 68. *Seien $\boldsymbol{x}$ und $\boldsymbol{y}$ primale und duale Lösungen, für die die primalen Komplementaritätsbedingungen exakt erfüllt*

$$x_j > 0 \quad \Rightarrow \quad \boldsymbol{y}^T \boldsymbol{a}^j = c_j$$

und die dualen Bedingungen approximativ mit Faktor $\beta \geq 1$ erfüllt sind

$$y_i > 0 \quad \Rightarrow \quad \boldsymbol{a}_i \cdot \boldsymbol{x} \leq \beta \cdot b_i.$$

Dann ist $\boldsymbol{x}$ eine β-approximative Lösung für das primale LP.

Beweis: Übung.

Greedy Algorithmus

Wir betrachten einen weiteren, einfachen Greedy-Algorithmus für Set Cover (Algorithmus 19). Er folgt *nicht direkt unserem Schema* für primal-duale Algorithmen. Wir analysieren ihn aber mit primal-dualen Argumenten.

Beobachtungen:

- Offensichtlich implementierbar in polynomieller Zeit.

Algorithmus 19: Greedy Algorithmus für gewichtetes Set Cover

```
1 Setze C ← ∅, E_C ← ∅
2 while E_C ≠ E do
3     Finde Menge S_k mit kleinsten Kosten pro unüberdecktem Element w_k / |S_k \ E_C|
4     Füge Menge hinzu C ← C ∪ {k} und aktualisiere überdeckte Elemente
      E_C ← E_C ∪ S_k
5 return C
```

- Erzielt H_m**-Approximation** (höchstens $1 + \ln m$, auch wenn f groß wird). Es gibt Instanzen von Vertex Cover mit $f = 2$, auf denen der Algorithmus nur Faktor H_m erzielt.
- Es ist NP-hart, eine $(1 - \varepsilon) \ln m$-approximative Lösung für Set Cover zu erzielen, für jede Konstante $\varepsilon > 0$. Faktor ist also optimal wenn $\mathsf{P} \neq \mathsf{NP}$.

Theorem 69. *Algorithm 19 berechnet eine H_m-Approximation für gewichtetes Set Cover in polynomieller Zeit.*

Proof. Betrachte primale und duale LPs (4.4) oben.

- Für Element $e \in E$ sei t die Iteration, in der e zum ersten Mal überdeckt wird durch Hinzufügen von Menge S_j zu C. Setze die *Zahlung* von e auf
$$p_e = \frac{w_j}{|S_j \setminus E_C^t|} \ ,$$
wobei E_C^t die Menge E_C in Iteration t bezeichnet.
- Insgesamt gilt $\sum_{e \in E} p_e = \sum_{j \in C} w_j$.
- Skaliere die Zahlungen runter, um eine gültige duale Lösung zu erhalten (sog. **dual fitting**):
$$y_e = p_e / H_m,$$
mit $H_m = \sum_{i=1}^{m} 1/i \leq 1 + \ln m$ der m-ten harmonischen Zahl.
- Für duale Gültigkeit betrachte eine beliebige Menge S_k und ihr duales Constraint.
- Sei $r = |S_k|$. Nummeriere die Elemente $S_k = \{e_1, e_2, \ldots, e_r\}$ in steigender Ordnung der Iteration, in der sie zuerst überdeckt werden.
- Am Anfang einer Iteration t, in der Element e_ℓ überdeckt wird, gibt es mindestens $r - (\ell - 1)$ unüberdeckte Elemente in S_k. Wenn nun e_ℓ durch Hinzufügen von S_j überdeckt wird, dann
$$p_{e_\ell} = \frac{w_j}{|S_j \setminus E_C^t|} \leq \frac{w_k}{|S_k \setminus E_C^t|} \leq \frac{w_k}{r - \ell + 1} \ ,$$
denn der Algorithmus wählt S_j mit minimalen Kosten pro unüberdecktem Element.
- Für die Summe aller Elemente in S_k gilt
$$\sum_{\ell=1}^{r} p_{e_\ell} \leq w_k \cdot \sum_{\ell=1}^{r} \frac{1}{r - \ell + 1} = w_k \cdot H_m \ .$$

- Daraus folgt
$$\sum_{e \in S_k} y_e = \sum_{e \in S_k} \frac{p_e}{H_m} \leq w_k \ ,$$
d.h. das duale Constraint für S_k ist erfüllt. $\boldsymbol{y}$ ist eine gültige duale Lösung.

Das Cover C ist eine primale Lösung $\boldsymbol{x}$ mit $x_j = 1$ für $j \in C$ und 0 sonst. Dualität und Optimalität ergeben

$$\sum_{j \in C} w_j = \sum_{j=1}^{n} x_j w_j = \sum_{e \in E} p_e = H_m \cdot \sum_{e \in E} y_e \leq H_m \cdot \sum_{e \in E} y_e^* = H_m \cdot \sum_{e \in E} x_j^* w_j \ ,$$

d.h. die Gesamtkosten der Ausgabe C sind höchstens um Faktor H_m größer als die Kosten einer optimalen (primalen) Lösung $\boldsymbol{x}^*$ der LP-Relaxierung. □

4.4.4 Steinerwald

Das (gewichtete) Steinerbaum Problem ist eine natürliche Verallgemeinerung von minimalen Spannbäumen und kürzesten s-t-Pfaden in Graphen. Wir suchen eine Teilmenge der Kanten mit minimalen Kosten, so dass alle Knoten aus einer *Teilmenge* $V_1 \subseteq V$ (genannt Terminals) paarweise verbunden sind. Der resultierende Baum kann auch andere, sog. *Steinerknoten* enthalten – diese werden aber nur für Erreichbarkeit oder Kostenminimierung eingefügt, um alle Terminals möglichst günstig zu verbinden.

- $G = (V, E)$ *ungerichteter* Graph, Kantenkosten $\ell_e \geq 0$ für jede $e \in E$
- Partition von V in *Terminals* $V_1 \subseteq V$ und *Steinerknoten* $V \setminus V_1$
- *Steinerbaum* $T \subseteq E$: $G' = (V, T)$ enthält einen u-v-Pfad für jedes Paar $u, v \in V_1$
- **Ziel:** Finde einen Steinerbaum mit kleinsten Gesamtkosten $\sum_{e \in T} \ell_e$

[Pic: Beispiel Steinerbaum]

Wir betrachten direkt eine Verallgemeinerung, das Steinerwald Problem. Dies ist ein einfaches Beispiel für (oftmals deutlich komplexere) Probleme des Netzwerkentwurfs, mit vielen wichtigen Anwendungen in Telekommunikation, Logistik, etc.

Eine Instanz von Steinerwald enthält k verschiedene *Typen* von Terminals $V_1, \ldots, V_k$. Wir müssen eine Kantenmenge F wählen, so dass, für jeden Typen i, alle Terminals von Typ i paarweise verbunden sind:

- $G = (V, E)$ ungerichteter Graph, Kantenkosten $\ell_e \geq 0$ für jede $e \in E$.
- k Knotenmengen $V_1, \ldots, V_k \subseteq V$ (*Terminals*)
- *Steinerwald* $F \subseteq E$: $G = (V, F)$ enthält einen u-v-Pfad für jedes Paar $u, v \in V_i$ gleichen Typs, von jedem Typen $i = 1, \ldots, k$.
- **Ziel:** Finde einen Steinerwald mit kleinsten Gesamtkosten $\sum_{e \in F} \ell_e$

Bemerkungen:

- F verbindet jedes Paar $u, v \in V_i$ von Terminals gleichen Typs, aber nicht notwendigerweise Paare $u \in V_i$, $v \in V_j$ mit unterschiedlichem Typ $i \neq j$.
- Terminals mit verschiedenem Typ werden nur verbunden, wenn dies notwendig ist, um Terminals mit gleichem Typ zu verbinden oder falls es eine billigere Lösung ist im Sinne der Kantenkosten.

Algorithmus 20: Primal-dualer Algorithmus für Steinerwald

Setze $F_1 \leftarrow \emptyset$, $\boldsymbol{x} \leftarrow 0$, $\boldsymbol{y} \leftarrow 0$, $t \leftarrow 1$
while *F_t kein gültiger Steinerwald* **do**
  $\mathcal{C}_t \leftarrow \{W \mid W$ Knotenmenge einer Zusammenhangskomponente in $(V, F_t)\}$
  $\mathcal{C}_t^1 \leftarrow \{W \in \mathcal{C}_t \mid f(W) = 1\}$
  Erhöhe *alle* y_W, $W \in \mathcal{C}_t^1$, *uniform* bis duales Constraint tight für ein $e' \in E_W, W \in \mathcal{C}_t^1$
  `// Primale Komplementarität: Tightes Constraint erlaubt` $x_{e'} > 0$.
  $\Delta_t \leftarrow$ Wert, um den wir jedes y_W, $W \in \mathcal{C}_t^1$ erhöht haben
  $F_{t+1} \leftarrow F_t \cup \{e'\}$ und $x_{e'} \leftarrow 1$
  $t \leftarrow t + 1$
$F \leftarrow F_t$
while *es gibt $e \in F$ mit $F \setminus \{e\}$ gültig* **do** $F \leftarrow F \setminus \{e\}$, $x_e \leftarrow 0$
return F

- O.B.d.A. können wir annehmen, dass die Terminalmengen disjunkt sind: $V_i \cap V_j = \emptyset$ für $i \neq j$. (Warum?)

[Pic: Beispiel Steinerwald, Terminaltypen, disjunkte Mengen]

Wir konstruieren ein ILP. Sei $f : 2^V \to \{0, 1\}$ eine Funktion, die für jede Teilmenge $W \subseteq V$ angibt, ob diese Knoten eine Verbindung nach draußen (also nach $V \setminus W$) benötigen:

$$f(W) = \begin{cases} 1 & \text{es gibt } i \in [k] \text{ und } u, v \in V_i \text{ mit } u \in W \text{ und } v \notin W \\ 0 & \text{sonst.} \end{cases}$$

Falls $f(W) = 1$, müssen wir mindestens ein Terminal aus W mit mindestens einem Terminal außerhalb von W verbinden. Sei $E_W = \{\{u, v\} \in E \mid u \in W, v \notin W\}$, die Menge der (ungerichteten) Kanten im Schnitt $(W, V \setminus W)$. Jeder Steinerwald F muss mindestens eine Kante aus E_W enthalten.

