Algorithmische Diskrete Mathematik

Prof. Dr. Alexander Martin

SS 2008

Inhaltsverzeichnis

1 Endliche Mengen

Definition 1.1 (Menge)
Eine **Menge** ist eine Ansammlung verschiedener Objekte, auch **Elemente** genannt.

Beispiel:
Die Menge aller Haare auf menschlichen Köpfen
Die Menge aller Kanten eines Kantenstapels
Die Menge aller Atome im Weltall
Die Menge der reellen Zahlen

$\mathbb{N} = \{1, 2, 3, \ldots\}$ - natürliche Zahlen
$\mathbb{N}_0 = \{0, 1, 2, 3, \ldots\}$
$\mathbb{Z} = \{0, 1, -1, 2, -2, \ldots\}$ - ganze Zahlen
$\mathbb{Q} = \left\{\frac{p}{q} : \ p \in \mathbb{Z}, q \in \mathbb{Z} \setminus \{0\}\right\}$ - rationale Zahlen
$\mathbb{R}$ - reelle Zahlen

Gibt man die Elemente einer Menge explizit an, so verwendet man geschweifte Klammern.

Beispiel:

$$P = \{\text{Gustav}, \text{Gregor}, \text{Georg}, \text{Gunter}\}$$
$$M = \{1, 17, 39, 34, 18\}$$

1.1 Schreibweisen und Mengenoperationen

Eine Menge, die keine Elemente enthält, wird als leere Menge bezeichnet und mit einer durchgestrichenen Null $\emptyset$ oder leere, Mengenklammern $\{\}$ abgekürzt.

Man beachte den Unterschied zwischen $\emptyset$ und $\{\emptyset\}$. Ersteres ist die leere Menge selbst, Zweiteres ist die Potenzmenge der leeren Menge, eine Menge, die genau ein Element enthält.
Die beiden Zeichen „ : “ und „ | “ stehen als Abkürzung für „mit der Eigenschaft“.

Beispiel:
$\{x \in P \mid$ Name endet mit 'v'$\}$
$\{x \in \mathbb{Z} \mid\ x \geq 0\}$

$\in$ steht für „ist ein Element von“
$\notin$ ist eine Abkürzung für „ist nicht Element von“

Aufpassen „Barbier von Sevillia“
„Der Barbier von Sevillia rasiert alle Männer, die sich nicht selbst rasieren.“ - Wer rasiert den Barbier? Dies ist ein Paradoxon.
Die Schreibweise „ $\mid$ “ erlaubt nicht notwendigerweise eine Zuordnung oder nicht Zuordnung von Elementen zu Mengen.

Seien A, B, C Mengen. Dann bezeichnet

$A \cap B = \{x : x \in A \text{ und } x \in B\}$

$A \cup B = \{x : x \in A \text{ oder } x \in B\}$

$A \setminus B = \{x : x \in A \text{ und nicht } x \in B\}$

$A \subset B$ bedeutet A ist Teilmenge von B

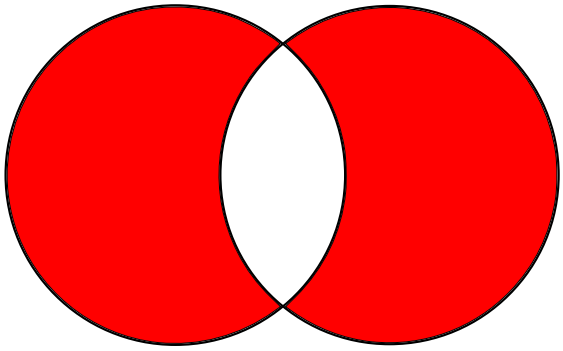
$A = B \Leftrightarrow\ A \subset B$ und $B \subset A$

$|A| =$ Kardinalität von A (Mächtigkeit von A, also die Anzahl Elemente in A)

Gilt $|A| < \infty$, so heisst A **endlich**.
Gilt $|A| = \infty$, so heisst A **unendliche Menge**.
Bei unendlichen Mengen wird noch weiter unterschieden in abzählbare und überabzählbare Mengen.

Definition: (symmetrische Differenz)
Die Menge $A \triangle B$ wird als **symmetrische Differenz** bezeichnet. Es handelt sich um die Menge aller Elemente, die jeweils in einer, aber nicht in beiden Mengen liegen.

$$A \triangle B := (A \setminus B) \cup (B \setminus A) = (A \cup B) \setminus (A \cap B)$$



$A \triangle B$

1.2 Rechenregeln

1.) $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$

Beweis:

I.) Sei $x \in A \cap (B \cup C) \Longrightarrow x \in A$ und $x \in B \cup C \Longrightarrow x \in A$ und ($x \in B$ oder $x \in C$). Im ersten Fall ist damit $x \in A \cap B$, im zweiten Fall $x \in A \cap C$. Also insgesamt $x \in (A \cap B) \cup (A \cap C)$

II.) $x \in (A \cap B) \cup (A \cap C) \Longrightarrow$ a.) $x \in A \cap B$ oder $x \in A \cap C \Longrightarrow x \in A$ und b.) $x \in B$ oder $x \in C \Longrightarrow x \in A \cap (B \cup C)$ □

Analogie: $\cap$ sei äquivalent zu $\cdot$ und $\cup$ sei äquivalent zu $+$. Es gelten damit:
1.) $A \cdot (B + C) = AB + AC$ (Distributivität)
Weitere Regeln sind:
2.) $A \cup B = B \cup A$, $A \cap B = B \cap A$ (Kommutativität)
3.) $(A \cup B) \cup C = A \cup (B \cup C)$, $(A \cap B) \cap C = A \cap (B \cap C)$ (Assoziativität)
4.) $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$ (Distributivität)

Achtung: 4.) gilt nicht bei Zahlen $a + (b \cdot c) \neq (a + b)\,(b + c)$

1.3 Teilmengen und Anordnungen einer Menge

Satz 1.2
Eine n-elementige Menge besitzt 2^n Teilmengen.

Beweis:
Jedes Element ist entweder in der Menge enthalten oder nicht. Es gibt also 2 Möglichkeiten pro Element und bei n Elementen insgesamt

$$\underbrace{2 \cdot 2 \cdot \ldots \cdot 2}_{n-\text{mal}} = 2^n.$$

□

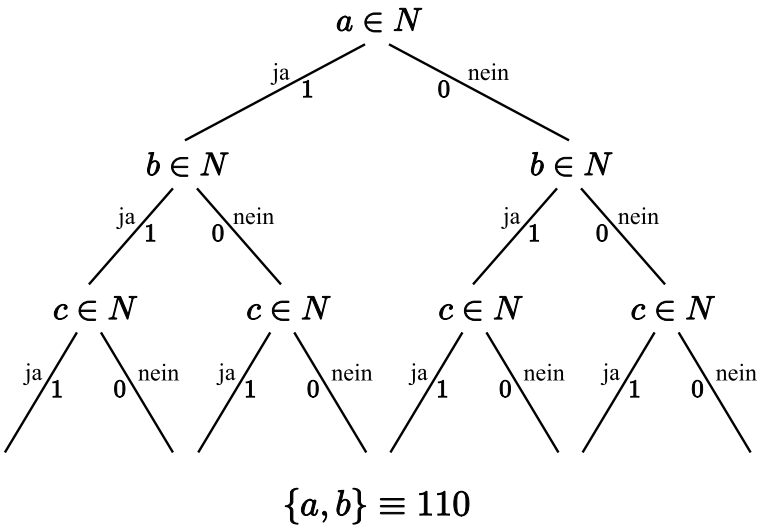
Beispiel:

M	Teilmengen von M
$\emptyset$	$\emptyset$
$\{a\}$	$\emptyset, \{a\}$
$\{a, b\}$	$\emptyset, \{a\}, \{b\}, \{a, b\}$
$\{a, b, c\}$	$\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}$

Darstellung über Binärzahlen mit n Stellen

$\{a\}$	$\{b\}$	$\{c\}$		
1	1	0	$\equiv$	$\{a, b\}$
0	0	0	$\equiv$	$\emptyset$
0	0	1	$\equiv$	$\{c\}$

Darstellung als Baum



Satz 1.3
Die Anzahl n-stelliger Strings, die aus k Elementen erzeugt werden können, beträgt k^n.

Beweis:
Für die erste Stelle gibt es k Möglichkeiten, für die Zweite $k \cdot k$, für die Dritte $k \cdot k \cdot k$, ..., für die n-te $\underbrace{k \cdot k \cdot k \cdot \ldots \cdot k}_{n\text{-mal}} = k^n$. □

Korollar 1.4
Stehen für Stelle i jeweils k_i Elemente zur Auswahl, so ist die Anzahl n-stelliger Strings $\prod_{i=0}^{n} k_i$.

1.4 Permutationen

Mengen sind an sich nicht geordnet, d.h.

$$\{a, b\} = \{b, a\}\,.$$

Legt man Wert auf die Anordnung, d.h. die Reihenfolge, in der die Elemente aufgelistet werden, so spricht man von einer **Ordnung** oder geordneten Liste. Tauscht man die Reihenfolge, so spricht man von einer **Permutation**.

Beispiel:
$M = \{1, 2, 3\}$
Permutationen: 123, 132, 213, 231, 231, 312, 321.

Satz 1.5
Die Anzahl Permutationen einer n-elementigen Menge beträgt $n!$.

Beweis:
Für das erste Element gibt es n mögliche Positionen, für das zweite $n-1$, für das dritte $n-2$, für das vierte $n-3$, ...,$3 \cdot 2 \cdot 1 = n!$ □

Möchte man nur k Elemente aus einer n-elementigen Menge anordnen, so ergibt sich

Korollar 1.6
Die Anzahl der geordneten k-elementigen Teilmengen einer Menge mit n Elementen beträgt

$$n \cdot (n-1) \cdot \ldots \cdot (n-k+1) = \frac{n!}{(n-k)!}$$

Verzichtet man auf die Reihenfolge, so ergibt sich mit Satz 1.5

Satz 1.7
Die Anzahl der k-Teilmengen einer n-elementigen Menge ist $\frac{n!}{(n-k)! \cdot k!}$.

Bezeichnung

$$\binom{n}{k} := \frac{n!}{(n-k)! \cdot k!}$$

(sprich „n über k“ oder „k aus n“) heißt **Binomialkoeffizient**.

Es gelten direkt folgende Bezeichnungen.

Satz 1.8

a.) $\binom{n}{k} = \binom{n}{n-k}$

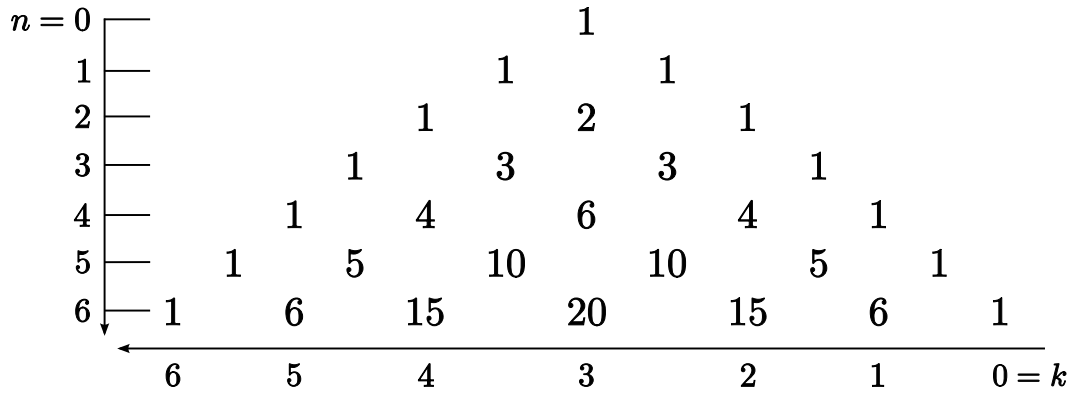
b.) $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$

Beweis:
a.) klar
b.) Eine k-elementige Teilmenge enthält entweder das n-te Element oder nicht. Im ersten Fall müssen aus $(n-1)$ Elementen noch $(k-1)$ ausgewählt werden, es gibt also $\binom{n-1}{k-1}$ Möglichkeiten, im zweiten Fall aus $(n-1)$ Elementen k, also $\binom{n-1}{k}$. Insgesamt also

$$\binom{n-1}{k} + \binom{n-1}{k-1} = \binom{n}{k}$$

Alternativ nachrechnen! □

1.5 Das Pascalsche Dreieck



Da eine Teilmenge einer n-elementigen Menge entweder $k = 0, 1, 2, 3, \ldots$ oder n Elemente enthält, ergibt sich mit Satz 1.2

Satz 1.9

Die Summe aller Binomialkoeffizienten „n über ...“ entspricht der Mächtigkeit der Potenzmenge einer n-elementigen Menge.

$$\binom{n}{0} + \binom{n}{1} + \ldots + \binom{n}{n-1} + \binom{n}{n} = 2^n$$

In Kurzschreibweise:

$$\boxed{\sum_{k=0}^{n} \binom{n}{k} = 2^n}$$

1.6 Exkurs: Grundbegriffe der Aussagenlogik

Definition:
Eine Aussage ist ein Ausdruck, der WAHR oder FALSCH sein kann.

Beispiel:
„Der Ball ist rund“.
„Das Papier ist weiß“.

Aussagentafeln und Verknüpfungen:

- $\wedge$ und
- $\vee$ oder
- $\neg$ Verneinung
- $A \Rightarrow B$ „aus A folgt B“
- $A \Leftrightarrow B$ „A ist äquivalent zu B“ $(\equiv (A \Rightarrow B) \wedge (B \Rightarrow A))$

A	B	$A \wedge B$	$A \vee B$	$\neg A$	$A \dot{\vee} B$	$A \Rightarrow B$	$\neg(A \Rightarrow B)$	$A \wedge \neg B$
0	0	0	0	1	0	1	0	0
0	1	0	1	1	1	1	0	0
1	0	0	1	0	1	0	1	1
1	1	1	1	0	0	1	0	0

2 Kombinatorische Werkzeuge

2.1 Vollständige Induktion

Behauptung
Es gilt eine Aussage $A(n)$ für alle $n \in \mathbb{N}$.

Beweis:

I.) Induktionsanfang beweisen!
$A(1)$ ist richtig.

II.) Induktionsschritt beweisen!
Wenn $A(n-1)$ richtig ist, dann auch $A(n)$ für alle $n \in \mathbb{N}$.

Aus I.) und II.) folgt die Behauptung, denn

I.) $\Longrightarrow$ $A(1)$ richtig
$\overset{II.)}{\Longrightarrow}$ $A(2)$ richtig
$\overset{II.)}{\Longrightarrow}$ $A(3)$ richtig
$\Longrightarrow$ ... usw. □

Satz 2.1 (Geometrische Summe)

$$\sum_{i=0}^{n} q^i = \frac{1-q^{n+1}}{1-q} \qquad \text{für } q \neq 1$$

Beweis:
$\underline{n=1:}$ Induktionsanfang

$$\sum_{i=0}^{1} q^i = 1+q$$
$$\frac{1-q^2}{1-q} = \frac{(1+q)\cdot(1-q)}{1-q} = 1+q$$

$\underline{(n-1) \to n}$:

$$\sum_{i=1}^{n} q^i = q^n + \sum_{i=1}^{n-1} q^i \overset{\text{Ind.ann.}}{=} q^n + \frac{1-q^n}{1-q}$$
$$= \frac{q^n \cdot (1-q) + 1 - q^n}{1-q} = \frac{q^n - q^{n+1} + 1 - q^n}{1-q} = \frac{1-q^{n+1}}{1-q}$$

□

Satz 2.2
$2^n > n^2 \qquad \forall\, n \in \mathbb{N},\ n \geq 5.$

Beweis:
$\underline{n = 5}$:

$$2^5 = 32 > 25 = 5^2$$

$\underline{(n-1) \to n}$:

$$2^n = 2 \cdot 2^{n-1} \overset{\text{Ind.ann.}}{>} 2 \cdot (n-1)^2 = 2n^2 - 4n + 1 = n^2 + n^2 - 4n + 1$$
$$\overset{n^2 > 4n \text{ (für } n>4)}{>} n^2 + 1 > n^2$$

□

Satz 2.3 (Bernoulli-Ungleichung)
Die Bernoullische Ungleichung ist eine recht einfache, aber wichtige Ungleichung um Potenzen nach unten abzuschätzen.
Für $h \in \mathbb{R}$, $h \geq -1$ und $n \in \mathbb{N}$ gilt

$$(1+h)^n \geq 1 + n \cdot h$$

Beweis:
$\underline{n = 1}$:

$$(1+h)^1 = 1 + h \geq 1 + 1 \cdot h$$

$\underline{(n-1) \to n}$:

$$(1+h)^n = (1+h)^{n-1} \cdot \underbrace{(1+h)}_{\geq 0} \overset{\text{Ind.ann.}}{\geq} (1 + (n-1) \cdot h) \cdot (1+h)$$
$$= 1 + h + n \cdot h - h + n \cdot h^2 - h^2$$
$$= 1 + n \cdot h + \underbrace{(n-1)h^2}_{\geq 0} \geq 1 + n \cdot h$$

□

2.2 Fibonacci-Zahlen

Motivation: Kaninchen-Beispiel von Fibonacci.
Ein Bauer züchtet Kaninchen. Jedes Kaninchen gebärt ein Junges jeden Monat, nachdem es 2 Monate alt ist.
Wir nehmen an, Kaninchen sterben nicht und wir ignorieren männliche Kaninchen. Wieviele Kaninchen wird der Bauer im n-ten Monat haben, wenn er mit einem beginnt?

$1, 1, 2, 3, 5, 8, \ldots$

Die Anzahl neuer Kaninchen ist die Anzahl der mindestens 2 Monate alten Kaninchen, allgemein

$$F_{n+1} = F_n + F_{n-1}.$$

Die Zahlen F_n werden Fibonacci-Zahlen genannt.

Satz 2.4

$$F_n = \frac{1}{\sqrt{5}} \cdot \left(\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right)$$

Die Zahl $\tau = \frac{1+\sqrt{5}}{2}$ heißt **goldener Schnitt**.

Beweis: (Induktion)
$\underline{n = 1}$:

$$\frac{1}{\sqrt{5}} \cdot \left(\left(\frac{1+\sqrt{5}}{2} \right)^1 - \left(\frac{1-\sqrt{5}}{2} \right)^1 \right) = 1$$

$\underline{n = 2}$

$$\frac{1}{\sqrt{5}} \cdot \left(\left(\frac{1+\sqrt{5}}{2} \right)^2 - \left(\frac{1-\sqrt{5}}{2} \right)^2 \right) = 1$$

$\underline{n \to (n+1)}$:

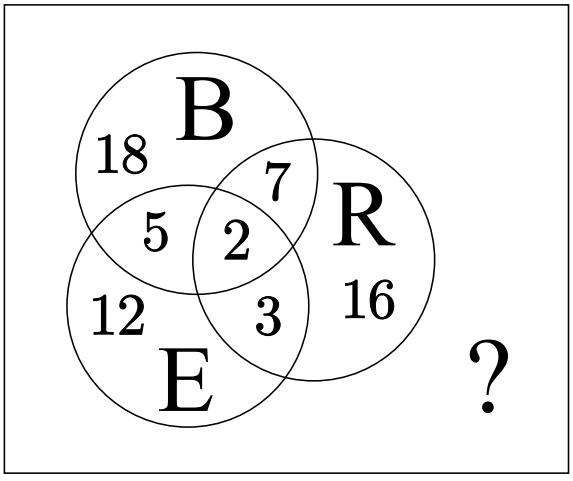
$$
\begin{aligned}
F_{n+1} = F_n + F_{n-1} &\overset{\text{Ind.ann.}}{=} \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right) \\
&\quad + \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{n-1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n-1}\right) \\
&= \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{n-1} \cdot \underbrace{\left(\frac{1+\sqrt{5}}{2}+1\right)}_{=\frac{1+2\cdot\sqrt{5}+5}{4}=\left(\frac{1+\sqrt{5}}{2}\right)^2} - \frac{1}{\sqrt{5}} \cdot \left(\frac{1-\sqrt{5}}{2}\right)^{n-1} \cdot \underbrace{\left(\frac{1-\sqrt{5}}{2}+1\right)}_{=\left(\frac{1-\sqrt{5}}{2}\right)^2} \\
&= \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+1}\right)
\end{aligned}
$$

□

2.3 Das Inklusions-Exklusionsprinzip

Beispiel:
Eine Schulklasse mit 40 Schülern mit unterschiedlichen Pop-Idolen: den Beatles (**B**), Rolling Stones (**R**) und Elvis Presley (**E**).

Die Pop-Idole sind gemäß folgender Grafik verteilt:



Wir wollen nun wissen, wie viele Schüler gar keine Pop-Idole haben.

	0-Eigenschaften		1-Eigenschaft		2-Eigenschaft		3-Eigenschaften
$40 =$	x	$+$	$(18+12+16)$	$-$	$(5+7+3)$	$+$	2
$40 \equiv$	$\binom{3}{0}$	$+$	$\binom{3}{1}$	$-$	$\binom{3}{2}$	$+$	$\binom{3}{3}$
$40 =$	x	$+$	46	$-$	15	$+$	2

$\implies x = 7$

Dieses Prinzip, das von einer Gesamtmenge die 1-elementigen (Elemente bezogen auf ihre Eigenschaften) Mengen abzieht, dann die 2-elementigen addiert, usw., nennt man **Inklusions-Exklusions-Prinzip**.

Satz 2.5

$$\sum_{k=0}^{n} \binom{n}{k} = 2^n$$

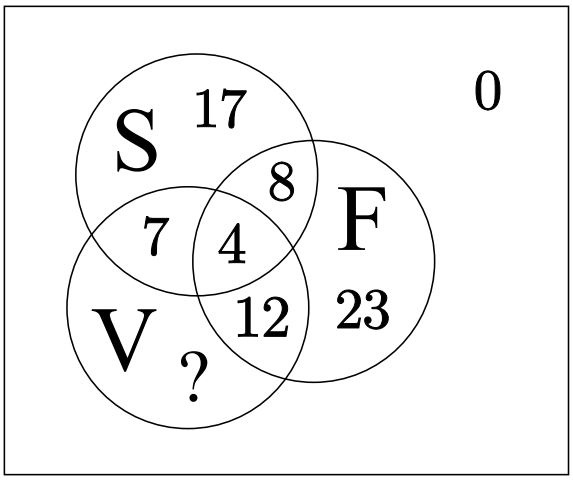
$$\boxed{\underbrace{\sum_{k=0}^{n} \binom{n}{k}}_{k \text{ gerade}} = \underbrace{\sum_{k=0}^{n} \binom{n}{k}}_{k \text{ ungerade}}}$$

Beweis:
Wir erinnern an die Darstellung von Teilmengen als Binärzahlen.

$$\{a, b, c\}$$
$$101 = \{a, c\}$$
$$000 = \emptyset$$

Jede k-elementige Teilmenge entspricht einem String der Länge n mit k Einsen. Da es genauso viele Strings mit einer geraden Anzahl von Einsen gibt wie mit einer ungeraden Anzahl, folgt die Behauptung. $\square$

Beispiel:
Eine Schulklasse mit 40 Mädchen, die gerne Schach (**S**), Fußball (**F**) und Volleyball (**V**) spielen. Die Interessen der 40 Mädchen sind gemäß folgender Grafik verteilt:



Wir interessieren uns für die Anzahl an Schülerinnen, die nur Volleyball spielen.

$$\begin{array}{rccccccc} & \binom{3}{0} & + & \binom{3}{1} & - & \binom{3}{2} & + & \binom{3}{3} \\ 40 = & 0 & + & (V+17+23) & - & (8+7+12) & + & 4 \\ 40 = & V & + & 40 & - & 27 & + & 4 \end{array}$$

$\Longrightarrow V = 23$

2.4 Das Taubenschlag-Prinzip (engl. pigeon-hole principle oder auch Schubfachprinzip)

Beobachtung 2.6
Wenn n Objekte auf k Schachteln verteilt werden sollen mit $k < n$, dann gibt es eine Schachtel mit mindestens 2 Objekten.

