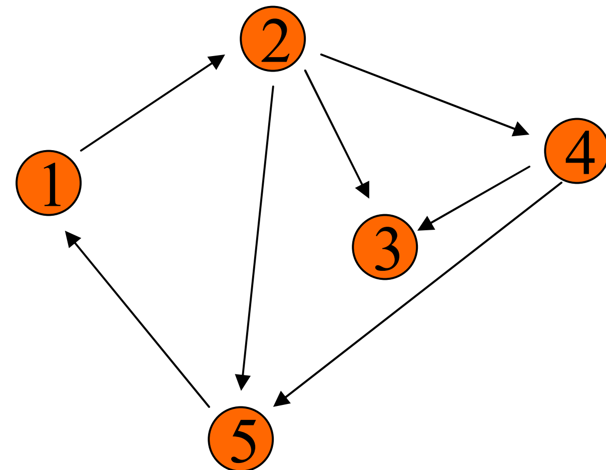
7. Graphenalgorithmen

Grundlagen:

1. Paar (V, E) heißt **gerichteter Graph *G***, wobei
   V endl. Menge von Knoten und $E \subseteq V \times V$
   eine endl. Menge von **Kanten** ist.



Element $e = (v,w) \in E$

heißt **Kante** von v nach w

(„ $v \longrightarrow w$ “)

v ist **Startknoten** von e

w ist **Zielknoten** von e

w ist **Nachbarknoten** von v (w ist **adjazent** zu v)

2. **Pfad** in G ist Folge $(v_0, ..., v_k)$ von Knoten mit $k \geq 0$ und $(v_i, v_{i+1}) \in E$ für $0 \leq i \leq k-1$.

Länge des Pfades ist k.

Falls $v_0 = v_k$ und $k \geq 1$, so heißt der Pfad **Kreis** bzw. **Zyklus**.

Falls $v_i \neq v_j$ für alle $i \neq j$, so heißt der Pfad **einfach**.

Falls es Pfad von v nach w in G gibt, so schreibt man

$$v \xrightarrow[G]{*} w .$$

Gerichteter Graph heißt **zyklisch**, falls er Kreis enthält; sonst **azyklisch**.

7.1 Darstellung von Graphen

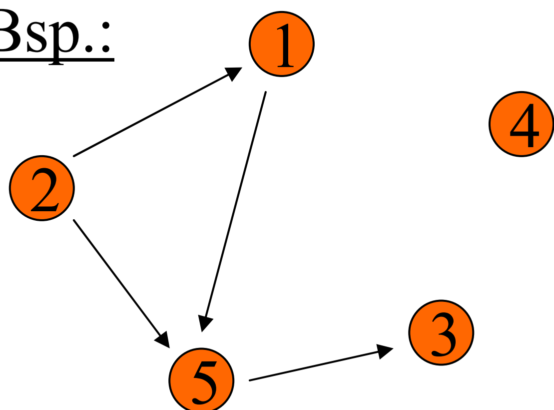
Annahme: $V = \{1, 2, ..., n\}$, $G = (V, E)$, $n = |V|$.

Zwei Darstellungsarten sind üblich:

1. Darstellung: **Adjazenzmatrix** $A = (a_{ij}),\ 1 \le i,j \le n$

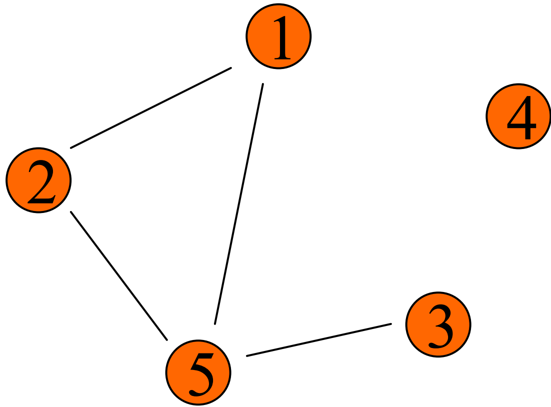
$$a_{ij} := \begin{cases} 1, & \text{falls } (i,j) \in E \\ 0, & \text{sonst.} \end{cases}$$

Bsp.:



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

Bei ungerichteten Graphen ist die Adjazenzmatrix symmetrisch:



hier: $a_{ij} := \begin{cases} 1, & \text{falls } \{i,j\} \in E \\ 0, & \text{sonst.} \end{cases}$

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

Platzbedarf: $O(n^2)$

ist günstig, falls $m := |E| \approx n^2$.

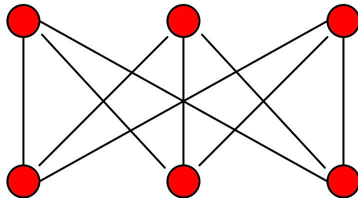
Aber: Oft sind Graphen dünn, d.h. $m \approx O(n)$.

Bsp.:

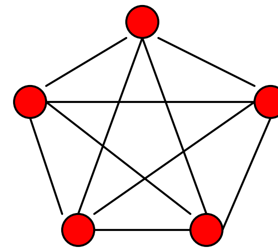
- Zahl der Kanten in Bäumen mit n Knoten ist $n - 1$.
- Planare Graphen: Graph heißt **planar**, falls er in Ebene gezeichnet werden kann, ohne dass sich Kanten überkreuzen.

Zwei nicht-planare, ungerichtete Graphen:

$K_{3,3}$



K_5



(„vollständiger" bipartiter Graph mit 3 und 3 Knoten)

<u>Mitteilung:</u> Ein planarer Graph mit $n>2$ Knoten enthält maximal $3n-6$ Kanten.

2. Darstellung: **Adjanzenzlisten**

Speichere für jeden Knoten *v* seine „Nachbarn“:

$G = (V, E)$ gerichtet:

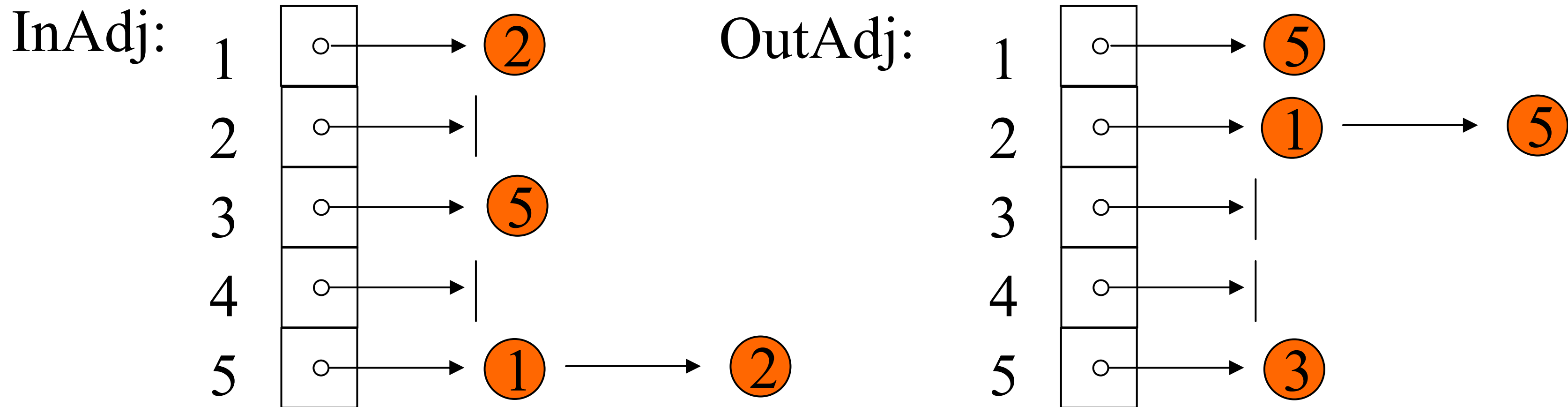
$$\text{InAdj}(v) = \{\, w \in V \mid (w, v) \in E \,\}$$

$$\text{OutAdj}(v) = \{\, w \in V \mid (v, w) \in E \,\}$$

G ungerichtet:

$$\text{Adj}(v) = \{\, w \in V \mid \{v, w\} \in E \,\}$$

Im vorigen Beispiel (gerichtet):



Platzbedarf: $O(n + m)$

Nachteil: Zugriffszeit auf Kante abhängig von Listenlänge, d.h. **Knotengrad**:

Eingangsgrad indeg(v) eines Knoten $v = \left| \left\{ (w,v) \middle| (w,v) \in E \right\} \right|$,

Ausgangsgrad outdeg(v) eines Knoten $v = \left| \left\{ (v,w) \middle| (v,w) \in E \right\} \right|$

(analog für ungerichtete Graphen).

Graph $G = (V, E)$ heißt **Baum**, falls

a) V enthält genau einen Knoten v_0 mit $\text{indeg}(v_0) = 0$.

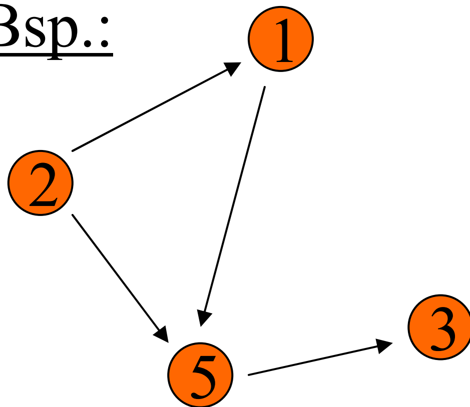
b) $\forall\, v \in V \setminus \{v_0\} : \text{indeg}(v) = 1$

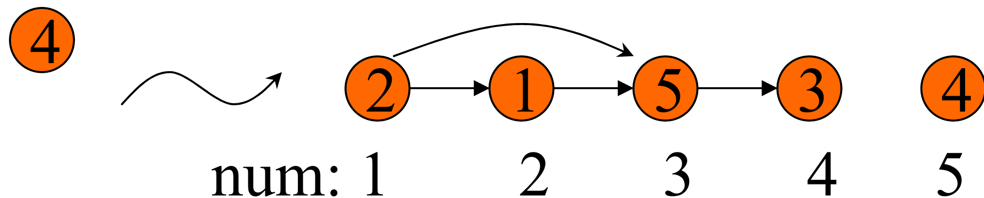
c) G ist azyklisch.

7.2 Topologisches Sortieren

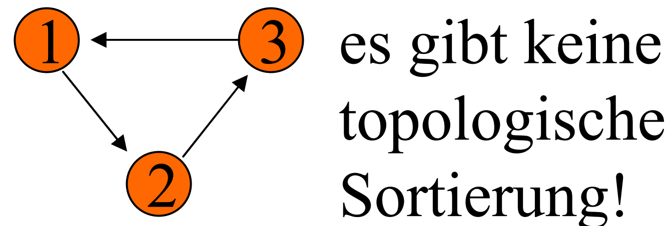
Sei $G = (V, E)$ gerichteter Graph.
Abbildung num: $V \longrightarrow \{1, ..., n\}$ mit $n = |V|$ heißt **topologische Sortierung** von G, falls num(v) < num(w) für alle $(v, w) \in E$.

Bsp.:





num: 1 2 3 4 5



es gibt keine topologische Sortierung!

Lemma:

Gerichteter Graph $G = (V, E)$ besitzt genau dann eine topologische Sortierung, wenn G azyklisch ist.

Beweis:

"$\Rightarrow$":

Sei G zyklisch. Dann ist etwa $(v_0, ..., v_k)$, $k \geq 1$, ein Kreis. Für eine topologische Sortierung muss aber gelten:

$$\text{num}(v_0) < \text{num}(v_1) < ... < \text{num}(v_k) = \text{num}(v_0).$$

Widerspruch!!

"$\Leftarrow$":

Sei G azyklisch.

Beh.: G enthält Knoten v mit indeg(v) = 0.

Wir finden solchen Knoten v wie folgt.

Starte mit bel. Knoten w und gehe entlang eingehender Kanten „rückwärts“. Da Graph azyklisch und endlich, wird kein Knoten auf diesem „Rückweg“ zweimal besucht und daher terminiert dieser Prozess mit einem Knoten v mit indeg(v) = 0.

Jetzt zur topologischen Sortierung:

Beweis per Induktion über Knotenzahl:

$|V| = 1$: trivial.

$|V| > 1$: Wähle (gemäß Beh.) v mit indeg(v) = 0.

Entferne v und sortiere den Restgraph G' topologisch induktiv, d.h.:

Sei $\text{num}' : V' \longrightarrow \{1, \ldots, |V'|\}$ top. Sortierung für G'.

Dann ergibt sich die top. Sort. num durch

$$\text{num}(w) = \begin{cases} \text{num}'(w) + 1, & \text{falls } w \neq v \\ 1, & \text{falls } w = v. \end{cases}$$

□

Grundalgorithmus für topologisches Sortieren:

count $\longleftarrow$ 0;
while $\exists\, v \in V$ mit indeg(v) = 0 do
  count ++;
  num(v) $\longleftarrow$ count;
  streiche v und alle von v ausgehenden Kanten aus G;
od
if count < $|V|$
  then ausgabe(G zyklisch!)
fi

Komplexitätsanalyse:

Problem: Test auf Existenz eines Knotens v mit indeg(v) = 0.

Lösung: Merke alle Knoten mit indeg = 0 in Menge ZERO.
Führe für jeden Knoten Buch über Anzahl jeweils einmündender Kanten.

