Graphen

Teil II: Graphen

1. Einführung
2. Wege und Kreise in Graphen, Bäume
3. Planare Graphen / Traveling Salesman Problem
4. Transportnetzwerke




1. Einführung

- Graphen
- Gerichtete / ungerichtete Graphen
- Isomorphie
- Grad eines Graphen
- Gradsequenz





Graphen

Graphen sind Objekte, bestehend aus **Knoten** und **Kanten**

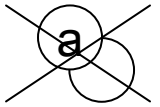
Formaler:

Ein Graph G wird spezifiziert durch

- V – Menge der Knoten („vertices“)
- E – Menge der Kanten („edges“)

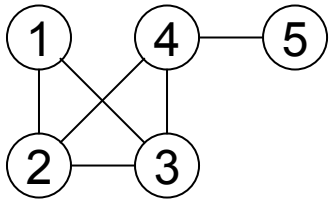
mathematisch verkürzt: G = (V, E)

Kanten sind 2-elementige Mengen!
(Es gibt also keine ungerichteten Schlingen)
(gerichtet heißt: mit Pfeilen)



Beispiel:

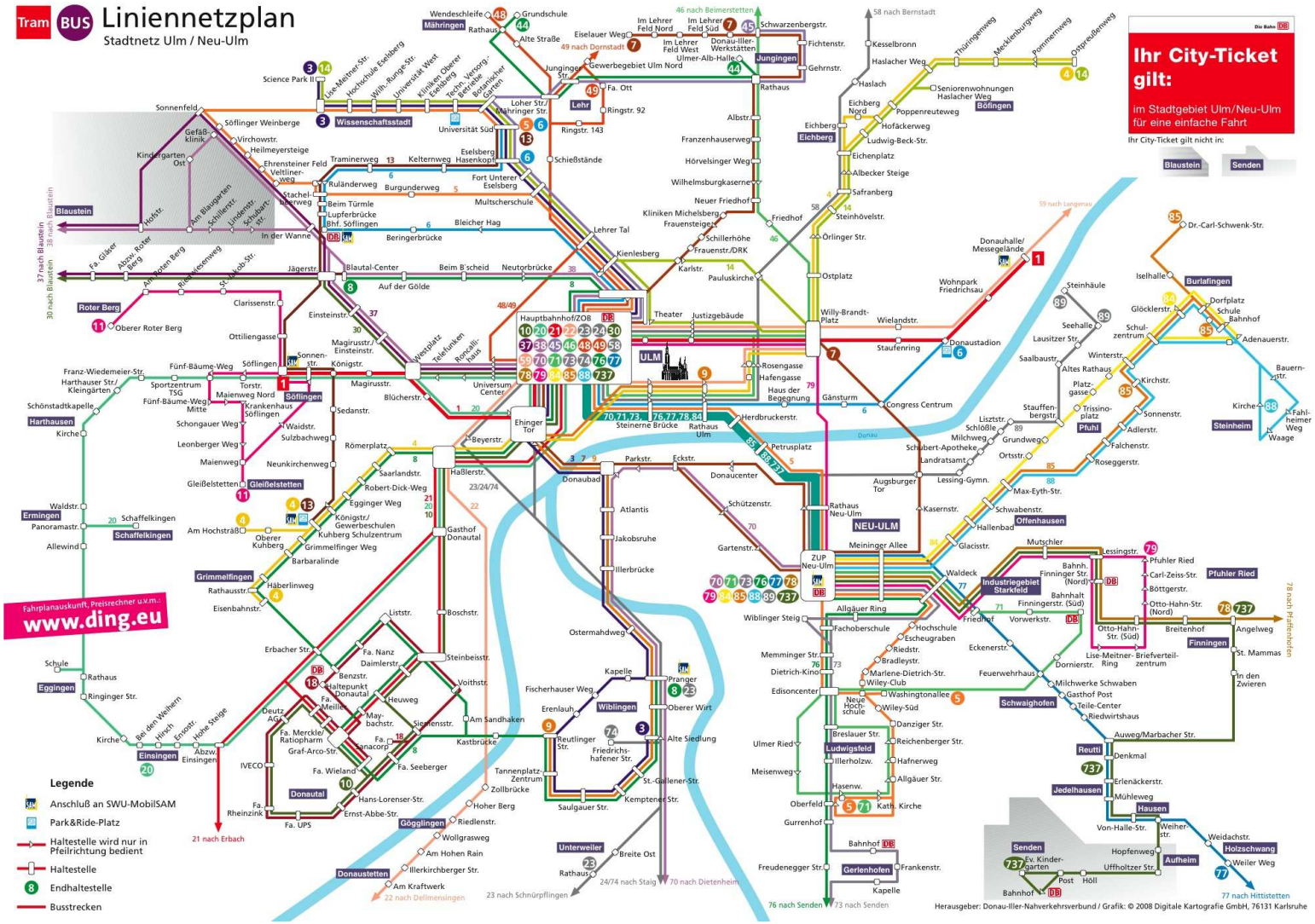
- V = {1, 2, 3, 4, 5}
- E = {{1, 2}, {1, 3}, {2, 4}, {2, 3}, {3, 4}, {4, 5}}



Die Reihenfolge spielt (bei ungerichteten Graphen) keine Rolle. (Bei gerichteten Graphen aber sehr wohl.)

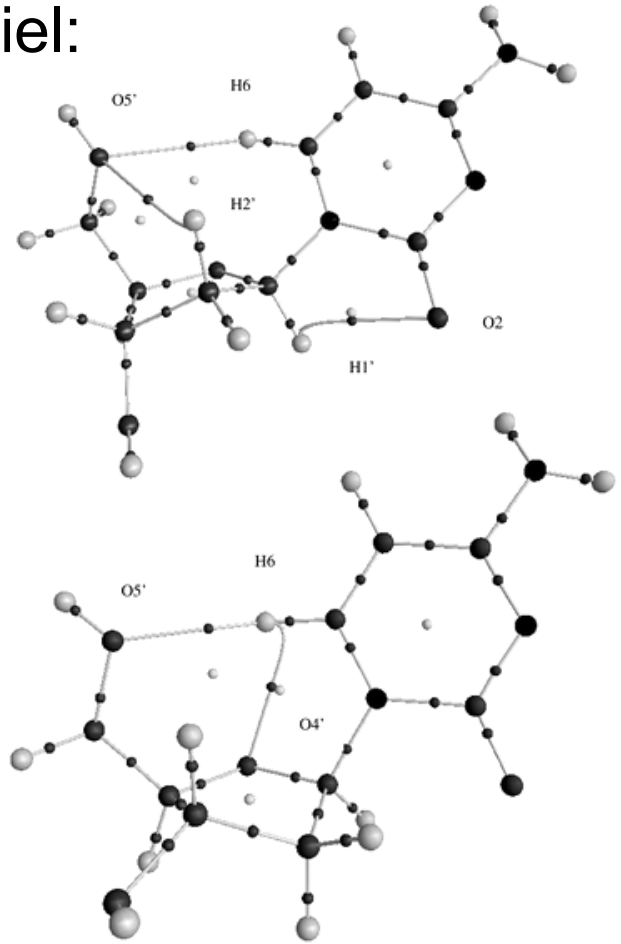
Ungerichtete Graphen

Beispiel:



Ungerichtete Graphen (1)

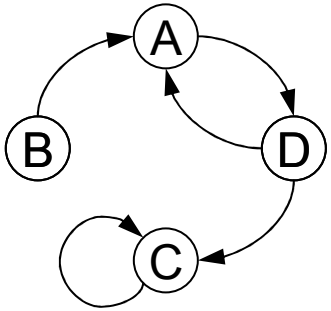
Beispiel:



Molekül - allgemein

Gerichtete Graphen

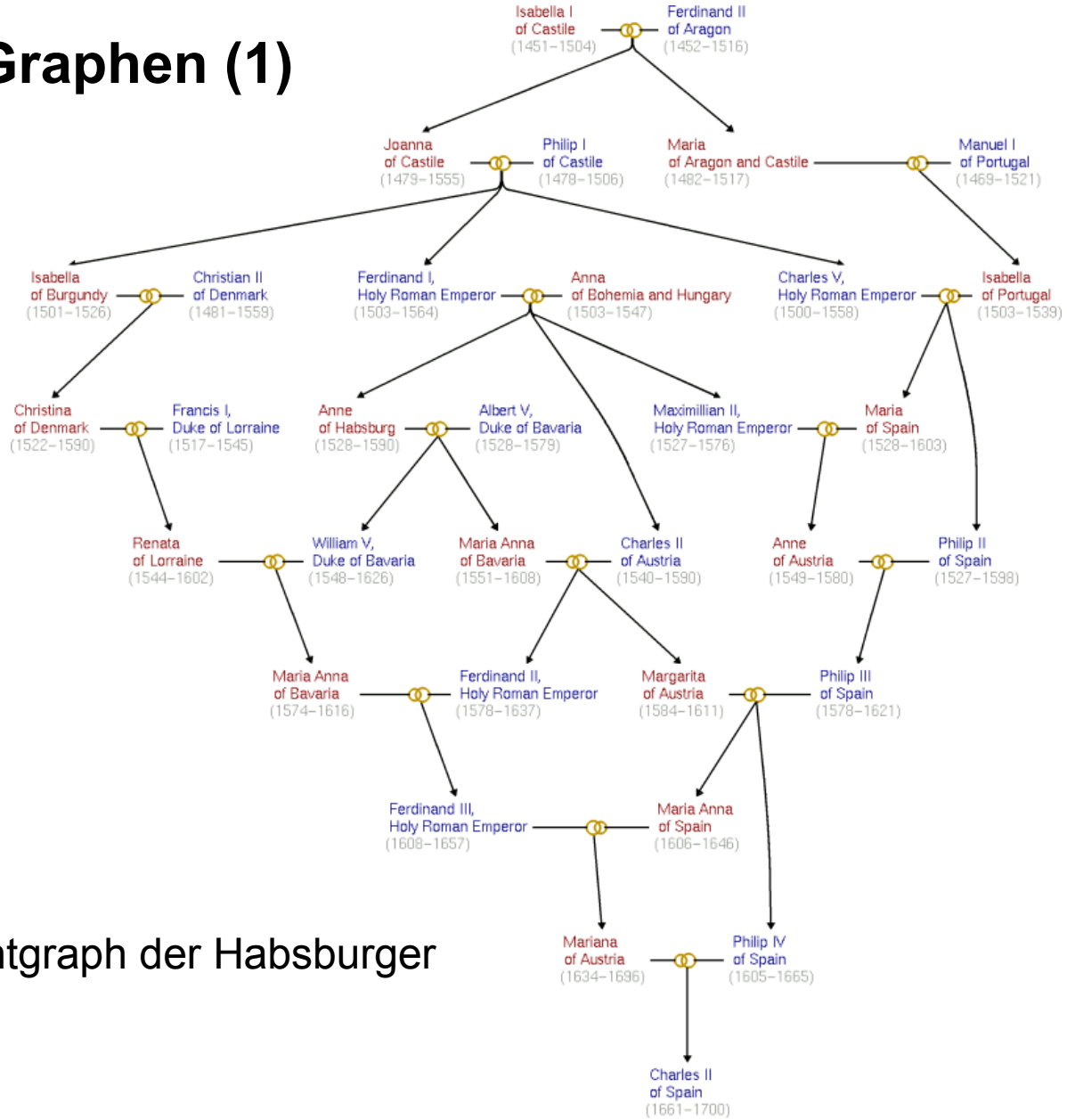
Beispiel:



Hier werden runde Klammern verwendet, da die Reihenfolge wichtig ist.

- $G = (\underbrace{\{A, B, C, D\}}_{= V}, \underbrace{\{(B, A), (A, D), (D, A), (D, C), (C, C)\}}_{= E \quad \text{(Menge von geordneten Paaren)}})$

Gerichtete Graphen (1)



Inzuchtgraph der Habsburger

Gerichtete Graphen (2)

- Gerichtete Graphen sind besonders zur Beschreibung von Zustandsübergängen geeignet.

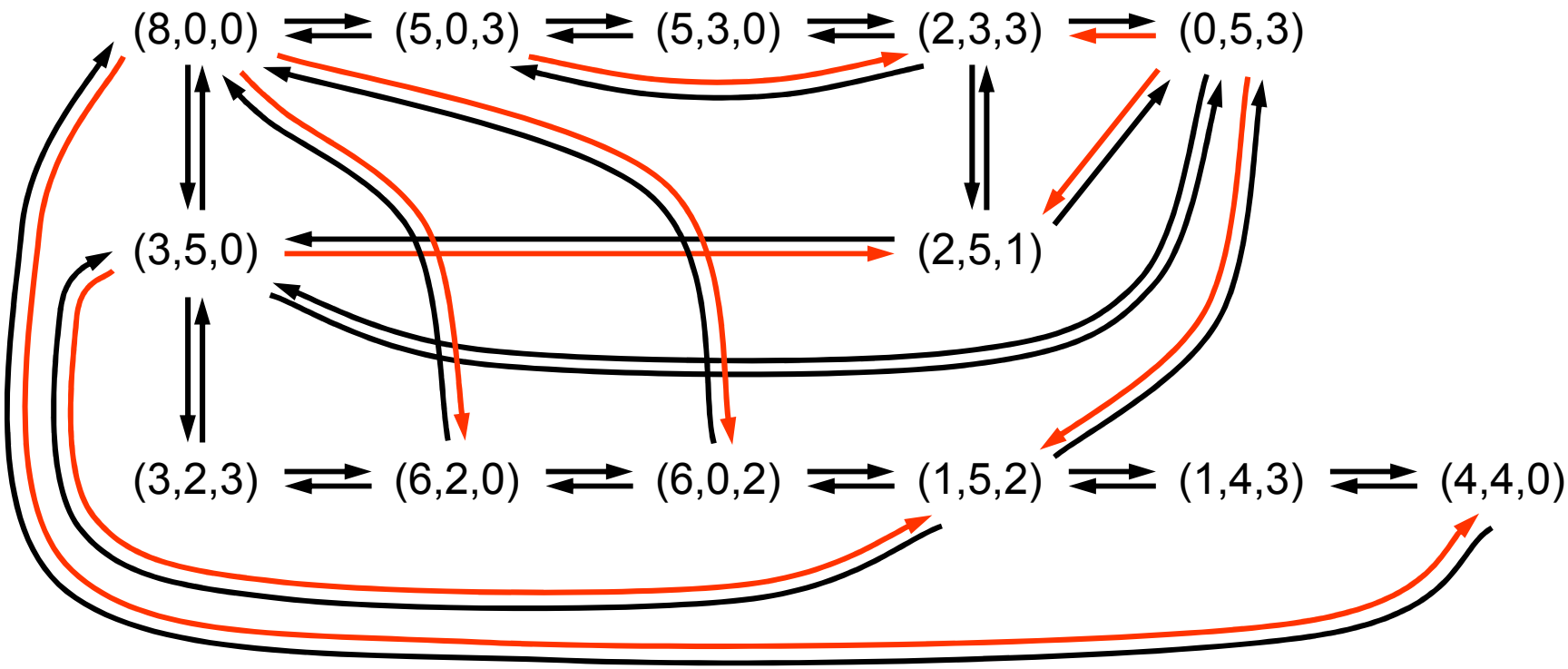
Umfüllproblem:

- Es sind 8l Flüssigkeit vorhanden, des Weiteren stehen ein 8l, 5l und ein 3l Behälter zur Verfügung.

- Aufgabe: Abzweigen von 4l ohne Schätzen

Gerichtete Graphen (3)

Systematische Lösung durch Umfüllgraph



1. Beobachtung: Übersichtliches Zeichnen von Graphen ist Problem für sich (Layoutproblem)
2. Beobachtung: Problemlösung durch Wegefindung

Isomorphie

Definition:

- Zwei Graphen $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ heißen **isomorph**, wenn sie "derselbe Graph" sind – bis auf die Bezeichnung der Knoten.

Formal:

- Es gibt eine Funktion $f : V_1 \rightarrow V_2$, die bijektiv (eineindeutig) ist, so dass gilt: $\{x, y\} \in E_1 \Leftrightarrow \{f(x), f(y)\} \in E_2$

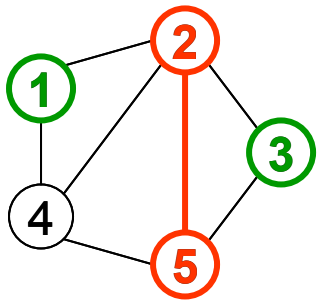
Beispiel:

- Hier ist: $f : 1 \rightarrow b$, $2 \rightarrow a$, $3 \rightarrow e$, $4 \rightarrow c$, $5 \rightarrow d$
- **$\{2, 5\} \in E_1$ und $\{\underbrace{f(2)}_{a}, \underbrace{f(5)}_{d}\} \in E_2$**
- **$\{1, 3\} \notin E_1$ und $\{\underbrace{f(1)}_{b}, \underbrace{f(3)}_{e}\} \notin E_2$**

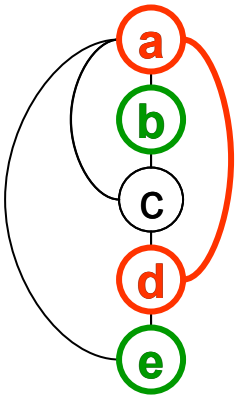
entweder sind beide $\in$ oder beide sind $\notin$

Beispiel:

G_1:



G_2:

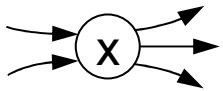


Grad eines Knoten

Definition:

- Falls $\{x, y\} \in E$, so heißen x und y **Nachbarn**.
  Falls $(x, y) \in E$ (bei gerichteten Graphen), so ist x **Vorgänger** von y und y **Nachfolger** von x.
- Sei d(x) die Anzahl der Kanten, die x enthalten ("degree", **Grad**).
  Bei gerichteten Graphen unterscheidet man zusätzlich **Eingangsgrad** $d_{in}(x)$ und **Ausgangsgrad** $d_{out}(x)$.

