Optimierung für Studierende der Informatik

Dieses Dokument wurde entworfen und erstellt von
THOMAS ANDREAE

Vorlesung gehalten von Matthias Voigt
Universität Hamburg
Wintersemester 2020/21

(Stand: 31. Januar 2018)

Vorwort

Das vorliegende Skript umfasst im Wesentlichen den Stoff der 2-stündigen Vorlesung „Optimierung für Studierende der Informatik“ an der Universität Hamburg im Wintersemester 2017/18.

Ich bedanke mich an dieser Stelle recht herzlich bei Steven Köhler, der – wie schon bei der Erstellung früherer Skripte – mit viel Engagement für das Setzen des Skripts in LaTeX gesorgt hat.

Hamburg, Januar 2018 Thomas Andreae

Inhaltsverzeichnis

1 Einführung

Wir gehen zunächst hauptsächlich nach folgendem Lehrbuch vor:

- V. Chvátal: *Linear Programming*. W. H. Freeman and Company. New York (2002, 16. Auflage).

Später wird weitere Literatur hinzugezogen werden, beispielsweise:

- Th. Cormen, Ch. Leiserson, R. Rivest, C. Stein: *Algorithmen - Eine Einführung*. Oldenburg Verlag. (2010, 3. Auflage).
- J. Matoušek, B. Gärtner: *Understanding and Using Linear Programming*. Springer-Verlag. Berlin, Heidelberg, New York (2007).
- J. Kleinberg, É. Tardos: *Algorithm Design*. Pearson. Boston (2006).
- D. Jungnickel: *Graphs, Networks and Algorithms*. Springer-Verlag (2013, 4. Auflage).

1.1 Ein Diätproblem

Paul möchte sich möglichst preiswert ernähren, allerdings so, dass er pro Tag mindestens 2000kcal, 55g Proteine und 800mg Kalzium erhält. Er wählt sechs Lebensmittel aus, die günstig zu erstehen sind:

1. *Haferflocken*: Eine 28g Packung liefert 110kcal, 4g Protein, 2mg Kalzium und kostet 25 Cent.
2. *Huhn*: 100g liefern 205kcal, 32g Protein und 12mg Kalzium; Preis: 130 Cent.
3. *Eier*: Eine Doppelpackung liefert 160kcal, 13g Protein und 54mg Kalzium; bezahlen muss man 85 Cent pro Doppelpack.
4. *Milch*: Eine kleine Packung liefert 160kcal, 8g Protein und 285mg Kalzium; Preis: 70 Cent.
5. *Kirschkuchen*: Ein Stück (170g) liefert 420kcal, 4g Protein und 22mg Kalzium; Preis: 95 Cent.
6. *Bohnen*: Eine Packung (260g) liefert 260kcal, 14g Protein und 80mg Kalzium; Preis: 98 Cent.

Dasselbe in der Übersicht (Angaben pro Portion des jeweiligen Lebensmittels; 1 Portion Haferflocken $\widehat{=}$ 28g, 1 Portion Huhn $\widehat{=}$ 100g, 1 Portion Eier $\widehat{=}$ 1 Doppelpackung, etc.):

	Energie (kcal)	Protein (g)	Kalzium (mg)	Preis (Cent)
Haferflocken	110	4	2	25
Huhn	205	32	12	130
Eier	160	13	54	85
Milch	160	8	285	70
Kirschkuchen	420	4	22	95
Bohnen	260	14	80	98

Paul denkt über seine Mahlzeiten nach: Beispielsweise würden 10 Portionen Bohnen alles Erforderliche an Energie, Protein und Kalzium liefern – zu einem Preis von nur (?) 980 Cent pro Tag. Mehr als 2 Portionen Bohnen pro Tag ist aber zu viel; er legt Obergrenzen fest:

- Haferflocken: höchstens 4 Portionen pro Tag;
- Huhn: höchstens 3 Portionen pro Tag;
- Eier: höchstens 2 Portionen pro Tag;

- Milch: höchstens 8 Portionen pro Tag;
- Kirschkuchen: höchstens 2 Portionen pro Tag;
- Bohnen: höchstens 2 Portionen pro Tag.

Ein Blick auf die Daten lässt erkennen: 8 Portionen Milch und 2 Portionen Kirschkuchen pro Tag liefern alles Nötige für nur 750 Cent. Man könnte auch etwas weniger Kuchen nehmen oder etwas weniger Milch – oder eine andere Kombination ausprobieren: *trial and error* nennt man das. Hilft das weiter?

Um systematisch vorzugehen, führen wir für jedes Lebensmittel eine Variable ein: x_1 bezeichnet die Anzahl der Portionen von Haferflocken pro Tag, x_2 die Anzahl der Huhn-Portionen pro Tag, x_3 die Anzahl der Ei-Portionen etc. Beispielsweise bedeutet

$$x_6 = 1.5,$$

dass Paul 1.5 Portionen Bohnen pro Tag zu sich nimmt.

Wir formulieren das vorgestellte Problem – man spricht von einem *Diätproblem* – auf die folgende Art:

$$\begin{aligned}
&\text{minimiere } 25x_1 + 130x_2 + 85x_3 + 70x_4 + 95x_5 + 98x_6 \\
&\text{unter den Nebenbedingungen} \\
&\qquad 110x_1 + 205x_2 + 160x_3 + 160x_4 + 420x_5 + 260x_6 \geq 2000 \\
&\qquad 4x_1 + 32x_2 + 13x_3 + 8x_4 + 4x_5 + 14x_6 \geq 55 \\
&\qquad 2x_1 + 12x_2 + 54x_3 + 285x_4 + 22x_5 + 80x_6 \geq 800 \\
&\qquad\qquad 0 \leq x_1 \leq 4 \\
&\qquad\qquad 0 \leq x_2 \leq 3 \\
&\qquad\qquad 0 \leq x_3 \leq 2 \\
&\qquad\qquad 0 \leq x_4 \leq 8 \\
&\qquad\qquad 0 \leq x_5 \leq 2 \\
&\qquad\qquad 0 \leq x_6 \leq 2.
\end{aligned}$$

Probleme dieser Art werden *lineare Programmierungsprobleme*[1] genannt. Hier sind zwei weitere **Beispiele**:

$$\begin{aligned}
&\text{maximiere } 5x_1 + 4x_2 + 3x_3 \\
&\text{unter den Nebenbedingungen}^{[2]} \\
&\qquad 2x_1 + 3x_2 + x_3 \leq 5 \\
&\qquad 4x_1 + x_2 + 2x_3 \leq 11 \\
&\qquad 3x_1 + 4x_2 + 2x_3 \leq 8 \\
&\qquad\qquad x_1, x_2, x_3 \geq 0
\end{aligned} \tag{1.1}$$

$$\begin{aligned}
&\text{minimiere } 3x_1 - x_2 \\
&\text{unter den Nebenbedingungen} \\
&\qquad -x_1 + 6x_2 - x_3 + x_4 \geq -3 \\
&\qquad\qquad 7x_2 + 2x_4 = 5 \\
&\qquad x_1 + x_2 + x_3 = 1 \\
&\qquad\qquad x_3 + x_4 \leq 2 \\
&\qquad\qquad x_2, x_3 \geq 0
\end{aligned} \tag{1.2}$$

[1] Kurz: *LP-Probleme*.

[2] Die Schreibweise $x_1, x_2, x_3 \geq 0$ ist eine Abkürzung für $x_1 \geq 0$, $x_2 \geq 0$, $x_3 \geq 0$.

Definition.
Eine Funktion $f : \mathbb{R}^n \to \mathbb{R}$ wird *lineare Funktion* genannt, falls

$$f(x_1, \ldots, x_n) = c_1 x_1 + \ldots + c_n x_n \tag{1.3}$$

für reelle Zahlen $c_1, \ldots, c_n$ gilt.

Mit dem Summenzeichen geschrieben lautet die Gleichung (1.3):

$$f(x_1, \ldots, x_n) = \sum_{j=1}^{n} c_j x_j. \tag{1.3'}$$

Ist b eine reelle Zahl, so nennt man

$$c_1 x_1 + \ldots + c_n x_n = b \tag{1.4}$$

bekanntlich eine *lineare Gleichung*. Neben linearen Gleichungen betrachten wir *lineare Ungleichungen*:

$$c_1 x_1 + \ldots + c_n x_n \leq b \tag{1.5}$$
$$c_1 x_1 + \ldots + c_n x_n \geq b. \tag{1.6}$$

Gleichungen bzw. Ungleichungen der Arten (1.4), (1.5) und (1.6) werden von uns *lineare Nebenbedingungen* oder einfach nur *Nebenbedingungen*[3] genannt.

Unter einem *linearen Programmierungsproblem*[4] verstehen wir das Problem, eine lineare Funktion unter einer endlichen Anzahl von linearen Nebenbedingungen zu minimieren oder zu maximieren.

Wir beschränken uns bis auf Weiteres auf lineare Programmierungsprobleme vom folgenden Typ:

$$\begin{aligned}
&\text{maximiere} \quad c_1 x_1 + \ldots + \quad c_n x_n \\
&\text{unter den Nebenbedingungen} \\
&\qquad a_{11} x_1 + \ldots + a_{1n} x_n \;\leq\; b_1 \\
&\qquad\qquad \vdots \\
&\qquad a_{m1} x_1 + \ldots + a_{mn} x_n \;\leq\; b_m \\
&\qquad\qquad x_1, \ldots, x_n \;\geq\; 0.
\end{aligned}$$

Dasselbe in knapper Form mit dem Summenzeichen geschrieben:

$$\begin{aligned}
&\text{maximiere} \quad \sum_{j=1}^{n} c_j x_j \\
&\text{unter den Nebenbedingungen} \\
&\qquad \sum_{j=1}^{n} a_{ij} x_j \leq b_j \quad (i = 1, \ldots, m) \\
&\qquad\qquad x_j \geq \; 0 \quad (j = 1, \ldots, n).
\end{aligned} \tag{1.7}$$

Wir nennen (1.7) ein LP-Problem in *Standardform*. (Warnung: Die Bezeichnung „Standardform“ wird in der Literatur nicht einheitlich verwendet; andere Autoren sagen stattdessen beispielsweise „Normalform“ und benutzen den Begriff „Standardform“ überhaupt nicht oder in anderer Bedeutung. Wir folgen hier der Bezeichnung von V. Chvátal: *Linear Programming*.)

[3] Strikte Ungleichungen ($<$ bzw. $>$ statt $\leq$ bzw. $\geq$) kommen nicht vor und fallen nicht unter den Begriff „Nebenbedingung“.

[4] Statt „lineares Programmierungsproblem“ sagt man auch *lineares Optimierungsproblem* oder (wie bereits erwähnt) *LP-Problem*.

In der Standardform (1.7) gibt es also $m+n$ Nebenbedingungen, von denen die letzten n recht speziell sind: Man nennt sie *Nichtnegativitätsbedingungen*.

Die lineare Funktion eines LP-Problems, die zu minimieren bzw. zu maximieren ist, wird die *Zielfunktion* des Problems genannt. In (1.1) ist also

$$5x_1 + 4x_2 + 3x_3$$

die Zielfunktion, während in (1.2) die Zielfunktion $3x_1 - x_2$ lautet.

Eine Belegung der Variablen $x_1, \ldots, x_n$, die die Nebenbedingungen eines LP-Problems erfüllt, nennt man eine *zulässige Lösung*. Zulässige Lösungen des LP-Problems (1.1) sind beispielsweise:

- $x_1 = 1$, $x_2 = 0$, $x_3 = 2$;
- $x_1 = 1$, $x_2 = \frac{1}{2}$, $x_3 = 1$;
- $x_1 = x_2 = x_3 = 0$.

Keine zulässigen Lösungen von (1.1) sind dagegen beispielsweise:

- $x_1 = 1$, $x_2 = 0$, $x_3 = 3$;
- $x_1 = 1$, $x_2 = 2$, $x_3 = -2$.

Eine zulässige Lösung, für die die Zielfunktion maximal bzw. minimal ist (je nach Art des LP-Problems), nennt man eine *optimale Lösung*. Den zu einer optimalen Lösung $x_1, \ldots, x_n$ gehörigen Zielfunktionswert nennt man den *optimalen Zielfunktionswert*.

Es gibt LP-Probleme, die keine zulässige Lösung besitzen; solche LP-Probleme nennt man *unlösbar*. Hier ein **Beispiel**:

$$\begin{aligned} &\text{maximiere} && 3x_1 - x_2 \\ &\text{unter den Nebenbedingungen} \\ & && x_1 + x_2 \leq 2 \\ & && -2x_1 - 2x_2 \leq -10 \\ & && x_1, x_2 \geq 0. \end{aligned}$$

Es gibt auch LP-Probleme, die zwar zulässige Lösungen besitzen, aber keine ihrer zulässigen Lösungen ist optimal. Hier ein **Beispiel**:

$$\begin{aligned} &\text{maximiere} && x_1 - x_2 \\ &\text{unter den Nebenbedingungen} \\ & && -2x_1 + x_2 \leq -1 \\ & && -x_1 - 2x_2 \leq -2 \\ & && x_1, x_2 \geq 0. \end{aligned}$$

In diesem Beispiel gibt es zu jeder Zahl M eine zulässige Lösung x_1, x_2 mit $x_1 - x_2 > M$. Derartige Probleme nennt man *unbeschränkt*.

In Matrixschreibweise lässt sich (1.7) besonders kompakt darstellen:

$$\begin{aligned} &\text{maximiere } c^T x \\ &\text{unter den Nebenbedingungen} \\ & && Ax \leq b \\ & && x \geq 0. \end{aligned} \tag{1.7'}$$

Hierin ist

$$c^T = (c_1, \ldots, c_n), \qquad x = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}, \qquad A = \begin{pmatrix} a_{11} & \ldots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \ldots & a_{mn} \end{pmatrix}, \qquad b = \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix}$$

und 0 bezeichnet den Nullvektor der Länge n; die beiden Ungleichungen sind komponentenweise zu verstehen.

Illustration von (1.7') anhand von Beispiel (1.1):

$$c^T = (5,4,3), \quad x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \quad A = \begin{pmatrix} 2 & 3 & 1 \\ 4 & 1 & 2 \\ 3 & 4 & 2 \end{pmatrix}, \quad b = \begin{pmatrix} 5 \\ 11 \\ 8 \end{pmatrix}, \quad 0 = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}.$$

Das LP-Problem (1.1) lässt sich mit diesen Bezeichnungen schreiben als:

$$\begin{aligned} &\text{maximiere } c^T x \\ &\text{unter den Nebenbedingungen} \\ &\qquad Ax \leq b \\ &\qquad x \geq 0. \end{aligned}$$

Einige Sprechweisen:

- Häufig wird auch einfach „lineares Programm“ anstelle von „lineares Programmierungsproblem“ („LP-Problem“) gesagt.
- Die Menge aller zulässigen Lösungen eines LP-Problems nennt man auch den *zulässigen Bereich*.
- Kommt eine Variable bei den Nichtnegativitätsbedingungen nicht vor, so spricht man von einer *freien Variablen*. Beispielsweise sind in (1.2) x_1 und x_4 freie Variablen.

Natürlich sollen Sie auch die englische Terminologie kennen. Hier sind die drei wichtigsten Vokabeln dieses Abschnitts:

constraint	–	Nebenbedingung
objective function	–	Zielfunktion
feasible solution	–	zulässige Lösung

Einfache Dinge wie

linear function – lineare Funktion

brauchen nicht erklärt zu werden, ebenso wenig wie die Bedeutungen von

linear constraints
nonnegativity constraints
standard form
optimal solution
optimal value
infeasible problem
unbounded problem
feasible region
free variable.

Ein weiteres **Beispiel**: **Ölraffinerie**[5]. Angeliefertes Rohöl wird in Ölraffinerien durch Anwendung von chemischen und physikalischen Verfahren in gewisse gewünschte Komponenten zerlegt. Die Ausbeute an verschiedenen Komponenten hängt vom eingesetzten Verfahren (Crackprozess) ab. Wir nehmen an, dass eine Raffinerie aus Rohöl drei Komponenten herstellen will:

[5] Dieses Beispiel ist u.a. im Skript zur Vorlesung *Algorithmen und Datenstrukturen* (Prof. Dr. Petra Mutzel, Universität Dortmund, WS 08/09) zu finden. Dort wird erwähnt, dass derartige Probleme in der Ölindustrie mithilfe von Linearer Programmierung gelöst werden – natürlich in ganz anderen Dimensionen. Im Skript von Grötschel (siehe Literaturverzeichnis) dient das Beispiel als Einstiegsbeispiel in die Lineare Optimierung; es wird dort noch wesentlich ausführlicher besprochen.

- schweres Öl: S;
- mittelschweres Öl: M;
- leichtes Öl: L.

Es stehen zwei Crackverfahren zur Verfügung, die die folgenden Einheiten an Ausbeute liefern, aber auch Kosten verursachen (alles bezogen auf eine Einheit Rohöl):

- Crackprozess 1: $2S$, $2M$, $1L$; Kosten: 3 Euro;
- Crackprozess 2: $1S$, $2M$, $4L$; Kosten: 5 Euro.

Aufgrund von Lieferbedingungen muss die Raffinerie die folgende Mindestproduktion erreichen:

$$3S, \quad 5M \quad \text{und} \quad 4L.$$

Aufgabe: Die Mengen sollen so kostengünstig wie möglich hergestellt werden.

Das hieraus resultierende LP-Problem erhalten wir nach Einführung von Variablen x_1 und x_2, die das *Produktionsniveau* der beiden Prozesse beschreiben: $x_1 = 2.5$ bedeutet beispielsweise, dass der Crackprozess 1 mit 2.5 Einheiten Rohöl beschickt wird. Man erhält das folgende LP-Problem:

$$\begin{aligned} &\text{minimiere } 3x_1 + 5x_2 \\ &\text{unter den Nebenbedingungen} \\ &\qquad 2x_1 + \ x_2 \ \geq 3 \\ &\qquad 2x_1 + 2x_2 \ \geq 5 \\ &\qquad \ x_1 + 4x_2 \ \geq 4 \\ &\qquad \ x_1, x_2 \ \geq 0. \end{aligned}$$

Zur Übung: Überführen Sie dieses Problem in Standardform.

Man kann – nebenbei gesagt – jedes LP-Problem durch sehr einfache, kleine „Tricks“ in Standardform überführen:

- Haben wir es beispielsweise mit einem Minimierungsproblem zu tun, lautet das Ziel etwa

  $$\text{minimiere } 3x_1 - 4x_2 + 5x_3,$$

  so kann man dies ersetzen durch die äquivalente Formulierung

  $$\text{maximiere } -3x_1 + 4x_2 - 5x_3.$$

- Eine Nebenbedingung der Form

  $$a_1x_1 + \ldots + a_nx_n \geq b$$

  kann gleichwertig ersetzt werden durch

  $$-a_1x_1 - \ldots - a_nx_n \leq -b.$$

- Eine Nebenbedingung der Form

  $$a_1x_1 + \ldots + a_nx_n = b$$

  kann gleichwertig ersetzt werden durch die folgenden beiden Nebenbedingungen

  $$\begin{aligned} a_1x_1 + \ldots + a_nx_n \ &\leq \ b \\ -a_1x_1 - \ldots - a_nx_n \ &\leq -b. \end{aligned}$$

Und was kann man machen, wenn man es mit einer *freien Variablen* zu tun hat? **Antwort**: Man kann anstelle von x_1 zwei neue Variablen einführen, etwa x_1' und x_1'', für die man $x_1' \geq 0$ und $x_1'' \geq 0$ fordert; anschließend ersetzt man überall x_1 durch $x_1' - x_1''$. Der Trick besteht also in diesem Fall darin, x_1 als Differenz zweier nichtnegativer Variablen darzustellen. Das funktioniert, da man (klarerweise) jede reelle Zahl als Differenz zweier nichtnegativer Zahlen schreiben kann.

1.2 Bemerkungen zur Grafischen Methode

Hat man es mit nur zwei Variablen x_1 und x_2 zu tun, so kann man LP-Probleme in Standardform mit der sogenannten *Grafischen Methode* lösen. Da es in der Praxis allerdings nicht nur um zwei, sondern meist um viele Tausend Variablen geht, wollen wir nur kurz anhand von Beispielen darauf eingehen.

Gegeben: $a_1, a_2, b \in \mathbb{R}$. Sind a_1 und a_2 nicht beide Null, so wird durch die Gleichung

$$a_1 x_1 + a_2 x_2 = b \tag{1.8}$$

bekanntlich eine *Gerade* im $\mathbb{R}^2$ dargestellt. Die Menge aller Paare $(x_1, x_2) \in \mathbb{R}^2$, für die

$$a_1 x_1 + a_2 x_2 \leq b$$

gilt, bilden eine *Halbebene*, die durch die Gerade (1.8) begrenzt wird.

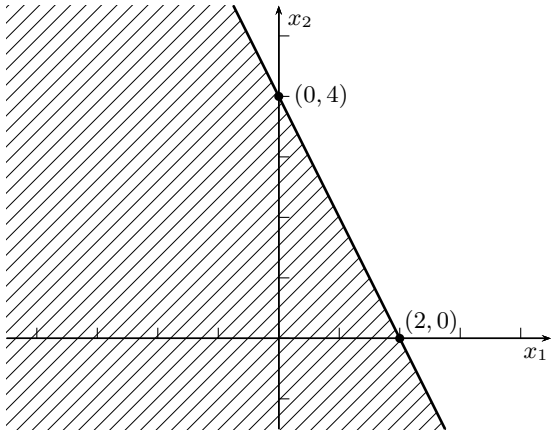
Beispielsweise wird durch die Gleichung

$$2x_1 + x_2 = 4$$

die Gerade durch die Punkte $(0, 4)$ und $(2, 0)$ beschrieben; die Menge aller Punkte (x_1, x_2), für die

$$2x_1 + x_2 \leq 4$$

gilt, bilden die durch Schraffur gekennzeichnete Halbebene:

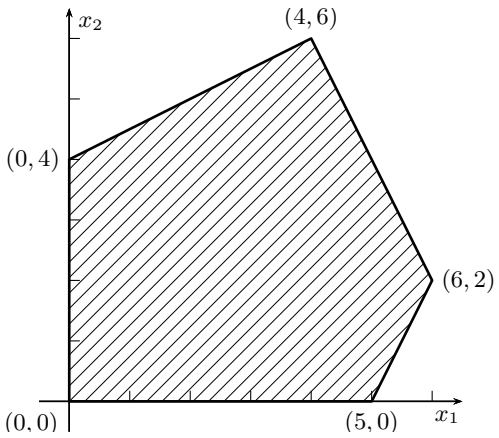


Hat man es nun mit zwei oder mehr Ungleichungen zu tun, so besteht die Menge aller Punkte (x_1, x_2), die alle diese Ungleichungen erfüllen, aus dem *Durchschnitt der entsprechenden Halbebenen.*

Betrachten wir beispielsweise das LP-Problem

$$\begin{aligned} &\text{maximiere} \quad x_1 + x_2 \\ &\text{unter den Nebenbedingungen} \\ &\qquad -x_1 + 2x_2 \leq 8 \\ &\qquad 2x_1 - x_2 \leq 10 \\ &\qquad 2x_1 + x_2 \leq 14 \\ &\qquad x_1, x_2 \geq 0, \end{aligned} \tag{1.9}$$

so haben wir es bei den Nebenbedingungen mit 5 Ungleichungen zu tun. Die Menge aller (x_1, x_2), die all diese Ungleichungen erfüllen, besteht also aus dem Durchschnitt von fünf Halbebenen:



Wir haben somit den zulässigen Bereich, d.h. die Menge der zulässigen Lösungen von (1.9), grafisch dargestellt. Es soll noch einmal genauer beschrieben werden, wie diese grafische Darstellung zustande kommt. Zu diesem Zweck betrachten wir zunächst einmal nur die erste Nebenbedingung; diese lautet

$$-x_1 + 2x_2 \leq 8. \tag{1.10}$$

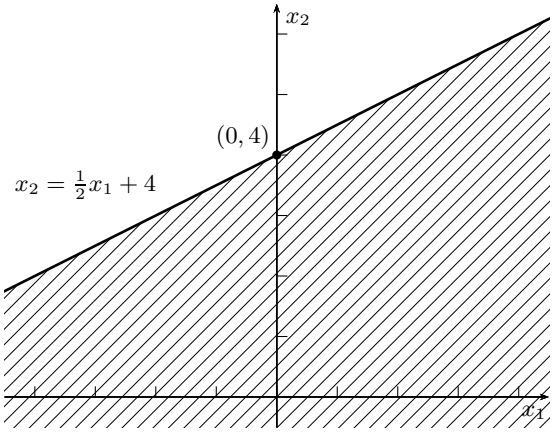
Wir formen die Ungleichung (1.10) äquivalent um, so dass auf der linken Seite nur noch die Variable x_2 steht; man erhält

$$x_2 \leq \frac{1}{2}x_1 + 4. \tag{1.11}$$

Durch (1.10) bzw. (1.11) wird eine *Halbebene* beschrieben; anhand von (1.11) erkennt man, dass diese Halbebene durch die *Gerade*

$$x_2 = \frac{1}{2}x_1 + 4 \tag{1.12}$$

berandet wird und dass die Punkte der Halbebene *unterhalb bzw. auf* der Geraden (1.12) liegen. Wir setzen das Gesagte um, indem wir die Gerade (1.12) darstellen und die Halbebene durch Schraffur andeuten:



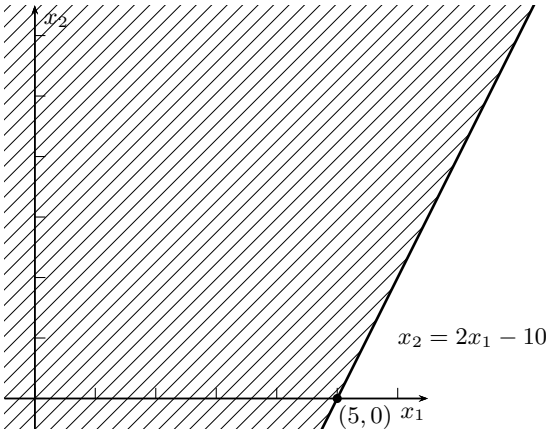
Im nächsten Schritt betrachten wir nun die zweite Nebenbedingung:

$$2x_1 - x_2 \leq 10. \tag{1.13}$$

Wir formen auch diese Ungleichung äquivalent um, so dass nur x_2 auf der linken Seite steht; man erhält

$$x_2 \geq 2x_1 - 10. \tag{1.14}$$

Diesmal gelangen wir zur folgenden Zeichnung:



Man beachte: Aufgrund der Ungleichung (1.14) liegt die von der Geraden $x_2 = 2x_1 - 10$ berandete Halbebene diesmal *oberhalb bzw. auf* dieser Geraden.

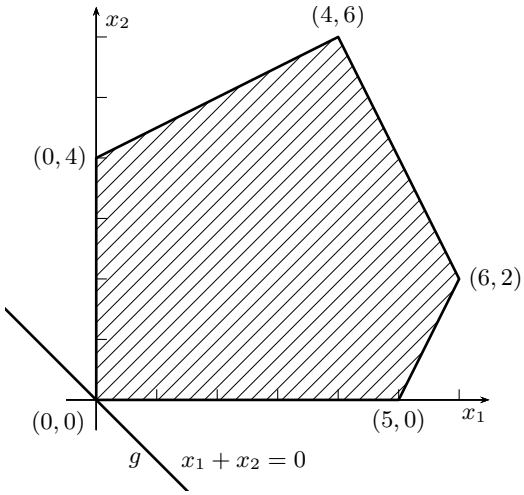
In ähnlicher Weise führen auch die übrigen Nebenbedingungen $2x_1 + x_2 \leq 14$, $x_1 \geq 0$ und $x_2 \geq 0$ zu Geraden und dazugehörigen Halbebenen.

Fasst man alles in einer einzigen Zeichnung zusammen, *wobei man auch einige Schnittpunkte von Geraden zu berechnen hat*, so gelangt man zur obigen Zeichnung, die den zulässigen Bereich von (1.9) darstellt.

Nun betrachten wir zusätzlich die Zielfunktion. Genauer: Wir betrachten diejenigen Punkte (x_1, x_2), für die die Zielfunktion den Wert 0 annimmt, für die also gilt:

$$x_1 + x_2 = 0.$$

Diese Punkte bilden eine Gerade g, die mit dem zulässigen Bereich von (1.9) zumindest einen Punkt gemeinsam hat. Wir sagen, dass die Gerade g den zulässigen Bereich *trifft* oder *schneidet*. Wir nehmen die Gerade g in unsere Zeichnung auf:



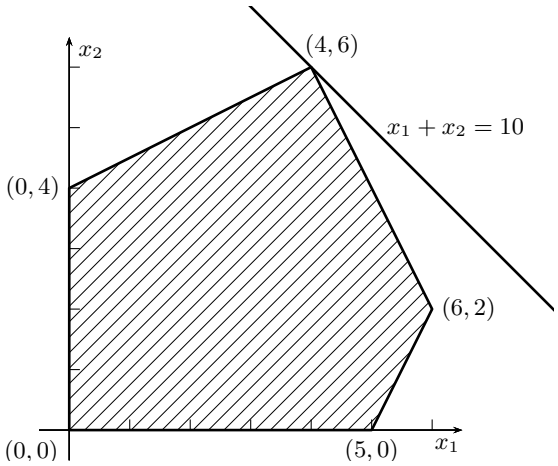
Betrachten wir für $d \in \mathbb{R}$ die Gleichung

$$x_1 + x_2 = d,$$

so wird hierdurch eine zu g parallele Gerade beschrieben. Wählen wir beispielsweise $d = 3$, so schneidet diese Gerade den zulässigen Bereich; auch wenn wir d etwas größer wählen, sagen wir $d = 4$ oder

$d = 6.5$, so trifft die dazugehörige Gerade immer noch den zulässigen Bereich. Nach diesen Beobachtungen sollte das weitere Vorgehen klar sein: Man verschiebt die Gerade g parallel und achtet darauf, dass die entstehende Gerade $x_1 + x_2 = d$ die Menge der zulässigen Lösungen immer noch schneidet, und bemüht sich außerdem, d möglichst groß werden zu lassen.

In unserem Beispiel ergibt sich, dass $d = 10$ der größtmögliche Wert ist und dass der Maximalwert $d = 10$ im Eckpunkt $(4, 6)$ angenommen wird; siehe nachfolgende Zeichnung:



Wir fügen noch eine Bemerkung zur Richtung an, in der die Gerade g verschoben wird. *Diese Richtung kann man leicht an der Zielfunktion ablesen*: In unserem Beispiel lautet die Zielfunktion $c_1x_1 + c_2x_2$ (mit $c_1 = c_2 = 1$) und die Verschiebung der Geraden g erfolgte senkrecht zu g in Richtung des Vektors $(1, 1)$. Allgemein gilt Folgendes, wobei wir voraussetzen, dass c_1, c_2 nicht beide gleich Null sind.

Feststellung.

Liegt ein Maximierungsproblem vor und lautet die Zielfunktion $c_1x_1 + c_2x_2$, so wird bei Ausführung der grafischen Methode eine Gerade der Form $c_1x_1 + c_2x_2 = d$, die den zulässigen Bereich trifft, in Richtung des Vektors $c^T = (c_1, c_2)$ parallel verschoben.

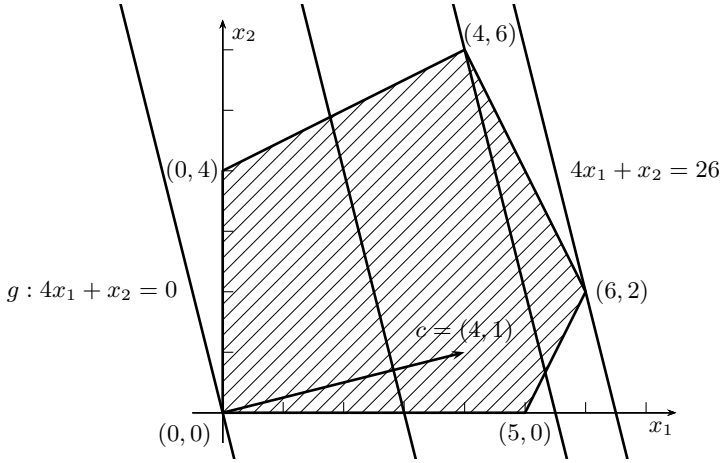
Im Zusammenhang mit dieser Feststellung sei darauf hingewiesen, *dass der Vektor* $c^T = (c_1, c_2)$ *senkrecht auf der Ursprungsgeraden steht, die durch* $c_1x_1 + c_2x_2 = 0$ *gegeben ist.*

Wir illustrieren das Gesagte anhand des folgenden Beispiels:

$$\begin{aligned} &\text{maximiere } 4x_1 + x_2 \\ &\text{unter den Nebenbedingungen} \\ &\qquad -x_1 + 2x_2 \leq 8 \\ &\qquad 2x_1 - x_2 \leq 10 \\ &\qquad 2x_1 + x_2 \leq 14 \\ &\qquad x_1, x_2 \geq 0, \end{aligned} \tag{1.15}$$

Der zulässige Bereich ist in diesem Beispiel derselbe wie in (1.9), nur die Zielfunktion hat sich geändert.

Für Beispiel (1.15) gelangt man zur folgenden Zeichnung:



Ergebnis: Die Gerade $4x_1 + x_2 = 0$ wird in Richtung des Vektors $c = (4, 1)$ verschoben und das Maximum wird im Punkt $(x_1, x_2) = (6, 2)$ angenommen (mit Zielfunktionswert $z = 26$).

Frage: Wie lautet das Ergebnis, wenn in (1.15) anstelle von $4x_1 + x_2$ die Zielfunktion $2x_1 + x_2$ betrachtet wird?

Weitere Beispiele:

1. Was ergibt die grafische Methode im folgenden Beispiel?

$$\begin{aligned} &\text{maximiere} && x_1 + x_2 \\ &\text{unter den Nebenbedingungen} \\ & && -x_1 + x_2 \leq 3 \\ & && x_1 - 2x_2 \leq 2 \\ & && x_1, x_2 \geq 0 \end{aligned}$$

2. Was ergibt sich in 1., wenn man anstelle der Zielfunktion $x_1 + x_2$ die folgende Zielfunktion wählt?

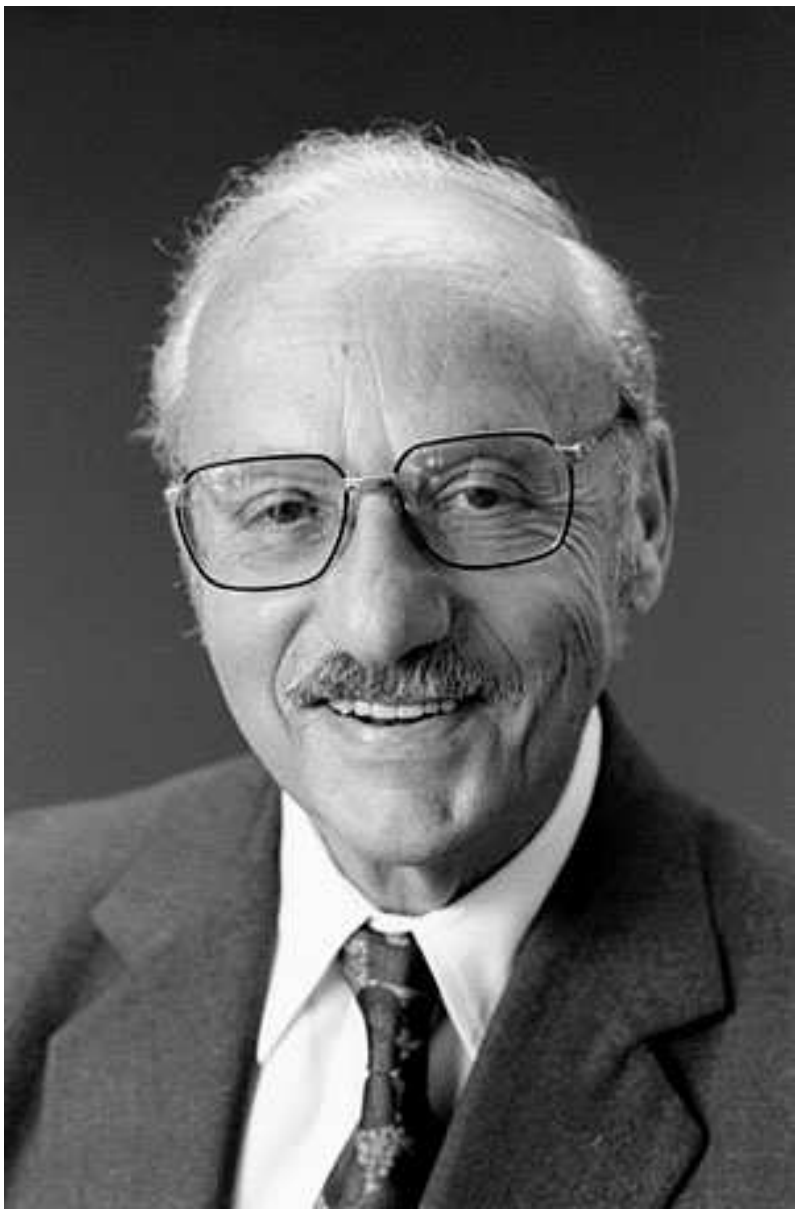
$$-2x_1 + x_2$$

3. Was ergibt sich im folgenden Beispiel?

$$\begin{aligned} &\text{maximiere} && 3x_1 - x_2 \\ &\text{unter den Nebenbedingungen} \\ & && x_1 + x_2 \leq 2 \\ & && -x_1 + x_2 \leq -3 \\ & && x_1, x_2 \geq 0 \end{aligned}$$

Im nächsten Abschnitt werden wir nun die Methode kennenlernen, die in der Praxis die wichtigste Methode zum Lösen von LP-Problemen ist: das *Simplexverfahren* (auch *Simplexalgorithmus* oder *Simplexmethode* genannt).

Das Simplexverfahren wurde von G. B. Dantzig im Jahre 1947 entwickelt.



George B. Dantzig
Geb. 1914 in Portland, Oregon
Gest. 2005 in Stanford, Kalifornien

2 Wie das Simplexverfahren funktioniert

Wir besprechen anhand des folgenden Beispiels, wie das Simplexverfahren verwendet wird, um LP-Probleme zu lösen, die *in Standardform* vorliegen:

$$\begin{aligned}
&\text{maximiere } 5x_1 + 4x_2 + 3x_3 \\
&\text{unter den Nebenbedingungen} \\
&\qquad \begin{aligned} 2x_1 + 3x_2 + x_3 &\leq 5 \\ 4x_1 + x_2 + 2x_3 &\leq 11 \\ 3x_1 + 4x_2 + 2x_3 &\leq 8 \\ x_1, x_2, x_3 &\geq 0. \end{aligned}
\end{aligned} \tag{2.1}$$

Wir führen sogenannte *Schlupfvariablen* (engl. *slack variables*) ein. Worum es dabei geht, erklären wir anhand der ersten Nebenbedingung:

$$2x_1 + 3x_2 + x_3 \leq 5. \tag{2.2}$$

Ist x_1, x_2, x_3 eine zulässige Lösung des LP-Problems (2.1), so erfüllen die Zahlen x_1, x_2, x_3 insbesondere die Ungleichung (2.2), wobei es möglich ist, dass $2x_1 + 3x_2 + x_3 < 5$ gilt, aber es könnte auch $2x_1 + 3x_2 + x_3 = 5$ gelten. Die Differenz der rechten und der linken Seite von (2.2) bezeichnet man als *Schlupf* (engl. *slack*). Der Schlupf gibt also an, um wie viel die rechte Seite die linke Seite übertrifft. Die Differenz zwischen der rechten und der linken Seite von (2.2) wollen wir x_4 nennen; wir *definieren* also:

$$x_4 = 5 - 2x_1 - 3x_2 - x_3.$$

Mit dieser neuen Bezeichnung können wir die Ungleichung (2.2) kurz und knapp wie folgt ausdrücken:

$$x_4 \geq 0.$$

In ähnlicher Weise definieren wir:

$$\begin{aligned} x_5 &= 11 - 4x_1 - x_2 - 2x_3 \\ x_6 &= 8 - 3x_1 - 4x_2 - 2x_3. \end{aligned}$$

Die neuen Variablen x_4, x_5, x_6 werden *Schlupfvariablen* (bzw. *slack variables*) genannt.

Darüber hinaus ist es üblich, noch eine weitere Variable einzuführen, die man mit z bezeichnet und die den Wert der Zielfunktion angibt; in unserem Beispiel:

$$z = 5x_1 + 4x_2 + 3x_3.$$

Unsere Überlegungen lassen sich wie folgt zusammenfassen: Zu jeder Wahl der Zahlen x_1, x_2, x_3 definieren wir die Zahlen x_4, x_5, x_6 und z, indem wir festlegen:

$$\begin{aligned} x_4 &= 5 - 2x_1 - 3x_2 - x_3 \\ x_5 &= 11 - 4x_1 - x_2 - 2x_3 \\ x_6 &= 8 - 3x_1 - 4x_2 - 2x_3 \\ z &= 5x_1 + 4x_2 + 3x_3. \end{aligned} \tag{2.3}$$

Unter Verwendung der Bezeichnungen x_4, x_5, x_6 und z aus (2.3) können wir unser LP-Problem (2.1) auch wie folgt formulieren:

$$\begin{aligned} &\text{maximiere } z \\ &\text{unter den Nebenbedingungen} \\ &\qquad x_1, x_2, x_3, x_4, x_5, x_6 \ \geq 0. \end{aligned} \tag{2.4}$$

Wir betonen noch einmal: Bei (2.4) handelt es sich nur um eine *Umformulierung* von (2.1), wobei die Bezeichnungen x_4, x_5, x_6 und z aus (2.3) verwendet werden. Es gilt:

(i) Jede zulässige Lösung x_1, x_2, x_3 von (2.1) kann auf eindeutige Art zu einer zulässigen Lösung von (2.4) erweitert werden, indem man x_1, x_2 und x_3 in (2.3) einsetzt.

(ii) Ist umgekehrt $x_1, x_2, x_3, x_4, x_5, x_6$ eine zulässige Lösung von (2.4), wobei x_4, x_5 und x_6 die in (2.3) angegebene Bedeutung haben, so kann man auf eine sehr einfache Art eine zulässige Lösung x_1, x_2, x_3 von (2.1) erhalten: Man braucht die Schlupfvariablen x_4, x_5, x_6 nur wegzulassen.

Aufgrund von (i) und (ii) entsprechen sich die zulässigen Lösungen von (2.1) und (2.4) also umkehrbar eindeutig, wobei auch jeder optimalen Lösung von (2.1) eine optimale Lösung von (2.4) entspricht; und umgekehrt. Man beachte, dass die Zielfunktion in beiden Fällen dieselbe ist.

Wir führen nun vor, wie man mithilfe des Simplexverfahrens eine optimale Lösung von (2.1) bzw. (2.4) findet. Die **Grundidee** ist einfach: Man versucht zulässige Lösungen schrittweise zu verbessern, wobei man anstrebt, nach endlich vielen Schritten bei einer optimalen Lösung anzukommen.

Mit anderen Worten: Wenn wir eine zulässige Lösung $x_1, \ldots, x_6$ von (2.4) haben, so versuchen wir eine zulässige Lösung $x_1', \ldots, x_6'$ von (2.4) zu finden, für die gilt:

$$5x_1' + 4x_2' + 3x_3' > 5x_1 + 4x_2 + 3x_3.$$

Damit dieser Prozess in Gang kommt, braucht man natürlich eine zulässige **Startlösung**. *In unserem Beispiel ist es leicht, eine solche zu finden*: Wir wählen $x_1 = x_2 = x_3 = 0$.

Mithilfe von (2.3) erhält man dann dazugehörige Werte für x_4, x_5 und x_6:

$$x_4 = 5, \qquad x_5 = 11, \qquad x_6 = 8.$$

Unsere Startlösung lautet also:

$$x_1 = 0, \qquad x_2 = 0, \qquad x_3 = 0, \qquad x_4 = 5, \qquad x_5 = 11, \qquad x_6 = 8. \tag{2.5}$$

Hierbei handelt es sich in der Tat um eine zulässige Lösung von (2.4), da $x_1, \ldots, x_6$ so gewählt wurden, dass (2.3) gilt, und da außerdem die Bedingungen $x_1, \ldots, x_6 \geq 0$ erfüllt sind.

Den zu dieser Lösung dazugehörigen *Zielfunktionswert* z erhalten wir ebenso, wie wir x_4, x_5, x_6 erhalten haben: Wir setzen in (2.3) für x_1, x_2, x_3 den Wert 0 sein. Man erhält

$$z = 0.$$

Dies gilt es nun zu verbessern, indem wir eine zulässige Lösung mit einem höheren Zielfunktionswert z finden. **Das ist nicht schwer**: Lassen wir beispielsweise die Werte für x_2 und x_3 unverändert bei $x_2 = x_3 = 0$ und vergrößern x_1, so erhalten wir

$$z = 5x_1 > 0.$$

Beispielsweise könnten wir $x_2 = x_3 = 0$ und $x_1 = 1$ wählen und erhielten die zulässige Lösung

$$x_1 = 1, \qquad x_2 = 0, \qquad x_3 = 0, \qquad x_4 = 3, \qquad x_5 = 7, \qquad x_6 = 5 \quad \text{mit} \quad z = 5.$$

Wählen wir $x_1 = 2$ und nach wie vor $x_2 = x_3 = 0$, so erhalten wir einen noch besseren Zielfunktionswert: $z = 10$. Außerdem gilt $x_4 = 1$, $x_5 = 3$, $x_6 = 2$.

Versuchen wir dasselbe mit $x_1 = 3$ und $x_2 = x_3 = 0$, so erhalten wir $z = 15$, was ein noch besserer Zielfunktionswert wäre. Aber gleichzeitig erhalten wir auch $x_4 = -1$. *Das bedeutet, dass wir den Bereich der zulässigen Lösungen verlassen haben*: Es muss ja immer $x_4 \geq 0$, $x_5 \geq 0$ und $x_6 \geq 0$ gelten. *Wir haben x_1 also zu stark erhöht.*

Frage: Um wie viel können wir x_1 maximal erhöhen (unter Beibehaltung von $x_2 = x_3 = 0$), ohne dass einer der Werte x_4, x_5, x_6 negativ wird?

Die Antwort auf diese Frage lässt sich leicht an den ersten drei Zeilen von (2.3) ablesen: Da $x_2 = x_3 = 0$ gelten soll, haben wir

$$\begin{aligned} x_4 &= 5 - 2x_1 \\ x_5 &= 11 - 4x_1 \\ x_6 &= 8 - 3x_1. \end{aligned}$$

Die Bedingung $x_4 \geq 0$ ist also gleichbedeutend mit $5 - 2x_1 \geq 0$, d.h., als Bedingung für x_1 erhalten wir $x_1 \leq \frac{5}{2}$. Entsprechend erhält man aus $x_5 \geq 0$ die Bedingung $x_1 \leq \frac{11}{4}$ und $x_6 \geq 0$ führt zu $x_1 \leq \frac{8}{3}$. Die erste Bedingung schränkt x_1 am stärksten ein; wir wählen also $x_1 = \frac{5}{2}$ und erhalten somit die zulässige Lösung

$$x_1 = \frac{5}{2}, \qquad x_2 = 0, \qquad x_3 = 0, \qquad x_4 = 0, \qquad x_5 = 1, \qquad x_6 = \frac{1}{2}. \tag{2.6}$$

Als verbesserten Zielfunktionswert erhalten wir $z = \frac{25}{2}$.

Wir fassen das bisher Erreichte zusammen: *Unsere Startlösung lautete*

$$x_1 = 0, \qquad x_2 = 0, \qquad x_3 = 0, \qquad x_4 = 5, \qquad x_5 = 11, \qquad x_6 = 8 \quad \text{mit} \quad z = 0,$$

und in der 1. Iteration haben wir die verbesserte zulässige Lösung (2.6) mit $z = \frac{25}{2}$ erhalten.

Dies gilt es nun weiter zu verbessern. Die entscheidende Rolle in der 1. Iteration spielte das Gleichungssystem (2.3): *Dort wurden die vier Variablen x_4, x_5, x_6 und z durch die drei übrigen Variablen (nämlich durch x_1, x_2, x_3) dargestellt und diese drei Variablen besaßen alle den Wert Null in unserer Startlösung.*

Diesen Zustand wollen wir nun (bezogen auf die verbesserte Lösung (2.6)) wiederherstellen. In (2.6) sind x_2, x_3 und x_4 diejenigen Variablen, die den Wert Null annehmen. Diese Variablen sollen nun die Rolle spielen, die zuvor von x_1, x_2 und x_3 gespielt wurde. Hierzu formen wir (2.3) so um, dass auf der linken Seite nun x_1, x_5, x_6 und z stehen, während rechts nur noch die Variablen x_2, x_3 und x_4 auftauchen.

Anders gesagt: *x_1 und x_4 sollen ihre Rollen tauschen; x_1 soll von rechts nach links wandern; x_4 umgekehrt von links nach rechts.*

Hierzu formen wir zunächst diejenige Zeile von (2.3) um, in der x_4 auf der linken Seite steht. In (2.3) ist das die erste Zeile; wir erhalten

$$x_1 = \frac{5}{2} - \frac{3}{2}x_2 - \frac{1}{2}x_3 - \frac{1}{2}x_4. \tag{2.7}$$

Einsetzen von (2.7) in die übrigen Zeilen von (2.3) ergibt:

$$\begin{aligned} x_5 &= 11 - 4 \cdot \left(\frac{5}{2} - \frac{3}{2}x_2 - \frac{1}{2}x_3 - \frac{1}{2}x_4\right) - x_2 - 2x_3 \\ &= 1 + 5x_2 + 2x_4 \\ x_6 &= 8 - 3 \cdot \left(\frac{5}{2} - \frac{3}{2}x_2 - \frac{1}{2}x_3 - \frac{1}{2}x_4\right) - 4x_2 - 2x_3 \\ &= \frac{1}{2} + \frac{1}{2}x_2 - \frac{1}{2}x_3 + \frac{3}{2}x_4 \\ z &= 5 \cdot \left(\frac{5}{2} - \frac{3}{2}x_2 - \frac{1}{2}x_3 - \frac{1}{2}x_4\right) + 4x_2 + 3x_3 \\ &= \frac{25}{2} - \frac{7}{2}x_2 + \frac{1}{2}x_3 - \frac{5}{2}x_4. \end{aligned}$$

Also lautet unser neues, durch Umformung von (2.3) entstandenes Gleichungssystem:

$$\begin{aligned} x_1 &= \frac{5}{2} - \frac{3}{2}x_2 - \frac{1}{2}x_3 - \frac{1}{2}x_4 \\ x_5 &= 1 + 5x_2 + 2x_4 \\ x_6 &= \frac{1}{2} + \frac{1}{2}x_2 - \frac{1}{2}x_3 + \frac{3}{2}x_4 \\ z &= \frac{25}{2} - \frac{7}{2}x_2 + \frac{1}{2}x_3 - \frac{5}{2}x_4. \end{aligned} \tag{2.8}$$

Analog zur 1. Iteration versuchen wir nun, den aktuellen Wert von z zu vergrößern, indem wir den Wert einer der drei Variablen auf der rechten Seite von (2.8) anheben, während wir gleichzeitig die beiden anderen Variablen der rechten Seite bei Null lassen. *Wir haben drei Möglichkeiten zur Auswahl*:

(i) x_2 wird angehoben, $x_3 = x_4 = 0$;

(ii) x_3 wird angehoben, $x_2 = x_4 = 0$;

(iii) x_4 wird angehoben, $x_2 = x_3 = 0$.

Die „Möglichkeiten“ (i) und (iii) scheiden sofort aus: Sie führen zu einer Verkleinerung von z – sehr entgegen unserer Absichten.

Es bleibt nur eine Wahl: Wir setzen $x_2 = x_4 = 0$ und versuchen durch Anheben von x_3 zu einem möglichst großen Zuwachs von z zu gelangen, wobei wir allerdings darauf achten müssen, dass weiterhin $x_1 \geq 0$, $x_5 \geq 0$ und $x_6 \geq 0$ gilt.

Wie stark können wir x_3 anheben? Die Antwort können wir direkt an unserem Gleichungssystem (2.8) ablesen: Wegen $x_2 = x_4 = 0$ ist die Bedingung $x_1 \geq 0$ äquivalent zu $\frac{5}{2} - \frac{1}{2}x_3 \geq 0$, woraus man $x_3 \leq 5$ erhält. Aus $x_6 \geq 0$ erhält man auf ähnliche Art $x_3 \leq 1$, während die Bedingung $x_5 \geq 0$ die Wahl von x_3 nicht beschränkt. **Also**: $x_3 = 1$ *ist das Beste, was wir erreichen können, und unsere neue Lösung ist dementsprechend*:

$$x_1 = 2, \qquad x_2 = 0, \qquad x_3 = 1, \qquad x_4 = 0, \qquad x_5 = 1, \qquad x_6 = 0. \tag{2.9}$$

Wir wissen bereits: *Damit das Verfahren weitergeht, brauchen wir nicht nur eine verbesserte Lösung, sondern auch eine neue Darstellung unseres Gleichungssystems (2.8), die zu (2.9) passt.* In (2.9) gibt es drei Variablen, die den Wert Null annehmen: $x_2 = x_4 = x_6 = 0$. Diese drei Variablen sollen nun auf der rechten Seite stehen, die übrigen Variablen (x_1, x_3, x_5 sowie z) sollen links auftauchen: *Ähnlich wie in der ersten Iteration ist also ein **Austausch** vorzunehmen; diesmal haben x_3 und x_6 die Seiten zu wechseln.* Dementsprechend stellen wir die dritte Gleichung von (2.8) um und erhalten

$$x_3 = 1 + x_2 + 3x_4 - 2x_6.$$

Setzt man dies für x_3 in die übrigen Gleichungen von (2.8) ein, so erhält man

$$\begin{aligned} x_3 &= 1 + x_2 + 3x_4 - 2x_6 \\ x_1 &= 2 - 2x_2 - 2x_4 + x_6 \\ x_5 &= 1 + 5x_2 + 2x_4 \\ z &= 13 - 3x_2 - x_4 - x_6. \end{aligned} \tag{2.10}$$

Einsetzen von $x_2 = x_4 = x_6 = 0$ in die letzten Zeile von (2.10) ergibt $z = 13$.

Während sich in der ersten Iteration eine Steigerung des Zielfunktionswerts von $z = 0$ zu $z = 12.5$ ergeben hat, hat die zweite Iteration nur zu einem bescheidenen Zuwachs geführt: $z = 13$.

Nun sollen Sie sich aber auch daran gewöhnen, dass das Lesen von englischen Lehrbüchern in der Regel sehr einfach ist. Darum gibt es den Rest auf Englisch (Originaltext von Vašek Chvátal).

Now it's time for the third iteration. First of all, from the right-hand side of (2.10) we have to choose a variable whose increase brings about an increase of the objective function. However, there is no such

variable: indeed, if we increase any of the right-hand side variables x_2, x_4, x_6, we will make the value of z *decrease*. Thus, it seems that we have come to a standstill. In fact, the very presence of this standstill indicates that we are done; we have solved our problem; the solution described by the last table is optimal. Why? The answer lies hidden in the last row of (2.10):

$$z = 13 - 3x_2 - x_4 - x_6. \tag{2.11}$$

Our last solution (2.9) yields $z = 13$; proving that this solution is optimal amounts to proving that every feasible solution satisfies the inequality $z \leq 13$. Since every feasible solution $x_1, \ldots, x_6$ satisfies, among other relations, the inequalities $x_2 \geq 0$, $x_4 \geq 0$, and $x_6 \geq 0$, the desired inequality $z \leq 13$ follows directly from (2.11).

Auf Deutsch (etwas frei übersetzt):

Bei der dritten Iteration stecken wir fest. Was nun zunächst wie eine Schwierigkeit aussieht, entpuppt sich als Erfolg: *Die Lösung (2.9) ist optimal, der Zielfunktionswert $z = 13$ ist bestmöglich. Weshalb ist das so?* **Die Antwort findet sich in der letzten Zeile von (2.10)**:

$$z = 13 - 3x_2 - x_4 - x_6. \tag{2.11}$$

Unsere letzte Lösung (2.9) hat zu $z = 13$ geführt. *Wir wollen uns davon überzeugen, dass für jede zulässige Lösung $z \leq 13$ gilt.* Aufgrund von (2.11) ist dies aber klar: Ist $x_1, x_2, x_3, x_4, x_5, x_6$ eine beliebige zulässige Lösung, so gilt insbesondere $x_2 \geq 0$, $x_4 \geq 0$ und $x_6 \geq 0$, woraus sich (aufgrund von (2.11)) $z \leq 13$ ergibt.

Nachdem wir das Simplexverfahren anhand eines Beispiels studiert haben, schauen wir uns nun den allgemeinen Fall an. Gegeben sei ein LP-Problem in Standardform.

LP-Problem in Standardform.

$$\begin{aligned} &\text{maximiere } \sum_{j=1}^{n} c_j x_j \\ &\text{unter den Nebenbedingungen} \\ &\qquad \sum_{j=1}^{n} a_{ij} x_j \leq b_i \qquad (i = 1, \ldots, m) \\ &\qquad\qquad x_j \geq 0 \qquad (j = 1, \ldots, n). \end{aligned} \tag{2.12}$$

Wollen wir das LP-Problem (2.12) mit dem Simplexverfahren lösen, so führen wir zunächst *Schlupfvariablen* $x_{n+1}, \ldots, x_{n+m}$ sowie eine Variable z ein, die den Wert der *Zielfunktion* angibt. *Mit anderen Worten: Wir definieren*

$$\begin{aligned} x_{n+i} &= b_i - \sum_{j=1}^{n} a_{ij} x_j \qquad (i = 1, \ldots, m) \\ z &= \sum_{j=1}^{n} c_j x_j. \end{aligned} \tag{2.13}$$

Unter Verwendung der Bezeichnungen aus (2.13) kann man das LP-Problem (2.12) auch wie folgt schreiben:

$$\begin{aligned} &\text{maximiere } z \\ &\text{unter den Nebenbedingungen} \\ &\qquad x_1, \ldots, x_{n+m} \geq 0. \end{aligned} \tag{2.12'}$$

Die folgende häufig verwendete Möglichkeit, das LP-Problem (2.12) zu formulieren, ergibt sich direkt aus der Definition der Schlupfvariablen:

$$\begin{aligned}
&\text{maximiere } \sum_{j=1}^{n} c_j x_j \\
&\text{unter den Nebenbedingungen} \\
&\qquad \sum_{j=1}^{n} a_{ij} x_j + x_{n+i} = b_i \qquad (i = 1, \ldots, m) \\
&\qquad\qquad x_j \geq 0 \qquad (j = 1, \ldots, n+m).
\end{aligned} \tag{2.12"}$$

Im Verlauf des Simplexverfahrens ersetzt man in jeder Iteration eine zulässige Lösung $x_1, \ldots, x_{n+m}$ von (2.12") durch eine zulässige Lösung $\overline{x}_1, \ldots, \overline{x}_{n+m}$; dabei strebt man an, dass die neue Lösung besser als die alte ist, d.h., man möchte erhalten, dass

$$\sum_{j=1}^{n} c_j \overline{x}_j > \sum_{j=1}^{n} c_j x_j$$

gilt. In unserem obigen Beispiel wurde dies in jeder Iteration erreicht; *wir werden jedoch (später) noch sehen, dass es auch nötig sein kann, Iterationen zuzulassen, in denen* $\sum_{j=1}^{n} c_j \overline{x}_j = \sum_{j=1}^{n} c_j x_j$ *gilt.*

In unserem obigen Beispiel haben wir gesehen, dass in jeder Iteration nicht nur eine zulässige Lösung $x_1, \ldots, x_{n+m}$ ermittelt wird, sondern dass in jeder Iteration auch ein lineares Gleichungssystem mit $m+1$ Gleichungen vorkommt. In unserem Beispiel waren dies die Gleichungssysteme (2.3), (2.8) und (2.10). Die Variablen dieser Gleichungssysteme waren $x_1, \ldots, x_6$ und z, *und diese Gleichungssysteme hatten eine besondere Form*: Links traten immer drei der Variablen $x_1, \ldots, x_6$ auf sowie (in der letzten Zeile) z, während rechts immer nur die drei übrigen der Variablen $x_1, \ldots, x_6$ vorkamen.

Außerdem sind (wie man sich unschwer überlegt) die drei Gleichungssysteme (2.3), (2.8) und (2.10) äquivalent in dem Sinne, dass sie *dieselbe Lösungsmenge* besitzen. Anders gesagt: Für jede Wahl der Zahlen $x_1, \ldots, x_6$ und z sind die folgenden drei Aussagen äquivalent:

- $x_1, \ldots, x_6$ und z bilden eine Lösung von (2.3);
- $x_1, \ldots, x_6$ und z bilden eine Lösung von (2.8);
- $x_1, \ldots, x_6$ und z bilden eine Lösung von (2.10).

Für Gleichungssysteme wie (2.3), (2.8) und (2.10) gibt es **unterschiedliche Bezeichnungen**:

- Chvátal benutzt beispielsweise die Bezeichnung *dictionary*;
- im Buch von Cormen et al wird die Bezeichnung *Schlupfform* benutzt;
- im Buch von Matoušek und Gärtner werden derartige Gleichungssysteme *Tableaus* genannt.

Wir schließen uns der Sprechweise von Matoušek und Gärtner an und verwenden ebenfalls den Begriff *Tableau* als unsere Bezeichnung für Gleichungssysteme wie (2.3), (2.8) und (2.10). Wenn wir von einem Tableau sprechen, so ist damit also kein „Zahlenschema“ oder „Koeffizientenschema“ gemeint, sondern ein lineares Gleichungssystem einer bestimmten Art. Diejenigen Variablen x_j, die in einem Tableau auf der linken Seite stehen, nennt man *Basisvariablen*, die übrigen Variablen x_j, also diejenigen, die auf der rechten Seite stehen, nennt man *Nichtbasisvariablen*. Die Menge der Basisvariablen nennt man eine *Basis*; wir bezeichnen die Menge der zu den Basisvariablen x_j gehörenden Indizes j mit B; die Menge der Indizes j, die bei den Nichtbasisvariablen vorkommen, bezeichnen wir mit N.

Im Verlauf des Simplexalgorithmus ändern sich B und N; wir erläutern dies anhand unseres Beispiels.

Für das Tableau (2.3) gilt $B = \{4, 5, 6\}$ und $N = \{1, 2, 3\}$. Beim Übergang von (2.3) zum Tableau (2.8) verlässt x_4 die Basis und x_1 wird neu in die Basis aufgenommen; für das Tableau (2.8) gilt also $B = \{1, 5, 6\}$ und $N = \{2, 3, 4\}$.

Einen Übergang von einem Tableau zum nächsten (wie von (2.3) zu (2.8)) nennt man *Pivotschritt*, *Basisaustauschschritt* oder *Basistausch*; diejenige Variable, die neu in die Basis aufgenommen wird, heißt *Eingangsvariable*; die Variable, die die Basis verlässt, heißt *Ausgangsvariable*. Diejenige Zeile, in der vor dem Basistausch links die Ausgangsvariable steht, heißt *Pivotzeile*. Diejenige Spalte, in der vor dem Basistausch die Eingangsvariable steht, heißt *Pivotspalte*.

Beim Übergang von (2.8) zu (2.10) ist x_3 die Eingangsvariable und x_6 ist die Ausgangsvariable. Die Zeile von (2.8), in der x_6 steht, ist die Pivotzeile dieses Basistauschs; die Spalte von (2.8), in der x_3 steht, ist die Pivotspalte.

Wir kehren zurück zum allgemeinen Fall (2.12). Unter einem zu (2.12) gehörigen *Tableau* (engl. Bezeichnung im Buch von Chvátal: *dictionary*) verstehen wir ein System von $m+1$ linearen Gleichungen mit den Variablen $x_1, \ldots, x_{n+m}$ und z sowie mit den folgenden Eigenschaften:

(i) Jede Lösung dieses Gleichungssystems ist eine Lösung von (2.13); und umgekehrt.

(ii) Die Gleichungen sind nach m der Variablen $x_1, \ldots, x_{n+m}$ (genannt *Basisvariablen*) und nach z aufgelöst; die übrigen n Variablen heißen *Nichtbasisvariablen*. Jede der Basisvariablen x_j sowie z ist im Gleichungssystem als Summe aus einer Konstanten und einer Linearkombination der Nichtbasisvariablen dargestellt.

Etwas vereinfacht kann man (ii) auch so aussprechen: *Jede Basisvariable x_j sowie z wird durch die Nichtbasisvariablen ausgedrückt.*

Die Eigenschaften (i) und (ii) definieren, was man unter einem *Tableau* (engl. *dictionary*) versteht. Die Tableaus (2.3), (2.8) und (2.10) besaßen darüber hinaus noch die folgende Eigenschaft:

(iii) Werden auf der rechten Seite alle Variablen gleich Null gesetzt, so erhält man eine zulässige Lösung. (Mit anderen Worten: Setzt man alle Nichtbasisvariablen gleich Null, so wird keine der Basisvariablen negativ.)

Tableaus mit dieser zusätzlichen Eigenschaft werden *zulässige Tableaus* (engl. *feasible dictionaries*) genannt.

Jedes zulässige Tableau beschreibt also eine zulässige Lösung von (2.12), die man erhält, wenn man alle Nichtbasisvariablen gleich Null setzt. Aber nicht jede zulässige Lösung entsteht auf diese Art aus einem zulässigen Tableau; beispielsweise ist

$$x_1 = 1, \quad x_2 = 0, \quad x_3 = 1, \quad x_4 = 2, \quad x_5 = 5, \quad x_6 = 3$$

eine zulässige Lösung von (2.1), die jedoch nicht von einem zulässigen Tableau auf die beschriebene Weise abstammt.

Zulässige Lösungen, die durch ein Tableau beschrieben werden (d.h., die aus einem Tableau dadurch entstehen, dass man alle Nichtbasisvariablen auf Null setzt), heißen *zulässige Basislösungen* (engl. *basic feasible solutions*).

Eine auffallende Eigenschaft des Simplexalgorithmus ist, dass er nur mit zulässigen Basislösungen arbeitet und alle anderen zulässigen Lösungen ignoriert.

Nun ist es Zeit für ein weiteres **Beispiel**, das aus Chvátal: *Linear Programming* entnommen wurde.

Second Example

We shall complete our preview of the simplex method by applying it to another LP problem:

$$\begin{aligned}
&\text{maximize} && 5x_1 + 5x_2 + 3x_3 \\
&\text{subject to} \\
&&& x_1 + 3x_2 + x_3 \leq 3 \\
&&& -x_1 + 3x_3 \leq 2 \\
&&& 2x_1 - x_2 + 2x_3 \leq 4 \\
&&& 2x_1 + 3x_2 - x_3 \leq 2 \\
&&& x_1, x_2, x_3 \geq 0.
\end{aligned}$$

In this case, the initial feasible dictionary reads

$$\begin{aligned} x_4 &= 3 - x_1 - 3x_2 - x_3 \\ x_5 &= 2 + x_1 - 3x_3 \\ x_6 &= 4 - 2x_1 + x_2 - 2x_3 \\ x_7 &= 2 - 2x_1 - 3x_2 + x_3 \\ \hline z &= 5x_1 + 5x_2 + 3x_3. \end{aligned} \tag{2.14}$$

(Even though the order of the equations in a dictionary is quite irrelevant, we shall make a habit of writing the formula for z last and separating it from the rest of the table by a solid line. Of course, that does *not* mean that the last equation is the sum of the previous ones.) This feasible dictionary describes the feasible solution

$$x_1 = 0, \quad x_2 = 0, \quad x_3 = 0, \quad x_4 = 3, \quad x_5 = 2, \quad x_6 = 4, \quad x_7 = 2.$$

However, there is no need to write this solution down, as we just did: the solution is implicit in the dictionary.

In the first iteration, we shall attempt to increase the value of z by making one of the right-hand side variables positive. At this moment, any of the three variables x_1, x_2, x_3 would do. In small examples, it is common practice to choose the variable that, in the formula for z, has the largest coefficient: the increase in that variable will make z increase at the fastest rate (but not necessarily to the highest level). In our case, this rule leaves us a choice between x_1 and x_2; choosing arbitrarily, we decide to make x_1 positive. As the value of x_1 increases, so does the value of x_5. However, the values of x_4, x_6, and x_7 decrease, and none of them is allowed to become negative. Of the three constraints $x_4 \geq 0$, $x_6 \geq 0$, $x_7 \geq 0$ that impose upper bounds on the increment of x_1 the last constraint $x_7 \geq 0$ is the most stringent: it implies $x_1 \leq 1$. In the improved feasible solution, we shall have $x_1 = 1$ and $x_7 = 0$. Without writing the new solution down, we shall now construct the new dictionary. All we need to know is that x_1 just made its way from the right-hand side to the left, whereas x_7 went in the opposite direction. From the fourth equation in (2.14), we have

$$x_1 = 1 - \frac{3}{2}x_2 + \frac{1}{2}x_3 - \frac{1}{2}x_7. \tag{2.15}$$

Substituting from (2.15) into the remaining equations of (2.14), we arrive at the desired dictionary

$$\begin{aligned} x_1 &= 1 - \frac{3}{2}x_2 + \frac{1}{2}x_3 - \frac{1}{2}x_7 \\ x_4 &= 2 - \frac{3}{2}x_2 - \frac{3}{2}x_3 + \frac{1}{2}x_7 \\ x_5 &= 3 - \frac{3}{2}x_2 - \frac{5}{2}x_3 - \frac{1}{2}x_7 \\ x_6 &= 2 + 4x_2 - 3x_3 + x_7 \\ \hline z &= 5 - \frac{5}{2}x_2 + \frac{11}{2}x_3 - \frac{5}{2}x_7. \end{aligned} \tag{2.16}$$

The construction of (2.16) completes the first iteration of the simplex method.

In our example, the variable to enter the basis during the second iteration is quite unequivocally x_3. This is the only nonbasic variable in (2.16) whose coefficient in the last row is positive. Of the four basic variables, x_6 imposes the most stringent upper bound on the increase of x_3, and, therefore, has to leave

the basis. Pivoting, we arrive at our third dictionary,

$$\begin{aligned} x_3 &= \frac{2}{3} + \frac{4}{3}x_2 + \frac{1}{3}x_7 - \frac{1}{3}x_6 \\ x_1 &= \frac{4}{3} - \frac{5}{6}x_2 - \frac{1}{3}x_7 - \frac{1}{6}x_6 \\ x_4 &= 1 - \frac{7}{2}x_2 + \frac{1}{2}x_6 \\ x_5 &= \frac{4}{3} - \frac{29}{6}x_2 - \frac{4}{3}x_7 + \frac{5}{6}x_6 \\ \hline z &= \frac{26}{3} + \frac{29}{6}x_2 - \frac{2}{3}x_7 - \frac{11}{6}x_6. \end{aligned} \tag{2.17}$$

In the third iteration, the entering variable is x_2 and the leaving variable is x_5. Pivoting yields the dictionary

$$\begin{aligned} x_2 &= \frac{8}{29} - \frac{8}{29}x_7 + \frac{5}{29}x_6 - \frac{6}{29}x_5 \\ x_3 &= \frac{30}{29} - \frac{1}{29}x_7 - \frac{3}{29}x_6 - \frac{8}{29}x_5 \\ x_1 &= \frac{32}{29} - \frac{3}{29}x_7 - \frac{9}{29}x_6 + \frac{5}{29}x_5 \\ x_4 &= \frac{1}{29} + \frac{28}{29}x_7 - \frac{3}{29}x_6 + \frac{21}{29}x_5 \\ \hline z &= 10 - 2x_7 - x_6 - x_5. \end{aligned} \tag{2.18}$$

At this point, no nonbasic variable can enter the basis without making the value of z decrease. Hence, the last dictionary describes an optimal solution of our example. That solution is

$$x_1 = \frac{32}{29}, \quad x_2 = \frac{8}{29}, \quad x_3 = \frac{30}{29}$$

and it yields $z = 10$.

Das *Ergebnis einer Iteration* ist immer ein *neues Tableau. Am Ende jeder Iteration wird dieses Ergebnis – das neue Tableau also – übersichtlich hingeschrieben*, wobei einige *Konventionen* zu beachten sind, die wir im zweiten Beispiel kennengelernt haben[1]:

1. Die z-Zeile wird **unten** notiert und durch einen Strich abgetrennt.
2. Im neuen Tableau wird die Zeile mit der Eingangsvariablen immer **oben** hingeschrieben.
3. Die Variable, die neu auf der rechten Seite auftaucht – die Ausgangsvariable – schließt immer **rechts** an.
4. Außerdem ist es für die Übersichtlichkeit wichtig, dass im Tableau gleiche Variablen immer **genau untereinander** geschrieben werden.

Wir schreiben das erste Beispiel noch einmal übersichtlich auf. Dieses Beispiel kann als *Muster für das Lösen von Übungsaufgaben* dienen.

[1] Diese Konventionen dienen der Übersichtlichkeit.

Aufgabe: Lösen Sie das folgende LP-Problem mit dem Simplexverfahren:

$$\begin{aligned}
&\text{maximiere } 5x_1 + 4x_2 + 3x_3 \\
&\text{unter den Nebenbedingungen} \\
&\qquad 2x_1 + 3x_2 + x_3 \leq 5 \\
&\qquad 4x_1 + x_2 + 2x_3 \leq 11 \\
&\qquad 3x_1 + 4x_2 + 2x_3 \leq 8 \\
&\qquad x_1, x_2, x_3 \geq 0.
\end{aligned}$$

Lösung.

Starttableau:

$$\begin{aligned}
x_4 &= 5 - 2x_1 - 3x_2 - x_3 \\
x_5 &= 11 - 4x_1 - x_2 - 2x_3 \\
x_6 &= 8 - 3x_1 - 4x_2 - 2x_3 \\
\hline
z &= 5x_1 + 4x_2 + 3x_3.
\end{aligned}$$

1. Iteration:

Eingangsvariable: x_1
Ausgangsvariable: x_4

Es folgt

$$\begin{aligned}
x_1 &= \frac{5}{2} - \frac{3}{2}x_2 - \frac{1}{2}x_3 - \frac{1}{2}x_4 \\
x_5 &= 11 - 4 \cdot \left(\frac{5}{2} - \frac{3}{2}x_2 - \frac{1}{2}x_3 - \frac{1}{2}x_4\right) - x_2 - 2x_3 \\
&= 1 + 5x_2 + 2x_4 \\
x_6 &= 8 - 3 \cdot \left(\frac{5}{2} - \frac{3}{2}x_2 - \frac{1}{2}x_3 - \frac{1}{2}x_4\right) - 4x_2 - 2x_3 \\
&= \frac{1}{2} + \frac{1}{2}x_2 - \frac{1}{2}x_3 + \frac{3}{2}x_4 \\
z &= 5 \cdot \left(\frac{5}{2} - \frac{3}{2}x_2 - \frac{1}{2}x_3 - \frac{1}{2}x_4\right) + 4x_2 + 3x_3 \\
&= \frac{25}{2} - \frac{7}{2}x_2 + \frac{1}{2}x_3 - \frac{5}{2}x_4.
\end{aligned}$$

Ergebnis der 1. Iteration:

$$\begin{aligned}
x_1 &= \frac{5}{2} - \frac{3}{2}x_2 - \frac{1}{2}x_3 - \frac{1}{2}x_4 \\
x_5 &= 1 + 5x_2 + 2x_4 \\
x_6 &= \frac{1}{2} + \frac{1}{2}x_2 - \frac{1}{2}x_3 + \frac{3}{2}x_4 \\
\hline
z &= \frac{25}{2} - \frac{7}{2}x_2 + \frac{1}{2}x_3 - \frac{5}{2}x_4.
\end{aligned}$$

2. Iteration:

Eingangsvariable: x_3
Ausgangsvariable: x_6

Es folgt

$$\begin{aligned} x_3 &= 1 + x_2 + 3x_4 - 2x_6 \\ x_1 &= \frac{5}{2} - \frac{3}{2}x_2 - \frac{1}{2}\Big(1 + x_2 + 3x_4 - 2x_6\Big) - \frac{1}{2}x_4 \\ &= 2 - 2x_2 - 2x_4 + x_6 \\ z &= \frac{25}{2} - \frac{7}{2}x_2 + \frac{1}{2}\Big(1 + x_2 + 3x_4 - 2x_6\Big) - \frac{5}{2}x_4 \\ &= 13 - 3x_2 - x_4 - x_6. \end{aligned}$$

Ergebnis der 2. Iteration:

$$\begin{aligned} x_3 &= 1 + x_2 + 3x_4 - 2x_6 \\ x_1 &= 2 - 2x_2 - 2x_4 + x_6 \\ x_5 &= 1 + 5x_2 + 2x_4 \\ \hline z &= 13 - 3x_2 - x_4 - x_6. \end{aligned}$$

Dieses Tableau liefert die optimale Lösung $x_1 = 2$, $x_2 = 0$, $x_3 = 1$ mit $z = 13$.

Anstelle von „Ergebnis der k-ten Iteration“ kann man jedes Mal auch kurz und knapp „Neues Tableau“ schreiben.

Hinweis: Das *Ergebnis einer Iteration* ist natürlich auch immer eine **neue zulässige Basislösung**. Die neue zulässige Basislösung braucht am Ende einer Iteration aber nicht unbedingt hingeschrieben zu werden, da sie implizit im neuen Tableau enthalten und sehr leicht ablesbar ist.

Der Deutlichkeit halber geben wir für das erste Beispiel die Folge der zulässigen Basislösungen noch einmal explizit an.

Startlösung („zulässige Basislösung am Anfang“):

$$x_1 = 0, \quad x_2 = 0, \quad x_3 = 0, \quad x_4 = 5, \quad x_5 = 11, \quad x_6 = 8 \quad \text{mit} \quad z = 0.$$

Zulässige Basislösung nach der 1. Iteration:

$$x_1 = \frac{5}{2}, \quad x_2 = 0, \quad x_3 = 0, \quad x_4 = 0, \quad x_5 = 1, \quad x_6 = \frac{1}{2} \quad \text{mit} \quad z = 12.5.$$

Zulässige Basislösung nach der 2. Iteration:

$$x_1 = 2, \quad x_2 = 0, \quad x_3 = 1, \quad x_4 = 0, \quad x_5 = 1, \quad x_6 = 0 \quad \text{mit} \quad z = 13.$$

Abschließend noch eine **Sprechweise**: Im LP-Problem (2.12) hatten wir es ursprünglich mit den Variablen $x_1, \ldots, x_n$ zu tun. Anschließend kamen sofort weitere Variablen $x_{n+1}, \ldots, x_{n+m}$ hinzu, die wir *Schlupfvariablen* genannt haben. Um uns besser ausdrücken zu können, fehlt noch ein Name für die „ursprünglichen Variablen“ $x_1, \ldots, x_n$: Es ist üblich, diese Variablen als *Problemvariablen* oder *Entscheidungsvariablen* (engl. *decision variables*) zu bezeichnen.

3 Schwierigkeiten und Hindernisse – und wie man sie überwindet

Die Beispiele des letzten Kapitels waren absichtlich so gewählt, dass alles glatt ging. Es kann in anderen Beispielen jedoch so sein, dass Schwierigkeiten auftreten. Der Zweck dieses Abschnitts ist, die Simplexmethode genau zu analysieren, die wichtigsten Details unter die Lupe zu nehmen und Hindernisse aus dem Weg zu räumen.

Schwierigkeiten könnte es in allen Phasen des Verfahrens geben:

(i) **Initialisierung**. Unsere bisherigen Beispiele waren so gewählt, dass es niemals schwierig war, ein geeignetes Starttableau zu finden. *Wir werden jedoch sehen, dass es nicht in jedem Fall so leicht ist, sich ein geeignetes Starttableau zu verschaffen.* Außerdem könnte es sein, dass es gar kein zulässiges Starttableau gibt, da das vorliegende LP-Problem unlösbar ist. Frage: *Wie findet man heraus, ob ein unlösbares LP-Problem vorliegt?*

(ii) **Iteration**. Eine der Fragen, die sich hier stellen: Was macht man, wenn im aktuellen Tableau keine geeignete Eingangs- oder Ausgangsvariable zu finden ist? Als Antwort wird sich ergeben, dass man nichts mehr tun braucht: *Falls keine geeignete Eingangsvariable existiert, so ist die vorliegende Lösung optimal; falls eine geeignete Eingangsvariable gefunden werden kann, aber keine dazugehörige Ausgangsvariable existiert, so ist man ebenfalls fertig, da das Problem in diesem Fall unbeschränkt ist.*

(iii) **Terminierung**. Könnte das Verfahren in eine unendliche Schleife geraten? *Antwort*: Ja, aber es handelt sich eher um eine theoretische Möglichkeit, die außerdem erfolgreich bekämpft werden kann.

3.1 Initialisierung (Vorbemerkungen)

Wir beschreiben hier zunächst nur, wo die Schwierigkeit liegt. Ist das LP-Problem

$$\begin{aligned} &\text{maximiere } \sum_{j=1}^{n} c_j x_j \\ &\text{unter den Nebenbedingungen} \\ &\qquad \sum_{j=1}^{n} a_{ij} x_j \ \le b_i \qquad (i = 1, \dots, m) \\ &\qquad\qquad x_j \ \ge \ 0 \qquad (j = 1, \dots, n) \end{aligned} \tag{3.1}$$

zu lösen, so haben wir bisher immer das Starttableau dadurch erhalten, dass wir die Formeln für die Schlupfvariablen hingeschrieben haben (zusammen mit z):

$$\begin{aligned} x_{n+1} &= b_1 - \sum_{j=1}^{n} a_{1j} x_j \\ &\vdots \\ x_{n+m} &= b_m - \sum_{j=1}^{n} a_{mj} x_j \\ \hline z &= \sum_{j=1}^{n} c_j x_j. \end{aligned}$$

Als dazugehörige Startlösung konnten wir bislang immer $x_1 = ... = x_n = 0$, $x_{n+1} = b_1, \ldots, x_{n+m} = b_m$ wählen. Das führt aber nur dann zu einer zulässigen Lösung, wenn $b_i \geq 0$ $(i = 1, \ldots, m)$ gilt!

Wenn eines der b_i negativ ist, so erhält man auf diese Art keine zulässige Lösung. Wie man diese Schwierigkeit bei der Initialisierung überwindet, soll jetzt noch nicht besprochen werden – gegen Ende des Kapitels greifen wir diese Frage wieder auf (vgl. Abschnitt 3.4).

Zunächst kümmern wir uns um die beiden anderen Punkte: Iteration und Terminierung.

3.2 Iteration

3.2.1 Wahl der Eingangsvariablen

Ist ein zulässiges Tableau gegeben, so hat man für die nächste Iteration zunächst eine Eingangsvariable auszuwählen.

Als Eingangsvariable wählt man, falls dies möglich ist, eine Nichtbasisvariable x_j, für die der Koeffizient $\overline{c}_j$ in der letzten Zeile (der „z-Zeile“) des aktuellen Tableaus positiv ist: $\overline{c}_j > 0$.

Falls es kein derartiges x_j gibt, falls also

$$\overline{c}_j \leq 0$$

für alle Koeffizienten $\overline{c}_j$ in der letzten Zeile des aktuellen Tableaus gilt, *so liegt eine optimale Lösung vor.* Genauer: Die letzte Zeile des aktuellen Tableaus lautet

$$z = z^* + \sum_{j \in N} \overline{c}_j x_j,$$

wobei N die Menge der Indizes j der Nichtbasisvariablen bezeichnet und z^* der aktuelle Wert der Zielfunktion ist. Falls nun $\overline{c}_j \leq 0$ für alle $j \in N$ gilt, so führt jede zulässige Lösung zu einem Wert der Zielfunktion z, der höchstens z^* beträgt. (Man beachte, dass dann wegen $\overline{c}_j \leq 0$ und $x_j \geq 0$ gilt: $\sum_{j \in N} \overline{c}_j x_j \leq 0$.)

Falls es mehrere Basisvariablen x_j gibt, für die $\overline{c}_j > 0$ gilt, so kann prinzipiell jede dieser Variablen als Eingangsvariable gewählt werden; rechnet man kleine Beispiele per Hand, so ist es jedoch üblich, ein x_j mit möglichst großem Koeffizienten $\overline{c}_j$ zu wählen[1].

3.2.2 Wahl der Ausgangsvariablen

Ist die Eingangsvariable x_j festgelegt, so ist als Nächstes die Ausgangsvariable auszuwählen.

Als Ausgangsvariable wählt man, falls dies möglich ist, eine Basisvariable x_i, für die die Bedingung $x_i \geq 0$ zu einer möglichst strengen oberen Schranke für die Eingangsvariable führt.

Falls kein geeigneter Kandidat hierfür zur Verfügung steht, so ist das Problem unbeschränkt. Wir erläutern diesen Fall an einem Beispiel:

$$\begin{aligned} x_2 &= 5 + 2x_3 - x_4 - 3x_1 \\ x_5 &= 7 - 3x_4 - 4x_1 \\ \hline z &= 5 + x_3 - x_4 - x_1. \end{aligned}$$

Hier ist die Eingangsvariable x_3, jedoch führt weder die Bedingung $x_2 \geq 0$ noch die Bedingung $x_5 \geq 0$ zu einer oberen Schranke für die Eingangsvariable x_3. Mit anderen Worten: *Setzen wir $x_1 = x_4 = 0$, so können wir x_3 so groß machen, wie wir wünschen, die Bedingungen $x_2 \geq 0$ und $x_5 \geq 0$ werden dadurch nicht verletzt. Das Problem ist also unbeschränkt.*

[1] Die Frage, ob diese Regel immer günstig ist, wird später aufgegriffen werden.

Falls es mehrere geeignete Kandidaten für die Wahl der Ausgangsvariablen gibt, d.h., falls es mehrere Basisvariablen gibt, die zur selben möglichst strengen oberen Schranke für die Eingangsvariable führen, so kann man eine dieser Variablen beliebig auswählen.

Hat man eine Eingangsvariable und eine Ausgangsvariable gewählt, so ist die Pivotierung immer möglich.

3.2.3 Entartung

Für die Wahl der Ausgangsvariable kann es in bestimmten Fällen mehrere Kandidaten geben. Zu welchen Konsequenzen dies führt, sei an einem **Beispiel** erläutert:

$$\begin{aligned} x_4 &= 1 - 2x_3 \\ x_5 &= 3 - 2x_1 + 4x_2 - 6x_3 \\ x_6 &= 2 + x_1 - 3x_2 - 4x_3 \\ \hline z &= 2x_1 - x_2 + 8x_3. \end{aligned}$$

Wählen wir x_3 als Eingangsvariable, so ergibt sich, dass alle drei Basisvariablen x_4, x_5 und x_6 den Zuwachs von x_3 auf $\frac{1}{2}$ beschränken. Alle drei Variablen x_4, x_5 und x_6 kommen also als Ausgangsvariable infrage; wir wählen (willkürlich) x_4. Pivotierung ergibt das folgende Tableau:

$$\begin{aligned} x_3 &= 0.5 - 0.5x_4 \\ x_5 &= -2x_1 + 4x_2 + 3x_4 \\ x_6 &= x_1 - 3x_2 + 2x_4 \\ \hline z &= 4 + 2x_1 - x_2 - 4x_4. \end{aligned}$$

Dieses Tableau unterscheidet sich von allen Tableaus, die bislang vorkamen, dadurch, dass für die dazugehörige zulässige Lösung gilt: Nicht nur die Nichtbasisvariablen x_1, x_2 und x_4 sind gleich Null, sondern es gibt auch Basisvariablen, die gleich Null sind. Die zugehörige Lösung lautet:

$$x_1 = 0, \quad x_2 = 0, \quad x_3 = 0.5, \quad x_4 = 0, \quad x_5 = 0, \quad x_6 = 0.$$

Eine zulässige Basislösung wird *degeneriert* (oder *entartet*) genannt, wenn eine oder mehrere Basisvariablen gleich Null sind (engl. *degenerate*).

Fortsetzung des Beispiels:

In der nächsten Iteration ist x_1 die Eingangsvariable und x_5 ist die Ausgangsvariable. Wie üblich berechnen wir aus $x_3 \geq 0$, $x_5 \geq 0$ und $x_6 \geq 0$ den größtmöglichen Zuwachs für die Eingangsvariable x_1. Wir erhalten diesmal (im Unterschied zu allen früheren Fällen), dass kein positiver Zuwachs möglich ist, da sich aus $x_5 \geq 0$ (und $x_2 = x_4 = 0$) ergibt, dass $x_1 = 0$ gilt.

Also: x_1 bleibt unverändert gleich Null; die übrigen Variablen x_2, x_3, x_4, x_5, x_6 und z ändern sich ebenfalls nicht, nur das Tableau geht durch den Austausch von x_1 und x_5 über in

$$\begin{aligned} x_1 &= 2x_2 + 1.5x_4 - 0.5x_5 \\ x_3 &= 0.5 - 0.5x_4 \\ x_6 &= -x_2 + 3.5x_4 - 0.5x_5 \\ \hline z &= 4 + 3x_2 - x_4 - x_5. \end{aligned}$$

Definition.
Eine Iteration im Simplexverfahren heißt *degeneriert* (oder *entartet*), wenn sich die zulässige Basislösung nicht ändert.

Unsere letzte Iteration war also degeneriert.

Übungsaufgabe. Zeigen Sie, dass in unserem Beispiel auch die nächste Iteration degeneriert ist, die übernächste jedoch nicht.

Auch in praktischen Anwendungen kommen degenerierte Iterationen vor (sogar häufig); typischerweise wird der Stillstand jedoch, wie in unserem Beispiel, nach einigen Schritten überwunden.

Entartung könnte aber auch – zumindest in der Theorie – dazu führen, dass sich das Verfahren im Kreis bewegt. Mit diesem Phänomen befassen wir uns im Folgenden; man spricht vom *Kreisen* des Verfahrens (engl. *cycling*).

3.3 Terminierung

Man kann – wie wir gleich sehen werden – Beispiele konstruieren, für die das Simplexverfahren in eine unendliche Schleife gerät. Aus der Sicht der Praxis ist die allerdings kein Problem: In praktischen Anwendungen scheint dieses Phänomen aller Erfahrung nach niemals aufzutreten.

Beispiel. Unser Anfangstableau lautet

$$\begin{aligned} x_5 &= -0.5x_1 + 5.5x_2 + 2.5x_3 - 9x_4 \\ x_6 &= -0.5x_1 + 1.5x_2 + 0.5x_3 - x_4 \\ x_7 &= 1 - x_1 \\ \hline z &= 10x_1 - 57x_2 - 9x_3 - 24x_4. \end{aligned}$$

Wir verwenden die folgenden (üblichen) *Regeln*:

- Als Eingangsvariable wird immer eine Nichtbasisvariable mit einem möglichst hohen Koeffizienten in der letzten Zeile gewählt.
- Falls es mehrere Basisvariablen gibt, die für das Verlassen der Basis infrage kommen, so wählen wir die Variable mit dem kleinsten Index.

In unserem Beispiel ergeben sich in den ersten sechs Iterationen die folgenden Tableaus.

Nach der ersten Iteration:

$$\begin{aligned} x_1 &= 11x_2 + 5x_3 - 18x_4 - 2x_5 \\ x_6 &= -4x_2 - 2x_3 + 8x_4 + x_5 \\ x_7 &= 1 - 11x_2 - 5x_3 + 18x_4 + 2x_5 \\ \hline z &= 53x_2 + 41x_3 - 204x_4 - 20x_5. \end{aligned}$$

Nach der zweiten Iteration:

$$\begin{aligned} x_2 &= -0.5x_3 + 2x_4 + 0.25x_5 - 0.25x_6 \\ x_1 &= -0.5x_3 + 4x_4 + 0.75x_5 - 2.75x_6 \\ x_7 &= 1 + 0.5x_3 - 4x_4 - 0.75x_5 - 13.25x_6 \\ \hline z &= 14.5x_3 - 98x_4 - 6.75x_5 - 13.25x_6. \end{aligned}$$

Nach der dritten Iteration:

$$\begin{aligned} x_3 &= \quad 8x_4 + 1.5x_5 - 5.5x_6 - 2x_1 \\ x_2 &= -\ 2x_4 - 0.5x_5 + 2.5x_6 + x_1 \\ x_7 &= 1 \qquad\qquad\qquad\qquad - x_1 \\ \hline z &= \quad 18x_4 + 15x_5 - 93x_6 - 29x_1. \end{aligned}$$

Nach der vierten Iteration:

$$\begin{aligned} x_4 &= -\ 0.25x_5 + 1.25x_6 + 0.5x_1 - 0.5x_2 \\ x_3 &= -\ 0.5x_5 + 4.5x_6 + 2x_1 - 4x_2 \\ x_7 &= 1 \qquad\qquad\qquad - x_1 \\ \hline z &= \quad 10.5x_5 - 70.5x_6 - 20x_1 - 9x_2. \end{aligned}$$

Nach der fünften Iteration:

$$\begin{aligned} x_5 &= \quad 9x_6 + 4x_1 - 8x_2 - 2x_3 \\ x_4 &= -\ x_6 - 0.5x_1 + 1.5x_2 + 0.5x_3 \\ x_7 &= 1 \qquad - x_1 \\ \hline z &= \quad 24x_6 + 22x_1 - 93x_2 - 21x_3. \end{aligned}$$

Nach der sechsten Iteration:

$$\begin{aligned} x_6 &= -\ 0.5x_1 + 1.5x_2 + 0.5x_3 - x_4 \\ x_5 &= -\ 0.5x_1 + 5.5x_2 + 2.5x_3 - 9x_4 \\ x_7 &= 1 - x_1 \\ \hline z &= \quad 10x_1 - 57x_2 - 9x_3 - 24x_4. \end{aligned}$$

Wir sind also wieder dort, wo wir angefangen haben. Dies führt zu folgender Definition.

Definition.

Wir sagen, dass *das Simplexverfahren kreist*, falls ein und dasselbe Tableau in zwei verschiedenen Iterationen auftritt.

Kreisen hängt immer mit degenerierten Iterationen zusammen: *Sämtliche Iterationen, die dazu geführt haben, dass ein Tableau wiederholt vorkam, müssen degeneriert gewesen sein.* (Weshalb nämlich?)

Es könnte also – zumindest in der Theorie – vorkommen, dass das Simplexverfahren nicht endet, da es im Sinne der obigen Definition kreist. *Könnte es noch andere Gründe geben, weshalb das Simplexverfahren nicht terminiert?* Die Antwort ist nein, wie der Beweis des folgenden Satzes zeigt.

Satz.

Falls das Simplexverfahren nicht terminiert, so kreist es.

Beweis. *Zunächst einmal halten wir fest, dass es nur endlich viele Möglichkeiten gibt, m Basisvariablen aus der Menge $\{x_1, \ldots, x_{n+m}\}$ aller Variablen auszuwählen.* Falls das Simplexverfahren nicht terminiert, so muss demnach ein und dieselbe Basis in zwei verschiedenen Iterationen auftreten.

Es bleibt also zu zeigen, dass durch die Wahl der Basis das Tableau bereits eindeutig bestimmt ist. Wenn wir dies gezeigt haben, so sind wir fertig, da damit feststeht, dass nur endlich viele verschiedene Tableaus auftreten können, d.h., bei Nichtterminierung muss sich ein Tableau wiederholen.

Betrachten wir also zwei Tableaus, die zur selben Basis gehören:

$$\begin{aligned} x_i &= b_i - \sum_{j \notin B} a_{ij} x_j \qquad (i \in B) \\ \hline z &= v + \sum_{j \notin B} c_j x_j \end{aligned} \tag{3.2}$$

und

$$\begin{aligned} x_i &= b_i^* - \sum_{j \notin B} a_{ij}^* x_j \qquad (i \in B) \\ \hline z &= v^* + \sum_{j \notin B} c_j^* x_j \end{aligned} \tag{3.3}$$

Da (3.2) und (3.3) Tableaus sind, die zum selben LP-Problem gehören, ist jede Lösung $x_1, \ldots, x_{n+m}, z$ des Gleichungssystems (3.2) auch eine Lösung von (3.3); und umgekehrt (vgl. die Definition des Begriffs „Tableau“ in Kapitel 2, Seite 23). Man kann in einer Lösung von (3.2) bzw. (3.3) die Nichtbasisvariablen frei vorgeben; dadurch sind dann die Werte der Basisvariablen x_i $(i \in B)$ und der Wert von z eindeutig bestimmt. Gibt man zum Beispiel $x_j = 0$ für alle $j \notin B$ vor, so erhält man $b_i = b_i^*$ für alle $i \in B$ sowie $v = v^*$.

Gibt man für eine beliebig gewählte Nichtbasisvariable x_j vor, dass $x_j = 1$ gelten soll, und setzt alle anderen Nichtbasisvariablen gleich Null, so erhält man für alle $i \in B$:

$$b_i - a_{ij} = x_i = b_i^* - a_{ij}^*.$$

Hieraus folgt (wegen $b_i = b_i^*$): $a_{ij} = a_{ij}^*$.

Ganz entsprechend erhält man $c_j = c_j^*$ für alle $j \notin B$.

Somit haben wir gezeigt, dass die Tableaus (3.2) und (3.3) gleich sind. Anders gesagt: *Wir haben gezeigt, dass durch die Wahl der Basis das zugehörige Tableau eindeutig bestimmt ist.* □

Beim Kreisen handelt es sich – wie bereits gesagt – um eine *theoretische Möglichkeit*. In der Praxis spielt Kreisen keine Rolle. Aus Sicht der Theorie möchte man Kreisen natürlich völlig ausschließen. Dies ist in der Tat möglich: Es gibt verschiedene Möglichkeiten hierfür, von denen die *Blandsche Regel* (*Bland's rule*) die interessanteste ist.

Bei dieser Regel geht es zunächst darum, wie die Eingangsvariable zu wählen ist, wenn es mehrere Nichtbasisvariablen gibt, für die der Koeffizient in der z-Zeile positiv ist. Wir wissen: Prinzipiell kommen alle diese Variablen infrage. Bei der Verwendung von Bland's rule spielt die Größe der positiven Koeffizienten in der z-Zeile gar keine Rolle; *man wählt unter den Variablen mit positivem Koeffizienten in der z-Zeile einfach die Variable x_k mit dem kleinsten Index k.* Nach erfolgter Wahl der Eingangsvariablen verfährt man bei der Wahl der Ausgangsvariablen entsprechend: Kommen mehrere Basisvariablen als Ausgangsvariablen in Betracht, so wähle man unter diesen die Variable mit dem kleinsten Index. (Man beachte: Nicht alle Basisvariablen kommen als Ausgangsvariablen in Betracht, da nur zulässige Tableaus entstehen dürfen.) Kurz zusammengefasst ergibt dies die folgende Regel:

Bland's rule.

Gibt es mehrere Möglichkeiten für die Wahl der Eingangs- bzw. Ausgangsvariablen, so wähle man immer den Kandidaten x_k mit dem kleinsten Index k.

Bland's rule ist leicht zu formulieren und leicht zu merken. Etwas schwieriger ist es dagegen, zu beweisen, dass bei Beachtung von Bland's rule Kreisen nicht möglich ist.

Satz.
Bei Verwendung von Bland's rule terminiert das Simplexverfahren.

Beweis. Aufgrund des vorangehenden Satzes haben wir zu zeigen, dass das Simplexverfahren bei Verwendung von Bland's rule nicht kreist. Zu diesem Zweck nehmen wir an, dass Bland's rule verwendet wird und dass es trotzdem ein Tableau T_0 gibt, das in einer Folge von degenerierten Iterationen in sich selbst übergeht: $T_0, T_1, \ldots, T_k = T_0$ seien die dabei auftretenden Tableaus ($k \geq 1$). Diese Annahme ist zum Widerspruch zu führen.

Wir nennen eine Variable *unbeständig*, falls sie in den Tableaus $T_0, T_1, \ldots, T_k = T_0$ mindestens einmal als Basis-, aber mindestens einmal auch als Nichtbasisvariable auftritt. Unter allen unbeständigen Variablen sei x_t die Variable mit dem größten Index t. Dann gibt es in der Folge $T_0, T_1, \ldots, T_k = T_0$ ein Tableau T, in dem x_t Ausgangsvariable ist, d.h., x_t ist Basisvariable in T und Nichtbasisvariable im darauffolgenden Tableau. Die entsprechende Eingangsvariable sei x_s, d.h., x_s ist Nichtbasisvariable in T und Basisvariable im darauffolgenden Tableau.

Da es sich bei $T_0, T_1, \ldots, T_k = T_0$ um eine zyklische Folge von Tableaus handelt, muss es in dieser Folge ein Tableau T^* geben, in dem x_t Eingangsvariable ist, und es gibt einen Abschnitt der Folge $T_0, T_1, \ldots, T_k, T_1, \ldots, T_k$, durch den T in T^* überführt wird.

Das Tableau T sei wie folgt dargestellt:

$$\begin{aligned} x_i &= b_i - \sum_{j \notin B} a_{ij} x_j \qquad (i \in B) \\ \hline z &= v + \sum_{j \notin B} c_j x_j \end{aligned}$$

Da alle Iterationen, die von T nach T^* führen, degeneriert sind, hat die Zielfunktion z in beiden Tableaus denselben Wert. Deshalb lässt sich die letzte Zeile von T^* wie folgt schreiben:

$$z = v + \sum_{j=1}^{m+n} c_j^* x_j, \tag{3.4}$$

wobei $c_j^* = 0$ gilt, falls x_j eine Basisvariable von T^* ist. Man beachte, dass Folgendes gilt: Da die Gleichung (3.4) aus T durch algebraische Umformungen entstanden ist, wird (3.4) von jeder Lösung $x_1, \ldots, x_{m+n}, z$ von T erfüllt. Insbesondere wird (3.4) von der folgenden Lösung von T erfüllt (für ein beliebig gewähltes $y \in \mathbb{R}$):

$$x_s = y, \quad x_j = 0 \ (j \notin B,\ j \neq s), \quad x_i = b_i - a_{is} y \ (i \in B) \quad \text{und} \quad z = v + c_s y.$$

Durch Einsetzen in (3.4) folgt für alle $y \in \mathbb{R}$

$$v + c_s y = v + c_s^* y + \sum_{i \in B} c_i^* \big(b_i - a_{is} y\big),$$

woraus man durch Umformung erhält:

$$\left(c_s - c_s^* + \sum_{i \in B} c_i^* a_{is}\right) y = \sum_{i \in B} c_i^* b_i \quad \text{für alle } y \in \mathbb{R}. \tag{3.5}$$

Da die rechte Seite von (3.5) eine von y unabhängige Konstante ist, ergibt sich als Konsequenz:

$$c_s - c_s^* + \sum_{i \in B} c_i^* a_{is} = 0. \tag{3.6}$$

Da x_s Eingangsvariable von T ist, gilt $c_s > 0$. Außerdem gilt $c_s^* \leq 0$. (Begründung: x_s ist eine unbeständige Variable mit $s \neq t$, woraus $s < t$ folgt. Wäre $c_s^* > 0$, so wäre x_s eine Nichtbasisvariable in T^* mit $c_s^* > 0$. Dies ist ein Widerspruch zur Tatsache, dass x_t die Eingangsvariable von T^* ist und dass Bland's rule verwendet wird.) Aus (3.6) folgt somit, dass es ein $r \in B$ geben muss, so dass gilt

$$c_r^* a_{rs} < 0. \tag{3.7}$$

Wegen $r \in B$ handelt es sich bei x_r um eine Basisvariable von T. Aufgrund von (3.7) gilt $c_r^* \neq 0$, weshalb x_r eine Nichtbasisvariable von T^* ist. Also ist auch x_r unbeständig und es gilt $r \leq t$. Da x_t Ausgangsvariable von T ist und x_s Eingangsvariable, gilt $a_{ts} > 0$; ferner gilt $c_t^* > 0$, da t Eingangsvariable von T^* ist. Es folgt $c_t^* a_{ts} > 0$. Vergleicht man dies mit (3.7), so erkennt man, dass $r \neq t$ gilt. Insgesamt haben wir also $r < t$ erhalten und außerdem ist x_r nicht die Eingangsvariable von T^*. Daraus ergibt sich (wegen Bland's rule), dass nicht $c_r^* > 0$ gelten kann. Somit folgt aus (3.7):

$$a_{rs} > 0.$$

Da alle Iterationen, die T in T^* überführen, degeneriert sind, besitzen T und T^* dieselbe zulässige Basislösung. In der zulässigen Basislösung für T^* gilt $x_r = 0$ (da x_r Nichtbasisvariable für T^* ist). Folglich gilt ebenfalls $x_r = 0$ in der zulässigen Basislösung für T, d.h. $b_r = 0$. Es folgt, dass x_r ein geeigneter Kandidat war für das Verlassen der Basis von T, für den $r < t$ gilt. Trotzdem wurde x_t gewählt. Dieser Widerspruch beweist unseren Satz. □

Nach diesen Ausführungen zu den Punkten Iteration und Terminierung kommen wir – wie zu Beginn des Kapitels angekündigt – auf den Punkt Initialisierung zurück.