Beispiel:

1.) Mindestens 2 Hessen haben gleich viele Haare auf dem Kopf

2.) 50 Schuss auf ein 70cm $\times$ 70cm Quadrat. Dann gibt es 2 Treffer, die höchstens 15cm Abstand haben.

3.) Betrachte Menge A mit $n+1$ Zahlen aus $\{1, 2, \ldots, 2n\}$. Dann gibt es 2 Zahlen, von denen die eine die andere teilt:

Beispiel:
$\underline{n=7} \quad \{1,2,3,\ldots,14\}$
$2,3,5,7,11,13,?$

Beweis:
Jedes $a \in A$ schreibe in der Form

$$a = 2^k \cdot m \quad k \in \mathbb{N}_0 \text{ und } m \text{ ungerade}$$

$2 = 2^1 \cdot 1 \quad 7 = 2^0 \cdot 7.$
In $\{1,2,\ldots,2n\}$ stecken n ungerade Zahlen.
$\overset{\text{Beobachtung 2.6}}{\Longrightarrow}$ Es muss 2 Zahlen aus A geben mit demselben m. Eine von diesen teilt die andere. □

4.) Betrachte n Zahlen $a_1,\ldots,a_n$. Dann gibt es eine Reihe aufeinanderfolgende Zahlen $a_{k+1}, a_{k+2}\ldots,a_l$ mit $l > k$, deren Summe $\sum_{i=k+1}^{l} a_i$ durch n teilbar ist.
$\underline{n=3} \quad 2,2,?$
$\underline{n=4} \quad 2,1,2,?$

Beweis:
$N = \{0, a_1, a_1+a_2, a_1+a_2+a_3, \ldots, a_1+a_2+\ldots+a_n\}$
$|N| = n+1$
Jedes $a \in N$ teile durch n mit Rest $\in R = \{0,1,2,\ldots,n-1\}$
$|R| = n$
$\overset{\text{Beobachtung 2.6}}{\Longrightarrow}$ es gibt 2 Elemente aus H mit demselben Rest

$$a_1+\ldots+a_k,\ a_1+a_2+\ldots+a_l \quad k<l$$

$$\Longrightarrow \quad \underbrace{\sum_{i=1}^{l} a_i - \sum_{i=1}^{k} a_i}_{\sum_{i=k+1}^{l} a_i \text{ ist durch } n \text{ teilbar}} \qquad \text{hat den Rest 0} \qquad \square$$

5.) Geburtstagswette
„Von 50 Menschen haben 2 am gleichen Tag Geburtstag“.

Beweis:
Ich verliere, wenn alle an verschiedenen Tagen Geburtstag haben. Die Wahrscheinlichkeit dafür ist

$$\frac{365 \cdot 364 \cdot 363 \cdot \ldots \cdot 316}{365 \cdot 365 \cdot \ldots \cdot 365} = \frac{\%}{365^{50}} \approx 0,0296\%$$

23 hätten gereicht um eine Gewinnwahrscheinlichkeit $> 50\%$ zu haben. □

2.5 Binomialkoeffizienten

Satz 2.7 (Binomialsatz)
Der binomische Lehrsatz ist ein Satz der Mathematik, der es in seiner einfachsten Form ermöglicht, die Potenzen eines Binoms $x + y$, also einen Ausdruck der Form $(x + y)^n$ mit $n \in \mathbb{N}$, als Polynom n-ten Grades in den Variablen x und y auszudrücken:

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} \cdot x^{n-k} \cdot y^k \qquad x, y \in \mathbb{R},\ n \in \mathbb{N}.$$

Beispiel:
$n = 2$

$$\begin{aligned}(x+y)^2 &= x^2 + 2xy + y^2 \\ &= \binom{2}{0} x^0 y^2 + \binom{2}{1} x^1 y^1 + \binom{2}{2} x^2 y^0 \\ &= y^2 + 2xy + x^2\end{aligned}$$

Beweisskizze:

$$\begin{aligned}(x+y)^n &= \underbrace{(x+y) \cdot (x+y) \cdot \ldots \cdot (x+y)}_{n-\text{mal}} \\ &= \sum_{k=0}^{n} \binom{n}{k} x^k \cdot y^{n-k}\end{aligned}$$

Korollar 2.8 (Satz 1.9)
Setzt man beim Binomialsatz $x = y = 1$, so erhält man die selbe Aussage, wie bereits in Satz 1.9 genannt.

$$x = y = 1 \Longrightarrow 2^n = \sum_{k=0}^{n} \binom{n}{k}$$

Korollar 2.9 (Satz 2.5)
Setzt man dagegen $x = -1$, $y = 1$ so erhält man die Summer der alternierenden Binomialkoeffizienten. Diese Formel folgt aus der Symmetrie des Binomialkoeffizienten.

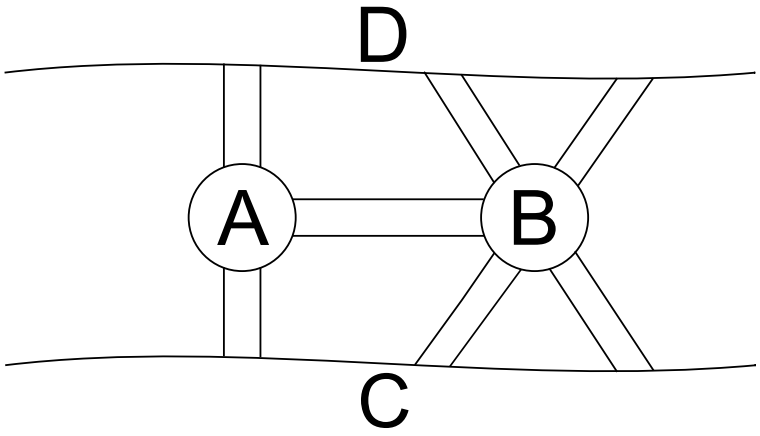
$$x = -1,\ y = 1 \Longrightarrow 0 = \sum_{k=0}^{n} (-1)^k \cdot \binom{n}{k}$$

3 Graphen-Grundlagen

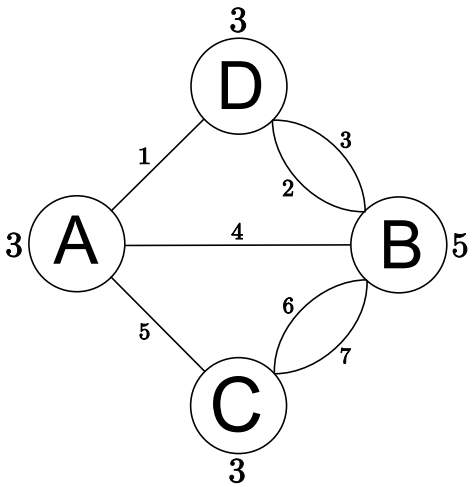
3.1 Grundlagen

Beispiel:

1.) Königsberger Brückenproblem (Euler, 1736)



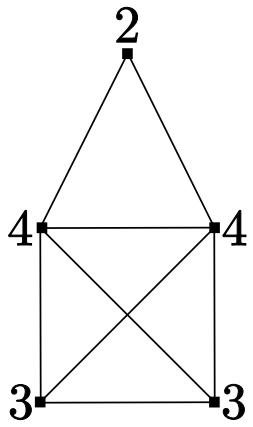
Frage: Gibt es eine Rundreise durch Königsberg, die jede Brücke genau einmal benutzt?



abstrahierte Darstellung des Brückenproblems

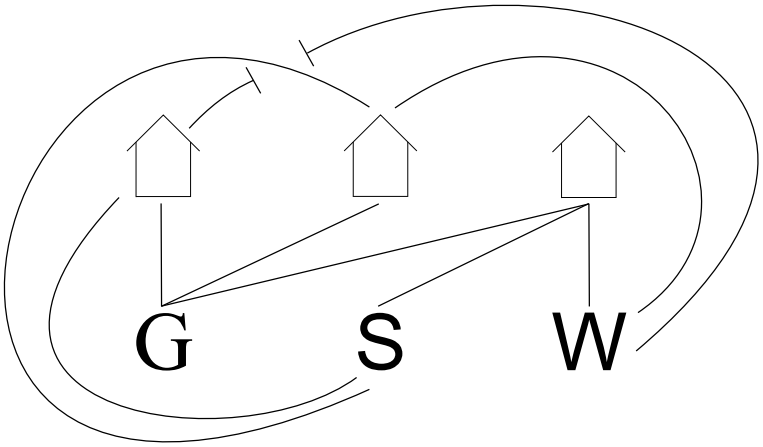
Nein, da mehr als 2 Knoten ungeraden Grad haben.

2.) Das Haus vom Nikolaus



Kann man das Haus vom Nikolaus zeichnen ohne den Stift abzusetzen?

3.) Ein Grundversorgungsproblem

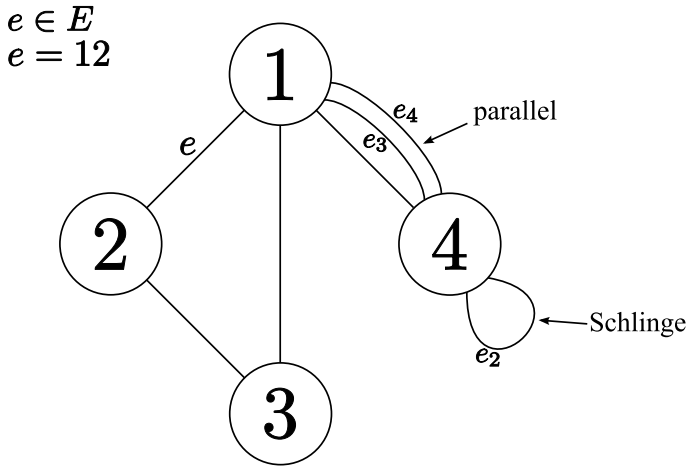


Verbinde alle 3 Häuser mit Gas, Wasser und Strom, ohne dass sich 2 Verbindungen kreuzen.

Definition 3.1
Ein ungerichteter Graph ist ein Tripel (V, E, Ψ) mit einer nicht leeren Menge V, den **Knoten**, einer Menge E, den **Kanten**, und einer **Inzidenzfunktion** $\Psi : E \to V \times V$. $V \times V$ bezeichnet die Menge der ungeordneten Paare von Elementen aus V. Die Funktion ψ weißt also jeder Kante e ein Paar von Knoten u und v zu durch $\psi(e) = uv = vu$.

Beispiel:
$V = \{A, B, C, D\}$, $E = \{1, 2, \ldots, 7\}$
$\Psi(1) = \{A, D\} = AD = [A, D]$
$\Psi(7) = \{B, C\} = BC = [B, C]$
$G = (V, E)$

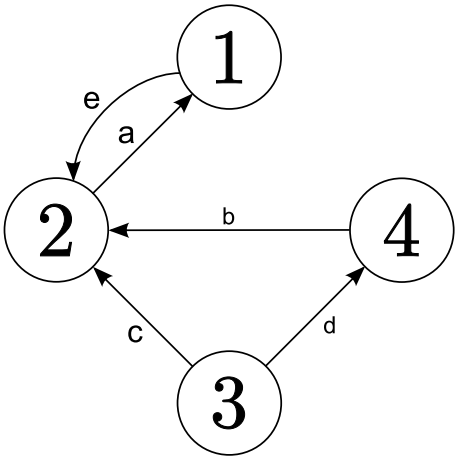
G heißt **endlich**, falls $|V|, |E| < \infty$, andernfalls heißt G **unendlich**.



1 und 2 liegen auf beziehungsweise sind **inzident** zu e.
e verbindet 1 und 2, die Knoten 1 und 2 sind Nachbarn beziehungsweise **adjazent**.
Zwei Kanten, die den selben Endknoten haben, weden sowohl **inzident** als auch **adjazent** genannt.
Die Kante e_2 mit $\Psi(e_2) = 44$ heißt Schlinge und die Kanten $e_3, e_4 \in E$ mit $\Psi(e_3) = \Psi(e_4)$ heißen parallel. Ein Graph ohne Schlingen und parallele Kanten heißt **einfach**.

Ein gerichteter Graph $D = (V, A, \Psi)$ ist ein Tripel bestehend aus einer Menge $V \neq \emptyset$, einer Menge A von **Bögen** und einer Inzidenzfunktion $\Psi : A \to V \times V$.

Beispiel:



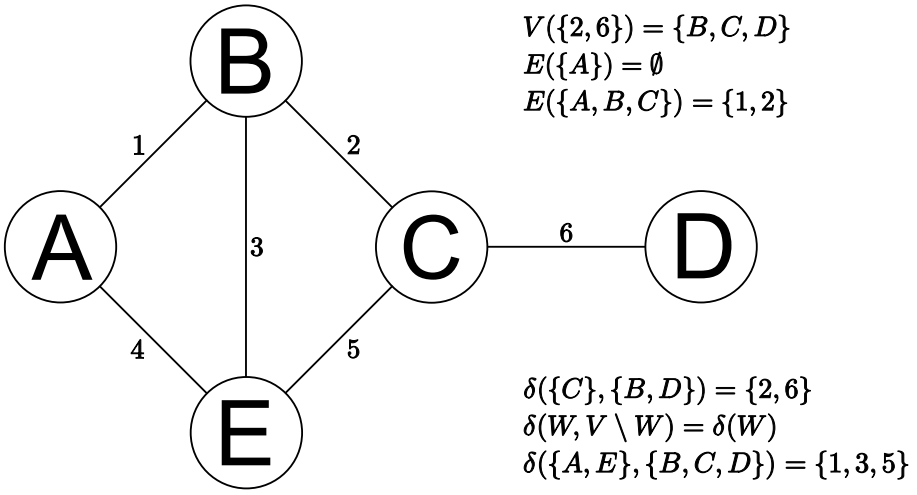
$\Psi(a) = (2, 1)$
$\Psi(e) = (1, 2)$

$\Psi(a) \neq \Psi(e)$
$\Psi(a) = \Psi(e)$ sind gerichtet und $\{2, 1\}$ und $\{1, 2\}$ sind ungerichtet.

A arc (engl.) Bogen
E edge (engl.) Kante
V vertex (engl.) Knoten

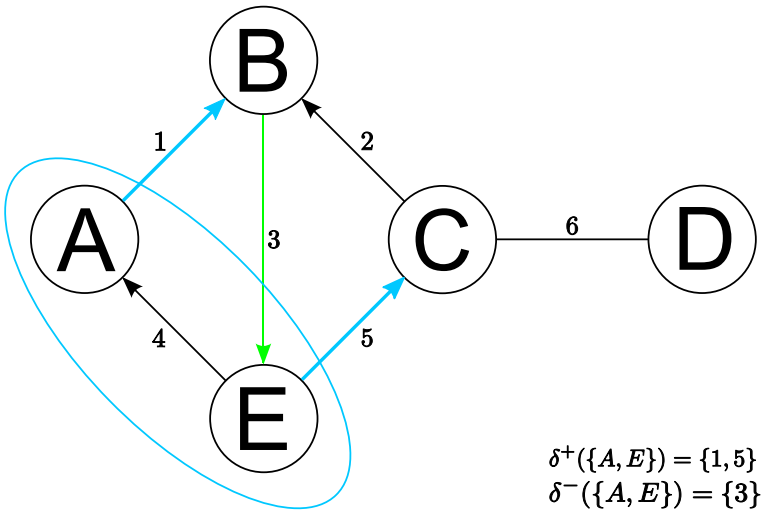
Für eine Kantenmenge $F \subseteq E$ bezeichnen wir $V(F)$ als die Menge aller Knoten, die zu einer Kante $f \in F$ inzident sind.
Für eine Knotenmenge $W \subseteq V$ bezeichnen wir $E(W)$ als die Menge aller Kanten mit beiden Endknoten in W.

Beispiel:



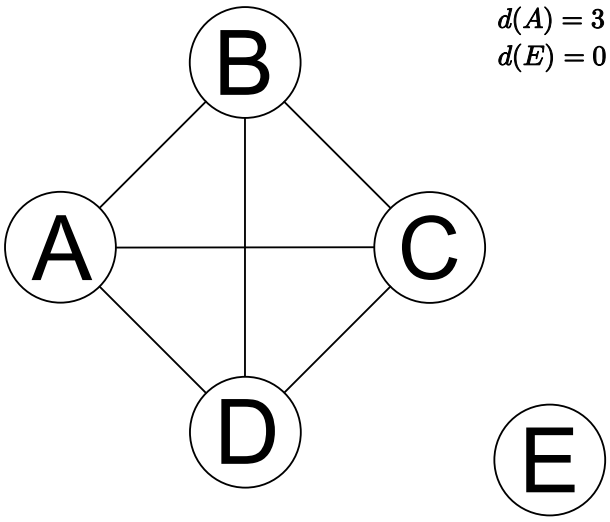
Für 2 Kantenmengen $U, W \subseteq V$ bezeichnen wir mit $[U : W]$ beziehungsweise $\delta(U, W)$ die Menge der Kanten mit einem Endknoten in U und einem Endknoten in W. Anstelle von $\delta(W, V \setminus W) = [W : V \setminus W]$ schreibt man auch kurz $\delta(W)$.
Eine Kantenmenge $F \subseteq E$ wird als **Schnitt** bezeichnet, wenn es eine Knotenmenge $W \subseteq V$ mit $\delta(W) = F$ gibt. $\delta(W)$ wird auch als **von W induzierter Schnitt** bezeichnet.
Im Falle eines gerichteten Graphen $D = (V, A)$ bezeichne $\delta^+(W) := \{(i, j) \in A : i \in W, j \in V \setminus W\}$ die Menge der ausgehenden Bögen und $\delta^-(W) := \{(i, j) \in A : i \in V \setminus W, j \in W\}$ die Menge der eingehenden Bögen.

Beispiel:



$d(v) = |\delta(\{v\})|$ bezeichnet den **Grad** eines Knotens v mit $v \in V$. Ist $d(0) = 0$, so nennen wir v einen **isolierten Knoten**. Ein Knoten v heißt **gerade** bzw. **ungerade**, falls $d(v)$ gerade bzw. ungerade ist.

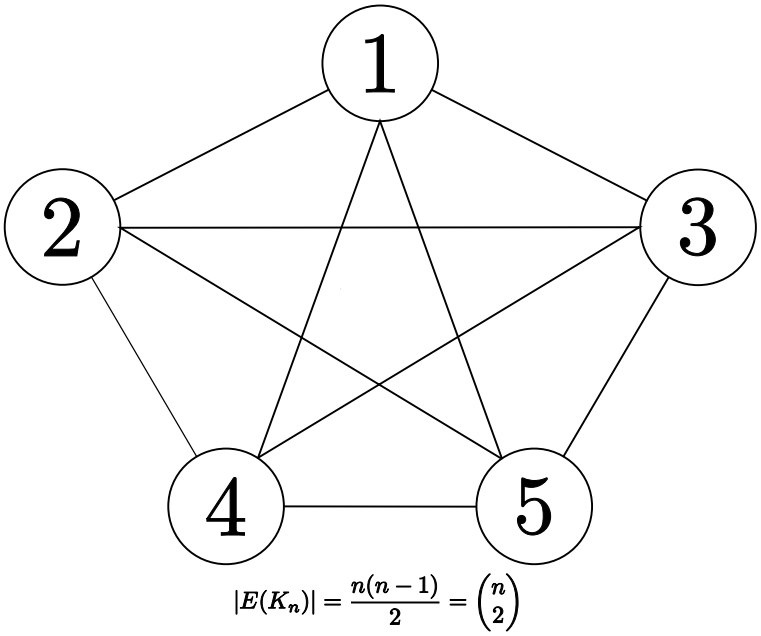
Beispiel:



$G = (V, E)$ heißt **vollständig**, falls jeder Knoten mit jedem anderen verbunden ist. Manchmal bezeichnen wir den vollständigen Graphen mit K_n, wenn n die Kardinalität der Knotenmenge ist.

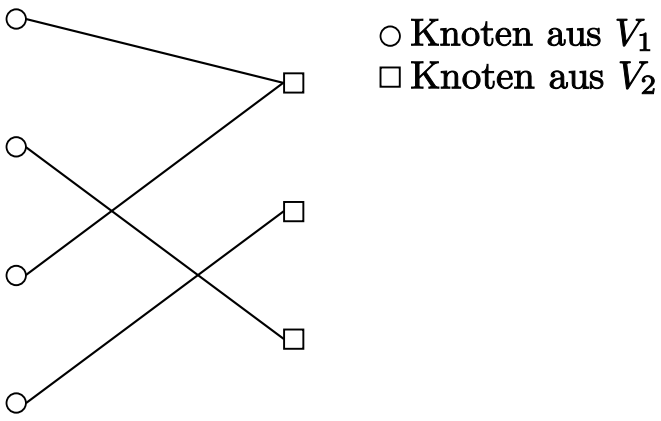
Sprechen wir von einem vollständigen Graphen G, so gehen wir immer davon aus, dass G vollständig ist.

Beispiel:



Ein Graph, dessen Knotenmenge V in zwei disjunkte, nicht-leere Teilmengen V_1, V_2 mit $V_1 \cup V_2 = V$ eingeteilt werden kann, so dass keine zwei Knoten in V_1 und keine zwei Knoten in V_2 benachbart sind, heißt **bipartit**.

