Vorlesungsmitschriften

Wintersemester 20/21

Universität Bonn *
Semester 1

Fabian

30.01.2022

Online aufrufbar auf
https://github.com/git-fabus/lecutres/blob/main/notes.pdf.
Veränderungsvorschläge und Verbesserungen bitte an git-fabus@uni-bonn.de.

*Letzte Änderung: `(None)` Commit: `(None)`

Vorwort

Dies sind meine persönlichen Vorlesungsmitschriften. Das Dokument ist in 3 große Teilbereiche unterteilt, dazu zählen:

- Algorithmische Mathematik 1
- Analysis 1
- Lineare Algebra 1

Das Dokument wird stetig verbessert und es werden nach und nach immer mehr Aufgaben, Lösungen etc. erscheinen.

Inhaltsverzeichnis

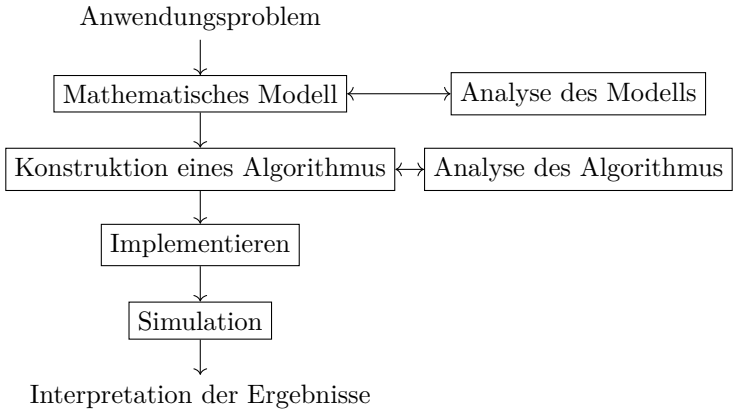
Teil I

Algorithmische Mathematik 1

Einführung

Algorithmische Mathematik -Was ist das?

Gegenstand der Alma ist die Konstruktion und Analyse effizienter Algorithmen zur Lösung mathematischer Problemstellungen mit Hilfe des Computers.
Damit liegt sie im Bereich der Angewandten Mathematik. Konkrete Problemstellungen ergeben sich oft aus technischen Problemen, Naturwissenschaften, Medizin etc.
Man kann hierbei annehmen, dass verschiedene Problemstellungen aus der Anwendungen oft zu ähnlichen oder gleichen mathematischen Modellen zurückgeführt werden können.
Diese Probleme können wie folgt aussehen:



0.1 **Beispiel.** Ein typisches Problem ist das lösen eines linearen Gleichungssystems.

Im Rahmen dieser Vorlesung konzentrieren wird uns auf die Teilbereiche

a) Numerik

b) Diskrete Mathematik

c) Statistik

der Angewandten Mathematik

Zahlendarstellung am Computer

1. Zahlensystem

Die Darstellung von Zahlen basiert auf sogenannten *Zahlensystemen*. Diese Zahlensysteme unterscheiden sich in der Wahl des zugrundeliegendes Alphabets. Unsere Zahl entspricht dann einem Wort, bestehend aus Elementen des Alphabets.

1.1 **Definition** (Alphabet). Es sei $\mathbb{N} = \{1, 2, 3 \ldots\}$ und $b \in \mathbb{N}$.
Wir bezeichnen mit $\sum_b$ das Alphabet des *b-adischen Zahlensystems*

1.2 **Beispiel.** Verschiedene Zahlensysteme

a) Dezimalsystem: $\sum_{10} = \{0, 1, \ldots, 9\}$
wie $(384)_10$

b) Dual bzw. Binärsystem $\sum_2 = \{0, 1\}$
wie $(42)_10 = (101010)_2$

c) ...

Um die Zahlendarstellung sinnvoll zu nutzen Ergibt sich folgendes:

1.3 **Satz.** Seien $b, n \in \mathbb{N}, b > 1$.
Dann ist jede ganze, nicht-negative Zahl z mit $0 \leq z \leq b^n - 1$ *eindeutig* als Wort der Länge n über $\sum_b$ darstellbar durch:

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

mit $z_i \in \sum_b$ für alle $i = 0, 1, \ldots, n-1$.
Wir schreiben vereinfachend:

$$z = (z_{n-1}, \ldots, z_0)_b$$

Beweis. • Induktion

◊ *Induktionsanfang*: $\underline{z < b}$ hat die eindeutige Darstellung $z_0 = z$ und $z_i = 0$ sonst.

◊ *Induktionsschritt*: $\underline{z - 1 \to z \geq b}$

Betrachte
$$z = \left\lfloor \frac{z}{b} \right\rfloor \cdot b + (z \mod b)$$
Da $\hat{z} < z$ besitzt die eindeutige Darstellung
$$\hat{z} = (z_{\hat{n-1}}, \ldots, \hat{z_0})_b$$
Bemerke $z_{\hat{n-1}} = 0$, da
$$\left(z_{\hat{n-1}} b^{n-1}\right) b \leq z \leq b^n - 1$$
Also ist z_b eine n-stellige Darstellung von z in b-adischen Zahlensystem

- Eindeutigkeit wird durch Widerspruch gezeigt.
  Angenommen: Es gibt zwei verschiedene Darstellungen
  $$z = \left(z_{n-1}^{(2)}, \ldots, z_0^{(2)}\right)_b = \left(z_{n-1}^{(1)}, \ldots, z_0^{(1)}\right)_b$$
  Sei $m \in \mathbb{N}$ der größte Index mit $z_m^{(1)} \neq z_m^{(2)}$,
  Ohne Beschränkung der Allgemeinheit kann gesagt werden $z_m^{(1)>z_m^{(2)}}$.
  Dann müssten die Stellen $0, 1, \ldots, m-1$ den niedrigeren Wert von $z_m^{(2)}$ kompensieren.
  Die größte mit diesen Stellen darstellbare Zahl ist aber:
  $$\sum_{i=0}^{m-1} (b-1) b^i = (b-1) \sum_{i=0}^{m-1} b^i = b^m - 1.$$
  Da b^m der kleinstmögliche Wert der fehlende Stelle m ist, kann diese aber nicht kompensiert werden. Dies ist ein Widerspruch.

□

Durch den Beweis ergibt sich sofort ein Algorithmus zur Umwandlung einer Zahl in ein anderes Zahlensystem:

1.4 Beispiel. Umwandlung von $(1364)_{10}$ in das Oktalsystem.

- $1364 = 170 \cdot 8 + 4$
- $170 = 21 \cdot 8 + 2$
- ...
- $0 \cdot 8 + 2$

$\Longrightarrow (2524)_8$

Daten: Dezimalzahl $z \in \mathbb{N}_0$, Basis $b \in \mathbb{N}$
Ergebnis: b-adische Darstellung $(z_{n-1}, \ldots, z_0)_b$
Initialisiere $i = 0$
solange $z > 0$ **tue**
  $z_i = z \mod b$
  $z = \lfloor \frac{z}{b} \rfloor$
  $i = i + 1$
Ende

Algorithmus 1: Bestimmung der b-adischen Darstellung

Beobachtung Wir sehen, dass das Honor-Schmea nur eine Schleife, Additionen und Multiplikationen benötigt. Diese Operationen können wir am Computer durchführen. Im Gegensatz dazu steht die Potenz b^i in modernen Programmiersprachen zwar zur Verfügung, wird aber im Hintergrund oft auf Multiplikationen zurückgeführt. Man überprüft leicht, dass das Honor-Schema weniger Multiplikationen benötigt und somit schnellt ist.

2. Vorzeichen-Betrag-Darstellung

Um auch Zahlen mit Vorzeichen am Computer darstellen zu können, betrachten wir im Folgenden die Vorzeichen-Betrag-Darstellung für Binärzahlen. Das Binäralphabet besteht nur aus 0 und 1, welche wir auch als *Bits* bezeichnen. Bei einer Wortlänge von n Bits wir das erste Bit als Vorzeichen verwendet, die restlichen $n - 1$-Bits für den Betrag der Zahl. Da die 0 die Darstellung $+0$ und $-O$ besitzt, können wir insgesamt $2^n - 1$ Zahlen darstellen.

2.5 Beispiel. Für $n = 3$

Bitmuster	Dezimaldarstellung
000	$+0$
001	$+1$
...	...
100	-0
...	...
111	-3

Aber: Diese Darstellung am Computer ist unpraktisch, da die vier Grundrechenarten auf Hardwareebene typischerweise mit Hilfe von Addition und Zusatz-Logik umgesetzt werden.

Lösung: Komplementdarstellung

3. Komplementdarstellung

3.6 Definition ((b-1)-Komplement). Sei $z = (z_{n-1} \dots z_1 z_0)_b$ eine n-stellige b-adische Zahl. Das *(b-1)-Komplement* $K_{b-1}(z)$ ist definiert als:

$$K_{b-1} = (b-1-z_{n-q}, \dots, b-1-z_0)_b$$

Geben wir hierzu direkt ein paar Beispiele an

3.7 Beispiel. Komplemente

- $K_9((325)_{10}) = (674)_{10}$ (9er-Komplement im 10-er System)
- $K_1((10110)_2) = (01001)_2$ (1er-Komplement im 2er System)

3.8 Definition (b-Komplement). Das b-Komplement einer b-adischen Zahl $z \neq 0$ ist definiert als

$$K_b(z) = K_{b-1}(z) + (1)_b$$

3.9 Beispiel. • $K_{10}((325)_{10}) = (674)_{10} + (1)_{10} = (675)_{10}$

3.10 Lemma. Für jede n-stellige b-adische Zahl z gilt:

- i) $z + K_{b-1}(z) = (b-1, \dots, b-1)_b = b^n - 1$
- ii) $K_{b-1}\left(K_{b-1}(z)\right) = z$

Ist außerdem $z \neq 0$ so gilt:

- iii) $z + K_b(z) = b^n$
- iv) $K_b(K_b(z)) = z$

Beweis. Hilfssatz

(i) Durch nachrechnen:

$$\begin{aligned}
z + K_{b-1}(z) &= (z_{n-1} \dots z_0)_b + (b-1-z_{n-1}, \dots, b-1-z_0)_b \\
&= \sum_{i=0}^{n-1} z_i b^i + \sum_{i=0}^{n-1} (b-1-z_i) b^i \\
&= \sum_{i=0}^{n-1} (b-1) b^i = (b-1, \dots, b-1)_b \\
&= (b-1) \sum_{i=0}^{n-1} b^i \\
&= (b-1) \left(\frac{b^n - 1}{b-1} \right) \\
&= b^n - 1
\end{aligned}$$

- (ii) per Definition
- (iii) Nachrechnen:

$$z + K_b(z) = z + K_{b-1} + 1 = b^n - 1 + 1 = b^n$$

- Definiere $\hat{z} = K_b(z) = K_{b-1}(z) + (1)_b > 0$ und rechne

$$z + K_b(z) = b^n = \hat{z} + K_b\hat{z} + K_b(K_b(z)) \implies \text{Behauptung.}$$

□

3.11 Bemerkung. Modifikation

- Die 3. Aussage gilt auch für $z = 0$, falls man dann bei der Addition von 1 die Anzahl der Stellen erweitert.
- die 4. Aussage gilt für $z = 0$, falls überall im Beweis modulo b^n gerechnet wird.

Außerdem impliziert die 3. Aussage des Lemmas, dass

$$K_b(z) = b^n - z. \tag{3.1}$$

Dies können wir geschickt zum Darstellen der b^n verschiedenen ganzen Zahlen z mit

$$-\left\lfloor \frac{b^n}{2} \right\rfloor \leq z \leq \left\lceil \frac{b^n}{2} \right\rceil - 1$$

nutzen. Diesen Bereich nennt man darstellbaren Bereich.

3.12 Definition (b-Komplement-Darstellung). Die b-Komplement-Darstellung $(z)_{K_b} = (z_{n-1} \dots z_0)_b$ einer Zahl $z \in \mathbb{Z}$ im darstellbaren Bereich ist definiert als:

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

3.13 Beispiel. Der darstellbare Bereich.

- Sei $b = 10$, $n = 2$.
  Dann impliziert (3.1), dass

$$K_{10}(50) = 10^2 - 50 = 50$$

$$K_{10}(49) = 100 - 49 = 51.$$

Der darstellbare Bereich ist nun

$$-50 \leq z \leq 49$$

und hat konkrete Darstellungen:

Darstellung	Zahl
0	+0
1	+1
...	...
49	+49
50	−50
...	...
99	−1

- Sei $b = 2$, $n = 3$ Der darstellbare Bereich ist $-4 \leq z \leq 3$.

Bitmuster	Dezimaldarstellung
000	0
001	1
...	...
100	−4
...	...
111	−1

Wir betrachten nun Addition und Subtraktion zweier Zahlen in b-Komplement-Darstellung. Hierzu bezeichne $(x)_{K_b} \oplus (y)_{K_b}$ die ziffernweise Addition der Darstellungen von x und y mit Übertrag ("schriftlich rechnen"), wobei ein eventueller Überlauf auf die $(n+1)$-te Stelle vernachlässigt wird (wir rechen also immer mit modulo b^n.

3.14 **Satz** (Addition in b-Komplement-Darstelllung). Seien x und y zwei n-stellige, b-adische Zahlen und x,y und $x + y$ im darstellbaren Bereich. Dann gilt:

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

Beweis. Wir betrachten dazu mehrere Fälle.

- Fall $x, y \geq 0$:

$$\begin{aligned}(x)_{K_b} \oplus (y)_{K_b} &\overset{\text{Def.}}{=} ((x)_b + (y)_b) \mod b^n \\ &\overset{\text{Def.}}{=} (x + y) \mod b^n \\ &\overset{\text{Def.}}{=} (x + y)_{K_b}\end{aligned}$$

Der letzte Schritt ist möglich, da $(x + y)_{K_b}$ im darstellbaren Bereich sind.

- Fall $x, y < 0$

$$\begin{aligned}(x)_{K_b} \oplus (y)_{K_b} &\overset{Def.}{=} ((K_b(|x|))_b + (K_b(|y|))_b) \mod b^n \\ &= ((K_b(|x|)) + (K_b(|x|))) \mod b^n \\ &= (b^n - |x| + b^n - |y|) \mod b^n \\ &= (x + y) \mod b^n \\ &= (x + y)_{K_b}\end{aligned}$$

- Fall $x \geq 0, y < 0$:

$$\begin{aligned}(x)_{K_b} \oplus (y)_{K_b} &\overset{Def.}{=} ((x)_b + (K_b(|y|))_b) \mod b^n \\ &= (x + K_b(|y|)) \mod b^n \\ &= (x + b^n - |y|) \mod b^n \\ &= (x + y) \mod b^n \\ &= (x + y)_{K_b}\end{aligned}$$

- Fall $x < 0, y \geq 0$: Analog

□

3.15 Satz (Subtraktion in b-Komplement-Darstellung). Seien x und y n-stellige b-adische Zahlen und x,y und x-y im darstellbaren Bereich. Dann gilt:

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

Beweis. • Fall $y = 0$ nichts zu zeigen.

- $y \neq 0$: (3.1) impliziert:

$$-y = K_b(y) - b^n$$

mit modulo b^n-rechnen folgt, dass

$$(-y)_{K_b} = (K_b(y))_{K_b}$$

ist und somit:

$$\begin{aligned}(x - y)_{K_b} &= (x + (-y))_{K_b} \\ &= (x)_{K_b} \oplus (-y)_{K_b} \\ &= (x)_{K_b} \oplus (K_b(y))_{K_b}\end{aligned}$$

□

3.16 Beispiel. Für $b = 10$ und $n = 2$ ist der darstellbare Bereich $-50 \leq z \leq 49$

- $(20 + 7)_{K_{10}} = (20)_{K_{10}} \oplus (7)_{K_{10}} = (27)_{K_{10}} = 27$
- $28 - 5 = 28 + (-5) = (28)_{K_{10}} \oplus (95)_{K_{10}} = (23)_{K_{10}} = 23$
- $-18 - 20 = (-18) + (-20) = \ldots = -28$

Die darstellbaren Zahlen kann man sich beim K_b-Komplement als Zahlenrad vorstellen.

Achtung Ein eventueller Überlauf bzw. Unterlauf wird im Allgemeinen nicht aufgefangen.

3.17 **Beispiel.** In n-stelliger Binärarithmetik ist die größte darstellbare Zahl $x_{max} = (011\ldots1)_{K_2}$ gleich $2^{n-1}-1$. Hingegen ist $x_{max}+1 = (100\ldots0)_{K_2}$ und wir als -2^{n-1} interpretiert.