3.4 Das Zweiphasen-Simplexverfahren

Wir betrachten ein Problem in Standardform, wobei mindestens eine der rechten Seiten negativ ist:

$$\begin{aligned}
&\text{maximiere } \sum_{j=1}^{n} c_j x_j \\
&\text{unter den Nebenbedingungen} \\
&\qquad \sum_{j=1}^{n} a_{ij} x_j \leq b_i \qquad (i = 1, \ldots, m) \\
&\qquad\qquad x_j \geq 0 \qquad (j = 1, \ldots, n).
\end{aligned}$$

Um ein zulässiges Starttableau und damit auch eine zulässige Startlösung zu finden, betrachtet man das folgende LP-Problem, das *Hilfsproblem* genannt wird:

$$\begin{aligned}
&\text{minimiere } x_0 \\
&\text{unter den Nebenbedingungen} \\
&\qquad \sum_{j=1}^{n} a_{ij} x_j - x_0 \leq b_i \qquad (i = 1, \ldots, m) \\
&\qquad\qquad x_j \geq 0 \qquad (j = 0, \ldots, n).
\end{aligned}$$

Eine zulässige Lösung des Hilfsproblems lässt sich leicht angeben: Man braucht nur die Variablen $x_1, \ldots, x_n$ gleich Null zu setzen und x_0 hinreichend groß zu wählen. *Das Hilfsproblem besitzt also immer eine zulässige Lösung.* Außerdem erkennt man sofort, dass Folgendes gilt:

Feststellung 1.

Das ursprüngliche Problem besitzt genau dann eine zulässige Lösung, wenn das Hilfsproblem eine zulässige Lösung mit $x_0 = 0$ besitzt bzw. (anders gesagt), wenn der optimale Wert des Hilfsproblems gleich Null ist.

Genauer gilt, dass sich die zulässigen Lösungen des ursprünglichen Problems und die optimalen Lösungen des Hilfsproblems, für die $x_0 = 0$ gilt, auf eine sehr einfache Art entsprechen:

(i) Ist $x_1, \ldots, x_n$ eine zulässige Lösung des ursprünglichen Problems, so erhält man eine optimale Lösung des Hilfsproblems, wenn man zusätzlich $x_0 = 0$ setzt.

(ii) Ist umgekehrt $x_0, \ldots, x_n$ eine optimale Lösung des Hilfsproblems, für die $x_0 = 0$ gilt, so ist $x_1, \ldots, x_n$ eine zulässige Lösung des ursprünglichen Problems.

Wir erläutern nun das weitere Vorgehen anhand des folgenden **Beispiels**:

$$
\begin{aligned}
&\text{maximiere} \quad x_1 - x_2 + x_3 \\
&\text{unter den Nebenbedingungen} \\
&\qquad\qquad 2x_1 - x_2 + 2x_3 \leq 4 \\
&\qquad\qquad 2x_1 - 3x_2 + x_3 \leq -5 \\
&\qquad\qquad -x_1 + x_2 - 2x_3 \leq -1 \\
&\qquad\qquad x_1, x_2, x_3 \geq 0.
\end{aligned}
$$

Das dazugehörige *Hilfsproblem* lautet, wenn es als *Maximierungsproblem* geschrieben wird:

$$
\begin{aligned}
&\text{maximiere } -x_0 \\
&\text{unter den Nebenbedingungen} \\
&\qquad\qquad 2x_1 - x_2 + 2x_3 - x_0 \leq 4 \\
&\qquad\qquad 2x_1 - 3x_2 + x_3 - x_0 \leq -5 \\
&\qquad\qquad -x_1 + x_2 - 2x_3 - x_0 \leq -1 \\
&\qquad\qquad x_0, x_1, x_2, x_3 \geq 0.
\end{aligned}
$$

Schreibt man die Formeln für die Schlupfvariablen x_4, x_5 und x_6 sowie für die Zielfunktion w auf, so erhält man für das Hilfsproblem das folgende Tableau:

$$
\begin{aligned}
x_4 &= 4 - 2x_1 + x_2 - 2x_3 + x_0 \\
x_5 &= -5 - 2x_1 + 3x_2 - x_3 + x_0 \\
x_6 &= -1 + x_1 - x_2 + 2x_3 + x_0 \\
\hline
w &= - x_0.
\end{aligned}
$$

Dieses Tableau ist ***kein zulässiges Tableau****, es kann jedoch durch eine einzige Pivotierung in ein zulässiges Tableau verwandelt werden.* Diese Pivotierung folgt jedoch einem etwas anderen Schema, als bislang gewohnt: Man wählt x_0 als Eingangsvariable, obwohl x_0 in der letzten Zeile des Tableaus mit einem negativen Koeffizienten auftritt. Wählt man zusätzlich x_5 als Ausgangsvariable, so erhält man das nunmehr zulässige Tableau:

$$
\begin{aligned}
x_0 &= 5 + 2x_1 - 3x_2 + x_3 + x_5 \\
x_4 &= 9 - 2x_2 - x_3 + x_5 \\
x_6 &= 4 + 3x_1 - 4x_2 + 3x_3 + x_5 \\
\hline
w &= -5 - 2x_1 + 3x_2 - x_3 - x_5.
\end{aligned}
\tag{3.8}
$$

Im allgemeinen Fall geht alles ganz entsprechend, man schreibt zunächst das Hilfsproblem als Maximierungsproblem auf:

$$\begin{aligned}
&\text{maximiere } -x_0\\
&\text{unter den Nebenbedingungen}\\
&\qquad \sum_{j=1}^{n} a_{ij}x_j - x_0 \le b_i \qquad (i = 1,\dots,m)\\
&\qquad\qquad x_j \ge 0 \qquad (j = 0,\dots,n).
\end{aligned}$$

Aufschreiben der Formeln für die Schlupfvariablen $x_{n+1},\dots,x_{n+m}$ und für die Zielfunktion w führt zum folgenden (nicht zulässigen) Tableau für das Hilfsproblem:

$$\begin{aligned}
x_{n+i} &= b_i - \sum_{j=1}^{n} a_{ij}x_j + x_0 \qquad (i = 1,\dots,m)\\
\hline
w &= \qquad\qquad - x_0.
\end{aligned}$$

Dieses Tableau kann durch eine einzige Pivotierung in ein zulässiges Tableau verwandelt werden: Als Eingangsvariable wählt man x_0 und als Ausgangsvariable x_{n+k} wählt man eine Variable mit negativem b_k, für die $|b_k|$ maximal ist[2].

Wir fahren mit unserem Beispiel fort. Da wir mit dem Tableau (3.8) nun ein *zulässiges* Tableau für unser Hilfsproblem vorliegen haben, können wir das Hilfsproblem wie gewohnt mit dem Simplexalgorithmus lösen. Als Starttableau dient dabei (3.8). Im ersten Iterationsschritt ist x_2 die Eingangs- und x_6 die Ausgangsvariable. Man erhält:

$$\begin{aligned}
x_2 &= 1 + 0.75x_1 + 0.75x_3 + 0.25x_5 - 0.25x_6\\
x_0 &= 2 - 0.25x_1 - 1.25x_3 + 0.25x_5 + 0.75x_6\\
x_4 &= 7 - 1.5x_1 - 2.5x_3 + 0.5x_5 + 0.5x_6\\
\hline
w &= -2 + 0.25x_1 + 1.25x_3 - 0.25x_5 - 0.75x_6.
\end{aligned}$$

Nach der zweiten Iteration (Eingangsvariable x_3 und Ausgangsvariable x_0) erhält man:

$$\begin{aligned}
x_3 &= 1.6 - 0.2x_1 + 0.2x_5 + 0.6x_6 - 0.8x_0\\
x_2 &= 2.2 + 0.6x_1 + 0.4x_5 + 0.2x_6 - 0.6x_0\\
x_4 &= 3 - x_1 - x_6 + 2x_0\\
\hline
w &= \qquad\qquad - x_0.
\end{aligned} \tag{3.9}$$

Das Tableau (3.9) ist optimal. Als optimale Lösung des Hilfsproblems erhält man:

$$x_0 = 0, \quad x_1 = 0, \quad x_2 = 2.2, \quad x_3 = 1.6.$$

Aufgrund von (ii) (Seite 36) ergibt sich als zulässige Lösung für das ursprüngliche Problem:

$$x_1 = 0, \quad x_2 = 2.2, \quad x_3 = 1.6.$$

Wir haben somit herausgefunden, dass das ursprüngliche Problem eine zulässige Lösung besitzt. Noch viel nützlicher ist jedoch die folgende Feststellung.

[2] Man überlege sich, dass dies immer zu einem zulässigen Tableau führt (vgl. auch V. Chvátal: *Linear Programming*, Seite 40).

Feststellung 2.

Darüber hinaus kann das Tableau (3.9) auf einfache Art in ein zulässiges Tableau für unser ursprüngliches Problem verwandelt werden (*Starttableau für unser ursprüngliches Problem!*).

Hier ist das zulässige Tableau für unser ursprüngliches Problem, das sich aus (3.9) ergibt:

$$\begin{aligned} x_3 &= 1.6 - 0.2x_1 + 0.2x_5 + 0.6x_6 \\ x_2 &= 2.2 + 0.6x_1 + 0.4x_5 + 0.2x_6 \\ x_4 &= 3 - x_1 - x_6 \\ \hline z &= -0.6 + 0.2x_1 - 0.2x_5 + 0.4x_6 \end{aligned} \tag{3.10}$$

Die ersten drei Zeilen von (3.10) sind aus (3.9) dadurch entstanden, dass der letzte Summand („derjenige, in dem x_0 vorkommt“) in jeder dieser Zeilen weggelassen wurde. Anders gesagt: Man betrachtet nur noch Lösungen von (3.9), für die $x_0 = 0$ gilt. Die letzte Zeile ergibt sich wie folgt: Für die ursprüngliche Zielfunktion z gilt

$$z = x_1 - x_2 + x_3.$$

Ersetzt man hierin x_2 und x_3 gemäß der ersten beiden Zeilen von (3.10), so erhält man die letzte Zeile von (3.10).

Aufgrund von (i) und (ii) auf Seite 36 erkennt man, dass die beschriebene Methode zu einem zulässigen Tableau für das ursprüngliche Problem führt, das dann als Starttableau dienen kann. *Klarerweise funktioniert das alles aber nur, wenn es einem wie im Beispiel gelingt, x_0 aus der Basis „rauszuschmeißen“.*

Um dies zu erreichen, verfährt man bei der Lösung des Hilfsproblems nach der folgenden **Regel**:

Gibt es nach erfolgter Wahl der Eingangsvariablen mehrere geeignete Kandidaten für die Wahl der Ausgangsvariablen und ist x_0 unter diesen Kandidaten, so entscheide man sich für x_0.

Dass es mehrere geeignete Kandidaten für die Wahl der Ausgangsvariablen gibt, bedeutet natürlich, dass es mehrere Basisvariablen gibt, die zur selben möglichst strengen oberen Schranke für die Eingangsvariable führen. Man beachte: Nachdem man (am Anfang) ein zulässiges Tableau hergestellt hat, löst man das Hilfsproblem wie gewohnt mit dem Simplexverfahren. Das bedeutet: *Es dürfen nur noch zulässige Tableaus auftreten und nur wenn x_0 eine Basisvariable ist, die zu einer möglichst strengen Schranke für die Eingangsvariablen führt, kann man x_0 aus der Basis ausscheiden lassen.*

Verfährt man nach der obigen Regel, so erhält man unmittelbar im Anschluss an die Anwendung dieser Regel ein Tableau, in dem folgende Situation vorliegt: x_0 ist nicht mehr in der Basis, weshalb für die zu diesem Tableau dazugehörige zulässige Basislösung $x_0 = 0$ gilt. Folglich gilt dann auch

$$w = -x_0 = 0.$$

Kommt die obige Regel zum Einsatz, gelingt es also, x_0 aus der Basis ausscheiden zu lassen, so erhält man eine zulässige Basislösung des Hilfsproblems, für die $x_0 = 0$ und folglich auch $w = 0$ gilt. Diese Lösung ist eine *optimale* Lösung des Hilfsproblems, da die Zielfunktion $w = -x_0$ nicht größer als Null werden kann (wegen $x_0 \geq 0$). **In diesem Fall hat man erreicht, was man wollte**: Aus dem Schlusstableau für das Hilfsproblem kann man – wie beschrieben – ein *Starttableau* für das ursprüngliche Problem gewinnen.

Es bleibt der Fall zu betrachten, dass wir mit dem Hilfsproblem fertig sind, obwohl die obige Regel nicht zum Einsatz gekommen ist: *Dann wäre das Hilfsproblem gelöst, aber x_0 wäre am Ende immer noch eine Basisvariable.*

Behauptung. Liegt dieser Fall vor, so gilt am Ende $w < 0$.

Beweis. Angenommen x_0 wäre am Ende immer noch eine Basisvariable und es würde $w = 0$ gelten. Wegen $w = -x_0$ würde am Ende dann auch $x_0 = 0$ gelten. Wir betrachten das vorletzte Tableau: Dieses war noch nicht optimal, d.h., es galt $w = -x_0 < 0$. Beim Übergang vom vorletzten zum letzten Tableau

hat sich der Wert von x_0 also von einem positiven Wert zu $x_0 = 0$ geändert. Man erkennt unschwer, dass hieraus folgt: Bei diesem Übergang wäre auch x_0 als Ausgangsvariable infrage gekommen, aber x_0 wurde entgegen der obigen Regel nicht gewählt. Dieser Widerspruch beweist die Behauptung. □

Man beachte: Dass man bei der Lösung des Hilfsproblems am Schluss den optimalen Zielfunktionswert $w < 0$ erhält, bedeutet, dass es nur Lösungen des Hilfsproblems mit $x_0 > 0$ gibt (wegen $w = -x_0$). Das heißt (vgl. Feststellung 1): *Das ursprüngliche Problem besitzt keine zulässige Lösung.*

Das beschriebene Verfahren ist unter dem Namen *Zweiphasen-Simplexverfahren* bekannt:

- Liegt ein *LP-Problem in Standardform* vor, so stellt man in der *ersten Phase* zunächst das Hilfsproblem auf und löst es wie beschrieben; falls dies mit $w < 0$ endet, so besitzt das ursprüngliche Problem keine zulässige Lösung; falls am Ende der ersten Phase $w = 0$ gilt, so erhält man ein *Starttableau für die zweiten Phase*[3].
- In der *zweiten Phase* löst man das ursprüngliche Problem.

Gilt $b_i \geq 0$ $(i = 1, \ldots, m)$, so entfällt die 1. Phase.

Auf Grundlage des beschriebenen Verfahrens ergibt sich folgender Satz, den man *Fundamentalsatz der Linearen Programmierung* nennt.

Satz (Fundamentalsatz der Linearen Programmierung).
Jedes LP-Problem in Standardform besitzt die folgenden Eigenschaften:

(i) Falls es eine zulässige Lösung gibt, so gibt es auch eine zulässige Basislösung.

(ii) Falls es eine optimale Lösung gibt, so gibt es auch eine optimale Basislösung.

(iii) Falls es keine optimale Lösung gibt, so ist das Problem entweder unlösbar oder unbeschränkt.

Beweis. In der ersten Phase des Zweiphasen-Simplexverfahrens wird entweder festgestellt, dass das Problem unlösbar ist, oder es wird eine zulässige Basislösung gefunden. Also gilt (i). In der zweiten Phase wird entweder festgestellt, dass das Problem unbeschränkt ist, oder man erhält eine optimale Basislösung. Also gelten (ii) und (iii). □

In manchen Lehrbüchern (vgl. etwa Cormen et al) wird auch die folgende Aussage als „Fundamentalsatz der Linearen Programmierung“ bezeichnet.

Satz.
Für jedes LP-Problem L in Standardform gilt genau eine der folgenden drei Aussagen:

(i) L besitzt eine optimale Lösung.

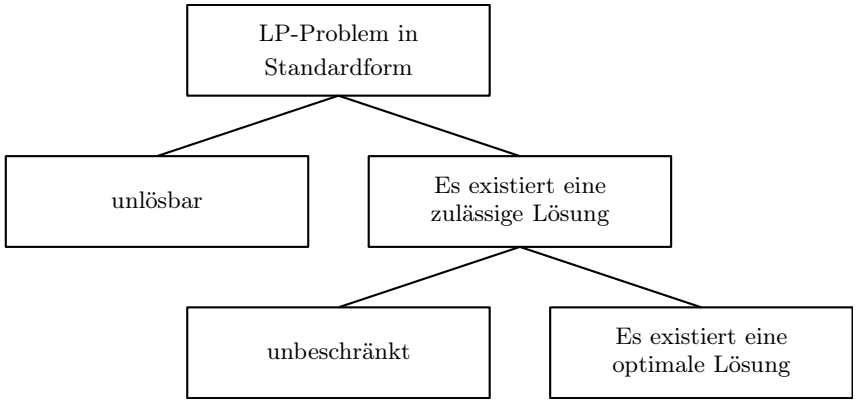
(ii) L ist unlösbar.

(iii) L ist unbeschränkt.

Beweis. Die erste Phase des Zweiphasen-Simplexverfahrens führt entweder zu der Feststellung, dass L unlösbar ist, oder man erhält ein Starttableau für die zweite Phase. In der zweiten Phase wird entweder festgestellt, dass das Problem unbeschränkt ist, oder man erhält eine optimale Lösung. □

[3] Vgl. Seite 37.

Die Aussage dieses Satzes lässt sich wie folgt darstellen:



4 Verbindungen zur Geometrie

4.1 Polyeder

Im Zusammenhang mit der grafischen Methode hatten wir bereits Geraden, Halbebenen und Durchschnitte von endlich vielen Halbebenen im $\mathbb{R}^2$ betrachtet (vgl. Kapitel 1). Wir wollen einige der dort auftretenden Begriffe verallgemeinern.

In Kapitel 1 hatten wir beispielsweise festgestellt: Sind $a_1, a_2, b \in \mathbb{R}$ gegeben und sind a_1, a_2 nicht beide gleich Null, so wird durch die Gleichung

$$a_1x_1 + a_2x_2 = b$$

eine *Gerade im* $\mathbb{R}^2$ dargestellt; die Menge aller Paare $(x_1, x_2) \in \mathbb{R}^2$, für die

$$a_1x_1 + a_2x_2 \leq b$$

gilt, bilden eine *Halbebene*, die durch diese Gerade begrenzt wird.

Ähnliches gilt im $\mathbb{R}^3$: Sind $a_1, a_2, a_3, b \in \mathbb{R}$ gegeben und sind a_1, a_2, a_3 nicht alle gleich Null, so wird durch die Gleichung

$$a_1x_1 + a_2x_2 + a_3x_3 = b$$

eine *Ebene im* $\mathbb{R}^3$ dargestellt, und die Menge aller Tripel $(x_1, x_2, x_3) \in \mathbb{R}^3$, für die

$$a_1x_1 + a_2x_2 + a_3x_3 \leq b$$

gilt, bilden einen *Halbraum*, der durch diese Ebene begrenzt wird.

Analoge Sprechweisen verwendet man auch im $\mathbb{R}^n$: Sind $a_1, \ldots, a_n, b \in \mathbb{R}^n$ gegeben und sind nicht alle $a_1, \ldots, a_n$ gleich Null, so nennt man die Menge aller n-Tupel $(x_1, \ldots, x_n) \in \mathbb{R}^n$, für die die Gleichung

$$a_1x_1 + \ldots + a_nx_n = b$$

gilt, eine *Hyperebene im* $\mathbb{R}^n$; und die Menge aller n-Tupel $(x_1, \ldots, x_n) \in \mathbb{R}^n$, für die

$$a_1x_1 + \ldots + a_nx_n \leq b$$

gilt, nennt man einen *Halbraum des* $\mathbb{R}^n$.

Dem Buch von Chvátal folgend, wählen wir für unsere weiteren Betrachtungen das folgende Beispiel eines LP-Problems:

$$\begin{aligned} &\text{maximiere } 3x_1 + 2x_2 + 5x_3 \\ &\text{unter den Nebenbedingungen} \\ &\qquad 2x_1 + x_2 \leq 4 \\ &\qquad x_3 \leq 5 \\ &\qquad x_1, x_2, x_3 \geq 0. \end{aligned}$$

Jede der fünf Nebenbedingungen beschreibt einen gewissen Halbraum des $\mathbb{R}^3$. Also ist die Menge der zulässigen Lösungen[1] dieses LP-Problems *gleich dem Durchschnitt von fünf Halbräumen*. Eine derartige Menge bezeichnet man als *Polyeder*.

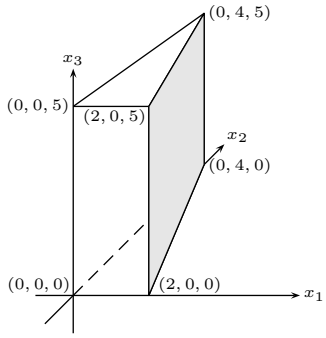
[1] Statt „Menge der zulässigen Lösungen“ sagt man auch *Bereich der zulässigen Lösungen* oder (kurz) *zulässiger Bereich* (vgl. Kapitel 1).

Genauer gilt:

Definition.
Eine Teilmenge P des $\mathbb{R}^n$ wird *Polyeder* genannt, falls P gleich dem Durchschnitt von endlich vielen Halbräumen des $\mathbb{R}^n$ ist oder falls $P = \mathbb{R}^n$ gilt.

Für $n = 2$ kamen Polyeder bereits in Kapitel 1 vor.

Das in unserem 3-dimensionalen Beispiel auftretende Polyeder ist in der folgenden Zeichnung dargestellt:



In diesem Beispiel hat das Polyeder, das den Bereich der zulässigen Lösungen beschreibt, also die Gestalt eines Prismas. Zu jeder der fünf Begrenzungsflächen des Prismas gehört eine der fünf Nebenbedingungen. Genauer gilt: *Die Punkte auf jeder der fünf Flächen*[2] *erfüllen die zugehörige Nebenbedingung* ***mit Gleichheit***.

Beispielsweise entspricht die Grundfläche (der dreieckige Boden) der Nebenbedingung $x_3 \geq 0$: Für die Punkte auf dem Boden des Prismas gilt $x_3 = 0$. Der grauen Fläche auf der rechten Seite des Prismas entspricht die Nebenbedingung $2x_1 + x_2 \leq 4$, und für die Punkte auf dieser Fläche gilt $2x_1 + x_2 = 4$. (Man überlege sich, welche Entsprechungen zwischen den verbleibenden Flächen und Nebenbedingungen außerdem gelten.)

Die Entsprechungen zwischen Flächen einerseits und Nebenbedingungen andererseits werden besonders deutlich, wenn man *Schlupfvariablen* x_4, x_5 wie üblich hinzunimmt. Es gelte also (Definition der Schlupfvariablen):

$$\begin{aligned} x_4 &= 4 - 2x_1 - x_2 \\ x_5 &= 5 \qquad\qquad - x_3. \end{aligned}$$

Jeder Fläche des Polyeders entspricht dann genau eine der Variablen $x_1, \ldots, x_5$ in dem Sinne, dass $x_j = 0$ für die Punkte auf der entsprechenden Fläche gilt. Im Einzelnen hat man Folgendes:

- für die Punkte auf der linken Fläche gilt $x_1 = 0$;
- für die Punkte auf der Frontfläche gilt $x_2 = 0$;
- für die Punkte auf der Grundfläche („Boden“) gilt $x_3 = 0$;
- für die Punkte auf der grauen Fläche gilt $x_4 = 0$;
- für die Punkte auf der oberen Fläche („Deckel“) gilt $x_5 = 0$.

Die Menge der zulässigen Lösungen haben wir bereits genau beschrieben: Es handelt sich um die Punkte des Prismas – natürlich einschließlich derjenigen, die im Inneren des Prismas liegen. Insbesondere können wir festhalten, dass es unendlich viele zulässige Lösungen gibt. *Die meisten dieser Lösungen werden – wie wir wissen – vom Simplexverfahren jedoch ignoriert*: Im Simplexverfahren kommen als Ergebnis einer Iteration nur **zulässige Basislösungen** vor.

[2] Statt „Begrenzungsflächen“ sagt man häufig auch „Seitenflächen“. Wir werden hier der Einfachheit halber immer „Flächen“ sagen.

In jeder zulässigen Basislösung sind drei der fünf Variablen $x_1, \ldots, x_5$ Nichtbasisvariablen. Diese drei Variablen sind gleich Null, und die Werte der beiden übrigen Variablen sind dadurch eindeutig bestimmt, dass die Nichtbasisvariablen gleich Null gesetzt wurden. *Geometrisch bedeutet dies, dass jeder Punkt, der eine zulässige Basislösung darstellt, ein eindeutig bestimmter Schnittpunkt von drei Flächen des Prismas ist*, dass es sich also um einen Eckpunkt des Prismas handelt (vgl. Zeichnung).

Statt „Eckpunkt“ sagt man gewöhnlich auch *Ecke* (engl. *vertex*) eines Polyeders.

4.2 Geometrische Interpretation des Simplexverfahrens

Wir wollen uns anschauen, was in unserem Beispiel geometrisch passiert, wenn wir es mit dem Simplexverfahren lösen. Wir interessieren uns dabei nicht für jede Einzelheit, sondern vor allem für die *zulässigen Basislösungen am Ende jeder Iteration*; außerdem geben wir an, welches am Ende jeder Iteration die Basisvariablen sind. Hier das Ergebnis:

Startlösung:

$$x_1 = 0, \quad x_2 = 0, \quad x_3 = 0, \quad x_4 = 4, \quad x_5 = 5; \quad \text{Basis: } x_4,\ x_5.$$

Am Ende der 1. Iteration:

$$x_1 = 0, \quad x_2 = 0, \quad x_3 = 5, \quad x_4 = 4, \quad x_5 = 0; \quad \text{Basis: } x_3,\ x_4.$$

Am Ende der 2. Iteration:

$$x_1 = 2, \quad x_2 = 0, \quad x_3 = 5, \quad x_4 = 0, \quad x_5 = 0; \quad \text{Basis: } x_1,\ x_3.$$

Am Ende der 3. Iteration:

$$x_1 = 0, \quad x_2 = 4, \quad x_3 = 5, \quad x_4 = 0, \quad x_5 = 0; \quad \text{Basis: } x_2,\ x_3.$$

Am Ende der dritten Iteration wurde eine optimale Lösung erreicht: Im Punkt $(x_1, x_2, x_3) = (0, 4, 5)$ ist die Zielfunktion optimal (mit $z = 33$). Auf dem Weg zu diesem Ergebnis erhielten wir die obigen zulässigen Basislösungen; die dazugehörigen Punkte waren:

$$(x_1, x_2, x_3) = (0, 0, 0)$$
$$(x_1, x_2, x_3) = (0, 0, 5)$$
$$(x_1, x_2, x_3) = (2, 0, 5)$$

sowie schließlich

$$(x_1, x_2, x_3) = (0, 4, 5).$$

Es handelt sich dabei – wie aufgrund unserer Vorüberlegungen nicht anders zu erwarten war – um *Ecken unseres Prismas* (siehe obige Zeichnung). *Die Übergänge von einer Ecke zur nächsten lassen sich als kontinuierliche Bewegung längs der Kanten interpretieren.*

Betrachten wir beispielsweise die 2. Iteration: Eingangsvariable war hier x_1. Die aktuelle Ecke vor der 2. Iteration war $(0, 0, 5)$. Nach Wahl von x_1 als Eingangsvariable wurde der Wert von x_1 kontinuierlich angehoben – unter Beibehaltung von $x_2 = 0$ und $x_5 = 0$. Die Bedingung $x_2 = x_5 = 0$ besagt, dass wir nur Punkte betrachten, die sowohl zur vorderen als auch zur oberen Fläche des Polyeders gehören. (Man beachte: $x_2 = 0$ ist die Bedingung für die vordere, $x_5 = 0$ die Bedingung für die obere Fläche.)

Beim Anheben von x_1 unter Beibehaltung von $x_2 = 0$ und $x_5 = 0$ bewegen wir uns also längs der Kante, die sowohl zur vorderen als auch zur oberen Fläche gehört – und zwar solange, bis wir die nächste Ecke erreicht haben. *In unserem Fall haben wir uns also von* $(0, 0, 5)$ *zu* $(2, 0, 5)$ *bewegt.*

Wir haben in unserem Beispiel *den typischen Ablauf des Simplexverfahrens* geometrisch beschrieben.

Feststellung.

Kommen keine degenerierten Iterationen vor, *so bewegt man sich längs bestimmter Kanten des zugehörigen Polyeders von einer Ecke zur nächsten, wobei der Zielfunktionswert steigt. Man stoppt, wenn man an einer optimalen Ecke angekommen ist* bzw. wenn man feststellt, dass das Problem unbeschränkt ist.

Bei dieser Beschreibung haben wir vorausgesetzt, dass das Problem lösbar ist und dass wir ein Starttableau kennen (und somit auch eine *Startecke*). Falls degenerierte Iterationen auftreten, so ändert sich an der Beschreibung nicht viel: Es kann dann zusätzlich vorkommen, dass man auf einer Ecke verweilt. (Zur Erinnerung: Bei einer degenerierten Iteration ändert sich nur die Basis, die zulässige Basislösung ändert sich nicht.)

Eine weitere Bemerkung ist angebracht: Wir haben zwar definiert, was man unter einem Polyeder versteht, bei anderen Begriffen (wie beispielsweise Ecke, Kante oder Seitenfläche eines Polyeders) haben wir uns aber lediglich an der Anschauung orientiert, ohne eine mathematische Definition zu geben. Wir wollen dies hier nicht nachholen, sondern verweisen auf die Literatur; man findet Näheres beispielsweise in

- D. Bertsimas, J. N. Tsitsiklis, *Introduction to Linear Optimization*. Athena Scientific.
- A. Schrijver: *Theory of Linear and Integer Programming*. Wiley.

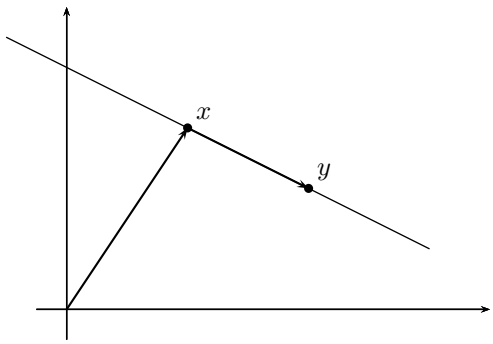
Einen Begriff wollen wir aber noch ansprechen: Es handelt sich um den Begriff der *Konvexität*, der in sehr vielen Bereichen der Mathematik und der Informatik eine wichtige Rolle spielt, u.a. auch in der Optimierung.

4.3 Konvexe Mengen

Es seien $x = (x_1, x_2)$ und $y = (y_1, y_2)$ zwei verschiedene Punkte im $\mathbb{R}^2$. Die Gerade durch x und y ist (in *Parameterform*) gegeben durch

$$z = x + t(y - x) \qquad (t \in \mathbb{R}). \tag{4.1}$$

Wie Sie wissen, nennt man x in dieser Darstellung *Stützvektor* und $y - x$ ist der *Richtungsvektor* (siehe Skizze).



Ein Punkt $z = (z_1, z_2)$ liegt genau dann auf der Geraden durch x und y, wenn es ein $t \in \mathbb{R}$ gibt, für das (4.1) gilt. Gilt dabei $t \geq 0$, so liegt z auf der *Halbgeraden*, die im Punkt x beginnt und y enthält; gilt $t \leq 0$, so liegt z auf der Halbgeraden, die in x beginnt und y nicht enthält; gilt $t = 0$, so liegt der Fall $z = x$ vor, und wenn $t = 1$ gilt, so folgt $z = y$; außerdem können wir feststellen: *Gilt* $0 \leq t \leq 1$, *so liegt* z *auf der Strecke, die die Punkte* x *und* y *verbindet.*

Umformung der Gleichung aus (4.1) ergibt

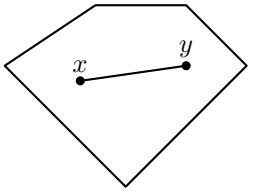
$$z = x + t(y - x) = x + ty - tx = (1 - t)x + ty.$$

Wir können also feststellen, dass ein Punkt z genau dann auf der Verbindungsstrecke der Punkte x und y liegt, wenn gilt:

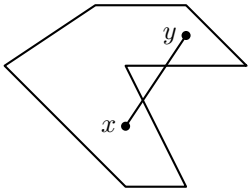
$$z = (1-t)x + ty \qquad (\text{für ein } t \text{ mit } 0 \leq t \leq 1).$$

Definition.
Man nennt eine Teilmenge $K \subseteq \mathbb{R}^2$ *konvex*, falls aus $x, y \in K$ und $0 \leq t \leq 1$ stets $(1-t)x + ty \in K$ folgt.

Mit anderen Worten: K heißt *konvex*, wenn für alle $x, y \in K$ die Verbindungsstrecke zwischen x und y vollständig in K liegt.

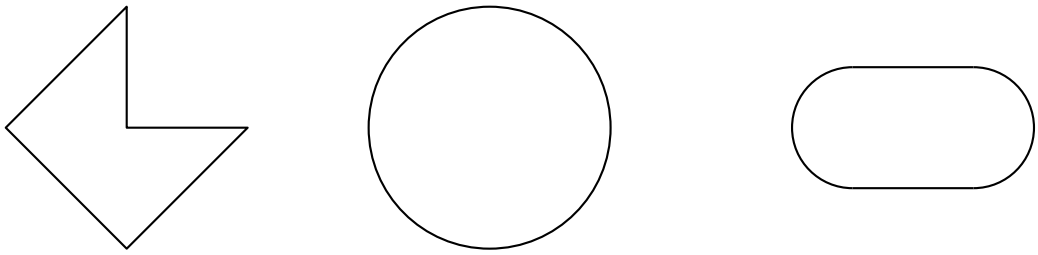


konvex



nicht konvex

Welche der skizzierten Mengen sind konvex?



Definition.

- Für $x, y \in \mathbb{R}^n$ definiert man die *Verbindungsstrecke von x und y* als die Menge aller Punkte $z \in \mathbb{R}^n$, für die es ein $t \in \mathbb{R}$ mit $0 \leq t \leq 1$ gibt, so dass gilt:

$$z = (1-t)x + ty.$$

- Eine Teilmenge $K \subseteq \mathbb{R}^n$ heißt *konvex*, falls für alle $x, y \in K$ gilt: Mit x und y ist auch die Verbindungsstrecke von x und y in K.

Einfache **Beispiele** konvexer Teilmengen des $\mathbb{R}^n$:

(i) jede Hyperebene im $\mathbb{R}^n$ ist konvex;

(ii) jeder Halbraum des $\mathbb{R}^n$ ist konvex;

(iii) $\mathbb{R}^n$ selbst ist konvex;

(iv) die leere Menge ist ebenfalls eine konvexe Teilmenge des $\mathbb{R}^n$.

Hat man es mit einer Menge $\mathcal{M}$ von Teilmengen T des $\mathbb{R}^n$ zu tun, so lässt sich bekanntermaßen der Durchschnitt D dieser Teilmengen bilden: D ist die Menge derjenigen $x \in \mathbb{R}^n$, die in *allen* Teilmengen $T \in \mathcal{M}$ enthalten sind. Man kann D auch so beschreiben:

$$D = \Big\{ x \in \mathbb{R}^n : x \in T \text{ für alle } T \in \mathcal{M} \Big\}.$$

Dasselbe, noch etwas kürzer ausgedrückt:

$$D = \bigcap_{T \in \mathcal{M}} T.$$

Dabei ist es durchaus möglich, dass $\mathcal{M}$ eine *unendliche* Menge ist, d.h., man bildet den Durchschnitt von unendlich vielen Teilmengen des $\mathbb{R}^n$.

Wir interessieren uns hier für Mengen $\mathcal{M}$ von *konvexen* Teilmengen T des $\mathbb{R}^n$. Als eine einfache Folgerung aus der Definition des Begriffs einer konvexen Menge erhält man die folgende Feststellung.

Feststellung.
Gegeben sei eine Menge $\mathcal{M}$ von *konvexen* Teilmengen T des $\mathbb{R}^n$. Dann ist der Durchschnitt D dieser Teilmengen ebenfalls eine konvexe Menge.

Der **Beweis** dieser wichtigen Feststellung ist sehr einfach: Gilt $x, y \in D$, so folgt $x, y \in T$ für alle $T \in \mathcal{M}$. Da alle $T \in \mathcal{M}$ konvex sind, folgt, dass auch die Verbindungsstrecke von x und y in allen $T \in \mathcal{M}$ enthalten ist. Folglich ist diese Verbindungsstrecke auch in D enthalten, wodurch die Konvexität von D nachgewiesen ist. □

Als Folgerung aus (ii) und (iii) sowie der letzten Feststellung erhält man, dass jedes Polyeder $P \subseteq \mathbb{R}^n$ konvex ist.

Wir halten noch einmal ausdrücklich fest:

Feststellung.

- Polyeder sind konvexe Mengen.
- Somit sind die zulässigen Bereiche von LP-Problemen ebenfalls konvex.

4.4 Eine Bemerkung zu konvexen Funktionen

In der Analysis hat man es mit *konvexen Funktionen* zu tun, während es in Abschnitt 4.3 um *konvexe Mengen* ging. Wir wollen kurz auf den Zusammenhang eingehen, der zwischen diesen beiden Begriffen besteht. Hier zunächst die übliche Definition des Begriffs „konvexe Funktion“ (für Funktionen $f : I \to \mathbb{R}$ mit $I \subseteq \mathbb{R}$).

Definition.
Es sei $I \subseteq \mathbb{R}$ ein Intervall und $f : I \to \mathbb{R}$ sei eine Funktion. Man nennt f eine *konvexe Funktion*, falls für alle $x_1, x_2 \in I$ mit $x_1 \leq x_2$ und für alle $t \in [0, 1]$ gilt:

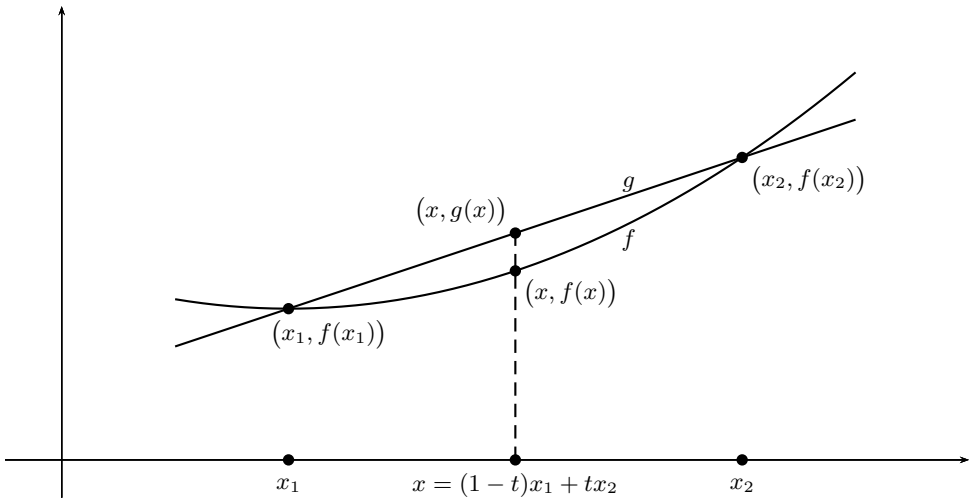
$$f\big((1-t)x_1 + tx_2\big) \leq (1-t)f(x_1) + tf(x_2). \tag{4.2}$$

Um die geometrische Bedeutung der Ungleichung (4.2) zu erkennen, betrachten wir die Funktion g, die die Strecke zwischen den Punkten $(x_1, f(x_1))$ und $(x_2, f(x_2))$ darstellt. In Parameterdarstellung lässt sich diese Strecke wie folgt angeben:

$$\big(x_1, f(x_1)\big) + t\big(x_2 - x_1, f(x_2) - f(x_1)\big) = \big((1-t)x_1 + tx_2, (1-t)f(x_1) + tf(x_2)\big) \quad (\text{für } t \in [0,1]).$$

Für ein $t \in [0, 1]$ setzen wir $x = (1-t)x_1 + tx_2$. Da der Punkt $((1-t)x_1 + tx_2, (1-t)f(x_1) + tf(x_2))$ auf der Strecke g liegt und da $x = (1-t)x_1 + tx_2$ gilt, handelt es sich bei diesem Punkt um den Punkt mit den Koordinaten $(x, g(x))$. Es gilt also $(1-t)f(x_1) + tf(x_2) = g(x)$ und die Ungleichung (4.2) geht über in $f(x) \leq g(x)$.

Die Ungleichung (4.2) besagt also: *Zwischen x_1 und x_2 verläuft der Graph von f niemals oberhalb der Geraden g, die durch die Punkte $(x_1, f(x_1))$ und $(x_2, f(x_2))$ geht* (siehe Zeichnung).



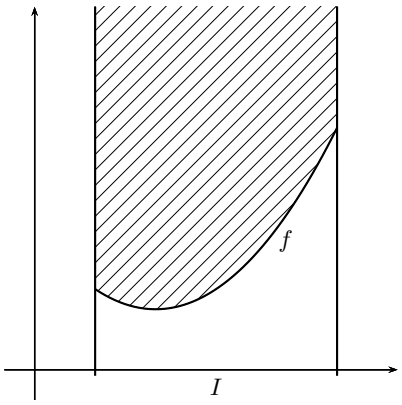
Für eine Funktion $f : I \to \mathbb{R}$ definiert man den Graphen von f bekanntlich als die Menge

$$\Big\{(x, y) \in \mathbb{R}^2 : x \in I, y = f(x)\Big\}.$$

Als *Epigraph* von f bezeichnet man die folgende Menge:

$$\Big\{(x, y) \in \mathbb{R}^2 : x \in I, y \geq f(x)\Big\}.$$

Beim Epigraphen einer Funktion $f : I \to \mathbb{R}$ handelt sich also um die Menge aller Punkte $(x, y) \in \mathbb{R}^2$, für die $x \in I$ gilt und die auf dem Graphen oder oberhalb des Graphen von f liegen.



Zwischen den Begriffen „konvexe Funktion“ und „konvexe Menge“ besteht der folgende Zusammenhang:

Feststellung.
Es sei $I \subseteq \mathbb{R}$ ein Intervall und $f : I \to \mathbb{R}$ sei eine Funktion. Dann gilt: f ist genau dann eine konvexe Funktion, wenn der Epigraph von f eine konvexe Menge ist.

Beweis. Wir bezeichnen den Epigraphen von f mit M, d.h. $M = \big\{(x, y) \in \mathbb{R}^2 : x \in I, y \geq f(x)\big\}$. Klar ist aufgrund der Definitionen: Wenn M konvex ist, so ist f eine konvexe Funktion. Um zu zeigen, dass auch die Umkehrung hiervon gilt, setzen wir nun voraus, dass f eine konvexe Funktion ist. Wir haben zu zeigen, dass M eine konvexe Menge ist. Dementsprechend betrachten wir zwei beliebige Punkte $P_1 = (x_1, y_1)$ und $P_2 = (x_2, y_2)$ aus M sowie einen beliebigen Punkt Q, der auf der Strecke von P_1 nach P_2 liegt. Es

gibt also ein $t \in [0,1]$, so dass gilt:

$$\begin{aligned} Q &= (x_1, y_1) + t(x_2 - x_1, y_2 - y_1) \\ &= \big((1-t)x_1 + tx_2, (1-t)y_1 + ty_2\big). \end{aligned}$$

Zu zeigen ist $Q \in M$, d.h., es ist $(1-t)y_1 + ty_2 \geq f((1-t)x_1 + tx_2)$ zu zeigen. Dies folgt so: Wegen $P_1, P_2 \in M$ gilt $y_1 \geq f(x_1)$ und $y_2 \geq f(x_2)$. Da f konvex ist, gilt außerdem die Ungleichung (4.2). Es folgt

$$\begin{aligned} f\big((1-t)x_1 + tx_2\big) &\leq (1-t)f(x_1) + tf(x_2) \\ &\leq (1-t)y_1 + ty_2, \end{aligned}$$

was zu zeigen war. □

Für diejenigen, die mehr über konvexe Mengen und konvexe Funktionen wissen möchten, sei auf das folgende einführende Lehrbuch hingewiesen:

- N. Lauritzen: *Undergraduate Convexity*. World Scientific (2013).

4.5 Der Begriff der konvexen Hülle

Es sei X eine beliebige Teilmenge des $\mathbb{R}^n$. Wir beginnen mit einer *sehr einfachen Frage*:

Gibt es immer eine konvexe Teilmenge K des $\mathbb{R}^n$, für die $X \subseteq K$ gilt?

Antwort: Ja, selbstverständlich gibt es eine solche Menge K; da $\mathbb{R}^n$ selbst eine konvexe Menge ist, braucht man nur $K = \mathbb{R}^n$ zu wählen.

Nächste Frage: Nun fragen wir nach *einer möglichst kleinen* konvexen Teilmenge K des $\mathbb{R}^n$, für die $X \subseteq K$ gilt.

Antwort: Eine solche möglichst kleine Menge K gibt es ebenfalls immer – man nennt diese Menge die *konvexe Hülle* von X.

Die Definition der konvexen Hülle wird im Folgenden gegeben.

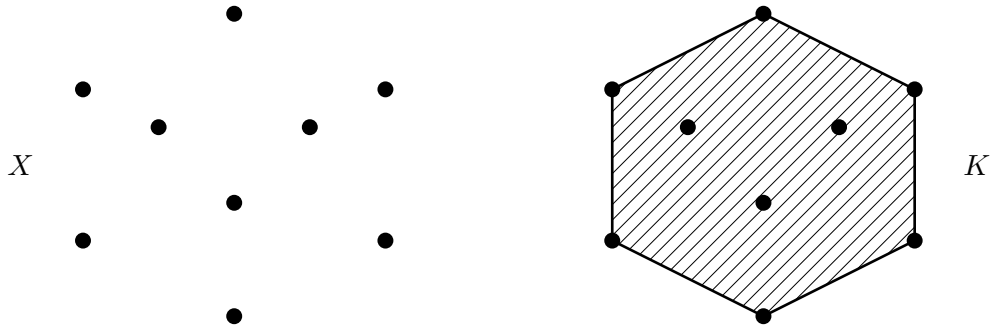
Definition.
Es sei X eine Teilmenge des $\mathbb{R}^n$. Unter der *konvexen Hülle* von X versteht man den Durchschnitt sämtlicher konvexer Teilmengen des $\mathbb{R}^n$, in denen X enthalten ist.

Wir wissen: *Der Durchschnitt von beliebig vielen konvexen Teilmengen des $\mathbb{R}^n$ ist wiederum eine konvexe Menge* (vgl. Abschnitt 4.3). Deshalb gilt für die konvexe Hülle K einer Menge X: K ist eine konvexe Menge, für die $X \subseteq K$ gilt. *Im folgenden Sinne ist K die kleinste konvexe Menge, für die $X \subseteq K$ gilt*: Ist L irgendeine konvexe Menge, für die $X \subseteq L$ gilt, so folgt (nach der Definition der konvexen Hülle), dass K in L enthalten ist.

Beispiele.

1. Die Menge $X \subseteq \mathbb{R}^2$ enthalte nur zwei Punkte, etwa $X = \{x_1, x_2\}$. Dann ist die konvexe Hülle K von X gleich der Strecke, die x_1 und x_2 verbindet.
2. Nun gelte $X \subseteq \mathbb{R}^2$ und $X = \{x_1, x_2, x_3\}$, wobei wir voraussetzen, dass die drei Punkte x_1, x_2, x_3 nicht auf einer Geraden liegen. Dann ist die konvexe Hülle K von X gleich dem Dreieck, das von x_1, x_2 und x_3 gebildet wird – *inklusive des Inneren dieses Dreiecks.*

3. Zur Illustration ein weiteres Beispiel: Links ist eine Teilmenge X abgebildet, die aus neun Punkten des $\mathbb{R}^2$ besteht, rechts ist die konvexe Hülle K von X dargestellt.



Wir nehmen das dritte Beispiel zum Anlass, um auf das folgende sehr bekannte Problem der *Algorithmischen Geometrie* hinzuweisen: Gegeben seien n Punkte des $\mathbb{R}^2$: $P_1 = (x_1, y_1), P_2 = (x_2, y_2), \ldots, P_n = (x_n, y_n)$. Zu bestimmen sind die „Extremalpunkte“ der konvexen Hülle, wobei die ermittelten Extremalpunkte „entgegen dem Uhrzeigersinn“ auszugeben sind. (Was unter dem Begriff „Extremalpunkt“ sowie unter „entgegen dem Uhrzeigersinn“ zu verstehen ist, soll hier nicht definiert werden – intuitiv sollte klar sein, was gemeint ist. Wir begnügen uns mit dem Hinweis, dass es im obigen Beispiel 3 genau sechs Extremalpunkte gibt.) Wer effiziente Algorithmen zur Lösung des Problems kennenlernen möchte, greife (beispielsweise) zu einem der folgenden Bücher:

- Th. Cormen, Ch. Leiserson, R. Rivest, C. Stein: *Algorithmen – Eine Einführung*. Oldenbourg-Verlag (2010, 3. Auflage).
- F. P. Preparata, M. I. Shamos: *Computational Geometry: An Introduction*. Springer (1985).
- M. de Berg, O. Cheong, M. van Kreveld, M. Overmars: *Computational Geometry: Algorithms and Applications*. Springer (2008).

Die zuvor gegebene Definition der konvexen Hülle ist zwar kurz und knapp – und für viele Zwecke auch sehr nützlich –, als „konstruktiv“ kann diese Definition aber nicht angesehen werden. Anders gesagt: *Eine Vorstellung davon, wie man die Elemente von K mithilfe der Elemente von X erhält, wird durch diese Definition nicht geliefert.*

Eine andere Möglichkeit, die konvexe Hülle einer Menge X zu beschreiben, erhält man mithilfe des aus der Linearen Algebra bekannten Begriffs der *Linearkombination*. Dies wird im Folgenden ausgeführt.

Es sei X eine beliebige Teilmenge des $\mathbb{R}^n$; es sei darauf hingewiesen, dass X nicht unbedingt endlich sein muss. Für reelle Zahlen $\lambda_1, \ldots, \lambda_m$ betrachten wir Linearkombinationen

$$\lambda_1 x_1 + \ldots + \lambda_m x_m,$$

wobei die Vektoren $x_1, \ldots, x_m$ aus X stammen sollen. Gilt für eine solche Linearkombination zusätzlich

$$\lambda_1, \ldots, \lambda_m \geq 0 \quad \text{und} \quad \sum_{i=1}^{m} \lambda_i = 1, \tag{4.3}$$

so spricht man von einer *Konvexkombination*.

Man beachte, dass aus (4.3) folgt, dass die Koeffizienten λ_i nicht größer als 1 sein können. *Bei Konvexkombinationen handelt es sich also um spezielle Linearkombinationen. Genauer: Die Koeffizienten einer Konvexkombinationen müssen aus dem Intervall* $[0,1]$ *stammen und sich außerdem zu 1 addieren.*

Beispiele. Gegeben seien die Vektoren $x_1, x_2, x_3 \in \mathbb{R}^n$. Die Linearkombination

$$\frac{1}{2}x_1 + \frac{1}{6}x_2 + \frac{1}{3}x_3$$

ist eine Konvexkombination dieser drei Vektoren und

$$\frac{1}{2}x_1 + \frac{1}{2}x_2 \quad \Big[= \frac{1}{2}x_1 + \frac{1}{2}x_2 + 0x_3\Big]$$

ist ebenfalls eine Konvexkombination von x_1, x_2, x_3. *Keine* Konvexkombination dieser drei Vektoren sind dagegen die folgenden Linearkombinationen:

$$\frac{1}{6}x_1 + \frac{1}{5}x_2 + \frac{2}{3}x_3$$

und

$$\frac{2}{3}x_1 + \frac{2}{3}x_2 - \frac{1}{3}x_3.$$

Ist $X \subseteq \mathbb{R}^n$ gegeben, so bezeichnen wir mit $\widetilde{X}$ die Menge *sämtlicher* Konvexkombination von Vektoren aus X. Der Deutlichkeit halber: In $\widetilde{X}$ sind recht kurze Konvexkombinationen enthalten, wie z.B. $\frac{1}{2}x_1+\frac{1}{2}x_2$ (für $x_1, x_2 \in X$), aber auch längere wie z.B. $\frac{1}{5}x_1 + \frac{2}{5}x_2 + \frac{1}{5}x_3 + \frac{1}{10}x_4 + \frac{1}{10}x_5$ (für $x_1, x_2, x_3, x_4, x_5 \in X$). Ein einzelner Vektor $x_1 \in X$ stellt für sich genommen ebenfalls eine Konvexkombination dar: Es gilt ja

$$x_1 = 1x_1.$$

Die Menge $\widetilde{X}$ enthält also Konvexkombinationen unterschiedlicher Länge und wegen $x = 1x$ gilt $X \subseteq \widetilde{X}$. Man kann $\widetilde{X}$ wie folgt angeben:

$$\widetilde{X} = \Big\{\sum_{i=1}^{m} \lambda_i x_i : m \geq 1,\ x_i \in X \text{ und } \lambda_i \geq 0\ (i = 1, \ldots, m),\ \sum_{i=1}^{m} \lambda_i = 1\Big\}.$$

Der Zusammenhang zwischen $\widetilde{X}$ und der konvexen Hülle K von X besteht nun einfach darin, dass es sich um ein und dieselbe Menge handelt. Dies ist der Inhalt des folgenden Satzes.

Satz.
Für $X \subseteq \mathbb{R}^n$ sei K die konvexe Hülle von X und mit $\widetilde{X}$ sei die Menge aller Konvexkombinationen von Elementen aus X bezeichnet. Dann gilt

$$K = \widetilde{X}. \tag{4.4}$$

Beweis. Zum Nachweis von (4.4) sind zwei Dinge zu zeigen:

(I) $\widetilde{X} \subseteq K$ und

(II) $K \subseteq \widetilde{X}$.

Wir kümmern uns zunächst um (I), d.h. wir zeigen, dass jede Konvexkombination $x \in \widetilde{X}$ ein Element der konvexen Hülle K ist. Es sei also $x \in \widetilde{X}$, d.h., x besitzt die Darstellung

$$x = \sum_{i=1}^{m} \lambda_i x_i = \lambda_1 x_1 + \ldots + \lambda_m x_m, \tag{4.5}$$

wobei $m \geq 1$ und $x_i \in X$ $(i = 1, \ldots, m)$ gilt. Außerdem gilt für die Koeffizienten λ_i:

$$\lambda_i \geq 0 \quad (i = 1, \ldots, m) \tag{4.6}$$

und

$$\sum_{i=1}^{m} \lambda_i = 1. \tag{4.7}$$

Zu zeigen ist $x \in K$. *Dies geschieht durch Induktion nach* m. Für $m = 1$ ist dies offensichtlich: Im Fall $m = 1$ lautet (4.5) nämlich $x = \lambda_1 x_1$ (für $x_1 \in X$) und wegen (4.7) muss $\lambda_1 = 1$ gelten. Es folgt $x = x_1$ und somit $x \in X$; wegen $X \subseteq K$ erhält man $x \in K$.

Es sei nun $m \geq 2$; wir nehmen an, dass die Behauptung für Konvexkombinationen mit weniger als m Summanden richtig ist (Induktionsannahme). Es sei x wie in (4.5) gegeben. Falls $\lambda_m = 1$ gilt, so folgt (aufgrund von (4.6) und (4.7)) $\lambda_i = 0$ für $i = 1, \ldots, m-1$, woraus $x = x_m$ folgt. Also gilt $x \in X$ und somit auch $x \in K$. Wir dürfen also annehmen, dass $\lambda_m < 1$ gilt. Dies erlaubt uns, Koeffizienten $\lambda'_1, \ldots, \lambda'_{m-1}$ wie folgt zu definieren:

$$\lambda'_i = \frac{\lambda_i}{1 - \lambda_m} \quad (i = 1, \ldots, m-1).$$

Man beachte, dass $\lambda'_i \geq 0$ $(i = 1, \ldots, m-1)$ gilt und dass sich die λ'_i zu 1 summieren, da (aufgrund von (4.7)) $\lambda_1 + \ldots + \lambda_{m-1} = 1 - \lambda_m$ gilt. Also handelt es sich bei

$$x' = \lambda'_1 x_1 + \ldots + \lambda'_{m-1} x_{m-1}$$

um eine Konvexkombination von $x_1, \ldots, x_{m-1}$. Nach Induktionsannahme folgt somit $x' \in K$. Außerdem gilt $x = (1 - \lambda_m)x' + \lambda_m x_m$ sowie $x_m \in K$ (wegen $x_m \in X$ und $X \subseteq K$). Folglich liegt x auf einer Strecke, die zwei Punkte von K verbindet, woraus wegen der Konvexität von K folgt, dass $x \in K$ gilt.

Es bleibt (II) nachzuweisen. *Hierzu zeigen wir zunächst, dass es sich bei* $\widetilde{X}$ *um eine konvexe Menge handelt.* Zu diesem Zweck seien x und y zwei Elemente aus $\widetilde{X}$. Da $x, y \in \widetilde{X}$ gilt, lassen sich x und y wie folgt darstellen:

$$x = \sum_{i=1}^{m} \lambda_i x_i \quad \text{und} \quad y = \sum_{i=1}^{\ell} \mu_i y_i,$$

wobei alle x_i und y_i aus X stammen und außerdem $m \geq 1$, $\ell \geq 1$, $\lambda_i \geq 0$ $(i = 1, \ldots, m)$, $\mu_i \geq 0$ $(i = 1, \ldots, \ell)$ gilt sowie

$$\sum_{i=1}^{m} \lambda_i = 1 \quad \text{und} \quad \sum_{i=1}^{\ell} \mu_i = 1.$$

Es sei $z = (1 - t)x + ty$ für $t \in [0, 1]$, d.h., z ist ein Punkt auf der Strecke, die x und y verbindet. Um nachzuweisen, dass $\widetilde{X}$ konvex ist, haben wir zu zeigen, dass $z \in \widetilde{X}$ gilt. Dies ergibt sich aus der folgenden einfachen Rechnung:

$$\begin{aligned} z &= (1-t)x + ty \\ &= (1-t)\sum_{i=1}^{m} \lambda_i x_i + t\sum_{i=1}^{\ell} \mu_i y_i \\ &= \sum_{i=1}^{m} (1-t)\lambda_i x_i + \sum_{i=1}^{\ell} t\mu_i y_i, \end{aligned}$$

wobei für die Koeffizienten $(1-t)\lambda_1, \ldots, (1-t)\lambda_m, t\mu_1, \ldots, t\mu_\ell$ gilt:

$$\begin{aligned} \sum_{i=1}^{m} (1-t)\lambda_i + \sum_{i=1}^{\ell} t\mu_i &= (1-t)\sum_{i=1}^{m} \lambda_i + t\sum_{i=1}^{\ell} \mu_i \\ &= (1-t) + t \\ &= 1. \end{aligned}$$

Die in der Darstellung von z auftretenden Koeffizienten addieren sich also zu 1; außerdem sind diese Koeffizienten alle nichtnegativ. Folglich ist z eine Konvexkombination von Elementen aus X, d.h., $z \in \widetilde{X}$.

Damit ist gezeigt, dass $\widetilde{X}$ konvex ist.

Erfreulicherweise sind wir damit bereits fertig: Es gilt nämlich – wie wir wissen – $X \subseteq \widetilde{X}$. Da K in jeder konvexen Menge enthalten ist, die X umfasst (vgl. Seite 48), folgt $K \subseteq \widetilde{X}$. □

Häufig wird die konvexe Hülle einer Menge $X \subseteq \mathbb{R}^n$ mit

$$\operatorname{conv}(X)$$

bezeichnet. Diese Bezeichnung wollen wir ebenfalls verwenden.

Zusammenfassung. Wir haben die konvexe Hülle $\operatorname{conv}(X)$ einer Menge $X \subseteq \mathbb{R}^n$ auf zwei sehr unterschiedliche Arten beschrieben, die wir als Beschreibung „von außen“ bzw. „von innen“ bezeichnen wollen.

(I) *Beschreibung der Menge* $\operatorname{conv}(X)$ *von außen*:

$\operatorname{conv}(X)$ ist gleich dem Durchschnitt aller konvexen Teilmengen des $\mathbb{R}^n$, die die Menge X enthalten.

(II) *Beschreibung der Menge* $\operatorname{conv}(X)$ *von innen*:

$\operatorname{conv}(X)$ ist gleich der Menge aller Konvexkombination von Elementen von X.

Wir haben nachgewiesen, dass beide Beschreibungen *äquivalent* sind.

Es ist sehr nützlich, zwei verschiedene Beschreibungen von $\operatorname{conv}(X)$ zur Verfügung zu haben, da dies mit einem *Gewinn an Flexibilität* einhergeht: Man kann immer mit der Beschreibung arbeiten, die besser zum jeweiligen Kontext passt. Beide Beschreibungen werden in der Literatur häufig verwendet.

4.6 Konvexe Kegel

Wir wissen: Ist $X \subseteq \mathbb{R}^n$ gegeben, so handelt es sich bei $\operatorname{conv}(X)$ um die *kleinste* konvexe Menge, die X umfasst. Wir wollen zusätzlich eine andere Art von konvexer Menge kurz ansprechen, die ebenfalls X umfasst und ebenso wichtig ist wie die konvexe Hülle $\operatorname{conv}(X)$: Die Rede ist von der *konischen Hülle* von X, die man mit

$$\operatorname{cone}(X)$$

bezeichnet.

Hier ist die Definition der konischen Hülle (für $X \neq \emptyset$):

Definition.
Es sei $X \subseteq \mathbb{R}^n$, $X \neq \emptyset$. Unter der *konischen Hülle* von X versteht man die Menge aller Linearkombinationen

$$\lambda_1 x_1 + \ldots + \lambda_m x_m,$$

für die gilt: $m \geq 1$, $x_i \in X$ und $\lambda_i \geq 0$ $(i = 1, \ldots, m)$.

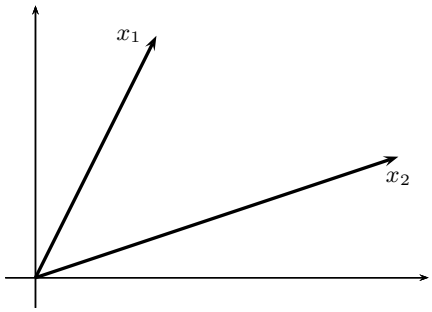
Unter Verwendung der Bezeichnung $\operatorname{cone}(X)$ gilt also (für $X \neq \emptyset$):

$$\operatorname{cone}(X) = \Big\{ \sum_{i=1}^{m} \lambda_i x_i : m \geq 1, x_i \in X \text{ und } \lambda_i \geq 0 \ (i = 1, \ldots, m) \Big\}.$$

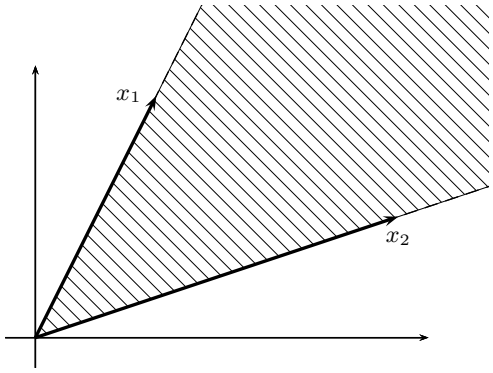
Im Unterschied zur Definition der konvexen Hülle fehlt also die Bedingung $\sum_{i=1}^{m} \lambda_i = 1$, während die Bedingung $\lambda_i \geq 0$ $(i = 1, \ldots, m)$ beibehalten wurde. (Ganz nebenbei gesagt: Ist $X = \emptyset$, so ergibt sich aus der Definition der konvexen Hülle, dass $\operatorname{conv}(X) = \emptyset$ gilt. Im Gegensatz dazu definiert man für $X = \emptyset$ zusätzlich zur obigen Definition: $\operatorname{cone}(X) = \{0\}$.)

Beispiele

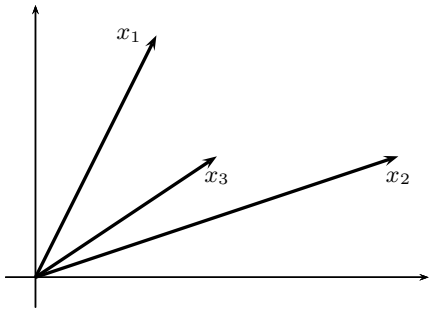
1. Die Vektoren $x_1, x_2 \in \mathbb{R}^2$ seien wie in der Zeichnung gegeben:



In der nachfolgenden Zeichnung ist die konische Hülle der Menge $X = \{x_1, x_2\}$ dargestellt:



2. Nun seien drei Vektoren des $\mathbb{R}^2$ wie in der folgenden Zeichnung gegeben:



Wie sieht die konische Hülle $\text{cone}(X)$ für $X = \{x_1, x_2, x_3\}$ aus?

3. Können Sie drei Vektoren $x_1, x_2, x_3 \in \mathbb{R}^2$ angeben, für die $\text{cone}(\{x_1, x_2, x_3\}) = \mathbb{R}^2$ gilt?

4. Können Sie Vektoren $x_1, x_2 \in \mathbb{R}^2$ angeben, für die $\text{cone}(\{x_1, x_2\})$ ein Halbgerade ist?

5. Gibt es auch $x_1, x_2 \in \mathbb{R}^2$, so dass $\text{cone}(\{x_1, x_2\})$ eine Gerade ist?

Zur Übung: Beweisen Sie, dass $\text{cone}(X)$ eine konvexe Menge ist, die X umfasst.

Abschließend noch *zwei Sprechweisen* (und ein Hinweis).

Ist eine Menge $X \subseteq \mathbb{R}^n$ gegeben, so nennt man die Menge

$$\text{cone}(X)$$

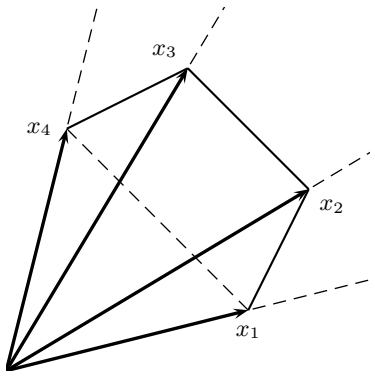
auch den von X erzeugten *konvexen Kegel* (engl. *convex cone*).

Eine Linearkombination

$$\lambda_1 x_1 + \ldots + \lambda_m x_m,$$

für die $\lambda_i \geq 0$ $(i = 1, \ldots, m)$ gilt, nennt man auch eine *konische Linearkombination*.

Hinweis: Häufig hat man es mit dem Fall zu tun, dass X eine endliche Menge ist. *Dies ist für uns der wichtigste Fall.* Man spricht in diesem Fall auch von einem *endlich erzeugten konvexen Kegel*. Dass solche Kegel nicht „rund", sondern „kantig" sind, haben Sie vermutlich längst bemerkt. Zur Illustration ist in der folgenden Zeichnung ein konvexer Kegel im $\mathbb{R}^3$ abgebildet, der von einer Menge $\{x_1, x_2, x_3, x_4\}$ von vier Vektoren erzeugt wird; es handelt sich also um den endlich erzeugten konvexen Kegel $\text{cone}(\{x_1, x_2, x_3, x_4\})$.



Beim abgebildeten konvexen Kegel handelt es sich um ein Polyeder. Die einzige Ecke dieses Polyeders ist der Ursprung („Spitze des konvexen Kegels"); neben seinen inneren Punkten umfasst das Polyeder die vier Halbgeraden $\lambda x_1, \ldots, \lambda x_4$ $(\lambda \geq 0)$ und außerdem besitzt das Polyeder vier Flächen.

5 Wie schnell ist das Simplexverfahren?

5.1 Beispiele von Klee und Minty

Im Buch von Chvátal wird berichtet, dass empirische Untersuchungen Folgendes ergeben haben: In der Praxis auftretende LP-Probleme

$$\begin{aligned}
&\text{maximiere } \sum_{j=1}^{n} c_j x_j \\
&\text{unter den Nebenbedingungen} \\
&\qquad \sum_{j=1}^{n} a_{ij} x_j \ \leq b_i \qquad (i = 1, \ldots, m) \\
&\qquad\qquad x_j \ \geq \ 0 \qquad (j = 1, \ldots, n).
\end{aligned}$$

benötigen häufig weniger als $\frac{3}{2}m$ Iterationen – und nur sehr selten werden mehr als $3m$ Iterationen benötigt. Außerdem wird der folgende empirische Befund erwähnt (siehe Chvátal, Kapitel 4): Bei festem m und wachsendem n nimmt die Anzahl der Iterationen nur sehr langsam zu; es wurde ein Wachstum ungefähr von der Größenordnung $\log n$ beobachtet. *Die Ergebnisse dieser empirischen Untersuchungen stehen im Einklang mit der Tatsache,* ***dass sich das Simplexverfahren seit mehr als 60 Jahren in der Praxis hervorragend bewährt hat.***

Auf der anderen Seite sind aber auch Beispiele bekannt, die eine außerordentlich hohe Anzahl von Iterationen erfordern. Derartige Beispiele wurden erstmals im Jahre 1972 in einer Arbeit von Klee und Minty vorgestellt. Bevor wir uns dieses Beispiel anschauen, sei an eine Pivotierungsregel erinnert, die wir bereits häufig verwendet haben.

Stehen mehrere Kandidaten für die Wahl einer Eingangsvariable zur Verfügung, so entscheide man sich für eine Variable, deren Koeffizient $\overline{c}_j$ in der z-Zeile des aktuellen Tableaus möglichst groß ist. ($\star$)

Diese Regeln nennt man die *Regel vom größten Koeffizienten* (engl. *largest-coefficient rule*).

Auch im Folgenden soll immer, wenn nichts anderes gesagt ist, die Regel vom größten Koeffizienten zugrunde liegen.

Hier sind nun die **Beispiele von Klee und Minty**:

$$\begin{aligned}
&\text{maximiere } \sum_{j=1}^{n} 10^{n-j} x_j \\
&\text{unter den Nebenbedingungen} \\
&\qquad 2 \cdot \sum_{j=1}^{i-1} 10^{i-j} x_j \ + \ x_i \ \leq 100^{i-1} \qquad (i = 1, \ldots, n) \\
&\qquad\qquad x_j \ \geq \qquad 0 \qquad (j = 1, \ldots, n).
\end{aligned}$$

Klee und Minty haben in ihrer Arbeit nachgewiesen, dass für dieses Beispiel $2^n - 1$ *Iterationen* nötig sind, bis das Simplexverfahren terminiert (bei Anwendung der Regel vom größten Koeffizienten).

Wir schauen uns den Fall $n = 3$ näher an:

$$\begin{aligned}
&\text{maximiere } 100x_1 + 10x_2 + x_3 \\
&\text{unter den Nebenbedingungen} \\
&\qquad\begin{aligned}
x_1 &\leq 1 \\
20x_1 + x_2 &\leq 100 \\
200x_1 + 20x_2 + x_3 &\leq 10000 \\
x_1, x_2, x_3 &\geq 0.
\end{aligned}
\end{aligned}$$

Unter Benutzung der Regel ($\star$) erhalten wir die folgenden Tableaus:
Starttableau:

$$\begin{aligned}
x_4 &= 1 - x_1 \\
x_5 &= 100 - 20x_1 - x_2 \\
x_6 &= 10000 - 200x_1 - 20x_2 - x_3 \\
\hline
z &= 100x_1 + 10x_2 + x_3.
\end{aligned}$$

Nach der 1. Iteration ergibt sich:

$$\begin{aligned}
x_1 &= 1 - x_4 \\
x_5 &= 80 + 20x_4 - x_2 \\
x_6 &= 9800 + 200x_4 - 20x_2 - x_3 \\
\hline
z &= 100 - 100x_4 + 10x_2 + x_3.
\end{aligned}$$

Nach der 2. Iteration ergibt sich:

$$\begin{aligned}
x_1 &= 1 - x_4 \\
x_2 &= 80 + 20x_4 - x_5 \\
x_6 &= 8200 - 200x_4 + 20x_5 - x_3 \\
\hline
z &= 900 + 100x_4 - 10x_5 + x_3.
\end{aligned}$$

Nach der 3. Iteration ergibt sich:

$$\begin{aligned}
x_4 &= 1 - x_1 \\
x_2 &= 100 - 20x_1 - x_5 \\
x_6 &= 8000 + 200x_1 + 20x_5 - x_3 \\
\hline
z &= 1000 - 100x_1 - 10x_5 + x_3.
\end{aligned}$$

Nach der 4. Iteration ergibt sich:

$$\begin{aligned}
x_4 &= 1 - x_1 \\
x_2 &= 100 - 20x_1 - x_5 \\
x_3 &= 8000 + 200x_1 + 20x_5 - x_6 \\
\hline
z &= 9000 + 100x_1 + 10x_5 - x_6.
\end{aligned}$$

Nach der 5. Iteration ergibt sich:

$$\begin{aligned} x_1 &= 1 - x_4 \\ x_2 &= 80 + 20x_4 - x_5 \\ x_3 &= 8200 - 200x_4 + 20x_5 - x_6 \\ \hline z &= 9100 - 100x_4 + 10x_5 - x_6. \end{aligned}$$

Nach der 6. Iteration ergibt sich:

$$\begin{aligned} x_1 &= 1 - x_4 \\ x_5 &= 80 + 20x_4 - x_2 \\ x_3 &= 9800 + 200x_4 - 20x_2 - x_6 \\ \hline z &= 9900 + 100x_4 - 10x_2 - x_6. \end{aligned}$$

Nach der 7. Iteration ergibt sich:

$$\begin{aligned} x_4 &= 1 - x_1 \\ x_5 &= 100 - 20x_1 - x_2 \\ x_3 &= 10000 - 200x_1 - 20x_2 - x_6 \\ \hline z &= 10000 - 100x_1 - 10x_2 - x_6. \end{aligned}$$

Beobachtung: Hätten wir in der 1. Iteration nicht x_1, sondern x_3 als Eingangsvariable gewählt, so wären wir schon nach einem Schritt fertig gewesen. *Es stellt sich also unter anderem die Frage nach alternativen Pivotierungsregeln.*

5.2 Alternative Pivotierungsregeln

Besonders naheliegend ist die folgende Pivotierungsregel, die man die *Regel vom größten Zuwachs* (engl. *largest-increase rule*) nennt. Nach dieser Regel wird der Kandidat für die Aufnahme in die Basis so gewählt, dass ein möglichst großer Zuwachs der Zielfunktion z dabei herauskommt.

Während die Regel vom größten Koeffizienten sehr einfach zu handhaben ist, ist die Regel vom größten Zuwachs deutlich rechenaufwendiger. Außerdem gilt (was auf den ersten Blick etwas überraschend sein mag): *Auch für die Regel vom größten Zuwachs lassen sich Beispiele angeben, die in Bezug auf diese Regel eine ähnliche Rolle spielen, wie die Klee-Minty-Beispiele für die largest-coefficient rule.* Derartige Beispiele wurden 1973 von R. G. Jeroslow vorgestellt.

Eine weitere Pivotierungsregel ist uns bereits in Kapitel 3 begegnet: die *Regel vom kleinsten Index* (*Blandsche Regel*, engl. *smallest-subscript rule* oder *Bland's rule*).

Weitere Pivotierungsregeln werden im Buch von Matoušek und Gärtner diskutiert. Für alle dort aufgeführten Regeln (einschließlich der Blandschen Regel) gilt: Es existieren Beispiele, die (ähnlich wie die Klee-Minty-Beispiele) zu einer außerordentlich hohen Zahl von Iterationen führen. Vielfach ist versucht worden, eine Pivotierungsregel zu finden, für die dies nicht der Fall ist – bislang ohne Erfolg. Bei der Frage, ob es eine Pivotierungsregel gibt, durch die das Simplexverfahren zu einem polynomiellen Algorithmus wird, handelt es sich um ein interessantes *offenes Problem.*

Diese Bemerkungen stehen im Übrigen keineswegs im Widerspruch zu der Tatsache, dass sich die Simplexmethode in der Praxis hervorragend bewährt hat: Die Beispiele zeigen ja nur, dass ***im schlechtesten Fall*** *außerordentlich viele Iterationen nötig sind.*

Keine der bekannten Pivotierungsregeln führt zu einem Algorithmus mit polynomieller Laufzeit. Es war lange Zeit offen, ob ein solcher Algorithmus zur Lösung von LP-Problemen überhaupt existiert – bis *L. G. Khachiyan* im Jahre 1979 einen polynomiellen Algorithmus zur Lösung von LP-Problemen vorstellte, der unter dem Namen *Ellipsoid-Methode* bzw. als *Khachiyan-Algorithmus* bekannt wurde. *Das war damals eine Sensation, über die sogar die New York Times auf ihrer ersten Seite berichtete.*

Ein weiterer Algorithmus mit polynomieller Laufzeit wurde ca. 5 Jahre später von *N. Karmarkar* vorgestellt; der Algorithmus von Karmarkar stellt ein Beispiel einer ganzen Klasse von Algorithmen zur Lösung von LP-Problemen dar, die man *Innere-Punkte-Algorithmen* nennt. In Kapitel 14 werden wir auf die Ellipsoid-Methode und auf Innere-Punkte-Methoden genauer eingehen und die Grundideen dieser Algorithmen vorstellen. Dabei werden wir uns an der Darstellung im folgenden Lehrbuch orientieren:

- Matoušek/Gärtner: *Understanding and Using Linear Programming*. Springer-Verlag.

Wir beenden Kapitel 5 mit einem Auszug aus dem Buch von Matoušek und Gärtner.

One of the key pieces of knowledge about linear programming that one should remember forever is this:

A linear program is efficiently solvable, both in theory and in practice.

- *In practice*, a number of software packages are available. They can handle inputs with several thousands of variables and constraints. Linear programs with a special structure, for example, with a small number of nonzero coefficients in each constraint, can often be managed even with a much larger number of variables and constraints.
- *In theory*, algorithms have been developed that provably solve each linear program in time bounded by a certain polynomial function of the input size. The input size is measured as the total number of bits needed to write down all coefficients in the objective function and in all the constraints.

These two statements summarize the results of long and strenuous research, and efficient methods for linear programming are not simple.

Bevor wir nun in Kapitel 7 mit dem zentralen Thema *Dualität* fortfahren, wollen wir in Kapitel 6 einen *Eindruck von den vielfältigen Einsatz- und Anwendungsmöglichkeiten der Linearen Programmierung* gewinnen.

Zu diesem Zweck schauen wir uns Beispiele aus verschiedenen Lehrbüchern an.

6 Einige Anwendungsbeispiele

In diesem Abschnitt werden Beispiele aus unterschiedlichen Lehrbüchern zusammengestellt und (teilweise) kommentiert. *Auf diese Art sollen Sie Einblicke in die vielfältigen Anwendungsmöglichkeiten der Linearen Programmierung erhalten.*

In der *1. Vorlesung* haben wir als einführendes Beispiel ein *Diätproblem* betrachtet, das Ihnen möglicherweise etwas speziell vorkam.

Das Problem ist jedoch weit weniger speziell, als es auf den ersten Blick erscheint: Viele Probleme aus der Praxis, die gar nichts mit gesunder oder preiswerter Ernährung zu tun haben, entpuppen sich bei genauerem Hinsehen ebenfalls als „Diätprobleme“. Um dies zu erkennen, betrachten wir das folgende Beispiel, das wir *das (allgemeine) Diätproblem* nennen[1].

6.1 Das allgemeine Diätproblem

Example 1 (The diet problem). How can we determine the most economical diet that satisfies the basic minimum nutritional requirements for good health? Such a problem might, for example, be faced by the dietician of a large army. We assume that there are available at the market n different foods and that the jth food sells at a price c_j per unit. In addition there are m basic nutritional ingredients and, to achieve a balanced diet, each individual must receive at least b_i units of the ith nutrient per day. Finally, we assume that each unit of food j contains a_{ij} units of the ith nutrient.

If we denote by x_j the number of units of food j in the diet, the problem then is to select the x_j's to minimize the total cost

$$c_1x_1 + \ldots + c_nx_n$$

subject to the nutritional constraints

$$\begin{aligned} a_{11}x_1 + \ldots + a_{1n}x_n &\geq b_1 \\ &\vdots \\ a_{m1}x_1 + \ldots + a_{mn}x_n &\geq b_m \end{aligned}$$

and the nonnegativity constraints

$$x_1 \geq 0, \ldots, x_n \geq 0$$

on the food quantities.

Ein Unterschied zwischen dem allgemeinen Diätproblem und Pauls Problem aus der 1. Vorlesung fällt auf: In Pauls Problem gab es obere Schranken für jedes Lebensmittel; im Einzelnen lauten diese:

$$\begin{aligned} x_1 &\leq 4 \\ x_2 &\leq 3 \\ x_3 &\leq 2 \\ x_4 &\leq 8 \\ x_5 &\leq 2 \\ x_6 &\leq 2. \end{aligned}$$

[1] Dieses Beispiel stammt aus D. Luenberger, Yinyu Ye: *Linear and Nonlinear Programming*. Springer (2008, 3. Auflage). *Ziel des Abschnitts 6 ist ebenfalls, Sie auf interessante Lehrbücher aufmerksam zu machen.*

Im allgemeinen Diätproblem kommen dagegen keine oberen Schranken vor; deshalb wollen wir zwischen den folgenden Problemen unterscheiden:

- dem allgemeinen *Diätproblem* (d.h. ohne obere Schranken)
- dem allgemeinen *Diätproblem mit oberen Schranken.*

Um zu erläutern, welcher Zusammenhang zwischen diesen beiden Varianten des Diätproblems besteht, betrachten wir noch einmal das Beispiel von Paul und seinen Lebensmitteln.

Wir stellen uns vor, dass Paul sein Problem *zunächst ohne obere Schranken* formuliert und auch gelöst hat (per Hand oder mithilfe eines der gängigen Softwarepakete). Möglicherweise wäre er dann mit dem Ergebnis nicht zufrieden gewesen, da der Speiseplan zu einseitig ausgefallen wäre. Um dies zu ändern, würde Paul dann *nachträglich* die zusätzlichen Nebenbedingungen hinzunehmen.

Die nachträgliche Hinzunahme von zusätzlichen Nebenbedingungen ist nichts Seltenes – im Gegenteil: *Es handelt sich um einen typischen Vorgang, der in der Praxis häufig vorkommt.* Das liegt daran, dass bei der Lösung eines LP-Problems natürlich nur diejenigen Nebenbedingungen Berücksichtigung finden, die explizit formuliert wurden. Das kann dazu führen, dass ein Anwender mit dem Ergebnis unzufrieden ist, da er noch zusätzliche Nebenbedingungen im Hinterkopf hatte. Dann kommt es darauf an, geeignete zusätzliche Nebenbedingungen explizit zu formulieren und einen weiteren Durchlauf zu starten.

Das nächste Beispiel, das wir uns anschauen, übernehmen wir aus dem bekannten Lehrbuch

- Th. Cormen, Ch. Leiserson, R. Rivest, C. Stein: *Algorithmen – Eine Einführung.* Oldenbourg-Verlag (2010, 3. Auflage).

6.1.1 Ein Beispiel aus der Politik

Nehmen Sie an, Sie wären ein Politiker, der eine Wahl gewinnen möchte. Ihr Wahlkreis besteht aus drei Typen von Gebieten – urbanen, suburbanen und ländlichen. Diese Gebiete haben 100 000, 200 000 beziehungsweise 50 000 registrierte Wahlberechtigte. Wenngleich nicht alle registrierte Wähler wirklich zur Wahlurne gehen, ist es Ihr Ziel, dass wenigstens die Hälfte der registrierten Wähler eines jeden Gebietstypen Ihnen ihre Stimme gibt, damit Sie effektiv regieren können. Sie sind ehrlich und würden niemals politische Ideen unterstützen, an die Sie nicht glauben. Sie erkennen jedoch, dass bestimmte Themen in bestimmten Gebieten besonders dazu geeignet sind, Stimmen zu gewinnen. Ihre Hauptthemen sind der Bau neuer Straßen, Sicherheitspolitik, Beihilfen für die Landwirtschaft und eine Mineralölsteuer für die Verbesserung des öffentlichen Nahverkehrs. Aufgrund der Untersuchungen Ihrer Wahlkampfberater können Sie für jedes Ihrer Wahlkampfthemen einschätzen, wie viele Stimmen Sie in jeder Bevölkerungsschicht verlieren oder gewinnen, wenn Sie jeweils 1 000 Dollar an Werbemitteln für ein Thema einsetzen. Diese Informationen sind in der Tabelle in Abbildung 29.1 enthalten.

Wahlkampfthema	urban	suburban	ländlich
Straßenbau	-2	5	3
Sicherheit	8	2	-5
Landwirtschaftsbeihilfe	0	0	10
Mineralölsteuer	10	0	-2

Abbildung 29.1: Die Auswirkungen von Wahlkampftaktiken auf die Wähler. Jeder Eintrag gibt die Anzahl der Wähler in Tausenden aus den urbanen, suburbanen und ländlichen Gebieten an, die durch den Einsatz von 1 000 Dollar Werbemitteln für ein bestimmtes Thema gewonnen werden können. Negative Einträge geben Stimmen an, die verloren gehen würden.

Jeder Eintrag dieser Tabelle gibt die Anzahl der Stimmberechtigten (in Tausenden) aus urbanen, suburbanen beziehungsweise ländlichen Gebieten an, die durch die Aufwendung von 1000 Dollar an Werbemitteln für ein bestimmtes Wahlkampfthema gewonnen werden können. Negative Einträge bedeuten, dass

Stimmen verloren gehen würden. Ihre Aufgabe ist es, den minimalen Geldbetrag zu bestimmen, den Sie aufwenden müssen, um 50 000 Stimmen in den urbanen Gebieten, 100 000 Stimmen in den suburbanen Gebieten und 25 000 Stimmen in den ländlichen Gebieten zu gewinnen.

Durch Ausprobieren (engl. *trial and error*) können Sie eine Strategie finden, mit der Sie die erforderlichen Stimmen bekommen, aber eine solche Strategie muss nicht die kostengünstigste sein. Sie könnten zum Beispiel in die Werbekampagnen für den Straßenbau 20 000 Dollar, für die Sicherheitspolitik 0 Dollar, für Landwirtschaftsbeihilfen 4 000 Dollar und für die Mineralölsteuer 9 000 Dollar stecken. In diesem Fall bekommen Sie $20(-2)+0(8)+4(0)+9(10) = 50$ Tausend Stimmen in den urbanen Gebieten, $20(5)+0(2)+4(0)+9(0) = 100$ Tausend Stimmen in den suburbanen Gebieten und $20(3)+0(-5)+4(10)+9(-2) = 82$ Tausend Stimmen in den ländlichen Gebieten. Sie würden in den urbanen und suburbanen Gebieten genau die gewünschte Anzahl von Stimmen und in den ländlichen Gebieten mehr Stimmen als notwendig bekommen. (Tatsächlich haben Sie in den ländlichen Gebieten sogar mehr Stimmen bekommen, als es Wähler gibt!) Um diese Stimmen zu bekommen, müssten Sie $20 + 0 + 4 + 9 = 33$ Tausend Dollar für Werbung bezahlen.

Natürlich würden Sie sich fragen, ob Ihre Strategie die bestmögliche ist, d.h., ob Sie Ihre Ziele mit weniger Werbeaufwand hätten erreichen können. Weiteres Ausprobieren könnte Ihnen helfen, diese Frage zu beantworten. Aber hätten Sie nicht lieber eine systematische Methode, um derartige Fragen zu beantworten? Um eine solche Methode zu entwickeln, werden wir die Frage mathematisch formulieren. Wir führen vier Variablen ein:

- x_1 ist die Anzahl der Dollar in Tausenden, die für die Kampagne für den Straßenbau ausgegeben werden,
- x_2 ist die Anzahl der Dollar in Tausenden, die für die Kampagne für die Sicherheitspolitik ausgegeben werden,
- x_3 ist die Anzahl der Dollar in Tausenden, die für die Kampagne für Landwirtschaftsbeihilfen ausgegeben werden,
- x_4 ist die Anzahl der Dollar in Tausenden, die für die Werbung für die Mineralölsteuer ausgegeben werden.

Die Forderung, mindestens 50 000 Stimmen in den urbanen Gebieten zu gewinnen, können wir durch die Ungleichung

$$-2x_1 + 8x_2 + 0x_3 + 10x_4 \geq 50 \tag{29.1}$$

ausdrücken. Entsprechend schreiben wir die Forderung, mindestens 100 000 Stimmen in den suburbanen und mindestens 25 000 Stimmen in den ländlichen Gebieten zu gewinnen, in der Form

$$5x_1 + 2x_2 + 0x_3 + 0x_4 \geq 100 \tag{29.2}$$

und

$$3x_1 - 5x_2 + 10x_3 - 2x_4 \geq 25. \tag{29.3}$$

Jede Belegung der Variablen x_1, x_2, x_3, x_4, die die Ungleichungen (29.1)-(29.3) erfüllt, führt zu einer Strategie, mit der Sie in jeder der Bevölkerungsgruppen die notwendige Stimmenanzahl bekommen. Um die Kosten so gering wie möglich zu halten, wollen Sie den Werbeaufwand, d.h. den Ausdruck

$$x_1 + x_2 + x_3 + x_4 \tag{29.4}$$

minimieren. Zwar ist Negativwerbung etwas, was in politischen Kampagnen häufig vorkommt, jedoch gibt es keine Werbung mit negativen Kosten. Folglich fordern wir, dass

$$x_1 \geq 0, \quad x_2 \geq 0, \quad x_3 \geq 0 \quad \text{und} \quad x_4 \geq 0 \tag{29.5}$$

gilt. Kombinieren wir die Ungleichungen (29.1)-(29.3) und (29.5) mit der zu minimierenden Zielfunktion

(29.4), so erhalten wir ein so genanntes „lineares Programm“. Wir schreiben dieses Problem in der Form

$$
\begin{aligned}
&\text{minimiere} \quad x_1 + x_2 + x_3 + x_4 \\
&\text{unter den Nebenbedingungen} \\
&\qquad -2x_1 + 8x_2 + 0x_3 + 10x_4 \geq 50 \\
&\qquad 5x_1 + 2x_2 + 0x_3 + 0x_4 \geq 100 \\
&\qquad 3x_1 - 5x_2 + 10x_3 - 2x_4 \geq 25 \\
&\qquad x_1, x_2, x_3, x_4 \geq 0.
\end{aligned}
$$

Die Lösung dieses linearen Programms liefert Ihnen eine optimale Strategie.

Man sieht: Auch dieses Beispiel ist ein „Diätproblem“, obwohl von gesunder oder preiswerter Ernährung nicht die Rede ist.

Wir schauen uns einige weitere Auszüge aus dem Buch von Cormen et al an:

6.1.2 Weitere Beispiele

Die Lineare Programmierung hat eine Vielzahl von Anwendungen. Jedes Lehrbuch über Operations Research enthält eine Fülle von Beispielen zur Linearen Programmierung. Lineare Programmierung ist ein Standardverfahren geworden, das den Studierenden in den meisten Wirtschaftsstudiengängen vermittelt wird. Das einführende Beispiel zu einer Wahlkampfstrategie ist ein typisches Beispiel. Zwei weitere Beispiele Linearer Programmierung sind die folgenden:

- Eine Fluggesellschaft möchte ihre Crews zusammenstellen. Von der zuständigen Luftfahrtbehörde werden viele Nebenbedingungen auferlegt, wie zum Beispiel die Beschränkung der Stundenzahl, die jedes Mitglied ohne Unterbrechung arbeiten darf, oder die Forderung, dass eine bestimmte Crew innerhalb eines Monats nur auf einem bestimmten Flugzeugtyp arbeiten darf. Die Fluggesellschaft will die Crews für alle Flüge so planen, dass so wenige Besatzungsmitglieder wie möglich gebraucht werden.
- Eine Ölgesellschaft muss entscheiden, wo nach Öl gebohrt werden soll. Einer Bohrung an einem bestimmten Ort sind bestimmte Kosten und, auf der Basis geologischer Gutachten, ein erwarteter Gewinn in Form einer bestimmten Anzahl Barrel Öl zugeordnet. Die Gesellschaft hat ein begrenztes Budget für neue Bohrungen und möchte die zu erwartende Ölmenge unter Vorgabe dieses Budgets maximieren.

Hier noch ein weiterer Absatz aus dem Buch von Cormen et al, *in dem besonders treffend die Rolle der Linearen Programmierung beschrieben wird.*

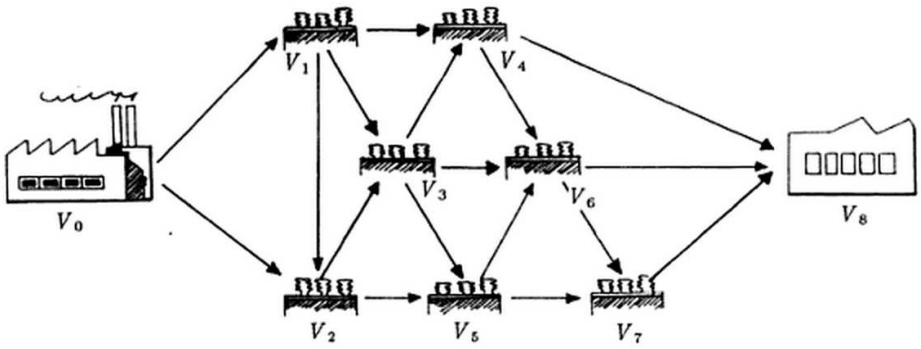
Die eigentliche Stärke der Linearen Programmierung liegt in ihrer Fähigkeit, neue Probleme zu lösen. Erinnern Sie sich an das Problem des Politikers zu Beginn dieses Kapitels. Das Problem, eine ausreichend große Anzahl von Stimmen zu bekommen und dabei nicht zu viel Geld auszugeben, wird durch keinen der Algorithmen, die wir in diesem Buch untersucht haben, gelöst; wir können es aber mit Linearer Programmierung lösen. Bücher enthalten eine Fülle solcher realitätsnaher Probleme, die durch Lineare Programmierung gelöst werden können. Die Lineare Programmierung ist auch dann besonders nützlich, wenn Varianten von Problemen zu lösen sind, für die wir noch nicht wissen, ob sie einen effizienten Algorithmus besitzen.

6.2 Energieflussproblem

Das nachfolgende Beispiel übernehmen wir aus

- K. Neumann, M. Morlock: *Operations Research*. Hanser-Verlag (2004, 2. Auflage).

Bei einem Problem der Versorgung mit elektrischer Energie soll von einer Quelle V_0 (Kraftwerk) über ein Leitungsnetz mit den Verteilerknoten (Umspannwerken) $V_1, \ldots, V_7$ die Senke V_8 (Fabrik) bedient werden. Weder in den Verteilerknoten noch in den Leitungen gehe Elektrizität verloren, d.h., der gesamte in V_0 eingespeiste Strom wird in V_8 abgenommen. Berücksichtigt man für die Leitungen Maximalbelastungen, so besteht eine Optimierungsaufgabe darin, die größtmögliche Strommenge zu bestimmen, die V_8 erreichen kann.



Ein anderes Optimierungsproblem ergibt sich, wenn elektrischer Strom einer vorgegebenen Stärke kostenoptimal durch ein Netz (von der Quelle zur Senke) geschickt werden soll und die Kosten aus Spannungsverlusten resultieren, die jeweils proportional zu den Stromstärken in den Leitungen sind.

Bezeichnen wir mit x_{ij} den von V_i nach V_j fließenden Strom[2], so erhalten wir neben einer Zielfunktion in der bekannten linearen Form für jeden Verteilerknoten die Bedingung, dass die hineinfließende gleich der herausfließenden Menge ist. Beispielsweise gilt für V_3

$$x_{13} + x_{23} - x_{34} - x_{35} - x_{36} = 0.$$

Eine entsprechende Beziehung gilt für die Quelle und die Senke. Die aus der Quelle herausfließende Menge w ist gerade gleich der in die Senke hineinfließenden Menge, da in den Verteilerknoten nichts „verlorengeht“. Für das obige Beispiel erhalten wir damit

$$x_{01} + x_{02} = w = x_{48} + x_{68} + x_{78}.$$

Neben diesen sogenannten Knotenbedingungen sind noch die Maximalbelastungen der Leitungen als (obere) Kapazitätsschranken zu beachten, d.h., für die Stärke x_{ij} des von V_i nach V_j fließenden Stroms gilt

$$0 \leq x_{ij} \leq \kappa_{ij},$$

wobei als untere Kapazitätsschranke 0 verwendet wird.

Den oben skizzierten Zielfunktionen und Nebenbedingungen ist zu entnehmen, dass wir es sowohl bei der Bestimmung eines *Flusses maximaler Stärke* (Maximierung von w) als auch bei der Berechnung eines *kostenoptimalen Flusses* vorgegebener Stärke mit sehr speziell strukturierten linearen Problemen zu tun haben, für die auch spezielle Verfahren entwickelt worden sind[3].

Eingesetzt werden derartige Modelle und Verfahren seit Anfang der 60er Jahre in einem breiten Spektrum von Anwendungen. Bereits zu dieser Zeit wurde z.B. vom Automobilhersteller Chrysler in den USA ein Planungsinstrument für die kostengünstigste Belieferung der Händler von den Produktionsstätten aus entwickelt (vgl. SHAPIRO (1984), Kapitel 5). Von 10 Werken aus sollen etwa 5000 Händler in den USA und in Kanada über ein gegebenes Vertriebsnetz beliefert werden. In Erweiterung der in Abb. 0.2.3 dargestellten Situation haben wir es mit einer Vielzahl von Quellen und Senken zu tun, was sich

[2] Genauer: Unter der Voraussetzung, dass die Kante von v_i nach v_j vorhanden ist, bezeichnen wir mir x_{ij} den über diese Kante von v_i nach v_j fließenden Strom.

[3] Flüsse maximaler Stärke werden im Rahmen der Vorlesung noch genauer behandelt werden.

jedoch ohne Schwierigkeiten auf den Fall einer Senke und einer Quelle zurückführen lässt. Ferner kann im Rahmen dieses Modells auch berücksichtigt werden, dass die Produktionskosten pro Fahrzeug in den einzelnen Werken unterschiedlich sein können und Beschränkungen hinsichtlich des Produktionsausstoßes bestehen.

6.3 Das Rucksackproblem (Knapsack Problem)

Die folgende Darstellung findet sich in

- S. Dasgupta, C. Papadimitriou, U. Vazirani: *Algorithms.* McGraw Hill (2008).

During a robbery, a burglar finds much more loot than he had expected and has to decide what to take. His bag (or "knapsack") will hold a total weight of at most W pounds. There are n items to pick from, of weight $w_1, \ldots, w_n$ and dollar value $v_1, \ldots, v_n$. What's the most valuable combination of items he can fit into his bag?[4]

For instance, take $W = 10$ and

Item	Weight	Value
1	6	\$30
2	3	\$14
3	4	\$16
4	2	\$9

There are two versions of this problem. If there are unlimited quantities of each item available, the optimal choice is to pick item 1 and two of item 4 (total: \$48). On the other hand, if there is one of each item (the burglar has broken into an art gallery, say), then the optimal knapsack contains items 1 and 3 (total: \$46).

Wir formulieren das beschriebene Problem wie folgt:

Version 1 („jeder Gegenstand nur einmal vorhanden"):

$$\begin{aligned} &\text{maximiere } 30x_1 + 14x_2 + 16x_3 + 9x_4 \\ &\text{unter den Nebenbedingungen} \\ &\qquad 6x_1 + 3x_2 + 4x_3 + 2x_4 \leq 10 \\ &\qquad x_1, x_2, x_3, x_4 \in \{0, 1\} \end{aligned}$$

Version 2 („beliebig viele Exemplare von jedem Gegenstand vorhanden"):

$$\begin{aligned} &\text{maximiere } 30x_1 + 14x_2 + 16x_3 + 9x_4 \\ &\text{unter den Nebenbedingungen} \\ &\qquad 6x_1 + 3x_2 + 4x_3 + 2x_4 \leq 10 \\ &\qquad x_1, x_2, x_3, x_4 \geq 0 \\ &\qquad x_1, x_2, x_3, x_4 \in \mathbb{Z} \end{aligned}$$

Version 1 wird für uns die wichtigere sein. *Offensichtlich handelt es sich weder bei Version 1 noch bei Version 2 um ein LP-Problem*: Bedingungen der Art $x_1, x_2, x_3, x_4 \in \{0, 1\}$ bzw. $x_1, x_2, x_3, x_4 \in \mathbb{Z}$ sind in LP-Problemen ja gar nicht vorgesehen.

[4] If this application seems frivolous, replace "weight" with "CPU time" and "only W pounds can be taken" with "only W units of CPU time are available." Or use "bandwidth" in place of "CPU time," etc. The knapsack problem generalizes a wide variety of resource-constrained selection tasks.

Bei Version 2 handelt es sich um ein *Ganzzahliges Lineares Programmierungsproblem* (engl. *integer linear programming problem*, kurz: *ILP-Problem*). Allgemein gilt: Ein Optimierungsproblem wird *Ganzzahliges Lineares Programmierungsproblem* oder *ILP-Problem* genannt, falls Folgendes erfüllt ist:

- Für jede Variable x_i wird $x_i \in \mathbb{Z}$ gefordert.
- Lässt man sämtliche Bedingungen $x_i \in \mathbb{Z}$ weg, so liegt ein LP-Problem vor.

Bei einem Ganzzahligen Linearen Programmierungsproblem dürfen die Werte der Variablen also nur ganze Zahlen sein. Bei Version 1 sind die Möglichkeiten sogar noch weiter eingeschränkt: Die Variablen dürfen nur einen der Werte 0 oder 1 annehmen. Man spricht von einem *0,1-Problem* (oder von einem *binären Problem*).

Wenn man in der Version 2 die Bedingungen $x_1, x_2, x_3, x_4 \in \mathbb{Z}$ weglässt, so gelangt man in der Tat zu einem LP-Problem. (Nebenbei: Wie lautet die optimale Lösung dieses LP-Problems?) Das entstehende LP-Problem nennt man übrigens die *LP-Relaxation* des ursprünglichen Problems[5].

Allgemein lautet das Rucksackproblem wie folgt:

Rucksackproblem (*Version 1*):

$$\begin{aligned} &\text{maximiere } c_1x_1 + \ldots + c_nx_n \\ &\text{unter den Nebenbedingungen} \\ &\qquad a_1x_1 + \ldots + a_nx_n \leq b \\ &\qquad\qquad x_i \in \{0,1\} \qquad (i = 1, \ldots, n) \end{aligned}$$

Rucksackproblem (*Version 2*):

$$\begin{aligned} &\text{maximiere } c_1x_1 + \ldots + c_nx_n \\ &\text{unter den Nebenbedingungen} \\ &\qquad a_1x_1 + \ldots + a_nx_n \leq b \\ &\qquad\qquad x_i \geq 0 \qquad (i = 1, \ldots, n) \\ &\qquad\qquad x_i \in \mathbb{Z} \qquad (i = 1, \ldots, n) \end{aligned}$$

Bei zahlreichen ganzzahligen oder binären Problemen handelt es sich um *NP-schwere Probleme* – das Rucksackproblem ist da keine Ausnahme (weder in der einen noch in der anderen Version).