Beispiel:



$d(x) = 5,\ d_{in}(x) = 2,\ d_{out}(x) = 3$

- Im ungerichteten Graphen ist:

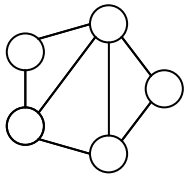
$$\sum_{x \in V} d(x) = 2 \cdot |E|$$

Gradsequenz

Definition:

- Die **Gradsequenz** eines Graphen ist die Folge seiner **Knotengrade** d(x), absteigend sortiert.

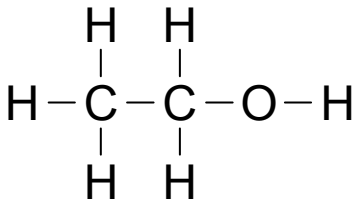
- Beispiel:



Gradsequenz:
(4, 3, 3, 2, 2)

- Beispiel (Chemie):

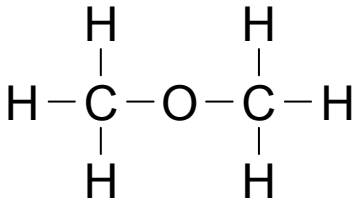
2 x C (Grad = 4)
1 x O (Grad = 2)
6 x H (Grad = 1)



Gradsequenz:
(4, 4, 2, 1, 1, 1, 1, 1, 1)

(Alkohol)

Beide Verbindungen besitzen die selbe Gradsequenz.



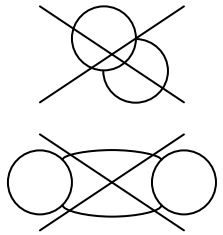
Gradsequenz:
(4, 4, 2, 1, 1, 1, 1, 1, 1)

(Dimethylether)

- Wenn G_1, G_2 isomorph sind, dann sind die Gradsequenzen identisch. (Die Umkehrung gilt i.A. nicht!)

Einige Begriffe

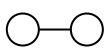
- Schlichter oder einfacher Graph:
  - keine Schlingen
  - keine Mehrfachkanten

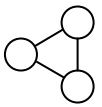


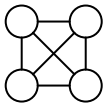
- Isolierter Knoten:
  - Knoten x, der mit keiner Kante verbunden ist, d.h. $d(x) = 0$
- Vollständiger Graph:
  - ungerichteter Graph, der alle möglichen Kanten besitzt.
  - K_n – vollständiger Graph mit n Knoten;

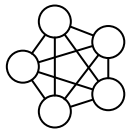
    dieser besitzt $\binom{n}{2} = \frac{n \cdot (n-1)}{2}$ viele Kanten.

K_1: 

K_2: 

K_3: 

K_4: 

K_5: 

2. Wege und Kreise in Graphen, Bäume

- Gradsequenztest
- Adjazenzmatrix
- Wege in Graphen
- Bäume
- Directed Acyclic Graph (DAG)
- Topologisches Sortieren
- Spannbaum (Algorithmus von Kruskal)
- Kürzeste Wege (Algorithmus von Dijkstra)
- Euler-Kreis





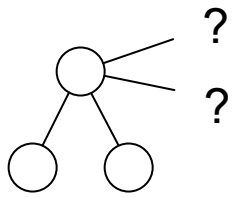




Gradsequenztest

Algorithmus zum Feststellen, ob eine Zahlenfolge $(u_1, u_2, \ldots, u_t)$ eine Gradsequenz ist oder nicht

Beispiel: (4, 1, 1)
ist keine Gradsequenz



1. Überprüfe, ob $\sum_{i=1}^{t} u_i$ eine gerade Zahl ist.
   (wenn nicht → keine Gradsequenz)

2. So lange möglich (bis eine leere Folge ( ) entsteht → dann ist es eine Gradsequenz) führe Reduktionsschritte durch.
   (wenn ein Reduktionsschritt nicht möglich ist → keine Gradsequenz)

Reduktionsschritt

a) Sortiere die momentane Reihenfolge absteigend:
$k = a_0 \geq a_1 \geq \ldots \geq a_m$

b) Reduktion ist möglich, falls $k \leq m$.
(sonst keine Gradsequenz)

c) Ersetze durch neue Folge:
$(a_1 - 1, a_2 - 1, \ldots, a_k - 1, a_{k+1}, \ldots a_m)$

d) Streiche alle Nullen in der Folge

a_0 lässt man weg. Von den nächsten k Elementen zieht man 1 ab.

Beispiel:

(5, 5, 4, 3, 2, 1)
Summe gerade
→ (4, 3, 2, 1, 0) → (4, 3, 2, 1), 4 > 3 → keine Gradsequenz
(b, c) (d) (b)
(3)

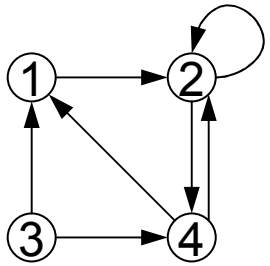
(5, 5, 4, 3, 2, 1, 1, 1)
Summe gerade
→ (4, 3, 2, 1, 0, 1, 1) → (4, 3, 2, 1, 1, 1) → (2, 1, 0, 0, 1) → (2, 1, 1)
(b, c) (d) (b, c) (d)
→ (0, 0) → ( ) → ist Gradsequenz
(b, c) (d) (2)

Adjazenzmatrix

(Computer-interne) Repräsentation eines Graphen mit n Knoten durch eine (n x n)-Matrix (die "Adjazenzmatrix")

Beispiel:

G =



Der Graph lässt sich auch als Matrix darstellen:

M_G =

	1	2	3	4
1	0	1	0	0
2	0	1	0	1
3	1	0	0	1
4	1	1	0	0

keine Kante von 1 nach 4

eine Kante von 2 nach 4

Die **Nummer der Zeilen** in der Matrix geben den Ursprung einer Verbindung an, die **Nummern der Spalten** das Ziel.

Matrix Multiplikation (1)

$(a_{ij}) \cdot (b_{ij}) = (c_{ij})$

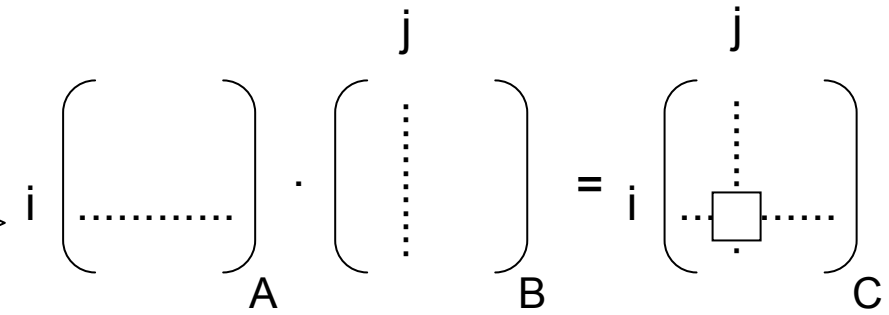
wobei

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

$i = 1, \ldots, n$

$j = 1, \ldots, n$

Alle Elemente der Matrix A in der Zeile i werden nacheinander mit allen Elementen der Matrix B in der Spalte j multipliziert und aufsummiert. Das Ergebnis steht in der Matrix C an der Stelle i, j.



oder auch die Bool'sche Matrix-Multiplikation
(keine natürlichen Zahlen im Ergebnis, sondern weiterhin 0 und 1)

$$c_{ij} = \bigvee_{k=1}^{n} (a_{ik} \wedge b_{kj})$$

logisches Oder

logisches Und

Matrix Multiplikation (2)

Beispiel: (normale Matrix Multiplikation)

$$M_G \cdot M_G = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \cdot \begin{pmatrix} 4 & 7 \\ 5 & 8 \\ 6 & 9 \end{pmatrix} = \begin{pmatrix} 1\cdot4+2\cdot5+3\cdot6 & 1\cdot7+2\cdot8+3\cdot9 \\ 4\cdot4+5\cdot5+6\cdot6 & 4\cdot7+5\cdot8+6\cdot9 \end{pmatrix}$$

$$= \begin{pmatrix} 4 + 10 + 18 & 7 + 16 + 27 \\ 16 + 25 + 36 & 28 + 40 + 54 \end{pmatrix}$$

$$= \begin{pmatrix} 32 & 50 \\ 77 & 122 \end{pmatrix}$$

Matrix Multiplikation (3)

Beispiel: (mit logischen Operatoren – und ($\wedge$) / oder ($\vee$))