4. Fest-Komma-Darstellung

4.18 **Definition.** Bei der Festkommadarstellung einer n-stelligen Zahl werden k Vorkomma und n-k Nachkommastellen definiert:

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

4.19 **Beispiel.** Im Zehner System wie gehabt.
Im Binärsystem ergibt sich folgendes: $(101.01)_2 = 2^2 + 2^0 + 2^{-2} = 5.25$

Achtung Im Gegensatz zur Darstellung ganzer Zahlen können bereits bei der Konvertierung von Dezimalzahlen in das b-adische Zahlensystem Rundungsfehler auftreten.

4.20 **Beispiel.** $(0.8)_{10} = (0.110\overline{1100})_2$

Das größte Problem der Festkommadarstellung ist alllerdings, dass der darstellbare Bereich stark eingeschränkt ist und schlecht aufgelöst ist, da der Abstand zwischen zweier Zahlen immer gleich ist.

4.21 **Beispiel.** Die kleinste darstellbare Zahl in Fixkommadarstellung ist

$$z_1 = (0\ldots0.0\ldots01)_b$$

Die zweitkleinste Zahl ist $z_2 = 2z_1$. Wir wollen $x = \frac{z_1+z_2}{2}$ in Fixkommadarstellung darstellen, müssen wir entweder zu z_1 abrunden oder zu z_2 aufrunden.

4.22 Der relative Fehler dieses Runden ist:

$$\frac{|x-z_1|}{|x|} = \frac{1}{3}$$

Analog für z_2.
Solche Fehler machen jede Rechnung unbrauchbar.

5. Gleitkommadarstellung

5.23 **Definition** (Gleitkommadarstellung). Die Gleitkommastellung einer Zahl $z \in \mathbb{R}$ ist gegeben durch:

$$z = +- m \cdot b^e$$

mit einer Matisse m, dem Exponenten e und der Basis b.

Der Einfachheit halber nehmen wir an, dass die Basis für alle Zahlen gleich ist, obwohl sie prinzipiell verschieden sein könnte.

5.24 Beispiel. Die Gleitkommadarstellungen von:

- $(-384.753)_{10} = -3.84753 \cdot 10^2$
- $(0.00042)_{10} = 4.2 \cdot 10^{-4d}$
- $(1010.101)_2 = (1.010101)_2 \cdot 2^3$

Achtung: Die Gleitkommadarstellung ist nicht eindeutig!

5.25 Definition. Die Matisse m heißt normalisiert, falls $m = m_1.m_2m_3 \dots m_t$ mit $1 \leq m_1 \leq b$.

Um die eindeutige, normalisierte Gleitkommadarstellung im Computer zu speichern, müssen wir festlegen, wie viele Stellen für Matisse und Exponent zur Verfügung gestellt werden. Dies resultiert in der Menge

$$F = F(b, t, e_{min}, e_{max}) = \{z = +- m_1.m_2m_3 \dots m_t \cdot b^e | e_{min} \leq e \leq e_{max}\}$$

Da 0 nicht in dieser Form dargestellt werden kann, reserviert man dafür eine spezielle Ziffernfolge in Matisse und Exponent.

5.26 Bemerkung. Das Hidden Bit ist hilfreich

- Für $b = 2$ kann die erste Ziffer m_1 der Matisse weggelassen werden (Hidden Bit)
- Um den Exponenten besser vergleichen zu können, verwendet man oft die Exzess- oder Bias-Darstellung. Durch Addition der Exzesser $|e_{min}| + 1$ wird der Exponetn auf den Bereich $1, 2, \dots, |e_{min}| + e_{max} + 1$ transformiert.

5.27 Beispiel (IEEE 754 Standard). Betrachte binäre Gleitkommazhalen mit 64 Bit auf dem Computer.

- 52 Bit für die Matisse in Hidden Bit Darstellung
- 11 Bit für den Exponenten mit $e_{min} = -1022$ und $e_{max} = 1023$ gespeichert in Exzessdarstellung.
- 1 Bit als Vorzeichen.

Weiter Fälle können Online nachgelesen werden.

Genauigkeit der Gleitkommadarstellung

Da die Menge aller Zahlen in $F = F(b, t, e_{min}, e_{max})$ endlich ist, müssen wir Zahlen in $\mathbb{R} \setminus F$ geeignet annähern.

5.28 Definition. Die Rundung ist eine Abbildung $rd \colon \mathbb{R} \to F$ mit

- $rd(a) = a$, $a \in F$
- $rd(z)$, $z \in \mathbb{R}$ ist gegeben so, dass $|z - rd(z)| = \min_{a \in F}(z - a)$ für alle $z \in \mathbb{R}$.

5.29 Definition (Maschinengenauigkeit). Den maximalen relativen Rundungsfehler ε_{mach} für $z_{min} \leq |z| \leq z_{max}$ nennt man Maschinengenauigkeit. Die Stellen der Mantisse heißen signfikante Stellen. Wenn die Mantisse t-stellig ist, spricht man von t-stelliger Arithmetik.

5.30 Satz (Maschinengenauigkeit von F). Die Maschinengenauigkeit ε_{mach} für $F = F(b, t, e_{min}, e_{max})$ ist:

$$\varepsilon_{mach} = \frac{1}{2} b^{1-t}.$$

Beweis. Betrachte relativen Rundungsfehler ε, der vom Abscheiden nicht signifikanten Stellen herrüht. Sei $z_{min} < x < z_{max}$ und $\tilde{x}$ die durch Abschneiden entstehende Zahl. Dann gilt:

$$\begin{aligned}
\varepsilon &= \frac{|x - \tilde{x}|}{|x|} \\
&= \frac{|x_1.x_2x_3 \ldots x_t x_{t+1} \ldots \cdot b^e - x_1x_2x_3 \ldots x_t \cdot b^e}{|x|} \\
&= \frac{|0.x_{t+1} \ldots | \cdot |b^{e+1-t}|}{|x|}
\end{aligned}$$

Da $|0.x_{t+1} \ldots | < 1$ und $b^e \leq |x|$ gilt:

$$\varepsilon < \frac{b^{e+1-t}}{b^e} = b^{1-t}$$

Da der Rundungsfehler ε_{mach} höchsten halb so groß ist wie der Abschneidefehler folgt die Behauptung. □

Die Maschinengenauigkeit ε_{mach} ist die wichtigste Größe zur Beurteilung der Genauigkeit von Gleitkommarechnungen am Computer. Sie gibt Aufschluss zur Anzahl signifikanter Stellen zur Basis b. Die zugehörige Anzahl s Stellen im Dezimalsystem erhält man durch das Auflösen von:

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

nach s. Es folgt daher:

$$s = \lfloor 1 + (t-1) \log_{10}(b) \rfloor$$

Für den IEEE 754 Standard mit $b = 2, t = 53$ erhält man $s = 16$ signifikante Stellen.

Fehleranalyse

6. Rechnerarithmetik

6.1 Beispiel. Betrachte $F = F(10, 5, -4, 5)$ und Maschinenzahlen

$$x = 2.5684 \cdot 10^0 = 2.56840000$$
$$y = 3.2791 \cdot 10^{-3} = 0.0032791$$

Es gilt:

$$\left.\begin{aligned} x + y &= 2.5716792 \\ x - y &= 2.5651209 \\ x \cdot y &= 0.00842204044 \\ \tfrac{x}{y} &= 783.2637004 \end{aligned}\right\} \notin F$$

6.2 Bemerkung. Die Menge $F(b, t, e_{min}, e_{max})$ ist nicht abgeschlossen bezüglich der Grundrechenarten und können somit im Allgemeinen nicht im Computer implementiert werden.

Lösung Wir runden das Ergebnis und implementieren so eine Pseudoarithmetik. Das bedeutet, wir ersetzen $\circ \in \{+, -, \cdot, \div\}$ durch $\boxdot \in \{\boxplus, \boxminus, \boxdot, \boxslash\}$ definiert durch:

$$x \boxdot y \coloneqq rd(x \circ y) \tag{6.1}$$

Auf Hardwareebene wird üblicherweise mit einer längeren Matisse gearbeitet und dann normalisiert und gerundet. Dies entspricht dem IEEE 754 Standard.

6.3 Bemerkung. Für $|x|, |y|, |x \circ y| \in [z_{min}, z_{max}]$ impliziert (6.1), dass

$$\frac{|x \boxdot y - x \circ y|}{|x \circ y|} = \frac{rd(x \circ y - x \circ y)}{|x \circ y|} \leq \varepsilon_{mach}$$

Das bedeutet, dass $\boxdot$ im Computer bestmöglich umgesetzt ist.

6.4 Beispiel. Betrachte $F = F(10, 5, -4, 5)$

- Setze:
  - $a = 0.98765$

- $b = 0.012424$
- $c = -0.0065432$

Dann gilt:

$$(a + b) + c = a + (b + c) = 0.9925208$$

Numerisch gilt:

$$(0.98765 \boxplus 0.012424) \boxminus 0.0065432 = rd(0.9935568) = 0.99356$$

und

$$0.98765 \boxplus (0.012424 \boxminus 0.0065432) = rd(0.9935308) = 0.99353$$

- Setze
  - $a = 4.2832$
  - $b = -4.2821$
  - 5.7632

  Dann gilt

$$(a + b) \cdot c = a \cdot c + b \cdot c = 0.006339520000001$$

  Numerisch gilt:

6.5 Bemerkung. Mathematisch äquivalente Algorithmen auf Fließkommazahlen können je nach Implementierung zu wesentlich unterschiedlichen Ergebnissen führen, selbst wenn die Eingangszahlen exakt dargestellt werden.

6.1. Auslöschung

Unglücklicherweise pflanzen sich numerische Fehler, zum Beispiel durch Rundung im Verlauf eines Algorithmus fort.

6.6 Lemma (Fehlerfortpflanzung). Es seien $x, y \in \mathbb{R}$ mit Datenfehlern $\Delta x, \Delta y \in \mathbb{R}$ behaftet, welche

$$|\frac{\Delta x}{x}|, |\frac{\Delta y}{y}| \ll 1$$

erfüllen. Für $\circ \in \{+, -, \cdot, \div\}$ gilt für den fortgepflanzten Fehler

$$\Delta(x \circ y) \coloneqq (x + \Delta x) \circ (y + \Delta y) - x \circ y,$$

dass

$$\begin{aligned} \frac{\Delta(x \pm y)}{x \pm y} &= \frac{x}{x \pm y}\frac{\Delta x}{x} \pm \frac{y}{x \pm y}\frac{\Delta y}{y} \\ \frac{\Delta(x \cdot y)}{xy} &\approx \frac{\Delta x}{x} + \frac{\Delta y}{y} \\ \frac{\Delta(\frac{x}{y})}{\frac{x}{y}} &\approx \frac{\Delta x}{x} - \frac{\Delta y}{y} \end{aligned}$$

Hierbei bedeutet "$\approx$", dass Terme mit $(\Delta x)^2, (\Delta y)^2, \Delta x \Delta y$ vernachlässigt werden.

Beweis. Übung. □

Für $\cdot, \div$ addieren bzw. subtrahieren sich die Fehler.

Achtung: Ist $|x \pm y|$ wesentlich kleiner als $|x|$ oder $|y|$, kann der relative Fehler massiv verstärkt werden. Dieses Phänomen heißt Auslöschung.
Bei der Konstruktion von Algorithmen sollte Auslöschung möglichst vermieden werden.

6.7 Beispiel. Betrachte

$$x^2 - 2px + q = 0 \tag{6.2}$$

mit Lösungen:

$$x_{1,2} = p \pm \sqrt[2]{p^2 - q}$$

Daten: $p, q \in \mathbb{R}$ so ,dass (6.2) lösbar ist
Ergebnis: Nullstellen x_1, x_2 von (6.2).
$d = \sqrt{p \cdot p - q}$
$x_1 = p + d$
$x_2 = p - d$

Algorithmus 2: naive Nullstellenberechung

Mit $p = 100$, $q = 1$ in dreistelliger, dezimaler Gleitkommaarithmetik ergibt sich:

$$\begin{aligned} d = \sqrt{10000 - 1} = \sqrt{rd(9999)} = \sqrt{10000} &= 100 \\ x_1 = 100 + 100 &= 200 \\ x_2 = 100 - 100 &= 0 \end{aligned}$$

Die exakten Werte sind $x_1 \approx 199.99, x_2 \approx 0.00500$, welche in dreistelliger, dezimaler Gleitkommaarithmetik als $x_1 = 200, x_2 = 0$ dargestellt werden.
Gemäß 5.30 ist die Maschinengenauigkeit:

$$\varepsilon_{mach} = \frac{1}{2} b^{1-t} = \frac{1}{2} 10^{1-3} = 0.005$$

Der relative Fehler $\frac{|0-0.005|}{|0.005|} = 1$ ist also zu 100 Prozent falsch.

Wir können die Genauigkeit von x_2 erhöhen, indem wir den Wurzelsatz von Vieta $x_1 x_2 = q$ verwenden.

Daten: $p, q \in \mathbb{R}$ so ,dass (6.2) lösbar ist.
Ergebnis: Nullstellen x_1, x_2 (6.2).
$d = \sqrt[2]{p \cdot p - q}$
wenn $q \geq 0$ **dann**
  $x_1 = p + d$ **sonst**
    $x_1 = p - d$
  Ende
Ende
$x_2 = \frac{q}{x_1}$

Algorithmus 3: verbesserte Nullstellenberechnung

Wir erhalten nun $d = 100, x_1 = 200, x_2 = \frac{1}{200} = 0.005$

7. Vorwärts- und Rückwertsanalyse

Abstrakt gesehen entspricht das Lösen eines Problems dem Auswerten einer Funktion f. Beim numerischen Auswerten von f können verschiedene Fehler passieren:

7.8 Definition (Fehlerarten). Sei $D \subset \mathbb{R}$ eine Menge von Eingangsdaten und $W \subset \mathbb{R}$ die Menge der möglichen Ergebnisse. Wir unterscheiden folgende Fehlerarten bei der Auswertung von $f\colon D \to W$.

- Datenfehler Typischerweise sind die Eingangsdaten $x \in D$ nicht exakt, sondern mit einem Datenfehler Δx behaftet. Die gestörten Eingangsdaten $\tilde{x} = x + \Delta x$ produzieren einen Fehler $f(x + \Delta x) - f(x)$.
- Verfahrensfehler Exakte Verfahren enden bei exakter Rechnung nach endlich vielen Operationen mit dem exakten Ergebnis. Näherungsverfahren enden in Abhängigkeit bestimmter Kriterien mit einer Näherung $\tilde{y}$ für die Lösung $y \in W$.
- Rundungsfehler Verursacht durch Maschinenzahlen und Rundungen während der Arithmetik.

Ein Teilgebiet der Numerik versucht diese Fehler durch eine Fehleranalyse zu quantifizieren. Hierzu verfolgt man die Auswirkungen von allen Fehlern, die in den einzelnen Schritten vorkommen können.

7.9 Definition (Analysearten). Bei der *Vorwärtsanalyse* wird der Fehler von Schritt zu Schritt verfolgt und der akkumulierte Fehler für jedes Teil-Ergebnis abgeschätzt.
Bei der *Rückwertsanalyse* geschieht die Verfolgung des Fehlers hingegen so, dass jedes Zwischenergebnis als exakt berechnetes Ergebnis zu gestörten Daten interpretiert wird, d.h , der akkumulierte Fehler wird als Datenfehler interpretiert.

7.10 Beispiel. Betrachte $f(x, y) = x + y$

- Vorwärtsanalyse: $\boxed{f} = x \boxplus y = (x + y)(1 + \varepsilon)$
- Rückwärtsanalyse: $x \boxplus y = x(1 + \varepsilon) + y(1 + \varepsilon) = f(x(1 + \varepsilon), y(1 + \varepsilon)$

mit $|\varepsilon| \leq \varepsilon_{mach}$

In der Praxis ist die Vorwärtsanalyse kaum durchführbar. Für die meisten Algorithmen ist, wenn überhaupt, nur eine Rückwärtsanalyse bekannt.

7.11 Beispiel. Fortsetzung von Beispiel 6.7
Führe Rückwärtsanalyse durch zu Algorithmus 2 für die Betrags-kleinere Nullstelle x_2 durch: Dann ist

$$f(p, q) = p - \sqrt[2]{p^2 - q}$$

mit $p > 0$. Für $|\varepsilon_i| \leq \varepsilon_{mach}, i = 1, 2, 3, 4$ betrachten wir:

$$\begin{aligned}\left(p - \sqrt[2]{(p^2(1+\varepsilon_1) - q)(1+\varepsilon_2)}(1+\varepsilon_3)\right)(1+\varepsilon_4) &= p(1+\varepsilon_4) - \sqrt[2]{(p^2(1+\varepsilon_1) - q)(1+\varepsilon_2)(1+\varepsilon_3)^2(1+\varepsilon_4)^2} \\ &= p^2(1+\varepsilon_1)(1+\varepsilon_3)^2(1+\varepsilon_4)^2 - q(1+\varepsilon_2)(1+\varepsilon_3)^2(1+\varepsilon_4)^2 \\ &= \dots \\ &= p(1+\varepsilon_4) - \sqrt[2]{p^2(1+\varepsilon_4)^2 - q(1+\varepsilon_7)} \\ &= f(p(1+\varepsilon_4), q(1+\varepsilon_7))\end{aligned}$$

Die Abschätzung für ε_7 explodiert, falls $0 < |q| \ll 1 < p$. Dies war in Beispiel 6.7 der Fall.