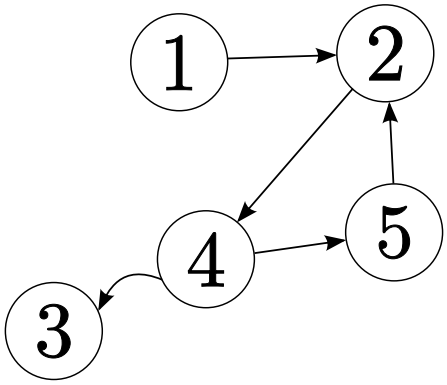
Beispiel:



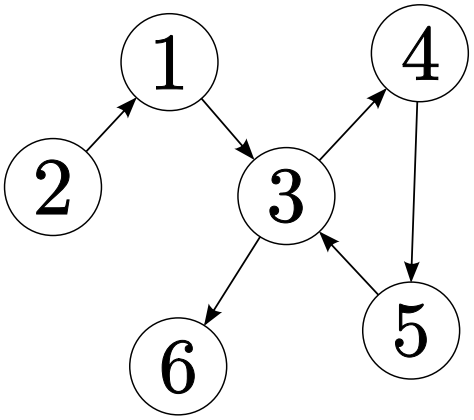
Falls $uv \in E$ für alle $u \in V_1$, $v \in V_2$ so spricht man von einem **vollständig bipartiten** Graphen. Diese eindeutig bestimmten Graphen werden manchmal auch einfach nur als $K_{m,n}$ bezeichnet, wobei $m = |V_1|$ sei, und sei $n = |V_2|$.

Eine endliche Folge $K = (v_0, e_1, v_1, e_2, v_2, \ldots, e_k, v_k)$, $k \geq 0$ heißt **Kette**, falls die Folge mit einem Knoten beginnt und endet und jede Kante e_i mit den Knoten v_{i-1} und v_i inzidiert. Der Knoten v_0 heißt **Anfangsknoten**, der Knoten v_k **Endknoten** und die Knoten $v_1, \ldots v_{k-1}$ **innere Knoten**.
Eine Kette, in der alle Knoten voneinander verschieden sind, heißt **Weg**. Beachte, dass in einem Weg auch alle Kanten voneinander verschieden sind.
Sind in einer Kette nur alle Kanten verschieden, so sprechen wir von einem **Pfad**.

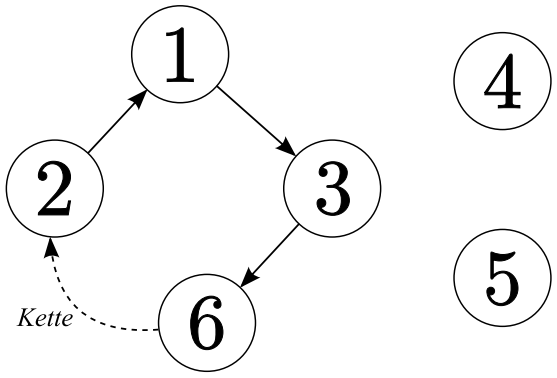
Beispiel:



Kette, kein Pfad, kein Weg



Kette, Pfad, kein Weg



Kette, Pfad, Weg

Ein Pfad, der jede Kante genau einmal enthält, heißt **Eulerpfad**.

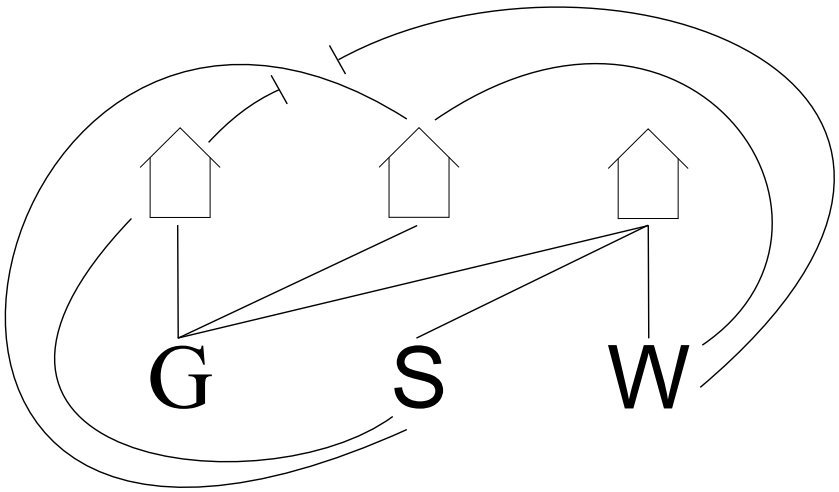
Ist der Pfad geschlossen so heißt er **Eulertour**. Ein Graph, der eine Eulertour enthält, heißt **eulersch**.

Bemerkung:
Es ist nicht schwer zu sehen, dass ein Graph genau dann eulersch ist, wenn jeder Knoten einen geraden Grad hat, siehe Übung.

Ein Pfad, der jeden Knoten genau einmal enthält, heißt **Hamiltonweg**.
Ist der Pfad geschlossen, dann heißt er **Hamiltontour** oder auch **Hamiltonkreis** genannt. Ein Graph, der einen Hamiltonkreis enthält, heißt **hamiltonsch**.

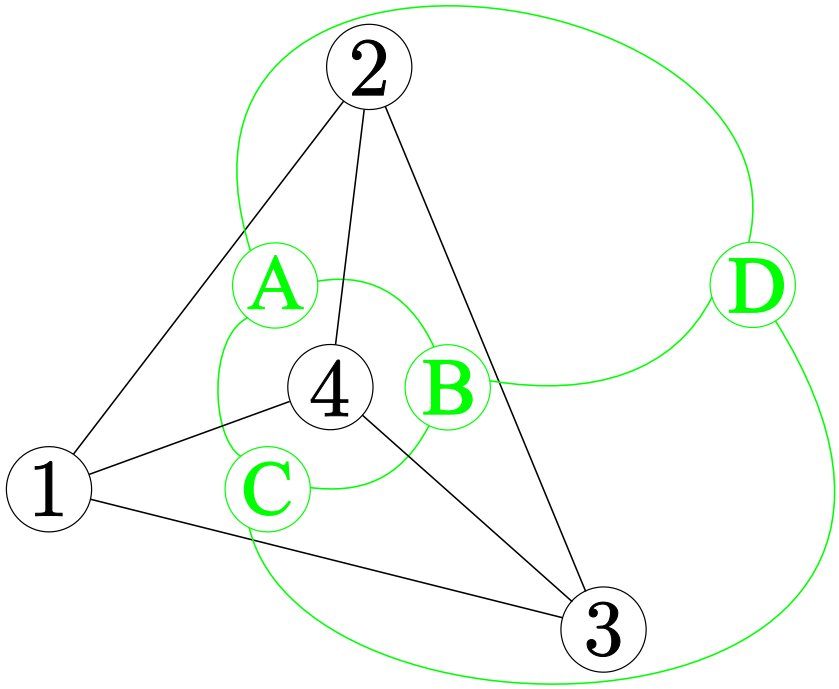
Ein Graph, der so in der Ebene zeichenbar ist, dass sich keine 2 Kanten kreuzen, nennt man **planar**.

Ist $K_{3,3}$ planar?



Ein planarer Graph teilt die Ebene in eine Menge von Flächen $\mathcal{F}_G$ ein. Diejenige Fläche, die den gesamten Graphen umschließt, heißt **äußere Fläche**. Zu jeder Fläche $F \in \mathcal{F}_G$, die nicht die äußere Fläche ist, gibt es genau einen Kreis $C_F \subseteq E$, der diese Fläche umschließt. Ein solcher Kreis heißt **Flächenkreis**. Ebenso gibt es eine Kantenmenge in G, die die äußere Fläche umschließt. Führen wir einen Knoten für jede Fläche aus $\mathcal{F}_G$ ein und Kanten zwischen Knoten, deren zugehörige Flächen benachbart sind (genauer gesagt, deren zugehörige Flächenkreise eine gemeinsame Kante haben), so erhalten wir einen neuen Graphen, den sogenannten **dualen Graphen** von G (in Zeichen $G^* = (V^*, E)$). Beachte, dass jede Kante im dualen Graphen genau einer Kante in G entspricht, so dass wir für beide Graphen die gleiche Kantenmenge E verwenden.

Beispiel:

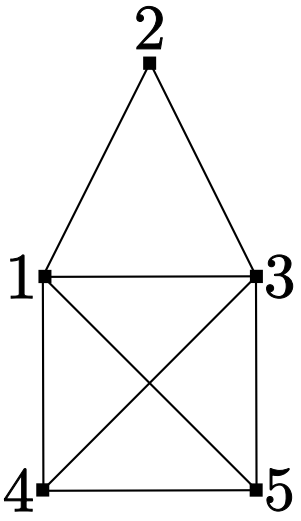


dualer Graph

3.2 Exkurs: Algorithmen

Im Laufe der Vorlesung wollen wir unterscheiden zwischen einfachen und schweren Problemen, „guten“ und „schlechten“ Algorithmen. Um mit der Komplexität der Lösung von Problemen umgehen zu können, müssen wir formalisieren, was wir unter einem Problem oder Algorithmus verstehen.
Ein **Problem** ist eine Fragestellung mit offenen Parametern und einer Spezifikation, wie eine Lösung aussieht.

Beispiel:
$G = (V, E)$, enthält G eine Eulertour?



Lösung: $4, 1, 3, 2, 1, 5, 4, 3, 5$
$V = \{1, 2, 3, 4, 5\}$
$E = \{12, 13, 14, \dots\}$

Wenn alle Parameter spezifiziert sind, so spricht man von einem **Problembeispiel**. Die „Größe“ eines Problembeispiels wird angegeben durch Verwendung eines Kodierungsschemas. Wir verwenden die Binärkodierung. Mit $\langle n \rangle$ bezeichnen wir die Kodierungslänge (Größe) einer Zahl $n \in \mathbb{Z}$

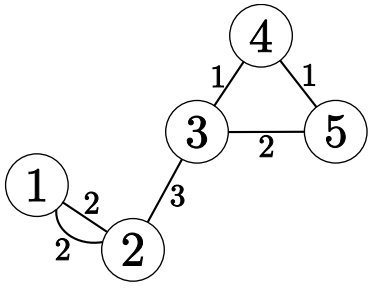
$$\langle n \rangle := \lceil \log_2(|n| + 1) \rceil + 1$$

$$\langle r \rangle := \langle p \rangle + \langle q \rangle \quad r = \frac{p}{q} \in \mathbb{Q} \text{ teilerfremd}$$

Die Kodierungslänge eines Graphen $G = (V, E)$ mit Kantengewichten $c_e \in E$ ist definiert durch:

$$\langle G \rangle = |V| + |E| + \sum_{e \in E} \langle c_e \rangle$$

Beispiel:



Graph G

$$\begin{aligned}\langle G\rangle &= |V| + |E| + \sum_{e\in E}\langle c_e\rangle \\ &= 5 + 6 + 2\cdot(\lceil\log_2(|1|+1)\rceil + 1) + 3\cdot(\lceil\log_2(|2|+1)\rceil + 1) \\ &\quad + 1\cdot(\lceil\log_2(|3|+1)\rceil + 1) \\ &= 5 + 6 + 2\cdot 2 + 3\cdot 3 + 1\cdot 3 \\ &= 27\end{aligned}$$

Die Kodierungslänge des Graphen G ist also $\langle G\rangle = 27$.

Ein **Algorithmus** ist eine Anleitung zur schrittweisen Lösung eines Problems. Wir sagen, ein Algorithmus A **löst** ein Problem Π, falls A für alle Problembeispiele $I \in \Pi$, eine Lösung in einer **endlichen** Anzahl an Schritten findet.
Ein **Schritt** ist eine **elementare Operation**:
Addieren, Subtrahieren, Vergleichen, $\left[\begin{array}{c}\text{Multiplikation}\\ \text{Division}\end{array}\right]$
Multiplikation und Division können hinzugenommen werden, wenn garantiert werden kann, dass die Größe der Zahlen, die auftreten, polynomial in der Größe des Inputs bleiben. Die **Laufzeit** eines Algorithmus ist die Anzahl der Schritte, die notwendig sind zur Lösung des Problembeispiels.
Sei $T(l)$ die Laufzeit eines Algorithmus A zur Lösung eines Problembeispiels mit Kodierungslänge höchstens $l \in \mathbb{N}$.

Ein Algorithmus A läuft in **Polynomialzeit**, falls es ein Polynom p gibt mit

$$\boxed{T(l) \leq p(l)} \qquad \forall\, l \in \mathbb{N}.$$

Die Menge der Probleme, die in Polynomialzeit lösbar sind, bezeichnen wir mit $\mathcal{P}$.

Definition 3.2 (Größenordnung von Funktionen)
Sei $M = \{f \mid f : \mathbb{N} \to \mathbb{R}\}$ die Menge der reellwertigen Funktionen (z.B. $T \in M$).
Für $g \in M$ sei

$$\begin{aligned}\mathcal{O}(g) &= \{f \in M \mid \exists\, c, n_0 \in \mathbb{N} : f(n) \leq c\cdot g(n),\ \forall\, n \geq n_0\} \\ \otimes(g) &= \{f \in M \mid \exists\, c, n_0 \in \mathbb{N} : f(n) \geq c\cdot g(n),\ \forall\, n \geq n_0\} \\ \Theta(g) &= \mathcal{O}(g) \cap \otimes(g)\end{aligned}$$

Bemerkung:

Notation	anschaulische Bedeutung
$f \in \mathcal{O}(g)$	f wächst höchstens so schnell wie g
$f \in \otimes(g)$	f wächst mindestens so schnell wie g
$f \in \Theta(g)$	f wächst genauso schnell wie g

Polynomiale Algorithmen vom Grad k haben stets einen Laufzeitaufwand von $\mathcal{O}(n^k)$, exponentielle Algorithmen den Laufzeitaufwand $\mathcal{O}(2^n)$, $\mathcal{O}(n!)$ oder $\mathcal{O}(n^n)$.

Beispiel:
Unsere Funktion sei gegeben durch $f(n) = 3n^2 + n + 1$. Da es sich hierbei um eine quadratische Funktion handelt, also Grad $k = 2$, gilt $f(n) \in \mathcal{O}(n^2)$, was wir aber noch zeigen müssen.

Um dies zu zeigen nutzen wir die gelernte Definition:

$$\mathcal{O}(g) = \{f \in M \mid \exists\, c, n_0 \in \mathbb{N} : f(n) \leq c \cdot g(n), \forall\, n \geq n_0\}$$

Im Prinzip bedeutet das eigentlich nur, dass es eine Funktion g gibt, die schneller oder zumindest gleich schnell wächst wie f. Dazu multiplizieren wir auch noch eine Konstante c, die wir für den Beweis frei wählen können. Das tun wir jetzt mal:

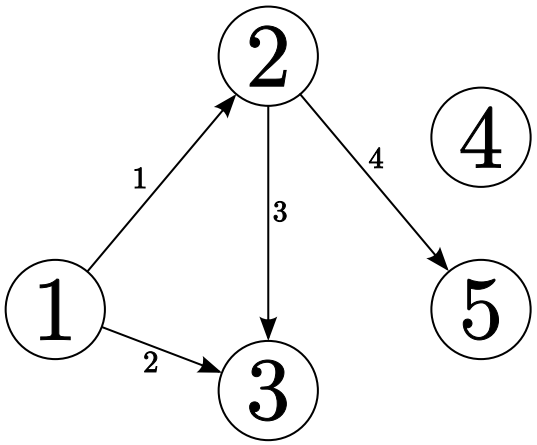
$$\begin{aligned} f(n) &\leq c \cdot g(n) \\ 3n^2 + n + 1 &\leq c \cdot n^2 \end{aligned}$$

Jetzt müssen wir ein n_0 und c finden, für die die Gleichung gilt. Da $n \geq n_0$ gilt, muss es auch für größere n gelten, damit der Beweis abgeschlossen ist.
Als erstes dividieren wir durch n^2, das ergibt dann:

$$3 + \frac{1}{n} + \frac{1}{n^2} \leq c$$

Nun können wir uns ein n_0 wählen, wir wählen $n_0 = 1$. In die Gleichung für n eingesetzt ergibt dann, dass $c \geq 5$ gewählt werden muss, wir wählen also $c = 5$. Offensichtlich gilt für $n_0 = 1$ und $c = 5$, dass die Gleichung für alle $n \geq n_0$ erfüllt ist. Der aufmerksame Leser möge dies durch vollständige Induktion validieren.
Somit ist also bewiesen, dass $f(n) \in \mathcal{O}(n^2)$ gilt, und wir sind fertig.

3.3 Speicherung von Graphen



Beispielgraph

(1) Kanten-Bogen-Liste:
Ist $G = (V, E)$. Die Bogenliste wäre dann:

$n, m, a_1, e_1, a_2, e_2, \ldots, a_m, e_m$ wobei a_i der Anfangsknoten von Kante i sei und e_i der Endknoten der Kante i.

Vorteil: Effiziente Speicherung $\mathcal{O}(m)$
Nachteil: Schlechter Zugriff $\mathcal{O}(m)$

In unserem *Beispielgraph* wäre die Kanten-Bogen-Liste wie folgt:

$5, 4, 1, 2, 1, 3, 2, 3, 2, 5$

(2) Adjazenzmatrizen:
Die (symmetrische) $n \times n$-Matrix $A = (a_{ij})$, wobei a_{ij} die Anzahl der Kanten sei, die i und j verbinden ($i \neq j$), und a_{ii} 2-mal die Anzahl der Schleifen an Knoten i, heißt **Adjazenzmatrix** von G.

Vorteil: Zugriff $\mathcal{O}(1)$
Nachteil: Speicher $\mathcal{O}(n^2)$

Für unseren *Beispielgraph* wäre die Adjazenzmatrix die folgende:

$$\begin{array}{c} \\ \mathbf{1} \\ \mathbf{2} \\ \mathbf{3} \\ \mathbf{4} \\ \mathbf{5} \end{array} \begin{array}{c} \begin{array}{ccccc} \mathbf{1} & \mathbf{2} & \mathbf{3} & \mathbf{4} & \mathbf{5} \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

(3) Adjazenzlisten:

- Anzahl Knoten
- Anzahl Kanten
- Für jeden Knoten seinen Grad und die Liste seiner inzidenten Kanten/Bögen oder seiner benachbarten Knoten (eventuell mit Kosten oder weitere Varianten)

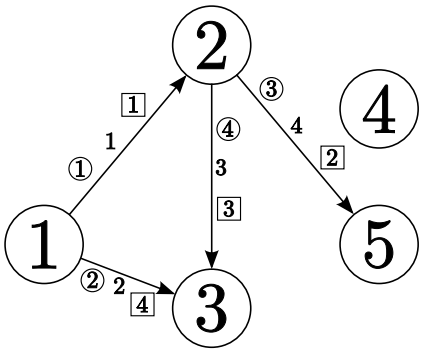
Speicheraufwand: $\mathcal{O}(n + m)$
Zugriff: $\mathcal{O}(n)$

Für den obigen *Beispielgraph* wäre die Adjazenzliste die folgende:

5
4
1,2,1,2,1,3
2,3,1,2,2,3,2,5
3,2,1,3,2,3
4,0
5,1,2,5

3.4 Suchalgorithmen auf Graphen

Gibt es einen Weg von 1 nach 4?



***Auswertungsreihenfolge*:**
◯ Breadth First Search (BFS)
☐ Depth First Search (DFS)

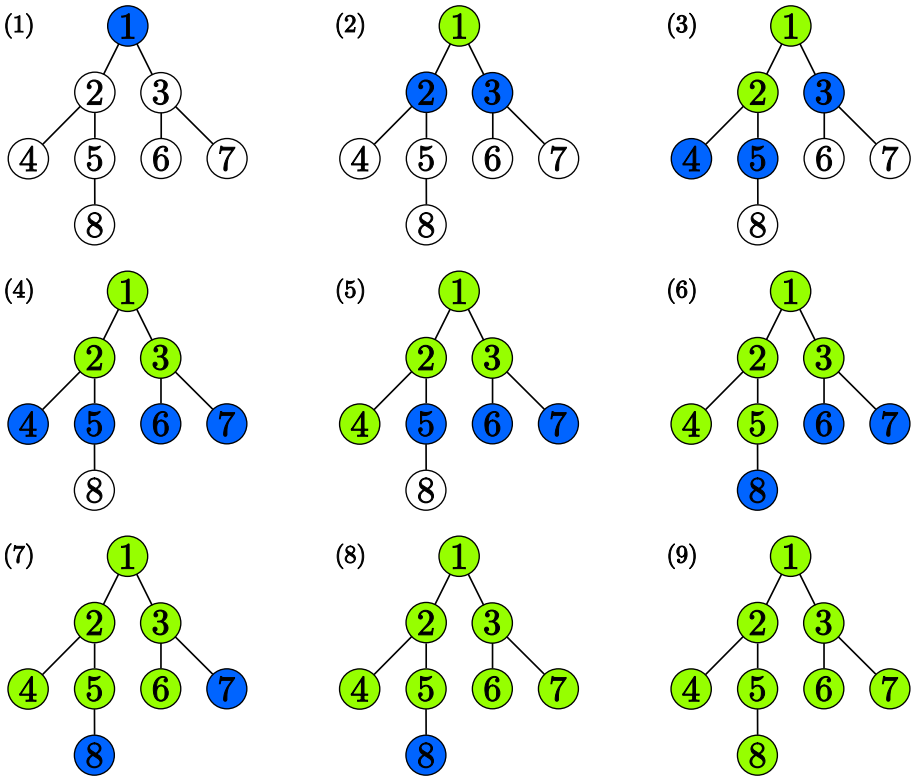
Algorithmus 3.3 (Breadth-First-Search Algorithmus)
Breadth-First-Search arbeitet nach dem FIFO-Prinzip (First-In-First-Out).

Input: Graph $G = (V, E)$, Liste L
Output: Anzahl der Zusammenhangskomponenten und die Menge der Knoten jeder Zusammenhangskomponente

1. Initialisierung:
   MARK(v) := -1 für alle $v \in V$
   NEXT := 1
2. For $v \in V$
   a) Füge v ans Ende der Liste L an
   b) While $L \neq \emptyset$
      i. Wähle v vom Anfang der Liste und entferne v aus L
      ii. If MARK(v) < 0 Then
         A. MARK(v) := NEXT
      iii. End If
      iv. For $w \in \delta(v)$ Do
         A. If MARK(w) < 0 Then
            - MARK(w) := NEXT
            - Füge w am Ende der Liste an
         B. End If
      v. End For
   c) End While
   d) $NEXT = NEXT + 1$
3. End For

$NEXT$ gibt die Anzahl der Zusammenhangskomponenten an. Die Knoten einer Zusammenhangskomponente sind mit demselben Wert markiert.

Beispiel:



Zuallererst wählen wir einen Startknoten. Wir wählen Knoten 1, in Grafik (1) blau markiert. In Grafik (2) wurde dieser nun markiert, daher grün. Die Nachbarknoten von Knoten 1 sind blau markiert Von diesen beiden Knoten wählen wir einen aus; wir wählen Knoten 2. Die Nachbarknoten von Knoten 2 sind ebenfalls blau markiert. Da aber Knoten 1 noch einen uns unbekannten Nachbar hat, nämlich Knoten 3, nehmen wir erstmal diesen in die Liste auf und merken uns schonmal seine Nachbarn vor, indem wir sie blau markieren. Nun gehen wir zurück zu unserem ersten gefunden Nachbar von Knoten 1, dem Knoten 2 und und schauen, welche uns unbekannten Nachbarn dieser hat. Der erste Unbekannte ist Knoten 4, welcher dann in die Liste aufgenommen wird. Anschließend folgt Knoten 5. Da Knoten 5 auch noch einen unbekannten Nachbar hat, wird dieser vorgemerkt und blau markiert. Damit hat Knoten 2 keine uns unbekannten Nachbarn mehr und wir kehren zurück zu Knoten 3. Knoten 3 hat als ersten Nachbar den Knoten 6, dieser ist uns nicht bekannt und wird daher in die Liste aufgenommen. Der nächste wäre Knoten 7, auch dieser ist unbekannt und wird in die Liste aufgenommen. Nun sind alle Nachbarn von Knoten 3 bekannt und wir kehren zum letzten Knoten zurück, der noch unbekannte Nachbarn hatte, das war Knoten 5. Der unbekannte Nachbar von Knoten 5 ist Knoten 8, welcher dann auch direkt in die Liste mit aufgenommen wird. Wie man in Grafik (9) erkennen kann, sind nun alle Knoten grün markiert und uns somit bekannt.

