Algorithmische Mathematik I

Vorlesungsskript WS 2009/10

Mario Bebendorf

Inhaltsverzeichnis

Vorwort

Dieses Skript fasst den Inhalt der von mir im Wintersemester 2009/10 an der Universität Bonn gehaltenen Vorlesung *Algorithmische Mathematik I* zusammen und ist eine Überarbeitung der Skripte zu den von Helmut Harbrecht und Stefan Hougardy im Wintersemester 2007/08 bzw. 2008/09 gehaltenen gleichnamigen Vorlesungen. Mein Dank gebührt Herrn Maximilian Kirchner (`mkirchner@uni-bonn.de`), der dieses LaTeX-Dokument aus der Vorlesungsmitschrift erstellt hat. Korrekturvorschläge sind willkommen.

Bonn, 29. März 2010

Einleitung

Die algorithmische Mathematik vereint die algorithmischen Grundlagen aus verschiedenen Bereichen der Mathematik

- Diskrete Mathematik
- Numerische Mathematik
- Stochastik

Die Aufgabe der Algorithmischen Mathematik ist die Konstruktion und Analyse von Algorithmen zur Lösung mathematischer Probleme. Ursprung dieser Probleme können Aufgabenstellungen aus Technik, Naturwissenschaften, Wirtschaft und Sozialwissenschaften sein. Von erheblicher praktische Bedeutung ist deshalb die Umsetzung der Algorithmen in ein Computerprogramm.

Literaturangaben:

- P. Deuflhard und A. Hohmann: *Numerische Mathematik*, de Gruyter Verlag
- M. Hanke-Bourgeois: *Grundlagen d. numerischen Mathematik und des wissenschaftlichen Rechnens*, Teubner-Verlag
- J. Stoer: *Numerische Mathematik I*, Springer-Verlag
- N. Blum: *Algorithmen und Datenstrukturen*, Oldenbourg Verlag
- B. Korte und J. Vygen: *Combinatorial Optimization: Theory and Algorithms*, Springer-Verlag

1 Maschinenzahlen und -arithmetik

Bei der mathematischen Analyse von Algorithmen wird das Rechnen mit reellen Zahlen als exakt vorausgesetzt. Tatsächlich werden im Rechner jedoch so genannte **Maschinenzahlen** verwendet, mit denen ein exaktes Rechnen nicht möglich ist.

1.1 Zahlensysteme

Die Darstellung einer Zahl basiert auf Zahlensystemen. Es sei $\mathbb{N} = \{1, 2, 3, \ldots\}$ die Menge der natürlichen Zahlen und $b \in \mathbb{N}$, $b > 1$, beliebig. Dann heißt die Menge $\Sigma_b := \{0, \ldots, b-1\}$ das **Alphabet** des b-adischen Systems. Ein Wort der Länge n ist eine Aneinanderreihung von n Symbolen aus Σ_b. b wird als **Basis** bezeichnet.

Beispiel 1.1.

- Dezimalsystem: $b = 10$, $\Sigma_b = \{0, 1, 2, \ldots, 9\}$. Worte aus diesem Alphabet sind zum Beispiel 123, 734, 7806. Eine feste Wortlänge $n = 4$ erreicht man durch Hinzufügen von Nullen: 0123, 0734, 7806. Im Rechner werden üblicherweise Worte fester Länge verwendet (8-, 16-, 32-, 64-Bit) (engl. binary digit).
- $\Sigma_2 = \{0, 1\}$ Dual- oder Binäralphabet
- $\Sigma_8 = \{0, \ldots, 7\}$ Octalalphabet
- $\Sigma_{16} = \{0, \ldots, 9, A, B, C, D, E, F\}$ Hexadezimalalphabet
- Alte Basen sind $b = 12$ (Dutzend) und $b = 60$ (Zeitrechnung).

Die Basen 2, 8, 16 spielen in der Informatik eine entscheidende Rolle.

Satz 1.2. *Sei $b \in \mathbb{N} \setminus \{1\}$ und $z \in \mathbb{N}_0 := \mathbb{N} \cup \{0\}$. Dann existiert $n \in \mathbb{N}$ mit $z < b^n$, und z ist eindeutig als Wort der Länge n über Σ_b darstellbar durch*

$$z = \sum_{i=0}^{n-1} z_i b^i$$

mit $z_i \in \Sigma_b$ für alle $0 \leq i < n$.
Vereinfacht wir die Ziffernschreibweise $z = (z_{n-1}, z_{n-2}, \ldots, z_1, z_0)_b$ verwendet.

Beweis. Der Beweis erfolgt durch vollständige Induktion nach z. Für den Induktionsanfang beachte man, dass für alle $z < b$ gilt, dass $n = 1$ und $z = z_0$.
Als Induktionsvoraussetzung nehmen wir an, die Behauptung gelte für alle Zahlen $1, 2, \ldots, z-1$. Im Induktionsschluss zeigen wir, dass unter dieser Annahme die Behauptung für $z > b$ gilt. Es ist nämlich

$$z = \hat{z} \cdot b + (z \mod b)$$

mit $\hat{z} := \left\lfloor \frac{z}{b} \right\rfloor$. Hier verwenden wir für $x \in \mathbb{R}$ die Notationen $\lfloor x \rfloor := \max\{k \in \mathbb{Z} : k \leq x\}$ und $(x \mod b) := x - \left\lfloor \frac{x}{b} \right\rfloor \cdot b$.
Wegen $\hat{z} < z$ besitzt $\hat{z}$ nach Induktionsvoraussetzung eine Darstellung $\hat{z} = (\hat{z}_{n-1}, \ldots, \hat{z}_0)_b$. Dabei ist $\hat{z}_{n-1} = 0$, weil aus $\hat{z}_{n-1} b^{n-1} \leq \hat{z}$ und $\hat{z} \cdot b \leq z < b^n$ folgt $\hat{z}_{n-1} b^n < b^n$. Also ist $\hat{z}_{n-1} < 1$ und somit $\hat{z}_{n-1} = 0$. Definiere $(z_{n-1}, \ldots, z_0)_b$ durch $z_i := \hat{z}_{i-1}$, $i = n-1, \ldots, 1$, und $z_0 := z \mod b$. Dann gilt

$$\begin{aligned} z &= \hat{z} \cdot b + (z \mod b) \\ &= b \cdot \sum_{i=0}^{n-2} \hat{z}_i \cdot b^i + (z \mod b) \\ &= \sum_{i=1}^{n-1} \hat{z}_{i-1} \cdot b^i + (z \mod b) \\ &= \sum_{i=0}^{n-1} z_i \cdot b^i. \end{aligned}$$

Wir müssen noch die Eindeutigkeit zeigen. Angenommen, es existieren zwei verschiedene Darstellungen $z = (z_{n-1}, \ldots, z_0)_b = (z'_{n-1}, \ldots, z'_0)_b$. Sei $p \in \mathbb{N}$ der größte Index mit $z_p \neq z'_p$. O.B.d.A. gilt $z_p > z'_p$. Dann müssen die niederwertigen Stellen $z'_{p-1}, \ldots, z'_0$ die kleinere $(p+1)$-te Stelle z'_p ausgleichen. Die größte durch diese p Stellen darstellbare Zahl ist aber

$$(b-1)b^0 + \ldots + (b-1)^{p-1} = (b-1) \sum_{i=0}^{p-1} b^i = b^p - 1.$$

Von der letzten Identität überzeugt man sich schnell durch Ausmultiplizieren. Da aber $1 \cdot b^p$ eine Einheit der niederwertigen $(p+1)$-ten Stelle ist, kann diese nicht ausgeglichen werden. Dieser Widerspruch beweist die Eindeutigkeit. □

Der Beweis ist konstruktiv: man erhält einen Algorithmus zur Darstellung einer Zahl in einem b-adischen System.

Beispiel 1.3. Darstellung von 1364 im Octalsystem

$$\begin{aligned} 1364 &= 170 \cdot 8 + 4 \\ 170 &= 21 \cdot 8 + 2 \\ 21 &= 2 \cdot 8 + 5 \\ 2 &= 0 \cdot 8 + 2 \end{aligned}$$

Hieraus erhält man $z = (1364)_{10} = (2524)_8$.

Algorithmus 1.4 (b-adische Darstellung $(x_{n-1}, \ldots, x_0)_b$ von $z \in \mathbb{N}_0$).

```
Input: unsigned int z,b,n;
Output: unsigned int x[n];
for (i=0; i<n; i++) x[i]=0;
i=0;
while (z>0) {
   x[i] = z % b; // entspricht (z mod b)
```

```
    z = z/b;       // entspricht ganzzahliger Division
    i++;
}
```

Die Umwandlung einer Zahl in b-adischer Darstellung in eine Dezimalzahl erfolgt mittels des **Horner-Schemas**.

$$\begin{aligned} z &= \sum_{i=0}^{n-1} z_i b^i = z_0 + b \sum_{i=0}^{n-2} z_{i+1} b^i \\ &= z_0 + b \left(z_1 + b \sum_{i=0}^{n-3} z_{i+2} b^i \right) \\ &= z_0 + b(z_1 + b(z_2 + b(z_3 + \ldots (z_{n-2} + b z_{n-1})))) \end{aligned}$$

Algorithmus 1.5 (Umwandlung aus b-adischer Darstellung)**.**

```
Input: unsigned int x[n],b,n;
Output: unsigned int z;
z=0;
for (i=n; i>0; i--) z = z * b + x[i-1];
```

Bemerkung. Man beachte, dass die Variante

```
for (i=n-1; i>=0; i--) z = z * b + x[i];
```

zwar naheliegend ist, aber tatsächlich eine Endlosschleife darstellt, falls i vom Typ `unsigned int` ist.

1.2 Darstellung ganzer Zahlen im Rechner

Um eine ganze Zahl in Binärdarstellung zu speichern, scheint es natürlich, ein Bit ($0 \mathrel{\widehat{=}}$ "+", $1 \mathrel{\widehat{=}}$ "−") für das Vorzeichen und bei einer Wortlänge von n weitere $n-1$ Bits für den Betrag der Zahl zu verwenden. Da die 0 zwei Darstellungen besitzt (± 0), können $2^n - 1$ Zahlen dargestellt werden.

Beispiel 1.6. Im Fall $n = 3$ ergeben sich folgende Zahlen.

Bitmuster	Dezimaldarstellung
0 0 0	+0
0 0 1	+1
0 1 0	+2
0 1 1	+3
1 0 0	−0
1 0 1	−1
1 1 0	−2
1 1 1	−3

Die beschriebene Darstellung ist auf Rechnern allerdings unpraktisch, weil auf Rechnern nur Additionswerke vorhanden sind. Daher benötigt man eine Darstellung, bei der die Subtraktion auf die ziffernweise Addition zurückgeführt werden kann. Dies gelingt mit der so genannten **Komplement-Darstellung**.

Definition 1.7. *Sei $z = (z_{n-1}, z_{n-2}, \ldots, z_0)_b$ eine n-stellige b-adische Zahl. Das **b-Komplement** $K_b(z)$ ist definiert als $K_b(z) = (b-1-z_{n-1}, b-1-z_{n-2}, \ldots, b-1-z_0)_b + 1$.*

Beispiel 1.8.

- $K_{10}((325)_{10}) = (674)_{10} + (1)_{10} = (675)_{10}$
- $K_2((10110)_2) = (01001)_2 + (1)_2 = (01010)_2$

Lemma 1.9. *Für jede n-stellige b-adische Zahl gilt*

(i) $z + K_b(z) = b^n$,

(ii) $K_b(K_b(z)) = z$.

Beweis. Nach Definition des b-Komplements ist

$$z + K_b(z) = (b-1, \ldots, b-1)_b + (1)_b = 1 + \sum_{i=0}^{n-1}(b-1)b^i$$
$$= 1 + (b-1)\sum_{i=0}^{n-1} b^i = b^n - 1 + 1 = b^n.$$

Für (ii) beachte man, dass aus (i) folgt

$$\underbrace{K_b(z)}_{z'} + K_b(\underbrace{K_b(z)}_{z'}) = b^n = z + K_b(z)$$

und hieraus $K_b(K_b(z)) = z$. □

Nach (i) von Lemma 1.9 ergibt sich das b-Komplement von z bei n Stellen als Differenz von b^n zu z; siehe Beispiel 1.8 für $n = 3$.

Definition 1.10. *Sei $z \in \mathbb{Z}$ mit $-\left\lfloor \frac{b^n}{2} \right\rfloor \leq z < \left\lceil \frac{b^n}{2} \right\rceil$. Die **$b$-Komplement-Darstellung** $(z)_{K_b}$ von z ist definiert als*

$$(z)_{K_b} = \begin{cases} (z)_b, & \text{falls } z \geq 0, \\ (K_b(|z|))_b, & \text{sonst.} \end{cases}$$

*Hierbei ist $\lceil x \rceil := \min\{k \in \mathbb{Z} : k \geq x\}$ für $x \in \mathbb{R}$. Das Intervall $\left[-\left\lfloor \frac{b^n}{2} \right\rfloor, \left\lceil \frac{b^n}{2} \right\rceil\right)$ heißt **darstellbarer Bereich**.*

Beispiel 1.11.

(a) Im Fall $b = 10, n = 2$ ist der darstellbare Bereich $-50 \leq z < 50$. Konkrete Darstellungen sind

Zahl	Darstellung
43	43
-13	87
-27	73
38	38

(b) Im Fall $b = 2, n = 3$ ist der darstellbare Bereich $-4 \leq z < 4$. Konkrete Darstellungen sind

Zahl	Bitmuster
-4	100
-3	101
-2	110
-1	111
0	000
1	001
2	010
3	011

Dies erklärt die Bemerkung vom Ende des Abschnitts 1.1. Die Zahl -1 hat die Darstellung $(111\ldots1)_{K_2}$. Diese Ziffern als `unsigned int` interpretiert ergeben $(1,\ldots,1)_2 = 2^n - 1 \geq 0$. Ein ähnliches Problem tritt beim so genannten Unter- bzw. Überlauf (Verlassen des darstellbaren Bereiches) beim Typ `int` auf. Die größte darstellbare Zahl ist $(011\ldots1)_{K_2} = 2^{n-1} - 1 =: x_{\max}$ während $x_{\max} + 1 = (10\ldots0)_{K_2}$ als -2^{n-1} interpretiert wird.

Betrachten wir nun die Addition von Zahlen in dieser Darstellung. Dazu sei mit $\oplus$ die ziffernweise Addition der Darstellungen zweier Zahlen bezeichnet, wobei ein eventueller Überlauf auf die $(n+1)$-te Stelle vernachlässigt wird, d.h. wir rechnen modulo b^n

Satz 1.12. *Seien x, y n-stellige Zahlen und $x, y, x + y$ im darstellbaren Bereich. Dann gilt $(x + y)_{K_b} = (x)_{K_b} \oplus (y)_{K_b}$.*

Beweis. Für $z \geq 0$ gilt per Definition $(z)_{K_b} = z$. Im Fall $z < 0$ haben wir nach Lemma 1.9 $(z)_{K_b} = K_b(-z) = b^n + z$. In beiden Fällen gilt daher $z = (z)_{K_b} \mod b^n$. Insbesondere ist $(x)_{K_b} + (y)_{K_b} \mod b^n = x + y = (x + y)_{K_b} \mod b^n$. □

Beispiel 1.13. Sei $b = 10, n = 2$. Dann ergibt sich ein darstellbarer Bereich $-50 \leq z < 50$. Wir erhalten

$$\begin{aligned}
(27)_{K_{10}} \oplus (12)_{K_{10}} &= 27 + 12 \mod 100 = 39 = (39)_{K_{10}},\\
(27)_{K_{10}} \oplus (-15)_{K_{10}} &= 27 + 85 \mod 100 = 12 = (12)_{K_{10}},\\
(27)_{K_{10}} \oplus (-34)_{K_{10}} &= 27 + 66 \mod 100 = 93 = (-7)_{K_{10}},\\
(-27)_{K_{10}} \oplus (-21)_{K_{10}} &= 73 + 79 \mod 100 = 52 = (-48)_{K_{10}}.
\end{aligned}$$

Die Subtraktion zweier Zahlen x, y läßt sich mittels $x - y = x + (-y)$ auf die Addition zurückführen.

Satz 1.14. *Seien x, y n-stellige Zahlen und $x, y, x - y$ im darstellbaren Bereich. Dann gilt $(x - y)_{K_b} = (x)_{K_b} \oplus K_b((y)_{K_b})$.*

Beweis. Wir zeigen zunächst, dass $(-y)_{K_b} = K_b((y)_{K_b})$ gilt. Ist nämlich $y > 0$, so gilt per Definition $(-y)_{K_b} = K_b(y) = K_b((y)_{K_b})$. Im Fall $y \leq 0$ hat man $(-y)_{K_b} = -y = K_b(K_b(-y)) = K_b((y)_{K_b})$. Nach Satz 1.12 gilt dann

$$(x - y)_{K_b} = (x)_{K_b} \oplus (-y)_{K_b} = (x)_{K_b} \oplus K_b((y)_{K_b}).$$

□

Beispiel 1.15. Es sei $b = 10$, $n = 2$. Hieraus folgt für den darstellbaren Bereich $-50 \leq z < 50$. Wir erhalten

$$\begin{aligned}
(37)_{K_{10}} \oplus K_{10}((28)_{K_{10}}) &= 37 + 72 \mod 100 = 9 = (37 - 28)_{K_{10}},\\
(37)_{K_{10}} \oplus K_{10}((48)_{K_{10}}) &= 37 + 52 \mod 100 = 89 = (37 - 48)_{K_{10}},\\
(-12)_{K_{10}} \oplus K_{10}((24)_{K_{10}}) &= 88 + 76 \mod 100 = 64 = (-12 - 24)_{K_{10}}.
\end{aligned}$$

1.3 Darstellung reeller Zahlen

1.3.1 Festkommazahlen

Definition 1.16. *Bei der **Festkommadarstellung** (engl. fixed point presentation) wird bei vorgegebener Stellenzahl n eine feste Anzahl an Nachkommastellen vereinbart, d.h.*

$$z = \pm(z_{k-1}, z_{k-2}, \ldots, z_0. z_{-1}, \ldots, z_{k-n}) = \pm \sum_{i=k-n}^{k-1} z_i b^i$$

mit k Vorkomma- und $n - k$ Nachkommastellen.

Beispiel 1.17.

- $(271.314)_{10} = 271.314$,
- $(101.011)_2 = 2^2 + 2^0 + 2^{-2} + 2^{-3} = 5.375$.

Der Hauptnachteil der Festkommazahlen ist, dass der Abstand $\delta := 2^{k-n}$ zwischen zwei benachbarten Zahlen immer gleich ist. Liegt beispielsweise ein Rechenergebnis x genau zwischen zwei Festkommazahlen x_0 und $x_0 + \delta$, so muss gerundet werden. Die Rundungsgenauigkeit beträgt $\frac{\delta}{2}$, d.h. $|x - x_0| = \frac{\delta}{2}$. Der relative Fehler hängt dann aber von der Größe von x ab:

$$\frac{|x - x_0|}{|x|} = \frac{\delta}{2|x|}.$$

Beispielsweise ist für $\delta = 10^{-3}$ und $x = 0.5$ der relative Fehler $\frac{10^{-3}}{2 \cdot 0.5} = 10^{-3}$, während für $x = 0.05$ der relative Fehler 10^{-2} beträgt. Weil man aber an einer relativen Genauigkeit interessiert ist, die nicht von der Größe der Zahl abhängt, verwendet man anstelle der Festkommazahlen die im Folgenden vorgestellten Gleitkommazahlen.

1.3.2 Gleitkommazahlen

Ähnlich wie in Satz 1.2 beweist man

Satz 1.18. *Sei $b \in \mathbb{N} \setminus \{1\}$. Jedes $x \in \mathbb{R}$ besitzt eine Darstellung $x = \pm \sum_{i=0}^{\infty} d_i b^{e-i}$ mit $d_i \in \Sigma_b$ und $e \in \mathbb{Z}$.*

Beweis. Sei $x_0 \in \mathbb{R}$ und o.E. sei $x_0 > 0$. Dann gibt es $e \in \mathbb{Z}$ mit $0 \leq x_0 < b^{e+1}$. Setzt man

$$d_0 = \left\lfloor \frac{x_0}{b^e} \right\rfloor,$$

so ist $d_0 \in \Sigma_b$. Wegen $d_0 \leq \frac{x_0}{b^e} < d_0 + 1 \iff d_0 b^e \leq x_0 < d_0 b^e + b^e$ folgt für

$$x_1 := x_0 - d_0 b^e$$

die Abschätzung $0 \leq x_1 < b^e$. Genauso wie x_1 aus x_0 generiert wurde, kann man nun x_2 aus x_1, x_3 aus x_2, usw. erzeugen. Nach s Schritten erhält man somit

$$x_0 = \sum_{i=0}^{s-1} d_i b^{e-i} + x_s$$

mit $d_i \in \Sigma_b$, und es gilt $0 \leq x_s < b^{e+1-s}$. Wegen $b^{e+1-s} \xrightarrow[s\to\infty]{} 0$ folgt $x_s \xrightarrow[s\to\infty]{} 0$ und daraus die Behauptung. □

Man beachte, dass die Darstellung aus Satz 1.18 nicht eindeutig ist. Beispielsweise hat man $1.4 \cdot 10^{-3} = 0.14 \cdot 10^{-2}$. Die Eindeutigkeit kann aber durch Normierung, d.h. z.B. durch die Forderung $d_0 \neq 0$, erzwungen werden. Wegen der Endlichkeit des Rechners beschränken wir uns auf Zahlen, die folgende Darstellung besitzen.

Definition 1.19. *Die Menge der **normalisierten Gleitkommazahlen** sei gegeben durch*

$$\mathbb{F}(b,t,p) := \left\{ \pm \sum_{i=0}^{t-1} d_i b^{e-i} : d_i \in \Sigma_b,\ d_0 \neq 0,\ e = \pm \sum_{j=0}^{p-1} e_j b^j,\ e_j \in \Sigma_b \right\} \cup \{0\}.$$

Dabei bezeichnet $b \in \mathbb{N} \setminus \{1\}$ die Länge der Basis, $t \in \mathbb{N}$ die Mantissenlänge und p die Exponentenlänge.
Man speichert also die Informationen

$$(v_m, d_{t-1}, \ldots d_0, v_e, e_{p-1}, \ldots, e_0)$$

*mit $v_m, v_e \in \{+,-\}$ und $d_i, e_j \in \Sigma_b$. Der zugeordnete Wert einer solchen Maschinenzahl ist $v_m \cdot m \cdot b^e$ mit dem Vorzeichen v_m, der **Mantisse** $m = \sum_{i=0}^{t-1} d_i b^{-i}$ und dem **Exponenten** $e = v_e \sum_{j=0}^{p-1} e_j b^j$.*

Beispiel 1.20. Die Dezimalzahl $x = 3.375$ hat die Gleitkommadarstellung ($b = 2$, $t = 5$, $p = 2$)

$$x = (+, 1, 1, 0, 1, 1, +, 0, 1).$$

Dies sieht man aus

$$\begin{aligned} m = \sum_{i=0}^{t-1} d_i b^{-i} &= 1 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4} \\ &= 1 + \frac{1}{2} + \frac{1}{8} + \frac{1}{16} = 1.6875, \\ e &= 1 \cdot 2^0 + 0 \cdot 2^1 = 1, \\ x &= 1.6875 \cdot 2^1 = 3.375. \end{aligned}$$

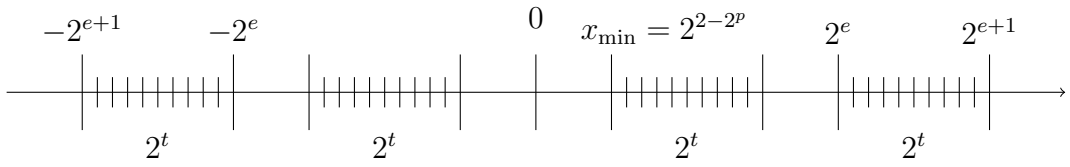
Bemerkung. Weil bei der Basis $b = 2$ das erste Bit $d_0 = 1$ feststeht, wird es oft nicht dargestellt (sog. "**hidden bit**"). In diesem Fall werden t Mantissenbits zur Darstellung von $d_1, \ldots, d_t$ verwendet. Die Null muss dann gesondert dargestellt werden, z.B. indem man die Bits der Mantisse als Null und den Exponenten als kleinster darstellbarer Exponent $1 - 2^p$ wählt (IEEE Standard 754). $(d_1, \ldots, d_t)$ wird oft auch als **Signifikant** bezeichnet. Beim "hidden bit"gelang man somit zu folgendem Zahlenbereich

- betragsmäßig größte Zahl: $x_{\max} = (1.1 \ldots 1)2^{\frac{(1\ldots1)_2}{p}} = 2^{e_{\max}}$ mit $e_{\max} = 2^p - 1$
- betragsmäßig kleinste Zahl ungleich Null: $x_{\min} = (1.0 \ldots 0)2^{-(1\ldots10)_2} = 2^{e_{\min}}$ mit $e_{\min} = 2 - 2^p$. Dabei beachte man, dass $1 - 2^p$ für die Darstellung der Null reserviert ist.

Bei vorgegebenem Exponenten e liegt die Menge der mit diesem Exponenten darstellbaren Zahlen im Intervall $2^e \leq x < 2^{e+1}$, weil (bei positivem Vorzeichen)

$$x = (1.d_1, \ldots, d_t)2^e = (1 + \frac{k}{2^t})2^e$$

für ein ganzzahliges k mit $0 \leq k < 2^t$.
Es können für einen Exponenten insgesamt 2^t Zahlen dargestellt werden.



1.3.3 Genauigkeit der Gleitkommadarstellung

Die Endlichkeit der Menge $\mathbb{F}$ der Maschinenzahlen erzwingt bei der Konvertierung oder der Darstellung von Zwischenergebnissen die Rundung auf Maschinenzahlen.

Definition 1.21. *Die **Rundung** $rd : \mathbb{R} \to \mathbb{F}(b, t, p)$ ist definiert durch*

$$|x - rd(x)| = \min_{a \in \mathbb{F}} |x - a|.$$

*Den maximalen relativen **Rundungsfehler** $\varepsilon_{\mathbb{F}}$ für $x_{\min} \leq |x| \leq x_{\max}$ nennt man **Maschinengenauigkeit**. Die Stellen der Mantisse heißen **signifikante Stellen**. Eine t-stellige Mantisse bezeichnet man auch als t-stellige Arithmetik.*

Format	Bytes $1+t+p$	Basis	Länge der Mantisse	hidden bit	Bits für den Exponenten
`single`	4	2	24	ja	8
`double`	8	2	53	ja	11

Tabelle 1.1: Zwei Zahlentypen mit ihren Parametern.

Bemerkung.

(a) Es gilt $\mathrm{rd}(x) = x(1+\varepsilon)$ mit $|\varepsilon| \leq \varepsilon_{\mathbb{F}}$ weil

$$\varepsilon_{\mathbb{F}} \geq \left|\frac{x - \mathrm{rd}(x)}{x}\right| = \left|\frac{x - x(1+\varepsilon)}{x}\right| = |\varepsilon|\,.$$

(b) Liegt x genau in der Mitte zwischen zwei benachbarten Gleitkommazahlen, so wird abgerundet.

Satz 1.22. *Die Maschinengenauigkeit für $\mathbb{F}(b,t,p)$ ist $\varepsilon_{\mathbb{F}} = \frac{1}{2}b^{1-t}$.*

Beweis. Sei $x = \sum_{i=0}^{\infty} d_i b^{e-i}$. Dann sind

$$x' := \sum_{i=0}^{t-1} d_i b^{e-i} \quad \text{und} \quad x'' := b^{e-t+1} + \sum_{i=0}^{t-1} d_i b^{e-i}$$

beide in $\mathbb{F}$, und es gilt $x' \leq x < x''$. Daher folgt $|x - \mathrm{rd}(x)| \leq \frac{1}{2}|x'' - x'| = \frac{1}{2}b^{e-t+1}$. Wegen $d_0 \neq 0$ gilt $|x| \geq b^e$. Also folgt

$$\frac{|x - \mathrm{rd}(x)|}{|x|} \leq \frac{1}{2}\frac{b^{e-t+1}}{b^e} = \frac{1}{2}b^{1-t}.$$

□

Die Maschinengenauigkeit $\varepsilon_{\mathbb{F}}$ ist die wichtigste Größe eines Gleitkommasystems. Die Anzahl signifikanter Stellen s im Dezimalsystem berechnet man durch Auflösen von

$$\varepsilon_{\mathbb{F}} = \frac{1}{2}b^{1-t} = \frac{1}{2}10^{1-s}$$

nach s. Danach ergibt sich $s = \lfloor 1 + (t-1)\log_{10} b \rfloor$.

Beispiel 1.23. Wir betrachten die im IEEE-Standard 754 definierten Zahlentypen `single` und `double`. Aus Tabelle 1.1 ergibt sich für den Typ `double` ein minimaler und maximaler Exponent $e_{\min} = 2 - 2^p = -1022$ bzw. $e_{\max} = 2^p - 1 = 1023$ und daraus

$$x_{\max} = (2 - 2^{-t})2^{2^p - 1} \approx 1.797 \cdot 10^{308},$$
$$x_{\min} = 2^{2-2^p} \approx 2.225 \cdot 10^{-308}.$$

Außerdem ist $\varepsilon_{\mathbb{F}} = \frac{1}{2}2^{1-53} \approx 1.1 \cdot 10^{-16}$, was 16 signifikanten Dezimalstellen entspricht.

Der Exponent $e_{\min} - 1 = -1023$ wird verwendet, um sog. **denormalisierte Gleitkommazahlen** zu speichern. Dabei repräsentiert die Mantisse $(0, 0, \ldots, 0)$ die Zahl 0 und eine positive Mantisse einen Wert zwischen 0 und $x_{\min}$ als Festkommazahl. Ein Overflow ($\pm$ inf) wird durch den maximalen Exponenten mit Bitmuster $(1 \ldots 1)$ und Mantisse 0 repräsentiert. Die Werte NaN (engl. not a number) werden als Darstellung für indefinite Werte verwendet. Diese treten z.B. auf, wenn man Operationen wie $\frac{0}{0}$ ausführt. NaN wird ebenfalls durch den maximalen Exponenten aber mit positiver Mantisse kodiert. Der Exponent wird um $|e_{\min}|+1$ verschoben, d.h. statt e wird $e + |e_{\min}|$ gespeichert (sog. **Biasdarstellung**).

Beispiel 1.24. Die Dezimalzahl 10.0 wird im IEEE-Standard 754 dargestellt als

0	010…0	10000000010
Vorzeichen	Signifikant	Exponent
+	1.25	$1026 - 1023 = 3$

Während $-0,8$ als (gerundetes) Bitmuster

1	1001101…10011010	01111111110
Vorzeichen	Signifikant	Exponent
−	1.599…	$1022 - 1023 = -1$

gespeichert wird.

Weil $x = 2\varepsilon_{\mathbb{F}}$ die kleinste Zahl ist, für die $\mathrm{rd}(1 + x) > 1$, kann die Maschinengenauigkeit auf einem Rechner mit folgendem Algorithmus bestimmt werden.

Algorithmus 1.25 (Bestimmung von $\varepsilon_{\mathbb{F}}$)**.**

```
double eps = 1.0;
while (1.0+eps > 1.0) eps = eps/2;
```

2 Fehleranalyse

2.1 Rechnerarithmetik

Wegen der Endlichkeit der Maschinenzahlen entstehen Fehler nicht nur bei der Eingabe von Daten, sondern auch bei den elementaren Rechenoperationen $+, -, *, /$. Man hat auf dem Rechner nur eine Pseudoarithmetik, d.h. die Menge $\mathbb{F}$ ist nicht abgeschlossen bezüglich dieser Operationen.

Beispiel 2.1. Betrachte Gleitkommazahlen $\mathbb{F}$ mit Mantissenlänge $t = 5$ und Basis $b = 10$. Mit $x = 2.5684$ und $y = 3.2791 \cdot 10^{-3}$ gilt, dass

$$\begin{aligned} x + y = 2.5715791 \notin \mathbb{F} \quad & \mathrm{rd}(x + y) = 2.5714, \\ x * y = 0.008422044044 \notin \mathbb{F} \quad & \mathrm{rd}(x * y) = 8.4220 \cdot 10^{-3}, \\ x/y = 783.2637004 \notin \mathbb{F}, \quad & \mathrm{rd}(x/y) = 7.8326 \cdot 10^{2}. \end{aligned}$$

Obiges Beispiel verdeutlicht, dass für $x, y \in \mathbb{F}$ nicht notwendigerweise $x \circ y \in \mathbb{F}$ gilt, wobei $\circ \in \{+, -, *, /\}$. Stattdessen wird auf dem Rechner eine Ersatzoperation $x \circledcirc y \in \mathbb{F}$ ausgeführt. Im Folgenden nehmen wir an, dass

$$x \circledcirc y = \mathrm{rd}(x \circ y) \quad \text{für alle } x, y \in \mathbb{F}. \tag{2.1}$$

Dies bedeutet, dass $x \circledcirc y$ denselben Wert hat, als würde man $x \circ y$ zunächst exakt berechnen und das Ergebnis dann runden.
Annahme (2.1) wird z.B. vom IEEE-Standard 754 gefordert. Hardwaremäßig wird dafür mit einer längeren Mantisse gerechnet und das Ergebnis dann gerundet. Unter der Annahme (2.1) gilt im Fall $x_{\min} \leq |x|, |y|, |x \circ y| \leq x_{\max}$ für den relativen Fehler

$$\left| \frac{x \circledcirc y - x \circ y}{x \circ y} \right| = \left| \frac{\mathrm{rd}(x \circ y) - x \circ y}{x \circ y} \right| \leq \varepsilon_{\mathbb{F}}.$$

Das Kommutativgesetz für Addition und Multiplikation gilt auch in der Rechnerarithmetik, Distributivgesetz- und Assoziativgesetz im Allgemeinen jedoch nicht.

Beispiel 2.2. Betrachte $\mathbb{F}$ mit $b = 10$, $t = 2$. Für $x = 4.1$, $y = 8.2 \cdot 10^{-1}$ und $z = 1,4 \cdot 10^{-1}$ hat man

$$\begin{aligned} (x \oplus y) \oplus z = 4.9 \oplus 0.14 = 5.0, \quad & x \oplus (y \oplus z) = 4.1 \oplus 0.96 = 5.1, \\ x \circledast (y \oplus z) = 4.1 \circledast 0.96 = 3.9, \quad & x \circledast y \oplus x \circledast z = 3.4 \oplus 0.57 = 4.0 \end{aligned}$$

Mathematisch äquivalente Ausdrücke können beim Ausführen auf dem Rechner zu wesentlich unterschiedlichen Ergebnissen führen, selbst dann, wenn die Eingangsdaten Maschinenzahlen sind.

2.2 Fehlerfortpflanzung

Bei der Fehlerfortpflanzung untersucht man, wie sich Fehler in den Eingabedaten im Laufe einer Berechnung verstärken. Man untersucht, welcher Fehler bei exakter Auswertung der fehlerhaften Eingabedaten auftritt.

Lemma 2.3. *Es seien $\tilde{x}$, $\tilde{y}$ die mit relativen Fehlern*

$$\varepsilon_x := \frac{x - \tilde{x}}{x}, \quad \varepsilon_y := \frac{y - \tilde{y}}{y}$$

behafteten Werten für x bzw. y. Dann gilt für den relativen Fehler

$$\varepsilon_\circ := \frac{(x \circ y) - (\tilde{x} \circ \tilde{y})}{x \circ y}, \quad \circ \in \{+, -, *, /\},$$

dass

$$\begin{aligned}
\varepsilon_\pm &= \varepsilon_x \frac{x}{x \pm y} \pm \varepsilon_y \frac{y}{x \pm y}, \\
\varepsilon_* &= \varepsilon_x + \varepsilon_y - \varepsilon_x \varepsilon_y, \\
\varepsilon_/ &= \varepsilon_x - \varepsilon_y + \frac{\varepsilon_y}{1 - \varepsilon_y}(\varepsilon_x - \varepsilon_y).
\end{aligned}$$

Beweis. in den Übungsaufgaben. □

Aus Lemma 2.3 erkennt man, dass sich die Fehler bei Multiplikation und Division im Wesentlichen addieren bzw. subtrahieren. Bei der Addition und der Subtraktion kann der relative Fehler jedoch sehr verstärkt werden, falls $|x \pm y|$ sehr klein ist im Vergleich zu $|x|$ und $|y|$. Diesen Effekt bezeichnet man als **Auslöschung**. Es gilt, ihn soweit wie möglich zu vermeiden.

Beispiel 2.4. Betrachte die quadratische Gleichung $x^2 + 2px + q = 0$. Die Lösungen dieser Gleichung sind $x_{1,2} = -p \pm \sqrt{p^2 - q}$. Diese lassen sich gemäß

```
d  = sqrt(p*p-q);
x1 = -p-d;
x2 = -p+d;
```

berechnen. Für $p = 100$ und $q = 1$ bei dreistelliger dezimaler Rechnerarithmetik ergibt sich

$$d = \sqrt{p^2 - d} = \sqrt{10000 - 1} = \sqrt{9999} = 100$$

und somit $x_1 = -p - d = -200$, $x_2 = -p + d = 0$.
Richtig wäre aber $x_1 = -200$ und $x_2 = -0.005$, wenn das Ergebnis mit drei Stellen angegeben werden soll. Weil die Maschinengenauigkeit $\varepsilon_{\mathbb{F}} = \frac{1}{2} \cdot 10^{1-3} = 0.005$ beträgt, ist das Ergebnis $x_2 = 0$ schlicht falsch.

Dieser Auslöschungseffekt bei der Berechnung von x_2 kann durch Änderung der Art und Weise, wie x_2 berechnet wird, vermieden werden. Wegen

$$x^2 - (x_1 + x_2)x + x_1 \cdot x_2 = (x - x_1)(x - x_2) = x^2 + 2px + q \quad \text{für alle } x \in \mathbb{R}$$

folgt durch Koeffizientenvergleich $x_1 + x_2 = -2p$ und $x_1 \cdot x_2 = q$, was als **Wurzelsatz von Vieta** bekannt ist.
Berechnet man nur die betragsmäßig größte Nullstelle und die andere mittels $x_1 \cdot x_2 = q$, so erhält man den folgenden Algorithmus:

```
d = sqrt(p*p-q);
if (p >= 0) x1 = -p -d;
else x1 = d-p;
x2 = q/x1;
```

Hier ergibt sich $d = 100$, $x_1 = -p - d = -200$, $x_2 = 1/-200 = -0.005$.