8. Kondition und Stabilität

Gegeben sei eine stetige und differenzierbare Funktion

$$f\colon \mathbb{R} \to \mathbb{R}, x \mapsto y = f(x)$$

Für fehlerhafte Daten $x + \Delta x$ mit kleinen Fehler Δx gilt:

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

Für den absoluten Datenfehler Δy gilt:

$$\Delta y = f(x+\Delta x) - f(x) \approx f'(x) \cdot \Delta x$$

und für den relativen Fehler:

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

8.12 Definition (Konditionszahlen). Die Zahl

$$K_{abs} = |f'(x)|$$

heißt *absolute Konditionszahl* des Problems $x \mapsto f(x)$. Für $f(x) \cdot x \neq 0$ heißt

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

die entsprechende *relative Konditionszahl*. Ein Problem heißt schlecht konditioniert, falls eine dieser beiden Konditionszahlen deutlich größer als 1 sind. Ansonsten heißt das Problem gut konditioniert.

8.13 Beispiel. Konditionierung der Addition und Multiplikation

- Für die Addition $f(x) = x + a$ gilt

$$K_{rel} = |\frac{f'(x)x}{f(x)}| = |\frac{x}{x+1}|$$

$\implies$ K_{rel} ist groß, falls $|x + a| \ll |x|$.

- Für die Multiplikation $f(x) = x \cdot a$ gilt

$$K_{rel} = \frac{|f'(x)x|}{|f(x)|} = \frac{|ax|}{|ax|} = 1$$

$\implies$ Die absolute Kondition ist schlecht, falls $1 \ll q$. Die relative Kondition ist immer gut.

8.14 Definition. Erfüllt die Implementierung eines Algorithmus $\boxed{f}$ zur Lösung eines Problems $x \mapsto f(x)$ die Abschätzung

$$|\frac{\boxed{f} - f(x)}{f(x)} \leq C_V K_{rel} \varepsilon_{mach}$$

mit einem mäßig großen $C_V > 0$, so wird der Algorithmus *vorwärtsstabil* genannt. Ergibt die Rückwärtsanalyse $\boxed{f}(x) = f(x + \Delta x)$ mit

$$|\frac{\Delta x}{x} \leq C_R \varepsilon_{mach}$$

mit $C_R > 0$ nicht zu groß, so heißt der Algorithmus $\boxed{f}$ *rückwärtsstabil*

8.15 Satz. Jeder rückwärtsstabile Algorithmus ist auch vorwärtsstabil

Beweis. Übung. □

Oft ist Rückwärtsstabilität einfacher nachzuweisen.

Faustregel zu konditionierten Probleme

- Gut konditioniertes Problem + stabiler Algorithmus
  $\implies$ Gute numerische Resultate.
- Schlecht konditioniertes Problem oder instabiler Algorithmus
  $\implies$ Fragwürdige Ergebnisse.

8.16 Beispiel. Fortsetzung von Beispiel 7.11
Die Rückwärtsanalyse hat gezeigt: Falls $0 < |q| \ll 1 < p$ wird der numerische Fehler untragbar. Unsere Abbildung ist:

$$f(q) = p - \sqrt[2]{p^2 - 1}$$

Als Konditionszahlen ergeben sich:

$$K_{abs} = |f'(q)| = |\frac{1}{2\sqrt[2]{p^2 - q}}| < 1$$

$$\begin{aligned} K_{rel} &= |\frac{f'(q)q}{f(q)}| \\ &= |\frac{q}{2\sqrt[2]{p^2-q}(p - \sqrt[2]{p^2-q})(p + \sqrt[2]{p^2-q})}| \\ &= \frac{1}{2}|\frac{p + \sqrt[2]{p^2-q}}{\sqrt[2]{p^2-q}} \\ &\approx \frac{1}{2}|\frac{p+p}{p}| \approx 1 \end{aligned}$$

$\Longrightarrow$ Nullstellenberechung ist ein gut konditioniertes Problem, aber Algorithmus 2 muss instabil sein

Dreitermrekursion

9. Theoretische Grundlagen

9.1 Definition. Für gegebene p_0 und p_1 heißt eine Rekursion der Form

$$p_k = a_k p_{k-1} + b_k p_{k-2} + c_k,\ k = 2, 3, \ldots \tag{9.1}$$

mit $b_k \neq 0$ eine *Dreitermrekursion*. Die zugehörige *Rückwärtsrekursion* ist

$$p_{k-2} = -\frac{a_k}{b_k} p_{k-1} + \frac{1}{b_k} p_k - \frac{c_k}{b_k},\ k = n, n-1, \ldots \tag{9.2}$$

Ist $b_k = 1$, das heißt (9.1) und (9.2) gehen durch vertauschen von p_{k-2} und p_k auseinander hervor, so heißt die Rekursion symmetrisch. Gilt $c_k = 0$ für alle k, so heißt die Rekursion homogen.

9.2 Beispiel. Die Fibonacci-Zahlen sind rekursiv definiert durch:

$$p_k = p_{k-1} + p_{k-2}$$

mit $p_0 = 0, p_1 = 1$. Es gilt $a_k = b_k = 1$.
Sei

$$p_k(x) = 2\cos(x) p_{k-1}(x) - 1 \cdot p_{k-2}(x)$$

mit $p_0 = 1$ und $p_1 = \cos(x)$. Man kann zeigen, dass

$$p_k(x) = 2\cos(kx)$$

ist.
Die Chebychev-Polynome T_k erfüllen

$$T_k(x) = 2xT_{k-1} - T_{k-2}(x)$$

mit $T_0(x) = 1$ und $T_1(x) = x$

Daten: Koeffizienten von $\{a_k\}_{k=2}^n$, $\{b_k\}_{k=2}^n$,$\{c_k\}_{k=2}^n$, Startwerte p_0, p_1
Ergebnis: Werte von $\{p_k\}_{k=2}^n$
für $k \leftarrow 2$ **bis** n **tue**
| $p_k = a_k p_{k-1} + b_k p_{k-2} + c_k$
Ende

Algorithmus 4: Dreitermrekursion

9.3 Beispiel. Betrachte: $p_0 = 1, p_1 = \sqrt[2]{2} - 1, p_k = -2p_{k-1} + p_{k-2}$, $k = 2, 3, \ldots$
Algorithmus 5 liefert:

Die Oszillationen für höhere k sollten uns misstrauische machen und uns motivieren das Problem genauer anzuschauen.

Wir können homogene Dreitermrekursion schreiben als:

$$\begin{bmatrix} p_k \\ p_{k-1} \end{bmatrix} = \begin{bmatrix} a_k p_{k-1} + b_k p_{k-2} \\ p_{k-1} \end{bmatrix} =: \begin{bmatrix} a_k & b_k \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} p_{k-1} \\ p_{k-2} \end{bmatrix} = A_k \begin{bmatrix} p_{k-1} \\ p_{k-2} \end{bmatrix}$$

Rekursiv folgt:

$$\begin{bmatrix} p_k \\ p_{k-1} \end{bmatrix} = A_k \begin{bmatrix} p_{k-1} \\ p_{k-2} \end{bmatrix} = A_k A_{k-1} \begin{bmatrix} p_{k-2} \\ p_{k-3} \end{bmatrix} = \ldots = A_k \ldots A_2 \begin{bmatrix} p_1 \\ p_0 \end{bmatrix}$$

Offensichtlich gilt für alle $\alpha, \beta \in \mathbb{R}$ und Startwerte p_0, p_1, q_0, q_1 und $B_k \coloneqq A_k \ldots A_2$, dass

$$B_k \begin{bmatrix} \alpha p_1 + \beta q_1 \\ \alpha p_0 + \beta q_0 \end{bmatrix} = \alpha B_k \begin{bmatrix} p_1 \\ p_0 \end{bmatrix} + \beta B_k \begin{bmatrix} q_1 \\ q_0 \end{bmatrix} = \alpha \begin{bmatrix} p_k \\ p_{k-1} \end{bmatrix} + \beta \begin{bmatrix} q_k \\ q_{k-1} \end{bmatrix}$$

Das heißt, die Lösungsfolge $\{p_k\}$ hängt *linear* von den Startwerten $\begin{bmatrix} p_1 \\ p_0 \end{bmatrix} \in \mathbb{R}^2$ ab. Im Falle unseres Beispiels gilt: $a_k = a = -2$, $b_k = b = 1$. Das bedeutet, es gilt:

$$B_k = A^{k-1}, A = \begin{bmatrix} a & b \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} -2 & 1 \\ 1 & 0 \end{bmatrix}$$

Aus den Eigenwerten λ_1, λ_2 der Matrix A kann man die Lösungen der homogenen Dreitermrekursion direkt angeben.

9.4 Satz. Seien λ_1, λ_2 die Nullstellen des *charakteristischen Polynoms*

$$q(\lambda) = \lambda^2 - a\lambda - b \tag{9.3}$$

Dann ist die Lösung der homogenen Dreitermrekursion:

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

gegeben durch:

$$p_k = \alpha\lambda_1^k + \beta\lambda_2^k, k = 2, 3, \ldots$$

mit $\alpha, \beta \in \mathbb{R}$ Lösungen des linearen Gleichungssystems.

$$\alpha + \beta = p_0$$

$$\alpha\lambda_1 + \beta\lambda_2 = p_1$$

Beweis. • Induktion über k

◊ Induktionsanfang: $\underline{k = 0}$: $p_0 = \alpha + \beta = \alpha\lambda_1^0 + \beta\lambda_2^0$

◊ Induktionsschritt: $\underline{k \to k+1}$:

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

Da nach (9.3) λ_1^2 und λ_2^2 Nullstellen sind.

□

Wende 9.4 auf das Beispiel 9.3 an: Mit $a = -2, b = 1$ hat das charakteristische Polynom (9.3) die Nullstellen:

$$\lambda_1 = \sqrt[2]{2} - 1, \lambda_2 = -\sqrt[2]{2} - 1$$

Aus

$$\alpha + \beta = p_0 = 1$$
$$\alpha\lambda_1 + \beta\lambda_2 = p_1 = \sqrt[2]{2} - 1$$

folgt $\alpha = 1, \beta = 0$. Der Satz 9.4 impliziert:

$$p_k = \lambda_1^k > 0, k = 0, 1, 2, \ldots$$

in Beispiel 9.3

Daten: Koeffizienten a,b, Startwerte p_0, p_1
Ergebnis: Werte $\{p_k\}_{k=2}^n$
Initialisiere:

- $\lambda_1 = \frac{a}{2} + \sqrt[2]{\frac{a^2}{4} + b}$
- $\lambda_2 = \frac{a}{2} - \sqrt[2]{\frac{a^2}{4} + b}$
- $\beta = \frac{p_1 - \lambda_1 p_0}{\lambda_2 - \lambda_2}$
- $\alpha = p_0 - \beta$

für $k \leftarrow 2$ **bis** n **tue**
  $p_k = \alpha\lambda_1^k + \beta\lambda_2^k$
Ende

Algorithmus 5: verbesserte Dreitermrekursion

Dieser Algorithmus liefert für Beispiel 9.3 folgende Grafik:
Dies ist in diesem Fall das richtige Ergebnis, da

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

Was geht schief in dem vorherigen Algorithmus 5?
Betrachten wir unser Problem $f\colon \mathbb{R}^2 \to \mathbb{R}$, $f(p_0, p_1) = p_k$ mit gestörten Eingangsdaten: $\hat{p_0} = 1(1+\varepsilon_0)$, $\hat{p_1} = \lambda_1(1+\varepsilon_1)$, $|\varepsilon_0|, |\varepsilon_1| \leq \varepsilon_{mach}$.
Dies ergibt:

$$\tilde{\beta} = \frac{\lambda_1(1+\varepsilon_1) - \lambda_1(1+\varepsilon_0)}{\lambda_2 - \lambda_1} = (\varepsilon_1 - \varepsilon_0)\frac{\lambda_1}{\lambda_2 - \lambda_1}$$

$$\tilde{\alpha} = 1 + \varepsilon_0\frac{\lambda_2 - \lambda_1}{\lambda_2 - \lambda_1} - (\varepsilon_1 - \varepsilon_0)\frac{\lambda_1}{\lambda_2 - \lambda_1} = 1 + \varepsilon_0\frac{\lambda_2}{\lambda_2 - \lambda_1} - \varepsilon_1\frac{\lambda_1}{\lambda_2 - \lambda_1}$$

und somit

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

Der relative Fehler von $\tilde{p_k}$ zu $p_k = \lambda_1$ ist somit :

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

Beobachte: Falls $\frac{\lambda_2}{\lambda_1} > 1$ explodiert der relative Fehler für wachsende k. Dies war in Beispiel 9.3 der Fall.
Falls wir den Begriff der Konditionszahl aus der Definition 8.12 geeignet für Funktion $\mathbb{R}^2 \to \mathbb{R}$ erweitern, sehen wir, dass das Problem schlecht konditioniert ist.

9.5 Definition (Minimallösung). Die Lösung $\{p_k\}$ der Dreiterm-Rekursion (9.1) zu den Startwerten p_0 und p_1 heißt *Minimallösung*, falls für jede Lösung $\{p_k\}$ zu den von p_0, p_1 linear unabhängigen Startwerten q_0, q_1 gilt, dass:

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

Die Lösung $\{p_k\}$ wird dominante Lösung genannt.

9.6 Beispiel. In Beispiel 9.3 ist $\{p_k\}$ Minimallösung genau dann, wenn $\beta = 0$.

Die Minimallösung ist nur bis auf einen skalaren Faktor eindeutig. Deshalb normieren wir sie mit der zusätzlichen Bedingung: $p_0^2 + p_1^2 = 1$.

Frage: Wie können wir eine normierte Minimallösung berechnen, obwohl das Problem schlecht konditioniert ist?

10. Miller-Algorithmus

Betrachte die Rückwärtsrekursion zu

$$q_k = a_k q_{k-1} + b_k q_{k-2}, k = 2, 3, \ldots$$

d.h.

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

mit Startwerten $q_n = 0, q_{n-1} = 1$. Für $a_k = a, b_k = b$ besitzt das charakteristische Polynom

$$q(\mu) = \mu^2 + \frac{a}{b}\mu - \frac{1}{b}$$

Die Nullstellen:

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

Idee: Wende Satz 9.4 "rückwärts" an.
Lösen wir

$$\alpha + \beta = 0$$

und

$$\alpha\mu_1 + \beta\mu_2 = q_{n-1} = 1$$

ergibt sich:

$$\alpha = \frac{\lambda_1\lambda_2}{\lambda_2 - \lambda_1} \neq 0$$

$$\beta = -\frac{\lambda_1\lambda_2}{\lambda_2 - \lambda_1} \neq 0$$

und

$$q_k = \alpha\mu_1^{n-k} + \beta\mu_2^{n-k} = \frac{\alpha}{\lambda_1^{n-k}} + \frac{\beta}{\lambda_2^{n-k}}$$

Für $|\lambda_1| < |\lambda_2|$ folgt, dass

$$\begin{aligned} p_k^{(n)} &:= \frac{q_k}{q_0} \\ &= \frac{\frac{\alpha}{\lambda_1^{n-k}} + \frac{\beta}{\lambda_2^{n-k}}}{\frac{\alpha}{\lambda_1^n} + \frac{\beta}{\lambda_2^n}} \\ &= \frac{\lambda_1^k + \frac{\beta}{\alpha}\lambda_2^k \left(\frac{\lambda_1}{\lambda_2}\right)^n}{1 + \frac{\beta}{\alpha}\left(\frac{\lambda_1}{\lambda_2}\right)^n} \\ &\to \lambda_1^k = p_k \end{aligned}$$

Beobachtung: Für großes n approximiert $p_k^{(n)}$ die Minimallösung.