Algorithmus TOPSORT:

(1) ZERO $\longleftarrow$ Ø;

(2) for all $v \in V$ do INDEG[v] $\longleftarrow$ 0 od;

(3) for all $e \in E$ do INDEG[target(e)] ++ od;

(4) for all $v \in V$ do

(5) if INDEG[v] = 0 then ZERO $\longleftarrow$ ZERO $\cup \{v\}$ fi

(6) od

(7) count $\longleftarrow$ 0;

Fortsetzung $\longrightarrow$ nächste Folie

```
(8)  while ZERO ≠ Ø do
(9)        wähle und streiche v ∈ ZERO ;
(10)       count ++;
(11)       num[v] ⟵ count;
(12)       for all (v, w) ∈ E do
(13)             INDEG[w] – – ;
(14)             if INDEG[w] = 0 then ZERO ⟵ ZERO ∪ {w} fi
(15)       od
(16) od;
(17) if count < |V| then ausgabe(G zyklisch!) fi
```

Satz:

Topologische Sortierung eines gerichteten Graphen $G = (V, E)$ kann in Zeit $O(n + m)$ (Linearzeit!) berechnet werden.
Hierbei: $n = |V|$, $m = |E|$.

Beweis: Die Korrektheit folgt unmittelbar mit dem Lemma.
Zur Komplexität: Übungsaufgabe!

□

7.3 Kürzeste/Billigste Wege in Graphen

Sei $G = (V, E)$ gerichteter Graph mit **Kostenfunktion** $c: E \longrightarrow \mathbf{R}$, ein sogenanntes **Netzwerk**.

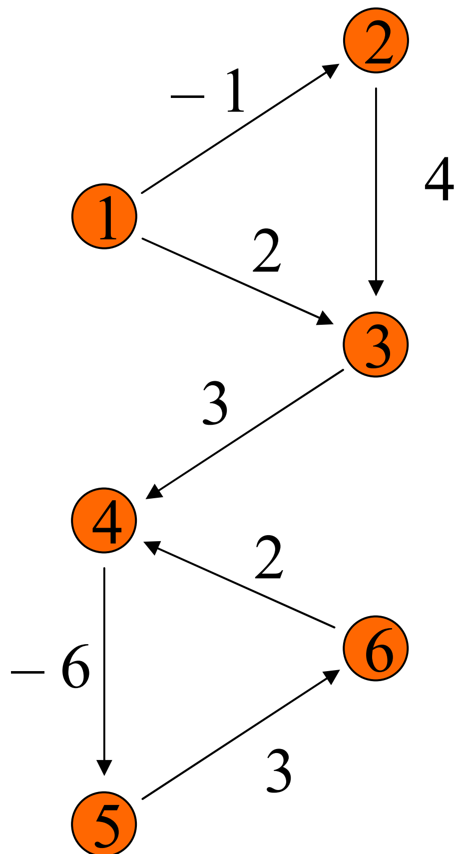
Sei $p = (v_0, ..., v_k)$ Pfad von $v_0 = v$ nach $v_k = w$.
Dann heißt

$$c(p) := \sum_{i=0}^{k-1} c(v_i, v_{i+1})$$

Kosten des Pfades von v nach w.

$\mathrm{dist}(v, w) = \inf \{c(p) \mid p \text{ ist Pfad von } v \text{ nach } w\}$
heißt **Entfernung**.

Beispiel:



dist(1,2) = – 1
dist(3,1) = ∞
dist(3,4) = – ∞, da
Pfad $(3,4,(5,6,4)^i)$ Kosten $3 - i$
hat für alle $i \geq 0$.

negativer Zyklus

3 grundsätzliche Problemstellungen:

1. single pair shortest paths
   (kürzeste Wege zwischen einem Knotenpaar)

2. single source shortest paths
   (kürzeste Wege von einem Knoten aus)

3. all pairs shortest paths
   (kürzeste Wege zwischen allen Knotenpaaren)

Bemerkung:
Unbekannt, ob (1) leichter/effizienter lösbar als (2) .

7.3.1 Single Source Shortest Paths

Sei s der „Quellknoten". $G = (V, E)$.
Gesucht für alle $u \in V$

$$\delta(u) := \begin{cases} \infty, & \text{falls kein Pfad von } s \text{ nach } u \text{ in } G, \\ \text{dist}(s,u), & \text{sonst.} \end{cases}$$

<u>Lemma:</u>

Sei $u \in V$. Dann gilt:

i) $\delta(u) = -\infty$ gdw. u ist erreichbar aus negativem Zyklus, der von s aus erreichbar ist.

ii) $\delta(u) \in \mathbf{R} \Rightarrow \exists$ billigster einfacher Pfad von s nach u mit Kosten $\delta(u)$.

Beweis:

i) "$\Leftarrow$": klar!

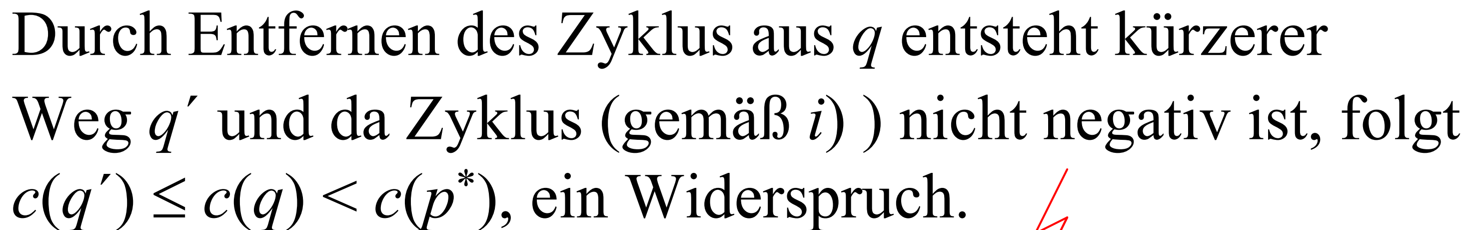
"$\Rightarrow$": Sei $c := \sum_{e \in E} |c(e)|$ und sei p ein kürzester Weg von s nach u mit $c(p) < -c$.

Dann muss p Zyklus q enthalten, d.h. p ist nicht einfach. Wäre nun $c(q) \geq 0$, so könnten wir q aus p entfernen und so p verkürzen/verbilligen.

ii) Falls $\delta(u) \in \mathbf{R}$, so gibt es Pfad von s nach u.

Wir zeigen: $\delta(u) = \min\{\, c(p) \mid p$ ist Pfad von s nach u und p ist einfach $\}$.

Sei p^* billigster einfacher Weg von s nach u.
Ist Behauptung falsch, dann gibt es einen nichteinfachen Weg q von s nach u mit $c(q) < c(p^*)$.

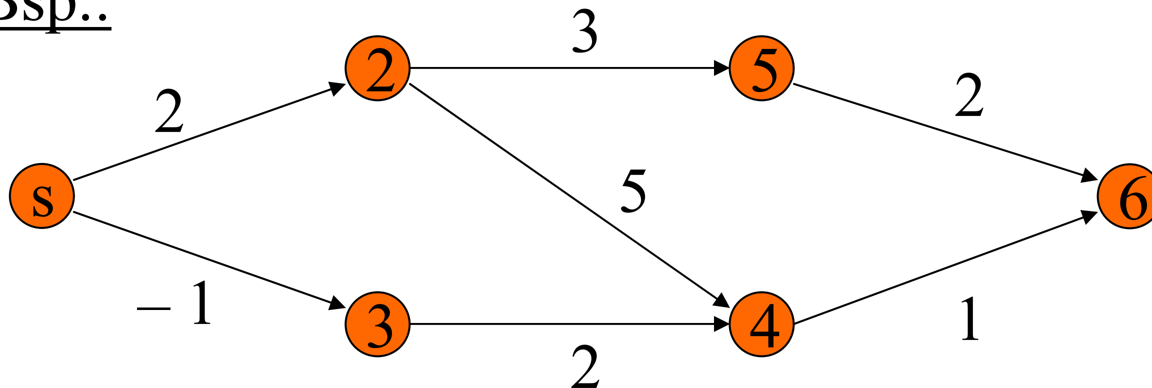


Durch Entfernen des Zyklus aus q entsteht kürzerer Weg q' und da Zyklus (gemäß i)) nicht negativ ist, folgt $c(q') \leq c(q) < c(p^*)$, ein Widerspruch. ↯

$\square$

7.3.1.1 Der geg. Graph $G = (V, E)$ ist azyklisch.

Bsp.:



$$\delta(s) = 0, \qquad \delta(2) = 2, \qquad \delta(3) = -1,$$

$$\delta(4) = \inf\{7,1\} = 1,$$

$$\delta(5) = 2 + 3 = 5,$$

$$\delta(6) = \inf\{5+2,\, 7+1,\, 1+1\} = 2$$

Nachfolgender Algorithmus setzt voraus, dass Graph topologisch sortiert ist, „also" $V = \{1, 2, \ldots, n\}$.

<u>Algorithmus:</u>

$d(s) \leftarrow 0$;

Pfad(s) $\leftarrow s$; (* Pfad als Liste repräsentiert *)

<u>for</u> <u>all</u> $v \in V \setminus \{s\}$ <u>do</u> $d(v) \leftarrow \infty$ <u>od</u>;

<u>for</u> $v = s + 1$ <u>to</u> n <u>do</u>

$d(v) \leftarrow \min \{d(u) + c(u,v) \mid (u,v) \in E\}$

(* u^* sei hierbei ein optimaler u-Wert *)

Pfad(v) $\leftarrow$ Pfad(u^*) $\circ\, v$

<u>od</u>

Satz:

Nach Ausführung von obigem Algorithmus gilt

1) $d(v) = \delta(v) \qquad \forall v \in V$
2) $d(v) < \infty \Rightarrow$ Pfad(v) ist billigster Weg von s nach v.
3) Algorithmus hat Laufzeit $O(n + m)$, wobei $n = |V|$ und $m = |E|$.

Beweis:

Korrektheit (1) + 2)): Induktion über v

$v < s$: $\delta(v) = \infty$, da v von s aus nicht erreichbar.
$d(v) = \infty$, da nach Init. nicht mehr verändert

$v = s$: $\delta(v) = d(v) = 0$. Pfad(s) = s ist einziger Weg.

$v > s$: Sei $\mathrm{In}(v) := \{ u \in V \mid (u, v) \in E \}$.
Nach Ind´vor. gilt für $u \in \mathrm{In}(v)$ schon
$d(u) = \delta(u)$ und Pfad(u) ist berechnet.

Ist $\mathrm{In}(v) = \emptyset$, dann $d(v) = \delta(v) = \infty$.

Ist $\mathrm{In}(v) \neq \emptyset$, so wähle u^* mit
$d(u^*) + c(u^*, v)$ minimal.

Falls $d(u^*) = \infty$, so auch $d(v) = \delta(v) = \infty$.
Sonst ergibt sich billigster Weg nach v
durch Konkatenation und

$$\delta(v) = \delta(u^*) + c(u^*, v) = d(v).$$

Laufzeit:

Topologisches Sortieren kostet Zeit $O(n + m)$.

Initialisierung: $O(n)$

„Hauptteil“: Komplexität wird i.w. bestimmt durch

$$\sum_{v \in V} | \operatorname{In}(v) | = m ,$$

also $O(n + m)$. □

7.3.1.2 Alle Kantenkosten nichtnegativ (d.h. $\forall e \in E, c(e) \geq 0$), Zyklen erlaubt

⟶ Dijkstras Algorithmus:

Nachfolgend nur Kosten kürzester Pfad berechnet;
eigentliche Pfadberechnung analog vorher.

Idee: Für Menge S von Knoten mit bereits bestimmtem kürzesten Pfad muss es Knoten $x \in V \setminus S$ geben, sodass kürzester Weg von s nach x über die Knoten in S geht und dann noch eine Kante aus S nach x.

Genauer: x ist Knoten, der für alle $u \in S$ den Ausdruck $\delta(u) + c(u, x)$ minimiert.

Nachfolgend:

$S :=$ Menge der Knoten u mit bekanntem $\delta(u)$.

$S' :=$ Menge der Knoten aus $V \setminus S$, die Nachbarn in S haben

Algorithmus:

$S \leftarrow \{s\};\ d(s) \leftarrow 0;$

$S' \leftarrow \text{OutAdj}(s);$

for all $u \in S'$ do;

$\quad d'(u) \leftarrow c(s,u);$

for all $u \in V - (S \cup S')$ do

$\quad d'(u) \leftarrow \infty;$

while $S' \neq \emptyset$ do

$x \leftarrow$ derjenige Knoten aus S' mit minimalem d' - Wert;

$d(x) \leftarrow d'(x);$

$S \leftarrow S \cup \{x\};$

$S' \leftarrow S' \setminus \{x\};$

for all $u \in \text{OutAdj}(x)$ do

if $u \notin S$

then $S' \leftarrow S' \cup \{u\};$

$d'(u) \leftarrow \min\{d'(u), d(x) + c(x,u)\}$

fi

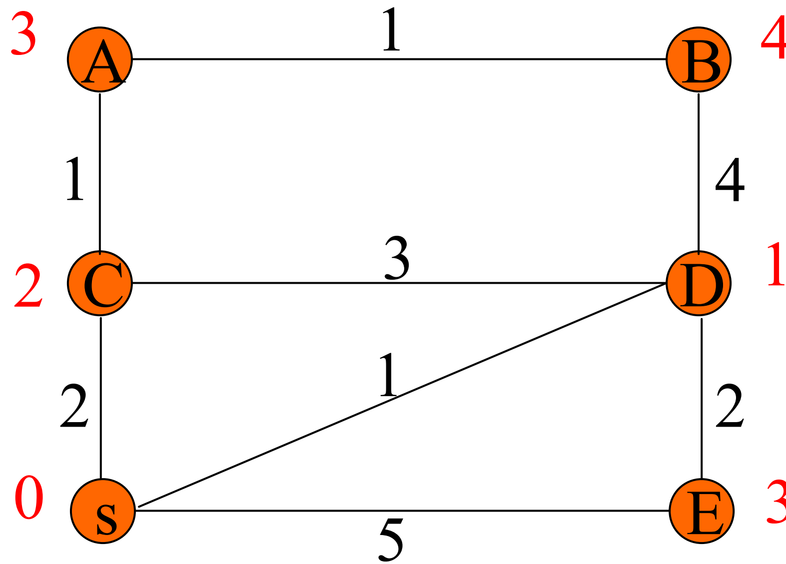
od

od

Beispiel (Dijkstra, Ungerichteter Graph):

S = {s},
S′={C, D, E}
d(s) = 0,
d′(C) = 2,
d′(D) = 1,
d′(E) = 5

S = {s, D},
S′={C, E, B}
d(D) = 1,
d′(C) = 2,
d′(E) = 3,
d′(B) = 5



S = {s, D, C},
S′={E, B, A}
d(C) = 2,
d′(B) = 5,
d′(E) = 3,
d′(A) = 3

S = {s, D, C, A},
S′={E, B}
d(A) = 3
d′(B) = 4,
d′(E) = 3

S = {s, D, C, A, E},
S′={B}
d(E) = 3
d′(B) = 4,

S = {s, D, C, A, E, B},
d(B) = 4,

Beispiel (Dijkstra, Gerichteter Graph):

Lemma:

Sei $x \in S'$ so gewählt, dass $d'(x)$ minimal ist.
Dann gilt $d'(x) = \delta(x)$.

Beweis:

Sei p ein billigster Weg von s nach x derart, dass alle Knoten (bis auf x) in S liegen.

Angenommen, es gäbe einen billigeren Weg q von s nach x.

q muss einen ersten Knoten v in $V \setminus S$ haben und nach Wahl von x ist $d'(v) \geq d'(x)$.

Da alle Kanten nichtnegativ sind, gilt aber

$$c(q) \geq d'(v) \geq d'(x) = d(x) = c(p),$$

ein Widerspruch.

□

Zur Laufzeit:

Implementiere S und S' als Bitvektoren und d und d' als Arrays.

- Aufwand für „for all $u \in \text{OutAdj}(x)$ ":

$$\sum_{x \in V} |\text{OutAdj}(x)| = O(m+n).$$

- n-maliges Minimieren über $S' : O(n^2)$.

Insgesamt ergibt sich Laufzeit $O(n^2+m)$,
für dichte Graphen $(m \approx n^2)$ ausreichend.

Alternativ:

Speichere S' in Heap (balancierter Baum), gemäß d'-Werten geordnet.

Damit Laufzeit $O((m+n)\log n)$, da nun Einfügen und Löschen in S' je $O(\log n)$ Zeit kostet.

Für dünne Graphen ist dies eine Verbesserung.

Satz:

Mit Hilfe des Algorithmus von Dijkstra läßt sich das Single Source Shortest Path Problem in Laufzeiten $O(n^2)$ bzw. $O((m+n)\log n)$ lösen.

Mitteilung:

Mit Hilfe sogenannter Fibonacci-Heaps läßt sich die Laufzeit des Algorithmus von Dijkstra asymptotisch zu $O(n \log n + m)$ verbessern.

7.3.1.3 Negative Kantenkosten erlaubt, aber keine negativen Zyklen.

→ Bellman-Ford-Algorithmus

Das „Entspannen einer Kante (v,w)" bezeichnen wir mit Relax $((v,w))$:

$$d(w) \leftarrow \min\{d(w), d(v) + c(v,w)\}$$

Offensichtlich: Entspannen einer Kante erhöht keinen d-Wert

Lemma:

Falls für alle $u \in V$ vor dem Entspannen einer beliebigen Kante (v,w) gilt, dass $d(u) \geq \delta(u)$, dann gilt dies auch hinterher.

Beweis:

Jeder Weg von s nach v zusammen mit Kante (v,w) ergibt Weg von s nach w.

Deshalb $\delta(v) + c(v,w) \geq \delta(w)$.

Weiterhin: $\delta(v) + c(v,w) \leq d(v) + c(v,w)$

$$\Rightarrow \delta(w) \leq d(v) + c(v,w)$$

$\Rightarrow \delta(w) \leq d(w)$ falls (v,w) „echt“ relaxiert.

□

Algorithmus:

$d(s) \leftarrow 0;$

for all $v \neq s$ do

$d(v) \leftarrow \infty$

while „möglich" do

Entspanne Kanten

od

Beobachtung: $d(u)$ wird immer kleiner, unterschreitet aber nie $\delta(u)$.

Offen bleibt noch die Frage nach der „Entspannungs-Reihenfolge“, sodass d schnell gegen δ konvergiert.

Lemma:

Sei $w \in V$ und sei $\delta(w) < \infty$ und sei (v, w) die letzte Kante auf dem billigsten Weg von s nach w.

Dann gilt:

Falls (v,w) entspannt wird, nachdem $d(v) = \delta(v)$ geworden ist, so ist danach $d(w) = \delta(w)$.

Beweis: Übung!

Algorithmus:

$d(s) \leftarrow 0;$

for all $v \neq s$ do

$d(v) \leftarrow \infty;$

for $i \leftarrow 1$ to $n-1$ do

for all $(v,w) \in E$ do

Relax $((v,w));$

Bem.: im Beispiel spielt die Reihenfolge der Kanten in der for all-Schleife eine Rolle, die Aussagen gelten aber für alle Reihenfolgen

Lemma:

Für i=0,..., n-1 gilt:

Nach dem i-ten Durchlauf der for-Schleife ist $d(w) = \delta(w)$ für alle $w \in V$, für die es einen billigsten Pfad der Länge i von s nach w gibt.

Beweis:

Induktion über i:

$i = 0: \quad d(s) = \delta(s)$, da es keine negativen Zykel gibt.

$i \rightarrow i+1:$

Sei w der Knoten mit billigstem Weg von s nach w der Länge i+1 und sei (v,w) die letzte Kante auf diesem Weg.

Also gibt es billigsten Weg von s nach v der Länge i und nach Induktionsannahme ist nach dem i-ten Durchlauf der for-Schleife $d(v) = \delta(\mathrm{v})$. Im ($i$+1)-ten Durchlauf wird insbesondere (v,w) entspannt:

$$\begin{aligned} \Rightarrow d(w) &= d(v) + c((v,w)) \\ &= \delta(v) + c((v,w)) \\ &= \delta(w) \end{aligned}$$

□

Korollar:

Nach dem (n-1)-ten Schleifendurchlauf gilt für alle v aus V, dass $d(v) = \delta(v)$.

Beweis:

Nach obigem Lemma und der Annahme der Nichtexistenz negativer Zyklen folgt, dass alle billigsten Wege Länge höchstens n-1 haben.

□

Insgesamt:

Satz:

In Laufzeit $O(n \cdot m)$ läßt sich das Single Source Shortest Path Problem lösen, falls der Graph keine negativen Zyklen enthält.

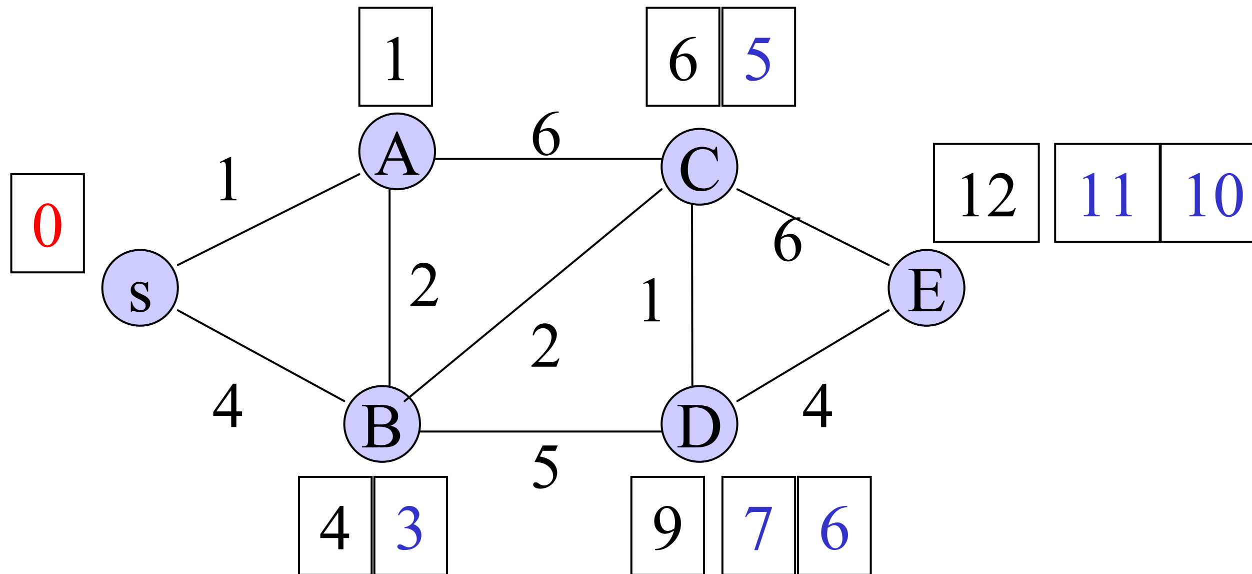
Beispiel (Bellmann-Ford, Gerichteter Graph, positive Kanten):

i = 1, i = 2, i = 3, i = 4, i = 5, i = 6, i = 7, i = 8,

<u>Beispiel</u> (Bellmann-Ford, Gerichteter Graph, negative Kanten, aber keine negativen Zyklen):

i = 1, i = 2, i = 3, i = 4, i = 5, i = 6, i = 7, i = 8,

Beispiel (Bellmann-Ford, Ungerichteter Graph, positive Kanten):



i = 1, i = 2, i = 3, i = 4, i = 5,

Warum braucht der Algorithmus hier n-1 Iterationen bis zur Konvergenz?

Welches wäre der einfachste Graph mit dieser Eigenschaft?

7.3.1.4: Auch negative Zyklen erlaubt.

Algorithmus:

① Führe zuerst die *n*-1 for-Schleifendurchläufe des Algorithmus von Bellman-Ford aus und merke in d_1 die so erzielten *d*-Werte.

② Führe *n* weitere for-Schleifendurchläufe aus und speichere das Ergebnis in d_2.

Lemma:

Für $w \in V$ gilt :

$$\text{i) } d_2(w) = d_1(w) \Rightarrow \delta(\mathrm{w}) = d_1(w)$$

$$\text{ii) } d_2(w) < d_1(w) \Rightarrow \delta(\mathrm{w}) = -\infty$$

Beweis: Übung!

Bemerkung:

Wenn sich nach nur einem weiteren Durchlauf der for-Schleife nach Abschluss von ① noch etwas ändert, so gibt es einen negativen Zyklus, sonst aber nicht.

Um aber alle Zyklen entdecken zu können, benötigt man in ② insgesamt n Durchläufe.

7.3.2 All Pairs Shortest Paths (Floyd-Warshall)

Annahme: Keine negativen Zyklen.

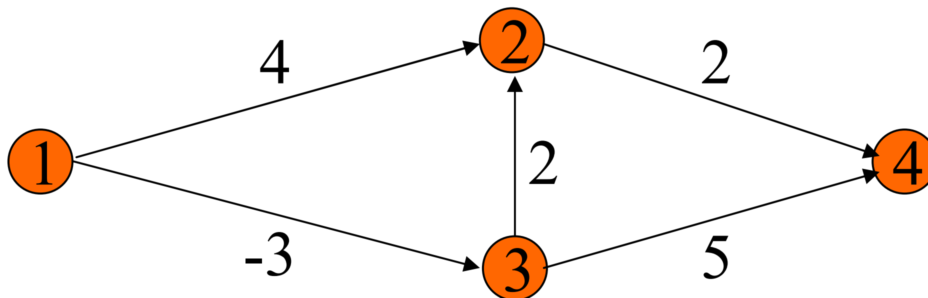
$$V = \{1,2,\ldots,n\}$$

Für $i, j \in V$ und $0 \leq k \leq n$ definiere

$$\delta_k(i, j) := \text{Kosten des billigsten Weges von } i \text{ nach } j,$$
$$\text{dessen innere Knoten} \leq k \text{ sind.}$$

Beispiel:

$$\delta_0(1,4) = \infty$$
$$\delta_1(1,4) = \infty$$
$$\delta_2(1,4) = 6$$
$$\delta_3(1,4) = 1$$
$$\delta_4(1,4) = 1$$

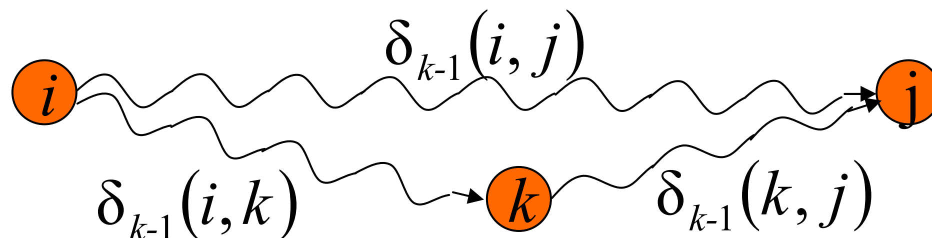


Also: $$\delta_0(i,j) = \begin{cases} c(i,j), \text{ falls } (i,j) \in E. \\ 0 \quad\quad , \text{ falls } i = j. \\ \infty \quad\quad , \text{ sonst.} \end{cases}$$

$\delta_n(i,j) = \delta(i,j) =$ Kosten des billigsten Weges von i nach j.

<u>Frage</u>: Wie berechnet sich δ_k aus δ_{k-1} ?

2 Möglichkeiten: Entweder k Teil eines billigeren Weges oder nicht:



Also gilt: $\delta_k(i,j) = \min\{\delta_{k-1}(i,j),\ \delta_{k-1}(i,k) + \delta_{k-1}(k,j)\}$

Algorithmus (Floyd-Warshall):

<u>for</u> <u>all</u> $i, j \in V$ <u>do</u>

$$\delta_0(i,j) \leftarrow \begin{cases} c(i,j), \text{falls } (i,j) \in E \\ 0 \qquad , \text{falls } i = j \\ \infty \qquad , \text{sonst} \end{cases} ;$$

<u>od</u>

<u>for</u> $k = 1$ <u>to</u> n <u>do</u>

    <u>for</u> $i = 1$ <u>to</u> n <u>do</u>

        <u>for</u> $j = 1$ <u>to</u> n <u>do</u>

$$\delta_k(i,j) \leftarrow \min\{\delta_{k-1}(i,j), \delta_{k-1}(i,k) + \delta_{k-1}(k,j)\}$$

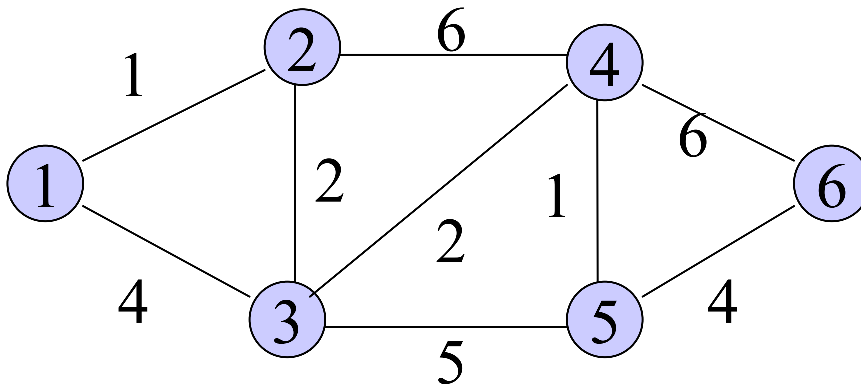
        <u>od</u>

    <u>od</u>

<u>od</u>

Laufzeit: $O(n^3)$

Beispiel (Floyd-Warshall, Ungerichteter Graph, positive Kanten):



$$D_0 = \begin{matrix} 0 & 1 & 4 & u & u & u \\ 1 & 0 & 2 & 6 & u & u \\ 4 & 2 & 0 & 2 & 5 & u \\ u & 6 & 2 & 0 & 1 & 6 \\ u & u & 5 & 1 & 0 & 4 \\ u & u & u & 6 & 4 & 0 \end{matrix}$$

$D_1 = D_0$

$$D_2 = \begin{matrix} 0 & 1 & 3 & 7 & u & u \\ 1 & 0 & 2 & 6 & u & u \\ 3 & 2 & 0 & 2 & 5 & u \\ 7 & 6 & 2 & 0 & 1 & 6 \\ u & u & 5 & 1 & 0 & 4 \\ u & u & u & 6 & 4 & 0 \end{matrix} \qquad D_3 = \begin{matrix} 0 & 1 & 3 & 5 & 8 & u \\ 1 & 0 & 2 & 4 & 7 & u \\ 3 & 2 & 0 & 2 & 5 & u \\ 5 & 4 & 2 & 0 & 1 & 6 \\ 8 & 7 & 5 & 1 & 0 & 4 \\ u & u & u & 6 & 4 & 0 \end{matrix} \qquad D_4 = \begin{matrix} 0 & 1 & 3 & 5 & 6 & 11 \\ 1 & 0 & 2 & 4 & 5 & 10 \\ 3 & 2 & 0 & 2 & 3 & 8 \\ 5 & 4 & 2 & 0 & 1 & 6 \\ 6 & 5 & 3 & 1 & 0 & 4 \\ 11 & 10 & 8 & 6 & 4 & 0 \end{matrix}$$

u.s.w.

Alternativer Algorithmus:

Benutze Algorithmus von Dijkstra bzw. Bellman-Ford

Idee: Führe n-mal für alle möglichen Startknoten Bellman-Ford-Algorithmus aus.

Laufzeit : $O(n \cdot nm)$

Beobachtung:

Es reicht sogar, einmal Bellman-Ford und (n-1)-mal Dijkstra auszuführen.

Laufzeit : $O(n \cdot (n+m) \log n)$

7.3.3 All Pairs Shortest Paths (Matrixmultiplikation)

Annahme: Keine negativen Zyklen.

$$G = (V, E), \quad V = \{1, 2, \ldots, n\}$$

Wir wissen (Lemma): Wenn ein Pfad von i über k nach j der kürzeste Pfad von i nach j ist, dann sind auch die Pfade von i nach k und von k nach j jeweils kürzeste Pfade.

Sei G als Adjazenzmatrix $W = (w_{ij})$ repräsentiert.
Sei p ein kürzester Pfad von i nach j mit m Kanten.
Es gilt: $m < n$.
Falls $i = j$, dann hat p die Länge 0 und keine Kanten.
Falls $i \neq j$, dann zerlegen wir p in Pfad p' von i nach k und Kante (k, j), wobei p' dann m-1 Kanten hat. Es gilt:

$$\delta(i, j) = \delta(i, k) + w_{kj}$$

Sei $l_{ij}^{(m)}$ der kürzeste Pfad zwischen Knoten i und j, der <u>höchstens</u> m Kanten hat. Für $m = 0$ gilt

$$l_{ij}^{(0)} = \begin{cases} 0 & \text{falls } i = j \\ \infty & \text{sonst} \end{cases}$$

Für $m \geq 1$ berechnen wir

$$\begin{aligned} l_{ij}^{(m)} &= \min\left(l_{ij}^{(m-1)}, \min_{1 \leq k \leq n} \left\{ l_{ik}^{(m-1)} + w_{kj} \right\}\right) \\ &= \min_{1 \leq k \leq n} \left\{ l_{ik}^{(m-1)} + w_{kj} \right\} \end{aligned}$$

denn $w_{jj} = 0$ für alle j

Da ein kürzester Weg von i nach j höchstens n-1 Kanten hat, gilt

$$\delta(i, j) = l_{ij}^{(n-1)} = l_{ij}^{(n)} = l_{ij}^{(n+1)} = \ldots$$

Wir berechnen also die $l_{ij}^{(m)}$ in einer Folge von Matrizen $L^{(1)}, L^{(2)}, \ldots L^{(n-1)}$, mit $L^{(m)} = (l_{ij}^{(m)})$.

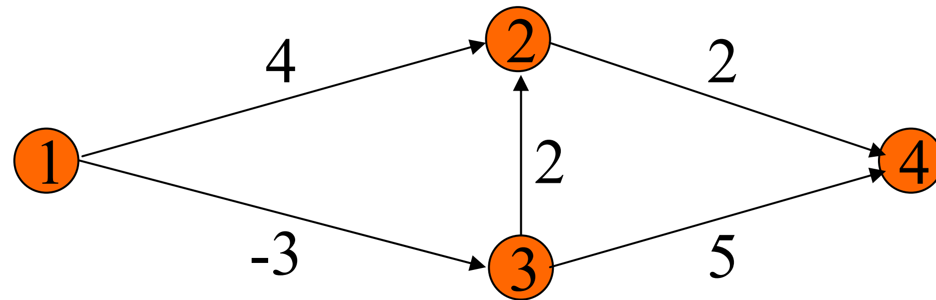
Algorithmus Slow-All-Pairs-Shortest-Paths (W)

1) $L^{(\text{old})} := W$
2) for $m := 2$ to n-1 do begin
3) for $i := 1$ to n do
4) for $j := 1$ to n do begin
5) $l_{ij}^{(new)} := \infty$
6) for $k := 1$ to n do
7) $l_{ij}^{(new)} := \min(l_{ij}^{(new)}, l_{ik}^{(old)} + w_{kj})$
8) end
9) $L^{(\text{old})} := L^{(\text{new})}$
10) end

Zeitaufwand: $O(n^4)$

Verwendung von nur zwei Matrizen $L^{(\text{new})}$ und $L^{(\text{old})}$

Beispiel:

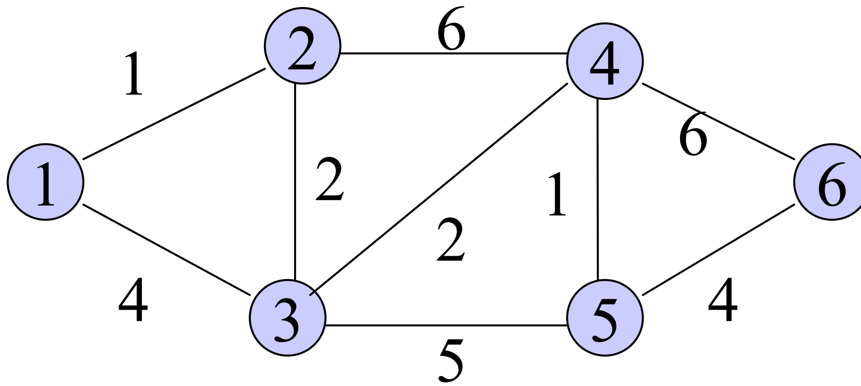


$$L^{(1)} = \begin{bmatrix} 0 & 4 & -3 & \infty \\ \infty & 0 & \infty & 2 \\ \infty & 2 & 0 & 5 \\ \infty & \infty & \infty & 0 \end{bmatrix}$$

$$L^{(2)} = \begin{bmatrix} 0 & -1 & -3 & 2 \\ \infty & 0 & \infty & 2 \\ \infty & 2 & 0 & 4 \\ \infty & \infty & \infty & 0 \end{bmatrix}$$

$$L^{(3)} = \begin{bmatrix} 0 & -1 & -3 & 1 \\ \infty & 0 & \infty & 2 \\ \infty & 2 & 0 & 4 \\ \infty & \infty & \infty & 0 \end{bmatrix}$$

Beispiel (Matrixmultiplikationsalgorithmus):



$$L^1 = W = \begin{matrix} 0 & 1 & 4 & u & u & u \\ 1 & 0 & 2 & 6 & u & u \\ 4 & 2 & 0 & 2 & 5 & u \\ u & 6 & 2 & 0 & 1 & 6 \\ u & u & 5 & 1 & 0 & 4 \\ u & u & u & 6 & 4 & 0 \end{matrix}$$

$$L^2 = \begin{matrix} 0 & 1 & 3 & 6 & 9 & u \\ 1 & 0 & 2 & 4 & 7 & 12 \\ 3 & 2 & 0 & 2 & 3 & 8 \\ 6 & 4 & 2 & 0 & 1 & 5 \\ 9 & 7 & 3 & 1 & 0 & 4 \\ u & 12 & 8 & 5 & 4 & 0 \end{matrix}$$

$$L^3 = \begin{matrix} 0 & 1 & 3 & 5 & 7 & 12 \\ 1 & 0 & 2 & 4 & 5 & 10 \\ 3 & 2 & 0 & 2 & 3 & 7 \\ 5 & 4 & 2 & 0 & 1 & 5 \\ 7 & 5 & 3 & 1 & 0 & 4 \\ 12 & 10 & 7 & 5 & 4 & 0 \end{matrix}$$

$$L^4 = \begin{matrix} 0 & 1 & 3 & 5 & 6 & \boxed{11} \\ 1 & 0 & 2 & 4 & 5 & 9 \\ 3 & 2 & 0 & 2 & 3 & 7 \\ 5 & 4 & 2 & 0 & 1 & 5 \\ 6 & 5 & 3 & 1 & 0 & 4 \\ \boxed{11} & 9 & 7 & 5 & 4 & 0 \end{matrix}$$

10

L^5: 10

Geht es auch schneller? Ja, durch Quadrierung!

Bisher hatten wir berechnet:

$$L^{(1)} = L^{(0)} * W \quad = W$$
$$L^{(2)} = L^{(1)} * W \quad = W^2$$
$$L^{(3)} = L^{(2)} * W \quad = W^3$$

...

$$L^{(n-1)} = L^{(n-2)} * W \quad = W^{n-1}$$

Jetzt berechnen wir schneller:

$$L^{(1)} = \quad = W$$
$$L^{(2)} = W * W \quad = W^2$$
$$L^{(4)} = W^2 * W^2 \quad = W^4$$
$$L^{(8)} = W^4 * W^4 \quad = W^8$$

...

Algorithmus Fast-All-Pairs-Shortest-Paths (W)

1) $L^{(\text{old})} := W$
2) $m := 1$
3) while $m < n$-1 do
4) for $i := 1$ to n do
5) for $j := 1$ to n do begin
6) $l_{ij}^{(new)} := \infty$
7) for $k := 1$ to n do
8) $l_{ij}^{(new)} := \min(l_{ij}^{(new)}, l_{ik}^{(old)} + l_{kj}^{(old)})$
9) end
10) $m := 2m$
11) $L^{(\text{old})} := L^{(\text{new})}$
12) endwhile

Zeitaufwand: $O(n^3 \log n)$, das ist aber immer noch langsamer als der Floyd-Warshall-Algorithm.

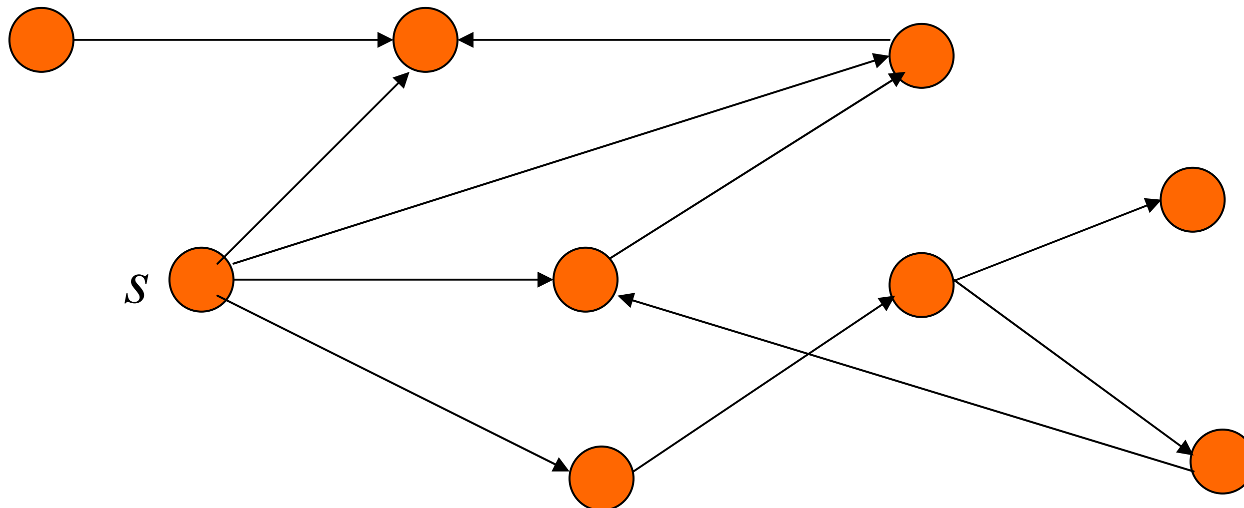
7.4 Depth First Search (Tiefensuche)

⟶ Durchmusterung von Graphen.

Wichtige Eigenschaften:

Mit DFS Graph systematisch von einem Knoten *s* aus zu durchmustern, dabei immer zuerst „in die Tiefe“.

Beispiel:



Rekursion:

```
procedure DFS (s)
        besucht [s] ←—— true;
        drucke (s);
        for all (s, v) ∈ E do
                if not besucht [v]
                  then DFS(v)
                fi;
        od
```

DFS teilt die Kanten des Graphen in vier Klassen ein, je nach Art des Besuchs:

Wir betrachten gerade die gerichtete Kante (v,w):

1. (v,w) gehört zu **Baumkanten** T, falls w noch nicht besucht.

2. (v,w) gehört zu **Vorwärtskanten** F, falls w schon besucht und „$v \xrightarrow[T]{*} w$", d.h. $\exists$ Pfad von v nach w bestehend aus lauter Baumkanten.

3. (v,w) gehört zu **Rückwärtskanten** B, falls w schon besucht und $w \xrightarrow[T]{*} v$.

4. (v,w) gehört zu **Querkanten** C, falls w schon besucht, aber weder $v \xrightarrow[T]{*} w$ noch $w \xrightarrow[T]{*} v$.

Erweiterung um dfsnum(v) und compnum(v):

- dfsnum(v): Reihenfolge der rekursiven DFS-Aufrufe.
- compnum(v): Abschlussreihenfolge der DFS-Aufrufe.

Hauptprogramm:

```
for all v ∈ V do besucht[v] ⟵ false od;
z1 ⟵ z2 ⟵ 0;
T ← B ← F ← C ← ∅;
for all v ∈ V do
        if not besucht[v] then DFS(v) fi
od
```

Für die Implementierung brauchen wir 2 Zähler $z1, z2$:

(1) procedure DFS(v)
(2) besucht[v] $\longleftarrow$ true; $z1 \longleftarrow z1+1$
(3) dfsnum[v] $\longleftarrow z1$;
(4) for all $(v,w) \in E$ do
(5) if not besucht[w] then $T \leftarrow T \cup \{(v,w)\}$
(6) DFS(w)
(7) else if $v \xrightarrow[T]{*} w$ then $F \leftarrow F \cup \{(v,w)\}$
(8) else if $w \xrightarrow[T]{*} v$ then $B \leftarrow B \cup \{(v,w)\}$
(9) else $C \leftarrow C \cup \{(v,w)\}$
(10) fi
(11) fi
(12) fi
(13) od
(14) $z2 \leftarrow z2+1$;
(15) compnum[v] $\longleftarrow z2$

Beispiel:

dfsnum[v]

s

compnum[v]

Knoten	dfsnum	compnum
i	9	9
b	2	1
d	4	2
f	6	3
a	1	8
c	3	6
e	5	5
g	7	4
h	8	7

Laufzeit: $O(n+m)$

da a) Einzellaufzeit eines Aufrufs (ohne Rekursion) ist $O(1+\text{outdeg}(v))$

b) für jeden Knoten einmal DFS-Aufruf

$$\Rightarrow O\left(\sum_{v\in V}(1+\text{out}\deg(v))\right) = O(n+m).$$

Beispiel (Klassifikation der Kanten durch DFS):

Knoten		
s	1	12
A	2	10
B	12	11
C	3	9
D	4	6
E	5	5
F	10	8
G	6	3
H	11	7
I	9	4
J	7	2
K	8	1
L	13	13

Beispiel (Klassifikation der Kanten durch DFS):

s 1 12
B 12 11
F 10 8
H 11 7
C 3 9
G 6 3
L 13 13
A 2 10
E 5 5
J 7 2
D 4 6
I 9 4
K 8 1

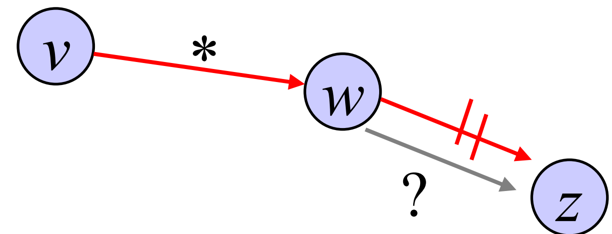
Lemma (Eigenschaften, Teil 1):

a) Kantenklassen T,B,F,C bilden Partition der Kantenmenge E

b) T entspricht dem Wald der rekursiven Aufrufe

c) $v \xrightarrow[T]{*} w$ gdw. $\text{dfsnum}[v] \leq \text{dfsnum}[w]$ und $\text{compnum}[w] \leq \text{compnum}[v]$.

d) Seien $v, w, z \in V$ mit $v \xrightarrow[T]{*} w, (w,z) \in E$ und $\neg(v \xrightarrow[T]{*} z)$, dann

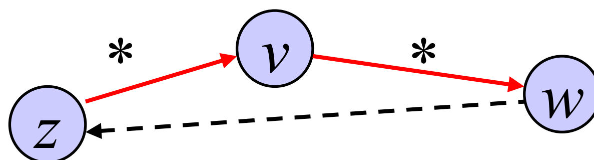


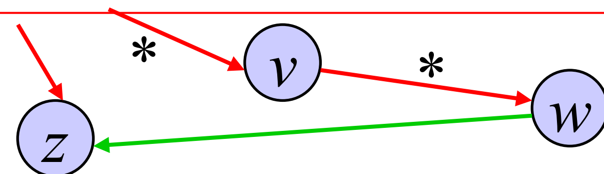
i) $\text{dfsnum}[z] < \text{dfsnum}[v]$

ii) $(w,z) \in B \cup C$

iii) $\text{compnum}[z] > \text{compnum}[v]$ gdw. $(w,z) \in B$.

iv) $\text{compnum}[z] < \text{compnum}[v]$ gdw. $(w,z) \in C$.





<u>Beweis</u>: a), b) klar: direkt aus Algorithmus.

c) $v \xrightarrow[T]{*} w \Leftrightarrow$ Aufruf DFS(w) geschachtelt in DFS(v)

$\Leftrightarrow \text{dfsnum}[v] \leq \text{dfsnum}[w]$ und

$\text{compnum}[v] \geq \text{compnum}[w]$.

d) • $v \xrightarrow[T]{*} w \Rightarrow \text{dfsnum}[v] \leq \text{dfsnum}[w]$

• $(w, z) \in E \Rightarrow$ DFS(z) wird aufgerufen, bevor DFS(w) endet, und auch bevor DFS(v) endet.

• $\neg\left(v \xrightarrow[T]{*} z\right) \Leftrightarrow$ DFS(z) *nicht* in DFS(v) geschachtelt

$\Rightarrow$ DFS(z) startet vor Aufruf DFS(v).

$\Rightarrow \text{dfsnum}[z] < \text{dfsnum}[v]$ (i)

$\xrightarrow{\text{e) (Teil 2)}} (w, z) \in B \cup C$ (ii)

$(w, z) \in B \Leftrightarrow z \xrightarrow[T]{*} w$ nach Definition von B

$\Leftrightarrow z \xrightarrow[T]{*} v$ da $v \xrightarrow[T]{*} w$ und $\neg\left(v \xrightarrow[T]{*} z\right)$

$\Leftrightarrow \text{compnum}[z] > \text{compnum}[v]$ (iii)

$\xRightarrow{\text{(ii), (iii)}}$ (iv) □

Lemma (Eigenschaften, Teil 2):

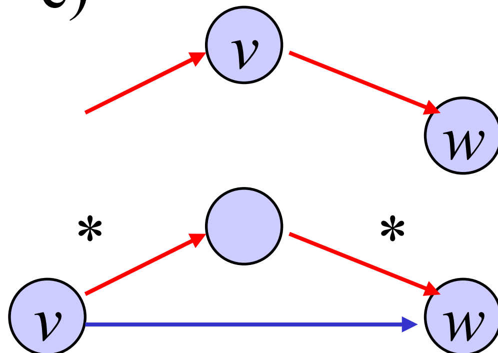
Sei $(v,w) \in E$:

e) $(v,w) \in T \cup F \Leftrightarrow \text{dfsnum}[v] \leq \text{dfsnum}[w]$.

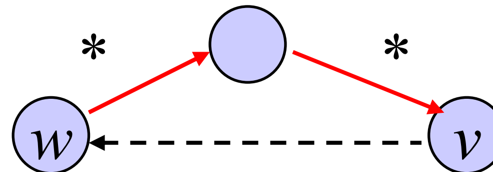
f) $(v,w) \in B \Leftrightarrow \text{dfsnum}[w] < \text{dfsnum}[v]$ und $\text{compnum}[w] > \text{compnum}[v]$.

g) $(v,w) \in C \Leftrightarrow \text{dfsnum}[w] < \text{dfsnum}[v]$ und $\text{compnum}[w] < \text{compnum}[v]$.

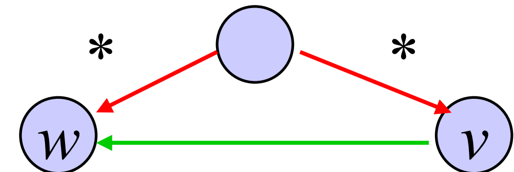
e)



f)



g)



w vor *v* besucht

Beweis:

e) $(v,w) \in T \cup F \Rightarrow v \xrightarrow[T]{*} w \Rightarrow \text{dfsnum}[v] \leq \text{dfsnum}[w]$

Sei $\text{dfsnum}[v] \leq \text{dfsnum}[w]$.

Da $(v,w) \in E$, muss $\text{DFS}(w)$ vor Abschluss von $\text{DFS}(v)$ aufgerufen werden.

$\Rightarrow$ echte Schachtelung
$v \xrightarrow[T]{*} w$

$\Rightarrow (v,w) \in T \cup F$

f), g) folgen aus d) und e) ...

$\square$

Bemerkungen:

- T, B, F, C werden im wesentlichen durch dfsnum, compnum festgelegt (vgl. Lemma!)

- In azyklischen Graphen gibt es keine Rückwärtskanten, d.h.
  $\forall (v,w) \in E : \text{compnum}[v] > \text{compnum}[w]$
  $\Rightarrow$ Nummerierung $\text{num}(v) := n + 1 - \text{compnum}[v]$
  ergibt topologische Sortierung.

Mitteilung:

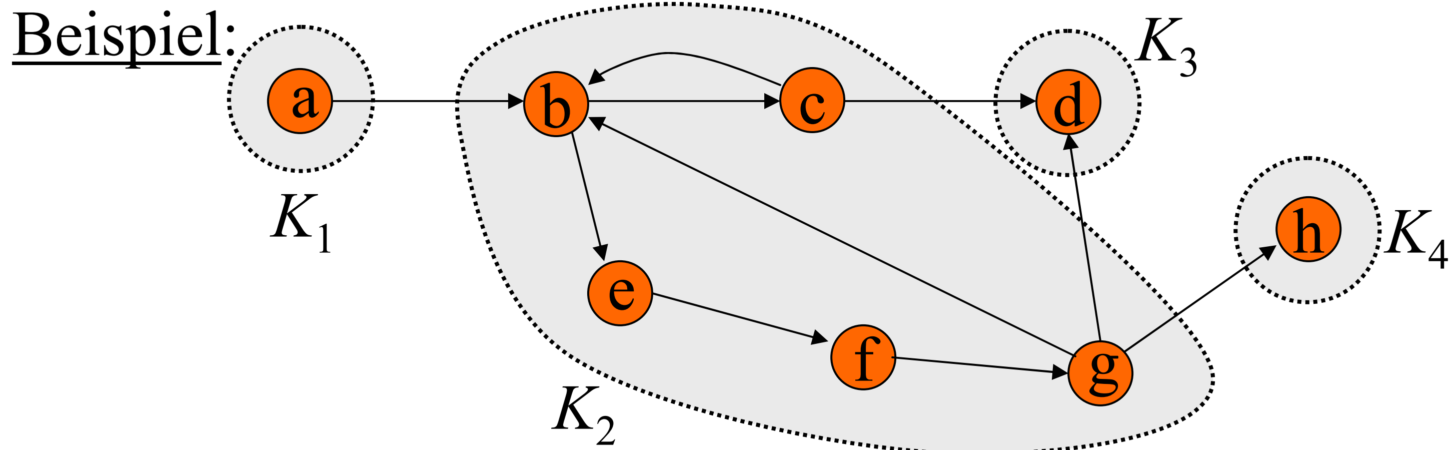
Zusammenhangskomponenten in ungerichteten Graphen können mit Hilfe von DFS in Zeit $O(n+m)$ berechnet werden.

Starke Zusammenhangskomponenten

Ein gerichteter Graph $G=(V,E)$ heißt **stark zusammenhängend** gdw. $\forall v,w \in V : v \xrightarrow[E]{*} w$.

Die maximalen (bzgl. Mengeninklusion) stark zusammenhängenden Teilgraphen von G heißen **starke Zusammenhangskomponenten (SZK's)**.

Beispiel:



Idee für Algorithmus: Erweitere DFS:

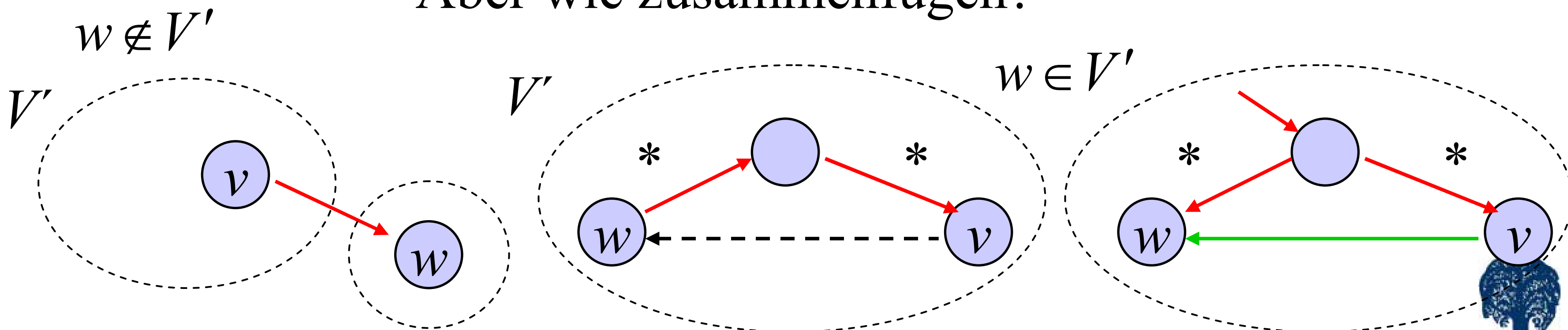
Sei $G' = (V', E')$ bisher erforschter Teilgraph von G.
Verwalte die SZK's von G:

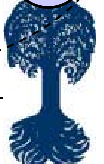
Starte mit $V' = \{a\}, E' = \phi, \text{SZK} = \{\{a\}\}$.
Sei (v,w) die nächste Kante in DFS. Dann ist $v \in V'$.

Ist $w \notin V'$, d.h. $(v,w) \in T$, dann erweitere SZK um $\{w\}$.

Ist $w \in V'$, dann füge mehrere SZK's zu einer zusammen
(falls $(v,w) \in B \cup C$)

Aber wie zusammenfügen?





Bezeichnungen:

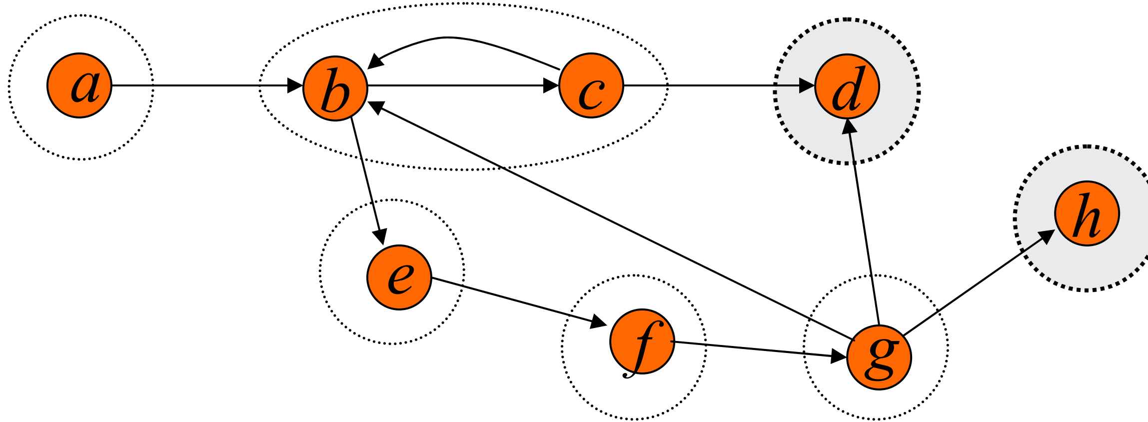
- SZK *K* heißt **abgeschlossen**, wenn alle DFS-Aufrufe für Knoten in *K* abgeschlossen sind.

- **Wurzel** von *K* ist der Knoten mit kleinster dfsnum in *K*.

- Folge von Knoten heißt **unfertig**, falls ihre DFS aufgerufen sind, aber ihre SZK noch nicht abgeschlossen ist. (Sortiert nach dfsnum!)

- **wurzeln** ist Folge der Wurzeln von noch nicht abgeschlossenen SZK´s. (Sortiert nach dfsnum!)

Beispiel:



unfertig:	a	$b\ c$	e	f	g
wurzeln:	a	b	e	f	g

Beobachtung:

1. $\forall v \in \text{unfertig}: \ v \xrightarrow[E]{*} g$
2. $\nexists (v, w) \in E : v$ in abgeschlossener und w in nicht abgeschlossener Komponente.

Von g ausgehende Kanten:

$(g,d) \in C$: tut nichts, d liegt in abgeschlossener SZK.

$(g,b) \in B$: vereinigt vier Komponenten $\{b,c\}, \{e\}, \{f\}, \{g\}$.

→

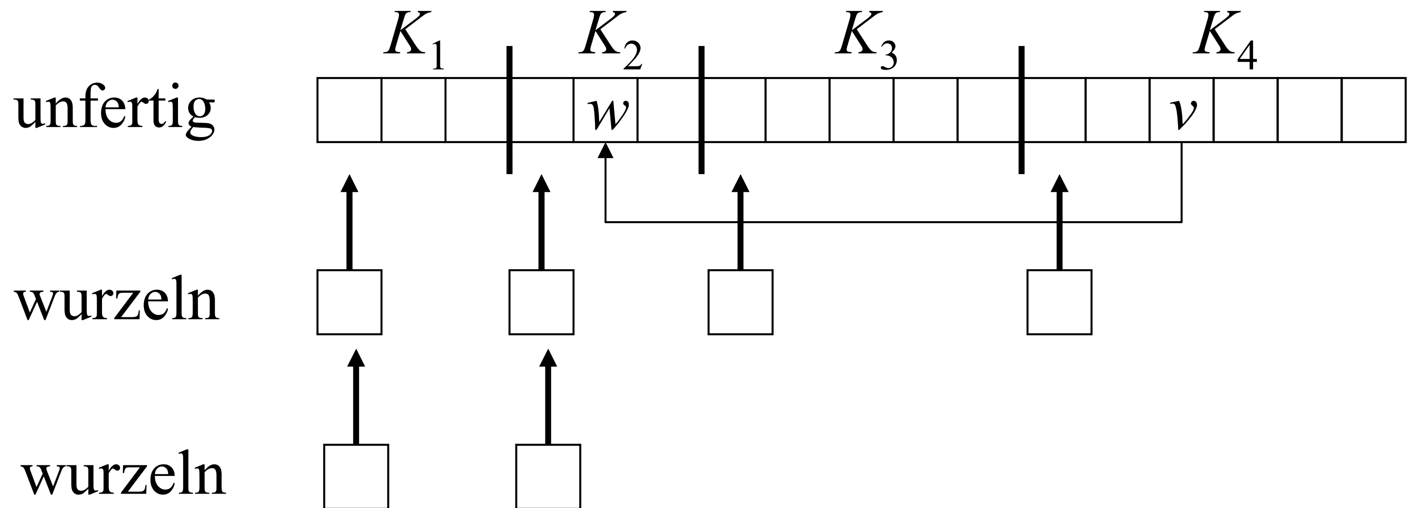
unfertig:	a	$b\ c\ e\ f\ g$
wurzeln:	a	b

$(g,h) \in T$: erzeuge neue Komponente $\{h\}$

Beim Abschluss von DFS(v) teste, ob v Wurzel.
Wenn ja, so gib Komponente mit v als Wurzel aus und streiche sie aus unfertig und v aus wurzeln.

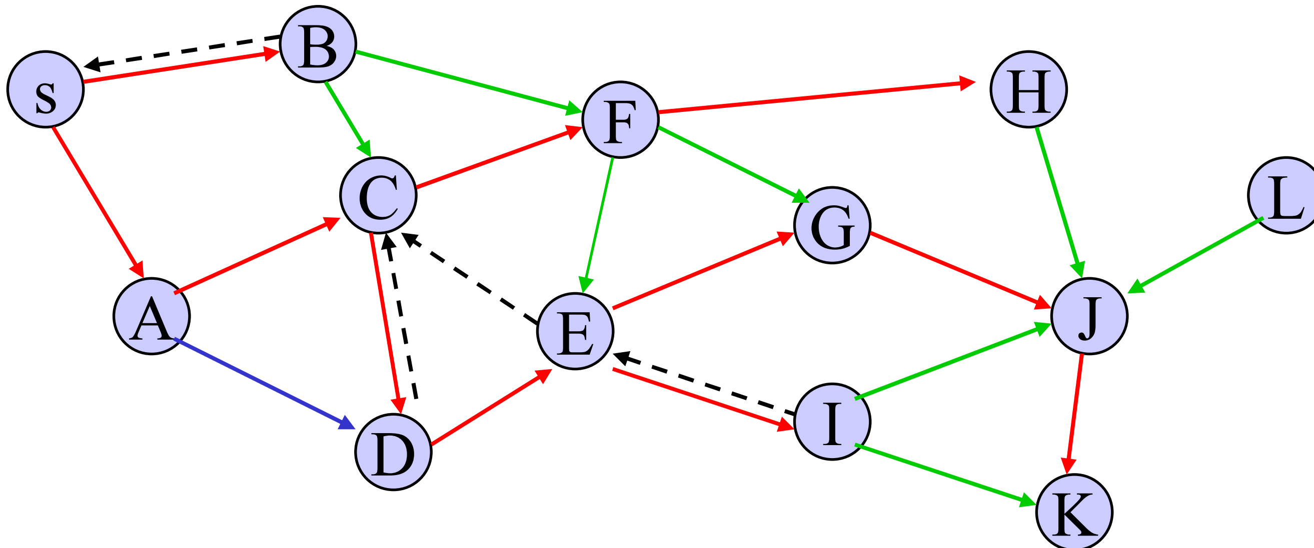
Operationen auf unfertig und wurzeln erlauben Implementierung als Keller (Stack):

z.B. Vereinigen von Komponenten:



K_2, K_3 und K_4 werden vereinigt durch pop(wurzeln) bis top(wurzeln) = Wurzel der SZK von w.

Beispiel (Bestimmung der SZKs durch DFS):



unfertig ~~s~~ ~~A~~ ~~C~~ ~~D~~ ~~E~~ ~~G~~ ~~J~~ ~~K~~ ~~I~~ ~~F~~ ~~H~~ ~~B~~ ~~L~~

wurzeln ~~s~~ ~~A~~ ~~C~~ ~~D~~ ~~E~~ ~~G~~ ~~J~~ ~~K~~ ~~I~~ ~~F~~ ~~H~~ ~~B~~ ~~L~~

print K J G H (F I E D C) A (B s) L

Korrektheit:

Invarianten:

I1: $\nexists (v,w) \in E$ mit v in abgeschlossener und w in nicht-abgeschlossener SZK.

I2: Nichtabgeschlossene SZK´s liegen auf einem Pfad, insbesondere ihre Wurzeln liegen auf einem Baumpfad

I3: Knoten jeder nicht abgeschlossenen Komponente K_i bilden Intervall in „unfertig“, erstes Element des Intervalls ist die Wurzel von K_i.

Zu zeigen: Invarianten bleiben beim Einfügen neuer Kanten $(v,w) \in E$ erfüllt.

<u>1.Fall</u>: $(v, w) \in T$:

push (w, unfertig); push (w, wurzeln)

I1: Es wird keine SZK abgeschlossen. ✓

I2: Sei r Wurzel der Komponente von v.
Dann $r \xrightarrow[T]{*} v$ und $(v, w) \in T$
$\Rightarrow r \xrightarrow[T]{*} w$

I3: Trivial!

2.Fall: $(v,w) \notin T$:

a) *w* in abgeschlossener SZK: Tue nichts.

b) *w* in nicht abgeschlossener SZK: Vereinige oberste SZK´s bis zu der, in der *w* liegt.

In a) passiert nichts. Nur b) zu betrachten:

I1: Es wird keine SZK abgeschlossen...

I2: Pfad wird verkürzt.

I3: Löschen der obersten Wurzel vereinigt Intervalle zu einem neuen. Da vorher I2 galt, bildet dieses eine Intervall eine einzige Komponente.

Durch nachfolgenden Algorithmus (Programm) erhalten wir:

Satz:
Starke Zusammenhangskomponenten eines gerichteten Graphen können in Zeit $O(n+m)$ berechnet werden.

Das Hauptprogramm wird erweitert um:

unfertig $\longleftarrow$ wurzeln $\longleftarrow$ leerer Keller

```
(1)  procedure DFS(v)
(2)  besucht[v] := true; z1:= z1+1; dfsnum[v] := z1;
(3)  push(v, unfertig); push(v, wurzeln);
(4)  for all (v, w) ∈ E  do
(5)       if not besucht[w] then DFS(w)
(6)       else if  w ∈ unfertig
(7)                 then while dfsnum[top(wurzeln)] > dfsnum[w] do
(8)                           pop(wurzeln) od ;
(9)            fi
(10)      fi
(11) od
(12) z2 := z2+1; compnum[v] := z2
(13) if v = top(wurzeln)
(14)      then repeat w := pop(unfertig)
(15)                          print(w)
(16)           until v = w ;
(17)           pop(wurzeln) ;
(18) fi
```

Zeilen (13)-(16) : Ausgabe der SZK mit Wurzel v.

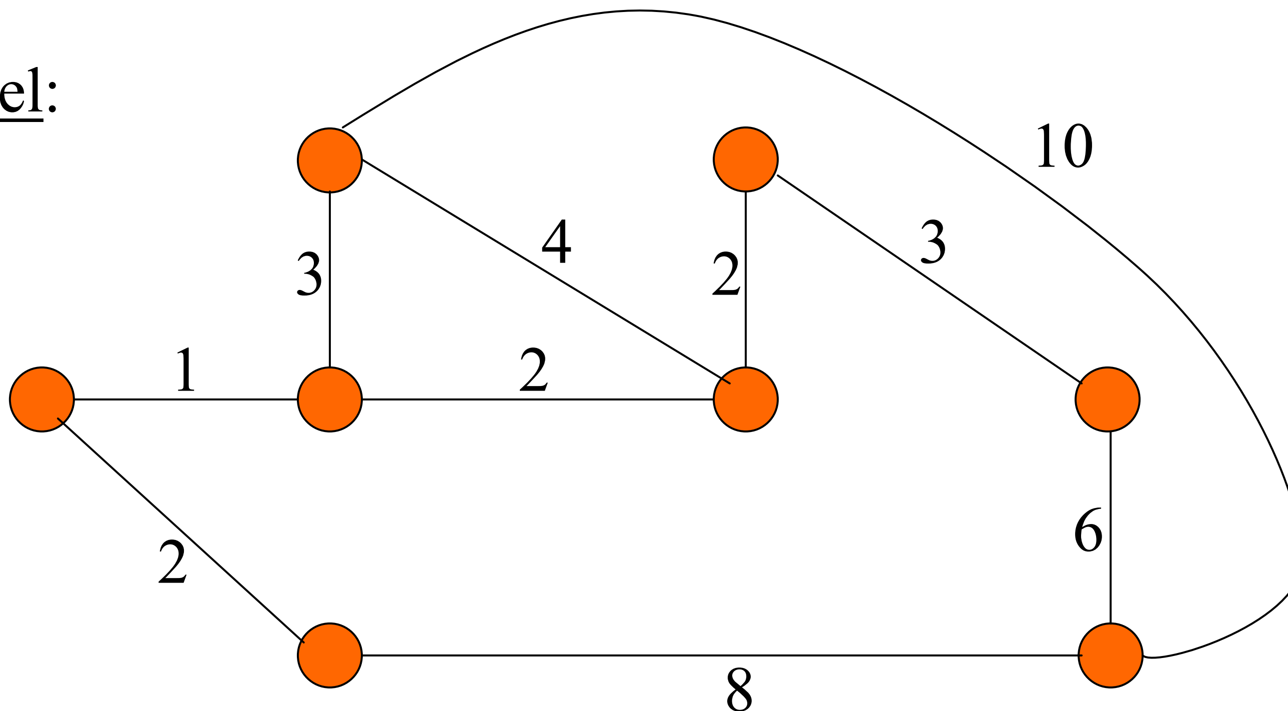
Bemerkungen:

- Test „$w \in$ unfertig“ wird mit zusätzlichem booleschen Feld in_unfertig unterstützt.
- Jeder Knoten wird höchstens einmal nach unfertig bzw. wurzeln aufgenommen.

7.5 Minimal aufspannende Bäume (MST)

Sei $G(V,E)$ ein zusammenhängender, ungerichteter Graph.
Sei $c: E \to \mathbf{R}^+$ eine Kostenfunktion.
Ziel: Berechne $E_T \subseteq E$, sodass $G(V,E_T)$ zusammenhängend ist
<u>und</u> $c(E_T) := \sum_{e \in E_T} c(e)$ kleinstmöglich ist.

<u>Beispiel</u>:



Anwendungsbeispiele:

- Computer-Netzwerke...
- Elektronische Schaltkreise...
- Netz von Telefonanschlüssen...

uvm.

Lemma:

$G(V,E_T)$ ist azyklisch.

Beweis:

Würde $G(V,E_T)$ einen Kreis enthalten, so könnte man eine Kante entfernen, der Graph wäre weiterhin zusammenhängend und die Gesamtkosten geringer.

□

Wir suchen also einen „**aufspannenden Baum**“ mit minimalen Kosten (**MST**, Minimum Spanning Tree).

Kruskals Greedy-Algorithmus für MST

Sortiere Kanten $e_1, e_2, \ldots, e_m$ so, dass
$c(e_1) \leq c(e_2) \leq \ldots \leq c(e_m)$;
$E_T \leftarrow \emptyset$;
for $i \leftarrow 1$ to m do
  if $(V, E_T \cup \{e_i\})$ azyklisch
    then $E_T \leftarrow E_T \cup \{e_i\}$
  fi
od

Satz:

Der Greedy-Algorithmus von Kruskal ist korrekt, d.h. er findet immer einen MST.

Beweis:

Eine Kantenmenge $E' \subseteq E$ heiße „**gut**", falls sie zu einem MST erweiterbar ist.

Behauptung:

Sei $E' \subseteq E$ gut und sei $e \in E \setminus E'$ eine billigste Kante, sodass der Graph $(V, E' \cup \{e\})$ azyklisch ist.
Dann ist auch $E' \cup \{e\}$ gut.

Beweis der Behauptung per Induktion:

Sei $T_1 = (V, E_1)$ ein MST mit $E' \subseteq E_1$
(T_1 existiert, da E' gut).
Ist $e \in E_1$, so fertig.

Ist $e \notin E_1$, so betrachte Graph $H = (V, E_1 \cup \{e\})$.
H enthält einen Kreis, auf dem e liegt. Da $(V, E' \cup \{e\})$ azyklisch ist, enthält dieser Kreis auch eine Kante aus $E_1 \setminus (E' \cup \{e\})$.
Sei eine solche Kante e_1.

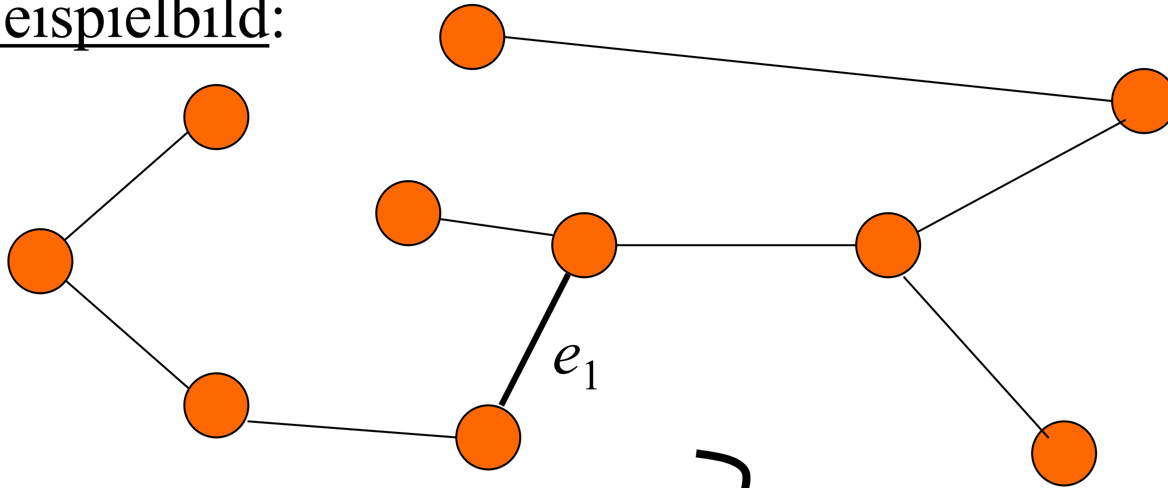
Betrachte $T_2 = (V, (E_1 \setminus \{e_1\}) \cup \{e\})$.
T_2 ist aufspannend und $c(T_2) = c(T_1) + c(e) - c(e_1)$.
Da e eine billigste Kante war, gilt $c(e) \leq c(e_1)$.
Also $c(T_2) \leq c(T_1)$.

Da nach Voraussetzung T_1 ein MST ist, gilt $c(T_2) = c(T_1)$.
D.h. T_2 auch ein MST, also $E' \cup \{e\}$ gut.

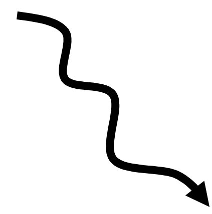
Mit der Behauptung folgt die Korrektheit von Kruskals Algorithmus unmittelbar.

$\square$

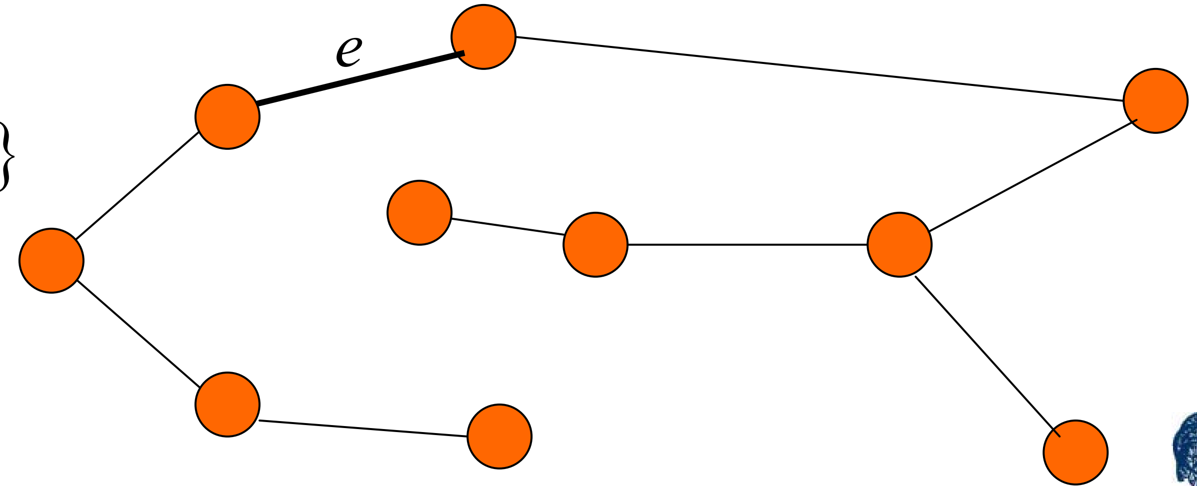
Beispielbild:



$T_1 : E_1$



$T_2 : (E_1 \setminus \{e_1\}) \cup \{e\}$



Zur Implementierung:

Wir halten uns eine Partition $V_1 \cup V_2 \cup \ldots \cup V_k = V$ von $V: \forall i \neq j: V_i \cap V_j = \emptyset$ und $\forall i: V_i \neq \emptyset$.

Operationen:

Starten mit Partition $\{\{1\}, \{2\}, \ldots, \{n\}\}$.

Benötigen:

- Find(x): Gibt Namen der Menge zurück, die x enthält.
- Union(a,b): Vereinigt die Mengen a und b.

Damit wird die for-Schleife in Kruskals Algorithmus zu:

```
for i ← 1 to m  do
        sei e_i = {u, v};
        a ← Find(u);
        b ← Find(v);
        if a ≠ b
           then E_T ← E_T ∪ {e_i};
                Union(a,b)
        fi
od
```

Bemerkung:

Auf diese Weise lassen sich i.w. auch die Zusammenhangs-komponenten eines Graphen berechnen.

Zur Realisierung der Union-Find-Datenstruktur:

Erste naheliegende Idee:

Realisierung als Feld $R[1 .. n]$, wobei $R[x]$ Name der Menge sein soll, die x enthält.

Damit:

Find(x): <u>return</u> $R[x]$

Union(a,b): <u>for</u> $i \leftarrow 1$ <u>to</u> n <u>do</u>

<u>if</u> $R[i] = a$ <u>then</u> $R[i] \leftarrow b$ <u>fi</u>

<u>od</u>

Laufzeit: Find(x): $O(1)$

Union(a,b): $O(n)$

Algorithmus von Kruskal:

$2m$ Find-Operationen und n-1 Union-Operationen.

⟶ Laufzeit Kruskal: $O(m+n^2+m \log m)$

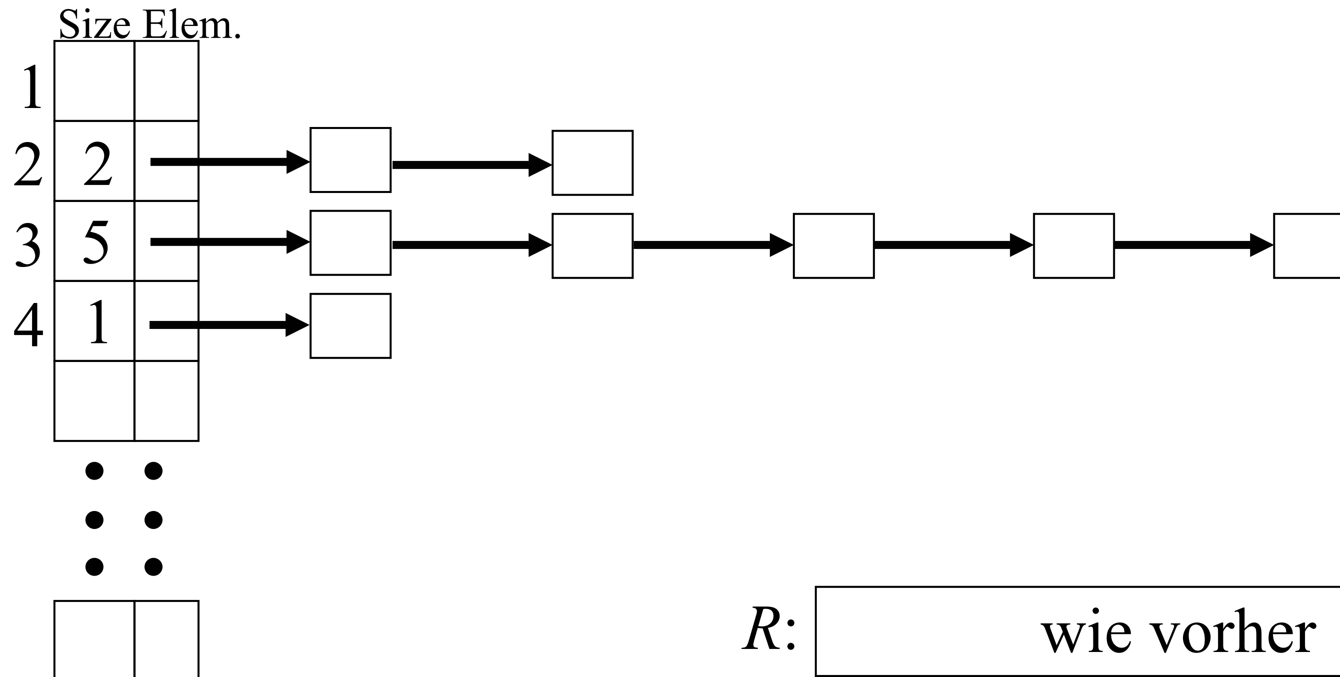
Heuristische Ideen zur Verbesserung:

- Union sollte nicht alle Elemente anschauen...
  ⟶ Verwalte Elemente jeder Menge separat...

- Behalte bei Union immer die Namen der größeren Mengen.

Also zusätzlich Verwaltung der Mengen als Listen (samt Größenangabe):

Bild:



Damit folgende Implementierung:

```
Initialis:      for x ← 1 to n do
                        R[x] ← x;
                        Elem[x] ← {x};
                        size[x] ← 1;
                od
Find(x):        return R[x]
Union(a,b):     if size[a] < size[b] then „vertausche a und b“ fi;
                for all x ∈ Elem[b] do
                        R[x] ← a;
                        insert(x, Elem[a])
                od;
                size[a] ← size[a] + size[b]
```

Beachte: Im Worst Case kann *ein* Union aber dennoch Zeit $O(n)$ benötigen.

Satz:

> Mit obiger Implementierung lassen sich die Initialisierungen sowie n-1 Union-Operationen und m Find-Operationen in Laufzeit $O(n \cdot \log n + m)$ realisieren.

Beweis:

Initialisierungen: $O(n)$, klar!

m Find-Operationen: $O(m)$, klar!

n Union-Operationen kosten Zeit $O(n \log n)$:

Union(a,b): Vereinige zwei Mengen mit n_a bzw. n_b Elementen; o.E. sei $n_b \leq n_a \Rightarrow$ Laufzeit $O(1 + n_b)$.

Sei n_i für i-tes Union die Größe der kleineren Menge. Damit Zeit für alle Union-Operationen: $O\left(\sum_{i=1}^{n-1}(n_i + 1)\right) = O\left(n + \sum_{i=1}^{n-1} n_i\right)$.

Immer wenn ein Element die Menge wechselt, trägt es genau 1 zu $\sum_{i=1}^{n-1} n_i$ bei.

Also $\sum_{i=1}^{n-1} n_i = \sum_{j=1}^{n} r_j$, wobei r_j angibt, wie oft Element j die Menge wechselt.

Behauptung: $r_j \leq \log_2 n$ für alle $j = 1, 2, \ldots, n$.

Beweis der Behauptung:

Wenn j die Menge wechselt und vorher in einer Menge mit l Elementen war, so ist j nachher in einer Menge mit mindestens $2l$ Elementen.

D.h. nach dem k-ten Wechsel ist j in einer Menge mit $\geq 2^k$ Elementen.

Da es nur n Elemente gibt, folgt $k \leq \log_2 n$.

Mit Hilfe der Behauptung folgt somit, dass n Union-Operationen Zeit $O(n \log n)$ kosten.

□

Korollar:

Der Algorithmus von Kruskal hat Laufzeit $O(m \cdot \log m)$.

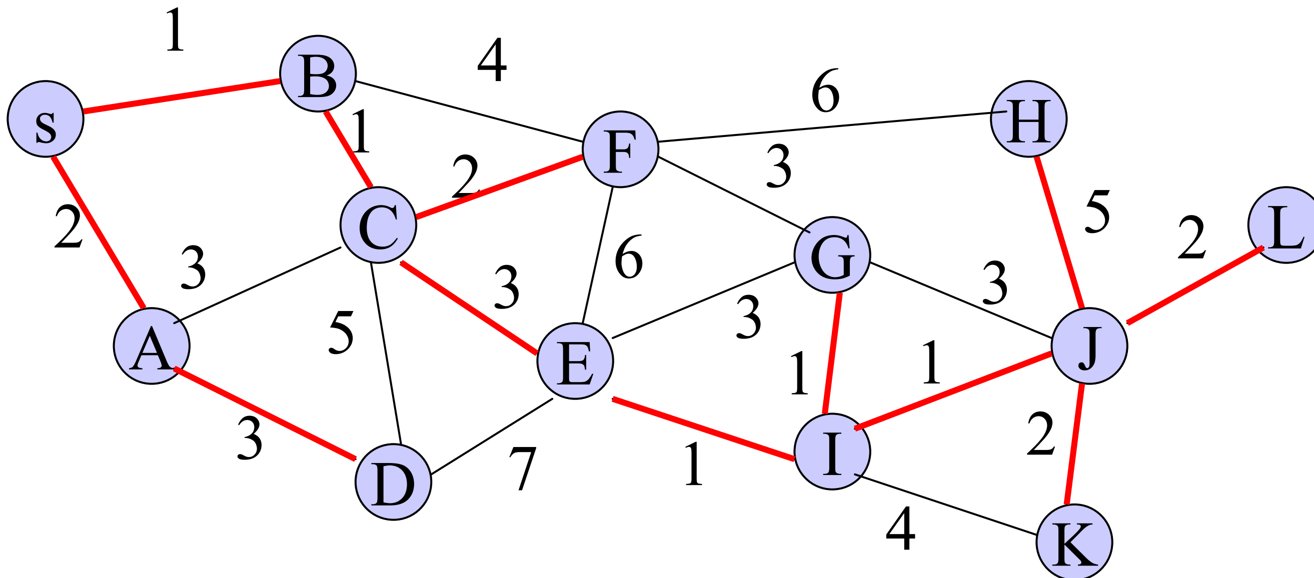
Bemerkung:

Obiges ist ein Beispiel für **amortisierte Analyse**:
Wir betrachten statt Einzeloperationen, welche verschieden teuer sind, eine Folge von Operationen und stellen fest, wieviel diese *zusammen* kosten.

Anderes Beispiel für amortisierte Analyse:

Binärzähler: Nicht trivial $n \log n$, sondern $2n$ amortisiert...

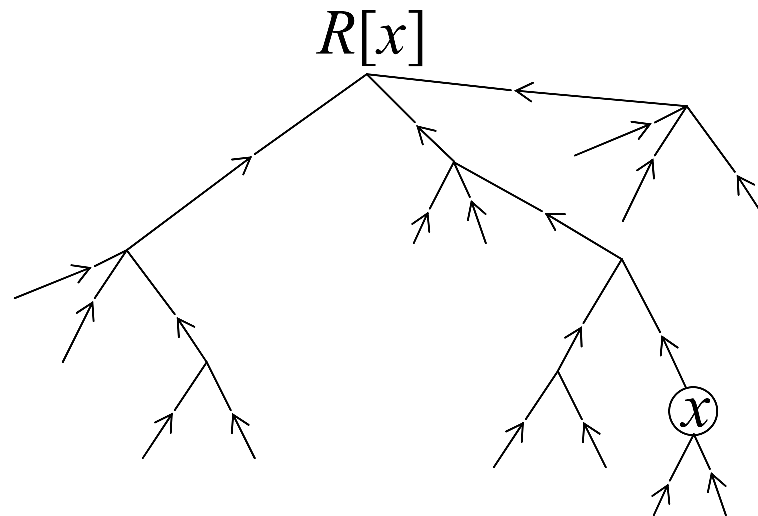
Beispiel (Kruskal, Ungerichteter Graph, nichtnegative Kanten):



Alternative zur vorigen Implementierung:

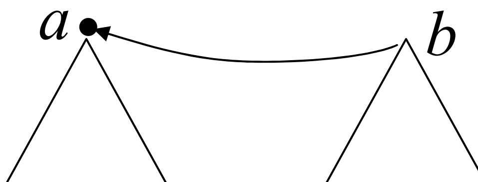
Idee: Repräsentiere jede Menge durch einen Baum (i.a. nicht binär):

Jeder Knoten (außer Wurzel) zeigt auf Elternknoten, Wurzel enthält Namen der Menge.



Damit:

Union(a,b): „Zusammenhängen“ von Bäumen.



Zeit: $O(1)$

Find(x): Folge dem Pfad von x zur Wurzel.

Zeit: $O(1+\text{Tiefe}(x))$.

Ideal wäre also Baumtiefe 1...

Im Worst Case können n Union-Op. aber zu Tiefe n-1 führen.

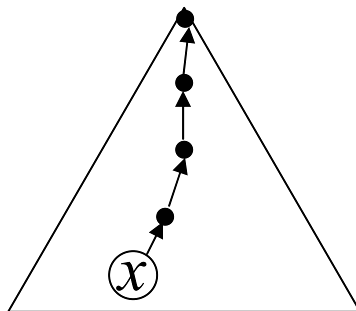
Deshalb: „**Gewichtete Vereinigungsregel**“:

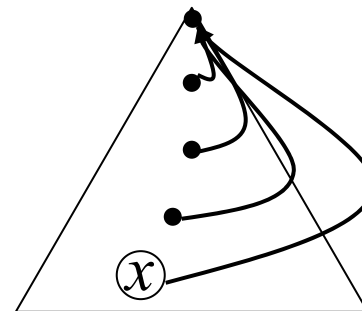
Hänge immer den kleineren an den größeren Baum (Baumgröße = Anzahl der Knoten).

Weitere Idee: **Pfadkomprimierung**

Wenn bei Find der Pfad zur Wurzel zurückverfolgt wird, so hänge alle Zeiger des Pfades zur Wurzel hin um:

Bild:





Mitteilung:

Mit gewichteter Verzweigungsregel und Pfadkomprimierung lassen sich die Initialisierungen sowie n-1 Union-Operationen und m Find-Operationen in Laufzeit $O(n + m \cdot \alpha(m + n, n))$ realisieren, wobei α die Inverse der Ackermannfunktion ist.

α wächst extrem langsam und ist für alle realistischen Werte ≤ 5.

Der Beweis obiger Mitteilung ist ziemlich kompliziert.