Inhaltsverzeichnis

1 Lineare Programmierung

1.1 Einführung

Eine Firma stellt PCs und Laptops her.

	Arbeitskosten	Materialkosten	Verkaufspreis
PC	4	200€	600€
Laptop	8	200€	800€

Es stehen ihr 160 Arbeitsstunden pro Woche zur Verfügung.
Außerdem ist der Kreditrahmen, aus dem Material gekauft werden soll, auf 6000€ beschränkt.
Was soll die Firma produzieren, um ihren Gewinn zu maximieren?

1. Nur Laptops
   maximal 20 Laptops (wegen Arbeitsstunden)
   Gewinn: 12000€

2. Nur PCs
   maximal 30 PCs (wegen Kreditrahmen)
   Gewinn: 12000€

3. Bessere Strategie
   10 Laptops, 20 PCs
   Gewinn: 14000€

Beobachtung: Neue Strategie erfüllt beide Bedingungen (Arbeitsstunden und Kreditrahmen) mit Gleichheit.

Mathematische Formulierung:

x_1 PCs
x_2 Laptops

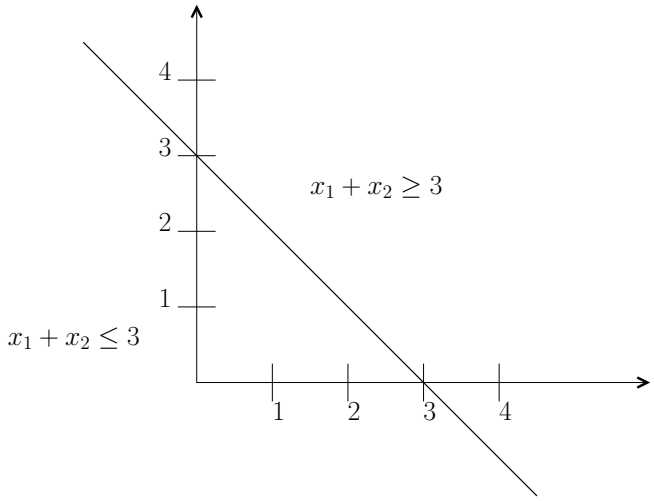
maximiere	$400x_1 + 600x_2$			
unter den Bedingungen	$4x_1 + 8x_2$	$\leq$	160	(Arbeitsstunden)
	$200x_1 + 200x_2$	$\leq$	6000	(Kreditrahmen)
	x_1	$\geq$	0	
	x_2	$\geq$	0	

Vereinfachung:

maximiere	$2x_1 + 3x_2$		
u.d.B.	$x_1 + 2x_2$	$\leq$	40
	$x_1 + x_2$	$\leq$	30
	x_1, x_2	$\geq$	0

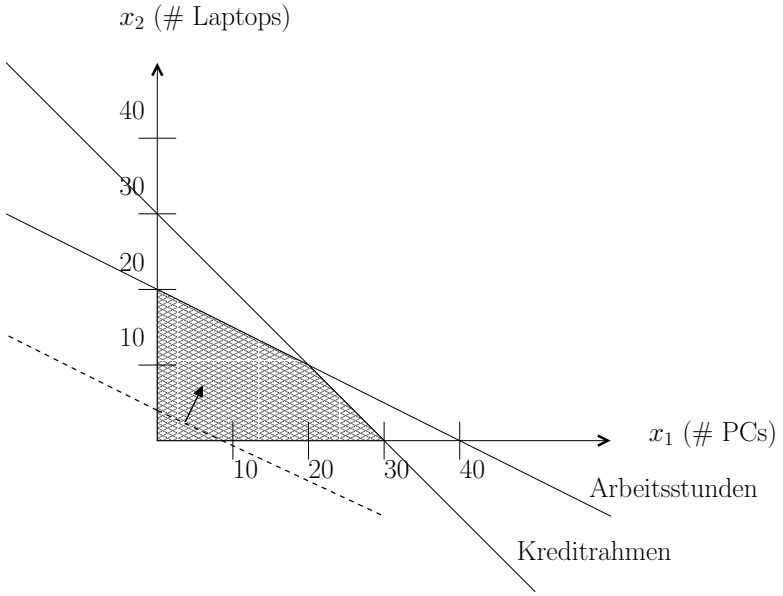
Geometrische Interpretation:
Die Gleichung $a_1x_1 + a_2x_2 = b$ (a_1, a_2, b Konstanten) ist eine Linie.

Beispiel 1.1.1 *Bsp.:* $x_1 + x_2 = 3$



$a_1x_1 + a_2x_2 \leq b$ ist der Bereich unterhalb einer Linie (einschließlich der Linie). Das nennt man auch Halbraum. $a_1x_1 + a_2x_2 \geq b$ ist der Bereich oberhalb einer Linie und wird auch Halbraum genannt.

Geometrische Interpretation unseres Bsp.



Aber was ist die beste Lösung?

– Betrachte alle Lösungen mit demselben Gewinn
z.B. Gewinn 1200 (3 PCs oder 2 Laptops; zur Vereinfachung: Auch Bruchteile von PCs bzw. Laptops erlaubt)

Alle Lösungen auf der Geraden $400x_1 + 600x_2 = 1200$ haben Wert 1200.
Nach Kürzen: Alle Lösungen auf der Geraden $2x_1 + 3x_2 = z$ haben dieselbe Qualität z.
Die Verschieberichtung ergibt sich dabei aus der Zielfunktion $2x_1 + 3x_2 = z$, nämlich durch den Vektor $\binom{2}{3}$.

Zur Erinnerung

Für zwei Vektoren $a = \begin{pmatrix} a_1 \\ \vdots \\ a_d \end{pmatrix}, x = \begin{pmatrix} x_1 \\ \vdots \\ x_d \end{pmatrix} \in \mathbb{R}^d$ bezeichnet $a^T \cdot x = a_1x_1 + ...a_dx_d$ das innere Produkt der Vektoren. Hat a (bzw. x) Länge 1, so gibt a^Tx die Länge des Vektors der orthogonalen Projektion des Punktes x (bzw. a) auf die durch $\lambda \cdot a$ ($\lambda \in \mathbb{R}$) definierte Gerade an. Dies ist auch einfach $\cos\alpha \cdot ||x||$, wobei α der Winkel zwischen a und x ist und $||x|| = \sqrt{x_1^2 + x_2^2 + ... + x_n^2}$ die euklidische Länge des Vektors x ist. Damit liefert die Gleichung $a_1x_1 + a_2x_2 = 0 \Leftrightarrow a^T \cdot x = 0$ mit $a = \begin{pmatrix} a_1 \\ a_2 \end{pmatrix}, x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$ für festes a die Menge aller Vektoren, die senkrecht auf a stehen. In unserem Fall ist $a = \binom{2}{3}$. Durch Addieren bzw. Subtrahieren von z wird diese Gerade dann um $\frac{z}{a_2}$ nach oben (bzw. unten) verschoben. Da eine Gerade beidseitig unendlich ist, kann man dies auch als eine Verschiebung in Richtung a um eine Distanz von $\frac{z}{a_2} \cdot (\cos((1,0), a))$ betrachten.

$$\begin{aligned} & 2x_1 + 3x_2 = 6 \\ \Leftrightarrow\ & 3x_2 = 6 - 2x_1 \\ \Leftrightarrow\ & x_2 = 2 - \tfrac{2}{3}x_1 \end{aligned}$$

Wichtige Beobachtung:

Wir können unsere Eingabe durch Drehen immer so transformieren, dass der Zielfunktionsvektor in eine bestimmte Richtung zeigt, z. B. nach unten (s. Übung).

Lineare Programmierung, allgemeine Formulierung:

$\max c_1x_1 + c_2x_2 + ... + c_dx_d$, $x \in \mathbb{R}^d$
unter den linearen Bedingungen:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + ... + a_{1d}x_d &\leq b_1 \\ a_{21}x_1 + a_{22}x_2 + ... + a_{2d}x_d &\leq b_2 \\ &\vdots \\ a_{m1}x_1 + a_{m2}x_2 + ... + a_{md}x_d &\leq b_m \end{aligned}$$

Matrixschreibweise: $\max c^Tx, x \in \mathbb{R}^d$
unter $Ax \leq b$, wobei $c \in \mathbb{R}^d, b \in \mathbb{R}^n$, A $n \times d$-Matrix

Wichtige Beobachtung:

Die optimale Lösung eines linearen Programms muss nicht ganzzahlig sein!

Ganzzahlige Lineare Programme (ILP)

$\max c^T x$, $x \in \mathbb{Z}^d$ unter $Ax \leq b$, wobei $c \in \mathbb{R}^d$, $b \in \mathbb{R}^m$, A $m \times d$ Matrix

Satz 1.1.2 *ILP ist NP-schwer.*

Beweis:
Vertex Cover (NP-vollständig)

Sei $G = (V, E)$ ein Graph und gesucht ist die kleinste Menge $U \subseteq V$, so dass für alle Kanten $(u, v) \in E$ $u \in U$ oder $v \in U$ gilt.
Man kann Vertex Cover auf ILP reduzieren. Sei $G = (U, E)$ ein Graph. Wir führen für jeden Knoten v eine Variable x_v ein.

$$\begin{aligned}
\min \quad & \sum_{v \in V} x_v \\
\text{unter } & 0 \leq x_v \leq 1,\ v \in V \\
& x_v + x_u \geq 1,\ (u, v) \in E \\
& x \in \mathbb{Z}^{|V|}
\end{aligned}$$

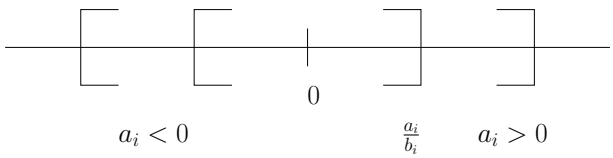
Offensichtlich ist das Problem äquivalent zu Vertex Cover und kann in Polynomialzeit konstruiert werden.

1.2 1D-LP

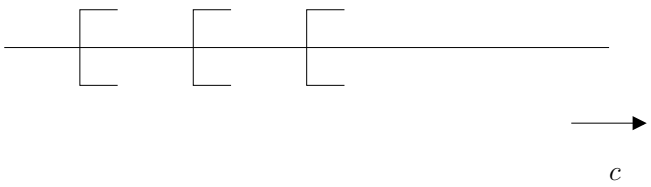
Es seien $c, a_1, \ldots, a_n, b_1, \ldots, b_n \in \mathbb{R}$.

max	cx
unter den Bedingungen	$a_1 x \leq b_1$
	$a_1 x \leq b_2$
	$\vdots$
	$a_n x \leq b_n$

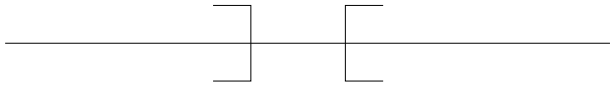
Mögliche Situationen:



LP hat optimale Lösung.



LP ist unbeschränkt.



LP ist unzulässig.

Im 1D haben Bedingungen die Form $x \geq b_i$ oder $x \leq b_i$. O.B.d.A. sei $c > 0$.

Zulässigkeitsbereich $\neq \emptyset$ und nach rechts beschränkt

$\Rightarrow$ rechtester Punkt ist optimale Lösung

Zulässigkeitsbereich ist nach rechts unbeschränkt

$\Rightarrow$ zu jeder Lösung gibt es eine bessere Lösung

Zulässigkeitsbereich ist leer

Man kann 1D-LD in $\mathcal{O}(n)$ Zeit lösen, indem man die maximale linke und die minimale rechte Schranke des Zulässigkeitsbereichs sucht.

1.3 2D-LP

Annahme: Zielfunktion zeigt „nach unten“(Übung!)

Mögliche Situationen:

1. Zulässigkeitsbereich ist leer
2. Die Zielfunktion ist unbeschränkt
3. Jeder Punkt einer Strecke ist optimale Lösung
4. Eine Ecke ist optimale Lösung

Definition 1.3.1 (Konvexivität) *$M \subseteq \mathbb{R}^d$ heißt konvex, wenn $\forall x, y \in M : \{\lambda x + (1-\lambda)y \mid 0 \leq \lambda \leq 1\} \subseteq M$ gilt.*

Lemma 1.3.2 *Sind $M_1, M_2, ..., M_e$ konvexe Mengen im $\mathbb{R}^d$, so ist auch der Schnitt dieser Mengen $\bigcap_{i=1}^{e} M_i$ konvex.*

Algorithmus 1.3.3 (Inkrementeller Algorithmus, Entwurf)

Start mit optimaler Lösung für h_1 und h_2 (Annahme: beschränkt)

füge in jedem Schritt eine neue Bedingung hinzu und aktualisiere Lösung

2 Fälle:

Die optimale Lösung v_{i-1} für $h_1, ..., h_{i-1}$ liegt in h_i. Dann gilt: Die optimale Lösung v_i für $C_i = h_1 \cap ... \cap h_i$ ist v_{i-1}.

Die optimale Lösung liegt nicht in h_i.

Lemma 1.3.4 *Liegt die optimale Lösung v_{i-1} für C_{i-1} nicht in h_i und ist C_i nicht leer, so liegt eine optimale Lösung für C_i auf l_i (wobei l_i die Linie ist, die h_i begrenzt).*

Beweis:
Sei $v_{i-1} \notin C_i$ und C_i nicht leer. Annahme: Es gibt keine optimale Lösung $v_i \in l_i$. Sei v_i eine optimale Lösung für C_i. Betrachte Strecke (v_{i-1}, v_i). Es gilt $v_{i-1} \in C_{i-1}$ und $v_i \in C_i \subseteq C_{i-1}$. $\underset{\text{Konvexivität von} C_{i-1}}{\Rightarrow} (v_{i-1}, v_i) \in C_{i-1}$. Da der Wert der Zielfunktion für v_i nicht größer ist als der für v_{i-1} und weil die Zielfunktion linear ist, wenn man sich von v_i nach v_{i-1} bewegt. Da (v_{i-1}, v_i) die Linie l_i schneidet, hat die Zielfunktion am Schnittpunkt q mindestens den Wert, den sie am Punkt v_i hat.

Damit ist aber entweder (a) q eine bessere Lösung als v_i oder
(b) q hat denselben Zielfunktionswert wie v_i, liegt aber auf l_i.

Fall (a) ist ein Widerspruch zur Optimalität von v_i und (b) ist ein Widerspruch zur Annahme, dass auf l_i keine optimale Lösung liegt.

□

Beobachtung:

Ist C_i leer, so ist auch $l_i \cap C_i$ leer.

Beobachtung:

Die Laufzeit beträgt $\sum_{i=1}^{n} \mathcal{O}(i) = \mathcal{O}(n^2)$

Wie kann man v_i finden?

Wir wissen $v_i \in l_i$. Dies bedeutet, dass wir $p \in L_i$ suchen, was die Zielfunktion maximiert und im Zulässigkeitsbereich liegt. O.B.d.A. sei L_i die x-Achse und seien die x_j die Schnittpunkte von L_j mit l_i.

max x_1
unter Bedingung $x_1 \geq x_j$ für alle $j \in \{1, ..., i-1\}$, für die $l_i \cap h_j$ nach links begrenzt
$x_1 \leq x_k$ für alle $k \in \{1, ..., i-1\}$, für die $l_i \cap h_k$ nach rechts begrenzt.

Algorithmus 1.3.5 (Inkrementeller Algorithmus für 2D-LP)

1. *if LP ist unbeschränkt then*
2. *gib Strahl aus, für den die Zielfunktion unbeschränkt ist*

3. *else*

4. *seien h_1, h_2 Zeugen für die Unbeschränktheit und v_2 ihr Schnittpunkt.*

5. *Berechne zufällige Reihenfolge $h_3, ..., h_m$ der übrigen Halbebenen*

6. *for $i = 3$ to n do*

7. *if $v_{i-1} \in h_i$ then $v_i = v_{i-1}$*

8. *else v_i = Punkt p auf l_i mit maximaler Zielfunktion*

9. *if p exisitiert nicht, dann gibt aus, dass LP nicht zulässig ist*

10. *return v_n*

Satz 1.3.6 *2D-LP mit n Bedingungen kann man in $\mathcal{O}(n)$ erwarteter Laufzeit lösen.*

Beweis:
Sei X_i die Zufallsvariable für das Ereignis $v_{i-1} \notin h_i$. Wir wissen, dass 1D-LP mit i Bedingungen in $\mathcal{O}(i)$ Zeit gelöst werden kann. Also ist die erwartete Gesamtlaufzeit $T(n)$.

$$\begin{aligned} T(n) &= \mathcal{O}(n) + E\left(\sum_{i=3}^{n} \mathcal{O}(i) \cdot X_i\right) \\ &= \mathcal{O}(n) + \sum_{i=3}^{n} \mathcal{O}(i) \cdot E(X_i) \text{ (wegen Linearität des Erwartungswertes)} \end{aligned}$$

Da X_i eine 0-1-Zufallsvariable ist, gilt:

$$\begin{aligned} E[X_i] &= 0 \cdot Pr[X_i = 0] + 1 \cdot Pr[X_i = 1] \\ &= Pr[x_i = 1] \\ &= Pr[v_{i-1} \notin h_i] \end{aligned}$$

Um diese Wahrscheinlichkeit zu bestimmen setzen wir sog. Rückwärtsanalyse ein, d.h. die Tatsache, dass sequentielles Einfügen von n Halbebenen in zufälliger Reihenfolge äquivalent zum Sequentiellen löschen der n Halbebenen aus der Menge aller n Halbebenen ist. Es gilt daher

$$\begin{aligned} Pr[v_{i-1} \notin h_i] &= \sum_{\substack{\text{alle Auswahlen} \\ \text{von } h_{i+1}, ..., h_n}} Pr[h_{i+1}, ..., h_n] \cdot Pr[v_{i-1} \notin h_i \mid h_{i+1}, ..., h_n \text{ schon gelöscht}] \\ &= \sum Pr[h_{i+1}, ..., h_n] \cdot Pr[v_{i-1} \neq v_i \mid h_{i+1}, ..., h_n] \\ &= \sum Pr[h_{i+1}, ..., h_n] \cdot \tfrac{2}{i-2} \end{aligned}$$

Denn v_i ist durch den Schnitt zweier eine Halbebene begrenzender Linien definiert. (Wir nehmen allg. Lage an, d.h. keine drei solchen Linien schneiden sich in einem Punkt.) Dann gilt $v_i \neq v_{i-1}$ nur dann, wenn eine dieser beiden Halbebenen h_i ist. Die Wahrscheinlichkeit hierfür ist gerade $\frac{2}{i-2}$ (weil wir zwei der $i-2$ Halbebenen $h_3, \ldots, h_n$ ziehen).

$$= \frac{2}{i-2} \sum_{\substack{\text{alle Auswahlen} \\ \text{von } h_{i+1},...,h_n}} Pr[h_{i+1},...,h_n] = \frac{2}{i-2}$$

$$(Pr[h_{i+1},...,h_n] \text{ ist } \frac{1}{\#\text{ möglicher Auswahlen}})$$

Damit folgt: $T(n) = \mathcal{O}(n) + \sum_{i=3}^{n} \mathcal{O}(i) \cdot \frac{2}{i-2} = \mathcal{O}(n)$.

Unbeschränkte LPs

$H = \{h_1, ..., h_n\}$ und v_i die nach außen gerichtete Normale von h_i. Sei l_i die Linie, die h_i begrenzt. Sei Φ_i der kleinere Winkel zwischen c und v_i. Sei $H_{\min} = \{h_j \in H \mid v_j = v_i\}$, wobei h_i die Halbebene ist, für die Φ_i minimal. Außerdem sei $H_{par} = \{h_j \in H \mid v_j = -v_i\}$. Ist der Streifen zwischen den Begrenzungen der Halbräume aus $H_{\min}$ und denen der Halbräume aus H_{par} leer, dann ist das LP nicht zulässig. Sei l_{i^*} die Linie mit h_{i^*} aus $H_{\min}$, die diesen Streifen begrenzt.

Lemma 1.3.7 *Es gilt:*
(i) Wenn $l_{i^*} \cap h_j$ in Richtung c unbeschränkt ist für alle h_j in $H \backslash (H_{\min} \cup H_{par})$, dann ist (H, c) unbeschränkt entlang eines Strahls in l_{i^*}. Diesen Strahl kann man in $\mathcal{O}(n)$ Zeit berechnen.

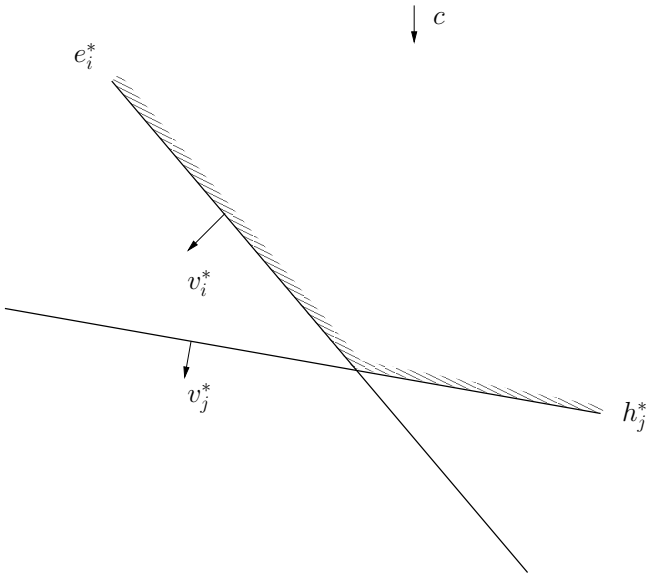
(ii) Wenn $l_{i^*} \cap h_{j^*}$ in Richtung c beschränkt ist für ein h_{j^*} aus $H \backslash (H_{\min} \cup H_{par})$ dann ist das lineare Programm $(\{h_{i^*}, h_{j^*}\}, c)$ beschränkt.

Beweis: (i) Der Schnitt zwischen l_{i^*} und h_j ist ein Strahl s_j der unbeschränkt ist. Der Schnitt aller s_j ist daher auch ein unbeschränkter Strahl und kann in $\mathcal{O}(n)$ Zeit berechnet werden.

(ii) Annahme: $l_{i^*} \cap h_{j^*}$ ist begrenzt in Richtung c, aber $(\{h_{i^*}, h_{j^*}\}, c)$ ist unbegrenzt. Dann muss der Strahl $l_{j^*} \cap h_{i^*}$ unbegrenzt in Richtung c sein. Das ist nur dann der Fall, wenn die Normale von h_{j^*} einen mindestens genauso kleinen Winkel mit c bildet wie l_{i^*}. Widerspruch zur Definition von $H_{\min}$, da $h_{j^*} \notin H_{\min} \cup H_{par}$ und somit der Winkel zu c sogar kleiner sein muss.

Algorithmus 1.3.8 (Unbeschränktes LP (H, c))

1. *Berechne $H_{\min}$ und H_{par}*
2. *$\hat{H} = H \backslash (H_{\min} \cup H_{par})$*
3. *Berechne den Schnitt der Halbebenen in $H_{\min} \cup H_{par}$*
4. *if Schnitt leer then Ausgbe: LP nicht zulässig*
5. *else*
6. *if es gibt Halbebene $h_{j^*} \in \hat{H}$, so dass $l_i \cap h_{j^*}$ in Richtung c beschränkt ist then*



Ausgabe $(\{h_{i^*}, h_{j^*}\}, c)$ *ist beschränkt*

7. *else Ausgabe* (H, c) *ist unbeschränkt entlang* $l_{i^*} \cap \left(\bigcap \hat{H}\right)$.

1.4 Wichtige Sätze der Linearen Algebra

Lineare Unterräume

$\mathbb{R}^d$: d-dim. euklidischer Raum
Punkte sind reellwertige d-Tupel $x = (x_1, ..., x_d)$
$\mathbb{R}^d$ ist Vektorraum, daher können wir über lineare Unterräume sprechen

Definition 1.4.1 *Ein linearer Unterraum ist eine Untermenge des* $\mathbb{R}^d$*, die abgeschlossen unter Addition von Vektoren und Multiplikation mit reellen Zahlen (Skalaren) ist.*

Beispiele 1.4.2 *Ursprung, alle Linien durch den Ursprung, alle Ebenen durch den Ursprung, usw.*

Affine Unterräume

Allgemeine Linien, Ebenen, usw.

Allgemein: $x + L$, wobei $x \in \mathbb{R}^d$ und L ein linearer Vektorraum.

Definition 1.4.3 *Die lineare (affine) Hülle einer Menge* $X \subseteq \mathbb{R}^d$ *ist der Schnitt aller linearen (affinen) Unterräume, die* X *enthalten. Die lineare Hülle kann auch als Menge aller Linearkombinationen von Vektoren aus* X *beschrieben werden.*

Was ist das affine Analog?
Betrachte die affine Hülle von $a_1, ..., a_m \in \mathbb{R}^d$.
Wir verschieben die Menge um $-a_m$, um einen linearen Unterraum zu erhalten. Nachher schieben wir um $+a_m$ zurück. Wir erhalten, dass

$$\underbrace{\beta_1(a_1 - a_m) + \beta_2(a_2 - a_m) + ... + \beta_m(a_{m-1} - a_m)}_{\text{linearer Unterraum}} + \underbrace{a_m}_{\text{zurückverschieben}}$$
$$= \beta_1 a_1 + \beta_2 a_2 + ... + \beta_{m-1} a_{m-1} + a_m(1 - \beta_1 - \beta_2 - ... - \beta_{m-1})$$

den affinen Raum beschreibt.
Damit ist eine affine Kombination ein Ausdruck der Form

$$\alpha_1 a_1 + \alpha_2 a_2 + ... + \alpha_m a_m \text{ mit } \sum \alpha_i = 1 \text{ und } \alpha_1, ..., \alpha_m \in \mathbb{R}$$

Affine Abhängigkeit

Eine Punktmenge $\{a_1, ..., a_m\}$ heißt affin abhängig wenn man einen der Punkte als affine Kombination der anderen Punkte schreiben kann, d.h. es gibt $\alpha_1, ..., \alpha_m$, mindestens ein $\alpha_i \neq 0$, so dass

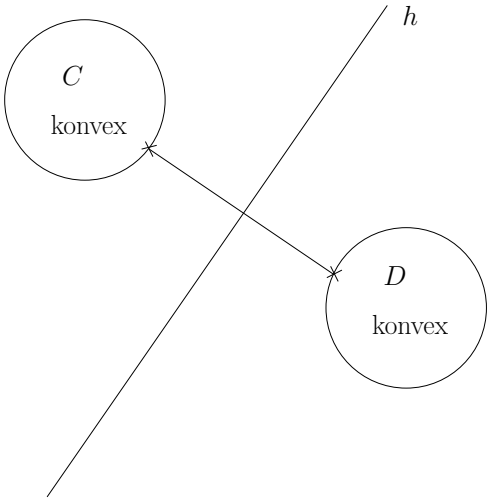
$$\alpha_1 a_1 + ... + \alpha_m a_m = 0 \text{ und } \alpha_1 + \alpha_2 + ... + \alpha_m = 0.$$

Begründung: Annahme: affin abhängig
$a_m = \alpha_1 a_1 + ... + \alpha_{m-1} a_{m-1}$ mit $\alpha_1 + ... + \alpha_{m-1} = 1$
(a_m ist affine Kombination von $a_1, ..., a_{m-1}$)
$\Leftrightarrow \alpha'_m a_m = \alpha'_1 a_1 + ... + \alpha'_{m-1} a_{m-1}$ mit $\alpha'_1, ..., \alpha'_{m-1} = \alpha'_m$
$\alpha'_i = \alpha'_m \cdot \alpha_i, 1 \leq i \leq m-1$
für alle $\alpha'_w \in \mathbb{R}$

Die konvexe Hülle $conv(X)$ einer Menge $X \subseteq \mathbb{R}^d$ ist der Schnitt aller konvexen Mengen in $\mathbb{R}^d$, die X enthalten.

Satz 1.4.4 (Separationssatz) *Seien $C, D \subseteq \mathbb{R}^d$ konvexe Mengen mit $C \cap D = \emptyset$. Dann gibt es eine Hyperebene h, so dass C in einem der durch h definierten abgeschlossenen Halbräume liegt und C in dem anderen abgeschlossenen Halbraum.*

Beweis:
Wir zeigen den Satz nur für den Fall, dass C und D kompakt sind ($\mathbb{R}^d$, daher äquivalent zu abgeschlossen und beschränkt, Bolzano-Weierstrass). Dann ist das kartesische Produkt $C \times D$ ebenfalls kompakt und die Distanzfunktion $(x, y) \mapsto ||x - y||_2$ nimmt ein Minimum auf $C \times D$ an. Hier bezeichnet $||z||_2 = \sqrt{\sum_{i=1}^{d} (z_i)^2}$ die euklidische Länge des Vektors $z = (z_1, ..., z_d)^T$. Es gibt also $p \in C$ und $q \in D$ so dass die Distanz von C und D gleich der Distanz von p und q ist. Die separierende Hyperebene h kann dann als senkrecht zur Strecke pq und durch ihren Mittelpunkt gelegt werden. Man kann leicht zeigen, dass h C und D nicht schneidet (Übung).



Affine Abhängigkeit von $a_1, ..., a_m$ ist äquivalent zu linearer Abhängigkeit der $m-1$ Vektoren $(a_1 - a_m), ..., (a_{m-1} - a_m)$. Damit gibt es maximal $d+1$ affine unabhängige Punkte in $\mathbb{R}^d$.

Definition 1.4.5 (affine Unterräume) *Wir verwenden folgende Bezeichnungen für bestimmte affine Unterräume:*

Hyperebene $d-1$*–dimensionaler affiner Unterraum in* $\mathbb{R}^d$

Ebene *2-dimensionaler affiner Unterraum in* $\mathbb{R}^d$

k**-Flach** k*-dimensionaler affiner Unterraum*

Hyperebene $\{x \in \mathbb{R}^d \mid a^T x = b\}$, $a \in \mathbb{R}^d \backslash \{0\}$, $b \in \mathbb{R}$

Halbraum $\{x \in \mathbb{R}^d \mid a^T x \leq b\}$, $a \in \mathbb{R}^d \backslash \{0\}$, $b \in \mathbb{R}$

Ein k-Flach kann man als Schnitt von Hyperebenen darstellen, d.h. als Lösung des linearen Gleichungssystems $Ax = b$ oder als Bild einer affinen Abbildung $f : \mathbb{R}^k \to \mathbb{R}^d$ der Form $y \to By + c$ für eine $d \times k$-Matrix B und $c \in \mathbb{R}^d$.

$$B \cdot y = \sum_{i=1}^{k} y_i b_i$$

Dabei ist f also eine Kombination einer linearen Abbildung und einer Translation. Das Bild von f ist ein k'-Flach, wobei $k' \leq \min(k, d)$ der Rang der Matrix B ist.

Satz 1.4.6 (Der Fundamentalsatz über Lineare Ungleichungen) *Seien* $a_1, ..., a_m, b$ *Vektoren im* $\mathbb{R}^d$*. Dann gilt entweder*

I. b ist nicht negative Linearkombination von linear unabhängigen Vektoren aus $a_1, ..., a_m$

oder

II. Es gibt eine Hyperebene $\{x \mid c^T x = 0\}$*, die* $t-1$ *linear unabhängige Vektoren aus* $a_1, ..., a_m$ *enthält, so dass* $c^T b < 0$ *und* $c^T a_1, ..., c^T a_m \geq 0$*, wobei* $t = Rang(a_1 ... a_m, b)$*.*

Beweis:
Wir können annehmen, dass $a_1, ...a_m$ den $\mathbb{R}^d$ aufspannen. Ist dies nicht der Fall, dann betrachten wir nur den von $a_1, \ldots, a_m$ aufgespannten Unterraum U. Liegt b nicht in diesem Unterraum, so ist I offensichtlich nicht erfüllt. Wir betrachten dann eine Hyperebene H, die U enthält und b nicht enthält. Offensichtlich kann man diese Normale von H so wählen, dass $c^T a_1, \ldots, c^T a_m \geq 0$ und $c^T b < 0$ gilt.
Offensichtlich schließen sich I und II gegenseitig aus, da ansonsten, wenn wir $b = \lambda_1 a_1 + ... + \lambda_m a_m$ mit $\lambda_1, ..., \lambda_m \geq 0$ einen Widerspruch wegen $0 > c^T \cdot b = \lambda_1 c^T a_1 + ... + \lambda_m c^T a_m \geq 0$ hätten.
Um zu zeigen, dass mindestens I oder II erfüllt ist, wählen wir linear unabhängige Vektoren $a_{i_1}, ..., a_{i_d}$ aus $a_1, ..., a_m$ aus und setzen $D = \{a_{i_1}, ..., a_{i_d}\}$. Dann wenden wir folgende Iteration an:

(i) Schreibe $b = \lambda_{i_1} a_{i_1} + ... + \lambda_{i_d} a_{i_d}$ (die anderen λ_j sind 0!). Falls $\lambda_{i_1}, ..., \lambda_{i_d} \geq 0$, so sind wir in Fall I.

(ii) Ansonsten wähle den kleinsten Index h aus $i_1, ..., i_d$ mit $\lambda_h < 0$. Sei $\{x \mid c^T x = 0\}$ die Hyperebene, die durch $D \backslash \{a_h\}$ aufgespannt wird. Wir normalisieren c, so dass $c^T \cdot a_h = 1$. Da $\lambda_h < 0$ und $c^T a_{i_j} = 0$ für alle $i_j \neq h$, folgt $c^T b = \lambda_h < 0$.

(iii) Falls $c^T a_1, ..., c^T a_m \geq 0$, so sind wir in Fall II.

(iv) Ansonsten wähle das kleinste s, so dass $c^T a_s < 0$. Dann ersetze D durch $(D \backslash \{a_h\}) \cup \{a_s\}$. Starte neue Iteration.

Wir sind fertig, wenn wir zeigen, dass dieser Prozess terminiert. Sei D_k die Menge D in der k-ten Iteration. Wenn der Prozess nicht terminiert, dann gilt $D_k = D_l$ für geeignete $k, l, k < l$, da es nur endlich viele Auswahlmöglichkeiten für D gibt.
Sei r der größte Index für den gilt, dass a_r am Ende der Iterationen $k, k+1, \ldots, l-1$ aus D entfernt wurde. Nennen wir diese Iteration p. Da $D_k = D_l$ ist, wissen wir, dass a_r in einer Iteration q mit $k \leq q \leq l$ zu D hinzugefügt wurde.
Damit gilt $D_p \cap \{a_{r+1}, \ldots, a_m\} = D_q \cap \{a_{r+1}, \ldots, a_m\}$.

$$
\begin{array}{ccccccc}
 & & & & a_r \text{ wird eingefügt} & & \\
 & & & & \downarrow & & \\
D_k & \cdots & D_p & \cdots & D_q & \cdots & D_l \\
 & & \uparrow & & & & \\
 & & a_r \text{ wird gelöscht} & & & &
\end{array}
$$

Sei $D_p = \{a_{i_1}, \ldots, a_{i_d}\}$, $b = \lambda_{i_1} a_{i_1} + \cdots \lambda_{i_d} a_{i_d}$ und sei c' der Vektor c, der in Schritt (ii) von Iteration q gefunden wurde.
Nun gilt aber nach (ii), dass $0 > (c')^T b$.
Außerdem gilt in Iteration p, dass $h = r$ ist, denn r wird ja aus D_p entfernt, und somit $\lambda_r < 0$. Da h der kleinste solche Index ist, folgt für alle $i_j < r$, dass $\lambda_{i_j} \geq 0$.
Außerdem folgt in Iteration q, dass $s = r$ ist (r wird hinzugefügt). Aber s ist das kleinste s mit $(c')^T a_s < 0$.

Somit gilt

wenn $i_r < r$, dann $\lambda_{i_j} \geq 0$, $(c')^T a_{i_j} \geq 0$
wenn $i_r = r$, dann $\lambda_{i_j} < 0$, $(c')^T a_{i_j} < 0$

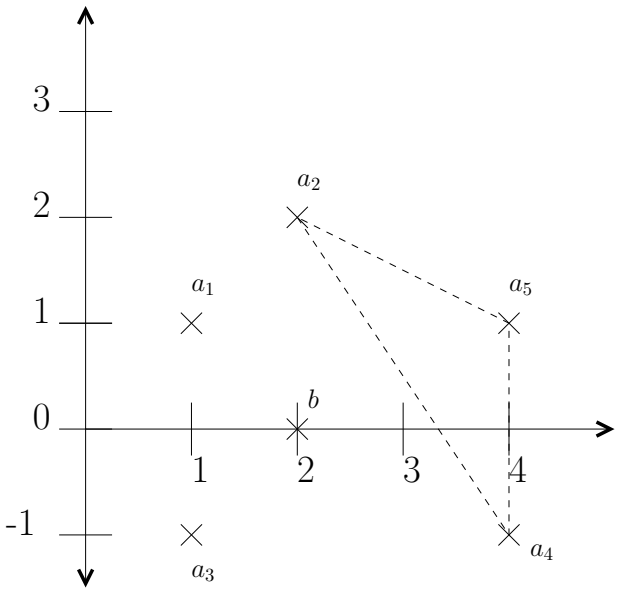
Da $D_p \cap \{a_{r+1}, \ldots, a_m\} = D_q \cap \{a_{r+1}, \ldots, a_m\}$, folgt aus (ii) (da die übrigen i_j die Hyperebene mit Normale c' aufspannen), dass $(c')^T a_{i_j} = 0$, wenn $i_j > r$ ist.