[Pic: Terminals, Schnitt W, Funktion $f(W) \in \{0, 1\}$]

Binäre Variablen $x_e \in \{0, 1\}$ sollen anzeigen ob $e \in F$ ($x_e = 1$) oder nicht ($x_e = 0$). Sei $\mathcal{S} = \{W \subset V \mid f(W) = 1\}$ die Menge der *notwendigen Schnitte* in G, die ein Steinerwald überbrücken muss. Das ILP und das Duale der LP-Relaxierung sind wie folgt gegeben:

$$\begin{array}{lll} \text{Min.} & \displaystyle\sum_{e \in E} x_e \ell_e & \\ \text{s.d.} & \displaystyle\sum_{e \in E_W} x_e \geq 1 & \text{für jede } W \in \mathcal{S} \\ & x_e \in \{0, 1\} & \text{für jede } e \in E \end{array} \qquad \begin{array}{lll} \text{Max.} & \displaystyle\sum_{W \in \mathcal{S}} y_W & \\ \text{s.d.} & \displaystyle\sum_{W : e \in E_W} y_W \leq \ell_e & \text{für jede } e \in E \\ & y_W \geq 0 & \text{für jede } W \in \mathcal{S} \end{array} \tag{4.5}$$

Unser Algorithmus 20 löst das Steinerwald Problem. Anmerkungen:

- Statt einzeln werden *alle* dualen Variablen y_W gleichmäßig um den gleichen Wert erhöht, für alle Mengen W die Zusammenhangskomponenten von (V, F_t) entsprechen, die eine Verbindung nach außen brauchen.
- In jeder Iteration t gibt es höchstens n Zusammenhangskomponenten in (V, F_t). Wir müssen höchstens n duale Variablen erhöhen.
- Ausserdem wird in der Iteration mindestens ein neues duales Constraint tight für eine Kante $e' \rightarrow$ höchstens $m = |E|$ Iterationen. Am Ende insgesamt höchstes nm positive Variablen. Statt alle (exponenziell viele) y_W verwalten wir nur die *strikt positiven* Variablen.
- In Iteration t kennen wir die Kanten, die Zusammenhangskomponenten von (V, F) verlassen und die zu erhöhenden dualen Variablen. Daher können wir effizient den kleinsten Wert Δ_t finden, bei dem ein weiteres duales Constraint tight wird.
- Der finale Schrumpfung von F in der While-Schleife kann effizient implementiert werden. Dies ist notwendig – ansonsten könnte F viel zu viele Kanten enthalten (Übung).

[Pic: Beispieldurchlauf des Algorithmus, "wachsende Ringe" um Terminals und Komponenten]

Unsere Beobachtungen zeigen das folgende Resultat:

Lemma 70. *Algorithm 20 kann effizient implementiert werden.*

Das Hauptresultat ist die Approximationsgarantie.

Theorem 71. *Algorithm 20 berechnet eine 2-Approximation für Steinerwald.*

Der Algorithmus hält *exakte primale* Komplementarität ein. Der Faktor 2 könnte aus *approximativer dualer* Komplementarität resultieren wenn für jede Menge $W \in \mathcal{S}$

$$y_W > 0 \quad \Rightarrow \quad \sum_{e \in E_W} x_e \leq 2 \ .$$

Leider ist es nicht so einfach – wir erhöhen immer wieder *alle* y_W der Komponenten. Es ist unklar, ob die Bedingung am Ende gilt *für jeden Schnitt* W *mit* $y_W > 0$. Stattdessen zeigen wir eine "Durchschnitts-Version" der Bedingung für alle Schnitte $W \in \mathcal{C}_t^1$, für die wir in einer Iteration t die Variablen y_W erhöhen.

Lemma 72. *Sei F (und $\boldsymbol{x}$) die Ausgabe des Algorithmus. Am Anfang jeder Iteration t gilt*

$$\sum_{W \in \mathcal{C}_t^1} |E_W \cap F| = \sum_{W \in \mathcal{C}_t^1} \sum_{e \in E_W} x_e \leq 2 \cdot |\mathcal{C}_t^1|.$$

Wir zeigen zuerst, wie man das Theorem mit Hilfe des Lemmas beweist.

Beweis von Theorem 71: Betrachte die primale Lösung $\boldsymbol{x}$ am Ende des Algorithmus:

$$\sum_{e \in E} x_e \ell_e = \sum_{e \in F} \ell_e = \sum_{e \in F} \sum_{W : e \in E_W} y_W \qquad \text{(primale Komplementarität)}$$

$$= \sum_{W \in \mathcal{S}} \sum_{e \in F \cap E_W} y_W \qquad \text{(gruppiere alle Vorkommen von } y_W\text{)}$$
$$= \sum_{W \in \mathcal{S}} y_W \cdot |E_W \cap F|$$

(nun wäre approx. duale Kompl. mit $|E_W \cap F| \leq 2$ hilfreich, aber gilt leider nicht...)

In jeder iteration t werden die dualen Variablen y_W jeweils um Δ_t erhöht, für jedes $W \in \mathcal{C}_t^1$. Insgesamt ist die Erhöhung der dualen Variablen

$$\sum_{W \in \mathcal{S}} y_W = \sum_t \sum_{W \in \mathcal{C}_t^1} \Delta_t = \sum_t \Delta_t \cdot |\mathcal{C}_t^1| \ . \tag{4.6}$$

$\boldsymbol{y}$ bleibt natürlich dual gültig. Also

$$\sum_{e \in E} x_e \ell_e \leq \sum_{W \in \mathcal{S}} y_W \cdot |E_W \cap F| = \sum_t \sum_{W \in \mathcal{C}_t^1} \Delta_t \cdot |E_W \cap F| \quad (\Delta_t \text{ Zuwachs in } y_W \text{ für jedes } W \in \mathcal{C}_t^1)$$
$$= \sum_t \Delta_t \cdot \sum_{W \in \mathcal{C}_t^1} |E_W \cap F| \leq \sum_t \Delta_t \cdot 2 \cdot |\mathcal{C}_t^1| \qquad \text{(mit Theorem 72)}$$
$$= 2 \cdot \sum_t \Delta_t \cdot |\mathcal{C}_t^1| = 2 \cdot \sum_{W \in \mathcal{S}} y_W \qquad \text{(mit Gleichung (4.6))}$$
$$\leq 2 \cdot \sum_{W \in \mathcal{S}} y_W^* \qquad (\boldsymbol{y} \text{ dual gültig})$$
$$= 2 \cdot \sum_{e \in E} x_e^* \ell_e. \qquad \text{(starke Dualität)} \quad \square$$

Als letzten Schritt beweisen wir Theorem 72.

Beweis von Theorem 72: Zu Beginn einer Iteration t nehmen wir den (finalen) Steinerwald F und "kontrahieren" alle Kanten aus F_t (d.h. verschmelzen alle Knoten zu einem Superknoten). Das ergibt H_t, einen "Wald aus Komponenten" in Iteration t:

- (V, F_t) ist auch ein Wald: Am Anfang ist F_1 ein leerer Wald. Kante e' wird nur zwischen verschiedenen Komponenten hinzugefügt $\rightarrow$ alle F_t sind azyklisch.
- Seien $\mathcal{C}_t = \{W_1, \ldots, W_d\}$ die (Knotenmengen der) Komponenten in Iteration t.
- Konstruiere den Graphen H_t: Füge einen Knoten v_i ein für jede Menge W_i. Füge Kante $\{v_i, v_j\}$ ein für jedes $\{u, v\} \in F$ mit $u \in W_i$ und $v \in W_j$.
- Da F ein Wald ist, muss H_t ein Wald sein.

[Pic: Finaler Wald F, Wald F_t in Iteration t, kontrahierter Wald H_t]

Jeder Knoten v_i in H_t ist *aktiv* ($f(W_i) = 1$) oder *inaktiv* ($f(W_i) = 0$).

- Jeder isolierte Knoten in H_t ist ein leerer Baum und hat Grad 0.
- Betrachte einen Knoten v_i mit Grad 1 in H_t. Es gibt eine Kante $\{v_i, v_j\}$ in H_t. Warum ist $\{v_i, v_j\} \in F$? Wenn $f(W_i) = 0$, dann würde die Kante nie hinzugefügt (oder gelöscht in der While-Schleife am Ende). Also muss v_i aktiv sein.
- Ein Knoten v_i mit Grad ≥ 2 kann inaktiv sein – evtl. $f(W_i) = 0$, aber andere Komponenten verbinden sich evtl. "durch v_i hindurch".

Wir beschränken uns auf alle nicht-leeren Bäume mit aktiven Knoten in H_t. Dann hat jeder *inaktive* Knoten Grad mindestens 2, also ist der Durchschnittsgrad inaktiver Knoten *mindestens* 2. H_t ist ein Wald mit $1 \leq r \leq d$ nicht-leeren Bäumen, also ist der Durchschnittsgrad aller Knoten in diesen Bäumen $2 - (2r/n) < 2$. Damit ist der Durchschnittsgrad von *aktiven* Knoten in H_t *höchstens* 2. Aktive Knoten entsprechen den Komponenten von $\mathcal{C}_t^1$, ihre inzidenten Kanten sind aus F. Daher

$$\frac{1}{|\mathcal{C}_t^1|} \sum_{W \in \mathcal{C}_t^1} |E_W \cap F| \leq 2 \ . \quad \square$$

Algorithmus 20 und Theorem 71 können auf Min-Cost-Wald Probleme erweitert werden, bei denen die Verbindungsanforderungen für Teilmengen $W \subset V$ ausgedrückt werden können durch eine Funktion $f : 2^V \to \{0, 1\}$ mit den folgenden Bedindungen:

1. $f(\emptyset) = f(V) = 0$
2. $f(W) = f(V \setminus W)$ für alle $\emptyset \subseteq W \subseteq V$
3. $f(W \cup U) \leq \max\{f(W), f(U)\}$ für je zwei disjunkte $W, U \subset V$

Wir nennen eine solche Funktion f *passend (engl. proper)*. Für passendes f gibt es immer eine optimale Lösung des ILP 4.5, die einen Wald $F \subseteq E$ darstellt. Wenn $f(W)$ für jede Komponente $W \in \mathcal{C}_t$ in polynomieller Zeit berechnet werden kann, dann können wir jede Iteration t in Algorithmus 20 in polynomieller Zeit implementieren. Die Beweise von Theorem 72 und Theorem 71 sind ohne Anpassung gültig für jedes Min-Cost-Wald Problem mit passendem f.