Algorithmus 3.4 (Depth-First-Search Algorithmus)
Depth-First-Search arbeitet nach dem LIFO-Prinzip (Last-In-First-Out).

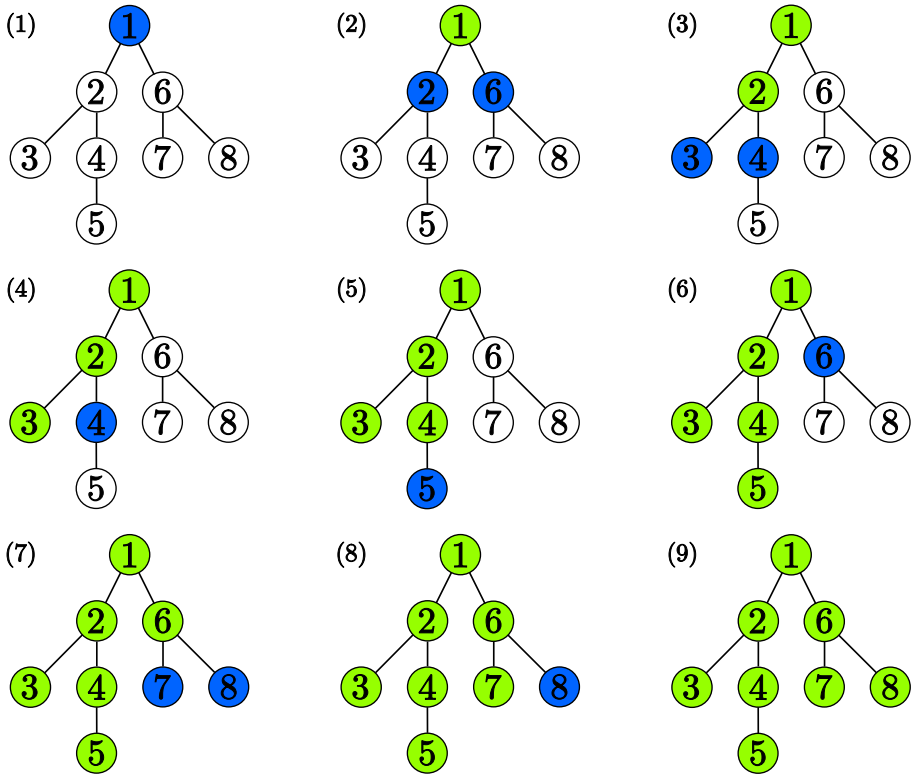
Input: Graph $G = (V, E)$
Output: Anzahl der Zusammenhangskomponenten und die Menge der Knoten jeder Zusammenhangskomponente

1. Initialisierung:
   MARK(v) := -1 für alle $v \in V$
   NEXT := 1
2. For $v \in V$ Do
   a) If MARK(v) < 0 Then
      i. MARK(v) := NEXT
      ii. For $w \in \delta(v)$ Do
         A. check_nachbar(w, NEXT)
      iii. NEXT := NEXT + 1
   b) End If
3. End For
4. Gib NEXT aus

check_nachbar(w, NEXT)
Input: $G = (V, E)$, $w \in V$, NEXT, MARK($\cdot$)

(1) If MARK(w) < 0 Do
   (2) MARK(w) = NEXT
   (3) For $u \in \delta^+(w)$ Do
      (4) check_nachbar(u, NEXT)
   (5) End For
(6) End If

Beispiel:



Auch hier wählen wir zuallererst einen Startknoten, dieser ist Knoten 1. Nun suchen wir uns einen Nachbarn von Knoten 1 aus, zur Auswahl stehen Knoten 2 und 6. Wir wählen Knoten 2. Nun schauen wir, welche Nachbarn Knoten 2 hat, wovon wir einen wählen. Wir wählen Knoten 3. Knoten 3 hat keine uns unbekannten Nachbarn, daher gehen wir wieder einen Knoten zurück, also zu Knoten 2. Der letzte unbekannte Nachbar von Knoten 2 ist Knoten 4, weshalb wir zu Knoten 4 gehen. Die möglichen Nachbarn von Knoten 4 ist nur ein Knoten, Knoten 5, welchen wir dann auch wählen. Knoten 5 hat keinen Nachbarn, also geht es zurück zu Knoten 4, auch dieser hat keine weiteren unbekannten Nachbarn, also weiter zurück zu Knoten 2. Auch dieser hat keine weiteren Nachbarn, daher gehen wir noch einen Schritt weiter zurück, also zu Knoten 1. Knoten 1 hat wieder einen uns unbekannten Nachbar, Knoten 6, welchen wir wählen. Knoten 6 hat die beiden Nachbarn 7 und 8, wovon wir Knoten 7 wählen, Nun wird wieder geschaut, welche Nachbarn Knoten 7 hat. Da Knoten 7 keine Nachbarn hat, geht es wieder einen Schritt zurück zu Knoten 6. Knoten 6 hat noch einen weiteren uns unbekannten Nachbarn, Knoten 8, welchen wir wählen. Da Knoten 8 keinen weiteren Nachbarn mehr hat und wir auch alle Knoten bereits erfasst haben, sind wir fertig.

4 Bäume und Wälder

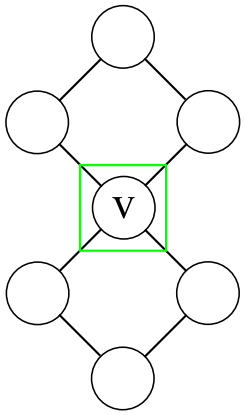
In diesem Kapitel beschäftigen wir uns mit dem Problem in einem Graphen mit Kantengewichten einen *aufspannenden Baum minimalen Gewichts* oder einen *Wald maximalen Gewichts* zu finden. Diese beiden Probleme sind in direkter Weise äquivalent, siehe Übung.

4.1 Grundlagen

Definition 4.1
Ein Graph G heißt **zusammenhängend**, falls es zu jedem Knotenpaar $s, t \in V$ einen $[s, t]$-Weg in G gibt.
Die **Komponenten** von G sind die bezüglich Kanteninklusion maximalen zusammenhängenden Teilgraphen von G.
Kanteninklusion kommt von inkludieren, also miteinbeziehen oder hinzuziehen. „Kanteninklusion maximal“ bedeutet, dass man keine Kante mehr hinzunehmen kann, da es sonst keine maximal zusammenhängenden Teilgraphen mehr wären.

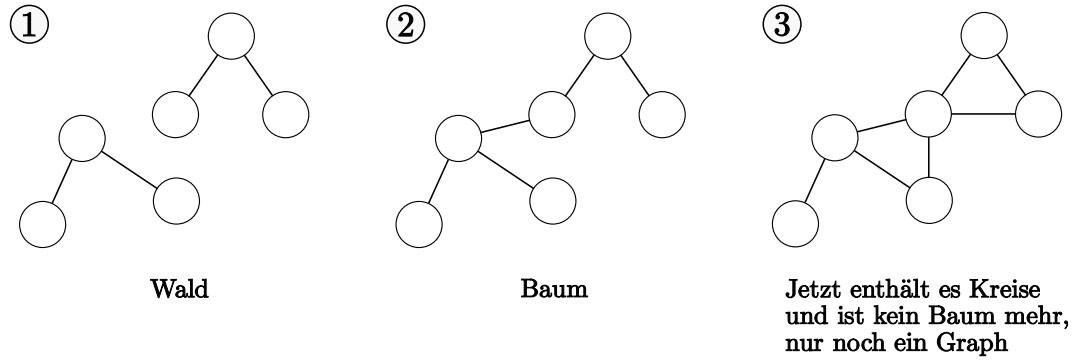
Wir bezeichnen mit $\zeta(G)$ die Anzahl der Komponenten von G. Ein Knoten v heißt **Artikulationsknoten** von G, falls $\zeta((G) - \{v\}) > \zeta(G)$

Beispiel:



Wir nennen eine Kantenmenge B einen **Wald** in G, falls $(V(B), B)$ keinen Kreis enthält. Ist $(V(B), B)$ außerdem zusammenhängend, so heißt B ein **Baum**.

Beispiel:



4.2 Algorithmen

Der folgende Algorithmus bestimmt einen Wald minimalen Gewichts (bzgl. Kanteninklusion).

Algorithmus 4.2 (Algorithmus von Kruskal, 1956)
Input: Graph $G = (V, E)$, Gewichte c_e für $e \in E$.
Output: maximaler Wald (bzgl. Kantenzahl) $T \subseteq E$ minimalen Gewichts.

(1) Sortiere die Kantengewichte in nicht-absteigender Reihenfolge:

$$c_{e_1} \leq c_{e_2} \leq \ldots \leq c_{e_m}$$

(2) Setze $T := \emptyset$

(3) For $i = 1$ To m Do

  (4) If $T \cup \{e_i\}$ enthält keinen Kreis Then $T := T \cup \{e_i\}$.

(5) Gib T aus.

Die Laufzeit des Algorithmus von Kruskal beträgt

$$\mathcal{O}(\underbrace{m \cdot \log m}_{\text{sortieren}} + \underbrace{m}_{\text{Schleife}} \cdot \underbrace{n}_{\text{Kreisprüfung}}) = \mathcal{O}(m \cdot n)$$

wobei $m = |E|$ und $n = |V|$ sei.

Satz 4.3
Der Algorithmus von Kruskal ist korrekt.

BEWEIS:
Behauptung: Nach dem i-ten Schritt gibt es einen (bzgl. Kanteninklusion) maximalen Wald $\overline{W}$ minimalen Gewichts mit $\overline{W} \cap \{e_1, \ldots, e_i\} = W \cap \{e_1, \ldots, e_i\}$, wobei W ein maximaler Wald minimalen Gewichts ist.

Induktion über i.
$i = 0$: Für $i = 0$ ist die Behauptung offensichtlich erfüllt.
$i - 1 \to i$: Hier wird zwischen 2 Fällen unterschieden:

1.) $W \cup \{e_i\}$ enthält einen Kreis, d.h. $e_i \notin W$.
Da $\overline{W} \cap \{e_1, \ldots, e_{i-1}\} = W \cap \{e_1, \ldots, e_{i-1}\}$ enthält auch $\overline{W} \cup \{e_i\}$ einen Kreis, somit ist $e_i \notin \overline{W}$.

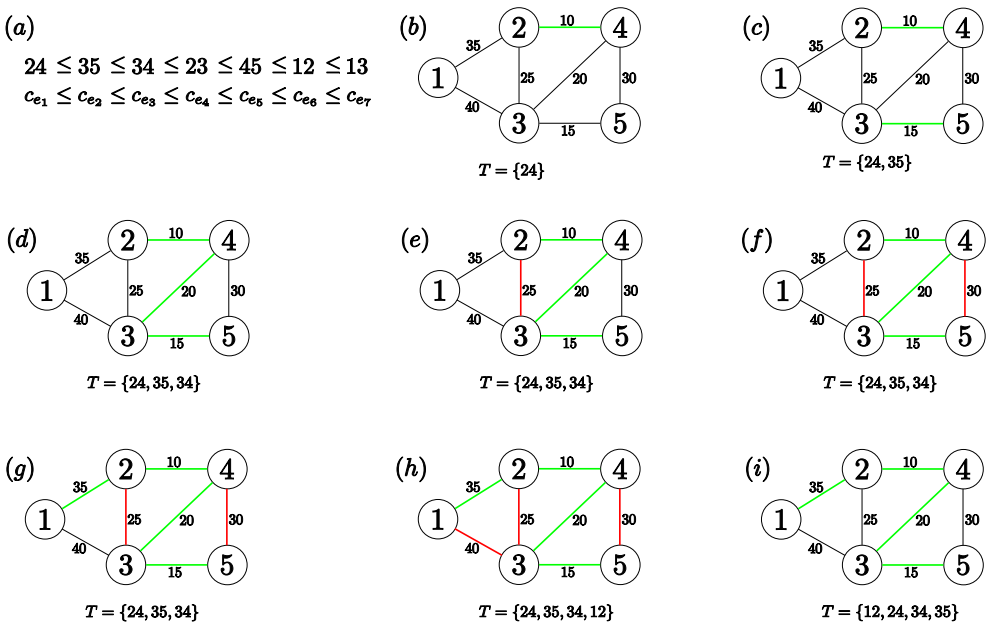
2.) $W \cup \{e_i\}$ enthält keinen Kreis, d.h. $e_i \in W$.

   a.) $e_i \in \overline{W}$, was die Behauptung ist.

   b.) $e_i \notin \overline{W}$. Damit enthält $\overline{W} \cup \{e_i\}$ einen Kreis C. Somit muss es ein $j > i$ geben mit $e_j \in \overline{W}, e_j \notin W$ und $c_{e_j} \geq c_{e_i}$, da $\overline{W} \cap \{e_1, \ldots, e_{i-1}\} = W \cap \{e_1, \ldots, e_{i-1}\}$ und $W \cup \{e_i\}$ keinen Kreis enthält. D.h. $\overline{W}' := (\overline{W} \setminus \{e_j\}) \cup \{e_i\}$ ist auch ein (bzgl. Kanteninklusion) maximaler Wald minimalen Gewichts, der die Behauptung erfüllt.

Für $i = m$ folgt die Behauptung. □

Beispiel:



Zu Beginn werden die Kanten nach ihrer Gewichtung sortiert, in unserem Beispiel wäre also $24 \leq 35 \leq 34 \leq 23 \leq 45 \leq 12 \leq 13$, wie es in Grafik (a) zu sehen ist.
Anschließend werden die Kanten die keinen Kreis bilden, von der kleinsten bis zur größten Kantengewichtung, vereinigt und bilden die Menge T. Die Kante mit der kleinsten Kantengewichtung ist 24, sie ist also die erste Kante (siehe Grafik (b)), die mit der Menge T vereinigt wird, die zweite wäre 35, da 24 und 35 keinen Kreis bilden, wird 35 mit T vereinigt, wie es in der Grafik (c) zu sehen ist. Die nächste Kante ist 34; da auch 24, 35 und 34 keinen Kreis bilden, werden diese zur Menge T vereinigt, wie es in Grafik (d) zu sehen ist. Die nächste Kante wäre 23. Die Kanten 24, 35, 34 und 23 bilden aber einen Kreis und verstoßen somit gegen Schritt (4) im Pseudocode, also wird die Kante 23 nicht genommen, daher ist sie in Grafik (e) rot markiert. Die nächste Kante ist 45, auch $T \cup \{45\}$ (zur Erinnerung: $T = \{24, 35, 34\}$) bilden einen Kreis, womit Kante 45 auch rausfällt, also bekommt 45 eine rote Färbung. Nun ist Kante 12 an der Reihe, da $T \cup \{12\}$ keinen Kreis bilden, wird 12 mit T vereinigt. Nun ist Kante 13 an der Reihe. Aber auch hier würde $T \cup \{13\}$ einen Kreis bilden, und fällt somit raus, wie in Grafik (h) zu sehen.
Die Schleife endet, wenn der Laufindex i bei m angekommen ist, also die Schleife m-mal durchlaufen wurde. Da unser Graph aus genau 7 Kanten besteht, weil wie man in Grafik (a) sieht, die Kante mit der größten Gewichtung den Index 7 trägt, ist m also 7. Die Schleife wurde 7-mal durchlaufen und endet somit. Der Code kommt also nun bei Schritt (5) im Pseudocode an, wo T ausgegeben wird. T, also der maximale Wald minimalen Gewichts, ist gegeben durch $T = \{24, 35, 34, 12\}$, wie in Grafik (i) zu sehen ist.

Bemerkung 4.4

a.) Wenn G zusammenhängend ist, so ist ein Wald maximaler Kantenanzahl und minimalen Gewichts identisch mit einem minimal aufspannenden Baum.

b.) Der Kruskal-Algorithmus gehört zu der Klasse der Greedy-Algorithmen. Greedy bedeutet soviel wie gefräßig, gierig. Der Algorithmus heißt daher Greedy, weil in jedem Schritt diejenige Kante ausgewählt wird, die hinsichtlich der Kostenfunktion bestmöglich ist. Siehe dazu auch ein späteres Kapitel.

c.) Der Kruskal-Algorithmus kann analog auch zur Bestimmung eines maximalen (bzgl. Gewichtsfunktion) Waldes benutzt werden, wobei man nur Kanten in Erwägung zieht, deren Gewichte positiv sind.

Ein gemeinsames Gerüst für viele Algorithmen ist der folgende Algorithmus:

Algorithmus 4.5
Input: Graph $G = (V, E)$ zusammenhängend, Gewichte c_e für $e \in E$.
Output: Aufspannender Baum $T \subseteq E$ minimalen Gewichts.

(1) Initialisierung: Für $i \in V$ setze $V_i = \{i\}$, $T_i := \emptyset$.

(2) Führe $|V| - 1$ mal aus:

(3) Wähle nichtleeres V_i.

(4) Wähle $uv \in \delta(V_i)$, $u \in V_i$ mit $c_{uv} \leq c_{pq}$ für alle $pq \in \delta(V_i)$

(5) Bestimme j, so dass $v \in V_j$.

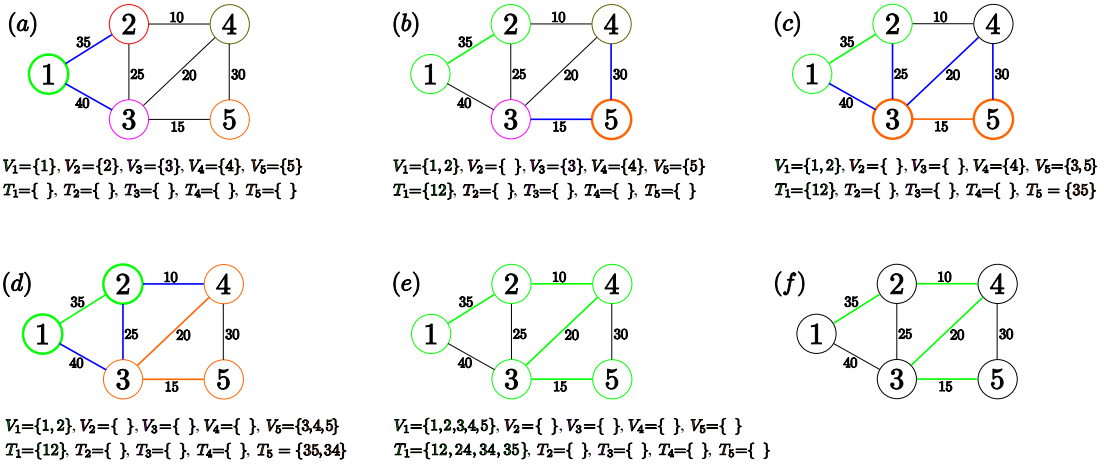
(6) Setze $V_i := V_i \cup V_j$, $V_j := \emptyset$ und $T_i := T_i \cup T_j \cup \{uv\}$, $T_j := \emptyset$

(7) Gib T_i mit $T_i \neq \emptyset$ aus!

Satz 4.6
Der Algorithmus arbeitet korrekt.

Beweis:
Wir zeigen durch Induktion über $k = \sum |T_i|$, dass G einen minimalen aufspannenden Baum T mit $T_i \subseteq T$ enthält.
$k = 0$: Trivial.
$k - 1 \to k$: Sei uv die hinzugefügte Kante. Nach Annahme gibt es einen minimalen aufspannenden Baum T mit $T_i \subseteq T$ für alle bisher bestimmten Mengen T_i.
$uv \in T$: erfüllt die Behauptung.
$uv \notin T$: So muss $T \cup \{uv\}$ einen Kreis enthalten. Also gibt es eine Kante $rs \in T$ mit $r \in V_i$ und $s \in V \setminus V_i$. Nach Wahl der in (4) gilt $c_{rs} \geq c_{uv}$. Also ist $(T \setminus \{rs\}) \cup \{uv\}$ ebenfalls ein minimal aufspannender Baum, der die Bedingung der Induktionsannahme erfüllt. □

Beispiel:



Als erstes müssen wir laut Pseudocode Schritt (1) die Initialisierung vornehmen. Da die Menge V genau 5 Elemente enthält, erstellen wir 5 Teilmengen $V_i \subseteq V$ mit $V_i = \{i\}$ für $i = 1, 2, 3, 4, 5$ und 5 Mengen T_i mit $i = 1, 2, 3, 4, 5$, wobei vorerst $T_i = \emptyset$ gilt.
Nun kommen wir zu Schritt (3), diesen führen wir genau $|V| - 1 = 5 - 1 = 4$-mal durch. Zuerst wählen wir ein beliebiges V_i, wir wählen V_1. In Grafik (a) ist diese nun fett hervorgehoben. Von diesem Knoten 1 müssen wir nun die Kante mit dem geringsten Kantengewicht wählen, die den Knoten 1 als Anfangsknoten hat. Die möglichen Kanten sind in Grafik (a) blau markiert. Diese Kante wäre 12. In Schritt (4) wählen wir nun ein j, so dass Knoten $2 \in V_j$ gilt, also ist $j = 2$, da der Knoten 2 ein Element der Teilmenge V_2 ist. In Schritt (6) wird nun die Menge V_i und V_j, also in unserem Fall V_1 und V_2, neu definiert, indem V_1 mit V_2 vereinigt und anschließend $V_2 := \emptyset$ gesetzt wird. Ebenso redefinieren wir die Menge T_1, indem wir die Kante 12 mit der Menge T_1 und T_2 vereinigen. Anschließend beenden wir damit Schritt (6) und kehren wieder zu Schritt (3) zurück.
In Grafik (b) sieht man nun, welche Mengen neu entstanden sind. Wir wählen nun ein neues, nichtleeres V_i. Man kann sehr wohl auch erneut die Menge V_1 verwenden, wir wählen aber V_5, dieser zugehörige Knoten ist nun in Grafik (b) fett hervorgehoben. Wieder wird die Kante mit dem geringsten Kantengewicht gewählt, die den Knoten 5 als Startknoten hat. Die möglichen Kanten sind wieder blau markiert und die gesuchte Kante ist die Kante 53. Nun benötigen wir wieder ein j, so dass $3 \in V_j$ gilt. Da der Knoten 3 Element der Menge V_3 ist, wählen wir $j = 3$. Wieder bei Schritt (6) angekommen, redefinieren wir die beiden Menge, V_5 und V_3, indem wir sagen $V_5 := V_5 \cup V_3$ und anschließend $V_3 := \emptyset$. Auch wird wieder T_i redefiniert: $T_5 := T_5 \cup T_3 \cup \{53\}$ und anschließend $T_j := \emptyset$. In Grafik (c) sieht man nun die neu entstandenen Mengen. Schritt (6) wäre somit beendet und der 2. Durchlauf abgeschlossen, es fehlen also noch 2. Daher geht es wieder zurück nach Schritt (3).
Wieder wählen wir ein nichtleeres V_i. Wir wählen die Menge V_5, welche in Grafik (c) dick hervorgehoben ist. Die zulässigen Kanten sind wieder blau markiert und die zu wählende Kante ist die Kante 34. Wir suchen also ein j, wofür gilt $4 \in V_j$, also ist $j = 4$. Nun wird in Schritt (6) erneut V_5 und V_4 redefiniert, indem wir V_5 als Vereinigung der beiden Mengen V_5 und V_4 definieren und anschliessend $V_4 := \emptyset$ setzen. Auch muss wieder T_5 und T_4 wie gehabt umdefiniert werden. T_5 wird als Vereinigung zwischen T_5, T_4 und $\{34\}$ definiert und anschließend $T_4 := \emptyset$ gesetzt. Die neu entstandenen Mengen kann man nun in Grafik (d) sehen. Da dies erst der 3. Durchlauf war, benötigen wir noch einen 4. und kehren daher erneut zu Schritt (3) im Pseudocode zurück.
Die einzigen beiden noch verbleibenden nichtleeren Mengen V_i sind V_1 und V_5. Dieses mal wählen wir die Menge V_1, welche in Grafik (d) fett hervorgehoben ist. Die gültigen Kanten sind wie gehabt blau markiert, also 12, 23 und 24. Die Kante mit dem geringsten Kantengewicht ist 24, welche wir daher auch wählen. Da der Knoten 4 ein Element der Menge V_5 ist, wählen wir $j = 5$ und kommen zu Schritt (6). Hier wird wie gehabt die Menge $V_1 := V_1 \cup V_5$ redefiniert und anschließend $V_5 := \emptyset$ gesetzt. Die Kantenmenge T_1 wird nun noch mit der Menge T_5 und der Kante $\{24\}$ vereinigt und als T_1 definiert. Die nun entstandenen Mengen sind in Grafik (e) zu sehen.
Da Schritt (2) nun 4 mal ausgeführt wurde, gehen wir nun weiter zu Schritt (7), wo T_i,

also T_1, ausgegeben wird. $T_1 = \{12, 24, 34, 35\}$ ist also der aufspannende Baum minimalen Gewichts.