Es folgt

$$0 > (c')^T b = (c')^T \cdot (\lambda_{i_1} a_{i_1} + \cdots \lambda_{i_d} a_{i_d})$$
$$\lambda_{i_1}(c')^T a_{i_1} + \cdots \lambda_{i_d}(c')^T a_{i_d} > 0. \qquad \lightning$$

□

Beispiel 1.4.7 *Gegeben sind* $a_1 = (1,1,1), a_2 = (2,2,1), a_3 = (1,-1,1), a_4 = (4,-1,1), a_5 = (4,1,1)$ *und* $b = (2,0,1)$. *Start mit* $D = \{a_2, a_4, a_5\}$



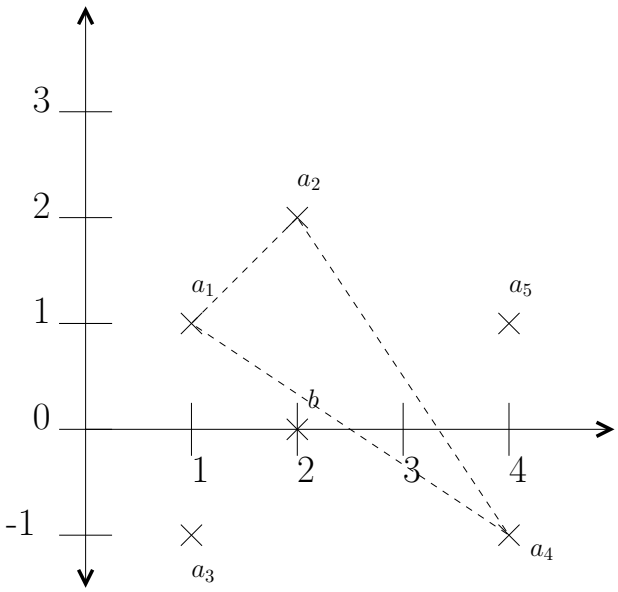
Berechne b *als Linearkombination von* a_2, a_4, a_5*:*

$$b = \lambda_2 a_2 + \lambda_4 a_4 + \lambda_5 a_5 = \begin{pmatrix} 2 \\ 2 \\ 1 \end{pmatrix} + \begin{pmatrix} 4 \\ -1 \\ 1 \end{pmatrix} - \begin{pmatrix} 4 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 2 \\ 0 \\ 1 \end{pmatrix}$$

$\Rightarrow \lambda_2 = 1, \lambda_4 = 1, \lambda_5 = -1$

Berechne neues D*:*

$\Rightarrow h = 5, s = 1 \Rightarrow D = \{a_1, a_2, a_4\}$



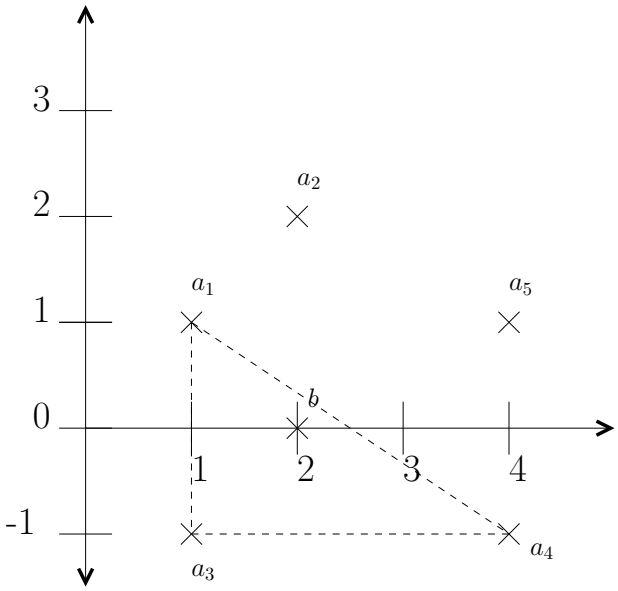
Nächste Iteration:

$b = \lambda_1 a_1 + \lambda - 2a_2 + \lambda_4 a_4 = \frac{4}{5}a_1 - \frac{1}{5}a_2 + \frac{2}{5}a_4$

$\Rightarrow h = 2$

$\Rightarrow D = \{a_1, a_3, a_4\}$

$\Rightarrow$ *b liegt im Kegel!*



Man kann b als nicht negative Linearkombination der Vektoren a_1, a_2, a_3 schreiben.

Ziel: (Dualität) $\max\left\{c^T x \mid Ax \leq b\right\} = \min\left\{y^T b \mid y \geq 0, y^T A = c^T\right\}$

Definition 1.4.8 (Konvexer Kegel) *$C \subseteq \mathbb{R}^d$, $C \neq \emptyset$ heißt konvexer Kegel, wenn aus $x, y \in C$ und $\lambda, \mu \geq 0$ immer $\lambda x + \mu y \in C$ folgt.*
Ein Kegel heißt polyedrisch, wenn $C = \{x \mid Ax \leq 0\}$ für eine $(m \times d)$-Matrix A, d.h. C ist Schnitt von endlich vielen Halbräumen. Der Kegel, der durch Vektoren $x_1, \ldots, x_m$ generiert wird, ist die Menge

$$cone\{x_1, \ldots, x_m\} = \{\lambda_1 x_1 + \cdots + \lambda_m x_m \mid \lambda_1, \ldots, \lambda_m \geq 0\},$$

d.h. der kleinste konvexe Kegel, der $x_1, \ldots, x_m$ enthält. Ein solcher Kegel heißt endlich erzeugt.

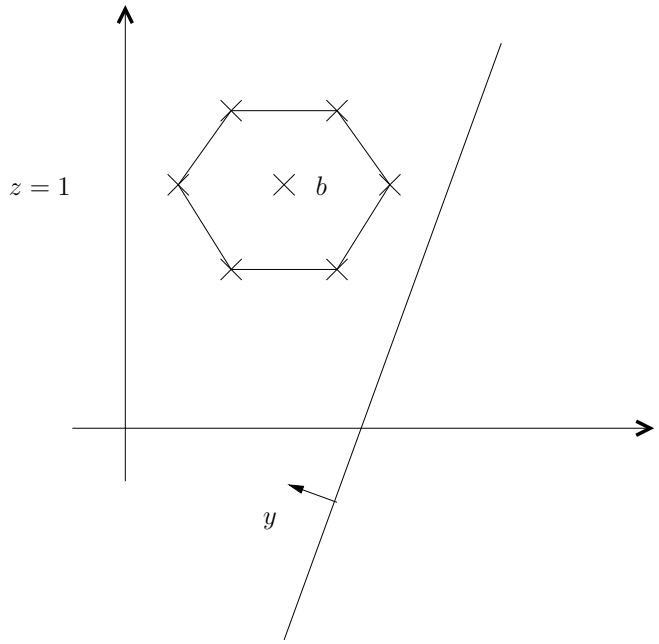
Lemma 1.4.9 (Farkas' Lemma) *Sei A eine $(m \times d)$-Matrix und $b \in \mathbb{R}^m$ ein Vektor.*
Dann gibt es einen Vektor $x \in \mathbb{R}^d$, $x \geq 0$ mit $Ax = b$, g.d.w. $y^T b \geq 0$ für jeden Vektor $y \in \mathbb{R}^m$ mit $y^T A \geq 0$.

Interpretation von $(\exists x \geq 0 : Ax = b)$:

$$\left(\begin{array}{c|c|c|c} a_1 & a_2 & \cdots & a_m \end{array} \right) \cdot \begin{pmatrix} x_1 \\ \vdots \\ x_m \end{pmatrix} = a_1 \cdot x_1 + \cdots + a_m \cdot x_m = b$$

Da $x \geq 0$ bedeutet, dass b eine nicht negative Linearkombination der Spaltenvektoren von A ist.

Interpretation von $(\forall y : y^T A \geq 0 \Rightarrow y^T b \geq 0)$:



b liegt genau dann im Kegel, wenn jeder Halbraum, der alle a_i enthält, auch b enthält.

Beweis:

„⇒“: Sei $x \geq 0$ mit $Ax = b$. Sei y ein Vektor mit $y^T A \geq 0$.
Dann gilt $y^T b = \underbrace{y^T A}_{\geq 0} \underbrace{x}_{\geq 0} \geq 0$.

„⇐“: Sei $y^T b \geq 0$ für jedes y mit $y^T A \geq 0$.
Angenommen, es gibt kein $x \geq 0$ mit $Ax = b$. Seien $a_1, \ldots, a_m$ die Spalten von A. Dann ist $b \notin \text{cone}\{a_1, \ldots, a_m\}$ und somit gibt es nach dem Fundamentalsatz ein $y := c$ mit $y^T b < 0$ und $y^T A \geq 0$.

□

Lemma 1.4.10 (Variante Farkas' Lemma) *Sei A eine $(m \times d)$-Matrix und $b \in \mathbb{R}^m$ ein Vektor.*
Dann hat das lineare Gleichungssystem $Ax \leq b$ eine Lösung $x \in \mathbb{R}^d$ g.d.w. $y^T b \geq 0$ für jeden Vektor $y \in \mathbb{R}^m$ mit $y \geq 0$ und $y^T A = 0$.

Beweis: Sei A' die $(m \times 2d + m)$-Matrix $[I\ A\ (-A)]$ (I ist die Einheitsmatrix).

Behauptung: $Ax \leq b$ hat eine Lösung x g.d.w. $A'x' = b$ eine nicht negative Lösung $x' \in \mathbb{R}^{m+2d}$ hat (Beweis Übung).

Das Lemma folgt nun durch Anwendung des vorherigen Lemmatas auf $A'x' = b$.

□

Satz 1.4.11 (Dualitätssatz der linearen Programmierung) *Sei A eine $(m \times d)$-Matrix, $b \in \mathbb{R}^m$ und $c \in \mathbb{R}^d$ Vektoren. Dann gilt:*

$$\max\left\{c^T x \mid Ax \leq b\right\} = \min\left\{y^T b \mid y \geq 0, y^T A = c^T\right\},$$

sofern beide Mengen nicht leer sind.

Beweis:

„≤“: Wenn $Ax \leq b$ und $y \geq 0$ mit $y^T A = c^T$ gilt, dann $c^T x = y^T A x \leq y^T b$.
Wenn also beide Mengen nicht leer sind, dann gilt
$\max\left\{c^T x \mid Ax \leq b\right\} \leq \min\left\{y^T b \mid y \geq 0, y^T A = c^T\right\}$.

„≥“: Wir müssen noch zeigen, dass es x, y gibt, so dass $Ax \leq b$, $y \geq 0$, $y^T A = c^T$ und $c^T x \geq y^T b$, d.h. es gibt x, y mit $y \geq 0$ und

$$\begin{bmatrix} A & 0 \\ -c^T & b^T \\ 0 & A^T \\ 0 & -A^T \end{bmatrix} \cdot \binom{x}{y} \leq \begin{pmatrix} b \\ 0 \\ c \\ -c \end{pmatrix}$$

Nach dem vorherigen Lemma ist dies äquivalent zu $(y')^T b' \geq 0$ für jedes y' mit

$$(y')^T \cdot \begin{pmatrix} A & 0 \\ -c^T & b^T \\ 0 & A^T \\ 0 & -A^T \end{pmatrix} = 0 \text{ und mit } b' = \begin{pmatrix} b \\ 0 \\ c^T \\ -c^T \end{pmatrix}$$

Wir schreiben nun $y' = \begin{pmatrix} u \\ \lambda \\ v \\ w \end{pmatrix}$.

Dann lässt sich diese Bedingung auch als

$u^T A - \lambda c^T = 0$ und $\lambda b^T + v^T A^T - w^T A^T \geq 0$ schreiben.

$(y')^T b' \geq 0$ lässt sich auch als

$u^T b + v^T c - w^T c \geq 0$ schreiben.

Wenn wir also zeigen:

Für $u, \lambda, v, w \geq 0$ mit $u^T A - \lambda c^T = 0$ und $\lambda b^T + v^T A^T - w^T A^T = 0$ gilt
$u^T b + v^T c - w^T c \geq 0$,

so haben wir den Satz bewiesen.

Fall 1:

Wenn nun $\lambda > 0$, dann ist

$$u^T b = \lambda^{-1} \cdot \underbrace{\lambda \cdot b^T}_{= w^T A^T - v^T A^T} \cdot u = \lambda^{-1}(w^T - u^T) \cdot A^T \cdot u$$

$(A^T u)^T = u^T A = \lambda c^T$

$$\downarrow$$
$$= \lambda^{-1}\lambda(w^T - v^T)c$$
$$= (w^T - v^T)c$$

Wir haben gezeigt: $u^T b = (w^T - v^T) \cdot c \Rightarrow u^T b + v^T c - w^T c = 0$
$u^T b + v^T c - w^T c \geq 0$

Fall 2:

Wenn $\lambda = 0$, dann sei $Ax_0 \leq b$ und $y_0 \geq 0$ mit $y_0^T = c$
(x_0, y_0 existieren, da beide Mengen nicht leer sind).
Dann folgt $u^T b \geq \underbrace{u^T A}_{=0} \cdot x_0 = 0$

$$\geq (w^T - v^T)A^T \cdot y_0 = (w^T - v^T) \cdot c.$$

□

Äquivalente Formulierungen von linearen Programmen

(i) $\max\{c^T x \mid Ax \leq b\}$

(ii) $\max\{c^T x \mid x \geq 0, Ax \leq b\}$

(iii) $\max\{c^T x \mid x \geq 0, Ax = b\}$

(i) $\min\{c^T x \mid Ax \geq b\}$

(ii) $\min\{c^T x \mid x \geq 0, Ax \geq b\}$

(iii) $\min\{c^T x \mid x \geq 0, Ax = b\}$

Äquivalente Formulierungen des Dualitätssatzes

(i) $\max\{c^T x \mid x \geq 0, Ax \leq b\} = \min\{y^T b \mid y \geq 0, y^T A \geq c^T\}$

(ii) $\max\{c^T x \mid x \geq 0, Ax = b\} = \min\{y^T b \mid y^T A \geq c^T\}$

Eine Interpretation von Dualität

Beispiel 1.4.12 *Beispiel-LP*

$$\begin{array}{lll} \textit{I.} & \textit{max} & 3x_1 - x_2 + x_3 \\ & \textit{unter} & 4x_1 - x_2 + 2x_3 \leq 8 \\ & & x_1 + x_2 - x_3 \leq -2 \\ & & x_1, x_2, x_3 \geq 0 \end{array}$$

Sei z^* optimaler Lösungswert des LP.

Wie können wir zeigen, dass $z^* \geq \alpha$ für ein festes α?
Antwort: Finde eine zulässige Lösung mit Zielfunktionswert $\geq \alpha$.
Im Beispiel; $z^* \geq 0$, da $x = (0, 0, 4)$ zulässig mit Zielfunktionswert 4.
Wie können wir obere Schranke herleiten?
Da die $x_i \geq 0$ folgt

$$3x_1 - x_2 + x_3 \leq 4x_1 - x_2 + 2x_3$$

durch paarweises Vergleichen der Summanden.
Dann gilt $3x_1 - x_2 + x_3 \leq 4x_1 - x_2 + 2x_3 \leq 8$.
Es geht aber besser:

$$\begin{aligned} 3x_1 - x_2 + x_3 \leq 5x_1 + 0x_2 + x_3 &= (4x_1 - x_2 + 2x_3) + (x_1 + x_2 - x_3) \\ &\leq 8 + (-2) = 6. \end{aligned}$$

Wir können nun das Problem, eine minimale obere Schranke zu finden, auch wieder als LP formulieren:

$$
\begin{array}{lll}
\textit{I.} & \textit{max} & 3x_1 - x_2 + x_3 \\
& \textit{unter} & 4x_1 - x_2 + 2x_3 \leq 8 \quad | \cdot y_1 \\
& & x_1 + x_2 - x_3 \leq -2 \quad | \cdot y_2 \\
& & x_1, x_2, x_3 \geq 0
\end{array}
$$

$$
\begin{array}{lll}
\textit{II.} & \textit{min} & 8y_1 - 2y_2 \\
& \textit{unter} & 4y_1 + y_2 \geq 3 \\
& & -y_1 + y_2 \geq -1 \\
& & 2y_1 - y_2 \geq 1 \\
& & y_1, y_2 \geq 0
\end{array}
$$

Dabei machen wir uns zunutze, dass jede nicht negative Linearkombination von Bedingungen aus I, die Koordinatenweise größer ist als der Zielfunktionsvektor, eine obere Schranke an den Wert einer optimalen Lösung liefert.

Wichtig: Die y_i müssen ≥ 0 sein, da sich ansonsten die Ungleichung umdreht.

I wird auch als das primale und II als das duale LP bezeichnet.

Das duale LP kann als Suche nach der besten oberen Schranke interpretiert werden.

1.5 Der Simplex-Algorithmus

Wir wollen

$$\begin{array}{ll} \max & c^T x \\ \text{unter} & a_1^T x \leq \beta_1 \\ & a_2^T x \leq \beta_2 \\ & \vdots \\ & a_m^T x \leq \beta_m \end{array}$$

lösen.

Sei A die $(m \times d)$-Matrix mit Zeilenvektoren $a_i^T \in \mathbb{R}^d$ und $b = \begin{pmatrix} \beta_1 \\ \vdots \\ \beta_m \end{pmatrix}$

Annahme:
Wir kennen bereits eine zulässige Ecke x_0^* des Zulässigkeitsbereichs $\{x \mid Ax \leq b\}$.
Diese muss mindestens d Ungleichungen mit Gleichheit erfüllen.
Wähle nun eine solche Menge $A_0 x \leq b_0$ von d Ungleichungen aus $Ax \leq b$ aus, so dass A_0 nicht singulär ist und x_0^* die Ungleichung mit Gleichheit erfüllt.
Bestimme einen Vektor u, so dass $c^T = u^T A$ und u ist 0 für die Ungleichungen, die nicht in A_0 sind, d.h. wir berechnen $c^T \cdot A_0^{-1}$ und fügen Nullen hinzu.

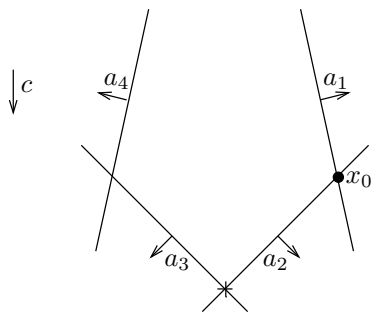
Fall 1:

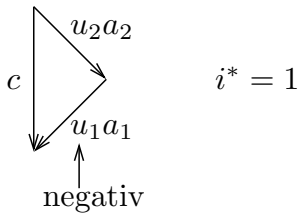
$u \geq 0$: Dann ist x_0 optimale Lösung, weil $c^T x_0^* = u^T A x_0^* = u^T b^*$,
wobei b^* ein Vektor ist mit $\beta_i = \beta_i^*$, falls Ungleichung i in A_0 auftaucht.
Da b an den Stellen, an denen u nicht 0 ist, mit b^* übereinstimmt, gilt weiter:

$$\begin{aligned} & u^T b^* = u^T b \\ & \geq \min\{y^T b \mid y \geq 0; y^T A = c^T\} \\ & \qquad\qquad \text{(da } c^T = u^T A \text{ und } u \geq 0 \text{ und } u \text{ somit in der Menge} \\ & \qquad\qquad \{y \mid y \geq 0; y^T A = c^T\} \\ & = \max\{c^T x \mid Ax \leq b\}. \\ & \uparrow \\ & \text{Dualitätssatz} \end{aligned}$$

Fall 2:

$u \not\geq 0$: Wähle den kleinsten Index i^*, für den u eine negative Komponente u_{i^*} hat.
Sei y der Vektor mit $a^T y = 0$ für jede Zeile von A_0 mit $a \neq a_{i^*}$ und
$a_{i^*}^T y = -1$ (d.h. y ist die entsprechende Spalte von $-A_0^{-1}$).





Anmerkung:
Für $\lambda \geq 0$ ist $x_0^* + \lambda y$ ein Strahl, der entweder

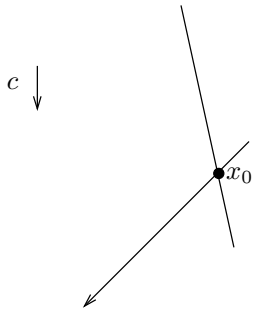
(a) eine Kante von P enthält,

(b) ein Strahl von P enthält, oder

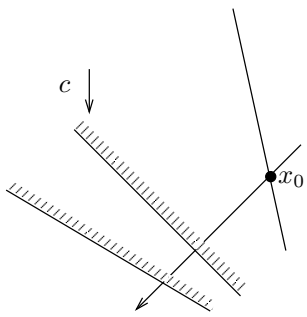
(c) außerhalb von P liegt.

Außerdem gilt $c^T y = u^T Ay = -u_{i^*} > 0$.

Fall 2a:
$a^T y \leq 0$ für jede Zeile von A. Dann ist $x_0^* + \lambda y$ in P für alle $\lambda \geq 0$ und das Maximum ist unbeschränkt.



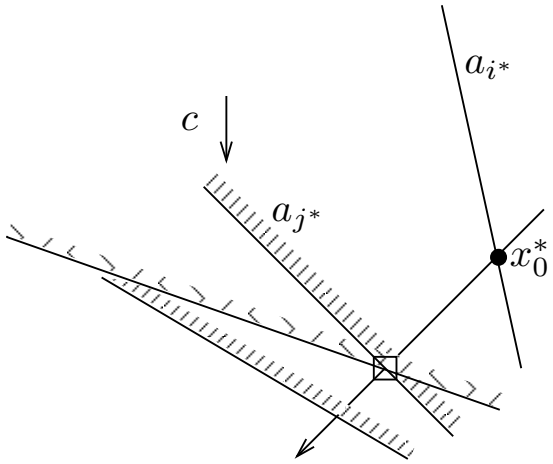
Fall 2b:
$a^T y > 0$ für eine Zeile von A.



Sei λ_0 das größte λ, so dass $x_0^* + \lambda y$ zu P gehört, d.h.

$$\lambda_0 := \min \left\{ \tfrac{\beta_j - a_j^T x_0^*}{a_j^T y} \mid j = 1, \ldots, m; a_j^T y > 0 \right\}.$$

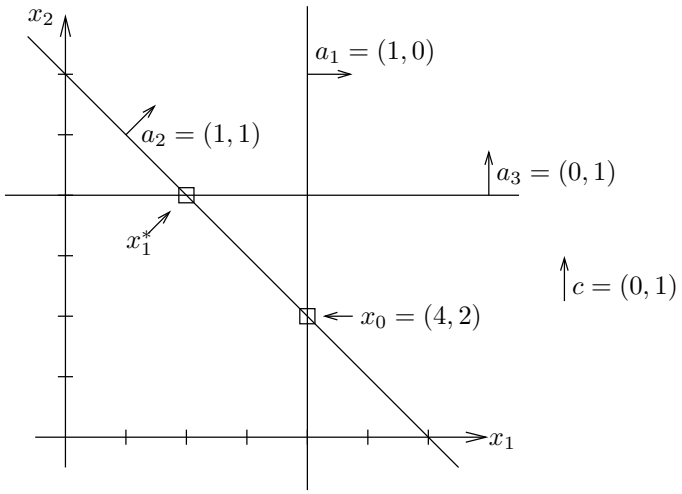
Sei j^* der kleinste Index, der dieses Minimum annimmt.



Es entstehe nun A_1 aus A_0 durch ersetzen von a_{i^*} durch a_{j^*} und es sei $x_1 = x_0^* + \lambda_0 \cdot y$. Damit ist $A_1 x_1^* = b_1$, wobei b_1 der Teil von b ist, der den Ungleichungen in A_1 entspricht. Wir starten die Iteration erneut, wobei A_1 und x_1^* die Werte A_0 und x_0^* ersetzen.

Beispiel 1.5.1

$$\begin{array}{lll} \max & & x_2 \\ \textit{unter} & & x_1 \leq 4 \\ & x_1 + & x_2 \leq 6 \\ & & x_2 \leq 4 \end{array}$$



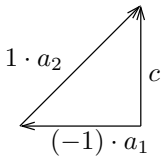
Für $x_0 = (4,2)$ *gilt:*
$a_1 \to x_1 = 4$
$a_2 \to x_1 + x_2 = 6$
$a_3 \to x_2 \leq 4$

$\rightarrow A_0 = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$, $b_0 = \begin{pmatrix} 4 \\ 6 \end{pmatrix}$, *das heißt* $A_0 x_0^* = b_0$.

Stelle $c^T = (0,1)$ *als Linearkombination der Zeilen von* A *dar. Verwende dabei nur die Zeilen aus* A_0*, das heißt berechne zunächst*

$$A_0^{-1} = \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix} \text{ und } c^T A_0^{-1} = (0,1) \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix} = (-1,1)$$

Damit folgt $c^T = u^T A$ *mit* $u^T = (-1,1,0)$*. Damit ist:*



D.h. um den Vektor c *darzustellen, muss man sich „entgegen“ der Beschränkungsrichtung von* a_1 *bewegen. Damit kann* a_1 *nicht relevant als einschränkende Ungleichung sein.*

Also ist $i^* = 1$*. Es folgt* $y = \begin{pmatrix} -1 \\ 1 \end{pmatrix}$.
Es gilt $c^T y = (0,1) \cdot \begin{pmatrix} -1 \\ 1 \end{pmatrix} = 1 > 0$.

D.h., wenn wir uns in Richtung y *bewegen, verbessern wir unsere Lösung.*
Außerdem gilt

$$x_1^* = \begin{pmatrix} 4 \\ 2 \end{pmatrix} + 2 \cdot \begin{pmatrix} -1 \\ 1 \end{pmatrix}, \text{ d.h. } \lambda_0 = 2.$$

Nun gilt für $x = x_1^*$*:*

$$\begin{aligned} x_1 &\leq 4 \\ x_1 + x_2 &= 6 \\ x_2 &= 4 \end{aligned}$$

Somit ist $A_1 = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$*. Wir stellen wieder* c^T *als Linearkombination der Zeilen von* A_1 *dar und erhalten*

$$c^T = u^T A \text{ mit } u^T = (0,0,1).$$

Damit ist die Lösung optimal.

Satz 1.5.2 *Der Simplex-Algorithmus (wie oben) terminiert.*

Beweis: Bezeichne mit $A_k x \leq b_k$, x_k, u_k, y_k das Untersystem von $Ax \leq b$, die Ecke und die Vektoren x und y aus der k-ten Iteration des Algorithmus.

Wegen $c^T y = u^T A y = -\nu_{i*} > 0$ (mit $\nu_{i*} = u_{i*}$ in unserem Algorithmus) folgt $c^T x_0^* \leq c^T x_1^* \leq c^T x_2^* \leq \ldots$ (*), wobei $c^T x_k^* = c^T x_{k+1}^*$ nur dann, wenn $x_k^* = x_{k+1}^*$.

Annahme: Die Methode terminiert nicht. Dann gibt es ein k, l, so dass $k < l$ und $A_k < A_l$, da es nur endlich viele Auswahlmöglichkeiten für A_k gibt. Also gilt auch $x_k^* = x_l^*$ und somit $x_k^* = x_{k+1}^* = \ldots = x_l^*$. Sei nun r der größte Index für den a_r aus A_t in einer Iteration $t = k, k+1, \ldots, l$ entfernt wurde. Nennen wir diese Iteration p. Da $A_k = A_l$ gilt, wissen wir, dass a_r in einer Iteration q, $k \leq q < l$ zu A_q hinzugefügt werden muss.

Es folgt, dass für $j > r$:

$$a_j \text{ erscheint in } A_p \;\Leftrightarrow\; a_j \text{ erscheint in } A_q$$

Nach (*) gilt $u_p^T A y_a = c^T y_q > 0$.
Damit gilt $\nu_{pj}(a_j^T y_q) > 0$ für mindestens ein j mit $u_p = (\nu_{p1}, \ldots, \nu_{pm})$.

Wenn a_j nicht in A_p auftaucht, ist $\nu_{pj} = 0$.

Wenn a_j in A_p auftaucht und $j > r$ ist, dann gilt $a_j^T \cdot y_p = a_j^T \cdot y_q = 0$.

Wenn a_j in A_p auftaucht und $j < r$ ist, dann ist $\nu_{pj} \geq 0$, da r zu Zeitpunkt p der kleinste Index ist mit $\nu_{pj} < 0$. Außerdem gilt $a_q^T y_p \leq 0$, da in Iteration q $r = j^*$ der kleinste Index j mit $a_j^T x_q^* = \beta_j$ und $a_q^T y_p > 0$.

Wenn a_j in A_p auftaucht und $j = r$ ist, dann gilt aus denselben Gründen $\nu_{pj} < 0$ und $a_j^T y_p > 0$.

Also ist $\nu_{pj}(a_j^T y_q) \leq 0$ für alle j. $\lightning$

□

Finden einer Anfangslösung

Idee: Finde zunächst LP, für das man eine Ecke kennt. Die Lösung dieses LPs ist Ecke des eigentlichen LPs.

Betrachte LP der Form

$$(**)\ \max\left\{c^T x \mid x \geq 0; Ax \leq b\right\}$$

Wir schreiben $Ax \leq b$ als $A_1 x \leq b_1$ und $A_2 x \geq b_2$ mit $b_1 \geq 0$ und $b2 > 0$, d.h. b_2 entspricht den negativen Einträgen in b.
Betrachte das LP

$$(***)\ \max\left\{(1, \ldots, 1) \cdot (A_2 x - \tilde{x}) \mid x, \tilde{x} \geq 0, A_1 x \leq b_1, A_2 x - \tilde{x} \leq b_2\right\}$$

Dann ist $x = 0, \tilde{x} = 0$ eine Ecke des Zulässigkeitsbereichs.

Daher wissen wir, dass wir (***) mit unserem Simplexalgorithmus lösen können.

Behauptung 1.5.3 $(***)$ *hat Maximum mit Wert* $(1,...,1)\cdot b_2$ *für eine Eckenlösung* $(x^*, \tilde{x}^*)$. $\Rightarrow$ x^* *ist Ecke des Zulässigkeitsbereichs von* $(**)$

Beweis: Wir zeigen zunächst die Zulässigkeit von x^*:

Es gilt $A_2x^* - \tilde{x}^* = b_2$ und somit (da $\tilde{x}^* \geq 0$) $A_2x^* \geq b_2$. Dies bedeutet, dass x^* zulässig für $(**)$ ist. Nun zeigen wir, dass x^* Ecke des Zulässigkeitsbereichs von $(**)$ ist. Da $(x^*, \tilde{x}^*)$ Ecke des ZLB von $(***)$ ist, gibt es ein System von $d+m'$ unabhängigen linearen Ungleichungen aus $A_1x \leq b_1$ und $A_2x - \tilde{x} \leq b_2$, das mit Gleichheit erfüllt ist; dabei bezeichne m' die Dimension von $\tilde{x}$. Da bei einem Zielfunktionswert von $(1,...,1)b_2$ $A_2x^* - \tilde{x}^* = b_2$ gilt, sind diese m' Ungleichungen alle mit Gleichheit erfüllt und es müssen noch d weitere Ungleichungen aus $A_1x \leq b$ mit Gleichheit erfüllt sein. Diese liefern eine Ecke für $(**)$.

Behauptung 1.5.4 *Ist der Zielfunktionswert kleiner als* $(1,...,1)\cdot b_2$, *so hat* $(**)$ *keine zulässige Lösung.*

Beweis: Wir zeigen: Wenn $(**)$ eine zulässige Lösung hat, dann ist der Zielfunktionswert $(1,...,1)\cdot b_2$. Ist $(**)$ zulässig mit Lösung x^*, dann gilt $A_1x^* \leq b_1$ und $A_2x^* \geq b_2$. Definiere $\tilde{x}^* = A_2x - b_2 \geq 0$. Dies impliziert $A_zx^* - \tilde{x}^* = b_2$. Also ist $(x^*, \tilde{x}^*)$ zulässig und hat Zielfunktionswert $(1,...,1)\cdot b_2$.

Bemerkung 1.5.5 *Laufzeit eines Sinplexschritts*

Berechne A_0^{-1}*:* $\mathcal{O}(d^3)$ *(*A_0 $(d\times d)$*-Matrix)*

Fülle u mit Nullen auf: $\mathcal{O}(m)$

Berechne λ_0*:* $\mathcal{O}(dm)$

$\Rightarrow$ *Insgesamt:* $\mathcal{O}(d^3 + dm)$ *Zeit*

Die Anzahl der Simplexschritte kann exponentiell groß sein.

1.6 Ellipsoidmethode

LP:

$$\begin{pmatrix} a_{11} & \cdots & a_{1d} \\ \vdots & & \vdots \\ a_{1m} & \cdots & a_{md} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ \vdots \\ x_d \end{pmatrix} \leq \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix}$$

Kodierung der Eingabe: $a_{11}\#a_{12}\#a_{13}\#...\#a_{1d}\#a_{2d}\#...\#a_{md}\#b_1\#...\#b_m$

Definition 1.6.1 *Sei* A *eine* $(m\times d)$*–Matrix über* $\mathbb{Z}$. *Als Größe von* A *definieren wir die Länge ihrer binären Kodierung*

$$L(A) = \sum_{i,j}(\log_2|a_{ij}| + 2),$$

$\lceil\log_2|a_{ij}|\rceil$ *Bits für* $|a_{ij}|$ *und ein Bit für das Vorzeichen.*

1.6.1 LP-Entscheidungsproblem

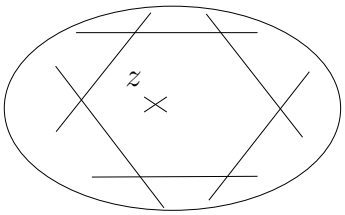
Definition 1.6.2 *Sei A eine $(m \times d)$–Matrix mit Einträgen aus $\mathbb{Z}$ und $b \in \mathbb{Z}^m$. Das Problem*

$$\exists x : Ax \leq b$$

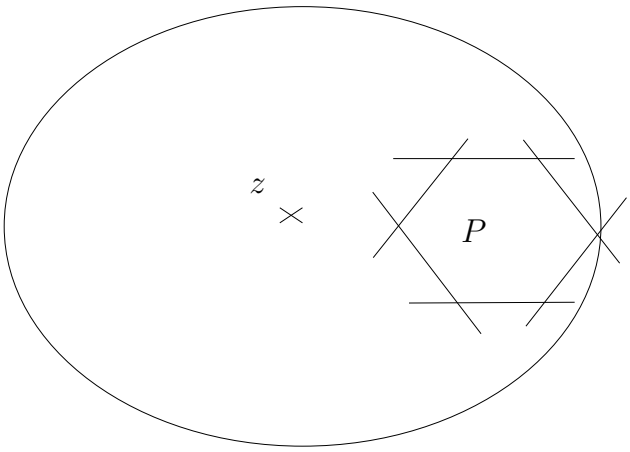
heißt LP-Entscheidungsproblem.

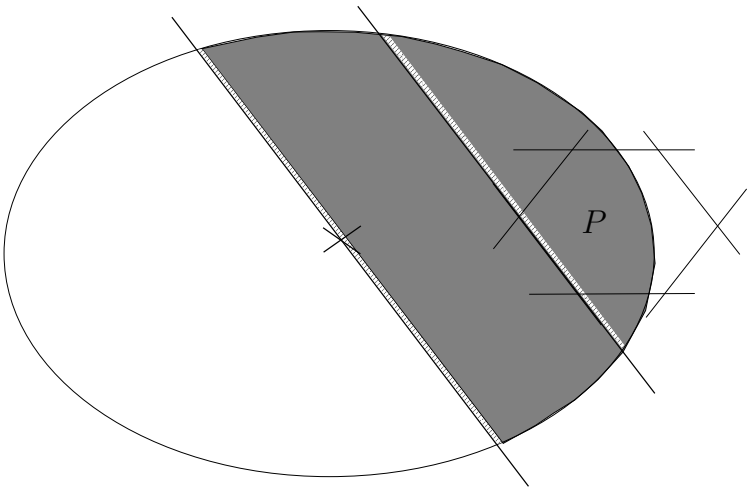
1.6.2 Ellipsoidmethode

Iterationsschritt
Sei P der Zulässigkeitsbereich, z Zentrum des Ellipsoids.
1. Fall: $z \in P$: In diesem Fall gilt $Az \leq b$, d.h. wir haben eine Lösung gefunden.



2. Fall: $z \notin P$: Dann gibt es eine separierende Ungleichung $a_i^T z > b_i$, da für alle $x \in P$ gilt $a_i^T x \leq b_i$.
(i) Schneide Ellipsoid E mit Halbraum H_i, der normal zu a_i liegt und z auf dem Rand hat.
(ii) Konstruiere ein Ellipsoid E' mit Zentrum z', das $E \cap H_i$ enthält und ein möglichst kleines Volumen hat.





Lemma 1.6.3 *Wenn LP-Entscheidungsprobleme der Form $\exists x : Cx \leq d$ in polynomieller Zeit gelöst werden können, dann kann man auch LP-Optimierungsprobleme* $\max c^T x, Ax \leq b$ *in polynomieller Zeit lösen.*

Beweis: Wir reduzieren das Optimierungsproblem auf ein LP-Entscheidungsproblem, dessen Eingabegröße nur polynomiell viel größer ist.
Gegeben sei nun ein LP-Optimierungsproblem

$$(*) \max c^T x, Ax \leq b$$

Wir wissen, dass aufgrund der Dualität gilt

$$\max\{c^T x \mid Ax \leq b\} = \min\{y^t b \mid y^T A = c^T, y \geq 0\},$$

falls keine der Mengen leer ist.
Sei nun

$$P = \{x \in \mathbb{R}^d \mid Ax \leq b\}$$

und

$$P' = \{(x, y) \in \mathbb{R}^{d+m} \mid Ax \leq b, y \geq 0, y^T A \leq c, c^T x \leq y^T b, -y^T A \leq -c, -c^T x \leq -y^T b\}$$

Algorithmus zum Lösen von $(*)$:

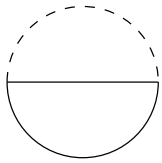
1. Teste, ob $P = \emptyset$. Falls ja, dann hat $(*)$ keine Lösung
2. Sonst, $P \neq \emptyset$. Teste, ob $P' = \emptyset$. Falls ja, dann ist P unbeschränkt.

   Andernfalls gibt es $(x_0, y_0) \in P'$ und x_0 ist aufgrund der Dualität eine optimale Lösung des Optimierungsproblems (denn x_0 ist primal zulässig und erreicht gleichen Zielfunktionswert wie eine duale zulässige Lösung y_0).

Das Entscheidungsproblem für P' ist polynomiell in der Eingabegröße, da sich die Bedingungen aus P' mit polynomiellem Mehraufwand darstellen lassen.

Warum approximiert man nicht mit Hilfe von Kugeln?
Es gibt keine Garantie, dass Kugel kleiner wird.

Beispiel 1.6.4 *Um eine halbe Kugel mit Radius r abzudecken, benötigt man eine Kugel mit Radius r:*



1.6.3 Anfangsellipsoid

Lemma 1.6.5 *Sei A eine $d \times d$-Matrix mit Einträgen aus $\mathbb{Z}$ mit Bitkomplexität $L = L(A) = \sum_{ij}(2 + \log_2 |a_{ij}|)$. Wir setzen $L' = L'(A) = L(A) + d \log d$.*
Dann gilt

$$|\det(A)| \leq 2^{L'}$$

Beweis: Es gilt: $\det(A) = \underbrace{\sum_{\pi}}_{\text{läuft über alle } d! \text{ Permutationen}} \operatorname{sign}(\pi) \cdot \prod_i a_{i\pi(i)} \in \mathbb{Z}$

Jeder Summand ist Produkt von d Elementen aus A (daher ist die Summe ganzzahlig, da $A \in \mathbb{Z}^{d \times d}$). Wenn man alle Elemente aus A aufmultipliziert, dann erhält man eine Zahl $\leq 2^L$, da $\log \left(\prod_{i,j} |a_{ij}| \right) = \sum_{ij} \lceil \log |a_{ij}| \rceil \leq L$ (und damit $2^{\log \left(\prod_{i,j} |a_{ij}| \right)} \leq 2^L \Leftrightarrow \prod_{ij} |a_{ij}| \leq 2^L$).

Das bedeutet:

$$\begin{aligned} \det(A) &\leq \sum_{\pi} \prod_i a_{i\pi(i)} \leq \sum_{\pi} \prod_{i,j} |a_{ij}| \\ &\leq \sum_{\pi} 2^L = d! 2^L \leq d^d \cdot 2^L = 2^{d \log d} \cdot 2^L = 2^{L'} \end{aligned}$$

□

Lemma 1.6.6 *Sei A eine invertierbare $d \times d$-Matrix aus $\mathbb{Z}$ und sei $b \in \mathbb{Z}^d$. Wir setzen $L := L(A, b)$, wobei (A, b) die $d \times (d+1)$-Matrix A mit zusätzlicher Spalte b ist. Sei ferner $L' = L + d \log d$ und $x = (x_1, \ldots, x_d)^T$ eine Lösung von $Ax = b$.*
Dann gibt es ganze Zahlen $D_1, \ldots, D_d$ und D mit

$$x_j = \frac{D_j}{D},\ |D_j| \leq 2^{L'},\ |D| \leq 2^{L'}$$

Beweis: Wir können die x_j mit der Cramerschen Regel ausrechnen.