Übrigens: Die Bedingung $x_i \in \{0,1\}$ kann ersetzt werden durch $0 \leq x_i \leq 1$, $x_i \in \mathbb{Z}$. Man kann also jedes $0,1$-Problem in ein Ganzzahliges Lineares Programmierungsproblem umschreiben. Anders gesagt: Bei den $0,1$-Problemen handelt es sich um spezielle ILP-Probleme.

Wir halten noch einmal ausdrücklich fest: *Bei beiden Versionen des Rucksackproblems handelt es sich* ***nicht*** *um ein LP-Problem, sondern um ein ILP-Problem.* Für ILP-Probleme kommen übrigens gänzlich andere Methoden zum Einsatz als für LP-Probleme; einen ersten Einblick in diese Methoden findet man beispielsweise im Lehrbuch von D. Bertsimas und J. N. Tsitsiklis.

6.4 Cutting Paper Rolls

Die Liste der Anwendungsmöglichkeiten Linearer Programmierung ist lang und wir können hier natürlich längst nicht alles besprechen. Wir präsentieren deshalb nur noch ein Beispiel, in dem es um ein *Verschnittproblem* geht. Derartige Probleme treten sehr häufigen in industriellen Fertigungsprozessen auf: Es geht darum, Rohmaterial optimal zu nutzen.

[5] Allgemein gilt: Lässt man in einem ILP-Problem sämtliche Bedingungen $x_i \in \mathbb{Z}$ weg, so nennt man das entstehende LP-Problem die *LP-Relaxation* des ursprünglichen Problems.

Der folgende Text stammt aus dem Buch

- Matoušek/Gärtner: *Understanding and Using Linear Programming.*

Here we have another industrial problem, and the application of linear programming is quite nonobvious. Moreover, we will naturally encounter an integrality constraint, which will bring us to the topic of the next chapter.

A paper mill manufactures rolls of paper of a standard width 3 meters. But customers want to buy paper rolls of shorter width, and the mill has to cut such rolls from the 3 m rolls. One 3 m roll can be cut, for instance, into two rolls 93 cm wide, one roll of width 108 cm, and a rest of 6 cm (which goes to waste).

Let us consider an order of

- 97 rolls of width 135 cm,
- 610 rolls of width 108 cm,
- 395 rolls of width 93 cm, and
- 211 rolls of width 42 cm.

What is the smallest number of 3 m rolls that have to be cut in order to satisfy this order, and how should they be cut?

In order to engage linear programming one has to be generous in introducing variables. We write down all of the requested widths: 135 cm, 108 cm, 93 cm, and 42 cm. Then we list all possibilities of cutting a 3 m paper roll into rolls of some of these widths (we need to consider only possibilities for which the wasted piece is shorter than 42 cm):

$$\begin{array}{ll} P_1: & 2 \cdot 135 \\ P_2: & 135 + 108 + 42 \\ P_3: & 135 + 93 + 42 \\ P_4: & 135 + 3 \cdot 42 \\ P_5: & 2 \cdot 108 + 2 \cdot 42 \\ P_6: & 108 + 2 \cdot 93 \end{array} \qquad \begin{array}{ll} P_7: & 108 + 93 + 2 \cdot 42 \\ P_8: & 108 + 4 \cdot 42 \\ P_9: & 3 \cdot 93 \\ P_{10}: & 2 \cdot 93 + 2 \cdot 42 \\ P_{11}: & 93 + 4 \cdot 42 \\ P_{12}: & 7 \cdot 42 \end{array}$$

For each possibility P_j on the list we introduce a variable $x_j \geq 0$ representing the number of rolls cut according to that possibility. We want to minimize the total number of rolls cut, i.e., $\sum_{j=1}^{12} x_j$, in such a way that the customers are satisfied. For example, to satisfy the demand for 395 rolls of width 93 cm we require

$$x_3 + 2x_6 + x_7 + 3x_9 + 2x_{10} + x_{11} \geq 395.$$

For each of the widths we obtain one constraint.

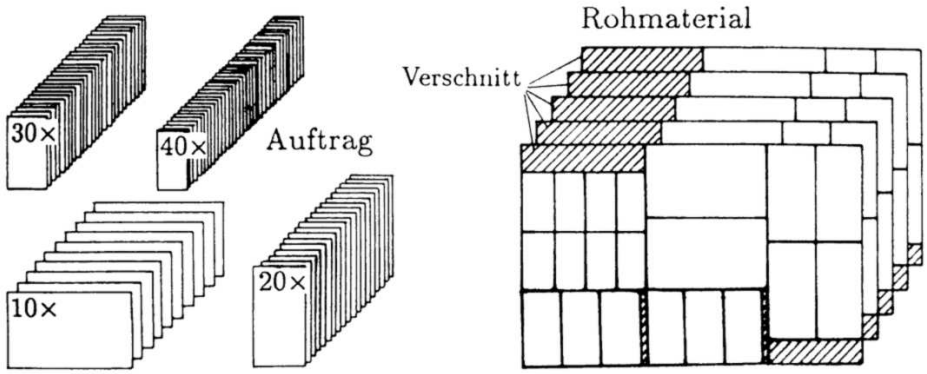
For a more complicated order, the list of possibilities would most likely be produced by computer. We would be in a quite typical situation in which a linear program is not entered "by hand," but rather is generated by some computer program. More-advanced techniques even generate the possibilities "on the fly," during the solution of the linear program, which may save time and memory considerably. See the entry "column generation" in the glossary or Chvátal's book cited in Chapter 9, from which this example is taken.

The optimal solution of the resulting linear program has $x_1 = 48.5$, $x_5 = 206.25$, $x_6 = 197.5$, and all other components 0. In order to cut 48.5 rolls according to the possibility P_1, one has to unwind half of a roll. Here we need more information about the technical possibilities of the paper mill: Is cutting a fraction of a roll technically and economically feasible? If yes, we have solved the problem optimally. If not, we have to work further and somehow take into account the restriction that only feasible solutions of the linear program with integral x_i are of interest. This is not at all easy in general, and it is the subject of Chapter 3.

In Chvátals Buch wird dieses Beispiel noch wesentlich genauer behandelt (Abschnitt 13: The Cutting-

Stock Problem). Auch im bereits öfter zitierten Buch von Neumann und Morlock findet sich einiges über verwandte Probleme, beispielsweise über so genannte *2-dimensionale Verschnittprobleme*.

Was unter einem 2-dimensionalen Verschnittproblem zu verstehen ist, wollen wir hier nur andeuten, indem wir die folgende Zeichnung aus dem Buch von Neumann/Morlock wiedergeben:



7 Dualität

7.1 Motivation: obere Schranken für den optimalen Wert

Wir betrachten das folgende LP-Problem in Standardform:

$$\begin{aligned}
&\text{maximiere } 4x_1 + x_2 + 5x_3 + 3x_4 \\
&\text{unter den Nebenbedingungen} \\
&\qquad\begin{aligned} x_1 - x_2 - x_3 + 3x_4 &\leq 1 \\ 5x_1 + x_2 + 3x_3 + 8x_4 &\leq 55 \\ -x_1 + 2x_2 + 3x_3 - 5x_4 &\leq 3 \\ x_1, x_2, x_3, x_4 &\geq 0. \end{aligned}
\end{aligned} \tag{7.1}$$

Anstatt das Problem zu lösen, wollen wir versuchen, möglichst gute *obere Schranken* für den optimalen Zielfunktionswert z^* unmittelbar am gegebenen LP-Problem abzulesen.

Beispielsweise könnte man die zweite Nebenbedingung mit 2 multiplizieren:

$$10x_1 + 2x_2 + 6x_3 + 16x_4 \leq 110.$$

Es folgt, dass für jede zulässige Lösung gilt:

$$4x_1 + x_2 + 5x_3 + 3x_4 \leq 10x_1 + 2x_2 + 6x_3 + 16x_4 \leq 110.$$

Da dies für jede zulässige Lösung gilt, gilt es insbesondere auch für jede optimale Lösung. Wir haben somit $z^* \leq 110$ erhalten.

Stellen wir uns etwas geschickter an, so können wir diese obere Schranke für z^* noch verbessern. Beispielsweise erhält man $z^* \leq \frac{275}{3}$, wenn man die zweite Nebenbedingung nicht mit 2, sondern mit $\frac{5}{3}$ multipliziert; in diesem Fall ergibt sich für jede zulässige Lösung:

$$4x_1 + x_2 + 5x_3 + 3x_4 \leq \frac{25}{3}x_1 + \frac{5}{3}x_2 + 5x_3 + \frac{40}{3}x_4 \leq \frac{275}{3}.$$

Also (insbesondere) $z^* \leq \frac{275}{3}$.

Mit etwas Eingebung und Fantasie können wir diese Schranke noch weiter verbessern. Addiert man beispielsweise die zweite und die dritte Nebenbedingung, so erhält man

$$4x_1 + 3x_2 + 6x_3 + 3x_4 \leq 58;$$

es folgt $z^* \leq 58$.

Wir wollen nun systematisch vorgehen und eine Strategie zum Auffinden von oberen Schranken für z^* beschreiben: *Wir bilden eine Linearkombination der Nebenbedingungen, d.h., wir nehmen die erste Nebenbedingung mit einer Zahl y_1 mal, die zweite Nebenbedingung mit y_2, die dritte mit y_3; danach addieren wir die erhaltenen Ungleichungen.*

Dabei setzen wir voraus, dass $y_1 \geq 0$, $y_2 \geq 0$ und $y_3 \geq 0$ gilt. (In unseren obigen Betrachtungen, die zu $z^* \leq \frac{275}{3}$ bzw. zu $z^* \leq 58$ geführt haben, galt $y_1 = 0$, $y_2 = \frac{5}{3}$, $y_3 = 0$ bzw. $y_1 = 0$, $y_2 = y_3 = 1$.)

Im allgemeinen Fall (d.h. mit beliebigen $y_1, y_2, y_3 \geq 0$) erhält man

$$\Big(y_1+5y_2-y_3\Big)x_1+\Big(-y_1+y_2+2y_3\Big)x_2+\Big(-y_1+3y_2+3y_3\Big)x_3+\Big(3y_1+8y_2-5y_3\Big)x_4 \leq y_1+55y_2+3y_3. \quad (7.2)$$

Nun möchte man erreichen, dass die linke Seite von (7.2) eine obere Schranke für die Zielfunktion

$$z = 4x_1 + x_2 + 5x_3 + 3x_4$$

ergibt. Dies ist gewiss der Fall, wenn Folgendes gilt:

$$\begin{aligned} y_1 + 5y_2 - \ y_3 &\geq 4 \\ -y_1 + \ y_2 + 2y_3 &\geq 1 \\ -y_1 + 3y_2 + 3y_3 &\geq 5 \\ 3y_1 + 8y_2 - 5y_3 &\geq 3. \end{aligned}$$

Wenn also die Faktoren y_i nichtnegativ sind und diese 4 Ungleichungen erfüllt sind, so können wir sicher sein, dass jede zulässige Lösung (x_1, x_2, x_3, x_4) die Ungleichung

$$4x_1 + x_2 + 5x_3 + 3x_4 \leq y_1 + 55y_2 + 3y_3$$

erfüllt. Da diese Ungleichung für alle zulässigen Lösungen gilt, also insbesondere auch für eine optimale Lösung, erhalten wir

$$z^* \leq y_1 + 55y_2 + 3y_3.$$

Wir möchten gerne, dass diese obere Schranke für z^* möglichst nahe bei z^* liegt. *Damit sind wir beim folgenden Minimierungsproblem angelangt*:

$$\begin{aligned} &\text{minimiere} \quad y_1 + 55y_2 + 3y_3 \\ &\text{unter den Nebenbedingungen} \\ &\qquad\qquad y_1 + \ 5y_2 - \ y_3 \geq 4 \\ &\qquad\qquad -y_1 + \ y_2 + 2y_3 \geq 1 \\ &\qquad\qquad -y_1 + \ 3y_2 + 3y_3 \geq 5 \\ &\qquad\qquad 3y_1 + \ 8y_2 - 5y_3 \geq 3 \\ &\qquad\qquad\qquad y_1, y_2, y_3 \geq 0. \end{aligned}$$

Das so erhaltene LP-Problem nennt man das *duale Problem* des ursprünglichen Problems[1].

7.2 Das duale Problem

Wir betrachten nun den allgemeinen Fall. Gegeben sei ein LP-Problem in Standardform:

$$\begin{aligned} &\text{maximiere} \quad \sum_{j=1}^{n} c_j x_j \\ &\text{unter den Nebenbedingungen} \\ &\qquad \sum_{j=1}^{n} a_{ij} x_j \ \leq b_i \qquad (i = 1, \ldots, m) \\ &\qquad\qquad x_j \ \geq \ 0 \qquad (j = 1, \ldots, n). \end{aligned} \quad (P)$$

[1] Statt „ursprüngliches Problem“ sagt man auch *primales Problem*.

Dann nennt man das folgende Problem das *duale Problem* zu (P):

$$\begin{aligned} &\text{minimiere } \sum_{i=1}^{m} b_i y_i \\ &\text{unter den Nebenbedingungen} \\ &\qquad \sum_{i=1}^{m} a_{ij} y_i \ \geq c_j \qquad (j = 1, \ldots, n) \\ &\qquad\qquad y_i \ \geq \ 0 \qquad (i = 1, \ldots, m). \end{aligned} \tag{D}$$

Man beachte: Das Duale eines Maximierungsproblems ist ein Minimierungsproblem; jeder der m *primalen Nebenbedingungen*

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_i$$

entspricht eine *duale Variable* y_i $(i = 1, \ldots, m)$; umgekehrt gilt: Jeder der n *dualen Nebenbedingungen*

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

entspricht eine *primale Variable* x_j $(j = 1 \ldots, n)$; die Koeffizienten c_j der primalen Zielfunktion tauchen im dualen Problem als rechte Seite auf; Entsprechendes gilt umgekehrt auch für die Koeffizienten b_i.

Besonders kurz kann man das alles aufschreiben, wenn man *Matrixschreibweise* verwendet; dann lauten das primale Problem (P) und das duale Problem (D) wie folgt:

Primales Problem.

$$\begin{aligned} &\text{maximiere } c^T x \\ &\text{unter den Nebenbedingungen} \\ &\qquad Ax \ \leq b \\ &\qquad\ \ x \ \geq 0 \end{aligned} \tag{P}$$

Duales Problem.

$$\begin{aligned} &\text{minimiere } \ b^T y \\ &\text{unter den Nebenbedingungen} \\ &\qquad A^T y \ \geq c \\ &\qquad\ \ y \ \geq 0 \end{aligned} \tag{D}$$

Hierin ist

$$c^T = (c_1, \ldots, c_n), \quad b^T = (b_1, \ldots, b_m), \quad x = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}, \quad y = \begin{pmatrix} y_1 \\ \vdots \\ y_m \end{pmatrix}, \quad A = \begin{pmatrix} a_{11} & \ldots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \ldots & a_{mn} \end{pmatrix};$$

0 bezeichnet in (P) den Nullvektor der Länge n und in (D) den Nullvektor der Länge m; die Ungleichungen sind komponentenweise zu lesen, beispielsweise bedeutet $y \geq 0$ dasselbe wie $y_i \geq 0$ für $i = 1, \ldots, m$.

Wie bereits in unserem Beispiel beobachtet, liefert jede zulässige Lösung des dualen Problems eine *obere Schranke* für den optimalen Wert des primalen Problems.

Feststellung.
Ist $(x_1, \ldots, x_n)$ eine primale zulässige Lösung und $(y_1, \ldots, y_m)$ eine duale zulässige Lösung[2], so gilt

$$\sum_{j=1}^{n} c_j x_j \leq \sum_{i=1}^{m} b_i y_i. \tag{7.3}$$

Die wichtige Feststellung (7.3) wird *schwache Dualität* genannt. Der Beweis von (7.3) ist sehr kurz.

Beweis.

$$\sum_{j=1}^{n} c_j x_j \leq \sum_{j=1}^{n} \left(\sum_{i=1}^{m} a_{ij} y_i \right) x_j = \sum_{i=1}^{m} \left(\sum_{j=1}^{n} a_{ij} x_j \right) y_i \leq \sum_{i=1}^{m} b_i y_i. \quad \square$$

Dasselbe in Matrixschreibweise:

$$c^T x \leq \Big(A^T y\Big)^T x = \Big(y^T A\Big) x = y^T \Big(Ax\Big) \leq y^T b = b^T y.$$

Die Beziehung (7.3) („schwache Dualität") ist sehr nützlich: Falls wir irgendwo her eine primale zulässige Lösung $(x_1^*, \ldots, x_n^*)$ und eine duale zulässige Lösung $(y_1^*, \ldots, y_m^*)$ haben und falls

$$\sum_{j=1}^{n} c_j x_j^* = \sum_{i=1}^{m} b_i y_i^* \tag{7.4}$$

gilt, so können wir sicher sein, dass $(x_1^*, \ldots, x_n^*)$ eine optimale Lösung des primalen Problems und $(y_1^*, \ldots, y_m^*)$ eine optimale Lösung des dualen Problems ist. (Wieso nämlich?)

Beispiel. Wir betrachten das LP-Problem (7.1), das uns bereits am Anfang dieses Abschnitts zur Illustration diente: $x_1 = 0$, $x_2 = 14$, $x_3 = 0$, $x_4 = 5$ ist eine zulässige Lösung dieses Problems – davon können wir uns leicht überzeugen. Wir brauchen die Zahlen $x_1 = 0$, $x_2 = 14$, $x_3 = 0$, $x_4 = 5$ ja nur in (7.1) einzusetzen.

Ich behaupte nun: *Die Zahlen* $x_1 = 0$, $x_2 = 14$, $x_3 = 0$ *und* $x_4 = 5$ *bilden sogar eine optimale Lösung von (7.1)*. Stellen wir uns vor, das Sie dies bezweifeln. *Wie kann ich Sie schnell davon überzeugen, dass ich Recht habe?*

Hier ist die Antwort: Ich präsentiere Ihnen zusätzlich die Zahlen

$$y_1 = 11, \quad y_2 = 0 \quad \text{und} \quad y_3 = 6,$$

die ich „magische Zahlen" nenne, da ich mit ihrer Hilfe sämtliche Zweifel zum Verschwinden bringe.

Bei diesen Zahlen handelt es sich um eine zulässige Lösung des dualen Problems – davon können wir uns ebenfalls ohne Mühe überzeugen. (Wie nämlich?)

Nun brauchen wir nur noch die Zielfunktionswerte zu vergleichen: Für $x_1 = 0$, $x_2 = 14$, $x_3 = 0$, $x_4 = 5$ erhält man $z = 29$. Und für $y_1 = 11$, $y_2 = 0$, $y_3 = 6$ erhalten wir

$$y_1 + 55y_2 + 3y_3 = 11 + 0 + 18 = 29.$$

Also ist $x_1 = 0$, $x_2 = 14$, $x_3 = 0$, $x_4 = 5$ wie behauptet eine optimale Lösung des primalen Problems (7.1) (und $y_1 = 11$, $y_2 = 0$, $y_3 = 6$ ist eine optimale Lösung des dazugehörigen dualen Problems).

Anknüpfend an die schwache Dualität und an das Beispiel mit den „magischen Zahlen" lernen wir jetzt einen zentralen Satz kennen.

[2] „Primale zulässige Lösung" bedeutet natürlich „zulässige Lösung des primalen Problems"; analog: „duale zulässige Lösung". Die Bezeichnungen c_j und b_i seien wie in (P) und (D).

7.3 Der Dualitätssatz und sein Beweis

Der Dualitätssatz hat seinen Ursprung in Diskussionen zwischen G. B. Dantzig und J. von Neumann aus dem Jahr 1947. Die erste explizite Version des Satzes stammt von D. Gale, H. W. Kuhn und A. W. Tucker (1951).

Bevor wir den Dualitätssatz vorstellen, überlegen wir uns zunächst, dass *das duale Problem des dualen Problems wieder das primale Problem ist*. Hierzu schreiben wir das duale Problem (D) in ein Maximierungsproblem in Standardform um:

$$
\begin{aligned}
&\text{maximiere} \quad \sum_{i=1}^{m} (-b_i) y_i \\
&\text{unter den Nebenbedingungen} \\
&\qquad \sum_{i=1}^{m} (-a_{ij}) y_i \;\leq -c_j \qquad (j = 1, \ldots, n) \\
&\qquad\qquad y_i \;\geq \quad 0 \qquad (i = 1, \ldots, m).
\end{aligned}
\tag{$\widetilde{D}$}
$$

Das Duale dieses Problems ist

$$
\begin{aligned}
&\text{minimiere} \quad \sum_{j=1}^{n} (-c_j) x_j \\
&\text{unter den Nebenbedingungen} \\
&\qquad \sum_{j=1}^{n} (-a_{ij}) x_j \;\geq -b_i \qquad (i = 1, \ldots, m) \\
&\qquad\qquad x_j \;\geq \quad 0 \qquad (j = 1, \ldots, n),
\end{aligned}
$$

was offensichtlich äquivalent zum Ausgangsproblem (P) (dem primalen Problem) ist.

Gibt es immer „magische Zahlen“ wie im obigen Beispiel?

Die Antwort auf diese Frage liefert der *Dualitätssatz*.

Satz 1 (Dualitätssatz).

Falls das primale Problem (P) eine optimale Lösung $(x_1^*, \ldots, x_n^*)$ besitzt, so besitzt auch das duale Problem (D) eine optimale Lösung $(y_1^*, \ldots, y_m^*)$ und es gilt

$$
\sum_{j=1}^{n} c_j x_j^* = \sum_{i=1}^{m} b_i y_i^*. \tag{7.5}
$$

Bevor wir den Satz beweisen, wollen wir den entscheidenden Punkt anhand eines Beispiels studieren. Dazu nehmen wir an, dass das primale Problem (P) eine optimale Lösung besitzt. Der Punkt, auf den es ankommt, ist der folgende:

Löst man das primale Problem mit dem Simplexverfahren, so kann man in der letzten Zeile („der z-Zeile“) des letzten Tableaus eine optimale Lösung des dualen Problems ablesen.

Wir schauen uns an unserem Beispiel an, wie das geht. Löst man das Problem (7.1) mit dem Simplex-

verfahren, so erhält man als letztes Tableau:

$$\begin{aligned} x_2 &= 14 - 2x_1 - 4x_3 - 5x_5 - 3x_7 \\ x_4 &= 5 - x_1 - x_3 - 2x_5 - x_7 \\ x_6 &= 1 + 5x_1 + 9x_3 + 21x_5 + 11x_7 \\ \hline z &= 29 - x_1 - 2x_3 - 11x_5 - 6x_7. \end{aligned} \qquad (\star)$$

Wir wissen: Zu den ersten drei Ungleichungen des LP-Problems (7.1) gehören die drei Schlupfvariablen

$$x_5, \quad x_6 \quad \text{und} \quad x_7.$$

Andererseits gehört zu jeder dieser Ungleichungen auch eine duale Variable:

$$y_1, \quad y_2 \quad \text{und} \quad y_3.$$

Durch die erste Ungleichung von (7.1) ist also x_5 mit y_1 verbunden; ebenso bestehen Verbindungen von x_6 zu y_2 und x_7 zu y_3:

$$\begin{array}{ccc} x_5 & x_6 & x_7 \\ \updownarrow & \updownarrow & \updownarrow \\ y_1 & y_2 & y_3 \end{array}$$

In der z-Zeile von $(\star)$ finden wir bei den Schlupfvariablen x_5, x_6 und x_7 die folgenden Koeffizienten vor: -11 bei x_5, 0 bei x_6 und -6 bei x_7.

Ordnet man diese Koeffizienten mit umgekehrtem Vorzeichen den entsprechenden dualen Variablen zu, so erhält man die gewünschte optimale Lösung des dualen Problems:

$$y_1 = 11, \quad y_2 = 0, \quad y_3 = 6.$$

Aus dem Beweis des Dualitätssatzes wird sich ergeben, dass diese Vorgehensweise immer funktioniert; dies ist der entscheidende Punkt im nachfolgenden Beweis.

Beweis des Dualitätssatzes. Es sei $(x_1^*, \ldots, x_n^*)$ eine optimale Lösung von (P). Wir haben eine zulässige Lösung $(y_1^*, \ldots, y_m^*)$ des dualen Problems (D) anzugeben, die die Gleichung (7.5) erfüllt. Aufgrund von (7.3) ist $(y_1^*, \ldots, y_m^*)$ dann eine optimale Lösung von (D), für die (7.5) gilt, womit wir fertig sind.

Wir gehen vor, wie zuvor in unserem Beispiel. Nachdem wir Schlupfvariablen

$$x_{n+i} = b_i - \sum_{j=1}^{n} a_{ij} x_j \qquad (i = 1, \ldots, m) \tag{7.6}$$

eingeführt haben, landen wir schließlich beim letzten Tableau des Simplexverfahrens; die letzte Zeile dieses Tableaus sei:

$$z = z^* + \sum_{k=1}^{n+m} \overline{c}_k x_k. \tag{7.7}$$

Ist x_k eine Basisvariable, so gilt $\overline{c}_k = 0$; für alle Koeffizienten $\overline{c}_k$, die zu Nichtbasisvariablen gehören, gilt $\overline{c}_k \leq 0$. Außerdem ist z^* der optimale Wert der Zielfunktion. Nach Voraussetzung ist $(x_1^*, \ldots, x_n^*)$ eine optimale Lösung von (P); deshalb gilt

$$z^* = \sum_{j=1}^{n} c_j x_j^*. \tag{7.8}$$

Wir definieren

$$y_i^* = -\overline{c}_{n+i} \qquad (i = 1, \ldots, m) \tag{7.9}$$

und behaupten, dass $(y_1^*, \ldots, y_m^*)$ eine zulässige Lösung des dualen Problems (D) ist, die (7.5) erfüllt.

Damit haben wir die entscheidende Idee des Beweises geschildert, nachdem wir diese Idee ja anhand unseres Beispiels kennengelernt hatten.

Der Rest des Beweises besteht darin, „nachzurechnen", dass $(y_1^*, \ldots, y_m^*)$ tatsächlich eine zulässige Lösung von (D) ist, die (7.5) erfüllt. Im Folgenden finden Sie den Rest des Beweises im englischen Original (vgl. Chvátal: *Linear Programming*); zur Erinnerung ein paar Vokabeln:

constraint	–	Nebenbedingung
objective function	–	Zielfunktion
feasible solution	–	zulässige Lösung
slack variable	–	Schlupfvariable

Defining

$$y_i^* = -\overline{c}_{n+i} \qquad (i = 1, \ldots, m) \tag{7.9}$$

we claim that $(y_1^*, \ldots, y_m^*)$ is a dual feasible solution satisfying (7.5); the rest of the proof consists of a straightforward verification of our claim. Substituting $\sum c_j x_j$ for z and substituting from (7.6) for the slack variables in (7.7) we obtain the identity

$$\sum_{j=1}^{n} c_j x_j \;=\; z^* + \sum_{j=1}^{n} \overline{c}_j x_j - \sum_{i=1}^{m} y_i^* \left(b_i - \sum_{j=1}^{n} a_{ij} x_j \right)$$

which may be written as

$$\sum_{j=1}^{n} c_j x_j \;=\; \left(z^* - \sum_{i=1}^{m} b_i y_i^* \right) + \sum_{j=1}^{n} \left(\overline{c}_j + \sum_{i=1}^{m} a_{ij} y_i^* \right) x_j .$$

This identity, having been obtained by algebraic manipulations from the definitions of the slack variables and the objective function, must hold for every choice of values $x_1, \ldots, x_n$. Hence we have

$$z^* \;=\; \sum_{i=1}^{m} b_i y_i^* \tag{7.10}$$

and

$$c_j \;=\; \overline{c}_j + \sum_{i=1}^{m} a_{ij} y_i^* \qquad (j = 1, \ldots, n). \tag{7.11}$$

Since $\overline{c}_k \le 0$ for every $k = 1, \ldots, n+m$, (7.9) and (7.11) imply

$$\begin{aligned} \sum_{i=1}^{m} a_{ij} y_i^* \;&\ge\; c_j \qquad (j = 1, \ldots, n) \\ y_i^* \;&\ge\; 0 \qquad (i = 1, \ldots, m). \end{aligned}$$

Finally, (7.8) and (7.10) imply (7.5). □

Wir formulieren den Dualitätssatz noch einmal *kurz zusammengefasst in Worten.*

Dualitätssatz (kurz zusammengefasst).
Wenn das primale Problem eine optimale Lösung besitzt, dann besitzt auch das duale Problem eine optimale Lösung *und die dazugehörigen Zielfunktionswerte stimmen überein.*

Anhand des Beispiels mit den „magischen Zahlen“ haben wir bereits gesehen, dass es sehr nützlich sein kann, neben einer optimalen Lösung $x_1^*, \ldots, x_n^*$ des primalen Problems auch eine optimale Lösung $y_1^*, \ldots, y_m^*$ des dazugehörigen dualen Problems zur Verfügung zu haben: Die Zahlen $y_1^*, \ldots, y_m^*$ können als ein *Zertifikat der Optimalität* (engl. *certificate of optimality*) angesehen werden, da man – wie wir gesehen haben – mithilfe dieser Zahlen eine andere Person schnell davon überzeugen kann, dass $x_1^*, \ldots, x_n^*$ tatsächlich eine optimale Lösung des primalen Problems ist.

Außerdem gilt[3]: *Falls man $x_1^*, \ldots, x_n^*$ mit dem Simplexverfahren ermittelt, so bekommt man die Zahlen $y_1^*, \ldots, y_m^*$ (also das Zertifikat) am Ende „kostenlos mitgeliefert“.* Man spricht in diesem Zusammenhang von einem *zertifizierenden Algorithmus* (engl. *certifying algorithm*).

Als Folgerung aus dem Dualitätssatz und aus der bereits vor dem Dualitätssatz festgestellten Tatsache, dass das Duale des dualen Problems wieder das primale Problem ist, erhalten wir den folgenden Satz.

Satz 2 (Folgerung aus dem Dualitätssatz).
Gegeben seien das LP-Problem (P) und das zu (P) duale Problem (D). Dann gilt:

(i) Besitzt eines dieser beiden Probleme eine optimale Lösung, so besitzt auch das andere eine optimale Lösung und die Zielfunktionswerte stimmen überein.

(ii) Ist eines der beiden Probleme unbeschränkt, so hat das andere keine zulässige Lösung.

Beweis.

(i) Dies ergibt sich aus dem Dualitätssatz sowie der Tatsache, dass das Duale des dualen Problems wieder das primale Problem ist.

(ii) Dies ergibt sich unmittelbar aus (7.3) („schwache Dualität“). □

Außer den beiden unter (i) und (ii) genannten Fällen gibt es auch noch den Fall, dass sowohl (P) als auch (D) keine zulässige Lösung besitzt. Dass dieser Fall tatsächlich vorkommen kann, zeigt das folgende **Beispiel**.

$$\begin{aligned} &\text{maximiere } 2x_1 - x_2 \\ &\text{unter den Nebenbedingungen} \\ &\qquad x_1 - x_2 \leq 1 \\ &\qquad -x_1 + x_2 \leq -2 \\ &\qquad x_1, x_2 \geq 0. \end{aligned}$$

Dass dieses Problem keine zulässige Lösung besitzt, erkennt man sofort: Addition der beiden Ungleichungen $x_1 - x_2 \leq 1$ und $-x_1 + x_2 \leq -2$ ergibt $0 \leq -1$.

Auch das duale Problem, das wie folgt lautet, ist nicht lösbar:

$$\begin{aligned} &\text{minimiere } y_1 - 2y_2 \\ &\text{unter den Nebenbedingungen} \\ &\qquad y_1 - y_2 \geq 2 \\ &\qquad -y_1 + y_2 \geq -1 \\ &\qquad y_1, y_2 \geq 0. \end{aligned}$$

[3] Dies ergibt sich, wie wir ausführlich besprochen haben, aus dem Beweis des Dualitätssatzes.

Feststellung.

Sind (P) und (D) wie oben gegeben, so sind also genau *drei Fälle* möglich:

(i) Sowohl (P) als auch (D) besitzt eine optimale Lösung; in diesem Fall stimmen die optimalen Zielfunktionswerte überein.

(ii) Eines der beiden Probleme (P) und (D) ist unbeschränkt und das andere besitzt keine zulässige Lösung.

(iii) Keines der beiden Probleme (P) und (D) besitzt eine zulässige Lösung.

In enger Weise mit dem Dualitätssatz verbunden ist der folgende Satz, den man den *Satz vom komplementären Schlupf* (engl. *Complementary Slackness Theorem*) nennt.

Satz 3 (Satz vom komplementären Schlupf).

Es sei $x_1^*, \ldots, x_n^*$ eine zulässige Lösung von (P) und $y_1^*, \ldots, y_m^*$ sei eine zulässige Lösung von (D). Notwendig und hinreichend dafür, dass es sich bei $x_1^*, \ldots, x_n^*$ und $y_1^*, \ldots, y_m^*$ gleichzeitig um optimale Lösungen von (P) bzw. (D) handelt, ist das Erfülltsein der folgenden $n+m$ Bedingungen:

$$x_j^* = 0 \quad \text{oder} \quad \sum_{i=1}^{m} a_{ij} y_i^* = c_j \qquad (j = 1, \ldots, n) \tag{7.12}$$

$$y_i^* = 0 \quad \text{oder} \quad \sum_{j=1}^{n} a_{ij} x_j^* = b_i \qquad (i = 1, \ldots, m). \tag{7.13}$$

Beweis. Da $x_1^*, \ldots, x_n^*$ und $y_1^*, \ldots, y_m^*$ zulässige Lösungen von (P) bzw. (D) sind, gilt

$$c_j x_j^* \leq \left(\sum_{i=1}^{m} a_{ij} y_i^* \right) x_j^* \qquad (j = 1, \ldots, n) \tag{7.14}$$

und

$$\left(\sum_{j=1}^{n} a_{ij} x_j^* \right) y_i^* \leq b_i y_i^* \qquad (i = 1, \ldots, m). \tag{7.15}$$

Es folgt

$$\sum_{j=1}^{n} c_j x_j^* \leq \sum_{j=1}^{n} \left(\sum_{i=1}^{m} a_{ij} y_i^* \right) x_j^* = \sum_{i=1}^{m} \left(\sum_{j=1}^{n} a_{ij} x_j^* \right) y_i^* \leq \sum_{i=1}^{m} b_i y_i^*. \tag{7.16}$$

Bei $x_1^*, \ldots, x_n^*$ und $y_1^*, \ldots, y_m^*$ handelt es sich (nach dem Dualitätssatz) genau dann um optimale Lösungen von (P) bzw. (D), wenn in (7.16) Gleichheit gilt. Dies ist genau dann der Fall, wenn in sämtlichen Ungleichungen in (7.14) und (7.15) Gleichheit gilt, was genau dann der Fall ist, wenn sämtliche Bedingungen (7.12) und (7.13) gelten. □

Es sei ausdrücklich darauf hingewiesen, dass das Wort „oder“ in (7.12) sowie in (7.13) wie üblich als „einschließendes Oder“ gemeint ist. Die erste der $m+n$ Bedingungen (7.12) und (7.13) besagt also, wenn man sie etwas anders formuliert: Es gilt $x_1^* = 0$ oder $a_{11} y_1^* + \ldots + a_{m1} y_m^* = c_1$ *oder beides*.

Der Inhalt der n Bedingungen (7.12) wird besonders deutlich, wenn man daran denkt, dass es in (P) genau n Variablen $x_1, \ldots, x_n$ gibt und dass diese n Variablen in natürlicher Weise den ersten n Ungleichungen

in (D) entsprechen:

$$\begin{aligned} x_1 &\longleftrightarrow a_{11}y_1 + \ldots + a_{m1}y_m \geq c_1 \\ &\vdots \\ x_n &\longleftrightarrow a_{1n}y_1 + \ldots + a_{mn}y_m \geq c_n. \end{aligned}$$

Wir können (7.12) also auch so aussprechen:

In $(x_1^*, \ldots, x_n^*)$ ist die j-te Variable gleich Null oder die entsprechende duale Ungleichung ist *tight*, d.h., diese Ungleichung ist mit Gleichheit erfüllt $(j = 1, \ldots, n)$.

Entsprechend kann man (7.13) wie folgt formulieren:

In $(y_1^*, \ldots, y_m^*)$ ist die i-te Variable gleich Null oder die entsprechende primale Ungleichung ist mit Gleichheit erfüllt $(i = 1, \ldots, m)$.

Beispiel. Wir greifen unser erstes Beispiel aus Kapitel 2 auf:

$$\begin{aligned} &\text{maximiere } 5x_1 + 4x_2 + 3x_3 \\ &\text{unter den Nebenbedingungen} \\ &\qquad \begin{aligned} 2x_1 + 3x_2 + x_3 &\leq 5 \\ 4x_1 + x_2 + 2x_3 &\leq 11 \\ 3x_1 + 4x_2 + 2x_3 &\leq 8 \\ x_1, x_2, x_3 &\geq 0. \end{aligned} \end{aligned} \tag{P}$$

Wir haben (P) bereits mit dem Simplexverfahren gelöst; das letzte Tableau lautete

$$\begin{aligned} x_3 &= 1 + x_2 + 3x_4 - 2x_6 \\ x_1 &= 2 - 2x_2 - 2x_4 + x_6 \\ x_5 &= 1 + 5x_2 + 2x_4 \\ \hline z &= 13 - 3x_2 - x_4 - x_6. \end{aligned}$$

Aufgabe.

a) Stellen Sie das zugehörige duale Problem (D) auf.

b) Lesen Sie aus dem letzten Tableau für (P) eine optimale Lösung (x_1^*, x_2^*, x_3^*) für (P) sowie eine optimale Lösung (y_1^*, y_2^*, y_3^*) für (D) ab.

c) Überprüfen Sie, ob (y_1^*, y_2^*, y_3^*) tatsächlich eine zulässige Lösung für (D) ist, und überprüfen Sie mithilfe des Dualitätssatzes, ob (y_1^*, y_2^*, y_3^*) tatsächlich optimal ist.

d) Bestätigen Sie noch einmal, dass es sich bei (x_1^*, x_2^*, x_3^*) und (y_1^*, y_2^*, y_3^*) um optimale Lösungen von (P) bzw. (D) handelt, indem Sie die Bedingungen (7.12) und (7.13) überprüfen.

Im Englischen heißen die Bedingungen (7.12) und (7.13) übrigens *complementary slackness conditions*; im Deutschen sagt man *komplementäre Schlupfbedingungen*.

7.4 Wie die komplementären Schlupfbedingungen eingesetzt werden können, um auf Optimalität zu testen

Stellen Sie sich vor, dass $(x_1^*, \ldots, x_n^*)$ eine zulässige Lösung eines LP-Problems (P) in Standardform ist. Sie vermuten, dass $(x_1^*, \ldots, x_n^*)$ optimal ist – sicher sind Sie aber nicht.

In dieser Situation können die komplementären Schlupfbedingungen sehr nützlich sein, um zu testen, ob $(x_1^*, \ldots, x_n^*)$ *tatsächlich optimal ist.*

Bevor wir uns anhand eines Beispiels anschauen, wie das geht, halten wir eine Folgerung aus dem Satz vom komplementären Schlupf fest, die besonders gut zu unserer Zielsetzung passt. (Man stellt leicht fest, dass Satz 3' aus Satz 3 folgt.)

Satz 3' (Folgerung aus dem Satz vom komplementären Schlupf).
Eine zulässige Lösung $(x_1^*, \ldots, x_n^*)$ von (P) ist genau dann optimal, wenn es Zahlen $y_1^*, \ldots, y_m^*$ gibt, für die gilt:

- Für $(x_1^*, \ldots, x_n^*)$ und $(y_1^*, \ldots, y_m^*)$ gelten die komplementären Schlupfbedingungen;
- $(y_1^*, \ldots, y_m^*)$ ist eine zulässige Lösung von (D).

Beispiel 1. Wir betrachten das folgende LP-Problem

$$\begin{aligned} &\text{maximiere } 3x_1 + x_2 + 2x_3 \\ &\text{unter den Nebenbedingungen} \\ &\qquad x_1 + x_2 + 3x_3 \le 30 \\ &\qquad 2x_1 + 2x_2 + 5x_3 \le 24 \\ &\qquad 4x_1 + x_2 + 2x_3 \le 36 \\ &\qquad x_1, x_2, x_3 \ge 0 \end{aligned} \tag{P}$$

und möchten prüfen, ob

$$x_1^* = 8, \quad x_2^* = 4, \quad x_3^* = 0$$

eine optimale Lösung von (P) ist.

Zu diesem Zweck betrachten wir das duale Problem (D):

$$\begin{aligned} &\text{minimiere } 30y_1 + 24y_2 + 36y_3 \\ &\text{unter den Nebenbedingungen} \\ &\qquad y_1 + 2y_2 + 4y_3 \ge 3 \\ &\qquad y_1 + 2y_2 + y_3 \ge 1 \\ &\qquad 3y_1 + 5y_2 + 2y_3 \ge 2 \\ &\qquad y_1, y_2, y_3 \ge 0. \end{aligned} \tag{D}$$

Wir wollen Satz 3' anwenden und müssen demnach herausfinden, ob es Zahlen y_1^*, y_2^* und y_3^* gibt, für die gilt:

(1) Für (x_1^*, x_2^*, x_3^*) und (y_1^*, y_2^*, y_3^*) gelten die komplementären Schlupfbedingungen.

(2) (y_1^*, y_2^*, y_3^*) ist eine zulässige Lösung von (D).

Wir betrachten zunächst nur (1): Setzt man $x_1^* = 8$, $x_2^* = 4$ und $x_3^* = 0$ in (P) ein, so stellt man fest, dass die 1. Ungleichung von (P) nicht mit Gleichheit erfüllt ist; soll (1) erfüllt sein, so muss also

$$y_1^* = 0$$

gelten. Wegen $x_1^* > 0$ und $x_2^* > 0$ muss außerdem gelten, dass die ersten beiden Ungleichungen von (D) mit Gleichheit erfüllt sind. Man erhält, wenn man $y_1^* = 0$ berücksichtigt, das folgende Gleichungssystem für y_2^* und y_3^*:

$$\begin{aligned} 2y_2^* + 4y_3^* &= 3 \\ 2y_2^* + y_3^* &= 1. \end{aligned} \tag{$\star\star$}$$

Dieses Gleichungssystem hat die eindeutige Lösung

$$y_2^* = \frac{1}{6}, \quad y_3^* = \frac{2}{3}.$$

Insgesamt gilt also

$$y_1^* = 0, \quad y_2^* = \frac{1}{6} \quad \text{und} \quad y_3^* = \frac{2}{3}\ .$$

Damit haben wir eindeutig bestimmte Zahlen y_1^*, y_2^*, y_3^* erhalten, die (1) erfüllen. Nun bleibt nur noch zu prüfen, ob auch (2) gilt. Einsetzen in (D) ergibt, dass die Antwort ja lautet.

Unser Test hat also ergeben, dass $x_1^* = 8$, $x_2^* = 4$, $x_3^* = 0$ eine optimale Lösung von (P) ist.

Beispiel 2. Wir betrachten dasselbe LP-Problem wie zuvor in Beispiel 1 und stellen uns vor, es wäre nicht die zulässige Lösung $x_1^* = 8$, $x_2^* = 4$, $x_3^* = 0$ auf Optimalität zu testen gewesen, sondern stattdessen

$$x_1^* = \frac{33}{4}, \quad x_2^* = 0, \quad x_3^* = \frac{3}{2}.$$

Aufgabe. Spielen Sie diesen Testfall durch![4]

7.5 Zur ökonomischen Bedeutung der dualen Variablen

Wir betrachten ein LP-Problem in Standardform:

$$\begin{aligned} &\text{maximiere} \quad \sum_{j=1}^{n} c_j x_j \\ &\text{unter den Nebenbedingungen} \\ &\qquad \sum_{j=1}^{n} a_{ij} x_j \ \leq b_i \qquad (i = 1, \ldots, m) \\ &\qquad\qquad x_j \ \geq \ 0 \qquad (j = 1, \ldots, n). \end{aligned} \tag{7.17}$$

Tritt ein LP-Problem in einem Anwendungszusammenhang auf, zum Beispiel in den *Wirtschaftswissenschaften*, so lassen sich die dualen Variablen $y_1, \ldots, y_m$ häufig inhaltlich interpretieren.

Einen Hinweis auf die inhaltliche Bedeutung der dualen Variablen liefert das folgende Beispiel.

Beispiel (Gewinn eines Kosmetikherstellers). Stellen wir uns vor, dass es bei (7.17) darum geht, den Gewinn eines Kosmetikherstellers zu maximieren. Dabei gibt jedes x_j das *Outputniveau für das j-te Produkt* an:

x_1 : Menge der pro Woche hergestellten Handcreme
x_2 : Menge der pro Woche hergestellten Gesichtscreme
$\vdots$
x_n : Menge der pro Woche hergestellten Körperlotion

Die im selben Zeitraum maximal zur Verfügung stehende Menge des i-ten Inhaltsstoffs (der *i-ten Ressource*) wird durch b_i angegeben:

Es stehen b_1 Einheiten gereinigtes Wasser zur Verfügung.
Es stehen b_2 Einheiten Glycerin zur Verfügung.
$\vdots$
Es stehen b_m Einheiten Olivenöl zur Verfügung.

[4] Anders gesagt: Man soll so tun, als hätte man die Lösung $x_1^* = 8$, $x_2^* = 4$, $x_3^* = 0$ noch gar nicht getestet, und soll stattdessen $x_1^* = \frac{33}{4}$, $x_2^* = 0$, $x_3^* = \frac{3}{2}$ testen.

Außerdem – das sollte klar sein – gibt c_j den Gewinn an, sagen wir in Dollar, den man mit einer Einheit des jeweiligen Produkts erzielt. Ferner: a_{ij} gibt an, wie viele Einheiten der i-ten Ressource pro Einheit des j-ten Produkts benötigt werden.

Wir schauen uns die Ungleichungen des dualen Problems an:

$$\sum_{i=1}^{m} a_{ij} y_i \geq c_j \qquad (j = 1, \ldots, n). \tag{7.18}$$

- *Rechts* haben wir es mit Dollar pro Einheit von Produkt j zu tun;
- *Links* haben wir es bei a_{ij} mit Einheiten von Ressource i pro Einheit von Produkt j zu tun.

Soll das zusammenpassen, so muss die Größe y_i also etwas in Dollar pro Einheit von Ressource i angeben $(i = 1, \ldots, m)$.

Die Größe y_i wird in Dollar pro Einheit von Ressource i gemessen und gibt deshalb einen *Preis oder Wert einer Einheit der i-ten Ressource an* $(i = 1, \ldots, m)$.

Dies soll nun präzisiert und ausgebaut werden. Entscheidendes Hilfsmittel ist der folgende Satz, den wir ohne Beweis angeben. (Bemerkungen zum Beweis findet man im Chvátal.)

Satz 4.
Falls das LP-Problem (7.17) mindestens eine nichtdegenerierte optimale Basislösung besitzt, so gibt es ein $K > 0$ mit folgender Eigenschaft: Falls $|t_i| \leq K$ für alle $i = 1, \ldots, m$ gilt, so besitzt das Problem

$$\begin{aligned} &\text{maximiere} \quad \sum_{j=1}^{n} c_j x_j \\ &\text{unter den Nebenbedingungen} \\ &\sum_{j=1}^{n} a_{ij} x_j \;\leq b_i + t_i \qquad (i = 1, \ldots, m) \\ &\qquad x_j \;\geq \quad 0 \qquad (j = 1, \ldots, n) \end{aligned} \tag{7.19}$$

eine optimale Lösung und der optimale Wert dieses Problems ist gleich

$$z^* + \sum_{i=1}^{m} y_i^* t_i. \tag{7.20}$$

Hierbei bezeichnet z^* den optimalen Wert von (7.17) und $y_1^*, \ldots, y_m^*$ bezeichnet die optimale Lösung[5] des dualen Problems von (7.17).

Dieser Satz beschreibt den Effekt, den kleine Veränderungen der zur Verfügung stehenden Ressourcen auf den Gewinn haben.

Für den Fall, dass $t_i > 0$ für ein i gilt, bedeutet die Formel (7.20):

Mit jeder zusätzlich zur Verfügung stehenden Einheit der i-ten Ressource nimmt der maximale Gewinn um y_i^* Dollar zu.

Man kann dies auch so formulieren:

y_i^* gibt den Höchstbetrag an, den die Firma für eine zusätzliche Einheit der i-ten Ressource zu zahlen bereit sein sollte.[6]

[5] Aufgrund der Voraussetzung von Satz 4, dass (7.17) mindestens eine nichtdegenerierte optimale Basislösung besitzt, kann gezeigt werden, dass das duale Problem eine *eindeutig bestimmte* optimale Lösung hat; insofern ist es gerechtfertigt, an dieser Stelle von *der* optimalen Lösung zu sprechen.

[6] Etwas genauer gilt: Mehr als y_i^* Dollar zu zahlen sollte die Firma nicht bereit sein; beträgt der Preis genau y_i^* Dollar, so liegt ein Grenzfall vor („weder Nutzen noch Schaden“).

Ist $t_i < 0$, so ergibt sich eine ähnliche Interpretation von y_i^*. Damit haben wir die gewünschte Interpretation der dualen Variablen erhalten.

Im beschriebenen Zusammenhang ist es üblich, y_i^* den *Schattenpreis* der i-ten Ressource zu nennen. Zur Illustration geben wir ein Beispiel.

Beispiel (Forstunternehmerin). Eine Forstunternehmerin besitzt 100ha Wald, der vollständig aus Laubbäumen besteht[7]. Es gibt die folgenden Möglichkeiten:

(i) Den Wald zu fällen und den Boden brach liegen zu lassen, würde zunächst \$10 Kosten pro ha verursachen und später durch den Verkauf des Holzes einen Erlös von \$50 pro ha einbringen.

(ii) Den Wald zu fällen und anschließend Pinien zu pflanzen, würde zunächst Kosten von \$50 pro ha verursachen; später würden jedoch (nach Abzug späterer Kosten) \$120 pro ha in die Kasse kommen.

Also: Die 2. Möglichkeit ist die bessere, da sie \$70 Gewinn pro ha verspricht, während der Gewinn bei der 1. Möglichkeit nur \$40 pro ha beträgt. Nun kann die 2. Möglichkeit aber nicht im vollen Umfang umgesetzt werden, da nur \$4000 zur Verfügung stehen, um die unmittelbar anfallenden Kosten zu bestreiten. Die Forstunternehmerin erkennt, dass sich ihr Problem so formulieren lässt:

$$\begin{aligned} &\text{maximiere } 40x_1 + 70x_2 \\ &\text{unter den Nebenbedingungen} \\ &\qquad x_1 + \quad x_2 \leq 100 \\ &\qquad 10x_1 + 50x_2 \leq 4000 \\ &\qquad x_1, x_2 \geq \quad 0. \end{aligned} \tag{7.21}$$

Die optimale Lösung ist $x_1^* = 25$ und $x_2^* = 75$.

Die Forstunternehmerin sollte also nach Fällung des gesamten Baumbestands 25% der Fläche brach liegen lassen und die restlichen 75% mit Pinien bepflanzen. Dies würde zunächst Investitionskosten von \$4000 verursachen; der Gewinn würde letztendlich \$6250 betragen.

Offensichtlich stellt das Anfangskapital von \$4000 eine wertvolle Ressource dar[8]. In der Tat wäre die Forstunternehmerin gut beraten, diese Ressource zu erhöhen und einen Kredit aufzunehmen: Die zu erwartenden zusätzlichen Einnahmen könnten möglicherweise sogar drastische Zinsen ausgleichen.

Beispielsweise könnte sie die Gelegenheit haben, sich \$100 zu borgen, für die sie später \$180 zurückzahlen müsste. **Sollte sie das machen?**

Die Antwort auf diese und ähnliche Fragen erhält man, wenn man die optimale Lösung des zu (7.21) dualen Problems berechnet; diese lautet:

$$y_1^* = 32.5, \quad y_2^* = 0.75.$$

Aufgrund der Erläuterungen zu Satz 4 und wegen $y_2^* = 0.75$ erkennt man: *Die Forstunternehmerin sollte (in begrenztem Umfang) Kapital aufnehmen, aber nur genau dann, wenn die zu zahlenden Zinsen kleiner als 75 Cent pro Dollar sind.*

Dieses Ergebnis hat sich aufgrund von Satz 4 ergeben. Es lässt sich aber auch direkt nachprüfen, wenn

[7] Im englischsprachigen Original (vgl. Chvátal: *Linear Programming*) ist von "100 acre of hardwood timber" die Rede (1 acre = $4047m^2$).

[8] Man beachte, dass es in diesem Beispiel zwei Ressourcen gibt: Laubwald und Kapital. Jeder der beiden Ressourcen entspricht eine Ungleichung von (7.21); auf der rechten Seite steht dabei die zur Verfügung stehende Menge der jeweiligen Ressource (100ha bzw. \$ 4000). Es ist also alles ganz ähnlich wie weiter oben im Beispiel des Kosmetikherstellers.

man sich anschaut, wie das in Satz 4 auftretende LP-Problem (7.19) in unserem Fall lautet:

$$\begin{aligned} &\text{maximiere } 40x_1 + 70x_2 \\ &\text{unter den Nebenbedingungen} \\ &\qquad\begin{aligned} x_1 + \quad x_2 &\le 100 \\ 10x_1 + 50x_2 &\le 4000 + t \\ x_1, x_2 &\ge 0. \end{aligned} \end{aligned} \tag{7.22}$$

Hierbei ist $4000 + t$ das zur Verfügung stehende Kapital. Die Größe t gibt das zusätzlich aufgenommene Kapital an.

Jede zulässige Lösung x_1, x_2 dieses Problems erfüllt die Ungleichung

$$40x_1 + 70x_2 = 32.5\big(x_1 + x_2\big) + 0.75\big(10x_1 + 50x_2\big) \le 3250 + 0.75(4000 + t) = 6250 + 0.75t. \tag{7.23}$$

Deshalb wird der zusätzliche Gewinn niemals größer als $0.75t$ sein.

Falls $t \le 1000$, so kann in der Tat ein zusätzlicher Gewinn von $0.75t$ erzielt werden, wenn man

$$x_1 = 25 - 0.025t, \quad x_2 = 75 + 0.025t \tag{7.24}$$

wählt. (Das Ergebnis (7.24) erhält man, wenn man (7.22) für festes t mit $0 \le t \le 1000$ mittels Simplexverfahren löst.)

Kreditaufnahme von mehr als \$1000 ist offensichtlich sinnlos, da insgesamt nicht mehr als \$5000 benötigt werden. Damit ist auch präzisiert, was mit der Formulierung gemeint ist, dass die Forstunternehmerin „in begrenztem Umfang“ Kapital aufnehmen sollte, falls die Zinsen kleiner als 75% sind: In unserem Fall bedeutet das $t \le 1000$.

Der Fall $t < 0$ lässt sich auf ganz ähnliche Art illustrieren: Anstelle der Gelegenheit, sich \$100 zu borgen, für die sie später \$180 zurückzahlen müsste, könnte sich für unsere Forstunternehmerin die Gelegenheit bieten, einen Teil ihrer \$4000 abzuzweigen und in ein anderes lukratives Unternehmen zu investieren. Beispielsweise könnte sie die Gelegenheit haben, \$100 zu investieren, um dann später \$180 zurückzubekommen.

Eine solche Investition in ein anderes Unternehmen führt zu einem negativen t in (7.22). Die Ungleichung (7.23) bleibt aber auch für $t < 0$ gültig; diese Ungleichung bedeutet für $t < 0$:

Falls $-t$ Dollar für eine alternative Investition abgezweigt werden ($-t$ ist positiv!), so beträgt der Gewinn aus der ursprünglichen Unternehmung nur noch höchstens $6250 + 0.75t$, d.h., der Gewinn aus der ursprünglichen Unternehmung fällt um mindestens $0.75(-t)$.

Falls man x_1 und x_2 wie in (7.24) wählt und falls $-t \le 3000$ gilt, so kann in der Tat erreicht werden, dass die Verringerung des Gewinns aus der ursprünglichen Unternehmung genau $0.75(-t)$ beträgt.

Die Berechnungen laufen in beiden Fällen ($t \ge 0$ und $t < 0$) also völlig gleich, nur der Gültigkeitsbereich von (7.24) ist im ersten Fall $0 \le t \le 1000$ und im zweiten Fall $-3000 \le t < 0$.

Zusammenfassung:

1. Hat die Forstunternehmerin die Gelegenheit, weiteres Kapital aufzunehmen, so sollte sie es tun, aber nur bis zu \$1000 und nur, falls die zu zahlenden Zinsen kleiner als 75 Cent pro Dollar sind.
2. Hat sie die Gelegenheit, einen Teil ihres Geldes in ein anderes Unternehmen zu investieren, so kann zugeraten werden, falls sie für einen Dollar mehr als 175 Cent zurückbekommt. Dies gilt aber nur dann, wenn die Investitionssumme nicht höher als \$3000 ist.

Dies ist nun aber noch nicht ganz das Ende der Geschichte.

Man stelle sich einmal vor, dass es eine überraschende Gelegenheit für unsere Forstunternehmerin gibt, ein möglicherweise noch besseres Geschäft zu machen. Beispielsweise könnte sich die Gelegenheit ergeben, den Wald zu fällen und – sagen wir – Koniferen zu pflanzen.

Um eine schnelle Beurteilung der Geschäftsaussichten zu erhalten, kann die Forstunternehmerin die *Schattenpreise* ihrer Ressourcen heranziehen:

$32.5 pro ha Laubwald;
$0.75 pro Dollar Kapital.

Falls die neue Aktivität d Dollar Investitionskosten pro ha erfordert, dann haben die Ressourcen, die durch die neue Unternehmung pro ha verbraucht werden, einen Wert von $\$(32.5 + 0.75d)$. Die neue Aktivität kommt demnach genau dann in Betracht, wenn der Gewinn pro ha diesen Wert übersteigt.

7.6 Dualität im Fall eines allgemeinen LP-Problems

Für jedes LP-Problem (P) in Standardform haben wir definiert, was wir unter dem dazugehörigen dualen Problem (D) verstehen. Es besitzen jedoch nicht nur Probleme in Standardform ein duales Problem, *sondern zu jedem LP-Problem gehört ein duales Problem.*

Wie sieht nun im allgemeinen Fall das duale Problem aus? Die Antwort auf diese Frage werden wir weiter unten in Form eines *Rezepts* präsentieren. Bevor wir dies tun, soll jedoch erläutert werden, was wir unter dem „allgemeinen Fall“ genau verstehen.

Klarerweise bedeutet es keine Einschränkung der Allgemeinheit, wenn wir annehmen, dass wir als Ausgangsproblem (primales Problem) ein Maximierungsproblem vorliegen haben: Jedes Minimierungsproblem lässt sich ja – wie wir wissen – auf eine ganz einfache Art in ein Maximierungsproblem verwandeln. (Wie nämlich?)

Ebenfalls bedeutet es keine Einschränkung, wenn wir annehmen, dass im primalen Problem (abgesehen von Nichtnegativitätsbedingungen) keine Ungleichungen des Typs $\sum_{j=1}^{n} a_{ij}x_j \geq b_i$ auftreten: Kommt eine solche Ungleichung (beispielsweise $3x_1 + 2x_2 - 5x_3 \geq 6$) vor, so braucht man diese ja nur mit -1 zu multiplizieren.

Als Nebenbedingungen, die keine Nichtnegativitätsbedingungen sind, können im primalen Problem (P) im allgemeinen Fall also auftreten:

- Ungleichungen vom Typ $\sum_{j=1}^{n} a_{ij}x_j \leq b_i$, wie beispielsweise $27x_1 - x_2 + 2x_3 \leq 5$;
- Gleichungen.

In ähnlicher Weise können im allgemeinen Fall zwei Typen von Variablen auftreten:

- Variablen, die einer Nichtnegativitätsbedingung unterliegen ($x_j \geq 0$);
- freie Variablen, d.h. Variablen, für die nicht $x_j \geq 0$ gefordert wird.

Ein allgemeines LP-Problem (P) ist beispielsweise:

$$
\begin{array}{l}
\text{maximiere} \quad 7x_1 + 3x_2 + x_3 + 5x_4 \\
\text{unter den Nebenbedingungen} \\
\qquad \begin{array}{rcl}
-2x_1 + x_2 + x_3 - 3x_4 & \leq & 1 \\
5x_1 + x_2 + 9x_4 & \leq & -2 \\
x_1 + 3x_2 + x_3 + 7x_4 & = & 5 \\
x_1 + x_3 - 6x_4 & = & 1 \\
x_2, x_3 & \geq & 0.
\end{array}
\end{array}
$$

Hierbei sind x_1 und x_4 freie Variablen.

Im Unterschied zu den bislang betrachteten primalen Problemen in Standardform können jetzt also *zusätzlich Gleichungen und freie Variablen* auftreten.

Wir gehen also von einem LP-Problem (P) des folgenden Typs aus:

$$\begin{aligned}&\text{maximiere } \sum_{j=1}^{n} c_j x_j\\ &\text{unter den Nebenbedingungen}\\ &\sum_{j=1}^{n} a_{ij} x_j \;\leq b_i \qquad (i \in I_1)\\ &\sum_{j=1}^{n} a_{ij} x_j \;= b_i \qquad (i \in I_2)\\ &\qquad x_j \;\geq\; 0 \qquad (j \in J_1).\end{aligned} \tag{7.25}$$

Was in (7.25) und im Folgenden die Bezeichnungen in I_1, I_2 und J_1 bedeuten, liegt auf der Hand:

- I_1 ist die Menge der Indizes $i \in \{1, \ldots, m\}$, für die die i-te Nebenbedingung eine *Ungleichung* ist;
- I_2 ist die Menge der Indizes $i \in \{1, \ldots, m\}$, für die die i-te Nebenbedingung eine *Gleichung* ist;
- J_1 ist die Menge der Indizes $j \in \{1, \ldots, n\}$, die zu *restringierten Variablen* x_j gehören, also zu denjenigen Variablen, die einer Nichtnegativitätsbedingung unterliegen.

Ergänzend definieren wir noch $J_2 = \{1, \ldots, n\} \setminus J_1$.

- Die Menge J_2 enthält diejenigen Indizes, die zu *freien Variablen* gehören.

Es ist durchaus möglich, dass einige dieser Indexmengen leer sind: Gilt beispielsweise $I_1 = \emptyset$, so treten in (P) nur Gleichungen auf; oder (eine andere Möglichkeit): Es gilt $J_1 = \emptyset$, d.h., wir haben es nur mit freien Variablen zu tun.

Gilt $I_2 = J_2 = \emptyset$, so liegt der Fall eines LP-Problems in Standardform vor.

Unter einer *Linearkombination der Nebenbedingungen*

$$\begin{aligned}&\sum_{j=1}^{n} a_{ij} x_j \;\leq b_i \qquad (i \in I_1)\\ &\sum_{j=1}^{n} a_{ij} x_j \;= b_i \qquad (i \in I_2)\end{aligned} \tag{7.26}$$

verstehen wir eine lineare Ungleichung, die dadurch entsteht, dass man jede dieser Nebenbedingungen mit einem Faktor y_i multipliziert, wobei $y_i \geq 0$ für alle $i \in I_1$ gelten soll, und anschließend die entstandenen Ungleichungen und Gleichungen aufsummiert.

Die hierdurch entstandene neue Ungleichung (Linearkombination) lautet also

$$\sum_{i=1}^{m} y_i \left(\sum_{j=1}^{n} a_{ij} x_j \right) \leq \sum_{i=1}^{m} b_i y_i. \tag{7.27}$$

Wichtig, damit die Ungleichung (7.27) tatsächlich zustande kommt: Für die Ungleichungen aus (7.26) müssen nichtnegative Faktoren gewählt werden. (Die Faktoren y_i, die zu den Gleichungen gehören, sind hingegen frei wählbar.)

Wir wollen die Faktoren y_i als *duale Variablen* bezeichnen; es gilt also:

- Zu jeder *Ungleichung* von (7.26) gehört eine *restringierte duale Variable* y_i, d.h., es wird $y_i \geq 0$ gefordert.
- Zu jeder *Gleichung* von (7.26) gehört eine *freie duale Variable* y_i.

Änderung der Summationsreihenfolge auf der linken Seite von (7.27) ergibt

$$\sum_{i=1}^{m} y_i \left(\sum_{j=1}^{n} a_{ij} x_j \right) = \sum_{j=1}^{n} \left(\sum_{i=1}^{m} a_{ij} y_i \right) x_j. \tag{7.28}$$

Ersetzt man die linke Seite von (7.27) entsprechend, so geht unsere Linearkombination in die folgende Darstellung über:

$$\sum_{j=1}^{n}\left(\sum_{i=1}^{m} a_{ij}y_i\right)x_j \le \sum_{i=1}^{m} b_i y_i. \tag{7.29}$$

Gelten für eine Wahl der Variablen $x_1,\dots,x_n$ sämtliche Nebenbedingungen (7.26), so gilt für diese Wahl von $x_1,\dots,x_n$ ebenfalls (7.29).

Bislang haben wir nur die Nebenbedingungen von (7.25) betrachtet – *nun kommen zusätzlich die Koeffizienten $c_1,\dots,c_n$ der Zielfunktion ins Spiel.*

Falls die Zahlen $y_1,\dots,y_m$ so gewählt werden, dass

$$\sum_{i=1}^{m} a_{ij}y_i \ge c_j \quad \text{für alle } j \in J_1 \tag{7.30}$$

und

$$\sum_{i=1}^{m} a_{ij}y_i = c_j \quad \text{für alle } j \in J_2 \tag{7.31}$$

gilt, so folgt für jede zulässige Lösung $x_1,\dots,x_n$ von (7.25):

$$c_j x_j \le \left(\sum_{i=1}^{m} a_{ij}y_i\right)x_j \quad \text{für alle } j \in J_1$$

und

$$c_j x_j = \left(\sum_{i=1}^{m} a_{ij}y_i\right)x_j \quad \text{für alle } j \in J_2,$$

woraus man durch Aufsummieren

$$\sum_{j=1}^{n} c_j x_j \le \sum_{j=1}^{n}\left(\sum_{i=1}^{m} a_{ij}y_i\right)x_j$$

erhält und somit (aufgrund von (7.29))

$$\sum_{j=1}^{n} c_j x_j \le \sum_{i=1}^{m} b_i y_i. \tag{7.32}$$

Zusammenfassend können wir feststellen:

Feststellung.

Falls die Zahlen $y_1,\dots,y_m$ die Bedingungen (7.30) und (7.31) erfüllen und falls außerdem $y_i \ge 0$ für alle $i \in I_1$ gilt, so ist die Zahl

$$\sum_{i=1}^{m} b_i y_i$$

eine *obere Schranke* für den optimalen Wert von (7.25).

Natürlich wünscht man sich eine *möglichst gute obere Schranke*, was zu folgendem LP-Problem führt:

$$\begin{aligned}
&\text{minimiere} \quad \sum_{i=1}^{m} b_i y_i \\
&\text{unter den Nebenbedingungen} \\
&\sum_{i=1}^{m} a_{ij} y_i \;\geq c_j \qquad (j \in J_1) \\
&\sum_{i=1}^{m} a_{ij} y_i \;= c_j \qquad (j \in J_2) \\
&\qquad y_i \;\geq\; 0 \qquad (i \in I_1).
\end{aligned} \tag{7.33}$$

Man nennt (7.33) das *duale Problem* (oder kurz: das *Duale*) zu (7.25); in diesem Zusammenhang wird (7.25) das *primale Problem* genannt.

Das duale Problem (D) für das Beispiel (P) vor (7.25) lautet also:

$$\begin{aligned}
&\text{minimiere} \quad y_1 - 2y_2 + 5y_3 + y_4 \\
&\text{unter den Nebenbedingungen} \\
&-2y_1 + 5y_2 + y_3 + y_4 = 7 \\
&\;\; y_1 + y_2 + 3y_3 \qquad \geq 3 \\
&\;\; y_1 \qquad + y_3 + y_4 \geq 1 \\
&-3y_1 + 9y_2 + 7y_3 - 6y_4 = 5 \\
&\qquad\qquad\qquad\qquad y_1, y_2 \geq 0.
\end{aligned}$$

Wir können uns das folgende *Dualisierungsrezept* merken.

Dualisierungsrezept.

- Ist die i-te Nebenbedingung im primalen Problem (P) eine Ungleichung, so ist y_i in (D) eine restringierte Variable.
- Ist die i-te Nebenbedingung in (P) eine Gleichung, so ist y_i eine freie Variable.
- Ist x_j eine freie Variable, so ist in (D) die j-te Nebenbedingung eine Gleichung.
- Ist x_j eine restringierte Variable, so ist in (D) die j-te Nebenbedingung eine Ungleichung.

Dasselbe in Tabellenform:

Maximierungsproblem (P)	*Minimierungsproblem (D)*
i-te Nebenbedingung enthält $\leq$	$y_i \geq 0$
i-te Nebenbedingung enthält $=$	y_i ist frei
$x_j \geq 0$	j-te Nebenbedingung enthält $\geq$
x_j ist frei	j-te Nebenbedingung enthält $=$

In Worten lässt sich das *Dualisierungsrezept* ebenfalls sehr eingängig ausdrücken:

Ungleichungen entsprechen restringierten Variablen im jeweils anderen Problem;
Gleichungen entsprechen freien Variablen.

Beispiel. Konstruieren Sie das Duale (D) des folgenden LP-Problems, das wir (P) nennen:

$$\begin{aligned}
&\text{maximiere } 3x_1 + 2x_2 + 5x_3 \\
&\text{unter den Nebenbedingungen} \\
&\qquad\begin{aligned}
5x_1 + 3x_2 + x_3 &= -8 \\
4x_1 + 2x_2 + 8x_3 &\leq 23 \\
6x_1 + 7x_2 + 3x_3 &\geq 1 \\
x_1 &\leq 4 \\
x_3 &\geq 0.
\end{aligned}
\end{aligned}$$

Lösung. Bevor wir das Dualisierungsrezept anwenden können, muss zunächst die 3. Nebenbedingung mit -1 multipliziert werden. Man erhält:

$$\begin{aligned}
&\text{maximiere } 3x_1 + 2x_2 + 5x_3 \\
&\text{unter den Nebenbedingungen} \\
&\qquad\begin{aligned}
5x_1 + 3x_2 + x_3 &= -8 \\
4x_1 + 2x_2 + 8x_3 &\leq 23 \\
-6x_1 - 7x_2 - 3x_3 &\leq -1 \\
x_1 &\leq 4 \\
x_3 &\geq 0.
\end{aligned}
\end{aligned}$$

Nun können wir das Dualisierungsrezept anwenden und erhalten (D):

$$\begin{aligned}
&\text{minimiere } -8y_1 + 23y_2 - y_3 + 4y_4 \\
&\text{unter den Nebenbedingungen} \\
&\qquad\begin{aligned}
5y_1 + 4y_2 - 6y_3 + y_4 &= 3 \\
3y_1 + 2y_2 - 7y_3 &= 2 \\
y_1 + 8y_2 - 3y_3 &\geq 5 \\
y_2, y_3, y_4 &\geq 0.
\end{aligned}
\end{aligned}$$

Die folgende Feststellung ist unschwer einzusehen.

Feststellung.
Bildet man von einem Problem (P) das Duale (D) und anschließend das Duale von (D), so gelangt man zurück zum primalen Problem (P), wobei es vorkommen kann, dass man (P) in leicht umgeschriebener Form erhält.

Wir halten fest: *Das Duale des dualen Problems ist das primale Problem.*

Ist (P) ein Minimierungsproblem und soll von (P) das Duale gebildet werden, so kann man zunächst einmal eine Überführung von (P) in ein Maximierungsproblem vornehmen. Danach lässt sich das Duale bilden, indem man wie in unserem letzten Beispiel vorgeht.

Da das Duale des dualen Problems das primale Problem ist, gibt es hierzu eine *alternative Vorgehensweise*: Man wendet die Tabelle, in der das Dualisierungsrezept beschrieben wird, *von rechts nach links* an. Dabei liest man die Tabelle von rechts nach links:

> Liegt ein Minimierungsproblem (D) vor, von dem das Duale zu bilden ist, so sorgt man zunächst dafür, dass alle Ungleichungsnebenbedingungen von der Form „$\geq$“ sind; dann kann man das Duale von (D) bilden, indem man zum Problem (P) der linken Spalte übergeht.

Für allgemeine LP-Probleme gelten ähnliche Sätze wie für LP-Probleme in Standardform. Dies trifft beispielsweise auf den Dualitätssatz zu. Auch für allgemeine LP-Probleme lässt sich der Dualitätssatz wie folgt formulieren (Beweis: siehe Chvatal).

Satz (Dualitätssatz für allgemeine LP-Probleme).
Falls ein LP-Problem eine optimale Lösung besitzt, so gilt dies auch für das duale Problem und die optimalen Werte beider Probleme stimmen überein.

7.7 Das Lemma von Farkas

Wir haben in Abschnitt 7.3 den Dualitätssatz mithilfe des Simplexalgorithmus bewiesen. Eine andere Möglichkeit, den Dualitätssatz zu beweisen, macht vom Lemma von Farkas Gebrauch. Da es sich bei dem „Lemma von Farkas“ genannten Satz um eine sehr bekannte und häufig benutzte Aussage handelt, die keineswegs nur beim Beweis des Dualitätssatzes eine Rolle spielt, soll das Lemma von Farkas hier vorgestellt werden.

Satz (Lemma von Farkas).
Es sei A eine reelle $m \times n$ - Matrix und $b \in \mathbb{R}^m$ sei ein Vektor. Dann ist genau eine der beiden Aussagen richtig:

(i) Es existiert ein $x \in \mathbb{R}^n$, für das $Ax = b$ und $x \geq 0$ gilt.

(ii) Es existiert ein $y \in \mathbb{R}^m$, für das $y^T A \geq 0$ und $y^T b < 0$ gilt[9].

Dass (I) und (II) nicht gleichzeitig gelten können, ist einfach einzusehen: Angenommen, es würden (I) und (II) gleichzeitig gelten. Dann erhielte man aus $y^T A \geq 0$ und $x \geq 0$, dass $(y^T A)x \geq 0$ gilt, woraus sich (unter Benutzung von $Ax = b$) Folgendes ergibt:

$$y^T b = y^T (Ax) = (y^T A)x \geq 0.$$

Dies steht im Widerspruch zu $y^T b < 0$, womit gezeigt ist, dass (I) und (II) nicht gleichzeitig gelten können.

Die eigentliche Aussage des Lemmas von Farkas ist die Feststellung, dass immer eine der beiden Aussagen (I) und (II) gelten muss.

Es gibt *verschiedene Versionen* des Lemmas von Farkas. Da es uns hier besonders auf die geometrische Interpretation des Lemmas von Farkas ankommt, haben wir eine Version ausgewählt, die für die geometrische Deutung besonders gut geeignet ist. Später werden wir noch eine andere Version kennenlernen.

Wir kommen zur *geometrischen Interpretation des Lemmas von Farkas.* Hierbei spielen die Spalten von A eine wichtige Rolle. Wir bezeichnen die Spalten von A mit

$$a_1, \ldots, a_n.$$

Bei den Spalten $a_1, \ldots, a_n$ handelt es sich um Vektoren des $\mathbb{R}^m$ – und ebenso ist b ein Vektor des $\mathbb{R}^m$. Bezeichnen wir die Einträge von x mit $x_1, \ldots, x_n$ (mit anderen Worten: $x^T = (x_1, \ldots, x_n)$), so können wir (I) auch wie folgt ausdrücken: Es existiert eine Linearkombination $x_1 a_1 + \ldots + x_n a_n$ der Spalten von A *mit nichtnegativen Koeffizienten* $x_i \in \mathbb{R}$ $(i = 1, \ldots, n)$, so dass gilt:

$$x_1 a_1 + \ldots + x_n a_n = b.$$

In geometrischer Terminologie lässt sich (I) demnach wie folgt aussprechen (vgl. Abschnitt 4.4):

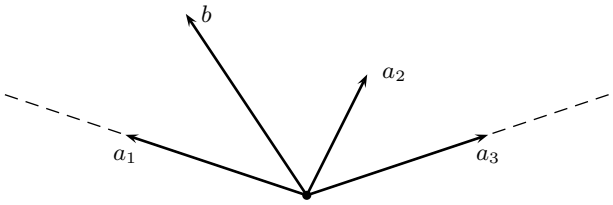
[9] Die Bedeutung des Symbols „0“ erklärt sich aus dem jeweiligen Zusammenhang: In der Ungleichung $x \geq 0$ ist 0 ein Spaltenvektor, in $y^T A \geq 0$ ist 0 ein Zeilenvektor und in $y^T b < 0$ ist 0 eine reelle Zahl.

(I) Der Vektor b liegt im konvexen Kegel, der von der Menge $\{a_1, \ldots, a_n\}$ erzeugt wird.

Dasselbe noch etwas knapper geschrieben:

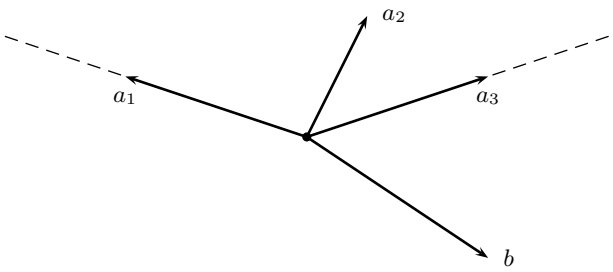
$$b \in \text{cone}(\{a_1, \ldots, a_n\}).$$

In der folgenden Zeichnung wird (I) für den Fall illustriert, dass $m = 2$ und $n = 3$ gilt („drei Vektoren a_1, a_2, a_3 im $\mathbb{R}^2$“):



Der Fall $b \in \text{cone}\left(\{a_1, a_2, a_3\}\right)$

Der Fall, dass (I) nicht gilt, d.h., dass b *nicht* im konvexen Kegel cone$(\{a_1, \ldots, a_n\})$ enthalten ist, wird in der nächsten Zeichnung dargestellt:



Der Fall $b \notin \text{cone}\left(\{a_1, a_2, a_3\}\right)$

Soviel zur geometrischen Interpretation von (I) bzw. zum Fall, dass (I) nicht gilt. Wir kommen nun zur geometrischen Interpretation von (II).

Zunächst beobachten wir, dass der Vektor $y \in \mathbb{R}^m$, der in (II) vorkommt, nicht der Nullvektor sein kann, da andernfalls nicht $y^T b < 0$ gelten würde. Zu y gehört also eine Hyperebene h des $\mathbb{R}^m$:

$$h = \left\{x \in \mathbb{R}^m : y^T x = 0\right\}.$$

Mit anderen Worten: *h ist die Menge aller Vektoren des $\mathbb{R}^m$, die senkrecht auf y stehen.*

Man beachte, dass h den Nullvektor enthält; im Fall $m = 2$ ist h also eine *Ursprungsgerade* und im Fall $m = 3$ ist h eine *Ursprungsebene.*

Sprechweise: Es seien y und h wie zuvor, d.h., es gelte $y \in \mathbb{R}^m$, $y \neq 0$ und $h = \{x \in \mathbb{R}^m : y^T x = 0\}$. Außerdem seien $b \in \mathbb{R}^m$ und eine Menge $C \subseteq \mathbb{R}^m$ gegeben. Gilt $y^T c \geq 0$ für alle $c \in C$ sowie $y^T b < 0$, so sagt man, *dass h den Vektor b von den Vektoren aus C trennt.* Oder etwas kürzer:

h trennt b von C.

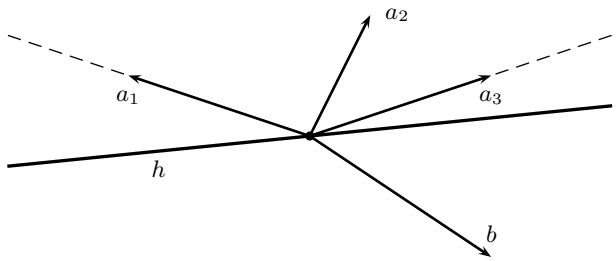
Man kann dies auch so ausdrücken: *Alle Vektoren von C liegen auf einer Seite von h und b liegt nicht auf dieser Seite.*

Damit ist klar, wie (II) geometrisch zu interpretieren ist: Die Ungleichungen $y^T A \geq 0$ und $y^T b < 0$ besagen, dass die Hyperebene $h = \{x \in \mathbb{R}^m : y^T x = 0\}$ den Vektor b von den Spaltenvektoren $a_1, \ldots, a_n$ der Matrix A trennt.

Es sei $C = \text{cone}(\{a_1, \ldots, a_n\})$ und es gelte $c \in C$. Man beachte (Beweis als Übungsaufgabe!): Aus $y^T A \geq 0$ folgt, dass ebenfalls $y^T c \geq 0$ gilt.

Die Ungleichungen $y^T A \geq 0$ und $y^T b < 0$ besagen also noch etwas mehr, nämlich dass die Hyperebene $h = \{x \in \mathbb{R}^m : y^T x = 0\}$ den Vektor b nicht nur von $a_1, \ldots, a_n$ trennt, sondern sogar vom gesamten konvexen Kegel $C = \text{cone}(\{a_1, \ldots, a_n\})$.

Natürlich haben wir unsere Sprechweisen wie z.B. „h trennt b von C“ nicht willkürlich gewählt, sondern so, dass sie für die Fälle $\mathbb{R}^2$ und $\mathbb{R}^3$ mit unserer geometrischen Anschauung übereinstimmen. In der folgenden Zeichnung wird der Fall $m = 2$ illustriert:



Unter Benutzung der eingeführten geometrischen Sprechweisen können wir das Lemma von Farkas wie folgt formulieren:

Lemma von Farkas („geometrische Formulierung“).
Gegeben seien Vektoren $a_1, \ldots, a_n$ des $\mathbb{R}^m$ sowie $b \in \mathbb{R}^m$. Dann ist genau eine der beiden Aussagen richtig:

(I) Der Vektor b ist im konvexen Kegel $\text{cone}(\{a_1, \ldots, a_n\})$ enthalten.

(II) Es gibt eine durch den Ursprung gehende Hyperebene des $\mathbb{R}^m$, die b von $\text{cone}(\{a_1, \ldots, a_n\})$ trennt.

Aus der geometrischen Formulierung wird übrigens klar, dass es sich beim Lemma von Farkas im Fall $m = 2$ um eine Feststellung handelt, die aufgrund der geometrischen Anschauung selbstverständlich richtig ist: Wenn b außerhalb des konvexen Kegels $C = \text{cone}(\{a_1, \ldots, a_n\})$ liegt, so gibt es eine Ursprungsgerade h, die b von C trennt. (Man mache sich klar, dass Entsprechendes auch im Fall $m = 3$ gilt: Auch in diesem Fall ist die Aussage des Lemmas von Farkas anschaulich einleuchtend; die Rolle der trennenden Ursprungsgerade wird hier von einer trennenden Ursprungsebene übernommen.)

Nachdem wir die geometrischen Aspekte des Lemmas von Farkas behandelt haben, kommen wir auf *Varianten des Lemmas von Farkas* zu sprechen. Wir wollen uns hier nur eine Variante anschauen; einen umfassenden Überblick über weitere Varianten und über eng verwandte Sätze erhält man im Buch von Schrijver:

- A. Schrijver: *Theory of Linear and Integer Programming.* Wiley (1998).

Bei der Variante des Lemmas von Farkas, die wir auf Seite 88 formuliert haben, geht es in (I) um ein Gleichungssystem $Ax = b$. Betrachtet man anstelle von $Ax = b$ das Ungleichungssystem $Ax \leq b$, so lautet die entsprechende Variante des Lemmas von Farkas wie folgt.

Lemma von Farkas (Version für $Ax \leq b$ und $x \geq 0$).
Es sei A eine reelle $m \times n$ - Matrix und $b \in \mathbb{R}^m$ sei ein Vektor. Dann ist genau eine der beiden Aussagen richtig:

(I') Es existiert ein $x \in \mathbb{R}^n$, für das $Ax \leq b$ und $x \geq 0$ gilt.

(II') Es existiert ein $y \in \mathbb{R}^m$, für das $y^T A \geq 0$, $y^T b < 0$ und $y \geq 0$ gilt.

Die beiden Versionen des Lemmas von Farkas sind äquivalent im folgenden Sinne: Aus der Version von Seite 88 (Version für $Ax = b$ und $x \geq 0$) lässt sich unschwer die neue Version (Version für $Ax \leq b$ und $x \geq 0$) folgern, und umgekehrt lässt sich ebenso leicht die ursprüngliche Version aus der neuen Version gewinnen. *Wir zeigen, wie sich die neue Version aus der ursprünglichen ergibt*: Zu diesem Zweck betrachten wir die Matrix

$$A^+ = \big(A \mid I_m\big).$$

Erläuterung dieser Schreibweise: I_m ist eine $m \times m$ - Einheitsmatrix; die neue Matrix A^+ entsteht also aus A, indem man zu A weitere m Spalten rechts hinzunimmt, wobei die neuen Spalten die Einheitsmatrix I_m bilden.

Man beachte, dass (I') äquivalent zur folgenden Aussage ist, die wir (I$^+$) nennen wollen.

(I$^+$) Es existiert ein $\overline{x} \in \mathbb{R}^{n+m}$, für das $A^+\overline{x} = b$ und $\overline{x} \geq 0$ gilt[10].

Außerdem ist (II') äquivalent zur folgenden Aussage:

(II$^+$) Es existiert ein $y \in \mathbb{R}^m$, für das $y^T A^+ \geq 0$ und $y^T b < 0$ gilt[11].

Wendet man die ursprüngliche Version des Lemmas von Farkas auf die Matrix A^+ und den Vektor b an, so erhält man, dass genau eine der beiden Aussagen (I$^+$) und (II$^+$) richtig ist. Es folgt (wegen (I') $\Leftrightarrow$ (I$^+$) und (II') $\Leftrightarrow$ (II$^+$)), dass genau eine der Aussagen (I') und (II') richtig ist. □

Nach einem ähnlichen Schema zeigt man, das auch umgekehrt die ursprüngliche Version des Lemmas von Farkas aus der neuen Version folgt. (Der Beweis sei dem Leser als Übungsaufgabe überlassen. *Hinweis*: Man denke unter anderem daran, dass sich eine lineare Gleichung äquivalent durch zwei Ungleichungen ausdrücken lässt.)

In vielen Lehrbüchern der Linearen Optimierung wird wie folgt vorgegangen: Zunächst wird das Lemma von Farkas in einer der üblichen Varianten bewiesen; danach wird das Lemma von Farkas benutzt, um den Dualitätssatz zu beweisen. *Da wir in der glücklichen Lage sind, den Dualitätssatz bereits bewiesen zu haben, können wir auch umgekehrt vorgehen*: Im Folgenden werden wir das Lemma von Farkas aus dem Dualitätssatz herleiten.

Genauer: *Wir weisen nach, dass sich das Lemma von Farkas (in der Version für $Ax \leq b$ und $x \geq 0$) auf eine sehr einfache Art aus dem Dualitätssatz ergibt.*

Hier ist der angekündigte **Beweis des Lemmas von Farkas**: Die Bezeichnungen A und b seien wie im Lemma von Farkas gewählt. Zu gegebenem A und b stellen wir das folgende LP-Problem auf, das wir (P) nennen:

$$\begin{aligned} &\text{maximiere } c^T x \\ &\text{unter den Nebenbedingungen} \\ &\qquad Ax \;\leq\; b \\ &\qquad\; x \;\geq\; 0 \end{aligned} \tag{P}$$

Die Bedeutung von A und b haben wir bereits besprochen. *Was ist nun aber c?* Antwort: c ist nichts weiter als der *Nullvektor* der Länge n.

[10] Der Übergang von (I') zu (I$^+$) entspricht der Einführung von Schlupfvariablen.
[11] Beim Übergang von (II') zu (II$^+$) wurden $y^T A \geq 0$ und $y \geq 0$ zu $y^T A^+ \geq 0$ zusammengefasst.

Dies ist ein *Trick*, den man sich merken sollte, da er auch in anderen Situationen nützlich ist: *Die Wahl von $c = 0$ bewirkt, dass jede zulässige Lösung von (P) eine optimale Lösung ist.*

Um das Lemma von Farkas zu beweisen, nehmen wir an, dass (I') nicht gilt. Wir weisen nach, dass dann (II') gelten muss. Dass (I') nicht gilt bedeutet, dass (P) keine zulässige Lösung besitzt. Aufgrund des Dualitätssatzes ist dann das zu (P) duale Problem (D) *unbeschränkt*. Das Duale (D) lautet wie folgt (Man beachte $c = 0$!):

$$\begin{aligned} &\text{minimiere} && b^T y \\ &\text{unter den Nebenbedingungen} \\ &&& A^T y \geq 0 \\ &&& y \geq 0 \end{aligned} \tag{D}$$

Da (D) unbeschränkt ist, existiert ein $y \in \mathbb{R}^m$, für das $A^T y \geq 0$, $b^T y < 0$ und $y \geq 0$ gilt. Das bedeutet aber, dass (II') gilt. (Man beachte, dass $y^T b = b^T y$ gilt; außerdem ist $A^T y \geq 0$ äquivalent zu $y^T A \geq 0$.)

Damit ist der Beweis des Lemmas von Farkas im Wesentlichen fertig. Es fehlt nur noch der Nachweis, dass (I') und (II') nicht gleichzeitig erfüllt sein können. Dies sei dem Leser als Übungsaufgabe überlassen. (*Hinweis*: Man gehe ähnlich vor wie auf Seite 88.) □

Wir schließen den Abschnitt über das Lemma von Farkas mit einigen *Literaturhinweisen*.

Neben dem Buch von Schrijver bietet auch das Buch von Matoušek/Gärtner einen guten Überblick über die unterschiedlichen Varianten des Lemmas von Farkas.

Interessante Anwendungen des Lemmas von Farkas findet man beispielsweise in

- D. Bertsimas, J. N. Tsitsiklis: *Introduction to Linear Optimization*. Athena Scientific (1997).
- N. Lauritzen: *Undergraduate Convexity*. World Scientific (2013).

8 Die revidierte Simplexmethode

8.1 Matrixdarstellung

Wir wollen zunächst das Simplexverfahren auf eine etwas andere Art beschreiben; in dieser neuen Beschreibung werden in wesentlich stärkerer Weise *Matrizen* zum Einsatz kommen.

Im Gegensatz zum vorausgegangenen Abschnitt betrachten wir in diesem Abschnitt ausschließlich LP-Probleme in Standardform. Wir erläutern die neue Matrizendarstellung anhand des folgenden Beispiels:

$$\begin{aligned}
&\text{maximiere } 19x_1 + 13x_2 + 12x_3 + 17x_4 \\
&\text{unter den Nebenbedingungen} \\
&\qquad\begin{aligned}
3x_1 + 2x_2 + x_3 + 2x_4 &\leq 225 \\
x_1 + x_2 + x_3 + x_4 &\leq 117 \\
4x_1 + 3x_2 + 3x_3 + 4x_4 &\leq 420 \\
x_1, x_2, x_3, x_4 &\geq 0.
\end{aligned}
\end{aligned} \tag{8.1}$$

Löst man (8.1) mit dem Simplexverfahren, so erhält man nach zwei Iterationen das folgende Tableau:

$$\begin{aligned}
x_1 &= 54 - 0.5x_2 - 0.5x_4 - 0.5x_5 + 0.5x_6 \\
x_3 &= 63 - 0.5x_2 - 0.5x_4 + 0.5x_5 - 1.5x_6 \\
x_7 &= 15 + 0.5x_2 - 0.5x_4 + 0.5x_5 + 2.5x_6 \\
\hline
z &= 1782 - 2.5x_2 + 1.5x_4 - 3.5x_5 - 8.5x_6.
\end{aligned} \tag{8.2}$$

Es sei an den Zusammenhang erinnert, der zwischen den ersten drei Zeilen von (8.2) und den ersten drei Ungleichungen von (8.1) besteht: Nach der Einführung von Schlupfvariablen gehen die ersten drei Ungleichungen von (8.1) über in

$$\begin{aligned}
3x_1 + 2x_2 + x_3 + 2x_4 + x_5 &= 225 \\
x_1 + x_2 + x_3 + x_4 + x_6 &= 117 \\
4x_1 + 3x_2 + 3x_3 + 4x_4 + x_7 &= 420.
\end{aligned} \tag{8.3}$$

Diese drei Gleichungen sind äquivalent zu den ersten drei Gleichungen des Tableaus (8.2).

Was für das Tableau (8.2) gilt, gilt natürlich auch für die anderen Tableaus, die sonst noch auftreten, wenn man das Problem (8.1) mit dem Simplexverfahren löst. Wir halten dies noch einmal ausdrücklich fest.

Feststellung.
Löst man das Problem (8.1) mit dem Simplexverfahren, so sind die ersten drei Zeilen jedes auftretenden Tableaus äquivalent zu den drei Gleichungen in (8.3).

Wir wollen das Gleichungssystem (8.3) in Matrixform darstellen. Zu diesem Zweck sei

$$A = \begin{pmatrix} 3 & 2 & 1 & 2 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 1 & 0 \\ 4 & 3 & 3 & 4 & 0 & 0 & 1 \end{pmatrix}, \quad b = \begin{pmatrix} 225 \\ 117 \\ 420 \end{pmatrix}, \quad x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{pmatrix}.$$

Mit diesen Bezeichnungen geht (8.3) über in („Matrixdarstellung von (8.3)“):

$$Ax = b.$$

Äquivalent zu (8.3) sind, wie gesagt, die ersten drei Zeilen von (8.2). In (8.2) sind x_1, x_3 und x_7 die Basisvariablen; x_2, x_4, x_5 und x_6 sind die Nichtbasisvariablen.

Um den Unterschied zwischen den Basis- und den Nichtbasisvariablen zu betonen, schreiben wir Ax in der Form

$$A_B x_B + A_N x_N$$

mit

$$A_B = \begin{pmatrix} 3 & 1 & 0 \\ 1 & 1 & 0 \\ 4 & 3 & 1 \end{pmatrix} \quad \text{und} \quad x_B = \begin{pmatrix} x_1 \\ x_3 \\ x_7 \end{pmatrix}$$

sowie

$$A_N = \begin{pmatrix} 2 & 2 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 3 & 4 & 0 & 0 \end{pmatrix} \quad \text{und} \quad x_N = \begin{pmatrix} x_2 \\ x_4 \\ x_5 \\ x_6 \end{pmatrix}.$$

Das Gleichungssystem $Ax = b$ geht somit über in $A_B x_B + A_N x_N = b$; hieraus erhält man

$$A_B x_B = b - A_N x_N. \tag{8.4}$$

Bei (8.4) handelt es sich um nichts weiter als eine andere Art, die Gleichung $Ax = b$ zu schreiben. Wir verdeutlichen dies, indem wir beide Gleichungen in expliziter Form angeben: Die Gleichung $Ax = b$ ist, wenn man sie in expliziter Form hinschreibt, nichts anderes als (8.3); und wenn man (8.4) in expliziter Form hinschreibt, so erhält man:

$$\begin{aligned} 3x_1 + x_3 &= 225 - 2x_2 - 2x_4 - x_5 \\ x_1 + x_3 &= 117 - x_2 - x_4 - x_6 \\ 4x_1 + 3x_3 + x_7 &= 420 - 3x_2 - 4x_4 \end{aligned}$$

Man erkennt, was beim Übergang von $Ax = b$ zu $A_B x_B = b - A_N x_N$ passiert ist: (8.3) wurde so umgeschrieben, dass die Nichtbasisvariablen alle nach rechts kamen, während die Basisvariablen links blieben.

Nun soll die Gleichung $A_B x_B = b - A_N x_N$ weiter umgeformt werden, nämlich so, dass links nur noch x_B steht. Da trifft es sich gut, dass A_B eine nichtsinguläre quadratische Matrix ist[1], und somit die inverse Matrix A_B^{-1} existiert. Nimmt man (8.4) auf beiden Seiten von links mit A_B^{-1} mal, so erhält man

$$x_B = A_B^{-1} b - A_B^{-1} A_N x_N. \tag{8.5}$$

[1] Vgl. auch (8.7) sowie den Anhang zu diesem Abschnitt (Seite 105).

Die Gleichung (8.5) ist nichts weiter als eine kompakte Art, die ersten drei Zeilen von (8.2) mittels Matrizen auszudrücken. Nun soll auch noch die letzte Zeile von (8.2) in eine entsprechende Form gebracht werden.

Ähnlich, wie wir zuvor von $Ax = b$ ausgegangen sind, gehen wir nun von der Gleichung

$$z = c^T x$$

aus; hierbei ist $c = \big(19,\ 13,\ 12,\ 17,\ 0,\ 0,\ 0\big)^T$ und x hat dieselbe Bedeutung wie oben. Ähnlich wie zuvor definieren wir $c_B = \big(19,\ 12,\ 0\big)^T$ und $c_N = \big(13,\ 17,\ 0, 0\big)^T$. Wir erhalten

$$z = c_B^T x_B + c_N^T x_N.$$

Setzt man hierin gemäß (8.5) ein, so ergibt sich

$$\begin{aligned} z &= c_B^T\Big(A_B^{-1}b - A_B^{-1}A_N x_N\Big) + c_N^T x_N \\ &= c_B^T A_B^{-1} b + \Big(c_N^T - c_B^T A_B^{-1} A_N\Big) x_N. \end{aligned}$$

Zusammenfassend können wir also feststellen, dass (8.2) in Matrixform wie folgt angegeben werden kann:

$$\begin{array}{rcl} x_B &=& A_B^{-1}b - A_B^{-1}A_N x_N \\ \hline z &=& c_B^T A_B^{-1} b + \Big(c_N^T - c_B^T A_B^{-1} A_N\Big) x_N. \end{array} \tag{8.6}$$

Nun betrachten wir den allgemeinen Fall. Gegeben sei ein beliebiges LP-Problem in Standardform:

$$\begin{array}{l} \text{maximiere } \displaystyle\sum_{j=1}^{n} c_j x_j \\ \text{unter den Nebenbedingungen} \\ \qquad \displaystyle\sum_{j=1}^{n} a_{ij} x_j \ \le\ b_i \qquad (i = 1, \ldots, m) \\ \qquad\qquad x_j \ \ge\ 0 \qquad (j = 1, \ldots, n). \end{array} \tag{$\star$}$$

Nach der Einführung von Schlupfvariablen lässt sich dies wie folgt schreiben:

$$\begin{array}{l} \text{maximiere } c^T x \\ \text{unter den Nebenbedingungen} \\ \qquad Ax \ = b \\ \qquad\ x \ \ge 0. \end{array}$$

Man beachte: Es wurden *Schlupfvariablen* eingeführt und $Ax = b$ ist eine *Gleichung*. Dementsprechend ist A eine Matrix mit m Zeilen und $n + m$ Spalten; die letzten m Spalten von A bilden eine Einheitsmatrix.

Außerdem gilt: x ist ein Spaltenvektor der Länge $n + m$; b ist ein Spaltenvektor der Länge m; c^T ist ein Zeilenvektor der Länge $n + m$, für den die letzten m Einträge gleich Null sind.

Explizit lassen sich diese Matrizen und Vektoren wie folgt angeben:

$$A = \begin{pmatrix} a_{11} & \ldots & a_{1n} & 1 & \ldots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{m1} & \ldots & a_{mn} & 0 & \ldots & 1 \end{pmatrix}, \quad x = \begin{pmatrix} x_1 \\ \vdots \\ x_{n+m} \end{pmatrix}, \quad b = \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix}, \quad c^T = (c_1, \ldots, c_n, 0, \ldots, 0).$$

Es liege nun ein zulässiges Tableau zum Problem ($\star$) vor. Wie wir wissen, gehört zu diesem Tableau eine zulässige Basislösung, die wir mit $x^* = \left(x_1^*, \ldots, x_{n+m}^*\right)^T$ bezeichnen wollen.

Außerdem liefert das Tableau eine Zerlegung der Variablen $x_1, \ldots, x_{n+m}$ in Basisvariablen und Nichtbasisvariablen. Genauer: Es gibt m Basisvariablen und n Nichtbasisvariablen.

Wie zuvor in unserem Beispiel führt die Aufteilung in Basis- und Nichtbasisvariablen zu Matrizen A_B und A_N, zu Vektoren x_B und x_N sowie zu c_B^T und c_N^T. Wir behaupten nun:

$$\textit{Die Matrix } A_B \textit{ ist nichtsingulär.} \tag{8.7}$$

Wir zeigen die Richtigkeit dieser Behauptung, indem wir nachweisen, dass das Gleichungssystem

$$A_B x_B = b$$

eindeutig lösbar ist (vgl. Seite 105 f.).

Dass dieses Gleichungssystem eine Lösung besitzt, ist schnell gezeigt: Für die zulässige Basislösung x^* wählen wir die Bezeichnungen x_B^* und x_N^* analog zu x_B und x_N. Für x^* gilt $Ax^* = b$ und $x_N^* = 0$; außerdem gilt $Ax^* = A_B x_B^* + A_N x_N^*$. Also haben wir

$$A_B x_B^* = Ax^* - A_N x_N^* = Ax^* = b,$$

d.h., x_B^* ist eine Lösung von $A_B x_B = b$.

Es bleibt zu zeigen, dass x_B^* die einzige Lösung von $A_B x_B = b$ ist. Um dies nachzuweisen, betrachten wir einen beliebigen Vektor $\widetilde{x}_B$, für den $A_B \widetilde{x}_B = b$ gilt. Wir bilden zu $\widetilde{x}_B$ einen Vektor $\widetilde{x}$ der Länge $n+m$, indem wir die Stellen von $\widetilde{x}$, die den n Nichtbasisvariablen entsprechen, gleich Null setzen, und ansonsten die Einträge gemäß $\widetilde{x}_B$ wählen[2]. Dann gilt also insbesondere $\widetilde{x}_N = 0$, woraus

$$A\widetilde{x} = A_B \widetilde{x}_B + A_N \widetilde{x}_N = A_B \widetilde{x}_B = b$$

folgt. Wir ziehen nun das Tableau heran, von dem wir oben ausgegangen waren, d.h. das Tableau, zu dem die zulässige Basislösung x^* gehört. Da $A\widetilde{x} = b$ gilt, erfüllt $\widetilde{x}$ ebenfalls die ersten m Gleichungen dieses Tableaus, woraus wegen $\widetilde{x}_N = 0$ die Behauptung $\widetilde{x}_B = x_B^*$ folgt. Damit ist (8.7) bewiesen. □

Die Feststellung (8.7) bedeutet unter anderem, dass A_B invertierbar ist (vgl. Abschnitt 8.5). Mit anderen Worten: A_B^{-1} existiert.

Mit A_B^{-1} können wir nun wie im Beispiel rechnen und gelangen auch diesmal zur Matrixdarstellung (8.6) eines Tableaus.

Man nennt die Matrix A_B die *Basismatrix*. (Häufig wird die Matrix A_B auch einfach nur die *Basis* genannt; da wir die Menge der Basisvariablen ebenfalls „Basis“ genannt haben, muss bei Verwendung dieser Sprechweise darauf geachtet werden, dass keine Missverständnisse möglich sind.)