Ein Spezialfall vom Algorithmus 4.5 ist der Algorithmus von Prim, der auf R.C. Prim zurückgeht.

Algorithmus 4.7 (Algorithmus von Prim, 1957)
Input: Graph $G = (V, E)$ zusammenhängend, Gewichte c_e für $e \in E$.
Output: Aufspannender Baum $T \subseteq E$ minimalen Gewichts.

(1) Initialisierung: Wähle $w \in V$ beliebig und setze $T := \emptyset$, $W := \{w\}$ und $V' := V \setminus \{w\}$.

(2) If $V' = \emptyset$ Then **Stop** (gib T aus).

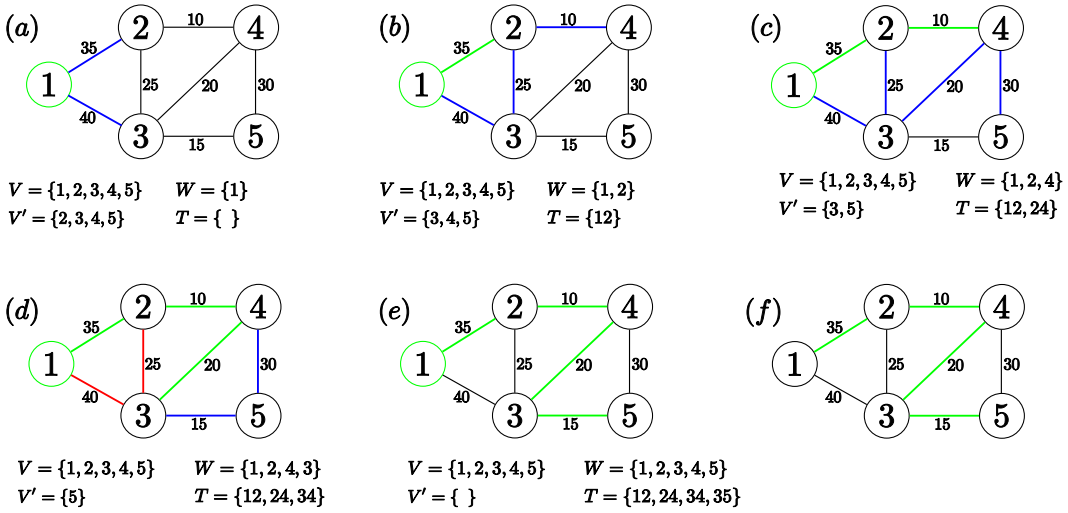
(3) Wähle $c_{uv} = \min \{c_e \mid e \in \delta(W)\}$ mit $u \in W$ und $v \in V'$.

(4) Setze $T := T \cup \{uv\}$, $W := W \cup \{v\}$ und $V' := V' \setminus \{v\}$.

(5) Gehe zu (2).

Die Laufzeit des Algorithmus von Prim ist $\mathcal{O}(n^2)$. Für vollständige Graphen ist dies bestmöglich.

Beispiel:



Wie aus dem Pseudocode erkennbar, dürfen wir uns wieder einen beliebigen Knoten aus V wählen, der Einfachheit halber wählen wir wieder Knoten 1. Nun werden wieder alle Kanten betrachtet, die vom Knoten 1 ausgehen, in der Grafik (a) blau gekennzeichnet; diese wären 12 und 13. Da 12 die kleinere Kantengewichtung hat, wählen wir diese und

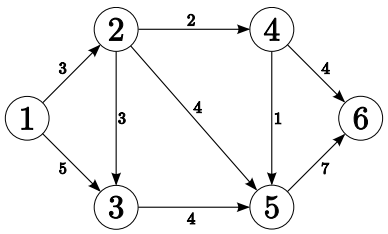
fügen sie wieder in T ein, fügen Knoten 2 in die Menge W ein und nehmen Knoten 2 aus der Menge V'. Der so entstandene Graph ist in Grafik (b) grün gekennzeichnet. Die nun zur Auswahl stehenden Kanten sind wieder blau gekennzeichnet, diese wären 13, 23 und 24. Aus diesen 3 Kanten wird wieder die mit der geringsten Kantengewichtung gewählt, das wäre die Kante 24. Kante 24 wird also wieder in die Menge T einsortiert, der Knoten 4 der Menge W hinzugefügt und aus der Menge V' genommen. Den neu entstandenen Graphen kann man in Grafik (c) wieder in grün sehen, die nun zur Auswahl stehenden Kanten sind wieder blau; diesen wäre 13, 23 und 45. Die Kante mit der geringsten Gewichtung ist die Kante 34. Wir fügen also Kante 34 wieder in T ein, fügen Knoten 3 in Menge W ein und entfernen ihn aus V'. Den nun entstanden Graphen kann man in Grafik (d) in grün sehen und die nun zur Auswahl stehenden Kanten sind wieder blau, also 35 und 45. Die beiden Kanten 13 und 23 stehen nicht zur Verfügung, daher sind sie rot gekennzeichnet, weil $v \in V'$ gilt, aber Knoten 3, 1 und 2 nicht mehr Elemente von V' sind. Wir wählen also die Kante 35 aus, fügen diese zu T, vereinigen Knoten 5 mit der Menge W und nehmen Knoten 5 aus V'. V' ist nun leer und laut dem Rekursionsanker in Punkt (2) soll T ausgegeben werden, wenn $V' = \emptyset$. Also wird T ausgegeben mit $T = \{1, 2, 4, 3, 5\}$. Der aufspannende Baum $T \subseteq E$ minimalen Gewichts ist nun in Grafik (e) in grün zu sehen.

5 Kürzeste Wege

Gegeben sei ein gerichteter Graph, auch Digraph genannt, $D(V, A)$ mit Bogengewichten c_a für $a \in A$ und zwei Knoten s und t.
Gesucht sei ein kürzester Weg von s nach t, d.h. ein Weg P von s nach t mit $\sum_{(i,j)\in P} c_{ij}$ minimal.

Annahme: Im folgenden nehmen wir an, dass alle Kantengewichte nicht-negativ sind, d.h. $c_a \geq 0$ für alle $a \in A$.

Beispiel:



Digraph D

Anwendungen für das Finden kürzester Wege sind beispielsweise

- Autopilotsystem
- Telekommunikation
- Internet

5.1 Eigenschaften kürzester Wege

Proposition 5.1
Ist $(s = i_0, i_1, \ldots, i_k = t)$ ein kürzester Weg von s nach t $(i_l \in V,\ l = 0, \ldots, k)$ so ist auch jeder Teilweg $(i_0, \ldots, i_l)$ für $(l = 1, \ldots, k-1)$ ein kürzester Weg von s nach i_l.

Beweis:
Übungsaufgabe □

Proposition 5.2
Seien $d : V \to \mathbb{R}$ die Werte der kürzesten Wege von s nach v für alle $v \in V$. Dann gilt: Ein gerichteter Weg P von s nach v ist genau dann kürzester Weg, falls $d(j) = d_i + c_{ij}$ für alle $(i, j) \in P$.

Beweis:
„$\Longrightarrow$“ folgt aus Proposition 5.1.

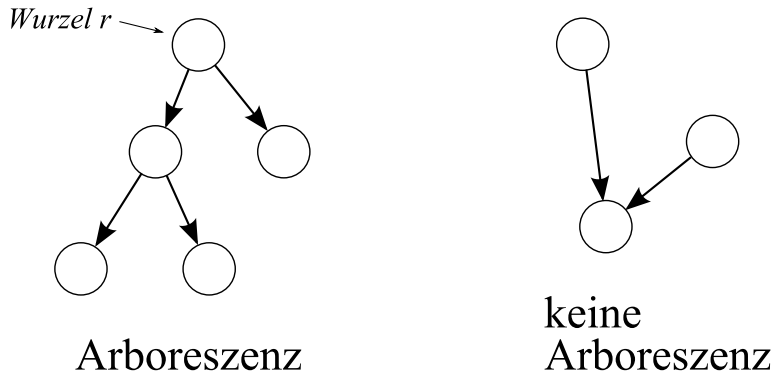
„$\Longleftarrow$“ Sei $P = (s = i_0, i_1, \ldots, i_k = v)$. Dann gilt:

$$\begin{aligned} d(v) = d(i_k) &= (d(i_k) - d(i_{k-1})) + (d(i_{k-1}) - d(i_{k-2})) + \ldots + (d(i_1) - d(i_0)) \\ &= \sum_{(i,j)\in P} c_{ij}, \end{aligned}$$

das heißt P hat die Länge $d(v)$, also ist P ein kürzester Weg. □

Definition 5.3 (Arboreszenz)
Ein kreisfreier, zusammenhängender Digraph $D = (V, A)$ mit $\delta(v) \leq 1$ für alle Knoten $v \in V$, d.h. in jeden Knoten geht höchstens eine Kante rein, heißt **Arboreszenz**.
Ein Knoten $r \in V$, von dem aus alle anderen Knoten in V über gerichtete Bögen erreichbar sind heißt **Wurzel**.
Eine Arboreszenz ist ein gerichteter Baum mit Wurzel.

Beispiel:



Definition 5.4 (Kürzester-Wege-Baum)
Gegeben sei ein Digraph $D = (V, A)$ und $B \subseteq A$.
Eine aufspannende Arboreszenz (V, B) mit Wurzel s heißt **Kürzester-Wege-Baum**, falls der eindeutige Weg von s nach v in B ein kürzester Weg von s nach v für alle $v \in V$ ist.

Proposition 5.5
Ein Kürzester-Wege-Baum existiert, sofern es einen gerichteten Weg von s nach v für alle $v \in V$ gibt.

Beweis:
Da es nur endlich viele Wege von s nach v gibt, existiert ein kürzester Weg P_v von s nach v für alle $v \in V$. Proposition 5.2 impliziert, dass $d(j) = d(i) + c_{ij}$ für alle $(i,j) \in P_v$. Starten wir in s und laufen per Depth-First-Search Algorithmus durch den Graphen entlang der Bögen (i,j), die $d(j) = d(i) + c_{ij}$ erfüllen, so müssen wir jeden Knoten erreichen. Die daraus resultierende Arboreszenz ist ein Kürzester-Wege-Baum. □

Satz 5.6 (Optimalitätsbedingung)
Seien $d(v)$ mit $v \in V$ obere Schranken für den Wert des kürzesten Weges von s nach v mit $d(s) = 0$. Dann gilt $d(v)$ ist der Wert eines kürzesten Weges von s nach v genau dann, wenn

$$d(v) \leq d(u) + c_{uv} \qquad \forall\, (u,v) \in A.$$

Beweis:
„$\Longrightarrow$“ klar nach Definition.

„$\Longleftarrow$“ Sei $P = (s = i_0, \ldots, i_k = v)$ ein Weg von s nach v. Dann gilt

$$d(v) \leq d(i_{k-1}) + c_{i_{k-1}i_k} \leq \ldots \leq \underbrace{d(i_0)}_{=0} + \sum_{(i,j)\in P} c_{ij},$$

d.h. $d(v)$ ist eine untere Schranke für den Wert eines kürzesten Weges von s nach v. Da $d(v)$ nach Voraussetzung auch obere Schranke ist, folgt die Behauptung. □

Definition 5.7
Für eine Distanzfunktion $d : V \to \mathbb{R}$ bezeichnen wir mit

$$c_{ij}^d := c_{ij} + d(i) - d(j)$$

die **reduzierten Bogengewichte bezüglich** d.

Proposition 5.8
a.) Für einen gerichteten Kreis W gilt

$$\sum_{(i,j)\in W} c_{ij}^d = \sum_{(i,j)\in W} c_{ij}.$$

b.) Für einen gerichteten Weg P von u nach v gilt

$$\sum_{(i,j)\in P} c_{ij}^d = \sum_{(i,j)\in P} c_{ij} + d(u) - d(v).$$

c.) Falls $d(\cdot)$ die Werte kürzester Wege sind, gilt

$$c_{ij}^d \geq 0 \qquad \forall\, (i,j) \in A$$

Beweis:
5.8a, 5.8b sind klar. 5.8c folgt aus Satz 5.5. □

Wir stellen nun einen Kürzeste-Wege-Algorithmus vor.

Algorithmus 5.9 (Dijkstra (1959))
Input: Digraph $D = (V, A)$, $c_a \geq 0$, $a \in A$, Startknoten $s \in V$.
Output: Kürzeste gerichtete Wege von s nach v für alle $v \in V$.
Datenstrukturen: $d(v)$ ist die Distanz von s nach v.
Vor(v) ist der Vorgänger von v auf dem Weg von s nach v.

(1) Setze $d(s) = 0$
Vor$(s) = s$
Vor$(v) = s$ $\forall\, (s,v) \in A$

$$d(v) = \begin{cases} c_{sv} & \text{falls } (s,v) \in A \\ \infty & \text{sonst.} \end{cases}$$

Markiere s.

(2) Bestimme unmarkierten Knoten u mit $d(u) = \min\{d(v) \mid v \text{ unmarkiert}\}$.
Markiere u.

(3) For All unmarkierte Knoten v mit $(u,v) \in A$ Do

(4) If $d(v) > d(u) + c_{uv}$ Then

(5) $d(v) := d(u) + c_{uv}$

(6) Vor$(v) := u$

(7) End If

(8) End For All

(9) Falls nicht alle Knoten markiert sind gehe nach (2)

(10) **Stop** gib $d(\cdot)$ und Vor$(\cdot)$ aus.

Satz 5.10
Der Dijkstra-Algorithmus arbeitet korrekt und die Laufzeit beträgt $\mathcal{O}(n^2)$.

Beweis:
Laufzeit:

Schritt (1)	$\mathcal{O}(n)$
Schritt (2)	$\mathcal{O}(n)$
Schritt (3)	$\mathcal{O}(n)$
Schritt (7)-(9)	$\mathcal{O}(n)$

Führen wir Schritt (2) bis Schritt (9) $\leq n$-mal durch, so erhalten wir die Laufzeit $\mathcal{O}(n^2)$

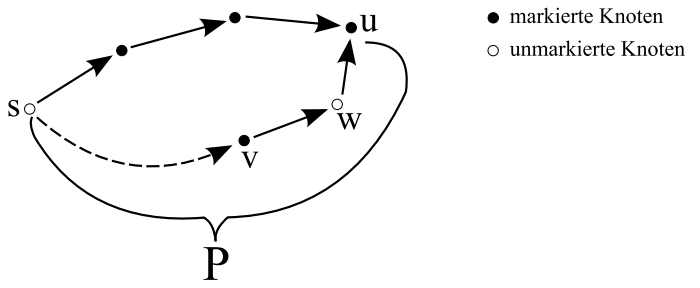
Um die **Korrektheit** des Algorithmus zu beweisen, zeigen wir per Induktion über die Anzahl der markierten Knoten Folgendes:

a.) Ist v markiert, so enthält $d(v)$ die Länge eines kürzesten Weges.

b.) Ist v unmarkiert, so enthält $d(v)$ die Länge eines kürzesten Weges, der als innerer Knoten nur markierte Knoten besitzt.

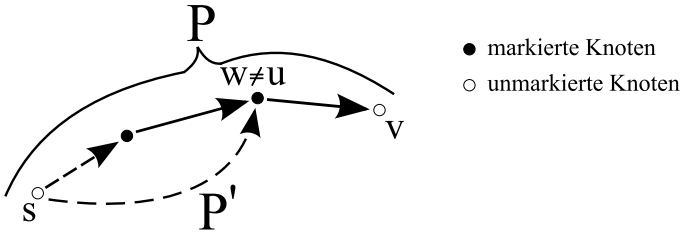
Induktionsanfang: Ist nur ein Knoten markiert (nämlich s), so ist die Behauptung erfüllt (Schritt (1)).

Induktionsschritt: Wir nehmen an, die Behauptung ist korrekt für k Knoten und wir markieren in Schritt (2) den $(k+1)$sten Knoten, sagen wir u. Nach Induktionsvoraussetzung wissen wir, $d(u)$ ist der Wert eines kürzesten Weges, der als innere Knoten nur markierte Knoten hat. Angenommen es gäbe einen kürzeren Weg P von s nach u. Sei $(v, w) \in P$ mit v markiert und w unmarkiert. Dann gilt $c(P) \underbrace{\geq}_{c_a \geq 0} d(w) \underbrace{\geq}_{\text{Schritt (2)}} d(u)$.



Also ist P nicht kürzer und a.) ist gezeigt.

Es bleibt noch zu zeigen, dass für die derzeit unmarkierten Knoten v, $d(v)$ der Wert eines kürzesten Weges ist, dessen innere Knoten alle markiert sind (b).
Sei wieder u der neu markierte Knoten. Angenommen es gäbe einen Weg P von s nach v, dessen innere Knoten alle markierte Knoten (inkl. u) sind und dessen vorletzter Knoten $w \neq u$ ist, und dessen Länge kürzer als die Länge der bislang betrachteten Wege ist.

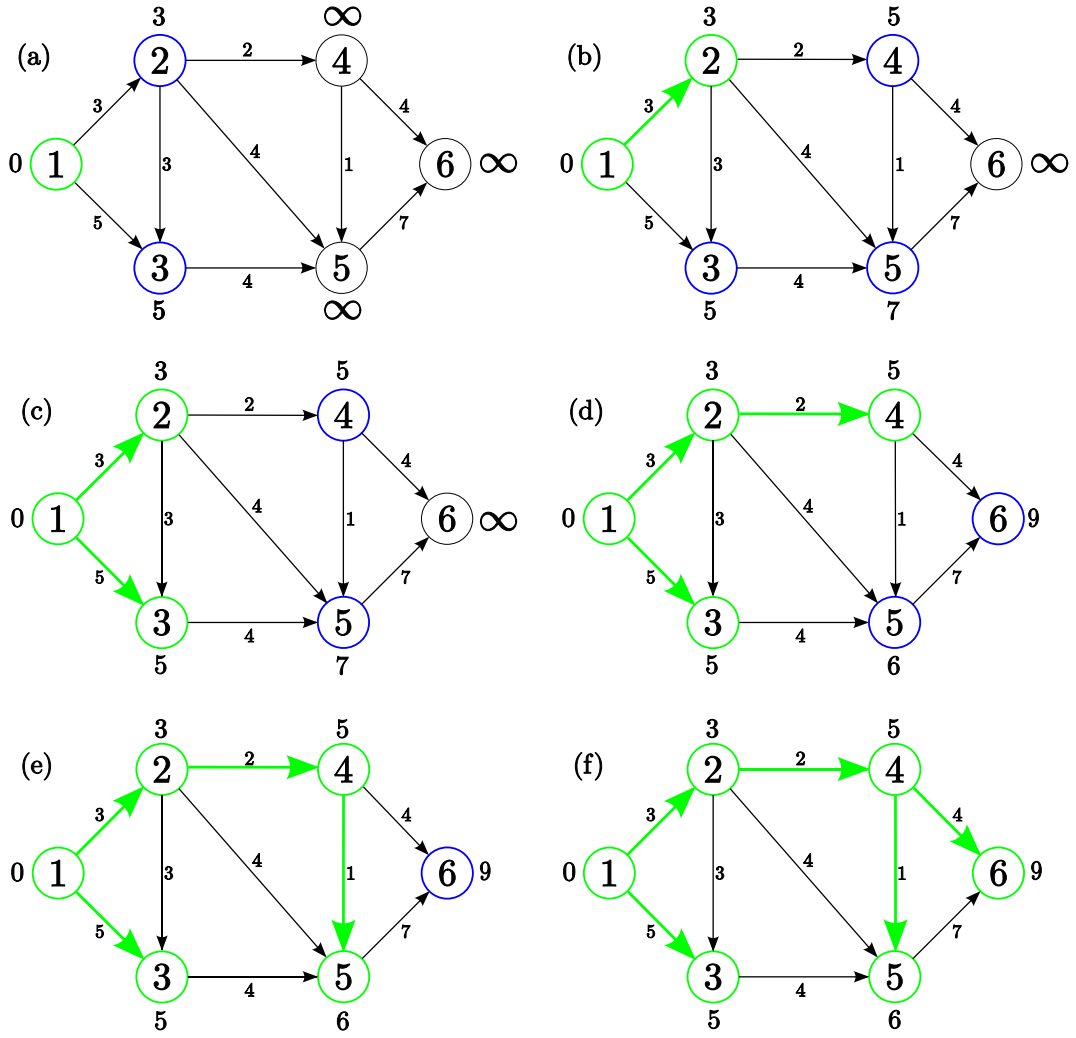


Da w markiert ist, enhält $d(w)$ die Länge eines kürzesten (s, w)-Weges. Also gibt es einen kürzesten Weg P' der nur markierte Knoten enthält, die verschieden von u sind (w wurde vor u markiert).