Bei der Fehleranalyse unterscheidet man drei Arten von Fehlern:

1. *Datenfehler:* Um ein Rechenverfahren zu starten, benötigt man Eingabedaten. Diese sind im Allgemeinen ungenau (z.B. Messdaten) oder lassen sich nicht exakt im Rechner darstellen.

2. *Rundungsfehler:* Wegen der begrenzten Rechengenauigkeit sind alle Zwischenergebnisse gerundet.

3. *Verfahrensfehler:* Viele numerische Verfahren liefern selbst bei einer exakten Arithmetik nach endlich vielen Schritten keine exakte Lösung, sondern nähern sich einer solchen nur beliebig genau an. Beispielsweise kann mit Hilfe der Iteration

$$x_{i+1} := \frac{1}{2}\left(x_i + \frac{a}{x_i}\right), \quad i = 0, 1, 2, \ldots,$$

mit Startwert $x_0 = 1$ die Quadratwurzel aus $a > 0$ berechnet werden. Kein x_i stimmt mit $\sqrt{a}$ überein, nur die Differenz $|x_i - \sqrt{a}|$ wird für wachsende i beliebig klein. Da man nur endlich viele Schritte ausführen kann, entsteht ein Fehler. Dieser wird als Verfahrensfehler bezeichnet.

Definition 2.5. *Bei der **Vorwärtsanalyse** untersucht man, wie sich der relative Fehler im Verlaufe einer Rechnung akkumuliert. Bei der **Rückwärtsanalyse** wird jeder Rechenschritt als exakt bei gestörten Daten interpretiert, und es wird abgeschätzt, wie groß die Eingangsdatenstörung ist.*

Beispiel 2.6. Wir untersuchen die Addition zweier Zahlen.
Vorwärtsanalyse: $x \oplus y = \mathrm{rd}(x + y) = (x + y)(1 + \varepsilon)$ mit $|\varepsilon| \leq \varepsilon_{\mathbb{F}}$,
Rückwärtsanalyse: $x \oplus y = x(1 + \varepsilon) + y(1 + \varepsilon) = \tilde{x} + \tilde{y}$ mit $|\varepsilon| \leq \varepsilon_{\mathbb{F}}$.

Beispiel 2.7. Berechnung $\sum_{i=1}^{n} a_i b_i$ für $a_i, b_i \in \mathbb{R}$

```
double s = 0.0;
for (i=0; i<n; ++i) s=s+a[i]*b[i];
```

Sei $t_i := a_i \circledast b_i = a_i * b_i(1 + \varepsilon_i)$ und $s_i := s_{i-1} \oplus t_i = (s_{i-1} + t_i)(1 + \gamma_i)$, $s_0 = 0$, mit $|\varepsilon_i|, |\gamma_i| \leq \varepsilon_{\mathbb{F}}$. Hierbei ist zu berücksichtigen, dass wegen $s_0 = 0$ die erste Addition exakt

durchgeführt werden kann, d.h. wir haben $\gamma_1 = 0$. Dann gilt

$$\begin{aligned} s_1 &= a_1 \cdot b_1(1+\varepsilon_1) \\ s_2 &= [a_1 \cdot b_1 \cdot (1+\varepsilon_1) + a_2 \cdot b_2(1+\varepsilon_2)](1+\gamma_2) \\ &= a_1 \cdot b_1(1+\varepsilon_1)(1+\gamma_2) + a_2 \cdot b_2(1+\varepsilon_2)(1+\gamma_2) \\ &\vdots \\ s_n &= \sum_{i=1}^{n} a_i \underbrace{b_i(1+\varepsilon_i) \prod_{j=i}^{n}(1+\gamma_j)}_{\tilde{b}_i}. \end{aligned}$$

Bei der Rückwärtsanalyse werden die Fehler als gestörte Eingabedaten interpretiert: $\tilde{b_i} = b_i(1+\delta_i)$. Hieraus folgt

$$1 + \delta_i = (1+\varepsilon_i) \prod_{j=i}^{n}(1+\gamma_j)$$

und somit

$$(1-\varepsilon_{\mathbb{F}})^{n-i+2} \leq 1 + \delta_i \leq (1+\varepsilon_{\mathbb{F}})^{n-i+2}.$$

Mit der Bernoullischen Ungleichung $(1-x)^n \geq 1 - nx$, $x \leq 1$, erhält man

$$\begin{aligned} \delta_i &\geq (1-\varepsilon_{\mathbb{F}})^{n-i+2} - 1 \geq -(n-i+2)\varepsilon_{\mathbb{F}}, \\ \delta_i &\leq (1+\varepsilon_{\mathbb{F}})^{n-i+2} - 1 = \frac{(1-\varepsilon_{\mathbb{F}}^2)^{n-i+2}}{(1-\varepsilon_{\mathbb{F}})^{n-i+2}} - 1 \\ &\leq \frac{1}{1-(n-i+2)\varepsilon_{\mathbb{F}}} - 1 = \frac{(n-i+2)\varepsilon_{\mathbb{F}}}{1-(n-i+2)\varepsilon_{\mathbb{F}}}. \end{aligned}$$

Hieraus folgt

$$|\delta_i| \leq \frac{(n-i+2)\varepsilon_{\mathbb{F}}}{1-(n-i+2)\varepsilon_{\mathbb{F}}}$$

und

$$s_n = \sum_{i=1}^{n} a_i b_i (1+\delta_i).$$

2.3 Kondition und Stabilität

Ein zu lösendes Problem ist dadurch gekennzeichnet, dass zu gegebenen Daten x ein Ergebnis y zu berechnen ist, wobei der Zusammenhang zwischen x und y durch

$$y = f(x) \tag{2.2}$$

beschrieben wird.

Definition 2.8. *Ein Problem (2.2) heißt* ***wohlgestellt****, wenn es zu jedem x genau eine Lösung y gibt und die Zuordnung $x \mapsto y$ stetig ist.*

Gegeben sei eine differenzierbare Funktion $f : \mathbb{R} \to \mathbb{R}$. Für fehlerbehaftete Daten $\tilde{x} = x + \Delta x$ gilt

$$\frac{f(x + \Delta x) - f(x)}{\Delta x} \approx f'(x)$$

Folglich gilt für den Datenfehler $\Delta y := f(\tilde{x}) - f(x)$

$$\frac{\Delta y}{y} = \frac{f'(x)\Delta x}{f(x)} = \frac{f'(x) \cdot x}{f(x)} \frac{\Delta x}{x}.$$

Definition 2.9. *Für $f(x) \neq 0$ definiert man die **Konditionszahl** von f an der Stelle x als*

$$\kappa(f, x) := \left| \frac{f'(x) \cdot x}{f(x)} \right|$$

*Ein Problem heißt **schlecht konditioniert**, falls die Konditionszahl deutlich größer ist als 1, andernfalls heißt es **gut konditioniert**.*

Beispiel 2.10.

- $f(x) = x + a$. Dann ist die Konditionszahl $\frac{x}{x+a}$. Das Problem ist schlecht konditioniert, falls $|x + a|$ klein ist im Vergleich zu x (Auslöschung).
- $f(x) = x \cdot a$. Dann ist $\kappa(f, x) = |\frac{a \cdot x}{x \cdot a}| = 1$. Folglich ist die Multiplikation immer gut konditioniert.
- $f(x) = \sqrt{x}$. Dann ist $\kappa(f, x) = \frac{x \cdot \frac{1}{2} x^{-1/2}}{\sqrt{x}} = \frac{1}{2}$. Somit ist auch das Radizieren stets gut konditioniert.

Definition 2.11. *Erfüllt ein Algorithmus $\tilde{f}$ zur Lösung eines Problems f*

$$\left| \frac{\tilde{f}(x) - f(x)}{f(x)} \right| \leq c_V \cdot \kappa(f, x) \cdot \varepsilon_{\mathbb{F}}$$

*mit einer mäßig großen Konstanten $c_V > 0$, so wird der Algorithmus $\tilde{f}$ als **vorwärtsstabil** bezeichnet. Ergibt die Rückwärtsanalyse $\tilde{f}(x) = f(x + \Delta x)$ mit*

$$\left| \frac{\Delta x}{x} \right| \leq c_R \cdot \varepsilon_{\mathbb{F}}$$

*und ist $c_R > 0$ nicht zu groß, so ist $\tilde{f}$ **rückwärtsstabil**.*

Bemerkung.

- Die Kondition ist eine Eigenschaft des Problems, wohingegen die Stabilität eine Eigenschaft eines Algorithmus ist. Ist ein Problem schlecht konditioniert, so kann kein Algorithmus verhindern, dass sich Eingabefehler vergrößern. Ein gut konditioniertes Problem mit einem stabilen Algorithmus liefert gute numerische Ergebnisse.

- Rückwärtsstabile Algorithmen zur Lösung von gut konditionierten Problemen sind auch vorwärtsstabil, denn es gilt

$$\frac{|\tilde{f}(x) - f(x)|}{|f(x)|} = \frac{|f(x + \Delta x) - f(x)|}{|f(x)|} \leq \kappa \frac{|\Delta x|}{|x|} \leq \kappa \cdot c_R \cdot \varepsilon_{\mathbb{F}}.$$

Beispiel 2.12. Wir kommen zurück zu Beispiel 2.4. Für $f(x) = -p + \sqrt{p^2 - x}$ mit $|x| \ll p$ gilt

$$\begin{aligned}
\kappa(f, x) &= \left| \frac{f'(x) \cdot x}{f(x)} \right| = \left| \frac{x}{2\sqrt{p^2 - x}(p - \sqrt{p^2 - x})} \right| \\
&= \frac{1}{2} \left| \frac{x(p + \sqrt{p^2 - x})}{\sqrt{p^2 - x}(p + \sqrt{p^2 - x})(p - \sqrt{p^2 - x})} \right| \\
&= \frac{1}{2} \left| \frac{p + \sqrt{p^2 - x}}{\sqrt{p^2 - x}} \right| \approx 1
\end{aligned}$$

Die Nullstellenberechnung ist also gut konditioniert. Folglich ist der erste Algorithmus aus Beispiel 2.4 instabil.

3 Dreitermrekursion

3.1 Theoretische Grundlagen

Definition 3.1. *Eine Rekursion der Form*

$$p_k = a_k p_{k-1} + b_k p_{k-2} + c_k, \quad k = 2, 3, \ldots \tag{3.1}$$

mit $a_k, b_k, c_k \in \mathbb{R}$ heißt ***Dreitermrekursion****. Gilt $c_k = 0$ für alle k, so heißt die Dreitermrekursion* ***homogen****, sonst* ***inhomogen****. Gilt $b_k \neq 0$ für alle k, so wird*

$$p_{k-2} = -\frac{a_k}{b_k} p_{k-1} + \frac{1}{b_k} p_k - \frac{c_k}{b_k} \tag{3.2}$$

als ***Rückwärtsrekursion*** *bezeichnet. Geht (3.2) aus (3.1) durch Vertauschen von p_k und p_{k-2} hervor, d.h. gilt $b_k = -1$, so heißt die Rekursion* ***symmetrisch****.*

Beispiel 3.2.

(a) Sei $p_k = a_k p_{k-1} + b_k p_{k-2}$, $a_k = 2\cos(x)$, $b_k = -1$, $k = 2, 3, \ldots$. Dies liefert mit $p_0 = 1$, $p_1(x) = \cos(x)$, dass

$$p_k(x) = \cos(k \cdot x), \quad k = 2, 3, \ldots.$$

Dies sieht man mit den trigonometrischen Beziehungen

$$\cos(x \pm y) = (\cos x)(\cos y) \mp (\sin x)(\sin y).$$

(b) Die Rekursionsformel für die **Tschebycheff-Polynome** lautet

$$T_k(x) = 2xT_{k-1}(x) - T_{k-2}(x), \quad k = 2, 3, \ldots$$

mit Startwerten $T_0(x) = 1$, $T_1(x) = x$. Diese Rekursionsformel ist vom Typ (a), falls man $x = \cos\theta$ einsetzt. Daher gilt $T_k(\cos\theta) = \cos(k\theta) \Leftrightarrow T_k(x) = \cos(k \arccos(x))$.

(c) Die **Fibonacci-Zahlen** sind rekursiv definiert durch $f_k = f_{k-1} + f_{k-2}$, $k = 2, 3 \ldots$ mit Startwerten $f_0 = 0$, $f_1 = 1$.

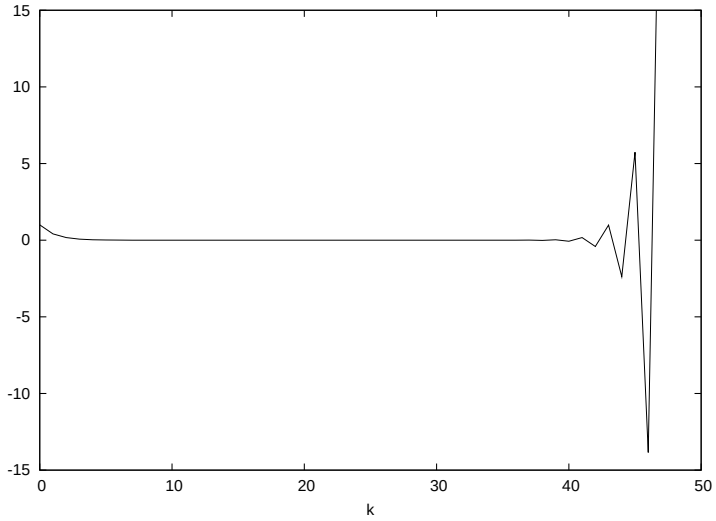
Algorithmus 3.3. (Dreitermrekursion)

```
Input: unsigned int n;
       double a[n-2],b[n-2],c[n-2],p0,p1; // Achtung: Indexshift
Output: double p[n];
p[0] = p0;
p[1] = p1;
for (k=2; k<n; k++) p[k]=a[k-2]*p[k-1]+b[k-2]*p[k-2]+c[k-2];
```

Beispiel 3.4. Im Fall der Dreitermrekursion

$$p_k = -2p_{k-1} + p_{k-2}, \quad k = 2, 3, \dots \text{ mit } p_0 = 1,\ p_1 = \sqrt{2} - 1$$

liefert Algorithmus 3.3 das folgende Ergebnis.



Wie wir im Folgenden sehen werden, handelt es sich hierbei um eine Instabilität. Um zu verstehen, warum diese auftritt, müssen wir Dreitermrekursionen zunächst etwas analysieren. Dazu benötigen wir den Begriff des *Eigenwerts* einer Matrix $A \in \mathbb{R}^{n\times n}$. Wir definieren das **charakteristische Polynom**

$$\chi(\lambda) := \det(A - \lambda I),$$

wobei $I \in \mathbb{R}^{n\times n}$ die **Einheitsmatrix**, d.h. die Matrix mit den Einträgen

$$I_{ij} = \begin{cases} 1, & i = j, \\ 0, & \text{sonst}, \end{cases}$$

bezeichnet. Jede Nullstelle $\lambda \in \mathbb{C}$ von χ heißt **Eigenwert** von A. Der Name Eigenwert rührt aus der Tatsache her, dass die Bedingung $\det(A - \lambda I) = 0$ äquivalent zur Existenz eines $0 \neq x \in \mathbb{R}^n$ mit $Ax = \lambda x$ ist. x heißt **Eigenvektor**.

Die homogene Dreitermrekursion

$$p_k = a_k p_{k-1} + b_k p_{k-2}, \quad k = 2, 3, \dots$$

kann umgeschrieben werden gemäß

$$\begin{bmatrix} p_k \\ p_{k-1} \end{bmatrix} = \begin{bmatrix} a_k & b_k \\ 1 & 0 \end{bmatrix} \begin{bmatrix} p_{k-1} \\ p_{k-2} \end{bmatrix} = A_k \begin{bmatrix} p_{k-1} \\ p_{k-2} \end{bmatrix}, \quad k = 2, 3, \dots.$$

Rekursiv folgt somit

$$\begin{bmatrix} p_k \\ p_{k-1} \end{bmatrix} = A_k A_{k-1} \dots A_2 \begin{bmatrix} p_1 \\ p_0 \end{bmatrix} = B_k \begin{bmatrix} p_1 \\ p_0 \end{bmatrix}$$

mit $B_k = A_k \cdot \dots \cdot A_2 \in \mathbb{R}^{2\times 2}$. Offensichtlich gilt für alle $\xi, \zeta \in \mathbb{R}$

$$B_k \left(\xi \begin{bmatrix} p_1 \\ p_0 \end{bmatrix} + \zeta \begin{bmatrix} q_1 \\ q_0 \end{bmatrix} \right) = \xi B_k \begin{bmatrix} p_1 \\ p_0 \end{bmatrix} + \zeta B_k \begin{bmatrix} q_1 \\ q_0 \end{bmatrix} = \xi \begin{bmatrix} p_k \\ p_{k-1} \end{bmatrix} + \zeta \begin{bmatrix} q_k \\ q_{k-1} \end{bmatrix}.$$

Die Folge $\{p_k\}$ hängt also linear von den Startwerten $\begin{bmatrix} p_1 \\ p_0 \end{bmatrix} \in \mathbb{R}^2$ ab.

Im Fall konstanter Koeffizienten $a_k = a$, $b_k = b$ folgt

$$B_k = A^{k-1}, \quad A = \begin{bmatrix} a & b \\ 1 & 0 \end{bmatrix}.$$

Beispiel 3.5. Wir bestimmen das charakteristische Polynom χ von A. Es gilt

$$\begin{aligned} \chi(\lambda) = \det(A - \lambda I) &= \det\left(\begin{bmatrix} a & b \\ 1 & 0 \end{bmatrix} - \lambda \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\right) = \det \begin{bmatrix} a-\lambda & b \\ 1 & -\lambda \end{bmatrix} \\ &= -\lambda(a-\lambda) - b = \lambda^2 - a\lambda - b \end{aligned}$$

Satz 3.6. *Seien λ_1, λ_2 die Nullstellen des charakteristischen Polynoms χ von A. Dann ist die Lösung der homogenen Dreitermrekursion*

$$p_k = ap_{k-1} + bp_{k-2}, \quad k = 2, 3, \ldots$$

gegeben durch

$$p_k = \alpha\lambda_1^k + \beta\lambda_2^k, \quad k = 0, 1, 2, \ldots$$

mit $\alpha, \beta \in \mathbb{R}$ Lösungen von $\alpha + \beta = p_0$ und $\alpha\lambda_1 + \beta\lambda_2 = p_1$.

Beweis. Wir führen den Beweis durch vollständige Induktion nach k. Für $k = 0, 1$ ist die Aussage trivial. Angenommen, die Behauptung gilt für alle natürlichen Zahlen bis zu einem $k \in \mathbb{N}$. Dann ist

$$\begin{aligned} p_{k+1} &= ap_k + bp_{k-1} \\ &= a(\alpha\lambda_1^k + \beta\lambda_2^k) + b(\alpha\lambda_1^{k-1} + \beta\lambda_2^{k-1}) \\ &= \alpha\lambda_1^{k-1}(a\lambda_1 + b) + \beta\lambda_2^{k-1}(a\lambda_2 + b) \end{aligned}$$

Wegen $\chi(\lambda_1) = 0$ und $\chi(\lambda_2) = 0$ folgt die Behauptung. □

Wenden wir nun den letzten Satz auf Beispiel 3.4 an. Das charakteristische Polynom $\chi(\lambda) = \lambda^2 + 2\lambda - 1$ hat die Nullstellen $\lambda_{1,2} = -1 \pm \sqrt{2}$. Aus $\alpha + \beta = 1$ und $\alpha\lambda_1 + \beta\lambda_2 = \sqrt{2} - 1$ erhält man $\alpha = 1, \beta = 0$. Nach Satz 3.6 ist die Lösung der Dreitermrekursion also

$$p_k = \lambda_1^k = (\sqrt{2} - 1)^k \ll 1.$$

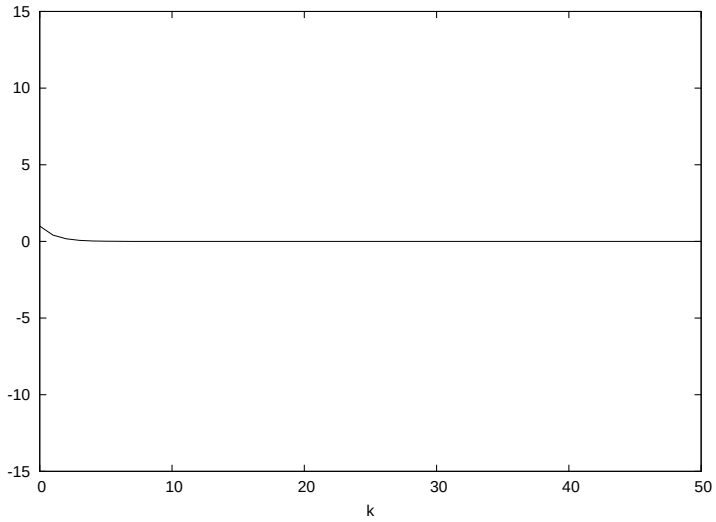
Dies erhält man auch bei folgendem Algorithmus:

Algorithmus 3.7. (geschlossene Lösung der Dreitermrekursion)

```
Input:  unsigned int n;
        double a,b,p0,p1;
Output: double p[n];
lambda1 = a/2 + sqrt(a*a/4+b);
lambda2 = a/2 - sqrt(a*a/4+b);
beta = (p1-lambda1*p0)/(lambda2-lambda1);
```

```
alpha = p0-beta;
p[0] = p0;
p[1] = p1;
l1 = lambda1;
l2 = lambda2;
for (k=2; k<n; ++k) {
   l1 *= lambda1;        // l1 = l1*lambda1;
   l2 *= lambda2;
   p[k] = alpha*l1 + beta*l2;
}
```

Im Fall der Werte aus Beispiel 3.4 liefert dieser Algorithmus das folgende Ergebnis.



Wir untersuchen die Kondition des Problems, um diese unterschiedlichen Ergebnisse zu erklären. Dazu betrachten wir $f : \mathbb{R} \to \mathbb{R}$ mit $f(p_0, p_1) = p_k$. Gestörte Daten

$$\tilde{p_0} = 1 + \varepsilon_0, \quad \tilde{p_1} = \lambda_1(1 + \varepsilon_1), \quad |\varepsilon_0|, |\varepsilon_1| \leq \varepsilon_{\mathbb{F}},$$

ergeben gestörte Koeffizienten

$$\begin{aligned} \tilde{\alpha} &= 1 + \varepsilon_0 \frac{\lambda_2}{\lambda_2 - \lambda_1} - \varepsilon_1 \frac{\lambda_1}{\lambda_2 - \lambda_1}, \\ \tilde{\beta} &= (\varepsilon_1 - \varepsilon_0) \frac{\lambda_1}{\lambda_2 - \lambda_1} \end{aligned}$$

und die gestörte Lösung

$$\tilde{p_k} = \left(1 + \varepsilon_0 \frac{\lambda_2}{\lambda_2 - \lambda_1} - \varepsilon_1 \frac{\lambda_1}{\lambda_2 - \lambda_1}\right) \lambda_1^k + (\varepsilon_1 - \varepsilon_0) \frac{\lambda_1}{\lambda_2 - \lambda_1} \lambda_2^k.$$

Folglich ergibt sich der relative Fehler

$$\begin{aligned} \left| \frac{\tilde{p_k} - p_k}{p_k} \right| &= \left| \varepsilon_0 \frac{\lambda_2}{\lambda_2 - \lambda_1} - \varepsilon_1 \frac{\lambda_1}{\lambda_2 - \lambda_1} + (\varepsilon_1 - \varepsilon_0) \frac{\lambda_1}{\lambda_2 - \lambda_1} \left(\frac{\lambda_2}{\lambda_1}\right)^k \right| \\ &= \left| \varepsilon_0 + (\varepsilon_1 - \varepsilon_0) \frac{\lambda_1}{\lambda_2 - \lambda_1} \left(\left(\frac{\lambda_2}{\lambda_1}\right)^k - 1 \right) \right|. \end{aligned}$$

Im Fall $|\lambda_2| > |\lambda_1|$ explodiert der relative Fehler. Genau dies passierte in Beispiel 3.4. Der Term λ_2^k wächst exponentiell, während die exakte Lösung exponentiell abfällt. Wie kann man dennoch so genannte Minimallösungen berechnen?

Definition 3.8. *Die Lösung $\{p_k\}$ der Dreitermrekursion (3.1) zu den Startwerten p_0, p_1 heißt **Minimallösung**, falls für jede Lösung $\{q_k\}$ zu den von p_0, p_1 linear unabhängigen Daten q_0, q_1 gilt*

$$\lim_{k\to\infty} \frac{p_k}{q_k} = 0.$$

*Die Lösung $\{q_k\}$ wird auch als **dominante Lösung** bezeichnet.*

Beispiel 3.9. In Beispiel 3.4 ist $\{p_k\}$ genau dann Minimallösung, wenn $\beta = 0$.

Es ist klar, dass die Minimallösung nur bis auf einen skalaren Faktor eindeutig bestimmt ist. Daher führen wir die Normierung $p_0^2 + p_1^2 = 1$ ein.

3.2 Miller-Algorithmus

Wir suchen die Minimallösung, d.h. diejenige Lösung, die zu dem kleineren Eigenwert korrespondiert. Betrachte die zu (3.1) gehörende Rückwärtsrekursion

$$q_{k-2} = -\frac{a}{b} q_{k-1} + \frac{1}{b} q_k, k = n, n-1, \ldots, 1$$

mit Startwerten $q_n = 0$, $q_{n-1} = 1$. Das charakteristische Polynom von

$$C = \begin{bmatrix} -\frac{a}{b} & \frac{1}{b} \\ 1 & 0 \end{bmatrix}$$

ist

$$\chi(\mu) = \det(C - \lambda I) = \det \begin{bmatrix} -\frac{a}{b} - \mu & \frac{1}{b} \\ 1 & -\mu \end{bmatrix} = \mu^2 + \frac{a}{b}\mu - \frac{1}{b}.$$

Daher besitzt C die Eigenwerte

$$\mu_{1,2} = \frac{-a \pm \sqrt{a^2 + 4b}}{2b} = \frac{1}{\lambda_{1,2}},$$

welche reziprok zu den Eigenwerten von A sind. Daher werden kleine Eigenwerte zu großen, gegen die dann die Rekursion stabil konvergiert. Aus $\alpha + \beta = 0$ und $\frac{\alpha}{\lambda_1} + \frac{1}{\lambda_2} = 1$ ergeben sich die Koeffizienten

$$\alpha = \frac{\lambda_1\lambda_2}{\lambda_2 - \lambda_1}, \quad \beta = -\frac{\lambda_1\lambda_2}{\lambda_2 - \lambda_1}.$$

Aufgrund der Lösungsdarstellung

$$q_k = \frac{\alpha}{\lambda_1^{n-k}} + \frac{\beta}{\lambda_2^{n-k}}$$

folgt im Fall $|\lambda_1| < |\lambda_2|$

$$\frac{q_k}{q_0} = \frac{\frac{\alpha}{\lambda_1^{n-k}} + \frac{\beta}{\lambda_2^{n-k}}}{\frac{\alpha}{\lambda_1^n} + \frac{\beta}{\lambda_2^n}} = \frac{\lambda_1^k + \frac{\beta}{\alpha}\lambda_2^k \frac{\lambda_1}{\lambda_2}^n}{1 + \frac{\beta}{\alpha}\frac{\lambda_1}{\lambda_2}^n} \xrightarrow{n\to\infty} \lambda_1^k.$$

Für $n = 100$ wird p_{50} aus Beispiel 3.4 auf 13 Nachkommastellen genau berechnet.

Algorithmus 3.10 (Miller-Algorithmus)**.**

(i) wähle n genügend größ.

(ii) Rückwärtsrekursion:

$$\hat{p}_{k-2} = -\frac{a}{b}\hat{p}_{k-1} + \frac{1}{b}\hat{p}_k, \quad k = n, n-1, \ldots, 2,$$

mit den Startwerten $\hat{p}_n = 0$, $\hat{p}_{n-1} = 1$.

(iii) Normierung: setze $p_k^{(n)} = \frac{\hat{p}_k}{\sqrt{\hat{p}_0^2+\hat{p}_1^2}}$, $k = 0, 1, 2, \ldots, n$.

Satz 3.11. *Sei $\{p_n\}$ eine Minimallösung der homogenen Dreitermrekursion mit $p_0^2 + p_1^2 = 1$. Dann gilt für die Lösung des Miller-Algorithmus*

$$\lim_{n\to\infty} p_k^{(n)} = p_k, \quad k = 0, 1, 2, \ldots, n.$$

Beweis. Sei $\{p_k\}$ eine normierte dominante Lösung mit zu $\left[p_1, p_0\right]^T$ linear unabhängigen Startvektoren $\left[q_1, q_0\right]^T$. Wegen

$$\begin{bmatrix} q_k \\ q_{k-1} \end{bmatrix} = A^{k-1} \begin{bmatrix} q_1 \\ q_0 \end{bmatrix} \quad \text{und} \quad \begin{bmatrix} p_k \\ p_{k-1} \end{bmatrix} = A^{k-1} \begin{bmatrix} p_1 \\ p_0 \end{bmatrix}$$

sind auch $\left[q_k, q_{k-1}\right]^T$, $\left[p_k, p_{k-1}\right]^T$ linear unabhängig, weil aus

$$0 = \alpha \begin{bmatrix} q_k \\ q_{k-1} \end{bmatrix} + \beta \begin{bmatrix} p_k \\ p_{k-1} \end{bmatrix} = A^{k-1} \left(\alpha \begin{bmatrix} q_1 \\ q_0 \end{bmatrix} + \beta \begin{bmatrix} p_1 \\ p_0 \end{bmatrix} \right)$$

folgt, dass

$$0 = \alpha \begin{bmatrix} q_1 \\ q_0 \end{bmatrix} + \beta \begin{bmatrix} p_1 \\ p_0 \end{bmatrix}$$

und hieraus $\alpha = \beta = 0$. Also können wir den Startvektor $\left[\hat{p}_n, \hat{p}_{n-1}\right]^T$ als Linearkombination von $\left[p_n, p_{n-1}\right]^T$ und $\left[q_n, q_{n-1}\right]^T$ darstellen:

$$\begin{bmatrix} \hat{p}_n \\ \hat{p}_{n-1} \end{bmatrix} = \xi \begin{bmatrix} p_n \\ p_{n-1} \end{bmatrix} + \zeta \begin{bmatrix} q_n \\ q_{n-1} \end{bmatrix} = \begin{bmatrix} p_n & q_n \\ p_{n-1} & q_{n-1} \end{bmatrix} \begin{bmatrix} \xi \\ \zeta \end{bmatrix}$$

und somit

$$\begin{aligned} \begin{bmatrix} \xi \\ \zeta \end{bmatrix} &= \begin{bmatrix} p_n & q_n \\ p_{n-1} & q_{n-1} \end{bmatrix}^{-1} \begin{bmatrix} \hat{p}_n \\ \hat{p}_{n-1} \end{bmatrix} \\ &= \frac{1}{p_n q_{n-1} - q_n p_{n-1}} \begin{bmatrix} q_{n-1} & -q_n \\ -p_{n-1} & p_n \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} \\ &= \frac{1}{p_n q_{n-1} - q_n p_{n-1}} \begin{bmatrix} -q_n \\ p_n \end{bmatrix} \end{aligned}$$

Wegen der Linearität folgt

$$\hat{p}_k = \xi p_k + \zeta q_k = \frac{q_n p_k - p_n q_k}{q_n p_{n-1} - p_n q_{n-1}} = \frac{q_n}{q_n p_{n-1} - p_n q_{n-1}} \left(p_k - \frac{p_n}{q_n} q_k \right).$$

Aus

$$\begin{aligned} \hat{p}_0^2 + \hat{p}_1^2 &= \frac{q_n^2}{(q_n p_{n-1} - p_n q_{n-1})^2} \left(p_0^2 + p_1^2 - 2\frac{p_n}{q_n}(p_0 q_0 + p_1 q_1) + (q_0^2 + q_1^2)\frac{p_n^2}{q_n^2} \right) \\ &= \frac{q_n^2}{(p_{n-1} q_n - q_{n-1} p_n)^2} \left(1 - 2\frac{p_n}{q_n} c + \frac{p_n^2}{q_n^2} \right) \end{aligned}$$

mit $c := p_0 q_0 + p_1 q_1$ erhalten wir

$$p_k^{(n)} = \frac{\hat{p}_k}{\sqrt{\hat{p}_0^2 + \hat{p}_1^2}} = \frac{p_k - \frac{p_n}{q_n} q_k}{\sqrt{1 - 2c\frac{p_n}{q_n} + \frac{p_n^2}{q_n^2}}} \xrightarrow{n \to \infty} p_k$$

wegen $p_n / q_n \to 0$ für $n \to \infty$. □

4 Sortieralgorithmen

Wir betrachten das folgende Sortierproblem: Gegeben seien n Zahlen $z_1, \ldots, z_n \in \mathbb{R}$. Gesucht ist eine Permutation π, so dass $z_{\pi_1} \leq z_{\pi_2} \leq \ldots \leq z_{\pi_n}$. Von Bedeutung ist die Sortierung, wenn z.B. Ausdrücke der Art $\sum_{i=1}^{n} z_i$ zu berechnen sind.

Definition 4.1. *Eine bijektive Abbildung $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ heißt **Permutation**. Wir schreiben $\pi(k) = \pi_k$, $k = 1, \ldots, n$.*

Satz 4.2. *Es gibt $n! = n \cdot (n-1) \cdot \ldots \cdot 2 \cdot 1$ Permutationen von $\{1, \ldots, n\}$.*

Beweis. Im Fall $n = 1$ ist die Behauptung klar. Angenommen, die Aussage gilt für ein $n \in \mathbb{N}$. Sei π eine Permutation über $\{1, \ldots, n+1\}$. Dann existiert ein $k \in \{1, \ldots, n+1\}$, so dass $\pi(k) = n + 1$. Die Abbildung

$$\pi^{(k)} : \{1, \ldots, k-1, k+1, \ldots, n+1\} \to \{1, \ldots, n\}$$

definiert durch $\pi^{(k)}(i) = \pi(i)$, $i \neq k$, ist eine Permutation über n Zahlen. Umgekehrt ist jede Abbildung dieser zusammengesetzten Form eine Permutation über $\{1, \ldots, n+1\}$. Dies ergibt $(n+1)n! = (n+1)!$ Permutationen. □

Ein naiver Algorithmus besteht darin, alle möglichen Permutationen zu testen, ob sie die gewünschte Sortierung garantieren. Für jede Permutation sind $n - 1$ Vergleiche durchzuführen. Also ergeben sich im ungünstigsten Fall (engl. worst case) $(n-1)n!$ Operationen.

4.1 Bubblesort

Aus der Transitivität der Ordnungsrelation "$\leq$"

$$x \leq y,\ y \leq z \ \Rightarrow\ x \leq z$$

ergibt sich folgende Idee, die eine bessere Effizienz verspricht.

Die Zahlenfolge wird in aufsteigender Reihenfolge durchlaufen. Dabei werden immer zwei benachbarte Elemente vertauscht, falls sie in falscher Reihenfolge stehen. Am Ende steht das größte Element am Ende der Folge und muss beim nächsten Durchlauf nicht mehr betrachtet werden. Der Name *bubblesort* rührt daher, dass die größeren Zahlen wie Blasen im Wasser aufsteigen.

Beispiel 4.3. Wir wollen die Zahlen $z_1 = 5,\ z_2 = 3,\ z_3 = 2$ aufsteigend sortieren.
Im 1. Durchlauf haben wir $(5, 3, 2) \to (3, 5, 2) \to (3, 2, \underline{5})$.
Im 2. Durchlauf ist $(3, 2, \underline{5}) \to (2, \underline{3}, \underline{5})$.

Die Anzahl der Vergleiche im $k = 1, \ldots, n-1$-ten Schritt ist $n - k$. Also ergeben sich insgesamt

$$\sum_{k=1}^{n-1}(n-k) = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2}$$

Vergleiche, was eine erhebliche Reduktion des Aufwandes gegenüber der naiven Vorgehensweise bedeutet.

Algorithmus 4.4 (Bubblesort)**.**

```
Input:  unsigned int n;
        double z[n];
Output: double z[n];                    // in aufsteigender Sortierung
for (k=1; k<n; ++k) {
  swapped = false;
  for (l=0; l<n-k; ++l) {
    if (z[l]>z[l+1]) {
      tmp = z[l];
      z[l] = z[l+1];
      z[l+1] = tmp;
      swapped = true;
    }
  }
  if (swapped == false) break;  // Folge ist bereits sortiert
}
```

Für die Komplexitätsanalyse ist die folgende Funktion nützlich:

Definition 4.5. *Seien $f, g : \mathbb{R}^+ \to \mathbb{R}^+$ zwei Funktionen. Dann schreibt man*

(i) $f = O(g)$*, "f ist groß O von g", falls es zwei Konstanten $x_0, c > 0$ gibt mit $f(x) \leq c \cdot g(x)$ für alle $x \geq x_0$,*

(ii) $f = o(g)$ *"f ist klein o von g", falls für alle $c > 0$ ein $x_0 \in \mathbb{R}^+$ existiert mit $f(x) \leq c \cdot g(x)$ für alle $x \geq x_0$,*

(iii) $f = \Theta(g)$*, falls $f = O(g)$ und $g = O(f)$.*

Beispiel 4.6. $f(x) = 2x + 25,\ g(x) = x^2 - 10$

(a) $f = O(g)$, weil für $c = 1$ und $x_0 = 7$ gilt $2x + 25 \leq 2x + (x-1)^2 - 11 = x^2 - 10$.

(b) $f = o(g)$, weil für alle $c > 0$ und alle $x \geq x_0 = \frac{1}{c} + \sqrt{10 + \frac{2}{5}c + \frac{1}{c^2}}$ gilt

$$c(x^2 - 10) = c(x - \frac{1}{c})^2 + 2x - \frac{1}{c} - 10c \geq 2x + 25.$$

(c) Die Komplexität von Bubblesort ist $O(n^2)$. Man beachte: Sie ist nicht $\Theta(n^2)$, weil nur $O(n)$ Operationen nötig sind, wenn die Folge bereits sortiert ist.

Satz 4.7. *Seien $f, g : \mathbb{R}^+ \to \mathbb{R}^+$ zwei Funktionen. Dann gilt*

(a) $f = o(g) \Leftrightarrow \lim_{x\to\infty} \frac{f(x)}{g(x)} = 0$.