Kapitel 5

Randomisierte Algorithmen

Für randomisierte Algorithmen nutzen wir zufällige Entscheidungen. Die Algorithmen haben dabei Zugriff auf **interne Random-Bits**, die evtl. abhängig von der Eingabeinstanz generiert werden.

Beispiel: QuickSort.
Gegeben sei ein unsortieres Array A mit n Zahlen.

- Wähle einen zufälligen Eintrag als Pivotelement p.
- Partitioniere den Array in $A_{<p}$, p und $A_{\geq p}$.
- Sortiere $A_{<p}$ und $A_{\geq p}$ rekursiv.

Das Array wird immer korrekt sortiert. Die zufällige Wahl des Pivotelements legt p mit konstanter Wahrscheinlichkeit (ungefähr) in die "Mitte" der (unbekannten) sortierten Folge. Dadurch sind die Größen von $A_{<p}$ und $A_{\geq p}$ im rekursiven Aufruf mit guter Wahrscheinlichkeit um einen **konstanten Faktor** kleiner als n.

Durch die zufällige Wahl erhält man eine **average-case** oder **erwartete Laufzeit**. Sie liegt bei $O(n \log n)$ – bei einer worst-case Auswahl kann das Verfahren bis zu $\Theta(n^2)$ Laufzeit benötigen.

Zwei grundlegende Arten, wie sich Randomisierung auswirken kann:

Las Vegas: Problem wird immer (exakt, approximativ) gelöst. Laufzeit nicht immer polynomiell, nur mit einer gewissen Wahrscheinlichkeit oder im Erwartungswert

Monte Carlo: Laufzeit immer polynomiell. Problem wird nicht immer (exakt, approximativ) gelöst, nur mit einer gewissen Wahrscheinlichkeit oder im Erwartungswert.

Beispiele für Las-Vegas-Algorithmen: Quicksort (A wird immer exakt sortiert, Laufzeit im Erwartungswert) oder Seidels LP-Algorithmus (immer optimale LP Lösung, Laufzeit im Erwartungswert).

5.1 Minimaler Schnitt

Das Problem des **minimalen Schnittes** ist definiert analog zum s-t-Schnitt, allerdings ist G ungerichtet, und es werden nur nicht-leere Partitionen S, T gefordert (anstatt $s \in S$ und $t \in T$).

Algorithmus 21: Kontraktionsalgorithmus von Karger für minimalen Schnitt

```
1 Setze S(v) ← {v} für alle v ∈ V
2 Setze H = (V_H, E_H) ← G
3 while |V_H| > 2 do
4 |   Wähle e = {v, w} ∈ E_H zufällig mit Wahrscheinlichkeit c(e)/ Σ_{e'∈E_H} c(e')
5 |   Setze S(v) ← S(v) ∪ S(w)
6 |   Kontrahiere Kante {v, w} in H
7 Sei v einer der zwei verbleibenden Knoten in H
8 return S(v)
```

- $G = (V, E)$ ungerichteter Graph mit Kantenkosten $c : E \to \mathbb{N}_{\geq 0}$.
- Wir repräsentieren einen **Schnitt** durch eine echte Teilmenge $\emptyset \neq S \subset V$. Daraus resultiert eine Partition von V in zwei Mengen S und $T = V \setminus S$, so dass $S, T \neq \emptyset$
- Kantenmenge im Schnitt $E_S = \{\{u, v\} \in E \mid u \in S, v \notin S\}$
- **Kosten des Schnittes** $c(S) = \sum_{e \in E_S} c(e)$
- **Ziel:** Finde einen Schnitt mit minimalen Kosten.

Man kann einen minimalen Schnitt leicht in polynomieller Zeit finden. Berechne den minimalen s-t-Schnitt, für jedes Paar $s, t \in V$. Schlauer: Fixiere *einen beliebigen Knoten* $s \in S$ und finde den minimalen s-t-Schnitt, für jeden anderen Knoten $t \neq s$. Laufzeit mit, z.B., $(n-1)$-Mal Preflow-Push (siehe Theorem 13) ist $O(n^4)$.

Kargers randomisierter **Kontraktionsalgorithmus** (Algorithmus 21) ist oft schneller. Die Idee:

- Beim minimalen Schnitt sind "teuere" Kanten in den Partitionen und "billige" im Schnitt
- Wähle zufällig eine Kante gemäß der Kosten und verschmelze die Knoten
- Dann werden häufig nur Kanten innerhalb der optimalen Partitionen verschmolzen.

Verschmelzung/Kontraktion einer Kante $\{v, w\}$ im Graphen $H = (V_H, E_H)$:

- Für alle Kanten $\{u, w\}$, für die $u \neq v$ und $\{u, v\} \notin E_H$, wird eine Kante $\{u, v\}$ mit gleichen Kosten $c(\{u, v\}) = c(\{u, w\})$ hinzugefügt.
- Für alle Kanten $\{u, w\}$, für die $u \neq v$ und $\{u, v\} \in E_H$, werden die Kantenkosten von $\{u, v\}$ erhöht: $c(\{u, v\}) = c(\{u, v\}) + c(\{u, w\})$.
- Danach werden w aus V_H und alle seine inzidenten Kanten aus E_H entfernt.
- Eine Kantenkontraktion kann in Zeit $O(n)$ ausgeführt werden.
- Gesamtlaufzeit des Algorithmus ist $O(n^2)$

[Pic: Kantenkontraktion]

Theorem 73. *Der Kontraktionsalgorithmus ist ein Monte-Carlo-Algorithmus mit Erfolgswahrscheinlichkeit mindestens* $\binom{n}{2}^{-1}$. *Nach* $t \cdot \binom{n}{2}$ *Wiederholungen ist der beste berechnete Schnitt ein minimaler Schnitt mit Wahrscheinlichkeit mindestens* $1 - (1/\mathsf{e})^t$.

Beweis: Definitionen:

- e_i die i-te kontrahierte Kante e_i, für $i = 1, 2, \ldots, n-2$

- $H_0 = G$ und H_i der Graph nach der i-ten Kontraktion.
- c_i die Kantenkosten in Graphen H_i
- S^* minimaler Schnitt, S Ausgabe des Algorithmus

Betrachten wir nun die Erfolgswahrscheinlichkeit:

$$\begin{aligned}
\Pr[S \text{ ist min. Schnitt}] \geq \Pr[S = S^*] &= \Pr\left[\bigcap_{i=1}^{k} e_i \notin E_{S^*}\right] \\
&= \prod_{i=1}^{n-2} \Pr\left[e_i \notin E_{S^*} \;\middle|\; \bigcap_{j=1}^{i-1} e_j \notin E_{S^*}\right] \\
&\geq \prod_{i=1}^{n-2} \left(1 - \frac{2}{n-i+1}\right) \qquad \text{(Lemma 74 unten)} \\
&= \prod_{i=1}^{n-2} \frac{n-i-1}{n-i+1} \\
&= \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdots \frac{2}{4} \cdot \frac{1}{3} \\
&= \frac{2}{n(n-1)} = \binom{n}{2}^{-1}
\end{aligned}$$

Bei $t \cdot \binom{n}{2}$ Wiederholungen beträgt die **Fehlerwahrscheinlichkeit** (dass wir **keinen minimalen Schnitt** gefunden haben) höchstens

$$\left(1 - \binom{n}{2}^{-1}\right)^{t \cdot \binom{n}{2}} \leq \left(\frac{1}{\mathsf{e}}\right)^{t}$$

Durch viele Wiederholungen können wir den Fehler reduzieren. $O(n^2)$ Wiederholungen sind ausreichend für konstante Erfolgswahrscheinlichkeit (Gesamtlaufzeit dann $O(n^4)$). Danach nimmt die Fehlerwahrscheinlichkeit exponentiell in t ab. □

Wir beweisen nun noch die Hilfsaussage:

Lemma 74. *Es gilt*

$$\Pr\left[e_i \notin E_{S^*} \;\middle|\; \bigcap_{j=1}^{i-1} e_j \notin E_{S^*}\right] \quad \geq \quad 1 - \frac{2}{n-i+1} \ .$$

Beweis: Wir zeigen die Aussage für die Gegenwahrscheinlichkeit.