Eine weitere **Konvention**: *Es ist üblich, für die Basismatrix die Bezeichnung B anstelle von A_B zu verwenden.* Wir schließen uns dieser Konvention an und erhalten dementsprechend unsere *Matrixdarstellung eines Tableaus* in folgender Form:

$$\begin{aligned} x_B &= B^{-1}b - B^{-1}A_N x_N \\ \hline z &= c_B^T B^{-1} b + \left(c_N^T - c_B^T B^{-1} A_N\right) x_N. \end{aligned} \tag{8.8}$$

Bei (8.8) handelt es sich um eine allgemeine Art, ein Tableau darzustellen. *In dieser Darstellung wird in präziser Weise angegeben, wie die Koeffizienten eines Tableaus von den Eingangsdaten des Problems (also von A, b und c^T) abhängen.*

Zur Erinnerung: Auch B ist ein Teil der Eingangsdaten, denn B ist ja nur eine Abkürzung für die Matrix A_B, die eine Teilmatrix von A ist.

[2] *Genauer*: Die Stellen von $\widetilde{x}$, die den Basisvariablen entsprechen, sollen die Einträge von $\widetilde{x}_B$ sein (in derselben Reihenfolge); $\widetilde{x}_N$ sei der Nullvektor der Länge n.

Schaut man sich (8.8) an, so erkennt man alles wieder, was bislang immer in Tableaus vorkam:

- $B^{-1}b$ ist nichts anderes als der Vektor x_B^*, der die aktuellen Werte der Basisvariablen angibt.
- $c_B^T B^{-1} b$ ist eine allgemeine Formel für den aktuellen Wert der Zielfunktion.
- $c_N^T - c_B^T B^{-1} A_N$ ist eine allgemeine Formel für den Vektor der Koeffizienten, die in der z-Zeile bei den Nichtbasisvariablen auftreten.
- $-B^{-1}A_N$ ist die Matrix der Koeffizienten, die oberhalb der z-Zeile bei den Nichtbasisvariablen anzutreffen sind.

Am konkreten Fall des Tableaus (8.2) erläutert bedeuten diese Feststellungen, dass die folgenden Gleichungen gelten:

$$\begin{aligned} B^{-1}b &= \begin{pmatrix} 54 \\ 63 \\ 15 \end{pmatrix} \\ c_B^T B^{-1} b &= 1782 \\ c_N^T - c_B^T B^{-1} A_N &= \begin{pmatrix} -2.5, & 1.5, & -3.5, & -8.5 \end{pmatrix} \\ -B^{-1}A_N &= \begin{pmatrix} -0.5 & -0.5 & -0.5 & 0.5 \\ -0.5 & -0.5 & 0.5 & -1.5 \\ 0.5 & -0.5 & 0.5 & 2.5 \end{pmatrix}. \end{aligned}$$

Wir kommen nun zur *Beschreibung des revidierten Simplexverfahrens*, das an die Darstellung (8.8) („Darstellung eines Tableaus mittels Matrizen“) anknüpft. Das Simplexverfahren in seiner bisherigen Form wollen wir auch *Standardsimplexverfahren* nennen.

8.2 Beschreibung des revidierten Simplexverfahrens anhand eines Beispiels

In jeder Iteration des Standardsimplexverfahrens wählt man zunächst eine Eingangsvariable, bestimmt danach eine Ausgangsvariable und nimmt anschließend ein Update des aktuellen Tableaus vor, wodurch man vor allem auch eine neue zulässige Basislösung erhält. Genaue Beobachtung, wie all dies im Standardsimplexverfahren geschieht, wird uns zum revidierten Simplexverfahren führen.

Wir steigen mit unseren Überlegungen an der Stelle ein, an der im Standardverfahren das Tableau (8.2) vorliegt. Im Standardverfahren führt man anhand dieses Tableaus die nächste Iteration aus. Im revidierten Simplexverfahren ist dies so nicht möglich, denn wir haben das Tableau in diesem Verfahren gar nicht vorliegen. Stattdessen liegen x_B^* und B vor:

$$x_B^* = \begin{pmatrix} x_1^* \\ x_3^* \\ x_7^* \end{pmatrix} = \begin{pmatrix} 54 \\ 63 \\ 15 \end{pmatrix} \quad \text{und} \quad B = \begin{pmatrix} 3 & 1 & 0 \\ 1 & 1 & 0 \\ 4 & 3 & 1 \end{pmatrix}.$$

Als Eingangsvariable kommt jede Nichtbasisvariable infrage, deren Koeffizient in der letzten Zeile des Tableaus positiv ist. Wir wissen (vgl. (8.8)), dass

$$c_N^T - c_B^T B^{-1} A_N$$

den Vektor der Koeffizienten der letzten Zeile angibt. *Benutzt man das Standardsimplexverfahren, so hat man diesen Vektor unmittelbar zur Verfügung*: Man kann ihn ja an der letzten Zeile des aktuellen Tableaus ablesen.

Benutzt man dagegen die revidierte Simplexmethode, so muss man sich diesen Vektor erst ausrechnen; dies geschieht in zwei Schritten: Zunächst berechnet man den Zeilenvektor

$$y^T = c_B^T B^{-1},$$

indem man das Gleichungssystem

$$y^T B = c_B^T$$

löst; im 2. Schritt berechnet man anschließend $c_N^T - y^T A_N$.

In unserem Beispiel sieht das so aus: Zunächst löst man das Gleichungssystem

$$\begin{pmatrix} y_1, & y_2, & y_3 \end{pmatrix} \begin{pmatrix} 3 & 1 & 0 \\ 1 & 1 & 0 \\ 4 & 3 & 1 \end{pmatrix} = \begin{pmatrix} 19, & 12, & 0 \end{pmatrix}.$$

In expliziter Form lautet dieses Gleichungssystem

$$\begin{aligned} 3y_1 + y_2 + 4y_3 &= 19 \\ y_1 + y_2 + 3y_3 &= 12 \\ y_3 &= 0. \end{aligned}$$

Als Lösung erhält man

$$y^T = \begin{pmatrix} y_1, & y_2, & y_3 \end{pmatrix} = \begin{pmatrix} 3.5, & 8.5, & 0 \end{pmatrix}.$$

Danach berechnet man $c_N^T - y^T A_N$; man erhält

$$\begin{aligned} c_N^T - y^T A_N &= \begin{pmatrix} 13, & 17, & 0, & 0 \end{pmatrix} - \begin{pmatrix} 3.5, & 8.5, & 0 \end{pmatrix} \begin{pmatrix} 2 & 2 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 3 & 4 & 0 & 0 \end{pmatrix} \\ &= \begin{pmatrix} -2.5, & 1.5, & -3.5, & -8.5 \end{pmatrix}. \end{aligned}$$

Man vergleiche dies mit den Koeffizienten in der letzten Zeile von (8.2). (Noch einmal der Deutlichkeit halber: *Im Standardverfahren hat man diesen Vektor bereits zu Beginn der Iteration vorliegen, im revidierten Verfahren muss man ihn erst berechnen.*)

Im Vektor $(-2.5,\ 1.5,\ -3.5,\ -8.5)$ ist nur die zweite Komponente positiv; also ist die zweite Komponente des Vektors $x_N^T = (x_2,\ x_4,\ x_5,\ x_6)$ die Eingangsvariable, d.h., x_4 *ist die Eingangsvariable.*

In unserem Beispiel kam nur x_4 als Eingangsvariable infrage. Allgemein gilt: Die Nichtbasisvariable x_j kommt als Eingangsvariable infrage, falls für die entsprechende Komponente c_j von c_N^T und für die entsprechende Spalte a von A_N gilt: $c_j - y^T a > 0$ bzw. (was dasselbe ist) $y^T a < c_j$. Wird x_j als Eingangsvariable gewählt, so bezeichnet man die Spalte a von A_N, die x_j entspricht, als *Eingangsspalte*.

Erläuterung an unserem Beispiel: x_4 ist die Eingangsvariable und

$$a = \begin{pmatrix} 2 \\ 1 \\ 4 \end{pmatrix}$$

ist die Eingangsspalte.

Um die *Ausgangsvariable* zu bestimmen, hebt man – wie wir wissen – den Wert der Eingangsvariablen von Null auf einen Wert t an, wobei die übrigen Nichtbasisvariablen den Wert Null behalten und sich die Werte der Basisvariablen entsprechend ändern. Solange alle Basisvariablen positiv bleiben, wird t weiter angehoben, bis es zum ersten Mal vorkommt, dass eine oder mehrere Basisvariablen auf Null absinken. Unter diesen Variablen wird dann eine ausgewählt, die die Basis verlässt.

Wird die *Standardsimplexmethode* benutzt, so kann man den Höchstwert von t und eine dazugehörige Ausgangsvariable leicht bestimmen – man hat ja das Tableau zur Verfügung, aus dem man die erforderlichen Informationen ablesen kann; in unserem Beispiel gilt

$$\begin{aligned} x_1 &= 54 \ \ldots - 0.5x_4 \ \ldots \\ x_3 &= 63 \ \ldots - 0.5x_4 \ \ldots \\ x_7 &= 15 \ \ldots - 0.5x_4 \ \ldots \end{aligned}$$

und somit (für $x_4 = t$ und $x_2 = x_5 = x_6 = 0$)

$$\begin{aligned} x_1 &= 54 - 0.5t \\ x_3 &= 63 - 0.5t \\ x_7 &= 15 - 0.5t. \end{aligned} \tag{8.9}$$

Aus (8.9) ergibt sich, dass t bis auf 30 angehoben werden kann und dass x_7 die Ausgangsvariable ist.

Soweit die Vorgehensweise bei der Standardsimplexmethode. *Wie verfährt man nun aber in der revidierten Simplexmethode, wenn man das Tableau (8.2) gar nicht zur Verfügung hat?*

Was hat man stattdessen zur Verfügung?

Antwort: x_B^* und B sowie die zuvor bestimmte Eingangsspalte a und die dazugehörige Eingangsvariable (im Beispiel: x_4); und außerdem natürlich A, b und c^T.

Wir wissen aus (8.8), dass sich die ersten m Zeilen eines Tableaus wie folgt darstellen lassen[3]:

$$x_B = x_B^* - B^{-1}A_N x_N.$$

Hebt man die Eingangsvariable von Null auf t (und lässt alle anderen Nichtbasisvariablen bei Null), so ändert sich x_B von x_B^* zu $x_B^* - td$, wobei d die Spalte von $B^{-1}A_N$ bezeichnet, die der Eingangsvariablen entspricht: Dies ist die Spalte

$$d = B^{-1}a,$$

wobei a wie zuvor die Eingangsspalte bezeichnet[4]. Benutzt man die revidierte Simplexmethode, so muss also $d = B^{-1}a$ berechnet werden. Dies geschieht dadurch, dass man das Gleichungssystem

$$Bd = a$$

löst. In unserem Beispiel sieht das wie folgt aus:

Schreibt man $d = (d_1,\ d_2,\ d_3)^T$, so lautet die Gleichung $Bd = a$:

$$\begin{pmatrix} 3 & 1 & 0 \\ 1 & 1 & 0 \\ 4 & 3 & 1 \end{pmatrix} \begin{pmatrix} d_1 \\ d_2 \\ d_3 \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \\ 4 \end{pmatrix}.$$

Löst man dieses lineare Gleichungssystem mit dem Gauß-Verfahren, so erhält man

$$d = \begin{pmatrix} d_1 \\ d_2 \\ d_3 \end{pmatrix} = \begin{pmatrix} 0.5 \\ 0.5 \\ 0.5 \end{pmatrix}.$$

Dies ist genau das, was sich bei Verwendung des Standardverfahrens direkt am Tableau ablesen ließ (vgl. (8.2) bzw. (8.9)).

Da wir d nun kennen, ist es einfach festzustellen, dass t bis auf 30 angehoben werden kann. Für $t = 30$ ergibt sich:

$$\begin{aligned} 54 - 0.5t &= 39 \\ 63 - 0.5t &= 48 \\ 15 - 0.5t &= 0. \end{aligned}$$

Es hat sich insbesondere ergeben, dass x_7 die Ausgangsvariable ist.

[3] Man beachte, dass $B^{-1}b = x_B^*$ gilt.

[4] Man beachte: Bei $-d$ handelt es sich um die *Pivotspalte*.

Bislang war es so, dass wir bei Verwendung der revidierten Simplexmethode Rechnungen durchführen mussten, die bei der Standardmethode nicht nötig waren. **Dies kehrt sich nun, am Ende der Iteration, um**: Während beim Standardverfahren nun noch das recht mühevolle Update des Tableaus zu leisten ist, sind beim revidierten Verfahren keine derartigen Rechnungen nötig. In unserem Beispiel beginnt man die nächste Iteration einfach mit

$$x_B^* = \begin{pmatrix} x_1^* \\ x_3^* \\ x_4^* \end{pmatrix} = \begin{pmatrix} 39 \\ 48 \\ 30 \end{pmatrix} \quad \text{und} \quad B = \begin{pmatrix} 3 & 1 & 2 \\ 1 & 1 & 1 \\ 4 & 3 & 4 \end{pmatrix}.$$

Die Reihenfolge der Spalten in der Matrix B ist übrigens unerheblich: *Das Einzige, was gewährleistet sein muss, ist, dass diese Reihenfolge zu der Reihenfolge der Variablen in x_B^* passt.* Die nächste Iteration könnte ebenso gut beginnen mit

$$x_B^* = \begin{pmatrix} x_1^* \\ x_4^* \\ x_3^* \end{pmatrix} = \begin{pmatrix} 39 \\ 30 \\ 48 \end{pmatrix} \quad \text{und} \quad B = \begin{pmatrix} 3 & 2 & 1 \\ 1 & 1 & 1 \\ 4 & 4 & 3 \end{pmatrix}.$$

Eine gute *Regel*, an die man sich auch in den Übungsaufgaben unbedingt immer halten sollte, wird im Folgenden beschrieben.

Regel zum Update von x_B^* und B bei der Durchführung des revidierten Simplexverfahrens.
Bei der Durchführung des revidierten Simplexverfahrens füge man beim *Update von x_B^** die Eingangsvariable an genau der Stelle ein, an der zuvor die Ausgangsvariable stand.
Beim *Update von B* ist analog vorzugehen: Man füge die Eingangsspalte genau dort ein, wo zuvor die Spalte stand, die aus B ausscheidet (*Ausgangsspalte*).

Der Übersichtlichkeit halber fassen wir den Ablauf einer Iteration noch einmal zusammen.

8.3 Eine Iteration im revidierten Simplexverfahren

1. Schritt:

Man löse das Gleichungssystem $y^T B = c_B^T$.

2. Schritt:

Wahl der Eingangsspalte; es kommt jede Spalte a von A_N infrage, für die $y^T a$ kleiner ist als die entsprechende Komponente von c_N^T. Falls es keine solche Spalte gibt, ist die aktuelle Lösung optimal.

3. Schritt:

Man löse das Gleichungssystem $Bd = a$.

4. Schritt:

Man finde das größte $t \geq 0$, für das $x_B^* - td \geq 0$ gilt. Falls es kein solches t gibt, ist das Problem unbeschränkt; andernfalls ist mindestens eine Komponente von $x_B^* - td$ gleich Null und eine zugehörige Variable verlässt die Basis.

5. Schritt:

Man setze den Wert der Eingangsvariablen auf t (für das größtmögliche t wie im 4. Schritt) und ersetze in x_B^* die Werte der übrigen Basisvariablen durch $x_B^* - td$. Außerdem ersetze man in B die Ausgangsspalte durch die Eingangsspalte. Beim Update von x_B^* und B beachte man die zuvor beschriebene Regel.

Beispiel. Wir greifen unser erstes Beispiel zur Standardsimplexmethode aus Kapitel 2 wieder auf und wollen es nun mit dem revidierten Simplexverfahren lösen. Dieses Beispiel lautet wie folgt:

$$
\begin{aligned}
&\text{maximiere } 5x_1 + 4x_2 + 3x_3 \\
&\text{unter den Nebenbedingungen} \\
&\qquad 2x_1 + 3x_2 + \ x_3 \ \leq \ 5 \\
&\qquad 4x_1 + \ x_2 + 2x_3 \leq 11 \\
&\qquad 3x_1 + 4x_2 + 2x_3 \ \leq \ 8 \\
&\qquad\qquad x_1, x_2, x_3 \ \geq \ 0.
\end{aligned}
$$

Die *Eingangsdaten* („original data“) für unser Beispiel sind gegeben durch[5]

$$
A = \begin{array}{c} \begin{matrix} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 \end{matrix} \\ \begin{pmatrix} 2 & 3 & 1 & 1 & 0 & 0 \\ 4 & 1 & 2 & 0 & 1 & 0 \\ 3 & 4 & 2 & 0 & 0 & 1 \end{pmatrix} \end{array}, \quad b = \begin{pmatrix} 5 \\ 11 \\ 8 \end{pmatrix} \quad \text{und} \quad c^T = \begin{array}{c} \begin{matrix} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 \end{matrix} \\ \begin{pmatrix} 5 & 4 & 3 & 0 & 0 & 0 \end{pmatrix} \end{array}.
$$

Das Verfahren startet mit

$$
x_B^* = \begin{pmatrix} x_4^* \\ x_5^* \\ x_6^* \end{pmatrix} = \begin{pmatrix} 5 \\ 11 \\ 8 \end{pmatrix} \quad \text{und} \quad B = \begin{array}{c} \begin{matrix} x_4 & x_5 & x_6 \end{matrix} \\ \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \end{array}.
$$

Auch bei B wurde eine *Kopfzeile* hinzugefügt, die angibt, wie sich Basisvariablen und Spalten entsprechen. Dies dient der Übersichtlichkeit. *Bei der Basismatrix B ist das Hinzufügen der Kopfzeile besonders wichtig*, da die Spalten von B nicht immer in derselben Reihenfolge wie in A vorkommen. Auch bei der Matrix A_N fügen wir im Folgenden eine Kopfzeile hinzu.

1. Iteration

1. Schritt:

Das Gleichungssystem $y^T B = c_B^T$ lautet

$$
\begin{aligned}
y_1 + 0y_2 + 0y_3 &= 0 \\
0y_1 + \ y_2 + 0y_3 &= 0 \\
0y_1 + 0y_2 + \ y_3 &= 0.
\end{aligned}
$$

Lösung: $y^T = (y_1, y_2, y_3) = (0, 0, 0)$.

2. Schritt:

Es gilt $A_N = \begin{array}{c} \begin{matrix} x_1 & x_2 & x_3 \end{matrix} \\ \begin{pmatrix} 2 & 3 & 1 \\ 4 & 1 & 2 \\ 3 & 4 & 2 \end{pmatrix} \end{array}$; für alle Spalten a von A_N gilt $y^T a = 0$ und die Komponenten von c_N^T lauten 5, 4 und 3. Es kommen also alle Spalten von A_N als Eingangsspalte a infrage; wir wählen $a = \begin{pmatrix} 2 \\ 4 \\ 3 \end{pmatrix}$. Dies entspricht der Regel des größten Koeffizienten. Mit der Wahl der Eingangsspalte a liegt auch die Eingangsvariable fest: In unserem Fall ist dies x_1.

[5] Es ist zweckmäßig und sehr zu empfehlen, sowohl bei A als auch bei c^T eine *Kopfzeile* der Form $x_1, \ldots, x_n$ hinzuzufügen. Schreibt man A und c^T genau untereinander, so genügt es, die Kopfzeile nur bei A anzugeben.

3. Schritt:

Das Gleichungssystem $Bd = a$ lautet

$$\begin{aligned} d_1 + 0d_2 + 0d_3 &= 2 \\ 0d_1 + d_2 + 0d_3 &= 4 \\ 0d_1 + 0d_2 + d_3 &= 3. \end{aligned}$$

Lösung: $d = \begin{pmatrix} 2 \\ 4 \\ 3 \end{pmatrix}$.

4. Schritt:

Es ist das größte $t \geq 0$ zu finden, für das gilt:

$$\begin{pmatrix} 5 \\ 11 \\ 8 \end{pmatrix} - t \begin{pmatrix} 2 \\ 4 \\ 3 \end{pmatrix} \geq \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}.$$

Für t soll demnach

$$\begin{aligned} 5 - 2t &\geq 0 \\ 11 - 4t &\geq 0 \\ 8 - 3t &\geq 0 \end{aligned}$$

gelten. Das größte t, das dies erfüllt, ist $t = 2.5$. Für diese Wahl von t gilt

$$x_B^* - td = \begin{pmatrix} 5 & - & 2t \\ 11 & - & 4t \\ 8 & - & 3t \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0.5 \end{pmatrix}.$$

Wegen $x_B^* = \begin{pmatrix} x_4^* \\ x_5^* \\ x_6^* \end{pmatrix}$ erhält man also x_4 als Ausgangsvariable.

5. Schritt (Update von x_B^* und B):

Man erhält

$$x_B^* = \begin{pmatrix} x_1^* \\ x_5^* \\ x_6^* \end{pmatrix} = \begin{pmatrix} 2.5 \\ 1 \\ 0.5 \end{pmatrix} \quad \text{und} \quad B = \begin{array}{c} \begin{matrix} x_1 & x_5 & x_6 \end{matrix} \\ \begin{pmatrix} 2 & 0 & 0 \\ 4 & 1 & 0 \\ 3 & 0 & 1 \end{pmatrix} \end{array}.$$

Mit dem neuen Vektor x_B^* und der neuen Basismatrix geht es nun in die nächste Runde.

2. Iteration

1. Schritt:

Das Gleichungssystem $y^T B = c_B^T$ lautet

$$\begin{aligned} 2y_1 + 4y_2 + 3y_3 &= 5 \\ y_2 &= 0 \\ y_3 &= 0. \end{aligned}$$

Lösung: $y^T = (y_1, y_2, y_3) = (2.5, 0, 0)$.

2. Schritt:

Es gilt $A_N = \begin{array}{c} \begin{matrix} x_2 & x_3 & x_4 \end{matrix} \\ \begin{pmatrix} 3 & 1 & 1 \\ 1 & 2 & 0 \\ 4 & 2 & 0 \end{pmatrix} \end{array}$. Für die Spalten a von A_N erhält man der Reihe nach als Wert von $y^T a$:

$$7.5, \quad 2.5 \quad \text{und} \quad 2.5.$$

Die entsprechenden Einträge von c_N^T lauten

$$4, \quad 3 \quad \text{und} \quad 0.$$

Folglich ist

$$a = \begin{pmatrix} 1 \\ 2 \\ 2 \end{pmatrix}$$

die Eingangsspalte und x_3 ist die Eingangsvariable.

3. Schritt:

Das zu lösende Gleichungssystem lautet

$$\begin{aligned} 2d_1 \qquad\quad &= 1 \\ 4d_1 + d_2 \qquad &= 2 \\ 3d_1 \qquad + d_3 \; &= 2. \end{aligned}$$

Lösung: $d = \begin{pmatrix} 0.5 \\ 0 \\ 0.5 \end{pmatrix}$.

4. Schritt:

Es ist das größte $t \geq 0$ zu finden, so dass gilt:

$$\begin{pmatrix} 2.5 \\ 1 \\ 0.5 \end{pmatrix} - t \begin{pmatrix} 0.5 \\ 0 \\ 0.5 \end{pmatrix} \geq \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}.$$

Es folgt $t = 1$. Für $t = 1$ gilt: $x_B^* - td = \begin{pmatrix} 2 \\ 1 \\ 0 \end{pmatrix}$, woraus wegen $x_B^* = \begin{pmatrix} x_1^* \\ x_5^* \\ x_6^* \end{pmatrix}$ folgt, dass x_6 die Ausgangsvariable ist.

5. Schritt:

Als Update von x_B^* und B erhält man

$$x_B^* = \begin{pmatrix} x_1^* \\ x_5^* \\ x_3^* \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \\ 1 \end{pmatrix} \quad \text{und} \quad B = \begin{array}{c} \begin{matrix} x_1 & x_5 & x_3 \end{matrix} \\ \begin{pmatrix} 2 & 0 & 1 \\ 4 & 1 & 2 \\ 3 & 0 & 2 \end{pmatrix} \end{array}.$$

Mit dem neuen Vektor x_B^* und der neuen Basismatrix geht es nun in die nächste Runde.

3. Iteration

1. Schritt:

Das zu lösende Gleichungssystem lautet[6]

$$\begin{aligned} 2y_1 + 4y_2 + 3y_3 &= 5 \\ y_2 &= 0 \\ y_1 + 2y_2 + 2y_3 &= 3. \end{aligned}$$

Lösung: $y^T = (y_1, y_2, y_3) = (1, 0, 1)$.

2. Schritt:

Es gilt $A_N = \begin{array}{c} \begin{matrix} x_2 & x_4 & x_6 \end{matrix} \\ \begin{pmatrix} 3 & 1 & 0 \\ 1 & 0 & 0 \\ 4 & 0 & 1 \end{pmatrix} \end{array}$. Für die Spalten a von A_N ergeben sich die folgenden Werte von $y^T a$: 7, 1, 1. Als dazugehörige Einträge von c_N^T haben wir 4, 0, 0. Es folgt, dass die aktuelle Lösung optimal ist. Als optimale Basislösung hat sich ergeben:

$$x_1^* = 2, \quad x_2^* = 0, \quad x_3^* = 1, \quad x_4^* = 0, \quad x_5^* = 1, \quad x_6^* = 0.$$

Als optimalen Zielfunktionswert erhält man $z = 13$.

8.4 Vergleich der Standardsimplexmethode mit dem revidierten Verfahren

Für beide Varianten gilt: *In jeder Iteration wird eine zulässige Basislösung durch eine in der Regel andere zulässige Basislösung ersetzt.* Oder, in geometrischer Terminologie ausgedrückt:

Man bewegt sich auf dem Rand des zulässigen Bereichs („längs einer Kante“) von einer Ecke zur nächsten.

Die *Details* sind jedoch sehr unterschiedlich:

- Im Standardverfahren arbeitet man mit *Tableaus* im Sinne von „dictionary“ (vgl. Chvátal) bzw. „Schlupfform“ (vgl. Cormen et al), in denen sich die aktuelle zulässige Basislösung unmittelbar ablesen lässt; *in jeder Iteration wird das neue Tableau direkt aus dem vorhergehenden Tableau berechnet.*
- Im Gegensatz dazu berechnet man in jeder Iteration des revidierten Simplexverfahrens die neue zulässige Basislösung *aus den Eingangsdaten* des Problems, wobei zwei lineare Gleichungssysteme zu lösen sind.

Beide Varianten haben *Vor- und Nachteile*: Diese werden im Buch von Chvátal ausführlich beschrieben und diskutiert; vgl. auch K. Neumann, M. Morlock: *Operations Research* (Hanser Verlag), wo ebenfalls die „pros and cons“ der beiden Varianten diskutiert werden. Hier soll nur ein Punkt etwas genauer besprochen werden.

In der *Praxis* hat man es meist mit *sehr großen Problemen* zu tun: Probleme mit mehr als 100000 Variablen und beispielsweise $m = 10000$ Gleichungen (Nebenbedingungen) sind keine Seltenheit. Häufig handelt es sich dabei um *Probleme, bei denen die meisten Koeffizienten a_{ij} gleich Null sind.* Man spricht dann von *schwach besetzten Problemen* und *schwach besetzten Matrizen*. Typische Größenordnungen:

[6] Man beachte: Zu lösen ist $y^T B = c_B^T$, wobei die Spalten von B in der Reihenfolge auftreten, die der Kopfzeile x_1, x_5, x_3 entspricht. *Diese Reihenfolge ist auch für die rechten Seiten des Gleichungssystems einzuhalten.* Dementsprechend gilt $c_B^T = (5, 0, 3)$.

$n = 100000$ Variablen, $m = 10000$ Gleichungen, aber niemals mehr als 10 Einträge in jeder Spalte der Matrix (a_{ij}), die ungleich Null sind.

Zur Behandlung von LP-Problemen mit schwach besetzten Matrizen gibt es ausgefeilte Methoden (vgl. etwa Chvátal, Kapitel 7 und 24). Man kann sagen: *Sind LP-Probleme sehr groß, so werden sie überhaupt nur dadurch für die Behandlung in einem Computer zugänglich, dass es sich um schwach besetzte Probleme handelt.* Beispielsweise müssen die vielen Nullen nicht gespeichert werden, sondern man speichert nur, an welchen Stellen Einträge ungleich Null stehen – und natürlich die Werte dieser Einträge.

Nach diesen Bemerkungen wird klar, weshalb in der Praxis, wenn es um sehr große, schwach besetzte Matrizen geht, *die revidierte Simplexmethode vorzuziehen ist*:

- Beim *Standardsimplexverfahren* kann aus dem Anfangstableau nach wenigen Iterationen ein „stark besetztes Tableau“ werden, d.h. ein Tableau, in dem viele Einträge ungleich Null sind. Diese Einträge müssen alle gespeichert werden, d.h., der Vorteil, den man am Anfang durch das Vorliegen eines schwach besetzten Problems hat, ist „futsch“.
- Dergleichen kann beim *revidierten Verfahren* nicht passieren, da man in jeder Iteration mit den Spalten der schwach besetzten Matrix A arbeitet und nicht mit einem völlig neuen Tableau, das in der Regel nicht mehr schwach besetzt ist.

Wir haben einen der Gründe angesprochen, weshalb praxistaugliche Softwarepakete auf dem revidierten Simplexverfahren basieren und nicht auf dem Standardverfahren. Hier noch ein anderer Grund: *Häufig ist die Anzahl n der Variablen wesentlich größer als m* (= Anzahl der Nebenbedingungen ohne die Nichtnegativitätsbedingungen). Daher kann es ein großer Vorteil sein, dass man im revidierten Verfahren im Wesentlichen mit der Basismatrix B arbeitet. (Man beachte: B ist eine $m \times m$ - Matrix, während in einem Tableau n Spalten vorkommen.)

Stichwort für einen weiteren Grund: Anfälligkeit für Rundungsfehler.

Für die Bearbeitung kleinerer Probleme per Hand sind beide Varianten – die Standardmethode und das revidierte Verfahren – sehr gut geeignet.

8.5 Anhang zu Abschnitt 8

In der linearen Algebra spielt der Unterschied zwischen invertierbaren und nicht invertierbaren $n \times n$ - Matrizen eine zentrale Rolle. Wir betrachten hier nur reelle $n \times n$ - Matrizen, d.h. quadratische Matrizen mit Einträgen aus $\mathbb{R}$, und sprechen von Matrizen der Klasse I und Matrizen der Klasse II. Eine andere (sehr verbreitete) Sprechweise: *Nichtsinguläre Matrizen* (Klasse I) und *singuläre Matrizen* (Klasse II). Statt „nichtsingulär“ wird häufig auch *regulär* gesagt. Hier noch einmal die wesentlichen Unterschiede:

Klasse I (invertierbare (n,n)-Matrizen A)	**Klasse II** (nicht invertierbare (n,n)-Matrizen A)
$\det A \neq 0$	$\det A = 0$
Die Spalten von A bilden eine Basis des $\mathbb{R}^n$.	Die Spalten von A bilden keine Basis des $\mathbb{R}^n$.
Die Spalten von A sind linear unabhängig.	Die Spalten von A sind linear abhängig.

Die Zeilen von A bilden eine Basis des $\mathbb{R}^n$.	Die Zeilen von A bilden keine Basis des $\mathbb{R}^n$.
Die Zeilen von A sind linear unabhängig.	Die Zeilen von A sind linear abhängig.
$Ax = b$ ist für jede rechte Seite b eindeutig lösbar.	Für jede rechte Seite b gilt: $Ax = b$ ist nicht eindeutig lösbar, d.h., $Ax = b$ ist entweder unlösbar oder es gibt mehr als eine Lösung.
Das homogene lineare Gleichungssystem $Ax = 0$ besitzt nur die triviale Lösung[7].	Das homogene lineare Gleichungssystem $Ax = 0$ besitzt neben der trivialen Lösung noch weitere (nichttriviale) Lösungen.
Mit dem Gauß-Jordan-Verfahren erhält man die reduzierte Zeilenstufenmatrix $Z = E_n$.	Das Gauß-Jordan-Verfahren führt zu einer reduzierten Zeilenstufenmatrix $Z \neq E_n$.
A^{-1} existiert.	A^{-1} existiert nicht.
Bei der linearen Abbildung $\mathbb{R}^n \to \mathbb{R}^n$, die durch $x \mapsto Ax$ gegeben ist, handelt es sich um eine *bijektive* lineare Abbildung („*Vektorraumisomorphismus*“).	Die lineare Abbildung $\mathbb{R}^n \to \mathbb{R}^n$, die durch $x \mapsto Ax$ gegeben ist, ist weder injektiv noch surjektiv.

[7] Mit 0 wird hier der Nullvektor der Länge n bezeichnet.

9 Maximale Flüsse und minimale Schnitte in Netzwerken

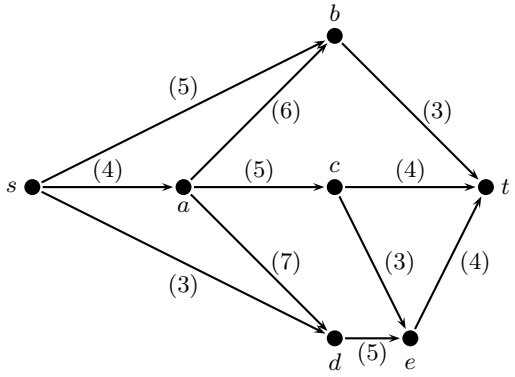
9.1 Flüsse in Netzwerken

Wir betrachten gerichtete Graphen, in denen jede Kante eine „Kapazität“ besitzt. Je nach Anwendungszusammenhang können diese Kapazitäten eine unterschiedliche Bedeutung haben. In jedem Fall geht es jedoch um obere Schranken: Beispielsweise besitzen Stromleitungen eine Maximalbelastung, durch eine Wasserleitung kann nur eine bestimmte Menge Wasser fließen oder auf einer Bahnstrecke kann nur eine gewisse Menge eines bestimmten Guts transportiert werden.

Zur Modellierung derartiger Situationen verwendet man sogenannte *Flussnetzwerke* (kurz: *Netzwerke*). Von grundlegender Bedeutung sind in diesem Zusammenhang *gerichtete Graphen* (kurz: *Digraphen*). Wir geben im Folgenden genaue Definitionen der erwähnten Begriffe.

Definition.
Es sei $G = (V, E)$ ein *schlingenloser Digraph*, d.h., V ist eine endliche Menge, deren Elemente *Knoten* genannt werden, und E ist eine Teilmenge von $V \times V$, die nur Paare (a, b) aus $V \times V$ enthält, für die $a \neq b$ gilt. Die Elemente von E heißen (gerichtete) *Kanten*[1].
Außerdem sei $c : E \to \mathbb{N} \cup \{0\}$ eine Abbildung, die jeder Kante $e \in E$ eine nichtnegative ganze Zahl $c(e)$ zuordnet, welche wir die *Kapazität* von e nennen.
Schließlich seien s und t zwei verschiedene Knoten von G, für die gilt: Zu s führen keine Kanten hin und von t führen keine Kanten weg.
Darüber hinaus wollen wir voraussetzen, dass es in G *keine isolierten Knoten* gibt, d.h., für jeden Knoten v soll es mindestens eine Kante geben, die zu v hin oder von v weg führt.
Unter diesen Voraussetzungen nennen wir $N = (G, c, s, t)$ *ein Flussnetzwerk mit Quelle s und Senke t* (kurz: *Netzwerk*).

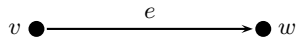
Beispiel.



Die eingeklammerten Zahlen bezeichnen die Kapazitäten der Kanten.

[1] Statt „Knoten“ sagt man im Deutschen auch *Ecke*, die englische Bezeichnung lautet *vertex* oder *node*; Kante heißt auf Englisch *edge*.

Ist $e = (v, w)$ eine Kante eines Netzwerks, so stellen wir uns e immer als einen Pfeil von v nach w vor:



Wir wollen uns nun für ein gegebenes Netzwerk zusätzlich vorstellen, dass in jeder Kante $e = (v, w)$ ein „Fluss“ von v nach w vorliegt: Beispielsweise kann man sich die Kanten als Wasserleitungen denken; oder man kann sich die Kanten $e = (v, w)$ als Straßen vorstellen, auf denen irgendeine Ware von v nach w transportiert wird. Durch $f(e) \in \mathbb{R}$ soll die Stärke des Flusses von v nach w modelliert werden.

Wir werden im Folgenden nur dann von einem Fluss auf dem Netzwerk N sprechen, wenn für alle Kanten $e \in E$ gilt:

$$0 \leq f(e) \leq c(e).$$

Mit anderen Worten: $f(e)$ soll immer nichtnegativ sein und $f(e)$ soll die Kapazität $c(e)$ niemals überschreiten. Außerdem soll eine *Erhaltungsregel* für alle Knoten $v \neq s, t$ gelten: *Es soll aus v ebenso viel herausfließen, wie in v hineinfließt.*

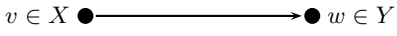
Bevor wir nun all dies in einer Definition zusammenfassen, führen wir einige Schreibweisen ein. Gegeben sei ein Netzwerk $N = (G, c, s, t)$ mit $G = (V, E)$. Sind X und Y Teilmengen von V, so bezeichnen wir mit

$$\big(X, Y\big)$$

die Menge aller Kanten $(v, w) \in E$, für die $v \in X$ und $w \in Y$ gilt. Dasselbe in Mengenschreibweise:

$$\big(X, Y\big) = \Big\{(v, w) \in E : v \in X \text{ und } w \in Y\Big\}.$$

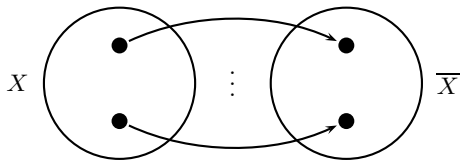
Mit anderen Worten: (X, Y) *bezeichnet die Menge aller Kanten von E, die von X nach Y zeigen.*



Für eine Teilmenge X von V bezeichnen wir mit $\overline{X}$ das *Komplement* von X in V, d.h.,

$$\overline{X} = V \setminus X.$$

$\overline{X}$ ist also die Menge der Knoten aus V, die nicht in X liegen, und $(X, \overline{X})$ ist die Menge der Kanten, die von X nach $\overline{X}$ zeigen.



Es sei nun $f : E \to \mathbb{R}$ eine Funktion; f ordnet also jeder Kante $e \in E$ eine reelle Zahl $f(e)$ zu. Dann definieren wir

$$f(X, \overline{X}) := \sum_{e \in (X, \overline{X})} f(e).$$

Um $f(X, \overline{X})$ zu berechnen, sind also alle Kanten $e \in E$ zu betrachten, die von X nach $\overline{X}$ zeigen; für diese Kanten sind die Werte $f(e)$ aufzusummieren.

Als Abkürzung schreiben wir auch

$$f^+(X) := f(X, \overline{X})$$

sowie

$$f^-(X) := f(\overline{X}, X).$$

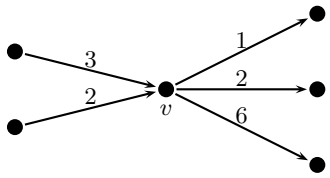
Gilt $X = \{v\}$, d.h., X enthält nur den Knoten v, so schreiben wir anstelle von $f^+(\{v\})$ auch einfach

$$f^+(v).$$

Folglich ist $f^+(v)$ die Summe der Werte $f(e)$ für alle Kanten e, die an v stoßen und von v weggerichtet sind.

Analog: Statt $f^-(\{v\})$ schreiben wir $f^-(v)$; der Wert von $f^-(v)$ ist die Summe der Werte $f(e)$ für alle Kanten e, die an v stoßen und zu v hingerichtet sind.

Beispiel. Es sollen, wie in der folgenden Zeichnung dargestellt, fünf Kanten an v stoßen.



Die Zahlen geben die Werte von f an. Dann gilt

$$f^-(v) = 3 + 2 = 5$$

und

$$f^+(v) = 1 + 2 + 6 = 9.$$

Nach diesem kleinen Einschub, in dem wir die Schreibweisen (X, Y), $(X, \overline{X})$, $f(X, \overline{X})$, $f^+(X)$, $f^-(X)$, $f^+(v)$ und $f^-(v)$ besprochen haben, kommen wir nun – wie weiter oben schon angekündigt – zur Definition des Begriffs eines Flusses auf einem Netzwerk.

Definition.
Gegeben sei ein Netzwerk $N = (G, c, s, t)$ mit $G = (V, E)$. Eine Abbildung

$$f : E \to \mathbb{R}$$

heißt ein *Fluss* auf N, wenn f den beiden folgenden Bedingungen genügt:

(F1) $0 \leq f(e) \leq c(e)$ für alle $e \in E$;

(F2) $f^-(v) = f^+(v)$ für alle Knoten $v \neq s, t$.

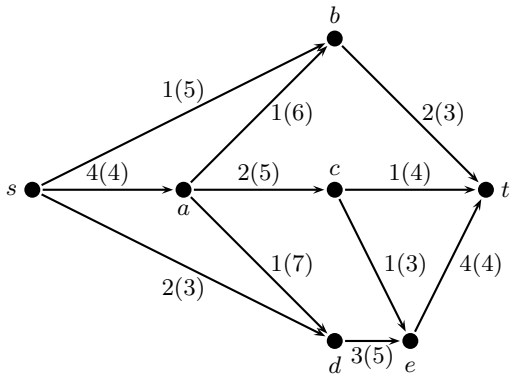
Die Bedingung (F2) bedeutet, dass für alle Knoten $v \neq s, t$ gilt[2]: *Aus v fließt ebenso viel hinaus, wie hineinfließt.*

Beispiel. Für das Netzwerk auf Seite 107 sei f gegeben durch:

(x, y)	$f(x, y)$
(s, a)	4
(s, b)	1
(s, d)	2
(a, b)	1
(a, c)	2
(a, d)	1
(c, e)	1
(d, e)	3
(b, t)	2
(c, t)	1
(e, t)	4

[2]Knoten $v \in V$ mit $v \neq s, t$ nennt man auch *innere Knoten* des Netzwerks N.

Trägt man diese Werte in die zugehörige Zeichnung ein, so erkennt man sofort, dass (F1) und (F2) erfüllt sind, dass also ein Fluss vorliegt:



Gilt $e = (v, w)$ für eine Kante $e \in E$, so nennen wir v den *Anfangsknoten* von e und w ist der *Endknoten* von e.

Da an den inneren Knoten $v \neq s, t$ die Erhaltungsregel (F2) gilt, ist es plausibel, dass aus s ebenso viel wegfließt, wie in t ankommt: Dies ist der Inhalt der folgende Feststellung.

Feststellung 1.

$$f^+(s) = f^-(t).$$

Beweis. Da es zu jeder Kante $e \in E$ genau einen Anfangsknoten gibt, haben wir

$$\sum_{e \in E} f(e) = \sum_{v \in V} f^+(v);$$

ganz entsprechend gilt auch

$$\sum_{e \in E} f(e) = \sum_{v \in V} f^-(v).$$

Es folgt

$$\begin{aligned} 0 &= \sum_{v \in V} f^+(v) - \sum_{v \in V} f^-(v) \\ &= \sum_{v \in V} \Big(f^+(v) - f^-(v) \Big) \\ &\overset{(F2)}{=} f^+(s) - f^-(s) + f^+(t) - f^-(t). \end{aligned}$$

Da zu s keine Kanten hinführen und von t keine Kanten wegführen, gilt $f^-(s) = f^+(t) = 0$. Also haben wir

$$0 = f^+(s) - f^-(t),$$

woraus die Behauptung folgt. □

Definition.

Den aufgrund von Feststellung 1 gemeinsamen Wert von $f^+(s)$ und $f^-(t)$ nennt man den *Wert des Flusses* f (Bezeichnung: $w(f)$). Man definiert also

$$w(f) := f^+(s) = f^-(t).$$

Der Wert $w(f)$ gibt also an, wie viel von der Quelle s wegfließt bzw. (was dasselbe ist), wie viel an der Senke t ankommt.

In unserem obigen Beispiel gilt

$$f^+(s) = 1 + 4 + 2 = 7$$

und

$$f^-(t) = 2 + 1 + 4 = 7.$$

Also gilt:

$$w(f) = 7.$$

Ein Fluss f^* auf einem Netzwerk N heißt *maximal*, falls $w(f^*) \geq w(f)$ für alle Flüsse f auf N gilt.

Eines der wichtigsten Probleme, um die es im Zusammenhang mit Flussnetzwerken geht, ist die Konstruktion eines maximalen Flusses[3].

9.2 Schnitte

Eine wichtige Rolle bei der Konstruktion eines maximalen Flusses spielen *obere Schranken*, die kein Fluss übertreffen kann. Ein einfaches Beispiel für eine solche obere Schranke erhält man, wenn man sich im Beispiel von Seite 107 die drei Kanten anschaut, die von s ausgehen: Die Summe ihrer Kapazitäten ist $5 + 4 + 3 = 12$; folglich kann es keinen Fluss mit einem Wert > 12 geben.

Oder, anders ausgedrückt: Für alle Flüsse f auf unserem Netzwerk gilt:

$$w(f) \leq 12.$$

Bei der Suche nach besonders guten oberen Schranken, die kein Fluss übertreffen kann, spielt der folgende Begriff eines Schnitts von $N = (G, c, s, t)$ eine besonders wichtige Rolle.

Definition.

Gegeben sei eine Zerlegung der Knotenmenge V von N in zwei disjunkte Mengen S und T derart, dass $s \in S$ und $t \in T$ gilt. (Mit anderen Worten: Es gelte $s \in S$, $t \notin S$ und $T = \overline{S}$.) Dann bezeichnen wir die Menge

$$(S, T)$$

als einen *Schnitt* von N.

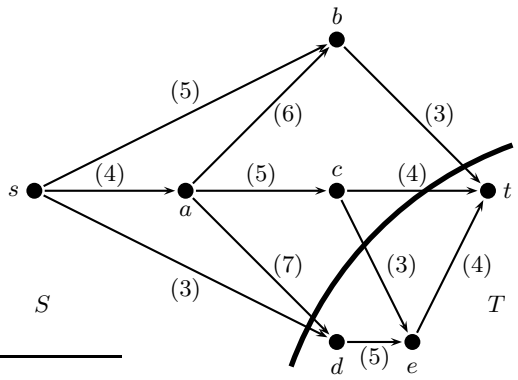
Ein Schnitt (S, T) ist also eine Menge von Kanten, nämlich die Menge aller Kanten $(u, v) \in E$, für die $u \in S$ und $v \in T$ gilt.

Wir illustrieren den Begriff eines Schnitts anhand unseres Beispiels (siehe Seite 107):

1. Es seien $S = \{s, a, b, c\}$ und $T = \{d, e, t\}$. Dann gilt

$$(S, T) = \Big\{(s, d),\ (a, d),\ (c, e),\ (c, t),\ (b, t)\Big\}.$$

Dasselbe als Zeichnung:

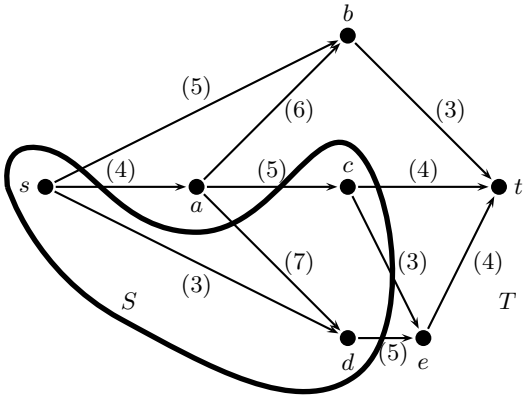


[3] Man sagt auch *Maximalfluss*.

2. Es seien $S = \{s, c, d\}$ und $T = \{a, b, e, t\}$. Dann gilt

$$(S,T) = \Big\{(s,a),\ (s,b),\ (c,e),\ (c,t),\ (d,e)\Big\}.$$

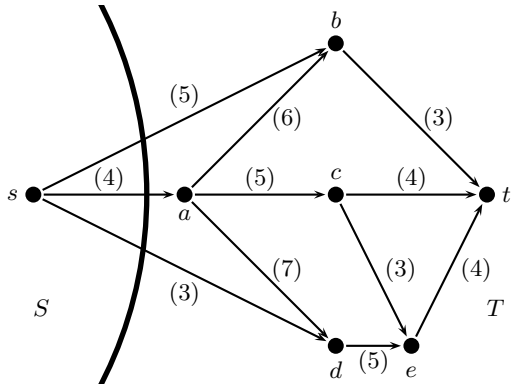
Dasselbe als Zeichnung:



3. Es seien $S = \{s\}$ und $T = \{a, b, c, d, e, t\}$. Dann gilt

$$(S,T) = \Big\{(s,a),\ (s,b),\ (s,d)\Big\}.$$

Dasselbe als Zeichnung:



Der Schnitt (S,T) aus dem letzten Beispiel ist derjenige, der zur Schranke

$$w(f) \leq 12$$

für alle Flüsse f auf N geführt hat.

In ähnlicher Weise führen alle Schnitte von N zu einer oberen Schranke von $w(f)$; dies soll im Folgenden präzisiert werden.

Zu diesem Zweck definieren wir, was die Kapazität eines Schnitts ist.

Definition.
Ist (S,T) ein Schnitt von N, so wird die Zahl

$$c(S,T) = \sum_{e \in (S,T)} c(e)$$

als *Kapazität* von (S,T) bezeichnet.

Die Kapazität eines Schnitts (S,T) ist somit die Summe der Kapazitäten aller Kanten, die von S ausgehen und nach T führen.

Beispiele. Es liege das Netzwerk von Seite 107 zugrunde und wir betrachten die obigen Beispiele für Schnitte (S,T).

1. $S=\{s,a,b,c\}$, $T=\{d,e,t\}$: Dann gilt $c(S,T)=3+7+3+4+3=20$.

2. $S=\{s,c,d\}$, $T=\{a,b,e,t\}$: Dann gilt $c(S,T)=4+5+3+4+5=21$.

3. $S=\{s\}$, $T=\{a,b,c,d,e,t\}$: Dann gilt $c(S,T)=4+5+3=12$.

Feststellung 2.
Ist f ein beliebiger Fluss auf $N=(G,c,s,t)$ und ist (S,T) ein beliebiger Schnitt, so gilt

$$w(f)\leq c(S,T). \tag{9.1}$$

Diese Feststellung gibt die anschaulich einleuchtende Tatsache wieder, dass von s nach t niemals mehr fließen kann, als die Kapazität eines Schnitts zulässt.

Zum Beweis von Feststellung 2 benötigen wir zunächst einen *Hilfssatz, der auch noch an anderer Stelle nützlich sein wird*. Die Richtigkeit von Feststellung 2 wird sich – wie wir unten sehen werden – als unmittelbare Folgerung aus dem Hilfssatz ergeben.

Hilfssatz.
Ist f ein Fluss auf $N=(G,c,s,t)$ mit $G=(V,E)$ und ist S eine Teilmenge von V mit $s\in S$ und $t\notin S$, so gilt

$$w(f)=f^+(S)-f^-(S). \tag{9.2}$$

Beweis des Hilfssatzes. Mit $E(S)$ bezeichnen wir die Menge aller Kanten (u,v) von N, für die sowohl $u\in S$ als auch $v\in S$ gilt. Man beachte, dass die folgenden Gleichungen gelten:

$$\sum_{v\in S} f^+(v)=f^+(S)+\sum_{e\in E(S)} f(e)$$
$$\sum_{v\in S} f^-(v)=f^-(S)+\sum_{e\in E(S)} f(e)$$

Es folgt

$$\sum_{v\in S} f^+(v)-\sum_{v\in S} f^-(v)=f^+(S)-f^-(S),$$

woraus sich für $w(f)$ die in (9.2) behauptete Identität wie folgt ergibt:

$$\begin{aligned} w(f)&=f^+(s)\\ &=f^+(s)-\underbrace{f^-(s)}_{=0}+\sum_{\substack{v\in S\\ v\neq s}}\underbrace{\Big(f^+(v)-f^-(v)\Big)}_{=0}\\ &=\sum_{v\in S}\Big(f^+(v)-f^-(v)\Big)\\ &=\sum_{v\in S} f^+(v)-\sum_{v\in S} f^-(v)\\ &=f^+(S)-f^-(S). \quad \square \end{aligned}$$

Erfüllt S die Voraussetzungen des Hilfssatzes, gilt also $S\subseteq V$, $s\in S$ und $t\notin S$, so wollen wir $f^+(S)-f^-(S)$ als den *Nettofluss* von S bezeichnen. Wir können den Hilfssatz also auch wie folgt aussprechen: Ist f ein Fluss auf N und S eine Teilmenge der Knotenmenge von N, für die $s\in S$ und $t\notin S$ gilt, *so ist der Flusswert von f gleich dem Nettofluss von S*.

Der **Beweis von Feststellung 2** ist nun ganz kurz; aufgrund des Hilfssatzes ergibt sich die Behauptung (9.1) wie folgt:

$$w(f) = f^+(S) - f^-(S) \leq f^+(S) = f(S,T) \leq c(S,T). \quad \square$$

Definition.
Ein Schnitt (S,T) von N heißt *minimal*, falls $c(S,T) \leq c(S',T')$ für alle Schnitte (S',T') von N gilt.

Ist f_0 ein maximaler Fluss und (S_0,T_0) ein minimaler Schnitt von N, so gilt nach Feststellung 2:

$$w(f_0) \leq c(S_0,T_0). \tag{9.3}$$

9.3 Das Max-Flow Min-Cut Theorem und der Labelling-Algorithmus von Ford und Fulkerson

Der folgende Satz von Ford und Fulkerson aus dem Jahre 1956 besagt, dass in (9.3) sogar Gleichheit gilt.

Satz (Max-Flow Min-Cut Theorem).
In einem Netzwerk ist der Wert eines maximalen Flusses immer gleich der Kapazität eines minimalen Schnittes.

Kurzfassung: max-flow = min-cut.

Bevor wir den Satz beweisen, beschreiben wir eine *Methode, mit der man einen gegebenen Fluss f verbessern kann* – vorausgesetzt natürlich, dass f nicht bereits maximal ist.

Diese Methode ist von zentraler Bedeutung: Sie liefert nicht nur einen Beweis des Max-Flow Min-Cut Theorems, sondern ist auch die Grundlage für einen *Algorithmus* zur Berechnung eines maximalen Flusses.

Wir erläutern die Methode anhand unseres obigen Beispiels.

Beispiel. Für das Netzwerk von Seite 107 sei f wie auf Seite 109 f. gegeben. Wir betrachten den (gerichteten) Pfad (s,b,t), der die Quelle s mit der Senke t verbindet. Weder für die Kante (s,b) noch für die Kante (b,t) wird durch f die Kapazität ausgeschöpft. Deshalb können wir in diesen beiden Kanten den Fluss soweit erhöhen, bis in einer der beiden Kanten die Kapazität erreicht ist. Dementsprechend definieren wir[4]:

$$\begin{aligned} f_1(s,b) &= 2, \\ f_1(b,t) &= 3, \\ f_1(x,y) &= f(x,y) \text{ für alle anderen Kanten.} \end{aligned}$$

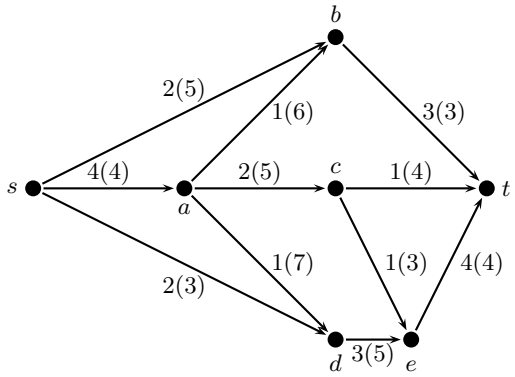
Da wir in beiden Kanten des Pfades (s,b,t) den Fluss um den gleichen Betrag angehoben haben (nämlich um 1), ist f_1 wiederum ein Fluss[5]:

$$w(f_1) = w(f) + 1 = 7 + 1 = 8.$$

Ersetzen wir in der Zeichnung von Seite 110 den alten Fluss f durch den neuen Fluss f_1, so erhalten wir die folgende Darstellung:

[4] Ist (u,v) eine Kante, so müsste man streng genommen $f((u,v))$, $f_1((u,v))$, $c((u,v))$ etc. schreiben; der Einfachheit halber schreiben wir stattdessen immer $f(u,v)$, $f_1(u,v)$, $c(u,v)$ etc., was „erlaubt“ und üblich ist, da Missverständnisse nicht möglich sind.

[5] Man beachte: (F2) gilt nach wie vor.



Wollen wir den Fluss weiter verbessern, so müssen wir etwas raffinierter vorgehen:

Wir durchlaufen Kanten *auch in entgegengesetzter Richtung.*

Genauer: Wir betrachten die Folge

$$(s, b, a, c, t) \tag{9.4}$$

von Knoten. Längs dieser Folge bewegen wir uns in unserem Netzwerk von s nach t, wobei wir die Kante zwischen a und b rückwärts durchlaufen.

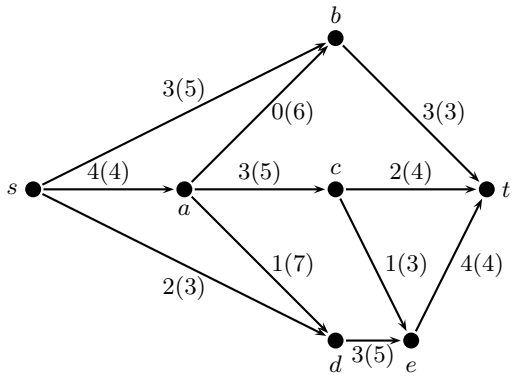
Man beachte: Für die „Vorwärtskanten“ ist dabei die Kapazität nicht ausgeschöpft, und für die rückwärts durchlaufene Kante $e = (a, b)$ gilt $f_1(e) > 0$. Wir definieren f_2 wie folgt:

$$\begin{aligned} f_2(s,b) &= f_1(s,b) + 1 \\ f_2(a,b) &= f_1(a,b) - 1 \\ f_2(a,c) &= f_1(a,c) + 1 \\ f_2(c,t) &= f_1(c,t) + 1 \end{aligned}$$

sowie

$$f_2(x, y) = f_1(x, y)$$

für die übrigen Kanten. Es ergibt sich die folgende Darstellung von f_2:



Da wir für alle „Vorwärtskanten“ den Fluss um denselben Betrag erhöht haben und gleichzeitig für alle „Rückwärtskanten“ den Fluss um genau diesen Betrag erniedrigt haben (ohne die Kapazitäten zu überschreiten bzw. 0 zu unterschreiten), ist f_2 wiederum ein Fluss[6]. Es gilt

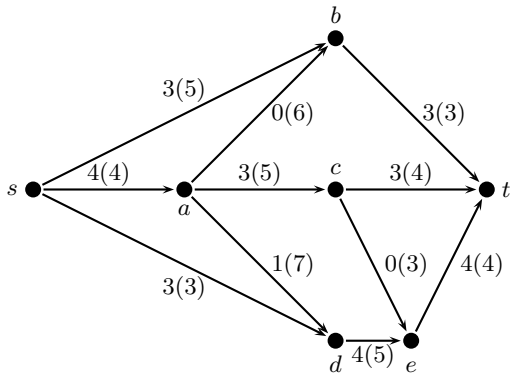
$$w(f_2) = w(f_1) + 1 = 8 + 1 = 9.$$

[6] Man beachte: (F2) gilt nach wie vor.

In ähnlicher Weise können wir den Fluss f_2 verbessern: Diesmal benutzen wir die Knotenfolge

$$(s, d, e, c, t) \tag{9.5}$$

und erhalten ganz entsprechend einen verbesserten Fluss f_3, für den $w(f_3) = 10$ gilt:



Nun können wir keine Knotenfolge mehr finden, die wie die Folgen (9.4) und (9.5) geeignet wäre, den Fluss f_3 zu verbessern. Wir haben daher den *Verdacht*, dass f_3 ein maximaler Fluss ist.

Frage: Wie können wir diesen Verdacht in eine Gewissheit verwandeln?

Antwort: Unser Verdacht wird Gewissheit, wenn wir einen *Beleg*[7] für die Optimalität von f_3 vorlegen.

Hier ist der gewünschte Beleg: Der Schnitt (S, T) mit $S = \{s, b\}$ und $T = \{a, c, d, e, t\}$ ist ein Zertifikat für die Optimalität von f_3, da für diesen Schnitt $c(S, T) = 10$ gilt. Da ebenfalls $w(f_3) = 10$ gilt, wissen wir (vgl. (9.1)): Einen besseren Fluss kann es nicht geben.

Im vorangegangenen Beispiel gab es drei Flussvergrößerungen: Beim Übergang zu f_1 kam zunächst die Knotenfolge (s, b, t) zum Einsatz, anschließend wurden die Knotenfolgen (9.4) und (9.5) verwendet. Derartige Knotenfolgen werden *flussvergrößernde Pfade* genannt. Bevor wir diesen Begriff zusammen mit verwandten Begriffen definieren, seien *zwei Bemerkungen* vorausgeschickt:

1. Wenn eine Rückwärtskante in einem flussvergrößernden Pfad P vorkommt, so handelt es sich bei P nicht um einen Pfad im gerichteten Graphen $G = (V, E)$. („Pfade im üblichen Sinne enthalten keine Rückwärtskanten.“) Es ist trotzdem üblich, von einem Pfad zu sprechen, denn: Die Knotenfolge von P beschreibt einen Pfad *im G zugrundeliegenden ungerichteten Graphen*. Dasselbe kurz und knapp gesagt: *P ist ein Pfad, wenn man die Kantenrichtungen ignoriert.*

Die zweite Bemerkung hat mit der Tatsache zu tun, dass in G Paare von *antiparallelen Kanten* zugelassen sind: Es könnte, wie in der nachfolgenden Figur, neben der Kante (a, b) auch die Kante (b, a) in G vorkommen.

Ein Paar von antiparallelen Kanten.

2. Da in G antiparallele Kanten vorkommen können, werden wir gelegentlich nicht nur die Knoten eines flussvergrößernden Pfades P angeben, sondern zusätzlich auch die Kanten von P. Dies wird zum Beispiel weiter unten in (9.6) so gemacht und dient der Klarheit und Eindeutigkeit: Falls es zwei Möglichkeiten gibt, soll eindeutig feststehen, welche Kante zu P gehört und welche nicht.

[7] Statt *Beleg* sagt man auch *Zertifikat*.

Definition.

Gegeben sei ein Netzwerk $N = (G, c, s, t)$ mit $G = (V, E)$ sowie ein Fluss f auf N; $k \geq 1$ sei eine ganze Zahl. Wir betrachten die Folge

$$P : s = v_1, e_1, \ldots, e_{k-1}, v_k, \tag{9.6}$$

die in s beginnt aber nicht unbedingt in t endet. Auf P wechseln sich Knoten und Kanten ab. Genauer: Es gilt $v_i \in V$ $(i = 1, \ldots, k)$ und $e_i \in E$ $(i = 1, \ldots, k-1)$. Außerdem soll es in P *keine Knotenwiederholungen* geben.

a) Eine solche Folge P nennt man einen *flussvergrößernden* oder *zunehmenden Pfad*, falls $v_k = t$ gilt und falls für jedes $i \in \{1, \ldots, k-1\}$ entweder

$$e_i = (v_i, v_{i+1}) \quad \text{und} \quad f(v_i, v_{i+1}) < c(v_i, v_{i+1}) \tag{9.7}$$

oder

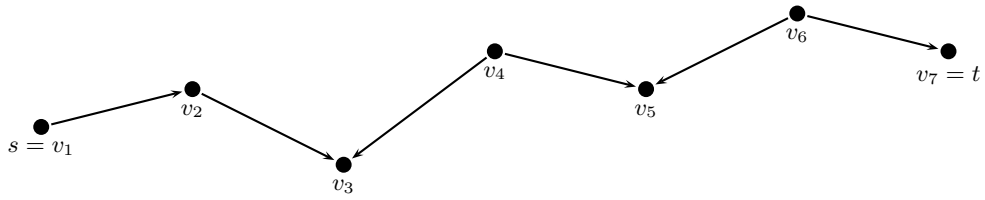
$$e_i = (v_{i+1}, v_i) \quad \text{und} \quad 0 < f(v_{i+1}, v_i) \tag{9.8}$$

erfüllt ist.

b) Wir werden auch den Fall betrachten, dass P alle unter a) genannten Bedingungen erfüllt, nur $v_k = t$ gilt möglicherweise nicht. Dann sprechen wir von einem *zunehmenden Pfad nach* v_k.

c) Gilt (9.7), so wird e_i *Vorwärtskante* von P genannt; gilt (9.8), so heißt e_i *Rückwärtskante* von P.

Weitere Sprechweisen: Unter den Voraussetzungen der obigen Definition sagen wir auch *f-vergrößernder Pfad* anstelle von flussvergrößernder Pfad. Die entsprechenden englischen Bezeichnungen sind übrigens *augmenting path* bzw. *f-augmenting path*. Wir betrachten das folgende Beispiel:



Die Vorwärtskanten in diesem Beispiel sind (v_1, v_2), (v_2, v_3), (v_4, v_5), (v_6, v_7) und die Rückwärtskanten sind (v_4, v_3) und (v_6, v_5). Soll dies ein flussvergrößernder Pfad sein, so muss also gelten:

$$\begin{aligned} f(v_1, v_2) &< c(v_1, v_2) \\ f(v_2, v_3) &< c(v_2, v_3) \\ f(v_4, v_3) &> 0 \\ f(v_4, v_5) &< c(v_4, v_5) \\ f(v_6, v_5) &> 0 \\ f(v_6, v_7) &< c(v_6, v_7). \end{aligned}$$

Gegeben sei, wie in obiger Definition, ein Netzwerk $N = (G, c, s, t)$ mit $G = (V, E)$ sowie ein Fluss f auf N. Existiert ein flussvergrößernder Pfad P, so kann man P dazu benutzen, um aus f einen Fluss f^+ zu gewinnen, für den $w(f^+) > w(f)$ gilt; anhand des Beispiels auf den Seiten 114 ff. hatten wir dies bereits gesehen – allgemein geht dies wie folgt (Bezeichnungen wie in (9.6)): Zu jeder Kante von P definieren wir eine Zahl d_i, indem wir *für Vorwärtskanten* (v_i, v_{i+1}) festsetzen:

$$d_i = c(v_i, v_{i+1}) - f(v_i, v_{i+1});$$

für Rückwärtskanten sei dagegen

$$d_i = f(v_{i+1}, v_i).$$

Dann gilt in jedem Fall $d_i > 0$. Wir setzen

$$d := \min\Big\{d_i : i = 1, \ldots, k-1\Big\},$$

d.h., d ist der kleinste Wert unter den d_i. Es gilt $d > 0$.

Wir definieren den neuen Fluss f^+ wie folgt:

- Es gelte $f^+(v_i, v_{i+1}) = f(v_i, v_{i+1}) + d$, falls (v_i, v_{i+1}) eine Vorwärtskante von P ist.
- Falls (v_{i+1}, v_i) eine Rückwärtskante von P ist, so sei $f^+(v_{i+1}, v_i) = f(v_{i+1}, v_i) - d$.
- Für alle anderen Kanten e lassen wir den Fluss durch e unverändert, d.h. $f^+(e) = f(e)$.

Aufgrund der Konstruktion von f^+ gilt dann: f^+ ist ein Fluss auf N mit

$$w(f^+) = w(f) + d. \tag{9.9}$$

Wegen $d > 0$ haben wir also $w(f^+) > w(f)$.

Man beachte: Alle Kapazitäten sind ganze Zahlen, da wir ja $c(e) \in \mathbb{N} \cup \{0\}$ für alle Kanten $e \in E$ voraussetzen.

Sind auch alle $f(e)$ ganze Zahlen, so sprechen wir von einem *ganzzahligen Fluss*.

Ist f ein ganzzahliger Fluss, so folgt aus der Definition von d, dass auch d eine ganze Zahl ist und dass somit $d \geq 1$ gilt. Wir halten fest:

$$\textit{Ist } f \textit{ ganzzahlig, so auch } f^+\textit{, und es gilt } w(f^+) \geq w(f) + 1. \tag{9.10}$$

Unsere Überlegungen legen das folgende *Verfahren zur Bestimmung eines maximalen Flusses* in einem Netzwerk nahe.

Ford-Fulkerson-Methode.

1. Man startet mit dem *Nullfluss* f_0: Dies ist der (ganzzahlige) Fluss, für den $f_0(e) = 0$ für alle $e \in E$ gilt.

2. Ist bereits ein ganzzahliger Fluss f_n konstruiert, so suche man nach einem f_n-vergrößernden Pfad P[8].

2.1. Existiert ein solches P, so benutze man P, um aus f_n einen ganzzahligen Fluss f_{n+1} mit $w(f_{n+1}) \geq w(f_n) + 1$ zu gewinnen; man wiederhole dann 2. mit f_{n+1} anstelle von f_n.

2.2. Existiert kein solcher Pfad P, so ist man fertig, da – wie wir gleich sehen werden – in diesem Fall ein Schnitt (S, T) mit $w(f_n) = c(S, T)$ existiert, was ja bedeutet, dass $w(f_n)$ maximal sein muss.

Man nennt das beschriebene Verfahren zur Bestimmung eines maximalen Flusses die *Ford-Fulkerson-Methode* (oder das Ford-Fulkerson-Verfahren oder den Ford-Fulkerson-Algorithmus).

Der **wichtigste Punkt**, der noch zu klären ist, betrifft die in **2.2.** gemachte Aussage, dass es im Fall, dass kein f_n-vergrößernder Pfad existiert, immer einen Schnitt (S, T) mit

$$w(f_n) = c(S, T) \tag{9.11}$$

geben muss.

Wir werden sehen, dass man als Ergebnis des Ford-Fulkerson-Algorithmus nicht nur einen Maximalfluss f_n erhält, sondern *dass man im letzten Schritt des Verfahrens gleichzeitig auch einen Schnitt (S, T), für den (9.11) gilt, ganz umsonst mitgeliefert bekommt.*

Das kommt Ihnen ja sicherlich bekannt vor. Schauen Sie sich noch einmal Seite 75 an!

[8] Es ist nicht schwer, diese Suche systematisch zu betreiben; Details hierzu werden später besprochen.

Am Ende des Verfahrens erhält man also neben einem maximalen Fluss auch einen minimalen Schnitt *und hat somit ein* ***Zertifikat für die Optimalität des gefundenen Flusses*** *in der Hand.* (Stellen Sie sich vor, dass jemand anderes bezweifelt, dass der von Ihnen gefundene Fluss f^* tatsächlich maximal ist. Dann brauchen Sie nur den ebenfalls gefundenen minimalen Schnitt (S^*, T^*) aus der Tasche zu holen und darauf hinzuweisen, dass

$$w(f^*) = c(S^*, T^*)$$

gilt: Ihr Gegenüber wird sofort verstummen.) Beim Ford-Fulkerson-Verfahren handelt es sich also (ebenso wie beim Simplexverfahren) um einen *zertifizierenden Algorithmus* (vgl. auch Seite 75).

Hier ist nun der noch fehlenden Baustein.

Feststellung 3.
Ist $N = (G, c, s, t)$ ein Netzwerk mit $G = (V, E)$ und ist f ein Fluss auf N, für den es keinen f-vergrößernden Pfad P gibt, so existiert ein Schnitt (S, T) mit $w(f) = c(S, T)$.

Bevor wir diese Feststellung beweisen, erläutern wir die *Grundidee*: Stellen Sie sich vor, dass Sie in der n-ten Iteration des Ford-Fulkerson-Verfahrens den Fluss f_n erhalten haben und dass f_n maximal ist – was Sie allerdings noch nicht wissen. Sie steigen also in die $(n+1)$-te Iteration ein und versuchen einen f_n-vergrößernden Pfad P zu konstruieren, wobei Sie bei der Quelle s starten und versuchen, die Senke t mit einem solchen Pfad zu erreichen. Da f_n bereits maximal ist, wird das nicht gelingen – Sie werden immer nur Knoten $v \neq t$ erreichen.

Die Menge aller Knoten v, die Sie auf diese Art erreichen können, nennen wir S; die Menge der übrigen Knoten nennen wir T.

Es ist nicht schwer, sich zu überlegen, dass für den auf diese Art gefundenen Schnitt (S, T) gilt: $w(f_n) = c(S, T)$; dass dies tatsächlich so ist, wird im Folgenden nachgewiesen.

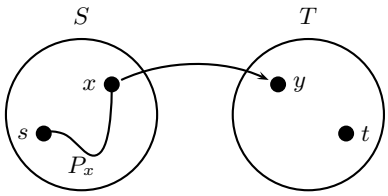
Beweis von Feststellung 3. Wir setzen voraus, dass $N = (G, c, s, t)$ ein Netzwerk mit $G = (V, E)$ ist und dass f ein Fluss auf N ist, für den es keinen f-vergrößernden Pfad gibt.

Wir definieren S als die Menge derjenigen $v \in V$, für die es einen zunehmenden Pfad nach v gibt. Dann gilt $s \in S$, da der einpunktige Pfad, der nur aus s besteht, ein zunehmender Pfad nach s ist. Außerdem gilt $t \notin S$, da es andernfalls einen f-vergrößernden Pfad geben würde. (Man beachte: Die Begriffe *zunehmender Pfad nach t* und *f-vergrößernder Pfad* bedeuten dasselbe.) Wir setzen $T = \overline{S}$. Es handelt sich bei (S, T) also um einen Schnitt.

Wir zeigen nun, dass für diesen Schnitt (S, T) *gilt*:

$$w(f) = c(S, T). \tag{9.12}$$

Zum Nachweis von (9.12) sei (x, y) eine beliebige Kante aus E mit $x \in S$ und $y \in T$. Wegen $x \in S$ existiert ein zunehmender Pfad nach x, den wir P_x nennen wollen (siehe Zeichnung). Es folgt $f(x, y) = c(x, y)$, *da man andernfalls* P_x *zu einem zunehmenden Pfad nach* y *verlängern könnte*, im Widerspruch zu $y \notin S$.



Da (x, y) eine beliebige Kante aus E mit $x \in S$ und $y \in T$ ist, gilt also $f(x, y) = c(x, y)$ für alle derartigen Kanten, d.h., es gilt:

$$f^+(S) = f(S, T) = c(S, T).$$

Auf eine ganz ähnliche Art findet man, dass gilt (Übungsaufgabe!):

$$f^-(S) = f(T, S) = 0.$$

Aufgrund des Hilfssatzes (vgl. Seite 113) wissen wir, dass

$$w(f) = f^+(S) - f^-(S)$$

gilt. Insgesamt ergibt sich demnach:

$$w(f) = f^+(S) - f^-(S) = c(S,T). \quad \square$$

Zu guter Letzt: Ist $K = \sum_{e \in E} c(e)$ die Summe aller in N vorkommenden Kantenkapazitäten, so gilt aufgrund von Feststellung 2 natürlich $w(f) \leq K$. *Dadurch ist sichergestellt, dass man beim Ford-Fulkerson-Verfahren nach endlich vielen Flussvergrößerungen fertig ist. (Man beachte vor allem (9.10)!)*

Das Verfahren terminiert also, man kommt nach endlich vielen Flussvergrößerungen zu **2.2.**, womit auch das *Max-Flow Min-Cut Theorem* bewiesen ist.

Das Ford-Fulkerson-Verfahren wird in der englischsprachigen Literatur auch

labelling algorithm

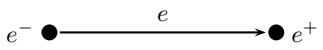
genannt; wir werden in Kürze sehen, woher der Name kommt. (Auf Deutsch sagt man übrigens Labelling-Algorithmus oder Markierungsalgorithmus.)

In diesem Abschnitt sind wir (zumindest teilweise) der Darstellung in den folgenden Büchern gefolgt:

- D. Jungnickel: *Graphs, Networks and Algorithms*. Springer-Verlag. 2012. 4. Auflage.
- A. Beutelspacher, M.-A. Zschiegner: *Diskrete Mathematik für Einsteiger*. Vieweg-Verlag. 2014. 5. Auflage.

Wir besprechen nun weitere Details des Labelling-Algorithmus, wobei wir verstärkt nach Jungnickel vorgehen; unter anderem lernen wir auch den *Algorithmus von Edmonds und Karp* kennen.

Eine Bezeichnung, die im Jungnickel häufig auftaucht: Ist e eine Kante eines Digraphen, so bezeichnet e^- den Anfangsknoten und e^+ den Endknoten von e.



Hier zunächst einmal der *Labelling-Algorithmus in Grobform*, wie er im Jungnickel beschrieben wird[9]:

(1) $f(e) \leftarrow 0$ for all edges e;
(2) **while** there exists an augmenting path with respect to f **do**
(3) let $P = (e_1, \ldots, e_r)$ be an augmenting path from s to t;
(4) $d \leftarrow \min\big(\{c(e_i) - f(e_i) : e_i \text{ is a forward edge in } P\}$
$\cup\ \{f(e_i) : e_i \text{ is a backward edge in } P\}\big)$;
(5) $f(e_i) \leftarrow f(e_i) + d$ for each forward edge e_i;
(6) $f(e_i) \leftarrow f(e_i) - d$ for each backward edge e_i;
(7) **od**

Die Markierungen (labels) sind in dieser Grobform noch nicht vorhanden. Sie kommen erst ins Spiel, wenn es um die *Details* geht. Insbesondere ist ja noch festzulegen, auf welche Art die Suche nach einem „augmenting path“ erfolgen soll. Außerdem bietet es sich an, die Suche nach einem solchen Pfad und die Berechnung von d geeignet zu kombinieren. Und schließlich wollen wir ja gar nicht nur einen maximalen Fluss, sondern auch einen minimalen Schnitt bestimmen. All dies führt zur Verwendung von

[9] Zur Erinnerung: Im Englischen heißt flussvergrößernder Pfad *augmenting path*. Um einen flussvergrößernden Pfad P zu beschreiben, genügt es, die Kanten von P anzugeben. In (3) wird deshalb $P = (e_1, \ldots, e_r)$ geschrieben.

„labels“. (Es sind übrigens die Knoten, die markiert werden; es geht also nicht um Kanten-, sondern um *Knotenmarkierungen*.)

Hier nun die Details des Labelling-Algorithmus (aus Jungnickel: *Graphs, Networks and Algorithms*, Springer-Verlag (2012, 4. Auflage)):

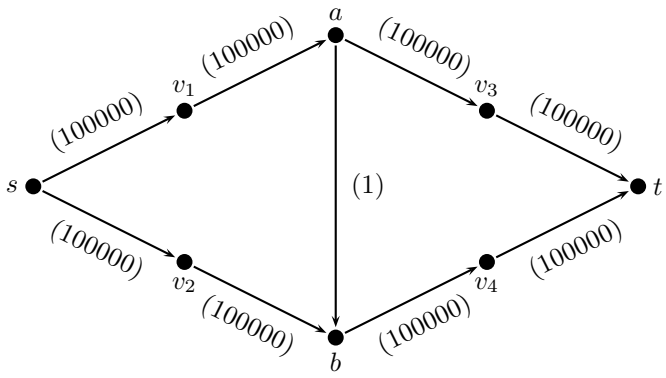
Labelling algorithm of Ford and Fulkerson.
Let $N = (G, c, s, t)$ be a flow network.

Procedure FORDFULK(N, f, S, T)
(1) **for** $e \in E$ **do** $f(e) \leftarrow 0$ **od**
(2) label s with $(-, \infty)$;
(3) **for** $v \in V$ **do** $u(v) \leftarrow$ false; $d(v) \leftarrow \infty$ **od**
(4) **repeat**
(5) choose a vertex v which is labelled and satisfies $u(v) =$ false;
(6) **for** $e \in \{e \in E : e^- = v\}$ **do**
(7) **if** $w = e^+$ is not labelled **and** $f(e) < c(e)$ **then**
(8) $d(w) \leftarrow \min\{c(e) - f(e), d(v)\}$; label w with $(v, +, d(w))$ **fi**
(9) **od**
(10) **for** $e \in \{e \in E : e^+ = v\}$ **do**
(11) **if** $w = e^-$ is not labelled **and** $f(e) > 0$ **then**
(12) $d(w) \leftarrow \min\{f(e), d(v)\}$; label w with $(v, -, d(w))$ **fi**
(13) **od**
(14) $u(v) \leftarrow$ true;
(15) **if** t is labelled
(16) **then** let d be the last component of the label of t;
(17) $w \leftarrow t$;
(18) **while** $w \neq s$ **do**
(19) find the first component v of the label of w;
(20) **if** the second component of the label of w is $+$
(21) **then** set $f(e) \leftarrow f(e) + d$ for $e = vw$
(22) **else** set $f(e) \leftarrow f(e) - d$ for $e = wv$
(23) **fi**
(24) $w \leftarrow v$
(25) **od**
(26) delete all labels except for the label of s;
(27) **for** $v \in V$ **do** $d(v) \leftarrow \infty$; $u(v) \leftarrow$ false **od**
(28) **fi**
(29) **until** $u(v) =$ true for all vertices v which are labelled;
(30) let S be the set of vertices which are labelled and put $T \leftarrow V \setminus S$

Eine Anmerkung zu den Bezeichnungen: Die Wahl des Buchstabens u leitet sich von „untersucht“ (engl. *scanned*) her; $u(v) =$ true bedeutet dementsprechend, dass der Knoten v in der jeweiligen Iteration bereits *untersucht* wurde. Entsprechend bedeutet $u(v) =$ false, dass v in der jeweiligen Iteration noch nicht untersucht wurde. Statt „untersucht“ sagt man auch *inspiziert*.

9.4 Der Algorithmus von Edmonds und Karp

Einen „Schönheitsfehler“ hat der vorgestellte Algorithmus allerdings noch. Das wird deutlich, wenn wir uns das folgende Beispiel anschauen:



Die geklammerten Zahlen an den Kanten sind die Kapazitäten. Man sieht sofort, dass der optimale Flusswert gleich 200000 ist.

Wendet man das Ford-Fulkerson-Verfahren an, so könnte es einem bei ungeschickter Wahl der flussvergrößernden Pfade passieren, dass in jeder Iteration der Fluss nur um den Wert 1 wächst. (Wie nämlich?) Man wäre also erst nach 200000 Iterationen fertig – und das, obwohl das Netzwerk nur 8 Knoten hat!

Dieses Defizit des Labelling-Algorithmus gilt es zu beseitigen. Der folgende Satz von Edmonds und Karp zeigt, wie das geschehen kann[10].

Satz (Edmonds und Karp, 1972).

Wählt man im Labelling-Algorithmus den flussvergrößernden Pfad P immer so, dass P möglichst wenige Kanten enthält, so terminiert der Algorithmus nach höchstens

$$\left\lfloor \frac{(n-1)m}{2} \right\rfloor$$

Flussvergrößerungen (für $n = |V|$ und $m = |E|$).

Im Satz von Edmonds und Karp wird also eine obere Schranke für die Zahl der Flussvergrößerungen angegeben, die *unabhängig von der Größe der Kapazitäten* ist. Stattdessen ist die angegebene Schranke nur von der Knotenzahl $n = |V|$ und der Kantenzahl $m = |E|$ abhängig.

Allerdings muss sichergestellt sein, dass auch immer ein möglichst kurzer flussvergrößernder Pfad ausgewählt wird, wobei „möglichst kurz“ natürlich bedeuten soll, dass P unter allen flussvergrößernden Pfaden möglichst wenige Kanten besitzt.

Dies ist nicht schwer zu erreichen, man erledigt dies, indem man in Zeile (5) des Algorithmus wie bei einer *Breitensuche* (engl. *breadth first search*; kurz: *BFS*) vorgeht.

Hierzu ist (5) nur durch die modifizierte Zeile (5') zu ersetzen:

(5') among all vertices with $u(v) =$ false, let v be the vertex which was labelled first.

Der Unterschied zwischen (5) und (5') liegt auf der Hand:

- In (5) wird ein Knoten v, der markiert ist, aber noch nicht inspiziert wurde, *beliebig* ausgewählt.
- Dagegen wird in (5') unter allen Knoten, die markiert sind, aber noch nicht inspiziert wurden, derjenige Knoten v ausgewählt, *der zuerst markiert wurde*.

Den derart modifizierten Labelling-Algorithmus („Ersetzung von (5) durch (5')“) nennt man den *Algorithmus von Edmonds und Karp*. Das Neue am Algorithmus von Edmonds und Karp ist, dass in (5') nach dem folgenden Motto vorgegangen wird:

[10] Der Beweis des Satzes von Edmonds und Karp ist nicht schwierig. Am Ende dieses Abschnitts wird ein Beweis vorgestellt (auf der Basis des Buchs von Jungnickel).

Umsetzen lässt sich (5') dadurch, dass man die markierten, aber noch nicht inspizierten Knoten mithilfe einer *Warteschlange* (engl. *queue*) verwaltet.

Das nachfolgende, etwas umfangreichere Beispiel illustriert die Arbeitsweise des Algorithmus von Edmonds und Karp.

Beispiel (aus Jungnickel). Es geht um das folgende Netzwerk N, bei dem die Kapazitäten in Klammern angegeben sind:

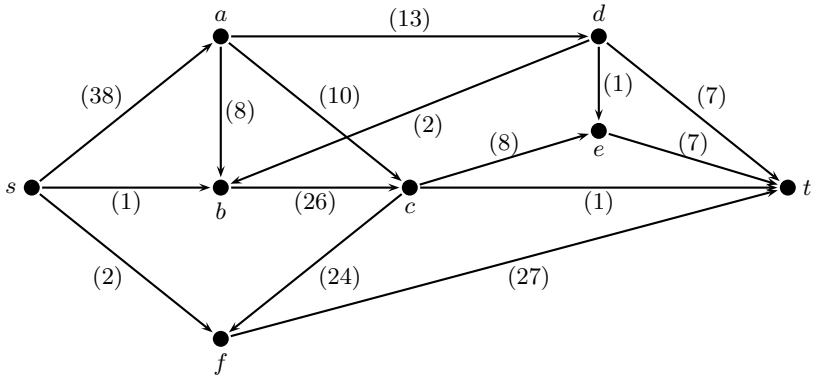


Abbildung 9.1

Im Folgenden wird der Algorithmus von Edmonds und Karp verwendet, um für N einen maximalen Fluss und einen minimalen Schnitt zu bestimmten. Die Zahlen ohne Klammern geben den Fluss durch die einzelnen Kanten an, wobei es natürlich mit dem Nullfluss (siehe Abbildung 9.2) losgeht. Die Knoten werden in Abbildung 9.2 in der Reihenfolge s, a, b, f, c, d, t markiert; e wird nicht markiert, da bereits zuvor in Zeile (15) des Labelling-Algorithmus festgestellt wurde, dass die Senke t mit einem Label versehen worden ist. Der gefundene flussvergrößernde Pfad wird immer fett gezeichnet. Im Anschluss an Abbildung 9.2 wird detailliert beschrieben, wie Abbildung 9.2 aus 9.1 entsteht und vor allem, wie die angegebene Reihenfolge s, a, b, f, c, d, t zustande kommt. Danach wird in den Abbildungen 9.3 - 9.11 der weitere Verlauf angegeben.

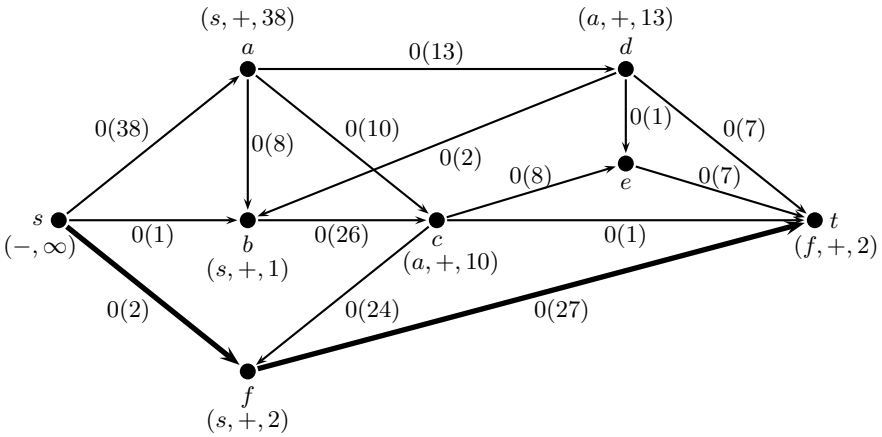


Abbildung 9.2: $w(f_0) = 0$.

Wir beschreiben im Folgenden, wie die zuvor angegebene Reihenfolge der Knotenmarkierungen in Abbildung 9.2 zustande kommt. Diese Reihenfolge ist nur zum Teil durch den Algorithmus von Edmonds und Karp vorgegeben; um eine vollständig festgelegte Reihenfolge zu erhalten, fügen wir die folgende **Regel** hinzu:

($\star$) Ist im Algorithmus von Edmonds und Karp die Reihenfolge der zu markierenden Knoten nicht festgelegt, so soll die *alphabetische Reihenfolge* den Ausschlag geben.

Wie wir wissen, geht es mit dem Nullfluss los (siehe Zeile (1) des Labelling-Algorithmus sowie Abbildung 9.2); anschließend erhält s in Zeile (2) die Markierung $(-, \infty)$. Die Warteschlange, die die markierten, aber noch nicht inspizierten Knoten enthält, wollen wir mit Q bezeichnen. Anfangs enthält Q also nur den Knoten s, d.h., nach abgeschlossener Initialisierung (Zeile (1)-(3)) bietet sich für Q das folgende Bild:

$$Q: \ s.$$

Beim ersten Durchlauf der in Zeile (4) beginnenden repeat-Schleife wird in (5') als Knoten v die Quelle s gewählt; kurz: $v = s$. Danach erhalten die Knoten a, b und f ihre jeweiligen Markierungen (siehe Zeile (6)-(9) bzw. Abbildung 9.2). Wegen ($\star$) erfolgt dies in der alphabetischen Reihenfolge: Zuerst wird a markiert, dann b und danach f. Die Ausführung der anschließenden Zeilen (10)-(13) entfällt, da es keine Kanten gibt, die in s hineinführen. Damit sieht unsere Warteschlange Q nun wie folgt aus:

$$Q: \ s\ a\ b\ f.$$

Da die Inspizierung von s in Zeile (13) zum Abschluss gekommen ist, wird in Zeile (14) $u(s) = \text{true}$ gesetzt. Für die Warteschlange bedeutet dies, dass s aus Q gelöscht wird; wir stellen den Vorgang wie folgt dar:

$$Q: \ \not{s}\ a\ b\ f.$$

In Zeile (15) wird gefragt, ob t bereits markiert ist; da dies zu verneinen ist, entfällt die Ausführung der Zeilen (16)-(28). In Zeile (29) wird geklärt, ob die repeat-Schleife noch ein weiteres Mal zu durchlaufen ist: Da es markierte, aber nicht inspizierte Knoten gibt (Q ist nicht leer!), steigen wir bei (4) wieder ein.

Nun wirkt sich zum ersten Mal die Tatsache aus, dass (5) durch (5') ersetzt wurde: In Q steht a an erster Stelle, d.h., in (5') setzen wir $v = a$.

Anschließend wird (6)-(9) ausgeführt, was dazu führt, dass c und d ihre jeweiligen Markierungen bekommen (in dieser Reihenfolge); b ist nicht zu berücksichtigen, da der Knoten b bereits markiert ist. Bei der anschließenden Ausführung von (10)-(13) passiert nichts Erwähnenswertes: (s, a) ist die einzige Kante, die in a hineinführt – und s ist bereits markiert! Nach Ausführung von (14) gilt $u(a) = \text{true}$; für unsere Warteschlange bedeutet dies, dass sich Q aktuell im folgenden Zustand befindet:

$$Q: \ \not{s}\ \not{a}\ b\ f\ c\ d.$$

Nun wird wie zuvor fortgefahren: In (15) wird festgestellt, dass t noch nicht markiert ist, weshalb es bei (29) weitergeht. Wegen $Q \neq \emptyset$ wird die repeat-Schleife ein weiteres Mal durchlaufen, diesmal für $v = b$. Dies führt zu keinen weiteren Markierungen, da a, c, d und s bereits markiert sind; es wird $u(b) = \text{true}$ gesetzt und man erhält

$$Q: \ \not{s}\ \not{a}\ \not{b}\ f\ c\ d.$$

Im nächsten Durchlauf der repeat-Schleife gilt $v = f$; zusätzlich markiert wird nur der Knoten t. Es wird $u(f) = \text{true}$ gesetzt, was zu

$$Q: \ \not{s}\ \not{a}\ \not{b}\ \not{f}\ c\ d\ t$$

führt. Bei der Abfrage in (15), ob t markiert ist, lautet die Antwort diesmal ja. Folglich werden diesmal die Zeilen (16)-(28) ausgeführt, was zu folgendem Resultat führt: Man erhält den flussvergrößernden

Pfad s, f, t (siehe Abbildung 9.2) und der Fluss wird in den Kanten dieses Pfades um 2 erhöht (siehe Abbildung 9.3). Die bisherigen Markierungen werden bis auf die Markierung von s gelöscht und es wird $u(v)$ für alle Knoten v auf false gesetzt. Für Q bedeutet dies, dass Q wie am Anfang nur noch s enthält:

$$Q: \ s.$$

Da anschließend in (29) festgestellt wird, dass es markierte, aber nicht inspizierte Knoten gibt (Es gilt $Q \neq \emptyset$!), geht es in den nächsten Durchlauf der repeat-Schleife, wobei $v = s$ gilt. Hier nun der weitere Verlauf:

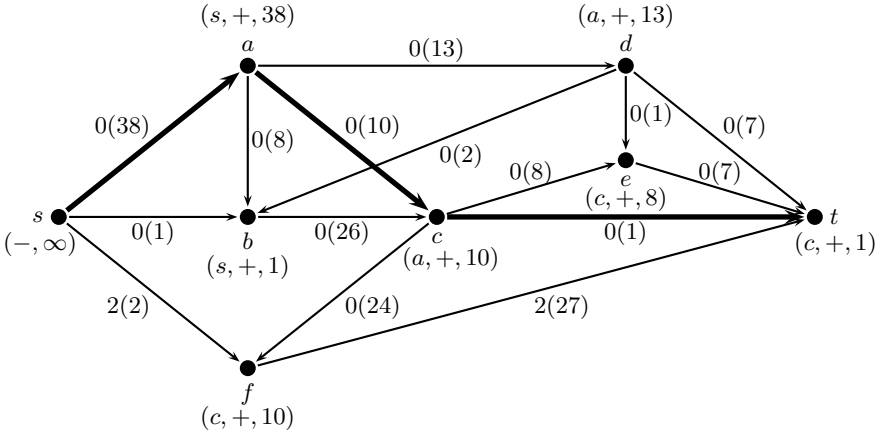


Abbildung 9.3: $w(f_1) = 2$.

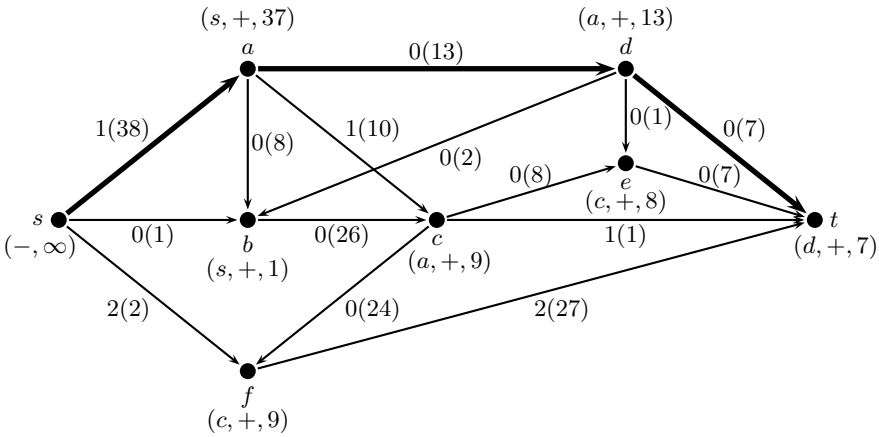


Abbildung 9.4: $w(f_2) = 3$.

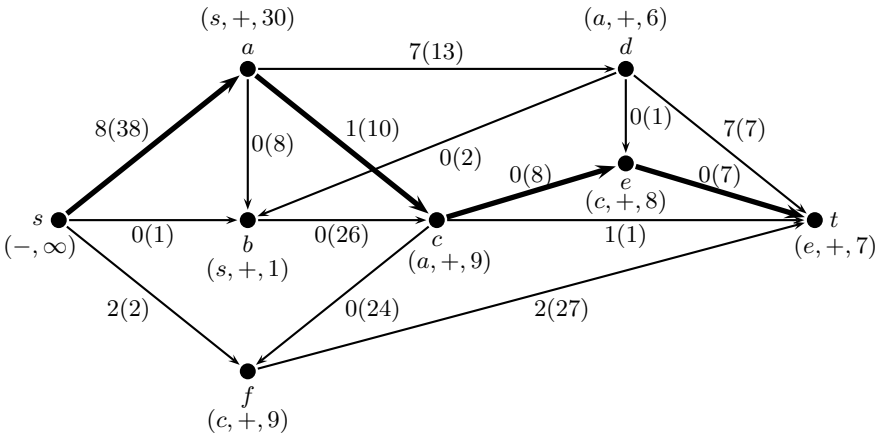


Abbildung 9.5: $w(f_3) = 10$.

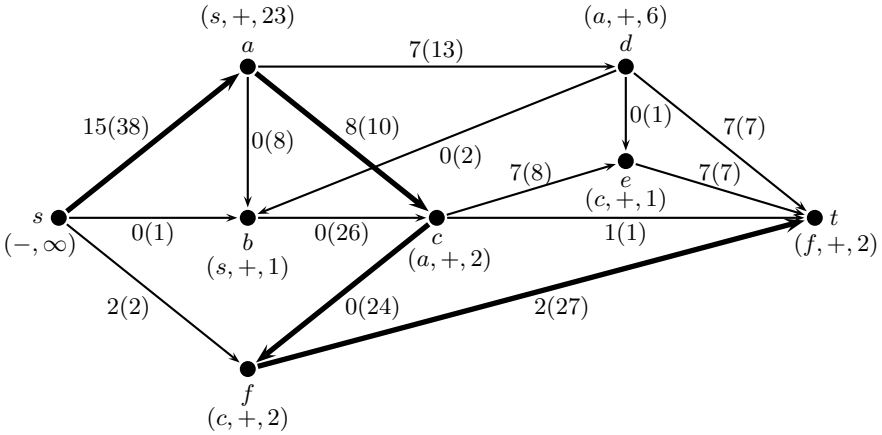


Abbildung 9.6: $w(f_4) = 17$.

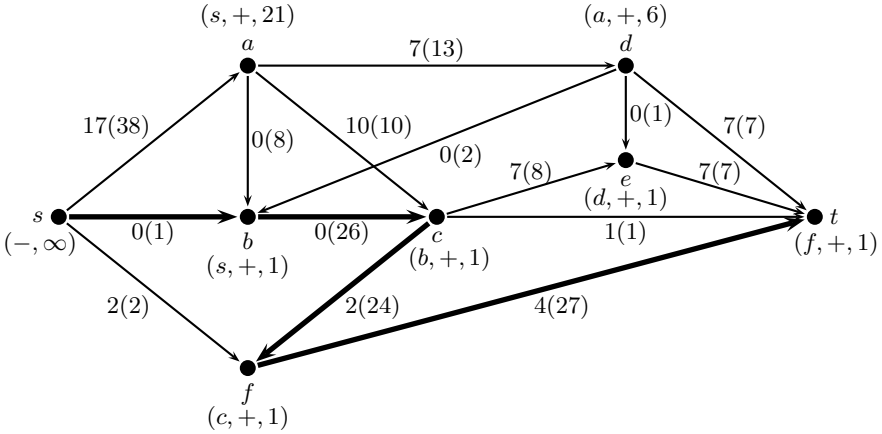


Abbildung 9.7: $w(f_5) = 19$.

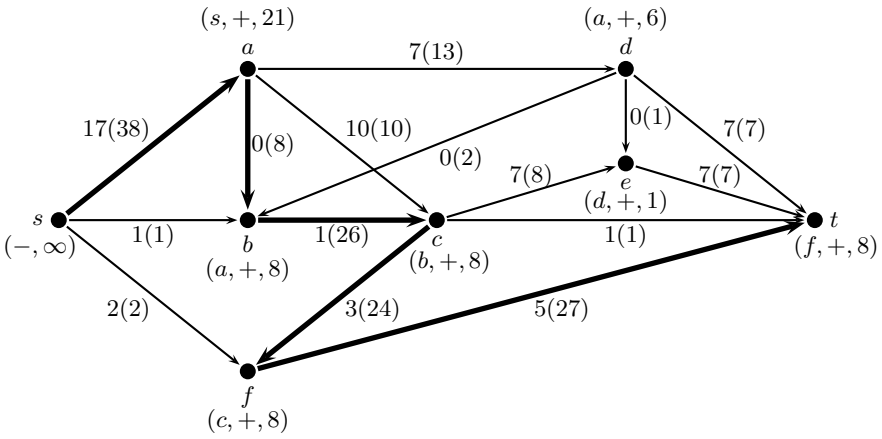


Abbildung 9.8: $w(f_6) = 20$.

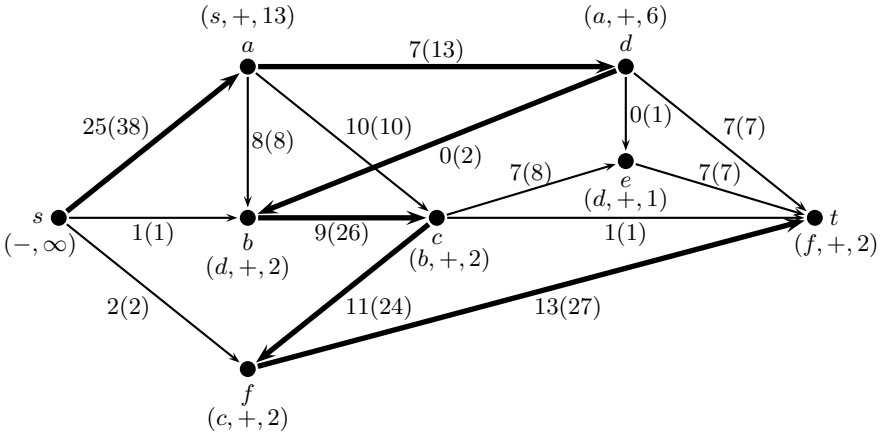


Abbildung 9.9: $w(f_7) = 28$.

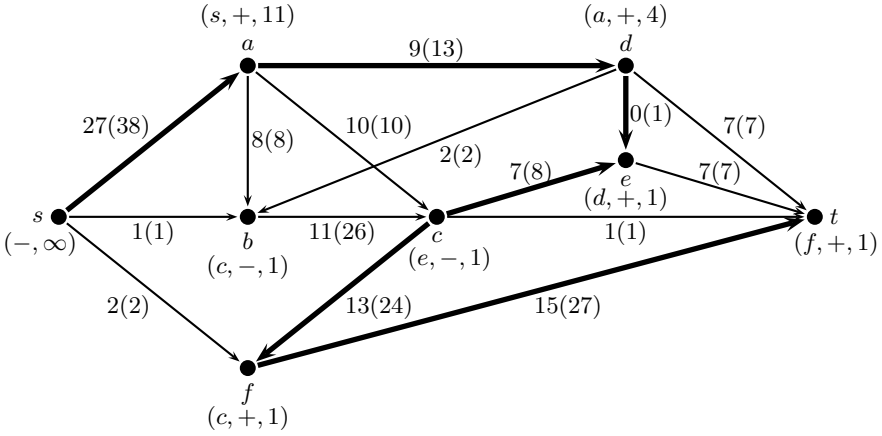


Abbildung 9.10: $w(f_8) = 30$.