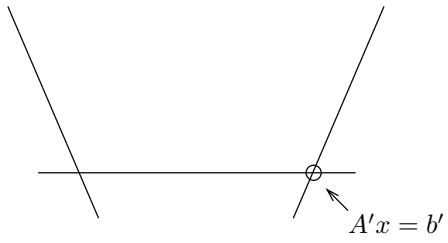
$$x_j = \frac{\det(B_j)}{\det(A)}, \text{ mit } B_j = (A_1, \ldots, A_{j-1}, b, A_{j+1}, \ldots, A_d),$$

d.h. um B_i zu erhalten ersetzt man die j-te Spalte von A durch b.
Mit Lemma 1.6.5 folgt die Behauptung.

□

Anfangsellipsoid

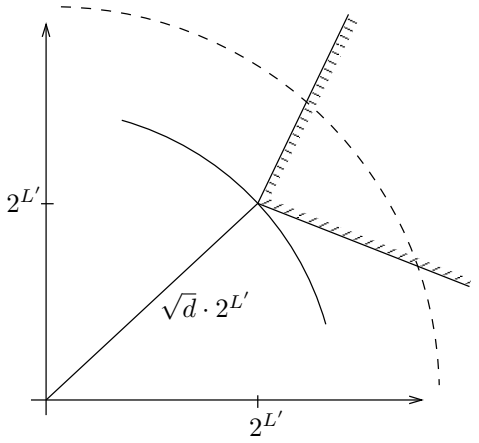
Sei $Ax \leq b$ eine Instanz eines LP-Entscheidungsproblems mit $A \in \mathbb{Z}^{m \times d}$.
Wir wissen, dass jede Ecke x des Zulässigkeitsbereichs ein quadratisches Teilsystem $A'x \leq b'$ mit Gleichheit erfüllt.



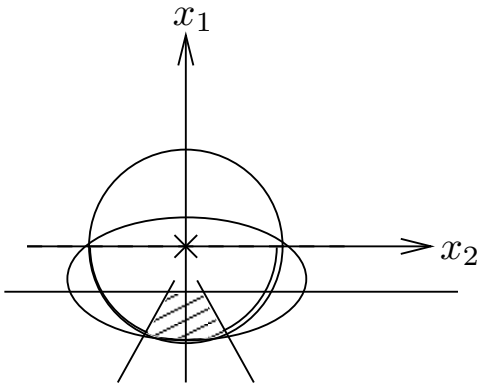
Nach Lemma 1.6.6 gilt dann für x:

$$|x_j| \leq 2^{L'},\ 1 \leq j \leq d.$$

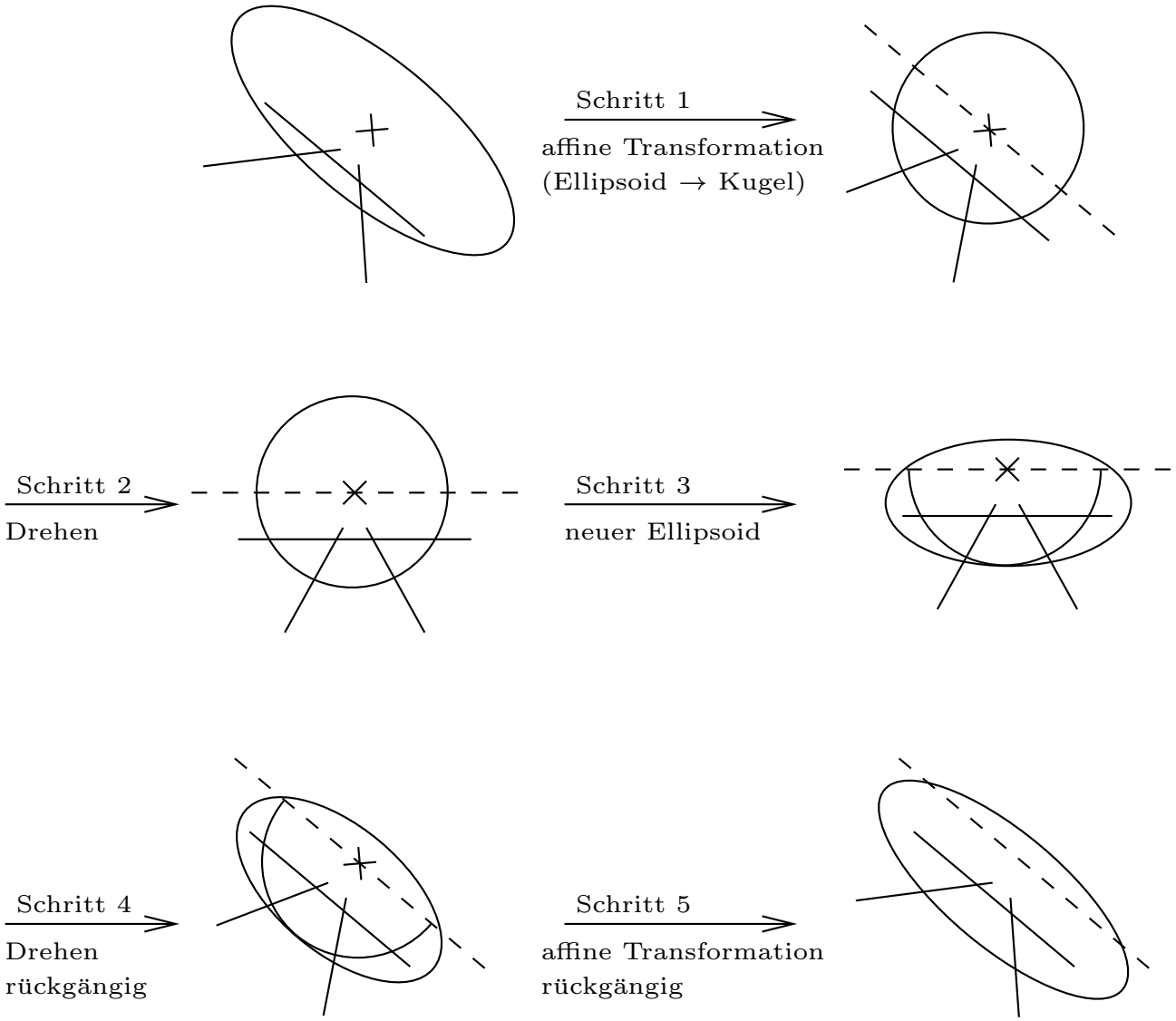
Wir wählen als Anfangsellipsoid E eine Kugel mit Zentrum im Ursprung und mit Radius $2^{L'} \cdot d$. Das heißt: Falls der Zulässigkeitsbereich nicht leer ist, so liegt eine Ecke in unserem Anfangsellipsoid.



Iterationsschritt:



Ablauf:

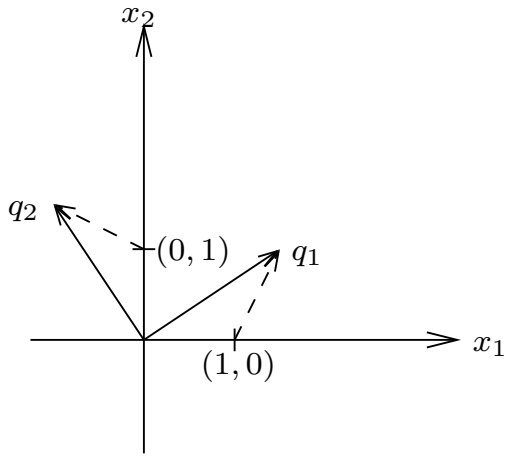


Definition 1.6.7 *Eine* affine Transformation T *ist eine Abbildung von* $\mathbb{R}^d$ *nach* $\mathbb{R}^d$ *mit* $T(x) = Q(x) + t$*, wobei* Q *eine invertierbare Matrix ist.*

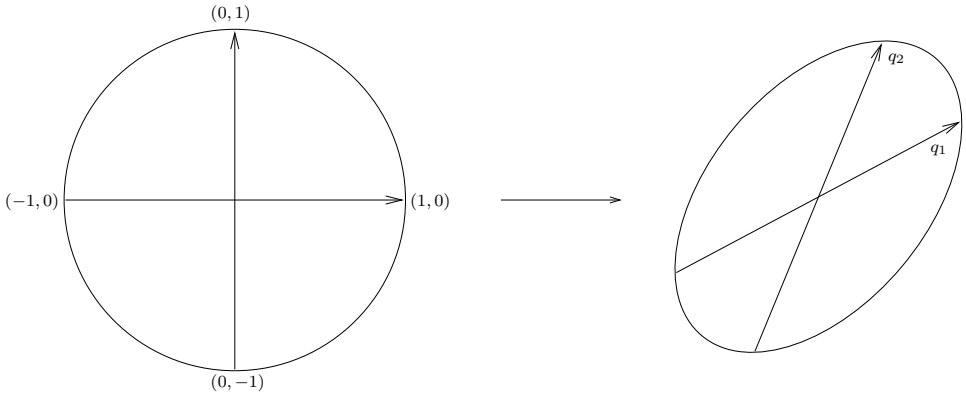
Veranschaulichung:

$$Qx = \left(\begin{array}{c|c|c|c} & & & \\ q_1 & q_2 & \cdots & q_d \\ & & & \end{array} \right) \cdot x = q_1x_1 + q_2x_2 + \ldots + q_dx_d$$

Auswirkung einer affinen Transformation auf die Einheitsvektoren:



Auswirkung einer affinen Transformation auf eine Kugel:



Definition 1.6.8 *Eine Kugel K mit Zentrum c und Radius r ist die Menge*

$$K(c,r) = \{x \mid (x-c)^T(x-c) \leq r^2\}$$

Definition 1.6.9 *Ein Ellipsoid E mit Zentrum z ist eine affine Transformation der Einheitskugel $K(0,1)$, d.h.*

$$E = \{y \mid (y-z)^T B^{-1}(y-z) \leq 1\}$$

Dabei muss für die Matrix B gelten:

$$B^{-1} = (Q^{-1})^T Q^{-1}$$

(d.h. B^{-1} ist eine symmetrische, positiv definierte Matrix).

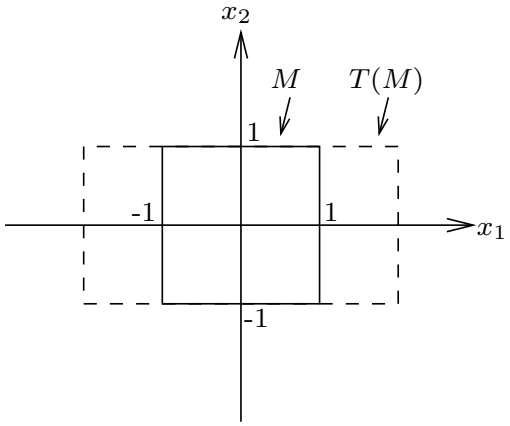
Bemerkung 1.6.10 *Die Einheitskugel ist die Menge $K(0,1) = \{x \mid x^T x \leq 1\}$.*
Sei T eine affine Transformation, d.h. $T(x) = Qx + t \Leftrightarrow x = Q^{-1}(T(x) - t)$.
Daraus folgt:

$$T(K(0,1)) = \left\{ \underbrace{y}_{=Qx+t} \mid x^T x \leq 1 \right\} = \left\{ y \mid (y-T)^T \underbrace{(Q^{-1})^T Q^{-1}}_{B^{-1}} (y-t) \leq 1 \right\}$$

Bemerkung 1.6.11 *Sei $T(x) = Qx + t$ eine affine Transformation und M ein Körper mit Volumen $vol(M) = \mu$.*
Dann gilt:

$$vol(T(M)) = \mu \cdot |\det(Q)|$$

Beispiel 1.6.12 *Sei M ein Quadrat mit Volumen 4, dessen Mittelpunkt im Ursprung liegt, und $Q = \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix}$, betrachte*
$T(x) = Qx + \overrightarrow{0}$, *d.h.* $\det(Q) = 2$.



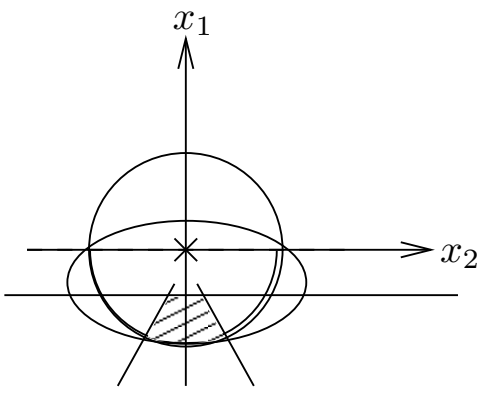
Beobachtung:

Affine Transformationen erhalten insbesondere Verhältnisse von Volumen. Z.B. seien vol(M) und vol(N) Volumen der Körper M und N und T eine affine Transformation; dann gilt:

$$\frac{\text{vol}(M)}{\text{vol}(N)} = \frac{\text{vol}(T(M))}{\text{vol}(T(N))}.$$

Iteration(Spezialfall):

Ellipsoid $E = K(0,1)$ und $a = e_1 = (1, 0, \ldots, 0)^T$, wobei a die Normale der verletzten Bedingung ist.



Bedingungen für den neuen Ellipsoid E'

Das Zentrum z' muss in der Halbebene $\{x \mid x_1 \leq 0\}$ liegen.

E' muss die Halbkugel $HK = \{x \in K(0,1) \mid x_1 \leq 0\}$ enthalten.

E' soll möglichst klein sein, d.h. wir suchen ein möglichst kleines Ellipsoid, das die Punkte $-e_1, e_2, e_3, \ldots, e_n, -e_2, \ldots, -e_d$ als Randpunkte beinhaltet.
So ein Ellipsoid hat sein Zentrum auf der Geraden $x_2 = x_3 = \ldots = x_d = 0$.

Spezialfall: Finde Zentrum $z' = t \cdot e_1$ $(t < 0)$ des neuen Ellipsoids sowie Streck-/Stauchfaktoren.

E' ist in Richtung x_1 gestaucht und in allen anderen Richtungen um einen Faktor gestreckt.

$$E' = \left\{ x \mid (x - z')^T \cdot \underbrace{\begin{pmatrix} \frac{1}{a^2} & 0 & 0 & \cdots & 0 \\ 0 & \frac{1}{b^2} & & & 0 \\ \vdots & & \ddots & & \\ 0 & 0 & \cdots & & \frac{1}{b^2} \end{pmatrix}}_{=(Q^{-1})^T Q^{-1}} (x - z') \leq 1 \right\}, \left(Q = \begin{pmatrix} a & & & 0 \\ & b & & \\ & & \ddots & \\ 0 & & & b \end{pmatrix} \right)$$

Die Ellipsoidgleichung ist für $-e_1, e_2, ..., e_d$ mit Gleichheit erfüllt.

$$\underbrace{(-e_1 - te_1)^T}_{-\begin{pmatrix} 1+t \\ 0 \\ \vdots \\ 0 \end{pmatrix}} \begin{pmatrix} \frac{1}{a^2} & 0 & 0 & \cdots & 0 \\ 0 & \frac{1}{b^2} & & & 0 \\ \vdots & & \ddots & & \\ 0 & 0 & \cdots & & \frac{1}{b^2} \end{pmatrix} (-e_1 - te_1) = 1$$

Also gilt $\frac{-(1+t)\cdot(-(1+t))}{a^2} = 1 \Leftrightarrow \frac{(1+t)^2}{a^2} = 1 \Leftrightarrow \frac{1}{a^2} = \frac{1}{(1+t)^2}$

$$\underbrace{(-e_2 - te_1)^T}_{\begin{pmatrix} -t \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}} \begin{pmatrix} \frac{1}{a^2} & 0 & 0 & \cdots & 0 \\ 0 & \frac{1}{b^2} & & & 0 \\ \vdots & & \ddots & & \\ 0 & 0 & \cdots & & \frac{1}{b^2} \end{pmatrix} (-e_2 - te_1) = 1$$

Also gilt: $\frac{t^2}{a^2} + \frac{1}{b^2} = 1 \Rightarrow \frac{1}{b^2} = \frac{1+2t}{(1+t)^2}$

Wir haben also $\frac{1}{a^2} = \frac{1}{(1+t)^2}$ und $\frac{1}{b^2} = \frac{1+2t}{(1+t)^2}$.

Welche Wahl für t minimiert das Volumen von E'?

$vol(E') = vol(E) \cdot \underbrace{a \cdot b^{d-1}}_{\mu(t)}$

$\mu^2(t) = a^2 + b^{2(d-1)} = (1+t)^2 + \left(\frac{(1+t)^2}{1+2t}\right)^{d-1} = \frac{(1+t)^{2d}}{(1+2t)^{d-1}}$

Ableiten und gleich 0 setzen liefert:

$t = -\frac{1}{d+1} \Rightarrow a = \frac{d}{d+1}$, $b = \frac{d}{\sqrt{d^2-1}}$

Sei $\gamma_d^2 = \left(\frac{d}{d+1}\right)^2 \cdot \left(\left(\frac{d}{\sqrt{d^2-1}}\right)^2\right)^{d-1} = \left(\frac{d}{d+1}\right)^2 \cdot \left(\frac{d^2}{d^2-1}\right)^{d-1} = \left(1 - \frac{1}{d+1}\right)^2 \cdot \left(\frac{(1-\frac{1}{d+1})^2}{1-\frac{2}{d+1}}\right)^{d-1}$

[die letzte Gleichung gilt, da $\frac{d^2}{d^2-1} = \frac{d^2}{(d+1)(d-1)} = \frac{d^2(d+1)}{(d+1)^2(d-1)} = \frac{\left(\frac{d}{d+1}\right)^2}{\left(\frac{d-1}{d+1}\right)} = \frac{\left(1-\frac{1}{d+1}\right)^2}{1-\frac{2}{d+1}}$]

Setze $v = d+1$:

$$\begin{aligned}\frac{\left(\frac{1}{v}\right)^2}{1-\frac{2}{v}} &= \frac{1-\frac{2}{v}+\frac{1}{v^2}}{1-\frac{2}{v}} \\ &= 1 + \frac{\frac{1}{v^2}}{1-\frac{2}{v}} = 1 + \frac{1}{v^2-2v} \\ &= 1 + \frac{1}{v(v-2)} = 1 + \frac{1}{(d+1)(d-1)}\end{aligned}$$

d.h. $\gamma_d^2 = \underbrace{\left(1 - \frac{1}{d+1}\right)^2}_{e^{-\frac{z}{d+1}}} \underbrace{\left(1 + \frac{1}{(d-1)(d+1)}\right)^{d-1}}_{e^{\frac{1}{d+1}}}$

$\leq e^{-\frac{z}{d+1}} \cdot e^{\frac{1}{d+1}}$, da $(1+x)^a \leq e^{ax} \forall a > 0$

$= e^{-\frac{1}{d+1}}$

$\Rightarrow \gamma_d \leq e^{-\frac{1}{z(d+1)}} = \frac{1}{e^{\frac{1}{z(d+1)}}} < 1$

Das bedeutet, dass man das Volumen mit einer polynomiellen Anzahl von Iterationen mehr als halbieren kann. Wiederholt man dies polynomiell oft, schrumpft das Volumen exponentiell.

Iteration(Allgemein):
Sei nun $E = \{x \mid (x-z)^T B^{-1}(x-z) \leq 1\}$ ein beliebiger Ellipsoid und sei a ein beliebiger Richtungsvektor. Sei $H = \{x \mid a^T(x-z) \leq 0\}$ der von a induzierte Halbraum.

(i) Transformiere E durch Transformation T^{-1} zur Einheitskugel $K(0,1)$

(ii) Rotiere den Halbraum H durch eine Rotation R^{-1} so, dass das Bild von H mit $\{x \mid x_1 \leq 0\}$ zusammenfällt.

(iii) Bilde $\hat{E}$ mit Zentrum $\hat{z}$ wie im Spezialfall angegeben.

(iv) Mache die Transformation (ii) und (i) rückgängig und bilde so E' mit Zentrum z'.

E ist nach Definition eine affine Transformation der einheitskugel $E = T(K(0,1))$ mit $T(x) = Qx + z$, Q invertierbar.
Die zu T inverse Transformation bildet E auf $K(0,1)$ ab, d.h. $T^{-1}(E) = K(0,1)$.

Außerdem gilt $T^{-1}(H) = \{x \mid (Q^T a)^T \cdot x \leq 0\}$, da

$$\begin{aligned} T^{-1}(H) &= T^{-1}(\{x \mid a^T(x-z) \leq 0\}) = \{T^{-1}(x) \mid a^T(x-z) \leq 0\} \\ &= \{x \mid a^T(T(x)-z) \leq 0\} = \{x \mid a^T(Qx+z-z) \leq 0\} \\ &= \{x \mid a^T Qx \leq 0\} = \{x \mid (Q^T a)^T x \leq 0\} \end{aligned}$$

Rotation R^{-1} : $R^{-1}\left(\frac{Q^T a}{||Q^T a||}\right) = e_1$ und damit auch $\frac{Q^T a}{||Q^T a||} = R_{e_1}$

Wir sind jetzt im Spezialfall und daher ist das neue Zentrum $\hat{z} = t \cdot e_1 = -\frac{1}{d+1} \cdot e_1$.
Mache R^{-1} rückgängig und erhalte $R\hat{z} = -\frac{1}{d+1} R_{e_1} = -\frac{1}{d+1} \frac{Q^T a}{||Q^T a||}$.
Wende nun die inverse Transformation zu T^{-1} an und erhalte für das neue Zentrum z':

$$z' = T(R\hat{z}) = -\tfrac{1}{d+1} \tfrac{Q \cdot Q^T a}{||Q^T a||} + z.$$

Wir haben damit eine direkte Formel für das neue Zentrum gefunden, so dass man z' berechnen kann ohne R zu berechnen.
Ähnlich: $B' = \frac{d^2}{d-1}\left(B - \frac{2}{d+1} \cdot \frac{Ba(Ba)^T}{a^T Ba}\right)$ [Übung!].

Beobachtung 1.6.13 *Eine Iteration erfordert nur polynomielle Zeit.*

Wieviele Iterationen?
Wir wissen, dass jede Ecke des Polytops eine maximale Entfernung $\sqrt{a} \cdot 2^{L'}$ vom Ursprung hat. Unser Anfangsellipsoid hat Durchmesser $d \cdot 2^{L'}$ und enthält auf jeden Fall alle Ecken.
Verschiedene Fälle können auftreten:

i) Der Zulässigkeitsbereich hat Dimension $< d$ und deshalb Volumen 0, obwohl er evtl. nicht leer ist. Man kann zeigen, dass man jedes LP so umformen kann, dass der Zulässigkeitsbereich positives Volumen hat (wenn er nicht leer ist). Diesen Fall betrachten wir nicht weiter.

ii) Fall i) liegt nicht vor und der Zulässigkeitsbereich ist beschränkt. Hier werden wir zeigen, dass die Ecken nicht beliebig weit voneinander entfernt sind, so dass das Polytop ein gewisses Mindestvolumen hat.

iii) Fall i) liegt nicht vor und der Zulässigkeitsbereich ist unbeschränkt. Füge Ungleichungen der Form $x_i \leq 2^{L'}$ und $x_i \geq -2^{2^{L'}}$ ein, so dass der Zulässigkeitsbereich beschränkt ist und wir in Fall ii) sind. Dabei wird die Bitkomplexität der Eingabe nur polynomiell erhöht.

Wir interessieren uns also nur für die Details von Fall ii), d.h. wir benötigen obere Schranke für das Volumen des Startellipsoids und eine untere Schranke für das Volumen von $\hat{P} = P \cap W_d$, wobei W_d der Würfel mit dem Ursprung als Mittelpunkt und Seitenlänge $2 \cdot 2^{L'}$ ist.
Wir wollen jetzt zeigen:

Volumen des Startellipsoids $\leq 2^{poly(L')}$

Volumen von $\hat{P} \geq 2^{-poly(L')}$

pro $2(d+1)$ Iterationen halbiert sich das Volumen des Ellipsoids

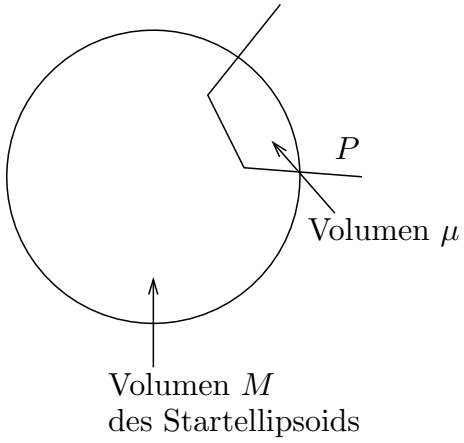
Dabei ist L' polynomiell in der Eingabegröße.

1.6.4 Laufzeit der Ellipsoidmethode

Sei P nicht leer und $\text{vol}(P) \neq 0$. Wir führen folgende Bezeichnungen ein:
E_l: Ellipsoid nach l Iterationen
μ: untere Schranke für $vol(P \cap E_0)$,
M : obere Schranke für $\text{vol}(E_0)$

Dann gilt: $\mu \leq \text{vol}(E_l) \leq \gamma^l \cdot \text{vol}(E_0) \leq \gamma^l \cdot M$ (da jedes Ellipsoid E_l den Schnitt von Zulässigkeitsbereich und E_0 enthält)

Faktor, um den sich das Volumen des Ellipsoids in einer Runde verkleinert



Wenn Schnitt nicht leer, dann terminiert der Algorithmus nach $\leq l$ Runden, wenn $\gamma^l \cdot M < \mu$. Auflösen nach l liefert:

$$\begin{aligned} & \gamma^l M < \mu \\ \Leftrightarrow & \gamma^l < \tfrac{\mu}{M} \\ \Leftrightarrow & l > \tfrac{\ln \frac{\mu}{M}}{\ln \gamma} \end{aligned}$$

Es gilt $\ln \gamma < -\frac{1}{2(d+1)}$, da $\gamma \leq e^{-\frac{1}{2(d+1)}}$.

Damit ist die kleinste Zahl, die die obige Ungleichung erfüllt, eine obere Schranke für die Anzahl der Iterationen. Und zwar $l > 2(d+1) \cdot \frac{M}{\mu}$.

Wir können o.B.d.A. annehmen, dass P beschränkt ist und komplett im Anfangsellipsoid liegt.
Ansonsten können wir noch Ungleichungen hinzufügen, die den „Anfangswürfel“ einschließen. Dieses erhöht die Eingabekomplexität nur um einen polynomiellen Faktor.

Bestimmung von M:
$vol(E_0) < M :=$ Volumen eines Würfels mit Seitenlänge $2 \cdot d \cdot 2^{L'}$ (dieser schließt das Anfangsellipsoid ein).
Also $M = (d \cdot 2^{L'+1})^d = d^d \cdot 2^{d(L'+1)}$

Bestimmung von μ:
Da $vol(P) \neq \emptyset$ ist, gibt es $d+1$ Ecken $v_0, \ldots, v_d$, die nicht alle in einer Hyperebene liegen und einen Simplex S aufspannen, dabei gilt

$$vol(S) \leq vol(P).$$

Volumen eines Simplex im $\mathbb{R}^d$:

$$vol(S) = \left| \frac{1}{d!} \cdot \begin{vmatrix} 1 & v_0^T \\ \vdots & \vdots \\ 1 & v_d^T \end{vmatrix} \right|$$

Jede Ecke ist Lösung eines Teilsystems von $Ax = b$ und daher gilt

$$v_{ij} = \frac{D_j^{(i)}}{D_i},\ 1 \leq i,j \leq d,\ |D_j^{(i)}| \leq 2^{L'},\ |D_i| < 2^{L'}$$
(siehe Lemma ...).

$$\begin{aligned}
\Rightarrow vol(S) &= \left| \frac{1}{d!} \cdot \begin{vmatrix} 1 & \frac{D_1^{(0)}}{D_0} & \cdots & \frac{D_d^{(0)}}{D_0} \\ \vdots & & & \vdots \\ 1 & \frac{D_1^d}{D_d} & \cdots & \frac{D_d^{(d)}}{D_d} \end{vmatrix} \right|, D_0, \ldots, D_d \neq 0 \\
&= \left| \frac{1}{d!} \frac{1}{D_0 \cdot D_1 \cdot \cdots \cdot D_d} \cdot \begin{vmatrix} D_0 & \frac{D_1^{(0)}}{D_0} & \cdots & \frac{D_d^{(0)}}{D_0} \\ \vdots & & & \vdots \\ D_d & \frac{D_1^d}{D_d} & \cdots & \frac{D_d^{(d)}}{D_d} \end{vmatrix} \right| \\
&\neq 0, da \begin{pmatrix} 1 & v_0^T \\ \vdots & \vdots \\ 1 & v_d^T \end{pmatrix} \text{ linear unabhängig} \\
&\quad \textit{da } v_0, \ldots, v_d \textit{ nicht auf einer Hyperebene liegen.}
\end{aligned}$$

Da die $D_j^{(i)}$ und die D_i alle ganzzahlig sind, folgt zusammen

$$\begin{Vmatrix} D_0 & D_1^{(0)} & \cdots & D_d^{(0)} \\ \vdots & & & \vdots \\ D_d & D_1^{(d)} & \cdots & D_d^{(d)} \end{Vmatrix} \geq 1$$

Außerdem gilt $|D - i| < 2^{L'} \Leftrightarrow \frac{1}{|D_i|} > 2^{-L'}$
und somit $vol(S) > \frac{1}{d!} \cdot 2^{-L' \cdot (d+1)} \cdot 1 > \frac{1}{d^d} \cdot 2^{-L'(d+1)}$

Bestimmung von l:
Es folgt:

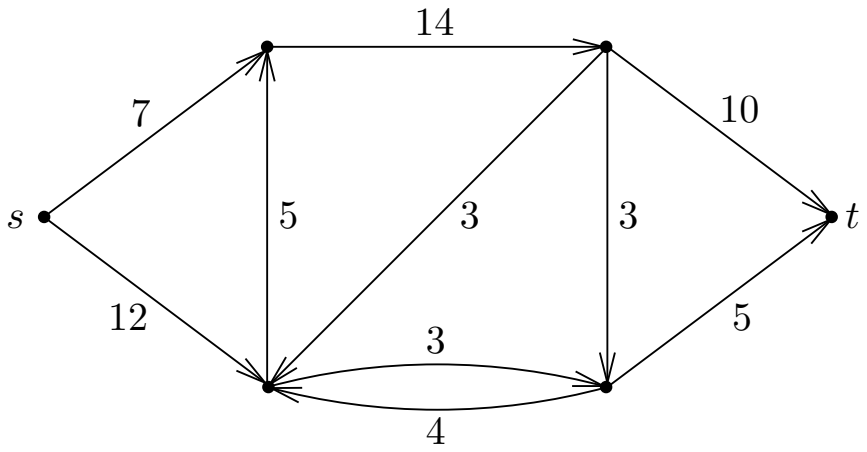
$$\begin{aligned} l &> 2(d+1) \cdot \ln \frac{M}{\mu} \\ &= 2(d+1) \cdot \ln \frac{d^d \cdot d^d}{2^{-d(L'+1)} 2^{-l'(d+1)}} \\ &= 2(d+1) \cdot \ln(2^{L'(2d+1)-d} \cdot d^{2d}) \\ &= 2(d+1)\big(\frac{L'(2d+1)-d}{\log_2 e} + 2d \log d\big) \\ &= \mathcal{O}(L'd^2 + d^2 \log d) \end{aligned}$$

Das ist polynomiell in der Eingabegröße.
Damit ist das LP-Entscheidungsproblem in P. Es gibt auch weitere polynomielle Verfahren (sog. Innere-Projekt-Methoden).

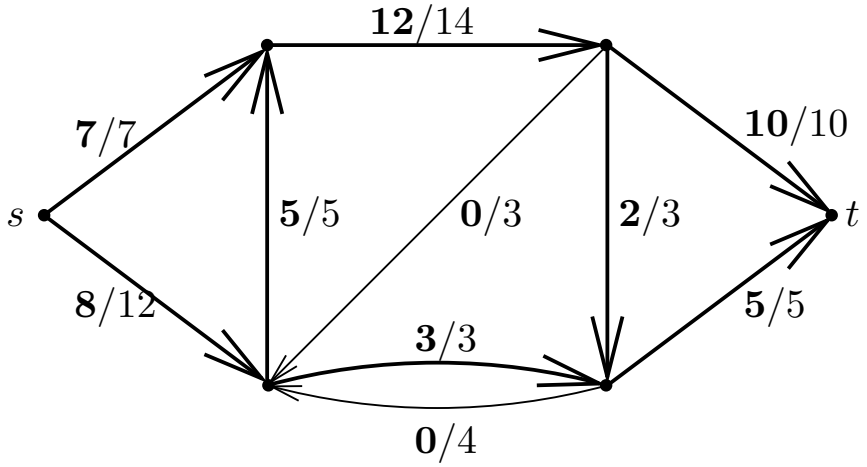
2 Approximationsalgorithmen

2.1 MaxFlow-MinCut-Theorem über Dualität

Beispiel 2.1.1 *Beispielgraph*



Maximaler Fluss



Der Fluss ist maximal, da beide eingehenden Kanten von t ausgelastet sind.

Problem 2.1.2 (Maximaler Fluss:) *Gegeben ist ein gerichteter Graph $G = (V, E)$ sowie zwei Knoten s (Quelle) und t (Senke), positive Kapazitäten $c : E \to \mathbb{R}^+$.*
Aufgabe: Berechne maximalen $s - t-$Fluss, der folgende Bedingung erfüllt:

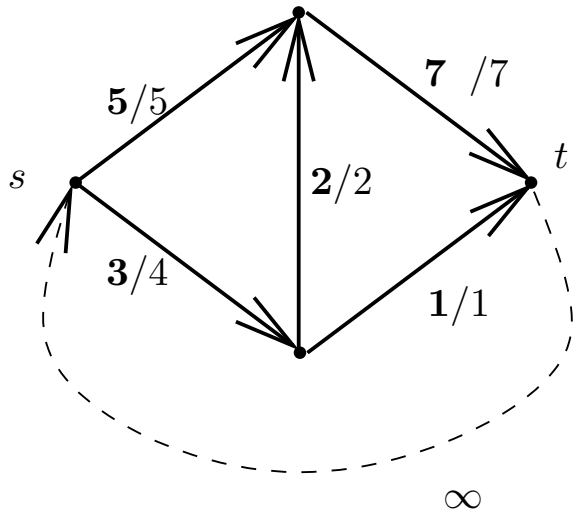
(1) (Kapazitätsbedingung)
Für jede Kante e ist der Fluss durch e maximal ihre Kapazität.

(2) (Flusserhaltung)
An jedem Knoten v außer s und t ist der gesamte eingehende Fluss gleich dem gesamten ausgehenden Fluss.

Definition 2.1.3 (s-t-Schnitt:) *Ein s-t-Schnitt ist definiert durch eine Partition von V in X und $\overline{X} = V \setminus X$, so dass $s \in X$ und $t \in \overline{X}$ und besteht aus allen Kanten von X nach $\overline{X}$. Die Kapazität eines s-t-Schnitts bezeichnen wir mit $c(X, \overline{X})$ und definieren sie als Summe der Kapazitäten der Schnittkanten.*

2.1.1 MaxFluss als LP:

Füge Kante von t nach s mit Kapazität ∞ hinzu.



Maximiere f_{ts}

unter	$f_{ij} \leq c_{ij}$,	$[i,j\rangle \in E \backslash \{[t,s\rangle\}$	$\cdot d_{ij}$	(Kapazitätsbedingung)
	$\underbrace{\sum_{j:[j,i\rangle \in E} f_{ji}}_{\text{eingehender Fluss}} - \underbrace{\sum_{j:[i,j\rangle \in E} f_{ij}}_{\text{ausgehender Fluss}} \leq 0$	$i \in V$	$\cdot p_i$	(Flusserhaltung)
	$f_{ij} \geq 0$	$[i,j\rangle \in E$		

Warum genügt die zweite Ungleichung? (Eigentlich brauchen wir Gleichheit!) Wenn für alle Knoten der Ausgangsfluss höchstens so groß ist wie der Eingangsfluss, dann muss Gleichheit gelten.
Außerdem lassen wir die Bedingung $f_{ts} \leq \infty$ weg. Sie stellt keine wirkliche Einschränkung dar.

Das duale LP

$$
\begin{aligned}
\text{(D) } \min & \sum_{[i,j\rangle \in E} c_{ij} \cdot d_{ij} \\
\text{unter } & d_{ij} - p_i + p_j \geq 0, \qquad [i,j\rangle \in E \\
& p_s - p_t \geq 1 \\
& d_{ij} \geq 0 \\
& p_i \geq 0
\end{aligned}
$$

Da $f_{ts} \leq \infty$ im primalen LP weggelassen wurde, erhalten wir statt $d_{ts} - p_t + p_s \geq 1$ vereinfacht $-p_t + p_s \geq 1$. Hätten wir $f_{ts} \leq \infty$ nicht weggelassen, würde die Zielfunktion den Summanden $\infty \cdot d_{ts}$ enthalten, woraus folgen würde, dass $d_{ts} = 0$ ist.

ILP
Fordere zusätzliche zu (D), dass

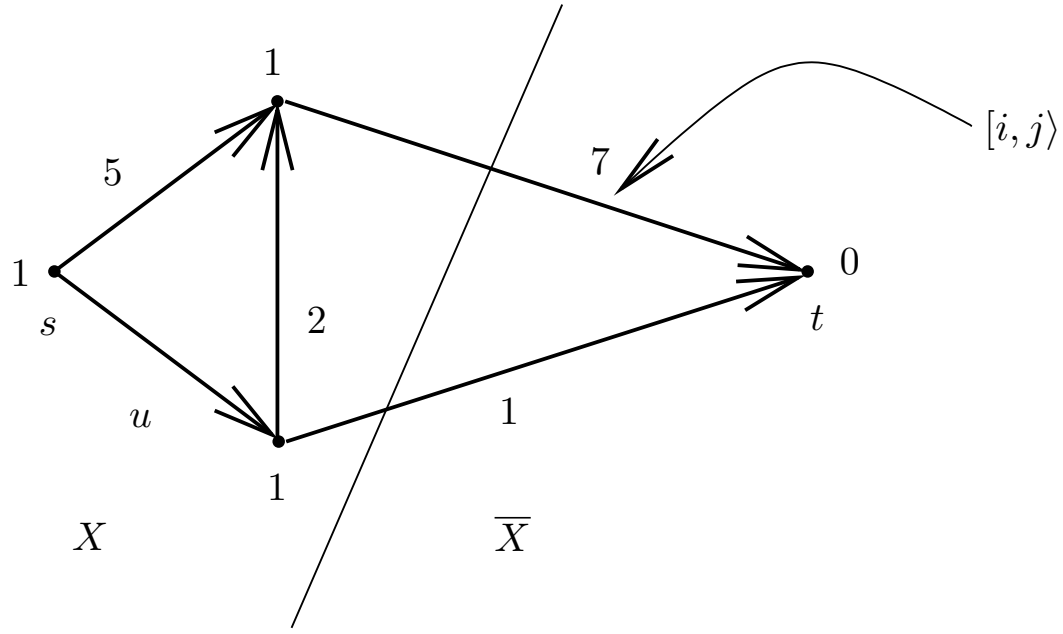
$$p_i \in \{0,1\}$$
$$d_{i,j} \in \{0,1\}$$

gilt.

Lemma 2.1.4 *Der optimale Zielfunktionswert des ILP ist der Wert eines minimalen Schnitts.*

Beweis: Sei (d^*, p^*) eine optimale Lösung des ILP.
Dann folgt $p_s^* = 1$ und $p_t^* = 0$, da $p_s - p_t \geq 1$ gelten muss.

$\Rightarrow$ Die Lösung (d^*, p^*) ist ein $s-t$-Schnitt mit $X = \{i : p_i = 1\}$ und $\overline{X} = \{i : p_i = 0\}$
(Das gilt sogar für jede zulässige Lösung des ILP.)



Betrachte Schnittkante $[i,j\rangle$, d.h. $i \in X$ und $j \in \overline{X}$. Die Lösung muss $d_{ij} - p_i + p_j \geq 0$ erfüllen. Wir wissen, dass $p_i = 1$ und $p_j = 0$ gilt, da wir X und $\overline{X}$ gerade so definiert haben.

Dies impliziert $\quad d_{ij}^* - 1 + 0 \geq 0$
$\Leftrightarrow d_{ij}^* \geq 1$
$\Rightarrow d_{ij}^* = 1$, da $d_{ij}^* \in \{0,1\}$

(Auch dieses gilt für jede zulässige Lösung).
Um die Zielfunktion zu minimieren, müssen alle anderen d_{ij} auf Null gesetzt werden (hier benötigen wir jetzt, dass (d^*, p^*) eine optimale Lösung ist.)
$\Rightarrow$ Zielfunktion ist gerade die Kapazität des Schnitts, und $(X, \overline{X})$ ist damit ein minimaler Schnitt.

Was ist mit LP?
Ersetze $d_{ij} \in \{0,1\}, p_i \in \{0,1\}$ durch $0 \leq d_{ij} \leq 1$ und $0 \leq p_i \leq 1$. Die oberen Schranken sind redundant.
$\Rightarrow$ altes LP (Relaxierung des ILP.)
Wir können eine Lösung des LP als fraktionalen $s-t$-Schnitt betrachten:
Die Distanz der Bezeichner p_i auf jedem $s-t$-Pfad $(p_0, ..., p_k)$ addiert sich wegen $\sum_{i=0}^{k-1}(p_i -$

$p_{i+1}) = p_s - p_t$ zu 1 auf.
Im Fall des $s-t$-Schnitts ist der beste ganzzahlige Schnitt genauso gut wie der beste fraktionale Schnitt (das Polyeder, das den Zulässigkeitsbereich beschreibt, hat eine ganzzahlige Ecke).

2.2 Set Cover über randomisiertes Runden

Definition 2.2.1 (Problem (Set Cover)) *Gegeben ist ein Universum U mit n Elementen und eine Menge $T = \{S_1, ..., S_k\}$ von Untermengen von U, d.h. $S_i \subseteq U$ für $i = 1, ..., k, k \geq 1$ und eine Kostenfunktion $c : T \to \mathbb{Q}^+$, die jeder Untermenge S_i einen Kostenwert zuweist.*
Aufgabe: Finde kostenminimale Teilmenge von T, die alle Elemente von U abdeckt.

ILP: Wir verwenden Variablen $x_{S_1}, ..., x_{S_k}$ wobei die Variable x_{S_i} eine 0-1-Indikatorvariable für $S_i \in T$ ist.

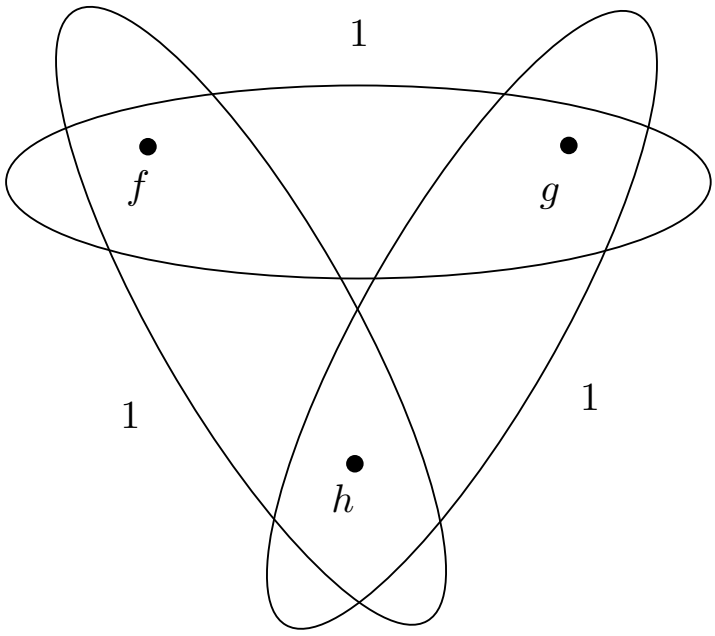
$$\begin{aligned} &\min \sum_{S \in T} c(S) \cdot x_S \\ &\text{unter} \sum_{S: e \in S} x_S \geq 1 && e \in U \\ &\quad x_S \in \{0, 1\} && S \in T \end{aligned}$$

LP-Relaxierung:

$$\begin{aligned} &\min \sum_{S \in T} c(S) \cdot x_S \\ &\text{unter} \sum_{S: e \in S} x_S \geq 1 && e \in U \\ &\quad 0 \leq x_S \underbrace{\leq 1}_{\text{redundant}} && S \in T \end{aligned}$$

Beobachtung 2.2.2 *Fraktionales Set Cover kann billiger sein als ganzzahliges Set Cover.*

Beweis: Betrachte folgendes Beispiel: $U = \{f, g, h\}$



Die beste ganzzahlige Lösung hat den Wert 2, aber die fraktionale Formulierung erlaubt $x_f = x_g = x_h = \frac{1}{2}$ und diese Lösung hat den Zielfunktionswert $\frac{3}{2} < 2$.
Duales LP:

$$\begin{aligned} &\max \sum_{e \in U} y_e \\ &\text{unter} \sum_{e \in S} y_e \leq c(S), && S \in T \\ &\quad y_e \geq 0 && e \in U \end{aligned}$$

Bemerkung 2.2.3 *Das duale LP ist ein Packungsproblem: Man darf Elemente $e \in U$ „einpacken", muss dabei aber für jede Menge S eine obere Schranke $c(S)$ einhalten.*
Überdeckungsprobleme sind häufig dual zu Packungsproblemen.

Algorithmus 2.2.4 (SetCover über LP-Runden)

1. *Finde optimale Lösung der LP-Relaxierung*
2. *Setze $f := \max_{X \in U} |\{S \mid X \in S\}|$*
3. *Wähle alle Mengen S, für die in der optimalen Lösung $x_S \geq \frac{1}{f}$ gilt.*

Satz 2.2.5 *Algorithmus 2.2.4 erzielt einen Approximationsfaktor von f für das SetCover-Problem. Insbesondere liefert er eine korrekte Lösung für das SetCover-Problem.*

Beweis: Sei $\mathcal{C}$ die Menge der ausgewählten Teilmengen. Betrachte ein beliebiges Element $e \in U$. Da e nur in maximal f Mengen ist und das LP die Ungleichung $\sum_{S:e \in S} x_S \geq 1$ enthält, muss mindestens für eine dieser Mengen im fraktionalen Cover $x_S \geq \frac{1}{f}$ erfüllt sein. Also wird e durch $\mathcal{C}$ überdeckt. $\mathcal{C}$ ist also eine gültige Lösung. Dieser Rundungsprozess erhöht jedes x_S maximal um Faktor f. Damit sind die Kosten für $\mathcal{C}$ höchstens f-mal die Kosten des fraktionalen Covers, was die Approximationsgüte zeigt.

□

2.3 Randomisiertes Runden

Idee: Betrachte x_S als **Wahrscheinlichkeit**, S auszuwählen. Wir werden zeigen, dass man dadurch jedes Element mit konstanter Wahrscheinlichkeit durch mindestens eine Menge überdeckt. Allerdings bedeutet das noch nicht, dass alle Elemente **gleichzeitig** überdeckt werden. Um dieses Problem zu beheben, wiederholen wir den Prozess und fügen iterativ Mengen zu den ausgewählten Mengen hinzu. Dadurch werden wir einen randomisierten $\mathcal{O}(\log n)$-Approximationsalgorithmus finden.

Details:
Sei p eine optimale Lösung des LP. Für jedes $S \in T$ wähle S mit Wahrscheinlichkeit p_S, dem Eintrag in p, der S entspricht. Sei $\mathcal{C}$ die Menge der so ausgewählten Mengen. Es gilt

$$\begin{aligned} E(c(\mathcal{C})) &= \sum_{S \in T} Prob[S \text{ ist ausgewählt}] \cdot c(S) \\ &= \sum_{S \in T} p_S \cdot c(S) \\ &= opt_{frac}, \end{aligned} \tag{1}$$

wobei opt_{frac} der Wert der optimalen fraktionalen Lösung ist.
Als nächstes berechnen wir die Wahrscheinlichkeit, dass ein Element $a \in U$ durch $\mathcal{C}$ überdeckt wird. Angenommen, a taucht in k Mengen aus T auf. Seien die mit diesen Mengen assoziierten Wahrscheinlichkeiten $p_1, p_2, \ldots, p_k$. Da a fraktional überdeckt ist, gilt $p_1 + \cdots + p_k \geq 1$. Man kann zeigen, dass diese Wahrscheinlichkeit minimiert wird, wenn alle $p_i = \frac{1}{k}$ sind. Damit folgt

$$Prob[a \text{ ist durch } \mathcal{C} \text{ überdeckt}] \geq 1 - (1 - \tfrac{1}{k})^k \geq 1 - \tfrac{1}{e},$$

da $(1 - \frac{1}{k})^k \leq \frac{1}{e}$ ist.
Jedes Element wird also mit konstanter Wahrscheinlichkeit überdeckt. Wähle nun $d \cdot \log n$ solche Mengen $\mathcal{C}$ und berechne ihre Vereinigung in $\mathcal{C}'$, wobei d eine Konstante ist, die $\left(\frac{1}{e}\right)^{d \log n} \leq \frac{1}{4n}$ erfüllt. Damit gilt

$$Prob[a \text{ ist nicht durch } \mathcal{C}' \text{ überdeckt}] \leq (1 - (1 - \tfrac{1}{e}))^{d \log n} = (\tfrac{1}{e})^{d \log n} \leq \tfrac{1}{4n}$$

Durch Aufsummierung aller $a \in U$ erhalten wir

$$\begin{aligned} & Prob[\mathcal{C}' \text{ ist kein gültiges SetCover}] \\ = \; & Prob[\exists a : a \text{ wird von } \mathcal{C}' \text{ nicht überdeckt}] \\ \leq \; & \sum_{a \in U} Prob[a \text{ wird von } \mathcal{C}' \text{ nicht überdeckt}] \\ = \; & n \cdot \frac{1}{4n} = \frac{1}{4}, \end{aligned}$$

indem wir die Union-Bound-Schranke anwenden.
Außerdem ist $E[c(\mathcal{C}')] = E[\sum_{i=1}^{d \log n} c(\mathcal{C}_i)]$, wobei $\mathcal{C}_i$ die Menge aus der Iteration i ist.
Aufgrund der Linearität des Erwartungswertes ist dies gleich $\sum_{i=1}^{d \log n} E[c(\mathcal{C}_i)] = d \cdot \log n \cdot opt_{frac}$.
Jetzt benötigen wir die folgende Aussage:

Bemerkung 2.3.1 (Markov-Ungleichung) *Für eine nicht-negative Zufallsvariable X gilt:*

$$Prob[x \geq k \cdot E[x]] \leq \tfrac{1}{k}.$$

Mit Hilfe der Markov-Ungleichung mit $k = 4$ folgt

$$Prob[c(\mathcal{C}') \geq opt_{frac} \cdot 4 \cdot d \cdot \log n] \leq \tfrac{1}{4}.$$

Zusammen gilt (wegen Union Bound):

$$Prob[\mathcal{C}' \text{ ist günstiges Set Cover mit Kosten } \leq 4 \cdot d \cdot \log n \cdot opt_{frac}] \geq \tfrac{1}{2}.$$

Wir können die Bedingung in Polynomzeit überprüfen und, falls $\mathcal{C}'$ die Bedingung nicht erfüllt, den Algorithmus wiederholen. Die erwartete Anzahl von Wiederholungen ist ≤ 2.

Bemerkung 2.3.2 *Der beste bekannte Approximationsalgorithmus erreicht eine Güte von $\ln n$, d.h. wir sind bzgl. $\mathcal{O}$-Notation bereits optimal (allerdings nicht in den Vorfaktoren).*

2.4 Ein Primal-Dualer-Algorithmus für Set Cover

2.4.1 Primal-Duale-Algorithmen

Primales LP:

$$\begin{aligned} \min \quad & \sum_{j=1}^{n} c_j x_j & \\ \text{unter} \quad & \sum_{j=1}^{n} a_{ij} x_j & \geq b_i \quad i = 1, \ldots, m \\ & x_j & \geq 0 \quad j = 1, \ldots, n \end{aligned}$$

Duales LP:

$$\begin{aligned} \max \quad & \sum_{i=1}^{m} b_i y_i & \\ \text{unter} \quad & \sum_{i=1}^{m} a_{ij} y_i & \leq c_j \quad j = 1, \ldots, n \\ & y_i & \geq 0 \quad i = 1, \ldots, m \end{aligned}$$

Es gilt:

$$\begin{aligned} \sum_{j=1}^{n} c_j x_j^* &\geq \sum_{j=1}^{n} \sum_{i=1}^{m} (a_{ij} y_i^*) x_j^* \\ &= \sum_{i=1}^{m} \sum_{j=1}^{n} (a_{ij} x_j^*) y_i^* \\ &\geq \sum_{i=1}^{m} b_i y_i^*, \end{aligned} \tag{2}$$

wobei x^* und y^* zulässige primale bzw. duale Lösungen sind.
Nach Satz 1.4.11 (Starke Dualität) gilt für optimale primale bzw. duale zulässige Lösungen sogar Gleichheit. Daraus folgt:

Satz 2.4.1 (Complementary Slackness) *Seien x, y primal bzw. dual zulässige Lösungen. Dann sind x, y optimal genau dann, wenn alle folgenden Bedingungen erfüllt sind:*

1. *Primale Complementary Slackness:*
   Für alle $1 \leq j \leq n$: Entweder $x_j = 0$ oder $\sum_{i=1}^{m} a_{ij} y_i = c_j$.
2. *Duale Complementary Slackness:*
   Für alle $1 \leq i \leq m$: Entweder $y_i = 0$ oder $\sum_{j=1}^{n} a_{ij} x_j = b_i$.

Idee: (Primal - duale Approximationsalgorithmen für NP-schwere Probleme):
Starte mit beliebiger i.A. nicht zulässiger ganzzahliger primaler und zulässiger dualer Lösung. Versuche Schritt für Schritt die Zulässigkeit der primalen Lösung und die Optimalität der dualen Lösung mit Hilfe der Complementary Slackness zu verbessern und dabei die Ganzzahligkeit der primalen Lösung zu erhalten.
Im Allgemeinen können wir die Complementary Slackness Bedingungen nicht erfüllen, da dies eine optimale ganzzahlige Lösung impliziert. Daher führen wir relaxierte Complementary Slackness Bedingungen ein:

Definition 2.4.2 (Relaxierte Complementary Slackness)

1. *Primale relaxierte complementary slackness:*
   Sei $\alpha \geq 1$: Für jedes $1 \leq j \leq n$: Entweder $x_j = 0$ oder $\frac{c_j}{\alpha} \leq \sum_{i=1}^{m} a_{ij} y_i \leq c_j$.

2. *Duale relaxierte complementary slackness:*
   Sei $\beta \geq 1$: Für jedes $1 \leq i \leq m$: Entweder $y_i = 0$ oder $b_i \leq \sum_{j=1}^{n} a_{ij} x_j \leq \beta b_i$.

Lemma 2.4.3 *Wenn x und y primal bzw. dual zulässige Lösungen sind, die die relaxierten complementary slackness Bedingungen erfüllen, dann gilt:*

$$\sum_{j=1}^{n} c_j x_j \leq \alpha \cdot \beta \cdot \sum_{i=1}^{m} b_i y_i$$

Beweis: Wir stellen uns vor, dass wir die rechte Seite der Ungleichung als Geld zur Verfügung haben und davon die linke Seite bezahlen müssen. Jede duale Variable y_i erhält einen Betrag von $\alpha \cdot \beta \cdot b_i \cdot y_i$. Damit haben die dualen Variablen einen Betrag von $\alpha \cdot \beta \cdot \sum_{i=1}^{m} b_i y_i$ erhalten. Die dualen Variablen zahlen nun an die primalen Variablen, so dass

(a) keine duale Variable mehr Geld bezahlt als sie hat und

(b) jede primale Variable x_j einen Betrag $\geq c_j x_j$ erhält.

Dann gilt $\sum_{j=1}^{n} c_j x_j \leq \alpha \cdot \beta \cdot \sum_{i=1}^{m} b_i y_i$.
Jede Variable y_i bezahlt $\alpha y_i a_{ij} x_j$ an die primale Variable x_j. Damit zahlt y_i

$$\alpha y_i \sum_{i=1}^{n} a_{ij} x_j \leq \alpha \cdot \beta \cdot y_i b_i$$

auf Grund der dualen relaxierten complementary slackness Bedingung. Also ist (a) erfüllt. Die Variable x_j erhält

$$\alpha x_j \sum_{i=1}^{m} a_{ij} y_i \geq c_j x_j$$

auf Grund der primalen relaxierten complementary slackness Bedingung. Also gilt auch (b). Daraus folgt das Lemma.

□

Definition 2.4.4 *Wir nennen eine Menge $\mathcal{S}$,* scharf, *wenn $\sum_{e \in \mathcal{S}} y_e = c(S)$ gilt.*

Unsere Strategie:
Nimm nur scharfe Mengen in die primale Lösung. Dadurch ist die exakte primale complementary slackness Bedingung immer erfüllt. Als relaxierte duale complementary slackness Bedingung verwenden wir

$$\forall e : y_e \neq 0 \Rightarrow \sum_{S : e \in S} x_S \leq f.$$

Da wir $x_S \in \{0,1\}$ wählen, ist dies äquivalent zu: Jedes Element mit dualer Variable $y_e \neq 0$ darf nur f-mal überdeckt werden.
$\Rightarrow$ Hier trivial, da jedes Element in maximal f Mengen ist.

Algorithmus 2.4.5 *(SetCover Primal Dual)*

1. $x \leftarrow 0$, $y \leftarrow 0$
2. *Solange noch nicht alle Elemente überdeckt:*
3. *Nimm nicht überdecktes Element e und erhöhe y_e, bis mindestens eine Ungleichung des dualen Programms mit Gleichheit erfüllt ist, d.h. mindestens eine Menge ist scharf.*
4. *Füge alle scharfen Mengen zur Überdeckung hinzu und aktualisiere x.*
5. *Nenne alle Elemente, die in diesen Mengen auftreten, überdeckt.*
6. *Gib Überdeckung x aus.*

Satz 2.4.6 *Der obige Algorithmus erreicht einen Approximationsfaktor von f.*

Beweis: Am Ende des Algorithmus sind sowohl die primale exakte als auch die duale relaxierte complementary slackness Bedingung mit $\alpha = 1$ und $\beta = f$ erfüllt.
Damit ist die Lösung nach 2.4.3 eine f-Approximation.

2.5 Max-SAT über randomisiertes Runden

Problem 2.5.1 *(Max-SAT):*
Gegeben ist eine CNF Formel f über den booleschen Variablen $x_1 \ldots, x_n$ und nicht negative Gewichte w_c für jede Klausel c von f. Eine Klausel ist eine Disjunktion von Literalen (z.B. $x_1 \vee \overline{x_2}$) und ein Literal ist eine Variable oder die Negation einer Variable.
Aufgabe: Finde Belegung, die das Gewicht der erfüllten Klauseln maximiert.

Beispiel 2.5.2 $\underbrace{(x_1 \vee \overline{x_2})}_{Gewicht\ w_1} \wedge \underbrace{(x_2 \vee x_3)}_{Gewicht\ w_2} \wedge \underbrace{(x_1 \vee x_2 \vee \overline{x_3})}_{Gewicht\ w_3}$

Notation 2.5.3 *Wir schreiben $f = \bigwedge_{c \in \mathcal{C}} c$ und bezeichnen mit $size(c)$ die Anzahl der Literale in c. Außerdem verwenden wir die folgenden Zufallsvariablen W und W_c, die von einer zufälligen Belegung abhängen:*
W: Gewicht der erfüllten Klauseln
W_c Beitrag von Klausel c zu W, d.h.

$$\begin{aligned} W &= \sum_{c \in \mathcal{C}} W_c \text{ und} \\ E[W_c] &= w_c \cdot Prob[c \text{ ist erfüllt}] \end{aligned}$$

(Wir verwenden diese Notation für unterschiedliche Algorithmen, die Bedeutung von W und W_c ist dann ebenfalls unterschiedlich).

Algorithmus 2.5.4 *(einfacher MaxSat Algorithmus)*
Setze jede Variable mit Wahrscheinlichkeit $\frac{1}{2}$ auf wahr (TRUE) und gib die resultierende Belegung aus.

Lemma 2.5.5 *Wenn size(c) = k, dann ist* $E[W_c] = (1 - 2^{-k}) \cdot w_c$

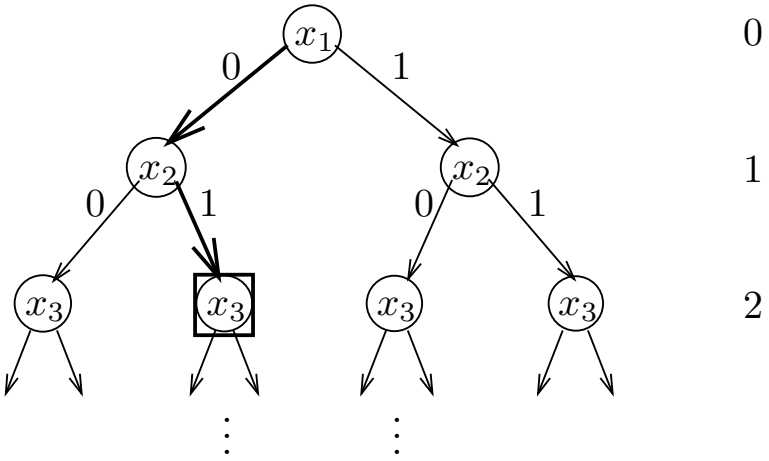
Beweis: Eine Klausel ist nicht erfüllt, wenn alle Literale auf falsch gesetzt wurden. Die Wahrscheinlichkeit hierfür ist 2^{-k}.

□

Es folgt $E[W] = E[\sum_{c \in \mathcal{C}} W_c] = \sum_{c \in \mathcal{C}} E[W_c] \overset{k \geq 1}{\geq} \frac{1}{2} \sum_{c \in \mathcal{C}} w_c \geq \frac{1}{2} \cdot Opt.$

2.5.1 Derandomisierung

Betrachte Baum T für Formel f:



Jeder Knoten auf Ebene i (hat Markierung x_{i+1} und) entspricht einer Belegung der Variablen $x_1, \ldots, x_i$. Jedes Blatt (= Knoten auf Ebene n) entspricht einer Belegung aller n Variablen.
Wir werden die Knoten von T mit folgenden bedingten Wahrscheinlichkeiten versehen:
Sei $a_1, \ldots, a_i$ eine Belegung der Variablen $x_1, \ldots, x_i$. Dann wird der Knoten, der dieser Belegung entspricht, mit $\mathrm{E}[W \mid x_1 = a_1, \ldots, x_i = a_i]$ annotiert.

Lemma 2.5.6 *Der bedingte Erwartungswert eines Knotens von T kann in Polynomialzeit berechnet werden.*

Beweis: Betrachte den Knoten, den wir durch $x_1 = a_1, \ldots, x_i = a_i$ erhalten. Sei ϕ die boolsche Formel über den Variablen $x_{i+1}, \ldots, x_n$ die man durch die Restriktion von f mit $x_1 = a_1, \ldots, x_i = a_i$ erhält.
Man kann den Erwartungswert von ϕ in Polynomialzeit ausrechnen. Wenn wir dazu noch das Gewicht der durch die partielle Belegung erfüllten Klauseln addieren, so erhalten wir den bedingten Erwartungswert.

□

Idee:
Wir nutzen jetzt folgende Gleichheit aus:

$$\mathrm{E}[W] = \tfrac{1}{2}\mathrm{E}[W \mid x_1 = 0] + \tfrac{1}{2}\mathrm{E}[W \mid x_1 = 1]$$

Die bedingten Erwartungswerte können wir in Polynomialzeit ausrechnen.
Mindestens einer der Erwartungswerte muss $\geq \mathrm{E}[W]$ sein.

Beispiel 2.5.7 *Sei* $E[W \mid x_1 = 0] \geq E[W]$.
Dann wissen wir

$$E[W \mid x_1 = 0] = \tfrac{1}{2}E[W \mid x_1 = 0, x_2 = 0] + \tfrac{1}{2}E[W \mid x_1 = 0, x_2 = 1]$$

Sei nun z.B. $E[W \mid x_1 = 0, x_2 = 1] \geq E[W \mid x_1 = 0]$*, dann fahren wir analog mit* $E[W \mid x_1 = 0, x_2 = 1]$ *fort. Nach* n *Schritten finden wir einen bedingten Erwartungswert* $E[W \mid x_1 = 0, x_2 = 1, \ldots, x_n = 0] \geq E[W]$ *und haben damit eine Belegung gefunden, die mindestens Wert* $E[W]$ *erreicht.*

Satz 2.5.8 *Wir können in Polynomialzeit einen Pfad von der Wurzel zu einem Blatt finden, so dass die bedingte Wahrscheinlichkeit jedes Knotens auf dem Pfad mindestens* $E[W]$ *ist.*

Beweis: Die bedingte Wahrscheinlichkeit eines Knotens ist die durchschnittliche bedingte Wahrscheinlichkeit seiner Kinder, d.h.

$$\begin{aligned} \mathrm{E}[W \mid x_1 = a_1, \ldots x_i = a_i] \quad = \quad & \tfrac{1}{2}\mathrm{E}[W \mid x_1 = a_1, \ldots x_i = a_i, x_{i+1} = 0] \\ + \quad & \tfrac{1}{2}\mathrm{E}[W \mid x_1 = a_1, \ldots x_i = a_i, x_{i+1} = 1]. \end{aligned}$$

Damit hat mindestens eines der Kinder einen genauso großen Erwartungswert wie sein Elternknoten. Damit existiert der gesuchte Pfad. Nach Lemma 2.5.6 kann er auch in Polynomialzeit berechnet werden.
Da die Blätter von T Belegungen $x_1, \ldots, x_n$ entsprechen, findet unser Algorithmus eine Belegung mit Gewicht $\geq \mathrm{E}[W]$.

Bisher: Einfacher Ansatz, der besser wird, wenn mehr Variablen in Klauseln vorkommen.
Jetzt: Ansatz für Klauseln mit wenigen Variablen, der $(1 - \frac{1}{e})$-Approximation liefert.
ILP für Max-Sat
Für alle Klauseln $c \in \mathcal{C}$ sei S_c^+, S_c^- Menge der nicht negierten bzw. der negierten Variablen in c. Die Belegung wird durch Vektor y kodiert:

$$\begin{aligned} y_i = 1 \quad &\Leftrightarrow \quad x_i \text{ wahr} \\ (y_i = 0 \quad &\Leftrightarrow \quad x_i \text{ falsch}) \end{aligned}$$

Für alle Klauseln $c \in \mathcal{C}$ gibt es eine Variable $z_c \in \{0, 1\}$, die nur auf 1 gesetzt werden können soll, wenn mindestens ein Literal aus c auf wahr gesetzt wird.

$$\begin{aligned} \max \quad & \sum_{c \in \mathcal{C}} w_c \cdot z_c \\ \text{unter} \quad & \sum_{i \in \mathcal{S}_c^+} y_i + \sum_{i \in \mathcal{S}_c^-} (1 - y_i) \geq z_i && \forall c \in \mathcal{C} \\ & y_i \in \{0, 1\} && \forall i = 1, \ldots, n \\ & z_c \in \{0, 1\} && \forall c \in \mathcal{C} \end{aligned}$$

LP-Relaxierung

$$\begin{aligned} \max \quad & \sum_{c \in \mathcal{C}} w_c \cdot z_c \\ \text{unter} \quad & \sum_{i \in \mathcal{S}_c^+} y_i + \sum_{i \in \mathcal{S}_c^-} (1 - y_i) \geq z_i && \forall c \in \mathcal{C} \\ & 1 \geq y_i \geq 0 && \forall i = 1, \ldots, n \\ & 1 \geq z_c \geq 0 && \forall c \in \mathcal{C} \end{aligned}$$

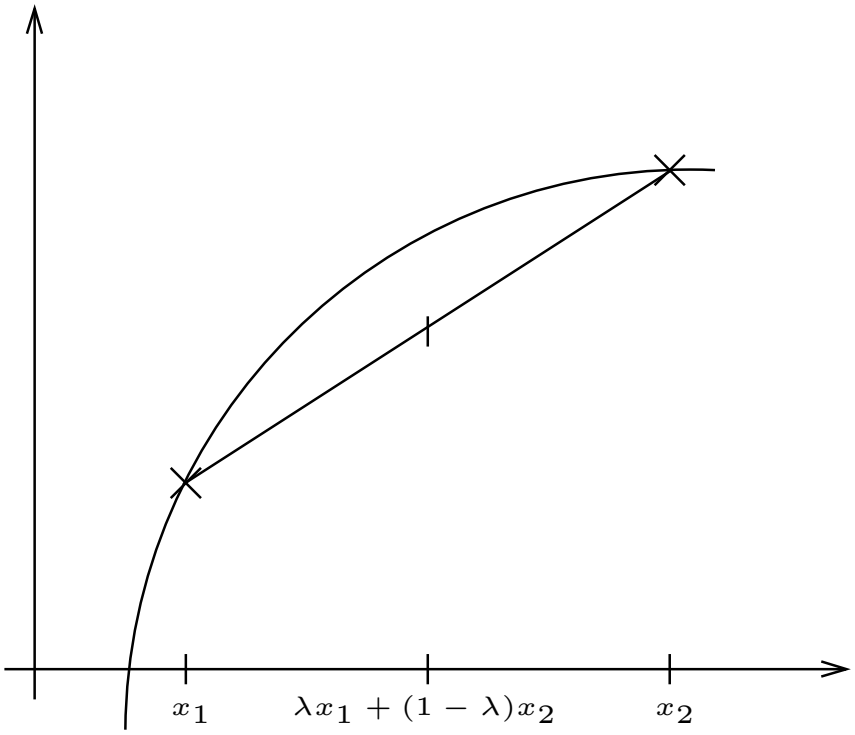
Algorithmus 2.5.9 *(Max-Sat-RR):*

1. *Löse die LP-Relaxierung. Sei* (y^*, z^*) *die optimale Lösung.*
2. *Setze jedes* x_i *unabhängig mit Wahrscheinlichkeit* y_i^* *auf wahr für alle* $i = 1, \ldots, n$.

3. Ausgabe der resultierenden Belegung τ.

Bemerkung 2.5.10 *Die gewählte Belegung ist immer zulässig. Das ist ein Vorteil von Max-Sat, der dieses Problem für eine Lösung mit randomisiertem Runden besonders geeignet macht.*

Definition 2.5.11 (konkav) *Eine Funktion $g : \mathbb{R} \to \mathbb{R}$ heißt konkav, wenn $g(\lambda x_1 + (1-\lambda)x_2) \geq \lambda g(x_1) + (1-\lambda)g(x_2)$ für $0 \leq \lambda \leq 1$ und $x_1, x_2 \in \mathbb{R}$.*



Lemma 2.5.12 *Wenn $size(c) = k$, dann gilt*

$$E[W_c] \geq (1 - (1 - \tfrac{1}{k})^k) \cdot w_c \cdot z_c^*$$

Beweis: Wir können annehmen, dass alle Literale in c nicht negiert auftreten (da wir nur eine einzelne Klausel betrachten und daher die Variablen in der Formel ggf. passend durch ihre Negationen tauschen können).
Sei o.B.d.A. $c = (x_1 \vee x_2 \cdots \vee x_k)$.
Klausel c ist erfüllt, wenn nicht alle $x_1, \ldots, x_k$ auf falsch gesetzt sind. Dies geschieht mit Wahrscheinlichkeit

$$1 - \underbrace{\prod_{i=1}^{k}(1-y_i)}_{\text{W'keit, dass alle } x_i \text{ auf falsch gesetzt werden}} .$$

Wir schätzen diese Wahrscheinlichkeit mit Hilfe der Abschätzung

$$\tfrac{a_1+\cdots+a_k}{k} \geq (a_1 \cdot a_2 \cdots \cdot a_k)^{\frac{1}{k}}$$

ab:

$$\begin{aligned}
1-\prod_{i=1}^{k}(1-y_i) &= 1-\left(\left(\prod_{i=1}^{k}(1-y_i)\right)^{\frac{1}{k}}\right)^k \\
&\geq 1-\left(\frac{\sum_{i=1}^{k}(1-y_i)}{k}\right)^k \\
&= 1-\left(1-\frac{\sum_{i=1}^{k}y_i}{k}\right)^k \\
&\geq 1-\left(1-\frac{z_c^*}{k}\right)^k,
\end{aligned}$$

wobei die letzte Ungleichung wegen $y_1+\ldots+y_k \geq z_c^*$ gilt, da z_c^* eine zulässige Lösung der LP-Relaxierung ist.
Betrachte nun die Funktion

$$g(z) = 1-\left(1-\tfrac{z}{k}\right)^k.$$

Es gilt $g(0) = 1-1 = 0$ und $g(1) = 1-\left(1-\frac{1}{k}\right)^k$.
Man kann zeigen, dass g konkav ist.
Damit gilt für $z \in [0,1]$, dass

$$\begin{aligned}
g(z) = g((1-z)\cdot 0 + z\cdot 1) &\underset{\substack{\uparrow \\ g \text{ konkav}}}{\geq} 0\cdot g(1-z) + z\cdot g(1) \\
&= z\cdot\left(1-\left(1-\tfrac{1}{k}\right)^k\right) \text{ ist.}
\end{aligned}$$

Es folgt:

$$\text{Prob}[c \text{ ist erfüllt}] \geq \left(1-\left(1-\tfrac{1}{k}\right)^k\right)\cdot z_c^*$$

und somit:

$$\begin{aligned}
\text{E}[W_c] &= w_c\cdot\text{Prob}[c \text{ ist erfüllt}] \\
&\geq \left(1-\left(1-\tfrac{1}{k}\right)^k\right)\cdot z_c^*\cdot w_c.
\end{aligned}$$

□

Da $1-\left(1-\frac{1}{k}\right)^k$ mit steigendem k monoton fallend ist (sie geht für $k\to\infty$ gegen $1-\frac{1}{e}$), folgt für Klauseln der Größe k:

$$\begin{aligned}
\text{E}[W] = \sum_{c\in\mathcal{C}}\text{E}[W_c] &\geq \left(1-\left(1-\tfrac{1}{k}\right)^k\right)\cdot\sum_{c\in\mathcal{C}} w_c\cdot z_c^* \\
&= \left(1-\left(1-\tfrac{1}{k}\right)^k\right)\cdot \text{Opt}_{frac} \\
&\geq \left(1-\left(1-\tfrac{1}{k}\right)^k\right)\cdot \text{Opt}
\end{aligned}$$

wobei Opt_{frac} der Wert einer optimalen Lösung der LP-Relaxierung ist und Opt der Wert der optimalen ganzzahligen Lösung.

Satz 2.5.13 *Der Algorithmus MAX-SAT-RR ist ein* $(1-\frac{1}{e})$*-Approximationsalgorithmus.*

Beweis: Folgt aus dem vorherigen Lemma, da $(1-\frac{1}{k})^k < \frac{1}{e}$ für alle $k\in\mathbb{Z}^+$.

□

2.6 Eine $\frac{3}{4}$-Approximation

Wir kombinieren unsere Algorithmen wie folgt:

Algorithmus 2.6.1

1. *Wirf eine Münze*
2. *Falls Kopf: Starte „einfachen" Algorithmus*
3. *Falls Zahl: Starte Algorithmus MAX-SAT-RR*

Lemma 2.6.2 *Sei (y^*, z^*) eine optimale Lösung der LP-Relaxierung. Es gilt*

$$E[W_c] \geq \tfrac{3}{4} w_c \cdot z_c^*.$$

Beweis: Sei $size(c) = k$. Nach Lemma 2.5.5 gilt

$$\begin{aligned} E[W_c|\text{Münze zeigt Kopf}] &= (1 - 2^{-k}) \cdot w_c \\ &\geq (1 - 2^{-k}) \cdot w_c \cdot z_c^* \end{aligned}$$

Nach Lemma 2.5.12 zu Algorithmus MAX-SAT-RR gilt

$$E[W_c|\text{Münze zeigt Zahl}] = (1 - (1 - \tfrac{1}{k})^k) \cdot w_c \cdot z_c^*.$$

Wir erhalten

$$\begin{aligned} E[W_c] &= \quad Prob[\text{Münze zeigt Kopf}] \cdot E[W_c|\text{Münze zeigt Kopf}] \\ &\quad + Prob[\text{Münze zeigt Zahl}] \cdot E[W_c|\text{Münze zeigt Zahl}] \\ &\geq \frac{1}{2} \cdot w_c \cdot z_c^* \cdot (1 - 2^{-k}) + \frac{1}{2} \cdot w_c \cdot z_c^* \cdot (1 - (1 - \frac{1}{k})^k) \\ &= w_c z_c^* \cdot \left(\frac{1 - 2^{-k} + 1 - (1 - \frac{1}{k})^k}{2} \right) \end{aligned}$$

Betrachte verschiedene Fälle:

$k = 1$: $1 - 2^{-k} + 1 - (1 - \frac{1}{k})^k = 1 - \frac{1}{2} + 1 - (1 - \frac{1}{1})^1 = \frac{3}{2}$
$k = 2$: $1 - 2^{-k} + 1 - (1 - \frac{1}{k})^k = 1 - \frac{1}{4} + 1 - (1 - \frac{1}{2})^2 = \frac{3}{4} + \frac{3}{4} = \frac{3}{2}$
$k \geq 3$: $1 - 2^{-k} + 1 - (1 - \frac{1}{k})^k \geq \frac{7}{8} + 1 - \frac{1}{e} \geq \frac{3}{2}$.

Damit folgt das Lemma.

□

Insgesamt folgt:

$$\begin{aligned} E[W] &= \sum_{c \in \mathcal{C}} E[W_c] \geq \sum_{c \in \mathcal{C}} \frac{3}{4} \cdot w_c \cdot z_c^* \\ &= \frac{3}{4} \sum_{c \in \mathcal{C}} w_c \cdot z_c^* = \frac{3}{4} Opt_{frac} \\ &\geq \frac{3}{4} Opt. \end{aligned}$$

2.7 Ein deterministischer Algorithmus

Beobachtung 2.7.1 *Algorithmus MAX-SAT-RR kann derandomisiert werden.*

Beweis: Übung.

□

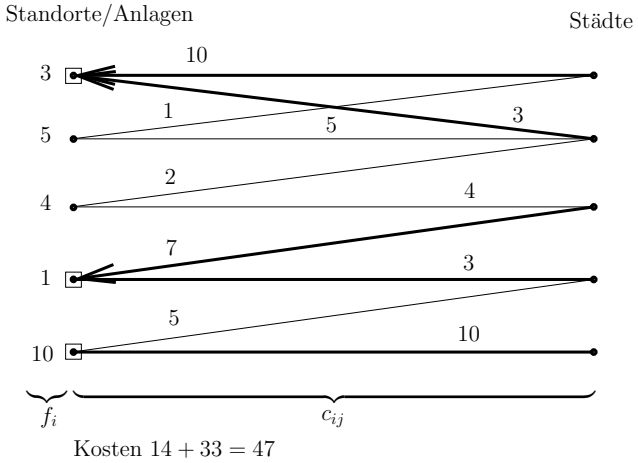
Algorithmus 2.7.2 *(MAX-SAT)*

1. *Berechne $\frac{1}{2}$-Approximation mit derandomisiertem „einfachen“ Algorithmus.*
2. *Berechne $\left(1-\frac{1}{e}\right)$-Approximation mit derandomisiertem MAX-SAT-RR.*
3. *Gib die bessere Lösung aus.*

Satz 2.7.3 *Algorithmus 2.7.2 ist ein deterministischer Faktor $\frac{3}{4}$-Approximationsalgorithmus für MAX-SAT.*

Beweis: Nach der Analyse des randomisierten Algorithmus ist der Durchschnitt der beiden Lösungen mindestens $\frac{3}{4}$-Opt. Damit muss die bessere Lösung ebenfalls diesen Wert erreichen.

2.8 Metrische Standortbestimmung (Metric Facility Location)



Problem 2.8.1 (Metrische Standortbestimmung)

Gegeben:

G: bipartiter Graph mit Bipartition (F, C)
F: Menge von Anlagen / Standorten
C: Menge von Städten
f: Kosten, um Anlage i zu öffnen
c_{ij}: Kosten, um Stadt j von Anlage i aus zu versorgen.
Die c_{ij} erfüllen die Dreiecksungleichung.

Aufgabe: *Finde $I \subseteq F$ und $\Phi : C \to I$, wobei I eine Menge von zu öffnenden Anlagen ist und Φ die Städte den offenen Anlagen zuweist.*
Dabei soll die Summe aus Öffnungs- und Versorgungskosten minimiert werden.

ILP
$$\begin{aligned} \min \sum_{i\in F, j\in C} c_{ij}x_{ij} + \sum_{i\in F} f_i y_i & \\ \text{unter} \quad \sum_{i\in F} x_{ij} &\geq 1 && j \in C \\ y_i - x_{ij} &\geq 0 && i \in F, j \in C \\ x_{ij} &\in \{0,1\} && i \in F, j \in C \\ y_i &\in \{0,1\} && i \in F \end{aligned}$$

Bedeutung der Variablen:

x_{ij} : ist genau dann 1, wenn Stadt j von Anlage i versorgt wird.
y_i : ist genau dann 1, wenn Anlage i geöffnet ist.

Bedingung (1): Jede Stadt muss versorgt werden.
Bedingung (2): Städte dürfen nur von offenen Anlagen versorgt werden.

LP:
$$\begin{aligned} \min \sum_{i\in F, j\in C} c_{ij}x_{ij} + \sum_{i\in F} f_i y_i & \\ \text{unter} \quad \sum_{i\in F} x_{ij} &\geq 1 && j \in C \\ y_i - x_{ij} &\geq 0 && i \in F, j \in C \\ 0 \leq x_{ij} &\leq 1 && i \in F, j \in C \\ 0 \leq y_i &\leq 1 && i \in F \end{aligned}$$

Das duale LP:
$$\begin{aligned} \max \sum_{j\in C} \alpha_j & \\ \text{unter} \quad \alpha_j - \beta_{ij} &\leq c_{ij} && i \in F, j \in C \\ \sum_{j\in C} \beta_{ij} &\leq f_i && i \in F \\ \alpha_j &\geq 0 && j \in C \\ \beta_{ij} &\geq 0 && i \in F, j \in C \end{aligned}$$

Das duale LP verstehen
Die dualen Variablen „bezahlen“ die primale Lösung.
(Vereinfachende) Annahme: Es gibt eine optimale Lösung, die ganzzahlig ist, d.h. es gibt $I \subseteq F$ und $\Phi : C \to I$, so dass $y_i = 1 \Leftrightarrow i \in I$ und $x_{ij} = 1 \Leftrightarrow i = \Phi(j)$ und $y_i = 0, x_{ij} = 0$, sonst.
Sei (α, β) eine optimale duale Lösung und (x, y) eine optimale primale Lösung.

Complementary Slackness Bedingungen

$$\begin{aligned} &(i)\ \forall i \in F, j \in C : x_{ij} > 0 && \Rightarrow \alpha_i - \beta_{ij} = c_{ij} \\ &(ii)\ \forall i \in F : y_i > 0 && \Rightarrow \sum_{j\in C} \beta_{ij} = f_i \\ &(iii)\ \forall j \in C : \alpha_j > 0 && \Rightarrow \sum_{i\in F} x_{ij} = 1 \\ &(iv)\ \forall i \in F, j \in C : \beta_{ij} > 0 && \Rightarrow y_i = x_{ij} \end{aligned}$$

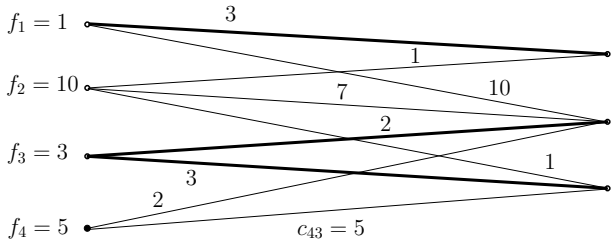
Bedingung (ii):
Wenn Anlage i geöffnet, dann wird sie durch die β_{ij} bezahlt, d.h. wenn $i \in I$, dann gilt

$$\sum_{j:\Phi(j)=i} \beta_{ij} = f_i$$

Bedingung (iv):
Jede Stadt bezahlt nur für Anlagen, die mit ihr verbunden sind, denn wenn Anlage i offen ist, und $\Phi(j) \neq i$, dann ist $y_i \neq x_{ij}$ und daher $\beta_{ij} = 0$.

Bedingung (i):
Wenn $\Phi(j) = i$, dann gilt $\alpha_j - \beta_{ij} = c_{ij} \Leftrightarrow \alpha_j = \beta_{ij} + c_{ij}$

Beispiel 2.8.2



Optimale primale Lösung:
$y_1 = 1$, $y_2 = 0$, $y_3 = 1$, $y_4 = 0$
$x_{11} = 1$, $x_{32} = 1$, $x_{33} = 1$, *alle anderen* $x_{ij} = 0$.
Mit Kosten 12.
Optimale duale Lösung:
$\alpha_1 = 4$, $\alpha_2 = 4$, $\alpha_3 = 4$
$\beta_{11} = 1$, $\beta_{32} = 2$, $\beta_{33} = 1$, *alle anderen* $\beta_{ij} = 0$.
Mit Kosten 12.

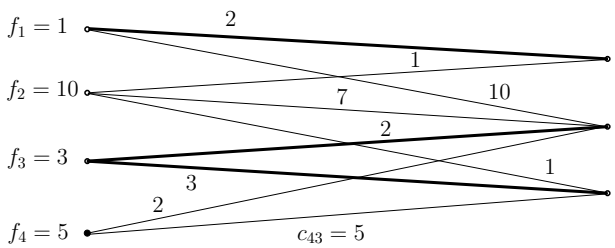
2.8.1 Relaxierung der Bedingungen

Die Städte werden in direkt und indirekt verbunden eingeteilt.
Indirekt verbunden $\Rightarrow \beta_{ij} = 0$ Für indirekt verbundene Städte relaxieren wir die primale Bedingung zu: $\frac{1}{3} \cdot c_{\Phi(j)j} \leq \alpha_j \leq c_{\Phi(j)j}$. (Hinweis: $x_{\Phi(j)j} = 1$!)
Alle anderen Bedingungen werden erfüllt, d.h. für eine direkt verbundene Stadt j gilt insbesondere:

$$\alpha_j - \beta_{\Phi(j)j} = c_{\Phi(j)j}$$

und für jede Anlage gilt $\sum_{j_\Phi(j)=1} \beta_{ij} = f_i$.

Beispiel 2.8.3



1. *Start mit dual zulässiger Lösung* $\alpha_1 = 0$, $\alpha_2 = 0$, $\alpha_3 = 0$, $\beta_{ij} = 0$, $I = \emptyset$, $y_i = 0$, $x_{ij} = 0$.

2. *Bedingung aus dualem LP:* $\alpha_j - \beta_{ij} \leq c_{ij}$

   Für die oberste Stadt erhalten wir daraus bei Standort 2 die stärkste Einschränkung, nämlich $\alpha_1 - \beta_{21} \leq 1$.

3. *Wir setzen* $\alpha_1 = 1, \alpha_2 = 1, \alpha_3 = 1$.

   Weiterhin ist $\beta_{ij} = 0, I = \emptyset, y_i = 0, x_{ij} = 0$.

4. $\alpha_1 = 2, \alpha_2 = 2, \alpha_3 = 2$,

   $\beta_{21} = 1, \beta_{23} = 1, \beta_{ij} = 0$

5. $\alpha_1 = 3, \alpha_2 = 3, \alpha_3 = 3$

   $I = \{1\}, y_1 = 1, x_{11} = 1, \beta_{11} = 1, \beta_{21} = 2, \beta_{23} = 2, \beta_{42} = 1$

6. $\alpha_1 = 3, \alpha_2 = 4, \alpha_3 = 4, I = \{1, 3\}$

Algorithmus 2.8.4

Phase I ignoriert zunächst Bedingung (iv) und verfolgt ansonsten den primal-dualen Ansatz, um möglichst gute Lösungen zu berechnen.
Phase II berechnet dann eine primal zulässige und ganzzahlige Lösung, die auch (iv) erfüllt.
Phase I: Erhöhe alle α_j *gleichmäßig und erhalte dabei durch Anpassen der übrigen primalen und dualen Variablen die duale Zulässigkeit und die primale complementary slackness.*
Wir definieren Zeit: Prozess startet bei Zeitpunkt 0. Zeit erhöht sich um 1 pro Zeiteinheit.

Ereignisse:

1. $\alpha_j = c_{ij}$ *für Kante* (i, j)*:*

   (i, j) *heißt scharf.*

   In Zukunft wird β_{ij} *erhöt, um die erste Ungleichung des Dualen zu erfüllen.*

Bedeutung: Die Stadt j *kann für die Kante* (i, j) *bezahlen,* β_{ij} *geht an Anlage* i*, Kante* (i, j) *mit* $\beta_{ij} > 0$ *heißt speziell.*

2. $\sum \beta_{ij} = f_i$*: Anlage* i *ist vorläufig offen*

   Alle nicht verbundenen Städte j *mit scharfer Kante* (i, j) *werden verbunden.* i *heißt Verbindungszeuge von* j*. Die* α_i *dieser Städte werden nicht weiter erhöht!*

   (wegen $\sum \beta_{ij} \leq f_i$ *und* $\alpha_{ij} - \beta_{ij} \leq c_{ij}$*).*

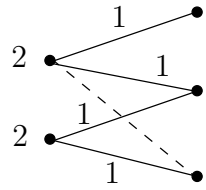
3. *Kante* (i, j) *wird scharf für offene Anlage:*

   j *wird verbunden und* i *ist der Verbindungszeuge.*

   (i, j) *ist nicht speziell, weil* $\beta_{ij} = 0$*!*

Phase I endet, wenn alle Städte verbunden sind.

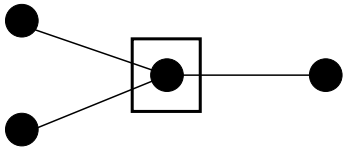
Unglücklicher Fall:



In dieser Situation werden in Schritt 2 beide Anlagen geöffnet, obwohl nicht genug Geld vorhanden ist. In dieser Situation wird uns die Dreiecksgleichung weiterhelfen.

Definition 2.8.5 *Sei $G = (V, E)$ ein ungerichteter Graph. Eine Knotenmenge $I \subseteq V$ heißt unabhängige Menge, wenn zwischen den Knoten in I keine Kanten verlaufen. Eine unabhängige Menge heißt inklusionsmaximal, wenn man keine Knoten zu I hinzufügen kann, ohne die Unabhängigkeitsbedingung zu verletzen.*

Beispiel 2.8.6 *Das folgende Beispiel zeigt eine inklusionsmaximale unabhängige Menge, deren Kardinalität nicht maximal ist.*



Algorithmus 2.8.7 *Phase II*

F_t: *Menge der vorläufigen offenen Anlagen*

T: *Untergraph von G aus speziellen Kanten*

T^2: *Graph mit Kanten (u, v), für die es Pfad der Länge 2 zwischen u und v in T gibt.*

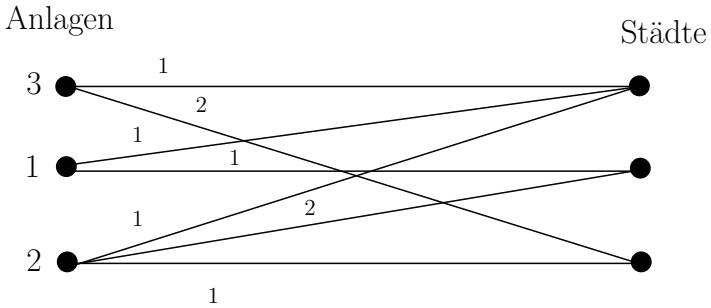
H: *Untergraph von T^2, der durch F_t induziert wird.*

Schritt 1: *Finde eine inklusionsmaximale unabhängige Menge I in H.*

Schritt 2: *Ist Stadt j über spezielle Kante mit $i \in I$ verbunden, so setze $\phi(j) = i$ und nenne j direkt verbunden (es gibt max eine solche Möglichkeit pro Stadt).*

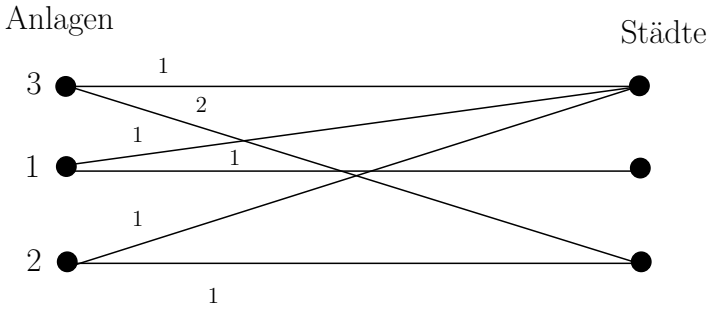
Ansonsten: *Sei (i', j) die scharfe Kante, die j mit seinem Verbindungszeugen i' verbindet.*
Falls $i' \in I$, so setze $\phi(j) = i'$ und nenne j direkt verbunden.
Ansonsten sei i der Nachbar von i' in H, so dass $i \in I$.
Setze $\phi(j) = i$ und nenne j indirekt verbunden.

Beispiel 2.8.8

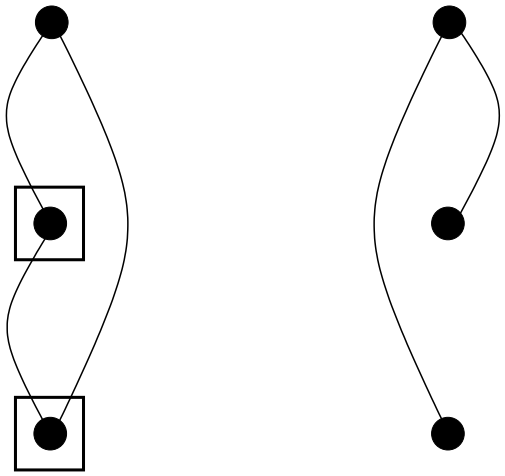


Zeitpunkt 1:
$\alpha_1 = \alpha_2 = \alpha_3 = 1$
Kanten $c_{11}, c_{21}, c_{31}, c_{22}, c_{32}$ *werden scharf*
Zeitpunkt 1.5:
$\alpha_1 = \alpha_2 = \alpha_3 = 1.5$
Anlage 2 wird vorläufig geöffnet und ist Verbindungszeuge für die Städte 1 und 2
Zeitpunkt 2:
$\alpha_1 = \alpha_2 = 1.5, \alpha_3 = 2$
Die Kante c_{13} *wird scharf.*
Zeitpunkt 2.5
$\alpha_1 = \alpha_2 = 1.5, \alpha_3 = 2.5$
Anlage 3 wird vorläufig geöffnet und ist Verbindungszeuge zu Stadt 3.
Ende von Phase I

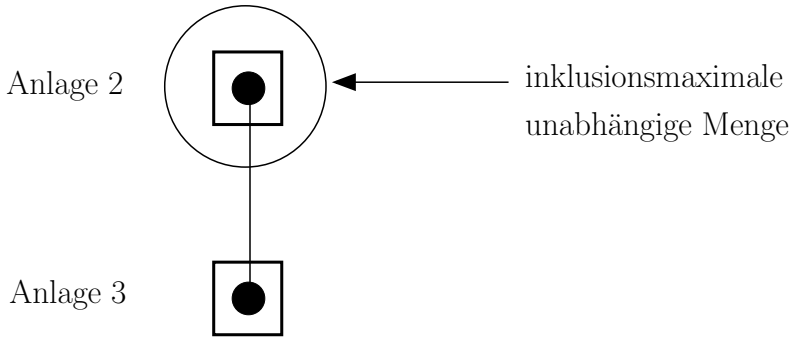
Phase II:
$F_t = \{2, 3\}$
T sieht folgendermaßen aus:



T^2 *hat folgende Form:*



Wir erhalten also für H:



Es wird nur Anlage 2 geöffnet.
Stadt 1 und 2 sind direkt mit Anlage 2 verbunden.
Stadt 3 ist indirekt (über Anlage 3) mit Anlage 2 verbunden.

Analyse 2.8.9 *Duale Variablen bezahlen primale Öffnungs- und Verbindungskosten. Seien α_j^f und α_j^e die Beiträge von j zu diesen beiden Kosten $\alpha_j = \alpha_j^f + \alpha_j^e$.*

j indirekt verbunden: $\alpha_j^f = 0$, $\alpha_j^e = \alpha_j$
j direkt verbunden: $\alpha_j = c_{ij} + \beta_{ij}$, *mit* $i = \phi(j)$
dann sei $\alpha_j^f = \beta_{ij}$ *und* $\alpha_j^e = c_{ij}$

Lemma 2.8.10 *Sei $i \in I$. Dann gilt*

$$\sum_{j:\phi(j)=i} \alpha_j^f = f_i$$

Beweis: i ist vorläufig offen am Ende von Phase I und somit gilt $\sum_{j:(i,j) ist\ speziell} \beta_{ij} = f_i$
Jede Stadt j, die zu f_i beiträgt (eine spezielle Kante zu f_i hat), ist direkt verbunden und somit gilt $\alpha_j^f = \beta_{ij}$. Für jede andere Stadt j' gilt $\alpha_{j'}^f = 0$. Damit folgt das Lemma.

Korollar 2.8.11 $\sum_{i \in I} f_i = \sum_{j \in C} \alpha_j^f$.

Lemma 2.8.12 *Für eine indirekt verbundene Stadt j gilt $c_{ij} \leq 3\alpha_j^e$, wobei $i = \phi(j)$.*

Beweis: Sei i' der Verbindungszeuge für Stadt j. Da j indirekt zu i verbunden ist, ist (i, i') Kante in H. Also gibt es Stadt j', so dass (i, j') und (i', j') spezielle Kanten sind. Es seien t_1 und t_2 die Zeitpunkte, zu denen i und i' vorläufig geöffnet werden. Da (i', j) scharf, gilt $\alpha_j \geq c_{i'j}$. Wir zeigen $\alpha_j \geq c_{ij'}$ und $\alpha_j \geq c_{i'j'}$. Dann gilt nach der Dreiecksungleichung $c_{ij} \leq c_{ij'} + c_{i'j'} + c_{i'j} \leq 3\alpha_j = 3\alpha_j^e$
Da (i', j') und (i, j') speziell sind, müssen sie beide scharf geworden sein, bevor eine Anlage von i oder i' vorläufig geöffnet wurde.
Betrachte den Zeitpunkt $\min\{t_1, t_2\}$.
Nach diesem Zeitpunkt wird $\alpha_{j'}$ nicht mehr erhöht. Also gilt $\alpha_{j'} \leq \min\{t_1, t_2\}$.
Da i' Verbindungszeuge für j ist, gilt $\alpha_2 \geq t_2$. Also $\alpha_j \geq \alpha_{j'}$ und somit folgt das Lemma.

Satz 2.8.13 *Unser Algorithmus für das metrische Standortbestimmungsproblem ist ein 3-Approximationsalgorithmus.*

Beweis: Wir zeigen:

$$\sum_{i \in F, j \in C} c_{ij} x_{ij} + 3 \cdot \sum_{i \in F} f_i y_i \leq 3 \cdot \sum_{j \in C} \alpha_j \leq 3 \cdot \mathrm{Opt}_{frac}. \tag{3}$$

Daraus folgt der Satz, da die Kosten einer dualen Lösung eine untere Schranke für die Kosten einer optimalen Lösung sind. Für jede direkt verbundene Stadt j ist $c_{ij} = \alpha_j^e \leq 3 \cdot \alpha_j^e$ mit $i = \phi(j)$. Mit Lemma 2.8.12 erhalten wir $\sum_{i \in F, j \in C} c_{ij} x_{ij} \leq 3 \cdot \sum_{j \in C} \alpha_j^e$. Durch Addition der mit 3 multiplizierten Ungleichung unseres Korrolars folgt (3) und damit der Satz.

3 Online-Algorithmen

3.1 Beispiel: Ski-Verleih-Problem

Angenommen Du willst zum ersten Mal in Deinem Leben Ski fahren gehen. Lohnt es sich, für Dich eine komplette Ski Ausrüstung zu kaufen oder solltest Du sie Dir zunächst einmal ausleihen? Eine komplette Ski Ausrüstung kostet eine Menge Geld, aber wenn Du häufig fährst, dann ist sie die günstigste Lösung. Wenn sich aber nun herausstellt, dass Dir Ski fahren keinen Spass macht, Du Dich verletzt oder aus anderen Gründen nicht mehr Ski fahren möchtest oder kannst? Dann wäre es sicherlich besser, die Ski nicht gekauft zu haben. Leider weiß man beim Ski fahren nicht vorher, ob es einem gefällt oder nicht. Deshalb muss man die Entscheidung, Ski zu kaufen, unabhängig vom Wissen über die Zukunft treffen. Wir wollen nun das Ski Verleih Problem formalisieren: Nehmen wir einmal an, dass eine Ski Ausrüstung 800 Euro kostet, das Leihen der Ski 50 Euro. Ausserdem nehmen wir zur Vereinfachung an, dass Ski ewig halten.
Wir wollen nun versuchen eine gute Strategie zum Ski Kaufen zu finden. Dazu müssen wir uns erst einmal überlegen, was eine gute Strategie auszeichnet. Eine Strategie ist dann gut, wenn sie sicherstellt, dass wir - auch ohne die Zukunft zu kennen - nie viel mehr ausgeben, als nötig. Das bedeutet, dass wir sicher nicht sofort Ski kaufen, denn wenn wir nur einmal Ski fahren gehen, hätten wir 800 Euro ausgegeben, aber im schlechtesten Fall hätten wir nur 50 Euro benötigt. Das heißt, in diesem Fall hätten wir das 16-fache des benötigten ausgegeben. Andererseits macht es auch keinen Sinn immer wieder Ski auszuleihen, denn dann werden wir beliebig viel zahlen, wenn uns das Ski fahren gefällt.
Daher werden wir folgende Strategie verfolgen: Wir werden Ski ausleihen bis wir insgesamt 750 Euro Leihgebühr bezahlt haben oder wir aus irgendeinem Grunde nicht mehr Ski fahren wollen. Danach werden wir uns eine Ausrüstung zulegen. Das bedeutet, dass wir 15 mal leihen und beim 16. Mal kaufen. Wie gut ist diese Strategie? Angenommen wir würden die Zukunft kennen und wüßten genau, dass wir k mal Ski fahren. Wir unterscheiden jetzt zwei Fälle:

> **Fall 1:** Wir fahren weniger als 15 mal Ski, d.h. $k = 15$ bzw. $50k < 800$.
> Beste Lösung unter Kenntnis der Zukunft ist, die Ski zu leihen.
> Der Online-Algorithmus ist optimal.
>
> **Fall 2:** Wir fahren mindestens 16 mal Ski, d.h. $k \geq 16$ bzw. $50k \geq 800$.
> Optimal: Ski kaufen (800 Euro)
> Online: 15 mal leihen und einmal kaufen (750 + 800 Euro).
> $\rightarrow$ Wir geben ungefähr doppelt so viel aus.

Etwas genauer ausgedrückt: Sei

$$r = \frac{\text{Kaufen}}{\text{Leihen}} = \frac{800}{50}$$

Dann zahlt man mit unserer Strategie immer maximal das $(2 - \frac{1}{r})$–fache der bestmöglichen Lösung.

3.2 Einige Grundbegriffe

Bei Online Problemen handelt es sich um eine spezielle Art von *Optimierungsproblemen*. Im Beispiel des Ski Verleih Problems ging es beispielsweise darum, die Ausgaben zu minimieren. Im allgemeinen sagen wir, dass ein Optimierungsproblem $\mathcal{P}$ (wir betrachten hier o.b.d.A nur Minimierungsprobleme) aus einer Menge von Eingaben $\mathcal{I}$ und einer Kostenfunktion C besteht. Für jede Eingabe $I \in \mathcal{I}$ gibt es eine Menge von zulässigen Ausgaben (Lösungen) $F(I)$. Mit jeder zulässigen Lösung O in $F(I)$ sind Kosten $C(I, O)$ assoziiert, die die Kosten für Ausgabe O bei Eingabe I darstellen.

Ein Algorithmus berechnet nun bei Eingabe $I \in \mathcal{I}$ eine Lösung $\text{ALG}[I] \in F(I)$. Die Kosten, die wir mit dieser Ausgabe assoziieren, sind dann $C(I, \text{ALG}[I])$. Nun können wir definieren, was ein optimaler Algorithmus OPT ist. Ein solcher Algorithmus berechnet für jede Eingabe die Lösung mit den geringsten Kosten, d.h.,

$$\text{OPT}[I] = \min_{O \in F(I)} C(I, O) \ .$$

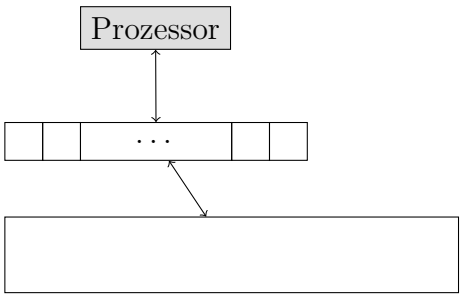
Bei Online Problem kommt nun noch ein weiterer Aspekt hinzu: Die Eingabe trifft als Sequenz ein. Wir bekommen also immer nur das nächste Element der Eingabe gezeigt und nicht wie bei 'normalen' Optimierungproblemen sofort die gesamte Eingabe.
Wir werden nun definieren, wie man die Güte eines Online Algorithmus bewertet. Wir sagen, dass ein Algorithm ALG *c-competitive* ist, wenn für jede (endliche) Eingabesequenz I gilt:

$$\text{ALG}[I] \leq c \cdot \text{OPT}[I] + \alpha \ .$$

Hierbei ist α eine Konstante. Ist $\alpha \leq 0$ so sagt man auch, dass der Algorithmus *streng c-competitive* ist. Das *competitive ratio* eines Algorithmus ALG bezeichnet das Infimum über alle Werte c mit der Eigenschaft, dass ALG c-competitive ist.

3.3 Paging-Problem

Jetzt werden wir uns mit dem sogenannten Paging Problem beschäftigen. Dabei geht es darum, dass wir einen Rechner haben, der eine gewisse Menge an schnellem Speicher (RAM) zur Verfügung hat. Ausserdem gibt es einen langsamen Sekundärspeicher (Festplatte), der eine größere Speicherkapazität hat. Wir messen die Speicherkapazität in Seiten und wir nehmen an, dass unser schneller Speicher Platz für k Seiten hat und in unserem langsamen Speicher N Seiten liegen.



Wenn nun ein Programm eine bestimmte Seite p_i benötigt, so muss das System diese Seite im schnellen Speicher bereit stellen. Ist die Seite bereits im schnellen Speicher vorhanden, so muss das System nichts tun. Ist die Seite jedoch nicht im schnellen Speicher, dann muss sie aus dem langsamen Speicher gelesen werden. Man sagt, das System macht einen *Seitenfehler*. Dabei muss natürlich Platz im schnellen Speicher geschaffen und eine alte Seite aus dem schnellen Speicher entfernt werden. Die Entscheidung, welche Seite aus dem Speicher entfernt wird, trifft ein Online Algorithmus und das zugehörige Problem wird als *Paging Problem* bezeichnet.

Das gleiche abstrakte Modell stellt natürlich auch andere zweistufige Speicherhierarchien dar, wie z.B. RAM und Prozessor Cache. Es gibt hierbei natürlich technische Unterschiede, aber diese werden in unserem Modell ignoriert.

3.4 Deterministische Ersetzungsstrategien

Wir werden nun zunächst einige deterministische Online Algorithmen für das Paging Problem kennenlernen und einige davon analysieren. Alle Strategien ersetzen natürlich nur dann eine Seite im schnellen Speicher, wenn eine Seite angefragt wird, die nicht im schnellen Speicher liegt.

Least Recently Used (LRU). Die LRU Strategie lagert immer die Seite aus, auf die am längsten *nicht* zugegriffen wurde.

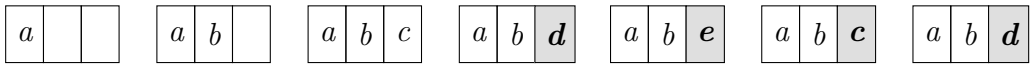
First In First Out (FIFO). Bei einem Seitenfehler ersetzt FIFO immer die Seite, die am längsten im schnellen Speicher liegt.

Last In First Out (LIFO). LIFO ersetzt die Seite, die als letztes in den schnellen Speicher eingefügt wurde.

Least Frequently Used (LFU). Der Algorithmus LFU lagert die Seite aus, die am wenigsten angefragt wurde, seit sie im schnellen Speicher ist.

Auf den ersten Blick erscheinen alle Strategien gewisse Vorzüge zu haben. Wir werden nun in diesem Kapitel bzw. in den Übungen diese Algorithmen in Bezug auf ihr competitive ratio analysieren.

Beispiel 3.4.1 *Betrachte die Anfragesequenz: a,b,c,a,d,e,a,a,b,c,d.*



LIFO erzeugt vier Seitenfehler (Laden einer Seite aus dem langsamen Speicher in den schnellen Speicher).
Noch schlechter: a,b,c,d,c,d,c,d,...

Beobachtung 3.4.2 *LIFO ist nicht competetive.*

Beweis: Es seien $p_1, \ldots, p_{k+1}$ die Seiten im langsamen Speicher. Wir nehmen an, dass LIFO am Anfang die Seiten $p_1, \ldots, p_k$ im schnellen Speicher hat. Nun betrachten wir die Sequenz:

$$\sigma = p_1, \ldots, p_k, p_{k+1}, p_k, p_{k+1}, \ldots, p_k, p_{k+1}.$$

LIFO macht auf dieser Sequenz ab Anfrage $k+1$ bei jeder Anfrage einen Seitenfehler. Ein optimalen Algorithmus hält aber einfach die Seiten p_k und p_{k+1} im Speicher und macht gar keinen Seitenfehler mehr.

Beobachtung 3.4.3 *LFU ist nicht competetive.*

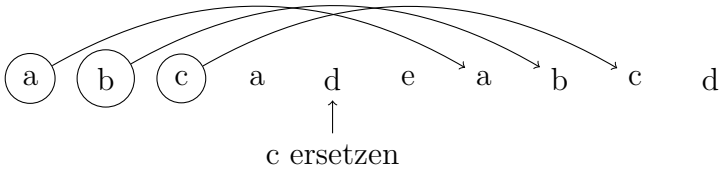
Beweis: Sei ℓ eine positive ganze Zahl. Wir betrachten die Sequenz:

$$\sigma = p_1^\ell, p_2^\ell, \ldots, p_{k-1}^\ell, (p_k, p_{k+1})^{\ell-1}.$$

Hier macht LFU bei jeder Anfrage nach den ersten $(k-1) \cdot \ell$ Anfragen einen Seitenfehler. OPT hingegen macht nur einen einziger Seitenfehler. Da man ℓ beliebig groß machen kann, ist LFU nicht competitive.

3.5 Ein optimaler Offline-Algorithmus für das Paging-Problem

Algorithmus(Longest Forward Distance, LFD)
LFD ersetzt die Seite, auf die als letztes wieder zugegriffen wird.



Satz 3.5.1 LFD *ist ein optimaler Offline Algorithmus für das Paging Problem.*

Beweis: Wir zeigen, dass jeder optimale Offline Algorithm OPT in LFD umgeformt werden kann, ohne dass zusätzliche Kosten auftreten. Die Umformung funktioniert Schritt für Schritt. Zunächst konstruieren wir aus OPT einen optimalen Offline Algorithmus OPT_1, der den ersten Zugriff wie LFD macht. Aus OPT_1 konstruieren wir dann einen optimalen Algorithmus OPT_2, der die ersten beiden Zugriffe wie LFD macht, usw.

Lemma 3.5.2 *Sei σ eine Anfragesequenz der Länge n. Sei OPT_i ein optimaler Paging Algorithmus für σ, der die ersten i Seitenfehler wie* LFD *bearbeitet. Dann gibt es eine optimalen Paging Algorithmus OPT_{i+1} fü4 σ, der die ersten $i+1$ Seitenfehler wie* LFD *behandelt.*

Beweis: Sei OPT_i gegeben. Wir nehmen an, dass OPT_i und OPT_{i+1} nach Seitenfehler $i+1$ die Seiten in $X \cup \{v\}$ bzw. $X \cup \{u\}$ im Speicher haben, wobei X eine Menge von $k-1$ Seiten ist. Gilt nun $u = v$, so folgt das Lemma mit $\text{OPT}_{i+1} = \text{OPT}_i$. Wir nehmen daher an, dass $u \neq v$ gilt.
Wir wollen nun OPT_{i+1} so konstruieren, dass er immer mindestens $k-1$ Seiten gemeinsam mit OPT_i im schnellen Speicher hat und nicht mehr Seitenfehler macht als OPT_i. Wir werden zeigen, dass irgendwann beide Algorithmen dieselben Seiten im Speicher haben. Falls dann OPT_{i+1} höchstens so viele Seitenfehler gemacht hat wie OPT_i sind wir fertig. Denn dann wird sich OPT_{i+1} genauso wie OPT_i verhalten.
Wird nun eine Seite ausser v angefragt und OPT_i ersetzt eine Seite ausser v, so macht OPT_{i+1} genau dasselbe wie OPT_i. (Wird u angefragt, so ersetzt OPT_i u und beide Algorithmen haben immer noch $k-1$ gemeinsame Seiten im Speicher.) Ersetzt OPT_i v, so ersetzt OPT_{i+1} u und beide haben dieselben Seiten im Speicher.
Wird v angefragt, so hat OPT_{i+1} einen Seitenfehler und OPT_i nicht. Da v aber nach der LFD Regel ersetzt wurde, muss es nach dem $(i+1)$-ten Seitenfehler Seite u mindestens einmal angefragt worden sein. Die erste solche Anfrage hatte einen Seitenfehler für OPT_i zur Folge, aber keinen für OPT_{i+1}. Daher ist die Gesamtanzahl Seitenfehler von OPT_{i+1} höchstens die von OPT_i. Zu guter letzt kann also OPT_{i+1} u durch v ersetzen und beide Algorithmen haben dieselben Seiten im Speicher und das Lemma folgt. Aus Lemma 3.5.2 folgt mit $\text{OPT} = \text{OPT}_0$ und $\text{LFD} = \text{OPT}_n$ sofort auch Satz 3.5.1.

3.6 Markierungsalgorithmen

Definition 3.6.1 *Sei δ eine Anfangssequenz. Wir teilen δ wie folgt in Phasen auf: Phase 0 ist die leere Sequenz. Für jedes i ist Phase i die maximale Sequenz, die auf Phase $i-1$ folgt und Anfragen nach höchstens k verschiedenen Seiten enthält. Man nennt die durch die Phasen gegebene Unterteilung auch k–Phasen Partition.*

Beispiel 3.6.2 *Betrachte folgende Sequenz und die angegebene 3-Phasen Partition:*
abcab|dea|bcaab|efa|gha

Definition 3.6.3 *(Markierungsalgorithmus):*
Ein Markierungsalgorithmus verwaltet für jede Seite im langsamen Speicher ein Bit. Ist das Bit gesetzt, so heißt die Seite markiert, an sonsten heißt sie unmarkiert.
Zu Beginn einer Phase werden alle Markierungen gelöscht. Während einer Phase wird eine Seite markiert, wenn auf sie zugegriffen wird. Keine markierte Seite wird aus dem schnellen Speicher entfernt.

Satz 3.6.4 *Sei ALG ein Markierungsalgorithmus mit schnellem Speicher der Größe k. Dann ist ALG k-competitive.*

Beweis: Wir zeigen zunächst, dass jeder Markierungsalgorithmus höchstens k Seitenfehler pro Phase macht. Dies folgt sofort, da jede Seite nach ihrem ersten Zugriff markiert ist und somit nicht mehr aus dem schnellen Speicher entfernt wird. Also kann ALG höchstens einen Fehler pro unterschiedlicher Seite in einer Phase machen.
Sei nun q die erste Anfrage in Phase i. Wir betrachten die Sequenz, die mit der zweiten Anfrage von Phase i beginnt und bis einschließlich zur ersten Anfrage in Phase $i+1$ reicht. Zu Anfang dieser Sequenz hat der optimale Algorithmus OPT Seite q und $k-1$ andere Seiten im schnellen Speicher. Unsere Sequenz enthält aber Anfragen nach k unterschiedlichen Seiten, die auch $\neq q$ sind. Also hat OPT mindestens einen Seitenfehler.
Für jede Phase (außer der letzten) hat ALG höchstens k Seitenfehler und OPT mindestens einen. Also folgt:

$$ALG(\delta) \leq k \cdot OPT(\delta) + \alpha \qquad \alpha = k.$$

Somit ist ALG K-competitive.

□

Beispiel 3.6.5 *Betrachte die Sequenz $abca|deadea|fa$. LRU lädt zuerst a, b, c in den Speicher. Wenn d angefragt wird, wird b durch d ersetzt. Wenn e angefragt wird, wird c durch e ersetzt. Wenn f angefragt wird, wird d durch f ersetzt. Alle anderen Anfragen können ohne Nachladen beantwortet werden.*

Um nun zu zeigen, dass LRU k-competitive ist, beweisen wir, dass LRU ein Markierungsalgorithmus ist.

Lemma 3.6.6 *LRU ist ein Markierungsalgorithmus.*

Beweis: Wir haben eine Anfangssequenz δ fest und betrachten ihre k Phasen Partition.
Annahme: LRU ist kein Markierungsalgorithmus. Dann ersetzt LRU während einer Phase eine markierte Seite x. Wir betrachten den ersten Zugriff auf x während dieser Phase. Hier wird x markiert und ist zu diesem Zeitpunkt die Seite, die als letztes angefragt wurde. Zum Zeitpunkt wenn LRU x ersetzt, ist x die Seite, die am längsten nicht angefragt wurde. Damit wurden alle k Seiten im schnellen Speicher sowie die Seite, die den Seitenfehler verursacht hat, alle in einer Phase angefragt. Dies sind aber $k+1$ Elemente. Widerspruch! Also ist LRU ein Markierungsalgorithmus.

□

Aus Satz 3.6.4 und Lemma 3.6.6 folgt sofort:

Korollar 3.6.7 *LRU ist k-competitive.*

3.7 Eine untere Schranke für deterministische Paging Algorithmen

Wir zeigen nun, dass kein deterministischer Paging Algorithmus ein besseres competitive ratio als k haben kann. Dazu benötigen wir zunächst folgendes Lemma:

Lemma 3.7.1 *Für jede endliche Sequenz δ von Anfragen, die aus einer Menge von $k+1$ Seiten ausgewählt wurden, gilt $LFD(\delta) \leq \frac{|\delta|}{k} + k$.*

Beweis: Die ersten k Seiten führen zu k Seitenfehlern. Danach betrachte die Seite p, die bei Anfrage r ersetzt wird. Nach der Definition von LFD und weil es nur $k+1$ verschiedene Seiten gibt, müssen alle k Seiten im schnellen Speicher angefragt werden, bevor p wieder angefragt wird. Also macht LFD höchstens einen Seitenfehler alle k Anfragen.

□

Als nächstes beweisen wir unsere untere Schranke.

Idee:
Ein „grausamer Gegenspieler“ fragt immer die Seite an, die gerade nicht im schnellen Speicher ist.

Satz 3.7.2 *Jeder deterministische Paging Algorithmus hat ein competitive ratio von mindestens k.*

Beweis: Wir nehmen an, dass nur $k+1$ verschiedene Seiten $p_1, \ldots, p_{k+1}$ angefragt werden. Wir betrachten nun einen beliebigen (aber festen) deterministischen Paging Algorithmus ALG. Konstruiere eine Sequenz δ mit

$$ALG(\delta) = |\delta| = \tfrac{k \cdot |\delta|}{k} + k^2 - k^2 = k \cdot \left(\tfrac{|\delta|}{k} + k\right) - k^2 \geq LFD(\delta) \cdot k - k^2 = k \cdot OPT - k^2.$$

Wir nehmen an, dass im schnellen Speicher von ALG am Anfang die Seiten $p_1, \ldots, p_k$ stehen und definieren die Sequenz dadurch, dass immer die Seite angefragt wird, die nicht im schnellen Speicher von ALG liegt. Diese Sequenz ist wohldefiniert, weil ALG deterministisch ist. ALG hat in jedem Schritt einen Seitenfehler, d.h. es gilt $ALG(\delta) = |\delta|$. Die untere Schranke folgt, weil man δ beliebig lang machen kann und damit das additive $-k^2$ ignoriert werden kann.

□

3.8 Paging mit verschieden viel Speicher

Wir werden nun ein weiteres Konzept zur Analyse von Online Algorithmen einführen. Dazu erlauben wir, dass der Online Algorithmus mehr Ressourcen benutzen darf als der Offline Algorithmus. Im Beispiel Paging heißt das, dass ein Online Algorithmus einen schnellen Speicher der Größe k besitzt während der Offline Algorithmus mit schnellem Speicher der Größe $h \leq k$ auskommen muss. Wir haben also im Fall Paging unser Modell einfach nur erweitert, denn unsere bisheriges Modell ergibt sich aus dem Fall $h = k$.
Wir werden nun unsere Ergebnisse über Markierungsalgorithmen auf dieses neue Modell erweitern. So werden wir zeigen, dass ein Online Markierungsalgorithmus mit doppelt so viel Speicher wie Offline 2-competitive ist. In diesem neuen Licht erscheinen nun Markierungsalgorithmen deutlich besser als in unserer bisherigen Analyse. Denn wir können die untere Schranke von k für deterministische Online Algorithmen dadurch umgehen, dass wir unserem Algorithmus doppelt so viel Speicher geben wie dem Offline Algorithmus.

Satz 3.8.1 *Sei* ALG *ein Markierungsalgorithmus mit schnellem Speicher der Größe k und sei* OPT *ein optimaler Offline Algorithmus mit schnellem Speicher der Größe $h \leq k$. Dann ist* ALG *$\frac{k}{k-h+1}$-competitive.*

Beweis: Unser Beweis ist eine einfache Erweiterung des Beweises von Satz 3.6.4.
Wir betrachten wieder die k-Phasen Partitionierung einer beliebigen Eingabesequenz σ. Wir wissen bereits aus dem Beweis von Satz 3.6.4, dass jeder Markierungsalgorithmus höchstens k Seitenfehler pro Phase macht.
Sei nun q die erste Anfrage in Phase i. Wir betrachten die Sequenz, die mit der zweiten Anfrage von Phase i beginnt und bis einschließlich zur ersten Anfrage der Phase $i+1$ reicht. Zu Anfang dieser Sequenz hat OPT Seite q und $h-1$ andere Seiten im schnellen Speicher. Außerdem gibt es Anfragen nach k verschiedenen Seiten (ohne q mitzuzählen) in dieser Sequenz. Also hat OPT mindestens $k-(h-1) = k-h+1$ Seitenfehler.
Für jede Phase (außer der letzten) hat ALG höchstens k Seitenfehler und OPT mindestens $k-h+1$. Also folgt:

$$\text{ALG}(\sigma) \leq \tfrac{k}{k-h+1} \cdot \text{OPT}(\sigma) + \alpha,$$

wobei α eine Konstante ist.

3.9 Randomisierte Paging Algorithmen

Wir werden einen randomisierten Markierungsalgorithmus entwickeln. Zunächst einmal müssen wir uns überlegen, was eigentlich das competitive ratio eines randomisierten Algorithmus sein soll.
Wir werden hier nur den sogenannten *blinden* Gegenspieler kennenlernen. Sei ALG ein randomisierter Online Algorithmus. Der blinde Gegenspieler („oblivious adversary“) wählt in Kenntnis des Algorithmus (und seiner Wahrscheinlichkeitsverteilung) eine Eingabesequenz σ aus. Er sieht jedoch nicht den Ausgang der Zufallsexperimente (daher blinder Gegenspieler). Wir sagen, dass ALG c-competitive gegen einen blinde Gegenspieler ist, wenn es eine Konstante α gibt, so dass für jedes solche σ gilt:

$$\mathbf{E}\big[\text{ALG}(\sigma)\big] \leq c \cdot \text{OPT}(\sigma) + \alpha.$$

Der Algorithmus, den wir entwickeln, nutzt eine k–Phasen Partition und verwaltet wie die bisherigen Markierungsalgorithmen für jede Seite im schnellen Speicher eine Markierung.

Algorithmus 3.9.1

Zu Beginn einer Phase werden alle Seiten im schnellen Speicher unmarkiert.

Wird eine neue Seite in den schnellen Speicher genommen oder auf eine im schnellen Speicher vorhandene zugegriffen, so wird diese markiert.

Muss eine Seite entfernt werden, so wird eine zufällige unmarkierte Seite entfernt.

Da in einer Phase genau k verschiedene Seiten auftreten, sind am Ende einer Phase alle Seiten markiert und eine neue Phase beginnt beim nächsten Seitenfehler.

Beispiel 3.9.2 *Betrachte die Sequenz | abacabc | daaba | cde | fabff.*

a, b, c werden in den Speicher geladen

bei d: alle Seiten werden unmarkiert,
b wird zufällig gewählt und durch d ersetzt,
d wird markiert

bei a: a wird markiert

bei b: nur c ist unmarkiert und wird durch b ersetzt,
b wird markiert

bei c: eine neue Phase beginnt
a wird zufällig gewählt und durch c ersetzt,
c wird markiert

bei d: d wird markiert

bei e: nur b ist unmarkiert und wird durch e ersetzt

bei f: eine neue Phase beginnt,
e wird zufällig gewählt und durch f ersetzt,
f wird markiert

bei a: d wird zufällig gewählt und durch a ersetzt,
d wird markiert

bei b: c wird zufällig gewählt und durch b ersetzt,
b wird markiert.

Definition 3.9.3 *Mit H_k bezeichnen wir die Harmonische Reihe, die durch*

$$H_k := \sum_{i=1}^{k} \frac{1}{i}$$

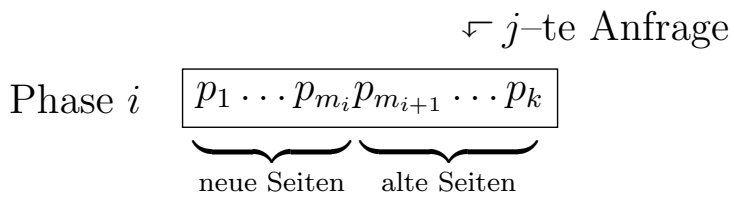
definiert ist.

Bemerkung 3.9.4 *Es gilt*

$$\ln(k) \leq H_k \leq \ln(k) + 1.$$

Satz 3.9.5 *Algorithmus* MARK *mit schnellem Speicher der Größe k ist $2H_k$-competitive gegen einen blinden Gegenspieler.*

Beweis: Wir betrachten eine beliebige Sequenz σ und die zugehörige k-Phasen Partition. Für Phase i bezeichnen wir die Seiten, die direkt vor Beginn von Phase i im Speicher liegen als *alt*. Jede nicht alte Seite, die in Phase i angefragt wird, heißt *neu*. Sei m_i die Anzahl neuer Seiten, die in Phase i angefragt werden. Da MARK ein Markierungsalgorithmus ist, können wir o.b.d.A. annehmen, dass jede Seite, die in Phase i angefragt wird, genau einmal angefragt wird. Dies ist richtig, weil jede Seite nach der ersten Anfrage markiert ist und somit bis zum Ende der Phase im schnellen Speicher bleibt. Damit können weitere Anfragen nach einer Seite auch keine Seitenfehler bei Online verursachen und wir brauchen sie daher auch nicht betrachten.
Wir zeigen zunächst, dass OPT mindestens $\frac{1}{2}\sum_i m_i$ Seitenfehler macht, unabhängig von der Anordnung der Seiten innerhalb der Phase. Wir stellen fest, dass in Phase $i-1$ und i zusammen mindestens $k + m_i$ verschiedene Seiten angefragt werden. Daher macht OPT mindestens m_i Seitenfehler. In der ersten Phase macht OPT mindestens m_1 Seitenfehler. Da wir immer zwei Phasen zusammenfassen können, können wir entweder die geraden oder die ungeraden m_i als untere Schranke benutzen. Eine der beiden Summen ist mindestens $\frac{1}{2}\sum_i m_i$. Also sind die Kosten eines optimalen Offline Algorihmus auch mindestens $\frac{1}{2}\sum_i m_i$.
Jetzt analysieren wir die Anzahl der Seitenfehler, die der Online Algorithmus erwartet macht. Innerhalb einer Phase ist es offensichtlich am schlechtesten für Online, wenn zunächst alle neuen Seiten angefragt werden. Wir müssen nun nur noch die erwartete Anzahl von Seitenfehlern für die restlichen $k - m_i$ Anfragen nach alten Seiten berechnen.
Wir betrachten nun die j-te Anfrage nach einer alten Seite p in Phase i.



Zu diesem Zeitpunkt sind noch genau $k - m_i - (j-1)$ unmarkierte alte Seiten im Speicher und es gibt insgesamt noch $k - (j-1)$ unmarkierte alte Seiten (von denen einige nur im langsamen Speicher sind). Wir erinnern uns, dass bei jedem Seitenfehler eine der alten unmarkierten Seite zufällig und gleichverteilt gewählt und aus dem schnellen Speicher entfernt wird. Daher gilt, dass zum Zeitpunkt von Anfrage j jede unmarkierte alte Seite mit derselben Wahrscheinlichkeit noch im schnellen Speicher steht. Also gilt:

$$\Pr[p \text{ ist im schnellen Speicher}] = \frac{k - m_i - (j-1)}{k - (j-1)} \ .$$

Und daraus folgt

$$\Pr[p \text{ nicht im schnellen Speicher}] = 1 - \frac{k - m_i - (j-1)}{k - (j-1)} = \frac{m_i}{k - j + 1}$$

Damit gilt für die erwartete Anzahl Seitenfehler in Phase i:

$$\mathbf{E}\big[\#\text{ Seitenfehler}\big] = m_i + \sum_{j=1}^{k-m_i} \frac{m_i}{k-j+1} = m_i + m_i(H_k - H_{m_i}) = m_i(H_k - H_{m_i} + 1) \leq m_i H_k \ .$$

Damit folgt, dass MARK $2H_k$–competitive ist.

4 Online-Algorithmen für das Listen-Zugriffsproblem

4.1 Amortisierte Analyse

Wir werden nun anhand eines Beispiels eine Analysetechnik kennenlernen, die zur Analyse von dynamischen Datenstrukturen eingeführt wurde und im Bereich Online Algorithmen eine große Rolle spielt. Es handelt sich um die sogenannte *amortisierte Analyse*. Bei dieser Technik geht man davon aus, dass jede Operation mit einer gewissen Menge Geld startet. Mit diesem Geld bezahlt man für das Ausführen von Operation und es ist möglich, Geld, das man für eine Operation nicht verbraucht hat, für zukünftige, unter Umständen teurere, Operationen zu sparen.
Man könnte auch sagen, dass man mit amortisierter Analyse die *durchschnittlichen Kosten* einer *worst-case Eingabe* analysiert.

4.1.1 Amortisierte Analyse von Datenstrukturen

Im Bereich Datenstrukturen wird amortisierte Analyse benutzt, um die durchschnittlichen Kosten einer Sequenz $\sigma_1, \ldots, \sigma_n$ von Anfragen mit Kosten $c_1, \ldots, c_n$ zu analysieren. Dabei gilt:

$$\text{Worst-cast Kosten} = \max_i c_i$$

$$\text{Gesamtkosten} = \sum_i c_i$$

$$\text{Amortisierte Kosten von } \sigma = \frac{\sum_i c_i}{n}$$

Als amortisierte Kosten (von Operationen auf) der Datenstruktur bezeichnet man dann das Maximum der amortisierten Kosten von σ über alle Folgen σ. Weitergehende Informationen zur amortisierten Analyse von Datenstrukturen findet man z.B. in 'Introduction to Algorithms' von Cormen, Leisserson und Rivest.

4.1.2 Beispiel: Amortisierte Analyse eines Binärzählers

Wir werden nun amortisierte Analyse anhand des Beispiels eines k-Bit Binärzählers kennenlernen. Zu Beginn ist ein solcher Zähler mit 0 initialisiert und es gibt nur eine Operation, nämlich das Erhöhen des Zählers um 1. Dies geschieht, indem man die Bits von rechts nach links durchgeht und flippt, bis man ein Bit von 0 auf 1 geflippt hat. Die Anzahl der Flips gibt dabei die Kosten des Erhöhens an.
Offensichtlich gilt, dass die Kosten für das Erhöhen des Zählers im worst-case k sind. Doch was sind die *amortisierten Kosten* für das Erhöhen des Zählers? Wir werden zeigen:

Satz 4.1.1 *Die amortisierten Kosten für das Erhöhen eines Zählers sind* $O(1)$.

Wir werden Satz 4.1.1 nun aus 3 verschiedenen Blickwinkeln beweisen. Aus der Sicht eines Ingenieurs, eines Bankers und eines Physikers.

Die Sichtweise des Ingenieurs Die Idee hier ist, das ganze Problem einfach durchzurechnen. Wir stellen fest, dass das rechteste Bit b_0 jedesmal geflippt wird. Das zweitrechteste Bit b_1 wird jedes zweite mal geflippt. Bit b_i wird jedes 2^i-te mal geflippt. Es ergibt sich für die amortisierten Kosten $\widehat{c}$:

$$\widehat{c} = \frac{\sum_{i=0}^{\lfloor \log n \rfloor} \lfloor \frac{n}{2^i} \rfloor}{n} < 2 \ .$$

Die Sichtweise des Bankers (Die Konto Methode) Bei jeder Operation werden zunächst $\widehat{c}$ Euro gezahlt. Von diesen $\widehat{c}$ Euro müssen zunächst die Kosten der Operation bezahlt werden. Evtl. übrig gebliebenes Geld kann auf ein Konto eingezahlt werden. Sind die Kosten für eine Operation größer als $\widehat{c}$, so müssen diese mit Geld vom Konto bezahlt werden. Es gilt:

Satz 4.1.2 *Hat ein System immer einen nicht negativen Kontostand, dann ist $\widehat{c}$ eine obere Schranke für die amortisierten Kosten des Systems.*

Um nun unser Beispiel des Binärzählers zu analysieren setzen wir $\widehat{c} = 2$. Wir können für jeden Euro ein Bit flippen. Wir werden für jedes Bit einen eigenen Kontostand einführen. Der Kontostand eines Bits ist 1, wenn der Wert des Bits 1 ist und 0 sonst. Nach der ersten Operation zeigt das kleinste Bit den Wert 1 und wir haben einen Euro für das flippen gezahlt und den übrigen Euro auf das Konto des kleinsten Bits gezahlt. Also stimmt unsere Behauptung.
Nehmen wir an, unsere Behauptung stimmt für die ersten m Operationen. In Operation $m+1$ werden nun die kleinsten j Bits des Zählers geflippt. Per Definition geschieht dies, wenn die kleinsten $j-1$ Bits den Wert 1 gezeigt haben. Wir zahlen das Flippen der $j-1$ Bits von den Konten dieser Bits und das Flippen des j-ten Bit von unseren 2 Euro für die Operation. Den übrigen Euro zahlen wir auf das Konto von Bit j ein. Da die letzen $j-1$ Bits auf 0 gestellt wurden, durften wir jeweils den Euro abheben. Einen Euro müssen wir einzahlen, weil das j-te Bit auf 1 gesetzt wird. Da wir diesen Euro eingezahlt haben, stimmt unsere Behauptung auch nach Operation m und ist damit per Induktion immer korrekt.

Die Sichtweise des Physikers (Die Potential Funktion Methode) Überschüssiges Geld wird als *Potential* (Energie) des gesamten Systems gespeichert. Dieses Potential wird angezapft, um für Operationen zu bezahlen, die mehr Energie kosten als sie bringen. Das Potential zum Zeitpunkt i wird als Φ_i bezeichnet. Wir setzen:

$$\widehat{c_i} = c_i + \Delta\Phi_i \ ,$$

wobei $\Delta\Phi_i = \Phi_i - \Phi_{i-1}$. Damit gilt:

$$\sum_{i=1}^{n} \widehat{c_i} = \sum_{i=1}^{n} (c_i + \Phi_i - \Phi_{i-1}) = \sum_{i=1}^{n} c_i + \Phi_n - \Phi_0 \ .$$

Satz 4.1.3 *Falls $\Phi_n \geq \Phi_0$, so ist $\widehat{c_i}$ eine obere Schranke für die amortisierten Kosten in Schritt i.*

Wir wenden nun die Potential Methode auf unser Binärzähler Problem an. Sei Φ_i die Anzahl Bits, die zum Zeitpunkt i auf 1 gesetzt sind. Sei $\Phi_0 = 0$. Wir betrachten die m-te Operation. Sei j die Position des Bits, das auf 1 geflippt wird (und alle Bits an Positionen kleiner j werden auf 0 geflippt). Dann gilt (da es j 0-Bits an den Positionen 0 bis $j-1$ gibt):

$$\Phi_i - \Phi_{i-1} = 1 - j \ .$$

Wir wollen zeigen, dass immer $\widehat{c_i} = 2$ gilt. Also:

$$\widehat{c_i} = 2 = (j+1) + (1-j) = c_i + \Delta\Phi_i \ .$$

4.2 Das Listen-Zugriff-Problem

Wir werden uns als erstes mit dem *Listen-Zugriff-Problem* beschäftigen. Dabei nehmen wir an, dass ℓ Datensätze in einer verketteten Liste abgespeichert sind. Zu jedem Datensatz gibt es einen Schlüssel. Datensätze könnten dann z.B. Kundendaten sein, der Schlüssel ist der Kundenname.

Zugegriffen wird auf die Datensätze wie folgt: Es wird die Liste von vorne nach hinten durchgesehen, bis man auf den Datensatz trifft bzw. weiß, dass der Datensatz nicht in der Liste vorhanden ist. Wir nehmen an, dass wir für eine solche Operation x Schritte benötigen, wobei x die Position des Datensatzes in der Liste ist. Desweiteren nehmen wir an, dass neue Datensätze eingefügt werden können und auch Datensätze gelöscht werden dürfen. Das Einfügen eines Datensatzes kostet $\ell+1$ Zeitschritte, wobei ℓ die Länge der Liste vor Einfügen des Datensatzes ist. Das Löschen des x-ten Datensatzes in der Liste kostet x Zeitschritte.
Bisher haben wir eine Implementierung (durch verkettete Listen) des abstrakten Datentyps Wörterbuch beschrieben. Die Implementierung durch verkettete Listen ist sehr einfach und wird daher häufig für Wörterbücher moderater Größe genutzt.

Aber wo ist nun ein Online Problem versteckt?

Dazu müssen wir nur folgendes erlauben: Nach einem Zugriff oder dem Einfügen bzw. Löschen eines Datensatzes in der Liste ist es erlaubt, die Liste neu zu organisieren. Wir messen die Kosten zur Reorganisation der Liste in Transpositionen (Vertauschungen benachbarter Elemente). Es gibt jedoch eine Sonderregel. Einen Datensatz, auf den gerade zugegriffen wurde, dürfen wir *umsonst* näher zum Listenanfang verschieben. Die Idee ist hier, dass wir ja bei der Suche nach diesem Datensatz Zeiger zu den bisher besuchten Positionen aufrecht erhalten können. Alle anderen Transpositionen heißen *bezahlt* und kosten einen Zeitschritt. Damit haben wir folgendes Online Problem:

Das Listen-Zugriff-Problem: Gesucht ist eine Strategie, unsere Liste nach jedem Zugriff geschickt zu reorganisieren, um die Gesamtzugriffszeit niedrig zu halten (ein gutes competitive ratio zu erzielen).

In unserer Beschreibung haben wir ein dynamisches Problem beschrieben; wir haben Einfügen und Löschen von Datensätzen erlaubt. Eine andere Variante des Problems ist das *statische Listen-Zugriff-Problems*. Dabei werden nur Zugriffsoperationen betrachtet.

4.2.1 Die 'move to front' Strategie

Wir werden nun den ersten Online Algorithmus für das Listen-Zugriff-Problem kennenlernen. Unsere Strategie ist sehr einfach: Nach jedem Zugriff oder dem Einfügen eines Datensatzes bewegen wir diesen an die erste Position der Liste, ohne die relative Ordnung der anderen Elemente zu verändern. Wir werden diesen Algorithmus mit MTF (für 'move to front') abkürzen.
Wir werden nun zeigen, dass MTF ein competitive ratio von 2 erreicht:

Satz 4.2.1 *Algorithmus* MTF *ist* 2*-competitive.*

Beweis: Wir müssen zeigen, dass für jede beliebige Sequenz von Anfragen (Zugriff, Einfügen, Löschen) die von MTF benötigten Zeitschritte höchstens doppelt so viele sind, wie die, die ein optimaler Offline Algorithmus benötigt. Wir stehen nun vor einem Problem: Wir kennen wir keine *einfache* Beschreibung der optimalen Offline Strategie. Ein Grund hierfür ist sicherlich, dass die Offline Variante des Problems NP-vollständig ist. Wie können wir dann jedoch eine Aussage über das competitive ratio machen?
Dazu werden wir Potentialfunktion nutzen. Der Wert der Potentialfunktion Φ_i zum Zeitpunkt i ist die Anzahl der Inversionen in der Liste von MTF bezüglich der Liste von OPT. Wir bezeichnen dazu mit c_i^{on} die Kosten des Online Algorithmus für die i-te Operation und mit c_i^{off} die entsprechenden Kosten des optimalen Offline Algorithmus. Mit a_i bezeichnen wir die amortisierten Kosten des Online Algorithmus. Wir wollen zeigen:

$$a_i = c_i^{\text{on}} + \Delta\Phi_i \leq 2c_i^{\text{off}} \ .$$

Im Unterschied zu unseren bisherigen Beispielen können die amortisierten Kosten für die einzelnen Operationen also verschieden sein.

Wir beobachten zunächst, das $\Phi_0 = 0$ gilt, da MTF und OPT zu Beginn dieselbe Liste von Datensätzen haben.
Wir wollen nun zeigen, dass die amortisierten Kosten für die i-te Anfrage höchstens $(2s - 1) + P - F$ betragen. Dabei bezeichnet s die Suchkosten und P die Anzahl bezahlter Transpositionen von OPT bei der Bearbeitung der i-ten Anfrage. F bezeichnet die entsprechende Anzahl Transpositionen von OPT, die umsonst sind.
Wir betrachten zunächst nur Zugriffsoperationen. Wir werden zunächst die Kosten analysieren, die MTF verursacht und danach die Kosten, die OPT verursacht. Sei x der Datensatz, auf den zugegriffen werden soll. Es bezeichne j die Position von x in der Liste von OPT und k die Position in der Liste von MTF. Weiterhin sei v die Anzahl der Datensätze, die in der Liste von MTF *vor* x stehen, aber in der Liste von OPT *hinter* x sind. Dann liegen $k - 1 - v$ Datensätze in beiden Listen vor x. Ausserdem gilt $k - 1 - v \leq j - 1$, da x an Position j in der Liste von OPT steht. Also gilt $k - v \leq j$.
Bewegt also nun MTF x an die erste Position der Liste, so erzeugt er $k-1-v$ neue Inversionen bezüglich der Liste von OPT. Desweiteren eliminiert diese Aktion auch v Inversionen. Also ist der Beitrag zu den amortisierten Kosten:

$$k + (k - 1 - v) - v = 2(k - v) - 1 \leq 2j - 1 = 2s - 1 \ .$$

Die Suche von OPT beeinflusst die Potentialfunktion und damit die amortisierten Kosten nicht. Jede bezahlte Transposition von OPT erhöht unser Potenzial höchstens um 1, jede Transposition, die umsonst ist, verringert es um 1. Daraus folgt sofort:

$$a_i \leq 2s - 1 + P - F \leq 2(s + P) = 2c_i^{\text{off}} \ .$$

Insgesamt ergibt sich für die Kosten von MTF

$$\text{MTF}(\sigma) = \sum_{i=1}^{n} c_i^{\text{on}} = \Phi_0 - \Phi_n + \sum_{i=1}^{n} a_i \leq 2\text{OPT}(\sigma) \ ,$$

Um dasselbe Ergebnis für Einfügen oder Zugriff auf nicht vorhandene Datensätze zu zeigen, können wir die gleichen Argumente verwenden und müssen $j = \ell + 1$ setzen, wobei ℓ die Länge der Listen bezeichnet. Der Fall, dass ein Datensatz gelöscht wird ist noch einfacher, da hier gar keine neuen Inversionen erzeugt werden und somit der Beitrag der Bewegung von MTF zu den amortisierten Kosten $k - v \leq j = s \leq 2s - 1$ ist.

5 Suchbäume

5.1 Selbstanpassende Suchbäume

Wir haben bereits in Kapitel 4.2 eine einfache Datenstruktur (lineare Liste) für das Wörterbuch kennengelernt und diese mit Hilfe von Online Analyse ausgewertet. Eine lineare Liste ist jedoch sicher keine effiziente Implementierung eines Wörterbuchs. Daher betrachten wir nun Suchbäume. Aufgrund der komplizierteren Struktur von Suchbäumen ist hier eine (nichttriviale) competitive Analyse nicht möglich. Allerdings werden wir in einem etwas vereinfachten Modell ein erstaunliches Ergebnis zeigen: Es gibt Suchbäume, die bei einer beliebigen Anfragesequenz σ asymptotisch genausogut sind wie ein optimalen statischer Suchbaum für σ. Ein optimaler statischer Suchbaum ist ein Suchbaum, der die Zugriffshäufigkeit auf die einzelnen Elemente berücksichtigt (d.h., die Elemente, auf die häufig zugegriffen wird, stehen nahe an der Wurzel). Eine Art von Suchbäumen mit dieser Eigenschaft sind die sogenannten Splay-Bäume. Diese benutzen eine Funktion Splay mit deren Hilfe ein Element des Suchbaums an die Wurzel rotiert wird. Dabei sorgen die Rotationen dafür, dass der Suchbaum auf lange Sicht balanziert wird. Man kann nämlich zeigen, dass die amortisierten Zugriffskosten $O(\log n)$ sind.

5.1.1 Operationen auf Splay-Bäumen

Folgende Operationen werden von Splay-Bäumen zur Verfügung gestellt.

Splay (x)**.** Die Operation Splay ist die wichtigste Operation in Splay-Bäumen. Mit ihrer Hilfe werden alle anderen Operationen implementiert. Eine Splay Operation holt das Element x mit Hilfe von Rotationen an die Wurzel des Suchbaums. Ist das Element x nicht im Suchbaum vorhanden, so wird der Vorgänger oder Nachfolger von x an die Wurzel rotiert.

Access (x)**.** Auf das Element x wird zugegriffen. Diese Operation wird durch Splay (x) und Ausgabe der Wurzel realisiert.

Insert (x)**.** Beim Einfügen eines Elements wird zunächst eine Splay (x) Operation durchgeführt. Diese rotiert den Vorgänger oder Nachfolger von x an die Wurzel. Ist nun die Wurzel r der Nachfolger von x, so wird x Wurzel des neuen Suchbaums. Der linke Teilbaum von r wird zum linken Teilbaum von x. Dann wird r rechter Nachfolger von x.
Ist die Wurzel r der Vorgänger von x, so erfolgt das Einfügen analog.

Delete (x)**.** Um ein Element x aus dem Splay-Baum zu löschen wird dieses zunächst mit der Splay Operation an die Wurzel geholt. Dann wird es gelöscht und die beiden dabei entstehenden Bäume werden mit Hilfe von Join wieder zusammengefügt.

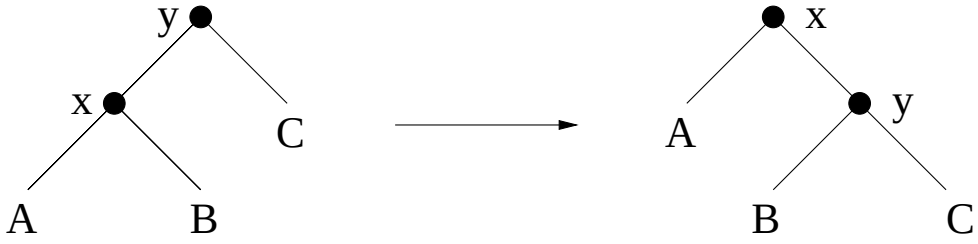
Split (x)**.** Auch bei der Operation Split wird zunächst ein Splay (x) durchgeführt. Danach sind alle Elemente im linken Unterbaum der Wurzel kleiner als x und alle Elemente im rechten Unterbaum gröser. Je nachdem, ob die Wurzel kleiner (oder gleich) oder größer als x ist, bleibt sie Wurzel des linken bzw. rechten Teilbaums.

Join (T_1, T_2)**.** Die Join Operation vereinigt zwei Bäume T_1 und T_2 unter der Voraussetzung, dass die Elemente in T_1 alle kleiner sind als die Elemente in T_2. Bei einem Join wird zunächst ein Splay (∞) in Baum T_1 durchgeführt. Damit wird das größte Element aus Baum T_1 an die Wurzel rotiert und der rechte Teilbaum unter der Wurzel ist leer. Dort wird nun Baum T_2 angehängt.

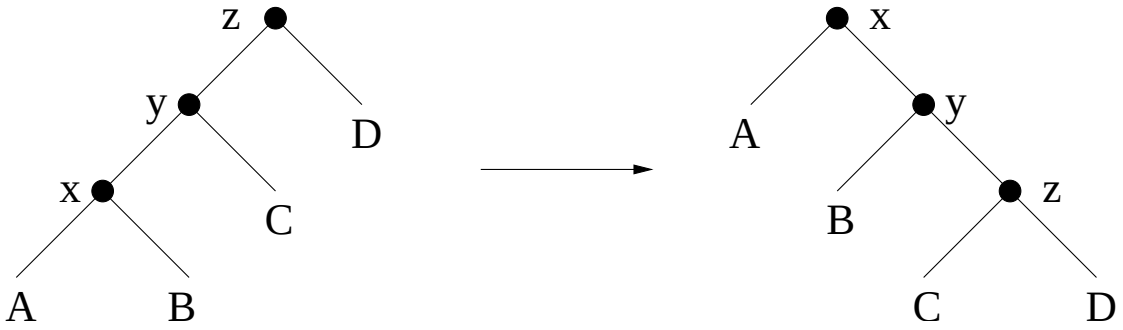
Zusammenfassung. Wir können also jede Operation mit Hilfe einer SPLAY Operation und konstant viel zusätzlichem Aufwand durchführen. Daher interessiert uns im folgenden in erster Linie die Implementierung der SPLAY Operation.

5.1.2 Die SPLAY Operation

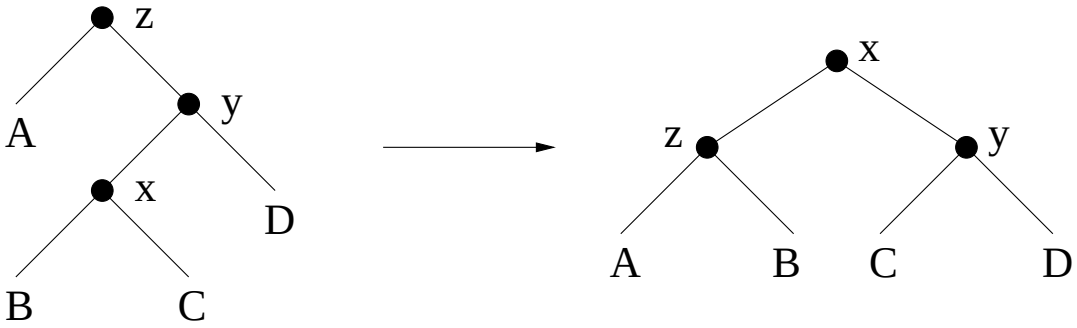
Fall 1:



Fall 2: Wir wenden zuerst eine Rechtsrotation auf z und anschließend auf y an.



Fall 3:



5.1.3 Die Analyse der SPLAY Operation.

Wir wollen nun die amortisierten Kosten von SPLAY (x) analysieren. Dazu führen wir eine Potentialfunktion Φ ein, die uns das Potential eines Splay-Baums T angibt. Wir nehmen an, dass jeder Knoten ein positives Gewicht $w(x)$ besitzt (zunächst können wir $w(x) = 1$ annehmen; später werden wir jedoch auch andere Gewichte benötigen). Wir definieren uns:

$$\begin{aligned} W(x) &= \sum_{y \text{ im Teilbaum von } x} w(y) \\ r(x) &= \log_2 W(x) \\ \Phi(T) &= \sum_{x \text{ ist Knoten in } T} r(x) \end{aligned}$$

Wir nennen $r(x)$ auch den *Rang* von x. Wir erinnern uns, dass die tatsächlichen Kosten einer Sequenz σ von m Operation höchstens die amortisierten Kosten von σ plus das Anfangspotential der Datenstruktur sind, d.h.:

$$\sum_{i=1}^{m} t_i \leq \sum_{i=1}^{m} a_i + \Phi_1 \ ,$$

wobei t_i, a_i die tatsächlichen bzw. amortisierten Kosten von Operation i sind und Φ_1 das Anfangspotential bezeichnet. Als erstes wollen wir nun die amortisierten Kosten einer SPLAY -Operation analysieren.

Lemma 5.1.1 *Die amortisierten Kosten von* SPLAY (x) *sind höchstnes* $3(r(t)-r(x))+1 = O(1+\log\frac{W(t)}{w(x)})$*, wobei* t *die Wurzel von* T *ist.*

Beweis: Um Lemma 5.1.1 zu zeigen, teilen wir eine SPLAY -Operation in eine Sequenz von Schritten auf. Dabei ist jeder Schritt gerade einer der drei oben beschriebenen Fälle, die bei einer SPLAY -Operation auftreten können. Wir betrachten nun per Fallunterscheidung die amortisierten Kosten eines Schrittes. Dabei bezeichne $r(x)$ den Rang von x vor dem Schritt und $r'(x)$ den Rang von x nach dem Schritt.

Behauptung 5.1.2 *Die amortisierten Kosten im Fall 1 sind höchstens* $1+3(r'(x)-r(x))$.

Beweis: Hier sind die amortisierten Kosten A

$$A = 1 + r'(x) + r'(y) - r(x) - r(y) \ ,$$

weil nur eine Rotation nötig ist. Aus $'r(x) = r(y)$ folgt

$$= 1 + r'(y) - r(x) \ .$$

Wegen $r'(y) \leq r'(x)$ folgt

$$\leq 1 + r'(x) - r(x) \ .$$

Aus $r(x) \leq r'(x)$ folgt dann

$$\leq 1 + 3(r'(x) - r(x))$$

Behauptung 5.1.3 *Die amortisierten Kosten im Fall 2 sind höchstens* $3(r'(x)-r(x))$.

Beweis: In diesem Fall sind die amortisierten Kosten A

$$A = 2 + r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z) \ ,$$

da zwei Rotationen benötigt werden. Nun folgt wegen $r'(x) = r(z)$

$$= 2 + r'(y) + r'(z) - r(x) - r(y) \ .$$

Dann ergibt sich wegen $r'(x) \geq r'(y)$ und $-r(x) \geq -r(y)$

$$\leq 2 + r'(x) + r'(z) - 2r(x) \ .$$

Wir wollen nun zeigen, dass

$$2 + r'(x) + r'(z) - 2r(x) \leq 3(r'(x) - r(x))$$

gilt. Dies ist äquivalent zu

$$\Leftrightarrow r'(z) + r(x) - 2r'(x) \leq -2 \ .$$

Wir setzen nun $r'(z) = \log_2 W'(z)$ ein und erhalten:

$$\Leftrightarrow \log_2 W'(z) + \log_2 W(x) - 2\log_2 W'(x) \leq -2$$

Nun folgt nach den Logarithmengesetzen ($\log a - \log b = \log(a/b)$):

$$\Leftrightarrow \log_2 \left(\frac{W'(z)}{W'(x)}\right) + \log_2 \left(\frac{W(x)}{W'(x)} \leq -2 \ .\right.$$

Wir können nun die Korrektheit der Ungleichung verifizieren. Links steht ein Ausdruck der Form $\log a + \log b$ mit $=\leq a, b \leq 1$ und $a + b \leq 1$, da $W'(z) + W(x) \leq W'(x)$. Der Term $\log a + \log b$ nimmt sein Maximum für $a = b = 1/2$ an.

Behauptung 5.1.4 *Die amortisierten Kosten im Fall 3 sind höchstens* $2(r'(x) - r(x))$.

Beweis: Siehe Übung.
Nun beobachten wir noch, dass Fall 1 bei einer SPLAY -Operation nur ein einziges mal auftritt. Damit folgt aus Behauptung 5.1.2 bis 5.1.4 nach der Teleskopsummenregel (und weil $2(r'(x)-r(x)) \leq 3r'(x)-r(x)$ gilt), dass die amortisierten einer SPLAY -Operation höchstens $3(r(t) - r(x)) + 1$ sind. Damit folgt das Lemma, denn $3(r(t) - r(x)) + 1 = O(1 + \log \frac{W(t)}{w(x)})$ gilt nach den Logarithmengesetzen.
Wir können nun folgenden Satz zeigen:

Satz 5.1.5 *Eine beliebige Folge σ von m Operationen auf einem Splay-Baum, der dabei maximal Größe n erreicht, wird in Zeit $O((m + n) \log n)$ abgearbeitet.*

Beweis: Sei T der Suchbaum vor der ersten Operation. Wir wissen, dass T höchstens n Knoten hat. Wir setzen für jeden Knoten x sein Gewicht $w(x) = 1$. Damit folgt sofort, dass $\Phi(T) = O(n \log n)$ gilt. Nach Lemma 5.1.1 gilt, dass die amortisierten Kosten einer SPLAY -Operation $O(1 + \log \frac{W(t)}{w(x)}) = O(\log n)$ sind. Damit folgt Satz 5.1.5 sofort, denn die tatsächlichen Kosten sind damit höchstens $O((m + n) \log n)$.

5.1.4 Statische Optimalität von Splay-Bäumen

Wir werden nun zeigen, dass Splay-Bäume auf einer genügend langen Anfragesequenz aus Zugriffsoperationen genausogut sind, wie ein beliebiger statischer Suchbaum (der Suchbaum wird also während der Anfragen nicht verändert). Wir könnten also auch den statischen Suchbaum nehmen, der optimal für unsere Folge von Anfragen ist und wir wären asymptotisch nicht besser als ein Splay-Baum.
Sei nun S eine Menge von n Elementen und σ eine feste Folge von Zugriffsoperationen auf Elemente von S. B sei ein beliebiger binärer Suchbaum für die Elemente aus S (B kann optimal für die Folge σ gewählt sein). Weiterhin sei k die Anzahl Vergleiche, die benötigt werden, um σ auf b abzuarbeiten. Dann gilt:

Satz 5.1.6 *Sei T ein beliebiger Suchbaum für S. Dann wird σ mit Hilfe der* SPLAY *-Methode mit Startbaum T in $O(k + n^2)$ Zeit abgearbeitet.*

Beweis: Sei t die Wurzel von T. Wir bezeichnen mit d die Tiefe von B und mit $d(x)$ die Tiefe von x in B (die Länge des Pfades von der Wurzel zu x). Wir werden nun als Gewichtsfunktion

$$w(x) = 3^{d-d(x)}$$

wählen und diese für unsere Potentialfunktion aus Lemma 5.1.1 verwenden. Wir zeigen zunächst:

Behauptung 5.1.7 *Das Gewicht der Wurzel t von T ist höchstens 3^{d+1}.*

Beweis: Wir wissen, dass B die Tiefe d hat, also hat jedes Element Tiefe höchstens d. Das Gewicht der Wurzel ist maximal, wenn der Baum groß ist, d.h. es wird am grössten, wenn B ein vollständiger Binärbaum ist. Damit ergibt sich

$$W(t) = \sum_{x \in S} w(x) = \sum_{0 \le i \le d} 2^i \cdot 3^{d-i} \le 3^d \sum_{0 \le i \le d} \left(\frac{2}{3}\right)^i \le 3^{d+1} \ .$$

Da das Gewicht jedes anderen Knotens kleiner ist, als das der Wurzel gilt:

Korollar 5.1.8 *Das Gewicht eines Knotens x aus T ist höchstens 3^{d+1}.*

Aus Lemma 5.1.1 folgt dann für die amortisierten Kosten A_x von SPLAY (x):

$$A_x = O(1 + \log \frac{W(t)}{w(x)} = O(\log(3^{1+d(x)})) = O(d(x)) \ .$$

Für das Anfangspotential $\Phi(T)$ gilt nach Korollar 5.1.8:

$$\Phi(T) = \sum_{x \in S} \log W(x) \le \sum_{x \in S} O(d) = O(nd) = O(n^2) \ .$$

Da amortisierte Kosten plus Anfangspotential eine obere Schranke für die tatsächlichen Kosten bilden, folgt Satz 5.1.6.

5.2 Randomisierte Suchbäume

In diesem Kapitel werden wir randomisierte Suchbäume kennen lernen. Zuerst werden wir uns mit dem randomisiertem Quicksort-Algorithmus beschäftigen und dabei eine grundlegende Technik zur Analyse von randomisierten Algorithmen kennenlernen, die die *Linearität des Erwartungswertes* von Zufallsvariablen ausnutzt. Damit werden wir die erwartete Laufzeit des randomisierten Quicksort-Algorithmus und ausschließlich die erwartete Zugriffszeit auf ein Elemenentin unserem randomisierten Suchbaum analysieren.

5.2.1 Der randomisierte Quicksort Algorithmus

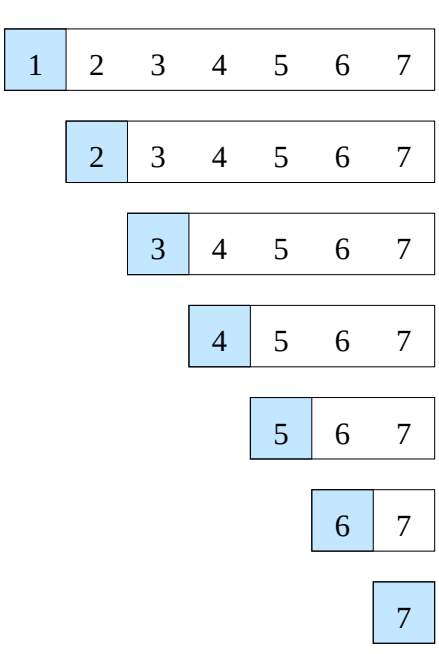
Als erstes werden wir nun den randomisierten Quicksort Algorithmus kennenlernen. Aufgabe dieses Algorithmus ist es, eine Menge S von n *unterschiedlichen* Zahlen in sortierter Reihenfolge auszugeben. Man kann den Algorithmus natürlich auch so modifizieren, dass eine sortierte Liste der Zahlen zurückgegeben wird. Ebenso kann man den Algorithmus erweitern, um auch Mengen von Zahlen zu sortieren, die dieselbe Zahl mehrmals enthalten dürfen. Wir werden jedoch auf diese Erweiterungen in unserer Darstellung verzichten, um die Analyse einfach zu halten.

RandQS(S)
1. Wähle $y \in S$ zufällig und gleichverteilt
2. $S_1 = \{x \in S | x < y\}$; $S_2 = \{x \in S | x > y\}$
3. **if** $S_1 \neq \emptyset$ **then** RandQS(S_1)
4. **output** y
5. **if** $S_2 \neq \emptyset$ **then** RandQS(S_2)

Wir werden nun zunächst einen bekannten Worst Case des Quicksort Algorithmus betrachten und bestimmen, mit welcher Wahrscheinlichkeit genau dieser Fall beim randomisierten Quicksort Algorithmus auftreten kann. Die Laufzeit des Quicksort Algorithmus messen wir durch die Anzahl der Vergleiche, die der Algorithmus benötigt. Diese Vergleiche werden für die Berechnung der Mengen S_1 und S_2 benötigt und dominieren die Laufzeit. Da jedes Paar von Elementen höchstens einmal miteinander verglichen wird, liefert die Anzahl Vergleiche eine Abschätzung der Laufzeit bis auf konstante Faktoren.

Ein Worst Case und seine Wahrscheinlichkeit

Wir nehmen an, dass die Eingabemenge S bereits in einem sortierten Feld vorliegt. Wir betrachten den Fall, dass immer das erste Element als Pivot-Element ausgewählt wird. Dies wird z.B. in vielen Implementierungen des deterministischen Quicksort Algorithmus gemacht. Wir betrachten zunächst ein Beispiel mit $S = \{1, \ldots, 7\}$.



Anzahl Vegleiche	Wahrscheinlichkeit
6	1/7
5	1/6
4	1/5
3	1/4
2	1/3
1	1/2
0	1
21	1/ (7 !)

Abbildung 1: Eine mögliche Ausführung des randomisierten Quicksort Algorithmus und die zugehörige Wahrscheinlichkeit. Die Pivot-Elemente sind unterlegt.

Wie man leicht sieht, ist die Anzahl Vergleiche für die Eingabe $1, \ldots, n$

$$\sum_{i=1}^{n-1} i = \frac{n \cdot (n-1)}{2} = \Theta(n^2) \ ,$$

wenn als Pivot-Element immer das erste Element gewählt wird. Die Wahrscheinlichkeit, dass dieser Fall beim randomisierten Quicksort Algorithmus eintritt, ist allerdings nur

$$\prod_{i=1}^{n} \frac{1}{i} = \frac{1}{n!} \ .$$

Dieser Fall tritt also mit sehr geringer Wahrscheinlichkeit auf. Andererseits gibt es natürlich auch sehr viele andere Möglichkeiten, dass unser Algorithmus eine relativ schlechte Auswahl für die Pivot-Elemente trifft. Daher wollen wir die *erwartete Laufzeit* unseres Algorithmus berechnen. Die erwartete Laufzeit gibt an, wie lange der Algorithmus 'durchschnittlich' für die *schlechteste* Eingabe rechnet. Der Durchschnitt wird dabei über die verschiedenen Ausgänge der Zufallsexperimente, die der Algorithmus ausführt, gebildet und mit der zugehörigen Wahrscheinlichkeit gewichtet.

Die erwartete Laufzeit des randomisierten Quicksort Algorithmus

Wir werden nun die *erwartete Anzahl* von Vergleichen bestimmen, die der randomisierte Quicksort Algorithmus benötigt. Dazu bezeichnen wir unsere Eingabemenge $S = \{s_1, \ldots, s_n\}$ und nehmen an, dass $s_i < s_j$ für $i < j$ gilt (d.h. s_i ist das i-kleinste Eingabeelement). Wir nehmen *nicht* an, dass die Elemente in der Eingabe in dieser Reihenfolge erscheinen. Wir führen nun die $0-1$ Zufallsvariablen $X_{i,j}$ für $1 \leq i < j \leq n$ ein. $X_{i,j}$ ist 1, wenn der randomisierte Quicksort s_i mit s_j vergleicht und 0 sonst. Der Wert von $X_{i,j}$ hängt also vom Ausgang des Zufallsexperimentes ab (daher auch der Name Zufallsvariable) und ist 1, wenn ein bestimmtes Ereignis eintritt (wenn s_i mit s_j verglichen wird). Daher nennen wir eine solche Zufallsvariable auch eine *Indikatorzufallsvariable* (für das Ereignis 'Quicksort vergleicht s_i mit s_j').
Damit können wir die Gesamtanzahl Vergleiche als

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{i,j}$$

schreiben. Die erwartete Laufzeit ist dann wegen der Linearität des Erwartungswertes

$$E\big[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{i,j}\big] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E\big[X_{i,j}\big] \ .$$

Linearität des Erwartungswertes sagt also, dass man den Erwartungswert der Summe von Zufallsvariables als die Summe der einzelnen Erwartungswerte schreiben kann. Als nächstes definieren wir

$$p_{i,j} = \mathrm{Prob}\big[X_{i,j} = 1\big] \ .$$

Nach der Definition des Erwartungswertes folgt dann

$$E\big[X_{i,j}\big] = 1 \cdot p_{i,j} + 0 \cdot (1 - p_{i,j}) = p_{i,j} \ .$$

Der Erwartungswert von $X_{i,j}$ ist also gerade die Wahrscheinlichkeit, dass $X_{i,j}$ den Wert 1 annimmt. Dies gilt natürlich für jede Indikatorzufallsvariable. Wir müssen nun also nur noch den Wert $p_{i,j}$ bestimmen, um die erwartete Laufzeit des randomisierten Quicksort Algorithmus zu bekommen.
Dazu betrachten wir den *Rekursionsbaum* T des Algorithmus. Jeder Knoten dieses Baumes entspricht einem Aufruf des Quicksort Algorithmus. Die Wurzel repräsentiert den ersten Aufruf RandQS(S). Die beiden Kinder der Wurzel entsprechen den Aufrufen RandQS(S_1)

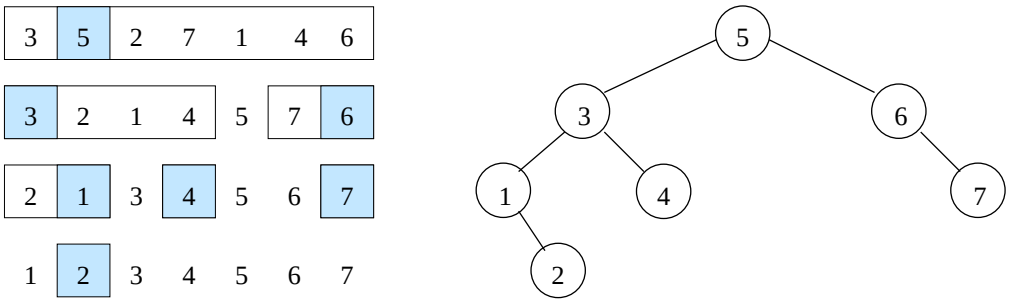


Abbildung 2: Ein Beispiel für den Verlauf des Quicksort Algorithmus mit zugehörigem Rekursionsbaum.

und RandQS(S_2). Wir führen diese Konstruktion rekursiv weiter und beschriften jeden Knoten von T mit dem zugehörigen Pivot-Element.
Nun bezeichne π die Permutation die durch die schichtweise Traversierung des Rekursionsbaums bestimmt ist (in unserem Beispiel ist $\pi = 5\ 3\ 6\ 1\ 4\ 7\ 2$). Wir werden nun einen strukturellen Zusammenhang zwischen Rekursionsbaum und dem Vergleich von Elementen zeigen.

Lemma 5.2.1 *Es gibt einen Vergleich zwischen s_i und s_j $(i < j)$ genau dann, wenn entweder s_i oder s_j eher in π auftreten als jedes andere Element aus der Menge $S_{i,j} = \{s_i, s_{i+1}, \ldots, s_j\}$.*

Beweis: Sei s^* das Element aus $S_{i,j}$, das als erstes in π auftritt. Dann ist $S_{i,j}$ komplett im Unterbaum mit Wurzel s^*, weil kein vorher auftretendes Pivot-Element in π die Menge $S_{i,j}$ aufteilt.

Fall 1. $s^* \in \{s_i, s_j\}$. Wenn $s^* = s_i$ ist, dann werden alle Elemente aus $S_{i,j}$ mit s_i verglichen, insbesondere auch s_j. Der Fall $s^* = s_j$ ist analog.

Fall 2. $s^* \notin \{s_i, s_j\}$. Dann werden s_i und s_j mit s^* verglichen. Dabei wird s_i in S_1 eingefügt und s_j in S_2. Damit erscheint s_i im linken Unterbaum von s^* und s_j im rechten und sie werden im weiteren Verlauf des Algorithmus nicht miteinander verglichen.

Nun wissen wir, dass zwei Elemente s_i und s_j genau dann miteinander verglichen werden, wenn eines der beiden Elemente als erstes aus $S_{i,j}$ in π auftritt. Das heisst wir müssen nur noch die Wahrscheinlichkeit dafür bestimmen, dass dies tatsächlich der Fall ist.

Lemma 5.2.2 *Die Wahrscheinlichkeit, dass entweder s_i oder s_j das Element aus $S_{i,j}$ ist, das als erstes in π auftaucht, ist genau $\frac{2}{j-i+1}$.*

Beweis: Wir betrachten den Verlauf des Quicksort Algorithmus bis zum ersten Mal ein Element aus $S_{i,j}$ als Pivot-Element gewählt wird. Bei jeder Wahl eines Pivot-Elements sind alle Elemente aus $S_{i,j}$ gleichwahrscheinlich, weil das Pivot-Element zufällig und gleichverteilt aus allen möglichen Elementen gewählt wird. Also ist jedes Element aus $S_{i,j}$ mit derselben Wahrscheinlichkeit das erste Element in π. Damit ist jedes Elemente aus $S_{i,j}$ mit Wahrscheinlichkeit

$$\frac{1}{|S_{i,j}|} = \frac{1}{j-i+1}$$

das erste Element in π.

Also ist die Wahrscheinlichkeit, dass s_i *oder* s_j das erste Element ist, $\frac{2}{j-i+1}$ und es folgt $p_{i,j} = \frac{2}{j-i+1}$. Dies brauchen wir nur noch einzusetzen, um die erwartete Anzahl Vergleiche im randomisierten Quicksort Algorithmus zu bestimmen.

$$\begin{aligned}
\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} p_{i,j} &= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1}\sum_{j=2}^{n-i+1}\frac{2}{j} \\
&= \sum_{i=2}^{n}\sum_{j=1}^{n-i+2}\frac{2}{j} \\
&= \sum_{i=2}^{n}\sum_{j=2}^{i}\frac{2}{j} \\
&= 2\sum_{i=1}^{n}(H_i - 1)
\end{aligned}$$

Dabei ist $H_i = \sum_{k=1}^{i}\frac{1}{k}$ die i-te harmonische Zahl. Aus $\ln i \leq H_i \leq \ln i + 1$ folgt

Satz 5.2.3 *Die erwartete Anzahl von Vergleichen in einer Ausführung des randomisierten Quicksort Algorithmus auf einer n-elementigen Eingabe ist*

$$2\sum_{i=1}^{n}(H_i - 1) = \Theta(n \log n) \ .$$

Bemerkung 5.2.4 *Jeder vergleichsbasierte Sortieralgorithmus benötigt $\Omega(n \log n)$ Laufzeit.*

Analysetechnik. Zur Analyse des Algorithmus haben wir die Laufzeit als Summe von Indikatorzufallsvariablen dargestellt. Dann haben wir ausgenutzt, dass sich aufgrund der Linearität des Erwartungswertes die erwartete Laufzeit als Summe der Erwartungswerte dieser Indikatorzufallsvariablen darstellen lässt. Da der Erwartungswert einer Indikatorzufallsvariable die Wahrscheinlichkeit ist, dass diese Zufallsvariable den Wert 1 annimmt (das zugehörige Ereignis eintritt), brauchten wir nur noch diese Wahrscheinlichkeit zu bestimmen und in die Formel für die Laufzeit einsetzen. Die Darstellung von komplexen Zufallsvariablen als Summe von Indikatorvariablen ist eine häufig verwendete Technik zur Analyse randomisierter Algorithmen.

5.2.2 Randomisierte Suchbäume

In diesem Kapitel werden wir *randomisierte Suchbäume* kennenlernen.

Definition 5.2.5 *Ein* Suchbaum *ist ein (binärer) Baum, der die* Suchbaumeigenschaft *erfüllt, d.h. für den gilt:*
Für jeden Knoten v mit Schlüssel $s(v)$ gilt:
Jeder Knoten im linken Teilbaum hat Schlüssel $\leq s(v)$ und jeder Knoten im rechten Teilbaum hat Schlüssel $> s(v)$.

Zufällig aufgebaute Suchbäume

Zunächst einmal wollen wir jedoch analysieren, wie gut Suchbäume sind bei denen eine Menge S von Elementen in zufälliger Reihenfolge eingefügt wird. Sei dazu wieder s_i das i-kleinste Element der Eingabe $s_1, \ldots, s_n$. Außerdem sei π eine zufällig und gleichverteilt gewählte Permutation von $\{1, \ldots, n\}$. Wir bauen abhängig von π den Baum T wie folgt auf:

1. Sei π eine zufällig und gleichverteil gewählte Permutation von $\{1, \ldots, n\}$
2. T = leerer Suchbaum
3. **for** $i = 1$ **to** n **do**
4. Insert($s_{\pi(i)}$, T)

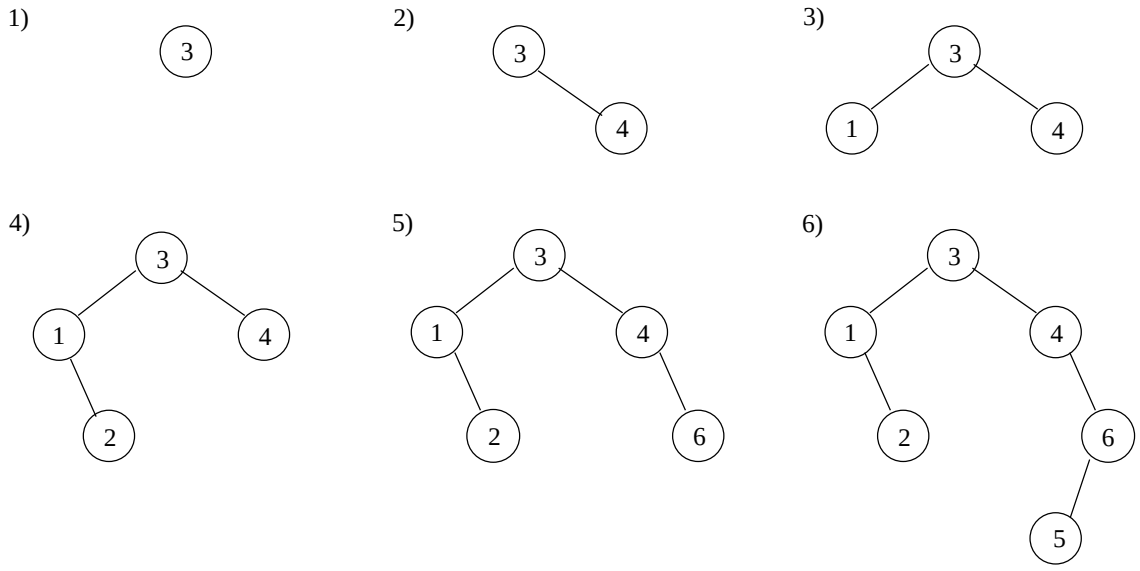


Abbildung 3: Einfügen der Elemente 1,2,3,4,5,6 in der Reihenfolge 3,4,1,2,6,5 in einen Suchbaum.

Als nächstes wollen wir die erwartete Zugriffszeit auf ein festes Element s_i bestimmen. Dazu sei D_i die Zufallsvariable für die Länge des Weges von s_i bis zur Wurzel von T, also die Anzahl der Vorgänger von Knoten s_i im Baum T. Wir definieren eine Indikatorzufallsvariable $X_{i,j}$ für das Ereignis Knoten s_j ist Vorgänger von Knoten s_i in T. Damit gilt also $X_{i,j} = 1$, wenn Knoten s_j Vorgänger von Knoten s_i in T ist und 0 sonst. Dann gilt

$$D_i = \sum_{1 \le j \le n} X_{i,j}$$

und

$$\begin{aligned} E\big[D_i\big] &= E\big[\sum_{1 \le j \le n} X_{i,j}\big] = \sum_{1 \le j \le n} E\big[X_{i,j}\big] = \sum_{1 \le j \le n} \text{Prob}\big[X_{i,j} = 1\big] \\ &= \sum_{1 \le j \le n} \text{Prob}\big[s_j \text{ ist Vorgänger von } s_i\big] \end{aligned}$$

Allgemein können wir für eine geordnete Menge mit Prioritäten einen eindeutig definierten Suchbaum wie folgt definieren:

Definition 5.2.6 *Sei $\{s_1, \ldots, s_n\}$ eine Menge mit $s_k < s_\ell$ für $k < \ell$. Jedes Element aus S hat eine Priorität $prio(s_\ell)$. Ein* Treap *(Tree + Heap) für S ist ein binärer Baum T für den gilt:*

T ist ein Suchbaum bzgl. S.

T ist ein Heap bzgl. der Prioritäten, d.h. für jeden Knoten außer der Wurzel ist die Priorität seiner Vorgänger kleiner als die eigene.

Als nächstes entwickeln wir ein Lemma, dass den Zusammenhang zwischen Prioritäten und Baumstruktur in einem Treap charakterisiert.

Lemma 5.2.7 *Sei $S = \{s_1, \ldots, s_n\}$ eine Menge von Elementen mit $s_k < s_\ell$ für $k < \ell$ und Prioritäten $prio(s_\ell)$. Dann ist in Treap T für S das Element s_j genau dann Vorgänger von s_i, wenn gilt:*

$$prio(s_j) = \begin{cases} \min\{prio(s_i), prio(s_{i+1}), \ldots, prio(s_j)\} & \text{falls } i < j \\ \min\{prio(s_j), prio(s_{j+1}), \ldots, prio(s_i)\} & \text{falls } i > j \end{cases}$$

Dabei bezeichnet $\min prio(S_{i,j})$ *die kleinste Priorität der Elemente aus $S_{i,j} = \{s_i, \ldots, s_j\}$.*

Beweis: Beweis in der Übung.
Wir betrachten nun Treaps, deren Prioritäten durch zufällige Permutationen gegeben sind.

Lemma 5.2.8 *Sei π zufällig und gleichverteilt gewählt aus der Gruppe Σ_n der Permutationen von n Elementen und sei $prio(s_\ell) = \pi(\ell)$. Dann gilt:*

$$Prob\big[s_j \text{ ist Vorgänger von } s_i\big] = \frac{1}{|j-i|+1} \ .$$

Beweis: Da π zufällig und gleichverteilt aus Σ_n gewählt wird, nimmt $\pi(\ell)$ jeden Wert aus $\{1, \ldots, n\}$ mit derselber Wahrscheinlichkeit an. Damit ist die Wahrscheinlichkeit, dass für festes i die Priorität $\pi(i)$ die kleinste Priorität der Elemente aus $S_{i,j}$ ist, genau

$$\frac{1}{|S_{i,j}|} = \frac{1}{|j-i|+1} \ .$$

Damit gilt:

$$\begin{aligned} E\big[D_i\big] &= \sum_{1\le j\le n} E\big[X_{i,j}\big] = \sum_{1\le j\le n} \mathrm{Prob}\big[X_{i,j}=1\big] \\ &= \sum_{1\le j\le n} \mathrm{Prob}\big[s_j \text{ ist Vorgänger von } s_i\big] \\ &= \frac{1}{i} + \cdots + \frac{1}{2} + 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n-i+1} \\ &= H_i + H_{n+1-i} - 1 \\ &\le 1 + 2\ln n \end{aligned}$$

Damit haben wir gezeigt:

Satz 5.2.9 *Wenn eine Menge $S = \{s_1, \ldots, s_n\}$ von n Elementen in zufälliger Reihenfolge in einen anfangs leeren binären Suchbaum eingefügt wird, dann ist die erwartete Tiefe eines festen Knotens s_ℓ höchstens $1 + 2\ln n$.*

Randomisierte Suchbäume

Bisher haben wir gesehen, dass jedes Element in einem Suchbaum erwartete Tiefe $O(\log n)$ hat, wenn die Elemente in zufälliger Reihenfolge in den Suchbaum eingefügt worden sind. Wir wollen nun effiziente Suchbäume entwickeln, die für beliebige Einfügereihenfolgen erwartete Tiefe $O(\log n)$ garantieren. Diese Suchbäume bezeichnen wir dann auch als *randomisierte Suchbäume*.
Wir wollen dabei unser Wissen über zufällig aufgebaute Suchbäume nutzen. Unser Ziel ist, einen Suchbaum zu entwickeln, in dem die Elemente angeordnet sind, als ob sie in zufälliger Reihenfolge eingefügt worden wären.

Die Idee. Wir würfeln für jedes Element beim Einfügen eine zufällige Priorität aus dem Intervall $[0, 1]$. Der Baum wird so aufgebaut, als wären die Elemente in der Reihenfolge ihrer Prioritäten eingefügt worden.

Definition 5.2.10 *Ein* randomisierter Suchbaum *für* $\{s_1, \ldots, s_n\}$ *ist ein Treap für* $\{s_1, \ldots, s_n\}$*, bei dem für jedes* s_i *die Priorität zufällig und gleichverteilt aus* $[0, 1]$ *gewählt wurde.*

Wir können nun ähnlich wie bei zufällig aufgebauten Suchbäumen folgendes Lemma zeigen:

Lemma 5.2.11 *Sei* $S = \{s_1, \ldots, s_n\}$ *eine Menge von Elementen und seien* $prio(s_1), \ldots, prio(s_n)$ *unabhängige und gleichverteile Zufallsvariablen aus* $[0, 1]$*. Dann gilt*

$$Prob\big[prio(s_j) = \min prio(S_{i,j})\big] = \frac{1}{|j - i| + 1} \ .$$

Beweis: Die Wahrscheinlichkeit, dass zwei Prioritäten identisch sind, ist 0. Da jede Priorität zufällig, gleichverteilt und unabhängig aus derselben Menge gewählt wurde, ist damit aus Symmetriegründen jedes Element mit derselben Wahrscheinlichkeit das Element mit der kleinsten Priorität. Daher ist die Wahrscheinlichkeit, das ein Element das kleinste einer k-elementigen Menge ist, genau $1/k$ und das Lemma folgt.
Es folgt

Satz 5.2.12 *Sei* $S = \{s_1, \ldots, s_n\}$ *eine Menge von Elementen, die in einem randomisierten Suchbaum gespeichert sind. Dann ist die erwartete Tiefe des Knotens für* s_i

$$H_i + H_{n+1-i} - 1 \leq 1 + 2 \ln n \ .$$

Einfügen in randomisierten Suchbäumen Wir müssen nun natürlich noch zeigen, dass man effizient Elemente in einen randomisierten Suchbaum einfügen kann. Dabei funktioniert das Einfügen wie in einem normalen Suchbaum. Wir merken uns zusätzlich beim Einfügen den Suchpfad (dies geschieht implizit durch Rekursion) und stellen nach dem Einfügen durch Rotationen entlang des Suchpfads die Heapeigenschaft bzgl. Prioritäten wieder her.
Wir geben nun Pseudocode für das Einfügen eines Elementes s in einen Treap. Dabei muss v im ersten Aufruf von TreapInsert die Wurzel des Treaps sein.

```
TreapInsert(s, prio(s), v)
  if v = NULL then v = new node(s, prio(s))
  else
    if s < v then
      TreapInsert(s, prio(s), leftchild(v))
      if prio(leftchild(v)) < prio(v) then RotateRight(v)
    else
```

if $s > v$ **then**
    TreapInsert$(s, prio(s), rightchild(v))$
    if $prio(rightchild(v)) < prio(v)$ **then** RotateLeft(v)
else (s ist schon im Baum enthalten)

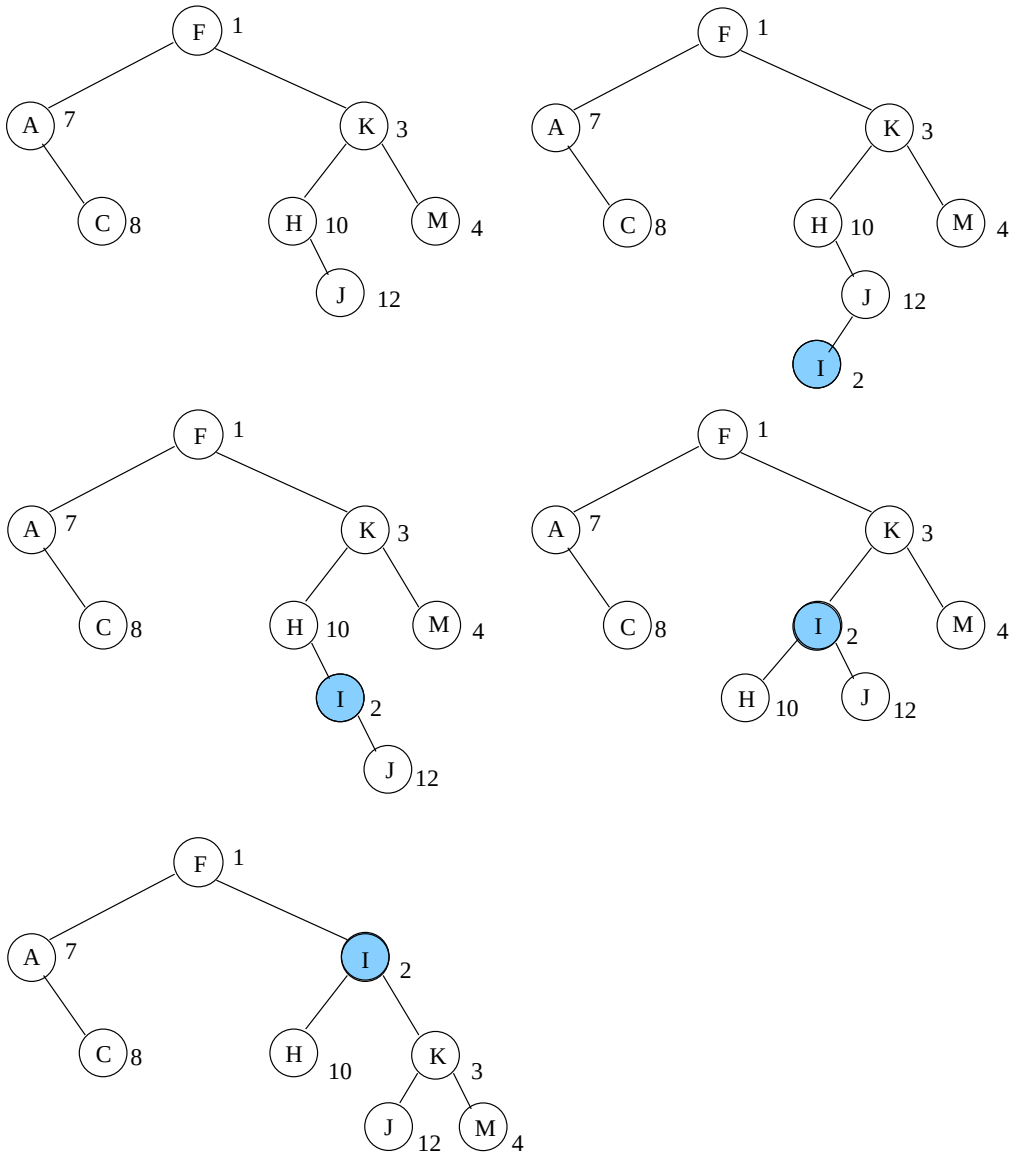


Abbildung 4: Ein Treap für die Elemente $\{A, C, F, H, J, K, M\}$ geordnet nach dem Alphabet. Einfügen eines Elements I in einen Treap.

Bemerkung 5.2.13 *Löschen kann ebenfalls in $\mathcal{O}(\log n)$ erwarteter Zeit implementiert werden.*

Beweis: Übung.

Satz 5.2.14 *In einem randomisierten Suchbaum für n Elemente können die Operationen Suchen, Einfügen und Löschen in $O(\log n)$ erwarteter Zeit durchgeführt werden.*

6 Minimale Schnitte in Graphen

In diesem Kapitel werden wir einen Algorithmus kennenlernen, der randomisiert arbeitet und nicht immer, sondern nur mit hoher Wahrscheinlichkeit, ein korrektes Ergebnis berechnet. Dafür können wir für diesen Algorithmus eine Laufzeitschranke angeben, die nicht vom Zufall abhängt, Solche Algorithmen nennt man auch Monte Carlo Algorithmen. Wir werden nun einen Monte Carlo Algorithmus kennenlernen, der einen *minimalen Schnitt* in einem Graph mit ganzzahligen Kantengewichten berechnet. Zur Vereinfachung der Präsentation des Algorithmus werden wir den Graph jedoch als Multigraph darstellen, d.h. eine Kante mit Gewicht k wird durch k einzelne Kanten dargestellt. Wir nehmen weiterhin an, dass der Graph keine Selbstloops hat. Ein Schnitt in einem Graph ist definiert wie folgt:

Definition 6.0.15 *Ein Schnitt in einem zusammenhängenden Multigraph ist eine Menge von Kanten C, so dass der Multigraph nach dem Entfernen von C nicht mehr zusammenhängend ist.*

Da wir gewichtete Graphen durch Multigraphen darstellen, ist der Wert eines Schnittes C gerade die Anzahl der Kanten $|C|$. Wir interessieren uns für minimale Schnitte. Interessanter Weise ist das Problem, maximale Schnitte zu berechnen, NP-schwer.

6.1 Ein einfacher Algorithmus

Als erstes werden wir nun einen sehr einfachen Algorithmus kennenlernen, der einen Schnitt in einem Multigraphen $G = (V, E)$ berechnet. Wir werden dann zeigen, dass der berechnete Schnitt mit Wahrscheinlichkeit $1/n^2$ ein minimaler Schnitt ist.
Der Algorithmus wählt in jedem Schritt eine zufällige Kante (u, v) aus dem aktuellen Multigraph aus. Danach wird diese Kante *kontrahiert*. Dabei werden die Knoten u und v durch einen neuen Knoten w ersetzt. Jede Kante der Form (x, u) für $x \neq v$ wird durch eine Kante (x, w) ersetzt. Jede Kante der Form (x, v) für $x \neq u$ ebenso. Kanten zwischen u und v fallen weg. Wird eine Kante kontrahiert, so liegen die beiden zugehörigen Knoten nach dem Entfernen des berechneten Schnitts in derselben Zusammenhangskomponente. Der neue Knoten repräsentiert die beiden alten Knoten.
Sind nur noch zwei Knoten übrig, so gibt der Algorithmus den Schnitt zwischen den Knotenmengen im Ursprungsgraph aus, die durch die übriggebliebenen Knoten repräsentiert werden.

CONTRACT(G)
1. H = G;
2. **while** H hat mehr als zwei Knoten **do**
3. Wähle Kante (x, y) zufällig und gleichverteilt aus den Kanten von H
4. Kontrahiere die Kante $(x; y)$ in H
5. Sei K die Menge der Kanten in H (zu diesem Zeitpunkt sind nur 2 Knoten übrig)
6. Gib die Kanten aus G aus, die K entsprechen

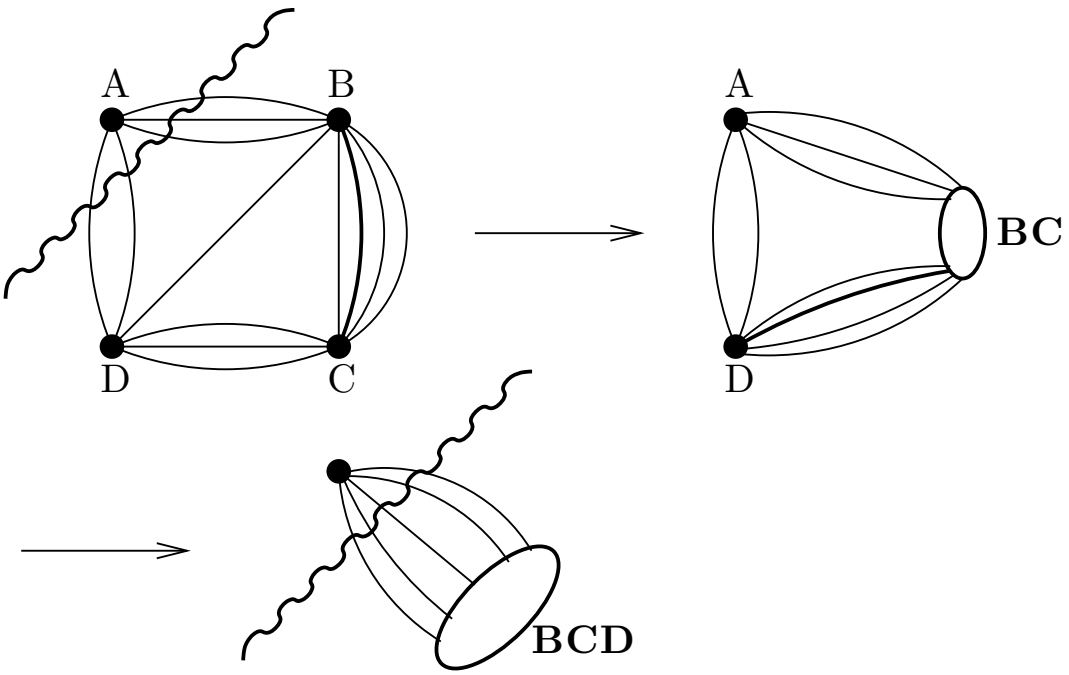


Abbildung 5: Ein beispielhafter Ablauf des Algorithmus.

6.2 Analyse des einfachen Algorithmus

Wir wollen nun zeigen, dass der von Algorithmus CONTRACT berechnete Schnitt mit Wahrscheinlichkeit mindestens $1/n^2$ ein minimaler Schnitt ist. Wir bezeichnen mit e_i die Kante, die in Iteration i des Algorithmus kontrahiert wird ($1 \leq i \leq n-2$) und mit H_i den Graph, der nach der Kontraktion von e_1 bis e_i entstanden ist. H_0 bezeichnet den Ausgangsgraph G und H_{n-2} den Graph, der am Ende des Algorithmus übrigbleibt.
Entfernt man eine Menge von Kanten C aus H_i und es entstehen dabei zwei oder mehr Zusammenhangskomponenten, so gilt dies auch, wenn man C aus $H_0 = G$ entfernt. Daher entspricht jeder Schnitt in einem der Graphen H_i einem Schnitt in G. Daraus folgt natürlich sofort, dass die Menge C, die der Algorithmus berechnet, ebenfalls einen Schnitt in G bildet. Außerdem gilt damit sofort, dass die Kardinalität eines minimalen Schnitts in H_i nicht kleiner wird als die eines minimalen Schnitts in G.

Satz 6.2.1 *Der von Algorithmus* CONTRACT *berechnete Schnitt ist mit Wahrscheinlichkeit* $1/n^2$ *ein minimaler Schnitt.*

Beweis: Sei C ein minimaler Schnitt in G und K der vom Algorithmus zurückgelieferte Schnitt. Wird im Verlauf des Algorithmus keine Kante von C kontrahiert, so ist der zurückgegebene Schnitt C. Wir werden zeigen, dass die Wahrscheinlichkeit dafür mindestens $1/n^2$ ist. Es folgt

$$\begin{aligned}
\text{Prob}\big[K \text{ ist ein minimaler Schnitt}\big] &\geq \text{Prob}\big[K = C\big] \\
&= \text{Prob}\Big[\bigwedge_{1\leq i\leq n-2} (e_i \notin C)\Big] \\
&= \prod_{1\leq i\leq n-2} \text{Prob}\big[e_i \notin C \,\big|\, \bigwedge_{1\leq j<i} (e_j \notin C)\big] \qquad (4)
\end{aligned}$$

Behauptung 6.2.2 *Für $1 \leq i \leq n-2$ gilt*

$$Prob\big[e_i \in C | \bigwedge_{1 \leq j < i} (e_j \notin C)\big] \leq \frac{2}{n-i+1} \ .$$

Beweis: Wir betrachten die Kontraktion von Kante e_i, die den Graph H_{i-1} in den Graph H_i überführt. Sei n_{i-1} die Anzahl der Knoten und m_{i-1} die Anzahl der Kanten in H_{i-1}. Da jede Kante mit derselben Wahrscheinlichkeit gewählt wird, gilt

$$\text{Prob}\big[e_i \in C | \bigwedge_{1 \leq j < i} (e_j \notin C)\big] = \frac{|C|}{m_{i-1}} \ .$$

Weiterhin gilt $m_{i-1} \geq |C| \cdot n_{i-1}/2$, weil es ansonsten einen Knoten mit Grad kleiner als $|C|$ gibt und seine inzidenten Kanten einen Schnitt mit Kardinalität $\leq |K| - 1$ bilden würden (Widerspruch zur Minimalität von C). Also folgt $\frac{|C|}{m_{i-1}} \leq \frac{2}{n_{i-1}}$. Nun folgt das Lemma aus $n_{i-1} = n - i + 1$.
Wir benutzen nun Behauptung 6.2.2, um den Beweis von Satz 6.2.1 zu vervollständigen. Dazu setzen wir das Ergebnis der Behauptung in Gleichung 4 ein und erhalten

$$\begin{aligned} \text{Prob}\big[K \text{ ist ein minimaler Schnitt}\big] &\geq \prod_{1 \leq i \leq n-2} \big(1 - \frac{2}{n-i+1}\big) \\ &= \prod_{1 \leq i \leq n-2} \frac{n-i-1}{n-i+1} \\ &= \frac{2}{n \cdot (n-1)} \geq 1/n^2 \end{aligned}$$

6.3 Ein schnellerer Algorithmus

Wir wollen nun den Algorithmus so modifizieren, dass er mit größerer Wahrscheinlichkeit einen minimalen Schnitt berechnet. Dabei wollen wir die Laufzeit möglichst wenig erhöhen. Unsere Verbesserung basiert auf folgender Beobachtung: Wenn wir den Algorithmus CONTRACT laufen lassen, bis ein Graph mit t Knoten entstanden ist, dann ist die Wahrscheinlichkeit, dass wir noch keine Kante eines festen minimalen Schnitts K genommen haben, mindestens (folgt aus dem Beweis von Satz 6.2.1)

$$\prod_{1 \leq i \leq n-t} \big(1 - \frac{2}{n-i+1}\big) = \prod_{1 \leq i \leq n-t} \frac{n-i-1}{n-i+1} = \frac{t \cdot (t-1)}{n \cdot (n-1)} \ .$$

Also ist die Wahrscheinlichkeit, dass der Algorithmus bei den ersten $n/2$ Kontraktionen keine Kante aus K nimmt, mindestens $\frac{n/2 \cdot (n/2-1)}{n(n-1)} \approx 1/4$. Obwohl unser Algorithmus nur mit Wahrscheinlichkeit $1/n^2$ einen minimalen Schnitt berechnet, nimmt er während der ersten $n/2$ Kontraktionen nur mit Wahrscheinlichkeit $1/4$ eine Kante des minimalen Schnitts. Also entsteht die hohe Fehlerwahrscheinlichkeit in erster Linie durch die letzten Kontraktionen des Algorithmus. Andererseits ist aber der Algorithmus auf kleinen Graphen viel schneller als auf großen.

Die Grundidee. Je kleiner der Graph im Verlauf des Algorithmus wird, umso mehr Durchläufe führen wir für den aktuellen Graph durch. Immer dann, wenn wir mehrere Kopien laufen lassen, wählen wir die Kopie, die den kleinsten Schnitt berechnet hat. Durch viele Kopien bei kleinen Graphen können wir die erhöhte Fehlerwahrscheinlichkeit ausgleichen.

FastCut(G)
  if $n \leq 6$ **then** berechne minimalen Schnitt 'brute force'
  else
    $t = \lceil 1 + n/\sqrt{2} \rceil$
    benutze Algorithmus Contract um unabhängig voneinander zwei Graphen H_A, H_B mit t Knoten zu berechnen
    FastCut(H_A);
    FastCut(H_B);
    gibt den kleineren der beiden berechneten Schnitte zurück

Die Laufzeit des Algorithmus $T(n)$ wird durch die Rekursion

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

beschrieben. Mit Hilfe des Master Satzs (siehe Cormen/Leiserson/Rivest) erhält man $T(n) = O(n^2 \log n)$.

Satz 6.3.1 *Algorithmus* FastCut *gibt mit Wahrscheinlichkeit* $\Omega(1/\log n)$ *einen minimalen Schnitt zurück.*

Beweis: Wir halten einen minimalen Schnitt K fest. Wir haben bereits gesehen, dass die Wahrscheinlichkeit, dass bei einer Kontraktion bis auf t Knoten keine Kante aus K gewählt wurde, mindestens

$$\frac{t \cdot (t-1)}{n \cdot (n-1)} = \frac{\lceil 1 + n/\sqrt{2} \rceil}{n} \cdot \frac{\lceil n/\sqrt{2} \rceil}{n-1} \geq 1/2$$

ist. Wir bezeichnen nun mit $P(G)$ die Wahrscheinlichkeit, dass FastCut(H) einen minimalen Schnitt berechnet. Wir wollen nun eine Rekursion für $P(G)$ aufstellen. Die Wahrscheinlichkeit, dass FastCut auf dem Rechenweg über H_A einen minimalen Schnitt berechnet, ist mindestens $1/2P(H_A)$. Gleiches gilt für den Rechenweg über H_B. Damit ist die Wahrscheinlichkeit, dass einer der beiden Wege zu einem minimalen Schnitt führt,

$$P(G) \geq 1 - \big(1 - \frac{P(H_A)}{2}\big) \cdot \big(1 - \frac{P(H_B)}{2}\big) \ . \tag{5}$$

Wir definieren nun die Höhe des Rekursionsbaums als die Anzahl Kanten auf einem kürzesten Weg zu einem Blatt, d.h. Blätter haben Tiefe 0. Sei $p(k)$ die Wahrscheinlichkeit, dass der Algorithmus FastCut einen minimalen Schnitt berechnet, wenn der Rekursionsbaum Tiefe k hat (die Tiefe des Rekursionsbaums hängt natürlich nur von der Anzahl der Knoten im Baum ab und nicht vom Verlauf des Algorithmus). Aus der Ungleichung 5 folgt

$$p(k) \geq 1 - \big(1 - \frac{p(k-1)}{2}\big)^2$$

für $k \geq 1$ und mit $p(0) = 1$. Wir wollen nun zeigen, dass $p(k) \geq \frac{1}{k+1}$ gilt. Der Induktionanfang gilt wegen $p(0) = 1$. Wir nehmen an, dass die Behauptung für $k-1$ gilt.

$$\begin{aligned}
p(k) &\geq 1 - \big(1 - \frac{p(k-1)}{2}\big)^2 \\
&\geq 1 - \big(1 - \frac{1}{2k}\big)^2 \\
&= \frac{1}{k} - \frac{1}{4k^2} \\
&= \big(\frac{(4k-1)(k-1)}{4k^2}\big) \cdot \big(\frac{1}{k+1}\big) \\
&= \big(\frac{4k^2+4k-k-1}{4k^2}\big) \cdot \big(\frac{1}{k+1}\big) \\
&\geq \frac{1}{k+1}
\end{aligned}$$

Da die Rekursiontiefe für einen Graphen mit n Knoten $O(\log n)$ ist, folgt Satz 6.3.1.

7 Markov Ketten und Random Walks

Wir werden in diesem Kapitel einen randomisierten 2-SAT Algorithmus kennenlernen. Um diesen Algorithmus zu analysieren, werden wir das Konzept der Markov Ketten einführen.

7.1 Ein 2-SAT Algorithmus

Das 2-SAT Problem ist definiert wie folgt. Wir haben eine Menge von n Variable $\{x_1, \ldots, x_n\}$ und eine boolesche Formel f in konjunktiver Normalform, deren Klauseln aus genau zwei Literalen bestehen. Die Literale zu einer Variable x_i sind x_i und $\bar{x}_i$, wobei $\bar{x}_i$ den booleschen Ausdruck $(\neg x_i)$ abkürzt. Wir sollen für f eine erfüllende Belegung finden, sofern f erfüllbar ist.
Beispiel: Sei die Menge der Variablen $\{x_1, x_2, x_3\}$. Dann ist die Formel

$$(x_1 \vee x_2) \wedge (x_1 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee x_3)$$

erfüllbar mit $x_1 = 1$, $x_2 = 1$ und $x_3 = 1$. Andererseits ist die Formel

$$(x_1 \vee x_2) \wedge (x_1 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_3)$$

nicht erfüllbar, da die ersten beiden Klauseln $x_1 = 1$ implizieren, während aus den letzten beiden Klauseln $x_1 = 0$ folgt.

7.1.1 Der Algorithmus

Als nächstes werden wir nun den randomisierten 2-Sat Algorithmus kennenlernen, den wir in diesem Kapitel analysieren wollen.

RAND2SAT(f)
1. Initialisiere alle Variablen mit 0
2. **while** f ist nicht erfüllt **do**
3. Sei C eine nicht erfüllte Klausel
4. Nimm eine Variable x aus C zufällig und gleichverteilt
5. Invertiere den Wert von x
6. Gib die aktuelle Belegung aus

Offensichtlich terminiert der Algorithmus nicht, wenn f nicht erfüllbar ist. Was passiert aber, wenn f eine erfüllende Belegung besitzt?

Terminiert der Algorithmus immer?

Was ist die erwartete Anzahl Iterationen des Algorithmus?

Sei A eine beliebige erfüllende Belegung. Wir messen den Fortschritt des Algorithmus daran, für wie viele Variablen die aktuelle Belegung des Algorithmus mit A übereinstimmt. Wir können also den Fortschritt des Algorithmus als ein Teilchen visualisieren, das sich auf den Zahlen $\{0, \ldots, n\}$ bewegt. Dabei bezeichnet die Zahl, auf der das Teilchen steht, die Anzahl der Variablen, für die die aktuelle Belegung des Algorithmus mit A übereinstimmt. Falls das Teilchen noch nicht auf Position n steht, so bewegt es sich mit Wahrscheinlichkeit mindestens $1/2$ von der aktuellen Position i nach $i + 1$. Der Grund hierfür ist, dass mindestens eine der beiden Variablen in einer nicht erfüllten Klausel anders als in A belegt sein muss (da A eine erfüllende Belegung ist). Damit ist die Wahrscheinlichkeit, dass eine Variable invertiert wird, die anders als in A belegt ist, mindestens $1/2$.

Der Algorithmus terminiert, wenn das Teilchen den Wert n erreicht. Zur Analyse werden wir nun annehmen, dass sich das Teilchen in jedem Schritt mit Wahrscheinlichkeit $1/2$ nach $i+1$ bzw. $i-1$ bewegt. Dies ist der schlechteste Fall für den Algorithmus. Wir werden zeigen, dass das Teilchen in diesem Fall nach $O(n^2)$ Schritten den Wert n erreicht hat. Daraus folgt

Satz 7.1.1 *Sei f erfüllbar. Dann terminiert* RAND2SAT *nach $O(n^2)$ erwarteten Iterationen und gibt eine erfüllende Belegung aus.*

Wir können nun aus diesem Algorithmus einen Monte-Carlo Algorithmus für Erfüllbarkeit von booleschen Formeln herleiten. Dazu sei X eine Zufallsvariable für die Anzahl Iterationen des Algorithmus. Weiterhin sei c eine Konstante für die gilt, $E\big[X\big] \leq c \cdot n^2$. Dann folgt nach der Markov Ungleichung

$$\text{Prob}\big[X \geq 4E\big[X\big]\big] \leq 1/4 \ .$$

Wenn wir also die Schleife nach $4cn^2$ Durchläufen abbrechen, dann wissen wir, dass unser Algorithmus mit Wahrscheinlichkeit $3/4$ eine erfüllende Belegung für f gefunden hat, sofern f erfüllbar ist. Hat der Algorithmus keine erfüllende Belegung gefunden, dann gibt der modifizierte Algorithmus aus, dass es keine solche Belegung gibt.

Korollar 7.1.2 *Der modifizierte* RAND2SAT *Algorithmus weist eine nicht erfüllbare Formel f immer zurück und akzeptiert eine erfüllbare Formel mit Wahrscheinlichkeit mindestens $3/4$ (diese Wahrscheinlichkeit kann natürlich mit Amplifikation auf eine beliebige Konstante verbessert werden).*

7.2 Markov Ketten

Zur Analyse des randomierten 2-SAT Algorithmuus führen wir nun Markov Ketten ein. Eine Markov Kette ist ein diskreter stochastischer Prozeß, der über einer endlichen oder abzählbar unendlichen Menge von Zuständen Ω definiert ist. Eine Markov Kette $\mathcal{M}$ ist in Zustand $X_t \in \Omega$ zu Zeitpunkt $t \geq 0$. Der Anfangszustand ist X_0. Die Übergangswahrscheinlichkeiten werden als Matrix P gegeben, die für jeden Zustand in Ω eine Reihe und eine Spalte besitzt, d.h.

$$P_{i,j} = \text{Prob}\big[X_{t+1} = j | X_t = i\big] \ .$$

Eine wichtige Eigenschaft von Markov Ketten ist, dass das zukünftige Verhalten nur vom aktuellen Zustand abhängt und nicht, wie man dorthin gekommen ist. Für Zustände $i, j \in \Omega$ definieren wir die t-Schritt Übergangswahrscheinlichkeit als

$$P_{i,j}^{(t)} = \text{Prob}\big[X_t = j | X_0 = i\big] \ .$$

Für den Anfangszustand $X_0 = i$ definieren wir die Wahrscheinlichkeit, dass der erste Übergang in Zustand j zum Zeitpunkt t erfolgt, als

$$r_{i,j}^{(t)} = \text{Prob}\big[X_t = j \text{ und } X_s \neq j \text{ für } 1 \leq s < t | X_0 = i\big] \ .$$

Damit ist die erwartete Anzahl Schritte bis Zustand j vom Startzustand i erreicht wird,

$$h_{i,j} = \sum_{t>0} t \cdot r_{i,j}^{(t)} \ .$$

Der $\mathcal{M}$ zugrundeliegende Graph besitzt einen Knoten für jeden Zustand und eine gerichtete Kante von i nach j, wenn $P_{i,j} > 0$. Eine Markov Kette ist *irreduzibel*, wenn der zugrundeliegende Graph aus einer starken Zusammenhangskomponente besteht.

Sei $\mathcal{M}$ eine Markov Kette mit Zustandsmenge Ω, $|\Omega| = n$. $\mathcal{M}$ habe einen festen Startzustand. Dann bezeichnet

$$q^{(t)} = (q_1^{(t)}, q_2^{(t)}, \ldots, q_n^{(t)})$$

die Verteilung der Kette zum Zeitpunkt t. Dabei gibt die i-te Komponente dieses Zeilenvektors die Wahrscheinlichkeit an, dass die Kette zum Zeitpunkt t in Zustand i ist. Es gilt

$$q^{(t)} = q^{(t-1)} \cdot P$$

und damit

$$q^{(t)} = q^{(0)} \cdot P^t \ .$$

Definition 7.2.1 *Eine* stationäre Verteilung *für eine Markov Kette ist eine Wahrscheinlichkeitsverteilung π mit $\pi = \pi \cdot P$.*

Die *Periode* eines Zustands i ist die größte Zahl k, für die es eine Anfangsverteilung $q^{(0)}$ gibt, so dass $q_i^{(t)} > 0$, genau dann wenn t ein Vielfaches von k ist. Eine Markov Kette mit Periode 1 heißt *aperiodisch*. Damit können wir eine (leicht vereinfachte) Version des Fundamentalsatzes für Markov Ketten formulieren. Wir werden diesen Satz nicht beweisen.

Satz 7.2.2 (Fundamentalsatz für Markov Ketten) *Jede irreduzible, endliche und aperiodische Markov Kette hat folgende Eigenschaften*

$\mathcal{M}$ hat eine eindeutige stationäre Verteilung π.

Für $1 \leq i \leq n$: $\pi_i > 0$ und $h_{i,i} = \frac{1}{\pi_i}$.

Sei $N(i,t)$ die Anzahl Besuche des Zustands i in t Schritten. Dann gilt

$$\lim_{t\to\infty} \frac{N(i,t)}{t} = \pi_i \ .$$

7.3 Random Walks

Wir werden in diesem Kapitel sogenannte *Random Walks* auf ungerichteten Graphen analysieren. Die Analyse unseres 2-SAT Algorithmus folgt als Spezialfall aus diesem Ergebnis.
Sei $G = (V, E)$ ein zusammenhängender, nicht bipartiter, ungerichteter Graph mit $|V| = n$ und $|E| = m$. Wir wollen folgenden stochastischen Prozeß analysieren. Ein Partikel startet an einem beliebigen Startknoten. In jedem Schritt sucht sich das Partikel einen der Nachbarknoten zufällig und gleichverteilt aus und bewegt sich dorthin. Man bezeichnet diesen Prozeß als *Random Walk*. Einen Random Walk kann man als Markov Kette $\mathcal{M}_G$ beschreiben. V ist die Menge der Zustände von $\mathcal{M}_G$. Außerdem gilt $P_{u,v} = 1/d(u)$, wenn $(u, v) \in E$ und $P_{u,v} = 0$, sonst. Dabei bezeichnet $d(u)$ den Grad von Knoten u. $\mathcal{M}_G$ ist irreduzibel, da G zusammenhängend ist. $\mathcal{M}_G$ ist aperiodisch, weil G nicht bipartit ist (Übung). Also folgt aus dem Fundamentalsatz, dass $\mathcal{M}_G$ eine eindeutige stationäre Verteilung hat.

Satz 7.3.1 *Für jeden Knoten $v \in V$, $\pi_v = \frac{d(v)}{2m}$.*

Beweis: Wir benutzen eine Hilfskette $\mathcal{M}'_G$, die denselben Random Walk beschreibt. Ungerichtete Kanten in G ersetzen wir dazu durch je zwei gerichtete. Die Menge der gerichteten Kanten ist dann der Zustandsraum Ω' von $\mathcal{M}'_G$. Der aktuelle Zustand ist die zuletzt durchlaufende (gerichtete) Kante. Die Übergangsmatrix ist gegeben durch

$$P_{[u,v\rangle,[v,w\rangle} = P_{v,w} = \frac{1}{d(v)}$$

und 0 an allen anderen Stellen. P' ist eine doppelt stochastische Matrix, weil sowohl die Zeilensummen als auch die Spaltensummen jeweils 1 sind, wie folgende Rechnung zeigt. Für $[u,w\rangle \in \Omega'$ ist die Summe der Spalte $[v,w\rangle$:

$$\begin{aligned}\sum_{x\in V, y\in\Gamma(x)} P'_{[x,y\rangle,[v,w\rangle} &= \sum_{u\in\Gamma(v)} P'_{[u,v\rangle,[v,w\rangle} \\ &= \sum_{u\in\Gamma(v)} P_{v,w} \\ &= d(v)\cdot\frac{1}{d(v)} = 1\end{aligned}$$

Dabei bezeichnet $\Gamma(v)$ die Menge aller Nachbarn von v.

Lemma 7.3.2 *Die stationäre Verteilung einer doppelt stochastische Matrix ist die Gleichverteilung.*

Beweis: Sei Ω^* der Zustandsraum von Markov Kette $\mathcal{M}^*$ mit der doppelt stochastischen Matrix P^*. Sei $k = |\Omega^*|$. Es gibt nach dem Fundamentalsatz nur eine stationäre Verteilung. Also müssen wir nur zeigen, dass $\pi^* = (1/k, \ldots, 1/k)$ eine stationäre Verteilung ist. Dies folgt aus:

$$[\pi^* \cdot P^*]_i = \sum_{j\in\Omega^*} \pi^*_j \cdot P^*_{j,i} = \frac{1}{k}\sum_{j\in\Omega^*} P^*_{j,i} = \frac{1}{k} \ .$$

Korollar 7.3.3 *Für jede gerichtete Kante $[u,v\rangle$ gilt, $\pi'_{[u,v\rangle} = \frac{1}{2m}$.*

Also gilt insbesondere:

$$\pi_v = \sum_{u\in\Gamma(v)} \pi'_{[u,v\rangle} = \frac{d(v)}{2m} \ .$$

Wir wollen analysieren, wie lange ein Random Walk braucht, um alle Knoten zu besuchen. Dazu benötigen wir noch zwei Definitionen.

Definition 7.3.4 *Die* Hitting Time $h_{u,v}$ *eines Random Walk ist die erwartete Anzahl Schritte, die benötigt wird, um zu Knoten v zu gelangen, wenn der Random Walk bei Knoten u startet. Die* Cover Time $C(G)$ *eines Random Walks ist gegeben durch $C(G) = \max_u C_u(G)$, wobei $C_u(G)$ die erwartete Anzahl von Schritten bezeichnet, die ein Random Walk, der bei u startet, benötigt, um alle Knoten zu besuchen und zu u zurückzukehren.*

Satz 7.3.5 *Für jeden Graph G gilt $C(G) \leq 2m(n-1)$.*

Beweis: Sei T ein Spannbaum von G. Für jeden Knoten gibt es eine Tour $v = v_0, \ldots, v_{2n-2} = v$, die jede Kante von T genau einmal in jede Richtung benutzt. Es folgt

$$C_v(G) \leq \sum_{j=0}^{2n-3} h_{v_j,v_{j+1}} = \sum_{(u,w)\in T} (h_{u,w} + h_{w,u}) \ .$$

Wir betrachten nun $h_{u,w}+h_{w,u}$ als die erwartete Zeit für einen Random Walk, der bei Knoten u startet, w besucht und zu u zurückkehrt. Aufgrund unserer Konstruktion gibt es die Kante (u,w).

Lemma 7.3.6 *Für jede Kante (u, w) gilt:*

$$h_{u,w} + h_{w,u} \leq 2m \ .$$

Beweis: Nach dem Fundamentalsatz und dem Korollar über den Random Walk $\mathcal{M}'_G$ folgt, dass die erwartete Anzahl Schritte, die ein Random Walk, der in $[w, u\rangle$ startet, benötigt, um zu $[w, u\rangle$ zurückzukehren, genau $2m$ ist. Da der Random Walk nur von der aktuelle Position abhängt, gilt für jeden Zustand $[x, u\rangle$ mit $(x, u) \in E$, dass die erwartete Anzahl Schritte, die ein Random Walk, der in $[x, u\rangle$ (mit $(x, u) \in E$) startet, benötigt, um zu $[w, u\rangle$ zu gelangen, genau $2m$ ist. Daraus folgt, dass für ein Partikel, das auf Knoten u steht, die erwartete Anzahl Schritte bis der Random Walk in Zustand $[w, u\rangle$ gelangt, genau $2m$ ist. Gelangt der Random Walk aber in Zustand $[w, u\rangle$, so hat sich das Partikel in der letzten Runde von w nach u bewegt. Also wissen wir, dass sich nach einer erwarteten Anzahl von höchstens $2m$ Schritten das Partikel von u nach w und wieder nach u bewegt hat. Damit folgt $h_{u,w} + h_{w,u} \leq 2m$.
Also folgt $C(G) = \max_v C_v(G) \leq 2m \cdot |T| = 2m(n-1)$.
Wenn wir diesen Satz auf den Linien-Graph anwenden, dann folgt der Satz über die 2-SAT Algorithmen.

8 String Matching

8.1 Definitionen

In diesem Kapitel geht es um Algorithmen für das folgende Problem:

Problem 8.1.1 (String Matching) *Gegeben sind ein Text $T[1 \dots n]$ und ein Muster $P[1 \dots m]$. Gesucht sind alle Vorkommen von P in T.*

Beispiel 8.1.2 *$T = agcabagbcd$, $P = ag$*
Das Muster taucht mit Verschiebung 0 und mit Verschiebung 5 auf.

Bemerkung 8.1.3 *Das Problem kann in Laufzeit $m \cdot n$ gelöst werden.*

Wir interessieren uns dafür, die Abhängigkeit von m abzuschwächen. Dies ist dann interessant, wenn das gesuchte Muster lang ist.

Notation 8.1.4
Σ: *Alphabet*
Σ^*: *Menge aller endlichen Zeichenketten aus Σ*
ϵ: *leeres Wort (insbesondere $\epsilon \in \Sigma^*$)*

Definition 8.1.5 *Eine Zeichenkette $w \in \Sigma^*$ heißt* Präfix *von x, geschrieben $w \sqsubset x$, wenn $x = wy$ für ein $y \in \Sigma^*$.*

Beispiel 8.1.6 *aab ist Präfix von aabxy. ab ist Präfix von ab.*

Definition 8.1.7 *Eine Zeichenkette $w \in \Sigma^*$ heißt* Suffix *von x, geschrieben $w \sqsupset x$, wenn $x = yw$ für ein $y \in \Sigma^*$.*

Beispiel 8.1.8 *aab ist ein Suffix von xyaab. ab ist Suffix von ab.*

Vorsicht: $w \sqsubset x \not\Leftrightarrow x \sqsupset w$

Definition 8.1.9 *Ein endlicher Automat M ist ein 5-Tupel $(Q, q_0, A, \Sigma, \delta)$, wobei*

Q eine endliche Zustandsmenge,

$q_0 \in Q$ der Startzustand des Automaten,

$A \subseteq Q$ die Menge der Endzustände,

Σ das Eingabealphabet und

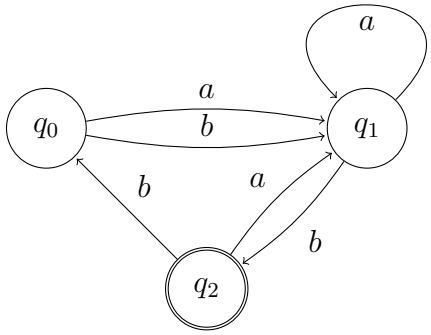
δ eine Funktion von $Q \times \Sigma$ nach Q ist, die man Übergangsfunktion *nennt.*

Die Erweiterung von δ auf Zeichenketten, $\phi : \Sigma^* \to Q$, nennen wir Endzustandsfunktion.
Wir definieren ϕ rekursiv durch

$$\phi(\epsilon) = q_0$$
$$\phi(wa) = \delta(\phi(w), a) \text{ für } w \in \Sigma^*,\ a \in \Sigma$$

Insbesondere akzeptiert ein Automat ein Wort w genau dann wenn $\phi(w) \in A$.

Beispiel 8.1.10

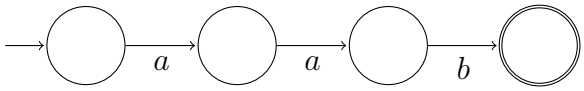


$Q = \{q_0, q_1, q_2\}$
$A = \{q_2\}$
$\Sigma = \{a, b\}$

Zustand	*Zeichen*	
	a	b
q_0	q_1	q_1
q_1	q_1	q_2
q_2	q_0	q_2

$\phi(aab) = q_2$
$\phi(aba) = q_1$

Wir wollen unser Problem mit Hilfe eines endlichen Automaten lösen.
Betrachte das Muster $P = aab$. Wir könnten den folgenden Automaten verwenden:



Dieser Automat kann P nur erkennen, wenn $T = P$ ist. Wir möchten aber P auch in $T = aaaabcaa$ erkennen können. Dazu müssen wir, wenn wir das dritte und vierte a finden, in unserem Automaten in den passenden Zustand „springen“. Diese Idee wollen wir jetzt genauer weiterverfolen.

Definition 8.1.11 (Suffix-Funktion) *Mit $\sigma(x)$ bezeichnen wir die Länge des längsten Präfix von P, das ein Suffix von x ist ($\sigma(x)$ ist also für ein spezielles Mustter P definiert), d.h.*

$$\sigma(x) = \max\{k : P_k \sqsupset x\},$$

wobei $P_k = P[1, \ldots, k]$ für $k \leq m$.

Beispiel 8.1.12
$x = aabaabca$
$P = bcaad$
$\sigma(x) = 3$

$T = aabaabca\ abadad$
$\uparrow$

Mit Hilfe der Suffix-Funktion entwerfen wir jetzt einen Algorithmus, der für ein festes Muster P auf allen eingegebenen Texten T funktioniert. Dazu entwickeln wir einen endlichen Automaten, dessen Zustände speichern, wie viele Buchstaben des Musters wir bereits gelesen haben.

8.2 String-Matching-Automat für $P[1, \ldots, m]$

$Q = \{0, 1, \ldots, m\}$
$q = 0$
$A = \{m\}$
$\delta(q, a) = \sigma(p_q a)$
Σ stimmt mit dem Alphabet überein, über dem P und T definiert sind.

Beispiel 8.2.1
$x = aabaa\underline{bca}$
$P = \underline{bca}ad$
$\sigma(x) = 3$

$T = aabaa\underline{bca}\ abadad$
$\qquad\qquad\quad\uparrow$

Nach dem Lesen von bca sind wir in Zustand 3. *Wenn wir a lesen, gehen wir in Zustand* $\sigma(bcaa) = 4$ *über.*

Wir erhalten einen Algorithmus, indem wir den endlichen Automaten simulieren:

Algorithmus 8.2.2 (Finite-Automaton-Matcher(T, δ, m)**)**

1. $n \leftarrow$ *length*$[T]$
2. $q \leftarrow 0$
3. for $i \leftarrow 1$ *to* n *do*
4. $\quad q \leftarrow \delta(q, T[i])$
5. $\quad$ *if* $q = m$ *then*
6. $\quad\quad s \leftarrow i - m$
7. $\quad\quad$ *print "Muster tritt mit Verschiebung* s *auf"*

Bemerkung 8.2.3 *Der Algorithmus hat Laufzeit* $\mathcal{O}(m)$*, unter der Annahme, dass* δ *bereits berechnet wurde.*

Unser Ziel ist es jetzt, zu zeigen, dass der String-Matching-Automat die Invariante $\phi(T_i) = \sigma(T_i)$ erfüllt. Dazu benötigen wir zuerst zwei Lemmata.

Lemma 8.2.4 *Für jeden String* x *und Zeichen* a *gilt* $\sigma(xa) \leq \sigma(x) + 1$.

Beweis: Sei $R = \sigma(xa)$.
Wenn $r = 0$, dann folgt das Lemma wegen der Nichtnegativität von r.
Sei also $r > 0$. Dann gilt $P_r \sqsupset xa$ nach Definition von σ. Also ist $P_{r-1} \sqsupset x$, da wir a bei beiden Strings weglassen können. Damit gilt $r - 1 \leq \sigma(x)$, da $\sigma(x)$ das größte k ist, so dass $P_k \sqsupset x$.

Lemma 8.2.5 *Für jeden String* x *und Zeichen* a *gilt:*
Wenn $q = \sigma(x)$*, dann ist* $\sigma(xa) = \sigma(P_q a)$.

Beweis: Nach der Definition von σ haben wir $P_q \sqsupset x$.
Weiterhin gilt $P_q a \sqsupset xa$. Wenn wir nun $r = \sigma(xa)$ setzen, dann gilt $r \leq q + 1$ nach Lemma 8.2.4 . Da $P_q a \sqsupset xa$, $P_r \sqsupset xa$ und $|P_r| \leq |P_q a|$, folgt $P_r \sqsupset P_q a$. Daher ist $r \leq \sigma(P_q a)$, d.h. $\sigma(xa) \leq \sigma(P_q a)$.
Aber wir haben auch $\sigma(P_q a) \leq \sigma(xa)$, da $P_q a \sqsupset xa$.
Also gilt $\sigma(xa) = \sigma(P_q a)$.

Satz 8.2.6 *Wenn ϕ die Endzustandsfunktion eines String-Matching-Automaten für gegebenes Muster P ist und $T[1,\ldots,n]$ Eingabetext für den Automaten ist, dann gilt:*

$$\phi(T_i) = \sigma(T_i) \text{ für } i = 0,\ldots,n.$$

Beweis:
Induktion über i.
$\underline{i=0}$: $T_0 = \epsilon$, $\phi(T_0) = \sigma(T_0) = 0$

$\underline{\text{Induktionsannahme:}}$ $\phi(T_i) = \sigma(T_i)$
$\underline{\text{Induktionsschluss:}}$ Zu zeigen: $\phi(T_{i+1}) = \sigma(T_{i+1})$.
Sei $q = \phi(T_i)$ und sei $a = T[i+1]$.
Dann gilt:

$$\begin{aligned}
\phi(T_{i+1}) &= \phi(T_i a) \\
&= \delta(\phi(T_i), a) && \text{(Def. von } \phi\text{)} \\
&= \delta(q, a) && \text{(Def. von } q\text{)} \\
&= \sigma(P_q a) && \text{(Def. von } \delta\text{)} \\
&= \sigma(T_i a) && \text{(Nach Lemma 8.2.4 und Induktion)} \\
&= \sigma(T_{i+1}),
\end{aligned}$$

wobei uns die Induktion $\phi(T_i) = \sigma(T_i)$ liefert.

Damit funktioniert unser Algorithmus, sofern wir dem String-Matching-Automaten berechnen können. Für den Automaten benötigen wir die Übergangsfunktion. Also:

Algorithmus 8.2.7 (Berechne Übergangsfunktion (P, Σ)**)**

```
1.  m ← length[P]
2.  for q ← 0 to m do
3.      for jedes Zeichen a ∈ Σ do
4.          k ← min(m + 1, q + 2)
5.          repeat
6.              k ← k − 1
7.          until
8.              P_k ⊐ P_q a
9.          δ(q, a) ← k
10. return δ
```

Bemerkung 8.2.8 *Algorithmus 8.2.7 hat Laufzeit $\mathcal{O}(m^3 \cdot |\Sigma|)$, da die Schleifen in Zeile 2 und 5 je $\mathcal{O}(m)$ Mal aufgerufen werden, die Schleife in Zeile 3 $|\Sigma|$ Aufrufe hat und die Ausführung von Zeile 8 $\mathcal{O}(m)$ Zeit benötigt.*

Bemerkung 8.2.9 *Es gibt ein verbessertes Verfahren für die Berechnung der Übergangsfunktion, das Laufzeit $\mathcal{O}(m \cdot |\Sigma|)$ erreicht. Damit hat das Gesamtverfahren Laufzeit $\mathcal{O}(n + m \cdot |\Sigma|)$.*

Literatur

[1] Alexander Schrijver. *Theory of Linear and Integer Programming.* Wiley, 1998.

[2] Raimund Seidel. Skript zur Vorlesung „Optimierung“ im SoSe 1996. Universität des Saarlands.

[3] Vijay V. Vazirani. *Approximation Algorithms.* Springer, 2001.