- Jeder Schnitt in H_{i-1} entspricht auch einem Schnitt in G.
- Nehmen wir an, dass keine vorherige Kante $e_j \in E_{S^*}$ und betrachten Runde i.
- Sei $k = \sum_{e \in E_{S^*}} c_{i-1}(e)$ und $n_{i-1} = |V_{H_{i-1}}|$ sowie $m_{i-1} = \sum_{e \in E_{H_{i-1}}} c_{i-1}(e)$
- Sei $\deg_{i-1}(u) = \sum_{\{u,v\} \in E_{H_{i-1}}} c_{i-1}(u)$ der gewichtete Knotengrad von Knoten u
- Abtrennen eines einzelnen Knotens ist möglicher Schnitt. Also sind die Gesamtkosten im Schnitt höchstens der mininmale Knotengrad: $k \leq \min_{u \in V_{H_{i-1}}} \deg_{i-1}(v)$

Algorithmus 22: FastCut(G) von Karger und Stein

```
1 Falls |V| ≤ 6 berechne optimalen Schnitt S, return S
2 Setze t ← ⌈1 + n/√2⌉
3 Setze H = H' = G
4 Kontrahiere n − t Kanten zufällig in H
5 Kontrahiere n − t Kanten zufällig in H'
6 Rekursive Aufrufe:
7   S_H = FastCut(H)
8   S_H' = FastCut(H')
9 return kleineren Schnitt S_H oder S_H' in G
```

- $n_{i-1} = n - i + 1$
- $m_{i-1} = \frac{1}{2} \sum_{v \in V_{H_{i-1}}} \deg_{i-1}(v) \geq \frac{1}{2} \cdot n_{i-1} \cdot k$

Da noch keine Kante $e \in E_{S^*}$ in $E_{H_{i-1}}$ kontrahiert ist, gilt

$$\Pr\left[e_i \in E_{S^*} \;\middle|\; \bigcap_{j=1}^{i-1} e_j \notin E_{S^*} \right] = \frac{k}{m_{i-1}} \leq \frac{2}{n_{i-1}} = \frac{2}{n-i+1} \quad . \quad \square$$

Wir betrachten eine verbesserte Version. Es gibt $\Theta(2^n)$ mögliche Schnitte im Graphen, aber nur $O(n^2)$ viele Kanten. Für ganz viele Schnitte könnten wir mit den gleichen ersten Kontraktionsschritten anfangen. Anstatt einfach nur sehr viele Wiederholungen von Algorithmus 21 zu machen (und in den ersten Schritten dann oft die gleichen Kanten zu kontrahieren), fassen wir in zwei Durchläufen $n - t$ Schritte zusammen. Dann rufen wir das Verfahren rekursiv auf, um zu verzweigen, um mit den verbleibenden Kanten unterschiedliche Durchläufe (und Schnitte) zu erzeugen.

Im FastCut-Algorithmus (Algorithmus 22) wird diese Idee umgesetzt. Für einen Graphen $G = (V, E)$ mit n Knoten werden in zwei Kopien jeweils nacheinander zufällige Kontraktionen gemacht bis jeweils nur noch $t \approx n/\sqrt{2}$ Knoten übrig sind. Danach wird rekursiv mit dem Verfahren ein Schnitt in den Kopien berechnet. Der günstigere der beiden resultierenden Schnitte wird (als unkontrahierte Knotenmenge von V) ausgegeben.

Lemma 75. *FastCut hat eine Laufzeit von $O(n^2 \log n)$.*

Beweisskizze: Die Laufzeit ist gegeben durch die Rekursionsgleichung

$$T(n) = 2 \cdot T(\lceil 1 + n/\sqrt{2} \rceil) + O(n^2)$$

für $n > 6$ und $T(n) = O(1)$ für $n \leq 6$.

- Rekursionstiefe ist $O(\log n)$, denn die Problemgröße sinkt konstanten Faktor in jedem Schritt.
- Sei n_ℓ die Anzahl verbleibender Knoten auf Tiefe ℓ
- Zwei rekursive Aufrufe $\Rightarrow$ Auf Tiefe ℓ sind 2^ℓ Teilprobleme zu lösen.

- Jedes Teilproblem hat eine Größe von $n_\ell = \Theta(n/(\sqrt{2})^\ell)$
- Laufzeit $O(n_\ell^2)$ für $\Theta(n_\ell)$ Kontraktionen in einem Graphen mit $\Theta(n_\ell)$ Knoten.
- Gesamter Zeitaufwand für Teilprobleme auf Tiefe ℓ ist

$$2^\ell \cdot O((n/(\sqrt{2})^\ell)^2) = 2^\ell \cdot O(n^2/((\sqrt{2}^2)^\ell)) = O(n^2)$$

- Jede der $O(\log n)$ Ebenen verursacht $O(n^2)$ Laufzeit, also Gesamtlaufzeit $O(n^2 \log n)$.

□

Theorem 76. *Die Wahrscheinlichkeit, dass FastCut einen minimalen Schnitt berechnet, beträgt mindestens* $\Omega(1/\log n)$.

Beweis: Für die Erfolgswahrscheinlichkeit sei S^* ein minimaler Schnitt. Wie im Beweis von Theorem 73 oben beschränken wir die Wahrscheinlichkeit, dass S^* eine Kontraktion von $n-t$ Kanten "überlebt", d.h. keine Kante aus E_{S^*} gewählt wird, durch mindestens

$$\begin{aligned}\prod_{i=1}^{n-t}\left(1-\frac{2}{n-i+1}\right) &= \frac{n-2}{n}\cdot\frac{n-1}{n-3}\cdots\frac{t}{t+2}\cdot\frac{t-1}{t+1} \\ &= \frac{t(t-1)}{n(n-1)} \quad = \quad \frac{\lceil 1+n/\sqrt{2}\rceil\lceil n/\sqrt{2}\rceil}{n(n-1)} \quad \geq \quad \frac{1}{2}\ .\end{aligned}$$

Wie ergibt sich nun die Erfolgswahrscheinlichkeit über die rekursiven Aufrufe?

- FastCut(G) erfolgreich wenn er minimalen Schnitt von G berechnet
- Seien $k = \sum_{e\in E_{S^*}} c(e)$ die optimalen Kosten. FastCut(G) erfolgreich, wenn
  (A) Minimaler Schnitt in H hat Kosten k, und FastCut(H) erfolgreich. **ODER**
  (B) Minimaler Schnitt in H' hat Kosten k, und FastCut(H') erfolgreich.
- Seien $p(H), p(H')$ die Erfolgswahrscheinlichkeiten für FastCut auf H und H'.
- Wahrscheinlichkeit für Event A mindestens $\frac{1}{2}\cdot p(H)$, Gegenw.keit höchstens: $1-p(H)/2$
- FastCut(G) erfolgreich, wenn mindestens ein Event A oder B eintritt, also W.keit mindestens:

$$p(G) \geq 1-\left(1-\frac{p(H)}{2}\right)\cdot\left(1-\frac{p(H)}{2}\right)$$

- Auf Höhe h des Rekursionsbaumes gilt für die Erfolgswahrscheinlichkeit:

$$p(h) \geq 1-\left(1-\frac{p(h-1)}{2}\right)^2$$

- Sei $d \in O(\log n)$ die Tiefe des Rekursionsbaumes (also die Höhe der Wurzel).
- An den Blättern (also wenn nicht mehr rekursiv aufgerufen wird) gilt $p(0) = 1$.
- Wir zeigen induktiv, dass $p(h) \geq \frac{1}{h+1}$ für alle $h = 0, 1, \ldots, d$,
- Start: $p(0) = 1 \geq 1/(0+1)$. Annahme: $p(i) \geq 1/(i+1)$ für alle $i = 0, 1, \ldots, h-1$. Schritt:

$$\begin{aligned}p(h) &\geq 1-\left(1-\frac{p(h-1)}{2}\right)^2 = p(h-1)-\frac{p(h-1)^2}{4} \\ &\geq \frac{1}{h}-\frac{1}{4h^2} = \frac{4h-1}{4h^2}\end{aligned}$$

$$
\begin{aligned}
&= \frac{(4h-1)(h+1)}{4h^2} \cdot \frac{1}{h+1} \\
&= \frac{4h^2+3h-1}{4h^2} \cdot \frac{1}{h+1} \\
&\geq \frac{1}{h+1} \ .
\end{aligned}
$$

- Damit gilt für die Wurzel $p(G) = p(d) \geq \frac{1}{d+1} = \Omega(1/\log n)$ und das Theorem ist gezeigt. □

Wie in Theorem 73 erhöht sich die Erfolgswahrscheinlichkeit durch Wiederholung. Hier sind aber nur $t \cdot O(\log n)$ Wiederholungen nötig, um eine Erfolgswahrscheinlichkeit von $1-(1/e)^t$ zu erreichen. Konstante Erfolgswahrscheinlichkeit schon nach $O(\log n)$ Wiederholungen (Gesamtlaufzeit dann nur $O(n^2 \log^2 n)$).

5.2 2-SAT und 3-SAT

Random Walks sind eine wichtige Analyse- und Entwurfstechnik für ganz unterschiedliche (verteilte) Problemstellungen, z.B. für das Schätzen der Anzahl unterschiedlicher Lösungen, die Erzeugung von zufälligen Elementen, das schnelle Testen von Graph-Eigenschaften, Lastbalancierung, usw.

Wir betrachten hier einen Ansatz für die Lösung von 2-SAT und 3-SAT Instanzen. Die Algorithmen kann man als lokale Suche auffassen, für die Analyse nutzen wir Random Walks.

2-SAT

Als einführendes Szenario analysieren wir das 2-SAT Problem. Es gibt einen deterministischen Polynomzeitalgorithmus zur Lösung von 2-SAT. Der folgende randomisierte Algorithmus und die Analyse dienen als Vorbereitung für das 3-SAT Problem unten.

Wir definieren allgemein das **k-SAT Problem**:

- n aussagenlogische Variablen $x_1, \ldots, x_n$
- m Klauseln, jeweils mit (höchstens) k Literalen $C_j = (\ell_{j,1} \vee \ldots \vee \ell_{j,k})$, wobei $\ell_{j,t} \in \{x_i, \bar{x_i} \mid i = 1, \ldots, n\}$ für $t = 1, \ldots, k$ (o.B.d.A. immer genau k Literale durch Duplizieren)
- Klauseln werden kombiniert zu einer KNF-Formel $(C_1 \wedge C_2 \wedge \ldots \wedge C_m)$
- Entscheide, ob die Formel erfüllbar ist – d.h. es eine Belegung $B : \{x_1, \ldots, x_n\} \to \{0, 1\}$ der Variablen gibt, so dass die Formel erfüllt wird.
- Beispietl 2-SAT: $((x_1 \vee x_2) \wedge (\bar{x_1} \vee \bar{x_3}) \wedge (x_1 \vee \bar{x_2})) \to$ ist erfüllbar.

Betrachte die lokale Suche in Algorithmus 23. Wenn die Formel nicht erfüllbar ist, gibt der Algorithmus *immer die korrekte Lösung aus*!