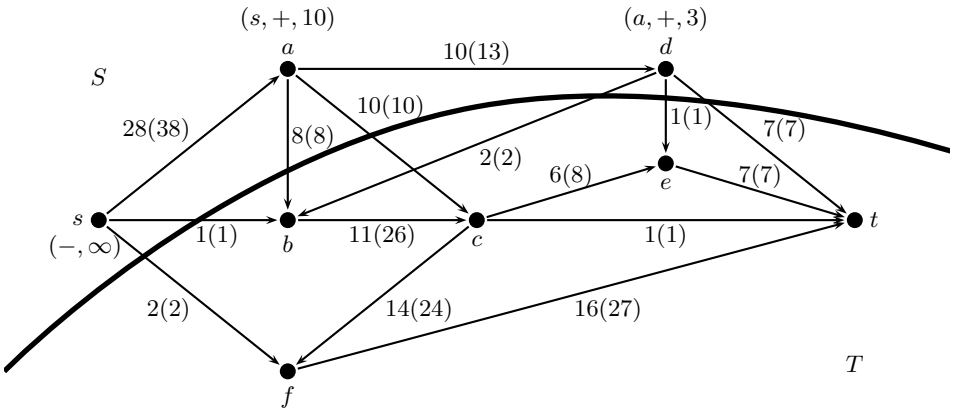


Abbildung 9.11: $w(f_9) = 31 = c(S,T)$.

Zur Übung:

a) Denken Sie sich in Abbildung 9.3 die Markierungen weg. Machen Sie sich Schritt für Schritt den Markierungsprozess klar, wobei Sie die Regel ($\star$) beachten. Geben Sie auch immer den jeweils aktuellen Zustand von Q an.

b) Wie kommt in Abbildung 9.3 der flussvergrößernde Pfad s, a, c, t zustande? Wie kommt der in Abbildung 9.4 angegebene Fluss zustande?

c) Führen Sie die Überlegungen aus a) und b) ebenso für Abbildung 9.10 (anstelle von 9.3) durch!

d) Führen Sie Entsprechendes auch für Abbildung 9.11 durch!

Zur Erinnerung: Auf Seite 117 haben wir definiert, was unter einem *zunehmenden Pfad zu einem Knoten* w zu verstehen ist; statt „zunehmender Pfad zu einem Knoten w“ haben wir auch kurz „zunehmender Pfad nach w“ gesagt.

Vor allem durch das vorangegangene, etwas umfangreichere Beispiel sollte klar geworden sein, dass es beim Algorithmus von Edmonds und Karp ganz wesentlich um *Erreichbarkeit* von Knoten durch zunehmende Pfade geht.

Wir führen die folgenden **Sprechweisen** ein: Anstatt zu sagen „es gibt einen zunehmenden Pfad nach w“ benutzen wir zukünftig auch Sprechweisen wie „w ist durch einen zunehmenden Pfad erreichbar“ oder „w wurde durch einen zunehmenden Pfad erreicht“.

Der Zusammenhang zwischen zunehmenden Pfaden und Knotenmarkierungen soll kurz angesprochen werden. Hierzu betrachten wir die Markierung

$$(c,+,9)$$

des Knotens f in Abbildung 9.5. Dass der Knoten f markiert wurde bedeutet, dass f im Laufe des Markierungsprozesses durch einen zunehmenden Pfad erreicht wurde; wir wollen diesen zunehmenden Pfad P_f nennen.

Der *erste Eintrag* in der Markierung $(c,+,9)$ zeigt an, dass c der Vorgänger von f auf P_f ist.

Der *zweite Eintrag* bedeutet, dass f auf einer Vorwärtskante erreicht wurde; mit anderen Worten: Die Vorwärtskante (c,f) ist die letzte Kante auf P_f. (Wäre f auf einer Rückwärtskante erreicht worden, so wäre der zweite Eintrag ein Minuszeichen gewesen.)

Der *dritte Eintrag* gibt die Größe des *Flaschenhalses* (engl. *bottleneck*) von P_f an. Um zu erkennen, was damit gemeint ist, bestimmen wir den gesamten Pfad P_f: Der Knoten c ist mit $(a,+,9)$ markiert, d.h., a ist der Vorgänger von c auf P_f; der Knoten a ist mit $(s,+,30)$ markiert, d.h., s ist der Vorgänger von a auf P_f. Somit lautet der Pfad P_f wie folgt:

$$P_f : \ s,\ a,\ c,\ f.$$

Der Pfad P_f durchläuft seine drei Kanten $e_1 = (s, a)$, $e_2 = (a, c)$ und $e_3 = (c, f)$ in Vorwärtsrichtung, weshalb für alle drei Kanten die Differenz zwischen der Kapazität und dem aktuellen Fluss betrachtet wird: Im Fall von e_1 beträgt diese Differenz $38 - 8 = 30$, im Fall von e_2 erhält man $10 - 1 = 9$, und für e_3 ergibt sich $24 - 0 = 24$. Der kleinste Wert ist also 9 – und genau dies liest man an der Markierung $(c, +, 9)$ in der dritten Stelle ab. Woher der Ausdruck „Größe des Flaschenhalses“ kommt und weshalb man e_2 als „Flaschenhalskante“ bezeichnet, sollte aufgrund der vorangegangenen Ausführungen klar sein.

Die Komplexität des Algorithmus von Edmonds und Karp

Die Komplexität des Algorithmus von Edmonds und Karp ist $O(nm^2)$ für $n = |V|$ und $m = |E|$. Wir skizzieren, weshalb dies so ist: Das Auffinden eines zunehmenden Pfades (bzw. im letzten Schritt die vergebliche Suche nach einem solchen) ist in $O(m)$ Schritten durchführbar, da jede Kante höchstens zweimal untersucht wird; und beim anschließenden Ändern des Flusses ist jede Kante höchstens einmal involviert. Da es nach dem Satz von Edmonds und Karp nur $O(nm)$ Flussvergrößerungen gibt, folgt insgesamt die Komplexitätsschranke $O(nm^2)$.

Wie bisher bezeichne $G = (V, E)$ den gerichteten Graphen des Netzwerks N. Wir schließen diesen Abschnitt mit einer Bemerkung zur *Darstellung von G*. Es bietet sich als Standardlösung an, G durch *Adjazenzlisten* darzustellen. Dabei ist es zweckmäßig, zu jedem Knoten v *zwei* Adjazenzlisten zur Verfügung zu haben:

- Die erste Liste enthält sämtliche Knoten y, für die $(v, y) \in E$ gilt.
- Die zweite Liste enthält sämtliche Knoten x, für die $(x, v) \in E$ gilt.

Beide Listen kommen zum Einsatz, wenn im Labelling-Algorithmus die repeat-Schleife durchlaufen wird: Die erste Liste wird durchlaufen, wenn die Zeilen (6)-(9) abgearbeitet werden; die zweite Liste wird anschließend verwendet, wenn es um die Zeilen (10)-(13) geht. (Sind die Knoten mit Buchstaben bezeichnet und sind beide Listen alphabetisch geordnet, so entspricht dies gerade der Regel ($\star$) auf Seite 124.)

Es gibt zahlreiche andere Algorithmen zum Auffinden eines Maximalflusses in einem Netzwerk – wir haben hier nur die ersten Anfänge behandelt. Wer mehr wissen möchte, kann beispielsweise zu den Büchern von Cormen et al oder Jungnickel greifen. Stellvertretend für die vielen anderen Lehrbücher, in denen Weiterführendes behandelt wird, sei außerdem genannt:

- Jon Kleinberg, Éva Tardos: *Algorithm Design*. Pearson-Verlag. 2006.

Beweis des Satzes von Edmonds und Karp

Wir beginnen mit der Einführung einiger Schreibweisen und schicken dem Beweis des Satzes von Edmonds und Karp zwei Hilfssätze voraus.

Mit f_0 sei der Fluss mit Wert 0 bezeichnet, der im Algorithmus von Edmonds und Karp unter (1) definiert wird; mit $f_1, \ldots, f_r$ bezeichnen wir die Folge der anschließend konstruierten Flüsse.

Für jedes $v \in V$ sei mit

$$x_v(k)$$

die minimale Länge eines zunehmenden Pfades nach v bezüglich des Flusses f_k bezeichnet.

Lemma 1.
Es gilt $x_v(k+1) \geq x_v(k)$ für alle $v \in V$ und k mit $0 \leq k < r$.

Beweis. Für $v = s$ gilt die Behauptung klarerweise. Angenommen, für ein Paar (v, k) mit $v \neq s$ gelte $x_v(k) > x_v(k+1)$. Dabei soll zu festem k der Knoten v so gewählt sein, dass der Wert $x_v(k+1)$ im folgenden Sinne *minimal* ist: Für alle Knoten $w \in V$, die die Ungleichung $x_w(k) > x_w(k+1)$ ebenfalls erfüllen, soll $x_w(k+1) \geq x_v(k+1)$ gelten.

Wir betrachten nun einen möglichst kurzen zunehmenden Pfad P nach v bezüglich des Flusses f_{k+1}. Es sei e die letzte Kante von P. Dann kann e entweder eine Vorwärts- oder eine Rückwärtskante von P sein.

Wir behandeln zunächst den Fall, dass e eine Vorwärtskante von P ist. Dann gilt also $e = (u, v)$ für einen bestimmten Knoten u. Man beachte, dass hieraus $f_{k+1}(e) < c(e)$ folgt. Außerdem gilt

$$x_v(k+1) = x_u(k+1) + 1,$$

da das Teilstück von P, das in u endet, ein zunehmender Pfad nach u bzgl. f_{k+1} mit minimaler Länge ist. Aufgrund der Minimalwahl von v gilt also

$$x_u(k+1) \geq x_u(k).$$

Es folgt

$$x_v(k+1) \geq x_u(k) + 1. \tag{9.13}$$

Außerdem muss $f_k(e) = c(e)$ gelten, da andernfalls $x_v(k) \leq x_u(k) + 1$ folgen würde, woraus sich mit (9.13) ein Widerspruch zur Annahme $x_v(k) > x_v(k+1)$ ergibt.

Aus $f_k(e) = c(e)$ und $f_{k+1}(e) < c(e)$ ergibt sich, dass e beim Übergang von f_k zu f_{k+1} eine *Rückwärtskante* gewesen sein muss. Aus der Tatsache, dass für diesen Übergang ein kürzester flussvergrößernder Pfad gewählt wurde, ergibt sich somit $x_u(k) = x_v(k) + 1$. Hieraus folgt zusammen mit (9.13), dass $x_v(k+1) \geq x_v(k) + 2$ gilt, im Widerspruch zur Annahme $x_v(k) > x_v(k+1)$.

Somit haben wir den Fall, dass e eine Vorwärtskante von P ist, erledigt. Den verbliebenen Fall, dass e eine Rückwärtskante von P ist, behandelt man auf die gleiche Art. □

Analog zum Begriff *zunehmender Pfad nach v* definiert man, was unter einem *zunehmenden Pfad von v nach t* zu verstehen ist. (Wie nämlich?) Daran anknüpfend sei mit

$$y_v(k)$$

die minimale Länge eines zunehmenden Pfades von v nach t bezüglich f_k bezeichnet. Analog zu Lemma 1 lässt sich Folgendes zeigen:

Lemma 2.
Es gilt $y_v(k+1) \geq y_v(k)$ für alle $v \in V$ und k mit $0 \leq k < r$.

Bevor wir den Satz von Edmonds und Karp mithilfe von Lemma 1 und 2 beweisen, sei an den Begriff der *Flaschenhalskante* oder, wie man auch sagt, *kritischen Kante* erinnert.

Wenn ein Fluss durch Augmentierung längs eines flussvergrößernden Pfads P verbessert wird, so enthält P immer mindestens eine *kritische Kante* e, d.h., der Fluss durch e wird entweder auf $c(e)$ angehoben oder auf 0 abgesenkt.

Nun zum **Beweis des Satzes von Edmonds und Karp**: Mit

$$f_0, \ldots, f_r$$

seien wie zuvor die im Laufe des Algorithmus von Edmonds und Karp produzierten Flüsse bezeichnet. Mit

$$P_0, \ldots, P_{r-1}$$

bezeichnen wir die dazugehörigen flussvergrößernden Pfade, d.h., P_k ist der flussvergrößernde Pfad, der beim Übergang von f_k zu f_{k+1} benutzt wird ($k = 0, \ldots, r-1$).

Für ein $k \in \{0, \ldots, r-1\}$ sei $e = (u, v)$ eine kritische Kante von P_k. Man beachte: Aufgrund der Regel, dass immer kürzeste flussvergrößernde Pfade gewählt werden, besitzt P_k genau

$$x_v(k) + y_v(k) = x_u(k) + y_u(k)$$

Kanten.

Wir setzen nun voraus, dass e ein weiteres Mal in einem augmentierenden Pfad vorkommt, sagen wir in P_h für $h > k$. Dabei sei h so klein wie möglich gewählt, d.h., wir betrachten *das nächste Mal*, dass e in einem flussvergrößernden Pfad auftritt. Es folgt (Wieso nämlich?):

War e in P_k eine Vorwärtskante, so ist e in P_h eine Rückwärtskante; und umgekehrt.

Wir betrachten den Fall, dass e in P_k eine Vorwärtskante war: In diesem Fall gilt

$$x_v(k) = x_u(k) + 1 \quad \text{und} \quad x_u(h) = x_v(h) + 1.$$

Aufgrund von Lemma 1 und 2 gilt $x_v(h) \geq x_v(k)$ und $y_u(h) \geq y_u(k)$. Es folgt

$$x_u(h) + y_u(h) = x_v(h) + 1 + y_u(h) \geq x_v(k) + 1 + y_u(k) = x_u(k) + y_u(k) + 2.$$

Dies bedeutet: *P_h besitzt mindestens zwei Kanten mehr als P_k.*

Dasselbe gilt auch im Fall, dass e in P_k eine Rückwärtskante war: Um dies zu erkennen, braucht man nur ebenso wie zuvor zu argumentieren – allerdings mit vertauschten Rollen für u und v.

Fazit. Da kein flussvergrößernder Pfad mehr als $|V| - 1 = n - 1$ Kanten enthalten kann, gilt für jede Kante e:

e kann höchstens $\dfrac{n-1}{2}$ mal als kritische Kante in einem der Pfade P_k auftreten.

Somit kann es höchstens $m \cdot \frac{n-1}{2}$ Flussvergrößerungen geben. □

9.5 Der Begriff des Residualgraphen

In diesem Abschnitt wird der Begriff des *Residualgraphen* vorgestellt – ein Begriff, der besonders nützlich ist, wenn es um etwas komplexere Netzwerk-Fluss-Algorithmen geht. Statt „Residualgraph“ sagt man auch *Restgraph*.

Mit $G = (V, E)$ bezeichnen wir wie bislang einen schlingenlosen Digraphen; zur Definition des Begriffs „schlingenloser Digraph“ siehe Seite 107. Man beachte: Aufgrund dieser Definition sind in G nicht nur Schlingen ausgeschlossen, sondern auch parallele Kanten; antiparallele Kanten (wie in der nachfolgenden Zeichnung) sind hingegen zugelassen.

Ein Paar von antiparallelen Kanten.

Bevor wir uns dem Begriff des Residualgraphen widmen, definieren wir zunächst einen zu G gehörigen Graphen G^+, der aus G dadurch entsteht, dass man zu jeder Kante e von G eine neue Kante hinzunimmt, die man *Rückwärtskante* von e nennt und mit e^{rev} bezeichnet.

Hier ist die genaue Definition:

Definition.

Gegeben sei ein schlingenloser Digraph $G = (V, E)$ mit m Kanten. Zu G definieren wir G^+ wie folgt:

(i) Die Knotenmenge von G^+ sei gleich V, d.h., G und G^+ besitzen dieselbe Knotenmenge.

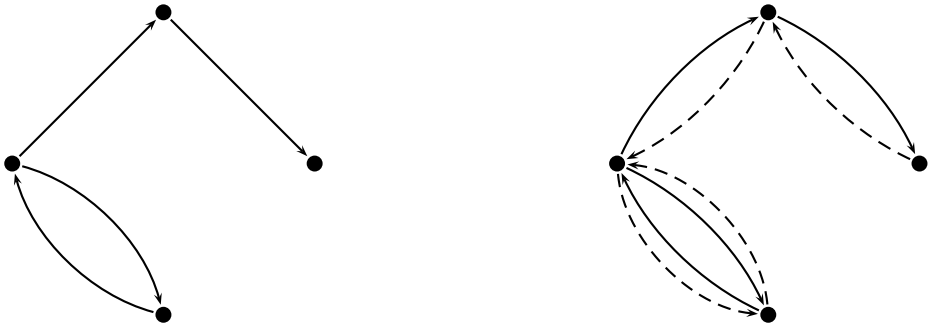
(ii) G^+ enthält sämtliche Kanten von G.

(iii) *Darüber hinaus gebe es in G^+ noch m weitere Kanten*: Zu jeder Kante $e = (x, y)$ von G enthalte G^+ eine neue Kante, die wir mit e^{rev} bezeichnen. Dabei gelte:

- Ist die Kante (y, x) nicht in G vorhanden, so sei $e^{\text{rev}} = (y, x)$.
- Ist die Kante (y, x) bereits in G vorhanden, so sei e^{rev} eine zusätzliche Kante, die ebenfalls von y nach x führt.

Insbesondere gilt also: G hat m Kanten $\Rightarrow$ G^+ hat $2m$ Kanten.

Beispiel. Es sei $G = (V, E)$ der links abgebildete Graph. Der dazugehörige Graph G^+ ist rechts dargestellt, wobei die neuen Kanten gestrichelt gezeichnet sind.



G $\qquad\qquad$ G^+

Man beachte: In G gibt es keine parallelen Kanten; in G^+ kann es jedoch wie im vorangegangenen Beispiel *parallele Kanten* geben. Solche Kanten treten allerdings nur auf, wenn es in G antiparallele Kanten gibt. Anders gesagt:

Sind in G keine antiparallelen Kanten vorhanden, so gibt es in G^+ keine parallelen Kanten.

Ein Wort zur Terminologie: Graphen, in denen parallele Kanten vorkommen (oder vorkommen dürfen), werden auch als *Multigraphen* bezeichnet. In anderen Fällen kann es durchaus wichtig sein, dass man sprachlich genau zwischen „Graphen“ und „Multigraphen“ unterscheidet. Im vorliegenden Kontext ist dies jedoch nicht unbedingt nötig: Auch in Fällen, in denen parallele Kanten zugelassen sind, werden wir von *Graphen* sprechen.

Mit E^{rev} bezeichnen wir die Menge der neu hinzugefügten Kanten. Es gilt also

$$E^{\text{rev}} = \Big\{ e^{\text{rev}} : e \in E \Big\}.$$

Die Kantenmenge von G setzt sich also aus zwei disjunkten Mengen zusammen: E (ursprüngliche Kanten) und E^{rev} (neue Kanten).

Es folgt die Definition des Residualgraphen G_f sowie des dazugehörigen Begriffs der Residualkapazität.

Definition.

Gegeben sei ein Netzwerk $N = (G, c, s, t)$ mit $G = (V, E)$ sowie ein Fluss f auf N. Der *Residualgraph* G_f wird wie folgt definiert:

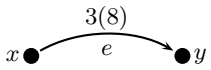
(i) G_f ist ein Teilgraph von G^+; die Knotenmenge von G_f ist V.

(ii) Für jede Kante $e \in E$ gilt
- Ist $f(e) < c(e)$ erfüllt, so wird e in G_f aufgenommen und außerdem $c_f(e) := c(e) - f(e)$ gesetzt.
- Ist $f(e) > 0$ erfüllt, so wird e^{rev} in G_f aufgenommen und $c_f(e^{\text{rev}}) := f(e)$ gesetzt.

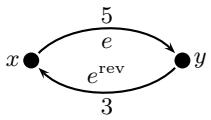
Weitere Kanten werden nicht in G_f aufgenommen. Die in (ii) auftretenden Größen $c_f(e)$ und $c_f(e^{\text{rev}})$ nennt man die *Residualkapazität* von e bzw. e^{rev}.

Man beachte: Liegt der Fall $0 < f(e) < c(e)$ vor, so wird sowohl e als auch e^{rev} in G_f aufgenommen. Diese Tatsache wird in der folgenden Darstellung illustriert.

Eine Kante $e = (x, y)$ in G mit $f(e) = 3$ und $c(e) = 8$:



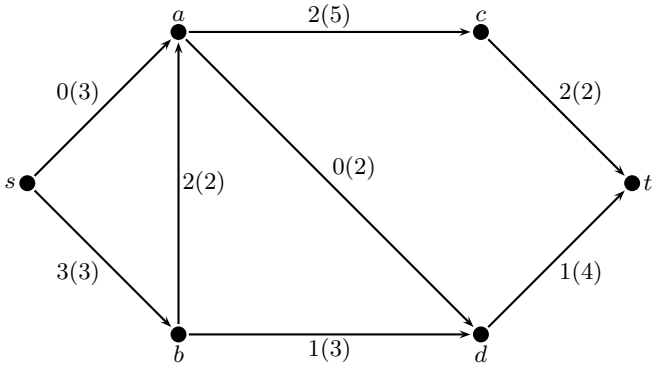
Die dazugehörigen Kanten in G_f besitzen die Residualkapazitäten 5 und 3:



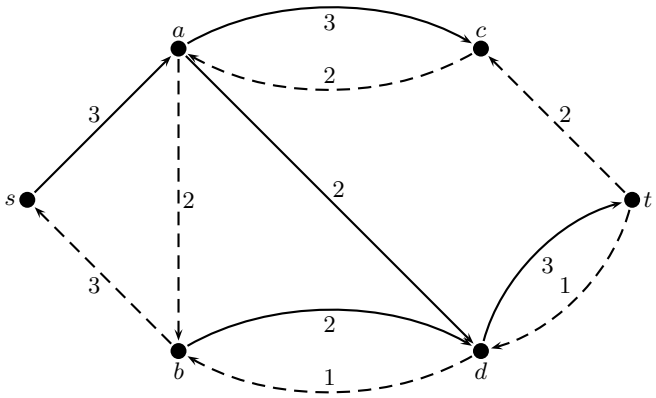
Die Interpretation der Residualkapazitäten liegt auf der Hand:

- Für $e \in E$ gibt $c_f(e)$ an, um wie viel der Fluss in e gegebenenfalls noch erhöht werden kann.
- Für $e \in E$ gibt $c_f(e^{\text{rev}})$ an, um wie viel der Fluss in e gegebenenfalls erniedrigt werden kann.

Beispiel.



Ein Netzwerk $N = (G, c, s, t)$ mit Fluss f.



Der dazugehörige Residualgraph G_f mit Residualkapazitäten (Kanten aus E^{rev}: gestrichelt).

In unserem Beispiel entspricht jedem f-vergrößernden Pfad in G auf naheliegende Weise ein s, t-Pfad in G_f: Beispielsweise ist

$$P : s, a, b, d, t$$

ein f-vergrößernder Pfad in G. Diesem Pfad entspricht in G_f ein s,t-Pfad mit derselben Knotenfolge. Der einzige Unterschied:

$$\textit{Die Rückwärtskante } e = (b,a) \textit{ wurde durch } e^{rev} \textit{ ersetzt.}$$

Es sei nun ein beliebiges Netzwerk $N = (G, c, s, t)$ und ein Fluss f auf N gegeben. Wie im Beispiel gilt dann: Jedem f-vergrößernden Pfad P in G entspricht in G_f ein s,t-Pfad mit derselben Knotenfolge, der aus P dadurch entsteht, dass man jede Rückwärtskante e von P durch e^{rev} ersetzt.

Umgekehrt: Ist in G_f ein s,t-Pfad ohne Knotenwiederholungen gegeben, so entspricht diesem Pfad auf analoge Weise ein f-vergrößernder Pfad in G.

Aufgrund der beschriebenen, einfach zu durchschauenden Beziehung zwischen f-vergrößernden Pfaden in G und s,t-Pfaden in G_f ergibt sich eine **alternative Möglichkeit**, zum Begriff *f-vergrößernder Pfad* zu gelangen:

> Man wählt einen etwas anderen Aufbau und *definiert* einen f-vergrößernden Pfad als einen s,t-Pfad ohne Knotenwiederholungen im Residualgraphen G_f.

Wegen der zuvor beschriebenen Entsprechung ist diese Möglichkeit, den Begriff „f-vergrößernder Pfad“ zu definieren, *äquivalent* zu unserer Definition in Abschnitt 9.3.

Die angesprochene Möglichkeit (oder ähnliche Möglichkeiten) einen f-vergrößernden Pfad zu definieren, werden Sie häufig in der Literatur antreffen – besonders, wenn es um etwas komplexere Themen geht, wie beispielsweise

- kostenoptimale Flüsse (minimum cost flows),
- Push-Relabel-Algorithmus von Goldberg und Tarjan.

Literatur zu den beiden genannten Themen:

- B. Korte, J. Vygen:

  Combinatorial Optimization. Theory and Algorithms. Springer. 2012. 5. Auflage.

- A. Schrijver:

  Combinatorial Optimization. Polyhedra and Efficiency. Volume A: Paths, Flows, Matchings. Springer. 2003.

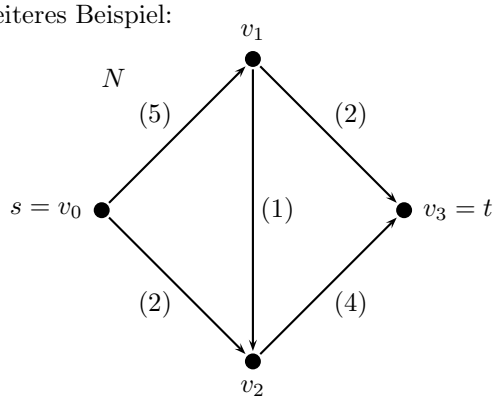
10 Max-Flow und Min-Cut als LP-Probleme

10.1 Ein einführendes Beispiel

Man kann viele Ähnlichkeiten zwischen dem Labelling-Algorithmus und dem Simplexalgorithmus beobachten, beispielsweise diese: Am Schluss, wenn keine Verbesserung mehr möglich ist, erhält man in beiden Fällen eine „Zugabe“; genauer gilt:

- Beim Simplexalgorithmus erhält man zusätzlich eine *optimale Lösung des dualen Problems*;
- Beim Labelling-Algorithmus erhält man neben einem maximalen Fluss auch einen *minimalen Schnitt*.

Diese Ähnlichkeiten hängen natürlich damit zusammen, *dass man das Problem, einen maximalen Fluss in einem Netzwerk zu finden, als ein LP-Problem formulieren kann*. Dies hatten wir in Abschnitt 6.2 bereits anhand eines Beispiels besprochen – in Abschnitt 10.2 wird der allgemeine Fall behandelt werden. Darüber hinaus wollen wir uns auch mit dem zum Maximalflussproblem dualen Problem befassen. Zum Einstieg betrachten wir ein weiteres Beispiel:



Die geklammerten Zahlen an den Kanten bezeichnen die Kapazitäten. *Für jede Kante führen wir eine Variable ein, die die Stärke des Flusses durch diese Kante angibt*: Führt die Kante e von v_i nach v_j, so bezeichnen wir die dazugehörige Variable mit x_{ij}.

Im obigen Beispiel haben wir es also mit den folgenden Variablen zu tun:

$$x_{01}, \quad x_{02}, \quad x_{12}, \quad x_{13}, \quad x_{23}.$$

Das dazugehörige LP-Problem – wir wollen es (P) nennen – lautet:

maximiere $x_{01} + x_{02}$

unter den Nebenbedingungen

$$\begin{aligned}
x_{01} - x_{12} - x_{13} &= 0 \\
x_{02} + x_{12} - x_{23} &= 0 \\
x_{01} &\leq 5 \\
x_{02} &\leq 2 \\
x_{12} &\leq 1 \\
x_{13} &\leq 2 \\
x_{23} &\leq 4 \\
x_{01}, \ldots, x_{23} &\geq 0.
\end{aligned}$$

Wir stellen außerdem das *duale Problem* (D) hierzu auf.

Die ersten beiden Nebenbedingungen von (P) beziehen sich auf die inneren Knoten v_1 und v_2; die dazugehörigen dualen Variablen wollen wir y_1 und y_2 nennen. Es handelt sich um freie Variablen, da die ersten beiden Nebenbedingungen von (P) Gleichungen sind.

Die anschließenden fünf Ungleichungen von (P) beziehen sich auf die fünf Kanten von N. Die dazugehörigen dualen Variablen wollen wir y_{01}, y_{02}, y_{12}, y_{13} und y_{23} nennen.

Das duale Problem (D) lautet wie folgt:

$$
\begin{aligned}
&\text{minimiere} && 5y_{01} + 2y_{02} + y_{12} + 2y_{13} + 4y_{23} \\
&\text{unter den Nebenbedingungen} \\
& && y_1 + y_{01} && && && \geq 1 \\
& && y_2 + y_{02} && && && \geq 1 \\
& && -y_1 + y_2 + y_{12} && && \geq 0 \\
& && -y_1 + y_{13} && \geq 0 \\
& && -y_2 + y_{23} && \geq 0 \\
& && y_{01}, \ldots, y_{23} && \geq 0.
\end{aligned}
$$

10.2 Das primale Problem (Max-Flow)

Nachdem wir in 10.1 anhand eines Beispiels gesehen haben, wie man zu (P) und (D) gelangt, *wollen wir nun für ein beliebiges Flussnetzwerk $N = (G, c, s, t)$ mit $G = (V, E)$ das Entsprechende durchführen.* Zu diesem Zweck benennen wir zunächst die Knoten von N wie folgt:

- v_0 sei die Quelle, d.h., $v_0 = s$;
- $v_1, \ldots, v_n$ seien die inneren Knoten;
- v_{n+1} sei die Senke, d.h., $v_{n+1} = t$.

Für jede Kante (v_i, v_j) führen wir eine Variable x_{ij} ein; die Kapazität der Kante (v_i, v_j) sei mit c_{ij} bezeichnet. Es folgt die *Beschreibung von (P)*:

Zielfunktion: maximiere $\displaystyle\sum_{(v_0, v_j) \in E} x_{0j}$

Nebenbedingungen:

(I) Für jeden inneren Knoten v_i gibt es eine Nebenbedingung, die wie folgt lautet[1]:

$$\sum_{(v_j, v_i) \in E} x_{ji} - \sum_{(v_i, v_j) \in E} x_{ij} = 0. \tag{10.1}$$

(II) Für jede Kante $(v_i, v_j) \in E$ gibt es die Nebenbedingung $x_{ij} \leq c_{ij}$.

(III) Für jede Variable x_{ij} gilt die Nichtnegativitätsbedingung $x_{ij} \geq 0$.

Damit ist (P) aufgestellt. Es gibt in (P) also – abgesehen von den Nichtnegativitätsbedingungen – zwei Typen von Nebenbedingungen

(I) die *Knotenbedingungen*,

(II) die *Kapazitätsbedingungen*.

Durch die Aufstellung von (P) haben wir gesehen, *dass man das Problem, einen maximalen Fluss in einem Netzwerk zu finden, als ein LP-Problem formulieren kann.*

[1] Man beachte: In (10.1) ist der Index i fest gewählt. Die Formel besagt, dass in v_i ebenso viel hinein wie hinaus fließt.

Feststellung 1.
Beim Maximalflussproblem handelt es sich um ein spezielles LP-Problem.

10.3 Das duale Problem (Min-Cut)

Nun wenden wir das *Dualisierungsrezept* auf das in 10.2 aufgestellte Problem (P) an (vgl. Seite 86): Zu jedem inneren Knoten v_i gehört dann eine duale Variable y_i, und zu jeder Kante $(v_i, v_j) \in E$ haben wir eine duale Variable y_{ij}. Unter Benutzung dieser Bezeichnungen stellen wir im Folgenden das duale Problem (D) auf. Dabei wird sich herausstellen, dass es einen sehr engen Zusammenhang zwischen minimalen Schnitten (S, T) von N und optimalen Lösungen von (D) gibt. Unser Ergebnis lässt sich so aussprechen: *Ein minimaler Schnitt (S, T) von N stellt eine optimale Lösung von (D) dar.*

Es folgt die *Beschreibung des zu (P) dualen Problems (D)*:

Zielfunktion: minimiere $\displaystyle\sum_{(v_i,v_j)\in E} c_{ij} y_{ij}$

Nebenbedingungen: Zu jeder Kante $(v_i, v_j) \in E$ gehört genau eine Nebenbedingung. Im Einzelnen gilt Folgendes:

(I) Gilt $i = 0$ und $j \leq n$, d.h., (v_i, v_j) führt von der Quelle zu einem inneren Knoten, so lautet die dazugehörige Nebenbedingung

$$y_j + y_{0j} \geq 1 \qquad \text{bzw.} \qquad y_{0j} \geq 1 - y_j.$$

(II) Gilt $1 \leq i \leq n$ und $1 \leq j \leq n$, d.h., (v_i, v_j) verbindet zwei innere Knoten, so lautet die dazugehörige Nebenbedingung

$$-y_i + y_j + y_{ij} \geq 0 \qquad \text{bzw.} \qquad y_{ij} \geq y_i - y_j.$$

(III) Gilt $i \geq 1$ und $j = n + 1$, d.h., (v_i, v_j) führt von einem inneren Knoten zur Senke, so lautet die dazugehörige Nebenbedingung

$$-y_i + y_{in+1} \geq 0 \qquad \text{bzw.} \qquad y_{in+1} \geq y_i.$$

(IV) Gilt $i = 0$ und $j = n + 1$, d.h., es liegt der Fall vor, dass (v_i, v_j) direkt von der Quelle zur Senke führt, so lautet die dazugehörige Nebenbedingung

$$y_{0n+1} \geq 1.$$

(V) Für die Variablen y_{ij} gilt $y_{ij} \geq 0$, während die y_i freie Variablen sind.

Es ist möglich, die vier verschiedenen Typen (I)-(IV) von Nebenbedingungen in einheitlicher Form zu schreiben. Zu diesem Zweck definieren wir zusätzlich y_0 und y_{n+1}, indem wir festlegen:

$$y_0 = 1 \qquad \text{und} \qquad y_{n+1} = 0.$$

Dann gilt *in jedem der Fälle (I)-(IV)*[2]: Die zu $(v_i, v_j) \in E$ gehörige Nebenbedingung lautet

$$y_{ij} \geq y_i - y_j.$$

[2] Prüfen Sie dies nach!

Wir können das zum Problem des maximalen Flusses duale Problem (D) also auch so schreiben:

$$\begin{aligned}
&\text{minimiere} \sum_{(v_i,v_j)\in E} c_{ij}y_{ij} \\
&\text{unter den Nebenbedingungen} \\
&\qquad y_{ij} \geq y_i - y_j \qquad \text{für alle } (v_i,v_j)\in E \qquad (10.2)\\
&\qquad y_0 = 1 \\
&\qquad y_{n+1} = 0 \\
&\qquad y_{ij} \geq 0 \qquad \text{für alle } (v_i,v_j)\in E
\end{aligned}$$

Um in einem Netzwerk einen maximalen Fluss zu bestimmen, könnte man auch mit dem *Simplexalgorithmus* arbeiten. Das tut man jedoch in der Regel nicht, sondern man verwendet Algorithmen, die – wie der Labelling-Algorithmus – von der speziellen Struktur des Maximalflussproblems Gebrauch machen. (Der Simplexalgorithmus ist ja eher eine Art Allzweckwaffe, mit der man *jedes* LP-Problem behandeln kann.)

Der Simplexalgorithmus liefert immer auch eine optimale Lösung des dualen Problems. *Lässt sich dies auch vom Labelling-Algorithmus sagen?*

Die Antwort auf diese Frage lautet ja!

Weshalb ist das so? Nun, der Labelling-Algorithmus liefert – wie wir wissen – einen minimalen Schnitt, *und aus einem minimalen Schnitt lässt sich (wie wir sehen werden) auf eine sehr einfache und direkte Weise eine optimale Lösung des dualen Problems gewinnen.*

Wir können also festhalten:

Feststellung 2.

Der Labelling-Algorithmus liefert neben einem maximalen Fluss einen minimalen Schnitt – und somit auch eine optimale Lösung des zum Maximalflussproblem dualen Problems.

Wie man aus einem minimalen Schnitt eine optimale Lösung des dualen Problems gewinnt, soll nun beschrieben werden. Zu diesem Zweck betrachten wir zunächst einen *beliebigen* Schnitt (S,T) von N, d.h., (S,T) kann (aber muss nicht) minimal sein.

Zu (S,T) definieren wir eine dazugehörige zulässige Lösung von (D) wie folgt[3]:

Wir setzen

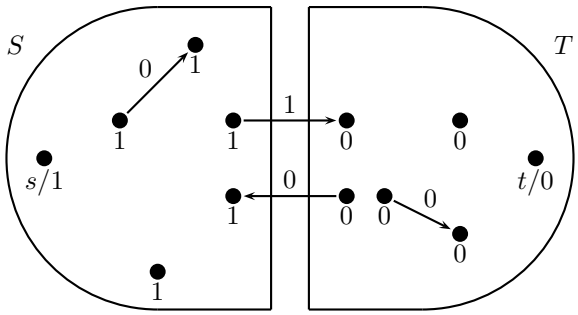
$$y_i = \begin{cases} 1, & \text{falls } v_i \in S; \\ 0, & \text{falls } v_i \in T \end{cases}$$

sowie

$$y_{ij} = \begin{cases} 1, & \text{falls } (v_i,v_j) \in (S,T); \\ 0, & \text{sonst.} \end{cases}$$

[3]Mit (D) ist hier und im Folgenden natürlich das zum Maximalflussproblem duale Problem gemeint; (D) sei wie in (10.2) dargestellt.

Diese Festlegung der dualen Variablen ist in der folgenden Skizze dargestellt:



Man überprüft leicht, dass dies eine zulässige Lösung von (D) ist, für die darüber hinaus gilt:

$$\begin{gathered}\textit{Der Zielfunktionswert } \sum_{(v_i,v_j)\in E} c_{ij}y_{ij} \textit{ für diese} \\ \textit{zulässige Lösung von (D) ist gleich } c(S,T).\end{gathered} \tag{10.3}$$

Die Feststellung (10.3) gilt für jeden beliebigen Schnitt (S,T) von N. Ist (S,T) nun ein *minimaler* Schnitt, so gilt nach dem Max-Flow Min-Cut Theorem $c(S,T) = w(f^*)$, wobei f^* einen Maximalfluss von N bezeichnet. Für einen minimalen Schnitt können wir (10.3) also auch so aussprechen:

$$\begin{gathered}\textit{Der Zielfunktionswert für eine zulässige Lösung von (D), die (wie beschrieben)} \\ \textit{von einem minimalen Schnitt } (S,T) \textit{ herrührt, ist gleich } w(f^*).\end{gathered} \tag{10.4}$$

Es folgt, dass die zulässige Lösung von (D), von der in (10.4) die Rede ist, eine optimale Lösung von (D) sein muss. (Denn: f^* ist eine zulässige Lösung des zu (D) gehörigen primalen Problems (P) und außerdem besitzt f^* den Zielfunktionswert $w(f^*)$ – ebenso wie die in (10.4) beschriebene zulässige Lösung von (D).)

Damit ist gezeigt, dass zu einem minimalen Schnitt (S,T) immer eine optimale Lösung von (D) dazugehört, die man (wie beschrieben und in der Zeichnung dargestellt) auf eine ganz einfache Art aus (S,T) erhalten kann.

11 Matchings und Knotenüberdeckungen in bipartiten Graphen

In diesem Abschnitt geht es größtenteils um eine Anwendung der Ergebnisse über Maximalflüsse auf *ungerichtete* Graphen.

11.1 Grundbegriffe

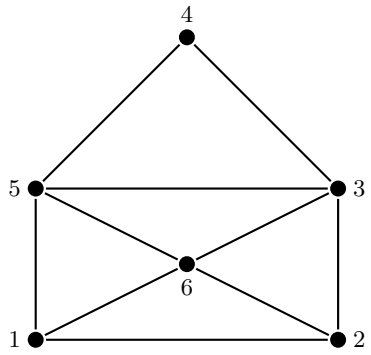
Wir wiederholen zunächst einmal die Definition eines ungerichteten Graphen, die bereits aus den Grundvorlesungen bekannt ist.

Definition.
Ein *ungerichteter Graph* G ist ein Paar (V, E), wobei V eine beliebige endliche Menge und E eine Teilmenge der Menge aller zweielementigen Teilmengen von V ist. Man nennt V die *Knotenmenge* und E die *Kantenmenge* von G.

Beispiel.

$$V = \Big\{1, 2, 3, 4, 5, 6\Big\}$$
$$E = \Big\{\{1,2\},\ \{2,3\},\ \{3,4\},\ \{4,5\},\ \{1,5\},\ \{3,5\},\ \{1,6\},\ \{2,6\},\ \{3,6\},\ \{5,6\}\Big\}$$

Den Graphen dieses Beispiels kann man wie folgt darstellen:

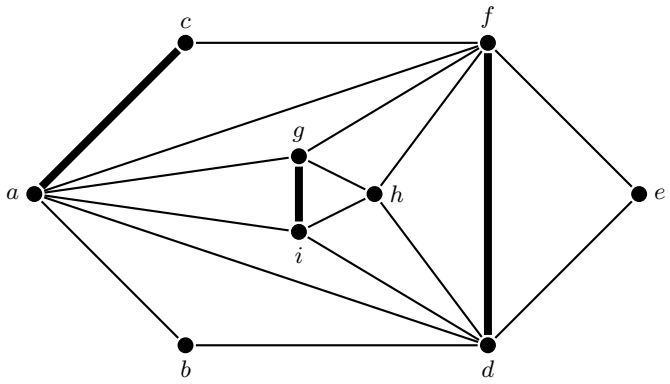


Wir setzen die in „Mathematik I (DM)“ behandelten Grundbegriffe über Graphen und die dort verwendeten Schreibweisen als bekannt voraus.

In der Überschrift dieses Abschnitts kommen die beiden Begriffe „Matching“ und „bipartiter Graph“ vor. Im Folgenden werden diese beiden Begriffe definiert und erläutert.

Definition.
Es sei $G = (V, E)$ ein (ungerichteter) Graph[1]. Eine Teilmenge M von E wird ein *Matching* von G genannt, falls je zwei verschiedene Kanten von M niemals einen Knoten gemeinsam haben.

In der folgenden Skizze wird ein Graph G und ein Matching M von G dargestellt:



Die Kanten von M wurden fett gezeichnet. M besteht aus drei Kanten:

$$M = \Big\{\{a,c\},\ \{g,i\},\ \{d,f\}\Big\}.$$

Können Sie ein Matching mit mehr als drei Kanten angeben?

Zur Übung: Geben Sie einen zusammenhängenden Graphen G mit 10 Kanten an, für den gilt: Es gibt in G ein Matching M mit $|M| = 2$, aber ein Matching mit mehr als 2 Kanten gibt es nicht.

Definition.
Die *Matchingzahl* $m(G)$ eines Graphen G wird definiert durch:

$$m(G) = \max\Big\{|M| : M \text{ ist ein Matching von } G\Big\}.$$

Für den oben abgebildeten Graphen G mit $V(G) = \{a, \ldots, i\}$ gilt $m(G) = 4$.

Unter dem *Matching-Problem* versteht man das Problem, für einen gegebenen Graphen G ein Matching M mit größtmöglicher Kantenzahl zu finden; mit anderen Worten: Gesucht ist ein Matching M, für das

$$|M| = m(G)$$

gilt.

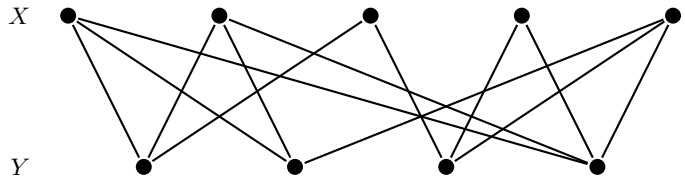
Wir werden das *Matching-Problem für bipartite Graphen* behandeln. Zunächst einmal ist die noch ausstehende Definition eines bipartiten Graphen zu geben.

Definition.
Ein Graph $G = (V, E)$ heißt *bipartit*, falls seine Knotenmenge V in zwei disjunkte Teilmengen X und Y zerlegt werden kann, so dass jede Kante von G einen Knoten aus X mit einem Knoten aus Y verbindet.

Kanten sollen also immer nur zwischen den Mengen X und Y verlaufen, aber *niemals innerhalb von X oder innerhalb von Y*.

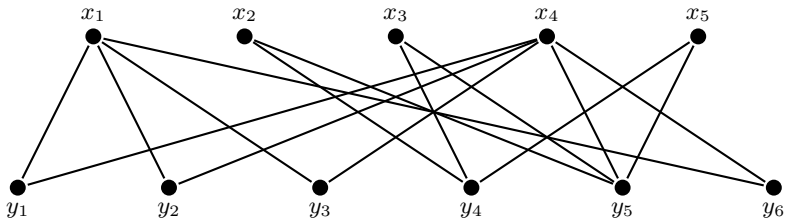
[1] Wir wollen in Kapitel 11 die folgende *Konvention* verwenden: Liegt ein ungerichteter Graph vor, so kann das Adjektiv „ungerichtet“ auch wegfallen; liegt hingegen ein gerichteter Graph vor, so soll das Adjektiv „gerichtet“ nicht weggelassen werden.

Hier ein **Beispiel** eines bipartiten Graphen:



Bipartite Graphen sowie Matchings in bipartiten Graphen treten besonders häufig in Situationen auf, in denen es um die Zuordnung von Personen oder Objekten zu anderen Personen oder Objekten geht – wie etwa im nachfolgenden Problem.

Gegeben seien m Personen und n Jobs. Die Personen seien mit $x_1, \ldots, x_m$ und die Jobs mit $y_1, \ldots, y_n$ bezeichnet. Ist eine Person x_i für einen Job y_j geeignet, so ziehen wir eine Kante von x_i nach y_j, wie beispielsweise im folgenden Graphen:



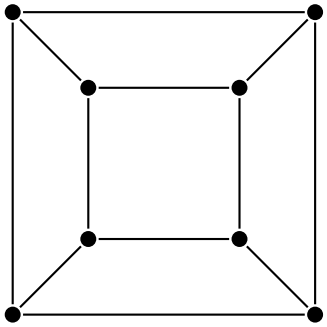
Die Aufgabe ist nun, möglichst vielen Personen einen Job zu geben, wobei Jobs jedoch nur mit geeigneten Personen zu besetzen sind; außerdem soll kein Job mehrfach vergeben werden und keine Person soll mehr als einen Job erhalten.

Perfekt wäre es natürlich, wenn jede Person einen Job bekäme und auch kein Job unbesetzt bliebe. In diesem Fall spricht man von einem *perfekten Matching*. Auch wenn es um nicht-bipartite Graphen geht, spricht man von perfekten Matchings.

Der Deutlichkeit halber sei noch einmal die genaue Definition gegeben.

Definition.
Es sei $G = (V, E)$ ein Graph. Ein Matching M von G heißt *perfekt*, falls es zu jedem Knoten v von G eine Kante $e \in M$ gibt, die v trifft.

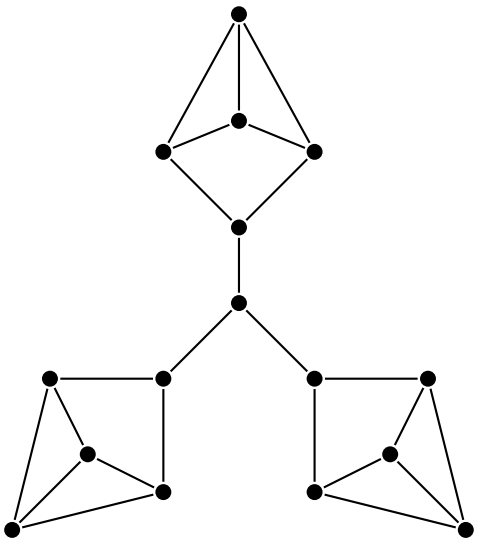
Zur Illustration ein Graph G, der ein perfektes Matching besitzt:



Klar ist: Besitzt ein Graph $G = (V, E)$ ein perfektes Matching, so ist $|V|$ eine gerade Zahl und es gilt

$$m(G) = \frac{|V|}{2}.$$

Zur Übung: Besitzt der folgende Graph ein perfektes Matching?



11.2 Ein Algorithmus zur Lösung des Matching-Problems für bipartite Graphen

Nach diesem einführenden Teil kommen wir nun wie angekündigt zum *Matching-Problem für bipartite Graphen*. Wir folgen dabei zum Teil der Darstellung im folgenden Lehrbuch:

- Jon Kleinberg, Éva Tardos: *Algorithm Design*. Pearson (2006).

11.2.1 Das Problem

Eingabe: ein bipartiter Graph $G = (V, E)$ mit zugehöriger Knotenpartition $V = X \cup Y$.

Gesucht: ein Matching von G mit maximaler Anzahl von Kanten.

Wir setzen im Folgenden stets voraus, dass der betrachtete bipartite Graph G, für den ein Matching mit maximaler Kantenzahl gefunden werden soll, keine Knoten vom Grad 0 („*isolierte Knoten*“) besitzt; dies ist möglich, da isolierte Knoten für das Matching-Problem offenbar keine Rolle spielen.

11.2.2 Der Algorithmus

Es soll der *Netzwerk-Fluss-Algorithmus* von Edmonds und Karp angewendet werden, um das Matching-Problem für bipartite Graphen zu lösen. Netzwerk-Fluss-Algorithmen lassen sich auf *gerichtete* Graphen anwenden, beim Matching-Problem geht es jedoch um *ungerichtete* Graphen; außerdem brauchen wir eine *Quelle* s, eine *Senke* t und *Kapazitäten*.

Dies alles stellt keine Schwierigkeit dar, da wir einen gegebenen bipartiten Graphen in ein passendes Netzwerk überführen können.

Es sei $G = (V, E)$ ein bipartiter Graph mit Knotenpartition $V = X \cup Y$. Wir bilden auf folgende Art ein Flussnetzwerk:

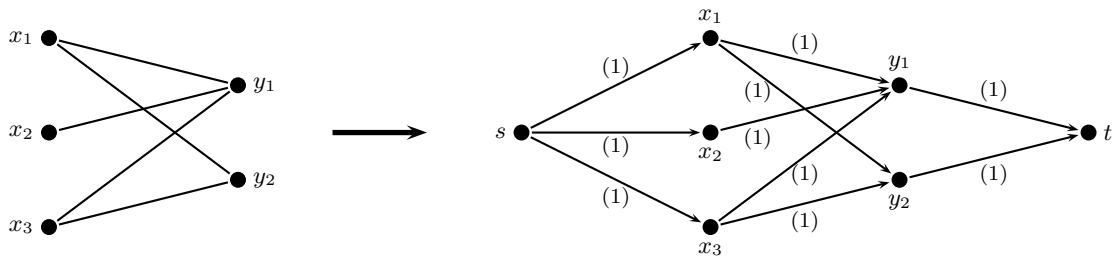
- Alle Kanten von G werden von X nach Y gerichtet.
- Wir fügen zwei neue Knoten s und t hinzu und verbinden s mit jedem Knoten $x \in X$ durch die (gerichtete) Kante (s, x); analog: Zu jedem $y \in Y$ wird die Kante (y, t) hinzugefügt.
- Jede Kante erhält die Kapazität 1.

Den so aus G entstandenen gerichteten Graphen wollen wir G' nennen; das entstandene Netzwerk ist also

$$N = \big(G', c, s, t\big)$$

mit $c(e) = 1$ für alle Kanten e von G'.

Illustration dieser Konstruktion:



Der angekündigte Algorithmus zur Berechnung eines Matchings von G mit maximaler Kantenzahl ist sehr einfach: Er besteht lediglich darin, dass man mit dem Algorithmus von Edmonds und Karp einen maximalen Fluss f^* von N berechnet. Wir werden sehen (siehe Analyse des Algorithmus), dass es ebenfalls ganz einfach ist, aus dem erhaltenen Maximalfluss f^* das gewünschte Matching zu gewinnen.

11.2.3 Analyse des Algorithmus

Wir erinnern uns zunächst an eine *Besonderheit des Netzwerk-Fluss-Problems*, die sich in unserem Zusammenhang als entscheidend erweist:

> Setzt man – wie wir es immer gemacht haben – voraus, dass alle Kapazitäten ganzzahlig sind, *so gibt es auch immer einen ganzzahligen Maximalfluss: Der Ford-Fulkerson-Algorithmus liefert bei ganzzahligen Kapazitäten auch immer einen ganzzahligen Maximalfluss.* (⋆)

Wenn Sie sich noch einmal von der Richtigkeit der Feststellung (⋆) überzeugen wollen: Ein kurzer Blick auf Seite 118 sollte genügen. Da die in (⋆) getroffenen Feststellungen für den Ford-Fulkerson-Algorithmus gelten, gelten sie natürlich ebenfalls für den Algorithmus von Edmonds und Karp.

Nun zur **Analyse unseres Matching-Algorithmus**: *Diese basiert auf dem Nachweis, dass sich Matchings in G und ganzzahlige Flüsse in $N = (G', c, s, t)$ auf eine leicht durchschaubare Art entsprechen.*

(I) Nehmen wir zunächst an, dass ein Matching M von G gegeben ist, das – sagen wir – aus k Kanten $\{x_{i_1}, y_{i_1}\}, \ldots, \{x_{i_k}, y_{i_k}\}$ besteht. Dann gehört zu M ein ganzzahliger Fluss f auf N, der wie folgt definiert wird:

$$f(s, x_{i_j}) = f(x_{i_j}, y_{i_j}) = f(y_{i_j}, t) = 1 \quad (j = 1, \ldots, k),$$
$$f(e) = 0 \quad \text{sonst.}$$

Man erkennt sofort, dass es sich hierbei um einen Fluss handelt, dass also (F1) und (F2) aus der Definition eines Flusses erfüllt sind; außerdem gilt $w(f) = k$.

(II) Nun wollen wir umgekehrt annehmen, dass ein ganzzahliger Fluss f auf N gegeben ist, für den $w(f) = k$ gilt. Aufgrund von (F1) gilt dann

$$0 \leq f(e) \leq c(e) = 1$$

für alle Kanten e von G'. Hieraus folgt wegen der Ganzzahligkeit von f, dass für jede Kante e von G'

$$f(e) = 0 \quad \text{oder} \quad f(e) = 1$$

gilt. Wir definieren nun M' als die Menge derjenigen (gerichteten) Kanten e von G', für die gilt: e führt von X nach Y und es gilt $f(e) = 1$.

Im Buch von Kleinberg und Tardos werden drei einfache Fakten über die Menge M' bewiesen. Wir schauen uns diese besonders wichtigen Feststellungen im englischsprachigen Original an:

(i) *M' contains k edges.*

Proof. To prove this, consider the cut (A, B) in G' with $A = \{s\} \cup X$. The value of the flow is the total flow leaving A, minus the total flow entering A[2]. The first of these terms is simply the cardinality of M', since these are the edges leaving A that carry flow, and each carries exactly one unit of flow. The second of these terms is 0, since there are no edges entering A. Thus, M' contains k edges. □

(ii) *Each node in X is the tail of at most one edge in M'.*

Proof. To prove this, suppose $x \in X$ were the tail of at least two edges in M'. Since our flow is integer-valued, this means that at least two units of flow leave from x. By conservation of flow, at least two units of flow would have to come into x – but this is not possible, since only a single edge of capacity 1 enters x. Thus x is the tail of at most one edge in M'. □

By the same reasoning, we can show

(iii) *Each node in Y is the head of at most one edge in M'.*

Bei den Kanten von M' handelt es sich um gerichtete Kanten, die von X nach Y führen. Mit M wollen wir die dazugehörige Menge von ungerichteten Kanten bezeichnen. Aufgrund von (i), (ii) und (iii) gilt dann: *M ist ein Matching von G mit $|M| = k$.*

Zusammenfassung von (I) und (II): In (I) haben wir gesehen, dass es zu jedem Matching M von G mit $|M| = k$ einen ganzzahligen Fluss f in N mit $w(f) = k$ gibt. In (II) haben wir erkannt, dass es umgekehrt zu jedem ganzzahligen Fluss f in N mit $w(f) = k$ ein Matching M von G mit $|M| = k$ gibt. Ist f^* ein ganzzahliger Maximalfluss auf N, so gilt also

$$w(f^*) = m(G).$$

Außerdem haben wir in (II) gesehen, wie man ein Matching M von G mit $|M| = m(G)$ erhält, wenn ein ganzzahliger Maximalfluss f^* von N vorliegt: Man hat nichts weiter zu tun, als die Menge M' derjenigen Kanten e von G' zu betrachten, die von X nach Y führen und für die $f^*(e) = 1$ gilt; um M mit $|M| = m(G)$ zu erhalten, braucht man nur noch die Orientierung dieser Kanten wegzulassen.

11.2.4 Bemerkung zur Komplexität

$G = (V, E)$ sei bipartit mit $|V| = n$ und $|E| = m$; isolierte Knoten soll es in G nicht geben.

Beobachtung: Jede Kante in G' hat die Kapazität 1. Die Summe der Kapazitäten der an s stoßenden Kanten ist demnach gewiss nicht größer als n. Für jeden ganzzahligen Maximalfluss f^* von N gilt deshalb

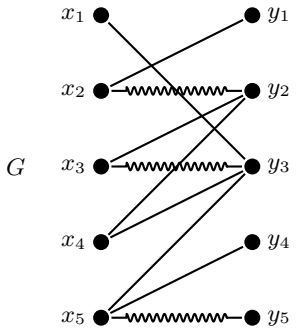
[2] Vgl. Formel (9.2) in Kapitel 9 (Stichwort: *Nettofluss*).

$w(f^*) \leq n$. Folglich benötigt der Algorithmus von Edmonds und Karp nicht mehr als n Iterationen, um einen Maximalfluss in $N = (G', c, s, t)$ zu finden.

Anknüpfend an diese Beobachtung lässt sich zeigen, dass Folgendes gilt (Details siehe Kleinberg/Tardos): Der Algorithmus von Edmonds und Karp kann benutzt werden, um in einem bipartiten Graphen ein Matching maximaler Größe in $O(nm)$ Zeit zu finden.

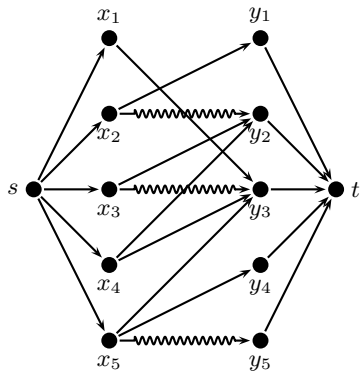
11.3 Alternierende und augmentierende Pfade

Es ist lohnend, sich genauer anzuschauen, wie flussvergrößernde Pfade im Falle des bipartiten Matching-Problems konkret aussehen. Hierzu schauen wir uns den folgenden bipartiten Graphen G an; die Kanten $\{x_2, y_2\}$, $\{x_3, y_3\}$ und $\{x_5, y_5\}$ bilden ein Matching M in G. (Hier und im Folgenden sind Matchingkanten, d.h. Kanten aus M, immer durch Wellenlinien dargestellt.)



Es sei f der ganzzahlige Fluss in $N = (G', c, s, t)$, der M entspricht (vgl. (I) in Abschnitt 11.2.3). Da $|M|$ nicht größtmöglich ist, ist f kein Maximalfluss und folglich muss es einen flussvergrößernden s,t-Pfad geben. Beispielsweise ist der folgende Pfad P' ein flussvergrößernder Pfad in $N = (G', c, s, t)$:

$$P': \quad s,\ x_4,\ y_3,\ x_3,\ y_2,\ x_2,\ y_1,\ t.$$



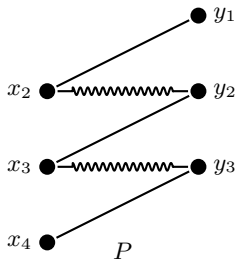
Das G entsprechende Netzwerk $N = (G', c, s, t)$.

Wir wissen: Alle Kapazitäten von N sind gleich 1. Deshalb konnte in der Abbildung, in der N dargestellt wird, auf die Angabe der Kapazitäten verzichtet werden. Außerdem: Für neun gerichtete Kanten e gilt $f(e) = 1$ (Für welche nämlich?); für die übrigen Kanten gilt $f(e) = 0$.

Dem Pfad P' entspricht in G ein ungerichteter Pfad, den wir P nennen wollen.

$$P: \quad x_4,\ y_3,\ x_3,\ y_2,\ x_2,\ y_1.$$

Man beachte, dass sich auf P Kanten, die nicht in M sind, mit Kanten aus M abwechseln:



Dasselbe, nur ein wenig anders formuliert: *Auf P wechseln sich Nicht-Matchingkanten mit Matchingkanten ab.* Außerdem beginnt P in einem Knoten von X, der in G *ungepaart* ist, d.h., es gibt im gesamten Graphen G keine Matchingkante, die an diesen Knoten stößt. Und schließlich gilt: P endet in einem Knoten von Y, der in G ebenfalls ungepaart ist[3].

Vergrößert man nun wie üblich den Fluss f mithilfe von P', so erkennt man, dass dies nichts anderes bedeutet, als das Matching M folgendermaßen mithilfe von P zu vergrößern:

Man entfernt aus M alle Matchingkanten von P und nimmt stattdessen die Nicht-Matchingkanten von P hinzu.

Kurz gesagt: *Man nimmt längs P einen Austausch von Matching- und Nicht-Matchingkanten vor.*

Dass dies so schön funktioniert, liegt natürlich an den drei bereits oben genannten Eigenschaften von P:

(1) Auf P wechseln sich Nicht-Matchingkanten mit Matchingkanten ab;
(2) P beginnt in einem ungepaarten Knoten von X;
(3) P endet in einem ungepaarten Knoten von Y.

Besitzt ein Pfad P die Eigenschaften (1) - (3), so nennt man P einen *augmentierenden Pfad* (engl. *augmenting path*). Wir haben gesehen, wozu augmentierende Pfade gut sind: *Mit ihrer Hilfe lässt sich aus einem gegebenen Matching M ein Matching mit $|M|+1$ Kanten gewinnen.*

Es lässt sich unschwer beweisen, dass einem flussvergrößernden Pfad in $N = (G', c, s, t)$ immer ein augmentierender Pfad in G entspricht (und umgekehrt). Da dies eine wichtige Feststellung ist, halten wir das Gesagte noch einmal fest:

Feststellung 1.
Es sei $G = (V, E)$ ein bipartiter Graph mit Knotenpartition $V = X \cup Y$; es gelte $X = \{x_1, \ldots, x_m\}$ und $Y = \{y_1, \ldots, y_n\}$. Ferner sei $N = (G', c, s, t)$ das zu G gehörige Flussnetzwerk. Es sei ein Matching M von G gegeben und f sei der zu M gehörige ganzzahlige Fluss in N. Dann gilt:

(i) Ist $P' = (s, x_{i_1}, y_{i_1}, \ldots, x_{i_k}, y_{i_k}, t)$ ein flussvergrößernder Pfad in N, so ist $P = (x_{i_1}, y_{i_1}, \ldots, x_{i_k}, y_{i_k})$ ein augmentierender Pfad in G.

(ii) Ist umgekehrt $P = (x_{i_1}, y_{i_1}, \ldots, x_{i_k}, y_{i_k})$ ein augmentierender Pfad in G, so ist $P' = (s, x_{i_1}, y_{i_1}, \ldots, x_{i_k}, y_{i_k}, t)$ ein flussvergrößernder Pfad in N.

Beweis. Wir zeigen (i); der Nachweis von (ii) sei dem Leser überlassen.

Es sei also $P' = (s, x_{i_1}, y_{i_1}, \ldots, x_{i_k}, y_{i_k}, t)$ ein flussvergrößernder Pfad in N. Da (s, x_{i_1}) eine Vorwärtskante von P' ist, gilt $f(s, x_{i_1}) < c(s, x_{i_1}) = 1$, woraus wegen der Ganzzahligkeit von P' folgt, dass $f(s, x_{i_1}) = 0$

[3] Um festzustellen, dass x_4 und y_1 tatsächlich ungepaarte Knoten sind, hat man einen Blick auf die obere Figur von Seite 146 zu werfen – sich nur die Darstellung von P anzuschauen, reicht natürlich nicht aus.

gilt. Dies impliziert, dass x_{i_1} in G ein ungepaarter Knoten ist.

Man beachte, dass Folgendes gilt: Die Kanten (x_{i_j}, y_{i_j}) sind Vorwärtskanten von P'; die Kanten $(y_{i_j}, x_{i_{j+1}})$ sind hingegen Rückwärtskanten. Hieraus ergibt sich $f(x_{i_j}, y_{i_j}) < c(x_{i_j}, y_{i_j}) = 1$ $(j = 1, \ldots, k)$ sowie $f(y_{i_j}, x_{i_{j+1}}) > 0$. Aufgrund der Ganzzahligkeit von f folgt also

$$f(x_{i_j}, y_{i_j}) = 0 \quad (j = 1, \ldots, k) \quad \text{und} \quad f(y_{i_j}, x_{i_{j+1}}) = 1 \quad (j = 1, \ldots, k-1).$$

Für G bedeutet dies, dass die Kanten (x_{i_j}, y_{i_j}) Nicht-Matchingkanten sind, während es sich bei den Kanten $(y_{i_j}, x_{i_{j+1}})$ um Matchingkanten handelt.

Schließlich erhält man (analog zu $f(s, x_{i_1}) = 0$), dass $f(y_{i_k}, t) = 0$ gilt. Dies bedeutet, dass y_{i_k} in G ein ungepaarter Knoten ist.

Insgesamt haben wir wie behauptet erhalten, dass P ein augmentierender Pfad in G ist. □

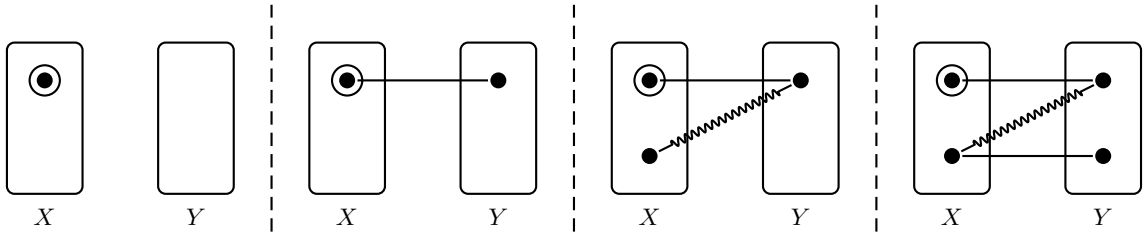
Neben dem Begriff des augmentierenden Pfads spielt der Begriff eines

alternierenden Pfads

eine zentrale Rolle.

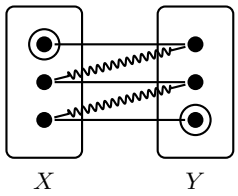
Definition.

Gegeben sei ein bipartiter Graph $G = (V, E)$ mit dazugehöriger Knotenpartition $V = X \cup Y$. Außerdem sei ein Matching M von G gegeben. Ein Pfad P in G wird *alternierender Pfad* (engl. *alternating path*) genannt, wenn er in einem ungepaarten Knoten x von X beginnt und wenn Folgendes gilt: P besteht entweder nur aus x oder auf P wechseln sich Nicht-Matchingkanten mit Matchingkanten ab.



Die Zeichnung stellt alternierende Pfade der Längen 0, 1, 2 und 3 dar; ungepaarte Knoten sind durch ◉ gekennzeichnet und Matchingkanten durch Wellenlinien.

Man beachte, dass jeder augmentierende Pfad auch ein alternierender Pfad ist: *Augmentierende Pfade sind also spezielle alternierende Pfade.* Die nachfolgende Zeichnung stellt einen augmentierenden Pfad der Länge 5 dar:



In Feststellung 1 haben wir festgehalten, dass sich augmentierende Pfade in G und flussvergrößernde Pfade in N auf eine ganz einfache Art entsprechen. *Eine analoge Feststellung lässt sich auch für alternierende Pfade treffen.* Dies wird im Folgenden ausgeführt.

Der Unterschied zwischen den Begriffen „augmentierender Pfad“ und „alternierender Pfad“ besteht darin, dass augmentierende Pfade immer in einem ungepaarten Knoten von Y enden, während alternierende Pfade in einem beliebigen Knoten w von G enden können.

Etwas ganz Ähnliches gilt in N für die Begriffe „flussvergrößernder Pfad“ und „zunehmender Pfad zum Knoten w“. (Zum Begriff „zunehmender Pfad zum Knoten w“: vgl. Seite 117.) Der Unterschied zwischen diesen beiden Begriffen besteht darin, dass flussvergrößernde Pfade immer in t enden, während ein zunehmender Pfad nach w in einem beliebigen Knoten w von N enden kann.

Es gilt nun die folgende Feststellung 2, die man analog zu Feststellung 1 beweist.

Feststellung 2.
Es sei $G = (V, E)$ ein bipartiter Graph mit Knotenpartition $V = X \cup Y$; es gelte $X = \{x_1, \ldots, x_m\}$ und $Y = \{y_1, \ldots, y_n\}$. Ferner sei $N = (G', c, s, t)$ das zu G gehörige Flussnetzwerk. Es sei ein Matching M von G gegeben und f sei der zu M gehörige ganzzahlige Fluss in N. Außerdem sei w ein Knoten aus G. Dann gilt:

(i) Ist $P' = (s, x_{i_1}, \ldots, w)$ ein zunehmender Pfad nach w im Netzwerk N, so ist $P = (x_{i_1}, \ldots, w)$ ein alternierender Pfad in G.

(ii) Ist umgekehrt $P = (x_{i_1}, \ldots, w)$ ein alternierender Pfad in G, so ist $P' = (s, x_{i_1}, \ldots, w)$ ein zunehmender Pfad nach w in N.

Es sei angemerkt, dass in Feststellung 2 auch $x_{i_1} = w$ gelten kann, d.h., der Fall, dass P nur aus einem einzigen Knoten besteht, ist nicht ausgeschlossen.

Wir können die Feststellungen 1 und 2 kurz und knapp wie folgt zusammenfassen:

- Jedem flussvergrößernden Pfad im Netzwerk N entspricht auf natürliche Weise ein augmentierender Pfad in G; und umgekehrt.
- Jedem zunehmenden Pfad in N zu einem Knoten w aus G entspricht in G auf natürliche Weise ein alternierender Pfad nach w; und umgekehrt.

Insbesondere können wir festhalten (Dies folgt unmittelbar aus Feststellung 2!):

Ein Knoten w ist in G genau dann mit einem alternierenden Pfad erreichbar, wenn er in N mit einem zunehmenden Pfad erreichbar ist.

11.4 Knotenüberdeckungen in bipartiten Graphen: der Satz von König

Wir behandeln in diesem Abschnitt den Begriff der *Knotenüberdeckung* in einem Graphen. Knotenüberdeckungen stehen in engem Zusammenhang mit Matchings; liegt ein bipartiter Graph vor, so handelt es sich um den zum Begriff des Matchings „dualen Begriff“: Matching und Knotenüberdeckung bilden für bipartite Graphen ein ähnliches Begriffspaar wie beispielsweise Fluss und Schnitt oder primales und duales Problem.

Es geht darum, alle Kanten eines Graphen durch Knoten zu überdecken; genauer: Die Kanten sollen durch möglichst wenige Knoten überdeckt werden. Es folgt die genaue Definition; man beachte, dass die Definition nicht nur für bipartite Graphen getroffen wird, sondern (allgemeiner) für beliebige Graphen.

Definition.
Es sei $G = (V, E)$ ein Graph. Eine Menge $U \subseteq V$ heißt *Knotenüberdeckung*[4] von G, falls jede Kante von G mit einem Knoten aus U inzidiert.

Anders gesagt: Jede Kante von G soll von mindestens einem Knoten aus U getroffen werden.

Wählt man $U = V$, so ist U offenbar eine Knotenüberdeckung von $G = (V, E)$. Worum es geht: Es soll eine möglichst kleine Knotenüberdeckung U gefunden werden, d.h., $|U|$ *soll minimal sein.*

[4] Engl. *node cover* oder *vertex cover*.

Definition.
Mit $c(G)$ bezeichnen wir die kleinstmögliche Anzahl von Knoten in einer Knotenüberdeckung von G, d.h., wir definieren

$$c(G) = \min\Big\{|U| : U \text{ ist eine Knotenüberdeckung von } G\Big\}.$$

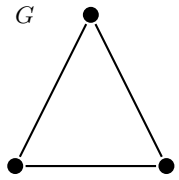
Man nennt $c(G)$ die *Knotenüberdeckungszahl* von G[5].

In jedem Graphen gilt

$$m(G) \leq c(G), \tag{11.1}$$

da für jedes Matching M und jede Knotenüberdeckung U von G gilt: $|M| \leq |U|$ („Je zwei Kanten aus M haben keinen Knoten gemeinsam; also benötigt man mindestens $|M|$ Knoten, um alle Kanten von G zu treffen.“)

Beispiel. G sei der vollständige Graph mit drei Knoten:



Dann gilt $m(G) = 1$ und $c(G) = 2$.

Das Beispiel zeigt, dass $m(G) < c(G)$ für nichtbipartite Graphen möglich ist. Für bipartite Graphen gilt dagegen der folgende *Satz von König*[6].

Satz (D. König, 1931).
Für jeden bipartiten Graphen gilt

$$m(G) = c(G). \tag{11.2}$$

Wir werden einen *konstruktiven Beweis des Satzes von König* geben: *Der Beweis wird eine Methode liefern, wie man in einem bipartiten Graphen eine minimale Knotenüberdeckung findet.*

Wenn man in einem bipartiten Graphen ein Matching M mit maximaler Kantenzahl gefunden hat, *so ist es sehr nützlich, gleichzeitig auch eine minimale Knotenüberdeckung U, also eine Knotenüberdeckung mit $|U| = |M|$ zu besitzen.*

Stellen Sie sich vor, Sie haben in mühevoller Rechnung für einen sehr großen bipartiten Graphen ein Matching M mit maximaler Kantenzahl gefunden. Dem Matching selbst kann man dann in der Regel nicht ansehen, dass es kein Matching mit größerer Kantenzahl gibt. Wenn Sie aber gleichzeitig eine Knotenüberdeckung U präsentieren, für die $|U| = |M|$ gilt, so wird jeder sofort einsehen, dass es sich bei M in der Tat um ein Matching mit maximaler Kantenzahl handelt. Die Knotenüberdeckung U ist in diesem Fall ein hervorragendes Mittel, um die Optimalität von M zu *verifizieren*. Anders gesagt: U ist ein *Zertifikat* für die Optimalität von M.

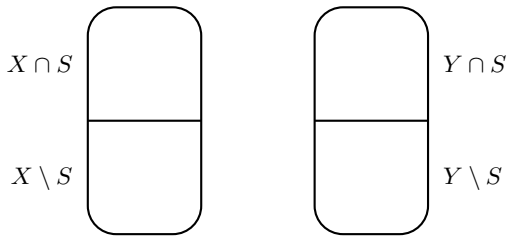
Der im Folgenden präsentierte konstruktive Beweis des Satzes von König knüpft an unseren Algorithmus zum Auffinden eines Matchings mit maximaler Kantenzahl an (vgl. Seite 143ff). Eine zentrale Rolle spielt dabei der Begriff des *alternierenden Pfades*, den wir im vorangegangenen Abschnitt kennengelernt haben.

[5] Engl. *node covering number* oder auch einfach nur *covering number*.
[6] Dénes König (1884-1944), ungarischer Mathematiker.

Beweis des Satzes von König. G sei bipartit. Zu zeigen ist $m(G) = c(G)$. Es sei M ein Matching von G mit $|M| = m(G)$. Zu finden ist eine Knotenüberdeckung U von G mit $|U| = |M|$. Wie üblich bezeichnen wir mit X und Y eine zu G gehörige Knotenpartition.

Mit S bezeichnen wir die Menge derjenigen Knoten von G, die von X aus mit einem alternierenden Pfad erreichbar sind. (Zur Erinnerung: Ein alternierender Pfad startet immer in einem ungepaarten Knoten von X.) Es sei

$$U := \big(X \setminus S\big) \cup \big(Y \cap S\big).$$



Es gelten die folgenden Feststellungen:

(i) In $X \setminus S$ gibt es keine ungepaarten Knoten (nach Definition von S und aufgrund der Tatsache, dass ein ungepaarter Knoten $x \in X$ für sich allein genommen bereits einen alternierenden Pfad darstellt).

(ii) In $Y \cap S$ gibt es ebenfalls keine ungepaarten Knoten (, da es sonst einen augmentierenden Pfad gäbe, im Widerspruch zu $|M| = m(G)$).

(iii) Es gibt keine Kante aus M, die $Y \cap S$ mit $X \setminus S$ verbindet (, da es sonst einen alternierenden Pfad gäbe, der einen Knoten aus $X \setminus S$ erreicht).

Aus (i)-(iii) folgt, dass sämtliche Knoten aus U auf Kanten aus M liegen, und zwar je zwei verschiedene Knoten von U auf unterschiedlichen Kanten aus M. Es folgt

$$|U| \leq |M|.$$

Darüber hinaus gilt:

(iv) Es gibt keine Kanten zwischen $X \cap S$ und $Y \setminus S$.

Begründung zu (iv). Angenommen, $e = \{x', y'\}$ wäre eine Kante von G mit $x' \in X \cap S$ und $y' \in Y \setminus S$. Wegen $x' \in X \cap S$ existiert ein alternierender Pfad P, der in x' endet. Dann ist P entweder nur einpunktig (d.h., x' ist ein ungepaarter Knoten) oder P durchläuft abwechselnd Knoten aus $X \cap S$ und $Y \cap S$, wobei sich Nicht-Matchingkanten und Matchingkanten abwechseln und die letzte Kante eine Matchingkante ist. In beiden Fällen würde $e \notin M$ gelten und y' wäre durch einen alternierenden Pfad erreichbar, im Widerspruch zu $y' \notin S$.

Feststellung (iv) bedeutet, dass U eine Knotenüberdeckung von G ist, weshalb insbesondere $|U| \geq |M|$ gilt. Oben hatten wir bereits $|U| \leq |M|$ festgestellt. Insgesamt haben wir also (wie gewünscht) eine Knotenüberdeckung U mit $|U| = |M|$ erhalten. □

Wir kommen nun zurück auf unseren Matching-Algorithmus für bipartite Graphen $G = (V, E)$ mit Knotenpartition $V = X \cup Y$ (vgl. Seite 143ff). In diesem wurde auf das zu G gehörige Netzwerk $N = (G', c, s, t)$ der Algorithmus von Edmonds und Karp angewandt. (Zur Definition von N siehe Seite 144.) Es sei (S', T') der vom Algorithmus von Edmonds und Karp gelieferte minimale Schnitt von $N = (G', c, s, t)$. Vergleicht man den Beweis des Max-Flow Min-Cut Theorems mit dem Beweis des Satzes von König, so erkennt man, dass für die im Beweis des Satzes von König definierte Menge S gilt: $S = S' \setminus \{s\}$. (Um zu erkennen, dass dies tatsächlich so ist, beachte man vor allem die am Ende von Abschnitt 11.3 formulierte Feststellung, dass ein Knoten w in G genau dann mit einem alternierenden Pfad erreichbar ist, wenn er in N mit einem zunehmenden Pfad erreichbar ist.)

Damit ist klar, dass unser Matching-Algorithmus für bipartite Graphen nicht nur ein Matching M mit maximaler Kantenzahl, sondern auch eine minimale Knotenüberdeckung U liefert, die man wie folgt

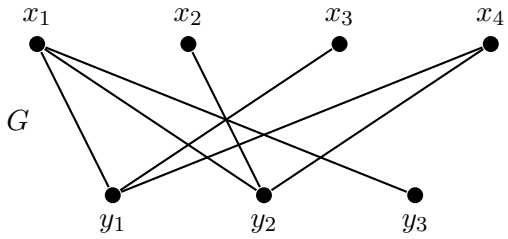
bekommt: Ist (S', T') der gefundene minimale Schnitt von $N = (G', c, s, t)$, so sei $S = S' \setminus \{s\}$. Dann gilt:

$$U = \big(X \setminus S\big) \cup \big(Y \cap S\big). \tag{11.3}$$

11.5 Zwei Beispiele

In diesem Abschnitt wird der zuvor besprochene Matching-Algorithmus anhand von zwei Beispielen illustriert. Im ersten Beispiel wird die Vorgehensweise sehr detailliert beschrieben, wobei zwischendurch auch immer das zum Graphen G gehörende Netzwerk $N = (G', c, s, t)$ betrachtet wird. Im zweiten Beispiel wird dann (wie allgemein üblich) das Netzwerk N überhaupt nicht mehr herangezogen: Das Vorgehen wird nur noch in der Sprache der ungerichteten Graphen, Matchings und alternierenden Pfade beschrieben.

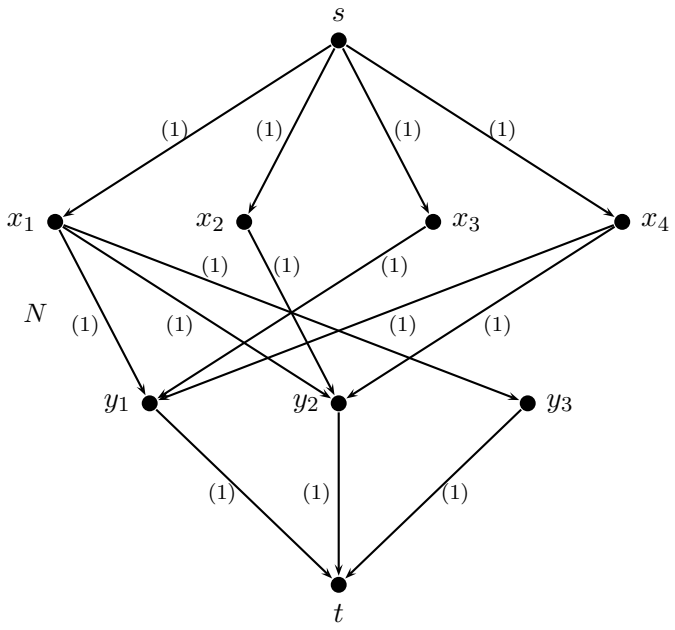
Beispiel 1. Wir betrachten den folgenden bipartiten Graphen $G = (V, E)$ mit $V = X \cup Y$ für $X = \{x_1, x_2, x_3, x_4\}$ und $Y = \{y_1, y_2, y_3\}$:



Die **Aufgabe** ist, ein Matching mit maximaler Kantenzahl und gleichzeitig eine minimale Knotenüberdeckung zu finden. Hierzu soll – wie in den Abschnitten 11.2 - 11.4 beschrieben – der Algorithmus von Edmonds und Karp (vgl. Abschnitt 9.4) verwendet werden, wobei die folgende Regel zu beachten ist:

($\star$) Ist im Algorithmus von Edmonds und Karp die Reihenfolge der zu markierenden Knoten nicht festgelegt, *so sind Knoten mit kleinerem Index vorzuziehen.*

Der Deutlichkeit halber sei angemerkt, dass es in dieser Aufgabe nicht darum geht, ein Matching mit maximaler Kantenzahl und eine minimale Knotenüberdeckung „durch scharfes Hinsehen“ zu ermitteln. Es geht darum, den Algorithmus von Edmonds und Karp anzuwenden. Wir gehen vor wie in Abschnitt 11.2 beschrieben, d.h., als erstes verwandeln wir den bipartiten Graphen G in ein Netzwerk $N = (G', c, s, t)$:



Los geht es wie immer mit dem Nullfluss, den wir f_0 nennen wollen (vgl. Abbildung 11.1). In Abbildung 11.1 sind außerdem die Markierungen eingetragen, die vom Algorithmus vergeben werden, bevor es zur ersten Flussvergrößerung kommt: Die Knoten wurden in der Reihenfolge $s, x_1, x_2, x_3, x_4, y_1, y_2, y_3, t$ markiert. Diese Reihenfolge ergibt sich aus Zeile (5') des Algorithmus von Edmonds und Karp („first labelled – first scanned") sowie aus der obigen Regel ($\star$).

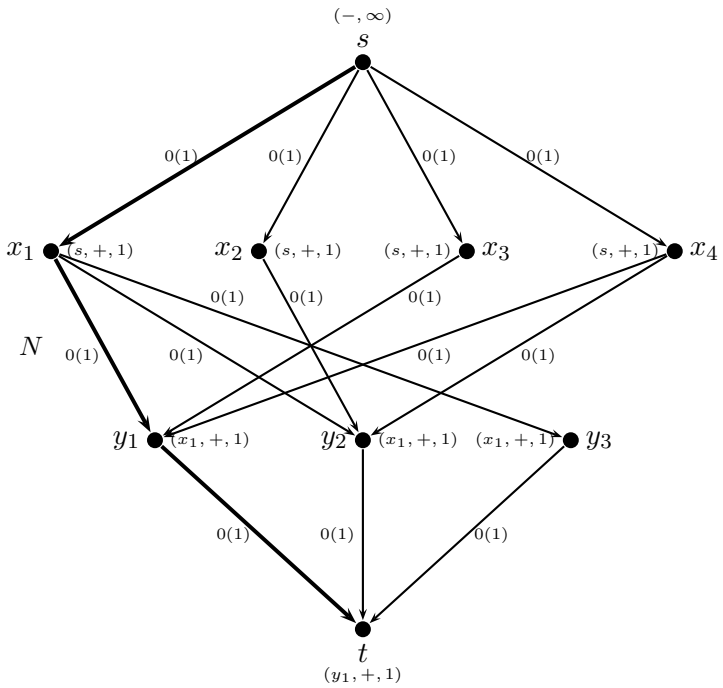


Abbildung 11.1

Der gefundene flussvergrößernde Pfad $P' = (s, x_1, y_1, t)$ wurde in Abbildung 11.1 fett eingezeichnet; der verbesserte Fluss, den wir f_1 nennen wollen, wurde weiter unten in Abbildung 11.3 eingetragen. Es gilt

$$\begin{aligned} f_1(s, x_1) &= 1 \\ f_1(x_1, y_1) &= 1 \\ f_1(y_1, t) &= 1 \end{aligned}$$

und $f_1(e) = 0$ für alle übrigen Kanten des Netzwerks N. Das Ergebnis der ersten Flussvergrößerung ist, dass wir den Fluss f_0 zu f_1 verbessert haben. Übertragen wir dieses Ergebnis auf den bipartiten Graphen G, so können wir feststellen: *Am Anfang waren noch keine Matchingkanten vorhanden und nach der ersten Flussvergrößerung besteht das aktuelle Matching aus genau einer Kante, nämlich der Kante* $\{x_1, y_1\}$:

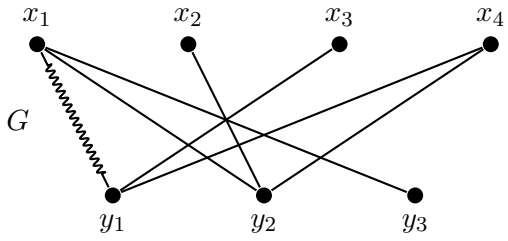


Abbildung 11.2

In Abbildung 11.3 sind neben dem Fluss f_1 die Markierungen eingetragen, die vom Algorithmus vergeben werden, bevor es zur Verbesserung von f_1 kommt; dabei wurden die Knoten in der Reihenfolge $s, x_2, x_3, x_4, y_2, y_1, t$ markiert. Als flussvergrößernden Pfad erhält man $P' = (s, x_2, y_2, t)$.

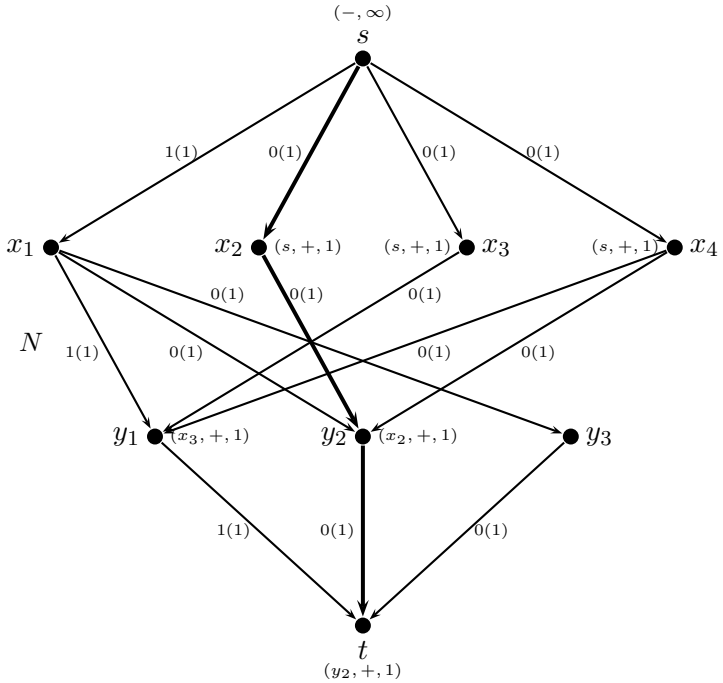


Abbildung 11.3

Verbesserung von f_1 mittels P' führt zum Fluss f_2 (wie in Abbildung 11.5 angegeben) sowie zum Matching $M_2 = \big\{\{x_1, y_1\}, \{x_2, y_2\}\big\}$:

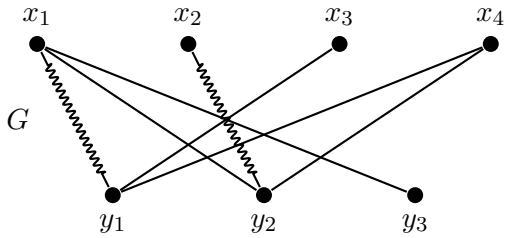


Abbildung 11.4

Bezeichnen wir mit M_0 die leere Menge sowie mit M_1 das Matching, das nach der ersten Flussvergrößerung aktuell war, so können wir den bisherigen Verlauf wie folgt darstellen:

$$\begin{array}{ll} f_0 & M_0 = \emptyset \\ \downarrow & \downarrow \\ f_1 & M_1 = \Big\{\{x_1, y_1\}\Big\} \\ \downarrow & \downarrow \\ f_2 & M_2 = \Big\{\{x_1, y_1\}, \{x_2, y_2\}\Big\} \end{array}$$

Alles was bislang passiert ist, lässt sich wie folgt in einem einzigen Satz zusammenfassen:

In den ersten beiden Iterationen wählt der Algorithmus die
Kanten $\{x_1, y_1\}$ und $\{x_2, y_2\}$ als Matchingkanten aus.

Nun geht es in die nächste Runde (3. Iteration): Es werden Markierungen wie in Abbildung 11.5 vergeben, wobei die Knoten in folgender Reihenfolge markiert werden: $s, x_3, x_4, y_1, y_2, x_1, x_2, y_3, t$. Dies führt in N zum flussvergrößernden Pfad $P' = (s, x_3, y_1, x_1, y_3, t)$ bzw. in G zum augmentierenden Pfad $P = (x_3, y_1, x_1, y_3)$; siehe Abbildung 11.5 bzw. 11.6.

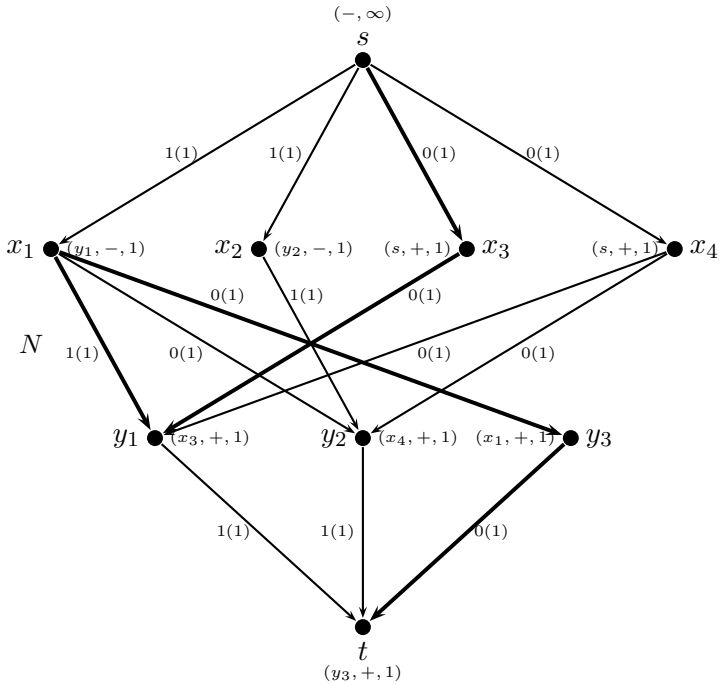


Abbildung 11.5: Der flussvergrößernde Pfad $P' = (s, x_3, y_1, x_1, y_3, t)$.

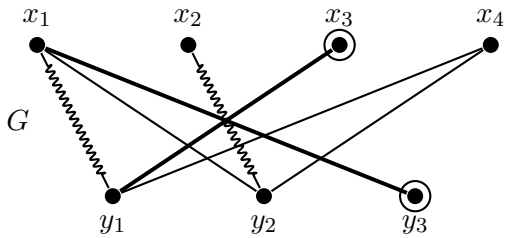


Abbildung 11.6: Der augmentierende Pfad $P = (x_3, y_1, x_1, y_3)$.

Der Austausch von Nicht-Matchingkanten und Matchingkanten von P führt zum Matching

$$M_3 = \Big\{\{x_1, y_3\}, \{x_2, y_2\}, \{x_3, y_1\}\Big\},$$

das in der folgenden Darstellung wiedergegeben wird:

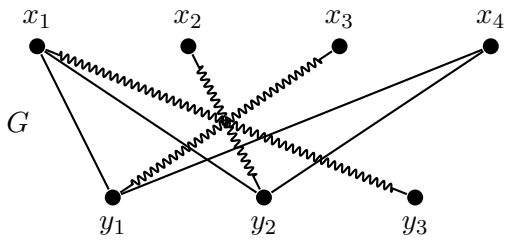


Abbildung 11.7

Nun geht es – wie man sich bereits denken kann – in die letzte Runde (4. Iteration), in der der Algorithmus kein verbessertes Matching findet, stattdessen aber ein *Zertifikat für die Optimalität von* M_3 liefert. Der Abbildung 11.8 entnimmt man, welche Knoten markiert werden; die Markierung erfolgt dabei in der Reihenfolge $s, x_4, y_1, y_2, x_3, x_2$. Die übrigen Knoten werden nicht markiert, da sie nicht durch einen zunehmenden Pfad erreicht werden. Man erhält den Schnitt (S', T') mit $S' = \{s, x_2, x_3, x_4, y_1, y_2\}$ und $T' = \{x_1, y_3, t\}$.

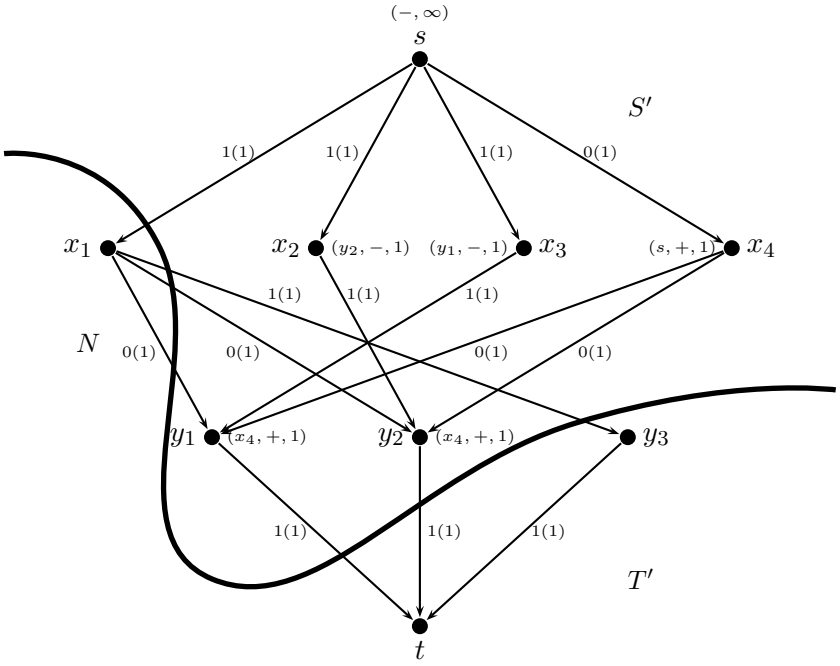


Abbildung 11.8

Für G bedeutet dies, wenn wir $S = S' \setminus \{s\}$ setzen: Als Zertifikat für die Optimalität des Matchings $M_3 = \{\{x_1, y_3\}, \{x_2, y_2\}, \{x_3, y_1\}\}$ hat der Algorithmus die minimale Knotenüberdeckung $U = (X \setminus S) \cup (Y \cap S) = \{x_1, y_1, y_2\}$ gefunden (vgl. Abbildung 11.9).

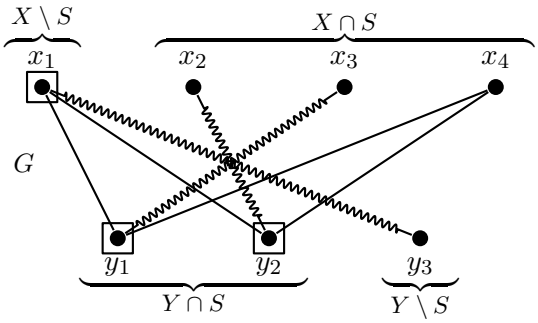
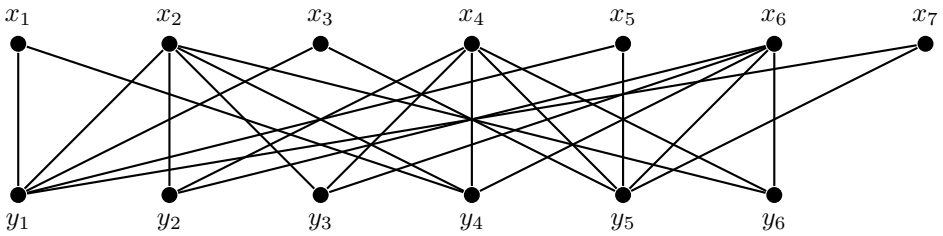


Abbildung 11.9: Die Knoten der minimalen Knotenüberdeckung U sind durch ein Quadrat gekennzeichnet.

Beispiel 2. Wir betrachten den folgenden bipartiten Graphen $G = (V, E)$:

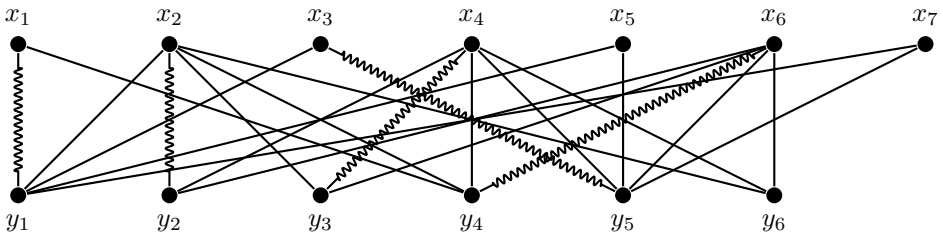


Es sei $X = \{x_1, \ldots, x_7\}$ und $Y = \{y_1, \ldots, y_6\}$. Wie in Beispiel 1 besteht die Aufgabe darin, ein Matching mit maximaler Kantenzahl und gleichzeitig eine minimale Knotenüberdeckung zu finden. Hierzu soll der Algorithmus von Edmonds und Karp verwendet werden, und auch diesmal soll die Regel ($\star$) angewandt werden.

Im Prinzip wird in Beispiel 2 alles wie in Beispiel 1 ablaufen. *Ein wesentlicher Unterschied besteht jedoch in der Art der Beschreibung*: Das Netzwerk $N = (G', c, s, t)$ wird nicht mehr erwähnt werden; wir haben es höchstens noch im Hinterkopf. In der folgenden Beschreibung verwenden wir nur noch Ausdrucksweisen, die mit Matchings in bipartiten Graphen zusammenhängen; Ausdrucksweisen, die mit Netzwerken und Flüssen zusammenhängen, werden vermieden. *Das hat den Vorteil, dass alles einfacher, übersichtlicher und kürzer wird.*

Lösung: Mit dem „leeren Matching“ $M_0 = \emptyset$ geht es los; in der ersten Iteration wählt der Algorithmus die Kante $\{x_1, y_1\}$ als Matchingkante aus. (Dies läuft im Detail wie in Beispiel 1 ab.) Am Ende der 1. Iteration gilt also $M_1 = \{\{x_1, y_1\}\}$. Im Verlauf der 2.-5. Iteration kommen dann – in der angegebenen Reihenfolge – die Kanten $\{x_2, y_2\}$, $\{x_3, y_5\}$, $\{x_4, y_3\}$ und $\{x_6, y_4\}$ hinzu[7]. Wir können also festhalten: Nach fünf Iterationen lautet das aktuelle Matching wie folgt (vgl. Zeichnung):

$$M_5 = \Big\{\{x_1, y_1\}, \{x_2, y_2\}, \{x_3, y_5\}, \{x_4, y_3\}, \{x_6, y_4\}\Big\}.$$



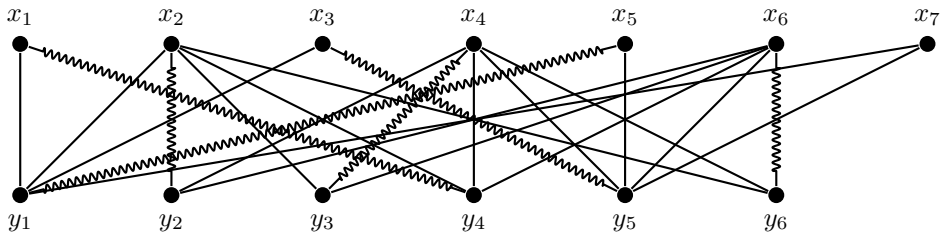
[7] Prüfen Sie nach, dass dies tatsächlich so ist; machen Sie sich auch klar, dass es an (5') („first labelled – first scanned“) sowie an der Regel ($\star$) liegt, dass es zur Auswahl der genannten fünf Kanten kommt.

In der 6. Iteration findet der Algorithmus den augmentierenden Pfad

$$\Big(x_5, y_1, x_1, y_4, x_6, y_6\Big)$$

und ändert das aktuelle Matching entsprechend ab (Austausch der Matchingkanten gegen die Nicht-Matchingkanten dieses Pfads): $\big\{x_5, y_1\big\}$, $\big\{x_1, y_4\big\}$ und $\big\{x_6, y_6\big\}$ werden in die aktuelle Menge der Matchingkanten aufgenommen, $\big\{x_1, y_1\big\}$ und $\big\{x_6, y_4\big\}$ verlassen diese Menge. Nach der 6. Iteration lautet das aktuelle Matching demnach (vgl. Zeichnung):

$$M_6 = \Big\{\big\{x_1, y_4\big\}, \big\{x_2, y_2\big\}, \big\{x_3, y_5\big\}, \big\{x_4, y_3\big\}, \big\{x_5, y_1\big\}, \big\{x_6, y_6\big\}\Big\}.$$

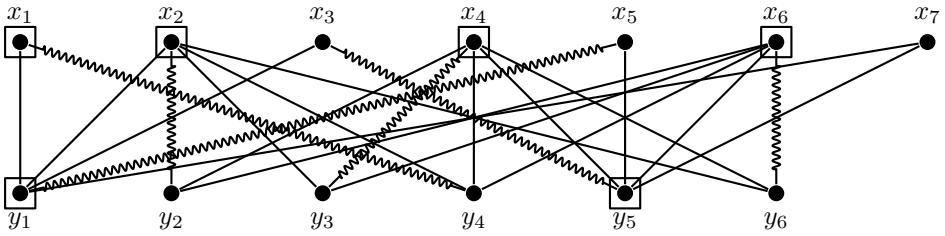


In der 7. Iteration versucht der Algorithmus, einen augmentierenden Pfad zu finden, der in x_7 startet – was nicht gelingt. Dabei werden die folgenden Knoten – in der angegebenen Reihenfolge – mit alternierenden Pfaden erreicht und markiert: x_7, y_1, y_5, x_5, x_3.

Ergebnis: Der Algorithmus liefert das obige Matching M_6 mit 6 Kanten zusammen mit der minimalen Knotenüberdeckung

$$U = \Big\{x_1, x_2, x_4, x_6, y_1, y_5\Big\}.$$

In der nachfolgenden Zeichnung wird dieses Ergebnis veranschaulicht; die Knoten aus U sind durch ein Quadrat gekennzeichnet.