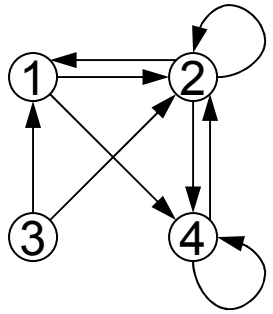
$$M_G \cdot M_G = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

$$c_{11} = \underbrace{(0 \wedge 0)}_{0} \vee \underbrace{(1 \wedge 0)}_{0} \vee \underbrace{(0 \wedge 1)}_{0} \vee \underbrace{(0 \wedge 1)}_{0} = 0 \vee 0 \vee 0 \vee 0 = 0$$

$$c_{12} = \underbrace{(0 \wedge 1)}_{0} \vee \underbrace{(1 \wedge 1)}_{1} \vee \underbrace{(0 \wedge 0)}_{0} \vee \underbrace{(0 \wedge 1)}_{0} = 0 \vee 1 \vee 0 \vee 0 = 1$$

$$c_{13} = \underbrace{(0 \wedge 0)}_{0} \vee \underbrace{(1 \wedge 0)}_{0} \vee \underbrace{(0 \wedge 0)}_{0} \vee \underbrace{(0 \wedge 0)}_{0} = 0 \vee 0 \vee 0 \vee 0 = 0$$

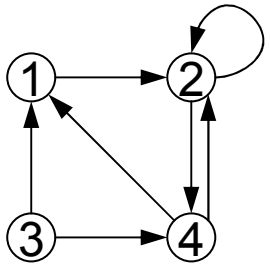
als Graph:



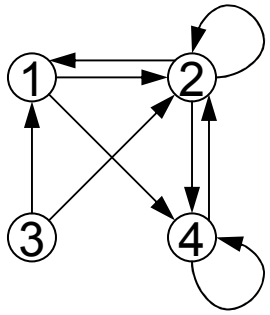
Interpretation

ursprünglicher Graph

G =

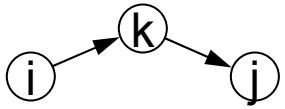


neuer Graph

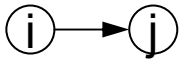


Was hat sich geändert?

- Der Graph zu $M_G \cdot M_G$ erhält eine (neue) Kante $i \rightarrow j$, falls in G die 2 Kanten $i \rightarrow k$, $k \rightarrow j$ existieren.
- bisherige Kanten



- neue Kante



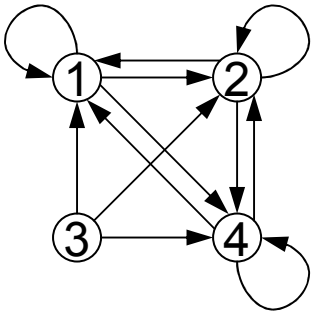
Transitive Hülle

Allgemein: $M_G^t = \underbrace{M_G \cdot M_G \cdot M_G \cdot ... \cdot M_G}_{\text{t - mal}}$

Der zugehörige Graph hat eine Kante i → j, falls es in G einen Weg der Länge t von i nach j gibt.

$M_G^* = M_G \cup M_G^2 \cup M_G^3 \cup ... \cup M_G^n$ **transitive Hülle** von M_G

G^* =



Wege in Graphen

Ein Weg (= Pfad) (von i nach j) ist eine Folge von Knoten
$i = i_0, i_1, ..., i_t = j$, so dass $(i_k, i_{k+1}) \in E \quad (k = 0, ..., t-1)$.

für ungerichtete Graphen stünden hier: { }

Ein Weg heißt **einfach**, falls sich in $i_1, ..., i_t$ kein Knoten wiederholt.

z.B. (1, 2, 3, 1)
i_0 i_t

Ein einfacher Weg mit $i_0 = i_t$ heißt ein **Kreis** (oder **Zyklus**).

Ein Graph, in dem keine Kreise vorkommen, heißt **azyklisch** (oder **kreisfrei**).

Wege in Graphen (1)

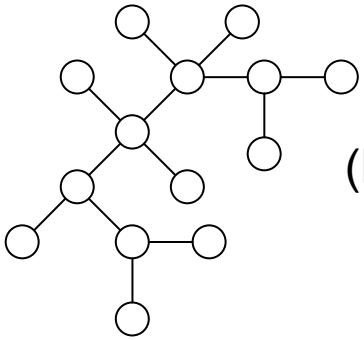
Ein ungerichteter Graph heißt **zusammenhängend**, falls für alle i, j $\in$ V gilt:
Es gibt einen Weg von i nach j.

Ein gerichteter Graph heißt **stark zusammenhängend**, falls für alle i, j $\in$ V gilt:
Es gibt einen gerichteten Weg von i nach j.

Ein gerichteter Graph heißt **schwach zusammenhängend**, falls für alle i, j $\in$ V gilt:
Es gibt einen Weg von i nach j – unter Ignorierung der Pfeilrichtung.

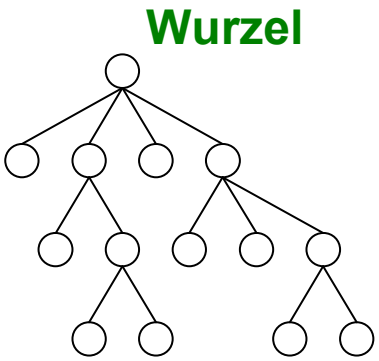
Bäume

Ein **Baum** ist ein ungerichteter, zusammenhängender, azyklischer Graph.



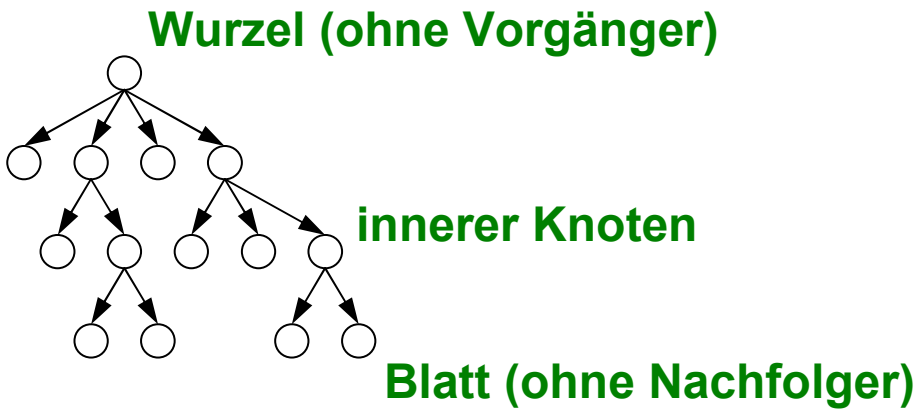
(keine Kreise!)

andere Darstellung (hierarchisch):



Bäume (1)

Ein **gerichteter Baum** ist ein gerichteter Graph mit einem ausgezeichneten Knoten, der so genannten Wurzel, für den gilt,
dass im Falle von **Out-Trees** jeder Knoten durch genau einen gerichteten Pfad von diesem aus erreichbar ist oder dass im Falle von **In-Trees** dieser von jedem Knoten aus durch genau einen gerichteten Pfad erreichbar ist.
(unterschiedliche Definitionen)

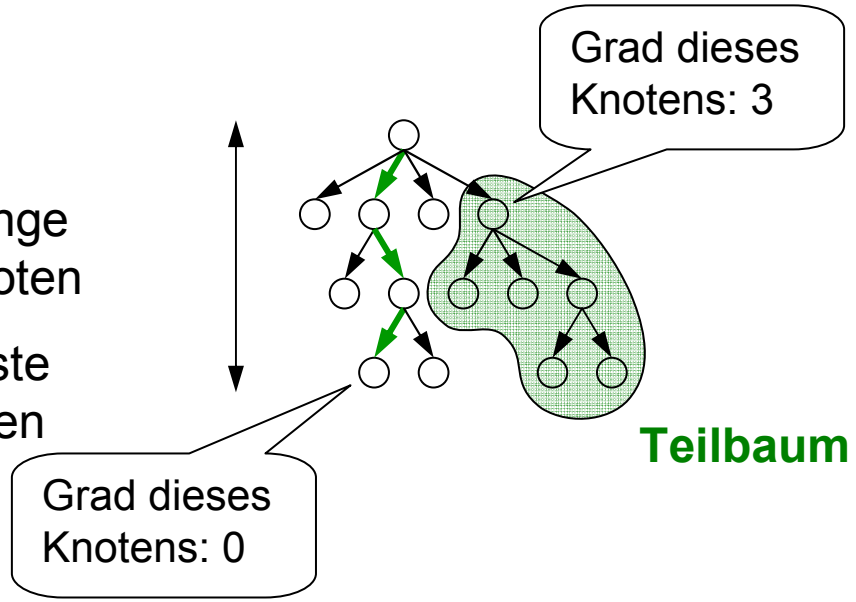


Bäume (2)

gerichtete Bäume stellen einen Spezialfall von gerichteten Graphen dar.

Einige Begriffe:

- Die **Höhe eines Knotens** ist die Länge des Pfades von der Wurzel zum Knoten
- Die **Höhe des Baumes** ist der längste Pfad von der Wurzel zu einem Knoten (hier Höhe = 3)
- Der **Grad eines Knotens** ist die Anzahl aller seiner direkten Nachfolger
- Ein **Teilbaum** ist ein Knoten (Wurzel des Teilbaums) und alle seine Nachfolger

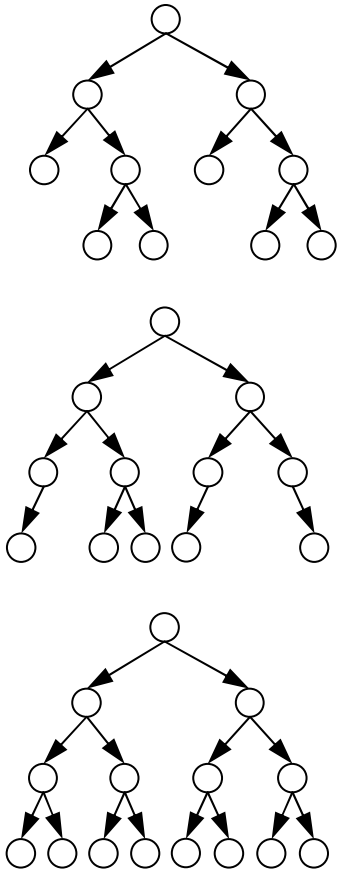


Bäume (3)

gerichtete Bäume stellen einen Spezialfall von gerichteten Graphen dar.

Einige Begriffe:

- Man spricht von einem **Binärbaum**, wenn jeder Knoten maximal zwei Nachfolger hat
- Man spricht von einem **der Höhe nach ausgeglichenen Baum**, wenn sich die Höhe aller seiner direkten Teilbäume maximal um 1 unterscheidet
- Man spricht von einem **dem Gewicht nach ausgeglichenen Baum**, wenn die Anzahl der Knoten aller seiner direkten Teilbäume sich maximal um 1 voneinander unterscheiden
- Man spricht von einem **vollständig ausgefüllten** Baum, wenn alle inneren Knoten vollständig aufgefüllt sind



Bäume (4)

Definition:

Ein Baum ist entweder leer
oder er besteht aus einem Knoten (Wurzel)
verknüpft über Kanten mit einer Anzahl disjunkter Bäume

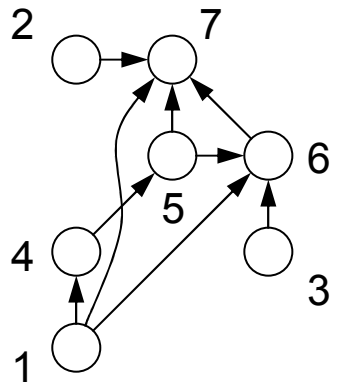
(disjunkt bedeutet hier: keiner der (Teil-)Bäume hat gemeinsame Knoten mit anderen (Teil-)Bäumen)

Bemerkung:

Man nennt dies eine *rekursive Definition*.

Directed Acyclic Graph – DAG

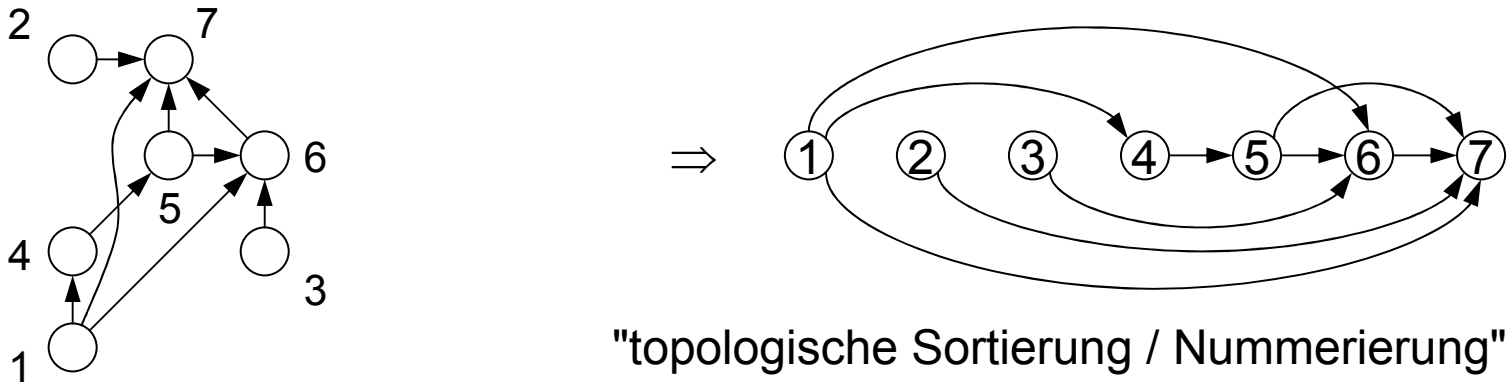
Gerichteter, zyklenfreier Graph



Knoten: **Ereignisse**, z.B. Projektstart, Projektende, Abschluss von Zwischentätigkeiten bei Fertigungsprozessen

Kanten: **Vorgänge**, z.B. Holz zuschneiden, transportieren, Holzteile lackieren, Holzteile montieren,…

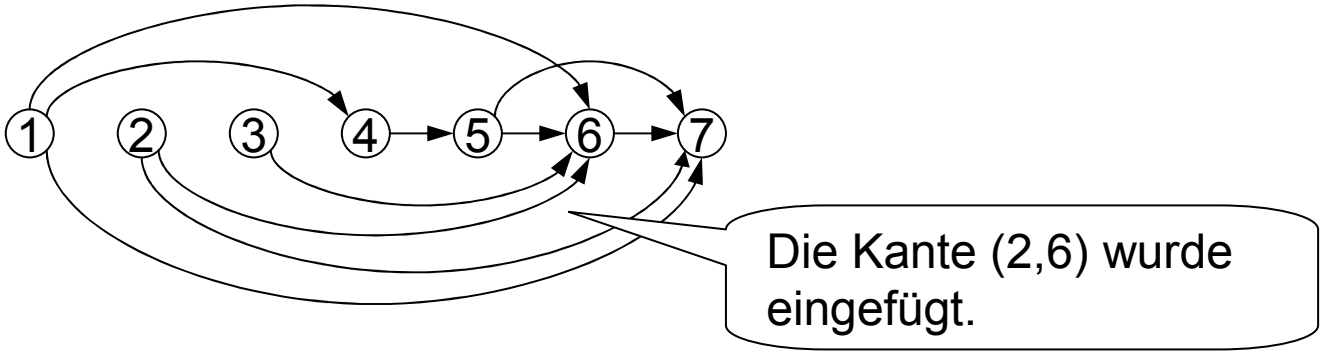
Directed Acyclic Graph – DAG (1)



"topologische Sortierung / Nummerierung"

Die Knoten jedes DAGs können so nummeriert werden, dass für alle Kanten $(n, m) \in E$ gilt: $n < m$.

Directed Acyclic Graph – DAG (2)



Die topologische Sortierung bleibt gültig!

Topologisches Sortieren

Problem:

Gegeben: endlicher gerichteter Graph G = (V, E)
(das entspricht einer partiellen Ordnung)

Gesucht: eine Aufzählung/Ordnung von Kanten, die die durch den gerichteten Graphen vorgegebene partielle Ordnung berücksichtigt.

Beispiel Studienplanung:

Gegeben: Menge von Vorlesungen $\{v_1, ..., v_n\}$
$(v_i, v_j) \in E$ bedeutet: Vorlesung v_i ist Voraussetzung für Vorlesung v_j.

Gesucht: zeitliche Anordnung der Vorlesungen, so dass alle zu einer Vorlesung notwendigen Kenntnisse in früheren Vorlesungen erworben werden können.

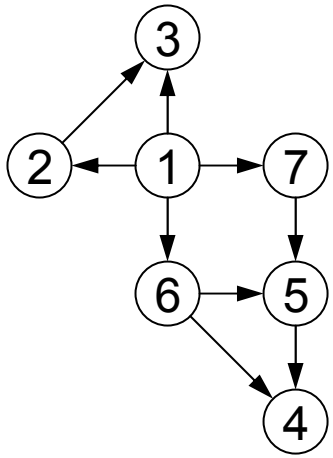
Topologisches Sortieren

Lösungsidee:

Solange der Graph G nicht leer ist:

1. Wähle "Quelle" (= Knoten ohne Vorgänger) v $\in$ V und füge ihn an die bisherige Knotenfolge (= Ergebnis) an
2. Entferne v und alle davon ausgehenden Kanten aus G
3. Fahre fort bei (1) bis alle Knoten aus G entfernt sind
4. Falls keine "Quellen" (mehr) vorhanden und
   - 4.1 Knotenmenge leer $\Rightarrow$ erfolgreich (fertig)
   - 4.2 Knotenmenge nicht leer $\Rightarrow$ Zyklus $\Rightarrow$ Misserfolg

Beispiel-Graph



Schritt	vorhandene Quellen	gewählte Quelle
1	{1}	1
2	{2, 6, 7}	2
3	{6, 7, 3}	6
4	{7, 3}	3
5	{7}	7
6	{5}	5
7	{4}	4

Eine mögliche Sortierung:

1, 2, 6, 3, 7, 5, 4

andere mögliche topologische Sortierungen durch andere Quellenwahl, wenn mehr als 1 Quelle vorhanden

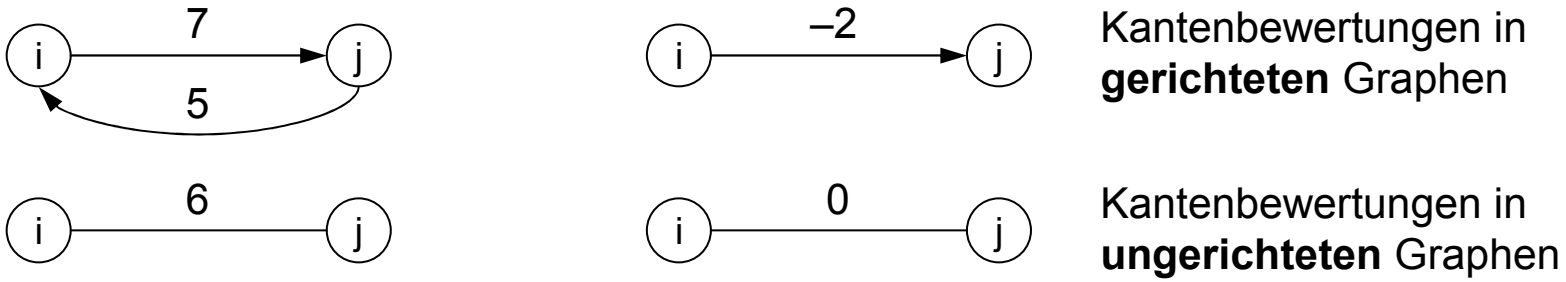
Kantenbewertungen

Bedeutung:

Kleinstmöglichstes Straßennetz oder Leitungsnetz oder …, welches alle Knoten untereinander erreichbar macht.

Aber:

Entfernungen zwischen Knoten noch nicht berücksichtigt. Dies geschieht mittels Kantenbewertungen bei Graphen



Notation: $w(i, j) = 7$, $w(j, i) = 5$ etc.

Spannbaum (spannender Baum)

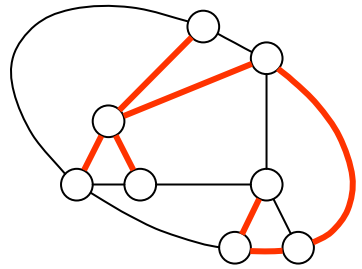
Gegeben sei ein zusammenhängender ungerichteter Graph G.

Ein **aufgespannter Baum** (Spannbaum, spannender Baum, Minimalgerüst) in G ist ein Teilgraph von G, welcher ein Baum ist, und alle Knoten von G enthält.

Bemerkung:

Sei n die Anzahl der Knoten von G.
Dann ist n – 1 die Anzahl der Kanten des Spannbaums.

G =



Spannbaum in G
(keine Kreise)

Algorithmus

- Definition:

  Endliche Folge von Operationen und Entscheidungen,
  die ein Problem in endlich vielen Schritten löst.

Algorithmus von Kruskal

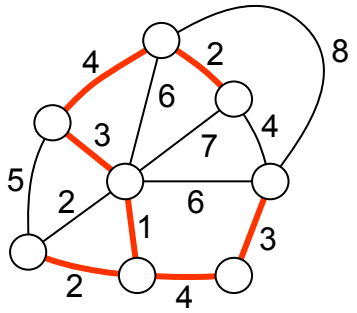
Gegeben:

Graph G = (V, E) mit Gewichten (Kosten) an den Kanten, $w : E \rightarrow R_+$

Aufgabe:

Finde einen Spannbaum in G mit minimalem Gewicht.

Beispiel:



$w : E \rightarrow R_+$

Gewicht des Spannbaums ist minimal:

1 + 2 + 2 + 3 + 3 + 4 + 4 = 19

- Wähle von den verbleibenden Kanten eine mit minimalem Gewicht
- Sofern die Kante keinen Kreis mit den bisher ausgewählten Kanten schließt:
  nehme die Kante in den Spannbaum auf (sonst auslassen).
- Wiederhole solange, bis der Spannbaum komplett ist
  (d.h. bis die Anzahl der Kanten = n – 1).

Algorithmus von Kruskal (1)

Eingabe: $V = \{ v_1, v_2, \ldots, v_n \}$ Menge von Knoten
$E = \{ e_1, e_2, \ldots, e_m \}$ Menge von Kanten
$w(e_i)$ Gewicht der Kante, $\forall e_i$ mit i = 1, ..., m

1. L := $\emptyset$ *// Lösungsmenge am Anfang noch leer*
2. M := min(E) *// Menge der Kanten mit minimalem Gewicht*
3. wähle ein e $\in$ M
4. Falls e keinen Kreis bildet mit den bisher ausgewählten Kanten (Menge L)
   dann E := E \ e *// streiche e aus E*
   L := L $\cup$ { e } *// füge e in L ein*
5. Wiederhole ab Schritt 2 solange Anzahl(L) < n – 1
   // d.h. solange Spannbaum noch nicht komplett
6. Ergebnis ist: L *// Lösungsmenge*

Algorithmus von Kruskal (1)

Eingabe: $V = \{ v_1, v_2, \ldots, v_n \}$ Menge von Knoten
$E = \{ e_1, e_2, \ldots, e_m \}$ Menge von Kanten
$w(e_i)$ Gewicht der Kante, $\forall e_i$ mit i = 1, ..., m

1. L := ∅ *// Lösungsmenge am Anfang noch leer*
2. M := **min(E)** *// Menge der Kanten mit minimalem Gewicht*
3. wähle ein neues e ∈ M
4. Falls e keinen Kreis bildet mit den bisher ausgewählten Kanten (Menge L)
   dann E := E \ e *// streiche e aus E*
   L := L ∪ { e } *// füge e in L ein*
5. Wiederhole ab Schritt 2 solange **Anzahl(L)** < n – 1
   // d.h. solange Spannbaum noch nicht komplett
6. Ergebnis ist: L *// Lösungsmenge*

Schritt 2 (etwas konkreter):

```
min(E):
    wiederhole ∀ ε ∈ E:
        wenn ε < ε_bisheriges Minimum
        dann   ε_bisheriges Minimum := ε
               Σ := {ε }
        sonst  wenn ε = ε_bisheriges Minimum
               dann Σ := Σ ∪ ε
        // Ende von wenn
        Ergebnis ist: Σ
```

Schritt 4 (etwas konkreter):

```
Anzahl(L):
    k := 0
    wiederhole solange L ≠ ∅
        L := L \ { l_1 }    // streiche erstes Element aus L
        k := k + 1          // erhöhe k um 1
    // Ende von wiederhole
    Ergebnis ist: k
```

Einschub: Pseudocode

- Ein Programm besteht aus folgenden Elementen
  (= verschiedene Arten von Anweisungen)

1. Sequenzen von Anweisungen

   <*Anweisung 1*>
   <*Anweisung 2*>
   ⋮
   <*Anweisung n*>

2. Variablenzuweisungen

   x := <*Wert*>

3. Verzweigungen (durch Bedingung)

   wenn <*Bedingung*> gilt
   dann <*Sequenz von Anweisungen*>
   sonst <*Sequenz von anderen Anweisungen*>

4. Wiederholungen

   z.B. wiederhole solange <*Bedingung*> gilt
   <*Sequenz von Anweisungen*>

5. Unterprogrammen / Funktionen / Methoden

   Platzhalter für Teilprogramme,
   die z.B. der Übersicht halber an anderer Stelle definiert werden

z.B. x := 5
a := x – 6
i := i + 1
$e_i := e_{i-1} + e_{i-2}$
$L := L \cup \{ e \}$
c := ‘B‘
s := „hallo“ + „Welt“

z.B. $i \leq 10$
$k = i$
$e \notin L$
$L \neq \emptyset$
$(a > 1) \wedge (a \leq 10)$

Einschub: Pseudocode (1)

- Algorithmus ist endliche Folge von Anweisungen, die nach endlich vielen Schritten terminiert

- Pseudocode beschreibt Abläufe (Verfahren / Algorithmen)
- kann verschieden abstrakt gehalten sein
- einzelne Schritte können bei Bedarf weiter verfeinert werden

- Computer „verstehen“ nur eine Sprache, die ein Problem in ***sehr*** kleine Schritte zerlegt.
- Pseudocode hilft ein Problem in immer kleinere Teilprobleme zu zerlegen, bis diese Abstraktionsstufe erreicht ist.
- Pseudocode ist noch unabhängig von der Schreibweise konkreter Programmiersprachen.