Wir beschränken unsere Analyse nur auf erfüllbare Formeln. Dafür benutzen wir einen Random Walk (allgemeiner: eine *Markov-Kette*), der Zustände der Belegung und Übergangswahrscheinlichkeiten durch die Änderungen des Algorithmus beschreibt/beschränkt. Wir schätzen

Algorithmus 23: Lokale Suche für k-SAT

```
1 Wähle eine Belegung B der Variablen uniform zufällig
2 for τ Iterationen do
3     if B erfüllt die Formel then return "erfüllbar" (und B)
4     Sei C_j eine Klausel, die von B nicht erfüllt wird
5     Wähle uniform zufällig eines der Literale ℓ_{j,t}
6     Invertiere in B die Belegung der Variable x_i des gewählten Literals
7 return "nicht erfüllbar"
```

damit die Wahrscheinlichkeit ab, dass der Algorithmus Fortschritt zur korrekten Entscheidung macht.

Zustände:

- Sei S eine erfüllende Belegung.
- Zustände $\{0, 1, \ldots, n\}$ sagen, **wie viele Variablen in B und S gleich belegt** sind
- "Erfolgszustand" n: Dann ist $B = S$, erfüllende Belegung gefunden.
- In allen anderen Zuständen nehmen wir an, dass B nicht erfüllend sei. Damit unterschätzen wir die Erfolgswahrscheinlichkeit des Algorithmus.

Übergangswahrscheinlichkeiten:

- Sei $p_{r,s}$ die Wahrscheinlichkeit, dass in einem Schritt des Algorithmus die Belegung von Zustand r zu s übergeht.
- Es wird genau eine Variable geändert, daher kann $p_{r,s} > 0$ nur für $s = r - 1$ oder $s = r + 1$.
- Offensichtlich: $p_{0,1} = 1$.
- Der Algorithmus terminiert in Zustand n. Daher $p_{n,n-1} = 0$. Wir setzen $p_{n,n} = 1$.
- Im Zustand $r \in \{1, \ldots, n-1\}$ wird eine Variable aus einer unerfüllten Klausel C_j geflippt.
- S erfüllt C_j, d.h. S und B unterscheiden sich in mind. einer Variablen x_i aus C_j (evtl. in mehreren)
- Bei 2-SAT: Algorithmus wählt und flippt x_i mit Wahrscheinlichkeit $1/2$. Dann erhöht sich der Zustand um 1. Sonst flippt er die andere Variable. Dann kann sich der Zustand um 1 erhöhen oder verringern.
- Insgesamt gilt also
$$p_{r,r+1} \;\geq\; 1/2 \;\geq\; p_{r,r-1}.$$
- Wir unterschätzen den Fortschritt des Algorithmus und setzen im Folgenden
$$p_{r,r+1} = p_{r,r-1} = 1/2.$$

[Pic: Markov-Kette als "Pfad" über Integers]

Der Algorithmus verhält sich also (besser als) ein Random Walk auf den Integers $\{0, 1, \ldots, n\}$. Wieviele Iterationen werden im Erwartungswert benötigt, bis er im Zustand n angekommen ist?

Lemma 77. *Für jede Anfangsbelegung B ist die erwartete Anzahl Iterationen, bis Algorithmus 23 eine erfüllende Belegung gefunden hat, höchstens n^2.*

Beweis: Sei h_r die **erwartete Anzahl Schritte**, dass n erreicht wird wenn wir mit der Belegung B in Zustand r starten. Dann gilt:

- $h_n = 0$ (wir sind da) und $h_0 = 1 + h_1$ (von 0 gehts immer nach 1). Für $r = 1, \ldots, n-1$ gehts nach einem Schritt weiter in $r-1$ oder $r+1$, jeweils mit Wkeit 1/2:

$$h_r = 1 + \frac{1}{2} \cdot h_{r-1} + \frac{1}{2} \cdot h_{r+1} \tag{5.1}$$

- Wir zeigen induktiv: $h_r = h_{r+1} + 2r + 1$ für alle $r = 0, \ldots, n-1$.
- Anfang: $h_0 = h_1 + 2 \cdot 0 + 1$.
- Schritt: Stelle (5.1) nach h_{r+1} um und setze Annahme für h_{r-1} ein:
  $h_{r+1} = 2h_r - 2 - h_{r-1} = 2h_r - 2 - (h_r + 2r - 2 + 1) = h_r - 2r - 1$
  Umstellen nach h_r liefert die Aussage.

Für Zustand 0 ergibt sich die größte erwartete Anzahl:

$$\begin{aligned} h_0 &= h_1 + 1 = h_2 + 3 + 1 = \ldots \\ &= 0 + (2n-1) + (2n-3) + \ldots + 5 + 3 + 1 \\ &= \sum_{r=0}^{n-1} (2r+1) \\ &= 2 \cdot \frac{(n-1)n}{2} + n \\ &= n^2 \qquad \square \end{aligned}$$

Lemma 78. *Für eine erfüllbare 2-SAT Formel erzielt die lokale Suche eine konstante Erfolgswahrscheinlichkeit nach $O(n^2)$ Iterationen.*

Beweis: Sei Z die Zufallsvariable, die die Anzahl Iterationen beschreibt, bis wir eine erfüllende Belegung gefunden haben. Es gilt $Z \in \mathbb{N}_0$ und mit dem obigen Lemma $\mathbb{E}[Z] \leq n^2$. Die Markov-Ungleichung ist anwendbar und liefert

$$\Pr[Z \geq 2n^2] \leq \frac{\mathbb{E}[Z]}{2n^2} \leq \frac{1}{2}$$

Mit $\tau = 2n^2$ ist die Erfolgswahrscheinlichkeit also mindestens 1/2. Mit $t \cdot 2n^2$ Wiederholungen erhalten wir eine Erfolgswahrscheinlichkeit von mindestens $1 - (1/2)^t$. $\square$

3-SAT

3-SAT ist NP-vollständig. Wir können nicht davon ausgehen, dass es einen Monte-Carlo Polynomialzeitalgorithmus mit konstanter Erfolgswahrscheinlichkeit (oder z.B. Las-Vegas Algorithmus mit erwartet polynomieller Laufzeit) gibt. Ziel daher: Möglichst gute exponentielle Laufzeit.

Wenden wir zuerst Algorithmus 23 für 3-SAT an. Die Analyse verläuft sehr ähnlich, aber es gibt andere Übergangswahrscheinlichkeiten.

Algorithmus 24: Schönings Algorithmus für k-SAT

```
for κ Wiederholungen do
    Starte eine neue lokale Suche (Algorithmus 23) mit τ = 3n Iterationen
    if erfüllende Belegung B gefunden then return "erfüllbar" (und B)
return "nicht erfüllbar"
```

- Es gilt weiterhin, dass nur $p_{r,r+1}$ und $p_{r,r-1}$ positiv sein können.
- Natürlich $p_{0,1} = 1$ und $p_{n,n-1} = 0$, und wir setzen wieder $p_{n,n} = 1$.
- Die Wahrscheinlichkeit beträgt nur 1/3, dass die (u.U. einzige) in S und B unterschiedlich belegte Variable x_i in C_j vom Algorithmus gewählt wird.
- Ansonsten wird evtl. eine der beiden anderen (u.U. in S und B gleich belegten) Variablen gewählt und der Zustand dadurch erhöht oder (u.U. immer) verringert.
- Damit gilt also
$$p_{r,r+1} \geq 1/3 \qquad p_{r,r-1} \leq 2/3,$$
und zur Unterschätzung des Fortschritts des Algorithmus ergibt sich lediglich
$$p_{r,r+1} = 1/3 \qquad \text{und} \qquad p_{r,r-1} = 2/3.$$
- Sei h_j die erwartete Anzahl Iterationen wie oben. Dann ergibt sich $h_n = 0$, $h_0 = 1 + h_1$ (wie oben) und hier
$$h_r = 1 + \frac{1}{3} \cdot h_{r+1} + \frac{2}{3} \cdot h_{r-1}$$
- Induktion für Lemma 77 kann man durchführen und erhält $h_r = h_{r+1} + 2^{r+1} - 3$.
- Daraus folgt
$$\begin{aligned} h_0 &= h_1 + 2^2 - 3 = h_2 + 2^3 + 2^2 - 2 \cdot 3 = \ldots \\ &= h_n + 2^{n+1} + 2^n + \ldots + 2^2 - 3n \\ &= 2^{n+2} - 1 - 3(n+1) \end{aligned}$$
- Ok, da können wir auch gleich **alle** 2^n **Belegungen direkt untersuchen**!
- Der Random Walk hat einen Bias in die falsche Richtung. Je länger der Algorithmus dauert, desto unwahrscheinlicher sind gute Ergebnisse.

In **Schönings Algorithmus** (Algorithmus 24) wird eine relativ kurze lokale Suche ganz oft neu gestartet. Offensichtlich ist der Algorithmus für nicht-erfüllbare Formeln weiterhin immer korrekt. Für erfüllbare Formeln erhalten wir folgende Garantie:

Theorem 79. *Für eine erfüllbare 3-SAT Formel erzielt Schönings Algorithmus eine konstante Erfolgswahrscheinlichkeit in einer Laufzeit von $O(m \cdot n^{3/2} \cdot (4/3)^n)$.*

Beweis (Skizze): Wir betrachten die Wahrscheinlichkeit q_s, in einem Aufruf der lokalen Suche von Zustand $r = n - s$ zu Zustand n zu kommen.

- Eine Sequenz von Schritten von Zustand r nach n ist eine Folge von Anstiegen ("+") oder Abstiegen ("-")

- Wenn sie am Ende bei n ankommt, muss sie genau s mehr + als - enthalten.
- Die Sequenz hat $s+2i$ Schritte ($s+i$ viele + und i viele -), für ein $i = 0, 1, 2, \ldots$
- Man kann zeigen, dass es genau $\binom{s+2i}{i} \cdot \frac{s}{s+2i}$ viele solcher Sequenzen gibt, für jedes $i = 0, 1, ...$
- Sei $q_{s,i}$ die Wahrscheinlichkeit, dass eine solche Sequenz auftritt. Dann gilt

$$q_{s,i} \geq \binom{s+2i}{i} \cdot \frac{j}{j+2i} \cdot \left(\frac{1}{3}\right)^{j+i} \cdot \left(\frac{2}{3}\right)^{i}$$

- Für Zustand 0 mit $s = n$ muss gelten, dass $s+2i = n+2i \leq 3n$, da wir nur $3n$ Schritte in der lokalen Suche machen. Also $i \leq n = s$.
- Wir beschränken allgemein auf $i \leq s$. Solche Sequenzen sind in der lokalen Suche möglich, da $s + 2i \leq 3s \leq 3n$.
- Mit geschickter Abschätzung kann man zeigen:

$$q_s \geq \sum_{i=0}^{s} q_{s,i} \;\geq\; \frac{c}{\sqrt{s}} \cdot \frac{1}{2^s},$$

  für eine Konstante c (mit $c \approx \frac{1}{2\sqrt{3\pi}}$).