Daten: n genügend groß $\{a_k\}_{k=2}^n, \{b_k\}_{k=2}^n$
Ergebnis: Approximation von $\{p_k^{(n)}\}_{k=0}^n$ um eine normierte Minimallösung
Setze $\hat{p_n} = 0, \hat{p_{n-1}} = 1$
für $k \leftarrow n$ **bis** 2 **tue**
  $\hat{p_{k-2}} = -\frac{a_k}{b_k}\hat{p_{k-1}} + \frac{1}{b_k}\hat{p_k}$
Ende
für $k \leftarrow 0$ **bis** n **tue**
  $p_k^{(n)} = \frac{\hat{p_n}}{\sqrt{\hat{p_0^2}+\hat{p_n^2}}}$
Ende

Algorithmus 6: Miller-Algorithmus

10.7 Satz (Miller-Algorithmus). Sei $\{p_k\}_{k=0}^\infty$ Minimallösung der normierten, homogenen Dreitermrekursion. Dann gilt für die Lösung des Miller-Algorithmus:

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

Beweis. Wir bemerken, dass sich jede Lösung der Dreitermrekursion als Linearkombination einer Minimallösung und einer dominanten Lösung schreiben lässt (Übung). Mit den richtigen α und β folgt (Übung):

$$\begin{aligned}\hat{p_k} &= \alpha p_k + \beta q_k \\ &= \frac{p_k q_n - q_k p_n}{p_{n-1}q_n - q_{n-1}p_n} \qquad = \frac{q_n}{p_{n-1}q_n - q_{n-1}p_n}\left(p_k - \frac{p_n}{q_n}q_k\right)\end{aligned}$$

Aus (Normierungs-Konstante)

$$\begin{aligned}\hat{p_0^2} + \hat{p_1^2} &= \frac{q_n^2}{(p_{n-1}q_n - q_{n-1}p_n)^2}\left(p_0^2 + p_1^2 - 2\frac{p_n}{q_n}(p_0q_0 + p_1q_1) + \frac{p_n^2}{q_n^2}(q_0^2 + q_1^2)\right) \\ &= \frac{q_n^2}{(p_{n-1}q_n - q_{n-1}p_n)^2}\left(1 - 2\frac{p_n}{q_n} + \frac{p_n^2}{q_n^2}\right)\end{aligned}$$

folgt:

$$\begin{aligned}p_k^{(n)} &= \frac{\hat{p_k}}{\sqrt{\hat{p_0^2} + \hat{p_1^2}}} \\ &= \frac{p_k - \frac{p_n}{q_n}q_k}{\sqrt{1 - 2\frac{p_n}{q_n} + \frac{p_n^2}{q_n^2}}} \qquad \to p_k\end{aligned}$$

□

Wir können also auch für schlecht konditionierte Probleme stabile Algorithmen finden.

Sortieren

11. Das Sortierproblem

Gegeben $n \in \mathbb{N}$ verschiedene Zahlen $z_1, \ldots, z_n \in \mathbb{R}$.

Gesucht Permutation $\pi_1, \ldots, \pi_n$, so dass $z_{\pi_1} < \ldots < z_{\pi_n}$

11.1 **Definition** (Permutation). Eine Permutation π von $\{1, 2, \ldots, n\}$ ist eine bijektive Abbildung von $\{1, 2, \ldots, n\}$ auf sich selbst. Wir schreiben $\pi(k) = \pi_k$ für $k = 1, \ldots, n$

11.2 **Bemerkung.** Da wir die Zahlen $z_1, \ldots, z_n$als verschieden annehmen ist das Sortierproblem eindeutig lösbar.

Probiere so lange alle möglichen Permutationen durch, bis die gewünschte Sortierung vorliegt

Algorithmus 7: Brute-Force

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

Beweis. Wir haben

- n Möglichkeiten die erste Zahl auszuwählen
- $n-1$ Möglichkeiten die zweite Zahl auszuwählen
- …
- 1 Möglichkeit die letzte Zahl auszuwählen

woraus die Behauptung folgt. □

Im schlimmsten Fall (worst case) muss der Algorithmus 7 also

$$(n-1) \cdot n!$$

Vergleiche durchführen. Da dies extrem aufwändig sein kann, betrachten wir im Folgenden einem Algorithmus, der die transitive Struktur

$$x < y \text{ und } y < z \implies x < z$$

der Ordnungsrelation ausnutzt.

Daten: Menge $S_n = \{z_1, \ldots, z_n\}$
Ergebnis: Sortierte Menge $S^\pi = \{z_{\pi_1}, \ldots, z_{\pi_n}\}$
für $k \leftarrow n$ **bis** 1 **tue**
  $z_{\pi_k} = \max(S_k)$
  $S_{k-1} = S_k \setminus \{z_{\pi_k}\}$
Ende

Algorithmus 8: Bubblesort

11.4 Beispiel. Die zu sortierende Menge ist: $\{4, 1, 2\}$.

- $S_3 = \{4, 1, 2\}$, $k = 3$, $z_{\pi_3} = 4$
- $S_2 = \{1, 2\}$, $k = 2$, $z_{\pi_2} = 2$
- $S_1 = \{1\}$, $k = 1$, $z_{\pi_1} = 1$

Die sortierte Liste ist: $\{1, 2, 4\}$.

Um den Aufwand von Algorithmen zu untersuchen, beschränken wir uns auf das asymptotische Verhalten für große n.

11.5 Definition (Landau-Notation). Wir schreiben

- $f(x) = \mathcal{O}(g(x))$, falls Zahlen $C > 0, x_0 > 0$ existieren, so dass: $|f(x)| \leq Cg(x), \forall_{x>x_0}$.
- $f(x) = \Omega(g(x))$, falls Zahlen $C > 0, x_0 > 0$ existieren, so dass $|f(x)| \geq Cg(x) \forall_{x>x_0}$.
- $f(x) = o(g(x))$, falls für jedes $c > 0$ ein $x_0 > 0$ existiert, so dass $|f(x)| \leq cg(x), \forall_{x>x_0}$.
- $f(x) = \Theta(g(x))$, falls $f(x) = \mathcal{O}(g(x))$, $g(x) = \mathcal{O}(f(x))$

11.6 Bemerkung. Genau dann wenn Beziehung der Landau-Notation

- $f(x) = \mathcal{O}(g(x)) \iff \limsup_{x\to\infty} |\frac{f(x)}{g(x)}| \leq C < \infty$
- $f(x) = \Omega(g(x)) \iff \liminf_{x\to\infty} |\frac{f(x)}{g(x)}| > 0$
- $f(x) = o(g(x)) \iff \lim_{x\to\infty} |\frac{f(x)}{g(x)}| = 0$
- $f(x) = \Theta(g(x)) \iff f(x) = \mathcal{O}(g(x)), f(x) = \Omega(g(x))$

11.7 Beispiel. Es gilt $\sin(x) = \mathcal{O}(1)$, da $|\sin(x)| \leq 1$ für alle $x \in \mathbb{R}$. Bei Polynomen gilt die Laufzeit ist die höchste Potenz, sofern $x \geq 1$.

11.8 Definition. Der Aufwand eines Algorithmus ist die kleinste obere Schranke für das betrachtete Aufwandsmaß

Für uns ist der Speicherbedarf irrelevant und benutzen als Aufwandsmaß für den Rechen die Anzahl der benötigten Vergleiche.
In der k-ten Iteration des Bubblesort-Algortihmus 8 müssen k Vergleiche ausgeführt werden. Das heißt, der Gesamtaufwand des Algorithmus ist:

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

Dies ist eine drastische Verbesserung im Vergleich zu $\mathcal{O}(n(n!))$ von Algorithmus 7

12. Mergesort

Zu erst wollen wir folgendes beobachten:

12.9 Lemma. Gegeben seien zwei sortiere Mengen

- $S_x = \{x_1 < \ldots < x_m\}$
- $S_y = \{y_1 < - \ldots < y_n\}$

Dann lässt sich die Menge $S = S_x \cup S_y$ mit linearem Aufwand sortieren. Genauer werden $m + n - 1$ Vergleiche benötigt.

Beweis. Konstruktiv, durch den entsprechenden Algorithmus. □

Daten: Sortierte Mengen S_x und S_y
Ergebnis: Sortierte Menge $S = S_x \cup S_y$
Initialisiere $i = j = k = 1$
solange $i \leq m$ *und* $j \leq n$ **tue**
  wenn $x_i < y_j$ **dann**
    $z_k = x_i$
    $i = i + 1$
  Ende
  sonst
    $z_k = y_j$
    $j = j + 1$
  Ende
  $k = k + 1$
Ende
für $l \leftarrow 0$ **bis** $m - i$ **tue**
  $z_{k+l} = x_{k+l}$
Ende
für $l \leftarrow 0$ **bis** $n - j$ **tue**
  $z_{k+l} = y_{k+l}$
Ende

Algorithmus 9: Merge

Basierend auf dieser Beobachtung können wir eine divide-and-conquer Strategie angeben um Mengen der Länge $n = 2^m, m \in \mathbb{N}$ zu sortieren.

Daten: Menge $S = \{z_1, \ldots, z_n\}$
Ergebnis: Sortierte Menge $S^\pi = \{z_{\pi_1} < \ldots < z_{\pi_n}\}$
wenn $n = 1$ **dann**
| $S^\pi = S$
Ende
sonst

- Sortiere
  $L = \{z_1, \ldots, z_{\frac{n}{2}}\}$
  $R = \{z_{\frac{n}{2}+1}, \ldots, z_n\}$ mittels Mergesort zu L^π und R^π
- Sortiere $L^\pi \cup R^\pi$ mittels Merge-Algorithmus 9 zu S^π

Ende

Algorithmus 10: Mergesort

12.10 Beispiel. Mergesort der Menge $\{20, 7, 84, 31, 71, 42, 18, 10\}$

12.11 Bemerkung. Da Mergesort sich selbst aufruft, sprechen wir von einem *rekursiven Algorithmus*. Im Allgemeinem ist es schwierig zu beurteilen, ob solche Algorithmen terminieren. Im Fall von Mergesort ist der Fall jedoch klar, da die Rekursion im Fall $n = 1$ abgebrochen wird.

12.12 Satz. Der Aufwand von Mergesort ist $\mathcal{O}(n \log n)$

Beweis. Bezeichne A(n) den Aufwand für das Sortieren einer $n = 2^m$ elementigen Menge mittels Mergesort. Dann gilt:

$$\begin{aligned} A(1) &= 0 \\ A(n) &= n - 1 + 2A(\frac{n}{2}) \end{aligned}$$

Auflösen der Rekursion ergibt:

$$\begin{aligned} A(n) &= n - 1 + 2A(\frac{n}{2}) \\ &= 2n - 1 - 2 + 4A(\frac{n}{4}) \\ &= \ldots \\ &= mn - \sum_{i=0}^{m-1} 2^i \\ &= mn - \frac{1 - 2^m}{1 - 2} \\ &= (m-1)n + 1 \end{aligned}$$

$m = \log_2(n) = \frac{\log(n)}{\log(2)}$ impliziert die Behauptung □

$\log n \ll n$, also ist die Verbesserung von $\mathcal{O}(n^2)$ nach $\mathcal{O}(n \log(n)$ signifikant. Man nennt das Wachstum auch beinahe linear.

12.13 Bemerkung. Die Implementierung von Mergesort als in-place-Algorithmus ist je nach Datenstrukturen trickreich. Deshalb wird oft nicht in-place implementiert, weshalb für jeden "divide"-Schritt zusätzlicher Speicherplatz implementiert werden muss.

13. Quicksort

Idee: Divide-and-Conquer Strategie basierend auf dem Inhalt der zu sortierenden Liste.

Daten: Menge $S = \{z_1, \ldots, z_n\}$
Ergebnis: Sortierte Menge $S^\pi = \{z_{\pi_1}, \ldots, z_{\pi_n}\}$
Wähle ein Pivot-Element $x \in S$
Bestimme eine Permutation π, so dass $x = z_{m_\pi}$
wenn $L = \{z_{\pi_1}, \ldots, z_{\pi_{m-1}} \neq \emptyset\}$ **dann**
| Sortiere L zu L^π mittels Quicksort
Ende
wenn $R = \{z_{\pi_{m+1}}, \ldots, z_{\pi_n}\} \neq \emptyset$ **dann**
| Sortiere R zu R^π mittels Quicksort
Ende
Vereinige $S^\pi = R^\pi \cup \{x\} \cup L^\pi$

Algorithmus 11: Quicksort

13.14 Beispiel. Beispiel anhand der gleichen Menge von Mergesort.

13.15 Lemma. Im schlimmsten Fall ist der Aufwand von Quicksort $\mathcal{O}(n^2)$

Beweis. $A(n)$ aus 12.12 wird umso größer, je unterschiedlicher die Größe der beiden Teilprobleme $A(m-1)$ und $A(n-m)$ ist. $A(n)$ ist also maximal für $m = 1$ oder $m = n$, also wenn das Pivot-Element das größte oder kleinste Element ist. Dann gilt:

$$A(n) = n - 1 + A(n-1)$$

Der Rest erfolgt Analog zu der Aufwandserklärung von Bubblesort 8:

$$A(n) = \frac{n(n-1)}{2} = \mathcal{O}(n^2)$$

Damit ist der Aufwand gezeigt. □

13.16 Satz. Alle Permutationen der Zahlen $\{1, 2, \ldots, n\}$ seien gleich wahrscheinlich. Dann benötigt Quicksort im Durchschnitt $\mathcal{O}(n \log(n)$ Vergleiche zum sortieren von Zahlen.

Beweis. Sie Π die Menge aller Permutationen von $\{1, 2, \ldots, n\}$ und sei $A(\pi), \pi \in \Pi$ die Anzahl Vergleiche um eine Permutation π mittels Quicksort zu sortieren. Der durchschnittliche Aufwand ist:

$$\overline{A}(n) = \frac{1}{n!} \sum_{\pi \in \Pi} A(\pi)$$

O.B.d.A: Sei das erste Element das Pivotelement. Definiere:

$$\Pi_k = \{\pi \in \Pi | \pi_1 = k\}, k = 1, \ldots, n$$

mit

$$|\Pi_k| = (n-1)!, k = 1, \ldots, n$$

Für k fix und $\pi \in \Pi_k$ teilt Quicksort im ersten Aufruf in zwei Mengen

$$\pi_< = \{\text{Permutationen von } 1, 2, 3 \ldots, k-1\}$$
$$\pi_> = \{\text{Permutationen von } k+1, \ldots, n\}$$

Analog zu 12.12 folgt

$$A(\pi) = n - 1 + A(\pi_<) + A(\pi_>)$$

und

$$\sum_{\pi \in \pi_k} A(\pi) = (n-1)!(n-1) + \sum_{\pi \in \Pi_k} A(\pi_<) + \sum_{\pi \in \Pi_k} A(\pi_>) \tag{13.1}$$

Wenn π alle Permutationen aus Π_k durchläuft, entstehen für $\Pi_<$ alle Permutationen von $\{1, 2, \ldots, k-1\}$. Dabei tritt jede Permutation genau $\frac{(n-1)!}{(k-1)!} = \frac{|\Pi_k|}{\Pi_<}$ mal auf.

$$\Longrightarrow \sum_{\pi \in \Pi_k} A(\pi_<) = \frac{(n-1)!}{(k-1)!} \sum_{\pi \in \Pi_k} A(\pi_<) = (n-1)!\overline{A}(k-1) \tag{13.2}$$

Analog:

$$\sum_{\pi \in \Pi_k} A(\pi_>) = (n-1)!\overline{A}(n-k) \tag{13.3}$$

Zusammensetzen:

$$\begin{aligned} \overline{A}(n) &= \frac{1}{n!} \sum_{\pi \in \Pi} A(\pi) \\ &= \frac{1}{n!} \sum_{k=1}^{n} \sum_{\pi \in \Pi_k} \end{aligned}$$

Unter Verwendung der Gleichungen 13.1, 13.2, 13.3 folgt weiter:

$$\begin{aligned}
&= \frac{1}{n!}\sum_{k=1}^{n}(n-1)!(n-1+\overline{A}(k-1)+\overline{A}(n-k) \\
&= n-1+\frac{1}{n}\sum_{k=1}^{n}\overline{A}(k-1)+\overline{A}(n-k) \\
&= n-1+\frac{2}{n}\sum_{k=0}^{n-1}\overline{A}(k)
\end{aligned}$$

mit Startwerten $\overline{A}(0) = \overline{A}(1) = 0$.
Per Induktion folgt:

$$\begin{aligned}
\overline{A}(n) &= 2(n+1)\sum_{i=1}^{n}\frac{1}{i} - 4n \\
&\leq 2(n+1)(1+\int_{1}^{n}\frac{1}{x}dx - 4n \\
&= 2(n+1)(1+\log(n)) - 4n \\
&= 2(n+1)\log(n) - 2(n-1) \\
&= \mathcal{O}(n\log(n))
\end{aligned}$$

□

14. Untere Schranke für das Sortierproblem

Frage: Gibt es einen Sortieralgorithmus, der schneller ist als $\mathcal{O}(n\log(n))$?

14.17 Satz. Jedes deterministische Sortierverfahren, das auf paarweisen Vergleichen basiert und keine Vorkenntnisse über die zu sortierende Menge hat, benötigt im schlimmsten Fall mindestens $\log_2(n!)$ Vergleiche zum Sortieren von n verschiedenen Zahlen.

Bevor wir den Beweis machen gibt es noch einige Hinweise, die wir für den Beweis benötigen.