(b) Gibt es ein $c > 0$ mit $\lim_{x\to\infty} \frac{f(x)}{g(x)} = c$, so gilt $f = \Theta(g)$.

Beweis.

zu (a): Es gilt

$$\begin{aligned} f = o(g) &\Leftrightarrow \forall c > 0\ \exists x_0 \in \mathbb{R}^+ \text{ mit } f(x) \le c \cdot g(x) \\ &\Leftrightarrow \forall x > 0\ \exists x_0 \in \mathbb{R}^+ \text{ mit } \frac{f(x)}{g(x)} \le c\ \forall x \ge x_0 \\ &\Leftrightarrow \lim_{x\to\infty} \frac{f(x)}{g(x)} = 0. \end{aligned}$$

zu (b): Wir haben

$$\begin{aligned} \lim_{x\to\infty} \frac{f(x)}{g(x)} = c &\Leftrightarrow \forall \varepsilon > 0\ \exists x_0 \text{ mit } |\frac{f(x)}{g(x)} - c| < \varepsilon\ \forall x \ge x_0 \\ &\Leftrightarrow \forall \varepsilon > 0\ \exists x_0 \text{ mit } c - \varepsilon < \frac{f(x)}{g(x)} < c + \varepsilon\ \forall x \ge x_0. \end{aligned}$$

Insbesondere existiert hat man für die Wahl $\varepsilon = \frac{c}{2}$

$$\left.\begin{array}{ll} f(x) \le \frac{3}{2}c \cdot g(x) & \Rightarrow f = O(g) \\ g(x) \le \frac{2}{c} f(x) & \Rightarrow g = O(f) \end{array}\right\} \Rightarrow f = \Theta(g).$$

□

4.2 Mergesort

Mergesort ist ein weiteres Sortierverfahren, das gegenüber Bubblesort den Vorteil besitzt, dass der Aufwand durch $O(n \log n)$ beschränkt ist, um n Zahlen zu sortieren. Die Steigerung der Effizienz beruht auf folgender Beobachtung. Gegeben seien die sortierten Menge

$$\begin{aligned} M_x &= \{x_1, \ldots, x_m,\ x_i \le x_j,\ i < j\}, \\ M_y &= \{y_1, \ldots, y_m,\ y_i \le y_j,\ i < j\} \end{aligned}$$

Dann lässt sich die Menge $M := M_x \cup M_y$ mit $m + n - 1$ Vergleichen sortieren, indem man das jeweils kleinste Element der beiden Mengen entfernt und M hinzufügt.

Beispiel 4.8. Es seien $M_x = \{1, 5, 9\}$ und $M_y = \{3, 7, 8\}$.

M_x	M_y	M
1, 5, 9	3, 7, 8	
5, 9	3, 7,8	1
5, 9	7, 8	1, 3
9	7, 8	1, 3, 5
9	8	1, 3, 5, 7
9		1, 3, 5, 7, 8
		1, 3, 5, 7, 8, 9

Dieses Zusammenfügen (engl. merge) bildet die Grundlage zu folgendem divide-and-conquer-Algorithmus:

Algorithmus 4.9 (Mergesort)**.**

(i) Ist $n = 1$, so gibt es nichts zu tun.
Andernfalls sortiere $M_x = \{z_1, \ldots, z_{\lfloor n/2 \rfloor}\}$ und $M_x = \{z_{\lfloor n/2 \rfloor}, \ldots, z_n\}$ mit Algorithmus 4.9.

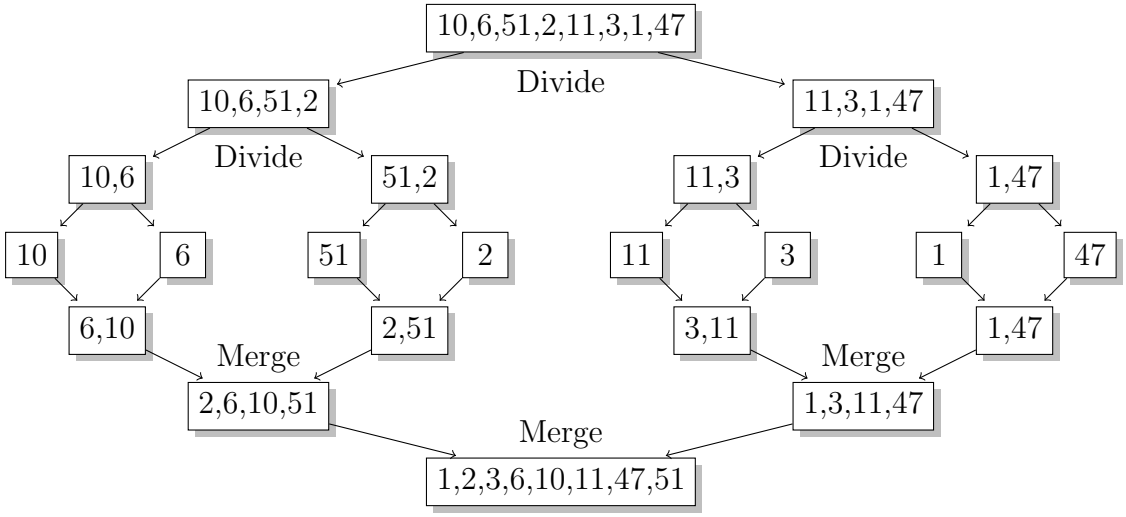
(ii) Füge die sortierten Mengen zu einer Menge zusammen.

Bemerkung.

(a) Man beachte, dass Mergesort sich selbst aufruft. Man spricht von einer **Rekursion** bzw. von einem rekursiven Algorithmus.

(b) Divide-and-conquer-Algorithmen verfahren immer nach folgendem Prinzip: Teile das Ausgangsproblem in etwa gleich große Teile, löse diese dann separat und füge die Lösungen zur Gesamtlösung zusammen.

Beispiel 4.10.



Eine mögliche Implementierung von Mergesort ist:

Algorithmus 4.11. `void merge(z,nbeg,nend,nsep)`

```
// vereinigt z[nbeg...nsep] und z[nsep+1...nend]
double *z;
unsigned int nbeg, nend, nsep;
{
  double tmp[nend-nbeg+1];
  unsigned int i = nbeg, j = nsep+1, k = 0;
  while ((i<=nsep) && (j<=nend)) {
    if (z[i]<=z[j]) tmp[k++] = z[i++];
    else tmp[k++] = z[j++];
  }
  while (i<=nsep) tmp[k++] = z[i++];        // Reste anhaengen
  while (j<=nend) tmp[k++] = z[j++];
  for (k=nbeg; k<=nend; ++k) z[k] = tmp[k-nbeg];
}

void mergesort(z, nbeg, nend)          // sortiert z[nbeg...nend]
{
  if (nbeg<nend) {
    unsigned int nsep = (nbeg+nend)/2;
    mergesort(z, nbeg, nsep);          // rekursiver Aufruf
    mergesort(z, nsep+1, nend);        // rekursiver Aufruf
    merge(z, nbeg, nend, nsep);        // conquer: merge
  }
}
```

Der Aufruf erfolgt mittels `mergesort(z, 0, n-1);`

Satz 4.12. *Der Aufwand des Mergesort-Algorithmus ist* $O(n \log n)$.

Beweis. Es bezeichne $T(n)$ die Anzahl der Operationen von Mergesort für n Elemente. Angenommen, $n = 2^k$ für ein $k \in \mathbb{N}$. Dann gilt

$$T(n) \leq 2 \cdot T(n/2) + c \cdot n \tag{4.1}$$

mit $c \in \mathbb{N}$. Wir zeigen induktiv, dass $T(2^k) \leq 2^k T(1) + c \cdot k \cdot 2^k$ gilt.
Für $k = 1$ ist wegen (4.1) $T(2) \leq 2 \cdot T(1) + 2 \cdot c$. Angenommen, die Aussage gilt für ein $k \in \mathbb{N}$. Dann ist

$$\begin{aligned} T(2^{k+1}) &\leq 2 \cdot T(2^k) + c \cdot 2^{k+1} \leq 2(2^k T(1) + c \cdot k \cdot 2^k) + c \cdot 2^{k+1} \\ &= 2^{k+1} T(1) + c \cdot 2^{k+1}(k+1). \end{aligned}$$

Wegen $k = \log_2 n$ gilt somit $T(n) = O(n \log n)$.

Die Behauptung gilt auch für beliebiges $n \in \mathbb{N}$. Es existiert $k \in \mathbb{N}$ mit $2^k \leq n < 2^{k+1}$. Wegen $T(n) \leq T(2^{k+1}) = O(2^{k+1}(k+1)) = O(2^k k) \leq O(n \log_2 n)$ folgt die Behauptung. □

Gegenüber Bubblesort haben wir somit einen signifikanten Vorteil bzgl. des Aufwandes. Allgemein gilt für die Komplexität von divide-and-conquer-Algorithmen der folgende Satz.

Satz 4.13 (Master-Theorem). *Wir betrachten Funktionen $T : \mathbb{N} \to \mathbb{N}$ definiert durch*

$$T(n) = aT(n/b) + f(n)$$

mit einer Funktion $f : \mathbb{N} \to \mathbb{N}$ und Konstanten $a \geq 1$, $b > 1$. Sei $f(n) = O(n^p)$ mit $p \geq 0$. Dann gilt

$$T(n) = \begin{cases} O(n^p), & \text{falls } a < b^p, \\ O(n^p \log n), & \text{falls } a = b^p, \\ O(n^{\log_b(a)}), & \text{falls } a > b^p. \end{cases}$$

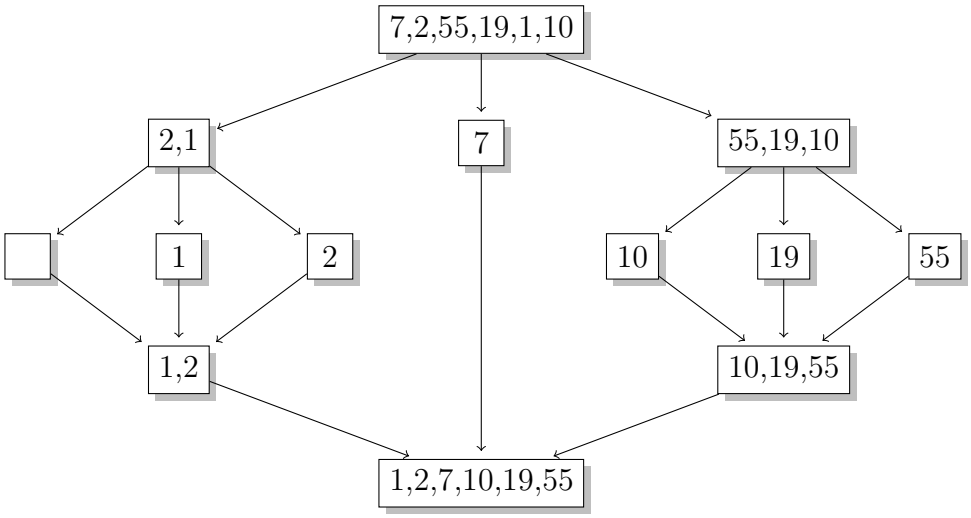
4.3 Quicksort

Quicksort ist ein weiteres Sortierverfahren. Es basiert ebenfalls auf dem divide-and-conquer-Prinzip, spart aber im Vergleich zu Mergesort den Merge-Schritt. Die Idee besteht darin, die Menge $Z := \{z_1, \ldots, z_n\}$ bzgl. eines Elementes $x \in Z$ in Untermengen

$$\{z \in Z : z \leq x\} \cup \{x\} \cup \{z \in Z : x \leq z\}$$

zu unterteilen, diese (rekursiv) zu sortieren und die sortierten Mengen zusammenzusetzen.

Beispiel 4.14.



Algorithmus 4.15 (Quicksort)**.**

```
unsigned partition(z, nbeg, nend)      // sortiert z[nbeg..nend]
double *z;  // liefert Position d. letzten Elementes <=x zurueck
unsigned nbeg, nend;
{
```

```
  double x = z[nend];   // x wird als letztes Element gewaehlt
  unsigned i = nbeg;    // kein Element vor i ist groesser als x
  unsigned j = nend-1;  // kein Element nach j ist kleiner als x
  do {
    while ((i<nend) && (z[i]<=x)) ++i;
    while ((j>nbeg) && (z[j]>=x)) --j;
    if (i<j) {
      double tmp = z[i];     // vertausche z[i] und z[j]
      z[i] = z[j];
      z[j] = tmp;
    }
  } while (i<j);
  z[nend] = z[i]; z[i] = x;      // vertausche x an Endposition
  return i;
}

void quicksort(z, nbeg, nend)
double *z;
unsigned nbeg, nend;
{
  if (nbeg<nend) {
    unsigned p = partition(z, nbeg, nend);
    quicksort(z, nbeg, p-1);
    quicksort(z, p+1, nend);
  }
}
```

Der Aufruf erfolgt mittels `quicksort(z, 0, n-1)`. Man beachte, dass Quicksort im Vergleich zu Mergesort keinen zusätzlichen Speicher benötigt, der so groß ist wie das Feld z. Man sagt, der Algorithmus läuft "in place".

Satz 4.16. *Der Quicksort-Algorithmus hat eine Worst-Case-Komplexität von $O(n^2)$.*

Beweis. Sei $T(n)$ die Anzahl der Operationen für n Elemente. Dann gilt

$$T(n) \leq c \cdot n + \max_{1 \leq k < n} [T(k) + T(n-k)].$$

Wir zeigen, dass $T(n) \leq c \cdot n^2$. Für $n = 1$ ist die Behauptung trivial. Angenommen, die Behauptung gilt für alle natürlichen Zahlen kleiner als $n \in \mathbb{N}$. Dann folgt wegen $\max_{1 \leq k < n}(k^2 + (n-k)^2) = 1 + (n-1)^2$

$$\begin{aligned} T(n) &\leq c \cdot n + \max_{1 \leq k < n} [T(k) + T(n-k)] \leq c \cdot n + \max_{1 \leq k < n} [c \cdot k^2 + c \cdot (n-k)^2] \\ &\leq c \cdot n + c + c \cdot (n-1)^2 = c \cdot n^2 - c \cdot n + 2c \leq c \cdot n^2. \end{aligned}$$

□

Bemerkung. Die Worst-Case-Komplexität wird angenommen, falls die Menge bereits sortiert ist.

Satz 4.17. *Der Aufwand von Quicksort ist im Durchschnitt $O(n \log n)$.*

Beweis. Sei P_n die Menge aller Permutationen über $\{1, \ldots, n\}$. Dann ist der mittlere Aufwand durch

$$\overline{T}(n) = \frac{1}{n!} \sum_{\pi \in P_n} T(\pi)$$

gegeben, wobei $T(\pi)$ den Aufwand von Quicksort für die Sortierung einer Permutation $\pi \in P_n$ bezeichnet. Wir teilen P_n in Mengen $P_n^{(k)}$, $k = 1, \ldots, n$, mit $P_n^{(k)} = \{\pi \in P_n : \pi_n = k\}$, $n \geq 2$. Weil das letzte Element aus $P_n^{(k)}$ fest ist, gilt $|P_n^{(k)}| = (n-1)!$, $k = 1, 2, \ldots, n$.

Für alle $\pi \in P_n^{(k)}$ ergibt der Teilungsschritt zwei Permutationen $\pi^{(1)}$ und $\pi^{(2)}$ über $\{1, \ldots, k-1\}$ bzw. $\{k+1, \ldots, n\}$. Somit folgt $T(\pi) = n - 1 + T(\pi^{(1)}) + T(\pi^{(2)})$ und somit

$$\sum_{\pi \in P_n^{(k)}} T(\pi) = |P_n^{(k)}|\,(n-1) + \sum_{\pi \in P_n^{(k)}} [T(\pi^{(1)}) + T(\pi^{(2)})].$$

Wenn π alle Permutationen aus $P_n^{(k)}$ durchläuft, entstehen für $\pi^{(1)}$ alle Permutationen von $\{1, \ldots, k-1\}$ und zwar jede genau $\frac{(n-1)!}{(k-1)!}$ -mal, weil $|P_n^{(k)}| = (n-1)!$. Also gilt

$$\sum_{\pi \in P_n^{(k)}} T(\pi^{(1)}) = \frac{(n-1)!}{(k-1)!} \sum_{\pi^{(1)} \in P_{k-1}} T(\pi^{(1)}) = (n-1)!\,\overline{T}(k-1)$$

und analog

$$\sum_{\pi \in P_n^{(k)}} T(\pi^{(2)}) = (n-1)!\,\overline{T}(k-1).$$

Als Folgerung erhält man die rekursive Abschätzung

$$\begin{aligned}
\overline{T}(n) &= \frac{1}{n!} \sum_{\pi \in P_n} T(\pi) = \frac{1}{n!} \sum_{k=1}^{n} \sum_{\pi \in P_n^{(k)}} T(\pi) \\
&= \frac{1}{n!} \sum_{k=1}^{n} (n-1)! \left(n - 1 + \overline{T}(k-1) + \overline{T}(n-k)\right) \\
&= n - 1 + \frac{1}{n} \sum_{k=1}^{n} [\overline{T}(k-1) + \overline{T}(n-k)] \\
&= n - 1 + \frac{2}{n} \sum_{k=0}^{n-1} \overline{T}(k).
\end{aligned}$$

Hieraus folgt insbesondere, dass

$$\begin{aligned}
\overline{T}(n) &= n - 1 - \frac{n-1}{n}(n-2) + \frac{n-1}{n}\,\overline{T}(n-1) + \frac{2}{n}\,\overline{T}(n-1) \\
&= \frac{2}{n}(n-1) + \frac{n+1}{n}\,\overline{T}(n-1)
\end{aligned}$$

mit $\overline{T}(0) = 1,\ \overline{T}(1) = 1$.

Wir zeigen induktiv, dass $\overline{T}(n) \leq 2(n+1)\sum_{i=1}^{n} \frac{1}{i}$. Für $n = 2$ gilt

$$\overline{T}(2) = 3 \leq 6(1 + \frac{1}{2}) = 2(n+1)\sum_{i=1}^{n} \frac{1}{i}.$$

Angenommen, die Aussage gilt für ein $n \in \mathbb{N}$. Dann folgt

$$\begin{aligned}\overline{T}(n+1) &= \frac{2}{n+1}n + \frac{n+2}{n+1}\overline{T}(n) \leq \frac{2}{n+1}n + 2(n+2)\sum_{i=1}^{n}\frac{1}{i} \\ &\leq 2\frac{n+2}{n+1} + 2(n+2)\sum_{i=1}^{n}\frac{1}{i} = 2(n+2)\sum_{i=1}^{n+1}\frac{1}{i}.\end{aligned}$$

Die Partialsumme $\sum_{i=1}^{n} \frac{1}{i}$ der harmonischen Reihe läßt sich durch ein Integral abschätzen. Die Rechtecke mit den Ecken $(i-1, 0)$ und $(i, 1/i)$, $i = 2, 3, \ldots$, sind Teil der Fläche, die der Graph der Funktion $1/x$, $x \geq 1$, mit der x-Achse einschliesst. Daher gilt

$$\sum_{i=1}^{n}\frac{1}{i} = 1 + \sum_{i=2}^{n}\frac{1}{i} \leq 1 + \int_1^n \frac{1}{x}dx = 1 + \log n,$$

woraus

$$\overline{T}(n) \leq 2(n+1)(1 + \log n) = O(n \log n)$$

folgt. □

4.4 Eine untere Schranke für das Sortierproblem

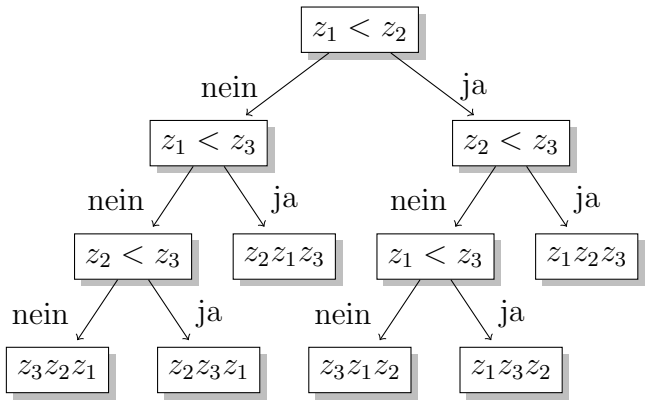
In diesem Abschnitt wollen wir untersuchen, welche Laufzeit ein Verfahren zur Sortierung von n unterschiedlichen Elementen mindestens haben muss.

Satz 4.18. *Jeder Algorithmus zur Sortierung von n Elementen, der die Information über die Elemente ausschließlich durch paarweise Vergleiche erhält, benötigt mindestens $\log n!$ viele Vergleiche.*

Für den Beweis betrachten wir den *Entscheidungsbaum*, der jedem Sortierverfahren mit paarweisen Vergleichen für festes n zugeordnet werden kann. Allgemein können wir den Entscheidungsbaum wie folgt beschreiben:

- Innere Knoten sind Vergleiche im Algorithmus
- Ein Weg von der Wurzel des Baumes zu einem Blatt entspricht der zeitlichen Abfolge der Vergleiche. Ein Weitergehen nach rechts entspricht einem richtigen Vergleich, nach links einem falschen.
- Die $n!$ Blätter des Baumes stellen die Permutationen der Eingabefolge dar, die jeweils zur Abfolge der Vergleiche gehört.

Beispiel 4.19. Der Entscheidungsbaum für $\{z_1, z_2, z_3\}$ ist wie folgt gegeben:



Beweis zu Satz 4.18. Wir müssen eine untere Schranke für die Höhe des Entscheidungsbaumes finden. Bestenfalls ist der Baum balanciert, d.h. alle Blätter liegen in der gleichen Ebene. Ein balancierter Baum hat also pro Ebene $1, 2, 4, 8, \ldots, 2^m$ Knoten. Da wir $n!$ Blätter haben, folgt $2^m = n!$ bzw.

$$\begin{aligned} m &= \log_2 n! \\ &\geq \log_2 \underbrace{\left(\frac{n}{2} \cdot \frac{n}{2} \cdot \ldots \cdot \frac{n}{2}\right)}_{\lceil \frac{n}{2} \rceil} \\ &\geq \log_2 \left(\left(\frac{n}{2}\right)^{\frac{n}{2}}\right) \\ &= \frac{n}{2}(\log_2 n - 1). \end{aligned}$$

□

Bemerkung. Die Funktion $f(n) = \log n!$ verhält sich im Wesentlichen wie $n \log n$, denn es gilt auch die Abschätzung

$$\log n! \leq \log n^n = n \log n.$$

Also folgt $\log n! = \Theta(n \log n)$. Daher ist Mergesort asymptotisch optimal. Allerdings benötigt Mergesort im Allgemeinen mehr als $\lceil \log n! \rceil$ viele Vergleiche.

4.5 Binäre Heaps und Heapsort

Für viele Zwecke ist es sinnvoll, Elemente einer Folge hinzuzufügen und Elemente aus einer Folge zu entfernen. Das Einfügen von Elementen in ein Array kann mit $O(1)$ Aufwand durchgeführt werden, während das Entfernen des kleinsten Elementes $O(n)$ Aufwand benötigt. Ist das Array sortiert, so benötigt man für das Einfügen $O(n)$ und für das Löschen $O(1)$.

Mit Hilfe von sog. **binären Heaps** können Einfügen und Entfernen mit jeweils $O(\log n)$ Aufwand durchgeführt werden.

Definition 4.20. *Eine Zahlenfolge $z_1, \ldots, z_n$ besitzt die **Heapeigenschaft**, falls gilt:*

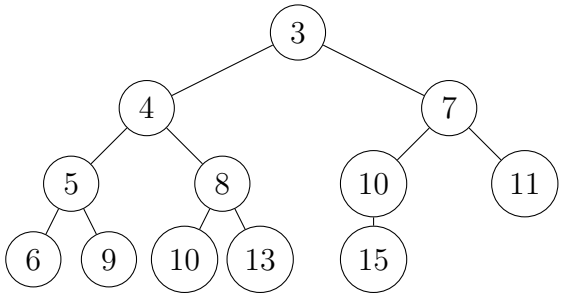
(i) $z_i \leq z_{2i}$, $i = 1, \ldots, \lfloor n/2 \rfloor$,

(ii) $z_i \leq z_{2i+1}$, $i = 1, \ldots, \lfloor (n-1)/2 \rfloor$.

Beispiel 4.21. Die folgende Zahlenfolge besitzt die Heapeigenschaft.

i	1	2	3	4	5	6	7	8	9	10	11	12
z_i	3	4	7	5	8	10	11	6	9	10	13	15

Indem man jeweils die Elemente z_{2i} und z_{2i+1} in einem Baum unter z_i anordnet, lässt sich die Heapeigenschaft leicht ablesen, da zu jedem Element die darunter liegenden mindestens genauso groß sein müssen.



Natürlich kann man Heaps auch als Bäume (d.h. mit Hilfe von Pointern) abspeichern. Dies kann allerdings die Datenlokalität beeinträchtigen, was sich negativ auf Cache-Effekte im Rechner auswirkt. Daher sollte die Speicherung in dem obigen Array mit Hilfe der sog. **Kekulé-Anordnung** bevorzugt werden.

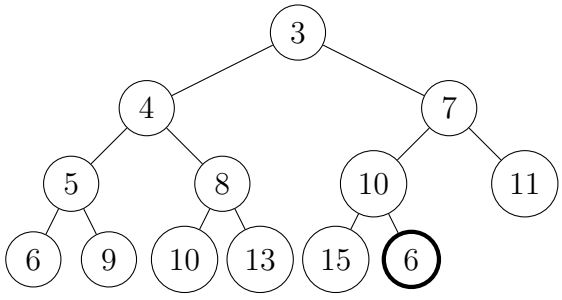
Lemma 4.22. *Falls die Folge $z_1, \ldots, z_n$ die Heapeigenschaft besitzt, so gilt $z_1 \leq z_i$, $i = 1, \ldots, n$.*

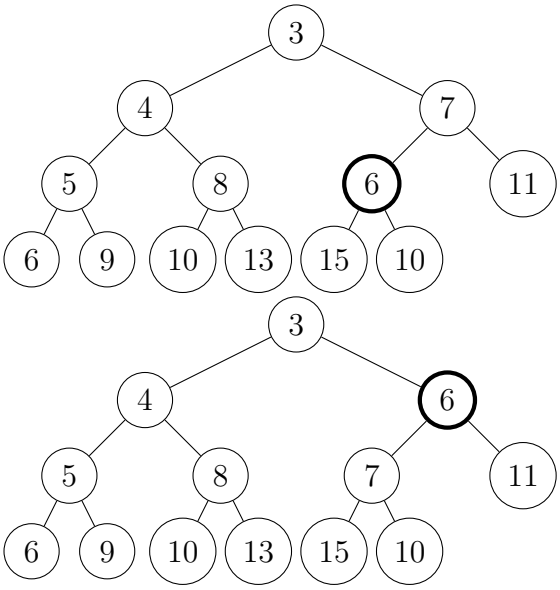
Beweis. Sei j der kleinste Index mit der Eigenschaft $z_j \leq z_i$, $i = 1, \ldots, n$. Ist $j = 1$, so ist nichts zu zeigen. Im anderen Fall ist $j > 1$. Dann gilt wegen der Heapeigenschaft, dass $z_{\lfloor j/2 \rfloor} \leq z_j$. Dies führt zum Widerspruch, weil j als kleinster Index definiert wurde. □

Ein kleinstes Element lässt sich daher bei einer Folge mit Heapeigenschaft in $O(1)$ finden. Im Folgenden zeigen wir, wie Elemente eingefügt bzw. entfernt werden können und man dabei die Heapeigenschaft erhält.

Zum Einfügen eines Elementes füge man dieses am Ende der Zahlenfolge als z_{n+1} an. Besitzt die neue Folge die Heapeigenschaft, so ist nichts zu tun. Andernfalls tauscht man das eingefügte Element sukzessive jeweils mit dem darüberstehenden Element.

Beispiel 4.23. Wir wollen die Zahl 6 in die Folge aus Beispiel 4.21 unter Erhaltung der Heapeigenschaft einfügen. Dann ergibt die beschriebene Vorgehensweise die folgende Folge von Bäumen.





Damit ist die Heapeigenschaft wiederhergestellt.

Algorithmus 4.24 (Einfügen in Heap)**.**

```
void Insert(x, z, heapsize)
int x, *z;
unsigned int heapsize;
{
  unsigned int i = heapsize;
  while (i>0 && z[(i-1)/2)]>x) {
    z[i] = z[(i-1)/2];
    i = (i-1)/2;
  }
  z[i] = x;
}
```

Die beiden folgenden Lemmata behandeln die Korrektheit des Algorithmus 4.24 (also die Frage, ob die Heapeigenschaft erhalten wird) und die Komplexität des Algorithmus.

Lemma 4.25. *Sei $z_1, \ldots, z_n$ eine Zahlenfolge, die die Heapeigenschaft besitzt, falls z_j durch $z_{\lfloor j/2 \rfloor} > z_j$ ersetzt wird. Durch Vertauschen von z_j mit $z_{\lfloor j/2 \rfloor}$ erhält man entweder eine Folge mit Heapeigenschaft oder es gilt $z_{\lfloor \lfloor j/2 \rfloor /2 \rfloor} > z_{\lfloor j/2 \rfloor}$ und man erhält eine Folge mit Heapeigenschaft, falls man $z_{\lfloor j/2 \rfloor}$ durch $z_{\lfloor \lfloor j/2 \rfloor /2 \rfloor}$ ersetzt.*

Beweis. Es bezeichne $\{z'_i\}$ die Folge, die durch Vertauschen von z_j mit $z_{\lfloor j/2 \rfloor}$ entsteht. Besitzt die Folge $\{z'_i\}$ die Heapeigenschaft, so ist nichts weiter zu tun. Andernfalls gilt nach Vorraussetzung, dass

(i) $z_{\lfloor \lfloor j/2 \rfloor /2 \rfloor} \leq \min\{z_{\lfloor j/2 \rfloor}, z_{j_1}\}$, $j_1 := \begin{cases} \lfloor j/2 \rfloor + 1, & \text{falls } \lfloor j/2 \rfloor \text{ gerade,} \\ \lfloor j/2 \rfloor - 1, & \text{falls } \lfloor j/2 \rfloor \text{ ungerade,} \end{cases}$

(ii) $z_{\lfloor j/2 \rfloor} \leq z_{j_2}$, $j_2 := \begin{cases} j+1, & j \text{ gerade}, \\ j-1, & j \text{ ungerade}, \end{cases}$

(iii) $z_{\lfloor j/2 \rfloor} \leq \min\{z_{2j}, z_{2j+1}\}$.

Hieraus folgt, dass

$$\begin{aligned} z'_j = z_{\lfloor j/2 \rfloor} &\overset{(iii)}{\leq} \min\{z_{2j}, z_{2j+1}\} = \min\{z'_{2j}, z'_{2j+1}\}, \\ z'_{\lfloor j/2 \rfloor} < z'_j = z_{\lfloor j/2 \rfloor} &\overset{(ii)}{\leq} z_{j_2} = z'_{j_2}. \end{aligned}$$

Da die Veränderung an den Positionen j und $\lfloor j/2 \rfloor$ nur die Relationen zwischen den Positionen $\lfloor \lfloor j/2 \rfloor /2 \rfloor$, $\lfloor j/2 \rfloor$, j_2, j, $2j$ und $2j+1$ eventuell ändern, muss $z'_{\lfloor \lfloor j/2 \rfloor /2 \rfloor} > z'_{\lfloor j/2 \rfloor}$ sein, weil die anderen Relationen gerade nachgewiesen wurden.

Ersetzt man $z'_{\lfloor j/2 \rfloor}$ durch $z'_{\lfloor \lfloor j/2 \rfloor /2 \rfloor}$, so gilt

$$\begin{aligned} z'_{\lfloor \lfloor j/2 \rfloor /2 \rfloor} = z_{\lfloor \lfloor j/2 \rfloor /2 \rfloor} &\overset{(i)}{\leq} z_{j_1} = z'_{j_1}, \\ z'_{\lfloor j/2 \rfloor} = z'_{\lfloor \lfloor j/2 \rfloor /2 \rfloor} = z_{\lfloor \lfloor j/2 \rfloor /2 \rfloor} &\overset{(i)}{\leq} z_{\lfloor j/2 \rfloor} \overset{(ii)}{\leq} z_{j_2} = z'_{j_2}. \end{aligned}$$

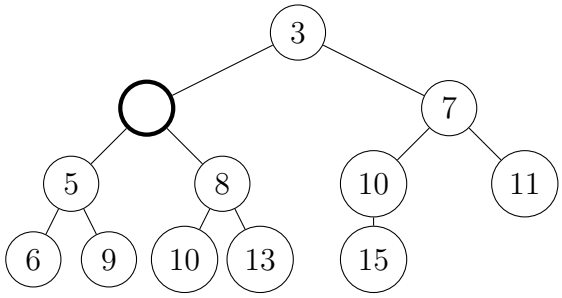
In diesem Fall ist also die Heapeigenschaft erfüllt. □

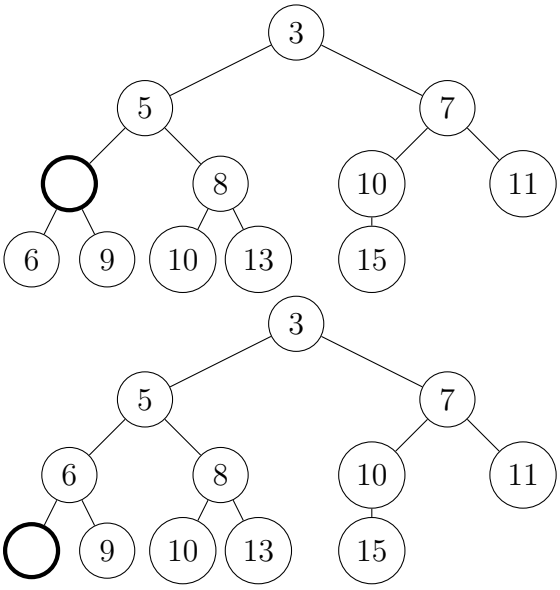
Satz 4.26. *Der Algorithmus 4.24 fügt ein Element z_{n+1} in die Folge $z_1, \ldots, z_n, z_{n+1}$ mit Aufwand $O(\log n)$ so ein, dass die Heapeigenschaft erhalten bleibt.*

Beweis. Die Erhaltung der Heapeigenschaft folgt durch sukzessive Anwendung des Lemmas 4.25 auf die Folge $z_1, \ldots, z_{n+1}$. Der Aufwand beträgt $O(\log n)$, weil die Folge $n+1, \lfloor (n+1)/2 \rfloor, \lfloor \lfloor (n+1)/2 \rfloor /2 \rfloor, \ldots$ nach spätestens $O(\log n)$ Schritten den Wert 1 erreicht. □

Ein Element kann unter Erhaltung der Heapeigenschaft dadurch entfernt werden, dass die entstehende Lücke jeweils durch das kleinere der beiden Elemente z_{2i} bzw. z_{2i+1} ersetzt wird, bis keine darunter liegenden Elemente mehr existieren. Man beachte, dass der Index des zu löschenden Elements bekannt sein muss, weil eine Suche im Heap nicht effizient durchgeführt werden kann.

Beispiel 4.27. Wir wollen die 4 aus der Folge von Beispiel 4.21 entfernen:





Satz 4.28. *Falls $z_1, \ldots, z_n$ die Heapeigenschaft besitzt, so kann durch obige Vorgehensweise ein Element mit Aufwand $O(\log n)$ so entfernt werden, dass die resultierende Folge die Heapeigenschaft besitzt.*

Beweis. analog zum Beweis von Satz 4.26. □

Bemerkung. Anstelle der obigen Vorgehensweise kann auch das letzte Element an die freiwerdende Position kopiert werden. In diesem Fall entsteht keine Lücke, nachdem die Heapeigenschaft wiederhergestellt wurde. Dazu muss aber das eingefügte Element zunächst wie bei Beispiel 4.23 aufwärts und anschließend wie bei Beispiel 4.27 abwärts den Heap durchlaufen. Wird das erste Element gelöscht, so genügt natürlich der Durchlauf nach unten.

Mit Hilfe binärer Heaps lassen sich nun n Zahlen mit Aufwand $O(n \log n)$ sortieren. Zunächst werden die Zahlen sukzessive mit Gesamtaufwand $O(n \log n)$ in den Heap eingefügt. Danach wird der Heap sukzessive abgebaut, indem jeweils das oben stehende Element (nach Lemma4.25 ist dies das jeweils kleinste Element) vom Heap wie oben beschrieben entfernt wird. Der resultierende Algorithmus wird entsprechend **Heapsort** genannt.

5 Graphen

5.1 Grundbegriffe

Definition 5.1. *Ein Graph ist ein Paar $G = (V, E)$ bestehend aus endlichen Mengen V von Knoten (engl. vertices) und E Kanten (engl. edges). Die Kanten stehen für Verbindungen zwischen den verschiedenen Knoten und können gerichtet oder ungerichtet sein.*

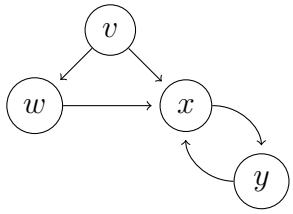
- *Für **gerichtete Graphen** (sog. Digraphen) ist $E \subset \{(v, w) \in V \times V : v \neq w\}$. Ist $e = (v, w) \in E$, dann heißt v der **Anfangsknoten** und w der **Endknoten** der Kante e.*
- *für **ungerichtete Graphen** ist E die Menge von ungeordneten Paaren (v, w) mit $v, w \in V$ und $v \neq w$.*

*Eine Kante $e = (v, w)$ heißt zu den Knoten $v, w \in V$ **inzident**.*

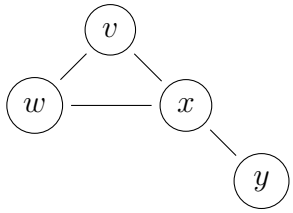
*Beachte: Es gilt $(v, w) = (w, v)$, falls (v, w) eine Kante in einem ungerichteten Graphen ist. Dies gilt jedoch nicht in einem gerichteten Graphen. Den **zugrundeliegenden ungerichteten Graphen** G' eines gerichteten Graphen G erhält man, indem man bei gleicher Knotenmenge die Kanten in G als ungeordnete Paare in G' interpretiert.*

Beispiel 5.2.

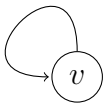
- Gerichteter Graph $G = \{(v, w), (w, x), (v, x), (x, y), (y, x)\}$:



- Ungerichteter Graph $G = \{(v, w), (w, x), (v, x), (x, y)\}$:



Bemerkung. In gerichteten Graphen werden manchmal auch Kanten der Form (v, v) zugelassen. Man spricht von **Schleifen**.



Definition 5.3. *Sei $G = (V, E)$ ein Graph. Ein* ***Weg*** *oder* ***Pfad*** *in G ist eine Knotenfolge*

$$\pi = v_0, v_1, \ldots, v_r$$

mit $r \geq 1$ und $(v_i, v_{i+1}) \in E$, $i = 0, \ldots, r-1$. Wir sprechen von einem ***Weg****, der* ***von v nach w*** *führt, wenn der Anfangsknoten $v_0 = v$ und der Endknoten $v_r = w$ ist. Ein Weg heißt* ***einfach****, falls gilt, dass $v_0, v_1, \ldots, v_{r-1}$ paarweise verschieden sind. Der Weg heißt* ***geschlossen****, falls $v_0 = v_r$. Die Anzahl der Knoten r, die in π durchlaufen werden, wird als* ***Länge*** *$|\pi|$ von π bezeichnet. Ein* ***kürzester Weg*** *von v nach w ist ein Weg von v nach w minimaler Länge. Ein Knoten w heißt von einem Knoten v* ***erreichbar****, falls ein Weg $\pi = v_0, \ldots, v_r$ in G existiert, so dass $v_0 = v$ und $v_r = w$.*

Beispiel 5.4. Beispiele für Wege in den beiden Graphen aus Beispiel 5.2 sind

(a) $\pi_1 = v, w$ (Länge 1),

(b) $\pi_2 = v, w, x$ (Länge 2),

(c) $\pi_3 = v, w, x, y, x, y$ (Länge 5).

Im ungerichteten Fall ist auch $\pi_4 = v, w, x, v$ (Länge 3) ein Weg.

Definition 5.5. *Sei $G = (V, E)$ ein Graph und $v \in V$. Wir definieren*

- *die Menge der* ***(direkten) Nachfolger*** *von v als $\mathrm{suc}(v) = \{w \in V : (v, w) \in E\}$,*
- *die Menge der* ***(direkten) Vorgänger*** *von v als $\mathrm{pre}(v) = \{w \in V : (w, v) \in E\}$,*
- *ein Knoten $w \in \mathrm{suc}(v) \cup \mathrm{pre}(v)$ ist ein* ***Nachbar*** *von v, und v, w heißen* ***benachbart*** *oder* ***adjazent****,*
- *die Menge aller von v erreichbaren Knoten als $\mathrm{suc}^*(v) = \{w \in V :$ w ist von v erreichbar$\}$,*
- *die Menge der Knoten, die v erreichen können $\mathrm{pre}^*(v) = \{w \in V : v \in \mathrm{suc}^*(w)\}$.*

Für ungerichtete Graphen gilt offenbar $\mathrm{pre}(v) = \mathrm{suc}(v)$ und auch $\mathrm{pre}^*(v) = \mathrm{suc}^*(v)$ für alle $v \in V$.

Beispiel 5.6. Für den gerichteten Graphen aus Beispiel 5.2 ist

$$\mathrm{suc}(v) = \{w, x\}, \quad \mathrm{suc}^*(v) = \{w, x, y\}, \quad \mathrm{pre}(v) = \mathrm{pre}^*(v) = \emptyset$$

und

$$\mathrm{suc}(y) = \{x\}, \quad \mathrm{suc}^*(y) = \{x, y\}.$$

Im ungerichteten Fall ist $\mathrm{suc}(x) = \{v, w, y\}$ und $\mathrm{suc}^*(x) = \{v, w, x, y\}$.

Definition 5.7. *Sei $G = (V, E)$ ein Graph. Wir definieren den* ***Grad*** *eines Knoten $v \in V$ als*

$$\deg v := |\{(x, y) \in E : x = v \text{ oder } y = v\}|$$

unter Berücksichtigung der Eigenschaft $(x, y) = (y, x)$ bei ungerichteten Graphen. Im gerichteten Fall heißt $\deg^+(v) = |\mathrm{suc}(v)|$ ***Ausgangsknotengrad*** *und $\deg^-(v) := |\mathrm{pre}(v)|$* ***Eingangsknotengrad*** *von v. Knoten mit Grad 0 heißen* ***isoliert****.*

Satz 5.8. *Für jeden Graphen $G = (V, E)$ gilt*

$$\sum_{v \in V} \deg v = 2|E|.$$

Beweis. Zu jeder Kante $e \in E$ gehören genau zwei Knoten. Auf der linken Seite der Gleichung zählt man daher jede Kante genau zweimal. □

Satz 5.9. *Für jeden Digraphen $G = (V, E)$ gilt*

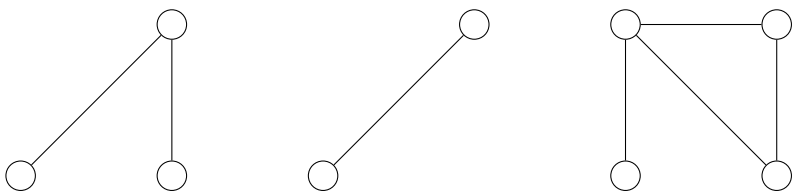
$$\sum_{v \in V} \deg^+(v) = \sum_{v \in V} \deg^-(v) = |E|$$

Beweis. Zu jeder Kante $e = (v, w) \in E$ ist $v \in \mathrm{pre}(w)$ und $w \in \mathrm{suc}(v)$. Daher zählt man bei beiden Summen jede Kante genau einmal. □

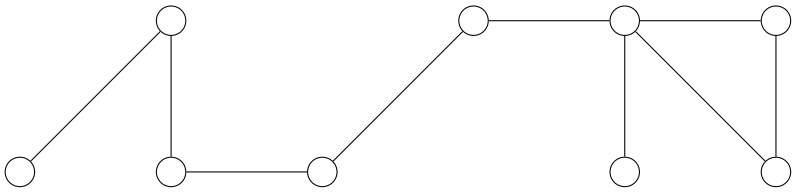
5.2 Zusammenhang

Definition 5.10. *Sei $G = (V, E)$ ein ungerichteter Graph und $C \subset V$. C heißt* ***zusammenhängend****, falls je zwei Knoten $v, w \in C$, $v \neq w$, voneinander erreichbar sind, d.h. falls gilt $w \in \mathrm{suc}^*(v)$ und $v \in \mathrm{suc}^*(w)$. Ist der Graph gerichtet, so heißt C* ***zusammenhängend****, falls C im zugrundeliegenden ungerichteten Graphen zusammenhängend ist. C heißt* ***Zusammenhangskomponente*** *von G, falls C eine nicht-leere maximale zusammenhängende Knotenmenge ist. Maximalität bedeutet hier, dass C in keiner anderen zusammenhängenden Menge $C' \subset V$ echt enthalten ist (also $C \neq C'$). Der Graph G heißt* ***zusammenhängend****, falls V zusammenhängend ist.*

Beispiel 5.11. Ein nicht zusammenhängender Graph mit drei Zusammenhangskomponenten ist



Ein zusammenhängender Graph ist



Bemerkung. Offenbar sind die Zusammenhangskomponenten eines ungerichteten Graphen die Äquivalenzklassen der Knotenmenge V unter der Äquivalenzrelation "$\sim$", wobei

$$v \sim w \quad \Longleftrightarrow \quad v \cup \operatorname{suc}^*(v) = w \cup \operatorname{suc}^*(w).$$

Insbesondere zerfällt G in paarweise disjunkte Zusammenhangskomponenten $C_1, C_2, \ldots, C_r$ mit

$$V = \bigcup_{i=1}^{r} C_i, \qquad E = \bigcup_{i=1}^{r} E_i,$$

wobei $E_i := E \cap (C_i \times C_i)$, $i = 1, \ldots, r$.

Satz 5.12. *Ein ungerichteter Graph $G = (V, E)$ ist genau dann zusammenhängend, wenn*

$$\{(v, w) \in E : v \in X,\, w \in V \setminus X\} \neq \emptyset$$

für alle $\emptyset \neq X \subsetneq V$.

Beweis. Sei $\emptyset \neq X \subsetneq V$ und $v \in X$, $w \in V \setminus X$. Weil G zusammenhängend ist, gibt es einen Weg von v nach w. Da $v \in X$, aber $w \notin X$, muss der Weg eine Kante (x, y) enthalten, sodass $x \in X$, $y \notin X$.

Umgekehrt sei angenommen, dass G nicht zusammenhängend ist. Dann existieren zwei Knoten $v, w \in V$, zwischen denen kein Weg existiert. Setze $X := \{v\} \cup \operatorname{suc}^*(v)$. Dann gilt $v \in X$ und $w \notin X$ und somit $\emptyset \neq X \subsetneq V$. Aus der Definition der Menge X folgt aber $\{(v, w) \in E : v \in X,\, w \in V \setminus X\} = \emptyset$. Dies liefert den Widerspruch, die Annahme ist somit falsch. □

Satz 5.13. *Ist $G = (V, E)$ ein ungerichteter Graph mit $n = |V| \geq 1$ Knoten und mit $m = |E|$ Kanten, so gilt: Ist G zusammenhängend, so gilt $m \geq n - 1$*

Beweis. Wir beweisen den Satz per vollständiger Induktion nach n. Für $n = 1$ folgt $m = 0 = n - 1$; für $n = 2$ ist G zusammenhängend genau dann, wenn $m = 1 = n - 1$. Wir betrachten nun den Fall $n \geq 3$. Wähle $v \in V$, so dass

$$k := \deg v = \min_{w \in V} \deg w.$$

Es gilt $k > 0$, denn sonst wäre v ein isolierter Knoten. Im Fall $k \geq 2$ folgt

$$2m = 2|E| = \sum_{w \in V} \underbrace{\deg w}_{\geq k} \geq n \cdot k \geq 2n$$

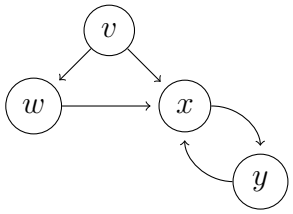
und folglich $m \geq n \geq n-1$. Für $k = 1$ ergibt sich die Aussage wie folgt: Es sei $G' = (V', E')$ derjenige Graph, der durch Entfernen des Knoten v und der inzidenten Kante entsteht. Mit G ist auch G' zusammenhängend und nach Induktionsvorraussetzung folgt wegen $|V'| = n-1$ und $|E'| = m - 1$

$$m - 1 = |E'| \geq (n-1) - 1 = n - 2$$

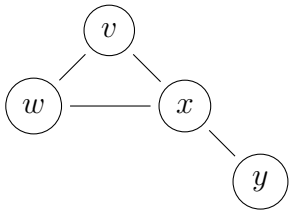
und daraus die Behauptung $m \geq n - 1$. □

Definition 5.14. *Ein **einfacher Zyklus** oder **einfacher Kreis** in einem Graphen $G = (V, E)$ ist ein einfacher, geschlossener Weg $\pi = v_0, \ldots, v_r$ mit $r \geq 2$ (gerichteter Graph) bzw. $r \geq 3$ (ungerichteter Graph).*

Beispiel 5.15. Der gerichtete Graph



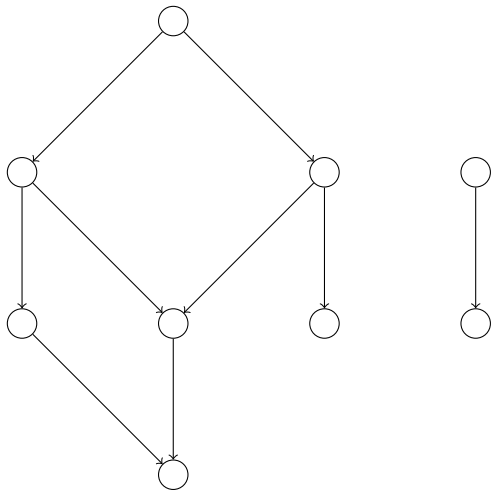
besitzt die einfachen Zyklen $\pi_1 = x, y, x$ und $\pi_2 = v, w, x, v$. Der ungerichtete Graph



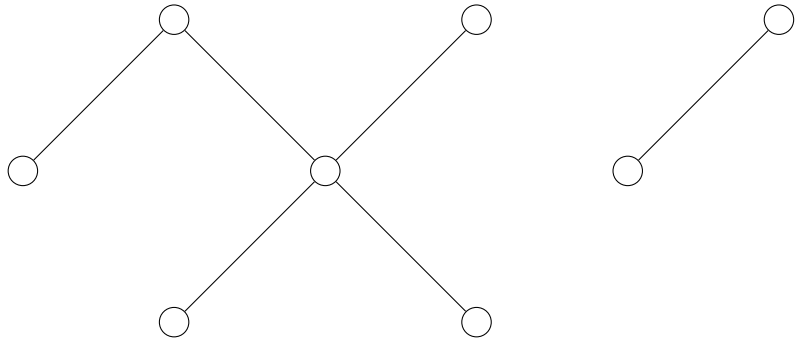
besitzt den einfachen Zyklus $\pi_1 = v, w, x, v$. Der Weg $\pi_2 = x, y, x$ ist hingegen kein Zyklus.

Definition 5.16. *Ein Graph heißt **azyklisch** oder **zyklenfrei**, falls es keine Zyklen in G gibt. Ein ungerichteter, azyklischer und zusammenhängender Graph ist ein **Baum**. Ein Knoten $v \in V$ eines Baumes mit $\deg v = 1$ heißt **Blatt**. Ist $v \in V$ kein Blatt, so wird v als **innerer Knoten** bezeichnet.*

Beispiel 5.17. Azyklischer gerichteter Graph:



Azyklischer ungerichteter Graph:



Satz 5.18. *Sei $G = (V, E)$ ein ungerichteter Graph mit n Knoten. Dann sind folgende Aussagen äquivalent:*

(i) *G ist ein Baum*

(ii) *G hat $n - 1$ Kanten und ist zusammenhängend.*

(iii) *G hat $n - 1$ Kanten und ist azyklisch.*

(iv) *G ist azyklisch und das Hinzufügen einer beliebigen Kante erzeugt einen Zyklus.*

(v) *G ist zusammenhängend und das Entfernen einer Kante erzeugt einen unzusammenhängenden Graphen.*

(vi) *Jedes Paar von verschiedenen Knoten in G ist durch genau einen einfachen Weg miteinander verbunden.*

Für den Beweis von Satz 5.18 benötigen wir das folgende

Lemma 5.19. *Für einen azyklischen und ungerichteten Graphen gilt $|V| = |E| + p$, wobei p die Anzahl der Zusammenhangskomponenten bezeichnet.*

Beweis. Die Behauptung ist klar für $m := |E| = 0$. Angenommen, die Aussage gilt für ein m. Fügen wir eine zusätzliche Kante hinzu, d.h. $|E| = m + 1$, so muss sich die Anzahl der Zusammenhangskomponenten p um Eins reduzieren, weil sonst ein Zyklus entsteht. □

Beweis von Satz 5.18.
(i) ⇒ (vi): Dies folgt aus der Tatsache, dass die Vereinigung zweier disjunkter einfacher Wege mit gleichem Anfangs- und Endknoten ein Zyklus ist.
(vi) ⇒ (v): G ist zusammenhängend nach Definition und das Entfernen der Kante (v, w) macht w unerreichbar von v.
(v) ⇒ (iv): G ist azyklisch, weil sonst eine Kante entfernt werden kann, so dass G weiterhin zusammenhängend ist. Da es in G stets einen Weg von v nach w gibt, liefert das Hinzufügen einer Kante einen Zyklus.
(iv) ⇒ (iii): Weil das Hinzufügen einer beliebigen Kante einen Zyklus erzeugt, muss G zusammenhängend sein. Nach Lemma 5.19 gilt dann $n = m + 1$.
(iii) ⇒ (ii): Weil G azyklisch ist mit $n - 1$ Kanten folgt nach Lemma 5.19, dass $p = 1$. Also

ist G zusammenhängend.
(ii) $\Rightarrow$ (i) Wir zerstören Zyklen in G durch Entfernen von Kanten. Haben wir k Kanten so entfernt, dass der resultierende Graph azyklisch ist, so folgt aus Lemma 5.19

$$\underbrace{n-1-k}_{\text{Kanten}} + \underbrace{p}_{=1} = n$$

und hieraus $k = 0$. G ist also bereits azyklisch. Weil G nach Voraussetzung ausserdem zusammenhängend ist, ist G ein Baum. □

Das folgende Lemma ist nützlich, um die Anzahl der Knoten in einem Baum ausgehend von einer bekannten Anzahl von Blättern abzuschätzen.

Lemma 5.20. *Sei L die Menge der Blätter in einem Baum mit $V \setminus L \neq \emptyset$. Weiter sei $v \in V \setminus L$ ein innerer Knoten, die sog. Wurzel. Wir definieren*

$$p_v := \min\{\deg w : w \in V \setminus (L \cup \{v\})\}.$$

Gilt $q_v := \min\{\deg v, p_v - 1\} \geq 2$, so folgt

$$|V| \leq \frac{q_v|L| - 1}{q_v - 1} \leq 2|L| - 1.$$

Beweis. Wir entfernen sukzessive die Blätter ausgehend vom Baum $V_0 := V$, indem wir in jedem Schritt alle Nachbarn vom Grad 1 eines inneren Knoten entfernen. Es bezeichne k die Anzahl der Schritte bis nur noch v übrig ist, und es sei V_ℓ der Baum nach ℓ Schritten. Dann gilt bis zum $(k-1)$-ten Schritt, dass $|V_\ell| = |V_{\ell-1}| - (p_\ell - 1)$ und $|L(V_\ell)| = |L(V_{\ell-1})| - (p_\ell - 2)$, $\ell = 1, \ldots, k-1$, wobei p_ℓ den Grad des ℓ-ten Knoten bezeichnet. Im letzten Schritt gilt $1 = |V_k| = |V_{k-1}| - \deg v$ und $1 = |L(V_k)| = |L(V_{k-1})| - (\deg v - 1)$. Also haben wir

$$|V_0| = |V_k| + \deg v + \sum_{\ell=1}^{k-1}(p_\ell - 1) = 1 + \deg v + \sum_{\ell=1}^{k-1}(p_\ell - 1),$$
$$|L(V_0)| = |L(V_k)| + \deg v - 1 + \sum_{\ell=1}^{k-1}(p_\ell - 2) = 1 - k + \deg v + \sum_{\ell=1}^{k-1}(p_\ell - 1).$$

Hieraus folgt $k = |V_0| - |L(V_0)|$ und mit $\min\{\deg v, p_\ell - 1\} \geq q_v \geq 2$, dass

$$|L(V_0)| \geq 1 + k(q_v - 1) \geq 1 + (q_v - 1)(|V_0| - |L(V_0)|) \quad \Longleftrightarrow \quad |V_0| \leq \frac{q_v|L(V_0)| - 1}{q_v - 1}.$$

□

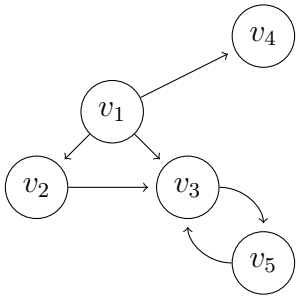
5.3 Speicherung von Graphen

Eine Möglichkeit, Graphen auf einem Rechner zu speichern, sind sog. Adjazenzmatrizen.

Definition 5.21. *Sei $G = (V, E)$ ein Graph mit Knoten v_i, $i = 1, \ldots, n := |V|$. Die Matrix $A \in \mathbb{R}^{n \times n}$ heißt* ***Adjazenzmatrix*** *von G, falls*

$$a_{ij} = \begin{cases} 1, & \text{falls } (v_i, v_j) \in E, \\ 0, & \text{sonst.} \end{cases}$$

Beispiel 5.22. Der gerichtete Graph



besitzt die Adiazenzmatrix

$$A = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Beim zugrundeliegendend ungerichteten Graphen ist

$$A = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}.$$

Die Adjazenzmatrix ungerichteter Graphen ist immer symmetrisch, d.h. es gilt $a_{ij} = a_{ji}$, $i, j, = 1, \ldots, n$, weil $(v_i, v_j) = (v_j, v_i)$ für ungerichtete Graphen. Speichert man jeden Eintrag der Adjazenzmatrix, so werden unabhängng von der Anzahl der Kanten $|V|^2$ Speichereinheiten benötigt. Effizienter ist es, Graphen durch sog. **Adjazenzlisten** darzustellen. Dabei speichert man zu jedem Knoten $v \in V$ eine Liste aller Knoten $w \in V$ mit der Eigenschaft, dass $(v, w) \in E$.

Beispiel 5.23. Die Adjazenzlisten zum gerichteten Graphen aus Beispiel 5.22 sind

v_1 : v_2, v_3, v_4
v_2 : v_3
v_3 : v_5
v_4 :
v_5 : v_3

Adjazenzlisten können als einfach verkettete Listen mit Elementen der Form

```
struct list_item {
  unsigned int idx;     // Nummer des Knotens
  list_item *next;      // Pointer auf naechstes Element
};
```

Sind a und b vom Typ `list_item`, so enthält `a.idx` die Nummer des Knoten und `a.next =&b;` definiert b als Nachfolger von a. Das Ende der Liste wird durch `b.next=NULL` markiert. Merken muss man sich jeweils den Anfang der Liste, wofür ein Array der Länge $|V|$ benötigt wird. Für gerichtete Graphen haben Adjazenzlisten den Speicherbedarf $|V|+|E|$; für ungerichtete Graphen werden $|V|+2|E|$ Speichereinheiten benötigt, da jede Kante in zwei Listen vorkommt. In der folgenden Tabelle sind Aufwand und Speicherbedarf von Adjazenzmatrizen bzw. -listen gegenübergestellt.

	Adjazenzmatrix	Adjazenzliste
Speicher	$O(n^2)$	$O(n+m)$
Knoten einfügen	$O(n)$	$O(1)$
Knoten entfernen	$O(n)$	$O(n+m)$
Kante einfügen	$O(1)$	$O(1)$
Kante entfernen	$O(1)$	$O(1)$
Testen, ob $(v,w) \in E$	$O(1)$	$O(m)$

Bemerkung. Kann man auf die Möglichkeit, Adjazenzlisten zu verändern, verzichten, so genügt es, die Position der von Null verschiedenen Einträge der schwachbesetzten Matrix zu speichern. Die Adjazenzmatrix zum gerichteten Graphen aus Beispiel 5.22 wird dann gespeichert als die Arrays

$$\texttt{cols} := (2,3,4,3,5,3) \quad \text{und} \quad \texttt{rptr} := (1,4,5,6,6,7).$$

Dabei enthält `cols` die Spaltenindizes der nicht verschwindenden Einträge und `rptr` legt fest, welches Element im Array `cols` das erste der jeweiligen Zeile ist. Genauer sind die Einträge mit den Positionen `rptr[i]` bis `rptr[i+1]-1` die Einträge der i-ten Zeile. Im letzten Eintrag von `rptr` wird daher die um Eins erhöhte Anzahl der nicht verschwindenden Einträge gespeichert. Bei dieser Datenstruktur handelt es sich um eine Spezialisierung des **CRS-Formats** (engl. compressed row storage), bei dem zusätzlich die Werte der Matrixeinträge in einem weiteren Feld der Länge des Feldes `cols` gespeichert werden. Bei ungerichteten Graphen ist die Adjazenzmatrix symmetrisch. Daher genügt es, den oberen Dreiecksanteil im CRS-Format zu speichern. Man erhält für den ungerichteten Graphen aus Beispiel 5.22 die Arrays $\texttt{cols} = (2,3,4,3,5)$ und $\texttt{rptr} = (1,4,5,6,6,6)$.

5.4 Graphendurchmusterung

Häufig muss ein Graph entlang seiner Kanten von einem Startknoten $s \in V$ durchgegangen werden (sog. Durchmusterung), um z.B. einen Knoten mit maximalem Abstand zu s zu finden. Die Hauptproblematik hierbei ist, dass die Durchmusterung wegen möglicher Zyklen im Graphen in eine Endlosschleife laufen kann. Populäre Graphendurchmusterungsmethoden sind die *Tiefensuche* und die *Breitensuche*. Beiden liegt folgender Algorithmus zugrunde:

Algorithmus 5.24.
`Input:` Graph $G=(V,E)$ und Startknoten $s \in V$
`Output:` azyklischer Graph $G'=(R,T)$ mit $R = \{s\} \cup \mathrm{suc}^*(s)$ und $T \subset E$

$R := \{s\}$; $Q := \{s\}$; $T := \emptyset$; `// R Resultat, Q ‘‘Queue’’, T ‘‘Tree’’`
`while` $(Q \neq \emptyset)$ {
  wähle $v \in Q$ beliebig;

```
    if (∃w ∈ V \ R mit e := (v, w) ∈ E) {
        R := R ∪ {w}; Q := Q ∪ {w}; T := T ∪ {e};
    }
    else Q := Q \ {v};
}
```

Satz 5.25. *Algorithmus 5.24 liefert einen azyklischen zusammenhängenden Graphen $G' = (R, T)$ mit $R = \{s\} \cup \mathrm{suc}^*(s)$ und $T \subset E$.*

Beweis. Zu jedem Zeitpunkt des Algorithmus ist (R, T) zusammenhängend, weil mit einem Knoten auch die verbindende Kante zu R bzw. T hinzugefügt wird. Ausserdem ist (R, T) azyklisch, denn eine neue Kante e verbindet stets Knoten $v \in Q \subset R$, $w \in V \setminus R$.
Angenommen, am Ende des Algorithmus existiert ein von s erreichbarer Knoten w, der nicht in R ist. Sei π ein einfacher Weg, der s mit w in G verbindet und $(x, y) \in E$ eine Kante mit $x \in R$ und $y \notin R$. Weil $x \in R$, muss x zu einem gewissen Zeitpunkt in Q gewesen sein. Der Algorithmus terminiert nicht, bevor x aus Q entfernt wurde. Dies geschieht jedoch nur, falls $(x, y) \notin E$. Dies liefert den Widerspruch. □

Satz 5.26. *Wir nehmen an, die Ausführung von "wähle $v \in Q$ beliebig" und "$\exists w \in V \setminus R$ mit $(v, w) \in E$" ist mit $O(1)$ Aufwand möglich. Dann besitzt Algorithmus 5.24 die Gesamtkomplexität $O(|V| + |E|)$.*

Beweis. Jeder Knoten wird höchstens $(\deg v + 1)$-mal und jede Kante $e \in E$ wird höchstens zweimal betrachtet. Damit ergibt sich eine Gesamtkomplexität von $O(|V| + |E|)$. □

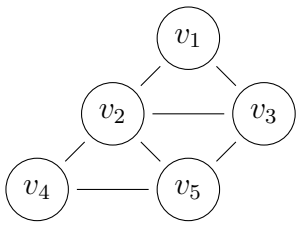
Algorithmus 5.24 kann beispielsweise verwendet werden, um die Zusammenhangskomponenten eines ungerichteten Graphen zu bestimmen, indem man ihn auf einen beliebigen Knoten $s \in V$ anwendet. Falls $R = V$, so ist G zusammenhängend. Andernfalls ist (R, T) eine Zusammenhangskomponente von G, und man fährt mit der Anwendung von Algorithmus 5.24 auf einen Knoten aus dem Rest $V \setminus R$ fort. Damit ergibt sich

Satz 5.27. *Die Zusammenhangskomponenten eines ungerichteten Graphen können mit $O(|V| + |E|)$ Aufwand bestimmt werden.*

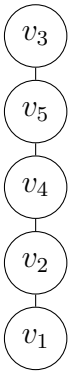
Je nachdem, wie der Knoten $v \in Q$ in Algorithmus 5.24 ausgewählt wird, ergibt sich eine andere Reihenfolge, in der der Graph durchmustert wird. Bei der **Tiefensuche** oder **DFS-Suche** (engl. Depth-First-Search) wird derjenige Knoten $v \in Q$ ausgewählt, der zuletzt zu Q hinzugefügt wurde, während bei der **Breitensuche** oder **BFS-Suche** (engl. Breadth-First-Search) der zuerst zu Q hinzugefügte Knoten ausgewählt wird.

Beispiel 5.28. Wir betrachten den Graphen G gegeben durch seine Adjazenzlisten

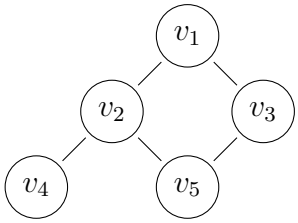
v_1 : v_2, v_3
v_2 : v_4, v_5, v_3, v_1
v_3 : v_1, v_2, v_5
v_4 : v_2, v_5
v_5 : v_2, v_3, v_4



Bei der Tiefensuche mit $s = v_1$ ergibt sich die Besuchsreihenfolge: v_1, v_2, v_4, v_5, v_3.



Die Breitensuche liefert die Reihenfolge v_1, v_2, v_3, v_4, v_5 und somit den Baum



Auf dem Graph G läßt sich eine Metrik definieren. Für $v, w \in V$ definiere

$$\mathrm{dist}_G(v, w) := \min\{|\pi|,\ \pi \text{ verbindet } v \text{ mit } w \text{ in } G\}$$

als **Abstand** von v und w, falls ein solcher Weg existiert. Andernfalls setze $\mathrm{dist}_G(V, W) = \infty$. Indem wir in Algorithmus 5.24 eine Variable ℓ einführen, die die Entfernung eines Knoten von s speichert, können wir mit Hilfe einer Breitensuche einen kürzesten Weg von s zu allen erreichbaren Knoten bestimmen.

Algorithmus 5.29.
`Input:` Graph $G = (V, E)$ und Startknoten $s \in V$
`Output:` azyklischer Graph $G' = (R, T)$ mit $R = \{s\} \cup \mathrm{suc}^*(s)$ und $T \subset E$, $\ell(v)$, $v \in R$

$R := \{s\}$; $Q := \{s\}$; $T := \emptyset$; $\ell(s) := 0$; `// Initialisierung`
`while` $(Q \neq \emptyset)$ {
  $v :=$ erster Knoten in Q;
  $Q := Q \setminus \{v\}$; `// Breitensuche`
  `for` $(w \in \mathrm{suc}(v) \setminus R)$ {
    $R := R \cup \{w\}$; $Q := Q \cup \{w\}$; $T := T \cup \{(v, w)\}$; $\ell(w) := \ell(v) + 1$;
  }
}

Satz 5.30. *Sei $G = (V, E)$ ein ungerichteter Graph. Dann enthält der BFS-Graph $G' = (R, T)$ von Algorithmus 5.29 zum Startknoten $s \in V$ einen kürzesten Weg zu jedem $v \in \mathrm{suc}^*(s)$, und es gilt, dass $\mathrm{dist}_G(s, v) = \ell(v)$ für alle $v \in R$*

Beweis. Wir beobachten zunächst, dass für das in Algorithmus 5.29 gewählte v gilt

$$\ell(w) \leq \ell(v) + 1 \quad \text{für alle } w \in Q \tag{5.1}$$

und

$$\ell(x) \geq \ell(y), \tag{5.2}$$

falls x nach y in Q aufgenommen wird. Wir zeigen nun, dass zu jedem Zeitpunkt des Algorihtmus gilt:

$$\ell(v) = \mathrm{dist}_{(R,T)}(s, v) = \mathrm{dist}_G(s, v)$$

für alle $v \in R$. Die Aussage $\ell(v) = \mathrm{dist}_{(R,T)}(s, v)$ ist klar, weil $G' = (R, T)$ azyklisch ist. Angenommen, es gibt einen Knoten w mit $\mathrm{dist}_{(R,T)}(s, w) \neq \mathrm{dist}_G(s, w)$, also $\mathrm{dist}_{(R,T)}(s, w) > \mathrm{dist}_G(s, w)$. Existieren mehrere solcher Knoten, so wähle w als denjenigen Knoten, der den geringsten Abstand zu s hat. Sei π ein kürzester Weg von s nach w in G, und es bezeichne $e = (v, w)$ die letzte Kante auf π. Dann gilt nach Konstruktion von w, dass $\mathrm{dist}_G(s, v) = \mathrm{dist}_{(R,T)}(s, v)$ und

$$\mathrm{dist}_G(s, v) + 1 = \mathrm{dist}_G(s, w) < \mathrm{dist}_{(R,T)}(s, w).$$

Also gilt

$$\ell(v) + 1 = \mathrm{dist}_{(R,T)}(s, v) + 1 = \mathrm{dist}_G(s, v) + 1 < \mathrm{dist}_{(R,T)}(s, w) = \ell(w). \tag{5.3}$$

Aus (5.1) folgt, dass $w \notin Q$. Wäre w bereits in Q aufgenommen gewesen, so hätte dies vor v sein müssen. Dann wäre aber nach (5.2) $\ell(v) \geq \ell(w)$ ein Widerspruch zu (5.3). Also war w noch nicht in Q und kann somit nicht in R sein. Es gilt also $w \in \mathrm{suc}(v) \setminus R$ und somit nach Algorithmus 5.29, dass $\ell(w) = \ell(v) + 1$ im Widerspruch zu (5.3). □

5.5 Minimal spannende Bäume

Sei $G = (V, E)$ ein ungerichteter Graph. Es sei eine Gewichts- bzw. Kostenfunktion $c : E \to \mathbb{R}$ für die Kanten des Graphen gegeben. Das Tripel (V, E, C) wird auch als **gewichteter Graph** bezeichnet. Für Teilmengen $F \subset E$ der Kantenmenge E definieren wir

$$c(F) = \sum_{e \in F} c(e).$$

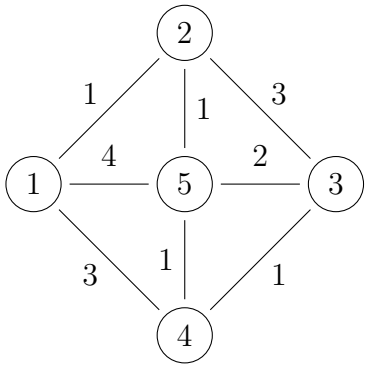
Definition 5.31. *Ein Graph $H = (V(H), E(H))$ heißt **Teilgraph** des Graphen $G = (V(G), E(G))$, falls $V(H) \subset V(G)$ und $E(H) \subset E(G)$. Gilt $V(H) = V(G)$, so wird H als **(auf)spannender Teilgraph** bezeichnet. Für einen Teilgraphen H sei $c(H) := c(E(H))$ die **Kosten** bzw. das **Gewicht** von H.*

Beim Minimum-Spanning-Tree-Problem ist ein spannender Teilgraph $T \subset G$ mit minimalen Kosten $c(T)$ gesucht. Anwendungen sind kostengünstige zusammenhängende Netzwerke wie beispielsweise Telefonnetze.

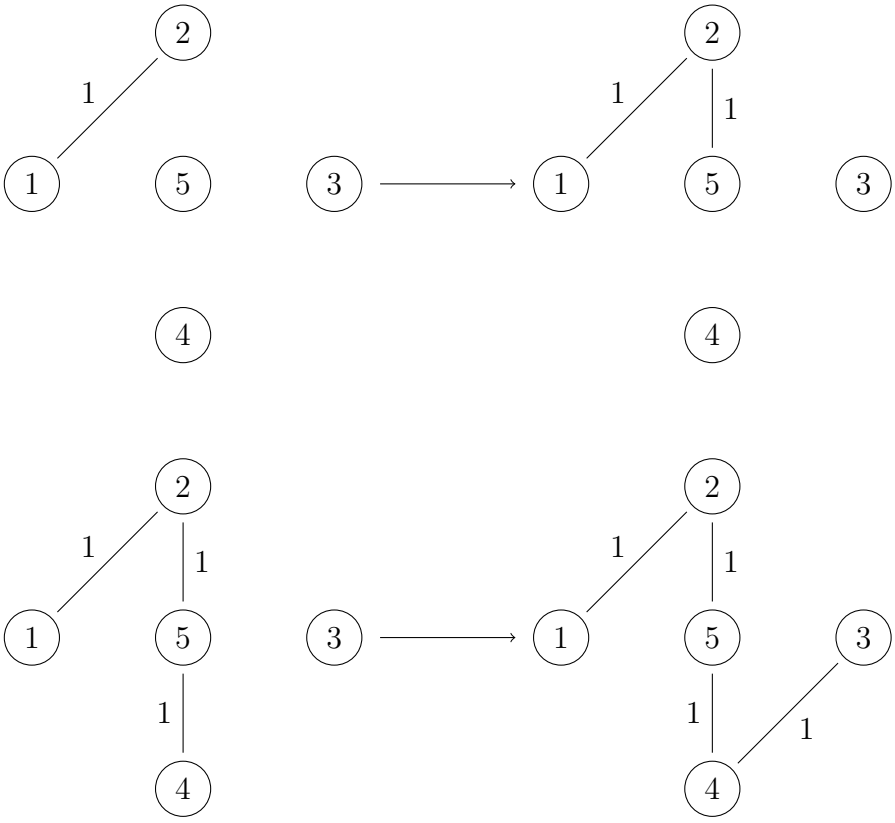
Algorithmus 5.32 (Kruskals Algorithmus)**.**
`Input:` Ein zusammenhängender ungerichteter Graph $G = (V, E)$
Gewichte $c : E \to \mathbb{R}$
`Output:` Ein minimal spannender Baum

Sortiere $E(G)$, so dass $c(e_1) \leq \ldots \leq c(e_m)$;
Setze $T := (V(G), \emptyset)$;
`for` $(i = 1,\ i \leq;\ ++i)$
  `if` ($T \cup \{e_i\}$ azyklisch) $T := T \cup \{e_i\}$;

Beispiel 5.33. Wir betrachten den folgenden Graphen. Die Kostenfunktion $c : E \to \mathbb{R}$ ist durch Zahlen entlang der Kanten dargestellt.



1: 2(1), 4(3), 5(4)
2: 1(1), 3(3), 5(1)
3: 2(3), 4(1), 5(2)
4: 1(3), 3(1), 5(1)
5: 1(4), 2(1), 3(2), 4(1)



Bemerkung. Der Test, ob $T \cup \{e_i\}$ azyklisch ist, läst sich mit $O(1)$ Aufwand durchführen, wenn man sich für jeden Knoten die ZusammenhangsKomponente merkt, in der er aktuell

liegt. Zu Beginn liegen alle Knoten in unterschiedlichen Komponenten. Wird eine Kante e_i zu T hinzugefügt, so werden zwei Komponenten vereinigt und die entsprechenden Werte der Knoten der einen Komponente auf die der anderen gesetzt. Daher muss lediglich geprüft werden, ob die Anfangs- und Endknoten der einzufügenden Kante in unterschiedlichen Komponenten liegen. Bei der Aktualisierung der Komponenteninformation sollte der Komponentenwert der Komponente mit der kleineren auf den Wert der Komponente mit der größeren Mächtigkeit gesetzt werden. Dies stellt sicher, dass jeder Knoten höchstens $O(\log n)$-mal eine neue Komponentennummer erhält.

Satz 5.34. *Der Algorithmus von Kruskal bestimmt einen minimal spannenden Baum und kann mit $O(m \log n)$ Aufwand durchgeführt werden.*

Beweis. Der vom Algorithmus von Kruskal zurückgelieferte Graph ist offensichtlich azyklisch. Angenommen, T ist nicht zusammenhängend. Dann gibt es eine Partition $V = A \cup B$, $A \cap B = \emptyset$ mit $|A|,\ |B| \geq 1$, sodass keine Kante von T zwischen A und B verläuft. Da G zusammenhängend ist, muss aber in G eine Kante zwischen A und B existieren. Diese wäre in der letzten Zeile des Algorithmus von Kruskal zu T hinzugefügt worden, was einen Widerspruch liefert.
Wir zeigen nun, dass T ein minimal spannender Baum ist. Sei T^* ein minimal spannender Baum, so dass der kleinste Index i, für den $e_i \in T$ aber $e_i \notin T^*$ größtmöglich ist. Falls dieser Index nicht existiert, so gilt $T = T^*$ und wir sind fertig. Ansonsten betrachte den Graphen $T^* \cup \{e_i\}$. Dieser enthählt einen Zyklus. Weil T azyklisch ist, muss dieser Kreis eine Kante $e_j \notin T$ enthalten. Die Kanten, die vor e_i in T eingefügt wurden, sind gemäß der Definition von i auch alle in T^* enthalten. Zu dem Zeitpunkt, zu dem Kruskals Algorithmus die Kante e_i hinzufügt, hätte daher auch e_j zu T hinzugefügt werden können, ohne einen Kreis zu erzeugen. Da aber $e_j \notin T$, muss $j > i$ und somit $c(e_j) \geq c(e_i)$ gelten. Dann ist aber $T' := (T^* \cup \{e_i\}) \setminus \{e_j\}$ ein minimalspannender Baum, der der Definition von T^* widerspricht, weil $e_\nu \in T'$, $\nu \leq i$, und somit der minimale Index i', für den $e_{i'} \in T$ aber $e_i \notin T'$, größer als i sein muss.
Zum Aufwand des Algorithmus: Das Sortieren der m Kanten benötigt $O(m \log m)$ Operationen. Nach Satz 5.13 gilt

$$n - 1 \leq m \leq \frac{n \cdot (n-1)}{2}$$

und somit $O(m \log m) = O(m \log n)$. Nach der letzten Bemerkung benötigt der Test, ob $T \cup \{e_i\}$ azyklisch ist, in jedem der m Durchläufe der `for`-Schleife $O(1)$ Operationen. Inklusive der Aktualisierung der Komponentenummern werden daher $O(m + n \log n)$ Operationen für die `for`-Schleife benötigt. □

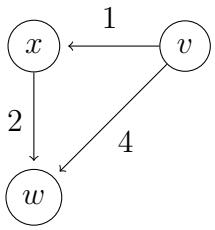
5.6 Kürzeste Wege

Gegeben sei ein Graph $G = (V, E)$ und eine Gewichtsfunktion $c : E \to \mathbb{R}$. Wir verallgemeinern die Definition des Abstands $\text{dist}_G(v, w)$ zweier Knoten $v, w \in V$, indem wir die Kosten c entlang der Kanten berücksichtigen

$$\text{dist}_{(G,c)}(v, w) := \begin{cases} \min\{c(\pi) : \pi \text{ ist ein Weg von } v \text{ nach } w\}, & \text{falls } w \text{ von } v \text{ erreichbar,} \\ \infty, & \text{sonst.} \end{cases}$$

Entsprechend verallgemeinern wir den Begriff des kürzesten Weges, indem wir diesen Abstandsbegriff zu Grunde legen.

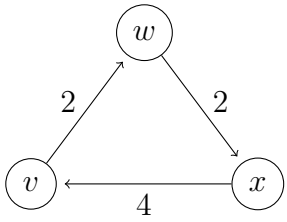
Beispiel 5.35. Gegeben sei der folgende gewichtete Graph (V, E, c)



Der kürzeste Weg von v nach w ist $\pi_1 = v, x, w$. Seine Weglänge ist $c(\pi_1) = 1 + 2 = 3$, während der direkte Weg $\pi_2 = v, w$ die Weglänge $c(\pi_2) = 4$ hat.

Beim kürzesten Wege-Problem besteht die Aufgabe darin, für ein Paar $s, t \in V$ einen kürzesten Weg von s nach t zu finden, bzw. die Angabe, dass ein solcher Weg nicht existiert. Eine offensichtliche Anwendung ist die Planung von Verkehrsrouten. Es zeigt sich, dass das Problem schwierig ist, falls negative Kantengewichte vorkommen.

Beispiel 5.36. Wir betrachten den folgenden Graphen



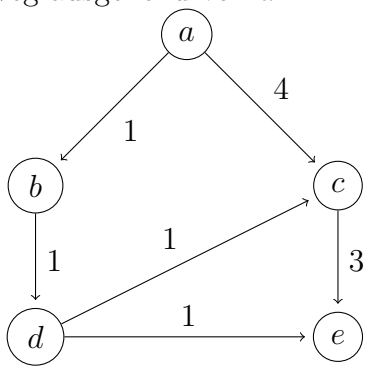
Der Weg $\pi_1 = v, w, x, v$ hat die Länge $c(\pi_1) = -1$, der Weg $\pi_2 = v, w, x, v, w, x, v$ die Länge $c(\pi_2) = -2$. Daher existiert kein kürzester Weg.

Wir setzen daher voraus, dass alle Gewichte positiv sind.

Algorithmus 5.37 (Dijkstras Algorithmus)**.**
`Input:` Ein Digraph $G = (V, E)$, $v : E \to \mathbb{R}^+$, $s \in V$
`Output:` $\ell(x) = \text{dist}_{(G,c)}(s, x)$ für alle $x \in V$,
$p(x)$ enthält den vorletzten Knoten auf dem minimalen Weg von s nach x, falls ein solcher existiert.

$\ell(s) := 0$; $\ell(x) = \infty$ für alle $x \in V \setminus \{s\}$; $Q := V$;
`while` $(Q \neq 0)$ {
  $x :=$ Knoten in Q mit $\ell(x)$ minimal;
  $Q := Q \setminus \{x\}$;
  `for` $(y \in Q$ mit $e := (x, y) \in E)$ {
    `if` $(\ell(x) + c(e) < \ell(y))$ {
      $\ell(y) := \ell(x) + c(e)$;
      $p(y) := x$;
    }
  }
}

Beispiel 5.38. Wir betrachten den folgenden gewichteten Graphen und suchen den kürzesten Weg ausgehend von a.



Iteration	x	$p(x)$	a	b	c	d	e
0			0	∞	∞	∞	∞
1	a	–	–	1	4	∞	∞
2	b	a	–	–	4	2	∞
3	d	b	–	–	3	–	5
4	c	d	–	–	–	–	4
5	e	c	–	–	–	–	–

Bemerkung.

(a) Der Algorithmus von Dijkstra berechnet einen kürzesten Weg von s zu allen erreichbaren Knoten $t \in \mathrm{suc}^*(s) \cup \{s\}$. Es ist kein Algorithmus mit geringerer Komplexität bekannt, der den kürzesten Weg zu genau einem Knoten $t \in \mathrm{suc}^*(s) \cup \{s\}$ bestimmt.

(b) $\ell(t)$ gibt über die Erreichbarkeit bzw. die Länge eines kürzesten Weges Auskunft. Der Weg kann rekonstruiert werden, indem mit dem Feld p der Weg rückwärts gegangen wird. In Beispiel 5.38 kann man einen kürzesten Weg von a nach e als $p(e) = c$, $p(c) = d$, $p(d) = b$, $p(b) = a$ rekonstruieren. Ist man nur an der Weglänge interessiert, so kann p aus Algorithmus 5.37 entfernt werden.

Satz 5.39. *Dijkstras Algorithmus arbeitet korrekt. Der Aufwand ist von der Ordnung $|V|^2$.*

Beweis. Angenommen, die Aussage ist falsch. Sei x der erste Knoten, der aus Q entfernt wird mit $\ell(x) \neq \mathrm{dist}_{(G,c)}(s,x)$. Dann gilt $x \neq s$ und $\ell(x) > \mathrm{dist}_{(G,c)}(s,x)$, weil die 7-te Zeile die Existenz eines Weges von s nach x der Länge höchstens $\ell(x)$ sichert. Wir betrachten den Zeitpunkt, bevor x aus Q entfernt wird. Sei $s = v_1, \ldots, v_k = x$ ein kürzester Weg von s nach x in G und i der größte Index eines Knoten, der bereits aus Q entfernt wurde. Dieser Index existiert, weil $s \notin Q$, und es gilt $i < k$, weil x noch in Q. Wegen $v_{i+1} \in Q$ gilt $\ell(v_{i+1}) \leq \ell(v_i) + c(v_i, v_{i+1})$ nach Zeile 6 und 7. Wegen Zeile 3 gilt

$$\ell(x) \leq \ell(v_{i+1}) \leq \ell(v_i) + c(v_i, v_{i+1}) \leq \mathrm{dist}_{(G,c)}(s,x).$$

Hierbei gilt die letzte Ungleichung, weil $v_1, \ldots, v_k$ ein kürzester Weg von s nach x ist. Dies ergibt einen Widerspruch zur Annahme; die Aussage ist also richtig. Wegen $\ell(x) = \mathrm{dist}_{(G,c)}(s,x)$ ist $p(x)$ der vorletzte Knoten auf einem kürzesten Weg von s nach x.
Zur Laufzeit: Die `while`-Schleife wird $|V|$-mal durchlaufen, wobei jeder Durchlauf $O(|V|)$ Aufwand benötigt (bereits in Zeile 3). □

Bemerkung. Im Fall von Gewichten $c \equiv 1$ kann Algorithmus 5.29 mit Aufwand $O(|V|+|E|)$ verwendet werden, um einen kürzesten Weg zu bestimmen.

Durch die Verwendung von Heaps läßt sich Dijkstras Algorithmus effizienter implementieren.

Satz 5.40. *Dijkstras Algorithmus kann so implementiert werden, dass der Aufwand von der Ordnung $|V| \cdot \log |V|$ ist.*

Beweis. Ist die Menge Q als binärer Heap implementiert, dann benötigen die Zeilen 3 und 4 $O(\log |V|)$ Operationen. Ebenso werden für die Zeilen 6 und 7 $O(\log |V|)$ Operationen benötigt. Die `while`-Schleife wird $|V|$-mal durchlaufen. □

Wir hatten bereits angemerkt, dass Dijkstras Algorithmus bei negativen Gewichten Schwierigkeiten hat. Im Fall von **konservativen Gewichten**, d.h. es gibt keinen Zyklus in G mit negativem Gewicht, gilt ein Prinzip, das wir bereits mehrfach verwendet haben.

Lemma 5.41. *Sei $G = (V, E)$ ein gerichteter Graph und $c : E \to \mathbb{R}$ konservativ. Sei $\pi : s = v_1, \ldots, v_{k+1} = w$ ein Weg zwischen $s, w \in V$ mit höchstens $k \in \mathbb{N}$ Kanten. Dann ist $\pi' : s = v_1, \ldots, v_k$ ein kürzester Weg zwischen s und v_k unter allen Wegen zwischen s und v_k mit höchstens $k - 1$ Kanten.*

Beweis. Angenommen, π' ist ein kürzester Weg zwischen s und v_k mit höchstens $k - 1$ Kanten und $c(\pi') < c(\pi \setminus \{(v_k, w)\}) = c(\pi) - c(v_k, w)$. Falls $w \notin V(\pi')$, so ist $\pi' \cup \{(v_k, w)\}$ ein kürzerer Weg von s nach w als π mit höchstens k Kanten. Im anderen Fall ($w \in V(\pi')$) gilt

$$c(\pi'_{[s,w]}) = c(\pi') + c(v_k, w) - c(\pi'_{[w,v_k]} \cup \{(v_k, w)\}) < c(\pi) - c(\pi'_{[w,v_k]} \cup \{(v_k, w)\}) \leq c(\pi),$$

weil c konservativ und $\pi'_{[w,v_k]} \cup \{(v_k, w)\}$ ein Zyklus ist. Dies widerspricht der Wahl von π. □

Algorithmus 5.42 (Moore-Bellman-Ford)**.**
`Input:` Ein gerichteter Graph G, konservatives $c : E \to \mathbb{R}$, $s \in V$
`Output:` $\ell(x) = \text{dist}_{(G,c)}(s, x)$ für alle $x \in V$, Vorgängerinformation p.

$\ell(s) = 0$; $\ell(x) = \infty$ für $x \in V \setminus \{s\}$;
`for` $(i = 1; i \leq |V|; ++i)$ {
  `for` $((x, y) \in E)$ {
    `if` $(\ell(y) > \ell(x) + c(x, y))$ {
      $\ell(y) := \ell(x) + c(x, y)$;
      $p(y) := x$;
    }
  }
}

Satz 5.43. *Der Moore-Bellman-Ford Algorithmus arbeitet korrekt und kann in $O(|V| \cdot |E|)$ Operationen ausgeführt werden.*

Beweis. Wir zeigen induktiv über die Anzahl der k Kanten eines kürzesten Weges zwischen s und y, dass $\ell(y) = \text{dist}_{(G,c)}(s, y)$ nach spätestens k Iterationen der `for`-Schleife in der 2. Zeile terminiert.
Der Fall $k = 1$ ist klar. Für den Induktionsschritt sei y ein Konten, so dass ein kürzester

Weg π zwischen s und y mit k Kanten existiert. Es bezeichne x den vorletzten Knoten dieses Weges. Dann ist $\pi_{[s,x]}$ nach Lemma 5.41 ein kürzester Weg zwischen s und x der Länge $k-1$. Hieraus folgt per Induktionsannahme $\ell(x) = \text{dist}_{(G,c)}(s,x)$ nach $k-1$ Iterationen und somit

$$\ell(y) \leq \ell(x) + c(x,y) = \text{dist}_{(G,c)}(s,x) + c(x,y) = \text{dist}_{(G,c)}(s,y).$$

Die Abschätzung $\ell(y) \geq \text{dist}_{(G,c)}(s,y)$ ist wieder klar. □

Bemerkung. Um die Abstände zwischen allen Paaren von Knoten zu berechnen, kann man $|V|$-mal den Moore-Bellman-Ford Algorithmus aufrufen. Dies erfordert $O(|V|^2 \cdot |E|)$ Operationen. Der aus der Literatur bekannte Algorithmus von Floyd-Warshall benötigt dafür $O(|V|^3)$ Operationen.

5.7 Netzwerkflüsse

Zur Motivation von Netzwerkflussproblemen betrachten wir das folgende aktuelle

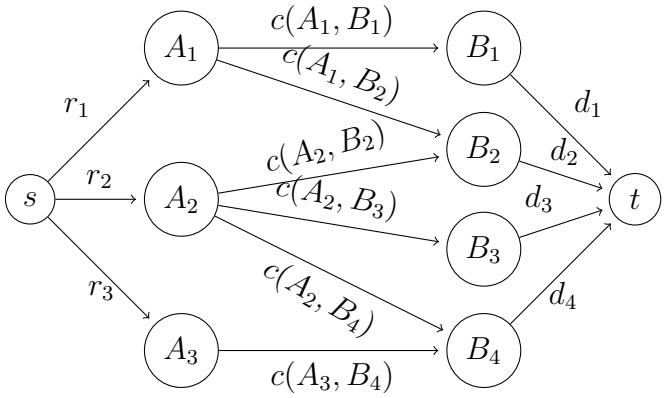
Beispiel 5.44. In den Depots $A_1, \ldots, A_p$ des Weihnachtsmannes liegen $r_1, \ldots, r_p$ Geschenke zum Transport bereit. In den Städten $B_1, \ldots, B_q$ warten $d_1, \ldots, d_q$ Kinder auf ihr Geschenk. Dabei nehmen wir an, dass jedes Kind das gleiche Geschenk bekommt. Von Depot A_i können an Heiligabend vom Knecht Ruprecht und seinen Kollegen höchstens $c(A_i, B_j)$ Geschenke zur Stadt B_j transportiert werden. Es ergeben sich folgende Fragen:

1. Ist es möglich, jedem Kind ein Geschenk zu bringen?

2. Falls nein, wieviele Geschenke können maximal zu den Kindern gebracht werden?

3. Wieviele Geschenke sollen von Depot A_i in die Stadt B_j transportiert werden?

Den Kanten (A_i, B_j) ordnen wir die Kapazität $c(A_i, B_j)$ zu. Um die Menge der vorhandenen bzw. der benötigten Geschenke zu erfassen, führen wir zwei weitere Knoten s, t und Kanten (s, A_i) bzw. (B_j, t) mit Kapazitäten v_i bzw. d_j ein.



Zur Beantwortung dieser drei Fragen lösen wir folgendes Problem: Was ist der maximale Fluss von s nach t und wie sieht dieser aus? Dabei ist der Fluss auf einer Kante durch ihre Kapazität beschränkt und der gesamte Fluss, der einen Knoten betritt, muss diesen auch wieder verlassen.

Definition 5.45. *Ein* ***Netzwerk*** *ist ein Tupel $N = (V, E, c, s, t)$ bestehend aus*

(i) einem Digraphen $G(V, E)$,

(ii) einer Kapazitätsfunktion $c : E \to \mathbb{R}^+$,

(iii) einer Quelle $s \in V$ mit $\mathrm{pre}(s) = \emptyset$,

(iv) einer Senke $t \in V$ mit $\mathrm{suc}(t) = \emptyset$.

Ein ***Fluss*** *$f : E \to \mathbb{R}_0^+$ ist eine Funktion, die folgende Bedingungen erfüllt:*

(i) Kapazitätsbedingung: $f(v, w) \leq c(v, w)$ für alle $(v, w) \in E$,

(ii) Flusserhaltung

$$\sum_{u \in \mathrm{pre}(v)} f(u, v) = \sum_{w \in \mathrm{suc}(v)} f(v, w)$$

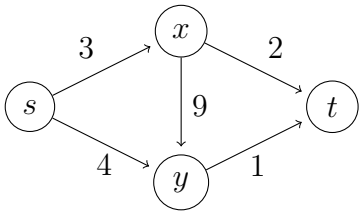
für alle $v \in V \setminus \{t, s\}$.

Der ***Wert*** *eines Flusses f ist definiert als*

$$|f| := \sum_{v \in \mathrm{suc}(s)} f(s, v).$$

Der ***maximale Fluss*** *eines Netzwerkes N ist der maximale Wert aller Flüsse für N.*

Beispiel 5.46. Wir betrachten das folgende Netzwerk



Der maximale Fluss ist 3.

Definition 5.47. *Ein* ***Schnitt*** *für $N = (V, E, c, s, t)$ ist eine Knotenmenge $S \subset V$ mit $s \in S$, $t \notin S$. Die* ***Kapazität*** *eines Schnittes ist gegeben durch*

$$\mathrm{cap}(S) := \sum_{v \in S} \sum_{w \in \mathrm{suc}(v) \setminus S} c(v, w).$$

Die ***minimale Schnittkapazität*** *von N ist die minimale Kapazität aller Schnitte für N.*

Lemma 5.48. *Sei S ein Schnitt für $N = (V, E, c, s, t)$. Dann gilt für jeden Fluss f*

(i)

$$|f| = \sum_{v \in S} \left(\sum_{w \in \mathrm{suc}(v) \setminus S} f(v, w) - \sum_{u \in \mathrm{pre}(v) \setminus S} f(u, v) \right),$$

(ii) $|f| \leq \operatorname{cap}(S)$.

Beweis. Aussage (i) folgt aus der Flusserhaltung

$$\begin{aligned}|f| &= \sum_{v\in\operatorname{suc}(s)} f(s,v) = \sum_{v\in S}\left(\sum_{w\in\operatorname{suc}(v)} f(v,w) - \sum_{u\in\operatorname{pre}(v)} f(u,v)\right)\\ &= \sum_{v\in S}\left(\sum_{w\in\operatorname{suc}(v)\setminus S} f(v,w) - \sum_{u\in\operatorname{pre}(v)\setminus S} f(u,v)\right)\\ &\quad + \underbrace{\sum_{v\in S}\left(\sum_{w\in\operatorname{suc}(v)\cap S} f(v,w) - \sum_{u\in\operatorname{pre}(v)\cap S} f(u,v)\right)}_{=0}.\end{aligned}$$

Die beiden letzten Summen stimmen überein, weil

$$\{(v,w) : v\in S,\, w\in\operatorname{suc}(v)\cap S\} = \{(v,w) : w\in S,\, v\in\operatorname{pre}(w)\cap S\}.$$

Weil $0 \leq f(e) \leq c(e)$ für alle $e\in E$, folgt (ii) aus

$$|f| \overset{(i)}{\leq} \sum_{v\in S}\sum_{w\in\operatorname{suc}(v)\setminus S} f(v,w) \leq \sum_{v\in S}\sum_{w\in\operatorname{suc}(v)\setminus S} c(v,w) = \operatorname{cap}(S).$$

□

Definition 5.49. *Sei $G=(V,E)$ ein Digraph. Für $e=(v,w)\in E$ bezeichne $e^{-1} := (w,v)$ als* ***gegenläufige Kante****. Ist $c : E\to\mathbb{R}^+$ eine Kapazitätenfunktion und $f : E\to\mathbb{R}_0^+$ ein Fluss, dann sind die* ***Restkapazitäten*** *c_f definiert durch*

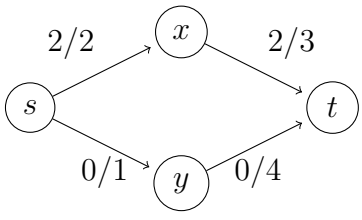
$$c_f(e) := c(e) - f(e) \quad \text{und} \quad c_f(e^{-1}) := f(e).$$

Der ***Restgraph*** *G_f besteht aus den Knoten V und den Kanten*

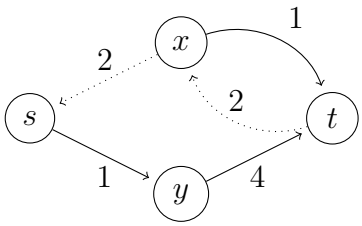
$$E(G_f) := \{e\in E : c_f(e) > 0\} \cup \{e^{-1} : e\in E \text{ und } c_f(e^{-1}) > 0\}.$$

Ein ***augmentierender Weg*** *ist ein Weg von s nach t im Restgraphen G_f.*

Beispiel 5.50. Wir betrachten folgendes Netzwerk mit Fluss/Kapazität.



Der Restgraph ist



Dabei sind gegenläufige Kanten gepunktet dargestellt.

Sei π ein augmentierender Weg und $0 < \gamma \leq \min_{e \in E(\pi)} c_f(e)$. Unter der Augmentierung des Flusses f entlang π um γ versteht man das Folgende. Für jedes $e \in E(\pi)$ erhöhe $f(e)$ um γ, falls $e \in E(G)$ und verringere $f(e^{-1})$ um γ, falls $e^{-1} \in E(G)$. Durch diese Augmentierung erhält man einen neuen Fluss f' entlang π mit $|f'| = |f| + \gamma$.

Algorithmus 5.51 (Ford-Fulkerson)**.**
`Input:` Netzwerk $N = (V, E, c, s, t)$
`Output:` Fluss f maximalen Wertes

Setze $f(e) = 0$ für alle $e \in E$;
`while` ($\exists$ augmentierender Weg π) {
  $\gamma := \min\limits_{e \in E(\pi)} c_f(e)$;
  augmentiere den Fluss f entlang π um γ;
}

Satz 5.52. *Ein Fluss von s nach t hat genau dann maximalen Wert, wenn es keinen augmentierenden Weg gibt.*

Beweis. Falls es einen augmentierenden Weg π gibt, so kann f entlang π zu einem Fluss mit größerem Wert augmentiert werden.
Falls es keinen augmentierenden Weg gibt, so ist t von s aus in G_f nicht erreichbar. Sei R die Menge der von s aus in G_f erreichbaren Knoten. Dann gilt $f(e) = c(e)$ für alle $e = (v, w)$, $v \in R$, $w \in \mathrm{suc}(v) \setminus R$ und $f(e) = 0$ für alle $e = (v, w)$, $w \in R$, $v \in \mathrm{pre}(w) \setminus R$, weil sonst nach Definition von G_f eine Kante aus R herausläuft in G_f. Nach Lemma 5.48 (i) gilt

$$|f| = \sum_{v \in R} \sum_{w \in \mathrm{suc}(v) \setminus R} f(v, w) - \sum_{v \in R} \sum_{u \in \mathrm{pre}(v) \setminus R} f(u, v) = \sum_{v \in R} \sum_{w \in \mathrm{suc}(v) \setminus R} c(v, w) = \mathrm{cap}(R).$$

Nach Lemma 5.48 (ii) hat f maximalen Wert. □

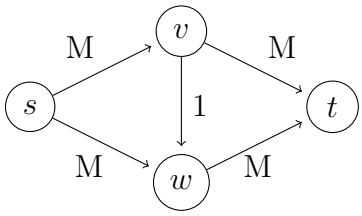
Satz 5.53 (Max-Flow-Min-Cut Theorem). *In einem Netzwerk (V, E, c, s, t) stimmt der maximalen Fluss von s nach t und die minimale Schnittkapazität überein.*

Beweis. Nach Lemma 5.48 (ii) ist der Wert eines Flusses von s nach t höchstens so groß wie die Kapazität eines Schnittes. Im Beweis von Satz 5.52 wurde ein Schnitt R definiert mit $|f| = \mathrm{cap}(R)$. □

Satz 5.54. *Der Algorithmus von Ford-Fulkerson ist korrekt und benötigt bei ganzzahligen Kapazitäten $O\left(|E| \cdot \sum_{e \in E} c(e)\right)$ Aufwand.*

Beweis. Die Korrektheit ergibt sich aus Satz 5.52. Für die Aufwandsabschätzung berücksichtige man, dass sich der Fluss bei jedem Durchlauf der `while`-Schleife um mindestens 1 erhöht. Ein Schleifendurchlauf benötigt $O(|E|)$ Aufwand. □

Beispiel 5.55. Wir betrachten folgendes Netzwerk.



Offensichtlich ist der maximale Fluss $2M$. Augmentieren wir abwechselnd entlang der Wege s, v, w, t und s, w, v, t, so erhöht sich der Wert von f um $\gamma = 1$. Folglich werden $2M$ Schritte benötigt.

Bemerkung. Ford und Fulkerson haben anhand eines Beispiels gezeigt, dass bei irrationalen Kapazitäten der Algorithmus möglicherweise nicht terminiert. Ferner kann der Algorithmus exponentiellen Aufwand verursachen.

Beispiel 5.55 zeigt, dass bei ungeschickter Wahl des augmentierenden Weges die Anzahl der Augmentierungsschritte sehr groß sein kann. Polynomielle Laufzeiten können im Ford-Fulkerson-Algorithmus durch die Wahl eines kürzesten Weges erreicht werden. Hierbei bezieht sich die Länge auf die Anzahl der Kanten.

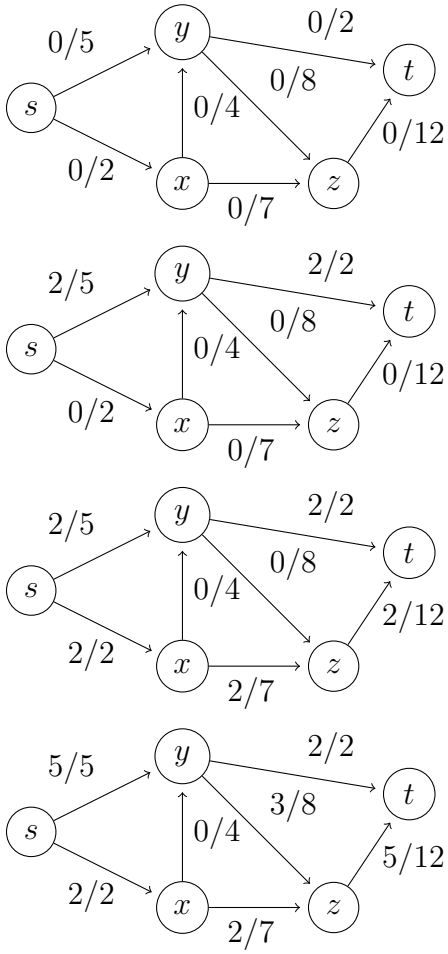
Algorithmus 5.56 (Edmonds-Karp).
`Input:` Netzwerk $N = (V, E, c, s, t)$
`Output:` Fluss f maximalen Wertes

Setze $f(e) = 0$ für alle $e \in E$;
`while` ($\exists$ augmentierender Weg) {
  wähle kürzesten augmentierenden Weg π von s nach t;
  $\gamma := \min_{e \in E(\pi)} c_f(e)$;
  augmentiere den Fluss f entlang π um γ;
}

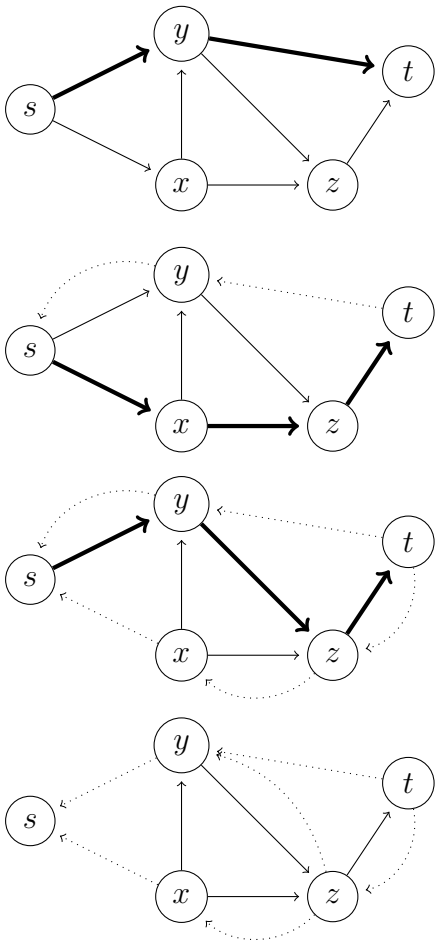
Bemerkung. Die Wahl eines kürzesten Weges kann durch eine Breitensuche im Restgraphen realisiert werden; vgl. Algorithmus 5.29.

Beispiel 5.57. Wir illustrieren den Edmonds-Karp-Algorithmus anhand des folgenden Netzwerks.

Netzwerk mit aktuellem Fluss



Restgraph und kürzester augmentierender Weg



Für die Laufzeitabschätzung des Algorithmus benötigen wir

Lemma 5.58. *Der Abstand* $\mathrm{dist}_{G_f}(s,x)$, $x \in V$, *wächst monoton mit jedem Schleifendurchlauf des Edmonds-Karp-Algorithmus.*

Beweis. Für $x = s$ ist dies sicher richtig. Sei also $x \neq s$. Angenommen, es gibt einen Knoten $x \in V$, sodass $\mathrm{dist}_{G_f}(s,x)$ in einer Iteration des Algorithmus kleiner wird. Sei f der Fluss, bevor $\mathrm{dist}_{G_f}(s,x)$ für ein $x \in V$ zum ersten Mal kleiner wird und sei f' der Fluss eine Iteration später. Ferner sei x so gewählt, dass $\mathrm{dist}_{G_{f'}}(s,x) < \mathrm{dist}_{G_f}(s,x)$ und $\mathrm{dist}_{G_{f'}}(s,x)$ unter allen möglichen Wahlen von x minimal ist. Sei π ein kürzester Weg von s nach x in G'_f und y der Vorgänger von x in π. Dann gilt

$$\mathrm{dist}_{G_{f'}}(s,x) = \mathrm{dist}_{G_{f'}}(s,y) + 1$$

und nach Wahl von x

$$\mathrm{dist}_{G_{f'}}(s,y) \geq \mathrm{dist}_{G_f}(s,y).$$

Wir zeigen, dass $(y,x) \notin E(G_f)$. Falls doch, so gälte

$$\mathrm{dist}_{G_f}(s,x) \leq \mathrm{dist}_{G_f}(s,y) + 1 \leq \mathrm{dist}_{G_{f'}}(s,y) + 1 \leq \mathrm{dist}_{G_{f'}}(s,x)$$

im Widerspruch zu unserer Annahme für x. Es gilt also $(y,x) \notin E(G_f)$ und $(y,x) \in E(G_{f'})$. Daher muss der Edmonds-Karp-Algorithmus den Fluss entlang der Kante (x,y) erhöht haben, um von f zu f zu kommen. Weil der Edmonds-Karp-Algorithmus immer einen kürzesten augmentierenden Weg auswählt, muss die Kante (x,y) in dem kürzesten Weg von s nach y enthalten sein. Es folgt

$$\text{dist}_{G_f}(s,x) = \text{dist}_{G_f}(s,y) - 1 \leq \text{dist}_{G_{f'}}(s,y) - 1 = \text{dist}_{G_{f'}}(s,x) - 2,$$

was $\text{dist}_{G_{f'}}(s,x) < \text{dist}_{G_f}(s,x)$ widerspricht. Unsere Annahme ist falsch. □

Satz 5.59. *Algorithmus 5.56 terminiert unabhängig von den Kapazitäten nach höchstens $\frac{1}{2}|V| \cdot |E|$ Schritten und benötigt daher $O(|V| \cdot |E|^2)$ Aufwand.*

Beweis. Sei (x,y) eine kritische Kante auf einem kürzesten ausgewählten augmentierenden Weg π, d.h. $c_f(x,y) = \gamma$. Es gilt $\text{dist}_{G_f}(s,y) = \text{dist}_{G_f}(s,x) + 1$. Die Kante (x,y) wird aus G_f entfernt. Sie kann zu einem späteren Zeitpunkt nur dann wieder hinzugefügt werden, falls (y,x) in einem augmentierenden (kürzesten) Weg auftritt. Sei f' der Fluss zu diesem Zeitpunkt. Dann gilt $\text{dist}_{G_{f'}}(s,x) = \text{dist}_{G_{f'}}(s,y) + 1$. Da $\text{dist}_{G_f}(s,y) \leq \text{dist}_{G_{f'}}(s,y)$ nach Lemma 5.58, folgt

$$\text{dist}_{G_{f'}}(s,x) = \text{dist}_{G_{f'}}(s,y) + 1 \geq \text{dist}_{G_f}(s,y) + 1 = \text{dist}_{G_f}(s,x) + 2.$$

Falls eine Kante mehrfach kritisch ist, so erhöht sich der Abstand von s nach x jedesmal. Weil der Abstand höchstens $|V| - 1$ ist, kann die Kante höchstens $\frac{1}{2}|V|$-mal kritisch sein. Es gibt $|E|$ Kanten, die kritisch werden können. Also gibt es höchstens $\frac{1}{2}|V| \cdot |E|$ Schritte. Bei Terminierung von Algorithmus 5.56 existiert kein augmentierender Weg von s nach t im Restgraphen G_f. Nach Satz 5.52 ist der Fluss dann maximal. Die Aufwandsabschätzung ergibt sich, weil die Breitensuche den Aufwand $O(|E|)$ hat. Weil G zusammenhängend ist, gilt nämlich $|V| - 1 \leq |E|$. □

5.8 Matchings in bipartiten Graphen

Definition 5.60. *Sei $G = (V,E)$ ein ungerichteter Graph. Ein **Matching** von G ist eine Kantenmenge $M \subset E$, sodass für alle $(v,w),(x,y) \in M$ gilt*

$$(v,w) \neq (x,y) \Rightarrow \{v,w\} \cap \{x,y\} = \emptyset,$$

*d.h. jeder Knoten von G liegt auf höchstens einer Kante von M. Ein Matching M heißt **maximal**, falls $M \cup \{e\}$ kein Matching ist für alle $e \in E \setminus M$. M ist **größtes Matching**, falls $|M| \geq |M'|$ für alle Matchings M' von G.*

Algorithmus 5.61 (Maximales Matching)**.**
`Input:` ungerichteter Graph $G = (V,E)$
`Output:` maximales Matching M

$M := \emptyset$;
`while` ($\exists e \in E \setminus M$ mit $M \cup \{e\}$ ist Matching) $M := M \cup \{e\}$;

Satz 5.62. *Algorithmus 5.61 ist korrekt und benötigt $O(|E|)$ Aufwand.*

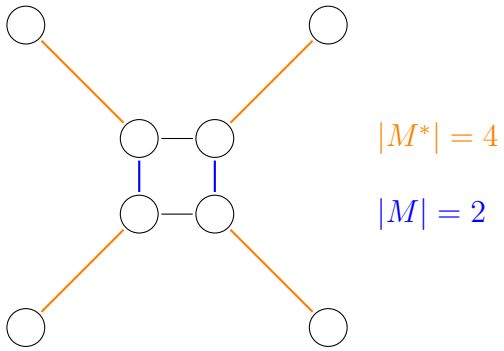
Beweis. klar □

Ein maximales Matching ist im Allgemeinen kein größtes Matching. Der folgende Satz besagt aber, dass sich die Kardinalität eines maximalen Matchings höchstens bis auf eine Konstante von der eines größten Matchings unterscheidet.

Satz 5.63. *Sei M ein maximales und M^* ein größtes Matching. Dann gilt $|M| \geq \frac{1}{2}|M^*|$.*

Beweis. Sei $S = \{x \in V : \exists e \in M \text{ mit } x \in e\}$. Dann ist $|S| = 2|M|$. Für $(x, y) \in M^*$ gilt $\{x, y\} \cap S \neq \emptyset$, sonst wäre $M \cup \{(x, y)\}$ ein Matching. Weil M^* selbst ein Matching ist, gibt es keine weitere Kante, die x oder y enthält. Hieraus folgt $|M^*| \leq |S| = 2|M|$. □

Beispiel 5.64. Die Schranke aus Satz 5.63 ist bestmöglich, wie folgender Graph zeigt.



Im Folgenden wollen wir uns darauf beschränken, größte Matchings in sog. bipartiten Graphen zu suchen.

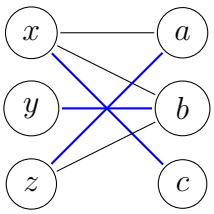
Definition 5.65. *Ein ungerichteter Graph $G = (V, E)$ heißt **bipartit** oder **zweigeteilt**, falls nicht-leere Knotenmengen V_1, V_2 existeren, sodass*

(i) $V = V_1 \cup V_2$, $V_1 \cap V_2 = \emptyset$,

(ii) für jede Kante $(v, w) \in E$ ist $\{v, w\} \cap V_1 \neq \emptyset$ und $\{v, w\} \cap V_2 \neq \emptyset$, d.h. alle Kanten verlaufen zwischen einem Knoten aus V_1 und einem Knoten aus V_2; es gibt also keine Kanten jeweils innerhalb von V_1 und V_2.

*Die Mengen V_1, V_2 heißen **Bipartition**. Ein Matching M heißt **perfekt**, falls $|M| = |V_1|$.*

Beispiel 5.66 (Arbeitsvermittlungsproblem)**.** Drei Personen $V_1 := \{x, y, z\}$ besitzen unterschiedliche Fähigkeiten, die sie für drei ausgeschriebene Stellen $V_2 := \{a, b, c\}$ qualifizieren. Ziel ist es, möglichst vielen Personen eine Stelle zu vermitteln. Wir erhalten beispielsweise den folgenden Graphen, in dem jede Kante die Eignung einer Person für eine Stelle symbolisiert.

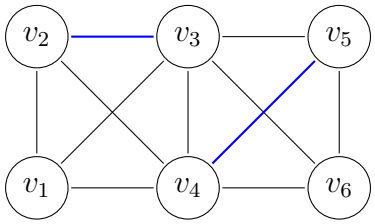


Ein größtes Matching ist $M := \{(x, c), (y, b), (z, a)\}$.

Die Bestimmung eines größten Matchings kann auf ein Flussproblem zurückgeführt werden. Die Lösung dieses Flussproblems mittels des Ford-Fulkerson-Algorithmus benötigt $O(|V| \cdot |E|)$ Aufwand. Mit dem folgenden Algorithmus von Hopcroft und Karp kann ein größtes Matching mit Aufwand $O(\sqrt{|V|} \cdot |E|)$ bestimmt werden. Um den Algorithmus vorstellen zu können, benötigen wir einige Begriffserklärungen.

Definition 5.67. *Sei $G = (V, E)$ ein ungerichteter Graph und $M \subset E$ ein Matching. Ein Pfad in G, bei dem sich Kanten $e \in M$ und Kanten $e \notin M$ abwechseln, heißt **M-alternierend**. Ein Knoten $v \in V$ heißt **frei**, falls $v \notin e$ für alle $e \in M$. Ein einfacher M-alternierender Pfand $\pi = (v_1, \ldots, v_k)$ mit freien Knoten v_1, v_k heißt **M-augmentierend**.*

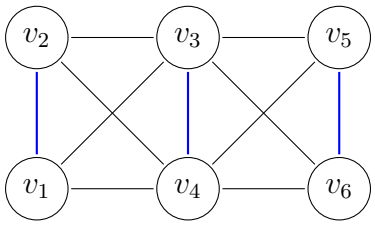
Beispiel 5.68. Wir betrachten das Matching $M = \{(v_2, v_3), (v_4, v_5)\}$ im folgenden Graphen.



Der Weg $\pi := (v_1, v_2, v_3, v_4, v_5, v_6)$ ist M-augmentierend. Betrachte das Matching

$$N := (M \setminus \pi) \cup (\pi \setminus M),$$

in dem im Vergleich zu M die Kanten $e \in \pi \cap M$ entfernt und $e \in \pi$ mit $e \notin M$ hinzugefügt werden. Dann gilt $N = \{(v_1, v_2), (v_3, v_4), (v_5, v_6)\}$ und somit $|N| = |M| + 1$.



Mit Hilfe eines M-augmentierenden Weges kann offenbar die Kardinalität des Matchings vergrößert werden.

Satz 5.69. *Sei $G = (V, E)$ ein ungerichteter Graph und $M \subset E$ ein Matching. M ist genau dann ein größtes Matching, falls es keinen M-augmentierendend Pfad gibt.*

Beweis. Wir zeigen die Negierung der Aussage. Sei $\pi = (v_1, \dots, v_k)$ ein M-augmentierender Pfad. Die Menge $N := (M \setminus \pi) \cup (\pi \setminus M)$ ist ein Matching, weil M ein Matching ist und v_1, v_k frei sind. Ferner gilt $|N| = |M| + 1$, weil die Anzahl der Kanten in π ungerade ist und $(v_1, v_2), (v_{k-1}, v_k) \notin M$. Also ist M kein größtes Matching.
Für die Gegenrichtung sei M' ein Matching mit $|M'| > |M|$. Sei $N := (M' \setminus M) \cup (M \setminus M')$ und betrachte den durch N induzierten Subgraphen G'. Dazu entferne alle Kanten, die nicht in N sind und danach alle isolierten Knoten. Die Knoten in G' haben Grad 1 oder 2, weil für einen größeren Grad mindestens zwei M-Kanten oder mindestens zwei M'-Kanten anliegen müssten, was der Matchingeigenschaft von M und M' widerspricht. Daher besteht G' aus disjunkten einfachen Pfaden oder Zyklen. Die Zyklen in G' enthalten die gleiche Anzahl von M- und M'-Kanten, weil weder zwei M-Kanten noch zwei M'-Kanten aneinander stoßen können. Weil aber $|M'| > |M|$ gilt, muss es in G' mehr M'-Kanten als M-Kanten geben. Also muss es mindestens einen einfachen Weg geben, der abwechselnd aus M- und M'-Kanten besteht und mit M'-Kante beginnt und endet. Die Anfangs- und Endknoten sind frei, weil in G keine weitere M'-Kante anliegen kann. Würde eine M-Kante anliegen, so wäre sie dann auch in G' enthalten. Dieser Pfad ist offenbar M-augmentierend. □

Aus dem letzten Satz ergibt sich ein einfacher Matching-Algorithmus. Um nur M-alternierende Wege zu beschreiten, definieren wir den gerichteten Graphen $G_M = (V_1 \cup V_2, E_M)$ mit

$$E_M = \{(u, v) : u \in V_1, v \in V_2, (u, v) \in M\} \cup \{(v, u) : u \in V_1, v \in V_2, (u, v) \in E \setminus M\}.$$

Beispiel 5.70. Wir betrachten den bipartiten Graphen $G = (V_1 \cup V_2, E)$.

Pfade in G_M sind offenbar M-alternierend.

Algorithmus 5.71 (Einfacher Matching-Algorithmus)**.**
`Input:` Bipartiter Graph $G = (V, E)$
`Output:` größtes Matching M

1: $M := \emptyset$;
2: konstruiere G_M;
3: suche mittels Breitensuche einen gerichteten Pfad π in G_M zwischen zwei freien Knoten;
4: `if` (π existiert) {
5: $M := (M \setminus \pi) \cup (\pi \setminus M)$;
6: `goto` 2;
7: }

Satz 5.72. *Algorithmus 5.71 berechnet ein größtes Matching für einen bipartiten Graphen $G = (V, E)$ mit Aufwand $O(|V| \cdot |E|)$.*

Beweis. Gerichtete Pfade in G_M zwischen freien Knoten sind offenbar M-augmentierend und umgekehrt. Wenn es einen solchen Pfad gibt, wird er auch per Breitensuche gefunden. Dies benötigt $O(|E|)$ Operationen. Die Kardinalität von M wird in jedem Schleifendurchlauf um 1 erhöht. Die Kardinalität eines größten Matchings ist durch $\lfloor \frac{1}{2}|V| \rfloor$ nach oben beschränkt. □

Dieser einfache Algorithmus soll nun verbessert werden, so dass ein Aufwand von der Ordnung $\sqrt{|V|}|E|$ entsteht. Dazu wird das Matching in jedem Schritt durch die Augmentierung von mehreren knotendisjunkten Pfaden vergrößert. Wir verwenden im Folgenden die Notation $M_1 \oplus M_2 := (M_1 \setminus M_2) \cup (M_2 \setminus M_1)$. Die Bedeutung der beiden folgenden Lemmata wird erst bei der Komplexitätsanalyse des Hopcroft-Karp-Algorithmus klar werden.

Lemma 5.73. *Sei $G = (V, E)$ ein ungerichteter Graph und $M \subset E$ ein Matching. Sei π ein kürzester M-augmentierender Pfad $N := M \oplus \pi$. Ist π' ein N-augmentierender Pfad, so gilt*

$$|\pi'| \geq |\pi| + |\pi \cap \pi'|$$

Beweis. Betrachte $N' := N \oplus \pi'$. Dann gilt $|N'| = |N| + 1 = |M| + 2$. Wegen

$$N'' := M \oplus N' = M \oplus ((M \oplus \pi) \oplus \pi') = \pi \oplus \pi'$$

enthält N'' zwei M-augmentierende Pfade π_1, π_2. Folglich ist $|N''| \geq |\pi_1| + |\pi_2|$. Weil π ein kürzester M-augmentierender Weg ist, gilt $|\pi| \leq \min\{|\pi_1|, |\pi_2|\}$. Somit ist $|N''| \geq 2|\pi|$. Andererseits gilt $|\pi \oplus \pi'| = |\pi| + |\pi'| - |\pi \cap \pi'|$. Hieraus folgt $|\pi| + |\pi'| - |\pi \cap \pi'| \geq 2|\pi|$. □

Das folgende Lemma verwendet die im letzen Lemma gemachte Beobachtung für eine Folge augmentierender Wege.

Lemma 5.74. *Sei $G = (V, E)$ ein ungerichteter Graph. Sei $M_0 := \emptyset$ und $M_{i+1} := M_i \oplus \pi_i$, $i > 0$, wobei π_i ein kürzester M_i-augmentierender Weg ist. Dann gilt*

(i) $|\pi_i| \leq |\pi_{i+1}|$,

(ii) $|\pi_i| = |\pi_j|$, $i \neq j \implies \pi_i$ *und* π_j *sind knotendisjunkt.*

Beweis. Der erste Teil folgt aus Lemma 5.73. Für den zweiten Teil nehmen wir an, es gäbe zwei nicht-knotendisjunkte Pfade π, π_j, $i \neq j$ mit $|\pi_i| = |\pi_j|$. Dann haben nach (i) auch alle Pfade π_ν, $i \leq \nu \leq j$, diese Länge. Wir wählen zwei Indizes k, l mit $i \leq k < \ell \leq j$, so dass π_k und π_ℓ nicht knotendisjunkt aber alle π_ν, $k < \nu < \ell$, knotendisjunkt zu π_k und π_ℓ sind. Die Existenz solcher Indizes ist gesichert, weil im Zweifel für $k = \ell - 1$ die Bedingung für ν trivial wird. Die Wahl von k und ℓ stellt sicher, dass π_ℓ ein M_{k+1}-augmentierender Pfad ist. Sei $v \in V(\pi_k) \cap V(\pi_l)$. Die zu v inzidente Kante aus M_{k+1} ist in π_ℓ und in π_k enthalten, weil sonst π_ℓ nicht M_{k+1}-augmentierend sein kann. Also ist $\pi_k \cap \pi_\ell \neq \emptyset$. Nach Lemma 5.73 folgt der Widerspruch $|\pi_\ell| \geq |\pi_k| + |\pi_k \cap \pi_\ell| > |\pi_k|$. □

Im letzten Beweis haben wir bereits verwendet, dass knotendisjunkte M-augmentierende Pfade gleichzeitig zu M "addiert" werden können, ohne dass diese sich gegenseitig beeinflussen. Wir können also gefahrlos $|M|$ um mehr als 1 in jedem Schleifendurchlauf vergrößern.

Algorithmus 5.75 (Hopcroft-Karp-Algorithmus)**.**
`Input:` Bipartiter Graph $G = (V, E)$
`Output:` größtes Matching M

1: $M := \emptyset$;
2: konstruiere G_M;
3: Berechne eine maximale Menge $\pi_1, \ldots, \pi_k$ knotendisjunkter kürzester Wege in G_M zwischen zwei freien Knoten;
4: `if` $(k \geq 1)$ {
5: $\quad M := M \oplus \pi_1 \oplus \pi_2 \oplus \ldots \oplus \pi_k$;
6: $\quad$ goto 2;
7: }

Bemerkung. Um jeden Schleifendurchlauf effizient durchführen zu können, fügen wir G_M zwei Knoten s und t und alle Kanten (s, v), (w, t) mit freien $v \in V_1$ bzw. freien $w \in V_2$ hinzu. Ausgehend von s wird eine Breitensuche durchgeführt, und alle Kanten, die nicht auf einem kürzesten Weg nach t liegen, werden entfernt. Offenbar sind dann alle kürzesten M-augmentierenden Pfade in diesem Graphen enthalten. Um sicher zu stellen, dass wir knotendisjunkte M-augmentierende Pfade erhalten, entfernen wir alle Kanten und Knoten eines bereits bestimmten kürzesten Weges. Dies garantiert einen Aufwand von der Ordnung $|E|$ für Zeile 3 von Algorithmus 5.75.

Satz 5.76. *Der Algorithmus von Hopcraft und Karp berechnet ein größtes Matching für einen bipartiten Graphen mit Aufwand* $O(\sqrt{|V|} \cdot |E|)$.

Beweis. Die Korrektheit haben wir schon in Satz 5.69 behandelt. Wir zeigen, dass $O(\sqrt{|V|})$ Schleifendurchläufe nötig sind. Dann ergibt sich die Komplexitätsabschätzung aus der letzten Bemerkung.

Betrachte nochmals den Graphen G' aus dem Beweis zu Satz 5.69. G' entsteht aus M und M', wobei M unser aktuelles Matching und M' ein größtes Matching bezeichnen. Seien $C_i := (V_i, E_i)$ die Zusammenhangskomponenten von G', die wir als einfache Wege oder Zyklen identifiziert hatten. Wir hatten auch gesehen, dass

$$\delta_i := |E_i \cap M'| - |E_i \cap M| \ \in \ \{-1, 0, 1\}.$$

C_i ist genau dann M-augmentierend, wenn $\delta_i = 1$. Weil aber

$$\sum_i \delta_i = |M'| - |M|,$$

muss es mindestens $|M'| - |M|$ knotendisjunkte M-augmentierende Pfade geben. Diese Pfade enthalten zusammen höchstens $|M|$ Kanten aus M. Daher muss ein M-augmentierender Weg existieren, der höchstens $\ell := \left\lfloor \frac{|M|}{|M'|-|M|} \right\rfloor$ Kanten aus M enthält. Weil M- und M'-Kanten alternieren, ist die Länge dieses Pfades durch $2\ell + 1$ nach oben beschränkt. Wir unterscheiden nun zwei Phasen des Hopcroft-Karp-Algorithmus. In der ersten Phase gelte

$|M| \leq \left\lfloor |M'| - \sqrt{|M'|} \right\rfloor$. Dann ist

$$2\ell + 1 \leq 2 \left\lfloor \frac{\left\lfloor |M'| - \sqrt{|M'|} \right\rfloor}{|M'| - \left\lfloor |M'| - \sqrt{|M'|} \right\rfloor} \right\rfloor + 1 = 2 \left\lfloor \frac{|M'| - \left\lceil \sqrt{|M'|} \right\rceil}{\left\lceil \sqrt{|M'|} \right\rceil} \right\rfloor + 1$$
$$= 2 \left\lfloor \frac{|M'|}{\left\lceil \sqrt{|M'|} \right\rceil} - 1 \right\rfloor + 1 \leq 2\sqrt{|M'|} - 1.$$

Für eine Abschätzung der Anzahl der Schritte ist eine obere Schranke für die Länge kürzester M-augmentierender Pfade ausreichend, weil nach Lemma 5.74 die Länge der Pfade bei jedem Schritt wächst. Daher ist nach $O(\sqrt{|M'|})$ Schleifendurchläufen die erste Phase beendet. In der zweiten Phase ist $|M|$ nur noch um höchstens $\sqrt{|M'|}$ von der Kardinalität eines größten Matchings entfernt. Weil jeder Schleifendurchlauf $|M|$ um mindestens 1 erhöht, kann auch die zweite Phase nicht mehr als $O(\sqrt{|M'|})$ Schleifendurchläufe benötigen. Wegen $|M'| \leq \frac{n}{2}$ folgt die Behauptung. □

Bemerkung. Wir haben uns beim Matching-Problem auf bipartite Graphen konzentriert. Will man die Inhalte dieses Abschnitts auf allgemeine Graphen anwenden, so stellt man fest, dass sowohl Satz 5.69 als auch Lemma 5.74 auch weiterhin gelten. Ein Problem tritt ausschließlich bei der effizienten Implementierung der Zeile 3 von Algorithmus 5.75 auf. Der Algorithmus von Micali und Vazirani verallgemeinert den Hopcroft-Karp-Algorithmus auf allgemeine Graphen und besitzt die gleiche asymptotischen Komplexität.

6 Lineare Gleichungssysteme

Im Folgenden bezeichne $\mathbb{K}^{m\times n}$, $\mathbb{K}\in\{\mathbb{R},\mathbb{C}\}$, den Raum der $m\times n$-Matrizen

$$A=\begin{bmatrix} a_{11} & \dots & a_{1n}\\ \vdots & & \vdots\\ a_{m1} & \dots & a_{mn}\end{bmatrix},\quad a_{ij}\in\mathbb{K}.$$

Dabei verwenden wir die komponentenweise Addition und Multiplikation mit Skalaren. Für $k=1$ ergibt sich der Raum der Vektoren $\mathbb{K}^m$.

Beispiel 6.1. Eine Matrix $A\in\mathbb{K}^{m\times n}$ heißt **Diagonalmatrix**, falls $a_{ij}=0$ für $i\neq j$. Die Matrix $I\in\mathbb{K}^{n\times n}$ mit

$$I_{ij}=\begin{cases}1, & i=j,\\ 0, & \text{sonst},\end{cases}$$

wird als **Identität** oder **Einheitsmatrix** bezeichnet. Die Vektoren $e_i\in\mathbb{K}^n$ mit

$$(e_i)_j=\begin{cases}1, & i=j,\\ 0, & \text{sonst},\end{cases}$$

heißen **kanonische Einheitsvektoren**.

Zwischen Matrizen kann ein Produkt $\cdot:\mathbb{K}^{m\times p}\times\mathbb{K}^{p\times n}\to\mathbb{K}^{m\times n}$ durch

$$(A\cdot B)_{ij}=\sum_{\ell=1}^{p}a_{i\ell}b_{\ell i},\quad i=1,\dots,m,\ j=1,\dots,n,$$

für $A\in\mathbb{K}^{m\times p}$, $B\in\mathbb{K}^{p\times n}$ definiert werden. Es gilt Assoziativität $(A\cdot B)\cdot C=A\cdot(B\cdot C)$. Den Multiplikationspunkt $\cdot$ lassen wir auch weg.

Die Matrix $A^T\in\mathbb{K}^{n\times m}$ mit den Einträgen $(A^T)_{ij}=a_{ji}$ wird als die zu $A\in\mathbb{K}^{m\times n}$ **transponierte Matrix** bezeichnet. A^H mit $(A^H)_{ij}=\overline{a}_{ji}$ wird die zu A **adjungierte Matrix** genannt. Gilt $A^T=A$ bzw. $A^H=A$, so heißt A **symmetrisch** bzw. **hermitesch**. Ferner gelten die Rechenregeln $(AB)^T=B^TA^T$, $(AB)^H=B^HA^H$, $(A+\lambda B)^T=A^T+\lambda B^T$, $(A+\lambda B)^H=A^H+\overline{\lambda}B^H$ für $\lambda\in\mathbb{K}$ sowie $(A^T)^T=A=(A^H)^H$.

Lineare algebraische Gleichungssysteme

$$Ax=b \tag{6.1}$$

mit gegebener Systemmatrix $A\in\mathbb{K}^{m\times n}$ und rechter Seite $b\in\mathbb{K}^m$ tauchen sehr oft als Endproblem in vielen Anwendungen auf. (6.1) ist offenbar genau dann lösbar, wenn $b\in\operatorname{Im}A$. Sei $x\in\mathbb{K}^n$ eine Lösung, dann ist die Lösungsmenge von (6.1) gegeben durch $x+\ker A:=\{x+y, y\in\ker A\}$. Insbesondere ist, falls $\operatorname{rank}A=\dim\operatorname{Im}A=n$, x die eindeutige Lösung von (6.1). Quadratische Matrizen $A\in\mathbb{K}^{n\times n}$ heißen **invertierbar**, falls $B\in\mathbb{K}^{n\times n}$ existiert mit $AB=BA=I$. Daher wird $B=A^{-1}$ als **Inverse** von A bezeichnet. A ist genau dann invertierbar, wenn $\operatorname{rank}A=n$ ist.

Eine weitere Charakterisierung der Lösbarkeit von (6.1) ergibt sich aus dem folgendem Lemma, in dem der **Orthogonalraum**

$$X^\perp = \{y \in \mathbb{K}^n : x^H y = 0 \text{ für alle } x \in X\}$$

von $X \subset \mathbb{K}^n$ verwendet wird. Man beachte, dass $(X^\perp)^\perp = X$.

Lemma 6.2. *Sei $A \in \mathbb{K}^{m\times n}$. Dann gilt $(\operatorname{Im} A)^\perp = \ker A^H$.*

Beweis. Sei $x \in \ker A^H$, d.h. $A^H x = 0$ und sei $z \in \operatorname{Im} A$. Dann existiert $y \in \mathbb{K}^n$ mit $z = Ay$. Dann ist $z^H x = (Ay)^H x = y^H(A^H x) = 0$ und somit $x \in (\operatorname{Im} A)^\perp$.
Mit der Umkehrung des Arguments erhält man die verbleibende Inklusion. □

Theorem 6.3. Das Gleichungssystem (6.1) ist genau dann lösbar, falls $b^H y = 0$ für alle $y \in \mathbb{K}^m$ mit $A^H y = 0$.

Beweis. Nach dem letzten Lemma gilt $\operatorname{Im} A = ((\operatorname{Im} A)^\perp)^\perp = (\ker A^H)^\perp$. □

6.1 Vektor- und Matrixnormen

Definition 6.4. *Eine Abbildung $\|\cdot\| : \mathbb{K}^n \to \mathbb{R}$ heißt **Vektornorm**, falls*

(i) $\|x\| > 0$ *für alle* $x \in \mathbb{K}^n \setminus \{0\}$, *(Positivität)*

(ii) $\|\lambda x\| = |\lambda|\,\|x\|$ *für alle* $x \in \mathbb{K}^n$, $\lambda \in \mathbb{K}$, *(Homogenität)*

(iii) $\|x + y\| \le \|x\| + \|y\|$ *für alle* $x, y \in \mathbb{K}^n$. *(Dreiecksungleichung)*

Bemerkung. Der Ausdruck $d(x, y) := \| x - y \|$ ist ein Maß für den Abstand von x und y. Außerdem gilt die **umgekehrte Dreiecksungleichung**

$$|\,\|x\| - \|y\|\,| \le \|x - y\|,$$

die aus $\|x\| = \|x - y + y\| \le \|x - y\| + \|y\|$ und $\|y\| = \|y - x + x\| \le \|x - y\| + \|x\|$ folgt.

Beispiel 6.5.

(a) **Betragssummennorm**

$$\|x\|_1 := \sum_{i=1}^{n} |x_i|$$

und **euklidische Norm**

$$\|x\|_2 := \sqrt{x^H x} = \sqrt{\sum_{i=1}^{n} |x_i|^2}$$

sind Spezialfälle der ***p*-Normen**

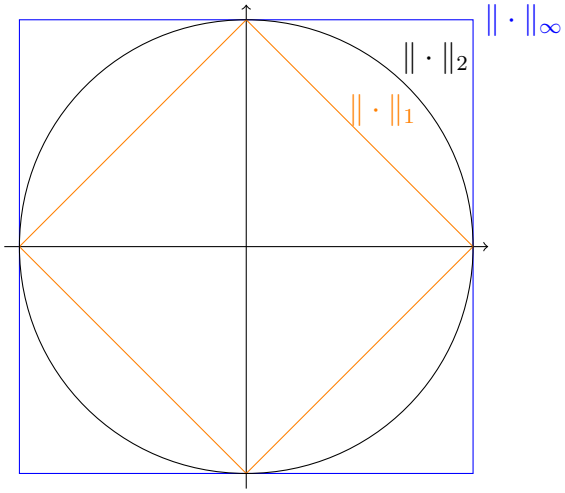
$$\|x\|_p = \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p}, \quad p \in \mathbb{N},$$

auf $\mathbb{K}^n$.

(b) Der punktweise Grenzwert von $\|\cdot\|_p$ für $p \to \infty$ ergibt die **Maximumnorm**

$$\|x\|_\infty := \max_{i=1,\dots,n} |x_i|.$$

Die folgende Abbildung zeigt die Einheitssphäre $S_p := \{x \in \mathbb{R}^2 : \|x\|_p = 1\}$ der p-Norm.



(c) Ist $\|\cdot\|$ eine Vektornorm, so ist auch $\|x\|_T := \|Tx\|$ mit $T \in \mathbb{K}^{n\times n}$, $\operatorname{rank} A = n$, eine Vektornorm.

Satz 6.6. *Alle Normen auf $\mathbb{K}^n$ sind äquivalent, d.h. für jeweils zwei Normen $\|\cdot\|$ und $\|\cdot\|'$ auf $\mathbb{K}^n$ existieren positive Konstanten $c, C > 0$ mit $c\|x\| \le \|x\|' \le C\|x\|$ für alle $x \in \mathbb{K}^n$.*

Beweis. Es genügt, die Behauptung für $\|\cdot\|' = \|\cdot\|_\infty$ und eine beliebige Norm $\|\cdot\|$ zu zeigen. Dazu seien $x, y \in \mathbb{K}^n$. Wegen

$$x - y = \sum_{i=1}^{n} (x_i - y_i) e_i$$

folgt aus der umgekehrten Dreiecksungleichung

$$| \|x\| - \|y\| | \le \|x - y\| \le \sum_{i=1}^{n} |x_i - y_i| \|e_i\| \le \|x - y\|_\infty \sum_{i=1}^{n} \|e_i\|.$$

Folglich ist $\|\cdot\| : (\mathbb{K}^n, \|\cdot\|_\infty) \to \mathbb{R}$ eine Lipschitz-stetige[1] Funktion mit Lipschitz-Konstante $L = \sum_{i=1}^{n} \|e_i\|$. Diese nimmt auf der kompakten Einheitssphäre $\{x \in \_K^n : \|x\|_\infty = 1\}$

[1] Eine Funktion $f : (\mathbb{K}^n, \|\cdot\|) \to \mathbb{R}$ heißt Lipschitz-stetig, falls $L > 0$ existiert mit $|f(x) - f(y)| \le L\|x - y\|$.

sowohl ihr Maximum C wie auch ihr Minimum c an. Wegen der Positivität der Norm ist $C \geq c > 0$. Also gilt für alle $z \in \mathbb{K}^n$

$$c \leq \left\| \frac{z}{\|z\|_\infty} \right\| \leq C \iff c\|z\|_\infty \leq \|z\| \leq C\|z\|_\infty.$$

□

Beispiel 6.7. Es sei $p \leq q$. Dann gilt für $x \in \mathbb{K}^n$

$$\|x\|_q \leq \|x\|_p \leq n^{\frac{1}{p}-\frac{1}{q}}\|x\|_q.$$

Speziell gilt

$$\|x\|_\infty \leq \|x\|_2 \leq \|x\|_1 \leq \sqrt{n}\|x\|_2 \leq n\|x\|_\infty, \quad x \in \mathbb{K}^n.$$

Definition 6.8. *Eine Abbildung $\|\cdot\| : \mathbb{K}^{m\times n} \to \mathbb{R}$ heißt **Matrixnorm**, falls*

(i) $\|A\| > 0$ *für alle* $A \in \mathbb{K}^{m\times n} \setminus \{0\}$,

(ii) $\|\lambda A\| = |\lambda|\,\|A\|$ *für alle* $A \in \mathbb{K}^{m\times n}$, $\lambda \in \mathbb{K}$,

(iii) $\|A + B\| \leq \|A\| + \|B\|$ *für alle* $A, B \in \mathbb{K}^{m\times n}$,

(iv) $\|AB\| \leq \|A\| \cdot \|B\|$ *für alle* $A \in \mathbb{K}^{m\times p}$, $B \in \mathbb{K}^{p\times n}$. *(Submultiplikativität)*

Beispiel 6.9.

(a) Auf $\mathbb{K}^{m\times n}$ definiere analog zur Maximumnorm die Abbildung

$$\|A\|_M := \max_{\substack{i=1,\dots,m \\ j=1,\dots,n}} |a_{ij}|.$$

Diese ist eine Vektornorm auf $\mathbb{K}^{m\cdot n}$ aber keine Matrixnorm. Denn für

$$A = B = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

gilt

$$AB = \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}$$

und somit $\|AB\|_M = 2 > 1 = \|A\|_M \cdot \|B\|_M$

(b) **Spaltensummenorm**

$$\|A\|_1 := \max_{j=1,\dots,n} \sum_{i=1}^{m} |a_{ij}|,$$

Zeilensummenorm

$$\|A\|_\infty := \max_{i=1,\dots,m} \sum_{j=1}^{n} |a_{ij}|$$

und **Frobeniusnorm**

$$\|A\|_F := \left(\sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|^2 \right)^{1/2}$$

sind Matrixnormen auf $\mathbb{K}^{m\times n}$. Man beachte $\|A^H\|_1 = \|A\|_\infty$.

Wir zeigen die Submultiplikativität der Frobeniusnorm. Sei dazu $A \in \mathbb{K}^{m\times p}$, $B \in \mathbb{K}^{p\times n}$. Dann gilt unter Verwendung der **Cauchy-Schwarzschen-Ungleichung**

$$\left| \sum_{i=1}^{n} x_i y_i \right| \le \left(\sum_{i=1}^{n} |x_i|^2 \right)^{1/2} \left(\sum_{i=1}^{n} |y_i|^2 \right)^{1/2}, \quad x, y \in \mathbb{K}^n,$$

dass

$$\|AB\|_F^2 = \sum_{i=1}^{m} \sum_{j=1}^{n} |(AB)_{ij}|^2 = \sum_{i=1}^{m} \sum_{j=1}^{n} \left| \sum_{\ell=1}^{p} a_{i\ell} b_{\ell j} \right|^2 \tag{6.2a}$$

$$\le \sum_{i=1}^{m} \sum_{j=1}^{n} \left(\sum_{\ell=1}^{p} |a_{i\ell}|^2 \right) \left(\sum_{\ell=1}^{p} |b_{\ell j}|^2 \right) \tag{6.2b}$$

$$= \|A\|_F^2 \cdot \|B\|_F^2. \tag{6.2c}$$

Definition 6.10. *Sind Vektornormen $\|\cdot\|$ und $\|\cdot\|'$ auf $\mathbb{K}^m$ bzw. $\mathbb{K}^n$ gegeben, so definiert man die* ***zugeordnete Norm*** *(oder induzierte Norm) auf $\mathbb{K}^{m\times n}$ durch*

$$|||A||| := \sup_{x\neq 0} \frac{\|Ax\|}{\|x\|'} = \max_{\|x\|'=1} \|Ax\|.$$

Eine Abbildung $f : \mathbb{K}^{m\times n} \to \mathbb{R}$ heißt ***verträglich*** *mit $\|\cdot\|$ und $\|\cdot\|'$, falls*

$$\|Ax\| \le f(A) \cdot \|x\|' \quad \text{für alle } A \in \mathbb{K}^{m\times n},\ x \in \mathbb{K}^n.$$

Beispiel 6.11.

(i) $\|\cdot\|_M$ aus Beispiel 6.9 (a) ist auf $\mathbb{K}^{m\times n}$ den Vektornormen $\|\cdot\|_\infty$ und $\|\cdot\|_1$ auf $\mathbb{K}^m$ bzw. $\mathbb{K}^n$ zugeordnet. Es gilt nämlich

$$\|Ax\|_\infty = \max_{i=1,\dots,m} |\sum_{j=1}^{n} a_{ij} x_j| \le \max_{i=1,\dots,m} \sum_{j=1}^{n} |a_{ij}|\,|x_j|$$

$$\le \max_{\substack{i=1,\dots,m \\ j=1,\dots,n}} |a_{ij}| \sum_{j=1}^{n} |x_j| = \|A\|_M \|x\|_1 \quad \text{für alle } x \in \mathbb{K}^n.$$

Sei (i_0, j_0) so gewählt, dass $|a_{i_0 j_0}| = \|A\|_M$. Dann wird das Maximum für $x := e_{j_0}$ angenommen

$$\|Ae_{j_0}\| = \max_{i=1,\dots,m} |a_{ij_0}| = |a_{i_0 j_0}| = \|A\|_M \|e_{j_0}\|_1.$$

Hieraus folgt $\|A\|_M = \max_{\|x\|_1=1} \|Ax\|_\infty$.

(ii) Die Frobeniusnorm ist mit der euklidischen Vektornorm verträglich. Dies folgt aus der Submultiplikativität (6.2) für den Fall $B = x \in \mathbb{K}^n$. Sie ist aber keine einer Vektornorm zugeordnete Norm, weil für einer Vektornorm zugeordnete Norm gilt $|||I||| = 1$ mit $I \in \mathbb{K}^{n\times n}$. Für die Frobeniusnorm gilt aber $\|I\|_F = \sqrt{n}$.

Lemma 6.12. *Sei $|||\cdot|||$ die $\|\cdot\| = \|\cdot\|'$ zugeordnete Norm. Dann ist $|||\cdot|||$ eine mit $\|\cdot\|$ verträgliche Matrixnorm. Ist $f : \mathbb{K}^{m\times n} \to \mathbb{R}$ eine mit $\|\cdot\|$ verträgliche Abbildung, so gilt $|||A||| \le f(A)$ für alle $A \in \mathbb{K}^{m\times n}$.*

Beweis. Die Aussage gilt offenbar für $0 = B \in \mathbb{K}^{p\times n}$. Daher sei $A \in \mathbb{K}^{m\times p}$ und $B \neq 0$. Dann gilt

$$\begin{aligned}
|||AB||| &= \sup_{x\neq 0} \frac{\|ABx\|}{\|x\|} = \sup_{Bx\neq 0} \frac{\|ABx\|}{\|x\|} = \sup_{Bx\neq 0} \frac{\|ABx\|}{\|Bx\|} \cdot \frac{\|Bx\|}{\|x\|} \\
&\le \sup_{Bx\neq 0} \frac{\|ABx\|}{\|Bx\|} \cdot \sup_{Bx\neq 0} \frac{\|Bx\|}{\|x\|} \le \sup_{y\neq 0} \frac{\|Ay\|}{\|y\|} \cdot \sup_{Bx\neq 0} \frac{\|Bx\|}{\|x\|} \\
&\le \sup_{y\neq 0} \frac{\|Ay\|}{\|y\|} \cdot \sup_{x\neq 0} \frac{\|Bx\|}{\|x\|} = |||A||| \cdot |||B|||.
\end{aligned}$$

Die übrigen Eigenschaften einer Matrixnorm sind offensichtlich.

Die Verträglichkeit mit $\|\cdot\|$ erhält man aus

$$|||A||| = \sup_{x\neq 0} \frac{\|Ax\|}{\|x\|} \ge \frac{\|Ax\|}{\|x\|} \qquad \text{for all } x \in \mathbb{K}^n \setminus \{0\}.$$

Hieraus folgt $\|Ax\| \le |||Ax|||\|x\|$ für alle $x \in \mathbb{K}^n$, weil die letzte Abschätzung für $x = 0$ trivialerweise gilt.

Der letzte Teil der Aussage ergibt sich aus

$$|||A||| = \sup_{x\neq 0} \frac{\|Ax\|}{\|x\|} \le \sup_{x\neq 0} \frac{f(A)\|x\|}{\|x\|} = f(A).$$

□

Bemerkung. Die Annahme $\|\cdot\| = \|\cdot\|'$ im letzten Lemma ist wichtig, weil $\|\cdot\|_M$ zwar durch $\|\cdot\|_\infty$ und $\|\cdot\|_1$ induziert (siehe Beispiel 6.11) aber nach Beispiel 6.9 (a) keine Matrixnorm ist.

In den Übungsaufgaben werden wir sehen, dass die Spaltensummen- und die Zeilensummennorm jeweils den Vektornormen $\|\cdot\|_1$ bzw. $\|\cdot\|_\infty$ zugeordnet sind. Da die Frobeniusnorm keine zugeordnete Norm ist und somit der euklidischen Norm nicht zugeordnet sein kann, stellt sich die Frage, welche Matrixnorm der euklidischen Vektornorm zugeordnet ist. Im Folgenden benötigen wir für die Charakterisierung der der euklidischen Vektornorm zugeordneten Matrixnorm $\|\cdot\|_2$ einige Aussagen aus der linearen Algebra.

Sei $\mathcal{P}_n$ die Menge der Permutationen (siehe Definition 4.1) über $\{1,\dots,n\}$. Ist $\pi \in \mathcal{P}_n$, so wird ein Paar $(i,j) \in \{1,\dots,n\} \times \{1,\dots,n\}$ mit $i < j$ und $\pi(i) > \pi(j)$ als **Fehlstand** bezeichnet. Man definiert das **Signum** einer Permutation $\pi \in \mathcal{P}_n$ durch

$$\operatorname{sign} \pi = \begin{cases} +1, & \text{falls die Anzahl der Fehlstände von } \pi \text{ gerade ist,} \\ -1, & \text{falls die Anzahl der Fehlstände von } \pi \text{ ungerade ist.} \end{cases}$$

Die **Determinante** einer Matrix $A \in \mathbb{K}^{n\times n}$ ist definiert durch

$$\det A = \sum_{\pi\in\mathcal{P}_n} (\operatorname{sign}\pi) \prod_{i=1}^{n} a_{i\pi(i)}.$$

Eine **reguläre** Matrix erfüllt $\det A \neq 0$. Im anderen Fall wird A als **singulär** bezeichnet. $A \in \mathbb{K}^{n\times n}$ ist genau dann regulär, wenn A invertierbar ist.

Aus der Definition der Determinanten folgt, dass $\det A^T = \det A$ und $\det A^H = \overline{\det A}$. Ferner kann gezeigt werden, dass für $A, B \in \mathbb{K}^{n\times n}$ gilt $\det AB = \det A \cdot \det B$ und $\det A^{-1} = (\det A)^{-1}$, falls A regulär ist.

Wir definieren (vgl. auch Kapitel 3)

$$\chi_A(\lambda) := \det(A - \lambda I)$$

als **charakteristisches Polynom**. Jede Zahl $\lambda \in \mathbb{C}$ heißt **Eigenwert** von $A \in \mathbb{K}^{n\times n}$, falls $\chi_A(\lambda) = 0$. Eine $n \times n$-Matrix besitzt n Eigenwerte.

Bemerkung. Weil für einen Eigenwert λ gilt $\det(A - \lambda I) = 0$, ist $\ker A - \lambda I$ nicht trivial. Daher existiert ein sog. **Eigenvektor** $0 \neq x \in \mathbb{K}^{n\times n}$ mit $Ax = \lambda x$. Wegen

$$A^k x = \lambda A^{k-1} x = \ldots = \lambda^k x$$

ist insbesondere λ^k Eigenwert von A^k.

Im folgenden Lemma verwenden wir auch die **Spur** einer Matrix $A \in \mathbb{K}^{n\times n}$ definiert durch

$$\operatorname{trace} A = \sum_{i=1}^{n} a_{ii}.$$

Man rechnet leicht nach, dass $\|A\|_F^2 = \operatorname{trace} A^H A$ gilt.

Lemma 6.13. *Seien $\lambda_1, \ldots, \lambda_n$ die Eigenwerte von $A \in \mathbb{K}^{n\times n}$. Dann gilt*

$$\operatorname{trace} A = \sum_{i=1}^{n} \lambda_i \quad \textit{und} \quad \det A = \prod_{i=1}^{n} \lambda_i.$$

Beweis. Es gilt $\chi_A(0) = \det A$. Daher stimmt der absolute Term der Polynoms χ_A mit $\det A$ überein. Man sieht ferner leicht, dass der Koeffizient vor λ^{n-1} mit der Spur $\operatorname{trace} A$ übereinstimmt. Auf der anderen Seite gilt

$$\chi_A(\lambda) = (\lambda - \lambda_1) \cdot \ldots \cdot (\lambda - \lambda_n).$$

Durch Koeffizientenvergleich ergibt sich die Behauptung. □

Definition 6.14. *Eine Matrix $A \in \mathbb{K}^{n\times n}$ heißt*

- ***normal**, falls $A^H A = A A^H$,*
- ***unitär** (im Fall $\mathbb{K} = \mathbb{R}$ orthogonal), falls $AA^H = A^H A = I$,*

- ***ähnlich*** *zu $B \in \mathbb{K}^{n\times n}$, falls $T \in \mathbb{K}^{n\times n}$ regulär existiert, so dass $A = TBT^{-1}$,*
- ***diagonalisierbar****, falls A ähnlich zu einer Diagonalmatrix ist,*
- ***unitär diagonalisierbar****, falls A diagonalisierbar mit unitärer Matrix T ist.*

Bemerkung.

(a) Ist A unitär, so ist A insbesondere invertierbar, denn $B := A^H$ erfüllt $AB = BA = I$.

(b) Seien A und B ähnlich. Dann gilt $A = TBT^{-1}$. Unter Verwendung der Rechenregeln der Determinanten folgt

$$\begin{aligned}\chi_A(\lambda) &= \det(A - \lambda I) = \det(TBT^{-1} - \lambda I) = \det(T(B - \lambda I)T^{-1}) \\ &= (\det T)(\det T^{-1}) \det(B - \lambda I) = \chi_B(\lambda).\end{aligned}$$

Daher besitzen A und B dieselben Eigenwerte.

(c) Hermitesche Matrizen sind normal.

Wir geben ohne Beweis an:

Satz 6.15. *$A \in \mathbb{K}^{n\times n}$ ist genau dann normal, falls A unitär diagonalisierbar ist.*

Definition 6.16. *Eine Matrixnorm $\|\cdot\|$ auf $\mathbb{K}^{m\times n}$ heißt* ***unitär invariant****, falls*

$$\|A\| = \|PAQ\|$$

für alle $A \in \mathbb{K}^{m\times n}$ und alle unitären Matrizen $P \in \mathbb{K}^{m\times m}$, $Q \in \mathbb{K}^{n\times n}$.

Satz 6.17. *Die Frobeniusnorm und die der euklidischen Vektornorm zugeordnete Matrixnorm $\|\cdot\|_2$ sind unitär invariant.*

Beweis. Seien $P \in \mathbb{K}^{m\times n}$ und $Q \in \mathbb{K}^{n\times n}$ unitär. Wegen $\|Px\|_2^2 = x^H P^H Px = x^H x = \|x\|_2^2$ folgt

$$\|PAQ\|_2 = \max_{\|x\|_2=1} \|PAQx\|_2 = \max_{\|x\|_2=1} \|AQx\|_2.$$

Weil Q unitär ist, gilt nach der Bemerkung zu Definition 6.14, dass $\operatorname{rank} Q = n$ und somit $\operatorname{Im} Q = \mathbb{K}^n$. Daher können wir das Maximum für $y = Qx$ bilden. Dann folgt aus $\|y\|_2 = \|x\|_2$

$$\|PAQ\|_2 = \max_{\|y\|_2=1} \|Ay\|_2 = \|A\|_2.$$

Für die unitäre Invarianz der Frobeniusnorm verwenden wir $\|A\|_F^2 = \operatorname{trace} A^H A$. Daher gilt

$$\|PAQ\|_F^2 = \operatorname{trace} Q^H A^H P^H PAQ = \operatorname{trace} Q^H A^H AQ.$$

Nach Lemma 6.13 stimmt $\operatorname{trace} A$ mit der Summe der Eigenwerte von A überein. Weil $Q^H A^H AQ$ ähnlich zu $A^H A$ ist, besitzen entsprechend der Bemerkung nach Definition 6.14 $Q^H A^H AQ$ und $A^H A$ dieselben Eigenwerte. Daher folgt $\operatorname{trace} Q^H A^H AQ = \operatorname{trace} A^H A$. □

Der folgende Satz erklärt die Bezeichnung **Spektralnorm** für die der euklidischen Vektornorm zugeordnete Matrixnorm $\|\cdot\|_2$.

Satz 6.18. *Für $A \in \mathbb{K}^{n\times n}$ gilt*

$$\|A\|_2 = \rho^{1/2}(A^H A),$$

wobei $\rho(A) := \max\{|\lambda| : \lambda \text{ ist Eigenwert von } A\}$ den Spektralradius von A bezeichnet. Ist $A \in \mathbb{K}^{n\times n}$ normal, so gilt $\|A\|_2 = \rho(A)$.

Beweis. Für $A \in \mathbb{K}^{m\times n}$ gilt

$$\|A\|_2^2 = \max_{\|x\|_2=1} \|Ax\|_2^2 = \max_{\|x\|_2=1} x^H A^H A x.$$

Die Matrix $A^H A$ ist offenbar hermitesch und damit normal. Nach Satz 6.15 ist sie unitär diagonalisierbar, d.h. es existiert $Q \in \mathbb{K}^{n\times n}$ unitär und $\Lambda \in \mathbb{K}^{n\times n}$ diagonal mit

$$A^H A = Q\Lambda Q^H.$$

Daher folgt mit $y = Q^H x$

$$x^H A^H A x = x^H Q\Lambda Q^H x = y^H \Lambda y = \sum_{i=1}^{n} \Lambda_{ii}|y_i|^2.$$

Die Diagonalmatrix Λ enthält nach der Bemerkung zu Definition 6.14 die Eigenwerte von $A^H A$. Also gilt

$$\|Ax\|_2^2 \le \rho(A^H A)\sum_{i=1}^{n} |y_i|^2 = \rho(A^H A)\|y\|_2^2 = \rho(A^H A)\|x\|_2^2.$$

Sei $i_0 \in \mathbb{N}$ ein Index mit $|\Lambda_{i_0 i_0}| = \rho(A^H A)$. Dann gilt mit $x = Qe_{i_0}$, $\|x\|_2 = 1$, und der Positivität der Vektornorm

$$\|Ax\|_2^2 = |e_{i_0}^H Q^H Q\Lambda Q^H Q e_{i_0}| = |e_{i_0}^H \Lambda e_{i_0}| = |\Lambda_{i_0 i_0}| = \rho(A^H A).$$

Insgesamt folgt also

$$\|A\|_2^2 = \max_{\|x\|_2=1} \|Ax\|_2^2 = \rho(A^H A).$$

Ist A normal, so folgt $A = Q\Lambda Q^H$ und mit der unitären Invarianz der Spektralnorm und der Ähnlichkeit von A und Λ

$$\|A\|_2 = \|Q\Lambda Q^H\|_2 = \|\Lambda\|_2 = \rho^{1/2}(\Lambda^2) = \rho(\Lambda) = \rho(A).$$

□

Lemma 6.19. *Für jede zugeordnete Norm $|||\cdot|||$ auf $\mathbb{K}^{n\times n}$ gilt $\rho(A) \le |||A|||$ für alle $A \in \mathbb{K}^{n\times n}$.*

Beweis. Sei λ Eigenwert von A und x ein zugehöriger Eigenvektor mit $\|x\| = 1$. Dann gilt $|\lambda| = |\lambda| \cdot \|x\| = \|\lambda x\| = \|Ax\| \le |||A|||\|x\| = |||A|||$. □

Das folgende Lemma wird für spätere Zwecke benötigt. Da der Beweis sehr technisch ist, wird er ausgelassen.

Lemma 6.20. *Zu jedem $\varepsilon > 0$ und jedem $A \in \mathbb{K}^{n\times n}$ existiert eine zugeordnete Norm $|\!|\!| \cdot |\!|\!|_\varepsilon$, sodass $|\!|\!| A |\!|\!|_\varepsilon \leq \rho(A) + \varepsilon$.*

Satz 6.21. *Sei $A \in \mathbb{K}^{m\times n}$. Dann gilt*

(i) $\|A\|_2^2 \leq \|A^H\|_\infty \|A\|_\infty = \|A\|_1 \|A\|_\infty$,

(ii) ist A normal, so gilt $\|A\|_2 \leq |\!|\!| A |\!|\!|$ für jede zugeordnete Norm $|\!|\!| \cdot |\!|\!|$,

(iii) besitzt A nur ν Einträge pro Zeile und pro Spalte, die nicht verschwinden, so gilt $\|A\|_2 \leq \nu \|A\|_M$,

(iv) $\|A\|_M \leq \|A\|_2$.

Beweis. Zu *(ii)*: folgt aus Satz 6.18 und Lemma 6.19.
Zu *(i)*: Aus *(ii)* folgt, weil $A^H A$ normal ist, dass

$$\|A\|_2^2 = \rho(A^H A) \leq \|A^H A\|_\infty \leq \|A^H\|_\infty \|A\|_\infty = \|A\|_1 \|A\|_\infty.$$

Zu *(iii)*: Folgt aus *(i)*, weil $\|A\|_1 = \max_{j=1,\dots,n} \sum_{i=1}^m |a_{ij}| \leq \nu \|A\|_M$.
Zu *(iv)*: Sei (i_0, j_0) so gewählt, dass $|a_{i_0 j_0}| = \|A\|_M$. Sei $x \in \mathbb{K}^n$ der Vektor mit den Komponenten $x_j = \overline{a}_{i_0 j}$, $j = 1, \dots, n$. Dann gilt

$$\begin{aligned} \|Ax\|_2^2 = \sum_{i=1}^m \left| \sum_{j=1}^n a_{ij} x_j \right|^2 &\geq \left| \sum_{j=1}^n a_{i_0 j} x_j \right|^2 = \left(\sum_{j=1}^n |a_{i_0 j}|^2 \right)^2 \\ &\geq |a_{i_0 j}|^2 \left(\sum_{j=1}^n |a_{i_0 j}|^2 \right) = \|A\|_M^2 \|x\|_2^2. \end{aligned}$$

Wir erhalten

$$\|A\|_2 = \sup_{y \neq 0} \frac{\|Ay\|_2}{\|y\|_2} \geq \frac{\|Ax\|_2}{\|x\|_2} \geq \|A\|_M.$$

□

Satz 6.6 läßt sich natürlich auf Matrixnormen übertragen. Wir geben einige Äquivalenzrelationen an.

Beispiel 6.22. Sei $A \in \mathbb{K}^{m\times n}$. Dann gilt

$$\begin{aligned} \|A\|_2 \leq \|A\|_F &\leq \sqrt{m} \|A\|_2, \\ \frac{1}{\sqrt{n}} \|A\|_\infty \leq \|A\|_2 &\leq \sqrt{m} \|A\|_\infty, \\ \frac{1}{\sqrt{n}} \|A\|_1 \leq \|A\|_2 &\leq \sqrt{m} \|A\|_1, \\ \frac{1}{n} \|A\|_\infty \leq \|A\|_M &\leq \|A\|_\infty. \end{aligned}$$

6.2 Störungstheorie für lineare Gleichungssysteme

Wir wollen die Störungsanfälligkeit der Lösung x des linearen Gleichungssystems $Ax = b$ bei invertierbarer Matrix $A \in \mathbb{K}^{n\times n}$ und $b \in \mathbb{K}^n$ untersuchen. Dazu nehmen wir zunächst an, dass b um Δb gestört ist und A exakt gegeben ist. Sei $\tilde{x}$ die Lösung des linearen Gleichungssystems mit gestörter rechter Seite, d.h. $A\tilde{x} = b + \Delta b$. Dann gilt für $\Delta x := \tilde{x} - x$

$$\Delta x = \tilde{x} - x = A^{-1}(b + \Delta b) - A^{-1}b = A^{-1}\Delta b$$

Für ein verträgliches Vektor-/Matrixnormpaar erhält man

$$\frac{\|\Delta x\|}{\|x\|} = \frac{\|A^{-1}\Delta b\|}{\|x\|} \leq \frac{\|A^{-1}\|\|\Delta b\|\|b\|}{\|x\|\|b\|} = \|A^{-1}\|\frac{\|\Delta b\|}{\|b\|}\frac{\|Ax\|}{\|x\|} \leq \|A^{-1}\|\|A\|\frac{\|\Delta b\|}{\|b\|}.$$

Definition 6.23. *Sei $\|\cdot\|$ eine Matrixnorm und $A \in \mathbb{K}^{n\times n}$ invertierbar. Dann wird der Ausdruck*

$$\operatorname{cond}_{\|\cdot\|}(A) := \|A\|\|A^{-1}\|$$

*als **Kondition** der Matrix A bzgl. $\|\cdot\|$ bezeichnet.*

Bemerkung. Verallgemeinert man Definition 2.9 auf vektorwertige Probleme $f : \mathbb{K}^n \to \mathbb{K}^n$, so sieht man, dass die Kondition einer Matrix mit dem Maximum der Konditionszahlen $\max_{x\in\mathbb{K}^n} \kappa(f, x)$ der Abbildung $f(b) := A^{-1}b$ im Fall zugeordneter Normen übereinstimmt.

Im Folgenden wollen wir untersuchen, wie die Lösung x von Störungen der Koeffizientenmatrix A abhängt. Das folgende Störungslemma benötigt die sog. **Neumannsche Reihe** $\sum_{k=0}^{\infty} A^k$ einer Matrix $A \in \mathbb{K}^{n\times n}$. Man rechnet leicht nach, dass

$$(I - A)\sum_{k=0}^{m} A^k = I - A^{m+1} \quad \text{für alle } m \in \mathbb{N}$$

Ist $I - A$ invertierbar, so folgt

$$\sum_{k=0}^{n} A^k = (I - A)^{-1}(I - A^{m+1}) \tag{6.3}$$

Wir zeigen im folgenden Satz, dass unter der Vorraussetzung $\rho(A) < 1$, $I - A$ invertierbar ist und $A^{m+1} \to 0$ für $m \to \infty$.

Lemma 6.24. *Sei $\|\cdot\|$ eine Matrixnorm. Dann gilt für $A \in \mathbb{K}^{n\times n}$*

(i) $\rho(A) = \lim_{k\to\infty} \|A^k\|^{\frac{1}{k}} \leq \|A\|$,

(ii) $(A^k)_{k\in\mathbb{N}}$ *ist genau dann eine Nullfolge, wenn* $\rho(A) < 1$.

Beweis.

(i) Sei $\varepsilon > 0$. Nach Lemma 6.20 existiert zu A eine zugeordnete Norm $\|\cdot\|_\varepsilon$ mit $\|A\|_\varepsilon \leq \rho(A) + \varepsilon$. Außerdem folgt aus der Äquivalenz der Normen

$$c_\varepsilon \|A\|_\varepsilon \leq \|A\| \leq C_\varepsilon \|A\|_\varepsilon \quad \text{für alle } A \in \mathbb{K}^{n\times n}.$$

Dann gilt nach Lemma 6.19 und der Bemerkung vor Lemma 6.13 für alle $k \in \mathbb{N}$

$$\rho(A) = \rho(A^k)^{\frac{1}{k}} \leq \|A^k\|_\varepsilon^{\frac{1}{k}} \leq c_\varepsilon^{-\frac{1}{k}} \|A^k\|^{\frac{1}{k}} \leq \left(\frac{C_\varepsilon}{c_\varepsilon}\right)^{\frac{1}{k}} \|A^k\|_\varepsilon^{\frac{1}{k}} \leq \left(\frac{C_\varepsilon}{c_\varepsilon}\right)^{\frac{1}{k}} (\|A\|_\varepsilon^k)^{\frac{1}{k}}$$
$$= \left(\frac{C_\varepsilon}{c_\varepsilon}\right)^{\frac{1}{k}} \|A\|_\varepsilon \leq \left(\frac{C_\varepsilon}{c_\varepsilon}\right)^{\frac{1}{k}} (\rho(A) + \varepsilon).$$

Im Grenzwert $k \to \infty$ erhält man

$$\rho(A) \leq \lim_{k\to\infty} \|A^k\|^{\frac{1}{k}} \leq \rho(A) + \varepsilon.$$

Weil ε beliebig ist, folgt die Behauptung.

(ii) Ist $\rho(A) < 1$, so folgt mit (i) aus dem Wurzelkriterium

$$\left\| \sum_{k=0}^{\infty} A^k \right\| \leq \sum_{k=0}^{\infty} \|A^k\| < \infty.$$

Daher konvergiert $\sum_{k=0}^{\infty} A^k$ und $(A^k)_{k\in\mathbb{N}}$ muss eine Nullfolge sein. Ist $\rho(A) \geq 1$, so sei $x \in \mathbb{K}^n$, $\|x\| = 1$, ein Eigenvektor zum betragsmaximalen Eigenwert λ. Dann gilt

$$\|A^k\| \geq \|A^k x\| = |\lambda|^k \|x\| = \rho^k(A) \geq 1.$$

Daher kann $(A^k)_{k\in\mathbb{N}}$ keine Nullfolge sein.

□

Satz 6.25. *Genau für $\rho(A) < 1$ konvergiert die Neumannsche Reihe. In diesem Fall ist*

$$\sum_{k=0}^{\infty} A^k = (I - A)^{-1}.$$

Ist $\|A\| < 1$, so gilt sogar $\|(I - A)^{-1}\| \leq \frac{1}{1-\|A\|}$.

Beweis. Im Fall $\rho(A) < 1$ ist 1 kein Eigenwert von A. Weil $I - A$ die Eigenwerte $1 - \lambda_i(A)$, wobei $\lambda_i(A)$ den i-ten Eigenwert von A bezeichnet, ist 0 kein Eigenwert von $I - A$. Da nach Lemma 6.13 die Determinante das Produkt der Eigenwerte ist, folgt $\det(I - A) \neq 0$, was zur Invertierbarkeit von $I - A$ äquivalent ist. Wegen Lemma 6.24 kann in (6.3) der Grenzwert $m \to \infty$ gebildet werden, und man erhält die gewünschte Darstellung.
Im Fall $\rho(A) \geq 1$ ist $(A^k)_{k\in\mathbb{N}}$ nach Lemma 6.24 keine Nullfolge. Daher kann die Neumannsche Reihe nicht konvergieren.
Ist $\|A\| < 1$, so folgt nach Lemma 6.24, dass $\rho(A) \leq \|A\| < 1$. Daher konvergiert die Neumannsche Reihe, und es gilt

$$\|(I - A)^{-1}\| = \left\| \sum_{k=0}^{\infty} A^k \right\| \leq \sum_{k=0}^{\infty} \|A\|^k = \frac{1}{1 - \|A\|}.$$

□

Lemma 6.26 (Störungslemma). *Sei $A \in \mathbb{K}^{n\times n}$ invertierbar und $\|\cdot\|$ eine Matrixnorm. Ist $\Delta A \in \mathbb{K}^{n\times n}$ mit $\|A^{-1}\|\|\Delta A\| < 1$, so ist auch $A + \Delta A$ invertierbar und*

$$\|(A+\Delta A)^{-1}\| \le \frac{\|A^{-1}\|}{1-\|A^{-1}\|\|\Delta A\|}.$$

Beweis. Wegen $\|A^{-1}\Delta A\| \le \|A^{-1}\|\|\Delta A\| < 1$ existiert nach Satz 6.25 $(I + A^{-1}\Delta A)^{-1}$ und es gilt

$$\|(1+A^{-1}\Delta A)^{-1}\| \le \frac{1}{1-\|A^{-1}\Delta A\|} \le \frac{1}{1-\|A^{-1}\|\|\Delta A\|}.$$

Weil $A + \Delta A = A(I + A^{-1}\Delta A)$, ist auch $A + \Delta A$ invertierbar. Die Behauptung folgt wegen $(A+\Delta A)^{-1} = (I + A^{-1}\Delta A)^{-1}A^{-1}$ aus $\|(A+\Delta A)^{-1}\| \le \|(I + A^{-1}\Delta A)^{-1}\|\|A^{-1}\|$. □

Satz 6.27. *Sei ein verträgliches Vektor-/Matrixnormpaar $\|\cdot\|$ gegeben. Seien $A \in \mathbb{K}^{n\times n}$ invertierbar und $\Delta A \in \mathbb{K}^{n\times n}$ mit $\|A^{-1}\|\|\Delta A\| < 1$. Mit vorgegebenen $b \in \mathbb{K}^n \setminus \{0\}$, $\Delta b \in \mathbb{K}^n$ seien x und $\tilde{x}$ definiert durch*

$$Ax = b, \quad (A+\Delta A)\tilde{x} = b + \Delta b. \tag{6.4}$$

Dann gilt für $\Delta x := \tilde{x} - x$

$$\frac{\|\Delta x\|}{\|x\|} \le \frac{\operatorname{cond}_{\|\cdot\|}(A)}{1-\operatorname{cond}_{\|\cdot\|}(A)\frac{\|\Delta A\|}{\|A\|}}\left\{\frac{\|\Delta A\|}{\|A\|} + \frac{\|\Delta b\|}{\|b\|}\right\}.$$

Beweis. Wegen $\|A^{-1}\|\|\Delta A\| < 1$ ist $A+\Delta A$ nach dem Störungslemma 6.26 invertierbar und

$$\|(A+\Delta A)^{-1}\| \le \frac{\|A^{-1}\|}{1-\|A^{-1}\|\|\Delta A\|}.$$

Aus (6.4) erhält man

$$\begin{aligned} &(A+\Delta A)(x+\Delta x) = b + \Delta b \\ \iff &Ax + \Delta A\cdot x + (A+\Delta A)\Delta x = b + \Delta b \\ \iff &(A+\Delta A)\Delta x = \Delta b - \Delta A\cdot x \\ \iff &\Delta x = (A+\Delta A)^{-1}(\Delta b - \Delta A\cdot x). \end{aligned}$$

Hieraus folgt

$$\begin{aligned} \|\Delta x\| &\le \frac{\|A^{-1}\|}{1-\|A^{-1}\|\|\Delta A\|}(\|\Delta A\|\|x\| + \|\Delta b\|) \\ &= \frac{\operatorname{cond}_{\|\cdot\|}(A)}{1-\operatorname{cond}_{\|\cdot\|}(A)\frac{\|\Delta A\|}{\|A\|}}\left\{\frac{\|\Delta A\|}{\|A\|}\|x\| + \frac{\|\Delta b\|}{\|A\|}\right\} \end{aligned}$$

Division durch $\|x\|$ und Verwendung von $\|b\| = \|Ax\| \le \|A\|\|x\|$ liefert die Behauptung. □

Beispiel 6.28.

(a) Sei

$$A = \begin{bmatrix} 3 & 1.001 \\ 6 & 1.997 \end{bmatrix}, \quad b = \begin{bmatrix} 1.999 \\ 4.003 \end{bmatrix}.$$

Dann ist

$$A^{-1} = \frac{1}{-0.015} \begin{bmatrix} 1.997 & -1.001 \\ -6 & 3 \end{bmatrix}$$

und somit $\|A\|_\infty = 7.997$, $\|A^{-1}\|_\infty = 600$. Hieraus erhält man $\operatorname{cond}_{\|\cdot\|_\infty}(A) = 4798.2$.
Sei

$$\tilde{A} = \begin{bmatrix} 3 & 1 \\ 6 & 1.997 \end{bmatrix}, \quad \tilde{b} = \begin{bmatrix} 2.002 \\ 4.003 \end{bmatrix}$$

Dann ist

$$\Delta A = \begin{bmatrix} 0 & 0.001 \\ 0 & 0 \end{bmatrix}, \quad \Delta b = \begin{bmatrix} 0.003 \\ 0 \end{bmatrix}$$

und somit $\|\Delta A\|_\infty = 0.001$, $\|\Delta b\|_\infty = 0.003$. Man überprüft leicht, dass $x = [1, -1]^T$ die Lösung von $Ax = b$ ist. Außerdem ist $\tilde{x} = [0.229, 4/3]^T$ die Lösung von $\tilde{A}\tilde{x} = \tilde{b}$.
Der relative Fehler beträgt

$$\frac{\|\Delta x\|_\infty}{\|x\|_\infty} = \frac{7}{3} = 2.\overline{3}.$$

(b) Die **Hilbert-Matrix** $H_n = \left(\frac{1}{i+j-1}\right)_{i,j=1,\ldots,n}$ weist eine Kondition auf, die exponentiell mit n wächst.

6.3 Gaußsche Elimination

Sei $A \in \mathbb{K}^{n\times n}$ eine **untere Dreiecksmatrix**, d.h. $a_{ij} = 0$ für $i < j$ mit nicht verschwindenden Diagonaleinträgen. Die Lösung x von $Ax = b$ erhält man wegen

$$b_i = \sum_{j=1}^{i} a_{ij}x_j = a_{ii}x_i + \sum_{j=1}^{i-1} a_{ij}x_j$$

$$\iff x_i = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j \right), \quad i = 1, \ldots, n.$$

Man beachte, dass auf der rechten Seite der letzten Gleichung ausschließlich bekannte Komponenten des Lösungsvektors auftreten.

Algorithmus 6.29.
`Input:` untere Dreiecksmatrix A, rechte Seite b
`Output:` Lösung von $Ax = b$

```
for (i=0; i<n; ++i) {
   x[i] = b[i];
   for (j=0; j<i; ++j) x[i] = x[i] - A[i+n*j]*x[j];
   x[i] = x[i]/A[(n+1)*i];
}
```

Der folgende Algorithmus liefert dieselbe Lösung, ist aber deutlich effizienter, weil er wegen der besseren Datenlokalität beim Zugriff auf die Matrix A den Cache des Prozessors nutzen kann.

Algorithmus 6.30.
`Input:` untere Dreiecksmatrix A, rechte Seite b
`Output:` Lösung von $Ax = b$

```
for (j=0; j<n; ++j) {
   x[j] = b[j]/A[j*(n+1)];
   for (i=j+1; i<n; ++i) b[i] = b[i] - A[i+n*j]*x[j];
}
```

Ist nun $A \in \mathbb{K}^{n\times n}$ eine **obere Dreiecksmatrix**, d.h. gilt $a_{ij} = 0$ für $i > j$, mit nicht verschwindenden Diagonaleinträgen, so erhält man

$$b_i = \sum_{j=i}^{n} a_{ij}x_j = a_{ii}x_i + \sum_{j=i+1}^{n} a_{ij}x_j$$

und somit

$$x_i = \frac{1}{a_{ii}}\left(b_i - \sum_{j=i+1}^{n} a_{ij}x_j\right), \quad i = 1, \ldots, n.$$

Hier muss der Vektor x beginnend beim größten Index, d.h. rückwärts, bestimmt werden.

Algorithmus 6.31.
`Input:` obere Dreiecksmatrix A, rechte Seite b
`Output:` Lösung von $Ax = b$

```
for (j=n; j>0;) {
  --j;
  x[j] = b[j]/A[j*(n+1)];
  for (i=j; i>0;) {
    --i;
    b[i] = b[i] - A[i+n*j]*x[j];
  }
}
```

Mit obigen Algorithmen kann $Ax = b$ mit Aufwand $O(n^2)$ gelöst werden, falls A eine untere oder eine obere Dreiecksmatrix ist. Üblicherweise ist A aber eine allgemeine Matrix. Wenn wir A in das Produkt einer unteren und einer oberen Dreiecksmatrix L bzw. R zerlegen können, d.h. $A = LR$, so kann die Lösung von $Ax = b$ in zwei Schritten geschehen:

$$Ax = b \iff LRx = b \iff \begin{cases} Ly = b, \\ Rx = y. \end{cases}$$

Das Lösen des Gleichungssystems $Ly = b$ nach y kann mit Algorithmus 6.30 geschehen und wird als **Vorwärtssubstitution** bezeichnet. Die anschließende Lösung von $Rx = y$ (siehe Algorithmus 6.31) wird als **Rückwärtssubstitution** bezeichnet.

Im Folgenden wird gezeigt, wie man eine LR-Zerlegung einer regulären Matrix $A \in \mathbb{K}^{n\times n}$ konstruiert.

Definition 6.32. *Jede Matrix der Form* $L_k = I - \ell^{(k)} e_k^T \in \mathbb{K}^{n\times n}$ *mit*

$$\ell^{(k)} = \left[0, \ldots, 0, \ell_{k+1}^{(k)}, \ldots, \ell_n^{(k)}\right]^T \in \mathbb{K}^n$$

heißt ***Gauß-Matrix****.*

Die Gauß-Matrix L_k besitzt die Einträge

$$L_k = \begin{bmatrix} 1 & & & & \\ & \ddots & & & \\ & & 1 & & \\ & & -\ell_{k+1}^{(k)} & 1 & \\ & & \vdots & & \ddots \\ & & -\ell_n^{(k)} & & 1 \end{bmatrix}.$$

Daher gilt $\det L_k = 1$ und L_k ist invertierbar. Ihre Inverse ist $L_k^{-1} = I + \ell^{(k)} e_k^T$, weil mit $e_k^T \ell^{(i)} = 0$, $i \geq k$, folgt

$$L_k^{-1} L_k = (I + \ell^{(k)} e_k^T)(I - \ell^{(k)} e_k^T) = I - \ell^{(k)} e_k^T + \ell^{(k)} e_k^T - \ell^{(k)} e_k^T \ell^{(k)} e_k^T = I.$$

Insbesondere ist L_k^{-1} ebenfalls eine Gauß-Matrix. Die Multiplikation von L_k mit $x \in \mathbb{K}^n$ ergibt

$$L_k x = \begin{bmatrix} x_1 \\ \vdots \\ x_k \\ x_{k+1} - \ell_{k+1}^{(k)} x_k \\ \vdots \\ x_n - \ell_n^{(k)} x_k \end{bmatrix}.$$

Dabei bleiben die ersten k Komponenten von x unverändert. Gilt $x_k \neq 0$, so kann durch die Wahl

$$\ell_i^{(k)} := \frac{x_i}{x_k}, \quad i = k+1, \ldots, n, \tag{6.5}$$

erreicht werden, dass $L_k x$ in den letzten $n - k$ Komponenten verschwindet.

Durch sukzessive Anwendung von Gauß-Matrizen wird nun eine gegebene reguläre Matrix $A \in \mathbb{K}^{n\times n}$ auf obere Dreiecksform gebracht. Die Bedingung $x_k \neq 0$ in (6.5) werden wir durch Vertauschung der Zeilen von A garantieren. Dazu multiplizieren wir A mit *Permutationsmatrizen.*

Definition 6.33. *Eine Matrix $P \in \mathbb{R}^{n\times n}$ heißt* ***Permutationsmatrix****, falls eine Permutation $\pi \in \mathcal{P}_n$ existiert, sodass*

$$P_{ij} = \begin{cases} 1, & \pi(i) = j, \\ 0, & \text{sonst.} \end{cases}$$

Permutationsmatrizen sind wegen

$$(P^T P)_{ij} = \sum_{\ell=1}^{n} (e_i)_{\pi(\ell)} (e_j)_{\pi(\ell)} = \begin{cases} 1, & i = j, \\ 0, & \text{sonst}, \end{cases}$$

und analog $PP^T = I$ orthogonal. Die Multiplikation einer Permutationsmatrix an eine Matrix A von links entspricht der Vertauschung der Zeilen und die Multiplikation von rechts bedeutet die Vertauschung der Spalten von A.

Sei $A \in \mathbb{K}^{n\times n}$ regulär. Die erste Spalte von A enthält einen nicht verschwindenden Eintrag, weil A sonst singulär wäre. Mit P_1 bezeichnen wir die Permutationsmatrix, die die zugehörige Zeile in A mit der ersten Zeile vertauscht. Wählt man für x in (6.5) die erste Spalte von $A^{(0)} := P_1 A$, so läßt sich eine Matrix L_1 definieren, sodass die erste Zeile von $L_1 A^{(0)}$ und $A^{(0)}$ übereinstimmen und die letzten $n-1$ Einträge der ersten Spalte von $L_1 A^{(0)}$ verschwinden:

$$L_1 A^{(0)} = \begin{bmatrix} a_{11}^{(0)} & \cdots & \cdots & a_{1n}^{(0)} \\ 0 & * & \cdots & * \\ \vdots & \vdots & & \vdots \\ 0 & * & \cdots & * \end{bmatrix}.$$

In den unteren $n-1$ Einträgen der zweiten Spalte von $L_1 A^{(0)}$ befindet sich wegen $|\det L_1 A^{(0)}| = |\det A|$ wiederum ein nicht verschwindender Eintrag. Die Permutation, die die entsprechende Zeile mit der zweiten Zeile von $L_1 A^{(0)}$ vertauscht, bezeichnen wir mit P_2. Man findet wieder eine Gauß-Matrix L_2, indem wir für x in (6.5) die zweite Spalte von $A^{(1)} := P_2 L_1 A^{(0)}$ einsetzen. Wir erhalten

$$L_2 A^{(1)} = \begin{bmatrix} a_{11}^{(0)} & \cdots & \cdots & \cdots & a_{1n}^{(0)} \\ 0 & a_{22}^{(1)} & \cdots & \cdots & a_{2n}^{(1)} \\ 0 & 0 & * & \cdots & * \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & * & \cdots & * \end{bmatrix}.$$

Durch sukzessive Multiplikation mit Permutationsmatrizen $P_1, \ldots, P_{n-1}$ und Gauß-Matrizen $L_1, \ldots, L_{n-1}$ erhält man auf diese Weise die gewünschte obere Dreiecksstruktur

$$R := L_{n-1} P_{n-1} L_{n-2} P_{n-2} \cdot \ldots \cdot L_1 P_1 A = \begin{bmatrix} a_{11}^{(0)} & \cdots & \cdots & \cdots & \cdots & a_{1n}^{(0)} \\ 0 & a_{22}^{(1)} & \cdots & \cdots & \cdots & a_{2n}^{(1)} \\ 0 & 0 & a_{33}^{(2)} & \cdots & \cdots & a_{3n}^{(2)} \\ \vdots & \vdots & 0 & \ddots & & \vdots \\ \vdots & \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & a_{nn}^{(n-1)} \end{bmatrix}.$$

Die letzte Gleichung kann wegen der Orthogonalität der Matrizen $P_1, \ldots, P_{n-1}$ auch in der Form

$$\hat{L}_{n-1} \cdot \ldots \cdot \hat{L}_1 P_{n-1} \cdot \ldots \cdot P_1 A = R$$

dargestellt werden. Dabei definieren wir

$$\begin{aligned}\hat{L}_k &= P_{n-1} \cdot \ldots \cdot P_{k+1} L_k P_{k+1}^T \cdot \ldots \cdot P_{n-1}^T \\ &= P_{n-1} \cdot \ldots \cdot P_{k+1} (I - \ell^{(k)} e_k^T) P_{k+1}^T \cdot \ldots \cdot P_{n-1}^T \\ &= I - P_{n-1} \cdot \ldots \cdot P_{k+1} \ell^{(k)} e_k^T P_{k+1}^T \cdot \ldots \cdot P_{n-1}^T \\ &= I - \hat{\ell}^{(k)} e_k^T\end{aligned}$$

mit $\hat{\ell}^{(k)} := P_{n-1} \cdot \ldots \cdot P_{k+1} \ell^{(k)}$. Dabei haben wir die Identität $e_k = P_{n-1} \cdot \ldots \cdot P_{k+1} e_k$ benutzt. Weil die ersten k Einträge von $\hat{\ell}^{(k)}$ verschwinden, ist $\hat{L}_k$ ebenfalls eine Gauß-Matrix. Mit $P := P_{n-1} \cdot \ldots \cdot P_1$ ist daher

$$PA = \hat{L}_1^{-1} \cdot \ldots \cdot \hat{L}_{n-1}^{-1} R = \left(I + \sum_{k=1}^{n-1} \hat{\ell}^{(k)} e_k^T \right) R.$$

Die letzte Gleichung folgt aus $e_k^T \hat{\ell}^{(i)} = 0$, $i \geq k$. Die Matrix

$$L := I + \sum_{k=1}^{n-1} \hat{\ell}^{(k)} e_k^T$$

liefert $PA = LR$. Aus der Struktur der Vektoren $\hat{\ell}^{(k)}$ erkennt man, dass L eine untere Dreiecksmatrix mit Einträgen 1 auf der Diagonalen ist. Wegen $|\det A| = |\det R|$ ist ferner R regulär und die Diagonaleinträge von R sind nicht Null.

Definition 6.34. *Eine **spaltenpivotisierte LR-Zerlegung** einer Matrix $A \in \mathbb{K}^{n \times n}$ ist eine Faktorisierung $PA = LR$ mit einer Permutationsmatrix $P \in \mathbb{R}^{n \times n}$, einer unteren Dreiecksmatrix L mit 1 auf der Diagonalen und einer oberen Dreiecksmatrix R mit nicht verschwindenden Diagonaleinträgen.*

Algorithmus 6.35 (Spaltenpivotisierte Gaußsche Elimination)**.**
`Input:` Reguläre Matrix $A \in \mathbb{K}^{n \times n}$
`Output:` Spaltenpivotisierte LR-Zerlegung von A

1: $R := A;\ L := I; P := I;$
2: `for` $(k = 1;\ k < n;\ ++k)$ {
3: wähle $i \geq k$, sodass $r_{ik} \neq 0$;
4: vertausche die i-te mit der k-ten Zeile in P, R und L;
5: `for` $(j = k + 1;\ j \leq n;\ ++j)$ {
6: $\ell_{jk} = \frac{r_{jk}}{r_{kk}}$;
7: $r_{j,k:n} = r_{j,k:n} - \ell_{jk} r_{k,k:n}$;
8: }
9: }

Bemerkung.

(a) Wir verwenden die MATLAB-Notation $a_{i,k:\ell}$ für den Vektor $[a_{ik}, \dots, a_{i\ell}]$ bestehend aus den Einträgen von Position k bis ℓ der i-ten Zeile von A.

(b) Die im k-ten Schritt berechneten Einträgen der Gauß-Matrix L_k önnen in den frei gewordenen Positionen der k-ten Spalte von A unterhalb der Diagonalen gespeichert werden. Auf diese Weise steht nach Abschluss des Verfahrens in der unteren Hälfte von A die untere Hälfte von L (ohne Diagonale), und in der oberen Hälfte von A (einschließlich der Diagonalen) befindet sich die obere Dreiecksmatrix R. Die Permutation muss in einem Feld der Länge n gesondert gespeichert werden.

(c) Durch das obige Gaußsche Eliminationsverfahren wird es möglich, mit

$$\sum_{k=1}^{n-1} \sum_{j=k+1}^{n} 2(n-k+1) = 2\sum_{k=1}^{n-1}(n-k)(n-k+1) \overset{k'=n-k}{=} 2\sum_{k=1}^{n-1} k^2 + k = \frac{2}{3}n^3 + O(n^2)$$

Operationen eine spaltenpivotisierte LR-Zerlegung von A zu berechnen.

Beispiel 6.36. Wir erhalten folgende LR-Zerlegung

$$A = \begin{bmatrix} -1 & 3 & -1 \\ 3 & -8 & 4 \\ 2 & -2 & 4 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ -3 & 1 & 0 \\ -2 & 0 & 1 \end{bmatrix} \begin{bmatrix} -1 & 3 & -1 \\ 0 & 1 & 1 \\ 0 & 4 & 2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ -3 & 1 & 0 \\ -2 & 4 & 1 \end{bmatrix} \begin{bmatrix} -1 & 3 & -1 \\ 0 & 1 & 1 \\ 0 & 0 & -2 \end{bmatrix}.$$

Die Lösung von $Ax = b := \begin{bmatrix} 2 & -3 & 6 \end{bmatrix}^T$ ergibt sich aus $Ly = b$

$$\begin{bmatrix} 1 & 0 & 0 \\ -3 & 1 & 0 \\ -2 & 4 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 2 \\ -3 \\ 6 \end{bmatrix} \quad \Rightarrow \quad y = \begin{bmatrix} 2 \\ 3 \\ -2 \end{bmatrix}$$

und $Rx = y$

$$\begin{bmatrix} -1 & 3 & -1 \\ 0 & 1 & 1 \\ 0 & 0 & -2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \\ -2 \end{bmatrix} \quad \Rightarrow \quad x = \begin{bmatrix} 3 \\ 2 \\ 1 \end{bmatrix}.$$

Der folgende Satz fasst unsere Konstruktion der LR-Zerlegung zusammen.

Satz 6.37. *Ist $A \in \mathbb{K}^{n\times n}$ regulär, so ist Algorithmus 6.35 durchführbar und produziert eine spaltenpivotisierte LR-Zerlegung $PA = LR$.*

In Beispiel 6.36 haben wir keine Permutation benötigt. Der folgende Satz klärt, unter welchen Umständen dies der Fall ist.

Satz 6.38. *Algorithmus 6.35 ist genau dann ohne Pivotisierung durchführbar, wenn die ersten $n-1$ Hauptabschnittsmatrizen $A_k := a_{1:k,1:k}$, $k = 1, \dots, n-1$, von A regulär sind. Die Zerlegung $A = LR$ ist dann eindeutig.*

Beweis. Wir zeigen per Induktion, dass im k-ten Schritt $a_{kk}^{(k-1)} \neq 0$ genau dann gilt, wenn $\det A_k \neq 0$. Für $k = 1$ ist die Aussage trivial. Wir nehmen an, dass die Aussage für $k-1$ gilt. Wegen

$$\det A_k = \underbrace{a_{11}^{(0)} \cdot a_{22}^{(1)} \cdot a_{33}^{(2)} \cdot \ldots \cdot a_{k-1k-1}^{(k-2)}}_{\det A_{k-1} \neq 0} \cdot a_{kk}^{(k-1)} \tag{6.6}$$

ist $a_{kk}^{(k-1)} \neq 0$ genau dann, wenn $\det A_k \neq 0$. In diesem Fall bestimmt die Darstellung

$$a_{ij} = \sum_{k=1}^{i-1} \ell_{ik} r_{kj} + r_{ij} \quad i,j = 1,\ldots,n,$$

die Koeffizienten auf eindeutige Weise. □

Beispiel 6.39. $A = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ ist regulär, aber die Hauptabschnittsmatrix $A_1 = a_{11}$ ist singulär. Hier benötigt man die Pivotisierung.

Bemerkung. Wegen (6.6) kann die Determinante einer Matrix aus einer LR-Zerlegung mit $O(n^3)$ Operationen berechnet werden.

6.4 Stabilität der Gaußschen Elimination

Im Folgenden wollen wir den Einfluss von Rundungsfehlern auf das Produkt LR betrachten. Die Genauigkeit der Faktoren L und R wird im Allgemeinen schlechter sein, was für die Numerik von untergeordnetem Interesse ist, weil L und R immer als Produkt auftreten.
Wir benötigen den **Betrag** $|A| \in \mathbb{R}^{m\times n}$ einer Matrix $A \in \mathbb{K}^{m\times n}$ definiert durch

$$|A|_{ij} = |a_{ij}|, \quad i = 1,\ldots,m,\ j = 1,\ldots,n.$$

Abschätzungen der Form $|A| \leq |B|$ sind einträgeweise zu verstehen.

Satz 6.40. *Sei $A \in \mathbb{K}^{n\times n}$. Wenn die LR-Zerlegung in endlicher Arithmetik durchführbar ist, dann gilt für die berechneten Matrizen $\tilde{L}$ und $\tilde{R}$, dass*

$$\tilde{L}\tilde{R} = A + \Delta A$$

mit $|\Delta A| \leq 2(n-1)\varepsilon_{\mathbb{F}}(|A| + |\tilde{L}|\,|\tilde{R}|) + O(\varepsilon_{\mathbb{F}}^2)$.

Beweis. Der Beweis erfolgt induktiv über n. Für $n = 1$ ist die Aussage klar. Angenommen, sie gilt für $n-1$. Wir verwenden die Zerlegung

$$A = \begin{bmatrix} \alpha & w^T \\ v & B \end{bmatrix} \quad \text{mit } \alpha \in \mathbb{K} \setminus \{0\}.$$

Im Algorithmus wird im ersten Schritt

$$\tilde{z} = \mathrm{rd}\left(\frac{v}{\alpha}\right) \quad \text{und} \quad \tilde{S} = \mathrm{rd}(B - \tilde{z}w^T)$$

bestimmt. Nach (2.1) und der Bemerkung nach Satz refsen:1.22 gilt

$$\tilde{z} = \frac{v}{\alpha} + f, \quad |f| \leq \varepsilon_{\mathbb{F}} \frac{|v|}{|\alpha|},$$

und

$$\tilde{S} = B - \tilde{z}w^T + F, \quad |F| \leq 2\varepsilon_{\mathbb{F}}(|B| + |\tilde{z}|\,|w|^T) + O(\varepsilon_{\mathbb{F}}^2). \tag{6.7}$$

Nach Induktionsannahme gilt für die in den nächsten $n-1$ Schritten berechneten Faktoren $\tilde{L}_1, \tilde{R}_1$

$$\tilde{L}_1\tilde{R}_1 = \tilde{S} + E, \quad |E| \leq 2(n-2)\varepsilon_{\mathbb{F}}(|\tilde{S}| + |\tilde{L}_1|\,|\tilde{R}_1|) + O(\varepsilon_{\mathbb{F}}^2). \tag{6.8}$$

Daher erhält man

$$\tilde{L}\tilde{R} = \begin{bmatrix} 1 & 0 \\ \tilde{z} & \tilde{L}_1 \end{bmatrix} \begin{bmatrix} \alpha & w^T \\ 0 & \tilde{R}_1 \end{bmatrix} = A + \begin{bmatrix} 0 & 0 \\ \alpha f & E+F \end{bmatrix} =: A + \Delta A.$$

Es bleibt, $|\Delta A|$ abzuschätzen. Zunächst erhält man aus (6.7), dass

$$|\tilde{S}| \leq (1 + 2\varepsilon_{\mathbb{F}})(|B| + |\tilde{z}|\,|w|^T) + O(\varepsilon_{\mathbb{F}}^2).$$

Ferner haben wir mit (6.8)

$$|E+F| \leq 2(n-1)\varepsilon_{\mathbb{F}}(|B| + |\tilde{z}|\,|w|^T + |\tilde{L}_1|\,|\tilde{R}_1|) + O(\varepsilon_{\mathbb{F}}^2).$$

Aus $|\alpha f| \leq \varepsilon_{\mathbb{F}}|v|$ folgt

$$\begin{aligned} |\Delta A| &\leq 2(n-1)\varepsilon_{\mathbb{F}} \left(\begin{bmatrix} |\alpha| & |w|^T \\ |v| & |B| \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ |\tilde{z}| & |\tilde{L}_1| \end{bmatrix} \begin{bmatrix} |\alpha| & |w|^T \\ 0 & |\tilde{R}_1| \end{bmatrix} \right) + O(\varepsilon_{\mathbb{F}}^2) \\ &= 2(n-1)\varepsilon_{\mathbb{F}}(|A| + |\tilde{L}|\,|\tilde{R}|) + O(\varepsilon_{\mathbb{F}}^2). \end{aligned}$$

□

Bemerkung.

(a) Satz 6.40 gilt natürlich auch für PA, wobei P eine Permutationsmatrix bezeichnet, mit der die LR-Zerlegung durchführbar ist.

(b) Man beachte, dass Satz 6.40 nicht die Rückwärtsstabilität der Gaußschen Elimination bedeutet. Dies gilt nur, falls $|\tilde{L}|\,|\tilde{R}| \approx |A|$.

Beispiel 6.41. Wendet man Algorithmus 6.35 auf die Matrix

$$A = \begin{bmatrix} \varepsilon & 1 \\ 1 & 1 \end{bmatrix}, \quad \varepsilon > 0,$$

an, so erhält man

$$\begin{bmatrix} \varepsilon & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1/\varepsilon & 1 \end{bmatrix} \begin{bmatrix} \varepsilon & 1 \\ 0 & 1 - 1/\varepsilon \end{bmatrix}.$$

Dabei sei $\varepsilon > 0$ so klein gewählt, dass bei endlicher Arithmetik $1 - 1/\varepsilon = -1/\varepsilon$ gilt. Anstelle von R würde mit

$$\tilde{R} = \begin{bmatrix} \varepsilon & 1 \\ 0 & -1/\varepsilon \end{bmatrix}$$

gerechnet. Das bedeutet

$$L\tilde{R} = \begin{bmatrix} \varepsilon & 1 \\ 1 & 0 \end{bmatrix}$$

und bei rechter Seite $b = \begin{bmatrix} 1 & 0 \end{bmatrix}^T$ würde man statt der exakten Lösung $x = \begin{bmatrix} -1 & 1 \end{bmatrix}^T$ die Lösung $\tilde{x} = \begin{bmatrix} 0 & 1 \end{bmatrix}^T$ erhalten. Fehler der Größenordnung ε können somit Fehler der Ordnung 1 hervorrufen. Eine solche Instabilität tritt immer dann auf, wenn im Vergleich zu den Einträgen von A große Einträge in L oder R vorkommen. Vertauscht man die Zeilen von A, so erhält man

$$\begin{bmatrix} 1 & 1 \\ \varepsilon & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ \varepsilon & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 0 & 1-\varepsilon \end{bmatrix}.$$

Hier tritt die Instabilität nicht auf.

Daher sollte im k-ten Schritt der Gaußschen Elimimation das betragsmäßig größte Element der letzten $n-k+1$ Einträge der k-ten Spalte pivotisiert werden. Diese Strategie wird als **partielle Pivotisierung** bezeichnet. Sie hat zur Folge, dass $|\ell_{ij}| \leq 1$, $i,j = 1,\dots,n$. Die Wahl des betragsmäßig größten Eintrags im Schnitt der unteren $n-k+1$ Zeilen und hinteren $n-k+1$ Spalten bezeichnet man als **vollständige Pivotisierung**. In diesem Fall werden auch Spaltenvertauschungen nötig. Man erhält daher eine Zerlegung $PAQ = LR$ mit Permuationsmatrizen P, Q. Diese Strategie ist aber aufwändig und der Zusatzaufwand hilft im Hinblick auf die Stabilität in der Regel nur wenig.

Weil im Fall der partiellen Pivotisierung $|\ell_{ij}| \leq 1$, $i,j = 1,\dots,n$, gilt, hängt die Rückwärtsstabilität nach Satz 6.40 nur von der Größe von $|R|$ ab. Wir definieren daher den **Wachstumsfaktor**

$$\rho_n := \frac{\|R\|_\infty}{\|A\|_\infty}.$$

Aus Satz 6.40 erhalten wir das folgende Resultat.

Satz 6.42. *Sei $A \in \mathbb{K}^{n\times n}$ regulär. Das Gaußsche Eliminationsverfahren mit Spaltenpivotisierung berechne Matrizen $\tilde{L}$, $\tilde{R}$ und $\tilde{P}$. Dann ist*

$$\tilde{L}\tilde{R} = \tilde{P}A + \Delta A$$

für ein $\Delta A \in \mathbb{K}^{n\times n}$ mit $\|\Delta A\|_\infty \leq c\rho_n \varepsilon_{\mathbb{F}} \|A\|_\infty$.

Bemerkung. Der Wachstumsfaktor ρ_n entscheidet über die Rückwärtsstabilität der Gaußschen Elimination. Diese ist nach Definition 2.11 rückwärtsstabil, falls ρ_n für alle Matrizen $A \in \mathbb{K}^{n\times n}$ gleichmäßig nach oben beschränkt ist.

Lemma 6.43. *Sei $PA = LR$ eine spaltenpivotisierte LR-Zerlegung von $A \in \mathbb{K}^{n\times n}$. Dann gilt $\rho_n \leq 2^{n-1}$.*

Beweis. Wegen $R = L^{-1}PA$ haben wir

$$\rho_n := \frac{\|R\|_\infty}{\|A\|_\infty} \leq \frac{\|L^{-1}\|_\infty \|A\|_\infty}{\|A\|_\infty} = \|L^{-1}\|_\infty.$$

Weil $L^{-1} = \hat{L}_{n-1} \cdot \ldots \cdot \hat{L}_1$ (siehe die Darstellung vor Definition 6.34) und $\|\hat{L}_i\|_\infty \leq 2$, folgt

$$\|L^{-1}\|_\infty \leq \|\hat{L}_{n-1}\|_\infty \cdot \ldots \cdot \|\hat{L}_1\|_\infty \leq 2^{n-1}.$$

□

Beispiel 6.44. Wir betrachten die **Wilkinson-Matrix**

$$W_n = \begin{bmatrix} 1 & & & 1 \\ -1 & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ -1 & \cdots & -1 & 1 \end{bmatrix} \in \mathbb{R}^{n\times n}.$$

Man überzeugt sich leicht davon, dass Spaltenpivotisierung $P = I$ liefert und dass

$$W_n = \begin{bmatrix} 1 & & & \\ -1 & \ddots & & \\ \vdots & \ddots & \ddots & \\ -1 & \cdots & -1 & 1 \end{bmatrix} \begin{bmatrix} 1 & & & 1 \\ & \ddots & & 2 \\ & & 1 & \vdots \\ & & & 2^{n-1} \end{bmatrix}.$$

Zusammenfassend ist die Gaußsche Elimination rückwärtsstabil. Die Konstanten hängen jedoch exponentiell von der Dimension ab, sodass die Stabilitätsabschätzung in der Praxis unbrauchbar ist. Die tatsächliche Größe von ρ_n und damit die Stabilität kann aber im Verlauf des Algorithmus auf einfache Weise inspiziert werden.

Die Gaußsche Elimination kann auch in Blockform angewendet werden. Wir betrachten die Blockpartitionierung von $A \in \mathbb{K}^{n\times n}$

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad A_{11} \in \mathbb{K}^{p\times p},\ A_{22} \in \mathbb{K}^{(n-p)\times(n-p)}. \tag{6.9}$$

Ist A_{11} regulär, so kann A wie folgt zerlegt werden:

$$A = \begin{bmatrix} I & 0 \\ -A_{21}A_{11}^{-1} & I \end{bmatrix} \begin{bmatrix} A_{11} & A_{12} \\ 0 & S \end{bmatrix} \tag{6.10}$$

mit dem **Schur-Komplement** $S := A_{22} - A_{21}A_{11}^{-1}A_{12}$ von A bzgl. A_{11}. Mann beachte, dass (6.10) eine Block-LR-Zerlegung ist.

6.5 Die Cholesky-Zerlegung

In diesem Abschnitt werden wir eine Spezialform der LR-Zerlegung für *positiv definite* Matrizen vorstellen.

Definition 6.45. *Eine hermitesche Matrix $A \in \mathbb{K}^{n\times n}$ heißt **positiv semidefinit**, falls $x^H Ax \geq 0$ für alle $x \in \mathbb{K}^n$. Gilt $x^H Ax > 0$ für alle $x \in \mathbb{K}^n \setminus \{0\}$, so heißt A **positiv definit**.*

Bemerkung. Ist A positiv definit, so ist A insbesondere regulär. Wegen $x^H Ax > 0$ für alle $x \neq 0$ ist nämlich $\ker A = \{0\}$. Daher sind die Spalten von A sind linear unabhängig, und es folgt $\det A \neq 0$. Ferner sind die Diagnaleinträge positiv: $a_{ii} = e_i^H Ae_i > 0$, $i = 1, \ldots, n$.

Das folgende Lemma wird von zentraler Bedeutung für die Durchführbarkeit der Cholesky-Zerlegung sein.

Lemma 6.46. *Sei $A \in \mathbb{K}^{n\times n}$ positiv definit. Dann ist das Schur-Komplement S wohldefiniert und sowohl A_{11} als auch S sind positiv definit.*

Beweis. Sei $x = [x_1, x_2]^T$ mit $x_1 \in \mathbb{K}^p$ partitioniert wie in (6.9). Aus

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}^H = \begin{bmatrix} A_{11}^H & A_{21}^H \\ A_{12}^H & A_{22}^H \end{bmatrix}$$

sieht man $A_{11} = A_{11}^H$, $A_{22} = A_{22}^H$ und $A_{12} = A_{21}^H$. Also ist A_{11} hermitesch, und es gilt

$$0 \le \begin{bmatrix} x_1 \\ 0 \end{bmatrix}^H \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ 0 \end{bmatrix} = \begin{bmatrix} x_1 \\ 0 \end{bmatrix}^H \begin{bmatrix} A_{11}x_1 \\ A_{21}x_1 \end{bmatrix} = x_1^H A_{11} x_1.$$

Gleichheit gilt genau für $x_1 = 0$. Daher ist A_{11} positiv definit und nach der letzten Bemerkung invertierbar. Dies beweist die Wohldefiniertheit von S, und wir haben

$$S^H = A_{22}^H - A_{12}^H A_{11}^{-H} A_{21}^H = A_{22} - A_{21}A_{11}^{-1}A_{12} = S.$$

Betrachte nun $x = [x_1, x_2]^T$ mit $x_1 = -A_{11}^{-1}A_{12}x_2$. Dann gilt

$$\begin{aligned} 0 \le x^H A x &= \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^H \begin{bmatrix} A_{11}x_1 + A_{12}x_2 \\ A_{21}x_1 + A_{22}x_2 \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^H \begin{bmatrix} 0 \\ -A_{21}A_{11}^{-1}A_{12}x_2 + A_{22}x_2 \end{bmatrix} \\ &= \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^H \begin{bmatrix} 0 \\ Sx_2 \end{bmatrix} = x_2^H S x_2. \end{aligned}$$

Weil Gleichheit nur im Fall $x_2 = 0$ gilt, ist S ebenfalls positiv definit. □

Definition 6.47. *Eine Zerlegung $A = LL^H$ mit unterer Dreiecksmatrix L, die positive Diagonaleinträge besitzt, heißt* ***Cholesky-Zerlegung*** *von A.*

Die Cholesky-Zerlegung ist also eine spezielle LR-Zerlegung. Im folgenden Satz sehen wir, dass eine Cholesky-Zerlegung einer positiv definiten Matrix ohne Pivotisierung berechnet werden kann.

Satz 6.48 (Existenz der Cholesky-Zerlegung). *Genau dann existiert die Cholesky-Zerlegung von A, wenn A positiv definit ist.*

Beweis. Sei $A = LL^H$. Dann gilt $A^H = (L^H)^H L^H = LL^H = A$. Wegen

$$x^H A x = x^H L L^H x = \|L^H x\|_2^2 \ge 0$$

und weil L^H regulär ist, gilt $x^H Ax > 0$ für alle $x \in \mathbb{K}^n \setminus \{0\}$.

Sei umgekehrt A positiv definit. Wir zeigen per Induktion, dass die Cholesky-Zerlegung existiert. Sei $n = 1$. Wegen $a_{11} > 0$ gilt mit $\ell_{11} = \sqrt{a_{11}}$, dass $[a_{11}] = [\ell_{11}][\overline{\ell}_{11}] = LL^H$. Angenommen, die Aussage gilt für $n-1$. Betrachte

$$A = \begin{bmatrix} a_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

mit $A_{21} = A_{12}^H$ und das Schur-Komplement $S = A_{22} - \frac{1}{a_{11}} A_{21} A_{12}$ von A bzgl. a_{11}. Nach Lemma 6.46 sind $a_{11} > 0$ und S positiv definit. Also ist $\ell_{11} := \sqrt{a_{11}} > 0$ und nach Induktionsannahme besitzt S eine Cholesky-Zerlegung $S = \hat{L}\hat{L}^H$. Definiere

$$L = \begin{bmatrix} \ell_{11} & 0 \\ \frac{1}{\ell_{11}} A_{21} & \hat{L} \end{bmatrix}.$$

Dann gilt

$$LL^H = \begin{bmatrix} \ell_{11} & 0 \\ \frac{1}{\ell_{11}} A_{21} & \hat{L} \end{bmatrix} \begin{bmatrix} \overline{\ell}_{11} & \frac{1}{\ell_{11}} A_{12} \\ 0 & \hat{L}^H \end{bmatrix} = \begin{bmatrix} a_{11} & A_{12} \\ A_{21} & \frac{1}{a_{11}} A_{21} A_{12} + \hat{L}\hat{L}^H \end{bmatrix} = \begin{bmatrix} a_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = A.$$

□

Eine Berechnungsformel für die Einträge von L ergibt sich aus

$$a_{ij} = \sum_{k=1}^{j} \ell_{ik} \overline{\ell}_{jk} = \sum_{k=1}^{j-1} \ell_{ik} \overline{\ell}_{jk} + \ell_{ij} \ell_{jj}, \qquad i \geq j,$$

zu

$$\ell_{jj} = \sqrt{a_{jj} - \sum_{k=1}^{j-1} |\ell_{jk}|^2}, \tag{6.11a}$$

$$\ell_{ij} = \frac{1}{\ell_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} \ell_{ik} \overline{\ell}_{jk} \right), \qquad i = j+1, \ldots, n, \tag{6.11b}$$

für $j = 1, \ldots, n$. Man beachte, dass auf der rechten Seite der Formeln nur Einträge der Spalten $1, \ldots, j-1$ von L erscheinen. Die Durchführbarkeit ist durch das Existenzresultat Satz 6.48 gesichert. Insbesondere folgt aus (6.11) die Eindeutigkeit der Cholesky-Zerlegung.

Algorithmus 6.49.
`Input:` positiv definite Matrix $A \in \mathbb{K}^{n \times n}$
`Output:` untere Dreiecksmatrix L mit $A = LL^H$

```
for (j=0; j<n; ++j) {
   l[j*(n+1)] = a[j*(n+1)];
   for (k=0; k+1<j; ++k)
      l[j*(n+1)] = l[j*(n+1)]-abs(l[j+k*n])*abs(l[j+k*n]);
   l[j*(n+1)] = sqrt(l[j*(n+1)]);
   for (i=j+1; i<n; ++i) {
      l[i+j*n] = a[i+j*n];
      for (k=0; k+1<j; ++k)
```

```
            l[i+j*n] = l[i+j*n]-l[i+k*n])*conj(l[j+k*n]);
        l[i+j*n] = l[i+j*n] / l[j*(n+1)];
    }
}
```

Zur Berechnung von ℓ_{ij}, $i \geq j$, sind $2(j-1)$ Multiplikationen und Additionen nötig. Somit werden insgesamt

$$\sum_{j=1}^{n} \sum_{i=j+1}^{n} 2(j-1) = 2 \sum_{j=1}^{n} (n-j)(j-1) = \frac{1}{3} n^3 + O(n^2)$$

Multiplikationen und Additionen sowie n Wurzeln benötigt. Der Aufwand ist deshalb etwa halb so groß wir der der LR-Zerlegung.

Beispiel 6.50. Die Cholesky-Zerlegung von

$$A = \begin{bmatrix} 1 & 3 & -2 \\ 3 & 10 & -10 \\ -2 & -10 & 21 \end{bmatrix}$$

ergibt sich wegen

$$\ell_{11} = \sqrt{1} = 1, \qquad \ell_{22} = \sqrt{a_{22} - \ell_{21}^2} = 1, \qquad \ell_{33} = \sqrt{a_{33} - \ell_{31}^2 - \ell_{32}^2} = 1$$
$$\ell_{21} = \frac{3}{1} = 3, \qquad \ell_{32} = \frac{a_{32} - \ell_{31}\ell_{21}}{\ell_{22}} = -4,$$
$$\ell_{31} = \frac{-2}{1} = -2,$$

zu

$$A = \begin{bmatrix} 1 & & \\ 3 & 1 & \\ -2 & -4 & 1 \end{bmatrix} \begin{bmatrix} 1 & 3 & -2 \\ & 1 & -4 \\ & & 1 \end{bmatrix}.$$

Die Stabilitätsprobleme der LR-Zerlegung finden man nicht bei der Cholesky-Zerlegung. Diese ist immer stabil. Die für die Instabilität der LR-Zerlegung verantwortlichen großen Einträge in den Faktoren L und R treten wegen des folgenden Lemmas bei der Cholesky-Zerlegung nicht auf.

Lemma 6.51. *Sei $A = LL^H$ die Cholesky-Zerlegung von $A \in \mathbb{K}^{n \times n}$. Dann gilt*

$$|\ell_{ij}| \leq \sqrt{a_{ii}} \quad \text{für } i \geq j.$$

Beweis. Vergleicht man die i-ten Diagonaleinträge von A und von LL^H, so erhält man

$$0 < a_{ii} = \sum_{j=1}^{i} |\ell_{ij}|^2.$$

Hieraus folgt $|\ell_{ij}|^2 \leq a_{ii}$, $j \leq i$. □

Wir erhalten als Folgerung von Satz 6.37 und Lemma 6.51 den folgenden Satz, der die Rückwärtsstabilität der Cholesky-Zerlegung belegt.

Satz 6.52. *Sei $A \in \mathbb{K}^{n\times n}$ positiv definit. Die berechnete untere Dreiecksmatrix $\tilde{L}$ erfüllt*

$$\tilde{L}\tilde{L}^H = A + \Delta A$$

für eine Matrix $\Delta A \in \mathbb{K}^{n\times n}$ mit $\|\Delta A\|_\infty \leq c\varepsilon_{\mathbb{F}}\|A\|_\infty$.

6.6 Die QR-Zerlegung nach Householder

Wir haben in den letzten Abschnitten gesehen, dass lineare Gleichungssysteme mit Hilfe einer LR-Zerlegung gelöst werden können. Anstelle der LR-Zerlegung kann aber auch eine QR-Zerlegung verwendet werden.

Definition 6.53. *Sei $A \in \mathbb{K}^{m\times n}$. Eine Zerlegung der Form $A = QR$ mit unitärem $Q \in \mathbb{K}^{m\times m}$ und einer oberen Dreiecksmatrix $R \in \mathbb{K}^{m\times n}$ heißt* ***QR-Zerlegung*** *von A.*

Ist $Ax = b$ zu lösen, so kann bei bekannter Faktorisierung $A = QR$ die Lösung x aus dem System

$$Ax = b \iff QRx = b \iff Rx = Q^H b \tag{6.12}$$

mittels Rückwärtssubstitution berechnet werden. Diese Vorgehensweise ist numerisch stabil im Vergleich zur Verwendung der LR-Zerlegung, weil unitäre Matrizen die Länge erhalten und somit Rundungsfehler nicht übermäßig verstärkt werden.

Definition 6.54. *Jede Matrix $Q \in \mathbb{K}^{n\times n}$ der Form*

$$Q = I - \beta\frac{uu^H}{u^H u}$$

mit einem $u \in \mathbb{K}^n\backslash\{0\}$ und einem $\beta \in \mathbb{K}\backslash\{0\}$, $|\beta|^2 = 2\operatorname{Re}\beta$, wird als ***Householder-Matrix*** *bezeichnet.*

Lemma 6.55. *Sei Q eine Householder-Matrix. Dann gilt*

(i) Q ist unitär. Im Fall $\mathbb{K} = \mathbb{R}$ ist Q symmetrisch.

(ii) Ist $\mathbb{K} = \mathbb{R}$, so gilt $x - Qx \in \operatorname{span}\{u\}$ und $\frac{1}{2}(x + Qx) \in (\operatorname{span}\{u\})^\perp$. Also spiegelt Q den Vektor $x \neq 0$ an der Hyperebene senkrecht zu u.

(iii) Sei $x \in \mathbb{K}^n \setminus \{0\}$. Für $u = x - \alpha e_1$, $|\alpha| = \|x\|_2$, und $\beta = 1 + \frac{x^H u}{u^H x}$ gilt $Qx = \alpha e_1$.

Beweis.

(i) Dass Q unitär ist, sieht man wegen

$$Q^HQ = \left(I - \beta\frac{uu^H}{u^Hu}\right)^H \left(I - \beta\frac{uu^H}{u^Hu}\right) = I - \overline{\beta}\frac{uu^H}{u^Hu} - \beta\frac{uu^H}{u^Hu} + |\beta|^2\frac{u(u^Hu)u^H}{(u^Hu)^2} = I$$

(ii) Ist $\mathbb{K} = \mathbb{R}$, so gilt $\beta^2 = 2\beta \Leftrightarrow \beta = 2$. Dann gilt

$$x - Qx = x - \left(x - 2\frac{u(u^Tx)}{u^Tu}\right) = 2\frac{u^Tx}{u^Tu}u$$

und

$$u^T(x + Qx) = u^Tx + u^Tx - 2\frac{(u^Tu)u^Tx}{u^Tu} = 0.$$

(iii) Wegen $u^Hx = (x - \alpha e_1)^Hx = \|x\|_2^2 - \overline{\alpha}x_1$ und

$$u^Hu = (x - \alpha e_1)^H(x - \alpha e_1) = \|x\|_2^2 - \alpha\overline{x}_1 - \overline{\alpha}x_1 + |\alpha|^2 = 2(\|x\|_2^2 - \mathrm{Re}\,(\overline{\alpha}x_1))$$

folgt

$$\begin{aligned} Qx &= x - \beta\frac{u^Hx}{u^Hu}u = x - \frac{u^Hx}{u^Hu}u - \frac{x^Hu}{u^Hu}u = x - 2\frac{\mathrm{Re}\,u^Hx}{u^Hu}u \\ &= x - \frac{\mathrm{Re}\,(\|x\|_2^2 - \overline{\alpha}x_1)}{\|x\|_2^2 - \mathrm{Re}\,(\overline{\alpha}x_1)}u = x - u = \alpha e_1. \end{aligned}$$

□

Bemerkung. Um Auslöschungseffekte zu vermeiden, sollte α in (iii) das entgegengesetzte Vorzeichen der ersten Komponente x_1 von x bekommen. Es sollte also

$$\alpha = -\frac{x_1}{|x_1|}\|x\|_2$$

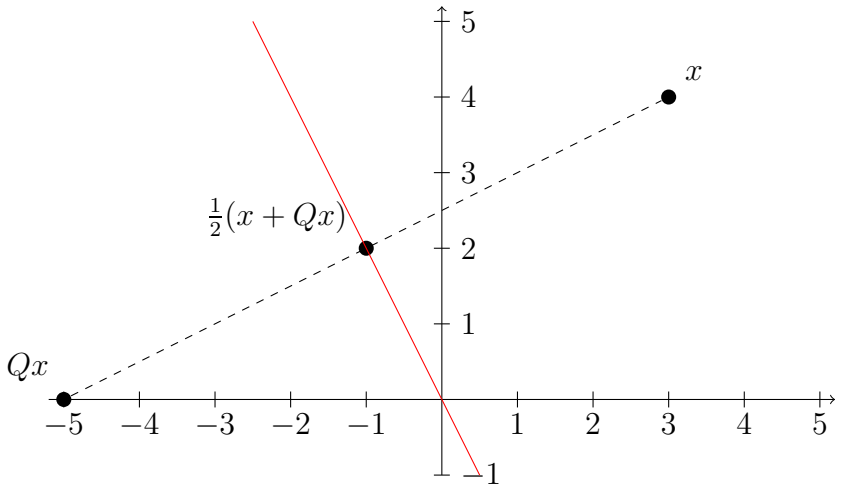
gewählt werden.

Beispiel 6.56. Der Punkt $x = [3, 4]^T \in \mathbb{R}^2$ soll durch Anwendung einer Householder-Matrix auf die x-Achse transformiert werden. Nach Lemma 6.55 (iii) wird dies durch

$$Q = I - \frac{1}{40}\begin{bmatrix}8\\4\end{bmatrix}\begin{bmatrix}8\\4\end{bmatrix}^T$$

erreicht. Tatsächlich gilt

$$Qx = \begin{bmatrix}3\\4\end{bmatrix} - \frac{1}{40}\begin{bmatrix}8\\4\end{bmatrix}\begin{bmatrix}8\\4\end{bmatrix}^T\begin{bmatrix}3\\4\end{bmatrix} = \begin{bmatrix}3\\4\end{bmatrix} - \begin{bmatrix}8\\4\end{bmatrix} = \begin{bmatrix}-5\\0\end{bmatrix}.$$



Sei nun $A \in \mathbb{K}^{m\times n}$, $m \geq n$, mit linear unabhängigen Spalten. Wie in Lemma 6.55 (iii) kann die erste Spalte von A durch Multiplikation mit einer Householder-Matrix $Q_1 \in \mathbb{K}^{m\times m}$ von links in ein Vielfaches des ersten kanonischen Einheitsvektors e_1 transformiert werden:

$$A^{(1)} := Q_1 A, \; A^{(1)} e_1 = \alpha e_1.$$

Sei nun angenommen, dass die Matrix A durch $k-1$ sukzessive Anwendungen von einer Householder-Matrix in folgende partielle obere Dreiecksform gebracht wurde:

$$A^{(k-1)} = Q_{k-1} \cdot \ldots \cdot Q_1 A = \begin{bmatrix} a_{11}^{(1)} & \cdots & \cdots & \cdots & a_{1n}^{(1)} \\ & \ddots & & & \vdots \\ & & a_{kk}^{(k-1)} & \cdots & a_{kn}^{(k-1)} \\ 0 & & \vdots & & \vdots \\ & & a_{mk}^{(k-1)} & \cdots & a_{mn}^{(k-1)} \end{bmatrix}.$$

Im k-ten Schritt soll $A^{(k-1)}$ so transformiert werden, dass $Q_k A^{(k-1)}$ bis zur k-ten Spalte eine obere Dreiecksmatrix ist. Um die ersten $k-1$ Zeilen und Spalten unverändert zu lassen, wählen wir

$$Q_k = \begin{bmatrix} I_{k-1} & 0 \\ 0 & \hat{Q}_k \end{bmatrix},$$

wobei $\hat{Q}_k \in \mathbb{K}^{(m-k+1)\times(m-k+1)}$ eine Householder-Matrix ist, die den Vektor

$$\begin{bmatrix} a_{kk}^{(k-1)} \\ \vdots \\ a_{mk}^{(k-1)} \end{bmatrix} \neq 0$$

auf ein Vielfaches von $e_1 \in \mathbb{K}^{m-k+1}$ transformiert. Weil $\hat{Q}_k$ unitär ist, trifft dies auch auf Q_k zu.

Nach n Schritten erhält man also die obere Dreiecksmatrix

$$A^{(n)} = Q_n \cdot \ldots \cdot Q_1 A = \begin{bmatrix} a_{11}^{(1)} & \cdots & a_{1n}^{(1)} \\ & \ddots & \vdots \\ & & a_{nn}^{(n)} \\ & & \end{bmatrix} =: R.$$

Setzen wir also $Q = Q_1^H \cdot \ldots \cdot Q_n^H$, so erhalten wir die gewünschte Zerlegung $A = QR$.

Bemerkung.

(a) Für die QR-Zerlegung von $A \in \mathbb{K}^{m\times n}$ mittels Householder-Spiegelung werden $2mn^2 - \frac{2}{3}n^3$ arithmetische Operationen benötigt.

(b) Die Einträge von Q sollten niemals berechnet werden. Effizienter ist es, die Vektoren u zu speichern. Diese können in gerade frei gewordene Spalten der Matrix A abgelegt werden.

(c) Auch für die Berechnung der Matrix-Vektor-Multiplikation wie bei der rechten Seite in (6.12) kann auf die explizite Darstellung der Matrix verzichtet werden. Es gilt nämlich

$$Qx = x - \beta \frac{u^H x}{u^H u} u,$$

d.h. Qx entsteht als Linearkombination der Vektoren x und u, was mit $O(m)$ Operationen im Vergleich zu $O(m \cdot n)$ Operationen bei einträgeweiser Multiplikation durchgeführt werden kann.

Index