Die Wahrscheinlichkeit p_s, dass wir anfangs in Zustand $r = n - s$ sind und genau s Variablen unterschiedlich von S gewählt werden, beträgt

$$p_s = \binom{n}{s} \cdot \frac{1}{2^n}$$

Die Wahrscheinlichkeit $\tilde{p}$, dass Zustand n erreicht wird in einem Aufruf der lokalen Suche, ist also

$$\begin{aligned}
\tilde{p} &\geq \sum_{s=0}^{n} p_s \cdot q_s \\
&\geq \sum_{s=0}^{n} \binom{n}{s} \cdot \frac{1}{2^n} \cdot \frac{c}{\sqrt{s}} \cdot \frac{1}{2^s} \\
&\geq \frac{c}{\sqrt{n}} \cdot \frac{1}{2^n} \cdot \sum_{s=0}^{n} \binom{n}{s} \cdot \frac{1}{2^s} \\
&= \frac{c}{\sqrt{n}} \cdot \frac{1}{2^n} \cdot \left(1 + \frac{1}{2}\right)^n \\
&= \frac{c}{\sqrt{n}} \cdot \left(\frac{3}{4}\right)^n.
\end{aligned}$$

Bei $\kappa = 1/\tilde{p} = O(\sqrt{n} \cdot (4/3)^n)$ Wiederholungen der lokalen Suche wird die Fehlerwahrscheinlichkeit höchstens konstant, da $(1-\tilde{p})^\kappa \leq (1/\mathsf{e})^{\tilde{p}\kappa} = (1/\mathsf{e})$. Damit erzielen wir Erfolgswahrscheinlichkeit mindestens $1 - 1/\mathsf{e}$.

Für die Laufzeitschranke beobachten wir:

- $O(\sqrt{n} \cdot (4/3)^n)$ Wiederholungen der lokalen Suche
- Jede Wiederholung $\tau = 3n$ Iterationen
- Laufzeit einer Iteration dominiert durch $O(m)$ zum Finden einer unerfüllten Klausel.

□

Kapitel 6

Online Algorithmen

Viele Rechnersysteme müssen sich adaptiv auf wechlsende Umgebungen einstellen, z.B. wenn nacheinander Anfragen gestellt werden, sich Datensätze ändern, gelöscht oder hinzugefügt werden, Tasks auf Maschinen gestartet oder beendet werden, etc. Die Herausforderung besteht dabei in der *Optimierung mit beschränktem Wissen über die Zukunft*, also welche Anfragen oder Anpassungen in der Zukunft eintreten werden und wie man bestmöglich darauf reagiert. Solche Problemstellungen heißen *Online-Probleme*, und wir nutzen *Online-Algorithmen* zur ihrer Lösung. Zur Bewertung vergleichen wir sie mit dem *Offline-Optimum*, d.h. der bestmöglichen Lösung mit Wissen über die Zukunft.

6.1 Ski-Rental

Ein elementares Beispiel für ein Online-Problem ist **Ski-Rental**:

- Sequenz σ von insgesamt n Tagen, n bekannt. Skifahren wann immer möglich
- Jeden Tag wird morgens bekannt, ob Skifahren heute möglich ist (Ski-Tag) oder nicht
- Jeden Tag kann man Equipment entweder mieten oder kaufen
- Miete beträgt (der Einfachheit halber) 1 EUR pro Tag, Kaufkosten B EUR, $B > 1$.
- Sobald man gekauft hat, braucht man nicht mehr mieten

Ziel: Minimiere die Gesamtkosten aus Miete plus Kaufkosten

Zur Bewertung von Online-Algorithmen nutzt man oft den **Wettbewerbsfaktor (engl: competitive ratio)**, analog zum Approximationsfaktor in Definition 42. Er gibt an, wieviel ein Algorithmus im Worst Case an Wert verliert gegenüber einem **(Offline-)Optimum mit vollständiger Kenntnis über die Zukunft**.

Eine Instanz eines Problems wird charakterisiert durch eine Eingabesequenz σ von Events. Algorithmus ALG lernt σ nur iterativ nacheinander kennen. ALG muss nach jedem Event unmittelbar eine (Teil-)Entscheidung treffen. Sei $ALG(\sigma)$ der Zielfunktionswert von ALG bei Eingabe σ, $OPT(\sigma)$ der Zielfunktionswert einer optimalen Lösung mit anfänglich voller Kenntnis von σ und $\alpha > 1$ eine Zahl.

Definition 80. Für ein Minimierungsproblem ist ALG **α-kompetitiv** oder hat **Wettbewerbsfaktor** α wenn für jede Eingabe σ des Problems gilt

$$ALG(\sigma) \leq \alpha \cdot OPT(\sigma) + b$$

für eine Konstante $b \geq 0$ unabhängig von der Eingabe. Für ein Maximierungsproblem ist ALG α-kompetitiv wenn

$$OPT(\sigma) \leq \alpha \cdot ALG(\sigma) + b$$

ALG heißt **strikt α-kompetitiv** wenn $b = 0$.

Für Ski-Rental untersuchen wir den **Greedy Algorithmus**:
Miete bis es $B - 1$ Ski-Tage gab. Am B-ten Ski-Tag kaufe das Equipment.

Theorem 81. *Der Greedy-Algorithmus ist $(2 - 1/B)$-kompetitiv.*

Beweis: Seien $G(\sigma)$ die Kosten von Greedy. Fallanalyse der (unbekannten) Anzahl Ski-Tage:

- σ hat höchstens $B - 1$ Ski-Tage: Greedy mietet nur, Optimum mietet nur. Greedy optimal: $G(\sigma) = OPT(\sigma)$
- σ hat mindestens B Ski-Tage: Greedy mietet $B - 1$ Tage und kauft, $G(\sigma) = 2B - 1$. Optimum kauft sofort, $OPT(\sigma) = B$. Also:

$$G(\sigma)) = 2B - 1 = \left(\frac{2B-1}{B}\right) \cdot B = \left(2 - \frac{1}{B}\right) \cdot OPT(\sigma). \quad \square$$

Tatsächlich kann kein deterministischer Algorithmus eine bessere Garantie erzielen

Theorem 82. *Kein deterministischer Algorithmus für Ski-Rental ist strikt α-kompetitiv mit $\alpha < 2 - 1/B$.*

Beweis: Sei ALG ein beliebiger deterministisher Algorithmus. ALG muss nicht mehr mieten nach einem Kauf, also wird erstmal gemietet dann evtl. gekauft. Wir beschränken uns auf Instanzen mit $n = 2B$. Sei $0 \leq \ell \leq n$ der letzte Ski-Tag, an dem ALG mietet.

[Pic: Kosten des Algorithmus, optimale Kosten]

Fall $\ell = n$:

- Betrachte eine Sequenz σ mit n Ski-Tagen.
- ALG kauft nie und hat Kosten $ALG(\sigma) = n = 2B$.
- Optimum kauft sofort, $OPT(\sigma) = B$.
- Wettbewerbsfaktor also

$$\frac{ALG(\sigma)}{OPT(\sigma)} = \frac{2B}{B} = 2.$$

Fall $\ell < n$:

- Betrachte eine Sequenz σ mit $\ell + 1$ Ski-Tagen.
- ALG mietet bis Tag ℓ, danach Kauf: $ALG(\sigma) = \ell + B$.
- Optimum wählt Minimum aus "immer mieten" und "sofort kaufen": $OPT(\sigma) = \min\{\ell + 1, B\}$.
- Wettbewerbsfaktor also mindestens

$$\frac{ALG(\sigma)}{OPT(\sigma)} = \frac{\ell + B}{\min\{\ell + 1, B\}} = \max\left(\frac{\ell + B}{\ell + 1}, \frac{\ell + B}{B}\right) \geq \frac{2B - 1}{B} = 2 - \frac{1}{B} \ .$$

Beachte: Im Maximum-Ausdruck ist der erste Term fallend in ℓ, der zweite wachsend in ℓ. Das Maximum wird also minimiert bei Gleichheit, d.h. wenn $\ell + 1 = B$.

Es gibt also für jeden Algorithmus ALG eine Eingabesequenz, so dass die Kosten von ALG und dem Optimum sich mindestens um Faktor $2 - 1/B$ unterscheiden. □

Können *randomisierte Algorithmen* diese Schranke schlagen? Wir untersuchen kurz eine einfache Randomisierung. Der Algorithmus erzielt **erwartete Kosten** für jede Eingabe σ. Entsprechend wird die Definition von "(strikt) α-kompetitiv" auf die erwarteten Kosten von ALG angewendet. Für *jede Eingabesequenz* σ muss gelten

$$\mathbb{E}[ALG(\sigma)] \leq \alpha \cdot OPT(\sigma) + b \qquad \text{für Minimierungsprobleme, und}$$

$$OPT(\sigma) \leq \alpha \cdot \mathbb{E}[ALG(\sigma)] + b \qquad \text{für Maximierungsprobleme}$$

Greedy ist optimal bei wenig Ski-Tagen. Ansonsten kauft er zu spät. Ein natürliche Idee ist daher, mit gewisser Wahrscheinlichkeit früher zu kaufen. Dann haben wir mehr Kosten bei wenig Ski-Tagen, aber weniger Kosten bei vielen Tagen. Insgesamt zahlt sich das aus.

Random-Greedy Algorithmus: Wähle uniform zufällig eine der zwei folgenden Optionen:
(1) Nutze Greedy Algorithmus.
(2) Miete die ersten $\frac{3}{5} \cdot B$ Ski-Tage, dann kaufe.