14.18 Bemerkung. Es gilt:

- $\log_2(n!) = \Theta(n\log(n)$
- $\log_2(n) \leq \log_2(n^n) = n\log_2(n) = \frac{1}{\log(2)}n\log n$
- $\log(n!) \geq \log(\frac{n}{2}\ldots\frac{n}{2}) \geq \log((\frac{n}{2})^{\frac{n}{2}} = \frac{n}{2}\log(\frac{n}{2}) = \frac{n}{2}\log(n) - n\log(2)$

Zum Beweis benötigen wir einen Entscheidungsbaum. Dieser kann jedem Sortieralgorithmus mit paarweisen Vergleichen zugeordnet werden, und illustriert das Verhalten des Algorithmus:

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

14.19 Beispiel. Baum für $\{z_1, z_2, z_3\}$:

Beweis. 14.17
Idee: Finde eine untere Schranke für die Tiefe des Baumes. Dies ist die Mindestanzahl der im schlimmsten Fall mötigen Vergleiche. Im besten Fall ist ein Baum ausbalanciert, d.h. alle Blätter haben die gleiche Tiefe m und wir haben 2^m Blätter. Da wir $n!$ Permutationen haben (also Blätter), muss $2^m = n!$ gelten. Es folgt $m \log_2(n!)$ □

Satz 14.17 sagt, dass Mergesort in der asymptotischen Laufzeit optimal ist.

Frage: Können wir etwas Ähnliches für Quicksort zeigen?

14.20 Satz. Alle Permutationen der Zahlen $z_1, \ldots, z_n$ seien gleich wahrscheinlich. Dann benötigen Sortierverfahren wie in Satz 14.17 im Mittel $\log_2(n!)$ Vergleiche.

Beweis. Die mittlere Höhe $\overline{H}(z)$ des Entscheidungsbaums τ ist gegeben als der Mittelwert der Tiefen t_τ aller seiner Blätter v. Sei $B(\tau)$ die Menge der Blätter und $\beta(\tau) = |B(\tau)|$ Dann gilt:

$$\overline{H}(\overline{c}) = \frac{1}{\beta(\overline{c})} \sum_{y=B(\overline{c}} t\overline{c}(v)$$

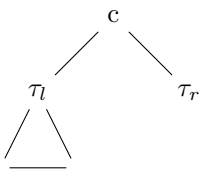
Zu zeigen ist, dass:

$$\overline{H}(\overline{c}) \geq \log_2(\beta(\overline{c}))$$

Vollständige Induktion über die Höhe des Entscheidungsbaums $\overline{c}$.

◊ Induktionsverankerung:$H(\overline{c}) = 0$
$\overline{c}$ besteht nur aus seiner Wurzel. $\implies \overline{H}(c) = \frac{1}{1} \sum_{y \in B(\tau)} t_\tau(v) = 0 = \log_2(1)$

◊ Induktionsschritt: $H(\overline{c}) \leq h - 1 \implies H(\overline{c}) = h$
Seien τ_l, τ_r die Wurzeln des linken bzw. rechten Teilbaums zum Wurzelknoten von c:



Beobachte:

- $H(\tau_l), H(\tau_r) < h$
- $B(\tau) = B(\tau_l) \cup B(\tau_r)$
- $B(\tau_l) \cap B(\tau_r) = \emptyset$
- $\beta(\overline{c}) = \beta(\tau_l) + \beta(\tau_r)$
- $\beta(\tau_l), \beta(\tau_r) \geq 1$
- $t_\tau(v) = t_{\tau_l}(v) + 1$
- $t_\tau(v) = t_{\tau_r}(v) + 1$

Rechne:

$$\begin{aligned}\overline{H}(\tau) &= \frac{1}{\beta(\tau)} \sum_{v \in B(\tau)} t_\tau(v) \\ &= \dots \\ &\geq 1 + \frac{1}{b} \min_{x \in [0,b]} (x \log_2(x) + (b-x) \log_2(b-x))\end{aligned}$$

Da

$$\begin{aligned}f'(x) &= \log_2(x) - \log_2(b-x) \\ &= \log_2(\frac{x}{b-x}) \qquad\qquad = 0\end{aligned}$$

genau dann, wenn $x = \frac{b}{2}$, ist dies die einzige Möglichkeit, wo das Minimum in $(0, b)$ angenommen werden kann. Die anderen Möglichkeiten sind die Intervallrandpunkte, was ausgeschlossen ist. Also gilt:

$$\begin{aligned}\overline{H}(\tau) &\geq 1 + \log_2(\frac{b}{2}) \\ &= 1 + \log_2(\beta(\tau)) - 1 \\ &= \log_2(\beta(\tau))\end{aligned}$$

□

Graphen

15. Grundlagen

15.1 **Definition** (Graph). Ein Graph ist ein Paar $G = (V, E)$, bestehend aus endlichen Mengen V von Knoten (verticles) und E von Kanten (edges). Die Kanten beschreiben die Verbindungen zwischen den Knoten und sind gerichtet oder ungerichtet.

- Für *gerichtete* Graphen oder *Digraphen* gilt:

$$E \subset \{(v, w) \in V \times V | v \neq w\}$$

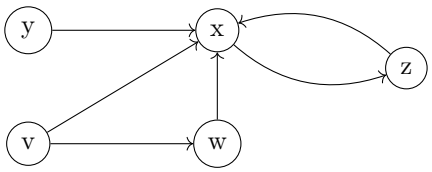
  Ist $e = (v, w) \in E$, so heißt v der Anfangsknoten und w der Endknoten.

- Für *ungerichtete* Graphen ist E eine Menge von ungeordneten Paaren $\{v, w\} \subset E, v \neq w$, das heißt:

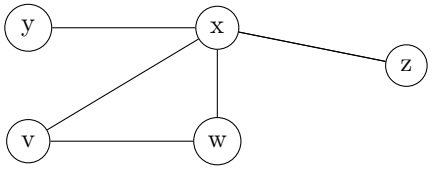
$$E \subset \{X \subset V || X| = 2\}$$

15.2 **Beispiel.** Sei $V = \{v, w, x, y, z\}$

- $G = (V, E), E = \{(v, w), (v, x), (w, x), (x, z), (z, x), (y, x)\}$ ist ein Digraph, der wie folgt illustriert werden kann:



- Der gleiche Graph sieht ungerichtet wie folgt aus:



  Wir bemerken insbesondere, dass $\{v, w\} = \{w, v\}$ im ungerichteten Graphen ist.

15.3 **Bemerkung.** In Digraphen werden manchmal auch Kanten der Form $\{v, v\}, v \in V$ zugelassen.

15.4 Definition. Sei $G = (V, E)$ ein Graph. Ein *Weg* oder *Pfad* von $v = v_0$ nach $w = v_r$ in G ist eine Kantenfolge

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

mit $r \geq 1$ und

- $(v_i, v_{i+1}) \in E, i = 0, \ldots, r-1$im Falle von Digraphen.
- $\{v_i, v_{i+1}\} \in E, i = 0, \ldots, r-1$ im Falle von ungerichteten Graphen.

Der Weg heißt *einfach*, falls

- $v_0, \ldots, v_r$ paarweise verschieden sind oder
- $v_0, \ldots, v_r$ paarweise verschieden mit $v_0 = v_r$.

Die *Länge* von π ist gegeben als $|\pi| = r$, ist also die Anzahl Knoten, die in π durchlaufen werden. Ein Knoten w heißt von Knoten v *erreichbar*, falls ein Weg von v nach w existiert.

15.5 Beispiel. Bezogen auf Beispiel 15.2 ist:

- $\pi_1 = v, w$ ein einfacher Weg der Länge 1
- $\pi_2 = v, w, x$ ein einfacher Weg der Länge 2
- $\pi_3 = v, w, x, z, x, z$ ein Weg der Länge 5

Im ungerichteten Fall ist

- $\pi_4 = v, w, x, v$ ein einfacher Weg der Länge 3.

In beiden Fällen ist z von v erreichbar, im ungerichteten Fall gilt dies auch umgekehrt. Im gerichteten Fall ist v nicht von z erreichbar.

15.6 Definition. Sei $G = (V, E)$ ein Digraph und $v \in V$. Wir definieren:

- die Menge der (direkten) *Nachfahren* von v:

$$post(v) = \{w \in V | (v, w) \in E\}$$

- die Menge der (direkten) *Vorfahren* von v:

$$pre(v) = \{w \in V | v \in post(w)\}$$

- die Menge der *erreichbaren Knoten* von v:

$$post^*(v) = \{w \in V | \text{ es gib einen Weg von v nach w}\}$$

- die Menge aller Knoten die v erreichen können:

$$pre^*(v) = \{w \in V | v \in post^*(w)\}$$

- die Nachbarn als die Menge $post(v) \cup pre(v)$ von v
- und den Knotengrad $deg(v)$, welcher der Anzahl seiner Kanten entspricht.

15.7 Bemerkung. Mit den offensichtlichen Modifikationen kann die Definition 15.6 auch auf ungerichtete Graphen angewendet werden. Wir erhalten in diesem Fall für $v \in V$, dass

$$post(v) = pre(v)$$
$$post^*(v) = pre^*(v)$$

15.8 Beispiel. Für den Digraphen aus Beispiel 15.2 gilt:

- $post(v) = \{w, x\}$
- $post^*(v) = \{w, x, z\}$
- $pre(v) = \emptyset$
- $pre^*(v) = \emptyset$
- $deg(v) = 2$

Für den ungerichteten Graphen gilt geändert:

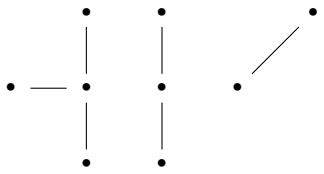
- $pre(v) = \{w, x\}$
- $pre^*(v) = \{w, x, y, z\}$
- $post^* = \{w, x, y, z\}$

16. Zusammenhang

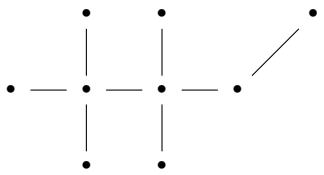
16.9 Definition. Sei $G = (V, E)$ ein ungerichteter Graph $\emptyset \neq C \subset V$

- Die Menge C heißt *zusammenhängend*, falls je zwei Knoten $v, w \in C, v \neq w$, voneinander erreichbar sind, das heißt, $v \in post^*(w), w \in post^*(v)$.
  Ist G ein Digraph, so heißt C zusammenhängend, falls C im zugrundeliegenden ungerichteten Graphen zusammenhängend ist.
- Eine zusammenhängende Knotenmenge heißt *Zusammenhangskomponente*, falls sie *maximal* ist. Das bedeutet, es gibt keine weitere zusammenhängende Knotenmenge $D \subset V$ mit $C \subsetneq V$.
- Ein Graph heißt zusammenhängend, falls V zusammenhängend ist.

16.10 Beispiel. Dies ist ein unzusammenhängender Graph mit drei Zusammenhangskomponenten:



Analog hierzu ist der folgende Graph zusammenhängend:



Wir können beobachten, dass die Zusammenhangskomponenten eines ungerichteten Graphen die Äquivalenzklassen der Knotenmenge V unter der Äquivalenzrelation

$$v = w \iff \{v\} \cup post^*(v) = \{w\} post^*(w)$$

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

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

wobei $E_i = E \cap \{X \subset C_i : |X| = 2\}$.

16.11 Satz. Sei $G = (V, E)$ ein ungerichteter Graph mit $n = |V| \geq 1$ Knoten und $m = |E|$ Kanten. Ist G zusammenhängend, so folgt für die Anzahl Knoten und Kanten, dass

$$m \geq n - 1$$

Beweis. □

Per vollständiger Induktion:

◊ IV $\underline{n=1}$
$m = 0 = n - 1$

◊ IS $\underline{n \geq 3}$
Wähle $v \in V$ mit:

$$deg(v) = \min_{v \in V} deg(v) =: k$$

Da G zusammenhängend ist, muss $k > 0$ gelten.
$\underline{k \geq 2}$

$$\begin{aligned} 2m &= 2|E| \\ &= \sum_{w \in V} |degw \\ &\geq 2|V| \\ &= 2n \qquad\qquad \implies m \geq n \geq n-1 \end{aligned}$$

$\underline{k = 1}$
Sei $G'(V', E')$ derjenige Graph, der durch das Streichen von v und der zugehörigen Kante entsteht. G' ist zusammenhängend, weil G zusammenhängend ist. Die Induktionsannahme impliziert:

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

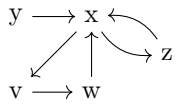
17. Zyklische Graphen

17.12 Definition. Ein *einfacher Zyklus* oder ein *einfacher Kreis* in einem Graphen $G = (V, E)$ ist ein einfacher Weg $\pi = v_0, \ldots, v_r$ mit $v_0 = v_r$ und $r \geq 2$ (im Falle von Digraphen) oder $r \geq 3$ (im Falle von ungerichteten Graphen). Ein *Zyklus* oder *Kreis* ist ein Weg, ‚der sich aus einfachen Zyklen zusammensetzt.

17.13 Bemerkung. Aus der Definition 17.12 folgt:

- Jeder einfache Zyklus ist ein Zyklus
- Sind $\pi_1 = v_0, \ldots, v_r$, $\pi_2 = w_0, \ldots, w_s$ mit $v_i = w_0 = w_r$ so ist auch $\pi = v_0, \ldots, v_{i-1}, w_0, \ldots, w_s, v_{i+1}, \ldots, v_r$ ein Zyklus
- Es gibt keine andere Möglichkeit um Zyklen zu generieren.

17.14 Beispiel. Der Digraph



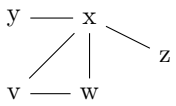
besitzt die einfachen Zyklen:

$$\pi_1 = x, z, x$$
$$\pi_2 = v, w, x, w$$

und die nicht einfachen Zyklen:

$$\pi_3 = x, z, x, z, x$$
$$\pi_4 = v, w, x, z, x, v$$

Der dazugehörige ungerichtete Graph:



besitzt den einfachen Zyklus

$$\pi_1 = v, w, x, v$$

Die Wege

$$\pi_2 = x, y, x$$
$$\pi_3 = v, w, x, y, x, v$$

sind keine Zyklen.

17.1. Eulergraphen

Im Folgenden betrachten wir exemplarisch das Königsberger Brückenproblem.

17.15 Das Königsbergerbrückenproblem ist ein von Leonhard Euler gelöstes mathematisches Problem. Es handelt konkret um die Stadt Königsberg und um die Frage, ob es einen Rundweg gibt, bei dem man alle sieben Brücken der Stadt über den Pregel exakt einmal überqueren kann und wieder zum Ausgangspunkt gelangt.
Euler zeigte, dass es keinen solchen Rundweg gibt.

17.16 **Definition.** Ein *Eulerscher Rundweg* in einem Graphen $G = (V, E)$ ist ein Zyklus, der jede Kante $e \in E$ genau einmal enthält. Im ungerichteten Fall nennen wir G *Eulersch*, falls der Grad jedes Knotens gerade ist.
Ein Digraph ist Eulersche, falls $indeg(v) = outdeg(v), v \in V$.

17.17 **Satz.** Ein zusammenhängender Graph $G = (V, E)$ besitzt genau dann einen Eulerschen Rundweg, falls der Graph Eulersch ist.

Beweis. Hin- und Rückrichtung

- $\underline{\Longrightarrow}$ Ein Knoten $v \in V$ in einem Eulerschen Graph, der k-mal in einem Eulerschen Rundweg vorkommt, muss im gerichteten Fall

  $$indeg(v) = outdeg(v)$$

  und im ungerichteten Fall

  $$deg(v) = 2k$$

  erfüllen.

- $\underline{\Longleftarrow}$ Sei G Eulersch und $\pi = v_0, \ldots, v_r$ der längste Weg, in dem jede Kante aus E höchstens einmal vorkommt. Dies bedeutet, dass jede ausgehende Kante von v_r im Weg enthalten sein muss (sonst wäre es nicht der längste Weg). Die Bedingung an den Knotengrad impliziert: $v_0 = v_r$.

Annahme:

- Digraphen: Es gibt eine Kante $e = (w_1, w_2) \in E$ mit $e \neq (v_i, v_{i+1}, i = 0, \ldots, r-1$
- ungerichtete Graphen: Es gibt eine Kante $e = \{w_1, w_2\} \in E$ mit $e \neq \{v_i, v_{i+1}\}, i = 0, \ldots, r-1$.

Fall 1: w_1 oder w_2 sind in π enthalten.
Ohne Beschränkung der Allgemeinheit: $v_r = w_1$

$$\Longrightarrow \tilde{\pi} = v_0, \ldots, v_r, w_2$$

Dies ist ein Widerspruch zur Annahme, da π nun nicht mehr der längste Weg ist.
Fall 2: w_1 und w_2 sind beide nicht in π enthalten. Da G zusammenhängend ist, gibt es einen Weg von v_0 nach w_1. Entlang dieses Weges muss es eine Kante geben, die einen Knoten in π und einen nicht in π hat.
Analog zum ersten Fall führt dies zum Widerspruch.