$$\Longrightarrow \quad d(v) \underbrace{>}_{\substack{\text{Annahme}\\ \text{P kürzer}}} c(P) \underbrace{\geq}_{c_a \geq 0} c(P') + c_{wv} = d(w) + c_{wv} \underbrace{\geq}_{\text{Schritt (4)-(6)}} d(v) \qquad \lightning$$

Womit auch b) gezeigt wäre. Also gelten nach jeder Iteration obige Behauptungen a.) und b.).
Insbesondere enthält also $d(\cdot)$ nach Beendigung des Algorithmus den kürzesten Weg von s zu jedem anderen Knoten. □

Beispiel:



Als Startknoten sei 1 vorgegeben. Dieser wird direkt markiert. Anschließend schreiben wir an alle verbliebenen Knoten die Distanz von unserem Startknoten 1 zu diesen Knoten. Wie in Grafik (a) zu sehen, sind die Distanzen folgendermaßen: $d(2) = 3$, $d(3) = 5$, $d(4) = \infty$, $d(5) = \infty$, $d(6) = \infty$. Da Knoten 4,5 und 6 nicht direkt von Knoten 1 erreichbar sind, bekommen sie erstmal die Distanz ∞. Nun wählen wir den nächsten Knoten, der (von 1 erreichbar ist und) die geringste Distanz d hat.
In Grafik (b) haben wir nun Knoten 2 markiert, da dieser die geringste Distanz zum Knoten 1 hat. Nun werden die Distanzen neu gesetzt. Würden wir von Knoten 2 nach Knoten 3 gehen, so wäre die Distanz von Knoten 1 über 2 nach 3 also 6. Da dieser Knoten aber direkt von 1 nach 3 mit $d(3) = 5$ erreichbar ist, bleibt $d(3) = 5$. Der nächste

erreichbare Knoten ist Knoten 4. Von Knoten 1 nach 2 ist $d(2) = 3$ und von Knoten 2 nach 4 ist die Distanz 2, also ist die Distanz von 1 nach 4 über Knoten 2 beträgt also 5. Da ∞ größer ist als 5, gilt nun $d(4) = 5$. Mit dieser Variante kommen wir nun auch auf $d(5) = 7$. Nun ist $d(3) = d(4) = 5$, wir können uns also aussuchen ob wir zuerst zum Knoten 3 oder 4 gehen, wir wählen Knoten 3 und markieren diesen.
In Grafik (c) ist zu sehen, dass wir nun eine weitere Möglichkeit dazu bekommen haben zu Knoten 5 zu kommen, nämlich den Weg von 1 über 3 nach 5. Die Kantengewichtung der Kante (35) ist 4. Da $5 + 4 = 9 > 7$ ist, bleibt also $d(5) = 7$. Nun ist also $d(4) = 5$ und $d(5) = 7$, also ist die Distanz zu Knoten 4 geringer, und wir markieren Knoten 4, wie in Grafik (d) zu sehen.
Vom Knoten 4 kommen wir nun auch nach Knoten 5, und da $5 + 1 = 6 < 7$ ist, wird nun $d(5) = 6$ gesetzt. Zusätzlich gibt es nun auch einen Weg über Knoten 4 nach Knoten 6. Da $5 + 4 = 9$ ist, ist also $d(6) = 9$. Der nächste unmarkierte Knoten mit minimaler Markierung ist Knoten 5, mit $d(5) = 6$. Wir markieren Knoten 5, wie in Grafik (e) zu sehen.
Die Distanz von Knoten 1 über Knoten 5 nach Knoten 6 wäre $9 + 7 = 16$, über Knoten 4 wäre die Distanz nur 9, also bleibt $d(6) = 9$ und da dies der letzte verbleibende Knoten ist, markieren wir also auch Knoten 6. Nun haben wir alle Knoten markiert und den kürzesten Weg von Knoten 1 zu allen anderen Knoten gefunden. Die Kürzeste-Wege-Arboreszenz kann man in grün in der Grafik (f) sehen.

5.2 Digraphen mit beliebigen Gewichten

Dijkstra funktioniert nicht für negative Gewichte (siehe Übung). Bei den bislang behandelten Algorithmen wurde in jeder Iteration das Distanzlabel $d(v)$ für ein $v \in V$ als permanent erklärt, d.h. es wurde gezeigt, dass $d(v)$ tatsächlich der Wert eines kürzesten Weges ist. Eine andere Idee, die auf Moore (1957) und Bellmann (1958) zurück geht, ist sich auf die reduzierten Bogengewichte zu kontentrieren. Satz 5.6 (und Proposition 5.8c) sagen aus, dass $d(\cdot)$ genau dann optimal ist, falls

$$c_{ij}^d = d_i + c_{ij} - d_j \geq 0 \qquad \forall\, (i,j) \in A$$

Die Idee ist nun, Bögen, die das reduzierte Bogenkriterium verletzen, auszuwählen, und die Labels zu reduzieren. Dieses Verfahren nennt man „labelcorrecting“-Algorithmen („Marken-Verbesserungs-Algorithmen“).
Wir wollen zunächst annehmen, dass D keine negativen Kreise (d.h. Kreis $C \subseteq A$ mit $c(C) < 0$) enthält.

Bemerkung:
Propositionen 5.1, 5.2 und 5.5 gelten auch, wenn man negative Bogengewichte zulässt. Es genügt vorauszusetzen, dass es keine negativen Kreise gibt, vergleiche Übungsaufgabe zum Beweis von Proposition 5.1.
Satz 5.6 und Proposition 5.8 gelten sogar für allgemeine Bogengewichte.

Algorithmus 5.11 (Grundversion von Moore-Bellmann)
Input: Digraph $D = (V, A)$ mit Gewichten c_a, $a \in A$ (D habe keine negativen Kreise), Startknoten $s \in V$
Output: Kürzeste Wege von s nach v für alle $v \in V$ und deren Länge.

1. Initialisierung:
$$d(v) = \begin{cases} 0 & \text{falls } v = s \\ \infty & \text{sonst} \end{cases}$$
$$\textsc{Vor}(v) = \text{ ungesetzt für alle } v \in V$$

2. While es gibt einen Bogen $(i, j) \in A$ mit $d(j) > d(i) + c_{ij}$ Do
$$d(j) = d(i) + c_{ij} \qquad \textsc{Vor}(j) = i$$

3. Gib $d(\cdot)$ und $\textsc{Vor}(\cdot)$ aus.

Definition 5.12
Sei
$$A_k := \{(\textsc{Vor}(i), i) : i \in V, \textsc{Vor}(i) \text{ gesetzt}\}$$
$$H_k := \{V(A_k), A_k\}$$
der **Vorgängergraph** von Algorithmus 5.11 nach der k-ten Iteration, sowie $d^k(\cdot)$ das entsprechende Distanzlabel.

Lemma 5.13
Für $k = 0, 1, 2, \ldots$ gilt
$$c_{ij}^{d^k} = c_{ij} + d^k(i) - d^k(j) \leq 0 \qquad \forall\, (ij) \in A_k$$

Beweis:
Induktion über k.

$\underline{k = 0}$: $A_0 = \emptyset$ ✓

$\underline{k \to k+1}$: Sei (i, j) der Bogen, der in der $(k+1)$sten Iteration hinzugefügt wird, also $\textsc{Vor}(j) = i$.
Offensichtlich bleiben alle Werte c_{rs}^d unberührt, die nicht Knoten j enthalten (also $r \neq j$, $s \neq j$).
Es gibt genau einen Bogen $(r, l) \in A_{k+1}$ mit $l = j$, nämlich $r = i$ und für den gilt
$$c_{ij}^{d^{k+1}} = c_{ij} + d^{k+1}(i) - d^{k+1}(j) = 0 \leq 0.$$
Für alle anderen Bögen $(r, l) \in A_{k+1}$ mit $r = j$ gilt
$$0 \geq c_{jl}^{d^k} = c_{jl} + d^k(j) - d^k(l) > c_{jl} + \underbrace{d^{k+1}(j)}_{<d^k(j)} - d^{k+1}(l) = c_{jl}^{d^{k+1}}.$$

Lemma 5.14
$H_k = (V(A_n), A_k)$ ist eine Arboreszenz mit Wurzel s für $k = 1, 2, \ldots$, falls D keinen negativen Kreis enthält.

Beweis:
Angenommen die Aussage ist falsch, dann sei k der kleinste Index, so dass H_k einen Kreis W enthält. Sei (i, j) der Bogen, der in der k-ten Iteration hinzugefügt wird, dann enthält H_k einen gerichteten Weg P von j nach i. Nach Lemma 5.13. gilt $c^{d^{k-1}}(P) \leq 0$ und somit:

$$\begin{aligned} c(W) &\overset{\text{Prop. 5.8b}}{=} c^{d^k}(W) = c^{d^k}(P) + c_{ij}^{d^k} \\ &\overset{\text{Prop. 5.8b}}{=} c(P) + d^k(j) - d^k(i) + \underbrace{(c_{ij} + d^k(i) - d^k(j))}_{=0} \\ &< c(P) + d^{k-1}(j) - d^{k-1}(i) \\ &\overset{\text{Prop. 5.8b}}{=} c^{d^{k-1}}(P) \leq 0 \end{aligned}$$

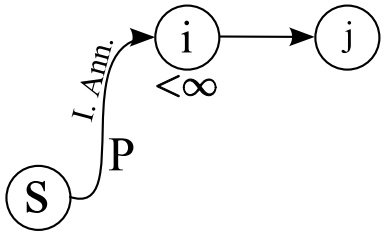
Was ein Widerspruch zu „D enthält keine negativen Kreise“ ist. $\lightning$

Offensichtlich ist H_k ein Branching, d.h. jeder Knoten ist Endknoten maximal eines Bogens aus A_k, da $\textsc{vor}(\cdot)$ eine Funktion von V nach V ist.

Zu zeigen ist noch, dass für alle $v \in V(A_k)$ H_k einen (s, v)-Weg enthält.
Induktion über k.

$\underline{k = 0:}$ $v \in V(A_0) \Leftrightarrow \textsc{vor}(v)$ gesetzt nach Initialisierung als $V(A_0) = \emptyset$.

$\underline{k \to k+1:}$ Sei (i, j) der Bogen, der hinzugefügt wird [(l, j) sei gelöscht worden].



Zu zeigen ist, dass ein gerichteter Weg von s nach j existiert. Nach Induktionsannahme existiert ein gerichteter Weg P von s nach i, beachte $d(i) < \infty$. Dieser kann nicht über j führen, da sonst H_{k+1} einen Kreis enthalten würde. Also ist $P \cup \{(i, j)\}$ der gesuchte Weg. $\square$

Satz 5.15
Algorithmus 5.11 arbeitet korrekt. Seine Laufzeit beträgt $\mathcal{O}\left(n^2 \cdot m \cdot C\right)$, wobei $C := \max_{a \in A} |c_a|$, falls alle Gewichte ganzzahlig sind.

BEWEIS:
Da alle Gewichte ganzzahlig sind, wird in jeder Iteration das Distanzlabel eines Knotens immer um mindestens 1 reduziert. Da der Wert eines kürzesten Weges im Intervall $[-n \cdot C, n \cdot C]$ liegt, endet der Algorithmus 5.11 nach $\mathcal{O}(n^2 \cdot m \cdot C)$ Iterationen und in jeder Iteration müssen maximal m Knoten betrachtet werden.

Sei k die letzte Iteration, die der Algorithmus 5.11 durchführt. Dann gilt:

$$\begin{aligned} &(2) \qquad c_{ij}^{d^k} \geq 0 \quad \forall (i,j) \in A. \\ &(3) \qquad \text{Da } c_{ij}^{d^k} \leq 0 \quad \forall (i,j) \in A_k \text{ (vgl. Lemma 13)}, \end{aligned}$$

ist $d^k(i)$ eine obere Schranke für den kürzesten Weg von s nach i, denn sei P der Weg von s nach i, dann gilt nach Proposition 5.8b

$$0 \geq c^{d^k}(P) = c(P) + d^k(s) - d^k(i) = c(P) - d^k(i).$$

Also folgt aus Satz 5.6 und 2, dass $d(\cdot)$ die Werte kürzester Wege von s nach v für alle $v \in V(A_k)$ sind.
Aus (2) und (3) folgt

$$c_{ij}^{d^k} = 0 \qquad \forall (i,j) \in A_k.$$

Damit ist H_k ein Kürzester-Wege-Baum und $v \in V(A_k)$ genau dann, wenn es einen Weg von s nach v gibt. □

Algorithmus 5.16 (Bellmann 1958)
Input: $D = (V, A)$, $c_a \in \mathbb{Z}$ oder $(\mathbb{Q}, \mathbb{R})$ für $a \in A$, $s \in V$
(D habe keine negativen Kreise)
Output: Kürzeste (s, v)-Wege mit Längen $d(\cdot)$

(1) Initialisierung

$$d(v) = \begin{cases} 0 & \text{falls } v = s, \\ \infty & \text{sonst.} \end{cases}$$
$$\text{VOR}(v) = \text{ ungesetzt} \quad \forall v \in V.$$

(2) Nummeriere die Bögen in $A = \{a_1, \ldots, a_m\}$ beliebig.

(3) FOR $k = 1$ TO n DO

(4) FOR $l = 1$ TO m DO

(5) Sei $a_l = (i, j)$

(6) IF $d(j) > d(i) + c_{ij}$ THEN

(7) Setze $d(j) := d(i) + c_{ij}$.

(8) Setze VOR$(j) := i$.

(9) Gib $d(\cdot)$ und VOR$(\cdot)$ aus.

Satz 5.17
Algorithmus 5.16. ist korrekt. Seine Laufzeit ist $\mathcal{O}(n \cdot m)$.

BEWEIS:
Wir zeigen zunächst per Induktion über die Anzahl Schleifendurchläufe (3) - (8), dass nach der k-ten Iteration $d(j)$, falls $d(j) < \infty$, eine untere Schranke für die Länge des kürzesten (s, j)-Weges ist, wobei nur Wege der Länge $\leq k$ zugelassen sind.

$\underline{k = 1:}$ ✓

$\underline{k \to k+1:}$ Betrachte einen Knoten j und einen kürzesten (s, j)-Weg
$P = (s = i_0, i_1, \ldots, i_l = j)$ mit der Länge $l \leq k + 1$.
Falls $l \leq k$ gilt, folgt die Behauptung aus der Induktionsannahme, da Distanzlabel nur verringert werden.
Sei also $l = k + 1$. Nach Proposition 5.1. ist $(s, \ldots, i_{l-1} = i_k) = P_k$ ein kürzester Weg der Länge k von s nach i_k. Nach Induktionsannahme und der Tatsache, dass Distanzlabel nicht erhöht werden, ist $d(i_k)$ eine untere Schranke für die Länge von P_k.
Beim $(k + 1)$sten Durchlauf wird bei Betrachtung des Bogens (i_k, i_{k+1}) $\quad d(j) := d(i_k) + c_{i_k i_{k+1}}$ gesetzt. Also ist $d(j)$ eine untere Schranke für die Länge von P.
Nach $k = |V|$ Iterationen erhalten wir somit, dass $d(j)$ eine untere Schranke eines kürzesten (s, j)-Weges ist, für alle $j \in V$.

Aus Lemma 5.13. folgt, dass $c_{ij}^{d^k} \leq 0$ für alle $k = 1, 2, \ldots$ und für alle $(i, j) \in A_k$.
Und aus Lemma 5.14. folgt, dass es einen (s, j)-Weg P_j in H_k gibt, für alle $j \in V(A_k)$.
Für diesen Weg gilt:

$$0 \geq c^{d^k}(P_j) = c(P_j) + d^k(s) - d^k(j) = c(P_j) - d^k(j)$$
$$\Longrightarrow d^k(j) \geq c(P_j)$$

Also ist am Ende von Alg. 5.16. $d^k(\cdot)$ eine obere Schranke für die Länge eines kürzesten (s, j)-Weges. Folglich muss Gleichheit gelten. Die Laufzeit ist offensichtlich $\mathcal{O}(m \cdot n)$. □

5.3 Behandlung negativer Kreise

Die Korrektheit der Algorithmen 5.11 und 5.16. benötigt die Eigenschaft, dass D keine negativen Kreise enthält, siehe Lemma 5.14. Enthält D einen negativen Kreis, so liefern beide Algorithmen nichts sinnvolles bzw. terminieren nicht.

Wir wollen nun unsere Algorithmen so modifizieren, dass sie als Eingabe beliebige Digraphen erlauben, und mit der Meldung „D hat einen negativen Kreis“ abbrechen, falls D einen besitzt.

Die Frage lautet also, wie entdeckt man negative Kreise in D?

Eine einfache Variante ist, zu überprüfen, ob ein Distanzlabel $d(j) < -nC$ ist. Offensichtlich muß dann D einen negativen Kreis enthalten. Um dies festzustellen, ist im „worst-case“ eine Laufzeit von $\mathcal{O}(nC)$ nötig und wir verlieren die strenge Polynomialität von Algorithmus 5.16.

Eine bessere Variante liefert folgende Beobachtung. Algorithmus 5.11 und 5.16 laufen korrekt, solange H_k keinen gerichteten Kreis enthält. Wir können nun nach jeder Iteration überprüfen, ob H_k einen gerichteten Kreis enthält. Dies geht mit Depth-First-Search in $\mathcal{O}(n)$. Haben wir einen solchen Kreis gefunden, so entspricht er einem negativen Kreis in D und wir können abbrechen (siehe Beweis von Lemma 5.14). Das verschlechtert die Laufzeit von Algorithmus 5.16 auf $\mathcal{O}(n^2 \cdot m)$. Die Laufzeit kann jedoch wieder auf $\mathcal{O}(n \cdot m)$ reduziert werden, wenn man diesen Check nur alle αn Iterationen durchführt, für ein konstantes α, denn

$$\mathcal{O}\left(nm + \frac{nm}{\alpha n}\mathcal{O}(n)\right) = \mathcal{O}(nm)$$

Man beachte hierbei, dass H_k eine Arboreszenz ist, woraus folgt $\mathcal{O}(n) = \mathcal{O}(m)$.

Algorithmus 5.18 (Floyd)
Input: $D = (V, A)$, $V = \{1, \ldots, n\}$ mit Gewichten c_a für alle $a \in A$. (D habe keine negativen Kreise)
Output: $n \times n$ Kürzeste-Weglängen-Matrix W, $n \times n$-Matrix P mit folgenden Eigenschaften:

w_{ij} = Länge eines kürzesten (i, j)-Weges.
w_{ii} = Länge eines kürzesten (i, i)-Kreises.
p_{ij} = vorletzter Knoten auf einem kürzesten (i, j)-Weg.
p_{ii} = vorletzter Knoten auf einem kürzesten (i, i)-Kreis.

(1) For $i = 1$ To n Do

(2) For $j = 1$ To n Do

$$w_{ij} := \begin{cases} c_{ij} & \text{falls } (i,j) \in A, \\ \infty & \text{sonst.} \end{cases}$$

$$p_{ij} := \begin{cases} i & \text{falls } (i,j) \in A, \\ 0 & \text{sonst.} \end{cases}$$

(3) End (j)

(4) End (i)

(5) For $l = 1$ To n Do

(6) For $i = 1$ To n Do

(7) For $j = 1$ To n Do

(8) If $w_{ij} > w_{il} + w_{lj}$ Then

Setze $w_{ij} := w_{il} + w_{lj}$ und $p_{ij} := p_{lj}$.
Falls $i = j$ und $w_{ii} < 0$ Stop, gehe nach (12)

(9) End (j)

(10) End (i)

(11) End (l)

(12) Gib W und P aus.

6 Maximale Flüsse

Definition 6.1
Sei $D = (V, A)$ ein Digraph mit Bogenkapazitäten $c_a \geq 0$ für alle $a \in A$. Seien $s, t \in V, s \neq t$, zwei verschiedene Knoten. s heißt **Quelle** (source), t heißt **Senke** (target). Eine Funktion $x : A \mapsto \mathbb{R}$ heißt **zulässiger** (s,t)**-Fluss**, wenn gilt:

(1)

$$0 \leq x_a \leq c_a \qquad \forall\, a \in A$$

(2)

$$\sum_{a \in \delta^-(v)} x_a = \sum_{a \in \delta^+(v)} x_a \qquad \forall\, v \in V \setminus \{s,t\} \qquad \text{[Flusserhaltung]}.$$

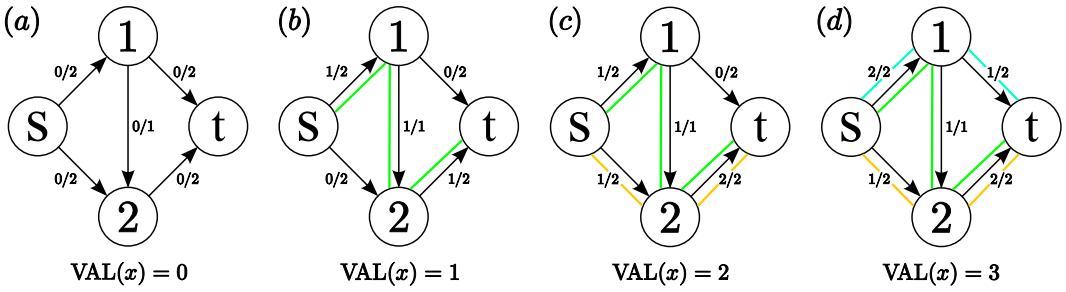
Wenn x ein zulässiger (s,t)-Fluss ist, dann ist

$$\mathrm{VAL}(x) := \sum_{a \in \delta^+(s)} x_a - \sum_{a \in \delta^-(s)} x_a$$

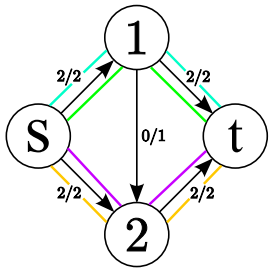
der **Wert** (Value) des Flusses.

Das Problem einen zulässigen Fluss x maximalen Wertes zu finden, heißt **Maximal-Fluss-Problem** (max flow problem).

Beispiel:



Die vier Abbildungen stellen zulässige Flüsse dar, aber keiner ist ein maximaler Fluss. Ein maximaler Fluss des gegebenen Graphen sähe folgendermaßen aus:



VAL(x) = 4

Anwendungen:

- Rohrleitungssyteme
- Telefonnetze
- Heiratsprobleme (können entsprechend transformiert werden)
- Unterproblem vieler NP-schwerer Probleme (TSP, Steinerbaum-Problem, ...).

6.1 Max-Flow-Min-Cut-Theorem

Proposition 6.2 (Flusszerlegung)
Sei $D = (V, A)$ ein Digraph mit Bogenkapazitäten $c_a \geq 0$ für alle $a \in A$ und $s, t \in V$ verschieden. Sei $\mathcal{P}$ die Menge aller gerichteten (s,t)-Wege in D und $\mathcal{C}$ die Menge aller gerichteten Kreise. Sei $x : A \to \mathbb{R}$ eine Zuordnung mit $0 \leq x_a \leq c_a$ für $a \in A$. Dann gilt: x ist ein zulässiger (s,t)-Fluss genau dann, wenn es Wege $P_1, \ldots, P_p \in \mathcal{P}$ und Kreise $C_1, \ldots, C_q \in \mathcal{C}$ sowie $\lambda_1, \ldots, \lambda_p > 0, \mu_1, \ldots, \mu_q > 0$ gibt mit

$$x_{uv} = \sum_{\{i | uv \in P_i\}} \lambda_i + \sum_{\{i | uv \in C_i\}} \mu_i.$$

Ferner gilt $p + q \leq |A|$.

Beweis:
Übungsaufgabe □

Lemma 6.3

1. Für jeden (s,t)-Fluss x gilt:

   $$\text{VAL}(x) = x(\delta^+(W)) - x(\delta^-(W))$$

   für alle $W \subseteq V$ mit $s \in W$ und $t \in V \setminus W$.

2. Für jeden zulässigen (s,t)-Fluss x und jeden (s,t)-Schnitt W gilt:

   $$\text{VAL}(x) \leq c(\delta^+(W)).$$

BEWEIS:

1. Nach Definition gilt:

$$\begin{aligned}\mathrm{VAL}(x) &= \underbrace{x(\delta^+(s))}_{\sum_{a\in\delta^+(s)} x_a} - x(\delta^-(s)) + \underbrace{\sum_{v\in W\setminus\{s\}} (x(\delta^+(v)) - x(\delta^-(v))}_{\text{nach (2)=0}} \\ &= \sum_{v\in W} (x(\delta^+(v)) - x(\delta^-(v))) \\ &= x(\delta^+(W)) - x(\delta^-(W)).\end{aligned}$$

2. Nach a) gilt:

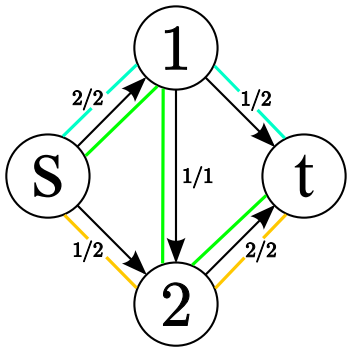
$$\begin{aligned}\mathrm{VAL}(x) &= x(\delta^+(W)) - \underbrace{x(\delta^-(W))}_{\geq 0} \\ &\leq x(\delta^+(W)) \\ &\leq c(\delta^+(W)).\end{aligned}$$

Folgerung 6.4
Der Wert eines minimalen (s,t)-Schnittes ist eine obere Schranke für den Wert eines maximalen Flusses.

Dieser Wert kann tatsächlich erreicht werden, wie wir gleich sehen werden. Eine Idee zur Vorgehensweise liefert uns Proposition 6.1:

Gegeben ein zulässiger (s,t)-Fluss x, z.B. $x = 0$. Erhöhe x auf gerichteten (s,t)-Wegen P mit $x_e < c_e$ für alle $e \in P$. Diese Idee reicht nicht aus, wie folgendes Beispiel zeigt:

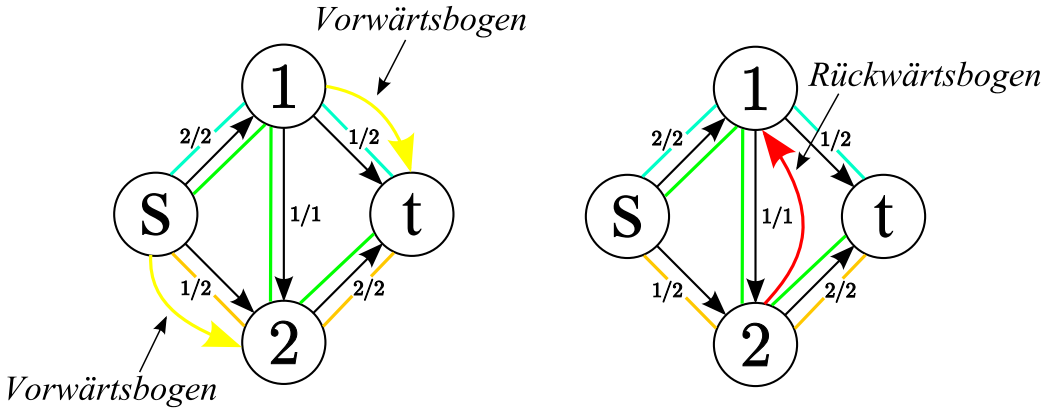
Beispiel:



Der Wert des Flusses ist 3, der minimale Schnitt hat jedoch Wert 4. Jedoch kann der Wert des Flusses mit obiger Idee alleine nicht vergrößert werden.

Definition 6.5
Sei x ein zulässiger (s,t)-Fluss und P ein (ungerichteter) (s,v)-Weg.
Ein Bogen $(i,j) \in P$ heißt **Vorwärtsbogen**, falls i vor j in P und $x_{ij} < c_{ij}$, andernfalls heißt er **Rückwärtsbogen**.
P heißt **augmentierender (s,v)-Weg**, falls $x_{ij} < c_{ij}$ für alle Vorwärtsbogen $(i,j) \in P$ und $x_{ji} > 0$ für alle Rückwärtsbögen $(i,j) \in P$.

Beispiel:



augmentierender Weg $P = (s, (s,2), 2, (1,2), 1, (1,t), t)$

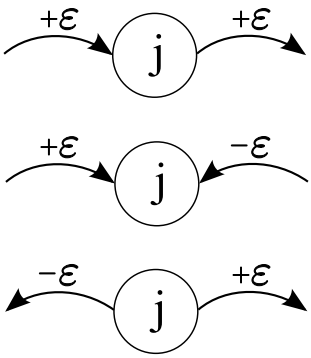
Satz 6.6
Ein (s,t)-Fluss x ist genau dann maximal, wenn es keinen augmentierenden (s,t)-Weg bezüglich x gibt.

Beweis:
„$\Longrightarrow$“ Sei P ein augmentierender (s,t)-Weg bezüglich x. Setze:

$$\varepsilon_{ij} := \begin{cases} c_{ij} - x_{ij} & , \text{ falls } (i,j) \text{ Vorwärtsbogen,} \\ x_{ij} & , \text{ sonst.} \end{cases}$$

Sei $\varepsilon := \min\{\varepsilon_{ij}\}$. Beachte $\varepsilon > 0$.
Dann ist

$$\hat{x}_{ij} := \begin{cases} x_{ij} + \varepsilon & , \text{ falls } (i,j) \in P \text{ Vorwärtsbogen,} \\ x_{ij} - \varepsilon, & , \text{ falls } (i,j) \in P \text{ Rückwärtsbogen,} \\ x_{ij} & , \text{ falls } (i,j) \in A \setminus P. \end{cases}$$



ein zulässiger (s,t)-Fluss mit

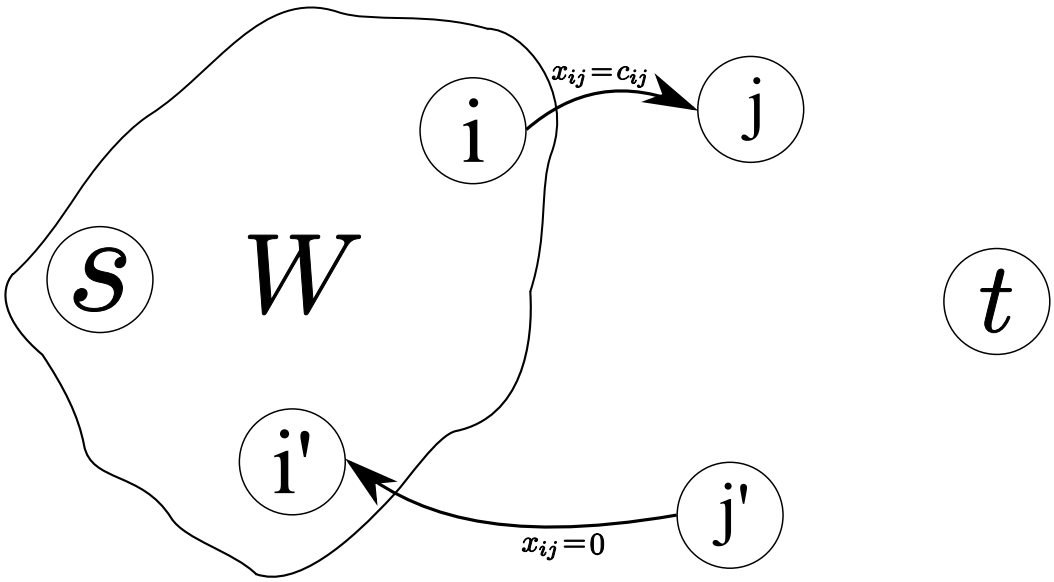
$$\text{VAL}(\hat{x}) = \text{VAL}(x) + \varepsilon \qquad \lightning$$

Also war x nicht maximal.

„$\Longleftarrow$“ Angenommen x besitzt keinen augmentierenden (s,t)-Weg. Sei

$$W := \{v \in V : \exists \text{ augmentierender } (s,v)\text{-Weg bzgl. } x\} \cup \{s\}.$$

Beachte $s \in W$, $t \notin W$.



Ferner gilt nach Definition von W:

$$\begin{aligned} x_a &= c_a \quad \text{für } a \in \delta^+(W), \\ x_a &= 0 \quad \text{für } a \in \delta^-(W). \end{aligned}$$

Wir erhalten:

$$\begin{aligned} \text{VAL}(x) &= x(\delta^+(W)) - x(\delta^-(W)) \\ &= c(\delta^+(W)), \end{aligned}$$

also ist x maximal nach Lemma 6.2. □

Satz 6.7 (Ford & Fulkerson 1956)
Der maximale Wert eines (s,t)-Flusses ist gleich der minimalen Kapazität eines (s,t)-Schnittes.

BEWEIS:
Sei $F = \delta^+(W)$ maximaler (s,t)-Fluss. Nach Lemma 6.3 genügt es einen (s,t)-Schnitt W zu finden mit

$$\text{VAL}(x) = c(\delta^+(W)).$$

Offenbar ist $\delta^+(W)$, mit $W := \{v \in V : \exists \text{ augmentierender } (s,v)\text{-Weg bzgl. } x\} \cup \{s\}$ definiert, ein Schnitt mit dieser Eigenschaft. □

Satz 6.8
Sei D ein Digraph mit ganzzahligen Kapazitäten $c_a \geq 0$ und $s,t \in V, s \neq t$. Dann gibt es einen maximalen (s,t)-Fluss, der ganzzahlig ist.

BEWEIS:
Wir führen den Beweis per Induktion über die Anzahl durchgeführter Augmentierungen. Wir starten mit $x = 0$, der ganzzahlig ist.

Haben wir nun einen ganzzahligen Fluss x, der nicht maximal ist, so bestimmen wir einen augmentierenden Weg P und ε wie im Beweis von Satz 6.6. Nach Voraussetzung ist ε ganzzahlig und damit auch der neue Fluss x.

Bei jeder Augmentierung erhöhen wir den Wert des Flusses um mindestens eins. Da der maximale Fluss endlich ist, folgt die Behauptung. □

Algorithmus 6.9 (Augmentierender-Wege-Algorithmus)

Input:	Digraph $D = (V,A)$ mit Bogenkapazitäten $c_a \geq 0$, $a \in A$ und zwei Knoten $s,t \in V, s \neq t$.
Output:	Ein zulässiger (s,t)-Fluss x mit max. Wert $\text{VAL}(x)$ und ein kapazitätsminimaler (s,t)-Schnitt $\delta^+(W)$.

(1) Bestimme einen zulässigen Fluss, z.B. $x = 0$.

(2) WHILE es existiert ein augmentierender Weg von s nach t DO

(3) Identifiziere einen augmentierenden (s,t)-Weg P.

(4) Bestimme ε durch

$$\varepsilon := \min_{(i,j)\in P} \begin{cases} c_{ij} - x_{ij} & , \text{ falls } (i,j) \text{ Vorwärtsbogen,} \\ x_{ij} & , \text{ falls } (i,j) \text{ Rückwärtsbogen.} \end{cases}$$

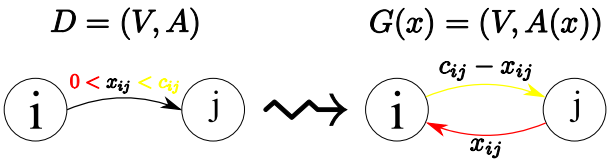
(5) Erhöhe x entlang P um ε:

$$x_{ij} := \begin{cases} x_{ij} + \varepsilon & , \text{ falls } (i,j) \in P \text{ Vorwärtsbogen,} \\ x_{ij} - \varepsilon & , \text{ falls } (i,j) \in P \text{ Rückwärtsbogen,} \\ x_{ij} & , \text{ sonst.} \end{cases}$$

(6) End While

(7) Bestimme $W := \{v \in V : \exists \text{ augmentierender } (s,v)\text{-Weg bzgl. } x\} \cup \{s\}$.

(8) Gib x, $\delta^+(W)$ und $\text{val}(x)$ aus.

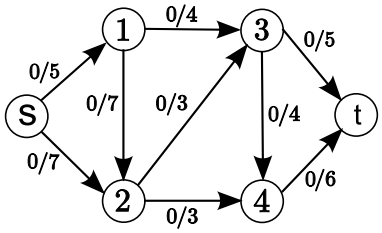


erklärende Grafik zu Schritt (4)

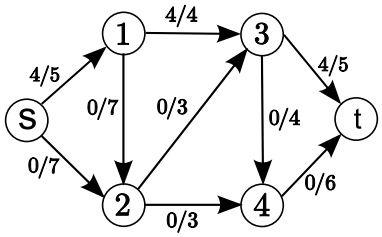
Satz 6.10
Algorithmus 6.8 arbeitet korrekt. Die Laufzeit beträgt $\mathcal{O}(m\nu)$ mit ν = maximaler Flusswert, falls c ganzzahlig ist.

Achtung: $\mathcal{O}(m^2 \cdot C)$ ist nicht polynomial!
In Schritt (3) von Algorithmus 6.9 einen augmentierenden Weg mit BFS bestimmt (kürzesten augm. Weg) $\Longrightarrow$ Laufzeit $\mathcal{O}(m^2 \cdot n)$

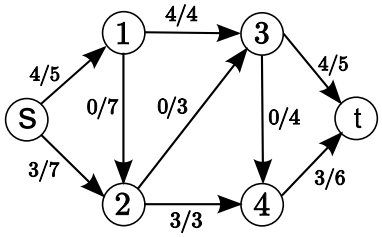
Beispiel: (ohne Rückwärtsbogen)
Gegeben sei das folgende Netzwerk:



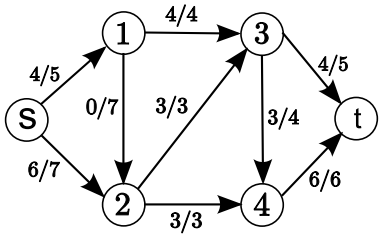
Als erstes suchen wir uns einen augmentierenden Weg von s nach t. Wir wählen den Weg $P = (s, (s,1), 1, (1,3), 3, (3,t), t)$. Unser ε wäre für diesen Weg $\varepsilon = 4$. Wir können über diesen augmentierenden Weg 4 Einheiten schicken. Womit die Kapazitäten nun so aussehen:



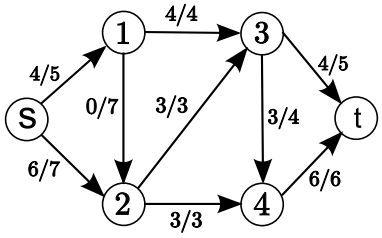
Nun wählen wir wieder einen augmentierenden Weg von s nach t. Diesmal den Weg $P = (s, (s, 2), 2, (2, 4), 4, (4, t), t)$. Diesesmal ist das $\varepsilon = 3$. Wir können also 3 Einheiten über diesen Weg laufen lassen, was wir auch tun. Die Kapazitäten sind nach diesem Durchlauf dann wie folgt verteilt:



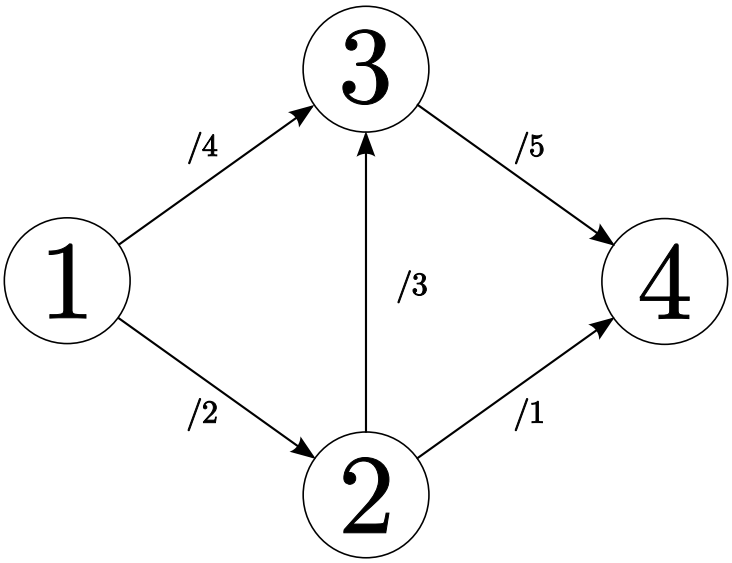
Nun wählen wieder einen augmentierenden Weg, nämlich
$P = (s, (s, 2), 2, (2, 3), 3, (3, 4), 4(4, t), t)$. Das ε wäre auch hier diesmal wieder $\varepsilon = 3$. Die Kapazitäten sind nun folgendermaßen verteilt:



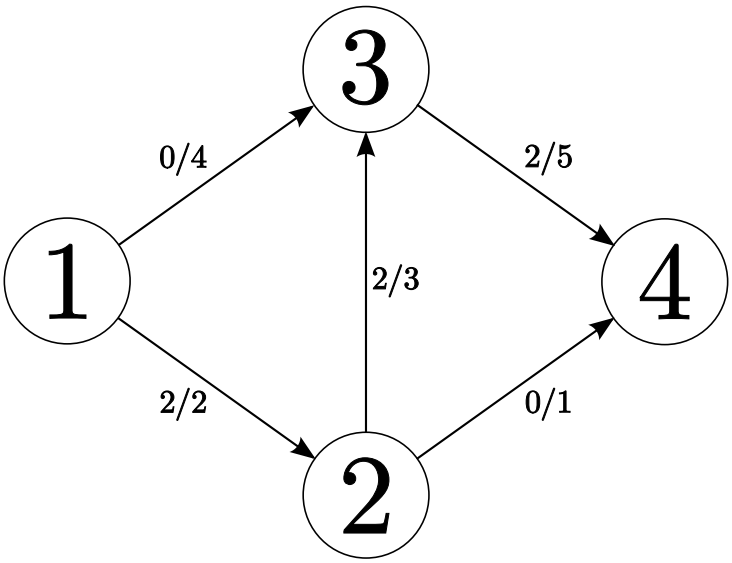
Der minimale Schnitt des Graphen ist 10. $S = \{s, 1, 2\}$ und $\overline{S} = \{3, 4, t\}$ ist ein solcher Schnitt. Unser $\textsc{Val}(x) = 10$. Wir haben also bereits den maximalen Wert erreicht und müssen daher nicht weiter nach augmentierende Wege suchen, sondern können hier aufhören. Unser maximale Fluss hat also $\textsc{Val} = 10$ und die Kapazitäten sind folgendermaßen verteilt:



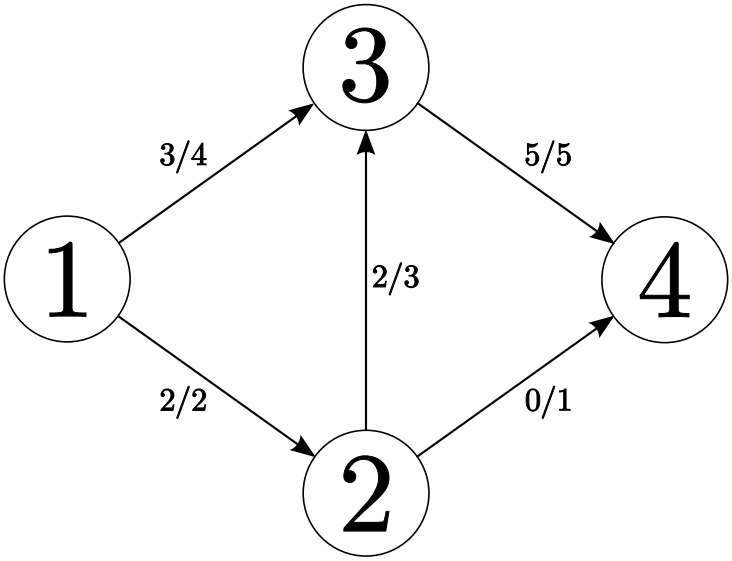
Beispiel: (mit Rückwärtsbogen)
gegeben sei folgendes Netzwerk:



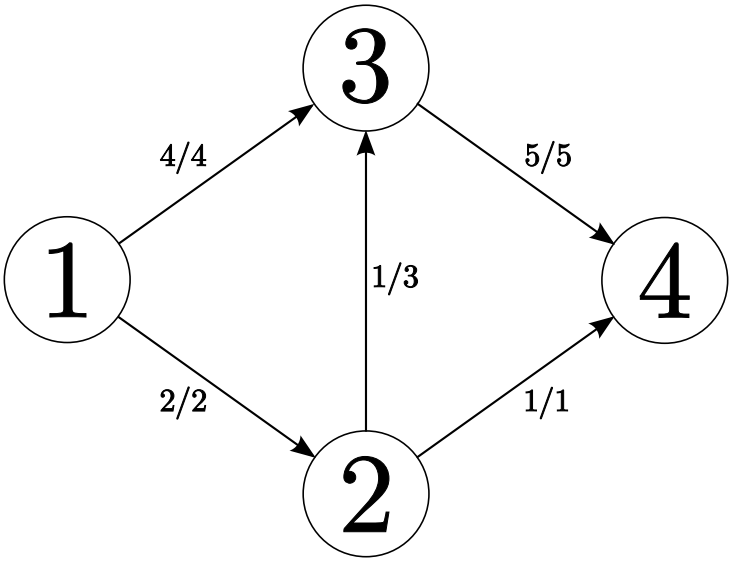
Wie im vorigen Beispiel suchen wir wieder einen augmentierenden Weg von 1 nach 4. Wir wählen den Weg $P = (1, (1, 2), 2, (2, 3), 3, (3, 4), 4)$. Das ε für diesen Weg ist $\varepsilon = 2$, wir können also 2 Einheiten über den Weg P schicken. Die Kapazitäten sind nun wie folgt verteilt:



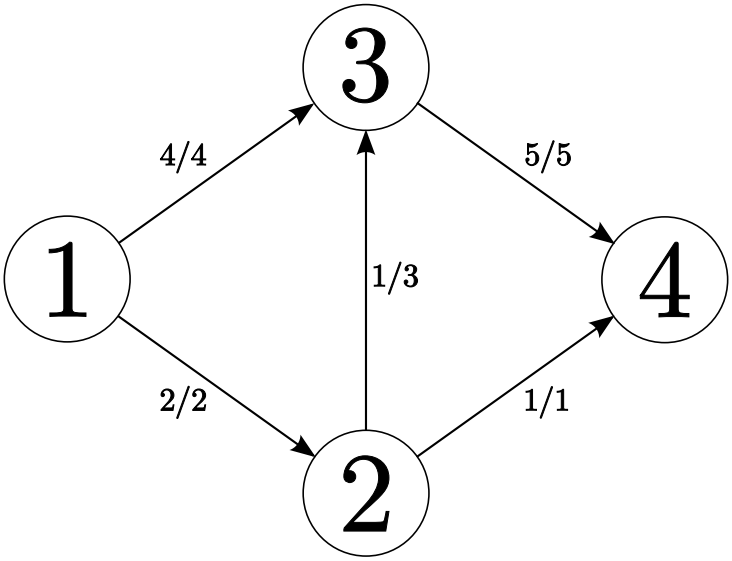
Erneut suchen wir uns einen augmentierenden Weg von 1 nach 4, wofür den Weg $P = (1, (1, 3), 3, (3, 4), 4)$ wählen. Diesmal ist $\varepsilon = 3$. Wir können also 3 Einheiten über diesen Weg schicken. Nun sieht die Kapazitätenverteilung folgendermaßen aus:



Der nächste augmentierende Weg den wir wählen ist der Weg $P = (1, (1, 3), 3, (3, 2), 2, (2, 4), 4)$ mit dem Rückwärtsbogen $(3, 2)$. Unser ε ist hier $\varepsilon = 1$. Die Kapazitäten sind dann folgendermaßen verteilt:



Der minimale Schnitt des Graphen ist 6. $S = \{1\}$ und $\overline{S} = \{2, 3, 4\}$ ist ein solcher Schnitt. Unser $textVAL(x) = 6$. Wir haben bereits den maximalen Wert erreicht und müssen nicht weiter nach augmentierenden Wegen suchen und sind damit fertig. Der maximale Fluss hat $textVAL(x) = 6$ und die Kapazitäten sind folgendermaßen verteilt:


3
4/4
5/5
1
1/3
4
2/2
1/1
2

7 Sortieren

In diesem Kapitel wollen wir uns mit Sortierproblemen beschäftigen, das heißt mit dem Problem, gegeben n Zahlen $a_1, \ldots, a_n$, finde eine Permutation $\pi : \{1, \ldots, n\} \mapsto \{1, \ldots, n\}$ von $a_1, \ldots, a_n$, so dass $a_{\pi(1)} \leq a_{\pi(2)} \leq \ldots \leq a_{\pi(n)}$.

In der Regel sind die Zahlen nicht isolierte Elemente, sondern Teil komplexer Strukturen oder Objekte. Jedes dieser Objekte enthält einen Schlüssel (engl. „key“), nach dem sortiert werden soll. Diese Schlüssel sind häufig Zahlen, können aber auch Characters, Strings oder dergleichen sein. Entsprechend muss man im Allgemeinen eine Ordnungsrelation auf diesen Elementen definieren. Wir beschränken uns in diesem Kapitel jedoch auf die „einfachste“ Form, nämlich auf ganze Zahlen als Schlüssel und auf die herkömmliche Ordnung auf Zahlen.

Sortieren an sich ist ein sehr großes Gebiet, es gibt ganze Bücher, die sich rein mit Sortieralgorithmen beschäftigen. Wir wollen in diesem Rahmen nur auf einige wichtige Algorithmen eingehen.

7.1 Sortieren durch Auswahl

Das wohl einfachste Verfahren zum Sortieren ist, dass man sich zunächst das (oder ein) kleinste Element sucht, dieses an die erste Stelle tauscht, danach das zweitkleinste unter den verbleibenden Zahlen sucht und dieses an die zweite Stelle tauscht. Dieses Vorgehen wiederholt man genau n mal, bis alle Elemente sortiert sind. Hier der Algorithmus.

Algorithmus 7.1 (Sortieren durch Auswahl)
Input: Ein Array a der Länge n mit $a[i] \in \mathbb{Z}$.
Output: Das Array a mit $a[1] \leq a[2] \leq \ldots \leq a[n]$.

(1) For $i = 1$ To n Do
  (2) $\min = i$.
  (3) For $j = i + 1$ To n Do
    (4) If $a[j] < \min$ Then $\min = j$.
  (5) End For

(6) Tausche $a[\min]$ und $a[i]$.

(7) End For

(8) Gib a aus.

Beachte, dass wir die Elemente innerhalb des Arrays getauscht haben. Möchte man die Elemente innerhalb des Arrays nicht tauschen, zum Beispiel weil sie Teil größerer Objekte oder Strukturen sind, so führt man wie bei der Definition des Sortierproblems einen zusätzlichen Permutationsvektor π mit, initialisiert diesen mit $\pi(i) = i$ und führt die Tauschoperationen auf diesem durch.

Die Laufzeit von Algorithmus 7.1 ist offensichtlich $\mathcal{O}(n^2)$ und der Algorithmus ist stabil, das heißt Elemente mit gleichen Wert behalten ihre relative Ordnung, und er läuft in-place, das heißt der zusätzliche Speicherbedarf ist konstant $\mathcal{O}(1)$. Der folgende Algorithmus hat ähnliche Eigenschaften.

7.2 Bubble Sort

Seinen Namen hat der folgende Algorithmus von der Eigenschaft, dass er im ersten Durchlauf das „schwerste“ Element sucht und dieses nach unten „durchblubbern“ lässt. Im zweiten Durchlauf sucht man sich unter den verbleibenden $n - 1$ Elementen das zweitschwerste, lässt dieses auch nach unten „blubbern“ usw., bis alle an die richtige Stelle „geblubbert“ sind.

Algorithmus 7.2 (Bubble Sort)
Input: Ein Array a der Länge n mit $a[i] \in \mathbb{Z}$.
Output: Das Array a mit $a[1] \leq a[2] \leq \ldots \leq a[n]$.

(1) For $i = 1$ To $n - 1$ Do

(2) For $j = 1$ To $n - i$ Do

(3) If $a[j] < a[j + 1]$ Then Tausche $a[j]$ und $a[j + 1]$.

(4) End For

(5) End For

(6) Gib a aus.

Dieser Algorithmus hat die gleichen Charakteristiken wie Algorithmus 7.2, das heißt seine Laufzeit ist $\mathcal{O}(n^2)$, er ist stabil und läuft in-place.

Beide bisher betrachteten Algorithmen haben eine Laufzeit von $\mathcal{O}(n^2)$. Es stellt sich die Frage, ob es nicht besser (schneller) geht. Es geht. Die beiden folgenden Algorithmen sind Beispiele hierfür.

7.3 Quick Sort

Die Idee des ersten, **Quick Sort** genannten, Algorithmus, ist es sich ein (beliebiges) Element r herauszugreifen und anschließend dafür zu sorgen, dass alle Elemente i links von r (d.h. $i \leq r$) kleiner sind als $a[r]$ und alle Elemente, die rechts von r sind (d.h. $j \geq r$) größer sind als r. Danach werden beide Hälften nach dem gleichen Prinzip (also rekursiv) behandelt. Im Detail sieht der Algorithmus wie folgt aus.

Algorithmus 7.3
Input: Ein Array a der Länge n mit $a[i] \in \mathbb{Z}$,
untere und obere Grenzen l, r mit $1 \leq l \leq r \leq n$.
Output: Das Array a mit $a[l] \leq a[l+1] \leq \ldots \leq a[r]$.

(1) Setze $i = l - 1$ und $j = r$.

(2) `While` $i < j$ `Do`

(3) `Do` i = i + 1 `While` $a[i] \leq a[r]$.

(4) `Do` j = j - 1 `While` ($a[j] \geq a[r]$ und $j \geq i$).

(5) Tausche $a[j]$ und $a[i]$.

(6) `End While`

(7) Tausche $a[i]$ und $a[r]$.

(8) `If` $l < i - 1$ `Then` QuickSort $(a, l, i-1)$.

(9) `If` $i + 1 < r$ `Then` QuickSort $(a, i+1, r)$.

(10) Gib a aus.

Der erste Aufruf erfolgt mit QuickSort(a,1,*length*(a)).

Satz 7.4
Die durchschnittliche Laufzeit des Quicksort beträgt $\mathcal{O}(n\log n)$.
Näheres in der Übung.

7.4 Heap Sort

Definition 7.5
Ein **Heap** ist ein binärer Baum, in dem für den Eintrag $a[i]$ eines jeden Knoten i des Baumes gilt:

$$a[LinkerSohn(i)] \leq a[i]$$
$$a[RechterSohn(i)] \leq a[i]$$

Wir sagen ein Array a hat die **Heap-Eigenschaft** im Bereich $[l, r]$, falls gilt:

$$\begin{array}{lllll} l \leq i \leq r & \text{und} & l \leq 2i \leq r & \Longrightarrow & a[i] \geq a[2i] \\ l \leq i \leq r & \text{und} & l \leq 2i+1 \leq r & \Longrightarrow & a[i] \geq a[2i+1]. \end{array}$$

Die Idee ist nun, zunächst die Heap-Eigenschaft für das gegebene Array a herzustellen. Danach haben wir das größte Element in $a[1]$ stehen. Nun tauschen wir das erste und das letzte Element und stellen die Heap-Eigenschaft wieder für die Elemente von 1 bis $n-1$ her. Dies wiederholen wir insgesamt n-mal, bis das gesamte Array sortiert ist. Dies liest sich detailliert wie folgt:

Algorithmus 7.6 (Heap Sort)
Input: Ein Array a der Länge n mit $a[i] \in \mathbb{Z}$.
Output: Das Array a mit $a[1] \leq a[2] \leq \ldots \leq a[n]$.

(1) For $i = \lfloor \frac{n}{2} \rfloor$ To 1 Do
  (2) heapify (a, i, n).
(3) End For
(4) For $i = n$ To 2 Do
  (5) Tausche $a[1]$ und $a[i]$.
  (6) heapify $(a, 1, i-1)$.
(7) End For
(8) Gib a aus.

heapify (a, l, r)

(1) Setze $f = l$.
(2) Setze $s = 2 \cdot f$.
(3) While $s \leq r$ Do
  (4) If $s+1 \leq r$ und $a[s] \leq a[s+1]$ Then
    (5) Setze $s = s+1$.
  (6) If $a[f] \leq a[s]$ Then
    (7) Tausche $a[f]$ und $a[s]$.
    (8) Setze $f = s$.
    (9) Setze $s = 2 \cdot f$.
  (10) Else Stop.

(11) End While

Satz 7.7
Die Laufzeit von Algorithmus 7.4 ist $\mathcal{O}(n \log n)$.

Satz 7.8
Die Laufzeit eines Algorithmus, der auf paarweisen Vergleichen beruht, ist $\mathcal{O}(\log n!)$.

Stirling Formel:

$$\sqrt{2\pi n} \cdot \left(\frac{n}{e}\right)^n \leq n! \leq \sqrt{2\pi n} \cdot \left(\frac{n}{e}\right)^{n+\frac{1}{12n}}$$

Satz 7.9
Jeder (deterministische) Sortieralgorithmus, der auf paarweisen Vergleichen der Schlüssel basiert, braucht im Worst Case $\otimes(n \log n)$ Vergleiche.

7.5 Bucket Sort

Die Idee ist, auf die Werte der Einträge selbst zu schauen. Nehmen wir einmal an, wir wüssten, dass alle auftretenden Zahlen nur zwischen 1 und 100 liegen würden. Dann könnten wir 100 Eimer bilden (engl. bucket), wobei wir in Eimer i alle Array-Elemente j nehmen mit $a[j] = i$. Nun laufen wir einmal durch das gesamte Array und weisen jedes Element seinem entsprechenden Eimer zu. Danach laufen wir noch einmal durch alle Eimer (beachte, innerhalb eines Eimers ist die Anordnung der Elemente egal, da sie alle denselben Wert haben) und ordnen die Elemente in der auftretenden (sortierten) Reihenfolge an. Die Laufzeit dieses Algorithmus ist offensichtlich linear, also $\mathcal{O}(n)$.

Im Allgemeinen ist es natürlich nicht so einfach, da das größte Element sehr groß sein kann. Damit muss die Anzahl Buckets reduziert werden, was dazu führen kann, dass innerhalb der einzelnen Buckets nochmal sortiert werden muss. Ziel ist es also, eine Anzahl B an Buckets zu finden, wobei B linear in n sein sollte, und eine Funktion $f : \{a[1], \ldots, a[n]\} \mapsto \{1, \ldots, B\}$ mit folgenden Eigenschaften zu finden:

(a) $f(a[i]) < f(a[j]) \implies a[i] \leq a[j]$ für alle i, j.

(b) $|\{a[i] : i \in \{1, \ldots, n\} \text{ und } f(a[i]) = b\}| \leq \text{const}$ für alle $b = 1, \ldots, B$.

Eigenschaft (a) ist Voraussetzung, um abschließend in einem Durchlauf durch die Buckets die endgültige Sortierung zu erhalten. Eigenschaft (b) ist dagegen nur notwendig, wenn man am Ende lineare Laufzeit erzielen will, siehe unten.

Wir wollen im Rahmen dieser Vorlesung nicht genauer auf solche Funktionen und deren Eigenschaften eingehen, sondern lediglich den allgemeinen Rahmen dieser Bucket Sorts im folgenden Algorithmus vorstellen.

Algorithmus 7.10 (Bucket Sort)
Input: Ein Array a der Länge n mit $a[i] \in \mathbb{Z}$.
Output: Das Array a mit $a[1] \leq a[2] \leq \ldots \leq a[n]$.

(1) Bestimme Anzahl Buckets B und eine Funktion $f : \{a[1], \ldots, a[n]\} \mapsto \{1, \ldots, B\}$.

(2) For $i = 1$ To n Do

  (3) Weise $a[i]$ Bucket $f(a[i])$ zu.

(4) End For

(5) For $b = 1$ To B Do

  (6) Sortiere Bucket b.

(7) End For

(8) Kopiere Elemente aus den Buckets in a zurück.

(9) Gib a aus.

Allgemein kann keine Aussage über die Laufzeit von Algorithmus 7.8 gemacht werden. Hat jeder Bucket konstante Größe, so läuft der Bucket Sort jedoch in linearer Zeit, also ist er schneller als die Sortieralgorithmen, die auf paarweises Vergleichen beruhen. Abschließend fassen wir in folgender Tabelle nochmal alle Algorithmen zusammen.

Algorithmus	stabil	in-place	asymp. Laufzeit
Sortieren durch Auswahl	ja	ja	$\mathcal{O}(n^2)$
Bubble-Sort	möglich	ja	$\mathcal{O}(n^2)$
Quick-Sort	?	nein (wegen Rekursion)	$\mathcal{O}(n \log n)$ bzw. $\mathcal{O}(n^2)$
Heap-Sort	?	nein (wegen Rekursion)	$\mathcal{O}(n \log n)$
Bucket-Sort	ja	nein($\mathcal{O}(n)$ Speicher)	$\mathcal{O}(n \cdot m)$

8 Algorithmische Prinzipien

Definition 8.1 (Rucksackproblem)
Gegeben sei eine Menge N von Gegenständen. Jeder Gegenstand $i \in N$ hat ein Gewicht $a_i \in \mathbb{N}$ sowie einen Profit $c_i \in \mathbb{N}$. Darüber hinaus gibt es eine Zahl $b \in \mathbb{N}$, die sogenannte Rucksackkapazität. Finde eine Auswahl an Gegenständen, deren gewichtete Summe die Rucksackkapazität nicht übersteigt und den Profit maximiert.

Will man das **Rucksackproblem** mathematisch modellieren, so erhält man ein spezielles ganzzahliges Programm:

Variablen:
$x_i = \begin{cases} 1 & \text{, falls Projekt } j \text{ gewählt wird,} \\ 0 & \text{, sonst.} \end{cases}$

$$\max \sum_{i=1}^{n} c_i x_i$$
$$\sum_{i=1}^{n} a_i x_i \leq b$$
$$x_i \in \{0,1\}, \text{ für } i = 1, \ldots, n.$$

8.1 Der Greedy-Algorithmus

Definition 8.2
Sei E eine endliche Grundmenge. Eine Menge $\mathcal{I} \subseteq \mathcal{P}(E)$ heißt **Unabhängigkeitssystem**, falls mit $G \subseteq F \in \mathcal{I}$ auch $G \in \mathcal{I}$ folgt.

Jede Menge $F \in \mathcal{I}$ heißt **unabhängig**, alle anderen Mengen **abhängig**. Ist $F \subseteq E$, so heißt eine unabhängige Teilmenge von F **Basis von** F, falls sie in keiner anderen unabhängige Teilmenge von F enthalten ist. Sei $c : E \mapsto \mathbb{R}$ eine Gewichtsfunktion auf E. Das Problem

(1) $$\max_{I \in \mathcal{I}} c(I)$$

heißt **Maximierungsproblem über einem Unabhängigkeitssystem**.

(2) $$\min_{B \text{ Basis}} c(B)$$

heißt **Minimierungsproblem über einem Basissystem**.

Algorithmus 8.3 (Greedy-Min für Basissystem)
Input: Ein Problem der Form $\min_{B \text{ Basis}} c(B)$.
Output: Basis B_{Greedy} von E.

(1) Sortieren: Wähle Funktion $w : E \mapsto \mathbb{R}$ und sortiere E, so dass

$$w(e_1) \leq w(e_2) \leq \ldots \leq w(e_m).$$

(2) Setze $B_{\text{Greedy}} = \emptyset$.

(3) For $k = 1$ To m Do

Falls $B_{\text{Greedy}} \cup \{e_k\}$ Basis ist oder eine Basis enthält setze

$$B_{\text{Greedy}} = B_{\text{Greedy}} \cup \{e_k\}.$$

(4) End For

(5) Gib B_{Greedy} aus.

Definition 8.3
Ein Unabhängigkeitssystem das folgende Eigenschaft erfüllt (*) heißt **Matroid**.
Für alle $F \subseteq E$ mit B, B' Basen von F. $\Longrightarrow |B| = |B'|$ (*).

Definition 8.4
Für $F \subseteq E$ bezeichne

$$r(F) := \max\{|B| : B \text{ ist Basis von } F\}$$

den **Rang** von F sowie

$$r_{\text{u}}(F) := \min\{|B| : B \text{ ist Basis von } F\}$$

den **unteren Rang** von F.

Für Matroide gilt offensichtlich:

$$r_{\text{u}}(F) = r(F).$$

Beispiel:
Rucksack:
$r_u(F) = 1$, $r(F) = 2$

Satz 8.1 (Jenkyns, 1976)

$$\min_{F \subseteq E} \frac{r_{\mathrm{u}}(F)}{r(F)} \leq \frac{c(I_{\mathrm{Greedy}})}{c(I_{\mathrm{OPT}})} \leq 1$$

und für jedes Unabhängigkeitssystem gibt es Gewichte $c_i \in \{0, 1\}$ für $i \in E$, so dass die erste Ungleichung mit Gleichheit angenommen wird.

Beweis:
o.B.d.A. gelte $c_i > 0$ für $i = 1, \ldots, n$ und $c_1 \geq c_2 \geq \ldots \geq c_n$.
Wir führen ein $(n+1)$tes-Element ein mit $c_{n+1} = 0$ und setzen

$$E_i = \{1, \ldots, i\} \quad \text{sowie} \quad q = \min_{F \subseteq E} \frac{r_{\mathrm{u}}(F)}{r(F)}$$

Für $F \subseteq E$ gilt:

$$c(F) = \sum_{i \in F} c_i = \sum_{i \in F} \left(\sum_{j=i}^{n} (c_j - c_{j+1}) \right) = \sum_{i=1}^{n} |F \cap E_i| \cdot (c_i - c_{i+1})$$

Da $I_{\mathrm{OPT}} \cap E_i \subseteq I_{\mathrm{OPT}}$ gilt $I_{\mathrm{OPT}} \cap E_i \in \mathcal{I}$, und somit $|I_{\mathrm{OPT}} \cap E_i| \leq r(E_i)$.
Die Vorgehensweise des Greedy-Algorithmus impliziert, dass $I_{\mathrm{Greedy}} \cap E_i$ eine Basis von E_i ist, also $|I_{\mathrm{Greedy}} \cap E_i| \geq r_{\mathrm{u}}(E_i)$.
Damit gilt:

$$|I_{\mathrm{Greedy}} \cap E_i| \geq |I_{\mathrm{OPT}} \cap E_i| \cdot \frac{r_{\mathrm{u}}(E_i)}{r(E_i)} \geq |I_{\mathrm{OPT}} \cap E_i| \cdot q$$

und

$$\begin{aligned} c(I_{\mathrm{Greedy}}) &= \sum_{i=1}^{n} |I_{\mathrm{Greedy}} \cap E_i| \cdot (c_i - c_{i+1}) \\ &\geq \sum_{i=1}^{n} |I_{\mathrm{OPT}} \cap E_i| \cdot q \, (c_i - c_{i+1}) \\ &= q \sum_{i=1}^{n} |I_{\mathrm{OPT}} \cap E_i| \cdot (c_i - c_{i+1}) \\ &= q \cdot c \, (I_{\mathrm{OPT}}) \end{aligned}$$

Die erste Ungleichung der Behauptung wäre damit bewiesen. Die zweite Ungleichung ist trivial, da jede Lösung höchstens schlechter als die Optimallösung sein kann.

Es verbleibt zu zeigen, dass das Minimum auch angenommen wird:
Sei nun $F \subseteq E$ mit $q = \frac{r_{\mathrm{u}}(F)}{r(F)}$. o.B.d.A. sei $F = \{1, \ldots, k\}$ und $B = \{1, \ldots, p\} \subseteq F$ eine Basis von F mit $|B| = r_{\mathrm{u}}(F)$. Setze $c_i = 1$ für $i = 1, \ldots, k$ und $c_i = 0$ sonst.
Der Greedy-Algorithmus liefert $I_{\mathrm{Greedy}} = B$ mit $c(I_{\mathrm{Greedy}}) = r_{\mathrm{u}}(F) = p$, während für jede Optimallösung I_{OPT} gilt $c(I_{\mathrm{OPT}}) = r(F)$. □

Folgerung 8.1
Sei $\mathcal{I}$ ein Unabhängigkeitssystem auf E. Dann sind äquivalent:

(i) $\mathcal{I}$ ist ein Matroid.

(ii) Greedy liefert optimale Lösung für alle $c \in \mathbb{R}^E$.

(iii) Greedy liefert optimale Lösung für alle $c \in \{0,1\}^E$