Theorem 83. *Random-Greedy ist strikt* 11/6*-kompetitiv.*

Beweis: Seien $RG(\sigma)$ die Kosten von Random-Greedy. Fallanalyse der (unbekannten) Anzahl Ski-Tage k:

Fall $k < \frac{3}{5}B$:
- Random-Greedy und Optimum mieten nur, $RG(\sigma) = OPT(\sigma) = k$.

Fall $k \geq B$:
- Mit W.keit 1/2 kauft Random-Greedy nach $\frac{3}{5}B$ Ski-Tagen.
- Mit W.keit 1/2 kauft er nach $B - 1$ Ski-Tagen.
- Erwartete Kosten $\mathbb{E}[RG(\sigma)] = \frac{1}{2}\left(\frac{3}{5} \cdot B + B\right) + \frac{1}{2}\left(B - 1 + B\right) < \frac{9}{5} \cdot B$.
- Optimale Kosten $OPT(\sigma) = B$. Wettbewerbsfaktor höchstens $9/5 = 1.8$.

Fall $\frac{3}{5}B \leq k < B$:
- Mit W.keit 1/2 kauft Random-Greedy nach $\frac{3}{5}B$ Ski-Tagen.
- Mit W.keit 1/2 mietet er nur.
- Erwartete Kosten $\mathbb{E}[RG(\sigma)] = \frac{1}{2}\left(\frac{3}{5} \cdot B + B\right) + \frac{1}{2} \cdot k = \frac{4}{5} \cdot B + \frac{k}{2}$.
- Optimale Kosten $OPT(\sigma) = k$. Wettbewerbsfaktor höchstens

$$\frac{\mathbb{E}[ALG(\sigma)]}{OPT(\sigma)} = \frac{4}{5} \cdot \frac{B}{k} + \frac{1}{2} \leq \frac{4}{5} \cdot \frac{5}{3} + \frac{1}{2} = \frac{11}{6} = 1.8\bar{3} \qquad \square$$

Bessere randomisierte Algorithmen nutzen schlauere Zufallsentscheidungen für den Kaufzeitpunkt. Durch sorgfältige Anpassung erhält man einen Wettbewerbsfaktor von $(1-1/\mathsf{e})^{-1} \approx 1.58$ und das ist bestmöglich.

Theorem 84. *Es gibt einen randomisierten Algorithmus für Ski-Rental, der strikt* $\frac{1}{\left(1-\frac{1}{\mathsf{e}}\right)}$*-kompetitiv ist, und kein randomisierter Algorithmus ist strikt* α*-kompetitiv mit* $\alpha < \frac{1}{\left(1-\frac{1}{\mathsf{e}}\right)}$*.*

6.2 Paging

Paging ist ein klassisches Problem bei der Verwaltung eines begrenzten schnellen Speichers.

- Schneller Cache-Speicher C der Größe k
- Größere, langsamere Speichermedien (z.B. Festplatte, weitere Level der Cache-Hierarchie)
- Programm fordert nacheinander eine Sequenz σ von Daten (sog. Seiten) an
- Ist die Seite in C gespeichert, entstehen quasi keine Kosten
- Sonst muss die Seite aufwändig aus großem Speicher geladen werden (Cache-Fehler)

Ziel: Wähle die in C zu speicherden Datensätze aus, um die Anzahl Cache-Fehler zu minimieren

Wir nehmen an, C und die ersten k unterschiedlichen angeforderten Seiten werden in C eingespeichert. Bei jeder weiteren Anfrage kann nach einem Cache-Fehler die angeforderte Seite nur gespeichert werden, wenn dafür eine Seite aus C entfernt wird. Dafür gibt es mögliche **Verdrängungsstrategien**. Hier einige Beispiele.

Entferne eine Seite aus dem aktuellen Cache C, ...
LFU (Least Frequently Used) ... die am seltensten angefragt wurde.
LRU (Least Recently Used) ... deren letzte Anfrage am längsten zurückliegt.
FIFO (First-In-First-Out) ... die schon am längsten in C liegt.
FWF (Flush When Full) Entleere C immer komplett wenn er voll ist und eine Seite ausserhalb von C angefragt wird.
LFD (Longest Forward Distance) ... deren nächster Zugriff am weitesten in der Zukunft liegt.

Bis auf LFD sind dies alles Strategien, die für den Einsatz als Online-Algorithmus geeignet sind. LFD ist dagegen eine Offline-Strategie mit Wissen über die Zukunft. Man kann zeigen:

Lemma 85. *LFD berechnet ein Offline-Optimum.*

Wir beobachten zuerst, dass z.B. LFU sehr schlechte Ergebnisse liefern kann.

Lemma 86. *Der Wettbewerbsfaktor von LFU ist unbeschränkt in* k*.*

Beweis: Betrachte eine Sequenz wie folgt. Sei $t \gg k$ beliebig groß.

- Es gibt insgesamt $k+1$ Seiten $p_1, \ldots, p_{k+1}$.
- In σ wird zuerst t-Mal p_1 angefragt, dann t-Mal p_2, ..., dann t-Mal p_{k-1}.
- Dadurch erhalten wir $k-1$ Cache-Fehler.
- Danach wird $(t-1)$-Mal abwechselnd p_k und p_{k+1} angefragt.

- LFU wird jedes Mal p_{k+1} durch p_k ersetzen und umgekehrt. Dadurch $\Omega(t)$ Cache-Fehler.
- Bei der ersten Anfrage von p_{k+1} wird LFD z.B. p_1 verdrängen. Dann sind beide p_k und p_{k+1} im Cache. Dadurch insgesamt nur $k+1$ Cache-Fehler.
- Wettbewerbsfaktor also $\Omega(t/k)$ □

[Pic: Schlechte Sequenz für LFU]

Man kann diese Idee leicht erweitern.

Lemma 87. *Kein deterministischer Algorithmus für Paging ist α-kompetitiv mit $\alpha < k$.*

Beweis: Betrachte ein System mit $k+1$ Seiten. Jeder deterministische Algorithmus ALG hat immer (maximal) k Seiten im Cache. Frage eine Seite an, die momentan nicht im Cache ist. Dadurch ergibt sich für ALG in jedem Schritt ein Cache-Fehler. LFD verdrängt die Seite, die am spätesten wieder angefragt wird, muss also höchstens alle k Schritte einen Fehler machen. □

Eine interessante Klasse von Algorithmen sind sog. Markierungsalgorithmen.
Dazu definieren wir zuerst eine **Phase** in σ:

- Phase 1 beginnt am Anfang. Phase i beginnt nach Ende von Phase $i-1$.
- Phase i endet zum spätesten Zeitpunkt, an dem nach dem Beginn von Phase i nur **höchstens k unterschiedliche Seiten** angefragt wurden.
- Die letzte Phase endet wenn σ endet.

Für einen **Markierungsalgorithmus** stellen wir uns Markierungen der Seiten vor:

- Zu Anfang einer Phase werden alle Markierungen gelöscht.
- Wenn eine Seite angefragt wird, wird sie markiert.

Ein Markierungsalgorithmus **verdrängt keine markierte Seite**!

[Pic: Schema Phase, Markierungsalgorithmus]

Beobachtungen:

- Das Beispiel in Lemma 86 zeigt, dass LFU kein Markierungsalgorithmus ist.
- FWF ist offensichtlich ein Markierungsalgorithmus. Er hält in jeder Phase immer genau die markierten Seiten im Cache.
- LRU ist ein Markierungsalgorithmus. Von den k Seiten im Cache wird immer eine verdrängt, deren letzte Anfrage am längsten zurück liegt. Das ist keine markierte Seite, denn dies sind immer die höchstens $k-1$ vielen Seiten mit den neuesten Anfragen.

Theorem 88. *Jeder Markierungsalgorithmus ist k-kompetitiv.*

Beweis: Wir analysieren die Cache-Fehler eines Markierungsalgorithmus ALG.

- Betrachte eine Phase i. Darin werden höchstens k unterschiedliche Seiten angefragt.
- Markierungsalgo: Keine in Phase i angefragte Seite wird verdrängt.
- ALG verdrängt nur unmarkierte Seiten, die zum Zeitpunkt der Verdrängung das letzte Mal vor Phase i angefragt wurden.
- Falls noch nicht vorhanden speichert er die markierten Seiten im Cache.
- ALG verdrängt also nur maximal k Seiten pro Phase. Maximal k Cache-Fehler pro Phase.

Sei t die Anzahl Phasen. Insgesamt also höchstens $k(t-1)+d$ Cache-Fehler, wobei d die Länge der letzten Phase.

Wir beschränken nun die Cache-Fehler des Optimums LFD.

- In Phase $i=1$ werden sowieso k Cache-Fehler produziert.
- Betrachte Phase $1 < i < t$.
- Wir nehmen an, in Phase $i-1$ hat LFD mindestens einen Cache-Fehler gemacht.
- Sei p die Seite, die den letzten Cache-Fehler in Phase $i-1$ erzeugt hat.
- In Phase i werden nun k verschiedene Seiten angefragt
- Damit dort kein Cache-Fehler passiert, müssen genau diese Seiten zu Beginn der Phase der Inhalt des Cache sein!
- Das bedeutet, p und alle nachfolgend markierten Seiten in Phase $i-1$ müssen eine der Seiten sein.
- Entweder ist dann Phase i noch Teil von Phase $i-1$ oder wir haben insgesamt zu viele Seiten.
- LFD muss einen Cache-Fehler in Phase i machen.

LFD erzeugt also mindestens $k+t-2 \geq t-1$ Cache-Fehler.

[Pic: Phase ohne Cache-Fehler]

Es gilt also $ALG(\sigma) \leq k \cdot OPT(\sigma) + d$, wobei $d \leq k$ die Länge der letzten Phase. □

Mit randomisierten Algorithmen geht es deutlich besser.

Wir betrachten den Markierungsalgrithmus **Random-Mark**: In jeder Runde mit Cache-Fehler wird eine Seite zur Verdrängung aus C *uniform zufällig aus den aktuell unmarkierten Seiten* ausgewählt.