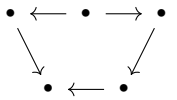
□

Wir beachten: Satz 17.17 ist die Antwort auf das Königsberger Brückenproblem. Demnach gibt es keinen solchen Weg und es ist nicht mögliche einen Euler-Weg zu finden.

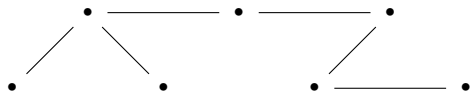
18. Bäume

18.18 **Definition.** Ein Graph G heißt *azyklisch* oder *zyklenfrei*, falls es keine Zyklen in G gibt. Ein ungerichteter, azyklischer und zusammenhängender Graph ist ein *Baum*

18.19 **Beispiel.** Ein azyklischer Graph kann wie folgt aussehen:



Sollte dieser ungerichtet sein könnte er so aussehen:



Dieser Graph ist sogar zusammenhängend.

18.20 **Satz.** Sei $G = (V, E)$ ein ungerichteter Graph mit $n = |V|$ knoten. Dann sind die folgenden Aussagen äquivalent.

a) G ist ein Baum

b) G hat n-1 Kanten und ist zusammenhängend

c) G hat n-1 Kanten und ist azyklisch

d) G ist azyklisch und das Hinzufügen einer beliebigen, noch nicht vorhandenen Kante erzeugt einen Zyklus.

e) G ist zusammenhängend und das Entfernen einer beliebigen Kante macht G unzusammenhängend.

f) Jedes Paar verschiedener Knoten in G ist durch genau einen einfachen Weg miteinander verbunden.

Beweis. Wir zeigen nicht jede Äquivalenz einzeln, sondern folgern aus anderen Aussagen.

- $\underline{a \implies f}$ Falls es für ein gegebenes Paar $v, w \in V$, verschiedene Wege gäbe, so würde die Vereinigung dieser beiden Wege einen Zyklus beinhalten (Widerspruch zu azyklisch in Baum).
- $\underline{f \implies e}$ "zusammenhängend" folgt per Definition. Falls wir eine beliebige Kante entfernen, so sind die beiden Endpunkte nicht mehr voneinander erreichbar ($\implies$ G wird unzusammenhängend).
- $\underline{e \implies d}$ G ist azyklisch, da wir sonst eine (ausgewählte) Kante entfernen könnten, so dass g immer noch zusammenhängend wäre. Seien $v, w \in V, w \neq v$, dann gibt es einen Weg von v nach w. Hinzufügen der Kante $\{v, w\}$ macht diesen Weg zum Zyklus.
- $\underline{d \implies c}$ Wir zeigen per Induktion über $m = |E|$, dass für einen azyklischen, ungerichteten Graph
$$n = m + p \tag{18.1}$$
gilt, wobei p die Anzahl Zusammenhangskomponenten ist.
  ◊ IV $\underline{m = 0}$ trivial

  ◊ IS $\underline{m \to m + 1}$ Beim Hinzufügen einer Kante muss eine Zusammenhangskomponente verschwinden, da sonst ein Zyklus entstehen würde.
  Da das Hinzufügen einer beliebigen neuen Kante einen Zyklus entstehen lässt, muss $p = 1$ gelten.
$$\implies |E| = m = n - 1$$
- $\underline{c \implies b}$ Folgt aus (18.1)
- $\underline{b \implies a}$ Zu zeigen: G ist azyklisch.
  Angenommen G hat k verschiedene einfache Zyklen. Dann können wir G durch entfernen von k Kanten zu einem azyklischen Graphen machen. (18.1) ist andwendbar, un impliziert, dass
$$\begin{aligned} n &= (m - k) + 1 \\ m + 1 &= (m - k) + 1 \end{aligned} \qquad \implies k = 0$$
  d.h. G keine Zyklen.

□

18.21 **Bemerkung.** Fixieren wir in einem Baum $G = (V, E)$ einen Wurzelknoten, so wird automatisch eine Richtung in den Kanten festgelegt. Für $v \in V$ heißt $pre(v), |pre(v)| = 1$ der *Elternteil* und $post(v)$ die *Kinder*.

19. Implementierung von Graphen

19.22 **Definition.** Ein Graph $G = (V, E)$ mit $V = \{1, \dots, n\}$ kann durch einen *Adjazenzmatrix* oder *Nachbarschftsmatrix*

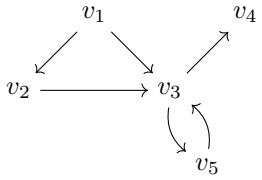
$$A = [a_{i,j}]_{i,j=1}^{n} \in \mathbb{R}^{n,n}$$

mit

$$a_{i,j} = \begin{cases} 1 & , \text{ falls } (i,j) \in E, \text{ bzw. falls } \{i,j\} \in E \\ 0 & , \text{ sonst} \end{cases}$$

dargestellt werden.

19.23 **Beispiel.** Der gerichtete Graph



besitzt die Adjazenzmatrix

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

und falls der Graph als ungerichteter Graph aufgefasst wird:

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

19.24 **Bemerkung.** Bei ungerichteten Graphen ist die Adjazenzmatrix immer symmetrisch. Der Speicherbedarf einer Adjazenzmatrix ist $|V|^2$, unabhängig von $|E|$. Für größere Graphen ist dies ineffizient. Wir werden beim Behandeln von dünnbesetzten Matrizen ein alternatives Speicherformat kennenlernen.
Zum Abschluss bemerken wir, dass sich viele graphentheoretische Konzepte mir Hilfe der Adjazenzmatrix in die Sprach der linearen Algebra übersetzten lässt.

Algorithmen auf Graphen

20. Graphendurchmusterung

Graphen müssen häufig durchmustert, d.h systematisch durchsucht, werden. Im Folgenden werden wir die "Tiefensuche" und die "Breitensuche" betrachten, welche sich beide auf den folgenden Algorithmus zurückführen lassen:

Daten: Graph $G = (V, E)$, Startknoten $s \in V$
Ergebnis: Zusammenhängender, azyklischer Graph $G'(R, T)$ mit $R = \{s\} \cup post^*(s)$ und $T \subset E$

- $R = \{s\}$
- $Q = \{s\}$
- $T = \emptyset$
- (a) **wenn** $Q = \emptyset$ **dann**
  | stop
  Ende
  sonst
  | Wähle $v \in Q \subset R$
  Ende
- Wähle $w \in V \setminus R \cap post(v)$ **wenn** *kein solches w exisitert* **dann**
  | setze $Q = Q \setminus \{v\}$ und gehe zu (a)
  Ende
- $R = R \cup \{w\}$
- $Q = Q \cup \{w\}$
- $T = T \cup \{e\}$ mit $e = (v, w)$ oder $e = \{v, w\}$
- Gehe zu (a)

Algorithmus 12: Algorithmische Suche

Je nachdem wie $v \in Q$ gewählt wird, bezeichnen wir den Suchalgorithmus unterschiedlich:

20.1 Definition. Bei der *Tiefensuche* oder *Depth-First-Search* (DFS) wird derjenige Knoten in $v \in Q$ ausgewählt, der zuletzt zu Q hinzugefügt wurde. Bei der *Breitensuche* oder *Breadth-First-Search* (BSF) wird derjenige Knoten $v \in Q$ ausgewählt, der zuerst zu Q hinzugefügt wurde.

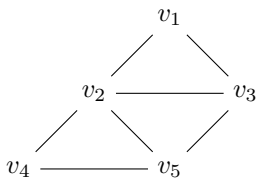
20.2 Satz. Algorithmus 12 liefert einen zusammenhängenden, azyklischen Graphen $G'(R, T)$, mit $R = \{s\} \cup post^*(s)$ und $T \subset E$.

Beweis. Per Konstruktion ist (R, T) zu jedem Zeitpunkt im Algorithmus zusammenhängend.
(R, T) ist außerdem zu jedem Zeitpunkt azyklisch, denn per Konstruktion gilt $Q \subset R$ und e verbindet jeweils $v \in Q \subset R$ mit $w \in V \backslash R$. Wir müssen also zeigen: $R = \{s\} \cup post^*(v)$ ($T \subset E$ ist klar).
Annahme: Nach Ende des Algorithmus gibt es $w \in V \setminus R$ von s erreichbar. Dann gibt es einen Weg

$$\pi = s, v_1, \ldots, v_r, w$$

und darin eine Kante $e = (x, y) \in E$ bzw. $e > \{x, y\} \in E$, mit $x \in R$ und $y \notin R$, $x, y \in \{s, v_1, \ldots, v_r, w\}$. Da $x \in R$, muss zu einem gewissen Zeitpunkt im Algorithmus auch $x \in Q$ gelten. Der Algorithmus terminiert, aber nur, falls x aus Q entfernt wurde, was nur geschieht, wenn $y \notin V \setminus R \cap post(x)$, also falls $e \notin E$ ist.
Dies ist ein Widerspruch zu unserer Annahme. □

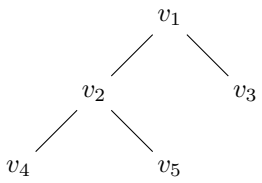
20.3 Beispiel. Betrachte



$s = \{v_1\}$. Bei der Tiefensuche entsteht folgender Baum:

$$v_1 \text{ — } v_2 \text{ — } v_4 \text{ — } v_5 \text{ — } v_3$$

Die Breitensuche hingegen ergibt:



20.4 Bemerkung. Ist G ein ungerichteter Graph, so ist das Resultat G' von Algorithmus 12 ein Baum mit Wurzel s, der *Spannbaum*

20.5 Satz. Die Ausführung von "wähle $v \in Q$" und "wähle $w \in V \setminus R \cap post(v)$" sei in $\mathcal{O}(1)$ durchführbar. Dann besitzt Algorithmus 12 den linearen Aufwand $\mathcal{O}(|V| + |E|)$.

Beweis. "wähle $v \in Q$" wird maximal $(|post(v)| + 1)$ mal aufgerufen.
"wähle $w \in V \setminus R \cap post(v)$" wird für jede Kante maximal ein mal aufgerufen.
Daraus folgt direkt die Behauptung. □

Wir hatten gezeigt, (Satz 16.11), dass $|V| - 1 \leq |E|$, d.h, es gilt $\mathcal{O}(|V| + |E|)$ kann nach oben beschränkt werden durch $\mathcal{O}(2|E|)$.

Beobachtung: Die Breitensuche beschreibt einen "kürzesten" Weg von Startknoten zu jedem anderen Knoten.

20.6 Satz. Sie $G = (V, E)$ ein ungerichteter Graph, $s, v \in V$ und
$dist_G(s, v) = \min\{|\pi| : \pi = s, u_1, \ldots, u_r, v$ ein Weg in G$\}$ mit $dist_G(s, v) = \infty$, falls es keinen Weg von s nach v gibt. Dann enthält der durch die Breitensuche erzeugte Graph $G' = (R, T)$ zum Startknoten $s \in V$ einen kürzesten Weg zu jedem Knoten $v \in post^*(s)$.Dies bedeutet, es gib einen einfachen Weg $\pi = s, u_1, \ldots, u_r, v$ in G' mit $|\pi| = dist_{G'}(s, v) = dist_G(s, v)$ minimal.

Beweis. Wir bemerken zuerst, dass G' azyklisch ist und somit π in G' eindeutig bestimmt ist. Modifiziere Algorithmus 12 nun wie folgt:

- In 1: l(s)=0
- In 4: l(w) = l(v)+1

Dies bedeutet, zu jedem Zeitpunkt im Algorithmus gilt:

$$l(v) = dist_{(R,T)}(s, v), v \in R$$

Insbesondere gibt es für kein in 2 ausgewähltes $v \in Q$ ein $w \in R$ mit:

$$l(w) > l(v) + 1 \tag{20.1}$$

Zu Zeigen: $dist_{R,T}(s, v) = dist_{G'}(s, v) = dist_G(s, v)$
Annahme: Nach Abbruch des Algorithmus gibt es ein $w \in V$ mit

$$dist_G(s, w) < dist_{G'}(s, w) \tag{20.2}$$

Falls es mehr als ein solches w gibt , wähle dasjenige mit den kleinsten Abstand $dist_G(s, w)$.
Sei $\pi = s, u_1, \ldots, u_r, v, w \implies dist_G(s, v) = dist_{G'}(s, v)$, da sonst v ein Knoten mit kleineren Abstand wäre, der (20.2) erfüllt.

$$\begin{aligned} \implies l(w) &= dist_{G'}(s, w) \\ &> dist_G(s, w) \\ &= dist_G(s, v) + 1 \\ &= dist_{G'}(s, v) + 1 \\ &= l(v) + 1 \end{aligned}$$

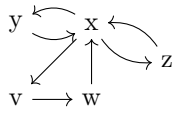
Dies ist ein Widerspruch zu (20.1). □

20.7 **Bemerkung.** Die Breitensuche berechnet also die Wege aller Knoten zur Wurzel in $\mathcal{O}(|V| + |E|)$.

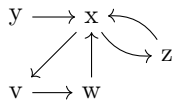
21. Starker Zusammenhang

21.8 **Definition.** Ein Digraph heißt *stark zusammenhängend*, falls für jedes Knotenpaar $v, w \in V$, $v \neq w$, sowohl $v \in post^*(w)$, als auch $w \in post^*(w)$ gilt. Es gibt also einen Weg von v nach w und umgekehrt.
Die *starken Zusammenhangskomponenten* sind die maximalen stark zusammenhängenden Teilgraphen.

21.9 **Beispiel.** Der Graph:



ist stark zusammenhängend.



Dieser Graph, wo der Weg von x nach y entfernt wurde, ist nicht stark zusammenhängend. Die starken Zusammenhangskomponenten sind y und der restliche Graph G'.

Im folgenden konstruieren wir einen Algorithmus zur Bestimmung solcher starker Zusammenhangskomponenten.

Daten: Digraph $G = (V, E)$
Ergebnis: Abbildung $comp\colon V \to \mathbb{N}$, welche die Zugehörigkeit zu einer starken Zusammenhangskomponente signalisiert.

- $R = \emptyset$
- $N = 0$
- **für** $v \in V$ **tue**
  wenn $v \notin R$ **dann**
  FirstVisit(v)
  Ende
  Ende
- $R = \emptyset$
- $K = 0$
- **für** $j \leftarrow |V|$ **bis** *1* **tue**
  wenn $\psi^{-1}(j) \notin R$ **dann**
  $K = K + 1$
  SecondVisit($\psi^{-1}(j)$)
  Ende
  Ende

Function `FirstVisit(`v`)`

- $R = R \cup \{v\}$
- **für** $w \in V \setminus R$, $(v, w) \in E$ **tue**
  FirstVisit(w)
  Ende
- $N = N + 1$
- $\psi(v) = N$
- $\psi^{-1}(N) = V$

zurück
Function `SecondVisit(`v`)`

- $R = R \cup \{w\}$
- **für** $w \in V \setminus R$, $(w, v) \in E$ **tue**
  SecondVisit(w)
  Ende
- $comp(v) = K$

zurück

Algorithmus 13: Bestimmung starker Zusammenhangskomponenten

21.10 Beispiel. Wir betrachten den Graph:

Der Startknoten v_1 für FirstVisit ergibt die Besuchsreihenfolge: v_1, v_7, v_2, v_4, v_5. Die Tiefensuche mittels FirstVisit bricht für v_3 und v_6 direkt ab. Wir erhalten:

$$\begin{array}{lll} \psi(v_2) = 1 & \psi(v_7) = 4 & \psi(v_6) = 7 \\ \psi(v_5) = 2 & \psi(v_1) = 5 & \\ \psi(v_4) = 3 & \psi(v_3) = 6 & \end{array}$$

SecondVisit für $v_6 = \psi^{-1}(7)$ bricht sofort ab $\implies$ $comp(v_6) = 1$
SecondVisit für $v_3 = \psi^{-1}(6)$ bricht sofort ab $\implies$ $comp(v_3) = 2$
SecondVisit für $v_1 = \psi^{-1}(5)$ ergibt die Besuchsreihenfolge: $v_2, v_7 \implies comp(v_1) = 3, v \in \{v_1, v_2, v_7\}$
SecondVisit für $v_4 = \psi^{-1}(3)$ ergibt die Besuchsreihenfolge: $v_5 \implies comp(v_4) = 4$.
Demnach ergibt sich:

$$\{v_6\}, \{v_3\}, \{v_1, v_2, v_7\}, \{v_4, v_5\}$$

21.11 Satz. Algorithmus 13 identifiziert die starken Zusammenhangskomponenten eines Digraphen $G = (V, E)$ in linearem Aufwand $\mathcal{O}(|V| + |E|)$

Beweis. Aufwand: Analog zur Tiefensuche (Satz 20.5)

- Gleiche starke Zusammenhangskomponente *implies* gleicher comp-Wert
  Seien $v, w \in V$ Knoten in der gleichen , starken Zusammenhangskomponente
  - $\implies$ Es gibt einen Weg von v nach w und umgekehrt.
  - $\implies$ SecondVisit weist $comp(v) = comp(w)$ zu
- Gleicher comp-Wert $\implies$ Gleiche starke Zusammenhangskomponente
  Seien $v, w \in V$ mit $comp(v) = comp(w)$. Definiere $r(v)$ ist derjenige von v erreichbare Knoten, der den höchsten ψ-Wert hat.
  Beobachte:
  - $comp(v) = comp(w) \implies v$ und w sind im gleichen, von SecondVisist erzeugten Subbaum. $\implies r(v) = r(w) =: r$ (d.h r ist von v, w erreichbar)
  - $\psi(r) > \psi(v) \implies v$ wurde in FirstVisit vor r nummeriert. $\implies$ Der von FirstVisit erzeugte Baum hat einen Weg von r nach v , d.h v ist von r erreichbar.
  - Analog: w ist von r erreichbar. $\implies v$ ist über r von w aus erreichbar und umgekehrt.