Einschub: Pseudocode (2)

➢ zu Unterprogrammen:

- Platzhalter für Teilprogramme
- lässt sich "aufklappen" wieder zu eigenem Programm-Code
- können manchmal ein Ergebnis zurückliefern

Ergebnis ist: <*Wert*>

Beispiel: n := Anzahl(L)

```
Anzahl(L):
    n := 0
    wiederhole solange L ≠ ∅
        L := L \ { l1 }   // streiche erstes Element aus L
        n := n + 1        // erhöhe n um 1
    // Ende von wiederhole
    Ergebnis ist: n
```

Einschub: Pseudocode (3)

➢ zu Wiederholungen:

- Sequenz von Anweisungen, die wiederholt wird, solange eine Bedingung gilt (bzw. nicht gilt)
- unterschiedliche Arten von Wiederholungen
  - feste Anzahl
    - wiederhole genau <*Anzahl*> mal
      <*Sequenz von Anweisungen*>
    - wiederhole $\forall e \in E$:
      <*Sequenz von Anweisungen*>
    - zähle i := 1 bis 10 und wiederhole dabei
      <*Sequenz von Anweisungen*>
  - Kopf gesteuert
    - wiederhole solange <*Bedingung*> gilt
      <*Sequenz von Anweisungen*>
  - Fuß gesteuert
    - wiederhole
      <*Sequenz von Anweisungen*>
      solange <*Bedingung*> gilt

Wird eine **bestimmte** (feste) **Anzahl** mal wiederholt

Wird **vielleicht** auch **nie** durchlaufen (Test am Anfang)

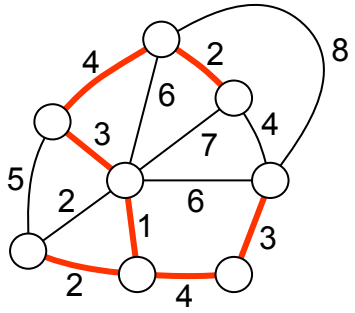
Wird **mindestens einmal** durchlaufen (Test erst am Ende)

Einschub: Pseudocode (4)

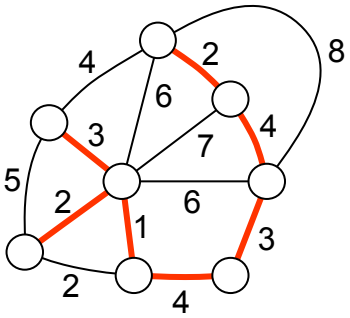
- Eingabe / Ausgabe:
  - manchmal sind Eingaben erforderlich:
    - Eingabe: <*Variable(n)*>
  - manchmal soll auch etwas ausgegeben werden:
    - Ausgabe: „Hallo“
    - Ausgabe: <*Variable(n)*>

Algorithmus von Kruskal (1)

- Auswahl des Kruskal Algorithmus ist nicht unbedingt eindeutig:



oder



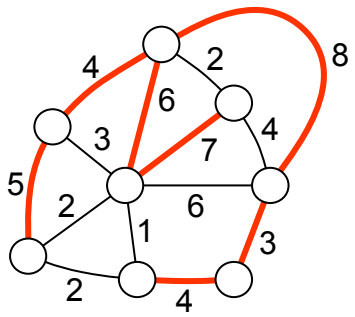
- Alternativer spannender Baum mit demselben Minimalgewicht

$$1 + 2 + 2 + 3 + 3 + 4 + 4 \quad = \quad 19$$

Algorithmus von Kruskal (2)

Bemerkung:

- Analog kann auch ein Spannbaum mit maximalem Gewicht konstruiert werden:



- Wähle (von den verbleibenden Kanten) eine Kante mit maximalem Gewicht
- Sofern die Kante keinen Kreis mit den bisher ausgewählten Kanten schließt:
  nehme die Kante in den Spannbaum auf (sonst auslassen).
- Wiederhole solange, bis alle Knoten erfasst sind

maximaler Spannbaum:

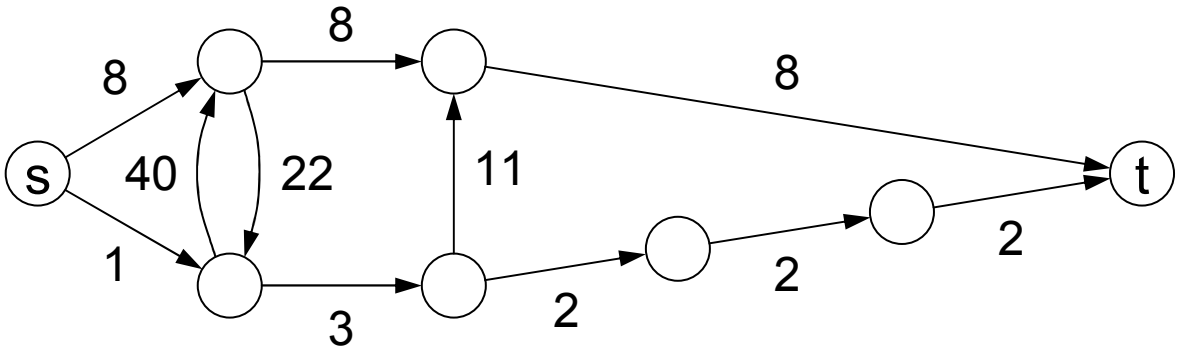
8 + 7 + 6 + 5 + 4 + 4 + 3 = 37

Algorithmustyp des Kruskal-Algorithmus:

Greedy-Algorithmus

Kürzeste Wege

- Kantenbewertungen können auch genutzt werden, um gewichtete oder kostenminimale Weg ("minimale Wege", "kürzeste Wege") zu berechnen.
- Das Gewicht eines Weges ist die Summe der beteiligten Kantengewichte.



- "Oberer" Weg von s nach t beinhaltet 3 Kanten mit Kosten (= Gewicht = Länge) 8 + 8 + 8 = 24.
- "Unterer" Weg von s nach t beinhaltet 5 Kanten mit Kosten (= Gewicht = Länge) 1 + 3 + 2 + 2 + 2 = 10.

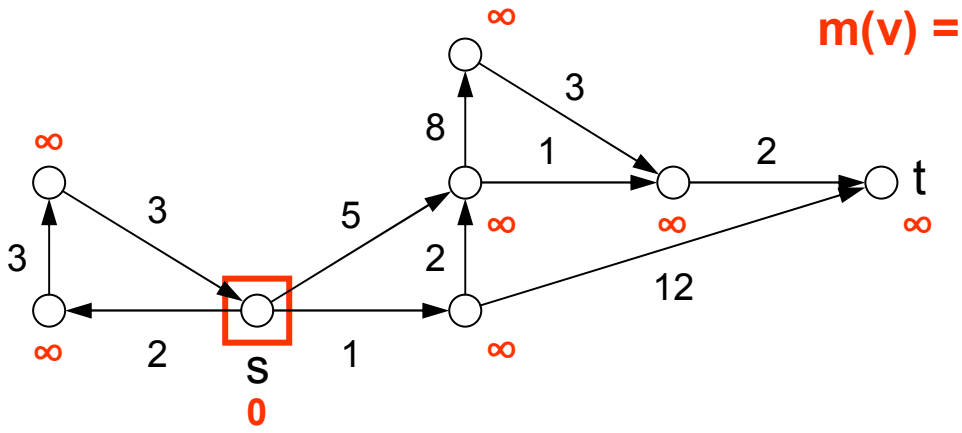
- Anwendung: Routenplanung.

Kürzeste Wege (1)

- Intuitives Vorgehen zur Berechnung des kürzesten Wegs von Startknoten s zu Zielknoten t.

- Es werden Knotenmarkierungen vergeben, die sich im Ablauf des Algorithmus ändern.
- Es wird eine Liste K von "unverbrauchten" Knoten geführt, die Schrittweise verkleinert wird.

- Beispiel: Graph mit Anfangssituation: **K = V, m(s) = 0**
  m(v) = ∞, ∀v∈V \ {s}



- Von einem besten Knoten ∈ K werden sukzessive Fortsetzungen konstruiert.

Algorithmus von Dijkstra (Routenplaner) (1)

- Dijkstra Algorithmus für kürzesten Weg von s nach t in gerichteten Graphen

1. Startsetzungen ("Initialisierungen")
   $K := V$, $m(s) := 0$,
   $m(v) := \infty$, $\forall\ v \in V \setminus \{s\}$.

2. Wiederhole solange $K \neq \emptyset$:
   Wähle $v_0 \in \min(K)$. // *Knoten mit minimalem m-Wert*
   $K := K \setminus \{ v_0 \}$ // *streiche v_0 aus K*
   $K' := \{ k \in K \mid (v_0, k) \in E \}$ // *Knoten, die von v_0 aus direkt erreichbar sind*

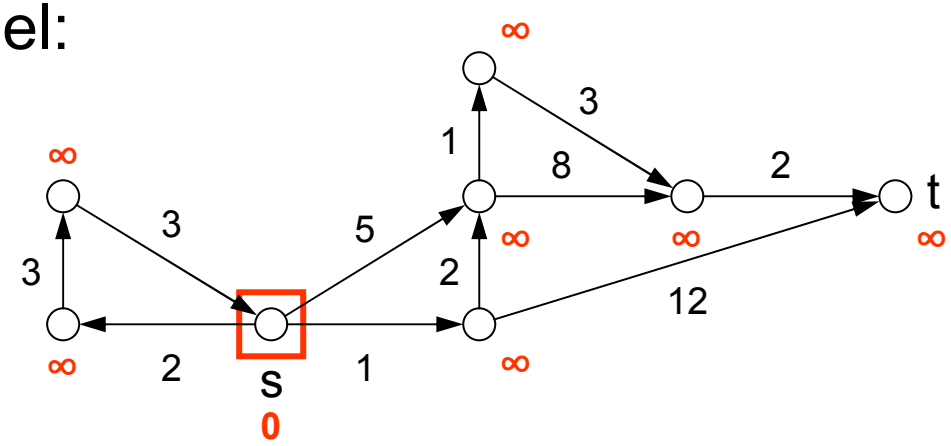
   Wiederhole $\forall\ k \in K'$:
   Wenn $m(v_0) + w(v_0, k) < m(k)$
   dann $m(k) := m(v_0) + w(v_0, k)$
   $pred(k) := v_0$

Beschreibung des kürzesten Weges über eine Liste der "Vorgänger".

3. Ausgabe m(t) und **t, pred(t), pred(pred(t)), ..., s**

Kürzeste Wege (2)

➢ 2. Beispiel:



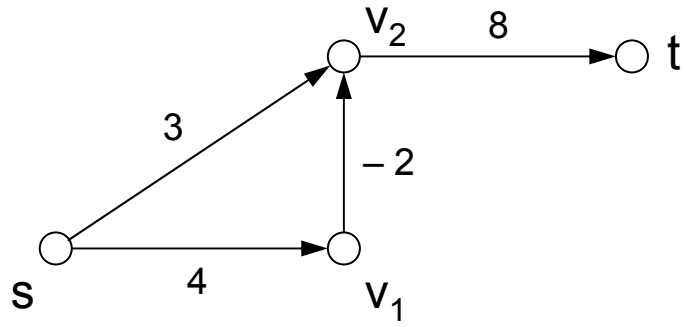
Algorithmus von Dijkstra (2)

- Der Dijkstra Algorithmus findet immer einen kürzesten Weg von einem festgelegten Start- zu einem festgelegten Zielknoten, wenn es einen solchen Weg gibt und alle Kantengewichte nicht-negativ sind.

Algorithmus von Dijkstra (3)

Beispiel:

- Der Dijkstra Algorithmus kann versagen, wenn Kanten negative Gewichte haben.



- Kürzester Weg von s nach t mit Länge 10 wird nicht gefunden:
  - von Knoten v_1 findet Dijkstra Algorithmus **keine** Fortsetzung, wenn die Marke $m(v_1) = 4$ gesetzt wird, da in dieser Situation v_2 schon aus der Liste K gestrichen wurde.

Algorithmus von Dijkstra (4)

Der Dijkstra Algorithmus findet kürzeste Wege von s zu **allen** anderen Knoten, wenn man ihn nur "lange genug" laufen lässt.

"One to many Algorithmus".

Um nach der Berechnung des kürzesten Weges von s nach t abzubrechen, wird die Bedingung

"solange $K \neq \emptyset$"

ersetzt durch die Bedingung

"solange Zielknoten $t \in K$".

Algorithmus von Dijkstra (5)

- Animation siehe http://www-b2.is.tokushima-u.ac.jp/~ikeda/suuri/dijkstra/Dijkstra.shtml

Algorithmus von Dijkstra (Alternative Einführung) (6)

Es existieren unterschiedliche Varianten des Dijkstra-Algorithmus in der Literatur. Hier ein Beispiel unter Verwendung von zwei Listen:

Gegeben:

- ungerichteter Graph $G = (V, E)$
  Startknoten $s \in V$,
  Kantengewichte $w : E \rightarrow R_+$
  ($\cong$ Entfernungen)

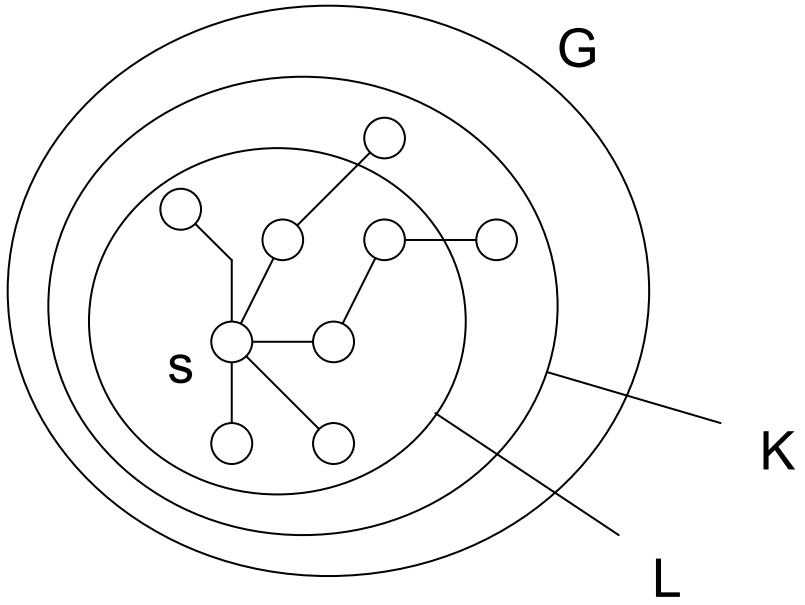
Gesucht:

- Kürzesten Entfernungen von s aus zu allen Knoten

Algorithmus von Dijkstra (Alternative Einführung) (7)

- Verwalte zwei Listen:
  - L enthält alle Knoten V, deren minimaler Abstand d vom Startknoten s bereits exakt feststeht.
    (genauer: Paare von Knoten und minimalem Abstand (x, d))
  - K "Kandidatenliste" (Nachbarn von Knoten, die nach L verschoben werden)

- Skizze:



Algorithmus von Dijkstra (Alternative Einführung) (8)

- Initialisierung:
  - L = { (s, 0) }, K = Ø

Der Minimale Abstand von x zu sich selbst beträgt 0.

- Sei (x, d) der letzte Eintrag in die Liste L.

  Trage alle Nachbarn von x, die noch nicht in der Liste L enthalten sind, in die Liste K ein.
- Sei w = w ({x, y}).

  w = Abstand von x nach y

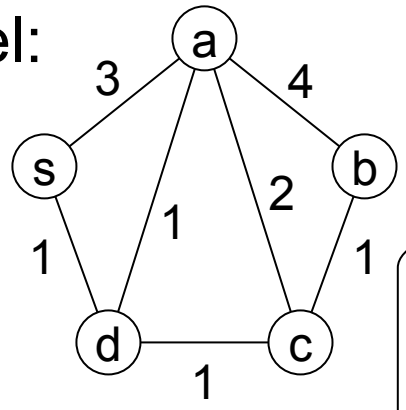
  Trage (y, d + w) in die Liste K ein.
  (Falls bereits ein Eintrag (y, t) in der Liste K existiert, dann trage ein: (y, min(t, d + w)))

  d + w = Abstand vom Anfang ausgehend
- Wähle in der Liste K den Eintrag (x, d), so dass d minimal ist.
- Verschiebe (x, d) von der Liste K in die Liste L.

- Wiederhole den Vorgang so lange, bis die Liste L alle Knoten $\in$ V enthält.

Algorithmus von Dijkstra (Alternative Einführung) (9)

➢ Beispiel:



Direkte Nachfolger von s ermitteln.

Nach a gibt es nun einen kürzeren Weg, daher Weglänge anpassen.

Direkte Nachfolger von d ermitteln. Weglänge zählt immer vom Startpunkt aus.

• Start:	L = { (s,0) }	K = Ø
• Schritt:	L = { (s,0) }	K = { (a,3), ~~(d,1)~~ }
•	L = { (s,0), (d,1) }	K = { ~~(a,2)~~, (c,2) }
•	L = { (s,0), (d,1), (a,2) }	K = { ~~(c,2)~~, (b,6) }
•	L = { (s,0), (d,1), (a,2), (c,2) }	K = { ~~(b,3)~~ }
•	L = { (s,0), (d,1), (a,2), (c,2), (b,3) }	K = Ø

Element mit dem kleinsten Weg auswählen,
in K streichen und in L einfügen.