Theorem 89. *Random-Mark ist $2H_k$-kompetitiv.*

Beweis (Skizze): In jeder Phase i unterscheiden wir **alte** und **frische** Seiten:
Alt: Unmarkierte Seite, auf die in Phase $i-1$ zugegriffen wurde.
Frisch: Unmarkierte Seite, auf die nicht in Phase $i-1$ zugegriffen wurde.
Sei f_i die Anzahl der frischen Seiten in Phase i.

Wir leiten eine obere Schranke auf die (erwartete) Anzahl Cache-Fehler von Random-Mark her.

- Phase 1 ist wieder geschenkt: OPT und Random-Mark machen k Cache-Fehler.
- Betrachte eine Phase $i \geq 2$. Da in Phase $i-1$ genau k unterschiedliche Seiten angefragt wurden (die Random-Mark nicht verdrängt hat), sind zu Beginn von Phase i genau diese Seiten im Cache $\Rightarrow$ Alle Seiten alt.
- Phase i hat Zugriffe auf $f_i \geq 1$ frische und $a_i = k - f_i \leq k-1$ alte Seiten.
- Alte Seiten erzeugen nur dann Cache-Fehler, wenn sie vorher in Phase i verdrängt wurden.
- Betrachte den ersten Zugriff auf die $(j+1)$-te alte Seite $(j = 0, \ldots, a_i - 1)$.
- Am Anfang gibt es k alte unmarkierte Seiten in C. Jetzt gibt es insgesamt noch $k-j$ alte unmarkierte Seiten (in C oder nicht).
- Vorherige erste Anfragen : $f_{ij} \leq f_i$ frische und j alte Seiten.

- Es sind $k - f_{ij} - j$ unmarkierte alte Seiten in C. Ist die $(j+1)$-te Seite noch dabei? Wahrscheinlichkeit dafür genau

$$\frac{\text{Anzahl unmarkiert alt in } C}{\text{Anzahl unmarkiert alt insgesamt}} = \frac{k - f_{ij} - j}{k - j}$$

- Wenn nicht, dann haben wir einen Cache-Fehler. Wahrscheinlichkeit dafür höchstens

$$1 - \frac{k - f_{ij} - j}{k - j} = \frac{f_{ij}}{k - j} \leq \frac{f_i}{k - j}$$

- Frische Seiten erzeugen immer Cache-Fehler. Also gilt für die erwartete Anzahl Cache-Fehler

$$f_i + \sum_{j=0}^{a_i - 1} \frac{f_i}{k - j} \;\leq\; f_i \cdot \left(1 + \sum_{j=2}^{k} \frac{1}{j}\right) = f_i \cdot H_k$$

  Für die Ungleichung nutzen wir $a_i \leq k - 1$, da Phase i immer mindestens eine frische Seite haben muss.
- Erwartete Gesamtanzahl Cache-Fehler von Random-Mark also höchstens

$$k + H_k \sum_{i=2}^{t} f_i \;\leq\; H_k \left(k + \sum_{i=2}^{t} f_i\right)$$

Für das Offline-Optimum LFD gilt, dass immer mindestens

$$\frac{1}{2}\left(k + \sum_{i=2}^{t} f_i\right)$$

Cache-Fehler erzeugt werden müssen. (ohne Beweis) □

6.3 Secretary-Problem

Beim Paging sind alle deterministischen Strategien bestenfalls k-kompetitiv, aber in der Praxis hat z.B. LRU nur einen sehr kleinen konstanten Verlust gegenüber LFD. Ähnliche Beispiele gibt es in vielen anderen Bereichen. Das Problem ist hierbei, dass die Analyse über Offline-Optima und Wettbewerbsfaktoren im Worst-Case sehr pessimistisch ist. Sie erlaubt nicht immer, gute und schlechte Algorithmen zu unterscheiden. In solchen Fällen benötigt man andere Benchmarks für den Vergleich, z.B. stochastische Eingabeverteilungen und Average-Case Analyse.

Hier betrachten wir als Beispiel zum Abschluss einen interessanten Mix aus stochastischen und Worst-Case-Elementen. Wir untersuchen das klassische **Gambler Problem**:

- Es kommt eine Sequenz σ von n Boxen an, n bekannt.
- In jeder Box $i = 1, \ldots, n$ liegt ein Preis mit unbekanntem Wert $v_i \geq 0$.
- Bei Ankunft von Box i wird sie geöffnet und v_i bekannt.
- Danach muss sofort entschieden werden, ob v_i akzeptiert oder endgültig verworfen wird.

Algorithmus 25: Greedy-Algorithmus für das Secretary Problem

```
1 Verwerfe die ersten s Boxen.
2 for Box i = s + 1, ..., n do
3 |   if v_i bester Preis bisher then return akzeptiere i
4 |     else verwerfe i
```

- Akzeptanz: Prozess endet, man erhält Preis v_i.
- Verwerfen: Nächste Box kommt an, v_i ist für immer verloren.

Ziel: Maximiere den Wert des akzeptierten Preises.

Ein einfacher randomisierter Algorithmus ALG: Wähle uniform zufällig eine der n Boxen und akzeptiere ihren Preis. Dann gilt $\mathbb{E}[ALG(\sigma)] = \sum_{i=1}^{n} v_i/n \geq OPT(\sigma)/n$, wobei $OPT(\sigma)$ der maximale Preis in den Boxen. ALG ist also (strikt) n-kompetitiv.

Man kann zeigen:

Theorem 90. *Jeder randomisierte Algorithmus für das Gambler Problem ist $\Omega(n)$-kompetitiv.*

Die Aussage wäre also, dass alle Algorithmen quasi gleich katastrophal schlecht sind. Das ist aber sicher nicht so, in der Praxis werden Probleme dieser Art täglich mit guten Ergebnissen gelöst.

Um gute Algorithmen von weniger guten zu unterscheiden, müssen wir die Worst-Case-Perspektive etwas auflockern. In vielen Entscheidungsprozessen ist es eher selten, dass die exakte Reihenfolge, in der die Dinge passieren, den Worst-Case darstellt. Um diese Eigenschaft zu modellieren, nehmen wir an, dass der **Wert der Preise durch einen Gegner** bestimmt werden kann, die **Ankunftsreihenfolge allerdings danach uniform zufällig** permutiert wird.

Dieses Szenario wird das **Secretary Problem** genannt. Betrachte den Greedy-Algorithmus 25.

Theorem 91. *Der Greedy-Algorithmus mit $s = \lfloor n/\mathsf{e} \rfloor$ ist $(\mathsf{e} + o(1))$-kompetitiv.*

Beweis: Sei i^* die optimale Box. Wir zeigen, dass der Algorithmus i^* **mit Wahrscheinlichkeit mindestens** $1/\mathsf{e} - 1/n$ **akzeptiert**.

- Akzeptanz von $i^* \iff$ Zwei Events treten ein:
  Event A_t: i^* kommt in Runde $t \geq s + 1$ an, und
  Event $R_{(1,t-1)}$: Wir haben in Runde $1, \ldots, t-1$ alles verworfen
- Wahrscheinlichkeit, dass A_t eintritt, ist $1/n$.
- Wenn A_t eintritt, was ist dann die bedingte Wkeit $\Pr[R_{(1,t-1)} \mid A_t]$?

[Pic: Ankunft, zwei Events]

Vorher keine Akzeptanz?

- Sei $[n] = \{1, \ldots, n\}$ die Menge aller Boxen. Betrachte die Menge $S \subseteq [n] \setminus \{i^*\}$ der Boxen, die in Runden $1, \ldots, t-1$ ankommen. Was ist mit der Box i^*_S mit dem besten Preis in S?

- i_S^* kommt in einer Runde $1, \ldots, s \Rightarrow$ verwerfe alles in Runde $1, \ldots, t-1$.
- i_S^* kommt in einer Runde $s+1, \ldots, t-1 \Rightarrow$ akzeptiert in einer Runde $s+1, \ldots, t-1$.
- Daher: i_S^* kommt in einer Runde $1, \ldots, s \iff R_{(1,t-1)}$
- Gegeben eine Menge S wird sie in der Ankunftsreihenfolge uniform zufällig auf die Runden $1, \ldots, t-1$ permutiert. Also ist die Wahrscheinlichkeit, dass i_S^* in einer Runde $1, \ldots, s$ ankommt, genau $s/(t-1)$

Insgesamt also: Bedingt auf A_t gilt für jede Menge S von Boxen, die in den ersten $t-1$ Runden eintreffen, dass $\Pr[R_{(1,t-1)} \mid A_t] = s/(t-1)$.

Die Akzeptanzwahrscheinlichkeit von i^* ist nach unten beschränkt durch

$$\begin{aligned}
\sum_{t=s+1}^{n} \Pr[A_t] \cdot \Pr[R_{(1,t-1)} \mid A_t] &= \sum_{t=s+1}^{n} \frac{1}{n} \cdot \frac{s}{t-1} = \frac{s}{n} \cdot \sum_{t=s}^{n-1} \frac{1}{t} \\
\geq \frac{s}{n} \cdot \left(\int_{t=s}^{n} \frac{1}{t} \right) &= \frac{s}{n} \cdot (\ln n - \ln s) = \frac{s}{n} \cdot \ln \frac{n}{s} \\
= \frac{\lfloor n/e \rfloor}{n} \cdot \ln \frac{n}{\lfloor n/e \rfloor} &\geq \left(\frac{1}{e} - \frac{1}{n} \right) \cdot \ln \frac{n}{n/e} = \frac{1}{e} - \frac{1}{n} \ .
\end{aligned}$$

[Pic: Summe-zu-Integral]

Der Algorithmus wählt i^* mit Wahrscheinlichkeit mindestens $1/\mathsf{e} - 1/n$. Also ist der erwartete Wert des Algorithmus mindestens $(1/\mathsf{e} - 1/n) \cdot v_{i^*}$. OPT nimmt immer i^*, d.h. $OPT(\sigma) = v_{i^*}$. Der Wettbewerbsfaktor ist also höchstens

$$\frac{v_{i^*}}{\left(\frac{1}{\mathsf{e}} - \frac{1}{n}\right) \cdot v_{i^*}} = \frac{e}{1 - \mathsf{e}/n} = \mathsf{e} + \frac{\mathsf{e}^2}{n - \mathsf{e}} = \mathsf{e} + o(1). \qquad \square$$