□

21.12 Definition. Sei $G = (V, E)$ ein Digraph. Zieht man die starken Zusammenhangskomponenten zu einem Knoten zusammen, entsteht ein *kondensierter Graph.*
Eine *Nummerierung* $v = \{v_1, \ldots, v_n\}$ der Knoten heißt *topologische Ordnung*, falls $(v_i, v_j) \in E$ nur, wenn $i < j$.

21.13 Lemma. Der Digraph $G = (V, E)$ besitzt eine topologische Ordnung genau dann, wenn er azyklisch ist.

Beweis. Übung □

21.14 Satz. Zu jedem Digraphen $G = (V, E)$ bestimmt Algorithmus 13 eine topologische Ordnung in linearem Aufwand $\mathcal{O}(|V| + |E|)$. Gibt es eine solche Ordnung nicht, so sagt uns das der Algorithmus ebenfalls in linearem Aufwand.

Beweis. Seien $V_i, V_j \subset V$ die (disjunkten) Knotenmengen zweier stabiler Zusammenhangskomponenten mit $comp(v_i) = i, comp(v_j) = j$ für $v_i \in V_i, v_j \in V_j$.
Ohne Beschränkung der Allgemeinheit: $i < j$
Annahme: Es gibt eine Kante $e = (v_j, v_i)$ mit $v_j \in V_j, v_i \in V_i$. *Beobacht:* In SecondVisit werden alle Knoten in V_i vor derjenigen in V_j zu R hinzugefügt. $\Longrightarrow$

- Beim Überprüfen von $e = (v_j, v_i)$ in SecondVisit gilt: $v_i \in R, v_j \notin R$
- v_j wird zu R hinzugefügt, bevor K erhöht wird.
- $comp(v_i) = comp(v_j)$

Dies ist ein Widerspruch.
$\Longrightarrow$ Algorithmus 13 erzeugt eine topologische Ordnung, falls diese existiert.
Aus Lemma 21.13 folgt:
Es gibt genau dann eine topologische Ordnung auf G, falls G azyklisch ist.
$\Longleftrightarrow$ alle starken Zusammenhangskomponenten sind eindeutig.
$\Longleftrightarrow$ Am Ende des Algorithmus ist $K = |V|$ □

21.15 Bemerkung. Der Beweis zeigt, dass Algorithmus 13 auch benutzt werden kann, um in linearer Zeit zu überprüfen, ob ein Graph Zyklen hat.

22. Kürzeste Wege Probleme

22.16 Definition (gewichteter Graph). Sei $G = (V, E)$ ein Graph. Eine *Gewichtsfunktion* für die Kanten von G ist eine Abbildung $w\colon E \to R$. Ist $\pi = v_0, v_1, \ldots, v_r$ ein Weg, so heißt

$$w(\pi) = \sum_{i=1}^{r-1} w(v_i, v_{i+1})$$

die *Weglänge* von π bezüglich w. Das Trippel $G = (V, E, w)$ heißt *gewichteter Graph.*

22.17 Definition. Sei $G = (V, E, w)$ ein gewichteter Graph und $v, \tilde{w} \in V$. Ein *kürzester Weg* von v nach $\tilde{w}$ in G bezüglich w ist ein v-$\tilde{w}$-Weg π mit der Eigenschaft $w(\pi) \leq w(\pi')$ für alle anderen v-w-Wege π' . Die kürzeste Weglänge $\delta(v, \tilde{w})$ von v nach $\tilde{w}$ ist definiert durch:

$$\delta(v, \tilde{w}) \begin{cases} \min\{w(\pi) | \pi \text{ ist } v\text{-}\tilde{w}\text{-Weg } \} & \text{, falls ein solcher Weg existiert} \\ \infty & \text{, sonst} \end{cases}$$

22.18 Beispiel. Betrachte den gewichteten Graph $G = (V, E, w)$ mit

Mögliche Wege von v_3 nach v_1 sind:

$$\begin{aligned} \pi_1 &= v_3, v_1 && \Longrightarrow\ w(\pi_1) = 6 \\ \pi_2 &= v_3, v_2, v_1 && \Longrightarrow\ w(\pi_2) = 5 \\ \pi_3 &= v_3, v_4, v_2, v_1 && \Longrightarrow\ w(\pi_3) = 9 \end{aligned}$$

22.19 Notation. Wir werden im Folgenden nur Digraphen behandeln, da sich ungerichtete Graphen immer auch als gerichtete Graphen interpretieren lassen. Sei $G = (V, E, w)$ ein gewichteter Digraph.

Wir unterscheiden drei verschiedene Varianten des *kürzeste-Wege-Problems:*

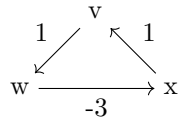
a) Einzelpaar-kürzeste-Wege-Problem
Gegeben $v, u \in V$, suche einen kürzesten Weg von v nach u.

b) Einzelquelle-kürzeste-Wege-Problem
Gegeben $v \in V$, suche einen kürzesten Weg von v zu allen $u \in post^*(v)$.

c) Alle-Paare-kürzeste-Wege-Problem
Finde alle Paare $v, u \in V$ einen kürzesten v-u-Weg

Wir bemerken, dass das Problem c die Probleme a und b löst, daher betrachten wir im Folgenden nur Problem c.
Negative Gewichte sind zwar nach Definition 22.17 zugelassen, können aber Probleme bereiten:

22.20 Beispiel. Betrachte:



In diesem Graphen gibt es keinen kürzesten Weg.

$$\begin{aligned} w(v, w, x) &= -2 \\ w(v, w, x, v, w, x) &= -3 \end{aligned}$$

Man kann den Weg also immer weiter verlängern und der Weg wird immer kürzer.

22.21 Lemma. Sei $G = (V, E, w)$ ein gewichteter Digraph. Falls es in G keine Zyklen mit negativer Weglänge gibt, dann gibt es für $v, u \in V, u \in post^*(v)$ einen kürzesten Weg π mit:

$$\delta(v, w) = w(\pi) > -\infty$$

Beweis. Da es keine negativen Zyklen gibt, genügt es alle einfachen Wege von v nach u zu betrachten. Da $|V|$ und $|E|$ endlich sind, sind dies endlich viele, so dass π durch einen Minimierer gegeben ist. □

22.22 Lemma. Sei $G = (V, E, w)$ ein gewichteter Digraph ohne negativen Zyklen. Ist $\pi = v_0, \ldots, v_r$ ein kürzester Weg von v_0 nach v_r dann ist der Teilweg

$$\pi_{i,j} = v_i, \ldots, v_j, 0 \leq i < j \leq r$$

ein kürzester Weg von v_i nach v_j.

Beweis. Angenommen: $\pi'_{i,j}$ ist ein kürzester Weg von v_i nach v_j mit $w(\pi'_{i,j} < w(\pi_{i,j})$

$$\begin{aligned} \implies \pi' &= \pi_{0,i}\pi'_{i,j}\pi_{j,r} \text{ erfüllt:} \\ w(\pi') &= w(\pi_{0,i}) + w(\pi'_{i,j}) + w(\pi_{j,r}) \\ &< w(\pi_{0,i}) + w(\pi_{i,j}) + w(\pi_{j,r}) \\ &= w(\pi') \\ &\implies w(\pi') < w(\pi) \end{aligned}$$

Dies ist ein Widerspruch, da π kürzester Weg nach Annahme ist. □

22.23 Lemma. Sei $G = (V, E, w)$ ein gewichteter Digraph ohne negative Zyklen und sei $\pi = v_0, v_1, \ldots, v_r$ ein kürzester Weg von v_0 nach v_r . Dann gilt:

$$\delta(v_0, v_r) = \delta(v_0, v_{r-1} + w(v_{r-1}, v_r)$$

Beweis. Aus Lemma 22.22 folgt:

- $\pi' = v_0, \ldots, v_{r-1}$ ist ein kürzester Weg von v_0 nach v_{r-1}
- $\delta(v_0, v_{r-1} = w(\pi')$
- $\delta(v_0, v_r) = w(\pi)$

$w(\pi)$ lässt sich umschreiben:

$$\begin{aligned} w(\pi) &= w(\pi') + w(v_{r-1}, v_r) \\ &= \delta(v_0, v_{r-1} + w(v_{r-1}, v_r) \end{aligned}$$

□

Daten: Gewichteter Graph $G = (V, E, w)$ mit nicht negativen Gewichten
Ergebnis: Kürzeste Wege von s nach $v \in post^*(s)$ samt Weglänge $l(v) = \delta(s, v)$

- $l(s) = 0$
  $l(v) = \infty$, $v \in V \setminus \{s\}$
  $R = \emptyset$
- Finde $u \in V \setminus R$ mit $l(n) = \min_{v \in V \setminus R} l(v)$
- $R = R \cup \{u\}$
- **für** $v \in V \setminus R$ *mit* $(u, v) \in E$ **tue**
  wenn $l(v) > l(u) + w(u, v)$ **dann**
  $l(v) = l(u) + w(u, v)$
  $p(v) = u$
  Ende
  Ende
- $R \neq V$ gehe zu 2.

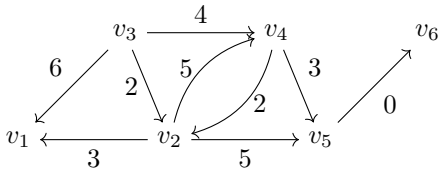
Algorithmus 14: Dijkstra Algorithmus

22.24 Bemerkung. Die kürzesten Wege im Output des Algorithmus sind durch die Abbildung $p\colon V \to V$ gegeben, denn

$$\pi = s, \ldots, p(p(p(v))), p(p(v)), p(v), v$$

ist ein kürzester s-v-Weg.

22.25 Beispiel. Der Graph



mit Startknoten v_3 hat nach Ausführung von Algorithmus 14 folgende Ausgabe:

Tabelle 1.: Dijkstra

Iteration	$l(v_1), l(v_2), \ldots, l(v_6)$						u, $l(u)$, $p(u)$		
0	∞	∞	0	∞	∞	∞	v_3	0	$-$
1	6	2		4	∞	∞	v_2	2	v_3
2	5				7	∞	v_4	4	v_3
3						∞	v_1	5	v_2
4						∞	v_5	7	v_2
5						7	v_6	7	v_5

22.26 **Satz.** Der Algorithmus von Dijkstra 14 arbeitet korrekt, mit einer Laufzeit von $\mathcal{O}(n^2), n = |V|$.

Beweis.

22.27 **Notation.** Schreibe in der k-ten Iteration $^{(k)}$ an alle Größen, die im Algorithmus vorkommen

Zeige: Bei jeder Ausführung von 2. gilt:

a) $l^{(k)}(v) \leq l^{(k)}(y)$ für alle $v \in R^{(k)}, y \in V \setminus R^{(k)}$

b) $l^{(k)}(v) = \delta(s, v)$ für alle $v \in R^{(k)}$
$l^{(k)}(v) < \infty, v \neq s \implies$ Es existiert ein kürzester s-v-Weg mit Knoten in $R^{(k)}$ und letzter Kante $(p(v), v)$

c) $v \in V \setminus R^{(k)} \implies l^{(k)}(v)$ ist die kürzeste Weglänge von s nach v im Teilgraphen mit Knoten $R^{(k)} \cup \{v\}$ in G
$v \in V \setminus R^{(k)}, l^{(k)}(v) < \infty \implies p^{(k)}(v) \in R^{(k)}$ und $l^{(k)}(v) = l^{(k)}(p(v)) + w(p^{(k)}(v), v)$

Per vollständiger Induktion:

$\Diamond$ IV $k = 0$ Trivialerweise erfüllt

$\Diamond$ IS $k \to k + 1$ Sei u der in 2. ausgewählte Knoten.

a) Für $v \in R^{(k)}, y \in V \setminus R^{(k)}$ gilt:

$$\begin{aligned} l^{(k+1)}(v) &= l^{(k)}(v) \\ &\leq l^{(k)}(u) \\ &= \min_{u \in V \setminus R^{(k)}} l^{(k)}(u) \\ &\leq l^{(k+1)}(y) \end{aligned}$$

Da $R^{(k+1)} = R^{(k)} \cup \{u\} \implies$ Behauptung a) für $k + 1$ gilt.

b) Wir müssen die Behauptung über die kürzeste Weglänge nur für u überprüfen.

c) für $k \implies l^{(k)}(u)$ ist die kürzeste Weglänge zum Weg π im Teilgraphen $R^{(k)} \cup \{u\}$.
zu Zeigen $l^{(k+1)}(u) = l^{(k)}(u) = w(\pi) = \delta(s,u)$ in G
Angenommen: Es gibt einen Weg π' in G mit mindestens einem Knoten $y \in V \setminus R^{(k)}$ und $w(\pi' < w(\pi)$
$\pi' = s, v_1, \ldots, v_{r-1}, y, v_{r+1}, \ldots, v_{r+m}, u$

$$\begin{aligned} \implies l^{(k)}(y) &\leq w(s, v_1, \ldots, v_{r-1}, y) \\ &\leq w(\pi') \\ &< w(\pi) \\ &= l^{(k)}(u) \end{aligned}$$

Dies ist ein Widerspruch zu $l^{(k)}(u) = \min_{v \in V \setminus R^{(k)}} l^{(k)}(v)$
$\implies$ b für $k+1$

c) Zeige, dass 3. und 4. die Aussage c) erhalten. Sei $v \in V \setminus R^{(k+1)}$ in 4.

$$\begin{aligned} \implies p^{(k+1)}(v) &= u \\ l^{(k+1)}(v) &= l^{(k+1)}(u) + w(u,v) \end{aligned}$$

c) für $k \implies$ Es gibt einen s-v-Weg im von $R^{(k+1)} \cup \{v\}$ aufgespannten Teilgraph $G'(v)$ in G , mit Länge $l^{(k+1)}(v) = l^{(k+1)}(u) + w(u,v)$
zu Zeigen Dieser Weg ist ein kürzester Weg in G
Angenommen: Für ein $v \in V \setminus R^{(k+1)}$ in 4. gibt es einen s-v-Weg π in $G'(v)$ mit $w(\pi) < l^{(k+1)}(v)$.
Beobachte: $l^{(k+1)}(v) \leq l^{(k)}(v)$
$\implies$ u muss π enthalten sein, da dies die einzige Veränderung zum k-ten Schritt ist und $l^{(k)}(v)$ nach c) im k-ten Schritt ein kürzester Weg in $R^{(k)} \cup \{u\}$ ist.
Sei x der Vorgänger von v in π.

$$\begin{aligned} l^{(k+1)}(v) &= l^{(k+1)}(x) + w(x,v) \\ &\leq l^{(k+1)}(u) + w(x,u) \\ &\leq w(\pi) \end{aligned}$$

Dies ist ein Widerspruch $\implies$ c) für $k+1$. Für $k = n$ impliziert b) die Behauptung.

Aufwand: $\mathcal{O}(n^2)$ ist trivial. □

22.28 Bemerkung. Clevere Implementierungen können den Dijkstra-Algorithmus 14 in $\mathcal{O}(m + n\log(n), n = |V|, m = |E|$ durchführen.