Wie kommt man auf die minimale Knotenüberdeckung $U = \big\{x_1, x_2, x_4, x_6, y_1, y_5\big\}$?

Die Antwort findet sich im Skript auf Seite 151: Bezeichnen wir (wie dort) mit S die Menge der in der 7. Iteration mittels alternierender Pfade erreichbaren Knoten, so ist

$$U = \Big(X \setminus S\Big) \cup \Big(Y \cap S\Big)$$

die gewünschte Knotenüberdeckung.

Wir betonen noch einmal: In der Darstellung von Beispiel 2 wurden keine Begriffe aus der „Welt der Flussnetzwerke“ verwendet, sondern nur noch Begriffe aus der „Welt der Matchings“. Beispielsweise war nicht mehr von zunehmenden Pfaden zu einem Knoten die Rede, sondern es ging um *Erreichbarkeit durch alternierende Pfade*. Dieses Umsteigen auf andere Sprechweisen ist üblich und vor allem auch angemessen: Es geht ja in der Fragestellung ausschließlich um Matchings in ungerichteten Graphen und nicht um Flüsse in Netzwerken. Ermöglicht wurde das Umsteigen auf andere Sprechweisen durch die

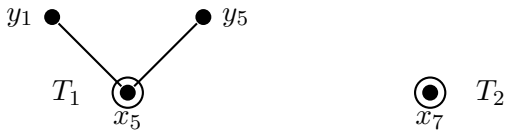
Ergebnisse des Abschnitts 11.3. Empfehlung: *Schauen Sie sich die zusammenfassende Bemerkungen am Ende von Abschnitt 11.3 noch einmal an.*

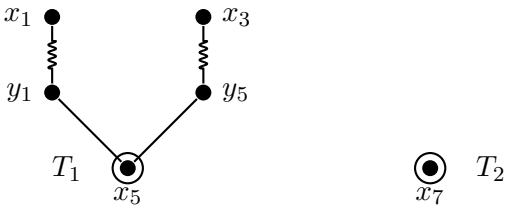
Die Vorgehensweise in Beispiel 2 wird *besonders anschaulich*, wenn man sich die 6. und 7. Iteration mithilfe von „alternierenden Bäumen“ vorstellt. Wir führen dies für die 6. Iteration vor.

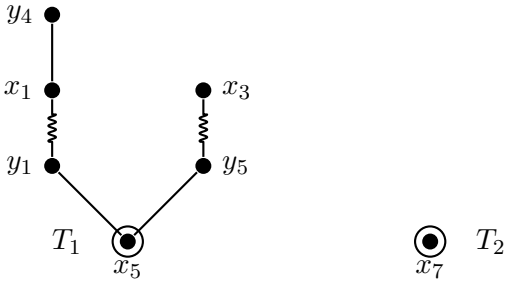
Ausgangspunkt in der 6. Iteration ist das Matching $M_5 = \{\{x_1, y_1\}, \{x_2, y_2\}, \{x_3, y_5\}, \{x_4, y_3\}, \{x_6, y_4\}\}$ (siehe entsprechende Abbildung weiter oben). Im Anschluss an diese Abbildung wurde gesagt, dass der Algorithmus in der 6. Iteration den augmentierenden Pfad $(x_5, y_1, x_1, y_4, x_6, y_6)$ findet. *Der Prozess, wie der Algorithmus diesen Pfad findet, lässt sich besonders anschaulich mit alternierenden Bäumen darstellen.*

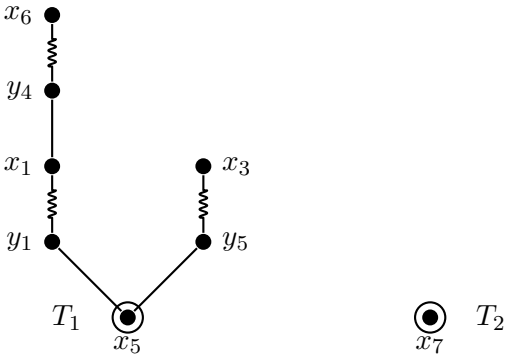
Da wir in X zwei ungepaarte Knoten (x_5 und x_7) haben, geht es um zwei Bäume T_1 und T_2. Zunächst sind diese Bäume noch sehr klein – sie bestehen nur aus ihren Wurzeln x_5 und x_7:

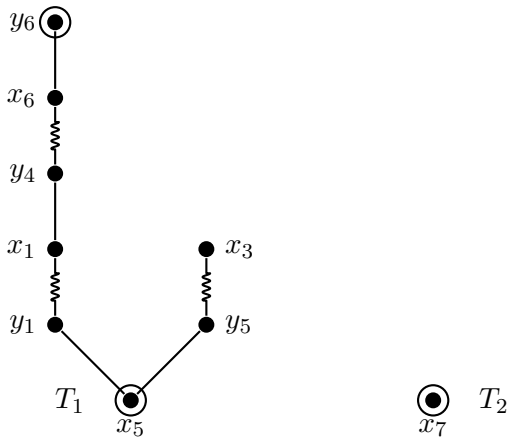
Das Bemühen, einen augmentierenden Pfad zu finden, führt dazu, dass diese Bäume wachsen. Im vorliegenden Fall wird sich herausstellen, dass nur T_1 wächst; bei T_2 liegt „Nullwachstum“ vor. Im Folgenden sind die Phasen dieses Wachstums in kleinen Schritten dargestellt:







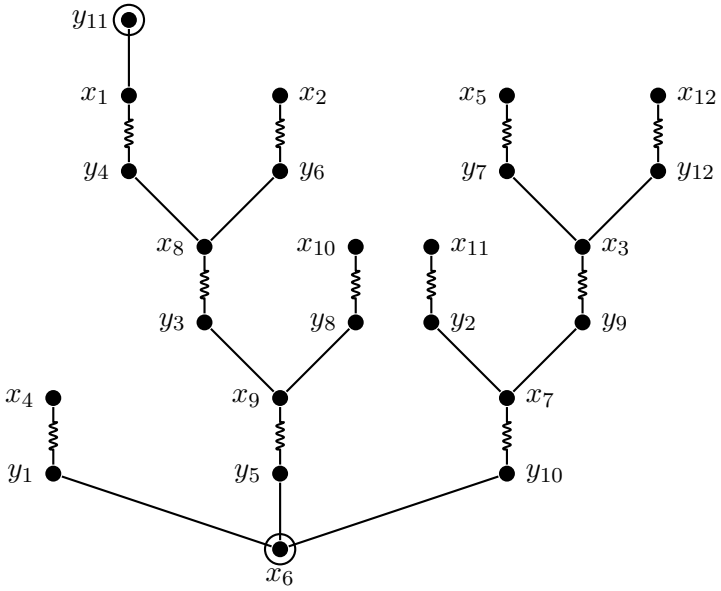




Der Knoten y_6 wurde mit einem Kreis versehen, da es sich – was besonders wichtig ist – um einen ungepaarten Knoten aus Y handelt.

Aufgabe: Stellen Sie auch für die 7. Iteration den dazugehörigen alternierenden Baum dar.

In unserem Beispiel hatten die auftretenden Bäume eine etwas spezielle Gestalt. Deshalb sei abschließend noch ein alternierender Baum angegeben, wie er typischerweise in einem größeren Beispiel vorkommen könnte:



Ein typischer alternierender Baum

Wir haben im Zusammenhang mit Beispiel 2 von „alternierenden Bäumen“ gesprochen, ohne diesen Begriff exakt definiert zu haben. Dies soll nun nachgeholt werden. Im Folgenden wird die übliche Definition des Begriffs *alternierender Baum* wiedergegeben. Man erkennt sofort, dass alle Bäume, die wir bislang als „alternierende Bäume“ bezeichnet haben, auch im Sinne dieser Definition alternierende Bäume sind.

Definition.

Es sei $G = (V, E)$ ein bipartiter Graph mit Knotenpartition $V = X \cup Y$; ein Matching M von G sei fest gegeben. Ein Teilgraph T von G heißt *alternierender Baum*, falls T ein Baum ist und falls es einen Knoten r von T gibt, für den gilt

(i) r ist ein ungepaarter Knoten aus X.

(ii) Für alle Knoten v von T gilt: Der eindeutig bestimmte r, v-Pfad in T ist ein alternierender Pfad.

11.6 Der Heiratssatz

Ist $G = (V, E)$ ein bipartiter Graph mit Knotenpartition $V = X \cup Y$ und ist $A \subseteq X$ gegeben, so bezeichnet man mit $\Gamma(A)$ die Menge aller Knoten von Y, die einen Nachbarn in A besitzen. Wir erwähnen einen der bekanntesten Sätze über Matchings in bipartiten Graphen, den sogenannten *Heiratssatz*.

Heiratssatz.

In einem bipartiten Graphen $G = (V, E)$ mit Knotenpartition $V = X \cup Y$ gibt es genau dann ein Matching M, für das $|M| = |X|$ gilt, wenn für alle $A \subseteq X$ die Bedingung $|A| \leq |\Gamma(A)|$ erfüllt ist.

Der Heiratssatz wurde in der ersten Hälfte des 20. Jahrhunderts von verschiedenen Mathematikern unabhängig voneinander in leicht unterschiedlichen Versionen formuliert und bewiesen; zu nennen sind neben anderen der ungarische Mathematiker D. König, der deutsche Mathematiker G. Frobenius sowie der Amerikaner Ph. Hall.

Der Heiratssatz ist nicht schwer zu beweisen; es gibt viele unterschiedliche Beweise. Einen algorithmischen Beweis, der mit alternierenden Bäumen und augmentierenden Pfaden arbeitet und der deshalb für uns besonders interessant ist, findet man beispielsweise im folgenden Lehrbuch

- A. Steger: *Diskrete Strukturen*. Band 1. Springer-Verlag (2007)[8].

Auf den nachfolgenden Seiten (bis zum Ende von Abschnitt 11.6) finden Sie (in leicht gekürzter Form) das entsprechende Kapitel aus dem Buch von Steger; dieses Kapitel ist von Ihnen selbstständig durchzuarbeiten.

Betrachten wir das folgende Zuordnungsproblem. Gegeben ist eine Menge von Rechnern mit verschiedenen Leistungsmerkmalen (Speicher, Geschwindigkeit, Plattenplatz, etc.) und eine Menge von Jobs mit unterschiedlichen Leistungsanforderungen an die Rechner. Gibt es eine Möglichkeit, die Jobs so auf die Rechner zu verteilen, dass alle Jobs gleichzeitig bearbeitet werden können? Graphentheoretisch können wir das Problem wie folgt formulieren: Wir symbolisieren jeden Job und jeden Rechner durch einen Knoten und verbinden einen Job mit einem Rechner genau dann, wenn der Rechner die Leistungsanforderungen des Jobs erfüllt. Gesucht ist dann eine Auswahl der Kanten, die jedem Job genau einen Rechner zuordnet und umgekehrt jedem Rechner höchstens einen Job. Eine solche Teilmenge der Kanten nennt man ein Matching des Graphen. Obiges Beispiel beschreibt ein Matching in einem bipartiten Graphen. In der folgenden Definition verallgemeinern wir diesen Begriff auf beliebige Graphen.

Definition 2.56.

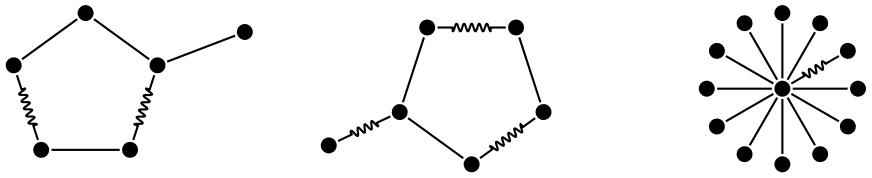
Eine Kantenmenge $M \subseteq E$ heißt *Matching* in einem Graphen $G = (V, E)$, falls kein Knoten des Graphen zu mehr als einer Kante aus M inzident ist, oder formal ausgedrückt, wenn

$$e \cap f = \emptyset \quad \text{für alle } e, f \in M \text{ mit } e \neq f.$$

Man sagt ein Knoten v wird von M *überdeckt*, falls es eine Kante $e \in M$ gibt, die v enthält. Ein Matching M heißt *perfektes Matching*, wenn jeder Knoten durch genau eine Kante aus M überdeckt wird, oder, anders ausgedrückt, wenn $|M| = |V|/2$.

[8] Dieses Buch steht Ihnen als eBook via SpringerLink vollständig zur Verfügung.

BEISPIEL 2.57. Ein Graph enthält im Allgemeinen sehr viele Matchings. Beispielsweise ist $M = \{e\}$ für jede Kante $e \in E$ ein Matching. Die folgende Abbildung zeigt ein Matching (links) und ein perfektes Matching (Mitte).



Nicht jeder Graph enthält jedoch ein perfektes Matching. Für Graphen mit einer ungeraden Anzahl an Knoten ist dies klar. Es gibt aber sogar Graphen mit beliebig vielen Knoten, deren größtes Matching aus einer einzigen Kante besteht. Dies sind die so genannten *Sterngraphen* (im Bild rechts), deren Kantenmenge genau aus den zu einem Knoten inzidenten Kanten besteht.

Wir werden uns in diesem Kapitel auf Matchings in bipartiten Graphen beschränken. Der folgende Satz von PHILIP HALL (1904-1982) gibt eine notwendige und hinreichende Bedingung an, unter der ein Matching in einem bipartiten Graphen existiert, das alle Knoten einer Partition überdeckt. Zur Formulierung des Satzes führen wir noch eine abkürzende Schreibweise für die *Nachbarschaft einer Knotenmenge* $X \subseteq V$ ein:

$$\Gamma(X) := \bigcup_{v \in X} \Gamma(v).$$

Satz 2.58 (Hall).
In einem bipartiten Graphen $G = (A \uplus B, E)$ gibt es genau dann ein Matching M der Kardinalität $|M| = |A|$, wenn gilt

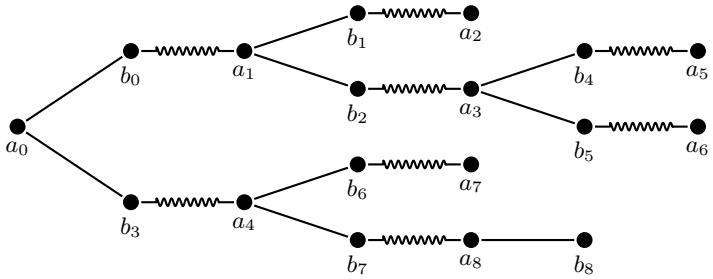
$$\big|\Gamma(X)\big| \geq \big|X\big| \quad \text{für alle } X \subseteq A. \tag{2.2}$$

Beweis. Wir beweisen zuerst die „⇒“-Richtung des Satzes. Sei M ein Matching der Kardinalität $|M| = |A|$. In dem durch M gegebenen Teilgraphen $H = (A \uplus B, M)$ hat jede Teilmenge $X \subseteq A$ nach Definition eines Matchings genau $|X|$ Nachbarn. Wegen $M \subseteq E$ gilt daher auch $|\Gamma(X)| \geq |X|$ für alle $X \subseteq A$.

Die „⇐“-Richtung des Satzes beweisen wir durch Widerspruch. Wir nehmen also an, es gibt einen Graphen $G = (A \uplus B, E)$, der die Bedingung (2.2) erfüllt, der aber kein Matching der Kardinalität $|A|$ enthält. Wir wählen uns nun ein beliebiges kardinalitätsmaximales Matching M in G. Da nach unserer Annahme $|M| < |A|$ gilt, gibt es mindestens einen Knoten in A, wir wollen ihn a_0 nennen, der von M nicht überdeckt wird. Wenden wir (2.2) für $X = \{a_0\}$ an, so sehen wir, dass a_0 mindestens einen Nachbarn in B hat. Wir wählen uns einen solchen beliebig und nennen ihn b_0. Ausgehend von a_0, b_0 konstruieren wir jetzt eine Folge von Knoten $a_i \in A$ und $b_i \in B$ wie folgt:

(1) $k \leftarrow 0$;
(2) **while** b_k wird von M überdeckt **do begin**
(3) $\quad a_{k+1} \leftarrow$ Nachbar von b_k in M;
(4) $\quad$ wähle einen beliebigen Knoten aus $\Gamma\big(\{a_0, \ldots, a_{k+1}\}\big) \setminus \{b_0, \ldots, b_k\}$ und nenne ihn b_{k+1};
(5) $\quad k \leftarrow k + 1$;
(6) **end**

Beachte, dass es wegen der Bedingung (2.2) in jeder Iteration der while-Schleife den Knoten b_{k+1} auch wirklich gibt. Jeder Knoten b_{k+1} ist nach Konstruktion zu mindestens einem Knoten in der Menge $\{a_0, \ldots, a_{k+1}\}$ benachbart. Die folgende Abbildung zeigt einen möglichen Ablauf des Algorithmus, der für $k = 8$ stoppt:



Die Abbildung verdeutlicht, dass es immer von a_0 zu dem letzten gefundenen Knoten, in unserem Beispiel also zu b_8, einen Pfad gibt, der abwechselnd aus Kanten besteht, die nicht zum Matching M gehören, und aus Kanten, die in M enthalten sind. Nach Konstruktion wissen wir zudem, dass sowohl a_0 also auch b_k nicht von M überdeckt werden. Daraus folgt aber, dass wir ein neues Matching M' wie folgt konstruieren können: Wir entfernen aus M alle Kanten des Pfades, die zu M gehören, und fügen stattdessen zu M alle Kanten des Pfades hinzu, die bislang nicht zu M gehört haben. (In obigem Beispiel würde man also die Kanten $\{b_3, a_4\}$ und $\{b_7, a_8\}$ aus M entfernen und dafür die Kanten $\{a_0, b_3\}$, $\{a_4, b_7\}$ und $\{a_8, b_8\}$ zu M hinzufügen.) Das so entstandene Matching M' enthält dann genau eine Kante mehr als das Matching M. Da wir M als kardinalitätsmaximales Matching gewählt haben, kann dies allerdings nicht sein. Wir haben also die Annahme, dass es in G kein Matching der Kardinalität $|A|$ gibt, zum gewünschten Widerspruch geführt.

BEISPIEL 2.59. Den Satz von Hall findet man oft auch unter der Bezeichnung „Heiratssatz“. Warum dies so ist, verdeutlicht die folgende Geschichte. Wenn die Menge A aus Prinzessinnen und die Menge B aus Rittern besteht und eine Kante jeweils bedeutet, dass die Prinzessin und der Ritter sich gerne sehen, so gibt der Satz von Hall die Bedingung an, die erfüllt sein muss, damit sich alle Prinzessinnen glücklich verheiraten können. Man beachte: Über die Vermählungschancen der Ritter wird dadurch noch nichts ausgesagt!

Betrachten wir den Beweis von Satz 2.58 noch einmal genauer. Bei der Konstruktion des Matchings M' haben wir die Annahme, dass M ein kardinalitätsmaximales Matching war, nur insofern ausgenutzt, als dass uns dies die Existenz des Knotens a_0 garantiert hat. Mit anderen Worten, den Algorithmus aus dem Beweis von Satz 2.58 kann man auch verwenden, um aus jedem beliebigen Matching M mit $|M| < |A|$ ein neues Matching M' mit $|M'| = |M| + 1$ zu konstruieren. Insbesondere kann man also ausgehend von $M = \emptyset$ das Matching sukzessive solange vergrößern, bis es aus genau $|A|$ vielen Kanten besteht. Die Ausarbeitung der Details dieses Verfahrens sei dem Leser als Übungsaufgabe überlassen.

Den im Beweis von Satz 2.58 konstruierten Pfad von a_0 zu b_k nennt man aus naheliegenden Gründen einen *augmentierenden Pfad*. Die Idee, maximale Matchings mithilfe von solchen augmentierenden Pfaden zu konstruieren, lässt sich auch auf bipartite Graphen, die die Bedingung (2.2) nicht erfüllen, und sogar auch auf nicht bipartite Graphen übertragen. Allerdings sind die Algorithmen hierfür (zum Teil erheblich) komplizierter[9].

11.7 Knotenüberdeckungen in beliebigen Graphen

Ist $G = (V, E)$ ein *bipartiter* Graph, so wissen wir, wie man in G eine Knotenüberdeckung mit möglichst wenigen Knoten findet: Dies haben wir ausführlich in den Abschnitten 11.2 - 11.5 besprochen, wo wir einen Algorithmus mit polynomieller Laufzeit kennengelernt haben, der sowohl ein Matching mit maximaler Kantenzahl als auch eine Knotenüberdeckung mit minimaler Knotenzahl liefert.

[9] Der Deutlichkeit halber: Für bipartite Graphen, die die Bedingung (2.2) nicht erfüllen, ist die Sache keineswegs komplizierter; Matching-Algorithmen für nicht bipartite Graphen sind dagegen erheblich komplizierter. Wer sich über Matching-Algorithmen für nicht bipartite Graphen informieren möchte, findet Genaueres u.a. in den Büchern von Jungnickel und Lovász/Plummer. Der Name des bekanntesten Matching-Algorithmus für nicht bipartite Graphen sei hier noch erwähnt: Dies ist der berühmte *Blütenalgorithmus von Edmonds*.

Ist $G = (V, E)$ ein *beliebiger*, nicht notwendig bipartiter Graph, so ist die Situation gänzlich anders: Bei dem Problem, in G eine Knotenüberdeckung mit minimaler Knotenzahl zu finden, handelt es sich um ein *NP-schweres Optimierungsproblem*[10]. Ein Algorithmus mit polynomieller Laufzeit existiert für dieses Problem demnach nur, wenn P = NP gilt, was allgemein als sehr unwahrscheinlich angesehen wird.

Was tun? Eine Möglichkeit ist, nach einem *Approximationsalgorithmus* Ausschau zu halten, d.h. nach einem Algorithmus mit polynomieller Laufzeit, der eine *Näherungslösung mit Gütegarantie* liefert. Einen solchen Algorithmus werden wir im Folgenden kennenlernen.

11.7.1 Ein 2-Approximationsalgorithmus für das Knotenüberdeckungsproblem

Es sei $\gamma \geq 1$ eine reelle Zahl. Bevor wir auf das Knotenüberdeckungsproblem zu sprechen kommen, soll definiert werden, was unter einer *Näherungslösung mit Gütegarantie* γ und unter einem *γ-Approximationsalgorithmus* zu verstehen ist.

Zu diesem Zweck nehmen wir an, dass ein *Minimierungsproblem* vorliegt[11]. Dabei muss es sich nicht unbedingt um ein Problem handeln, das als ganzzahliges lineares Programmierungsproblem formulierbar ist, sondern es kann ein beliebiges Optimierungsproblem vorliegen, bei dem eine *reelle nichtnegative Zielfunktion* zu minimieren ist. Wir setzen voraus, dass das Problem eine optimale Lösung besitzt. *Mit L^* sei der (unbekannte) optimale Zielfunktionswert bezeichnet.*

Außerdem: $\mathcal{A}$ sei ein Algorithmus für das vorliegende Problem, der eine Lösung liefert, die nicht notwendigerweise optimal ist. *Der Wert der von $\mathcal{A}$ gelieferten Lösung sei mit $L_{\mathcal{A}}$ bezeichnet.*

Definition.

Man sagt, dass $\mathcal{A}$ eine *Näherungslösung mit Gütegarantie* γ liefert, falls stets (d.h. für alle Instanzen des betrachteten Problems) gilt:

$$L_{\mathcal{A}} \leq \gamma L^*. \tag{11.4}$$

Die Ungleichung (11.4) bedeutet, dass der Wert der von $\mathcal{A}$ gelieferten Lösung niemals schlechter als das γ-fache des Optimalwerts L^* ist. Gilt zusätzlich zu (11.4), dass es sich bei $\mathcal{A}$ um einen polynomiellen Algorithmus handelt, so spricht man von einem *γ-Approximationsalgorithmus*.

Liegt ein polynomieller Algorithmus $\mathcal{A}$ vor, von dem man nachweisen möchte, dass es sich für ein bestimmtes $\gamma \geq 1$ um einen γ-Approximationsalgorithmus handelt, *so steht man vor der Schwierigkeit, dass L^* unbekannt ist.*

Es stellt sich die Frage, wie man (11.4) nachweisen kann, wenn man L^* gar nicht kennt. Die entscheidende Rolle hierbei spielen *untere Schranken* für L^*: Der Wert L^* ist zwar unbekannt, aber häufig kennt man eine untere Schranke B für L^*, d.h., man kennt eine Größe B, für die gilt:

$$B \leq L^*. \tag{11.5}$$

Gelingt es einem nun nachzuweisen, dass

$$L_{\mathcal{A}} \leq \gamma B, \tag{11.6}$$

gilt, so erhält man wegen $B \leq L^*$, dass (11.4) erfüllt ist:

$$L_{\mathcal{A}} \leq \gamma B \leq \gamma L^*.$$

Man hat also Zweierlei zu tun, um zu (11.4) zu gelangen:

- Erstens ist eine geeignete untere Schranke B zu finden;
- zweitens ist für diese Schranke (11.6) nachzuweisen.

[10] Zum Begriff *NP-schwer* und verwandten Begriffen siehe Kleinberg/Tardos (Kapitel 8).
[11] Für Maximierungsprobleme trifft man ähnliche Definitionen und verwendet ähnliche Sprechweisen.

Im Folgenden demonstrieren wir die geschilderte Vorgehensweise anhand des *Knotenüberdeckungsproblems*. Dies lässt sich wie folgt formulieren.

> **Knotenüberdeckungsproblem**
>
> **Eingabe**: ein Graph $G = (V, E)$.
>
> **Gesucht**: eine Knotenüberdeckung von G mit minimaler Knotenzahl.

Anders gesagt: Es ist nach einer Knotenüberdeckung U von G mit $|U| = c(G)$ gefragt. Zur Definition der *Knotenüberdeckungszahl* $c(G)$: vgl. Abschnitt 11.4. Es sei noch einmal betont:

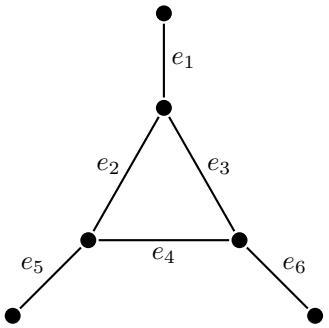
- Wir können nicht erwarten, einen polynomiellen Algorithmus zur exakten Lösung des Knotenüberdeckungsproblems zu finden, da es sich um ein NP-schweres Optimierungsproblem handelt.
- Aus diesem Grund halten wir nach einem γ-Approximationsalgorithmus Ausschau, wobei wir uns wünschen, dass γ „nicht allzu groß“ ist.

Es wird ein 2-Approximationsalgorithmus für das Knotenüberdeckungsproblem vorgestellt werden. Hierzu benötigen wir den Begriff eines *nicht erweiterbaren* Matchings, der wie folgt definiert wird:

Definition.
Es sei $G = (V, E)$ ein Graph. Ein Matching M von G wird *nicht erweiterbar* genannt, wenn es maximal in Bezug auf die Inklusion $\subseteq$ ist, d.h., wenn es unmöglich ist, M durch Hinzunahme einer weiteren Kante $e \in E$ zu einem größeren Matching von G zu erweitern.

Es ist sorgfältig zwischen den Begriffen *nicht erweiterbares Matching* und *Matching mit maximaler Kantenzahl* zu unterscheiden.

Als **Beispiel** betrachten wir den folgenden Graph:



In diesem Graph ist die Menge $M = \{e_1, e_4\}$ ein nicht erweiterbares Matching, das aber keineswegs maximale Kantenzahl besitzt; $M' = \{e_1, e_5, e_6\}$ besitzt dagegen maximale Kantenzahl.

Einschub zur Terminologie in der englischsprachigen Literatur: Im Englischen wird ein nicht erweiterbares Matching *maximal matching* genannt, während *maximum matching* die Bezeichnung für ein Matching mit maximaler Kantenzahl ist.

Für einen gegebenen Graphen $G = (V, E)$ ein nicht erweiterbares Matching zu finden, ist sehr leicht:

Man beginnt mit dem leeren Matching $M = \emptyset$ und geht die Kanten von G in einer beliebigen Reihenfolge durch. Stößt man dabei auf eine Kante e, durch deren Hinzunahme das aktuelle Matching M zu einem größeren Matching wird, so nehme man e in M auf; andernfalls wird e nicht aufgenommen.

Hieran anknüpfend erhält man den gewünschten Approximationsalgorithmus für das Knotenüberdeckungsproblem:

2-Approximationsalgorithmus für das Knotenüberdeckungsproblem.

Bestimme ein nicht erweiterbares Matching M von G und wähle als Output U die Menge aller Knoten, die mit einer Kante von M inzidieren.

Satz.

Der beschriebene Algorithmus ist ein 2-Approximationsalgorithmus für das Knotenüberdeckungsproblem.

Beweis. Es handelt sich um einen polynomiellen Algorithmus, da ein nicht erweiterbares Matching M (wie zuvor beschrieben) in polynomieller Zeit gefunden werden kann.

Das vom Algorithmus gelieferte U ist eine Knotenüberdeckung: Würde U eine Kante e von G nicht treffen, so könnte man e zu M hinzunehmen, d.h., M wäre erweiterbar. Es bleibt zu zeigen:

$$|U| \leq 2c(G). \tag{11.7}$$

Die Rolle der unteren Schranke B aus (11.5) wird von $|M|$ übernommen: Es gilt $|M| \leq c(G)$, da eine Knotenüberdeckung alle Kanten aus M treffen muss. Es folgt $2|M| \leq 2c(G)$, woraus man wegen $|U| = 2|M|$ die gewünschte Ungleichung (11.7) erhält. □

Frage: Ist es möglich, den gefundenen Faktor 2 zu verbessern, indem man den Algorithmus unverändert lässt, aber eine raffiniertere Analyse des Algorithmus vornimmt?

Antwort: Dass dies nicht möglich ist, erkennt man anhand des vollständig bipartiten Graphen $K_{n,n}$. (Details in den Übungen.)

Nächste Frage: Gibt es einen γ-Approximationsalgorithmus für das Knotenüberdeckungsproblem für ein $\gamma < 2$?

Antwort: Das ist nicht bekannt. Es handelt sich um ein berühmtes offenes Problem.

Bekannt ist jedoch Folgendes:

Satz (Dinur und Safra 2001).

Falls P $\neq$ NP, so existiert kein γ-Approximationsalgorithmus für das Knotenüberdeckungsproblem für $\gamma < 1.3606$.

11.7.2 Gewichtete Knotenüberdeckungen

Es sei ein Graph $G = (V, E)$ gegeben, wobei wir $V = \{1, \ldots, n\}$ annehmen. Jeder Knoten i besitzt ein *Gewicht* $w_i \geq 0$ mit $w_i \in \mathbb{Q}$ $(i = 1, \ldots, n)$. Zu finden ist eine Knotenüberdeckung $U \subseteq V$ mit möglichst kleinem Gewicht, wobei das Gewicht von U definiert wird durch

$$w(U) = \sum_{i \in U} w_i.$$

Sind alle Gewichte gleich 1, so liegt der ursprünglich betrachtete Fall vor, in dem die Anzahl der Knoten in einer Knotenüberdeckung zu minimieren ist. Das ursprünglich betrachtete Problem ist also ein Spezialfall des allgemeinen gewichteten Problems.

Das gewichtete Knotenüberdeckungsproblem lässt sich wie folgt als ein ganzzahliges lineares Optimierungsproblem schreiben, das wir mit (ILP) bezeichnen:

$$\begin{aligned} &\text{minimiere} && w_1 x_1 + \ldots + w_n x_n \\ &\text{unter den Nebenbedingungen} \\ &&& x_i + x_j \geq 1 && \{i, j\} \in E \\ &&& x_i \in \{0, 1\} && i \in V \end{aligned} \tag{ILP}$$

Jedem Knoten i entspricht hierbei eine Variable x_i. Außerdem entspricht jeder zulässigen Lösung $x = (x_1, \ldots, x_n)$ von (ILP) eine Knotenüberdeckung U, wenn man die folgende (naheliegende) *Interpretation* vornimmt: $x_i = 1$ bedeutet, dass der Knoten i in U liegt, während $x_i = 0$ gleichbedeutend mit $i \notin U$ ist. Man beachte: Für jede Kante $\{i, j\}$ wird durch die Bedingung $x_i + x_j \geq 1$ gesichert, dass $i \in U$ oder $j \in U$ gilt.

Umgekehrt erhält man auf entsprechende Art zu jeder Knotenüberdeckung U eine zulässige Lösung von (ILP). Zusammenfassend können wir feststellen:

Feststellung.
Ist $x = (x_1, \ldots, x_n)$ eine zulässige Lösung dieses ganzzahligen linearen Programmierungsproblems, so ist $U = \{i \in V : x_i = 1\}$ eine Knotenüberdeckung (und umgekehrt entspricht auch jeder Knotenüberdeckung eine zulässige Lösung von (ILP)).

Außerdem gilt:

Feststellung.
Ist $x^* = (x_1^*, \ldots, x_n^*)$ eine optimale Lösung des ganzzahligen linearen Programmierungsproblems (ILP), so ist $U^* = \{i \in V : x_i^* = 1\}$ eine minimale Knotenüberdeckung (und umgekehrt), wobei „minimale Knotenüberdeckung“ bedeuten soll: Knotenüberdeckung minimalen Gewichts.

11.7.3 Relaxieren und Runden: Ein 2-Approximationsalgorithmus für das gewichtete Knotenüberdeckungsproblem

Im vorangegangenen Abschnitt 11.7.2 haben wir das gewichtete Knotenüberdeckungsproblem in ein ganzzahliges lineares Optimierungsproblem (ILP) umgeschrieben. Dies ermöglicht uns, die *LP-Relaxation* des Problems zu betrachten:

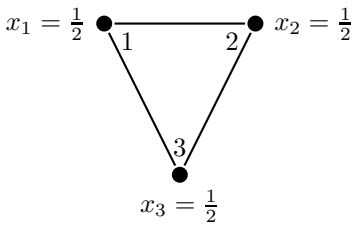
$$\begin{aligned} &\text{minimiere} \quad w_1 x_1 + \ldots + w_n x_n \\ &\text{unter den Nebenbedingungen} \\ &\qquad x_i + x_j \geq 1 \qquad \{i, j\} \in E \\ &\qquad 0 \leq x_i \leq 1 \qquad i \in V \end{aligned} \tag{LP}$$

Beobachtung. Es gilt

$$\text{Optimalwert von (LP)} \leq \text{Optimalwert von (ILP)}. \tag{11.8}$$

Denn: Jede zulässige Lösung von (ILP) ist auch eine zulässige Lösung von (LP).

Das folgende **Beispiel** zeigt, dass Optimalwert von (LP) < Optimalwert von (ILP) möglich ist: Es sei $G = (V, E)$ mit $V = \{1, 2, 3\}$ und $E = \{\{1, 2\}, \{2, 3\}, \{3, 1\}\}$. Alle Gewichte seien gleich 1, d.h. $w_1 = w_2 = w_3 = 1$. Der Optimalwert von (ILP) ist offenbar gleich 2. Für (LP) gibt es dagegen eine zulässige Lösung mit Zielfunktionswert $\frac{3}{2}$: Man braucht nur $x_1 = x_2 = x_3 = \frac{1}{2}$ zu wählen (siehe Zeichnung).



Die Methode, die zum Einsatz kommen wird, lässt sich wie folgt beschreiben:

Man löst die LP-Relaxation und rundet.

Genauer: Es sei $x^* = (x_1^*, \ldots, x_n^*)$ eine optimale Lösung der LP-Relaxation (LP). Dann betrachten wir die Menge $U \subseteq V$, die wie folgt definiert ist:

$$U = \left\{ i \in V : x_i^* \geq \frac{1}{2} \right\}.$$

Satz.
Es sei $x^* = (x_1^*, \ldots, x_n^*)$ eine optimale Lösung der LP-Relaxation (LP) des gewichteten Knotenüberdeckungsproblems (ILP). Dann ist $U = \left\{ i \in V : x_i^* \geq \frac{1}{2} \right\}$ eine Knotenüberdeckung mit einem Gewicht $w(U)$, das höchstens doppelt so groß ist wie das Gewicht einer minimalen Knotenüberdeckung.

Beweis. Wir weisen zunächst nach, dass die Menge $U = \left\{ i \in V : x_i^* \geq \frac{1}{2} \right\}$ tatsächlich eine Knotenüberdeckung ist. Zu diesem Zweck betrachten wir eine beliebige Kante $e = \left\{ i, j \right\} \in E$. Da $(x_1^*, \ldots, x_n^*)$ eine Lösung von (LP) ist, gilt $x_i^* + x_j^* \geq 1$. Deshalb muss $x_i^* \geq \frac{1}{2}$ oder $x_j^* \geq \frac{1}{2}$ (oder beides) gelten. Es folgt, dass $i \in U$ oder $j \in U$ erfüllt ist, d.h., U ist eine Knotenüberdeckung.

Es bleibt zu zeigen: $w(U) = \sum\limits_{i \in U} w_i$ ist höchstens doppelt so groß wie das minimale Gewicht einer Knotenüberdeckung. Dies ergibt sich wie folgt.

Es sei $U_{\min}$ eine Knotenüberdeckung minimalen Gewichts. Die Rolle der unteren Schranke B aus (11.5) wird in diesem Fall vom Zielfunktionswert $\sum\limits_{i=1}^{n} w_i x_i^*$ der optimalen Lösung $x^* = (x_1^*, \ldots, x_n^*)$ von (LP) übernommen. Es gilt nämlich

$$\sum_{i=1}^{n} w_i x_i^* \leq w\left(U_{\min}\right). \tag{11.9}$$

Zum Nachweis von (11.9) ist nur zu beachten, dass $w\left(U_{\min}\right) = \sum\limits_{i \in U_{\min}} w_i$ Zielfunktionswert einer optimalen Lösung von (ILP) ist, nämlich der Lösung $x = (x_1, \ldots, x_n)$ mit

$$x_i = \begin{cases} 1 & \text{falls } i \in U_{\min} \\ 0 & \text{sonst.} \end{cases}$$

Also: (11.9) gilt aufgrund der Beobachtung (11.8).

Außerdem gilt

$$\sum_{i=1}^{n} w_i x_i^* \geq \sum_{i \in U} w_i x_i^* \overset{(\star)}{\geq} \frac{1}{2} \sum_{i \in U} w_i = \frac{1}{2} w(U), \tag{11.10}$$

wobei sich die mit $(\star)$ gekennzeichnete Ungleichung von (11.10) aus der Tatsache ergibt, dass $x_i^* \geq \frac{1}{2}$ für alle $i \in U$ gilt. Aus (11.9) und (11.10) ergibt sich, dass wie behauptet

$$w(U) \leq 2w\left(U_{\min}\right)$$

gilt. □

Den Inhalt des vorangegangenen Satzes können wir auch so aussprechen: *Der Algorithmus „Lösen der LP-Relaxation und Runden“ ist ein 2-Approximationsalgorithmus für das gewichtete Knotenüberdeckungsproblem.* (Man beachte: (LP) lässt sich in polynomieller Zeit lösen; vgl. Abschnitt 14 dieses Skripts.)

12 Greedy-Algorithmen oder: Is Greed Good? Does Greed work?

Wir folgen in diesem Kapitel größtenteils der Darstellung in Jon Kleinberg, Éva Tardos: *Algorithm Design* (Pearson 2006) und geben als Einstieg zwei Absätze aus diesem Lehrbuch wieder.

In Wall Street, that iconic movie of the 1980s, Michael Douglas gets up in front of a room full of stockholders and proclaims, "Greed ... is good. Greed is right. Greed works." In this chapter, we'll be taking a much more understated perspective as we investigate the pros and cons of short-sighted greed in the design of algorithms. Indeed, our aim is to approach a number of different computational problems with a recurring set of questions: Is greed good? Does greed work?

It is hard, if not impossible, to define precisely what is meant by a *greedy algorithm*. An algorithm is greedy if it builds up a solution in small steps, choosing a decision at each step myopically to optimize some underlying criterion[1]. One can often design many different greedy algorithms for the same problem, each one locally, incrementally optimizing some different measure on its way to a solution.

Zu ein und demselben Problem kann man sich häufig unterschiedliche Greedy-Algorithmen („gierige Algorithmen") ausdenken, *die aber ebenso häufig ihr Ziel, eine optimale Lösung zu finden, verfehlen.* Andererseits gibt es aber auch Fälle, in denen man mit einer Greedy-Strategie Erfolg hat: Die vielleicht prominentesten Beispiele sind

- der *Algorithmus von Dijkstra* zum Auffinden kürzester Pfade in Graphen,
- der *Algorithmus von Kruskal* sowie der *Algorithmus von Prim* zur Bestimmung eines minimalen aufspannenden Baums.

Wir werden die genannten Algorithmen erst etwas später besprechen. Der Darstellung von Kleinberg und Tardos folgend werden wir zunächst *zwei Methoden* herausarbeiten, mit deren Hilfe man nachweisen kann, dass eine vorgeschlagene Greedy-Strategie tatsächlich funktioniert, d.h., dass man mit dieser Strategie immer eine optimale Lösung erhält. Um es noch einmal mit den Worten von Kleinberg/Tardos zu sagen:

> It is easy to invent greedy algorithms for almost any problem; finding cases in which they work well, and proving that they work well, is the interesting challenge.

Der ersten der beiden grundlegenden Methoden wird von Kleinberg und Tardos der Name "*the greedy algorithm stays ahead*" gegeben, die zweite Methode wird *Austauschargument* (engl. *exchange argument*) genannt.

Im Folgenden werden die beiden Methoden kurz geschildert, wobei keine präzise Beschreibung angestrebt wird, sondern an einigen Stellen bewusst etwas vage geblieben wird.

Zur 1. Methode ("the greedy algorithm stays ahead"): Bei dieser Methode misst man in gewissen Abständen, welchen Fortschritt der Greedy-Algorithmus erzielt hat, und stellt jedes Mal fest, dass er „vorne liegt" – typischerweise im Vergleich zu einem beliebigen optimalen Algorithmus. Daraus ergibt sich dann, dass der Greedy-Algorithmus auch am Schluss „vorne liegt", d.h. ein optimales Ergebnis abliefert. Der Greedy-Algorithmus hat dann sozusagen einen Start-Ziel-Sieg eingefahren.

Zur 2. Methode („Austauschargument"): Die Grundidee dieser Methode ist es nachzuweisen, dass man eine optimale Lösung $\mathcal{O}$ schrittweise in die vom Greedy-Algorithmus gefundene Lösung A umformen

[1] Mit *decision* ist hier eine *nicht-revidierbare Entscheidung* gemeint, d.h., man hat nicht die Möglichkeit, die getroffene Entscheidung im weiteren Verlauf des Algorithmus noch einmal abzuändern; *myopically* bedeutet *kurzsichtig*.

kann – und zwar so, dass in keinem Schritt eine Verschlechterung der betrachteten Lösung eintritt[2]. Da bei dieser Vorgehensweise eine Umformung häufig aus einem Austausch besteht, spricht man von einem *Austauschargument*.

Wir studieren beide Methoden anhand von sogenannten *Schedulingproblemen*.

12.1 Intervall-Scheduling: The Greedy Algorithm Stays Ahead

12.1.1 Das Intervall-Scheduling-Problem

Gegeben seien eine *Ressource* (etwa ein Hörsaal, ein Elektronenmikroskop oder ein Arbeitsplatz an einem Computer) sowie zahlreiche Anfragen; eine *Anfrage* enthält die Angabe eines Zeitintervalls $[s, f]$[3] („Ich würde die Ressource gerne vom Zeitpunkt s bis zum Zeitpunkt f benutzen."). Die Ressource kann immer nur von einer Person zur selben Zeit benutzt werden. Die Aufgabe des Schedulers besteht darin, möglichst viele Anfragen in einem Zeitplan unterzubringen.

Etwas formaler: Wir bezeichnen die Anfragen mit $1, \ldots, n$; $\{1, \ldots, n\}$ ist also die *Menge der Anfragen*. Zur i-ten Anfrage gehört das Zeitintervall $[s(i), f(i)]$ $(i = 1, \ldots, n)$. Zwei Anfragen heißen *kompatibel*, wenn die dazugehörigen Intervalle sich *nicht überlappen*, d.h., die Intervalle haben höchstens einen Randpunkt gemeinsam. *Gesucht ist eine Teilmenge von möglichst vielen paarweise kompatiblen Anfragen.*

12.1.2 Entwurf eines Greedy-Algorithmus

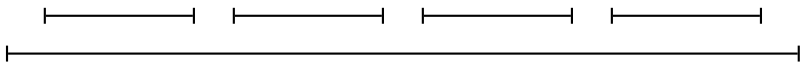
Mit dem Intervall-Scheduling-Problem haben wir ein Beispiel an der Hand, durch das unsere Diskussion über Greedy-Algorithmen *wesentlich konkreter* wird.

*Die **grundlegende Idee** in einem Algorithmus für das Intervall-Scheduling-Problem ist, eine **einfache Regel** anzugeben, mit deren Hilfe man die erste Anfrage i_1 auswählt, die akzeptiert werden soll.* Hat man einmal i_1 nach dieser Regel ausgewählt, so sortiert man alle Anfragen aus, die nicht kompatibel mit i_1 sind; danach wählt man i_2 nach derselben Regel aus, und anschließend werden alle Anfragen aussortiert, die nicht mit i_2 kompatibel sind, usw.

Die Herausforderung beim Entwurf eines guten Greedy-Algorithmus besteht also darin zu entscheiden, *welche* einfache Regel verwendet werden soll. Sorgfalt ist dabei geboten, denn häufig gibt es naheliegende Regeln, die nicht zu guten Lösungen führen. Schauen wir uns einige naheliegende Regeln für das Intervall-Scheduling-Problem an:

(1) Eine besonders naheliegende Regel ist möglicherweise, immer diejenige Anfrage zu wählen, die am frühesten beginnt, für die $s(i)$ also so klein wie möglich ist; auf diese Art wird die Ressource immer so schnell wie möglich wieder benutzt.

Da das Ziel ist, möglichst viele Anfragen zu akzeptieren, kann dies jedoch zu sehr schlechten Ergebnissen führen, wie das folgende Beispiel zeigt:

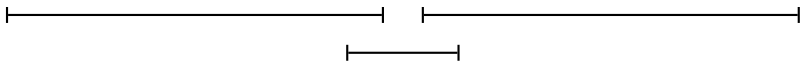


(2) Versucht man aus dem Fehlschlag der ersten Regel zu lernen, so könnte man auf die Idee kommen, immer ein möglichst kurzes Intervall zu wählen – dann können Effekte wie unter (1) nicht auftreten. Neue Regel: Man wähle immer diejenige Anfrage, für die $f(i) - s(i)$ so klein wie möglich ist.

[2] Natürlich gibt es Varianten: Beispielsweise kann es in bestimmten Fällen reichen, $\mathcal{O}$ in eine Lösung umzuformen, die sich von A ein wenig unterscheidet.

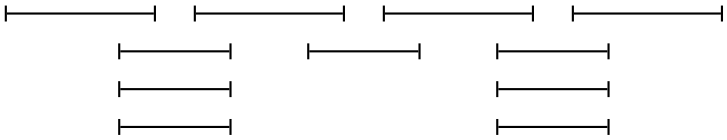
[3] s steht für *starting time*, f steht für *finishing time*; wir nehmen stets $s < f$ an.

Diese Regel scheint etwas besser zu sein – aber leider führt auch diese zu suboptimalen Ergebnissen, wie man am folgenden Beispiel erkennt:



③ Bei der vorherigen Regel ② lag das Problem darin, dass die zweite Anfrage sowohl mit der ersten als auch mit der dritten Anfrage inkompatibel war. Da ist es naheliegend, eine Regel aufzustellen, die besagt, dass immer eine Anfrage zu wählen ist, die mit möglichst wenigen anderen Anfragen kollidiert. Das würde zumindest im Falle des Beispiels ② helfen und scheint auch ansonsten recht vielversprechend zu sein: Man möchte ja insgesamt möglichst viele Anfragen akzeptieren; deshalb erscheint es einleuchtend, die Wahl immer so zu treffen, dass unmittelbar nach einer getroffenen Wahl möglichst wenige Anfragen ausscheiden müssen.

In der Tat muss man sich diesmal etwas mehr anstrengen, um ein entsprechendes Beispiel zu finden. Das folgende Beispiel zeigt jedoch, dass auch diese Regel suboptimale Ergebnisse liefert:



④ Nächster Versuch: Man richtet sich nach den Endzeiten der Anfragen und wählt immer diejenige Anfrage aus, für die $f(i)$ minimal ist. Auch für diese Regel gibt es ein Plausibilitätsargument: Je eher eine Anfrage beendet ist, desto eher kann die nächste Anfrage zum Zuge kommen.

Nach unseren Erfahrungen mit den Regeln ① - ③ sollten wir allerdings skeptisch sein: Auch diesen Regeln lagen auf den ersten Blick einleuchtende Plausibilitätsargumente zugrunde.

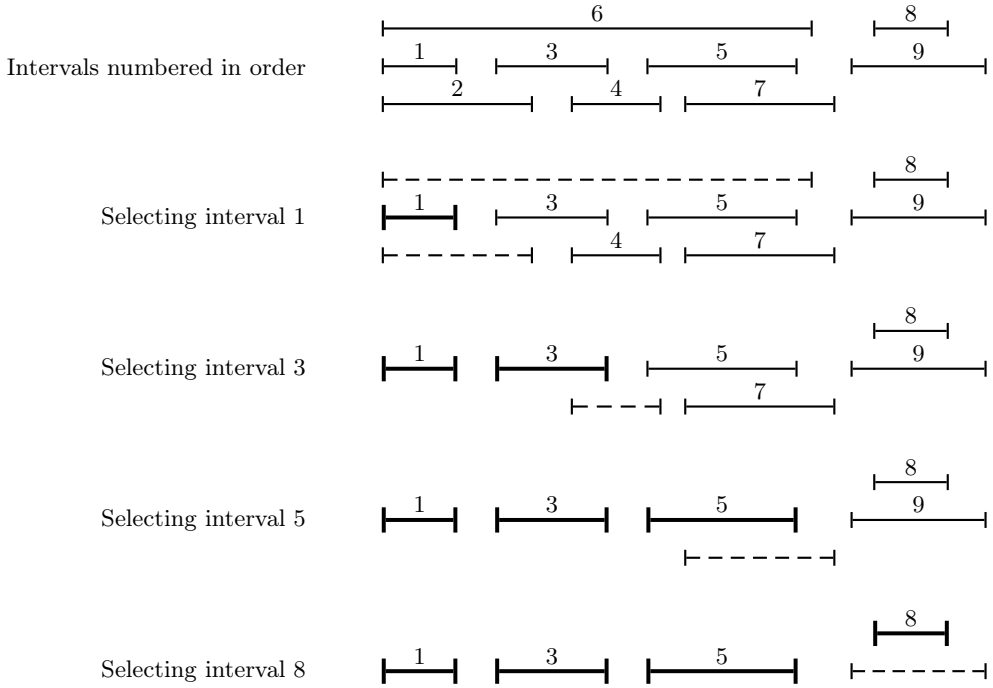
Allerdings fallen einem diesmal keine Beispiele ein, die die Suboptimalität der Regel ④ belegen. So hat man den Verdacht, dass die vierte Regel (möglicherweise) immer eine optimale Lösung liefert. Um dies nachzuweisen, formulieren wir den dazugehörigen Algorithmus etwas formaler: Mit R bezeichnen wir die Menge der Anfragen (engl. requests), über die noch zu entscheiden ist, die also bislang weder akzeptiert noch zurückgewiesen wurden; mit A bezeichnen wir die Menge der bereits akzeptierten Anfragen.

Hier nun der Algorithmus, den wir den *Intervall-Scheduling-Algorithmus* nennen (aus Kleinberg/Tardos: *Algorithm Design*):

Interval Scheduling Algorithm

(1) Initially let R be the set of all requests, and let A be empty
(2) **While** R is not yet empty
(3) Choose a request $i \in R$ that has the smallest finishing time
(4) Add request i to A
(5) Delete all requests from R that are not compatible with request i
(6) **EndWhile**
(7) **Return** the set A as the set of accepted requests

Ein **Beispiel**, das den Ablauf des Intervall-Scheduling-Algorithmus illustriert (ebenfalls aus dem Buch von Kleinberg/Tardos):



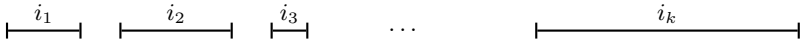
Sample run of the Interval Scheduling Algorithm. At each step the selected intervals are darker lines, and the intervals deleted at the corresponding step are indicated with dashed lines.

Im Folgenden befassen wir uns mit der *Analyse des Intervall-Scheduling-Algorithmus*. Dabei geht es uns weniger um die Laufzeit, sondern vielmehr um die Korrektheit des Algorithmus: Nachdem wir gesehen haben, dass das Greedy-Verfahren bei Verwendung der Regeln (1) - (3) nicht optimal arbeitet, wollen wir uns nun davon überzeugen, dass man bei Verwendung der Regel (4) *immer* eine optimale Lösung erhält.

In Zukunft wollen wir nicht mehr streng zwischen Anfragen und den dazugehörigen Intervallen unterscheiden. Mit A bezeichnen wir die Menge der Anfragen (Intervalle), die der Intervall-Scheduling-Algorithmus am Schluss liefert. Zum Zweck des Vergleichs betrachten wir außerdem eine optimale Lösung $\mathcal{O}$ des Problems. Wir haben zu zeigen, dass A ebenfalls optimal ist, d.h., wir müssen

$$|A| = |\mathcal{O}|$$

zeigen. Mit $i_1, \ldots, i_k$ wollen wir die Intervalle von A bezeichnen – in der Reihenfolge, in der sie vom Intervall-Scheduling-Algorithmus ausgewählt wurden. Dann gilt $f(i_r) \leq s(i_{r+1})$ für $r = 1, \ldots, k-1$, d.h., die Intervalle von A sind paarweise kompatibel und wie in der folgenden Zeichnung illustriert „von links nach rechts angeordnet“.



Da $\mathcal{O}$ eine Lösung des Problems ist, überlappen sich auch die Intervalle von $\mathcal{O}$ nicht, d.h., die Intervalle von $\mathcal{O}$ liegen ebenfalls auf der reellen Achse „von links nach rechts angeordnet“. Dementsprechend wollen wir die Intervalle von $\mathcal{O}$ mit $j_1, \ldots, j_m$ bezeichnen, wobei $f(j_r) \leq s(j_{r+1})$ für alle $r = 1, \ldots, m-1$ gilt.

Da $\mathcal{O}$ eine optimale Lösung ist, gilt $k \leq m$; unser Ziel ist es, $k = m$ nachzuweisen. Um dies zu erreichen, zeigen wir (Dies ist der entscheidende Schritt!), dass Folgendes gilt:

$$f(i_r) \leq f(j_r) \quad \text{für alle } r = 1, \ldots, k. \tag{12.1}$$

Dasselbe in Worten: *Für alle Intervalle i_r von A vergleichen wir den rechten Randpunkt $f(i_r)$ mit dem rechten Randpunkt $f(j_r)$ des entsprechende Intervalls j_r von $\mathcal{O}$ und behaupten, dass $f(i_r)$ niemals rechts von $f(j_r)$ liegt.* Dies ist der präzise Inhalt der Feststellung, dass unser Intervall-Scheduling-Algorithmus „immer vorne liegt".

Beweis von (12.1). Zunächst einmal stellen wir fest, dass $f(i_1) \leq f(j_1)$ gilt; der einfache Grund hierfür ist, dass unser Intervall-Scheduling-Algorithmus i_1 so wählt, dass $f(i_1)$ so klein wie möglich ist. Die Behauptung (12.1) gilt also für $r = 1$.

Wir nehmen nun an, dass (12.1) für ein $r \geq 2$ nicht gilt und führen diese Annahme zum Widerspruch. Hierzu betrachten wir den *kleinsten* Index $r \geq 2$, für den (12.1) falsch ist. Dann gilt also $f(j_r) < f(i_r)$ und $f(i_{r-1}) \leq f(j_{r-1})$.

Aus $f(i_{r-1}) \leq f(j_{r-1})$ folgt (wegen $f(j_{r-1}) \leq s(j_r)$), dass $f(i_{r-1}) \leq s(j_r)$ gilt. Das bedeutet, dass das Intervall j_r „noch im Rennen ist", wenn unser Intervall-Scheduling-Algorithmus das r-te Intervall auswählt. Wegen $f(j_r) < f(i_r)$ hätte der Algorithmus also nicht i_r als r-tes Intervall auswählen dürfen.

Dieser Widerspruch beweist (12.1). □

Nachdem (12.1) als richtig erkannt wurde, fällt der Nachweis von $k = m$ nicht schwer: Angenommen es gelte $k < m$. Dann gibt es ein Intervall j_{k+1} in $\mathcal{O}$ und es gilt $f(j_k) \leq s(j_{k+1})$. Wegen (12.1) gilt außerdem $f(i_k) \leq f(j_k)$. Es folgt $f(i_k) \leq s(j_{k+1})$. Dies würde jedoch Folgendes bedeuten: Nachdem am Ende unseres Intervall-Scheduling-Algorithmus i_k ausgewählt wurde, ist immer noch das Intervall j_{k+1} in R. Dieser Widerspruch beweist $k = m$.

Wir haben somit ein Beispiel für die Methode

The Greedy Algorithm Stays Ahead

kennengelernt. Im Buch von Kleinberg und Tardos finden sich noch viele interessante Ergänzungen sowohl zu dieser Methode als auch zum Thema Intervall-Scheduling. Wir fahren nun fort mit der zweiten Methode, die den Namen

Austauschargument

trägt.

12.2 Ein Algorithmus, der auf einem Austauschargument basiert

Es geht auch hier wieder um ein Schedulingproblem – diesmal sollen jedoch *alle* Anfragen akzeptiert werden, wobei es allerdings zu *Verspätungen* kommen kann.

Genauer: Wir haben wieder eine Ressource, die wir uns als eine *Maschine* vorstellen wollen, an der immer nur ein *Job* zur selben Zeit erledigt werden kann; statt „Anfrage" wollen jetzt immer „Job" sagen. Es gebe n Jobs und jeder Job besitze eine *Ausführungszeit* $t_i > 0$; Anfangs- und Endpunkt eines Jobs sollen diesmal aber nicht feststehen – stattdessen soll es eine *Deadline* d_i für jeden Job geben.

Wir nehmen an, dass die Maschine zum Zeitpunkt s bereitsteht. Die Jobs seien mit $1, \ldots, n$ bezeichnet. Jedem Job i ist ein Intervall $[s(i), f(i)]$ zuzuordnen, wobei gelten soll:

$$s \leq s(i) \quad \text{und} \quad f(i) - s(i) = t_i \ (i = 1, \ldots, n).$$

Klar ist: Die Intervalle $[s(i), f(i)]$ sollen sich nicht überlappen. *Was ist nun aber zu optimieren*? Es gibt verschiedene Möglichkeiten, die zu Problemen von unterschiedlichem Schwierigkeitsgrad führen. Wir wollen hier die *Variante* betrachten, *bei der die maximale Verspätung zu minimieren ist.* Genauer:

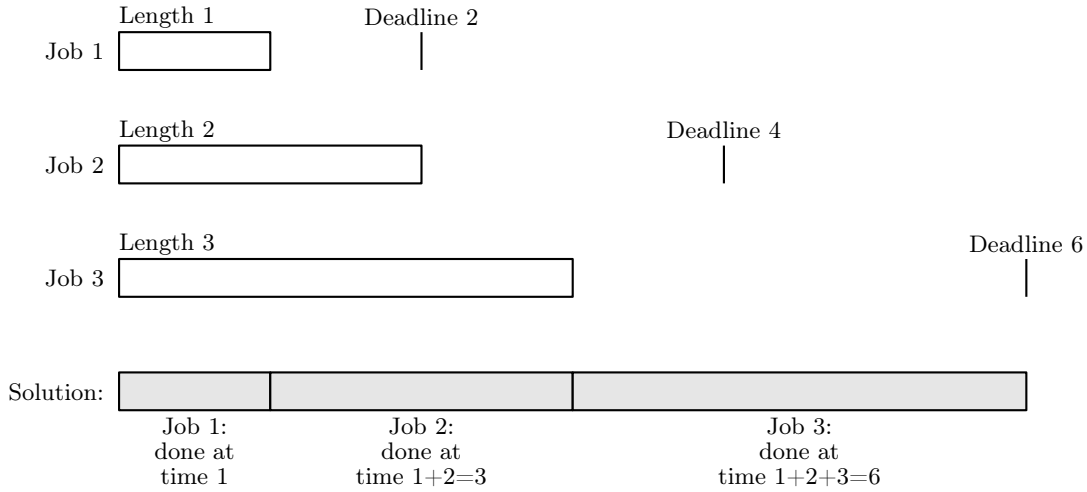
Wir nennen den Job i *verspätet* (engl. *late*), falls $d_i < f(i)$ gilt; wir setzen

$$\ell_i = \begin{cases} f(i) - d_i & \text{, falls der Job } i \text{ verspätet ist;} \\ 0 & \text{, falls der Job } i \text{ nicht verspätet ist}^4. \end{cases}$$

Zu minimieren ist

$$L = \max\Big\{\ell_i : i = 1, \ldots, n\Big\}.$$

Hier ein **Beispiel**, für das $L = 0$ gilt (aus Kleinberg/Tardos):



A sample instance of scheduling to minimize lateness.

Wie könnte ein Greedy-Algorithmus für unser Problem aussehen?

Es bieten sich wieder etliche Möglichkeiten an, den Jobs mithilfe einer einfachen Regel Zeitintervalle zuzuordnen:

(1) Eine dieser Möglichkeiten wäre, die Jobs zunächst nach aufsteigender Länge t_i zu ordnen und ihnen dann in dieser Reihenfolge nach dem Motto „die Kleinen zuerst“ ein Zeitintervall zuzuordnen; zumindest im obigen Beispiel hat das geklappt – die kleinen Jobs so früh wie möglich aus dem Weg zu kriegen, könnte eine gute Idee sein.

Andererseits werden bei dieser Strategie die Deadlines überhaupt nicht berücksichtigt; man braucht sich also nicht zu wundern, dass es „schlechte Beispiele“ bereits dann gibt, wenn nur zwei Jobs im Spiel sind:

$$\begin{array}{ll} \text{1. Job:} & t_1 = 1, d_1 = 100; \\ \text{2. Job:} & t_2 = 10, d_2 = 10. \end{array}$$

(2) Das Beispiel zu (1) legt nahe, mehr auf die „slack time“ $d_i - t_i$ zu achten und die Jobs vorzuziehen, für die $d_i - t_i$ klein ist: Diese Jobs vertragen nur wenig Aufschub und sollten deshalb so schnell wie möglich erledigt werden.

[4] Man nennt ℓ_i die *Verspätung* (engl. *lateness*) des i-ten Jobs.

Leider verfehlt auch diese Greedy-Regel ihr Ziel; man betrachte das folgende Beispiel:

1. Job: $t_1 = 1, d_1 = 2$;
2. Job: $t_2 = 10, d_2 = 10$.

③ Nun kommt eine Regel, die genauso einfach wie die beiden anderen ist, von der sich aber herausstellen wird, dass sie immer eine optimale Lösung liefert: Man ordnet die Jobs nach ihren Deadlines, wobei die frühen Deadlines zuerst drankommen; diese Regel ist unter dem folgenden Namen bekannt:

Earliest Deadline First

So einfach geht das? Und das soll funktionieren? *Man hat allen Grund gegenüber dieser Regel skeptisch zu sein*: Beispielsweise war einer der Einwände gegen die Regel ①, dass sie die Hälfte der Eingangsdaten – die Deadlines d_i – gar nicht berücksichtigt. Und nun soll eine Regel optimal sein, die die andere Hälfte der Eingangsdaten – die Intervalllängen t_i – komplett ignoriert?

Es gibt natürlich Plausibilitätsargumente für die Regel ③: Beispielsweise könnte man argumentieren, dass ein Job, der früh erledigt sein muss, auch früh angefangen werden sollte. Andererseits: Was von solchen Plausibilitätsargumenten zu halten ist, haben wir ja bereits gesehen.

Es soll nun *nachgewiesen* werden, dass die Regel *Earliest Deadline First* immer eine optimale Lösung hervorbringt.

Wir starten damit, dass wir die Jobs neu benennen: Die Bezeichnungen $1, \ldots, n$ für die Jobs sollen in der Reihenfolge der Deadlines vergeben werden, d.h., es soll

$$d_1 \leq \ldots \leq d_n$$

gelten. Unsere Strategie besagt dann, dass Job 1 die Startzeit $s(1) = s$ und die Endzeit $f(1) = s(1) + t_1$ erhält; und dass Job 2 die Startzeit $s(2) = f(1)$ und die Endzeit $f(2) = s(2) + t_2$ bekommt; usw.

Hier ist die Beschreibung des Algorithmus, wie sie im Buch von Kleinberg/Tardos zu finden ist:

(1) Order the jobs in order of their deadlines
(2) Assume for simplicity of notation that $d_1 \leq \ldots \leq d_n$
(3) Initially, $f = s$
(4) **Consider** the jobs $i = 1, \ldots, n$ in this order
(5) Assign job i to the time interval from $s(i) = f$ to $f(i) = f + t_i$
(6) Let $f = f + t_i$
(7) **End**
(8) **Return** the set of scheduled intervals $[s(i), f(i)]$ for $i = 1, \ldots, n$

Als erstes beobachten wir, dass dieser Algorithmus einen Zeitplan (engl. schedule) liefert, bei dem es keine Lücken ("no idle time") gibt. Ferner ist klar, dass es auch einen optimalen[5] Zeitplan ohne Lücken gibt, da Lücken leicht beseitigt werden können, ohne die Optimalität zu zerstören.

Wir nennen den von unserem Greedy-Algorithmus produzierten Zeitplan A; mit $\mathcal{O}$ sei ein optimaler Zeitplan bezeichnet.

Unser Ziel ist, $\mathcal{O}$ durch schrittweise Änderung in A zu überführen, wobei in jedem Änderungsschritt die Optimalität erhalten bleiben soll. Diese Vorgehensweise nennen wir *Austauschargument* (engl. *exchange argument*).

[5] Optimal bedeutet in unserem Fall natürlich immer, dass $L = \max\{\ell_i : i = 1, \ldots, n\}$ so klein wie möglich ist.

Wir sagen, dass in einem gegebenen Zeitplan eine *Inversion* vorkommt, falls es in diesem Zeitplan zwei Jobs i und j gibt, für die gilt: i liegt in diesem Zeitplan vor j, obwohl $d_j < d_i$ gilt.

Man beachte, dass unser Zeitplan A keine Inversionen besitzt. Falls es verschiedene Jobs mit gleicher Deadline gibt, so könnte es außerdem etliche andere Zeitpläne ohne Inversionen und ohne Lücken geben. Wir zeigen nun, dass alle diese Zeitpläne dieselbe maximale Verspätung L besitzen.

Alle Zeitpläne ohne Inversionen und ohne Lücken besitzen dieselbe maximale Verspätung. ($\star$)

Beweis. Falls zwei verschiedene Zeitpläne weder Inversionen noch Lücken besitzen, dann können sie sich nur dadurch unterscheiden, dass Jobs mit gleicher Deadline in unterschiedlicher Reihenfolge ausgeführt werden. Es sei d eine solche Deadline. In beiden Zeitplänen liegen die Jobs mit Deadline d dann lückenlos aneinandergereiht – nur eben in unterschiedlicher Reihenfolge. Unter diesem Jobs mit Deadline d besitzt der letzte die größte Verspätung – und diese ist unabhängig von der Reihenfolge dieser Jobs. □

Wie funktioniert nun das Austauschargument? Wir schildern hier nur die *Grundidee*; die Details findet man im Buch von Kleinberg/Tardos. Man startet von einem optimalen Zeitplan $\mathcal{O}$ ohne Lücken; dass es einen solchen Zeitplan gibt, haben wir bereits festgestellt. $\mathcal{O}$ wird nun schrittweise abgeändert, *wobei in jedem Schritt ein Paar von benachbarten Intervallen die Reihenfolge wechselt und alle anderen Intervalle unverändert bleiben.* Dies geschieht so, dass gilt:

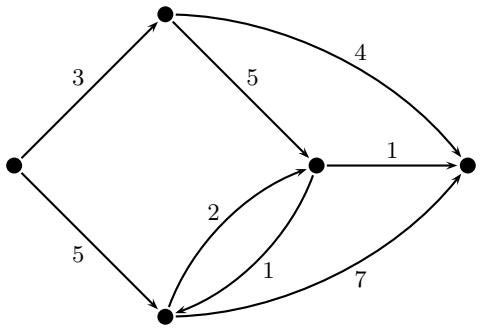
- Die auftretenden Zeitpläne haben niemals Lücken.
- In jedem Schritt sinkt die Anzahl der Inversionen um 1.
- In keinem Schritt vergrößert sich L, d.h., alle auftretenden Zeitpläne bleiben optimal.

Nach endlich vielen Schritten erhält man somit einen optimalen Zeitplan, der keine Inversionen und keine Lücken aufweist.

Damit ist gezeigt, dass ein optimaler Zeitplan ohne Inversionen und ohne Lücken existiert. Da unser Zeitplan A ebenfalls weder Lücken noch Inversionen aufweist, folgt wegen ($\star$), dass auch A optimal ist.

12.3 Kürzeste Pfade in Graphen

Wir betrachten gerichtete Graphen $G = (V, E)$, in denen jeder Kante e eine reelle Zahl $\ell(e) \geq 0$ zugeordnet ist, die wir die *Länge* von e nennen.



Je nach Anwendung kann man sich unter den Zahlen $\ell(e)$ auch etwas anderes als Längen vorstellen, beispielsweise Kosten, Zeitangaben oder Wahrscheinlichkeiten.

Ein grundlegendes algorithmisches Problem: Gegeben seien $u, v \in V$. Finde in G einen kürzesten Pfad von u nach v (sofern ein solcher existiert). Die Länge $\ell(P)$ eines u, v-Pfades P ist dabei wie folgt definiert:

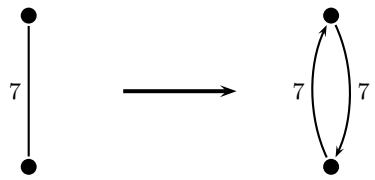
Durchläuft P nacheinander die Kanten $e_1, \ldots, e_t$, so gilt

$$\ell(P) = \sum_{i=1}^{t} \ell(e_i).$$

Wir haben unser „grundlegendes Problem“ als ein „point-to-point“ Problem formuliert. In vielen Fällen will man aber mehr wissen: Man fragt nach kürzesten Pfaden von einem festen Knoten s („Startpunkt“) *zu allen anderen Knoten* v. Wenn nichts anderes gesagt ist, wollen wir voraussetzen, dass es zu jedem Knoten v des betrachteten Graphen mindestens einen s, v-Pfad gibt. Unser Problem lässt sich dann wie folgt formulieren.

Problem der kürzesten Pfade (Version: “one-to-all”).
Gegeben seien ein gerichteter Graph $G = (V, E)$ mit Kantenlängen $\ell(e) \in \mathbb{R}$, $\ell(e) \geq 0$, sowie ein Knoten $s \in V$. Finde kürzeste Pfade von s zu allen anderen Knoten.

Wir haben das Problem der kürzesten Pfade für *gerichtete* Graphen formuliert. Ein entsprechendes Problem für ungerichtete Graphen gesondert zu betrachten, ist nicht nötig, da man den ungerichteten Fall auf den gerichteten zurückführen kann. Dies ist durch einen sehr einfachen Trick zu erreichen: Man ersetzt, wie in der folgenden Zeichnung angedeutet, jede ungerichtete Kante durch zwei gerichtete Kanten derselben Länge.



Der Algorithmus, den wir im Folgenden besprechen, ist der sehr bekannte *Algorithmus von Dijkstra*, bei dem es sich – wie wir noch sehen werden – um einen Greedy-Algorithmus handelt. Wir erwähnen bereits jetzt, dass der Algorithmus nur deshalb korrekt arbeitet, weil wir $\ell(e) \geq 0$ für alle Kanten e vorausgesetzt haben. Der Fall, dass auch Kanten negativer Länge ($\ell(e) < 0$) auftreten, erfordert etwas kompliziertere Methoden – da kommt man mit einem einfachen Greedy-Verfahren nicht mehr zum Ziel. Wir werden diesen Fall später behandeln, wenn wir im Kapitel über *Dynamisches Programmieren* den *Algorithmus von Bellman und Ford* sowie den *Algorithmus von Floyd und Warshall* kennenlernen.

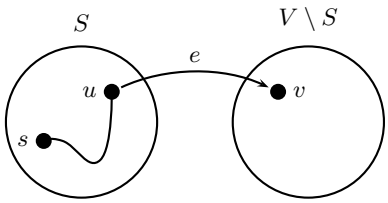
Wir beschreiben den Algorithmus von Dijkstra in einer Variante, bei der *nur die Längen der kürzesten Pfade* ermittelt werden. Es ist jedoch einfach, den Algorithmus so zu ergänzen, dass auch die kürzesten Pfade selbst mitgeliefert werden.

Im Algorithmus von Dijkstra spielt eine Menge $S \subseteq V$ eine wichtige Rolle: Wir nennen S den *bereits erforschten Teil des Graphen* (“set of explored nodes”), da in jeder Phase des Algorithmus für die Knoten u der aktuellen Menge S ein Wert $d(u)$ vorliegt, von dem wir nachweisen werden, dass er die Länge eines kürzesten s, u-Pfades von G bereits korrekt angibt. Darüber hinaus gilt sogar, dass es einen solchen Pfad gibt, der ganz in S verläuft.

Am Anfang gilt $S = \{s\}$ und $d(s) = 0$; im Laufe des Algorithmus kommen dann schrittweise Knoten zu S hinzu, bis am Ende $S = V$ gilt. Genauer: Gilt $S \neq V$, so betrachtet man diejenigen Knoten $v \in V \setminus S$, die von S aus direkt erreichbar sind, d.h., zu denen mindestens eine Kante $e = (u, v)$ mit $u \in S$ führt. Für jeden solchen Knoten $v \in V \setminus S$ stellt man sich vor, dass die minimale Länge eines s, v-Pfades zu berechnen ist, der – abgesehen von v und seiner letzten Kante $e = (u, v)$ – ganz in S verläuft (siehe Zeichnung). Dementsprechend betrachtet man die Größe

$$d'(v) = \min \Big\{ d(u) + \ell(u, v) : u \in S \text{ und } (u, v) \in E \Big\}.$$

In Worten: Für alle $u \in S$, von denen eine Kante $(u, v) \in E$ zu v führt, betrachtet man die Summe $d(u) + \ell(u, v)$; $d'(v)$ ist dann der kleinste unter den betrachteten Werten.



Im Algorithmus von Dijkstra wird nun dasjenige $v \in V \setminus S$ ausgewählt, für das $d'(v)$ so klein wie möglich ist; dieses v wird zu S hinzugefügt und $d(v)$ wird dadurch definiert, dass man $d(v) = d'(v)$ setzt.

Es ergibt sich der folgende Algorithmus (Darstellung nach Kleinberg/Tardos):

Dijkstra's Algorithm $(\mathbf{G}, \ell, \mathbf{s})$

(1) Let S be the set of explored nodes
(2) For each $u \in S$, we store a distance $d(u)$
(3) Initially $S = \{s\}$ and $d(s) = 0$
(4) **While** $S \neq V$
(5) Select a node $v \in V \setminus S$ with at least one edge from S to v such that
$d'(v) = \min\{d(u) + \ell(u, v) : u \in S \text{ and } (u, v) \in E\}$ is as small as possible
(6) Add v to S and define $d(v) = d'(v)$
(7) **EndWhile**

Am Ende liefert der Algorithmus von Dijkstra zu jedem $v \in V$ einen Wert $d(v)$. Noch wissen wir nicht, ob $d(v)$ tatsächlich die Länge eines kürzesten s, v-Pfades von G angibt – das wird erst später nachgewiesen werden. Wir wollen uns zunächst nur davon überzeugen, dass es überhaupt einen s, v-Pfad P_v in G gibt, der die Länge $d(v)$ besitzt. Dies ist nicht schwer einzusehen; man erhält einen solchen Pfad P_v für $v \neq s$ dadurch, dass man im Algorithmus von Dijkstra eine leichte Erweiterung vornimmt: Bei Aufnahme von $v \neq s$ in die Menge S speichert man zusätzlich immer eine Kante (u, v), die für die Aufnahme von v in S „verantwortlich" ist, d.h., man speichert eine Kante (u, v) mit $u \in S$, für die $d(u) + \ell(u, v)$ minimal ist.

Um einen s, v-Pfad P_v der Länge $d(v)$ für $v \neq s$ zu erhalten, braucht man dann nur noch die entsprechenden Kanten rückwärts zu durchlaufen: Man startet in v und durchläuft die für v gespeicherte Kante (u, v) rückwärts; dadurch kommt man von v zum Knoten u, der früher als v in S aufgenommen wurde. Falls $u \neq s$, so durchläuft man danach die zu u gespeicherte Kante (w, u) rückwärts und kommt zum Knoten w, der noch früher in S aufgenommen wurde. Dies führt man fort, bis man bei s angelangt ist. Durchläuft man diese Kanten umgekehrt von s nach v, so erhält man den gewünschten s, v-Pfad P_v der Länge $d(v)$.

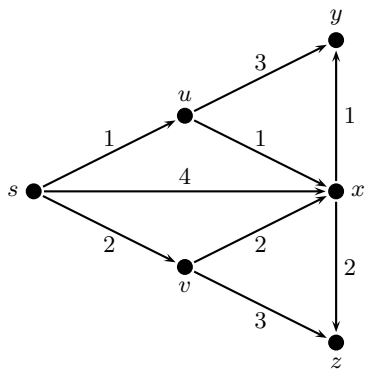
Man definiert zusätzlich den Pfad P_s als den nur aus s bestehenden Pfad der Länge 0.

Übungsaufgabe. Man beweise durch vollständige Induktion nach der Anzahl k der Kanten von P_v, dass P_v tatsächlich die Länge $d(v)$ besitzt. (*Hinweis*: Gilt $v \neq s$, so betrachte man die letzte Kante von P_v.)

Zu jedem vom Algorithmus gelieferten Wert $d(v)$ gehört also ein s, v-Pfad P_v, der die Länge $d(v)$ besitzt und den man wie beschrieben erhält. Weiter unten werden wir uns davon überzeugen, dass dieser s, v-Pfad P_v sogar immer ein *kürzester s, v-Pfad von G* ist. Wir halten dieses Ergebnis bereits hier fest:

Der vom Algorithmus von Dijkstra gelieferte Wert $d(v)$ gibt für jeden Knoten v die Länge eines kürzesten s, v-Pfades in G an und der dazugehörige Pfad P_v ist ein solcher kürzester Pfad. (12.2)

Wir schauen uns den Ablauf des Algorithmus von Dijkstra anhand eines **Beispiels** an; G sei der folgende Graph:



Initialisierung:

Es sei $S = \{s\}$ und $d(s) = 0$.

1. Durchlauf der While-Schleife:

Die Knoten aus $V \setminus S$, zu denen Kanten aus S führen, sind u, v, x; man erhält $d'(u) = 1$, $d'(v) = 2$, $d'(x) = 4$. Es folgt $S = \{s, u\}$ und $d(u) = 1$.

2. Durchlauf der While-Schleife:

Die Knoten aus $V \setminus S$, zu denen Kanten aus S führen, sind v, x, y; man erhält $d'(v) = 2$, $d'(x) = 2$, $d'(y) = 4$. Man kann v oder x in S aufnehmen; wir entscheiden uns (willkürlich) für v. Es folgt $S = \{s, u, v\}$ und $d(v) = 2$.

3. Durchlauf der While-Schleife:

Die Knoten aus $V \setminus S$, zu denen Kanten aus S führen, sind x, y, z; man erhält $d'(x) = 2$, $d'(y) = 4$, $d'(z) = 5$. Es folgt $S = \{s, u, v, x\}$ und $d(x) = 2$.

4. Durchlauf der While-Schleife:

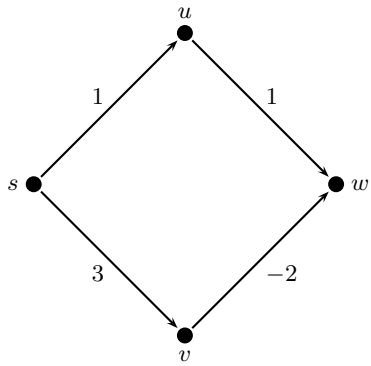
Die Knoten aus $V \setminus S$, zu denen Kanten aus S führen, sind y und z; man erhält $d'(y) = 3$, $d'(z) = 4$. Es folgt $S = \{s, u, v, x, y\}$ und $d(y) = 3$.

5. Durchlauf der While-Schleife:

Für den verbliebenen Knoten z erhält man $d'(z) = 4$. Es folgt $S = \{s, u, v, x, y, z\}$ und $d(z) = 4$.

Der Algorithmus von Dijkstra ist ein Greedy-Algorithmus, da er in jedem Schritt *kurzsichtig* vorgeht: Wenn es darum geht, S zu erweitern, so schaut sich der Algorithmus nur Kanten an, die direkt von S ausgehen – gerade so, als ob er Kanten, die weiter von S entfernt sind, nicht erkennen könnte.

Bei so viel Kurzsichtigkeit braucht man sich nicht zu wundern, dass der Algorithmus im Allgemeinen keine korrekten Ergebnisse liefert, wenn Kanten negativer Länge im Spiel sind. Hier ein **Beispiel**, an dem zu erkennen ist, was im Falle negativer Kantenlängen schiefgehen kann:



Wir setzen nun wieder $\ell(e) \geq 0$ für alle Kanten e voraus und kommen zum Beweis von (12.2) („Korrektheitsbeweis für den Algorithmus von Dijkstra“). Da der Algorithmus von Dijkstra der vielleicht bekannteste Algorithmus zum Berechnen von kürzesten Pfaden in Graphen ist, findet man Beweise für seine Korrektheit in vielen Lehrbüchern. Wir schauen uns die Darstellung von Kleinberg und Tardos an:

$$\begin{array}{c} \textit{Consider the set } S \textit{ at any point in the algorithm's execution.} \\ \textit{For each } u \in S\textit{, the path } P_u \textit{ is a shortest } s-u \textit{ path.} \end{array} \tag{4.14}$$

Note that this fact immediately establishes the correctness of Dijkstra's algorithm, since we can apply it when the algorithm terminates, at which point S includes all nodes.

Proof. We prove this by induction on the size of S. The case $|S| = 1$ is easy, since then we have $S = \{s\}$ and $d(s) = 0$. Suppose the claim holds when $|S| = k$ for some value of $k \geq 1$; we now grow S to size $k+1$ by adding the node v. Let (u, v) be the final edge on our $s-v$ path P_v.

By induction hypothesis, P_u is the shortest $s-u$ path for each $u \in S$. Now consider any other $s-v$ path P; we wish to show that it is at least as long as P_v. In order to reach v, this path P must leave the set S *somewhere*; let y be the first node on P that is not in S, and let $x \in S$ be the node just before y.

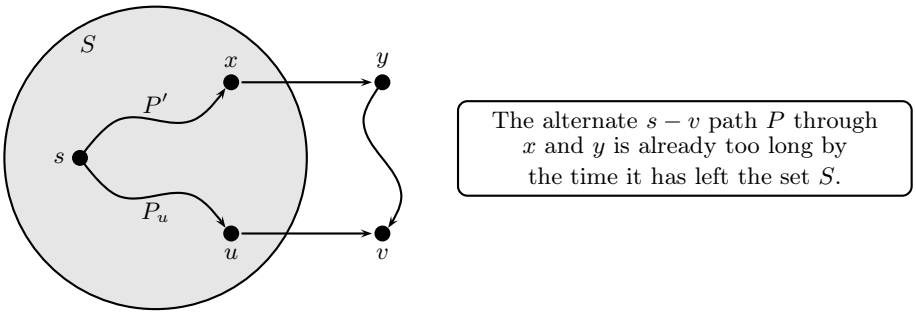


Figure 4.8 The shortest path P_v and an alternate $s-v$ path P through the node y.

The situation is now as depicted in Figure 4.8, and the crux of the proof is very simple: P cannot be shorter than P_v because it is already at least as long as P_v by the time it has left the set S. Indeed, in iteration $k+1$, Dijkstra's Algorithm must have considered adding node y to the set S via the edge (x, y) and rejected this option in favor of adding v. This means that there is no path from s to y through x that is shorter than P_v. But the subpath of P up to y is such a path, and so this subpath is at least as long as P_v. Since edge length are nonnegative, the full path P is at least as long as P_v as well.

This is a complete proof; one can also spell out the argument in the previous paragraph using the following inequalities. Let P' be the subpath of P from s to x. Since $x \in S$, we know by the induction hypothesis that P_x is a shortest $s-x$ path (of length $d(x)$), and so $\ell(P') \geq \ell(P_x) = d(x)$. Thus the subpath of P out to node y has length $\ell(P') + \ell(x, y) \geq d(x) + \ell(x, y) \geq d'(y)$, and the full path P is at least as long as this subpath. Finally, since Dijkstra's Algorithm selected v in this iteration, we know that $d'(y) \geq d'(v) = \ell(P_v)$. Combining these inequalities shows that $\ell(P) \geq \ell(P') + \ell(x, y) \geq \ell(P_v)$. □

Der obige Beweis der Korrektheit des Algorithmus von Dijkstra läuft übrigens nach dem Schema “the greedy algorithm stays ahead” ab: Der Algorithmus von Dijkstra baut die Gesamtlösung schrittweise auf und im Beweis wird gezeigt, dass die vom Dijkstra-Algorithmus in diesem Prozess gelieferten Teillösungen für die Mengen S bereits optimal sind (und deshalb von keinem anderen Algorithmus übertroffen werden können). Anders gesagt: Es wird gezeigt, dass der Dijkstra-Algorithmus für jede Menge S „vorne liegt“.

Bei der auf Seite 178 präsentierten Darstellung des Dijkstra-Algorithmus fehlt noch ein *wichtiges Detail*: Die Frage, wie man in jedem Durchlauf der While-Schleife den Knoten v auf eine effiziente Art findet,

wurde bislang ausgespart. Auf den ersten Blick hat man den Eindruck, dass man wie folgt vorgehen müsste: Für jeden Knoten $v \notin S$, zu dem mindestens eine von S ausgehende Kante hinführt, hat man für sämtliche Kanten $e = (u, v)$ mit $u \in S$ den Wert $d(u) + \ell(e)$ zu bilden und v das Minimum $d'(v)$ all dieser Werte zuzuordnen. Unter allen $v \notin S$, denen auf diese Art ein $d'(v)$ zugeordnet wurde, wählt man sich dann einen Knoten v, für den $d'(v)$ so klein wie möglich ist. Dieses v nimmt man zu S hinzu, setzt $d(v) = d'(v)$ und steigt ggf. in den nächsten Durchlauf der While-Schleife ein.

Im nächsten Durchlauf der While-Schleife könnte man nun ganz entsprechend vorgehen; dabei würde man jedoch merken, dass etliche Rechnungen, die bereits im vorangegangenen Durchlauf der While-Schleife durchgeführt wurden, nur wiederholt werden. In manchen Fällen würden sich die im vorangegangenen Durchlauf berechneten Werte nicht ändern:

- Wurde beim vorangegangenen Durchlauf der While-Schleife v zu S hinzugefügt und ist für einen Knoten $w \notin S$ die Kante (v, w) nicht vorhanden, so ändert sich der Wert $d'(w)$ gegenüber dem vorangegangenen Durchlauf nicht, da es immer noch genau dieselben Kanten $e = (u, w)$ sind, die zur Berechnung von $d'(w)$ herangezogen werden.
- Und wie ändert sich der Wert $d'(w)$ für ein $w \notin S$, wenn die Kante (v, w) vorhanden ist? Offenbar hat man den alten Wert $d'(w)$ mit dem Wert $d(v) + \ell(v, w)$ zu vergleichen und hierbei nach der folgenden Update-Formel vorzugehen:

$$d'(w) = \min\Big\{d'(w), d(v) + \ell(v, w)\Big\}.$$

Es wäre also unökonomisch, die erhaltenen Werte $d'(w)$ am Ende des vorangegangenen Durchlaufs der While-Schleife einfach „wegzuschmeißen“. Diese Überlegungen führen zur folgenden *modifizierten Version des Algorithmus von Dijkstra*.

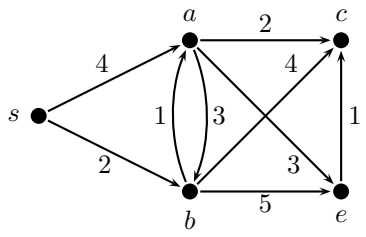
Dijkstra's Algorithm (modified version) $(\mathbf{G}, \ell, \mathbf{s})$

(1) Let S be the set of explored nodes
(2) For each $u \in V$, we store a value $d(u)$
(3) Initially $S = \big\{s\big\}$ and $d(s) = 0$
(4) For each $u \neq s$ let $d(u) = \ell(s, u)$ if $(s, u) \in E$ and $d(u) = \infty$ otherwise
(5) **While** $S \neq V$
(6) Select a node $v \in V \setminus S$ with $d(v) = \min\big\{d(u) : u \in V \setminus S\big\}$
(7) Add v to S
(8) For each $u \in V \setminus S$ with $(v, u) \in E$, let $d(u) = \min\big\{d(u), d(v) + \ell(v, u)\big\}$
(9) **EndWhile**

Die Laufzeit dieser Version des Dijkstra-Algorithmus ist $O(n^2)$ für $n = |V|$, da die While-Schleife $n - 1$ mal durchlaufen wird und die Zeilen (6)-(8) offenbar in $O(n)$ Zeit ausgeführt werden können. Liegt ein dichtbesetzter Graph vor, d.h., für $m = |E|$ gilt $m = \Theta(n^2)$, so ist die Laufzeitschranke $O(n^2)$ bestmöglich, da jede Kante inspiziert werden muss. Ist m dagegen asymptotisch kleiner als n^2, so lassen sich durch Einsatz geeigneter Datenstrukturen Laufzeitverbesserungen erreichen (vgl. etwa Kleinberg/Tardos, Seite 141f). Stichwort hierzu: Prioritätswarteschlange.

Bevor wir die Arbeitsweise des Algorithmus anhand eines Beispiels illustrieren, noch ein Wort zu den Werten $d(u)$: Für $u \in S$ gibt $d(u)$ wie bisher die Länge eines kürzesten s, u-Pfades in G an. Für Knoten $u \notin S$, zu denen mindestens eine von S ausgehende Kante hinführt, bezeichnet $d(u)$ die Größe, die wir bislang $d'(u)$ genannt haben; für die übrigen Knoten $u \notin S$ gilt $d(u) = \infty$. Für alle $u \in V \setminus S$ gibt $d(u)$ somit eine obere Schranke für die Länge eines kürzesten s, u-Pfades in G an; der Wert $d(u)$ kann in diesem Fall noch verändert werden; er ist *vorläufig*. Nach Aufnahme von u in S ändert sich der Wert von $d(u)$ nicht mehr – er ist *endgültig* und gibt den Abstand von s und u in G an.

Beispiel. Der Graph $G = (V, E)$ mit Längenfunktion ℓ sei durch die folgende Zeichnung gegeben:



Wir geben für jeden Knoten u an, welchen Wert $d(u)$ nach der Initialisierung und nach jedem Durchlauf der While-Schleife besitzt. Außerdem wird das jeweils aktuelle S angegeben. Ist ein Wert $d(u)$ endgültig (man sagt auch *permanent*), so wird er unterstrichen.

Initialisierung:

$$S = \Big\{ s \Big\}$$

u	s	a	b	c	e
$d(u)$	$\underline{0}$	4	2	∞	∞

1. Durchlauf der Schleife:

$$S = \Big\{ s, b \Big\}$$

u	s	a	b	c	e
$d(u)$	$\underline{0}$	3	$\underline{2}$	6	7

2. Durchlauf der Schleife:

$$S = \Big\{ s, b, a \Big\}$$

u	s	a	b	c	e
$d(u)$	$\underline{0}$	$\underline{3}$	$\underline{2}$	5	6

3. Durchlauf der Schleife:

$$S = \Big\{ s, b, a, c \Big\}$$

u	s	a	b	c	e
$d(u)$	$\underline{0}$	$\underline{3}$	$\underline{2}$	$\underline{5}$	6

4. Durchlauf der Schleife:

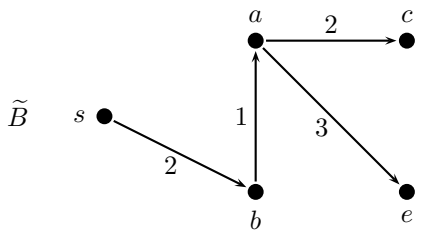
$$S = \Big\{ s, b, a, c, e \Big\}$$

u	s	a	b	c	e
$d(u)$	$\underline{0}$	$\underline{3}$	$\underline{2}$	$\underline{5}$	$\underline{6}$

Übersichtliche Zusammenfassung:

	s	a	b	c	e	S
0	$\underline{0}$	4	2	∞	∞	$\{s\}$
1	$\underline{0}$	3	$\underline{2}$	6	7	$\{s, b\}$
2	$\underline{0}$	$\underline{3}$	$\underline{2}$	5	6	$\{s, b, a\}$
3	$\underline{0}$	$\underline{3}$	$\underline{2}$	$\underline{5}$	6	$\{s, b, a, c\}$
4	$\underline{0}$	$\underline{3}$	$\underline{2}$	$\underline{5}$	$\underline{6}$	$\{s, b, a, c, e\}$

Einen kürzeste-Pfade-Baum $\widetilde{B}$ kann man leicht am Graphen ablesen:



Eine Bemerkung zur Terminologie: Unter einem *Baum* versteht man bekanntlich einen ungerichteten Graphen, der kreislos und zusammenhängend ist. Im Sinne dieser Definition ist der „kürzeste-Pfade-Baum“ $\widetilde{B}$ (streng genommen) gar kein Baum, da $\widetilde{B}$ ein gerichteter Graph ist. Die Bezeichnung von $\widetilde{B}$ als „kürzester-Pfade-Baum“ ist trotzdem üblich: Das rechtfertigt sich dadurch, dass es einen Baum B mit Wurzel s gibt, von dem $\widetilde{B}$ „abstammt“. Das soll heißen: $\widetilde{B}$ entsteht aus B dadurch, dass man alle Kanten von s weg orientiert.

Um einen kürzeste-Pfade-Baum *systematisch* zu finden, ist der Algorithmus von Dijkstra zu erweitern: Für $u \neq s$ speichert man im Fall $d(u) \neq \infty$ nicht nur den Wert $d(u)$, sondern auch einen dazugehörigen „Vorgängerknoten“ w aus S, d.h., nach dem i-ten Durchlauf der While-Schleife gilt $d(u) = d(w) + \ell(w,u)$ für den angegebenen Knoten $w \in S$. Im Beispiel sieht das so aus:

	s	a	b	c	e	S
0	$\underline{0}$	4 s	2 s	∞	∞	$\{s\}$
1	$\underline{0}$	3 b	$\underline{2}$ s	6 b	7 b	$\{s,b\}$
2	$\underline{0}$	$\underline{3}$ b	$\underline{2}$ s	5 a	6 a	$\{s,b,a\}$
3	$\underline{0}$	$\underline{3}$ b	$\underline{2}$ s	$\underline{5}$ a	6 a	$\{s,b,a,c\}$
4	$\underline{0}$	$\underline{3}$ b	$\underline{2}$ s	$\underline{5}$ a	$\underline{6}$ a	$\{s,b,a,c,e\}$

Anhand der letzten Zeile kann man zu jedem $u \neq s$ einen Vorgänger auf einem kürzesten s,u-Pfad ablesen.

Auf Seite 141f findet man im Buch von Kleinberg und Tardos weitere Ausführungen zum Algorithmus von Dijkstra, vor allen Dingen zur Verbesserung der Laufzeit und, damit zusammenhängend, zum Einsatz einer *Prioritätswarteschlange* (engl. *priority queue*) bei der Implementierung des Algorithmus von Dijkstra. *Empfehlung*: Schauen Sie sich diese Ergänzungen an.

12.4 Minimale aufspannende Bäume

Wir befassen uns in diesem Abschnitt mit einem Problem, das ebenso grundlegend wie das zuvor behandelte Problem der kürzesten Pfade ist.

Gegeben sei eine Menge V von n Knoten, sagen wir $V = \{v_1, \ldots, v_n\}$. Wir stellen uns vor, dass zwischen diesen Knoten ein *Kommunikationsnetzwerk* eingerichtet werden soll. Dabei muss es nicht unbedingt für alle Paare v_i, v_j möglich sein, eine direkte Verbindung einzurichten – für gewisse Paare v_i, v_j von verschiedenen Knoten soll dies aber infrage kommen, wobei der Aufbau einer solchen direkten Verbindung *Kosten* verursacht.

Kommunikationsverbindungen sollen immer in beide Richtungen nutzbar sein, d.h., zur Modellierung werden wir ungerichtete Graphen verwenden.

Das zu errichtende Kommunikationsnetzwerk soll natürlich *zusammenhängend* sein, d.h., zwischen zwei verschiedenen Knoten v_i und v_j soll es immer einen Pfad geben, der v_i mit v_j verbindet. *Das Ziel ist, die Gesamtkosten zu minimieren.* Wir gelangen zu folgendem *Problem.*

Gegeben: ein zusammenhängender Graph $G = (V, E)$; jeder Kante $e \in E$ sei eine reelle Zahl $c(e) > 0$ zugeordnet („Kosten von e“).

Gesucht: eine Teilmenge F der Kantenmenge E, so dass der Graph (V, F) zusammenhängend ist und außerdem die Gesamtkosten

$$\sum_{e \in F} c(e)$$

minimal sind.

Wir erinnern noch einmal an die Definition eines Baums.

Definition.

Unter einem *Baum* versteht man einen ungerichteten Graphen, der zusammenhängend und kreislos ist.

Wir schließen eine *einfache Beobachtung* an.

Feststellung.

Es sei (V, F) eine Lösung unseres Problems, d.h., der Graph (V, F) ist zusammenhängend und die Gesamtkosten $\sum_{e \in F} c(e)$ sind minimal. Dann ist (V, F) ein Baum. $(\star\star)$

Beweis. Da (V, F) zusammenhängend ist, muss nur noch gezeigt werden, dass (V, F) kreislos ist. Angenommen, (V, F) enthielte einen Kreis C. Dann könnte man eine beliebige Kante f von C weglassen: $(V, F \setminus \{f\})$ wäre immer noch ein zusammenhängender Graph und wegen $c(f) > 0$ wären die Gesamtkosten für $(V, F \setminus \{f\})$ kleiner als $\sum_{e \in F} c(e)$, d.h., $\sum_{e \in F} c(e)$ wäre nicht minimal. Dieser Widerspruch zeigt, dass (V, F) kreislos ist. □

Ist $G = (V, E)$ ein Graph und $T = (V, F)$ ein Baum mit $F \subseteq E$, so nennt man T einen *aufspannenden Baum* von G. Anders gesagt: Ein aufspannender Baum von G ist ein Teilgraph von G, der alle Knoten von G enthält und bei dem es sich um einen Baum handelt.

Die Feststellung ($\star\star$) besagt, dass als Lösung unseres Problems nur aufspannende Bäume infrage kommen. Gesucht ist also ein *minimaler aufspannender Baum* von G (engl. *minimum spanning tree*, kurz: *MST*).

Ein Graph G hat – wenn man einmal von sehr einfachen Fällen absieht – in der Regel sehr viele aufspannende Bäume. Beispielsweise besagt ein bekannter Satz von Cayley, dass ein vollständiger Graph mit n Knoten genau

$$n^{n-2}$$

aufspannende Bäume besitzt. Es ist also auf den ersten Blick alles andere als klar, wie man unter all diesen Bäumen einen Baum $T = (V, F)$ finden soll, für den

$$\sum_{e \in F} c(e)$$

minimal ist.

12.4.1 Algorithmen für das Minimum Spanning Tree Problem

In früheren Beispielen aus dem Gebiet des Schedulings haben wir gesehen, wie leicht es ist, sich eine Reihe von natürlich erscheinenden Greedy-Algorithmen auszudenken, für die es in jedem Einzelfall durchaus einleuchtende Plausibilitätsargumente gibt. Häufig funktionierten diese Algorithmen jedoch nicht.

Auch beim Problem des minimalen aufspannenden Baums ist es möglich, sich etliche Greedy-Algorithmen einfallen zu lassen – diesmal ist es aber erstaunlicherweise so, dass viele der Greedy-Algorithmen, die einem in den Sinn kommen, auch tatsächlich funktionieren. Für das Problem des minimalen aufspannenden Baums kann demnach völlig zu Recht behauptet werden: “Greed works.”

Hier sind *drei einfache Greedy-Algorithmen* für das Problem des minimalen aufspannenden Baums, die alle drei korrekt arbeiten, d.h., einen minimalen aufspannenden Baum auch tatsächlich finden. Wir beschreiben zunächst diese Algorithmen und kümmern uns erst danach um die Frage nach ihrer Korrektheit.

- Der erste dieser drei Greedy-Algorithmen geht wie folgt vor. Zunächst werden die Kanten von E nach ihren Kosten sortiert, und zwar in aufsteigender Reihenfolge: $e_1, \ldots, e_m$ mit $c(e_i) \leq c(e_{i+1})$ $(i = 1, \ldots, m-1)$. Man startet nun mit dem kantenlosen Graphen $(V, \emptyset)$, geht die aufsteigend sortierten Kanten von links nach rechts durch und stellt jedes Mal die Frage: *Entsteht ein Kreis, wenn e_i zum bereits konstruierten Teilgraphen hinzugenommen wird*? Falls ja, so wird e_i zurückgewiesen; falls nein, so wird e_i zum bereits konstruierten Teilgraphen hinzugefügt. Diese Vorgehensweise wird **Kruskals Algorithmus** genannt.
- Ein anderer, ebenso einfacher Algorithmus weist große Ähnlichkeit mit Dijkstras Algorithmus zum Auffinden kürzester Pfade auf. Man startet mit einem einzelnen Knoten s („Startknoten", „Wurzel") und baut ausgehend von s einen Baum auf, indem man in jedem Schritt eine weitere Kante $e = \{u, v\}$ hinzunimmt, die genau einen Knoten mit dem bereits konstruierten Teilbaum gemeinsam hat; dabei wird e jedes Mal so gewählt, dass $c(e)$ möglichst klein ist. Dieses Verfahren wird **Prims Algorithmus** genannt.
- Der dritte diese Algorithmen kann als eine „Rückwärtsversion" von Kruskals Algorithmus angesehen werden. Zunächst werden die Kanten von E wieder nach ihren Kosten sortiert, diesmal aber in absteigender Reihenfolge: $e_1, \ldots, e_m$ mit $c(e_i) \geq c(e_{i+1})$ $(i = 1, \ldots, m-1)$. Man startet mit dem Graphen $G = (V, E)$, geht die Kanten in der Reihenfolge $e_1, \ldots, e_m$ durch und stellt jedes Mal die Frage: *Bleibt der aktuelle Teilgraph zusammenhängend, wenn man die Kante e_i entfernt*? Falls ja, so wird e_i aus dem aktuellen Teilgraphen entfernt; falls nein, so bleibt e_i im aktuellen Teilgraphen. Diese Methode wird **Reverse-Delete-Algorithmus** genannt.

Woran liegt es nun, dass alle drei beschriebenen Algorithmen korrekt arbeiten, d.h. einen minimalen aufspannenden Baum liefern? Die Antwort, die sich aufgrund der nachfolgenden Analyse ergeben wird, ist, dass ein *Austauschargument* dahinter steckt.

Genauer: Um die Korrektheit der Algorithmen nachzuweisen, werden wir zunächst einen Hilfssatz vorstellen, der für unsere Zwecke sehr nützlich sein wird. Schaut man sich den Beweisteil (d) $\Rightarrow$ (a) dieses Hilfssatzes an, so erkennt man: Hier geht es um ein Austauschargument.

12.4.2 Analyse der Algorithmen

Alle drei Algorithmen arbeiten ähnlich, nämlich mit wiederholtem Hinzu- bzw. Wegnehmen von Kanten. Um diese Vorgehensweise zu analysieren fragen wir:

- Kann man beim Algorithmus von Kruskal bzw. Prim sicher sein, dass man nichts falsch macht, wenn man eine Kante zu einer bereits existierenden Teillösung hinzunimmt?
- Kann man beim Reverse-Delete-Algorithmus sicher sein, dass man keine Kante vorschnell aussortiert hat?

Die Antworten auf die aufgeworfenen Fragen werden sich aus dem Folgenden ergeben, wobei wir den Schwerpunkt auf die Analyse der *Algorithmen von Kruskal und Prim* legen.

Wir werden größtenteils nach dem folgenden Lehrbuch vorgehen:

- B. Korte, J. Vygen:

  Combinatorial Optimization. Theory and Algorithms. Springer. 2012. 5. Auflage.

Das Problem, um das es geht, lässt sich wie folgt beschreiben:

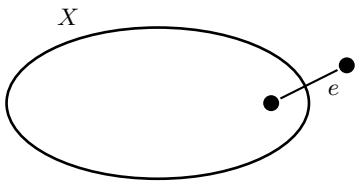
MINIMUM-SPANNING-TREE-PROBLEM

Instanz: Ein ungerichteter, zusammenhängender Graph G mit Kantengewichten $c(e) > 0$.

Aufgabe: Bestimme einen aufspannenden Baum von G mit minimalem Gewicht.

Wir beginnen mit der *Zusammenstellung einiger Bezeichnungen.*

Ist ein Graph G gegeben, so bezeichnen wir die Knotenmenge von G mit $V(G)$; $E(G)$ bezeichnet die Kantenmenge von G. Für $X \subseteq V(G)$ sei mit $\delta(X)$ die Menge derjenigen Kanten $e \in E(G)$ bezeichnet, für die gilt: Genau ein Endpunkt von e liegt in X (siehe Zeichnung).



Man kann die Menge $\delta(X)$ auch so beschreiben:

$$\delta(X) = \Big\{ e \in E(G) : |e \cap X| = 1 \Big\}.$$

Soll betont werden, dass wir uns auf den Graphen G beziehen, so schreiben wir $\delta_G(X)$ anstelle von $\delta(X)$.

Für $e \in E(G)$ bezeichnet $G - e$ den Graphen, den man aus G dadurch erhält, dass man die Kante e löscht. Falls eine neue Kante e zu G hinzugenommen wird, so wird der entstehende Graph mit $G + e$ bezeichnet.

Mit c bezeichnen wir die Funktion, die jeder Kante $e \in E(G)$ ihr Gewicht zuordnet. Ist T ein minimaler aufspannender Baum, so sagen wir hierfür kurz: T ist *optimal*. Im Folgenden sei (wie üblich) $n = |V(G)|$ und $m = |E(G)|$.

Es werden nun drei *Optimalitätsbedingungen* vorgestellt, die wir anschließend benutzen werden, um die Korrektheit der Algorithmen von Kruskal und Prim sowie des Reverse-Delete-Algorithmus nachzuweisen.

Hilfssatz.

Es sei (G, c) eine Instanz des MINIMUM-SPANNING-TREE-PROBLEMS und T sei ein aufspannender Baum von G. Dann sind die folgenden vier Aussagen äquivalent:

(a) T ist optimal.

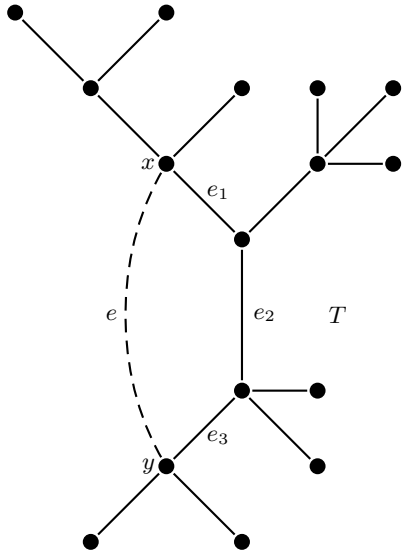
(b) Für jede Kante $e = \{x, y\} \in E(G) \setminus E(T)$ gilt: Keine Kante des x, y-Pfades in T hat höheres Gewicht als e.

(c) Für jedes $e \in E(T)$ gilt: Ist B eine der beiden Zusammenhangskomponenten von $T - e$, so ist e eine Kante aus $\delta(V(B))$, die unter allen Kanten aus $\delta(V(B))$ minimales Gewicht hat.

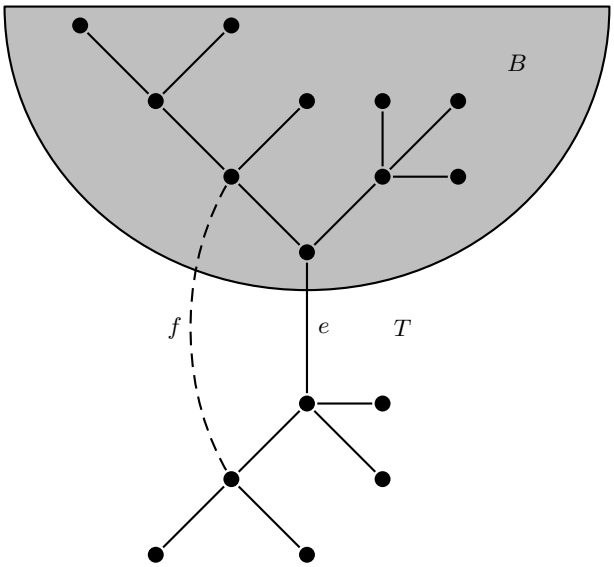
(d) Es existiert eine Reihenfolge $e_1, \ldots, e_{n-1}$ der Kanten von T mit der Eigenschaft, dass es zu jeder der Kanten e_i eine Menge $X \subseteq V(G)$ gibt, die Folgendes erfüllt:
- $e_i \in \delta(X)$ und unter allen Kanten aus $\delta(X)$ hat e_i minimales Gewicht;
- $e_j \notin \delta(X)$ für alle $j \in \{1, \ldots, i-1\}$.

Bevor wir diesen Satz beweisen, schauen wir uns die Bedingungen (b), (c) und (d) genauer an. Wir beginnen mit (b).

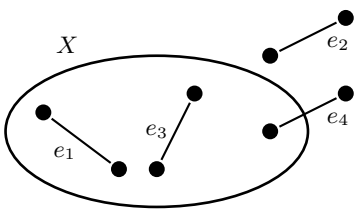
In der folgenden Zeichnung wird die Bedingung (b) veranschaulicht: Der x, y-Pfad in T besteht aus den Kanten e_1, e_2, e_3. Ist (b) erfüllt, so gilt $c(e_i) \leq c(e)$ für alle drei Kanten e_i.



In der nächsten Figur wird die Bedingung (c) illustriert: Ist (c) erfüllt, so gilt $c(e) \leq c(f)$ für alle Kanten $f \in \delta(V(B))$.



Die folgende Figur dient der Illustration der Bedingung (d). Dabei wird angenommen, dass eine Reihenfolge $e_1, \ldots, e_{n-1}$ wie in (d) gegeben ist; e_1, e_2, e_3, e_4 seien die ersten vier Kanten dieser Reihenfolge und es gelte $i = 4$. Die Menge $X \subseteq V(G)$ sei die zu e_4 gehörige Menge. Es gilt $e_4 \in \delta(X)$, aber $e_1, e_2, e_3 \notin \delta(X)$. Außerdem hat e_4 minimales Gewicht unter allen Kanten aus $\delta(X)$.



In der Bedingung (b) steht der x, y-Pfad von T im Mittelpunkt. Deshalb wollen wir (b) die *Pfadbedingung* nennen. In der Bedingung (c) geht es vor allem um einen Schnitt von G, nämlich um den Schnitt $(V(B), V(G) \setminus V(B))$[6]. Deshalb wollen wir (c) die *Schnittbedingung* nennen. Auch (d) soll einen Namen bekommen: Da es um die Reihenfolge der Kanten geht, nennen wir (d) die *Reihenfolgebedingung*.

Beweis des Hilfssatzes. Wir weisen die Äquivalenz der vier Aussagen dadurch nach, dass wir nacheinander die folgenden Implikationen zeigen: (a) ⇒ (b), (b) ⇒ (c), (c) ⇒ (d) und (d) ⇒ (a).

(a) ⇒ (b):

Wir setzen voraus, dass T optimal ist, und haben zu zeigen, dass T die Pfadbedingung (b) erfüllt. Hierzu nehmen wir an, dass (b) nicht gilt und führen diese Annahme zum Widerspruch.

Aufgrund der Annahme, dass (b) nicht gilt, gibt es eine Kante $e = \{x, y\} \in E(G) \setminus E(T)$ und eine Kante e' auf dem x, y-Pfad von T mit $c(e') > c(e)$. Dann ist aber $(T - e') + e$ ein aufspannender Baum von G mit kleineren Kosten als T – im Widerspruch zur Voraussetzung, dass T optimal ist.

(b) ⇒ (c):

Wir setzen voraus, dass T die Pfadbedingung (b) erfüllt, und haben zu zeigen, dass dann für T auch die Schnittbedingung (c) erfüllt ist. Wir gehen analog zum Beweis der Implikation (a) ⇒ (b) vor: Es wird angenommen, dass (c) nicht gilt, und nachgewiesen, dass sich aus dieser Annahme ein Widerspruch ergibt.

Aufgrund der Annahme, dass (c) nicht gilt, gibt es eine Kante $e \in E(T)$, eine Komponente[7] B von $T - e$ sowie eine Kante $f = \{x, y\} \in \delta(V(B))$, so dass $c(f) < c(e)$ gilt. Man beachte, dass der x, y-Pfad P in T eine Kante aus $\delta_G(V(B))$ enthalten muss. Die einzige Kante von G, die sowohl in T als auch in $\delta_G(V(B))$ liegt, ist jedoch e. Die Kante e liegt also auf P. Aus (b) folgt demnach $c(e) \leq c(f)$ – im Widerspruch zu $c(f) < c(e)$.

(c) ⇒ (d):

Wir setzen voraus, dass (c) gilt und wählen eine *beliebige* Reihenfolge $e_1, \ldots, e_{n-1}$ der Kanten von T. Wir betrachten die Kante e_i (i-te Kante in dieser Reihenfolge) und wählen eine der beiden Komponenten B von $T - e_i$.

Wir setzen $X = V(B)$. Aufgrund von (c) gilt dann (d).

(d) ⇒ (a):

Es sei $e_1, \ldots, e_{n-1}$ eine Reihenfolge der Kanten von T, für die (d) erfüllt ist. Wir haben nachzuweisen, dass T optimal ist. Hierzu betrachten wir einen optimalen Baum T^*.

Der Baum T^* sei unter allen optimalen Bäumen so gewählt, dass T^* ein *möglichst langes Anfangsstück* der Folge $e_1, \ldots, e_{n-1}$ enthält.

Falls T^* sämtliche Kanten $e_1, \ldots, e_{n-1}$ enthält, so folgt $T = T^*$, da T^* (wegen $|E(T^*)| = n - 1$) keine weiteren Kanten enthalten kann. In diesem Fall sind wir fertig.

Andernfalls bezeichnen wir mit h den kleinsten Index aus $\{1, \ldots, n-1\}$, für den $e_h \notin E(T^*)$ gilt.

Hieraus leiten wir nun einen Widerspruch ab. (Man beachte: Gelingt uns dies, so sind wir fertig.)

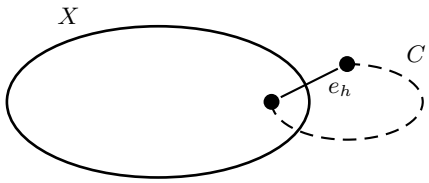
Da (d) für die Reihenfolge $e_1, \ldots, e_{n-1}$ erfüllt ist, gibt es ein $X \subseteq V(G)$, für das Folgendes erfüllt ist:

- $e_h \in \delta(X)$ und unter allen Kanten aus $\delta(X)$ hat e_h minimales Gewicht;
- $e_j \notin \delta(X)$ für alle $j \in \{1, \ldots, h-1\}$.

Wegen $e_h \notin E(T^*)$ enthält $T^* + e_h$ einen Kreis C. Da T^* keinen Kreis enthält, folgt $e_h \in E(C)$. Somit gilt $e_h \in E(C) \cap \delta(X)$ (siehe Zeichnung).

[6] Unter einem *Schnitt* des ungerichteten Graphen G verstehen wir eine Zerlegung von $V(G)$ in zwei disjunkte, nichtleere Teilmengen.

[7] Es ist üblich, statt *Zusammenhangskomponente* nur *Komponente* zu sagen.



Da C ein Kreis ist, muss es mindestens eine weitere Kante f auf C geben, für die $f \in \delta(X)$ gilt ($f \neq e$). Da e_h unter allen Kanten aus $\delta(X)$ minimales Gewicht hat, folgt

$$c(f) \geq c(e_h).$$

Man beachte, dass $(T^* + e_h) - f$ ein aufspannender Baum von G ist. Da T^* optimal ist, kann nicht $c(f) > c(e_h)$ gelten, da andernfalls $(T^* + e_h) - f$ ein aufspannender Baum von G mit kleinerem Gewicht als T^* wäre. Es folgt

$$c(f) = c(e_h).$$

Somit ist $(T^* + e_h) - f$ ein aufspannender Baum von G, der dasselbe Gewicht wie T^* besitzt, d.h., $(T^* + e_h) - f$ ist ebenfalls optimal.

Dies ist jedoch ein Widerspruch zur Wahl von T^*. Denn: $(T^* + e_h) - f$ enthält das Anfangsstück $e_1, \ldots, e_{h-1}, e_h$ der Folge $e_1, \ldots, e_{n-1}$ und dies ist ein längeres Anfangsstück als das entsprechende Anfangsstück für T^*. □

Zusammenfassung. In unserem Hilfssatz haben wir drei Möglichkeiten beschrieben, die optimalen Bäume zu charakterisieren. Insbesondere gilt:

- T ist genau dann optimal, wenn T die *Pfadbedingung* (b) erfüllt.

Ebenso gilt:

- T ist genau dann optimal, wenn T die *Schnittbedingung* (c) erfüllt.
- T ist genau dann optimal, wenn T die *Reihenfolgebedingung* (d) erfüllt.

Wir werden sehen, dass der Hilfssatz äußerst nützlich ist, wenn es darum geht, die Korrektheit unserer drei Algorithmen für das MINIMUM-SPANNING-TREE-PROBLEM nachzuweisen.

Zunächst schauen wir uns dies für den Algorithmus von Kruskal an. Dieser lässt sich auch wie folgt beschreiben.

Kruskals Algorithmus

(1) Sortiere die Kanten von G, so dass $c(e_1) \leq c(e_2) \leq \ldots \leq c(e_m)$ gilt.
(2) Setze $T = (V(G), \emptyset)$.
(3) **For** i=1 **to** m **do**
    If $T + e_i$ ist kreislos **then** setze $T = T + e_i$.

Theorem 1.
Kruskals Algorithmus arbeitet korrekt.

Beweis. Zunächst überlegen wir uns, dass Kruskals Algorithmus tatsächlich einen Baum abliefert. Offensichtlich ist das am Schluss erhaltene T kreislos. Es bleibt also zu zeigen, dass T zusammenhängend ist.

Angenommen, T wäre unzusammenhängend. Da G zusammenhängend ist, existiert eine Kante $e_i \in E(G) \setminus E(T)$, die Knoten aus zwei verschiedenen Komponenten von T verbindet. Es folgt: $T + e_i$ ist kreislos, und somit ist auch jeder Teilgraph von $T + e_i$ kreislos. Bei der Durchführung von Kruskals Algorithmus hätte e_i demnach in T aufgenommen werden müssen – im Widerspruch zu $e_i \in E(G) \setminus E(T)$. *Fazit*: Das am Schluss erhaltene T ist zusammenhängend und somit ein Baum.

Es bleibt zu zeigen: Das am Schluss vorliegende T ist optimal.

Wir zeigen dies, indem wir für dieses T das Erfülltsein der *Schnittbedingung* (c) nachweisen. Angenommen (c) wäre nicht erfüllt. Dann gibt es eine Kante $e \in E(T)$, eine Komponente B von $T - e$ und ein $f \in \delta(V(B))$ mit $c(f) < c(e)$. Da e die einzige Kante von T ist, die in $\delta(V(B))$ liegt, gilt $f \notin E(T)$.

Aus $c(f) < c(e)$ folgt, dass $f = e_i$ und $e = e_j$ für $i < j$ gilt. Die Kante f wurde in Zeile (3) also vor e betrachtet. Hieraus ergibt sich zusammen mit der Tatsache, dass $(T + f) - e$ kreislos ist: Als die Kante f bei der Durchführung von Zeile (3) an der Reihe war, hätte f in T aufgenommen werden müssen – im Widerspruch zur obigen Feststellung, dass am Schluss $f \notin E(T)$ gilt. □

Zur Laufzeit des Algorithmus von Kruskal. Wir werden, nachdem wir die Union-Find-Datenstruktur behandelt haben, eine wesentlich bessere Laufzeitschranke als die hier gegebene herleiten (vgl. Abschnitt 12.6). Hier begnügen wir uns zunächst mit der folgenden Feststellung: Die Laufzeit des Algorithmus von Kruskal ist $O(mn)$.

Begründung. Die Kanten von G können in $O(m \log m)$ Zeit sortiert werden. Wegen $m \leq n^2$ gilt $\log m \leq 2 \log n \leq 2n$; wir können also festhalten: Der Schritt (1) im Algorithmus von Kruskal lässt sich in $O(mn)$ Zeit erledigen. Ob in Zeile (3) $T + e_i$ kreislos ist, lässt sich in $O(n)$ Zeit ermitteln: Man braucht nur die Komponenten von T mittels BFS (oder DFS) zu bestimmen und dabei zu prüfen, ob e_i zwei Knoten aus derselben Komponente verbindet oder nicht. Da dies m mal auszuführen ist, ergibt sich die behauptete Laufzeitschranke $O(mn)$.

Prims Algorithmus lässt sich wie folgt beschreiben.

Prims Algorithmus

(1) Wähle $s \in V(G)$ und setze $T = (\{s\}, \emptyset)$.
(2) **While** $V(T) \neq V(G)$ **do**
Wähle eine Kante $e \in \delta_G(V(T))$ mit minimalem Gewicht und setze $T = T + e$.

Theorem 2.
Prims Algorithmus arbeitet korrekt.

Beweis. Während des gesamten Algorithmus ist T ein Baum. Zu zeigen ist, dass der am Schluss abgelieferte Baum T optimal ist. Wir zeigen dies, indem wir für dieses T nachweisen, dass die Reihenfolgebedingung (d) erfüllt ist.

Da die Reihenfolgebedingung Prims Algorithmus auf den Leib geschneidert ist, gibt es nicht viel zu tun: Als Reihenfolge $e_1, \ldots, e_{n-1}$ wählen wir gerade die Reihenfolge, in der die Kanten von T von Prims Algorithmus ausgewählt werden. Für e_i (i-te Kante in dieser Reihenfolge) betrachten wir den Baum T direkt vor dem Hinzufügen von e_i und setzen $X := V(T)$. Die Bedingung (d) ist dann aufgrund der Auswahl von e in (2) und aufgrund der Definition von X erfüllt. □

Die Laufzeit von Prims Algorithmus wird in Abschnitt 12.6 behandelt werden.

Aufgabe.

a) Formulieren Sie den Reverse-Delete-Algorithmus auf ähnliche Weise wie Kruskals Algorithmus, d.h. ähnlich zur Darstellung von Kruskals Algorithmus vor Theorem 1.

b) Beweisen Sie die Korrektheit des Reverse-Delete-Algorithmus, indem Sie dem Aufbau des Beweises von Theorem 1 folgen.

Hinweise zu b).

1. Der Nachweis, dass tatsächlich ein Baum abgeliefert wird, ist keineswegs schwieriger als im Fall von Kruskals Algorithmus.

2. Es bietet sich an, die Pfadbedingung (b) anstelle von (c) zum Einsatz zu bringen.

12.5 Die Union-Find-Datenstruktur

Wir gehen auch in diesem Abschnitt nach dem bekannten Lehrbuch *Algorithm Design* von Kleinberg und Tardos vor.

Gegeben sei eine endliche Knotenmenge. Wir befassen uns mit dem Vorgang, dass aus dieser Knotenmenge ein Graph dadurch „heranwächst“, dass Kanten schrittweise hinzugenommen werden (pro Schritt niemals mehr als eine Kante).

Am Anfang besteht der Graph nur aus isolierten Knoten, von denen jeder für sich eine Zusammenhangskomponente darstellt. Im Laufe des Verfahrens können Zusammenhangskomponenten zu größeren Komponenten[8] zusammenwachsen.

Wenn eine Kante hinzugefügt wird, so wollen wir die Komponenten natürlich nicht jedes Mal völlig neu berechnen; stattdessen soll eine Datenstruktur zum Einsatz kommen, die ein schnelles Update der Komponenten unterstützt. Außerdem soll diese Datenstruktur erlauben, zu einem gegebenen Knoten u den Namen der Komponente, der u angehört, schnell zu ermitteln. Man spricht von einer *Union-Find-Datenstruktur*.

Um den Algorithmus von Kruskal effizient zu implementieren, benötigt man eine Datenstruktur, die das Beschriebene leistet. Wird im Algorithmus von Kruskal für eine Kante $e = \{u, v\}$ gefragt, ob durch die Hinzunahme von e zum aktuellen Graphen ein Kreis entsteht, so kommt es darauf an, die aktuellen Komponenten von u und v zu bestimmen; man fragt dann, ob diese Komponenten gleich sind:

- Falls ja, so wird e zurückgewiesen, da sonst ein Kreis entstünde.
- Falls nein, so wird e zum aktuellen Graphen hinzugefügt, wodurch kein Kreis entsteht. Anschließend hat man ein Update der Komponenten vorzunehmen („Zusammenlegung der Komponenten von u und v“).

Union-Find-Datenstrukturen spielen nicht nur im Zusammenhang mit Kruskals Algorithmus eine wichtige Rolle, sondern ebenso im Zusammenhang mit anderen Fragestellungen, bei denen *Partitionen* von Mengen zu verwalten sind. Aus diesem Grund behandeln wir das Problem, eine geeignete Union-Find-Datenstruktur zu entwickeln, ab jetzt unabhängig von Kruskals Algorithmus und kommen nur gelegentlich darauf zurück.

12.5.1 Das Problem

Wir beschreiben noch einmal, worum es beim Union-Find-Problem geht. *Die zu entwickelnde Union-Find-Datenstruktur soll es ermöglichen, disjunkte Mengen (wie beispielsweise die Knotenmengen der Komponenten eines Graphen) zu verwalten, womit Folgendes gemeint ist*:

- Für jeden Knoten[9] u soll die *Operation Find(u)* den Namen der Menge liefern, die u enthält. Die Operation $\mathrm{Find}\,(u)$ kann benutzt werden, um zu testen, ob u und v derselben Menge angehören: Man muss lediglich prüfen, ob $\mathrm{Find}\,(u) = \mathrm{Find}\,(v)$ gilt.
- Neben der Operation $\mathrm{Find}\,(u)$ soll es eine *Operation Union(A, B)* geben, durch die die beiden Mengen A und B zu einer einzigen Menge vereinigt werden.

Wie wir bereits gesehen haben, können diese Operationen eingesetzt werden, um die Zusammenhangskomponenten eines Graphen $G = (V, E)$ zu verwalten, dessen Knotenmenge V fest gegeben ist, dessen Kantenmenge E sich jedoch im Aufbau befindet. Die zu verwaltenden Mengen sind in diesem Prozess die Knotenmengen der Zusammenhangskomponenten und für einen Knoten u liefert $\mathrm{Find}\,(u)$ den Namen der aktuellen Komponente von u. Wenn eine Kante (u, v) zum Graphen hinzugefügt werden soll, so ist zunächst zu testen, ob

$$\mathrm{Find}\,(u) = \mathrm{Find}\,(v)$$

[8] Statt „Zusammenhangskomponente“ sagen wir im Folgenden meist „Komponente“.

[9] Die Elemente der betrachteten Mengen bezeichnen wir auch dann als Knoten, wenn keine Graphen im Spiel sind.

gilt. Ist dies *nicht* der Fall, so ist

$$\text{Union}\,(\text{Find}\,(u), \text{Find}\,(v))$$

auszuführen, wodurch die Komponenten von u und v zusammengelegt werden.

Neben den Operationen Union (A, B) und Find (u) spielt außerdem die *Operation MakeUnionFind(S)* eine Rolle. Die Operation MakeUnionFind(S) dient der Initialisierung. Zusammenfassend halten wir fest, dass die Union-Find-Datenstruktur drei Operationen unterstützen soll:

- Für eine Menge S soll *MakeUnionFind(S)* eine Partition von S in disjunkte Klassen[10] liefern, die alle nur aus einem einzigen Element bestehen. MakeUnionFind (S) wird in $O(n)$ Zeit implementiert werden (für $n = |S|$).
- Für ein $u \in S$ soll *Find(u)* den Namen der Klasse liefern, die u enthält. Unser Ziel ist, Find (u) so zu implementieren, dass Find (u) nur $O(\log n)$ Zeit benötigt. (In einigen der vorgestellten Implementierungen wird Find (u) sogar nur $O(1)$ Zeit in Anspruch nehmen.)
- Durch *Union(A,B)* sollen zwei Klassen A und B vereinigt werden. Unser Ziel wird sein, die Operation Union (A, B) so zu implementieren, dass sie nur $O(\log n)$ Zeit in Anspruch nimmt.

Eine Bemerkung dazu, was unter dem *Namen einer Klasse* zu verstehen ist: Das Wichtigste in diesem Zusammenhang ist, dass die Namensgebung *konsistent* ist, d.h., dass Find (v) und Find (w) denselben Namen liefern, wenn v und w aus derselben Klasse stammen und außerdem muss natürlich auch Find $(v) \neq$ Find (w) gelten, wenn dies nicht der Fall ist. Darüber hinaus hat man große Freiheiten bei der Namenswahl. *Im Folgenden wird immer die naheliegende Lösung gewählt werden, dass eine Klasse nach einem ihrer Elemente benannt wird.*

12.5.2 Eine einfache Datenstruktur für das Union-Find-Problem

Eine einfache Art, eine Union-Find-Datenstruktur einzurichten, ist mittels eines Arrays, in dem für jedes Element der Name der zugehörigen Klasse angegeben wird. *Genauer*: Es sei S die betrachtete Menge, es gelte $|S| = n$ und die Elemente von S seien mit $1, \ldots, n$ bezeichnet. Es sei Component ein Array der Länge n, wobei Component$[s]$ der Name der Klasse sei, der s angehört $(s = 1, \ldots, n)$. Zur Durchführung von MakeUnionFind (S) wird Component$[s] = s$ gesetzt $(s = 1, \ldots, n)$; dies bedeutet, dass am Anfang jedes Element s seine eigene individuelle Klasse bildet. MakeUnionFind (S) ist klarerweise in $O(n)$ Zeit durchführbar.

Die Implementierung einer Union-Find-Datenstruktur mittels eines Arrays führt dazu, dass die Durchführung von Find (u) sehr einfach ist: Find (u) *ist in* $O(1)$ *Zeit durchführbar, da nichts weiter zu tun ist, als den aktuellen Wert von Component*$[u]$ *abzulesen.*

Im Gegensatz dazu kann die Operation Union (A, B) *jedoch einen Zeitaufwand von* $O(n)$ *erfordern*, da im Fall $A \neq B$ für eine der beiden Mengen A, B sämtliche Einträge Component$[s]$ abzuändern sind. (Man beachte: Beide Mengen A und B können einen konstanten Anteil von S ausmachen, etwa $|A| = \frac{1}{3}n$ und $|B| = \frac{1}{4}n$.)

Wir betrachten *zwei Maßnahmen*, um die Operation Union (A, B) zu verbessern:

- Es ist nützlich, für jede Klasse zusätzlich eine Liste ihrer Elemente zu führen; dadurch erreicht man, dass nicht jedes Mal das gesamte Array durchgegangen werden muss, wenn für die Elemente einer Klasse ein Update durchzuführen ist.
- Darüber hinaus spart man einige Zeit, wenn als Name für die durch Zusammenlegung entstandene Klasse der Name von A oder von B übernommen wird. Das hat den Vorteil, dass nur für die Elemente von einer der beiden Klassen ein Update vorgenommen werden muss. Dabei gilt aus naheliegenden Gründen die folgende *Regel*:

Falls A und B unterschiedliche Größe haben, so übernimmt man den Namen der größeren Klasse.

[10] Statt Partition sagt man bekanntlich auch Zerlegung und spricht von *Klassen* der Partition (bzw. Zerlegung).

Um festzustellen, welche der beiden Klassen die größere ist, kann ein zusätzliches Array Size der Länge n verwendet werden, in dem für jedes s, das aktuell als Name einer Klasse auftritt, die Größe Size[s] dieser Klasse abgelesen werden kann; hierzu sind pro Union-Operation zwei Einträge des Arrays Size abzuändern.

Auch mit diesen beiden Verbesserungen bleibt es *im schlechtesten Fall* bei einem Zeitaufwand der Größenordnung $O(n)$ für die Union-Operation: Dieser Fall tritt ein, wenn es sich bei beiden Mengen A und B um große Mengen handelt, was heißen soll, dass sowohl A als auch B einen konstanten Anteil der Elemente von S umfasst. *Solche schlechten Fälle für* Union (A, B) *können allerdings nicht allzu häufig auftreten, da es sich bei* $A \cup B$ *um eine noch größere Menge handelt.* Diese wichtige, aber noch etwas vage Feststellung soll im Folgenden präzisiert werden. Dies geschieht dadurch, dass wir uns weniger die Laufzeit einer einzelnen Operation Union (A, B) anschauen, sondern die Folge der ersten k Union-Operationen unter die Lupe nehmen, wobei es um die *Gesamtlaufzeit* (bzw. die *durchschnittliche Laufzeit*) dieser k Operationen geht.

Satz 1.

Betrachtet werde die *Array-Implementierung der Union-Find-Datenstruktur* für eine Menge S mit $|S| = n$, wobei Vereinigungen den Namen der größeren Menge übernehmen sollen. Eine Find-Operation benötigt dann $O(1)$ Zeit, MakeUnionFind (S) erfordert $O(n)$ Zeit und für die Folge der ersten k Union-Operationen wird insgesamt höchstens $O(k \log k)$ Zeit benötigt (für alle k, die kleiner oder gleich der Gesamtzahl der Union-Operationen sind).

Beweis. Die Behauptungen über die Find-Operation und über MakeUnionFind (S) sind leicht einzusehen. Es geht also um die letzte Behauptung, zu deren Beweis wir die ersten k Union-Operationen betrachten. Der einzige Teil einer Union-Operation, der mehr als $O(1)$ Zeit in Anspruch nimmt, ist das Update des Arrays Component. *Wir betrachten ein festes* $v \in S$ *und fragen uns, wie oft* Component[v] *bei der Durchführung unserer* k *Union-Operationen höchstens geändert werden kann.*

Es sei daran erinnert, dass am Anfang, nachdem MakeUnionFind (S) ausgeführt wurde, alle n Elemente von S eine eigene Klasse bilden. Eine einzige Union-Operation kann immer nur für höchstens zwei dieser 1-elementigen Klassen bewirken, dass sie zu einer 2-elementigen Klasse verschmelzen. Dies bedeutet, dass nach k Union-Operationen alle bis auf höchstens $2k$ Elemente von S noch völlig unberührt sind und immer noch lauter 1-elementige Klassen bilden.

Wir betrachten ein festes Element v aus S. Sobald die Klasse von v in eine unserer k Union-Operationen involviert ist, wächst sie an. Es kann nun sein, dass bei einigen dieser Operationen der Eintrag von Component[v] geändert wird. Aufgrund unserer Regel, dass immer der Name der größeren Klasse übernommen wird, ist jede Änderung von Component[v] mit mindestens einer Verdopplung der Größe der Klasse von v verbunden. Da – wie oben ausgeführt – höchstens $2k$ Elemente von S von den ersten k Union-Operationen betroffen sind, kann die Klasse von v auf höchstens $2k$ anwachsen, d.h., *es kann nicht mehr als* $\log_2(2k)$ *Updates von* Component[v] *während der ersten* k *Union-Operationen geben.*

Insgesamt bedeutet das, dass es in den ersten k Union-Operationen höchstens $2k \cdot \log_2(2k)$ Updates von irgendwelchen Einträgen des Arrays Component geben kann. (Man beachte: Nur höchstens $2k$ Elemente sind involviert!) Es folgt (wie behauptet), dass für die ersten k Union-Operationen höchstens $O(k \log k)$ Zeit benötigt wird. □

Satz 1 besagt, dass die Array-Implementierung der Union-Find-Datenstruktur bereits ein gutes Ergebnis liefert: MakeUnionFind (S) und Find (u) benötigen $O(n)$ bzw. O(1) Zeit und erfüllen somit unsere Vorgaben. Aber auch Union (A, B) kommt unseren Zielvorstellungen bereits nahe: Im Einzelfall kann Union (A, B) zwar $O(n)$ Zeit in Anspruch nehmen, aber *im Durchschnitt* benötigen die ersten k Union-Operationen $O(\log k)$ Zeit. (Außerdem ist die Einfachheit der Array-Implementierung positiv hervorzuheben.)

Defizit der Array-Implementierung von Union-Find: Wie bereits gesagt, kann eine einzelne Union-Operation $O(n)$ Zeit erfordern, d.h., unsere Zielsetzung für Union (A, B) ist nicht erfüllt.

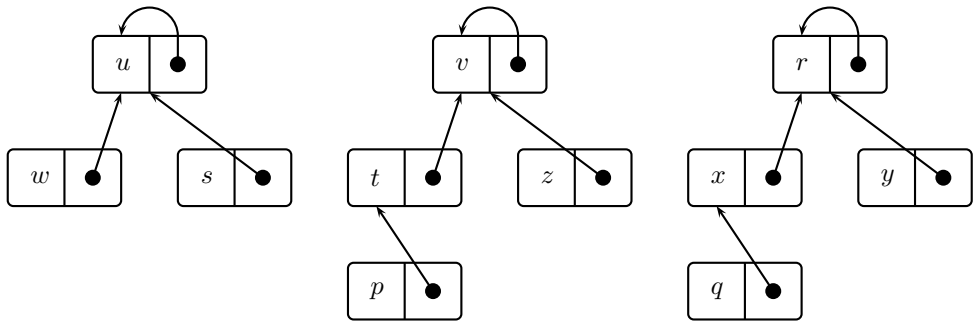
12.5.3 Eine bessere Datenstruktur für Union-Find

Die Datenstruktur für diese alternative Implementierung verwendet Zeiger. Jeder Knoten $v \in S$ ist in einem Record gespeichert – zusammen mit einem dazugehörigen Zeiger, durch den man direkt oder dadurch, dass man weiteren Zeigern folgt, zum Namen der Klasse gelangt, die v enthält[11]. Wie zuvor benutzen wir die Elemente der Menge S als mögliche Namen für die Klassen und betrachten Partitionen, in denen jede Klasse nach einem ihrer Elemente benannt ist.

Der *Initialisierung* dient die *Operation MakeUnionFind(S)*, durch die zu jedem $v \in S$ ein Record eingerichtet wird, in dem v zusammen mit einem Zeiger gespeichert wird, der auf v selbst verweist[12]. Das bedeutet, dass am Anfang jedes v eine 1-elementige Klasse bildet.

Als nächstes betrachten wir die *Union-Operation*: Wir nehmen an, dass A und B zwei verschiedene Klassen sind, die zusammengelegt werden sollen; die Klasse A sei nach dem Knoten $v \in A$ benannt, während die Klasse B nach dem Knoten $u \in B$ benannt sei. Es wird dann entweder u oder v als Name für die Vereinigungsmenge gewählt, sagen wir, v sei der Name für die Vereinigungsmenge $A \cup B$. *Dies wird dadurch umgesetzt, dass nichts weiter getan wird, als den Zeiger von u abzuändern: Der Zeiger von u wird so geändert, dass er nun auf v verweist.* Die Zeiger, die zu anderen Elementen von B gehören, werden nicht abgeändert. Für Elemente $w \in B$, $w \neq u$, bedeutet dies, dass man einer Reihe von Zeigern folgen muss, um den Namen der Klasse zu erfahren, der w angehört: Zunächst wird man dabei zum „alten Namen“ u hingeführt und erst über den Zeiger von u nach v gelangt man zum „neuen Namen“ v.

Zur Illustration anhand eines Beispiels sei $S = \{p, q, r, s, t, u, v, w, x, y, z\}$. Es gebe drei Klassen $\{s, u, w\}$, $\{p, t, v, z\}$ und $\{q, r, x, y\}$. Es soll die folgende Situation vorliegen:

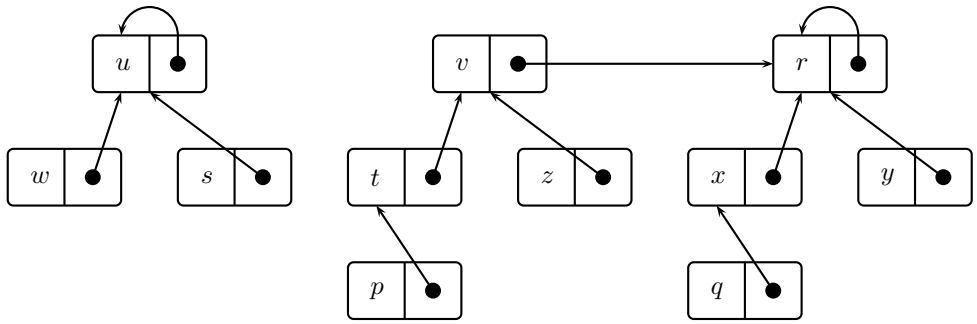


Die drei Mengen dieses Beispiels könnten etwa durch den folgenden Ablauf von Union-Operationen entstanden sein: Union (w, u), Union (s, u), Union (p, t), Union (z, v), Union (t, v), Union (q, x), Union (y, r), Union (x, r).

[11] Falls Sie mit Zeigern wenig vertraut sind: Im vorliegenden Kontext reicht es aus, sich einen Zeiger als einen Verweis auf ein Objekt vorzustellen.

[12] Alternativ könnte man hier den Nullzeiger verwenden.

Wenn nun als nächste Vereinigung die Operation Union (v, r) ausgeführt wird, wobei r der Name der durch Vereinigung entstandenen neuen Klasse sein soll, so ergibt sich die folgende Darstellung:



In dieser zeigerbasierten Datenstruktur ist eine *Union-Operation in $O(1)$ Zeit* durchführbar: Man hat nichts weiter zu tun, als einen einzigen Zeiger abzuändern. *Die Find-Operation ist dagegen nicht mehr in konstanter Zeit durchführbar*, da wir einer Reihe von Zeigern folgen müssen, um den Namen der Klasse zu erfahren, der ein Element angehört. Bevor wir diesen aktuellen Namen mitgeteilt bekommen, durchlaufen wir zunächst die komplette Historie der alten Namen von Klassen, denen das betreffende Element früher einmal angehört hat.

Wie viele Schritte benötigt eine einzelne Find-Operation? Die Antwort fällt leicht: Die Anzahl der Schritte, die Find (u) benötigt, ist gleich der Anzahl der Namen, die die Klasse von u bislang gehabt hat. Ebenso wie bei der Array-Implementierung verfahren wir nach der *Regel, dass bei unterschiedlicher Größe der beiden zu vereinigenden Mengen immer der Name der größeren Menge übernommen wird.*

Man beachte: Die genannte Regel über die Union-Operation dient jetzt dazu, die Zeit zu reduzieren, die für eine Find-Operation aufgewendet werden muss.

Die Regel kann dadurch effizient implementiert werden, dass man für jeden Knoten v in dem dazugehörigen Record *ein drittes Datenfeld* einrichtet: *Für alle Knoten v, die aktuell als Name für eine Klasse dienen, soll in diesem Datenfeld die Größe dieser Klasse gespeichert sein.* Um bei Ausführung einer Union-Operation die Einträge im dritten Datenfeld zu aktualisieren, ist pro Union-Operation nur eine Addition und nur ein Update nötig.

Unsere Überlegungen laufen auf den folgenden Satz hinaus.

Satz 2.

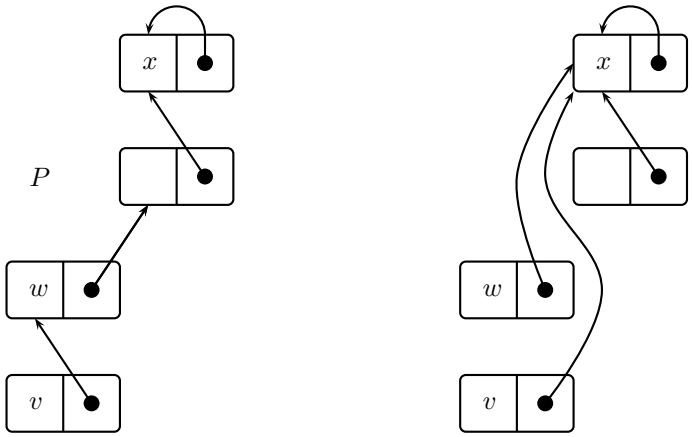
Betrachtet werde die zuvor beschriebene *zeigerbasierte Implementierung der Union-Find-Datenstruktur* für eine Menge S mit $|S| = n$, wobei Vereinigungen den Namen der größeren Menge übernehmen sollen. Eine Union-Operation benötigt dann $O(1)$ Zeit, MakeUnionFind (S) erfordert $O(n)$ Zeit und eine Find-Operation nimmt $O(\log n)$ Zeit in Anspruch.

Beweis. Die Behauptungen über Union und MakeUnionFind sind leicht zu prüfen: Die Richtigkeit ergibt sich direkt aus der Beschreibung der zeigerbasierten Implementierung. Auch die über die Find-Operation gemachte Behauptung ist leicht einzusehen: Die Zeit, die für die Durchführung von Find (v) für einen Knoten v in einer bestimmten Situation nötig ist, ergibt sich direkt aus der Häufigkeit, mit der der Name der Klasse von v bislang geändert wurde. Da bei Vereinigungen immer der Name der größeren Menge übernommen wird, findet bei jeder Namensänderung der Klasse von v mindestens eine Verdopplung der Klassengröße statt. Da es insgesamt nur n Elemente gibt, kann es demnach höchstens $\log_2 n$ Namensänderungen der Klasse von v geben. □

12.5.4 Weitere Verbesserungen durch Pfadverkürzung (path compression)

Wir begnügen uns in diesem Abschnitt damit, die *Idee* anzudeuten, auf der Verbesserungen durch Pfadverkürzung beruhen.

Stellen Sie sich vor, dass wir (wie im letzten Abschnitt) eine zeigerbasierte Union-Find-Datenstruktur vorliegen haben und dass Find (v) für einen Knoten v auszuführen ist. Es kann – wie wir wissen – einige Zeit in Anspruch nehmen, den Namen x der Klasse von v zu ermitteln: Zunächst muss ein Pfad P durchlaufen werden (siehe linke Zeichnung).



Nun kann es durchaus sein, dass Find (v) im Laufe des weiteren Verfahrens noch öfter auszuführen ist – möglicherweise sogar sehr oft. In diesem Fall möchte man nicht jedes Mal denselben Pfad P durchlaufen müssen, nur um die längst bekannte Information zu erhalten, dass v in derselben Klasse wie x liegt. Entsprechendes gilt, wenn Find (w) für einen Knoten w durchzuführen ist, der auf P liegt: Auch in diesem Fall möchte man gerne erneutes Durchlaufen von w nach x längs P vermeiden.

Aus den genannten Gründen geht man wie folgt vor: Direkt nach der Ausführung von Find (v) durchläuft man den Pfad P noch ein zweites Mal, wobei man für alle Knoten von P den Zeiger so abändert, dass dieser nun auf x verweist (siehe rechte Zeichnung).

Das Verfahren, dessen Grundidee wir soeben beschrieben haben, nennt man *Pfadverkürzung* (engl. *path compression*).

Durch den Einsatz von Pfadverkürzung lässt sich die Union-Find-Datenstruktur noch einmal wesentlich verbessern: Details hierzu findet man in Lehrbüchern über Algorithmen und Datenstrukturen, beispielsweise in dem bekannten Lehrbuch von Cormen, Leiserson, Rivest und Stein.

12.6 Die Laufzeit der Algorithmen von Kruskal und Prim

12.6.1 Zur Laufzeit von Kruskals Algorithmus

Wir wollen die Union-Find-Datenstruktur verwenden, um Kruskals Algorithmus zu implementieren.

Zunächst haben wir allerdings die Kanten von G in aufsteigender Reihenfolge zu sortieren. Dies geht in $O(m \log m)$ Zeit. Da es zwischen zwei Knoten nur höchstens eine Kante gibt, gilt $m \leq n^2$, woraus man $\log m \leq \log (n^2) = 2 \log n$ erhält. Deshalb lässt sich die *Laufzeit für das Sortieren der Kanten* auch angeben als

$$O(m \log n).$$

Nachdem die Kanten sortiert sind, verwenden wir die Union-Find-Datenstruktur, *um die Zusammenhangskomponenten zu verwalten*, die beim schrittweisen Aufbau eines minimalen aufspannenden Baums entstehen.

Wie wir wissen, geht man die Kanten in aufsteigender Reihenfolge durch. Ist dabei die Kante $e = \{u, v\}$ an der Reihe, so sind $\text{Find}(u)$ und $\text{Find}(v)$ zu berechnen und zu testen, ob $\text{Find}(u) \neq \text{Find}(v)$ gilt.

Falls ja, so wird e in den Baum aufgenommen und es hat

$$\text{Union}(\text{Find}(u), \text{Find}(v))$$

zu erfolgen.

Im Laufe des Kruskal-Algorithmus finden also statt:

$$2m \text{ Find-Operationen und } n-1 \text{ Union-Operationen.}$$

Aus Satz 1 für die Array-Implementierung der Union-Find-Datenstruktur bzw. Satz 2 für die zeigerbasierte Implementierung ergibt sich zusammen mit den obigen Anzahlangaben, dass in beiden Fällen

$$O(m \log n)$$

eine obere Schranke für die Zeit ist, die im Algorithmus von Kruskal für die Union- und Find-Operationen aufgewendet wird.

Man kann für die zeigerbasierte Union-Find-Datenstruktur durch Pfadverkürzung (path compression) noch bessere Laufzeiten für die Find-Operation herausholen (vgl. Abschnitt 12.5.4) – für den Algorithmus von Kruskal bringt das jedoch keine Verbesserung der Gesamtlaufzeit: Die Laufzeit des Algorithmus von Kruskal wird unabänderlich durch den Term $O(m \log n)$ dominiert, der durch das Sortieren der Kanten ins Spiel kommt.

Kurz zusammengefasst können wir feststellen:

Kruskals Algorithmus kann so implementiert werden, dass sich für Graphen mit n Knoten und m Kanten die Laufzeit $O(m \log n)$ ergibt.

12.6.2 Zur Laufzeit von Prims Algorithmus

Der *Algorithmus von Prim* besitzt große Ähnlichkeit mit dem *Algorithmus von Dijkstra*. (Die Gültigkeitsbeweise dieser beiden Algorithmen sind jedoch sehr unterschiedlich.)

Um die Ähnlichkeit zu verdeutlichen, wird der Algorithmus von Prim so beschrieben, dass er der Formulierung des Algorithmus von Dijkstra auf Seite 181 *möglichst nahe kommt*.

Dabei wird vorausgesetzt, dass der betrachtete Graph $G = (V, E)$ zusammenhängend ist. Der Knoten s („Startknoten“, „Wurzel“) sei vorgegeben.

Prim's Algorithm $(\mathbf{G}, \mathbf{c}, \mathbf{s})$

(1) Let T denote a tree contained in G and let S be the node set of T
(2) For each $u \in V$, we store a value $d(u)$ and, if $u \neq s$ and $d(u) \neq \infty$,
    we also store a node $\text{pred}(u) \in S$
(3) Initially let T be the tree with node set $S = \{s\}$ and let $d(s) = 0$
(4) For each $u \neq s$ let $d(u) = c(\{s, u\})$ if $\{s, u\} \in E$ and $d(u) = \infty$ otherwise
(5) Let $\text{pred}(u) = s$ for all u with $\{s, u\} \in E$
(6) **While** $S \neq V$
(7) Select a node $v \in V \setminus S$ with $d(v) = \min\{d(u) : u \in V \setminus S\}$
(8) Add v to S and add the edge $\{v, \text{pred}(v)\}$ to T
(9) For each $u \in V \setminus S$ with $\{v, u\} \in E$ and $c(\{v, u\}) < d(u)$,
    let $d(u) = c(\{v, u\})$ and $\text{pred}(u) = v$
(10) **EndWhile**

Aufgrund der Ähnlichkeit zum Algorithmus von Dijkstra ergeben sich Feststellungen zur Laufzeit, die wir auf ähnliche Art für den Algorithmus von Dijkstra getroffen haben.

- Die Laufzeit für die beschriebene Version des Algorithmus von Prim ist $O(n^2)$ für $n = |V|$, da die While-Schleife $n - 1$ mal durchlaufen wird und die Zeilen (7)-(9) offenbar in $O(n)$ Zeit ausgeführt werden können.
- Mithilfe einer Prioritätswarteschlange lässt sich die Laufzeit $O(m \log n)$ erzielen, was beispielsweise für Graphen mit $m = \Theta(n)$ eine Verbesserung gegenüber der Laufzeitschranke $O(n^2)$ bedeutet.

Genauer gilt sowohl für Dijkstra als auch für Prim: Die genannte Laufzeit $O(m \log n)$ ergibt sich, wenn die Prioritätswarteschlange mittels eines Binärheaps implementiert wird (vgl. Kleinberg/Tardos).

Wie im Fall von Kruskals Algorithmus können wir also zusammengefasst feststellen:

Prims Algorithmus kann so implementiert werden, dass sich für Graphen mit n Knoten und m Kanten die Laufzeit $O(m \log n)$ ergibt.

13 Dynamisches Programmieren

Auch in diesem Kapitel gehen wir größtenteils nach dem Lehrbuch von Kleinberg und Tardos vor. Beim *Dynamischen Programmieren* (engl. *dynamic programming*) handelt es sich um eine Entwurfsmethode für Algorithmen, die *häufig bei Optimierungsproblemen* zum Einsatz kommt.

Im letzten Kapitel haben wir Probleme kennengelernt, die erfolgreich mit einer Greedy-Strategie behandelt werden konnten. Dass man mit einer Greedy-Strategie Erfolg hat, ist jedoch eher die Ausnahme: *Für die meisten Probleme sind keine funktionierenden Greedy-Verfahren bekannt.* Man ist also auf andere Methoden angewiesen. Eine dieser Methoden, die die meisten von Ihnen vermutlich kennen, wird *Divide and Conquer* genannt.

Beim Divide-and-Conquer-Verfahren zerlegt man ein gegebenes Problem in Teilprobleme, die man rekursiv löst; aus den Lösungen der Teilprobleme gewinnt man dann die Lösung des Ausgangsproblems. Ein typisches Beispiel hierfür ist der *Mergesort-Algorithmus*. Auch der berühmte *Algorithmus von Strassen* zur Matrizenmultiplikation ist ein Divide-and-Conquer-Algorithmus, ebenso wie der *Karatsuba-Algorithmus* zur Multiplikation zweier ganzer Zahlen. Häufig führt aber auch Divide and Conquer nicht zum Erfolg; dann ist nicht selten *Dynamisches Programmieren* das Mittel der Wahl.

Worum es beim Dynamischen Programmieren geht, erläutert man am besten anhand eines Beispiels. Zu diesem Zweck betrachten wir ein Beispiel aus dem Bereich des Schedulings, nämlich das *gewichtete Intervall-Scheduling-Problem*.

13.1 Das gewichtete Intervall-Scheduling-Problem

In Abschnitt 12.1 haben wir das *ungewichtete* Intervall-Scheduling-Problem behandelt und mithilfe eines Greedy-Verfahrens gelöst. *Hier betrachten wir nun die gewichtete Variante des Intervall-Scheduling-Problems*: Wir bezeichnen die Anfragen wie zuvor mit $1, \ldots, n$; zur i-ten Anfrage gehört (ebenfalls wie zuvor) ein Zeitintervall $[s(i), f(i)]$ $(i = 1, \ldots, n)$. Darüber hinaus besitzt jedes Intervall einen *Wert* $v_i \geq 0$ $(i = 1, \ldots, n)$; statt vom Wert sprechen wir auch vom *Gewicht* v_i $(i = 1, \ldots, n)$.

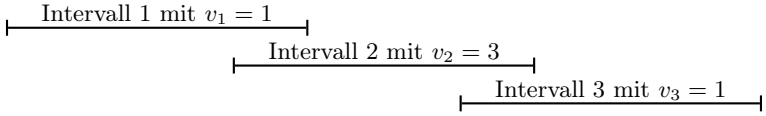
Gesucht ist eine Menge $S \subseteq \{1, \ldots, n\}$ von paarweise kompatiblen Anfragen, *für die die Summe*

$$\sum_{i \in S} v_i$$

maximal ist.

Das ursprüngliche (ungewichtete) Intervall-Scheduling-Problem ist ein Spezialfall des gewichteten Intervall-Scheduling-Problems: Um dies zu erkennen, setzt man $v_i = 1$ für alle $i \in \{1, \ldots, n\}$.

Der Greedy-Algorithmus, den wir in Abschnitt 12.1 für das ungewichtete Problem erhalten haben („wiederholtes Wählen eines kompatiblen Intervalls, das zuerst endet“), funktioniert im gewichteten Fall nicht mehr, wie das folgende einfache Beispiel zeigt:



Für das gewichtete Intervall-Scheduling-Problem ist auch kein alternativer Greedy-Algorithmus bekannt – dies ist für uns der Anlass, das Problem mit der Methode des Dynamischen Programmierens anzugehen.

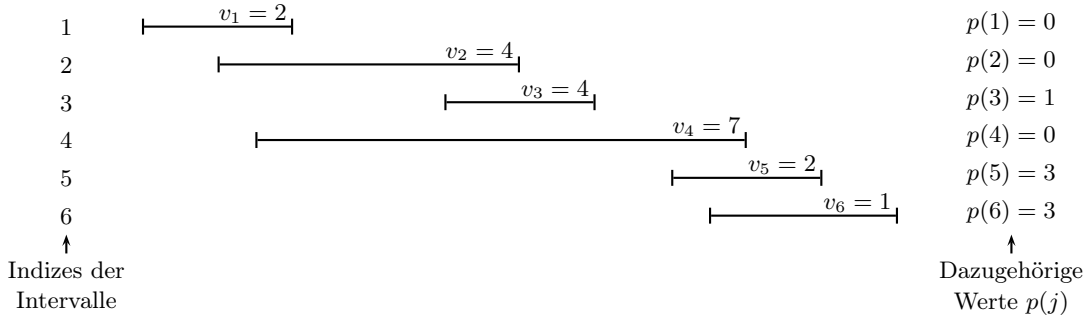
Wir setzen ab jetzt immer voraus, dass die Nummerierung der Anfragen so gewählt ist, dass

$$f(1) \leq \ldots \leq f(n)$$

gilt[1]. Für jedes Intervall $j \in \{1, \ldots, n\}$ definieren wir $p(j) \in \{0, \ldots, n\}$ wie folgt:

- Falls es ein Intervall $i < j$ gibt, so dass i und j sich nicht überlappen, so sei mit $p(j)$ das größte $i < j$ bezeichnet, für das dies gilt;
- Falls es kein solches i gibt, so setzen wir $p(j) = 0$.

Das folgende **Beispiel** aus dem Buch von Kleinberg/Tardos illustriert die Definition von $p(j)$:



Gegeben seien nun Intervalle $1, \ldots, n$ mit Gewichten (Werten) $v_1, \ldots, v_n$ und $\mathcal{O}$ sei eine dazugehörige optimale Lösung des (gewichteten) Intervall-Scheduling-Problems. Dann gibt es offensichtlich zwei Möglichkeiten: $n \in \mathcal{O}$ oder $n \notin \mathcal{O}$.

1. Fall: $n \in \mathcal{O}$.

In diesem Fall können wir feststellen, dass sämtliche Intervalle $p(n) + 1, \ldots, n - 1$ nicht zu $\mathcal{O}$ gehören, da sich diese Intervalle mit n überlappen.

Außerdem: Lässt man das Intervall n aus $\mathcal{O}$ weg, so bilden die übrigen Intervalle von $\mathcal{O}$ eine optimale Lösung für das Teilproblem, bei dem die Eingabe aus den Intervallen $1, \ldots, p(n)$ besteht, denn andernfalls könnte man die Intervalle aus $\mathcal{O} \cap \{1, \ldots, p(n)\}$ durch eine bessere Wahl ersetzen.

2. Fall: $n \notin \mathcal{O}$.

In diesem Fall ist $\mathcal{O}$ eine optimale Lösung des Problems, bei dem die Eingabe nur aus den Intervallen $1, \ldots, n - 1$ besteht, denn andernfalls hätte man eine bessere Lösung für das Problem mit Eingabe $1, \ldots, n$.

Die vorangegangenen Überlegungen legen Folgendes nahe: *Um eine optimale Lösung für eine gegebene Intervallmenge*

$$\{1, \ldots, n\}$$

zu finden, hat man sein Augenmerk auf die optimalen Lösungen für kleinere Intervallmengen der Form

$$\{1, \ldots, j\}$$

zu richten.

Für jeden Wert j mit $1 \leq j \leq n$ wollen wir mit $\mathcal{O}_j$ eine optimale Lösung des *Teilproblems* bezeichnen, das aus den Intervallen $\{1, \ldots, j\}$ besteht, und mit $\mathrm{OPT}(j)$ bezeichnen wir den dazugehörigen optimalen Wert, d.h.

$$\mathrm{OPT}(j) = \sum_{i \in \mathcal{O}_j} v_i.$$

Darüber hinaus definieren wir noch

$$\mathrm{OPT}(0) = 0.$$

[1] Zur Erinnerung: Zu jeder Anfrage i gehört ein Zeitintervall $[s(i), f(i)]$ $(i = 1, \ldots, n)$. Wir unterscheiden nicht immer streng zwischen einer Anfrage und dem zugehörigen Zeitintervall und sprechen auch vom Intervall i.

Also: Unser ursprüngliches Problem ist die Bestimmung von $\mathcal{O}_n$ und $\mathrm{OPT}(n)$; zu diesem Zweck betrachten wir *geeignet gewählte Teilprobleme*, für die wir die optimalen Lösungen $\mathcal{O}_j$ bzw. $\mathrm{OPT}(j)$ zu bestimmen haben.

Für $\mathcal{O}_j$ und die Intervallmenge $\{1, \ldots, j\}$ erhält man analog zu unseren obigen Überlegungen:

- Entweder gilt $j \in \mathcal{O}_j$; in diesem Fall erhält man $\mathrm{OPT}(j) = v_j + \mathrm{OPT}\,(p(j))$.
- Oder es gilt $j \notin \mathcal{O}_j$; in diesem Fall hat man $\mathrm{OPT}\,(j) = \mathrm{OPT}\,(j-1)$.

Da eine der beiden Möglichkeiten ($j \in \mathcal{O}$ oder $j \notin \mathcal{O}$) vorliegen muss, können wir als Ergebnis also festhalten:

$$\mathrm{OPT}\,(j) = \max\big(v_j + \mathrm{OPT}\,(p(j)), \mathrm{OPT}\,(j-1)\big) \tag{13.1}$$

Aus dem Vorangegangenen ergibt sich ebenfalls, wie über die Zugehörigkeit von j zu $\mathcal{O}_j$ entschieden werden kann:

$$\begin{gathered} \textit{Anfrage } j \textit{ gehört zu einer optimalen Lösung für das Teilproblem } \{1, \ldots, j\} \\ \textit{genau dann, wenn} \\ v_j + \mathrm{OPT}\,(p(j)) \geq \mathrm{OPT}\,(j-1). \end{gathered} \tag{13.2}$$

(13.1) stellt (zusammen mit (13.2)) die erste wichtige Komponente dar, die bei jeder Lösung durch Dynamisches Programmieren vorhanden sein muss: Es muss eine **Rekursionsgleichung** vorliegen, die die optimale Lösung bzw. ihren Wert mithilfe von optimalen Lösungen kleinerer Teilprobleme ausdrückt.

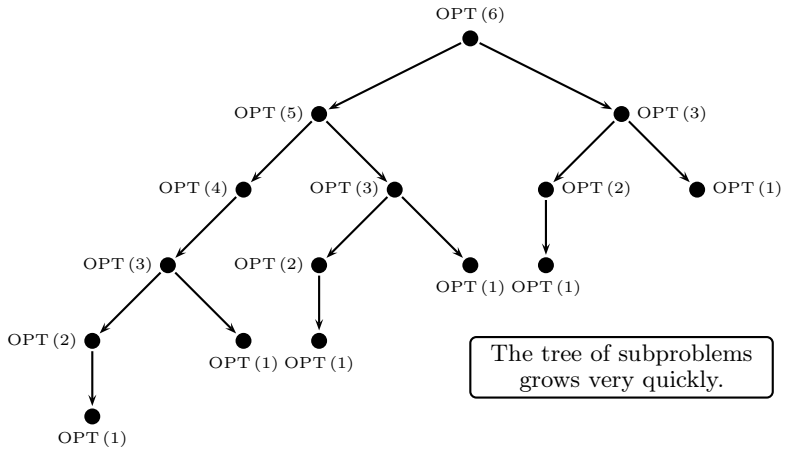
Anknüpfend an (13.1) ist der folgende rekursive Algorithmus zur Berechnung von $\mathrm{OPT}\,(j)$ naheliegend, bei dem vorausgesetzt wird, dass die Anfragen nach ihren Endzeiten aufsteigend geordnet sind und dass die Werte $p(j)$ für alle j vorliegen.

Compute-OPT (j)

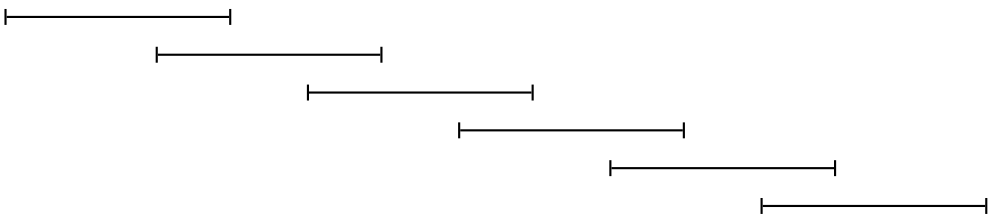
(1) **If** $j = 0$ **then**
(2) **Return** 0
(3) **Else**
(4) **Return** $\max\big(v_j + \text{Compute-OPT}\,(p(j)), \text{Compute-OPT}\,(j-1)\big)$
(5) **Endif**

Die Korrektheit des Algorithmus ergibt sich leicht durch vollständige Induktion (vgl. Kleinberg/Tardos). *Nun gibt es aber einen wichtigen Grund, weshalb man sich mit diesem Algorithmus auf keinen Fall zufrieden geben kann.*

Dieser Grund wird sehr schnell klar, wenn man sich die zugehörigen Rekursionsbäume anschaut. Die folgende Abbildung gibt beispielsweise den Rekursionsbaum wieder, der zu unserem früheren Beispiel mit sechs Intervallen gehört:



Wie man sieht, steigt die Zahl der rekursiven Aufrufe stark an, da etliche Berechnungen mehrfach ausgeführt werden. *Bereits bei sehr einfachen Beispielen kommt es zu extrem vielen rekursiven Aufrufen.* Um dies zu erkennen, betrachten wir ein recht „harmlos“ wirkendes Beispiel: Es seien n Intervalle $1, \ldots, n$ nach dem Schema der folgenden Figur angeordnet:



In diesem Beispiel gilt $p(j) = j - 2$ für alle $j = 2, \ldots, n$. Für $j \geq 2$ erzeugt Compute-OPT (j) also immer die rekursiven Aufrufe von Compute-OPT $(j-1)$ und von Compute-OPT $(j-2)$, d.h., die Gesamtzahl der rekursiven Aufrufe wächst wie die Fibonacci-Zahlen, welche exponentiell zunehmen. (Man beachte, dass für die Fibonacci-Zahlen $f(0), f(1), f(2), \ldots$ gilt: $f(n+2) = f(n+1) + f(n) \geq 2f(n)$. Das heißt: $f(n+2)$ ist bereits mindestens doppelt so groß wie $f(n)$.)

Aufgrund der hohen Zahl der rekursiven Aufrufe können wir mit unserem Algorithmus alles andere als zufrieden sein. Trotzdem sind wir gar nicht so weit davon entfernt, unser Problem auf eine weitaus bessere Art zu lösen: *Die entscheidende Beobachtung ist, dass unser rekursiver Algorithmus* Compute-OPT *in Wirklichkeit nur* $n+1$ *verschiedene Teilprobleme löst*:

$$\text{Compute-OPT}\,(0), \ldots, \text{Compute-OPT}\,(n).$$

Als *erste wichtige Komponente*, die bei einer Lösung mittels Dynamischer Programmierung vorhanden sein muss, hatten wir das *Vorliegen einer Rekursionsgleichung* bezeichnet. Die *zweite wichtige Komponente* haben wir soeben kennengelernt: Es muss eine Sammlung von „nicht allzu vielen[2]“ Teilproblemen vorliegen, auf die sich die rekursiven Aufrufe beschränken.

Das große Defizit unseres rekursiven Algorithmus lag in der extremen Häufigkeit, mit der immer wieder dieselben Teilprobleme aufgerufen werden. Wie kann man dies nun aber vermeiden? Es gibt zwei Möglichkeiten, die im Wesentlichen äquivalent sind:

- Die eine nennt man *Memoisation*; diese Möglichkeit soll hier nicht besprochen werden – wir verweisen auf die Literatur (z.B. Kleinberg/Tardos oder Cormen et al.).
- Die andere Möglichkeit ist, *iterativ* vorzugehen.

Im Fall unseres Scheduling-Problems geht das wie folgt: Wir nutzen aus, dass die Teilprobleme bereits in einer sich anbietenden Reihenfolge vorliegen; deswegen benutzen wir ein Array $M[0 \ldots n]$ der Länge $n+1$, in dem die Werte $\text{OPT}\,(0), \ldots, \text{OPT}\,(n)$ gespeichert werden. Beachtet man $\text{OPT}\,(0) = 0$ sowie die Rekursionsformel (13.1), *so ergibt sich der folgende iterative Algorithmus.*

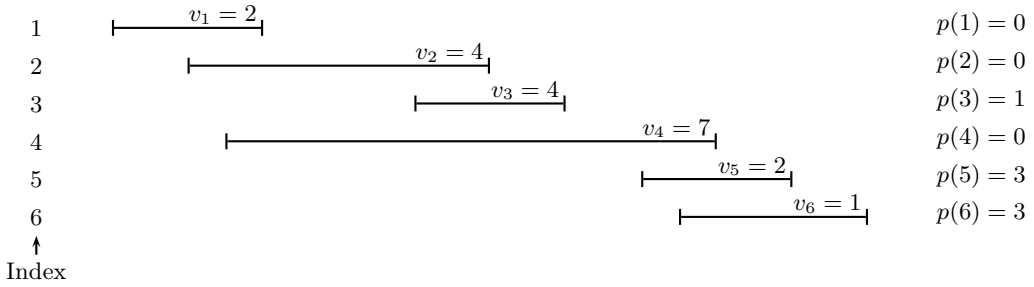
Iterative-Compute-OPT

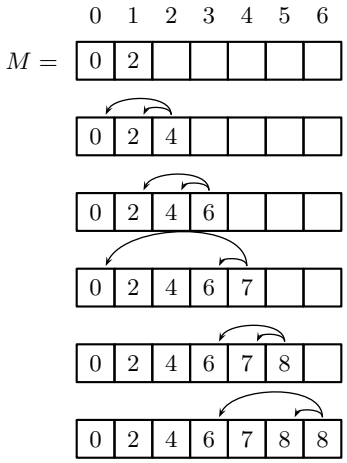
(1) $M[0] = 0$
(2) **For** $j = 1, \ldots, n$
(3) $\quad M[j] = \max\big(v_j + M[p(j)], M[j-1]\big)$
(4) **Endfor**

[2] Die vage Formulierung „nicht allzu viele“ Teilprobleme lässt sich dadurch präzisieren, dass man stattdessen von einer *polynomiellen Anzahl* von Teilproblemen spricht. Genauer: Die Anzahl der Teilprobleme soll *polynomiell* in n sein, wobei n ein Maß für die Größe der Eingabe ist. Anhand unseres Beispiels erläutert: n ist die Anzahl der Intervalle und es geht um $n+1$ Teilprobleme. In unserem Beispiel ist die Anzahl der Teilprobleme also sogar *linear* in n.

Die Korrektheit dieses Algorithmus ergibt sich aus (13.1) und $\mathrm{OPT}(0) = 0$. Die Laufzeit von Iterative-Compute-Opt ist $O(n)$, da $n+1$ Einträge von $M[0 \ldots n]$ berechnet werden und die Berechnung jedes Eintrags in konstanter Zeit erfolgt.

Wir greifen noch einmal das obige Beispiel aus Kleinberg/Tardos auf, um die Arbeitsweise von Iterative-Compute-OPT zu illustrieren:





Möchte man nicht nur $\mathrm{OPT}(n)$, sondern auch die dazugehörigen Intervalle der Menge $\mathcal{O}_n$ bestimmen, so ist dies auf der Basis von (13.2) leicht möglich; die (einfachen) Details findet man im Buch von Kleinberg und Tardos.

Unser Beispiel des gewichteten Intervall-Schedulings liefert eine *grobe Richtlinie* für den Entwurf von Algorithmen mittels Dynamischer Programmierung. Man benötigt vor allem eine *Menge von Teilproblemen* des Ausgangsproblems, die gewissen Grundbedingungen genügen:

(i) Es handelt sich um „nicht allzu viele“ Teilprobleme.

(ii) Die Lösung des Ausgangsproblems kann leicht aus den Lösungen der Teilprobleme berechnet werden. (Einfachster Fall: Das Ausgangsproblem selbst befindet sich unter den Teilproblemen.)

(iii) Es gibt eine „natürliche Ordnung“ unter den Teilproblemen sowie eine rekursive Beziehung, die es erlaubt, die Lösung eines Teilproblems auf die Lösung von kleineren Teilproblemen zurückzuführen. („Kleiner“ bezieht sich dabei auf die erwähnte Ordnung.)

Natürlich ist dies nur eine informelle Richtlinie, die teilweise etwas vage bleiben muss. *Die Schwierigkeit besteht meist darin, geeignete Teilprobleme zu einem gegebenen Problem aufzuspüren.*

Eine Vielzahl von interessanten Problemen, die mit Dynamischer Programmierung gelöst werden, findet man in den folgenden bekannten Lehrbüchern:

- Th. Cormen, Ch. Leiserson, R. Rivest, C. Stein: *Introduction to Algorithms*;
- S. Dasgupta, Ch. Papadimitriou, U. Vazirani: *Algorithms*;
- J. Kleinberg, É. Tardos: *Algorithm Design*.

Wir behandeln hier *zwei sehr grundlegende Probleme*, die häufig als Teilaufgabe bei der Lösung komplexerer Probleme auftreten:

- das *Rucksackproblem* (engl. *Knapsack Problem*);
- das *Problem der kürzesten Wege* in einem Graphen, in dem auch negative Kantenkosten vorkommen.

13.2 Ein Algorithmus zur Lösung des Rucksackproblems

Wir haben das *Rucksackproblem* (engl. *Knapsack Problem*) bereits zuvor kennengelernt. Hier befassen wir uns zunächst mit einem *Spezialfall des Rucksackproblems*, dass man das *Subset-Sum-Problem* nennt.

13.2.1 Das Subset-Sum-Problem

Gegeben seien n *Gegenstände* (engl. *items*), die wir mit $1, \ldots, n$ bezeichnen wollen. Der Gegenstand i habe das Gewicht $w_i \geq 0$, $w_i \in \mathbb{Z}$ $(i = 1, \ldots, n)$. Außerdem ist eine Schranke $W \geq 0$, $W \in \mathbb{Z}$ gegeben. Gesucht ist eine Teilmenge S von $\{1, \ldots, n\}$, für die

$$\sum_{i \in S} w_i \leq W$$

gilt und für die die links stehende Summe so groß wie möglich ist.

Vergleichen wir das Subset-Sum-Problem mit dem Rucksackproblem, so stellen wir fest, dass es sich beim Subset-Sum-Problem um einen Sonderfall des Rucksackproblems (in binärer Variante) handelt: um den Spezialfall nämlich, dass $v_i = w_i$ gilt $(i = 1, \ldots, n)$[3]. Die deutsche Bezeichnung für Subset-Sum-Problem ist *Teilsummenproblem*.

Entwurf des Algorithmus

Zunächst: *ein falscher Start.*

Da wir das Problem mit Dynamischer Programmierung behandeln wollen, müssen wir zunächst geeignete Teilprobleme finden. Wir orientieren uns am Problem des gewichteten Intervall-Schedulings und versuchen ähnlich vorzugehen, d.h., wir betrachten Teilprobleme der Form $\{1, \ldots, i\}$ und benutzen die Bezeichnung $\mathrm{OPT}(i)$ analog zur Bedeutung in Abschnitt 13.1.

Mit $\mathcal{O} \subseteq \{1, \ldots, n\}$ wollen wir eine optimale Lösung des Subset-Sum-Problems bezeichnen. Ebenso wie beim gewichteten Intervall-Scheduling-Problem unterscheiden wir die beiden Fälle $n \notin \mathcal{O}$ und $n \in \mathcal{O}$. Der Fall $n \notin \mathcal{O}$ geht ebenso wie zuvor, d.h., wir können auch diesmal feststellen:

- Falls $n \notin \mathcal{O}$, so gilt $\mathrm{OPT}(n) = \mathrm{OPT}(n-1)$.

Nun zum Fall $n \in \mathcal{O}$. Beim gewichteten Intervall-Scheduling war auch dieser Fall einfach: Wir konnten alle Anfragen weglassen, die nicht kompatibel mit n waren, und dann mit den restlichen Anfragen weiterarbeiten. *Diesmal hat $n \in \mathcal{O}$ aber etwas anderes zur Folge*: Die Aufnahme von n in $\mathcal{O}$ impliziert, dass für die Gegenstände aus $\{1, \ldots, n-1\}$ nur noch das Gewicht

$$W - w_n$$

zur Verfügung steht. Das bedeutet, dass es nicht ausreicht, Teilprobleme der Form $\{1, \ldots, i\}$ zu betrachten – es müssen zusätzlich auch noch kleinere Schranken w mit $0 \leq w \leq W$ Berücksichtigung finden.

[3] v_i bezeichnet den *Wert* (engl. *value*) des Gegenstands i. Mit „Rucksackproblem“ meinen wir im Folgenden immer die Version 1 auf Seite 65, bei der jeder Gegenstand nur einmal vorhanden ist (0,1-Problem).

Ganz so falsch war unser Start also doch nicht – wir müssen unseren Ansatz nur modifizieren, indem wir weitere Teilprobleme hinzunehmen. Dies wird im Folgenden durchgeführt.

Wie wir gesehen haben, ist es also nötig, für jede Menge $\{1, \ldots, i\}$ von Gegenständen und für jede Gewichtsobergrenze w mit $0 \leq w \leq W$ und $w \in \mathbb{Z}$ ein eigenes Teilproblem zu betrachten. Dementsprechend bezeichnen wir mit $\mathrm{OPT}(i, w)$ das Gewicht einer optimalen Lösung des Subset-Sum-Problems für die Menge $\{1, \ldots, i\}$ und für das maximal zulässige Gewicht w.

Man beachte, dass im Fall $w = 0$ immer $\mathrm{OPT}(i, w) = 0$ gilt, da das maximal zulässige Gewicht gleich 0 ist. Es gilt also immer $\mathrm{OPT}(i, 0) = 0$. Darüber hinaus definiert man noch $\mathrm{OPT}(0, w) = 0$. (Dies entspricht dem einleuchtenden Umstand, dass das Gewicht einer optimalen Lösung gleich 0 ist, wenn keine Gegenstände vorhanden sind.)

Für alle $i = 0, \ldots, n$ und alle ganzen Zahlen $0 \leq w \leq W$ haben wir somit den Wert $\mathrm{OPT}(i, w)$ definiert. Es sei daran erinnert, dass die Größe, die wir letzten Endes haben möchten, der Wert von $\mathrm{OPT}(n, W)$ ist. Wie zuvor sei $\mathcal{O} \subseteq \{1, \ldots, n\}$ eine dazugehörige optimale Lösung. Aufgrund unserer Überlegungen können wir feststellen:

- Falls $n \notin \mathcal{O}$, so gilt $\mathrm{OPT}(n, W) = \mathrm{OPT}(n-1, W)$.
- Falls $n \in \mathcal{O}$, so gilt $\mathrm{OPT}(n, W) = w_n + \mathrm{OPT}(n-1, W - w_n)$.

Falls $W < w_n$ gilt, d.h., falls der n-te Gegenstand zu groß ist, so liegt klarerweise der Fall $n \notin \mathcal{O}$ vor und es gilt $\mathrm{OPT}(n, W) = \mathrm{OPT}(n-1, W)$. Andernfalls ergibt sich, da einer der beiden Fälle $n \notin \mathcal{O}$ oder $n \in \mathcal{O}$ vorliegen muss:

$$\mathrm{OPT}(n, W) = \max\big(\mathrm{OPT}(n-1, W), w_n + \mathrm{OPT}(n-1, W - w_n)\big).$$

Entsprechendes gilt auch, wenn wir für $i \geq 1$ die Menge $\{1, \ldots, i\}$ anstelle von $\{1, \ldots, n\}$ sowie die Schranke w anstelle von W betrachten. Somit erhalten wir das folgende Ergebnis (für $i \geq 1$).

$$\mathrm{OPT}(i, w) = \begin{cases} \mathrm{OPT}(i-1, w) & \text{, für } w < w_i \\ \max\big(\mathrm{OPT}(i-1, w), w_i + \mathrm{OPT}(i-1, w - w_i)\big) & \text{, sonst} \end{cases} \tag{13.3}$$

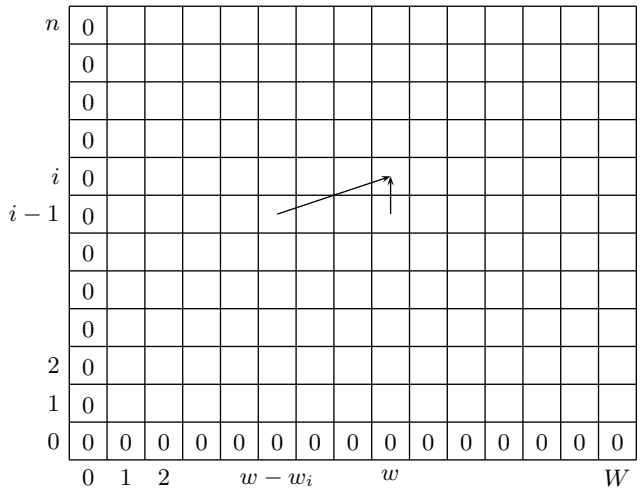
Damit haben wir die gewünschte *Rekursionsgleichung* aufgestellt. Anknüpfend an (13.3) erhält man den folgenden Algorithmus zur Berechnung von $\mathrm{OPT}(n, W)$.

Subset-Sum (n, W)

(1) Array $M[0 \ldots n, 0 \ldots W]$
(2) Initialize $M[0, w] = 0$ **for each** $w = 0, \ldots, W$
(3) **For** $i = 1, \ldots, n$
(4) **For** $w = 0, \ldots, W$
(5) Use the recurrence (13.3) to compute $M[i, w]$
(6) **Endfor**
(7) **Endfor**
(8) **Return** $M[n, W]$

Aufgrund von (13.3) ergibt sich (durch vollständige Induktion), dass dieser Algorithmus korrekt ist, d.h., dass der gelieferte Wert von $M[n, W]$ gleich $\mathrm{OPT}(n, W)$ ist.

Ähnlich wie wir im Fall des gewichteten Intervall-Scheduling-Problems das Array M in Form einer Tabelle dargestellt haben, deren Einträge schrittweise aus früheren Einträgen berechnet wurden, können wir uns M auch diesmal als eine Tabelle vorstellen. *Unterschied: Im vorliegenden Fall handelt es sich um eine 2-dimensionale Tabelle* (siehe nachfolgende Figur aus Kleinberg/Tardos), während wir es beim Intervall-Scheduling-Problem aus Abschnitt 13.1 nur mit einer einzigen Zeile zu tun hatten.



The two-dimensional table of OPT values. The leftmost column and bottom row is always 0. The entry $\mathrm{OPT}(i, w)$ is computed from two other entries $\mathrm{OPT}(i-1, w)$ and $\mathrm{OPT}(i-1, w-w_i)$, as indicated by the arrows.

Das folgende **Beispiel** aus Kleinberg/Tardos zeigt, wie die Tabelle zeilenweise von unten nach oben mit den Werten $\mathrm{OPT}(i, w)$ aufgefüllt wird.

Knapsack size $W = 6$, items $w_1 = 2$, $w_2 = 2$, $w_3 = 3$

	0	1	2	3	4	5	6
3							
2							
1							
0	0	0	0	0	0	0	0

Initial values

	0	1	2	3	4	5	6
3							
2							
(1)	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0

Filling in values for i = 1

	0	1	2	3	4	5	6
3							
(2)	0	0	2	2	4	4	4
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0

Filling in values for i = 2

	0	1	2	3	4	5	6
(3)	0	0	2	3	4	5	5
2	0	0	2	2	4	4	4
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0

Filling in values for i = 3

The iterations of the algorithm on a sample instance of the Subset-Sum-Problem.

Zur Laufzeit von Subset-Sum(n,W)

Es werden $(n+1)(W+1)$ Einträge berechnet und die Berechnung jedes Eintrags erfordert eine konstante Zahl von Operationen. Also beträgt die Laufzeit $O(nW)$. *Damit ist der* Subset-Sum-*Algorithmus nicht so effizient wie unser ebenfalls auf Dynamischer Programmierung basierender Algorithmus zum gewichteten Intervall-Scheduling-Problem* (vgl. Abschnitt 13.1). Insbesondere handelt es sich **nicht** um einen polynomiellen Algorithmus. (Man beachte: Um die Schranke W zu kodieren benötigt man lediglich $\lceil \log_2(W+1) \rceil$ Bits, während beim Aufstellen einer Zeile der 2-dimensionalen Tabelle $W+1$ Einträge zu berechnen sind.) Subset-Sum (n, W) arbeitet jedoch effizient, wenn die Schranke W im Vergleich zu

n nicht „übermäßig groß“ ist[4].

Der Algorithmus Subset-Sum (n, W) lässt sich leicht zu einem Algorithmus erweitern, der nicht nur den optimalen Wert $\mathrm{OPT}(n, W)$ liefert, sondern auch eine dazugehörige optimale Menge $\mathcal{O} \subseteq \{1, \ldots, n\}$ von Gegenständen. Um $\mathcal{O}$ zu erhalten, führt man zunächst den Algorithmus Subset-Sum (n, W) wie beschrieben durch. Anschließend bestimmt man $\mathcal{O}$ mithilfe des Arrays $M[0 \ldots n, 0 \ldots W]$. Wir nehmen an, dass $M[0 \ldots n, 0 \ldots W]$ in Form einer Tabelle vorliegt, die – wie zuvor beschrieben – von unten nach oben aufgebaut wurde. Das von Subset-Sum (n, W) gelieferte Ergebnis ist dann der Eintrag $M[n, W]$, der in dieser Tabelle an der Stelle $[n, W]$ steht („rechts oben“). Ausgehend von der Stelle $[n, W]$ wandert man nun durch die Tabelle, bis man unten angekommen ist, wobei man nach der folgenden Regel verfährt: Steht man auf der Stelle $[i, w]$ für $i > 0$ und gilt $M[i, w] = M[i-1, w]$, so nehme man i nicht in $\mathcal{O}$ auf und gehe nach $[i-1, w]$; gilt dagegen $M[i, w] > M[i-1, w]$, so nehme man i in $\mathcal{O}$ auf und gehe nach $[i-1, w-w_i]$.

Nun sind wir auch nicht mehr weit davon entfernt, einen Dynamischen-Programmierungs-Algorithmus für das *Rucksackproblem* zu haben. Kurz gesagt: *Beim Rucksackproblem läuft alles analog zum Subset-Sum-Problem; einziger Unterschied*: Es gilt nicht mehr unbedingt $v_i = w_i$. Dies wirkt sich nur insofern aus, dass man an einigen Stellen v_i statt w_i schreiben muss – ansonsten bleibt alles beim Alten.

13.2.2 Das Rucksackproblem

Hier ist die Originaldarstellung aus dem Buch von Kleinberg und Tardos:

The Knapsack Problem is a bit more complex than the scheduling problem we discussed earlier. Consider a situation in which each item i has a nonnegative weight w_i as before, and also a distinct *value* v_i. Our goal is now to find a subset S of maximum value $\sum_{i \in S} v_i$, subject to the restriction that the total weight of the set should not exceed W: $\sum_{i \in S} w_i \leq W$.

It is not hard to extend our dynamic programming algorithm to this more general problem. We use the analogous set of subproblems, $\mathrm{OPT}(i, w)$, to denote the value of the optimal solution using a subset of the items $\{1, \ldots, i\}$ and maximum available weight w. We consider an optimal solution $\mathcal{O}$, and identify two cases depending on whether or not $n \in \mathcal{O}$:

- If $n \notin \mathcal{O}$, then $\mathrm{OPT}(n, W) = \mathrm{OPT}(n-1, W)$.
- If $n \in \mathcal{O}$, then $\mathrm{OPT}(n, W) = v_n + \mathrm{OPT}(n-1, W-w_n)$.

Using this line of argument for the subproblems implies the following:

$$\mathrm{OPT}(i, w) = \begin{cases} \mathrm{OPT}(i-1, w) & \text{, if } w < w_i; \\ \max\big(\mathrm{OPT}(i-1, w), v_i + \mathrm{OPT}(i-1, w-w_i)\big) & \text{, otherwise.} \end{cases}$$

Using this recurrence, we can write down a completely analogous dynamic programming algorithm, and this implies the following fact.

The Knapsack Problem can be solved in $O(nW)$ time.

13.3 Der Algorithmus von Bellman und Ford

Es sei $G = (V, E)$ ein gerichteter Graph; jede Kante e von G sei mit einem *Gewicht* $c(e) \in \mathbb{R}$ versehen, wobei auch $c(e) < 0$ gelten kann.

Mögliche Interpretationen: Die Zahlen $c(e)$ geben *Kosten* an; negative Kosten lassen sich als Einnahmen interpretieren. Statt „Gewicht“ werden wir im Folgenden häufig auch „Kosten“ oder „Länge“ sagen.

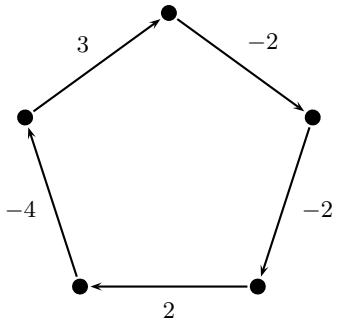
[4] Man spricht von einem *pseudo-polynomiellen Algorithmus*; Genaueres hierzu findet man beispielsweise im Buch von Kleinberg und Tardos.

Ist $G' = (V', E')$ ein Teilgraph von $G = (V, E)$, so bezeichnen wir mit $c(G')$ die Summe der Gewichte $c(e)$ für alle Kanten e von G'. Es gilt also

$$c(G') = \sum_{e \in E'} c(e).$$

Definition.
Ein gerichteter Kreis C von G wird ein *negativer Kreis* genannt, falls $c(C) < 0$ gilt.

Ein negativer Kreis:



Zwei eng verwandte Probleme:

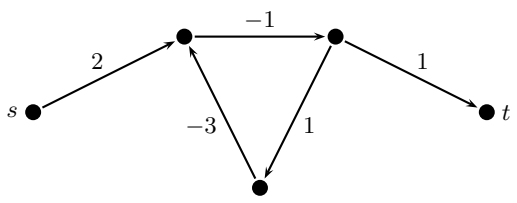
- *Das Problem der negativen Kreise*: Entscheide, ob G einen negativen Kreis besitzt.
- *Das Problem der kostenminimalen Pfade*: Gegeben seien ein Graph G ohne negative Kreise sowie ein Knoten s von G. Finde für jedes t, das von s aus erreichbar ist, einen s,t-Pfad P : $s = v_0, \ldots, v_k = t$ mit Kanten $e_i = (v_{i-1}, v_i)$ $(i = 1, \ldots, k)$, so dass die Summe

$$c(P) = \sum_{i=1}^{k} c(e_i)$$

  so klein wie möglich ist.

Statt „kostenminimaler s,t-Pfad“ sagen wir auch *kürzester s,t-Pfad*.

Wir wollen uns zunächst mit dem zweiten Problem, dem Problem der kostenminimalen Pfade beschäftigen. Der Grund, weshalb in diesem Problem die Existenz negativer Kreise ausgeschlossen wird, liegt auf der Hand: Andernfalls könnte es vorkommen, dass t zwar von s aus erreichbar ist, es aber trotzdem keinen kürzesten s,t-Pfad gibt. Man erkennt dies beispielsweise an der folgenden Darstellung: Zu jeder Zahl $K < 0$ kann man im dargestellten Graphen einen s,t-Pfad finden, dessen Länge kleiner als K ist – man muss den Kreis nur oft genug durchlaufen.



Der Dynamische-Programmierungs-Algorithmus, den wir entwickeln werden, fußt auf der folgenden Feststellung.

Feststellung 1.
Der Graph G besitze keine negativen Kreise und t sei ein von s aus erreichbarer Knoten. Dann gibt es immer einen kürzesten s,t-Pfad, der einfach ist und demnach höchstens $n-1$ Kanten besitzt[5].

Beweis. Wir zeigen zunächst, dass unter den Voraussetzungen von Feststellung 1 Folgendes gilt:

$$\text{In } G \text{ gibt es zu jedem } s,t\text{-Pfad } Q \text{ einen einfachen } s,t\text{-Pfad } P \text{ mit } c(P) \leq c(Q). \tag{13.4}$$

Nachweis von (13.4): Ist Q einfach, so ist (13.4) richtig, da man nur $P = Q$ zu wählen braucht. Wir können also annehmen, dass der Pfad

$$Q: \ s = v_0, \ldots, v_k = t$$

nicht einfach ist. Es gibt also eine Knotenwiederholung in Q, d.h., es gibt $i < j$ mit $v_i = v_j$. Wir betrachten eine Knotenwiederholung, für die $j - i$ so klein wie möglich ist. Dann sind die Knoten $v_i, v_{i+1}, \ldots, v_{j-1}$ alle verschieden, d.h., bei

$$C: \ v_i, v_{i+1}, \ldots, v_{j-1}, v_j$$

handelt es sich um einen Kreis. Wir betrachten den s,t-Pfad Q', bei dem der Kreis C ausgelassen wird:

$$Q': \ s = v_0, \ldots, v_i, v_{j+1}, \ldots, v_k = t.$$

Da C kein negativer Kreis ist, gilt $c(Q') \leq c(Q)$. Falls es in Q' nun immer noch Knotenwiederholungen gibt, so kann man das beschriebene Verfahren wiederholen, wodurch man nach endlich vielen Wiederholungen einen einfachen s,t-Pfad P mit $c(P) \leq c(Q)$ erhält. Dies zeigt die Richtigkeit von (13.4).

Nach dieser Vorarbeit ist es nicht schwer, die Richtigkeit von Feststellung 1 zu erkennen. Es sei $\mathcal{Q}$ die Menge sämtlicher s,t-Pfade in G und $\mathcal{P}$ sei die Menge aller einfachen s,t-Pfade in G. Aufgrund der Voraussetzung, dass t von s aus erreichbar ist, gilt $\mathcal{Q} \neq \emptyset$, woraus sich aufgrund von (13.4) ergibt, dass auch $\mathcal{P} \neq \emptyset$ gilt. Ein wesentlicher Unterschied zwischen $\mathcal{Q}$ und $\mathcal{P}$ besteht darin, dass $\mathcal{Q}$ eine unendliche Menge sein kann, während $\mathcal{P}$ garantiert eine *endliche* Menge ist. („Es kann unendlich viele s,t-Pfade in G geben; da jeder einfache s,t-Pfad in G höchstens $n-1$ Kanten besitzt, gibt es aber nur endlich viele einfache s,t-Pfade in G.“)

Da $\mathcal{P}$, wie soeben festgestellt, eine endliche nichtleere Menge ist, gibt es ein $P_0 \in \mathcal{P}$, so dass $c(P_0) \leq c(P)$ für alle $P \in \mathcal{P}$ gilt.

Es sei $Q \in \mathcal{Q}$ beliebig. Dann gibt es nach (13.4) ein $P \in \mathcal{P}$ mit $c(P) \leq c(Q)$. Es folgt

$$c(P_0) \leq c(P) \leq c(Q).$$

Der Pfad P_0 ist also ein kürzester s,t-Pfad, der einfach ist. Damit ist Feststellung 1 bewiesen. □

Im Folgenden sei immer ein Knoten s als „Startpunkt“ fest gewählt. Für $v \in V$ und $i \in \{0, 1, 2, \ldots\}$ bezeichne $\mathrm{OPT}(i, v)$ die kleinstmöglichen Kosten $c(P)$ eines s,v-Pfades P *mit höchstens i Kanten* – vorausgesetzt, ein solcher s,v-Pfad existiert.

Falls kein s,v-Pfad mit höchstens i Kanten existiert, so setzen wir $\mathrm{OPT}(i, v) = \infty$. Beispielsweise gilt $\mathrm{OPT}(0, s) = 0$ und $\mathrm{OPT}(0, v) = \infty$ für alle Knoten $v \neq s$.

Aufgrund der vorangegangenen Feststellung 1 handelt es sich bei

$$\mathrm{OPT}(n-1, t)$$

um die Größe, die wir berechnen wollen (für alle $t \in V$).

Zunächst geht es aber darum, für $i > 0$ die Größe $\mathrm{OPT}(i, v)$ mithilfe von kleineren Teilproblemen auf eine möglichst einfache Art auszudrücken. Bislang war dies immer dadurch geschehen, dass wir zwei Fälle

[5] Erklärung: Unter einem *einfachen Pfad* wird hier ein Pfad ohne Knotenwiederholung verstanden. Mit n sei immer die Anzahl der Knoten von G bezeichnet.

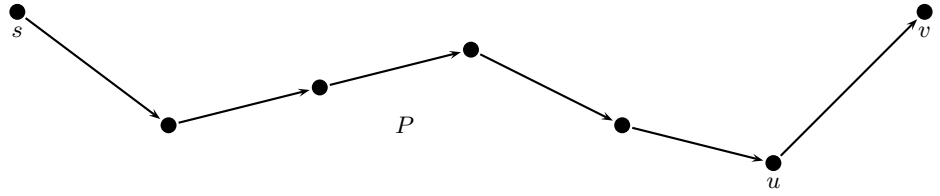
unterschieden haben ($n \in \mathcal{O}$ und $n \notin \mathcal{O}$). *Diesmal werden wir jedoch wesentlich mehr Fälle unterscheiden.* Wir nehmen an, dass $i > 0$ gilt und stellen uns vor, dass P ein s, v-Pfad mit höchstens i Kanten ist, für den $c(P) = \mathrm{OPT}\,(i, v)$ gilt.

Wir unterscheiden die Fälle, dass P höchstens $i-1$ Kanten besitzt und dass P genau i Kanten hat. Im ersten Fall ergibt sich Folgendes:

- Falls P höchstens $i-1$ Kanten besitzt, so gilt $\mathrm{OPT}\,(i, v) = \mathrm{OPT}\,(i-1, v)$.

Im Fall, dass P genau i Kanten hat, sei der Vorgänger von v auf P mit u bezeichnet (siehe Zeichnung). Es folgt:

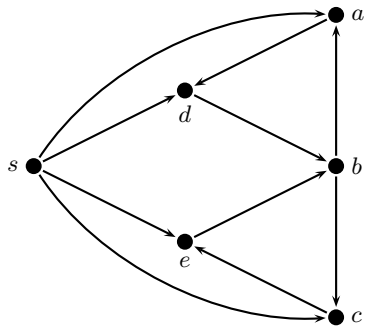
- Falls P genau i Kanten enthält, so gilt $\mathrm{OPT}\,(i, v) = \mathrm{OPT}\,(i-1, u) + c(u, v)$.



Bezeichnung: Ist ein beliebiges $v \in V$ gegeben, so betrachten wir die Menge aller Knoten $w \in V$, für die die Kante (w, v) in G vorhanden ist. Da diese Menge im Folgenden eine besondere Rolle spielen wird, führen wir eine Bezeichnung für sie ein:

$$N^-(v) = \Big\{ w \in V : (w, v) \in E \Big\}$$

Zur Illustration ein **Beispiel**: $G = (V, E)$ sei der folgende Graph.



Dann gilt:

$$\begin{aligned}
N^-(s) &= \emptyset \\
N^-(a) &= \big\{ s, b \big\} \\
N^-(b) &= \big\{ d, e \big\} \\
N^-(c) &= \big\{ s, b \big\} \\
N^-(d) &= \big\{ s, a \big\} \\
N^-(e) &= \big\{ s, c \big\}.
\end{aligned}$$

Aus den zuvor gemachten Beobachtungen ergibt sich die folgende rekursive Formel:

$$\mathrm{OPT}\,(i, v) = \min\Big(\mathrm{OPT}\,(i-1, v), \min_{w \in N^-(v)} \big(\mathrm{OPT}\,(i-1, w) + c(w, v) \big) \Big) \quad (\text{für } i > 0). \tag{13.5}$$

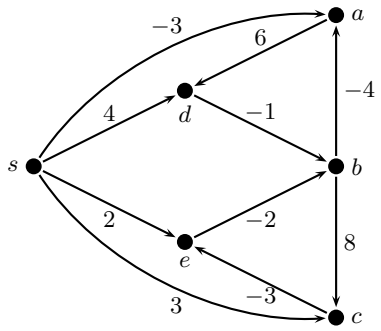
Unter Verwendung der Rekursionsformel (13.5) erhält man den folgenden Dynamischen-Programmierungs-Algorithmus zur Berechnung von $\mathrm{OPT}\,(n-1, t)$ für alle $t \in V$, der als *Algorithmus von Bellman und Ford* bekannt ist.

Shortest-Path(G, c, s)

(1) n = number of nodes in G
(2) Array $M[0 \ldots n-1, V]$
(3) Define $M[0, s] = 0$ and $M[0, v] = \infty$ for all $v \neq s$
(4) **For** $i = 1, \ldots, n-1$
(5) **For** $v \in V$ in any order
(6) Compute $M[i, v]$ using (13.5)
(7) **Endfor**
(8) **Endfor**
(9) **Return** $M[n-1, t]$ for all $t \in V$

Die Korrektheit dieser Methode ergibt sich direkt aus der Rekursion (13.5) (durch vollständige Induktion). Für die Laufzeit erhält man Folgendes: Das Array M hat n^2 Einträge und zur Berechnung eines einzelnen Eintrags sind höchstens n Additionen und höchstens n Vergleiche zweier Zahlen erforderlich (vgl. (13.5)). Insgesamt benötigt der Algorithmus also $O(n^3)$ Operationen, wobei unter einer Operation hier eine Addition oder ein Vergleich zweier Zahlen verstanden werden soll.

Wir erläutern die Arbeitsweise des Algorithmus von Bellman-Ford anhand des folgenden **Beispiels**:



Das Array M stellen wir als eine 6×6-Matrix dar, in die anfangs nur die Werte der ersten Zeile eingetragen sind. Danach werden die Einträge zeilenweise berechnet, wobei sich die Werte der i-ten Zeile unter Benutzung von (13.5) aus der $(i-1)$-ten Zeile ergeben. Man erhält die folgende Matrix M:

	s	a	b	c	d	e
0	0	∞	∞	∞	∞	∞
1	0	-3	∞	3	4	2
2	0	-3	0	3	3	0
3	0	-4	-2	3	3	0
4	0	-6	-2	3	2	0
5	0	-6	-2	3	0	0

Für jeden Knoten $t \in V$ lässt sich in der letzten Zeile von M die Länge eines kürzesten s, t-Pfades in G ablesen. Beispielsweise erhält man:

- die Länge eines kürzesten s, a-Pfades ist gleich -6;
- die Länge eines kürzesten s, d-Pfades ist gleich 0.

Aufgrund der Konstruktion besitzen auch die Einträge der übrigen Zeilen eine anschauliche Bedeutung; beispielsweise bedeutet der Eintrag 2 in der vorletzten Zeile:

- Die Länge eines kürzesten s, d-Pfades mit höchstens vier Kanten ist gleich 2.

Will man nicht nur die Längen kürzester Pfade berechnen, sondern auch entsprechende Pfade P selbst, so ist es zweckmäßig, zusätzlich zum Wert $\mathrm{OPT}(i, v)$ einen Knoten w zur Verfügung zu haben, der auf einem entsprechenden Pfad der direkte Vorgänger von v ist (für $v \neq s$ und falls $\mathrm{OPT}(i, v) \neq \infty$).

Zu jedem Eintrag $\mathrm{OPT}(i, v)$, für den $v \neq s$ und $\mathrm{OPT}(i, v) \neq \infty$ gilt, speichern wir deshalb in M als zusätzliche Information einen Knoten:

- Falls $\mathrm{OPT}(i, v) = \mathrm{OPT}(i-1, v)$ gilt, so sei dies derselbe Knoten, der zuvor zusammen mit $\mathrm{OPT}(i-1, v)$ gespeichert wurde.
- Andernfalls speichere man einen Knoten $w \in N^-(v)$, für den gilt:

$$\mathrm{OPT}(i, v) = \mathrm{OPT}(i-1, w) + c(w, v).$$

Mithilfe dieser zusätzlichen Einträge lässt sich dann P zurückverfolgen: Wir schauen uns dies in unserem Beispiel an. Die durch Zusatzeinträge ergänzte Matrix sieht in unserem Beispiel wie folgt aus:

	s	a	b	c	d	e
0	0 $-$	∞ $-$	∞ $-$	∞ $-$	∞ $-$	∞ $-$
1	0 $-$	-3 s	∞ $-$	3 s	4 s	2 s
2	0 $-$	-3 s	0 e	3 s	3 a	0 c
3	0 $-$	-4 b	-2 e	3 s	3 a	0 c
4	0 $-$	-6 b	-2 e	3 s	2 a	0 c
5	0 $-$	-6 b	-2 e	3 s	0 a	0 c

Wir haben weiter oben bereits festgestellt: Die Länge eines kürzesten s, d-Pfades ist gleich 0. Um zusätzlich einen kürzesten s, d-Pfad P zu ermitteln, benötigen wir nur die letzte Zeile: Wir entnehmen der letzten Zeile zunächst, dass a als Vorgänger von d auf P zu wählen ist; nun schauen wir in die Spalte, die zu a gehört, und ermitteln b als Vorgänger von a auf dem zu konstruierenden Pfad P; entsprechend geht es weiter: Ein Blick in die Spalte von b führt zu e als Vorgänger von b; danach ermittelt man c als Vorgänger von e, und abschließend erhält man s als Vorgänger von c. Es ergibt sich:

$$P = (s, c, e, b, a, d).$$

Wir haben bislang vorausgesetzt, dass G keine negativen Kreise enthält. Der Grund dafür war, dass negative Kreise – sofern sie von s aus erreichbar sind – für unsere Fragestellung äußerst „schädlich“ sind: vgl. das Beispiel vor Feststellung 1.

Wie findet man nun heraus, ob von s aus erreichbare, negative Kreise vorhanden sind? Zum Glück ist das ganz einfach: Man hat nichts weiter zu tun, als den Algorithmus von Bellman und Ford wie gewohnt auf G anzuwenden und ganz am Ende noch zusätzlich eine n-te Runde einzulegen, d.h., man berechnet durch Anwenden der Formel (13.5) zusätzlich Werte $M[n, v]$ für alle $v \in V$; dadurch erhält die Matrix M eine zusätzliche Zeile. Es gilt die folgende Feststellung (Beweis: siehe unten):

Feststellung 2.
G enthält genau dann einen negativen Kreis, der von s aus erreichbar ist, wenn $M[n, v] \neq M[n-1, v]$ für mindestens ein $v \in V$ gilt.

Anders gesagt: Stimmt die letzte Zeile (d.h. die Zeile mit den Werten $M[n, v]$) **nicht** mit der vorletzten Zeile überein, so brechen wir das Verfahren ab mit dem Ergebnis, dass ein von s aus erreichbarer negativer Kreis vorhanden ist.

Stimmen die beiden genannten Zeilen dagegen überein, so gibt es keine von s aus erreichbaren negativen Kreise in G. In diesem Fall können wir die Länge kürzester s, t-Pfade an der letzten (oder vorletzten) Zeile von M ablesen; dabei bedeutet der Eintrag ∞, dass der entsprechende Knoten nicht von s erreichbar ist.

Wir fügen einen Beweis von Feststellung 2 an, wobei einige Details dem Leser überlassen bleiben. Zunächst beobachten wir, dass die folgende leichte Verallgemeinerung von Feststellung 1 ebenfalls gültig ist.

Feststellung 1'.

Der Graph G besitze keine negativen Kreise, die von s aus erreichbar sind, und t sei ein von s aus erreichbarer Knoten. Dann gibt es immer einen kürzesten s,t-Pfad, der einfach ist und folglich höchstens $n-1$ Kanten besitzt.

Man beweist Feststellung 1' auf die gleiche Art wie Feststellung 1.

Beweis von Feststellung 2. Gilt $M[n,v] \neq M[n-1,v]$ für mindestens ein $v \in V$, so erhält man mithilfe von Feststellung 1', dass G einen negativen Kreis enthalten muss, der von s aus erreichbar ist. Gilt dagegen $M[n,v] = M[n-1,v]$ für alle $v \in V$, so stelle man sich vor, dass mithilfe der Formel (13.5) weitere Zeilen der Matrix M berechnet werden: Man erkennt, dass sich auch alle nachfolgenden Zeilen nicht mehr ändern, d.h., für alle $i \geq n$ und alle $v \in V$ gilt $M[i,v] = M[n-1,v]$. Folglich kann G keinen negativen Kreis enthalten, der von s aus erreichbar ist: Für die Knoten v eines solchen Kreises würden die Werte $M[i,v]$ nicht für alle $i \geq n$ gleich bleiben. □

13.4 Kürzeste Pfade „all-to-all": Der Algorithmus von Floyd und Warshall

Gegeben sei ein gerichteter Graph $G = (V,E)$. Die Knoten von G seien mit $1,\ldots,n$ bezeichnet, es gelte also $V = \{1,\ldots,n\}$. Außerdem sei für jede gerichtete Kante $(i,j) \in E$ ein Gewicht w_{ij} gegeben, wobei $w_{ij} < 0$ erlaubt ist. Mit w sei die Abbildung bezeichnet, die jeder Kante von G ihr Gewicht w_{ij} zugeordnet. Unter den genannten Voraussetzungen sprechen wir auch von einem *gewichteten Graphen* (G,w).

Ist nichts anderes gesagt, so seien negative Kreise ausgeschlossen. Statt vom „Gewicht" einer Kante sprechen wir auch von ihrer *Länge* oder ihren *Kosten*. Unter dem *Abstand* (oder der *Distanz*) von i nach j verstehen wir die Länge eines kürzesten i,j-Pfades in G.

In manchen Situationen möchte man nicht nur die Abstände von einem festen Knoten s berechnen („one-to-all"), sondern man interessiert sich für die *Abstände zwischen sämtlichen Knotenpaaren* („all-to-all").

Eine Möglichkeit, das Problem „all-to-all" zu behandeln, besteht darin, den Algorithmus von Bellman und Ford wiederholt anzuwenden, wobei jeder der n Knoten genau einmal die Rolle des Ausgangsknotens s übernimmt. Da der Algorithmus von Bellman und Ford die Komplexität $O(n^3)$ besitzt, gelangt man auf diese Art zu einem Algorithmus der Komplexität $O(n^4)$.

Der häufig benutzte Algorithmus von Floyd und Warshall, den wir im Folgenden besprechen werden, löst ebenfalls das Problem „all-to-all"; er ist jedoch effizienter als die zuvor beschriebene Methode: Die Komplexität des Algorithmus von Floyd und Warshall ist $O(n^3)$.

Für $i \neq j$ setzen wir zusätzlich $w_{ij} = \infty$, falls es in G keine von i nach j gerichtete Kante gibt.

Der Algorithmus von Floyd und Warshall ist sehr einfach zu beschreiben und auszuführen. Wir legen hier die Darstellung aus dem Buch von Jungnickel zugrunde (mit leichten Modifikationen):

Algorithmus von Floyd und Warshall

```
(1)  for i = 1 to n do
(2)      for j = 1 to n do
(3)          if i ≠ j then d(i,j) ← w_ij else d(i,j) ← 0 fi
(4)      od
(5)  od
(6)  for k = 1 to n do
(7)      for i = 1 to n do
(8)          for j = 1 to n do
(9)              if i ≠ k and j ≠ k then d(i,j) ← min{d(i,j), d(i,k) + d(k,j)} fi
(10)         od
(11)     od
(12) od
```

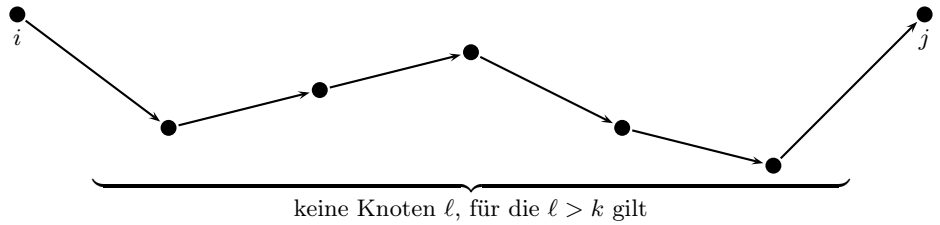
Mit $D_0 = \left(d_{ij}^{(0)}\right)$ sei die $n \times n$ - Matrix bezeichnet, die durch Zeile (3) definiert wird. Außerdem: $D_k = \left(d_{ij}^{(k)}\right)$ sei die Matrix, die im k-ten Durchlauf der äußeren For-Schleife (Zeile (6)-(12)) generiert wird $(k = 1, \ldots, n)$.
Für alle $k \in \{0, \ldots, n\}$ und alle $i, j \in \{1, \ldots, n\}$ bezeichnen wir mit

$$\mathcal{P}_{ij}^{(k)}$$

die Menge aller i, j-Pfade P von G, für die gilt:

$$\text{Es gibt in } P \text{ keine inneren Knoten } \ell, \text{ für die } \ell > k \text{ gilt.} \qquad (\star)$$

Etwas anders gesagt: $P \in \mathcal{P}_{ij}^{(k)}$ bedeutet, dass P ein i, j-Pfad ist, bei dem Knoten ℓ mit $\ell > k$ als innere Knoten nicht vorkommen (siehe Skizze).



In der nachfolgenden Feststellung, die man unschwer durch vollständige Induktion beweist (als Übungsaufgabe empfohlen!), wird die Bedeutung der Einträge $d_{ij}^{(k)}$ der Matrizen D_k festgehalten.

Feststellung.
Für alle $k = 0, 1, \ldots, n$ sowie für alle $i, j \in \{1, \ldots, n\}$ gilt:

- Falls $\mathcal{P}_{ij}^{(k)} \neq \emptyset$, so gibt $d_{ij}^{(k)}$ die kleinstmögliche Länge eines Pfades $P \in \mathcal{P}_{ij}^{(k)}$ an.
- Falls $\mathcal{P}_{ij}^{(k)} = \emptyset$, so gilt $d_{ij}^{(k)} = \infty$.

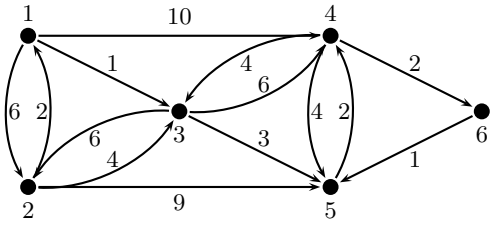
Da $D_n = \left(d_{ij}^{(n)}\right)$ die Matrix ist, die der Algorithmus von Floyd und Warshall am Ende abliefert, interessiert uns natürlich der Fall $k = n$ ganz besonders. Schaut man sich die Bedingung $(\star)$ für den Fall $k = n$ an, so erkennt man, dass es sich bei

$$\mathcal{P}_{ij}^{(n)}$$

um die Menge *sämtlicher* i, j-Pfade in G handelt. Für den Fall $k = n$ bedeutet unsere Feststellung also:

- Falls es in G einen i,j-Pfad gibt, so gibt $d_{ij}^{(n)}$ die Länge eines kürzesten i,j-Pfades in G an.
- Falls es in G keinen i,j-Pfad gibt, so gilt $d_{ij}^{(n)} = \infty$.

Beispiel. Wir betrachten den folgenden Graphen mit Kantengewichten w_{ij} wie angegeben:



Man erhält die folgende Matrix D_0:

$$D_0 = \begin{bmatrix} 0 & 6 & 1 & 10 & \infty & \infty \\ 2 & 0 & 4 & \infty & 9 & \infty \\ \infty & 6 & 0 & 6 & 3 & \infty \\ \infty & \infty & 4 & 0 & 4 & 2 \\ \infty & \infty & \infty & 2 & 0 & \infty \\ \infty & \infty & \infty & \infty & 1 & 0 \end{bmatrix}.$$

Es gibt im Folgenden keine Veranlassung mehr, auf die Zeichnung zu schauen: Die gesamte Information der Zeichnung wurde in die Matrix D_0 übertragen. Die Matrix D_k erhält man aus der Matrix D_{k-1}, indem man die folgenden Rekursionsformeln anwendet:

Für $k = 1, \ldots, n$ gilt:

$$d_{ij}^{(k)} = d_{ij}^{(k-1)} \text{ falls } i = k \text{ oder } j = k \tag{13.6}$$

$$d_{ij}^{(k)} = \min\left\{ d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \right\} \text{ falls } i \neq k \text{ und } j \neq k. \tag{13.7}$$

Man mache sich klar, dass sich die Formeln (13.6) und (13.7) unmittelbar aus Zeile (9) des Algorithmus von Floyd und Warshall ergeben.

Aufgrund von (13.6) ändern sich beim Übergang von D_{k-1} zu D_k die k-te Zeile und die k-te Spalte nicht. Führt man den Algorithmus von Floyd und Warshall per Hand aus, so ist es beim Aufstellen der Matrix D_k zweckmäßig, zunächst die k-te Zeile und die k-te Spalte von D_{k-1} direkt in D_k zu übernehmen. Gehen wir in unserem Beispiel von D_0 zu D_1 über, so schreiben wir also zunächst die 1. Zeile und die 1. Spalte von D_0 ab; man erhält eine Matrix, in der die meisten Einträge noch fehlen:

$$\begin{bmatrix} 0 & 6 & 1 & 10 & \infty & \infty \\ 2 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \infty & \cdot & \cdot & \cdot & \cdot & \cdot \\ \infty & \cdot & \cdot & \cdot & \cdot & \cdot \\ \infty & \cdot & \cdot & \cdot & \cdot & \cdot \\ \infty & \cdot & \cdot & \cdot & \cdot & \cdot \end{bmatrix}.$$

Danach trägt man dann gemäß (13.7) die übrigen Einträge ein; man erhält

$$D_1 = \begin{bmatrix} 0 & 6 & 1 & 10 & \infty & \infty \\ 2 & 0 & 3 & 12 & 9 & \infty \\ \infty & 6 & 0 & 6 & 3 & \infty \\ \infty & \infty & 4 & 0 & 4 & 2 \\ \infty & \infty & \infty & 2 & 0 & \infty \\ \infty & \infty & \infty & \infty & 1 & 0 \end{bmatrix}.$$

In ähnlicher Weise erhält man D_2, indem man zunächst die 2. Zeile und die 2. Spalte von D_1 abschreibt:

$$\begin{bmatrix} \cdot & 6 & \cdot & \cdot & \cdot & \cdot \\ 2 & 0 & 3 & 12 & 9 & \infty \\ \cdot & 6 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \infty & \cdot & \cdot & \cdot & \cdot \\ \cdot & \infty & \cdot & \cdot & \cdot & \cdot \\ \cdot & \infty & \cdot & \cdot & \cdot & \cdot \end{bmatrix}.$$

Anschließendes Auffüllen der Matrix gemäß (13.7) ergibt

$$D_2 = \begin{bmatrix} 0 & 6 & 1 & 10 & 15 & \infty \\ 2 & 0 & 3 & 12 & 9 & \infty \\ 8 & 6 & 0 & 6 & 3 & \infty \\ \infty & \infty & 4 & 0 & 4 & 2 \\ \infty & \infty & \infty & 2 & 0 & \infty \\ \infty & \infty & \infty & \infty & 1 & 0 \end{bmatrix}.$$

Fährt man entsprechend fort, so erhält man der Reihe nach die folgenden Matrizen (Prüfen Sie dies nach!):

$$D_3 = \begin{bmatrix} 0 & 6 & 1 & 7 & 4 & \infty \\ 2 & 0 & 3 & 9 & 6 & \infty \\ 8 & 6 & 0 & 6 & 3 & \infty \\ 12 & 10 & 4 & 0 & 4 & 2 \\ \infty & \infty & \infty & 2 & 0 & \infty \\ \infty & \infty & \infty & \infty & 1 & 0 \end{bmatrix}.$$

$$D_4 = \begin{bmatrix} 0 & 6 & 1 & 7 & 4 & 9 \\ 2 & 0 & 3 & 9 & 6 & 11 \\ 8 & 6 & 0 & 6 & 3 & 8 \\ 12 & 10 & 4 & 0 & 4 & 2 \\ 14 & 12 & 6 & 2 & 0 & 4 \\ \infty & \infty & \infty & \infty & 1 & 0 \end{bmatrix}.$$

$$D_5 = \begin{bmatrix} 0 & 6 & 1 & 6 & 4 & 8 \\ 2 & 0 & 3 & 8 & 6 & 10 \\ 8 & 6 & 0 & 5 & 3 & 7 \\ 12 & 10 & 4 & 0 & 4 & 2 \\ 14 & 12 & 6 & 2 & 0 & 4 \\ 15 & 13 & 7 & 3 & 1 & 0 \end{bmatrix}.$$

$$D_6 = \begin{bmatrix} 0 & 6 & 1 & 6 & 4 & 8 \\ 2 & 0 & 3 & 8 & 6 & 10 \\ 8 & 6 & 0 & 5 & 3 & 7 \\ 12 & 10 & 4 & 0 & 3 & 2 \\ 14 & 12 & 6 & 2 & 0 & 4 \\ 15 & 13 & 7 & 3 & 1 & 0 \end{bmatrix}.$$

Die Matrix D_6 des Beispiels – bzw. allgemein die vom Algorithmus von Floyd und Warshall gelieferte Matrix D_n – nennt man die *Distanzmatrix* des gewichteten Graphen (G, w). Es ist üblich, auch der Matrix D_0 einen Namen zu geben: Man nennt D_0 die *Adjazenzmatrix* von (G, w). Der Algorithmus von Floyd und Warshall wandelt also die Adjazenzmatrix D_0 in die Distanzmatrix D_n um. Dies geschieht stufenweise: Den Matrizen $D_1, \ldots, D_{n-1}$ kommt die Rolle von Zwischenstufen zu. Man beachte, dass für diesen Umwandlungsprozess nur ein einziges $n \times n$ - Array zum Speichern der Matrizeneinträge benötigt wird (vgl. Zeile (9) des Algorithmus).

Die Komplexität $O(n^3)$ des Algorithmus von Floyd und Warshall ergibt sich unmittelbar aus den Zeilen (6)-(12).

Auf Seite 202 f. haben wir dargelegt, unter welchen Bedingungen man von einer algorithmischen Lösung mittels *Dynamischer Programmierung* spricht.

Aufgabe: Überlegen Sie sich, dass der Algorithmus von Floyd und Warshall den auf Seite 202 f. genannten Anforderungen genügt.

Der Algorithmus von Floyd und Warshall ist also ein weiteres Beispiel für einen Algorithmus der Gattung *Dynamische Programmierung*.

Weitere Aufgaben:

1. Es sei (G, w) ein gewichteter Graph, von dem nicht bekannt ist, ob er einen negativen Kreis enthält. Wie kann der Algorithmus von Floyd und Warshall eingesetzt werden, um herauszufinden, ob ein negativer Kreis in (G, w) vorhanden ist?
2. Es ist möglich, den Algorithmus von Floyd und Warshall so zu modifizieren, dass nicht nur die Distanzen für alle Knotenpaare, sondern auch die dazugehörigen kürzesten Pfade geliefert werden. Wie kann das geschehen?

14 Polynomielle Algorithmen für Lineare Programmierung

14.1 Einführende Bemerkungen

Der Simplexalgorithmus bewährt sich seit Jahrzehnten in der Praxis hervorragend, obwohl er vom Standpunkt der Theorie einen „Schönheitsfehler“ hat: Es handelt sich nicht um einen polynomiellen Algorithmus. Genauer: *Es ist keine Pivotierungsregel bekannt, bei deren Verwendung das Simplexverfahren zu einem Algorithmus mit polynomieller Laufzeit wird.* Verwendet man beispielsweise die Regel vom größten Koeffizienten, so erkennt man anhand der Klee-Minty-Beispiele, dass kein polynomieller Algorithmus vorliegt.

Es war viele Jahre ebenfalls eine offene Frage, ob es überhaupt einen polynomiellen Algorithmus für Lineare Programmierung gibt.

Im Jahr 1979 gelang es *L. Khachiyan*, diese Frage zu beantworten: Das erregte damals großes Aufsehen und die größten Tageszeitungen der Welt berichteten darüber – teilweise auf der ersten Seite. Allerdings stellten die meisten Journalisten die Sache als einen Durchbruch mit unmittelbaren Auswirkungen auf die Praxis dar – was gewiss nicht zutraf.

Was Khachiyan in einer wissenschaftlichen Publikation dargestellt hatte, war dies: Er griff ein Verfahren namens *Ellipsoid-Methode* auf, das zuvor von Shor, Judin und Nemirovski für nichtlineare Optimierungsprobleme entwickelt worden war, und wies nach, dass diese Methode bei entsprechender Anpassung zu einem polynomiellen Algorithmus für Lineare Programmierung führt.

Mit der Ellipsoid-Methode lag also zum ersten Mal ein polynomieller Algorithmus für Lineare Programmierung vor.

Besonders interessant ist die Ellipsoid-Methode auch deshalb, weil an LP-Probleme auf eine völlig andere Art herangegangen wird als beim Simplexverfahren. *Schon allein wegen dieses frappierenden Unterschieds lohnt es sich, die Ellipsoid-Methode zumindest in ihren Grundzügen zu kennen.*

Im Folgenden werden die *Grundideen der Ellipsoid-Methode* vorgestellt, wobei wir teilweise der Darstellung des folgenden Lehrbuchs folgen:

- J. Matoušek, B. Gärtner: *Understanding and Using Linear Programming.* Springer-Verlag. Berlin Heidelberg (2007).

Ein weiterer polynomieller Algorithmus für Lineare Programmierung wurde im Jahre 1984 von *N. Karmarkar* präsentiert; es handelt sich dabei um eine *Innere-Punkte-Methode*. Ebenfalls auf der Basis des genannten Lehrbuchs werden wir auch Innere-Punkte-Methoden behandeln – zumindest werden wir einige der Grundideen kennenlernen.

14.2 Die Eingabelänge eines LP-Problems und der Begriff des polynomiellen Algorithmus

Um genau festzulegen, was unter einem polynomiellen Algorithmus für Lineare Programmierung verstanden werden soll, haben wir zunächst die Eingabelänge eines LP-Problems zu definieren. *Grob gesagt versteht man unter der Eingabelänge eines LP-Problems die Gesamtzahl der Bits, die nötig sind, um*

sämtliche in der Zielfunktion und in den Nebenbedingungen auftretenden Koeffizienten in Binärdarstellung aufzuschreiben (vgl. auch Seite 58).

Dies soll nun präzisiert werden. Ist eine ganze Zahl i gegeben, so verstehen wir unter der *Bitlänge von* i die folgende mit $\langle i \rangle$ bezeichnete Zahl:

$$\langle i \rangle = \lceil \log_2 (|i| + 1) \rceil + 1.$$

Erläuterung: Für $i \neq 0$ ist $\lceil \log_2 (|i| + 1) \rceil$ die Anzahl der benötigten Bits, um $|i|$ in Binärdarstellung zu kodieren; ein weiteres Bit kommt hinzu, um das Vorzeichen festzulegen. Für $i = 0$ gilt $\langle i \rangle = 1$ (wegen $\log_2 (1) = 0$).

Liegt eine rationale Zahl $r = \frac{p}{q}$ vor (für $p, q \in \mathbb{Z}$ mit $q \neq 0$), so definieren wir die *Bitlänge von* r durch

$$\langle r \rangle = \langle p \rangle + \langle q \rangle.$$

Haben wir es mit einem Vektor $v = (v_1, \ldots, v_n)$ zu tun, dessen Einträge v_i rationale Zahlen sind, so definieren wir die *Bitlänge von* v durch

$$\langle v \rangle = \sum_{i=1}^{n} \langle v_i \rangle.$$

Ähnlich verfahren wir mit Matrizen: Ist $A = (a_{ij})$ eine $m \times n$ - Matrix mit rationalen Einträgen (d.h. $a_{ij} \in \mathbb{Q}$), so sei

$$\langle A \rangle = \sum_{i=1}^{m} \sum_{j=1}^{n} \langle a_{ij} \rangle.$$

Bislang haben wir meistens angenommen, dass ein LP-Problem in Standardform (vgl. Skript Seite 7) vorliegt. Davon soll jetzt leicht abgewichen werden: In diesem Abschnitt soll – sofern nichts anderes gesagt ist – vorausgesetzt werden, dass LP-Probleme in der folgenden Form vorliegen:

$$\begin{aligned} &\text{maximiere } c^T x \\ &\text{unter den Nebenbedingungen} \\ &\qquad Ax \leq b. \end{aligned} \tag{14.1}$$

In der Darstellung (14.1) ist A eine $m \times n$ - Matrix; c und x sind Vektoren der Länge n und b ist ein Vektor der Länge m.

Der Unterschied zwischen der Darstellung (14.1) und der bislang bevorzugten Standardform besteht lediglich darin, dass mit Nichtnegativitätsbedingungen etwas anders umgegangen wird: Während in der Standardform die Nichtnegativitätsbedingungen gesondert hingeschrieben werden (in der Form $x \geq 0$), haben möglicherweise vorhandene Nichtnegativitätsbedingungen in der Darstellung (14.1) keinen Sonderstatus mehr. Anders gesagt: Nichtnegativitätsbedingungen können in (14.1) vorhanden sein; sie sind jedoch – wie alle anderen Nebenbedingungen – Teil der Forderung $Ax \leq b$.

Jedes LP-Problem kann auf einfache Weise in die Form (14.1) umgeschrieben werden. Ein **Beispiel** hierzu: Es liege das LP-Problem

$$\begin{aligned} &\text{maximiere } 3x_1 + 2x_2 + \ x_3 \\ &\text{unter den Nebenbedingungen} \\ &\qquad x_1 + 5x_2 + 2x_3 \ \leq 1 \\ &\qquad\quad x_2 - 3x_3 \ \leq 5 \\ &\qquad\quad\ \ x_1 \ \geq 0 \end{aligned}$$

vor. Dann kann die Nichtnegativitätsbedingung in die Form $-x_1 \leq 0$ gebracht werden. In der Darstellung (14.1) lautet dieses LP-Problem dann

$$\begin{aligned} &\text{maximiere } c^T x \\ &\text{unter den Nebenbedingungen} \\ &\qquad Ax \leq b, \end{aligned}$$

wobei $c^T = (3, 2, 1)$, $A = \begin{pmatrix} 1 & 5 & 2 \\ 0 & 1 & -3 \\ -1 & 0 & 0 \end{pmatrix}$ und $b = \begin{pmatrix} 1 \\ 5 \\ 0 \end{pmatrix}$ und gilt.

Liegt ein LP-Problem L in der Form (14.1) vor und sind alle Koeffizienten in A, b und c rational, so definieren wir die *Bitlänge von* L durch

$$\langle L \rangle = \langle A \rangle + \langle b \rangle + \langle c \rangle.$$

Statt „Bitlänge“ sagen wir auch *Eingabelänge* oder *Kodierungslänge*.

Wir betrachten auch im Folgenden nur LP-Probleme der Form (14.1), bei denen alle Einträge der Matrix A sowie der Vektoren b und c aus der Menge $\mathbb{Q}$ der rationalen Zahlen stammen, was vom Standpunkt der Praxis und der Laufzeitanalyse eine vernünftige Annahme ist. Wir kommen nun zur grundlegenden Definition.

Definition.

Ein Algorithmus wird *polynomieller Algorithmus für Lineare Programmierung* genannt, falls ein Polynom $p(x)$ existiert, für das gilt: Ist L ein LP-Problem der Form (14.1) mit rationalen Einträgen für A, b und c, so findet der Algorithmus eine optimale Lösung von L in höchstens $p(\langle L \rangle)$ Schritten bzw. liefert in höchstens $p(\langle L \rangle)$ Schritten eine der beiden Feststellungen „L ist unlösbar“ oder „L ist unbeschränkt“.

Dabei werden die Schritte in einem der üblichen Modelle gezählt, beispielsweise im Turingmaschinen-Modell oder im RAM-Modell von Shepherdson und Sturgis. Im vorliegenden Kontext spielt es keine Rolle, welches der üblichen Maschinenmodelle zugrunde liegt: Was beispielsweise im RAM-Modell mit einer polynomiellen Anzahl von Schritten durchführbar ist, ist im Turingmaschinen-Modell ebenfalls mit einer polynomiellen Anzahl von Schritten durchführbar; und umgekehrt. Die entsprechenden Polynome können jedoch – abhängig vom zugrundeliegenden Maschinenmodell – für ein und denselben Algorithmus höchst unterschiedlich ausfallen.

Auf eines sei ausdrücklich hingewiesen: Bei Operationen mit Zahlen werden die Schritte *bitweise* gezählt. Werden beispielsweise zwei Zahlen addiert, so wird dies nicht – wie in anderen Situationen üblich – als ein einzelner Schritt angesehen, sondern die Zahl der benötigten Schritte ist abhängig von der Bitlänge der beteiligten Zahlen. („Große Zahlen zu addieren dauert länger als die Addition kleiner Zahlen.“)

Wir kommen noch einmal kurz auf die *Klee-Minty-Beispiele* zurück: Um anhand dieser Beispiele zu erkennen, dass es sich beim Simplexverfahren (mit der Regel vom größten Koeffizienten) nicht um einen polynomiellen Algorithmus handelt, reicht es nicht aus festzustellen, dass für das Klee-Minty-Beispiel mit n Variablen $2^n - 1$ Iterationen nötig sind. Man hat sich außerdem um die Eingabelänge der Klee-Minty-Beispiele zu kümmern. Weshalb? Nun, es muss sichergestellt sein, dass die Eingabelänge dieser Beispiele nicht ebenfalls exponentiell in n wächst. *Dass dies nicht der Fall ist, erkennt man anhand der folgenden Feststellungen*: Ist L_n das Klee-Minty-Beispiel mit n Variablen (vgl. Abschnitt 5.1), so ist der größte Koeffizient, der in diesem Beispiel als Eintrag von A, b oder c auftritt, gleich 100^{n-1}. Obwohl 100^{n-1} auf den ersten Blick recht groß erscheint, ist die Bitlänge dieser Zahl gar nicht so „schrecklich“ groß, sondern nur von der Größenordnung $O(n)$. (Man beachte, dass $\log_2\left(100^{n-1}\right) = (n-1) \cdot \log_2(100)$ gilt.) Da in L_n insgesamt nur $O(n^2)$ Zahlen vorkommen, erhält man $\langle L_n \rangle = O(n^3)$, d.h., die Eingabelänge wächst (wie behauptet) keineswegs exponentiell in n.

14.3 Ellipsen im $\mathbb{R}^2$

Wir beginnen mit grundlegenden geometrischen Sachverhalten, wobei wir teilweise der Darstellung in W. Schäfer, K. Georgi, G. Trippler: *Mathematik-Vorkurs* (Teubner, 2006) folgen.

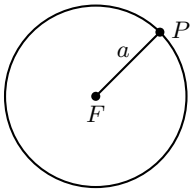
Sind zwei Punkte P und F des $\mathbb{R}^2$ gegeben, so bezeichnen wir mit $\overline{PF}$ den euklidischen Abstand dieser Punkte. Für $P = (x_1, y_1)$ und $F = (x_2, y_2)$ gilt demnach

$$\overline{PF} = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}.$$

Die Verbindungsstrecke zwischen P und F bezeichnen wir mit PF. Unter einem *Kreis mit Mittelpunkt* F versteht man bekanntlich die Menge aller Punkte $P = (x, y)$, für die der Abstand von F gleich einer positiven Konstanten ist. Bezeichnen wir diese Konstante mit a, so gilt demnach für alle Punkte P auf dem betrachteten Kreis:

$$\overline{PF} = a.$$

Dabei gilt $a > 0$ und, wie jeder weiß, nennt man a den *Radius* des Kreises; die Größe $2a$ ist der *Durchmesser*.



Während bei einem Kreis ein einziger Punkt F betrachtet wird, der Mittelpunkt des Kreises genannt wird, sind bei einer Ellipse zwei Punkte F_1 und F_2 gegeben, die *Brennpunkte* der Ellipse genannt werden. Bei einem Kreis wird verlangt, dass der Abstand von F konstant ist, während es bei einer Ellipse um die *Abstandssumme*

$$\overline{PF_1} + \overline{PF_2}$$

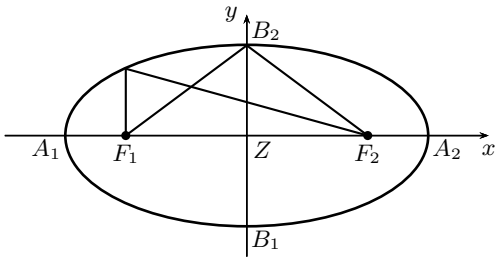
geht: *Bei einer Ellipse wird verlangt, dass die Abstandssumme* $\overline{PF_1} + \overline{PF_2}$ *konstant ist* und man bezeichnet diese konstante Abstandssumme mit $2a$; es soll also $\overline{PF_1} + \overline{PF_2} = 2a$ gelten. Ähnlich wie wir bei einem Kreis vorausgesetzt haben, dass $a > 0$ gilt, setzen wir hierbei $2a > \overline{F_1F_2}$ voraus; ebenso gut können wir hierfür natürlich auch $a > \frac{\overline{F_1F_2}}{2}$ schreiben.

In der nachfolgenden Definition wird das Gesagte zusammengefasst.

Definition.
Gegeben seien Punkte $F_1, F_2 \in \mathbb{R}^2$ sowie $a \in \mathbb{R}$ mit $a > \frac{\overline{F_1F_2}}{2}$. Unter einer *Ellipse mit den Brennpunkten* F_1 *und* F_2 versteht man die Menge aller Punkte $P = (x, y) \in \mathbb{R}^2$, für die gilt:

$$\overline{PF_1} + \overline{PF_2} = 2a.$$

Ist eine Ellipse mit den Brennpunkten F_1 und F_2 gegeben, so bezeichnen wir den Mittelpunkt der Strecke F_1F_2 als den *Mittelpunkt der Ellipse*. In der folgenden Zeichnung liegen die Brennpunkte auf der x-Achse und der Mittelpunkt Z der Ellipse ist gleich dem Ursprung des Koordinatensystems:



Wir betrachten zunächst nur Ellipsen, deren Brennpunkte (wie in der Zeichnung) auf der x-Achse symmetrisch zum Ursprung liegen.

Wie in der Zeichnung angegeben, seien A_1 und A_2 die Schnittpunkte der Ellipse mit der x-Achse; B_1 und B_2 seien die Schnittpunkte mit der y-Achse. Wir wollen die Größe a anhand der Zeichnung geometrisch interpretieren. Da der Punkt A_2 auf der Ellipse liegt, gilt

$$\overline{A_2F_1} + \overline{A_2F_2} = 2a. \tag{14.2}$$

Wegen der symmetrischen Lage der Ellipse gilt $\overline{A_1F_1} = \overline{A_2F_2}$; man erhält

$$\overline{A_1A_2} \;=\; \overline{A_2F_1} + \overline{A_1F_1} \;=\; \overline{A_2F_1} + \overline{A_2F_2} \;\overset{(14.2)}{=}\; 2a.$$

Damit haben wir die geometrische Interpretation der Größen $2a$ bzw. a gefunden: Es gilt $\overline{A_1A_2} = 2a$ und somit $a = \overline{A_1Z} = \overline{A_2Z}$.

Die Strecke A_1A_2 auf der x-Achse sowie die Strecke B_1B_2 auf der y-Achse nennt man die *Achsen der Ellipse*; entsprechend werden die Strecken A_1Z, A_2Z, B_1Z und B_2Z *Halbachsen der Ellipse* genannt. Wir halten fest:

a gibt die Länge der Halbachse A_1Z (bzw. A_2Z) an („lange Halbachse“).

Nun betrachten wir die andere Achse der Ellipse und setzen $b = \overline{B_1Z} = \overline{B_2Z}$. Somit gilt:

b gibt die Länge der Halbachse B_1Z (bzw. B_2Z) an („kurze Halbachse“).

Durch Betrachtung des Dreiecks B_2ZF_2 erkennt man übrigens, dass tatsächlich $a \geq b$ gilt. (Hinweis: Aus $\overline{B_2F_1} + \overline{B_2F_2} = 2a$ folgt $\overline{B_2F_2} = a$.) Die Namen „lange Halbachse“ und „kurze Halbachse“ sind also gerechtfertigt.

Wir betrachten nach wie vor eine Ellipse mit Mittelpunkt $Z = (0,0)$, deren Brennpunkte auf der x-Achse liegen; die Bezeichnungen seien wie bisher gewählt. Wir wollen eine solche Ellipse jetzt mithilfe der sogenannten *Mittelpunktsgleichung* beschreiben. Um zu erkennen, worum es dabei geht, orientieren wir uns am Beispiel eines Kreises. Ist ein Kreis mit Mittelpunkt $Z = (0,0)$ und Radius $a > 0$ gegeben, so liegt ein Punkt $P = (x,y)$ genau dann auf diesem Kreis, wenn der Abstand von P und Z gleich a ist, d.h., wenn

$$\sqrt{x^2 + y^2} = a$$

gilt. Durch Quadrieren und anschließendes Teilen durch a^2 geht diese Gleichung in die folgende äquivalente Form über:

$$\frac{x^2}{a^2} + \frac{y^2}{a^2} = 1. \tag{14.3}$$

Ein Punkt $P = (x,y)$ liegt also genau dann auf dem Kreis mit Mittelpunkt $Z = (0,0)$ und Radius a, wenn die Gleichung (14.3) erfüllt ist. *Wir behaupten nun, dass unsere Ellipse durch eine Gleichung beschrieben werden kann, die der Gleichung (14.3) sehr ähnlich ist.* Dies ist der Inhalt des folgenden Satzes.

Satz (Mittelpunktsgleichung der Ellipse mit Mittelpunkt $\mathbf{Z = (0,0)}$ und Brennpunkten auf der x-Achse).

Gegeben sei eine Ellipse mit Mittelpunkt $Z = (0,0)$, deren Brennpunkte auf der x-Achse liegen; die Bezeichnungen a und b für die Halbachsenlängen seien wie bisher gewählt. Dann gilt: Ein Punkt $P = (x,y)$ liegt genau dann auf der Ellipse, wenn die Gleichung

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1. \tag{14.4}$$

erfüllt ist.

Einen Beweis dieses Satzes findet man beispielsweise im zuvor genannten Buch von W. Schäfer et al.

Es sei darauf hingewiesen, dass unsere Definition des Begriffs *Ellipse* so abgefasst ist, dass Kreise spezielle Ellipsen sind: Ein Kreis liegt genau dann vor, wenn in der Definition einer Ellipse der spezielle Fall $F_1 = F_2$ vorliegt; in diesem Fall gilt $a = b$. (*Aufgabe*: Man überlege sich, weshalb in der Definition einer Ellipse die Voraussetzung $a > \frac{\overline{F_1F_2}}{2}$ gemacht wurde.)

Bislang haben wir Ellipsen mithilfe von Brennpunkten beschrieben. Für die Ellipsoid-Methode benötigen wir allerdings eine etwas andere Beschreibung, in der unter anderem *Matrizen* vorkommen. In dieser Beschreibung wird an die anschauliche Vorstellung angeknüpft, dass Ellipsen „deformierte Kreise“ sind; es werden also Streckungen und Stauchungen eine Rolle spielen. Außerdem sollen Ellipsen beispielsweise auch gedreht oder parallel verschoben werden. Um all dies zu präzisieren, benötigen wir den Begriff der *linearen Abbildung* und darüber hinaus auch den Begriff der *affinen Abbildung*.

14.4 Lineare und affine Abbildungen

Gegeben sei eine reelle $m \times n$ - Matrix A. Mithilfe von A definieren wir eine Abbildung

$$f : \mathbb{R}^n \to \mathbb{R}^m,$$

indem wir festlegen: $f(x) = Ax$ für alle $x \in \mathbb{R}^n$; hierbei ist $x = (x_1, \ldots, x_n)^T$ ein Spaltenvektor.

Da die obige Abbildung f mithilfe der Matrix A gebildet wurde, bezeichnen wir sie gelegentlich auch mit f_A. Um das Bild von x unter der Abbildung f_A zu bestimmen, hat man also nichts weiter zu tun, als das Produkt Ax zu bilden. Gilt beispielsweise

$$A = \begin{pmatrix} 7 & 5 & 2 \\ 3 & -1 & 3 \end{pmatrix},$$

so handelt es sich bei f_A um eine Abbildung vom Typ $\mathbb{R}^3 \to \mathbb{R}^2$ und das Bild eines Vektors $x = (x_1, x_2, x_3)^T$ ist gegeben durch

$$f_A(x) = \begin{pmatrix} 7x_1 + 5x_2 + 2x_3 \\ 3x_1 - \ x_2 + 3x_3 \end{pmatrix}.$$

Ist $f : \mathbb{R}^n \to \mathbb{R}^m$ eine Abbildung, die wie zuvor beschrieben von einer $m \times n$ - Matrix A abstammt (d.h. $f(x) = Ax$ für alle $x \in \mathbb{R}^n$), so nennt man f eine *lineare Abbildung* von $\mathbb{R}^n$ nach $\mathbb{R}^m$.

Zwei wichtige Rechenregeln, die für alle linearen Abbildungen gelten:

$$f(x + y) = f(x) + f(y) \tag{14.5}$$
$$f(c \cdot x) = c \cdot f(x) \qquad (\text{für } c \in \mathbb{R}). \tag{14.6}$$

Dass diese beiden Regeln tatsächlich für alle $x, y \in \mathbb{R}^n$ und alle $c \in \mathbb{R}$ gelten, ergibt sich unmittelbar aus entsprechenden Regeln der Matrizenrechnung: Wenn f eine lineare Abbildung ist, so gibt es eine Matrix A mit $f(x) = Ax$ für alle $x \in \mathbb{R}^n$; es folgt (wie behauptet):

$$\begin{aligned} f(x + y) = A(x + y) &= Ax + Ay = f(x) + f(y) \\ f(c \cdot x) = A(c \cdot x) &= c \cdot (Ax) = c \cdot f(x). \end{aligned}$$

Eine kurze *Nebenbemerkung* zur Rolle der Rechenregeln (14.5) und (14.6): Es lässt sich (unschwer) zeigen, dass es außer den Abbildungen, die wie beschrieben mithilfe von Matrizen A gebildet werden, keine weiteren Abbildungen $f : \mathbb{R}^n \to \mathbb{R}^m$ gibt, die die beiden Eigenschaften (14.5) und (14.6) besitzen. Aus diesem Grund hätten wir ebenso gut definieren können, dass eine Abbildung $f : \mathbb{R}^n \to \mathbb{R}^m$ *lineare Abbildung* genannt wird, falls (14.5) und (14.6) für alle $x, y \in \mathbb{R}^n$ und $c \in \mathbb{R}$ gelten. Diese Möglichkeit, den Begriff der linearen Abbildung zu definieren, werden Sie in den meisten Lehrbüchern der Linearen Algebra finden.

Nach dieser Nebenbemerkung wollen wir uns einige Beispiele für lineare Abbildungen $f : \mathbb{R}^2 \to \mathbb{R}^2$ anschauen. Wir beginnen mit einem besonders einfachen Beispiel.

Beispiel 1. Es sei $A = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ und $x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$. Dann gilt $Ax = x$, d.h., bei der dazugehörigen linearen Abbildung handelt es sich um die *Identität*.

In den folgenden Beispielen gelte, wenn nichts anderes gesagt ist, $x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \in \mathbb{R}^2$.

Beispiel 2. Es sei $A = \begin{pmatrix} a & 0 \\ 0 & a \end{pmatrix}$. Dann gilt $Ax = \begin{pmatrix} a & 0 \\ 0 & a \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} ax_1 \\ ax_2 \end{pmatrix} = ax$. Jeder Vektor x wird also auf sein a-faches abgebildet. Ist $a > 1$, so handelt es sich um eine *Streckung* mit dem Faktor a. (Man gebe ähnliche Beschreibungen für die Fälle $0 < a < 1$, $a = 0$ und $a < 0$.)

Beispiel 3. Es sei $A = \begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix}$ für $a > 0$ und $b > 0$. Dann gilt $Ax = \begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} ax_1 \\ bx_2 \end{pmatrix}$. Man erhält also das Bild eines Vektors $x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$, indem man die erste Komponente von x mit a und die zweite mit b multipliziert. Wir wollen uns anschauen, wohin bei dieser Abbildung die Einheitsvektoren

$$e_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad \text{und} \quad e_2 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

abgebildet werden. Es gilt

$$Ae_1 = A \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} a \\ 0 \end{pmatrix} \quad \text{und} \quad Ae_2 = A \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ b \end{pmatrix},$$

d.h. für $a > 1$ und $b > 1$, dass der erste Einheitsvektor e_1 um den Faktor a gestreckt wird, während e_2 um den Faktor b gestreckt wird. (Für $a \leq 1$ bzw. $b \leq 1$ gebe man ähnliche Interpretationen.)

Wir wollen uns zusätzlich anschauen, wohin bei der Abbildung aus Beispiel 3 der Einheitskreis abgebildet wird. Zur Erinnerung: Unter dem *Einheitskreis* versteht man die Menge aller $(x_1, x_2) \in \mathbb{R}^2$, für die gilt:

$$x_1^2 + x_2^2 = 1. \tag{14.7}$$

Behauptung.
Es gelte (wie in Beispiel 3 vorausgesetzt) $a > 0$, $b > 0$. Dann wird der Einheitskreis durch die lineare Abbildung

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \mapsto \begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} ax_1 \\ bx_2 \end{pmatrix}$$

auf die Ellipse abgebildet, die gegeben ist durch

$$\frac{x_1^2}{a^2} + \frac{x_2^2}{b^2} = 1. \tag{14.8}$$

Die Richtigkeit dieser Behauptung ist leicht ersichtlich: Der Punkt (x_1, x_2) liegt genau dann auf dem Einheitskreis, wenn $x_1^2 + x_2^2 = 1$ gilt; der Punkt (ax_1, bx_2) liegt genau dann auf der Ellipse, die durch (14.8) gegeben ist, wenn

$$\frac{(ax_1)^2}{a^2} + \frac{(bx_2)^2}{b^2} = 1$$

gilt. Hieraus ergibt sich die Behauptung, da offenbar Folgendes gilt:

$$x_1^2 + x_2^2 = 1 \quad \Longleftrightarrow \quad \frac{(ax_1)^2}{a^2} + \frac{(bx_2)^2}{b^2} = 1.$$

Bislang hatten wir uns vornehmlich auf Ellipsen konzentriert, deren Brennpunkte symmetrisch zum Ursprung auf der x_1-Achse liegen: Ist eine derartige Ellipse gegeben, so haben wir weiter oben festgestellt, dass $a \geq b$ gilt. Dabei ist der Fall $a = b$ der Spezialfall, dass ein Kreis vorliegt.

Frage: Welcher Typ von Ellipse wird in (14.8) beschrieben, wenn $b > a$ gilt?

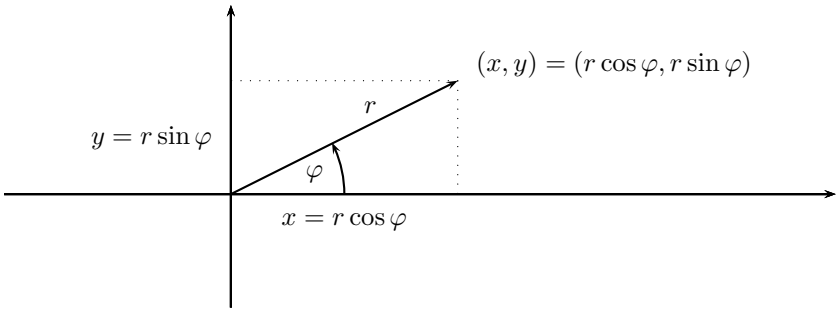
Antwort: In diesem Fall liegt eine Ellipse vor, deren Brennpunkte symmetrisch zum Ursprung auf der x_2-Achse liegen. In diesem Fall ist der Mittelpunkt der Ellipse nach wie vor $Z = (0, 0)$, *die lange Achse und die Brennpunkte liegen jedoch auf der x_2-Achse*. Man kann die Sache auch so sehen: Eine Ellipse mit Brennpunkten auf der x_1-Achse wurde um 90° gedreht, wodurch eine Ellipse entstanden ist, deren Brennpunkte auf der x_2-Achse liegen.

Natürlich gibt es auch Ellipsen, deren Brennpunkte weder auf der x_1- noch auf der x_2-Achse liegen. Diese Ellipsen lassen sich mithilfe von *Drehungen* und *Translationen* (Parallelverschiebungen) beschreiben. Dabei kann man sich auf Drehungen beschränken, deren Drehzentrum der Ursprung ist. Derartige Drehungen werden im nachfolgenden Beispiel behandelt.

Beispiel 4. Es sei $A = \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix}$ für $0 \leq \alpha \leq 2\pi$. Es handelt sich um eine *Drehung* um den Winkel α (entgegen dem Uhrzeigersinn und mit dem Ursprung als Drehzentrum).

Um dies einzusehen, verwenden wir Polarkoordinaten: Für $(x, y) \neq (0, 0)$ gelte (siehe Skizze)

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} r\cos\varphi \\ r\sin\varphi \end{pmatrix}.$$



Es folgt

$$\begin{aligned} A\begin{pmatrix} r\cos\varphi \\ r\sin\varphi \end{pmatrix} &= \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix}\begin{pmatrix} r\cos\varphi \\ r\sin\varphi \end{pmatrix} \\ &= \begin{pmatrix} r(\cos\alpha\cdot\cos\varphi - \sin\alpha\cdot\sin\varphi) \\ r(\sin\alpha\cdot\cos\varphi + \cos\alpha\cdot\sin\varphi) \end{pmatrix} \\ &\overset{(\star)}{=} \begin{pmatrix} r\cos(\varphi+\alpha) \\ r\sin(\varphi+\alpha) \end{pmatrix}. \end{aligned}$$

A bewirkt also eine Drehung der beschriebenen Art um den Winkel α.

Die Gleichheit $(\star)$ gilt aufgrund der bekannten *Additionstheoreme* für sin und cos, welche besagen, dass für alle $x_1, x_2 \in \mathbb{R}$ gilt:

$$\begin{aligned} \sin(x_1 + x_2) &= \sin x_1 \cos x_2 + \cos x_1 \sin x_2 \\ \cos(x_1 + x_2) &= \cos x_1 \cos x_2 - \sin x_1 \sin x_2. \end{aligned}$$

Durch die Betrachtung von linearen Abbildungen haben wir eine neue Möglichkeit gewonnen, Ellipsen mit Mittelpunkt $Z = (0, 0)$ darzustellen: *Jede Ellipse mit Mittelpunkt $Z = (0, 0)$ lässt sich aus dem Einheitskreis gewinnen, indem man zwei lineare Abbildungen nacheinander ausführt.*

Um welche Typen von linearen Abbildungen es sich dabei handelt, wird im Folgenden beschrieben:

- Die erste dieser beiden linearen Abbildungen ist eine Abbildung vom Typ

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \mapsto \begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} ax_1 \\ bx_2 \end{pmatrix},$$

  wobei $a \geq b > 0$ gilt. Durch diese lineare Abbildung wird – wie wir gesehen haben – der Einheitskreis zu einer Ellipse „deformiert“, deren Brennpunkte symmetrisch zum Ursprung auf der x_1-Achse liegen; a ist die Länge der langen Halbachse dieser Ellipse; b ist die Länge der kurzen Halbachse.
- Durch die zweite dieser linearen Abbildungen wird die Ellipse anschließend gedreht, wobei der Ursprung $Z = (0, 0)$ das Drehzentrum ist. Ist α der Drehwinkel, so lautet die dazugehörige Matrix

$$A = \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix}.$$

Die Matrix der ersten linearen Abbildung wollen wir mit B bezeichnen, d.h.

$$B = \begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix}.$$

Führt man die beiden linearen Abbildungen nacheinander aus („erst B, dann A“), so geht jeder Vektor $\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \in \mathbb{R}^2$ über in den Vektor

$$A\left(B\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}\right).$$

Handelt es sich bei dieser Nacheinanderausführung wiederum um eine lineare Abbildung?

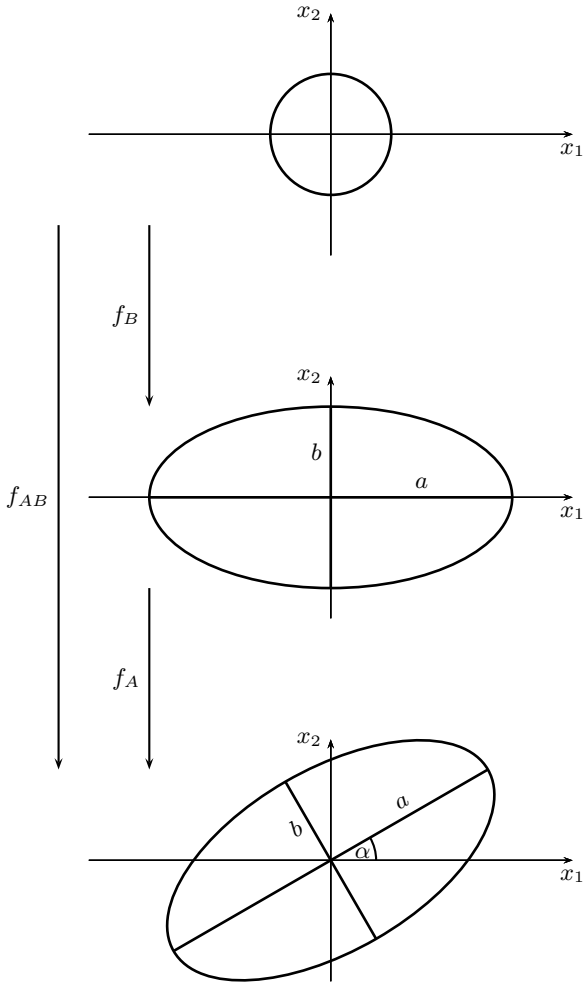
Die Antwort lautet ja, wie man anhand der folgenden Gleichung sofort erkennt:

$$A\left(B\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}\right) = (AB)\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}. \tag{14.9}$$

Die Gleichung (14.9) beruht auf dem Assoziativgesetz für die Matrizenmultiplikation. Man erkennt anhand von (14.9), welche Matrix zur Nacheinanderausführung der beiden linearen Abbildungen gehört: *Die Nacheinanderausführung wird durch die Produktmatrix AB beschrieben.* Diese lautet wie folgt:

$$AB = \begin{pmatrix} a\cos\alpha & -b\sin\alpha \\ a\sin\alpha & b\cos\alpha \end{pmatrix}. \tag{14.10}$$

Wir illustrieren das Gesagte noch einmal anhand einer Skizze:



Es sei darauf hingewiesen, dass die Determinante der Matrix AB aus (14.10) ungleich 0 ist. Dies ergibt sich wie folgt:

$$\begin{aligned}\det(AB) &= ab\cos^2\alpha + ab\sin^2\alpha \\ &= ab\left(\cos^2\alpha + \sin^2\alpha\right) \\ &= ab \;\neq\; 0.\end{aligned}$$

Wir setzen $M = AB$. Es gilt also $\det M \neq 0$, was bedeutet, dass es sich bei M um eine reguläre Matrix handelt[1]. Das bedeutet insbesondere, dass die Zeilen von M linear unabhängig sind – ebenso wie die Spalten – und dass M^{-1} existiert. Außerdem gilt aufgrund der Regularität von M, dass die zu M gehörige lineare Abbildung f_M bijektiv ist. (Die Bijektivität von f_M ergibt sich unmittelbar aus dem ersten Eintrag der linken Spalte auf Seite 106.)

Wir können das Ergebnis unserer bisherigen Überlegungen wie folgt aussprechen.

Feststellung 1.

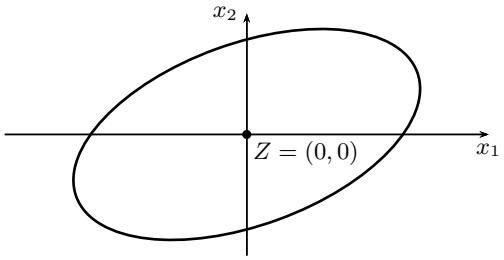
Ist $\mathcal{E}$ eine Ellipse mit Mittelpunkt $Z = (0,0)$, so gibt es eine reguläre Matrix M, so dass $\mathcal{E}$ das Bild des Einheitskreises unter der zu M gehörigen linearen Abbildung f_M ist.

Zu Feststellung 1 gibt es eine Umkehrung, die sich mit Mitteln der Linearen Algebra (Stichwort: Klassifikation der Quadriken im $\mathbb{R}^2$) beweisen lässt. Wir geben diese Umkehrung ohne Beweis als Feststellung 2 an.

[1] „Regulär“ bedeutet dasselbe wie „nichtsingulär“; zum Unterschied zwischen singulären und nichtsingulären Matrizen, siehe auch Abschnitt 8.5.

Feststellung 2.
Ist M eine beliebige reguläre 2×2 - Matrix und $f_M : \mathbb{R}^2 \to \mathbb{R}^2$ die zu M gehörige lineare Abbildung, so ist das Bild $f_M(C)$ des Einheitskreises C immer eine Ellipse mit Mittelpunkt $Z = (0, 0)$.

Bislang haben wir ausschließlich Ellipsen behandelt, für die $Z = (0, 0)$ galt, d.h., deren Mittelpunkt der Koordinatenursprung ist – wie beispielsweise in der folgenden Zeichnung.



Nun sollen die übrigen Ellipsen in die Betrachtung einbezogen werden. Das ist sehr einfach: Durch eine Translation, die sich an die lineare Abbildung anschließt, kann der Ellipsenmittelpunkt an jeden beliebigen Ort verlegt werden. Statt „Translation“ kann man auch „Parallelverschiebung“ sagen. Hier ist die genaue Definition.

Definition.
Ist $s = \begin{pmatrix} s_1 \\ s_2 \end{pmatrix} \in \mathbb{R}^2$ ein fest gewählter Vektor, so wird unter der *Translation* T mit Translationsvektor s die folgende Abbildung verstanden:

$$T : \mathbb{R}^2 \longrightarrow \mathbb{R}^2$$
$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \longmapsto \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} s_1 \\ s_2 \end{pmatrix}$$

Bei einer Translation wird also zu jedem $x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$ ein fester Vektor $s = \begin{pmatrix} s_1 \\ s_2 \end{pmatrix}$ addiert. Eine gegebene Ellipse wird dadurch parallel verschoben. Dabei ist auch der Fall, dass s der Nullvektor ist, mit einbezogen; in diesem Fall ist T nichts anderes als die Identität.

Definition.
Unter einer *affinen Abbildung* $\mathbb{R}^2 \to \mathbb{R}^2$ versteht man eine Abbildung vom Typ

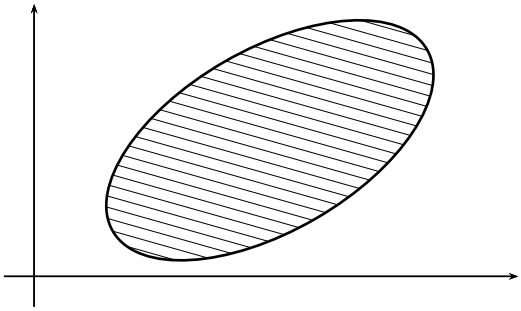
$$x \longmapsto Mx + s, \tag{14.11}$$

wobei M eine 2×2 - Matrix ist und $s \in \mathbb{R}^2$ ein fester Vektor.

Mit anderen Worten: *Eine affine Abbildung ist die Komposition einer linearen Abbildung mit einer anschließenden Translation.* (Beachte: Jede lineare Abbildung ist eine spezielle affine Abbildung, da für s auch der Nullvektor zugelassen ist.)

Aufgrund der vorangegangenen Ausführungen können wir feststellen: *Ellipsen im $\mathbb{R}^2$ sind nichts anderes als die Bilder des Einheitskreises unter einer affinen Abbildung (14.11) mit einer regulären Matrix M.*

Unter einem *Ellipsoid* im $\mathbb{R}^2$ versteht man eine Ellipse zusammen mit ihrem Inneren (siehe Zeichnung).



Unter einer affinen Abbildung (14.11) mit regulärer Matrix M geht das Innere des Einheitskreises in das Innere einer Ellipse über[2]. Mit B^2 wollen wir den Einheitskreis zusammen mit seinem Inneren bezeichnen, d.h.[3]

$$B^2 = \Big\{x \in \mathbb{R}^2 : x^T x \leq 1\Big\}. \tag{14.12}$$

Nach dem zuvor Gesagten ist ein Ellipsoid im $\mathbb{R}^2$ eine Menge der Form

$$E = \Big\{Mx + s : x \in B^2\Big\}, \tag{14.13}$$

wobei M eine reguläre 2×2 - Matrix und $s \in \mathbb{R}^2$ ein Vektor ist.

14.5 Ellipsoide im $\mathbb{R}^n$

Die Einheitskreisscheibe (mit Rand) haben wir in (14.12) mit B^2 bezeichnet und Ellipsoide im $\mathbb{R}^2$ haben wir in (14.13) beschrieben als Bilder der Einheitskreisscheibe B^2 unter affinen Abbildungen $x \mapsto Mx + s$ mit regulärer 2×2 - Matrix M. *Damit haben wir Ellipsoide in einer Form dargestellt, die für die Beschreibung der Ellipsoid-Methode günstig ist.* Wir folgen nun der Darstellung des Lehrbuchs

- J. Matoušek, B. Gärtner: *Understanding and Using Linear Programming*. Springer-Verlag. Berlin Heidelberg (2007)

und geben zum Einstieg die dort verwendete *Definition eines Ellipsoids im* $\mathbb{R}^n$ wieder[4]:

Definition (Ellipsoid).
A two-dimensional ellipsoid is an ellipse plus its interior. An ellipsoid in general can most naturally be introduced as an affine transformation of a ball. We let

$$B^n = \Big\{x \in \mathbb{R}^n : x^T x \leq 1\Big\}$$

be the n-dimensional ball of unit radius centered at 0. Then an n-dimensional *ellipsoid* is a set of the form

$$E = \Big\{Mx + s : x \in B^n\Big\},$$

where M is a nonsingular $n \times n$ matrix and $s \in \mathbb{R}^n$ is a vector. The mapping $x \mapsto Mx + s$ is a composition of a linear function and a translation; this is called an *affine map*.

Unter einem *n-dimensionalen Ellipsoid* verstehen wir also eine Menge der Form

$$E = \Big\{Mx + s : x \in B^n\Big\}, \tag{14.14}$$

[2] Auf den (nicht schwierigen) Beweis dieser anschaulich einleuchtenden Feststellung wollen wir verzichten.

[3] Man beachte, dass für $x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$ gilt: $x^T x = x_1^2 + x_2^2$.

[4] Zur Erinnerung: „Reguläre $n \times n$ - Matrix“ und „nichtsinguläre $n \times n$ - Matrix“ bedeutet dasselbe. Eine $n \times n$ - Matrix M ist genau dann nichtsingulär, wenn die inverse Matrix M^{-1} existiert.

wobei M eine nichtsinguläre $n \times n$ - Matrix und $s \in \mathbb{R}^n$ ein Vektor ist. Anders gesagt: *Ein n-dimensionales Ellipsoid ist das Bild der n-dimensionalen Kugel $B^n = \{x \in \mathbb{R}^n : x^T x \leq 1\}$ unter einer affinen Abbildung $x \mapsto Mx + s$, wobei M nichtsingulär ist.* Man nennt B^n die *n-dimensionale Einheitskugel*.

Damit die Beschreibung der Menge E genau den Anforderungen der Ellipsoid-Methode entspricht, wollen wir sie noch etwas umformen. Dabei wird die Matrix MM^T ins Spiel kommen; wir wollen diese Matrix mit Q bezeichnen, d.h., wir definieren (für M wie zuvor)

$$Q = MM^T. \tag{14.15}$$

Da M nichtsingulär ist, sind auch M^T und Q nichtsingulär, d.h., dass neben M^{-1} auch $\left(M^T\right)^{-1}$ und Q^{-1} existieren. Aufgrund bekannter Rechenregeln für Matrizen gilt

$$Q^{-1} = \left(MM^T\right)^{-1} = \left(M^T\right)^{-1} M^{-1} = \left(M^{-1}\right)^T M^{-1}. \tag{14.16}$$

Ist $x \in \mathbb{R}^n$ gegeben, so bezeichnen wir das Bild von x unter der affinen Abbildung $x \mapsto Mx + s$ mit y, d.h., wir setzen

$$y = Mx + s. \tag{14.17}$$

Auflösung der Gleichung (14.17) nach x ergibt

$$x = M^{-1}(y - s). \tag{14.18}$$

Unter Verwendung von (14.18) erhält man

$$y \in E \quad \Leftrightarrow \quad x \in B^n \quad \Leftrightarrow \quad x^T x \leq 1 \quad \Leftrightarrow \quad \left(M^{-1}(y-s)\right)^T M^{-1}(y-s) \leq 1.$$

Den rechts stehenden, etwas „wild“ aussehenden Ausdruck $\left(M^{-1}(y-s)\right)^T M^{-1}(y-s)$ wollen wir noch etwas vereinfachen; dafür haben wir in (14.16) bereits vorgearbeitet. Es gilt

$$\begin{aligned} \left(M^{-1}(y-s)\right)^T M^{-1}(y-s) &= (y-s)^T \left(M^{-1}\right)^T M^{-1}(y-s) \\ &\overset{(14.16)}{=} (y-s)^T Q^{-1}(y-s) \end{aligned}$$

Insgesamt können wir also feststellen:

$$y \in E \quad \Leftrightarrow \quad (y-s)^T Q^{-1}(y-s) \leq 1. \tag{14.19}$$

Damit haben wir die angekündigte Beschreibung des Ellipsoids E erhalten, die genau den Anforderungen der Ellipsoid-Methode entspricht: Bislang haben wir ein Ellipsoid E in der Form $E = \{Mx+s : x \in B^n\}$ beschrieben; ab jetzt werden wir meist die folgende Darstellung verwenden, die aufgrund von (14.19) gleichwertig ist:

$$E = \left\{y \in \mathbb{R}^n : (y-s)^T Q^{-1}(y-s) \leq 1\right\}. \tag{14.20}$$

Ein Wort zur Matrix Q: Es ist nicht sonderlich schwierig, sich zu überlegen, dass $Q = MM^T$ eine symmetrische, positiv definite Matrix ist[5]. Mit etwas fortgeschritteneren Hilfsmitteln der Linearen Algebra (Stichwort: Hauptachsentransformation) lässt sich zeigen, dass es auch umgekehrt zu jeder symmetrischen, positiv definiten Matrix Q eine nichtsinguläre Matrix M gibt, so dass $Q = MM^T$ gilt. *Das bedeutet*: Sind eine symmetrische, positiv definite Matrix Q und ein Vektor s gegeben, so wird durch (14.20) immer ein Ellipsoid beschrieben. Wir nennen dieses Ellipsoid das *durch Q und s erzeugte Ellipsoid*.

[5] Q ist *symmetrisch* bedeutet, dass $Q = Q^T$ gilt; Q ist *positiv definit* bedeutet, dass $x^T Q x > 0$ für alle $x \in \mathbb{R}^n$ mit $x \neq 0$ gilt.

Geometrisch handelt es sich bei s um den *Mittelpunkt* des Ellipsoids (14.14) bzw. (14.20).

Abschließende Bemerkungen: Durch die Formeln (14.14) und (14.20) werden Ellipsoide im $\mathbb{R}^n$ beschrieben. Es leuchtet ein, dass derartige Formeln nützlich und wichtig sind – darüber hinaus ist es aber auch wichtig, sich Ellipsoide im $\mathbb{R}^2$ und $\mathbb{R}^3$ konkret vorstellen zu können. Ein Ellipsoid im $\mathbb{R}^2$ ist eine Ellipse zusammen mit ihrem Inneren – das hatten wir ja bereits besprochen. Sich ein Ellipsoid im $\mathbb{R}^3$ vorzustellen ist ebenfalls nicht schwierig. Tipp: Denken Sie an einen *Rugbyball*. Oder geben Sie das Stichwort *Ellipsoid* bei Google ein: Sie werden haufenweise schöne Bilder von Ellipsoiden im $\mathbb{R}^3$ geliefert bekommen. Und Formeln werden Sie dort ebenfalls finden: Dabei ist zu beachten, dass die Matrix, die wir Q^{-1} genannt haben, häufig auch mit Q bezeichnet wird.

14.6 Eine Reduktion

Gegeben sei eine $m \times n$ - Matrix A und ein Vektor b der Länge m; wir betrachten das folgende System von Ungleichungen:

$$Ax \leq b. \tag{14.21}$$

Bei der Ellipsoid-Methode handelt es sich um ein Verfahren, das für jedes derartige System von Ungleichungen feststellt, ob eine Lösung existiert, und das – falls vorhanden – eine Lösung x liefert.

Frage: Wenn die Ellipsoid-Methode „nur“ das eben Gesagte leistet, wie kann man sie dann einsetzen, um jedes LP-Problem zu lösen?

Im Folgenden wird beschrieben, wie dies geht. Hierzu betrachten wir ein beliebiges LP-Problem in Standardform:

$$\begin{aligned} &\text{maximiere } c^T x \\ &\text{unter den Nebenbedingungen} \\ &\qquad Bx \leq d \\ &\qquad x \geq 0. \end{aligned} \tag{P}$$

Das zu (P) duale Problem lautet

$$\begin{aligned} &\text{minimiere } \ d^T y \\ &\text{unter den Nebenbedingungen} \\ &\qquad B^T y \geq c \\ &\qquad y \geq 0. \end{aligned} \tag{D}$$

Aus (P) und (D) formen wir nun ein Ungleichungssystem:

$$\begin{aligned} Bx &\leq d \\ -x &\leq 0 \\ -B^T y &\leq -c \\ -y &\leq 0 \\ d^T y - c^T x &\leq 0. \end{aligned} \tag{14.22}$$

Das Ungleichungssystem (14.22) besitzt also genau $m + n$ Variablen: Die Variablen von (14.22) sind $x_1, \ldots, x_n$ und $y_1, \ldots, y_m$. Man beachte, dass $-x \leq 0$ äquivalent zu $x \geq 0$ ist; außerdem ist $-B^T y \leq -c$ äquivalent zu $B^T y \geq c$ und $-y \leq 0$ ist äquivalent zu $y \geq 0$. Besonders wichtige Beobachtung: Die letzte Zeile von (14.22) ist äquivalent zu $d^T y \leq c^T x$. Demnach gilt (vgl. (7.3) und den anschließenden Kommentar):

(i) Bilden x und y eine Lösung des Ungleichungssystems (14.22), so ist x eine optimale Lösung von (P) und y ist eine optimale Lösung von (D) (und es gilt $d^T y = c^T x$).

Außerdem gilt aufgrund des *Dualitätssatzes*:

(ii) Ist (14.22) unlösbar, so besitzt (P) keine optimale Lösung.

Begründung zu (ii): Ist (14.22) unlösbar, so kann (P) keine optimale Lösung besitzen, denn: Würde (P) eine optimale Lösung x besitzen, so würde (nach dem Dualitätssatz) auch (D) eine optimale Lösung y besitzen und die Zielfunktionswerte würden übereinstimmen, d.h., es würde $d^T y = c^T x$ gelten. Dann würden x und y aber eine Lösung von (14.22) bilden – im Widerspruch zur Annahme, dass (14.22) unlösbar ist.

Aufgrund von (i) und (ii) ist die Frage geklärt, die wir oben im Anschluss an (14.21) gestellt haben.

Besitzt man einen Algorithmus $\mathcal{A}$, der für jedes Ungleichungssystem (14.21) herausfindet, ob es lösbar ist, und gegebenenfalls eine Lösung liefert, so kann man diesen Algorithmus auch zur Lösung von (P) verwenden: *Man hat aufgrund von (i) und (ii) nichts weiter zu tun, als $\mathcal{A}$ auf das Ungleichungssystem (14.22) anzuwenden.*

Für Abschnitt 14.6 wurde die Überschrift „Eine Reduktion“ gewählt. **Erklärung**: Aufgrund der Ergebnisse dieses Abschnitts lässt sich die Aufgabe, einen Algorithmus zu entwerfen, der jedes LP-Problem (P) löst, auf die folgende Aufgabe zurückführen (Statt „zurückführen“ sagt man auch „reduzieren“!): *Finde einen Algorithmus, der für jedes Ungleichungssystem $Ax \leq b$ entscheidet, ob eine Lösung existiert, und der ggf. auch eine Lösung x liefert.* Es handelt sich also um eine *Reduktion* einer gegebenen Aufgabenstellung auf eine andere Aufgabenstellung. Da man klarerweise die Erzeugung von (14.22) aus (P) in polynomieller Zeit durchführen kann, spricht man auch von einer *polynomiellen Reduktion*.

14.7 Die Ellipsoid-Methode

Es kann hier nur darum gehen, die *Grundideen der Ellipsoid-Methode* kennenzulernen – die Details können schon allein aus Zeitgründen nicht behandelt werden.

Zur Erinnerung: Sind eine $m \times n$ - Matrix A und eine dazugehörige rechte Seite b gegeben (beide mit Einträgen aus $\mathbb{Q}$), so geht es darum festzustellen, ob

$$Ax \leq b \tag{14.23}$$

eine Lösung besitzt; außerdem soll, falls die Lösungsmenge nicht leer ist, auch tatsächlich eine Lösung x ermittelt werden.

Die Lösungsmenge von (14.23) sei mit P bezeichnet; der Buchstabe P weist darauf hin, dass es sich bei der Lösungsmenge von (14.23) um ein *Polyeder* handelt (vgl. Kapitel 4).

Für $x = (x_1, \ldots, x_n) \in \mathbb{R}^n$ bezeichnen wir mit $|x|$ die *Länge* von x, d.h.

$$|x| = \sqrt{\sum_{i=1}^{n} x_i^2}.$$

Für zwei Punkte $x = (x_1, \ldots, x_n)$ und $z = (z_1, \ldots, z_n)$ des $\mathbb{R}^n$ ist der *Abstand* von x und z gegeben durch

$$|x - z| = \sqrt{\sum_{i=1}^{n} (x_i - z_i)^2}.$$

Sind $r > 0$ und $z \in \mathbb{R}^n$ gegeben, so bezeichnen wir mit $B(z, r)$ die *n-dimensionale Kugel mit Mittelpunkt z und Radius r*, d.h.

$$B(z, r) = \Big\{ x \in \mathbb{R}^n : |x - z| \leq r \Big\}.$$

Wir werden im Folgenden den besonders wichtigen Fall diskutieren, dass neben der Matrix A und dem Vektor b außerdem noch zwei rationale Zahlen R und ε gegeben sind, für die wir annehmen, dass

$$0 < \varepsilon < R$$

gilt. Die *Rolle von R* ist einfach zu beschreiben: Da es uns darum geht, die Grundidee der Ellipsoid-Methode zu erfassen, betrachten wir hier ausschließlich den besonders wichtigen Fall, dass P *beschränkt* ist. Genauer: *Wir nehmen an, dass uns bekannt ist, dass die Menge P aller Lösungen von $Ax \leq b$ in der Kugel $B(0, R)$ mit Mittelpunkt 0 und Radius R enthalten ist.*

Die *Rolle von ε* ist etwas diffiziler. *Es geht darum, die Anforderungen an unseren Algorithmus etwas abzuschwächen*:

- Falls P eine Kugel vom Radius ε (mit beliebigem Mittelpunkt) enthält, so wollen wir weiterhin „streng“ sein, d.h., wir wollen weiterhin verlangen, dass der Algorithmus uns ein Element $x \in P$ liefert.
- Falls sich im Laufe des Verfahrens jedoch herausstellt, dass P keine Kugel vom Radius ε enthält, so soll der Algorithmus entweder ein korrektes $x \in P$ abliefern oder aber die Antwort NO SOLUTION geben. Dabei ist es auch erlaubt, dass die Antwort NO SOLUTION lautet, obwohl $P \neq \emptyset$ gilt.

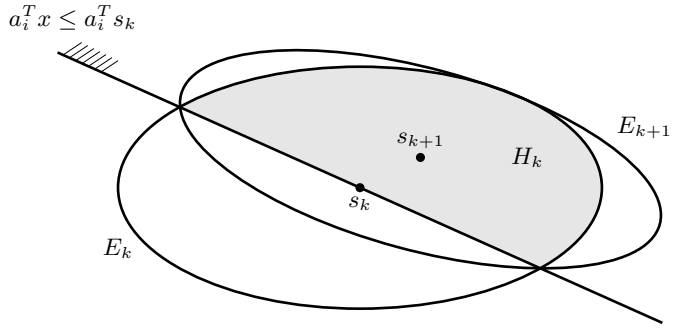
Vom Algorithmus wird demnach eine korrekte Antwort erwartet, wenn P an irgendeiner Stelle „hinreichend dick“ ist, d.h., wenn P irgendwo eine Kugel vom Radius ε enthält. Ist dies nicht der Fall, so ist es dem Algorithmus erlaubt, mit der Meldung NO SOLUTION abzubrechen.

Unter den genannten Annahmen liefert die *Ellipsoid-Methode* auf folgende Art eine Folge $E_0, \ldots, E_t$ von Ellipsoiden, die alle die Menge P umfassen, für die also $P \subseteq E_k$ $(k = 0, \ldots, t)$ gilt:

1. (**Initialisierung**): Setze $k = 0$ und $E_0 = B(0, R)$.

2. Für das aktuelle Ellipsoid E_k gelte $E_k = \left\{y \in \mathbb{R}^n : (y - s_k)^T Q_k^{-1} (y - s_k) \leq 1\right\}$. Falls der Mittelpunkt s_k von E_k sämtliche Ungleichungen des Systems $Ax \leq b$ erfüllt: Ausgabe von s_k und **stop**.

3. Andernfalls wähle man eine Ungleichung des Systems $Ax \leq b$, die von s_k nicht erfüllt wird; dies sei, sagen wir, die i-te Ungleichung. Bezeichnen wir die i-te Zeile von A mit a_i^T, so gilt also $a_i^T s_k > b_i$. Wir definieren E_{k+1} als das Ellipsoid mit kleinstmöglichem Volumen, das die Menge $H_k = E_k \cap \left\{x \in \mathbb{R}^n : a_i^T x \leq a_i^T s_k\right\}$ umfasst. Wir können uns die Menge H_k als ein „Halbellipsoid“ vorstellen (siehe Zeichnung):



4. Falls das Volumen des Ellipsoids E_{k+1} kleiner ist als das Volumen einer n-dimensionalen Kugel vom Radius ε, so lautet das Ergebnis NO SOLUTION; **stop**. Andernfalls erhöht man k um 1 und fährt bei **2.** fort.

Einige *Bemerkungen* zu diesem Algorithmus:

a) Es ist nicht schwer, das in **1.** auftretende Ellipsoid $E_0 = B(0,R)$ in der Form $E_0 = \left\{y \in \mathbb{R}^n : (y-s_0)^T Q_0^{-1}(y-s_0) \leq 1\right\}$ anzugeben. Hierzu hat man lediglich s_0 als den Ursprung des $\mathbb{R}^n$ zu wählen, also $s_0 = 0$, und ferner daran zu denken, dass $B(0,R) = \left\{y \in \mathbb{R}^n : |y| \leq R\right\} = \left\{y \in \mathbb{R}^n : y^T y \leq R^2\right\}$ gilt. *Aufgabe*: Überlegen Sie sich, wie man Q_0 zu wählen hat, damit

$$E_0 = \left\{y \in \mathbb{R}^n : y^T Q_0^{-1} y \leq 1\right\}$$

gilt.

b) Es lässt sich beweisen, dass das Ellipsoid E_{k+1}, d.h. das Ellipsoid mit kleinstmöglichem Volumen, das die Menge H_k umfasst, immer eindeutig bestimmt ist. *Genauer*: Es gilt $E_{k+1} = \left\{y \in \mathbb{R}^n : (y-s_{k+1})^T Q_{k+1}^{-1}(y-s_{k+1})\right\}$, wobei sich s_{k+1} und Q_{k+1} mithilfe der folgenden *Update-Formeln* aus s_k und Q_k berechnen lassen:

$$s_{k+1} = s_k - \frac{1}{n+1} \cdot \frac{Q_k a_i}{\sqrt{a_i^T Q_k a_i}}$$
$$Q_{k+1} = \frac{n^2}{n^2-1}\left(Q_k - \frac{2}{n+1} \cdot \frac{Q_k a_i a_i^T Q_k^T}{a_i^T Q_k a_i}\right).$$

Um diesen Formeln eine etwas übersichtlichere Gestalt zu geben, setzen wir

$$v = \frac{Q_k a_i}{\sqrt{a_i^T Q_k a_i}}.$$

Dann gilt

$$s_{k+1} = s_k - \frac{1}{n+1} v$$
$$Q_{k+1} = \frac{n^2}{n^2-1}\left(Q_k - \frac{2}{n+1} \cdot v v^T\right).$$

Beweise für die in b) aufgeführten Tatsachen findet man beispielsweise in

- D. Bertsimas, J. N. Tsitsiklis: *Introduction to Linear Optimization.* Athena Scientific (1997).[6]

Die nächste Bemerkung c) spielt für die Korrektheit und Effizienz der Ellipsoid-Methode eine besonders wichtige Rolle. In dieser Bemerkung wird es um das Volumen der Ellipsoide E_k und E_{k+1} gehen; auch das Volumen der n-dimensionalen Kugel $B(0,R)$ bzw. einer n-dimensionalen Kugel vom Radius ε wird eine Rolle spielen. Wir wollen nicht ausführen, wie all diese Volumina definiert sind, und auch keine Formeln oder Abschätzungen für die Volumina geben, sondern wir konzentrieren uns hier allein auf die für unsere Zwecke relevanten Fakten, wobei wir auf Beweise verzichten. Wer an weiteren Details interessiert ist, findet diese im oben genannten Buch von Bertsimas und Tsitsiklis, im Skript von M. Grötschel (siehe Literaturverzeichnis) oder auch in

- M. Grötschel, L. Lovász, A. Schrijver: *Geometric Algorithms and Combinational Optimization.* Springer (1994).

c) Die Volumina von E_k und E_{k+1} seien mit $\mathrm{vol}\,(E_k)$ bzw. $\mathrm{vol}\,(E_{k+1})$ bezeichnet.

Dann gilt

$$\mathrm{vol}\,(E_{k+1}) \leq e^{-\frac{1}{2n+2}} \cdot \mathrm{vol}\,(E_k).$$

[6] Man kann die im Buch von Bertsimas und Tsitsiklis gegebenen Beweise nicht 1–1 übernehmen; die ein oder andere Anpassung ist nötig, beispielsweise weil Bertsimas und Tsitsiklis andere Bezeichnungen benutzen und meistens Minimierungsprobleme betrachten.

Mit jeder Iteration verkleinert sich das Volumen des jeweiligen Ellipsoids also um den Faktor $e^{-\frac{1}{2n+2}}$. Folglich gilt

$$\operatorname{vol}(E_k) \leq e^{-\frac{k}{2n+2}} \cdot \operatorname{vol}(E_0). \tag{14.24}$$

Damit man mit der Formel (14.24) etwas anfangen kann, ist es natürlich wichtig, über $\operatorname{vol}(E_0)$ Bescheid zu wissen. Es sei daran erinnert, dass

$$E_0 = B(0, R)$$

gilt, d.h., bei E_0 handelt es sich um eine Kugel im $\mathbb{R}^n$ mit Radius R. Es ist eine bekannte Tatsache, dass das Volumen einer solchen Kugel proportional zur n-ten Potenz ihres Radius R ist (vgl. beispielsweise das oben genannte Buch von Grötschel, Lovász und Schrijver). Genauer gilt Folgendes: Zu jedem $n \in \mathbb{N}$ existiert eine Konstante c_n, so dass gilt:

$$\operatorname{vol}(E_0) = \operatorname{vol}(B(0, R)) = c_n R^n. \tag{14.25}$$

Die auftretende Konstante wurde mit c_n bezeichnet, da sie von der Dimension n abhängt. Zum besseren Verständnis der Formel (14.25) betrachten wir die Fälle $n = 2$ und $n = 3$:

- Es gilt $c_2 = \pi$, da πR^2 die allseits bekannte Formel für die Fläche eines Kreises vom Radius R ist.
- Es gilt $c_3 = \frac{4}{3}\pi$, da $\frac{4}{3}\pi R^3$ die ebenso bekannte Formel für das Volumen einer Kugel mit Radius R ist (im $\mathbb{R}^3$).

Frage: Nach wie vielen Iterationen können wir sicher sein, dass das Volumen $\operatorname{vol}(E_k)$ kleiner als das Volumen einer n-dimensionalen Kugel vom Radius ε geworden ist?

Antwort: Eine n-dimensionale Kugel vom Radius ε besitzt das Volumen $c_n \varepsilon^n$, d.h., für k soll $\operatorname{vol}(E_k) < c_n \varepsilon^n$ gelten. Wegen (14.24) und (14.25) ist dies gewiss dann der Fall, wenn gilt:

$$e^{-\frac{k}{2n+2}} \cdot c_n R^n < c_n \varepsilon^n. \tag{14.26}$$

Äquivalente Umformung von (14.26) ergibt:

$$\begin{aligned}
e^{-\frac{k}{2n+2}} \cdot c_n R^n < c_n \varepsilon^n \quad &\Leftrightarrow \quad \left(\frac{R}{\varepsilon}\right)^n < e^{\frac{k}{2n+2}} \\
&\Leftrightarrow \quad \frac{R}{\varepsilon} < e^{\frac{k}{n(2n+2)}} \\
&\Leftrightarrow \quad \ln\left(\frac{R}{\varepsilon}\right) < \frac{k}{n(2n+2)} \\
&\Leftrightarrow \quad n(2n+2)\ln\left(\frac{R}{\varepsilon}\right) < k.
\end{aligned}$$

Damit ist unsere Bemerkung c) über die Volumina von E_k, E_{k+1} und $B(0, R)$ zum Abschluss gekommen. Als Ergebnis unserer Ausführungen können wir festhalten:

Feststellung.
Gilt $k = \left\lceil n(2n+2)\ln\left(\frac{R}{\varepsilon}\right)\right\rceil + 1$, so ist das Volumen $\operatorname{vol}(E_k)$ garantiert kleiner als das Volumen einer n-dimensionalen Kugel vom Radius ε.

Da das Ellipsoid E_k das Polyeder P umfasst, kann P keine n-dimensionale Kugel vom Radius ε mehr enthalten, wenn das Volumen $\operatorname{vol}(E_k)$ unter das Volumen einer solchen Kugel gesunken ist. Wir halten fest:

$$\text{Unser Algorithmus stoppt nach höchstens } k = \left\lceil n(2n+2)\ln\left(\frac{R}{\varepsilon}\right)\right\rceil + 1 \text{ Iterationen.} \tag{14.27}$$

Das Ergebnis (14.27) ist ein erster Schritt auf dem Weg zu einer kompletten Laufzeitanalyse der Ellipsoid-Methode. Am Ende dieser Analyse steht die Feststellung, dass es sich bei der Ellipsoid-Methode um einen Algorithmus mit polynomieller Laufzeit handelt. Es bleiben jedoch noch viele Details zu klären und etliche Schwierigkeiten müssen aus dem Weg geräumt werden:

- Beispielsweise muss geklärt werden, wie mit der Tatsache umzugehen ist, dass in der Update-Formel für s_{k+1} ein Wurzelausdruck vorkommt, der im Allgemeinen ja nur näherungsweise berechnet werden kann.
- Außerdem ist zu bedenken, dass unser Algorithmus nicht das ursprüngliche Problem löst, bei dem es ausschließlich um das System $Ax \leq b$ geht (ohne R und ε). Es stellt sich also – etwas salopp ausgedrückt – die Frage, wie man ohne R und ε zurechtkommt.

Fazit: Wir haben das modifizierte Problem besprochen, bei dem R und ε hinzugenommen wurden, wobei wir nicht auf alle Einzelheiten eingegangen sind. Allerdings – und darauf kam es uns an – *waren die geometrischen Grundideen der Ellipsoid-Methode bereits zu erkennen.* Deshalb verzichten wir auf die Darstellung weiterer Details und verweisen auf die Literatur.

Erste Einblicke, wie es weitergeht, erhält man im Buch von Matoušek und Gärtner; *für ausführliche Darstellungen der Ellipsoid-Methode verweisen wir auf das bereits genannte Buch von Bertsimas und Tsitsiklis sowie auf das Werk von Grötschel, Lovász und Schrijver und auf A. Schrijver, Theory of Linear and Integer Programming; empfehlenswert ist auch das Skript von Grötschel* (siehe Literaturverzeichnis).

14.8 Theorie und Praxis

Bei der Ellipsoid-Methode handelt es sich um einen Algorithmus mit polynomieller Laufzeit, was auf den Simplexalgorithmus nicht zutrifft.

In zahlreichen Lehrbüchern wird andererseits darauf hingewiesen, dass die Ellipsoid-Methode *in der Praxis* allem Anschein nach nicht konkurrenzfähig zum Simplexalgorithmus ist; vgl. etwa Cormen et al.

Einer der möglichen Gründe, weshalb der Simplexalgorithmus in der Praxis nicht leicht zu schlagen ist: Das Simplexverfahren benötigt möglicherweise nur für wenige „künstliche“ Beispiele eine exponentiell wachsende Anzahl von Schritten, also für Beispiele, die in der Praxis nicht vorkommen.

Im Jahre 1984, also schon recht bald nach Vorstellung der Ellipsoid-Methode, präsentierte *N. Karmarkar*, ein Forscher von Bell Labs (New Jersey), einen polynomiellen Algorithmus für Lineare Programmierung, der auf ganz anderen Grundideen beruht als die Ellipsoid-Methode bzw. das Simplexverfahren. Beim Algorithmus von Karmarkar handelt es sich um eine *Innere-Punkte-Methode*.

In der Zwischenzeit sind eine ganze Reihe von verwandten Algorithmen entwickelt worden, die alle zur Klasse der Innere-Punkte-Methoden (engl. *interior-point-methods*) gehören.

Nach Meinung von Experten haben Innere-Punkte-Methoden durchaus das Zeug dafür, dem Simplexalgorithmus auch in der Praxis Konkurrenz zu machen (vgl. etwa Matoušek/Gärtner).

14.9 Innere-Punkte-Methoden

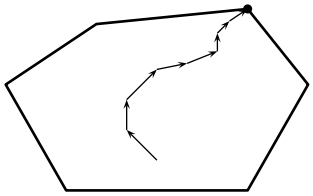
Innere-Punkte-Methoden basieren – ebenso wie der Ellipsoidalgorithmus und das Simplexverfahren – auf einer *geometrischen Grundidee*. Interessanterweise unterscheiden sich die Grundideen in allen drei Fällen erheblich:

- Beim *Simplexverfahren* bewegt man sich längs Kanten von einer Ecke eines Polyeders zur nächsten.
- Beim *Ellipsoidalgorithmus* geht es darum, eine Lösung mithilfe von Ellipsoiden, deren Volumen man fortlaufend verkleinert, einzukesseln.
- Bei den *Innere-Punkte-Methoden* wandert man – im Gegensatz zum Simplexverfahren – durch das Innere des Polyeders, wobei man sorgfältig darauf achtet, den Rand nicht zu berühren; im letzten Schritt hüpft man dann aber doch auf den Rand – nämlich auf eine Ecke, der man sich zuvor

bereits genähert hat.

Natürlich ist diese Beschreibung der geometrischen Grundideen der drei Methoden etwas grob – und es kann auch Varianten geben. Es sollte aber zumindest klar geworden sein, woher der Name „Innere-Punkte-Methoden“ stammt.

Wir illustrieren das Vorgehen einer Innere-Punkte-Methode noch einmal anhand einer Skizze:



Es gibt viele verschiedene Arten von Innere-Punkte-Methoden, mit jeweils etlichen Varianten. Wir wollen uns hier nur eine Klasse von Innere-Punkte-Methoden etwas genauer anschauen, ohne auf allzu viele Details einzugehen. Diese Klasse ist unter dem Namen *Zentrale-Pfad-Methoden* bekannt.

Der Einfachheit halber setzen wir ab jetzt immer voraus, dass uns ein innerer Punkt, d.h. eine zulässige Lösung, die nicht auf den Rand liegt, als *Startpunkt* zur Verfügung steht.

14.9.1 Logarithmische Barriere-Funktionen

Wir betrachten ein beliebiges konvexes Polyeder P im $\mathbb{R}^n$, das durch ein System $Ax \leq b$ von m linearen Ungleichungen gegeben ist; außerdem sei eine lineare Zielfunktion $f(x) = c^T x$ gegeben, die zu maximieren ist. Das dazugehörige LP-Problem lautet also:

$$\begin{aligned} &\text{maximiere } c^T x \\ &\text{unter den Nebenbedingungen} \\ &\qquad Ax \leq b. \end{aligned}$$

Die i-te Nebenbedingung (Ungleichung) wollen wir mit $a_i^T x \leq b_i$ bezeichnen ($i = 1, \ldots, m$).

Bislang ist also alles wie gewohnt; das Neue kommt jetzt: *Wir betrachten zusätzlich für jedes $\mu \in [0, \infty)$ eine weitere Zielfunktion, die wir mit*

$$f_\mu(x)$$

bezeichnen. Die Zielfunktion $f_\mu(x)$ ist folgendermaßen definiert:

$$f_\mu(x) = c^T x + \mu \cdot \sum_{i=1}^{m} \ln\left(b_i - a_i^T x\right). \tag{14.28}$$

Ist $\mu = 0$, so handelt es sich hierbei also um nichts weiter, als die ursprünglich gegebene Zielfunktion $f(x) = c^T x$.

Gilt dagegen $\mu > 0$, so handelt es sich bei $f_\mu(x)$ offenbar um eine *nichtlineare Zielfunktion*: Dass $f_\mu(x)$ für $\mu > 0$ nichtlinear ist, liegt an der auf der rechten Seite von (14.28) auftretenden Logarithmusfunktion. Man beachte auch, dass es sich bei $b_i - a_i^T x$ um den Schlupf der i-ten Ungleichung handelt.

Wir können die Zielfunktionen $f_\mu(x)$ für $\mu > 0$ auch als *Hilfszielfunktionen* bezeichnen, da sie dabei helfen, dass man bei der Durchführung des Innere-Punkte-Verfahrens dem Rand nicht unbeabsichtigt zu nahe kommt. Es werden Hilfsprobleme betrachtet, in denen es darum geht, jeweils eine der Funktionen $f_\mu(x)$ für $\mu > 0$ zu maximieren. *Die Idee dabei*: Wenn x dem Rand von P zu nahe kommt, d.h., wenn für ein i der Schlupf $b_i - a_i^T x$ gegen 0 tendiert, so tendiert $f_\mu(x)$ in Richtung $-\infty$. Anders gesagt: *Wenn man sich darum bemüht $f_\mu(x)$ zu maximieren, so wirkt man gleichzeitig darauf hin, dass man dem Rand fernbleibt.*

Man nennt den in (14.28) zu $c^T x$ hinzugefügten Ausdruck

$$\mu \cdot \sum_{i=1}^{m} \ln\left(b_i - a_i^T x\right) \tag{14.29}$$

eine *Barriere-Funktion* oder auch (genauer) *logarithmische Barriere-Funktion* (engl. *barrier function* bzw. *logarithmic barrier function*).

Im Folgenden präzisieren wir einige der vorangegangenen Bemerkungen.

Unter dem *Rand von* P versteht man die Menge aller $x \in P$, für die

$$b_i - a_i^T x = 0$$

für mindestens ein i gilt $(1 \leq i \leq m)$; unter dem *Inneren von* P versteht man die Menge aller $x \in P$, die nicht zum Rand von P gehören.

Wir bezeichnen im Folgenden das Innere von P mit $\operatorname{int}(P)$; diese Bezeichnung leitet sich von "interior of P" ab. Damit die nachfolgenden Ausführungen sinnvoll sind, *setzen wir ab jetzt immer* $\operatorname{int}(P) \neq \emptyset$ *voraus.*

Da die Funktion $f_\mu(x)$ für $\mu > 0$ auf dem Rand von P nicht definiert ist, betrachten wir nur das Innere von P. Unser *Hilfsproblem* lautet somit:

Für gegebenes $\mu > 0$ finde man (sofern vorhanden) ein $x \in \operatorname{int}(P)$, für das $f_\mu(x)$ maximal ist.

Es gilt nun folgender Satz, der für den Fall, dass P beschränkt ist, die eindeutige Lösbarkeit unseres Hilfsproblems feststellt (Beweis siehe Matoušek/Gärtner).

Satz.
Es sei P ein beschränktes Polyeder mit $\operatorname{int}(P) \neq \emptyset$; außerdem sei $\mu > 0$ gegeben. Dann existiert immer ein eindeutig bestimmter Punkt $x^* \in \operatorname{int}(P)$, für den die Funktion f_μ maximal ist, d.h. $f_\mu(x^*) > f_\mu(x)$ für alle $x \in \operatorname{int}(P)$ mit $x \neq x^*$.

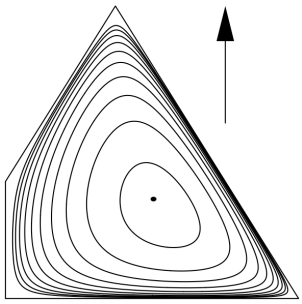
Welcher Punkt $x^* \in \operatorname{int}(P)$ der genannte *Maximalpunkt* ist, hängt vom betrachteten μ ab. Wir bezeichnen den Maximalpunkt daher im Folgenden immer mit $x^*(\mu)$.

Typischerweise gilt: Ist μ eine sehr große Zahl, so ist der Einfluss des Terms $c^T x$ auf den Wert von $f_\mu(x)$ nur sehr gering und $x^*(\mu)$ ist ein Punkt, der in allen Richtungen weit vom Rand entfernt ist; anders gesagt: Der Maximalpunkt $x^*(\mu)$ nimmt in P eine zentrale Lage ein.

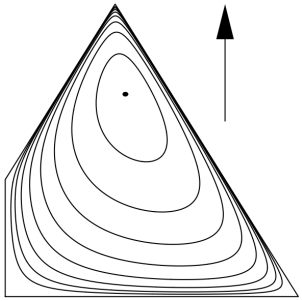
Ist μ dagegen klein, so besitzt der Term $c^T x$ einen bestimmenden Einfluss auf den Wert von $f_\mu(x)$. In diesem Fall unterscheiden sich die Funktionen $f_\mu(x)$ und $f(x) = c^T x$ nur wenig und der Maximalpunkt $x^*(\mu)$ von f_μ wird in der Nähe eines Maximalpunktes für $f(x) = c^T x$ liegen.

In den folgenden Zeichnungen, die aus dem Buch von Matoušek und Gärtner stammen, wird der beschriebene Effekt illustriert. Der Pfeil gibt die Richtung des Vektors c an; es sind Höhenlinien für die Funktion $f_\mu(x)$ eingezeichnet und der Punkt gibt die Lage von $x^*(\mu)$ an. Man stelle sich vor, dass sich an der Stelle $x^*(\mu)$ ein Gipfel („globales Maximum der Funktion $f_\mu(x)$") befindet. Die erste Zeichnung gibt – für ein 2-dimensionales Polyeder P – den Fall $\mu = 100$ wieder; die beiden weiteren Zeichnungen illustrieren die Funktion $f_\mu(x)$ für die Fälle $\mu = 0.5$ und $\mu = 0.1$.

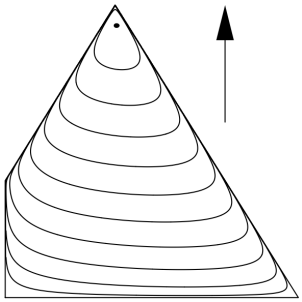
Der sehr anschauliche Begriff der *Höhenlinie* – man denke an Landkarten – wird anschließend exakt definiert.



Der Graph der Funktion $f_\mu(x)$ für den Fall $\mu = 100$.



Der Graph der Funktion $f_\mu(x)$ für den Fall $\mu = 0.5$.



Der Graph der Funktion $f_\mu(x)$ für den Fall $\mu = 0.1$.

Definition der Höhenlinien.
Ist $f : \mathbb{R}^2 \to \mathbb{R}$ eine Funktion und $\alpha \in \mathbb{R}$, so definiert man die *Höhenlinie von f zum Wert α* als die Menge der Punkte $(x_1, x_2) \in \mathbb{R}^2$, für die $f(x_1, x_2) = \alpha$ gilt.

Anders gesagt: Unter einer Höhenlinie versteht man die Menge aller Punkte (x_1, x_2), für die die Funktion f einen vorgegebenen Wert („Höhe“) annimmt[7].

[7] Höhenlinien werden natürlich analog definiert, wenn die betrachtete Funktion nur auf einer Teilmenge D von $\mathbb{R}^2$ existiert.

14.9.2 Der Begriff des zentralen Pfades

Wir betrachten die im vorangegangenen Abschnitt definierten Maximalpunkte $x^*(\mu)$ und führen den Begriff des „zentralen Pfads“ ein: Unter dem *zentralen Pfad* versteht man die Menge

$$\Big\{x^*(\mu) : \mu > 0\Big\}.$$

Der zentrale Pfad ist also die Menge aller Maximalpunkte $x^*(\mu)$ für $\mu > 0$. Es sei betont, dass der zentrale Pfad nicht nur von P und der Zielfunktion c abhängt, sondern auch von der Darstellung von P, d.h. vom Ungleichungssystem $Ax \leq b$, durch das P gegeben ist: Dass dies so ist, erkennt man anhand von (14.28).

Wir sind nun in der Lage, die *Grundidee von Zentralen-Pfad-Methoden* zu formulieren.

Grundidee von Zentralen-Pfad-Methoden.

Man startet mit einem $x^*(\mu)$ für ein geeignetes großes μ und folgt dann dem zentralen Pfad, indem man μ fortlaufend verkleinert.

Folgt man dieser Grundidee, so hat man – wie wir zuvor gesehen haben – in jedem Schritt ein Hilfsproblem mit einer nichtlinearen Zielfunktion $f_\mu(x)$ zu lösen. Dadurch erhält man fortlaufend verbesserte Näherungswerte für die angestrebte Lösung des Problems

$$\begin{aligned} &\text{maximiere } c^T x \\ &\text{unter den Nebenbedingungen} \\ &\qquad Ax \leq b. \end{aligned} \tag{14.30}$$

Zum Schluss geht es dann darum, wie man aus den gewonnenen Näherungslösungen eine exakte Lösung von (14.30) gewinnt.

Es gäbe noch viele weitere Details zu besprechen. Da es uns nur darum ging, die Grundideen darzulegen, brechen wir an dieser Stelle ab und verweisen auf das Buch von Matoušek und Gärtner sowie auf die dort genannte Literatur.

15 Literatur

- D. Bertsimas, J. N. Tsitsiklis:
  Introduction to Linear Optimization. Athena Scientific. Belmont, Massachusetts. 1997.
- A. Beutelspacher, M.-A. Zschiegner:
  Diskrete Mathematik für Einsteiger. Vieweg-Verlag. 2014. 5. Auflage.
- V. Chvátal:
  Linear Programming. W. H. Freeman and Company. New York. 2002. 16. Auflage.
- Th. Cormen, Ch. Leiserson, R. Rivest, C. Stein:
  Algorithmen – Eine Einführung. Oldenbourg-Verlag. 2010. 3. Auflage.
- S. Dasgupta, C. Papadimitriou, U. Vazirani:
  Algorithms. McGraw Hill. 2008.
- R. Diestel:
  Graph Theory. Springer. 2016. 5. Auflage.
- M. Grötschel, L. Lovász, A. Schrijver:
  Geometric Algorithms and Combinational Optimization. Springer. 1993. 3. Auflage.
- D. Jungnickel:
  Graphs, Networks and Algorithms. Springer-Verlag. Berlin, Heidelberg, New York. 2012. 4. Auflage.
- J. Kleinberg, É. Tardos:
  Algorithm Design. Pearson. Boston. 2006.
- B. Korte, J. Vygen:
  Combinatorial Optimization. Theory and Algorithms. Springer. 2012. 5. Auflage.
- N. Lauritzen:
  Undergraduate Convexity. World Scientific. 2013.
- L. Lovász, M. D. Plummer:
  Matching Theory. North-Holland. 1986.
- D. Luenberger, Yinyu Ye:
  Linear and Nonlinear Programming. Springer. 2008. 3. Auflage.
- J. Matoušek, B. Gärtner:
  Understanding and Using Linear Programming. Springer-Verlag. Berlin, Heidelberg, New York. 2007.
- K. Neumann, M. Morlock:
  Operations Research. Hanser-Verlag. 2004. 2. Auflage.
- A. Schrijver:
  Combinatorial Optimization. Polyhedra and Efficiency. Volume A: Paths, Flows, Matchings. Springer. 2003.

- A. Schrijver:
  Theory of Linear and Integer Programming. Wiley. 1998.
- A. Steger:
  Diskrete Strukturen. Band 1. Springer-Verlag. 2007.

Weitere nützliche Hinweise auf Literatur zur Linearen Optimierung sowie auf Software findet man in

- M. Grötschel:
  Lineare Optimierung. Skriptum zur Vorlesung im WS 2003/04. Technische Universität Berlin.

Zu guter Letzt sei der Klassiker auf dem Gebiet der Linearen Optimierung aufgeführt:

- G. B. Dantzig:
  Linear Programming and Extensions. Princeton University Press. 1963.

Index