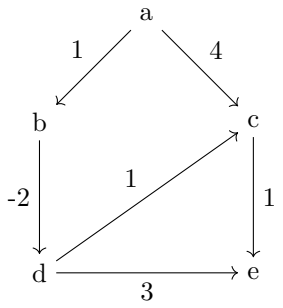
Für die Verallgemeinerung für negative Gewichte betrachten wir folgend den Moore-Bellmann-Ford Algorithmus:

Daten: Gewichteter Digraph $G = (V, E, w)$ ohne negative Zyklen, Startknoten $s \in V$
Ergebnis: kürzeste Wege von s nach v, $v \in post^*(s)$ samt Weglänge $l(v) = \delta(s, v)$

- $l(s) = 0$
  $l(v) = \infty, v \in V \setminus \{s\}$
- **für** $e = (u, v) \in E$ **tue**
  wenn $l(v) > l(u) + w(u, v)$ **dann**
  $l(v) = l(u) + w(u, v)$
  $p(v) = u$
  Ende
  Ende

Algorithmus 15: Moore-Bellmann-Ford

22.29 Beispiel. Betrachte den Graphen:



mit Startknoten a.

Tabelle 2.: Iteration 1

Kante	$l(a), \ldots, l(e)$					$p(b), \ldots, p(e)$			
(a,b)	0	1	∞	∞	∞	a	-	-	-
(a,c)			4	∞	∞		a	-	-
(b,d)			4	-1	∞		a		c
(c,e)			4		5		a		c
(d,c)			0		5		d		c
(d,e)					2				d

Tabelle 3.: Iteration 2

Kante	$l(a), \ldots, l(e)$					$p(b), \ldots, p(e)$			
(a,b)	0	1	0	-1	2	a	d	b	d
(a,c)									
(b,d)									
(c,e)									
(d,c)					1				c
(d,e)									

Keine Veränderung mehr ab der 3. Iteration.

22.30 **Satz.** Der Moore-Bellman-Ford Algorithmus 15 arbeitet korrekt mit Aufwand $\mathcal{O}(m \cdot n), m = |E|, n = |V|$.

Beweis. Bezeichne zu jedem Zeitpunkt:

$$R = \{v \in V | l(v) < \infty\}$$
$$F = \{(u,v) \in E | u = p(v)\}$$

Zeige zu erst:

a) $l(v) \geq l(u) + w(u,v), (u,v) \in E$

b) (R,F) ist ein azyklischer Graph

c) (R,F) ist ein gerichteter Baum mit Wurzel s, d.h jeder Knoten in R ist über genaue einen Weg von s aus erreichbar.

a) Bei $p(v) = u$ gilt:

$$l(v) = l(u) + w(u,v)$$

Danach wird $l(u)$ höchstens verkleinert.

b) Angenommen: Es gibt einen Zyklus $\pi = v_0, v_1, \ldots, v_{r-1}, v_r$ mit $v_0 = v_r$
Ohne Beschränkung der Allgemeinheit entstanden durch Setzen von $p(v_r) = v_{r-1}$
$\Longrightarrow$ Davor galt:

$$l(v_0) = l(v_r) > l(v_{r-1}) + w(v_{r-1}, v_r)$$

Wegen a) gilt:

$$l(v_{i+1}) \geq l(v_i) + w(v_i, v_{i+1}, i = 0, \ldots, r-2)$$

$$\begin{aligned} \Longrightarrow \sum_{i=1}^{r} w(v_{i-1}, v_i) &= \left(\sum_{i=1}^{r-1} w(v_{i-1}, v_i) \right) + w(v_{r-1}, v_r) \\ &< \sum_{i=1}^{r} (l(v_i) - l(v_{i-1}) \\ &= 0 \end{aligned}$$

Widerspruch dazu, dass es keine negativen Zyklen gibt.

c) $x \in R \setminus \{s\} \Longrightarrow p(x) = R$ Gehe von Blättern zu Wurzel dann folgt c)
$a, b, c \Longrightarrow l(y)$ ist mindestens die Länge des (eindeutigen) s-y-Wegs in (R, F)
Behauptung: Nach k Iterationen ist $l(y)$ auch die maximale Länge eines kürzesten s-y-Wegs in G mit maximal k Kanten.
Zeige durch Induktion:

$\Diamond$ IV $k = 1$ klar

$\Diamond$ $k - 1 \to k$ Sei

$$\pi = s, v_1, \ldots, v_r, x, y$$

ein kürzester s-y-Weg in G mit höchsten k Kanten und $r \leq k - 2$.
Dann folgt aus Lemma 22.23:

- $\pi' = s, v_1, \ldots, v_r, x$ ist ein kürzester s-k-Weg mit k-1 Kanten
- $l(x) \leq w(\pi')$
-

$$\begin{aligned} l(y) &\leq l(x) + w(x, y) \\ &\leq w(\pi') + w(x, y) \\ &= w(\pi) \end{aligned}$$

Wir haben gezeigt: $w(\pi) = \delta(s, y)$, wobei π der eindeutige Weg in (R, T) mit maximal k Kanten.
Daraus folgt die Behauptung.

Aufwand: Trivial. □

Die letzte Frage dieses Kapitels: Algorithmus für das Alle-Paare-kürzeste-Wege Problem lässt eine naive Antwort zu. Diese lautet: Wende Dijkstra oder Moore-Belmann-Ford auf jede Quelle an.Das Resultat ist jedoch, dass man keine brauchbaren Datenstrukturen mehr hat. Demnach gibt es folgenden Algorithmus:
Ohne Beschränkung der Allgemeinheit sei $V = \{1, \dots, n\}$

Daten: Gewichteter Digraph $G = (V, E, w), V = \{1, \dots, n\}$ ohne negative Zyklen
Ergebnis: Matrizen $[l_{i,j}]_{i,j=1}^{n}, [p_{i,j}]_{i,j=1}^{n}$

Algorithmus 16: Floyd-Warshall

22.31 **Satz.** Der Algorithmus 16 arbeitet korrekt, mit Aufwand $\mathcal{O}(n^3), n = |V|$

Beweis.

22.32 **Notation.** Im Iterationsschritt j_0 ist $l_{ik}^{(j_0)}$ die Länge eines kürzesten i-k-Weges im von $\{1, \dots, j_0\}$ aufgespannten Teilgraphen mit Endkante $(p_{ik}^{(j_0)}, k)$.

□

23. Netzwerkflussprobleme

23.33 **Beispiel.** Ausgehend von einer Bananenplantage s sollen alle geernteten Bananen zum Lagerhaus t transportiert werden. Für den Transport stehen Straßen mit $r_1, \dots, r_p$ $\frac{kg}{n}$ Transportkapazität zu den Seehäfen $A_1, \dots, A_p$ zu Verfügung. Von den Zielhäfen $B_1, \dots, B_q$ stehen Transportkapazitäten von $d_1, \dots, d_q$ $\frac{kg}{n}$ zum Supermarkt t bereit. Die Transportkapazität zwischen den Seehäfen werden mit $c(A_i, A_j), 1 \leq i \leq p, 1 \leq j \leq p$ bezeichnet.

Fragen:

- Ist es mögliche, alle Transportkapazitäten auszuschöpfen?
- Falls nein, was ist die maximal mögliche Transportkapazität?
- Wie sollen die Bananenladungen aufgeteilt werden?

Konstruiere einen gewichteten Digraph $G = (V, E, w)$ mit

- $V = \{s, A_1, \dots, A_p, B_1, \dots, q, t\}$
- $E = \{(s, A_i), (A_i, B_j), (B_j, t); 1 \leq i \leq p, 1 \leq j \leq p\}$
- $w(e) = \begin{cases} r_i & , e = (s, A_i) \\ c(A_i, B_j) & , e(A_i, B_j) \\ d_i & , e(B_i, t) \end{cases}$

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

- einem gewichteten Digraphen $G = (V, E, c)$
- einer *Kapazitätsfunktion* $c\colon E \to R_{\geq 0}$
- einer Quelle $s \in V$ mit $pre(s) = \emptyset$
- einer Senke $t \in V$ mit $post(t) = \emptyset$

Ein Fluss $f\colon E \to R_{\geq 0}$ ist eine Funktion, die folgende Bedingungen erfüllt:

a) *Kapazitätsbedingung:*

$$f(v, w) \leq c(v, w)$$

b) *Kirchhoffsches Gesetz:*

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

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

Der *Wert* des Flusses ist:

$$flow(f) = \sum_{w \in post(s)} f(s, w)$$

Der *maximale FLuss* von N wird bezeichnet als:

$$MaxFlow(N) = \max\{flow(f) \text{ ist Fluss für N}\}$$

Eine Flussfunktion f wird *optimal* genannt, falls

$$flow(f) = MaxFlow(N)$$

ist.
Ein *Schnitt* für N ist eine Knotenmenge $S \subset V$ mit $s \in S, t \notin S$.
Die *Kapazität* eines Schnittes ist gegeben durch:

$$cap(S) = \sum_{v \in S w \in post(v) \setminus S} c(v, w)$$

Die minimale Schnittkapazität von N ist:

$$MinCut(N) = \min\{cap(S) | \text{ S ist Schnitt für N}\}$$

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

$$flow(f) = \sum_{w \in post(v) \setminus S} f(v, w) - \sum_{u \in pre(v)} f(u, v)$$

$$flow(f) \leq cap(S)$$

Beweis. Rechne:

$$\begin{aligned} flow(f) &= \sum_{w \in post(s)} f(s,w) \\ &= \sum_{v \in S} \left(\sum_{w \in post(v)} f(v,w) - \sum_{u \in pre(v)} f(u,v) \right) \\ &= \sum_{w \in post(v)} f(v,w) - \sum_{u \in pre(v) \setminus S} f(u,v) \end{aligned}$$

Für die nächste Behauptung können wir ebenfalls nachrechnen:

$$\begin{aligned} flow(f) &= \sum_{w \in post(v) \setminus S} f(v,w) - \sum_{u \in pre(v) \setminus S} f(u,v) \\ &\leq \sum_{w \in post(v) \setminus S} c(v,w) \\ &= cap(S) \end{aligned}$$

□

23.36 Satz (Max-FLow-Min-Cut Theorem). Sei $N = (V, E, c, s, t)$ ein Netzwerk, dann gilt:

$$MaxFlow(N) = MinCut(N)$$

Beweis. Zu zeigen: Es gibt einen Schnitt für den die Gleichheit gilt.
Idee: Gegeben sei ein Fluss f mit flow(f)=MaxFlow(N), konsturiere einen Schnitt S mit flow(f)=cap(S).

Daten: Netzwerk N, Fluss f mit $flow(f) = MaxFlow(N)$.
Ergebnis: S mit $flow(f) = cap(S)$

- $S = \{s\}$
- **solange** $x \in S, y \in V \setminus S$ *existieren mit* $\begin{cases} f(x,y) < c(x,y) & , \textit{falls } (x,y) \in E \\ 0 < f(y,x) & , \textit{falls } (y,x) \in E \end{cases}$

  dann **tue**
  | $S = S \cup \{y\}$

  Ende

Behauptung Das Resultat S des Algorithmus ist ein Schnitt von N

Beweis. Zu Zeigen: $t \notin S$
Angenommen: $t \in S, v_r = t$
$\Longrightarrow$ Es gibt $v_{r-1} \in S$ mit $f(v_{r-1}, v_r) < c(v_{r-1}, v_r)$.

Iterativ: Es gibt einen ungerichteten Weg π mit $v_0 = s, v_r = t$
Definiere für $i = 0, \ldots, r-1$:

$$\varepsilon_i = \begin{cases} c(e) - f(e) & , \text{ falls } e = (v_i, v_{i+1}) \in E, e^{-1} = (v_{i+1}, v_i) \notin E \\ f(e^{-1}) & , \text{ falls } e \notin E, e^{-1} \in E \\ \max\{c(e) - f(e), f(e^{-1})\} & , \text{ falls } e \in E, e^{-1} \in E \end{cases}$$

Beobachte: $\varepsilon_i > 0$

$$\varepsilon = \min_{\leq i \leq r-1} \varepsilon_i > 0 \tag{23.1}$$

Zeige: Es gib einen Fluss f^* in N mit $flow(f^*) = flow(f) + \varepsilon$
Konstruiere:

$$f^*(e) = \begin{cases} f(e) + \varepsilon & , \text{ falls } e = (v_i, v_{i+1} \in E, e^{-1} \notin E \\ f(e) - \varepsilon & , \text{ falls } e = (v_{i+1}, v_i) \in E, e^{-1} \notin E \\ f(e) + \varepsilon & , \text{ falls } e = (v_i, v_{i+1}), e^{-1} \in E \text{ und } c(e) - f(e) > f(e^{-1}) \\ f(e) - \varepsilon & , \text{ falls } e = (v_{i+1}, v_i), e^{-1} \in E \text{ und } c(e) - f(e) \leq f(e^{-1}) \\ f(e) & , \text{ sonst} \end{cases}$$

Bemerke:

- Kapazitätsbedingung bleibt erfüllt.
- Kirchpffsches Gesetz bleibt erfüllt, da f^* weiterhin ein Fluss ist.

Das heißt:

$$\begin{aligned} flow(f^*) &= \sum_{v \in post(s)} f^*(s, v) \\ &= \sum_{v \in pre(t)} f^*(v, t) \\ &= \sum_{v \in pre(t) \setminus \{v_{r-1}\}} f^*(v, t) + f^*(v_{r-1}, t) \\ &= flow(f) + \varepsilon \end{aligned}$$

Dies ist ein Widerspruch dazu, dass $flow(f) = MaxFlow(N) \implies S$ ist ein Schnitt. □

S ist ein Schnitt in N mit:

$$f(x, y) = c(x, y)$$
$$f(y, x) = 0$$

für alle $x \in S, y \in V \setminus S \implies flow(f) = cap(S)$ nach Kirchoff. □

Wir formalisieren nun weiter:

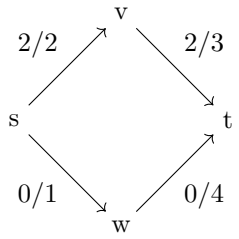
23.37 Definition. Sei f ein Fluss im Netzwerk $N = (V, E, c, s, t)$. Eine Kante $e = (x, y) \in E$ heißt *Vorwärtskante*, falls $f(e) < c(e)$
Eine Kante $e^{-1}(y, x) \in E$ heißt *Rückwärtskante*, falls $f(e^{-1}) > 0$.
Der *Restgraph* für f ist der Digraph $G' = (V, E')$ mit

$$E' = \{(x, y) \in V \times V | (x, y) \text{ ist Vorwärts- oder Rückwärts-Kante}\}$$

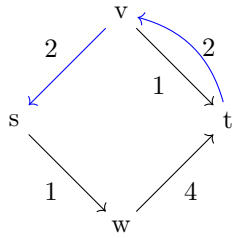
$c(e) - f(e)$ bzw. $f(e^{-1})$ heißen *Restkapazitäten*. Ein *augmentierender Weg* ist ein s-t-Weg im Restgraphen.

Im vorherigen Beweis haben wir einen solchen augmentierenden Weg konstruiert.

23.38 Beispiel. Betrachte



Der Restgraph mit Restkapazitäten ist:



Blau: Rückwärtskante, Rot: Vorwärtskante

Der Beweis von Satz 23.36 gibt uns außerdem einen Algorithmus, wie wir, für einen nicht-optimalen Fluss, ein Fluss mit höheren Wer finden können. Idee: Benutze dies um einen optimalen Fluss zu finden.

Daten: Netzwerk $N = (V, E, c, s, t)$
Ergebnis: Fluss f mit $flow(f) = MaxFlow(N)$

- $f(e) = 0, e \in E$
- Suche einen augmentierenden Weg von s nach t
- **wenn** *keiner existiert* **dann**
  | stop

  Ende
- Berechne ε (23.1)
- Augmentiere f um ε, gehe zu 2

Algorithmus 17: Ford-Fulkerson

23.39 **Bemerkung.** Im Falle von irrationalen Kapazitäten muss der Algorithmus nicht notwendigerweise terminieren (Beweis durch Gegenbeispiel)

23.40 **Satz** (Integral-Flow-Theorem). Sei $N = (V, E, c, s, t)$ ein Netzwerk mit ganzzahligen Kapazitäten. Dann terminiert der Algorithmus 18 nach maximal $\sum_{e \in E} c(e)$ Augmentierungsschritten mit einem ganzzahligen, optimalen Fluss.

Beweis. Betrachte:

Ganzzahlig: Wir starten mit $f(e) = 0, e \in E$. Die Restkapazitäten im Algorithmus und somit auch ε sind zu jedem Zeitpunkt ganzzahlig.

Optimal: Der Algorithmus terminiert, falls kein augmentierender Weg mehr gefunden werden kann. Der Beweis des Max-Flow-Min-Cut-Theorem zeigt, dass dies den maximalen Fluss impliziert.

Anzahl Schritte Im schlimmsten Fall wird pro Iteration der Fluss von genau einer Kante um 1 erhöht $\Longrightarrow$ Behauptung □

Es gibt eine Möglichkeit, die Laufzeit des Algorithmus zu verbessern, indem man immer den kürzesten augmentierenden Weg wählt. Dies bringt uns zum nächsten Algorithmus: