AVL-Bäume

Binäre Suchbäume optimal im Average-Case ($h = \Theta(\log n)$), aber schlecht im Worst-Case ($h = \Theta(n)$)

Definition

Ein Baum heißt balanciert, wenn für jeden Knoten die Höhen von dessen linkem und rechtem Teilbaum sich um höchstens 1 unterscheiden.

Satz

Ein balancierter Baum mit n Knoten hat Höhe $\Theta(\log n)$.

Beweis: Der vollständige Binärbaum der Höhe h ist der balancierte Binärbaum der Höhe h mit maximaler Knotenanzahl n. In dem Fall ist $n = 2^h - 1$.
Daher gilt

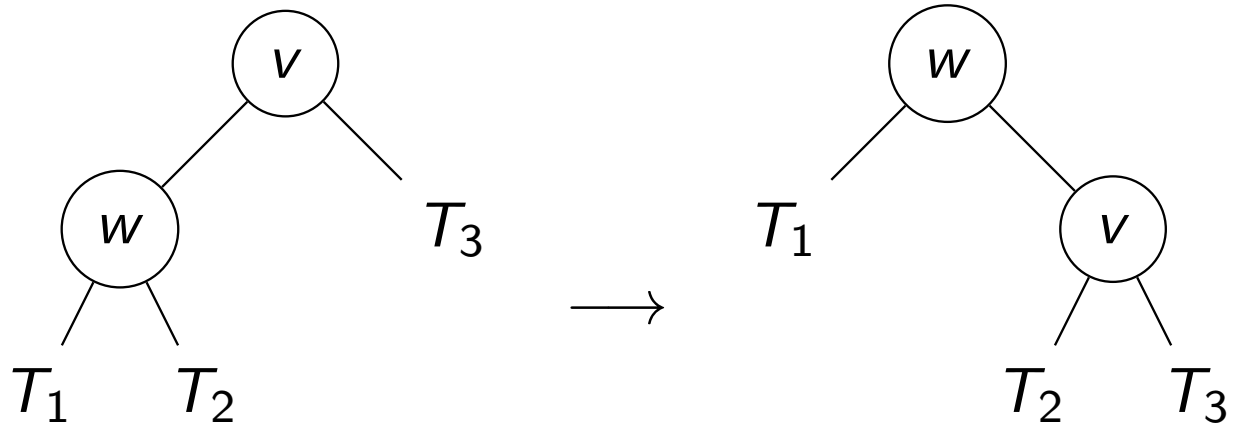
$$n \leq 2^h - 1 \text{ und somit } h = \Omega(\log n).$$

$n_h \ldots$ Mindestanzahl an Knoten bei Höhe h.
Behauptung: $n_h > F_h$
Induktion nach h: $h = 0$, $n_0 = 1 > 0 = F_0$, $h = 1$, $n_1 > 1 = F_1$ ✓
Die Teilbäume der Wurzel haben die Höhen $h-1$ und $h-2$ und n_{h-1} bzw n_{h-2} Knoten. Dann folgt

$$n_h = n_{h-1} + n_{h-2} + 1 > F_{h-1} + F_{h-2} + 1 > F_h.$$

Daher gilt $n \geq n_h > F_h$ und somit $h = \mathcal{O}(\log n)$. □

Balanciertheit muss beim Einfügen und Löschen erhalten bleiben!
Umsetzung mit Hilfe von lokalen Korrekturtransformationen.
Rechtsrotation:



Linksrotation analog. Bem.: Suchbaumeigenschaft bleibt erhalten.

Definition

Die Balanciertheit eines Baumes T, dessen Wurzel T_L als linken Teilbaum und T_R als rechten Teilbaum besitzt, ist definiert als $b(T) = h(T_R) - h(T_L)$.

Erhalt der Balanciertheit heißt im Detail:

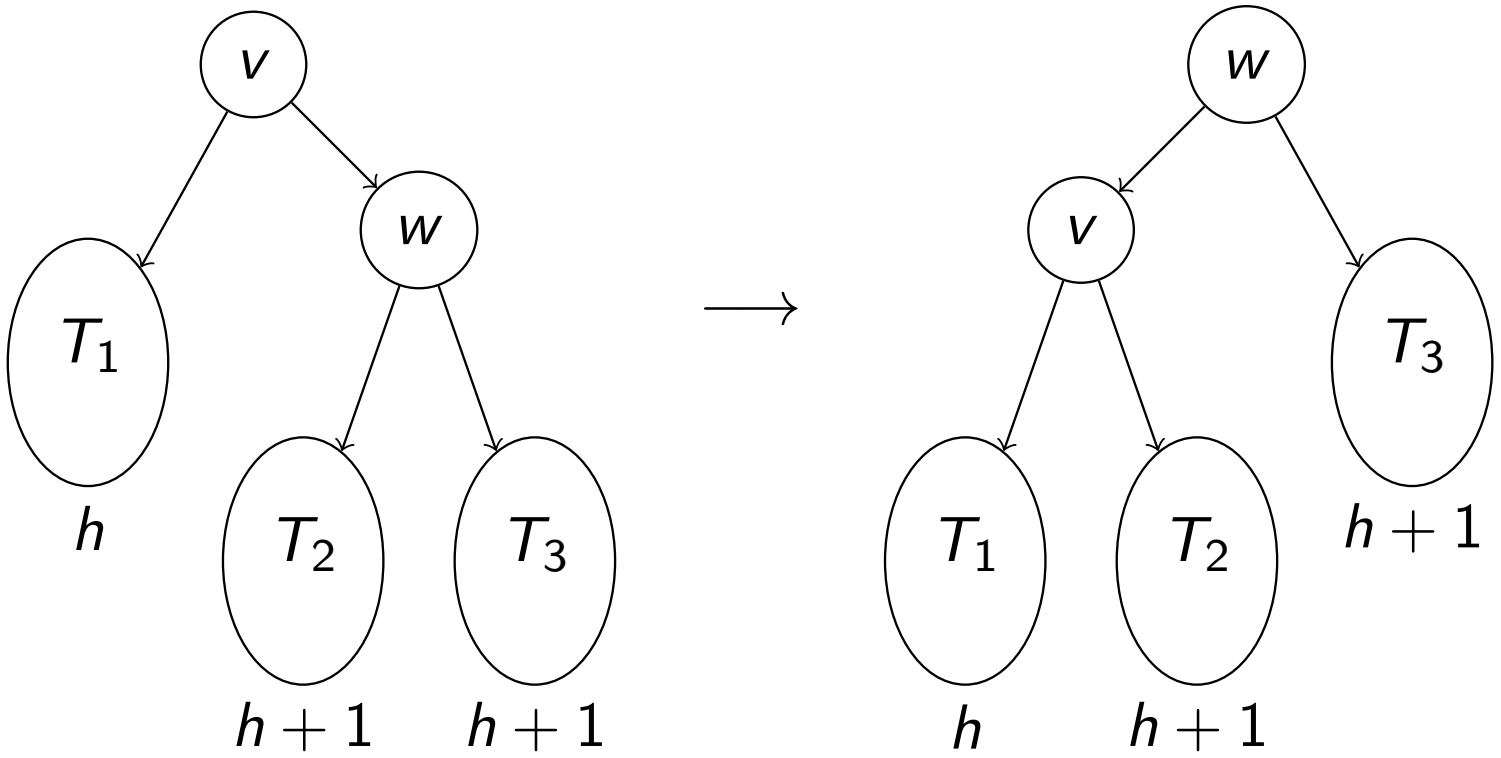
- Suche erfolgt wie im binären Suchbaum.
- Bei jedem Knoten wird die Balanciertheit dynamisch mitgespeichert.
- Nach dem Einfügen oder Löschen wird die Prozedur BALANCE ausgeführt, die im Fall $|b(T)| \geq 2$ die notwendigen Reparaturen durchführt und $|b(T)| \leq 1$ wiederherstellt.

BALANCE

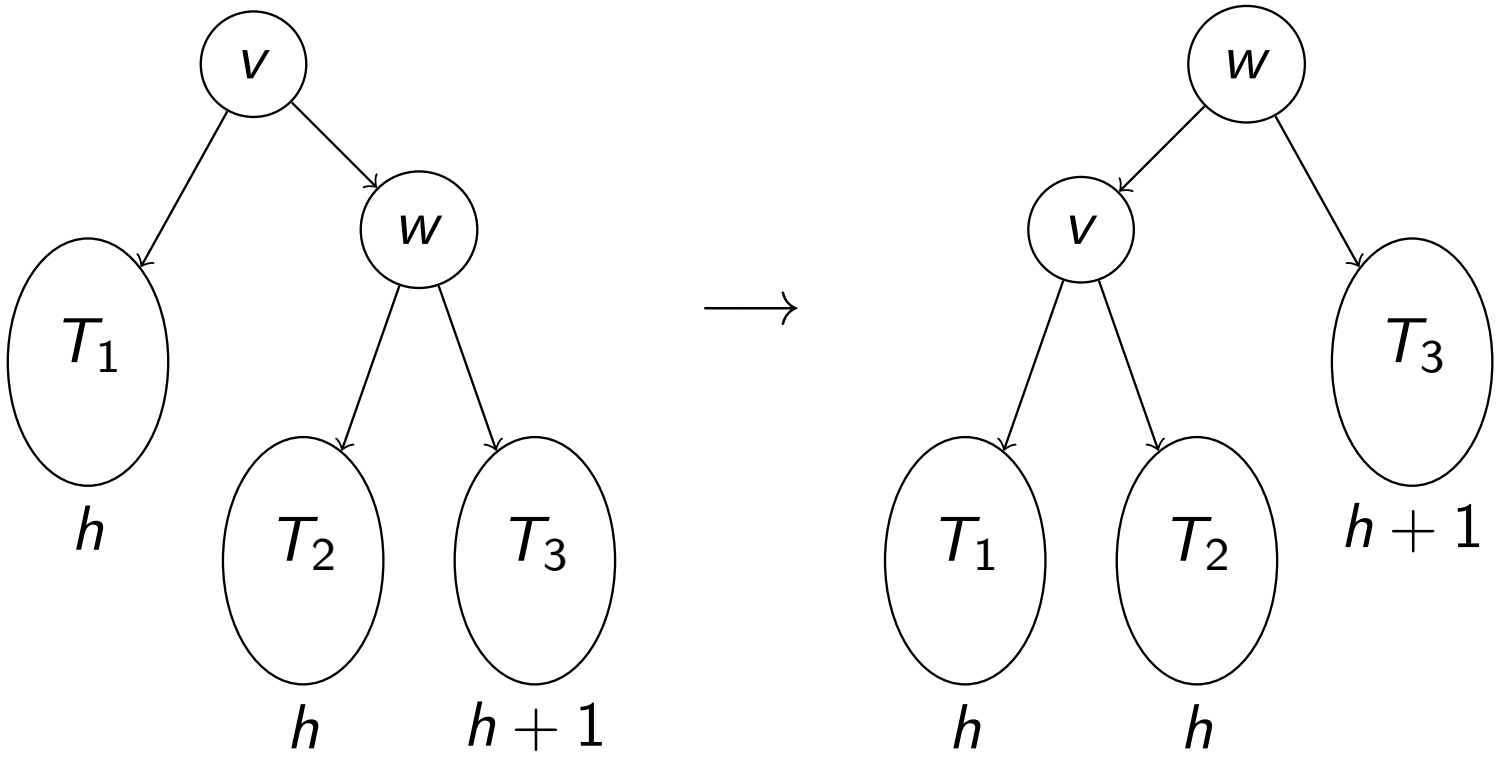
Übergeben wird ein Baum T mit Wurzel v.

1. Falls $|b(T)| \leq 1$, gib T zurück.
2. Falls $b(T) = 2$, dann ist $\mathrm{R}(v)$ nicht leer; Wurzel von $\mathrm{R}(v)$ sei w.

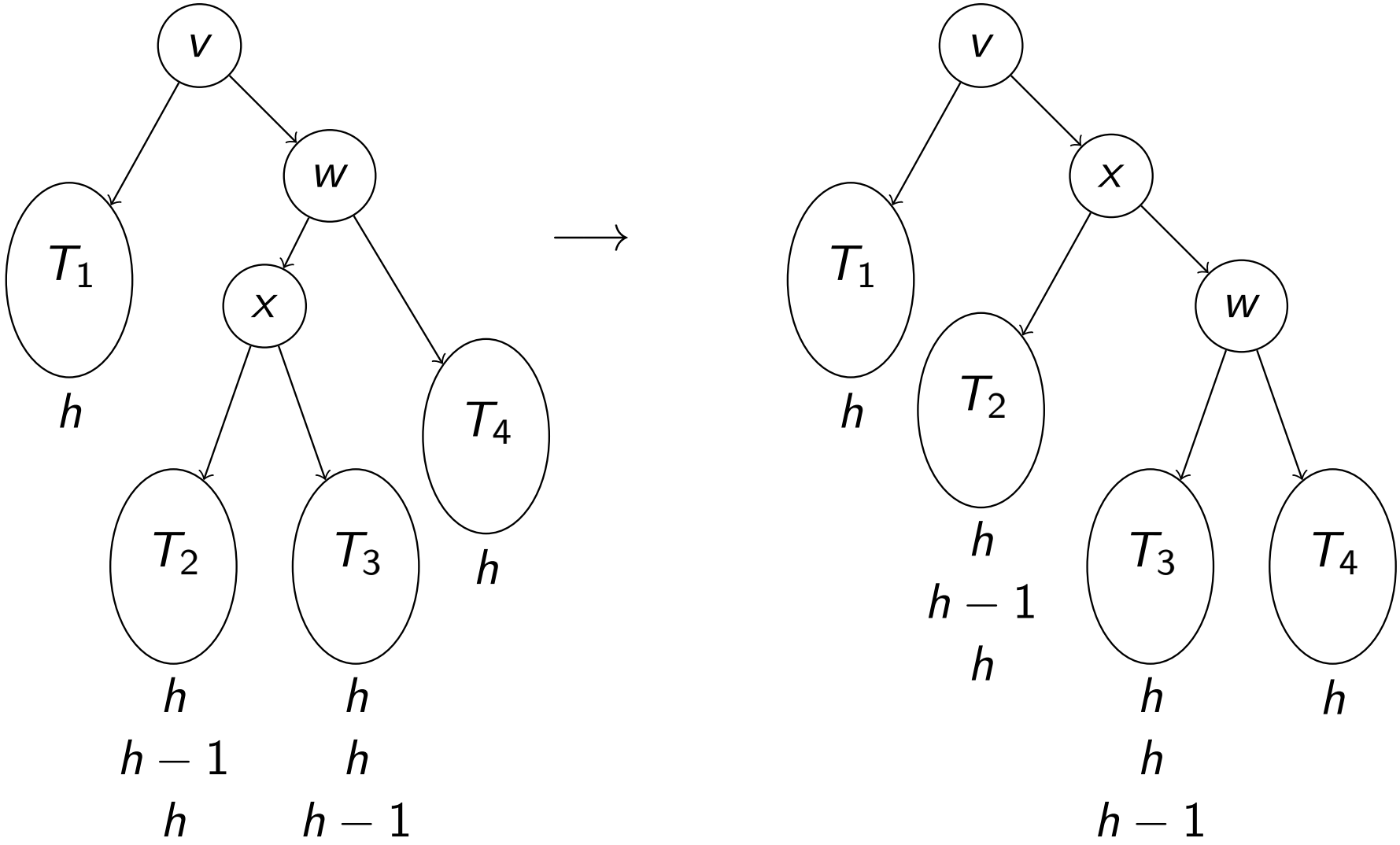
A Falls $b(w) = 0$, dann Linksrotation von T.



- **B** Falls $b(w) = 1$, dann Linksrotation von T.

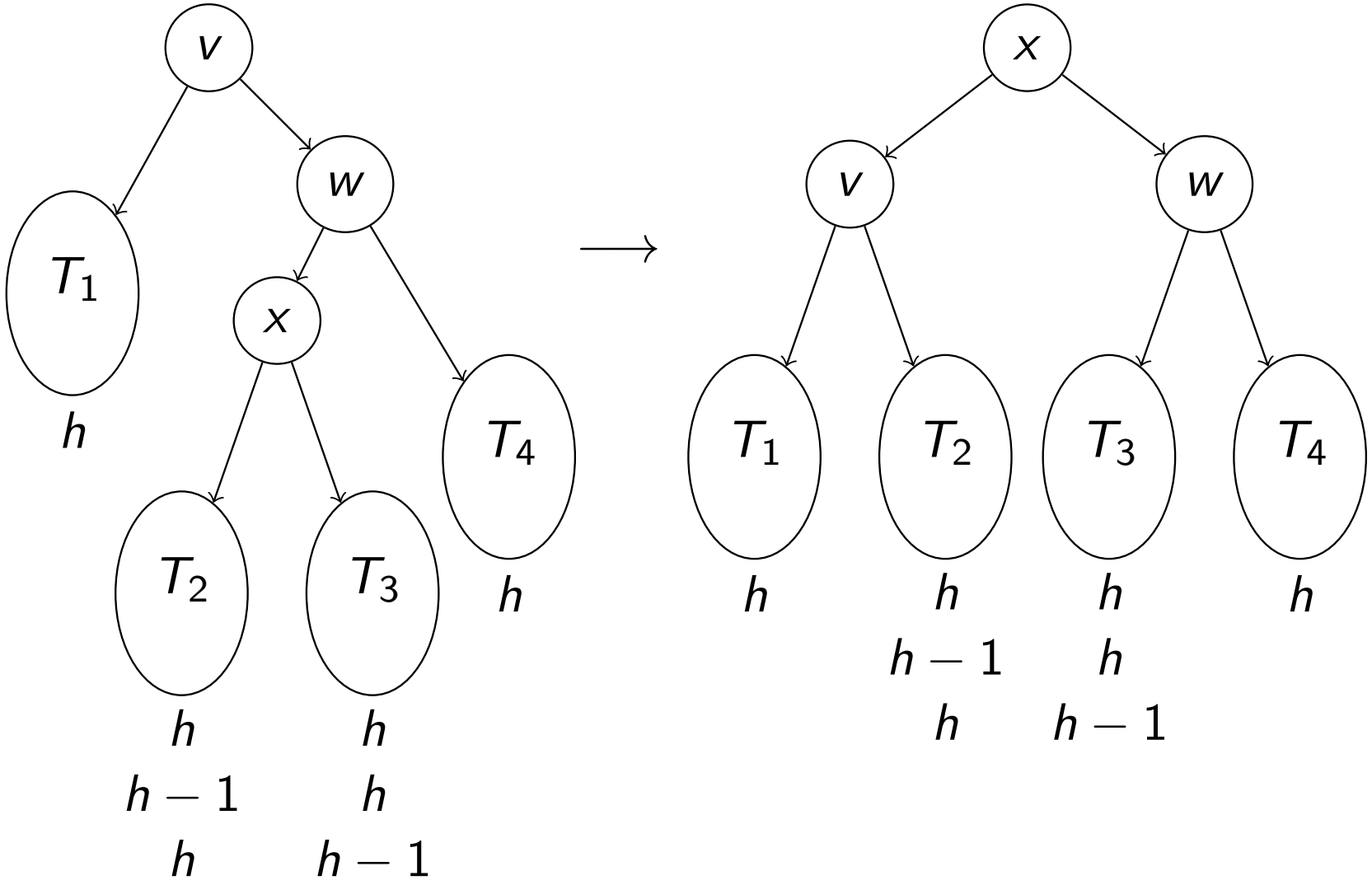


- **C** Falls $b(w) = -1$, dann ist $\mathrm{L}(w)$ nicht leer. Rechtsrotation auf $\mathrm{R}(v)$ und anschließend Linksrotation auf T.



3. Falls $b(v) = -2$, dann verfährt man analog zum Fall $b(v) = 2$.

- **c** Falls $b(w) = -1$, dann ist $\mathrm{L}(w)$ nicht leer. Rechtsrotation auf $\mathrm{R}(v)$ und anschließend Linksrotation auf T.



3. Falls $b(v) = -2$, dann verfährt man analog zum Fall $b(v) = 2$.

Laufzeit:

- Da die Balanciertheit in den Knoten gespeichert wird, benötigt BALANCE nur $\Theta(1)$ Schritte.
- Die Prozeduren EINFÜGEN und LÖSCHEN werden so modifiziert, dass statt eines Baumes T nun ein Paar $(T, \Delta h)$ ausgegeben wird, wobei Δh die Höhendifferenz zwischen altem und neuem Baum ist.
- Auch die Prozedur BALANCE wird so gestaltet, dass sie Paare $(T, \Delta h)$ ausgibt.
- Wenn EINFÜGEN die Ausgabe $(T, \Delta h)$ liefert, dann gilt

$$b(T_{\text{neu}}) = b(T_{\text{alt}}) + \Delta h_L - \Delta h_R.$$

  Danach wird BALANCE aufgerufen und liefert $(T', \Delta h')$ und analog kann dann $b(T')$ in der Zeit $\Theta(1)$ berechnet werden.
- Insgesamt ergibt sich logarithmische Laufzeit, auch im Worst-Case.

Direkte Adressierung in einer Tabelle

- Aufgabe: Speicherung einer dynamischen Menge unter Bereitstellung der grundlegenden "Wörterbuchfunktionen" Einfügen, Suchen und Löschen.
- Annahme: Menge möglicher Schlüssel sei das Universum $U = \{0, 1, \ldots, m-1\}$, wobei m relativ klein sei.

Tabelle zur direkten Adressierung durch Datenfeld $A[0..m-1]$ realisierbar; $A[k]$ ist Zeiger zum Element x mit Schlüssel $S(x) = k$.
Wörterbuchfunktionen sind einfach:

Algorithm D-A-EINFÜGEN(A, x)

1: $A[S(x)] := x$

Algorithm D-A-SUCHEN(A, k)

1: **return** $A[k]$

Algorithm D-A-LÖSCHEN(A, x)

1: $A[S(x)] :=$NIL

Alle Operation brauchen nur eine Zeit von $\mathcal{O}(1)$.

Hashtabellen

Nachteile der direkten Adressierung:

- schlecht oder sogar unmöglich, wenn Universum U riesig;
- Speicherverschwendung, wenn Anzahl der gespeicherten Schlüssel klein im Vergleich zu U.

Hashtabellen brauchen (deutlich) weniger Speicher als Tabellen zur direkten Adressierung.

- Direkte Adressierung: Element mit Schlüssel k wird in Position k gespeichert.
- Hashtabelle: Element mit Schlüssel k wird in Position $h(k)$ gespeichert; $h : U \to \{0, 1, \ldots, m-1\}$ heißt Hashfunktion.
- Meistens gilt $m \ll |U|$.
- Achtung: Hashfunktionen sind nicht injektiv!
  Es braucht also eine effektive Kollisionsbehandlung!

Kollisionsbehandlung durch Verkettung

Falls zwei Schlüssel k und ℓ den gleichen Hashwert $r = h(k) = h(\ell)$ haben, so werden die entsprechenden Elemente am gleichen Platz als verkettete Liste gespeichert.

Wörterbuchfunktionen:

Algorithm H-EINFÜGEN(A, x)

1: LISTE-EINFÜGEN$(A[h(S(x))], x)$

Algorithm H-SUCHEN(A, k)

1: SUCHE$(A[h(k)], k)$

Algorithm H-LÖSCHEN(A, x)

1: LISTE-LÖSCHEN$(A[h(S(x))], x)$

Laufzeiten:
H-EINFÜGEN: $\mathcal{O}(1)$,
H-SUCHEN: proportional zur Listenlänge,
H-LÖSCHEN: $\mathcal{O}(1)$, falls Listen doppelt verkettet.

Analyse

n Schlüssel, m Positionen.

Performance abhängig vom Belegungsfaktor $\alpha = n/m$.

- Worst-Case: alle Schlüssel haben gleichen Hashwert
  $\Longrightarrow$ Listenlänge n.
- Gute Hashfunktionen benötigt!
- Ideal: gleichmäßiges Hashing, d.h. Schlüsselverteilung so, dass jede der m Positionen gleich wahrscheinlich ist.

Satz (Erfolglose Suche)

Die Suche eines in einer Hashtabelle mit Verkettung nicht vorhandenen Schlüssels mittels gleichmäßigem Hashing dauert $\Theta(1+\alpha)$.

Beweis: Mittlere Suchzeit ist die mittlere Zeit zum Durchlaufen der Liste $A[h(k)]$.

Mittlere Listenlänge: α. Also Suchzeit $\Theta(\alpha)$.

Rechenzeit zur Berechnung von $h(k)$: $\Theta(1)$. □

Satz (Erfolgreiche Suche)

Die Suche eines in einer Hashtabelle mit Verkettung zufällig gewählten vorhandenen Schlüssels mittels gleichmäßigem Hashing dauert $\Theta(1+\alpha)$.

Beweis: Um x mit $S(x) = k$ zu finden, müssen die Elemente vor x in der Liste $A[h(k)]$ und x selbst untersucht werden.
Sei x_i das $i$te in die Tabelle eingefügte Element und $S(x_i) = k_i$.
Weiters sei $X_{i,j} = \mathbb{1}_{[h(k_i)=h(k_j)]}$.
Gleichmäßiges Hashing impliziert $\mathbb{P}\{h(k_i) = h(k_j)\} = \mathbb{E}X_{i,j} = \frac{1}{m}$.
Die mittlere Anzahl der untersuchten Elemente ist daher

$$\mathbb{E}\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{i,j}\right)\right] = \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right)$$

$$= 1+\frac{1}{nm}\sum_{i=1}^{n}(n-i) = 1+\frac{1}{nm}\left(n^2-\binom{n+1}{2}\right) = 1+\frac{\alpha}{2}-\frac{\alpha}{2n} \qquad \square$$

Gute Hashfunktionen

Eine gute Hashfunktion erfüllt möglichst gut die Annahme des gleichmäßigen Hashings, also jeder Hashwert tritt mit gleicher Wahrscheinlichkeit auf.

Beispiel: Schlüssel seien $U[0,1]$-Zufallsvariable, dann liefert $h(k) := \lfloor km \rfloor$ eine solche Hashfunktion.
Andere Möglichkeiten: Schlüssel aus $\mathbb{N}$

- Divisionsmethode: $h(k) = k \mod m$. Berechnung schnell. Wichtig ist gute Wahl des Moduls. ($m = 2^\ell$ schlecht!)
- Multiplikationsmethode: $h(k) = \lfloor m(kA \mod 1) \rfloor$, wobei $0 < A < 1$ eine a priori festgelegte Konstante ist. Schnelle Berechnung möglich, wenn $m = 2^\ell$ und $A = r/2^s$, $0 < r < 2^s$.
- Universelles Hashing: Hashfunktion zufällig aus einer sorgfältig gestalteten Funktionenklasse $\mathcal{F}$ wählen.
  $\mathcal{F}$ heißt universell, wenn für alle $k, \ell \in U$ höchstens $|\mathcal{F}|/m$ Funktionen $h \in \mathcal{F}$ zu einer Kollision führen, d.h. $h(k) = h(\ell)$.

Beispiel für universelles Hashing:
$U = \mathbb{Z}_p$, wobei p eine Primzahl mit $m \ll p$,

$$h_{a,b}(k) = ((ak + b) \mod p) \mod m.$$

Satz

Die Funktionenklasse $\mathcal{F} = \{h_{a,b} \mid 1 \leq a \leq p-1, 0 \leq b \leq p-1\}$ ist universell.

Beweis: Sei für $0 \leq k \leq p-1$, $0 \leq \ell \leq p-1$ $(k \neq \ell)$

$$r = (ak + b) \mod p \quad \text{und} \quad s = (a\ell + b) \mod p.$$

Aus $k \neq \ell$ folgt $r \neq s$. Außerdem ist $(a, b) \mapsto (r, s)$ injektiv. Wegen $h_{a,b}(k) = r \mod m$ folgt daraus (bei zufälliger Wahl von $h_{a,b}$)

$$\mathbb{P}\{h_{a,b}(k) = h_{a,b}(\ell)\} = \mathbb{P}\{r \equiv s \mod m\}$$

bei zufälliger Wahl eines der $p(p-1)$ Paare (r, s).
Aber, bei gegebenem r gibt es höchstens $\lfloor \frac{p-1}{m} \rfloor$ von r verschiedene s mit $r \equiv s \mod m$. Also gilt $\mathbb{P}\{r \equiv s \mod m\} \leq \frac{1}{m}$. □

Offenes Adressieren

- Alle Elemente in der Hashtabelle, nicht in externen Listen. ($\Longrightarrow \alpha < 1$)
- Im Falle einer Kollision: Sondierung der Tabelle, um freie Plätze zu finden.
- Lineares Sondieren: Sondierungsfolge

$$h(k,i) = (h(k) + i) \mod m, \quad i = 0, 1, \ldots, m-1.$$

  Nachteil: Tendenz zur Clusterbildung!
- Verbesserung: Double Hashing. Zwei Hashfunktionen, Sondierungsfolge

$$h(k,i) = (h_1(k) + ih_2(k)) \mod m, \quad i = 0, 1, \ldots, m-1.$$

  Voraussetzung: $\mathrm{ggT}(h_2(k), m) = 1$, da sonst die Hashtabelle nur teilweise sondiert wird.
  Z.B.: Für m Primzahl,

$$h_1(k) = k \mod m, \quad h_2(k) = k \mod m-1$$

  erreicht man bessere Performance als beim linearen Sondieren.

Satz (Gleichmäßiges Hashing mit Sondierung)

Beim offenen Adressierung und Belegungsfaktor

$$\alpha = \frac{n}{m} < 1$$

ist die mittlere Anzahl an Sondierungen

$$\frac{1}{1-\alpha}$$

bei erfolgloser Suche und höchstens

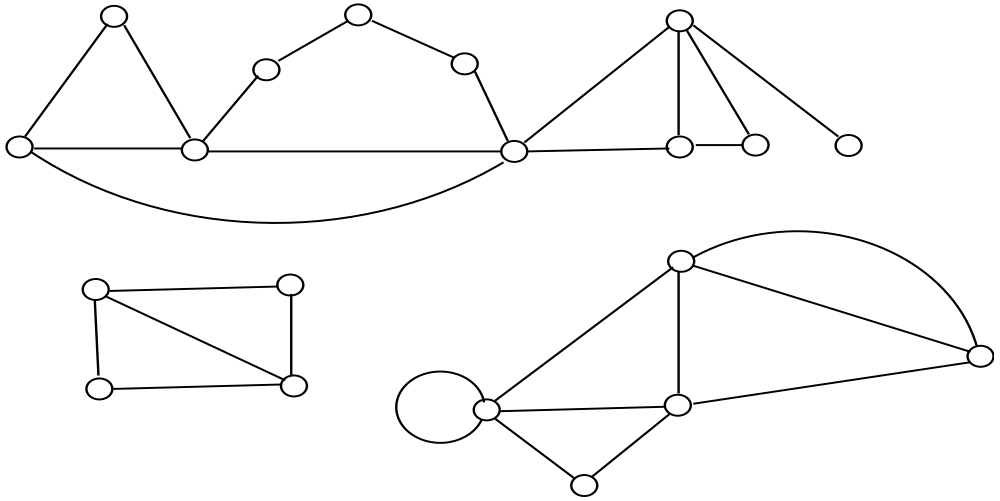
$$\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$$

bei erfolgreicher Suche.

10) Grundlegende Graphenalgorithmen

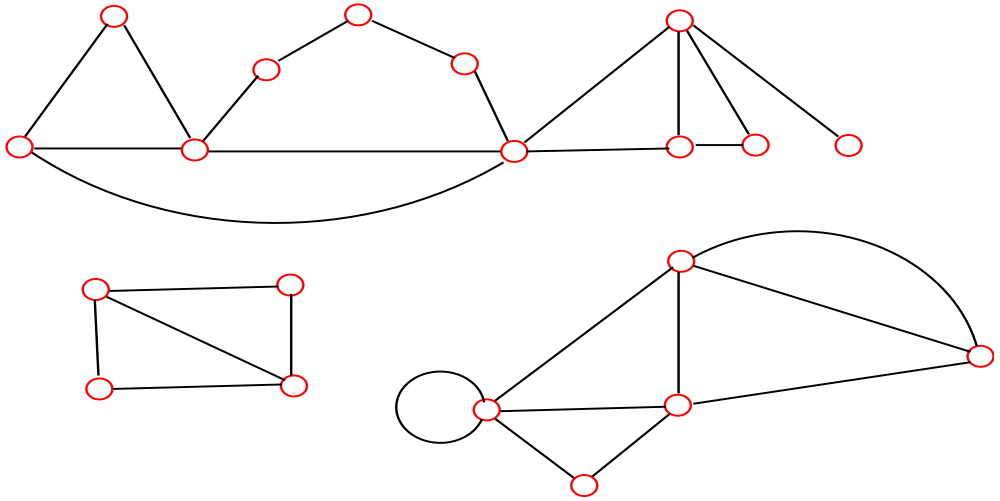
Darstellung von Graphen

Ungerichteter Graph



Darstellung von Graphen

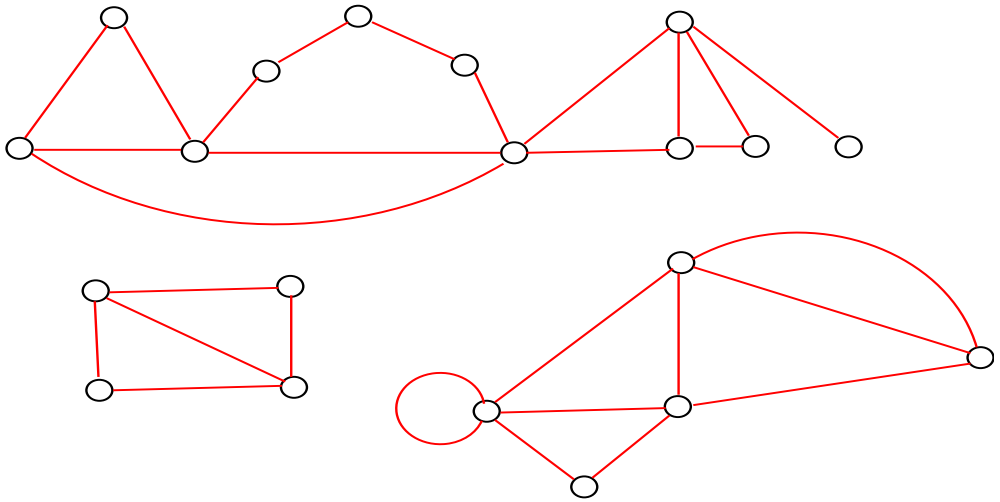
Die Knoten des Graphen



Knotenmenge V, $\alpha_0 := |V|$

Darstellung von Graphen

Die Kanten des Graphen



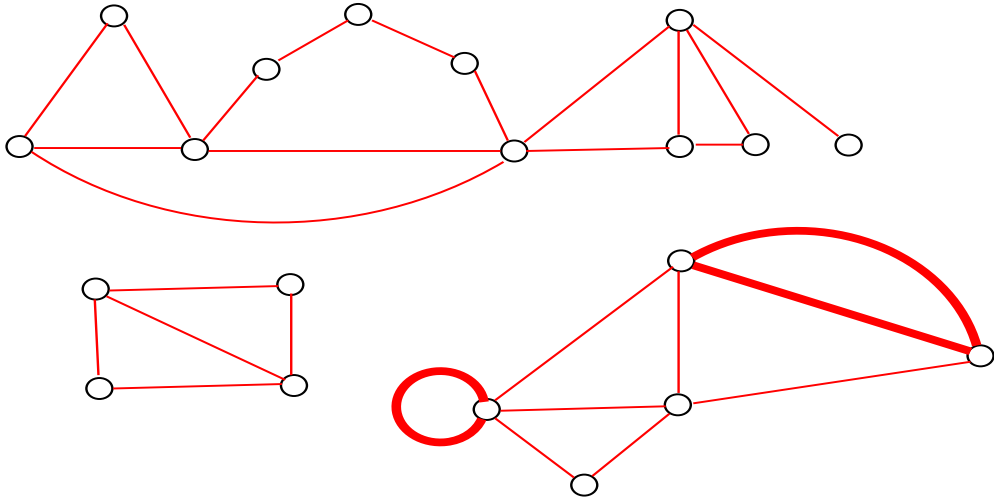
Kantenmenge E, $\alpha_1 := |E|$, $\longrightarrow$ Graph $G = (V, E)$

Dichte $\varepsilon(G) = \frac{|E|}{|V|}$

dünn: $\varepsilon(G) = o(1)$, dicht: $\varepsilon(G) = \Omega(1)$

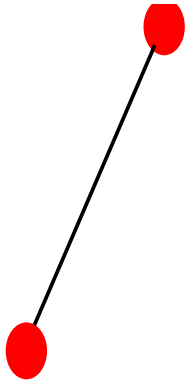
Darstellung von Graphen

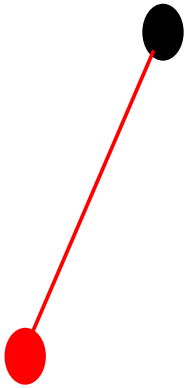
Spezielle Kanten: Schlingen und Mehrfachkanten



Graphen ohne Schlingen und Mehrfachkanten: **schlichte** Graphen

Adjazenz und Inzidenz



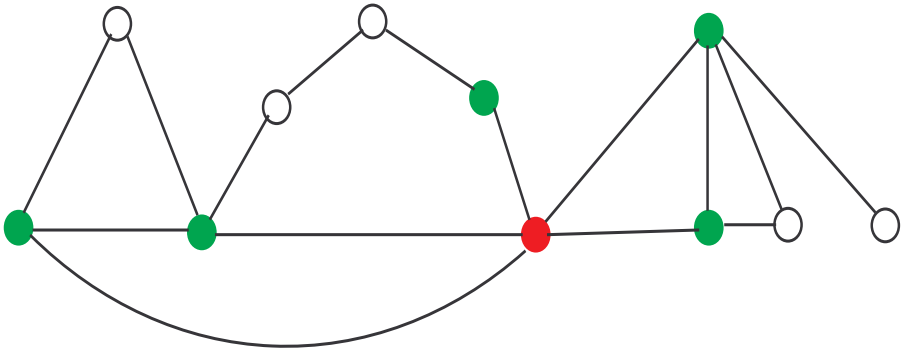


adjazente Knoten

$v \sim w$

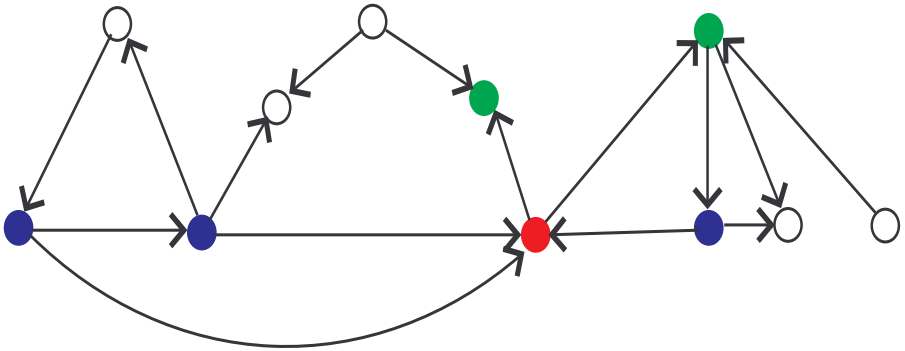
mit Knoten inzidierende Kante

Ein Knoten v und die Menge seiner Nachbarn $\Gamma(v)$
$d(v) = d_G(v) = |\Gamma(v)| =$ der Grad von v



$\delta(G) = \min_{v \in V} d(v), \qquad \Delta(G) = \max_{v \in V} d(v)$

Gerichteter Fall: Nachfolger $\Gamma^+(v)$ und Vorgänger $\Gamma^-(v)$
$d^+(v) = |\Gamma^+(v)|$ bzw. $d^-(v) = |\Gamma^-(v)|$: Weggrad bzw. Hingrad von v



Satz (Handschlaglemma)

$$\sum_{x \in V(G)} d(x) = 2|E(G)| \text{ bzw. } \sum_{x \in V(G)} d^+(x) = \sum_{x \in V(G)} d^-(x) = |E(G)|$$

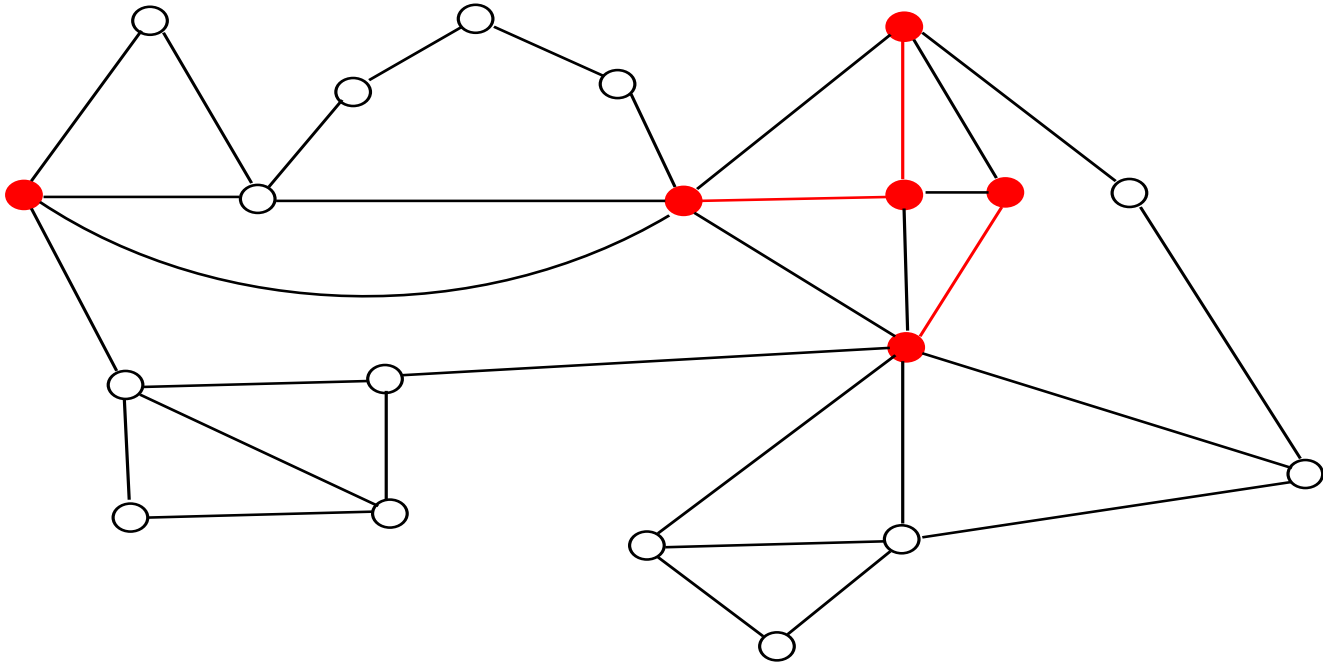
Definition

Ein Graph $G' = (V', E')$ heißt Teilgraph eines Graphen $G = (V, E)$, wenn

$$V' \subseteq V \text{ und } E' \subseteq E.$$

Beispiel

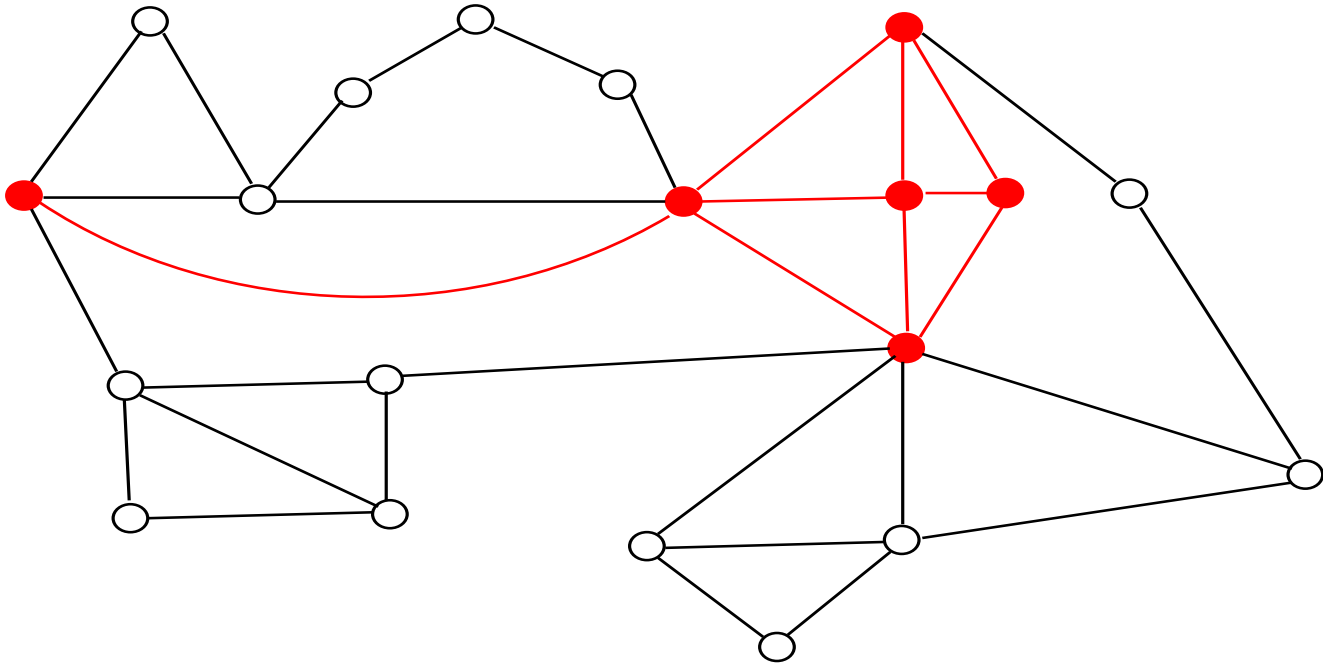
Ein Graph $G = (V, E)$ und einer seiner Teilgraphen $G' = (V', E')$:



Definition

Der Teilgraph $G[V_0] = (V_0, E_0)$ eines Graphen $G = (V, E)$ heißt von V_0 induzierter Teilgraph, wenn unter allen Teilgraphen $G' = (V_0, E')$ die Kantenmenge E_0 maximal bezüglich der Inklusion ist, d.h. für alle solchen G' gilt $E' \subseteq E_0$.

Beispiel



Die Adjazenzmatrix von $G = (V, E)$:

$V = \{v_1 \ldots, v_n\}$, $A = (a_{i,j})_{i,j=1,\ldots,n}$ mit

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

Beispiel

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

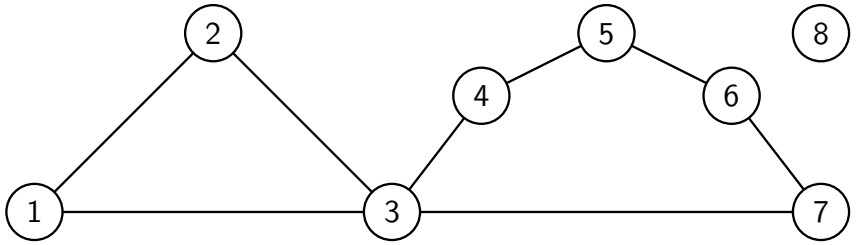
Es gilt: $d(v_i) = \sum_{j=1}^{n} a_{i,j} = \sum_{j=1}^{n} a_{j,i}$

Die Adjazenzmatrix von $G = (V, E)$:

$V = \{v_1 \ldots, v_n\}$, $A = (a_{i,j})_{i,j=1,\ldots,n}$ mit

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

Beispiel



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

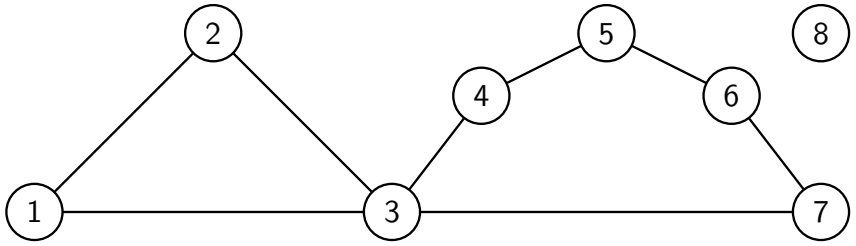
Es gilt: $d(v_i) = \sum_{j=1}^{n} a_{i,j} = \sum_{j=1}^{n} a_{j,i}$

Die Adjazenzmatrix von $G = (V, E)$:

$V = \{v_1 \ldots, v_n\}$, $A = (a_{i,j})_{i,j=1,\ldots,n}$ mit

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

Beispiel



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

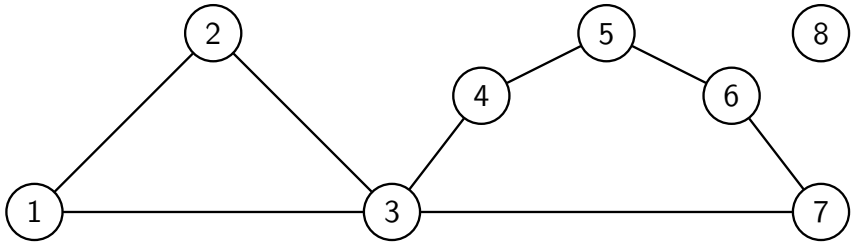
Es gilt: $d(v_i) = \sum_{j=1}^{n} a_{i,j} = \sum_{j=1}^{n} a_{j,i}$

Die Adjazenzmatrix von $G = (V, E)$:

$V = \{v_1 \ldots, v_n\}$, $A = (a_{i,j})_{i,j=1,\ldots,n}$ mit

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

Beispiel



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

Es gilt: $d(v_i) = \sum_{j=1}^{n} a_{i,j} = \sum_{j=1}^{n} a_{j,i}$

Adjazenzlisten: Feld aus $|V|$ Listen

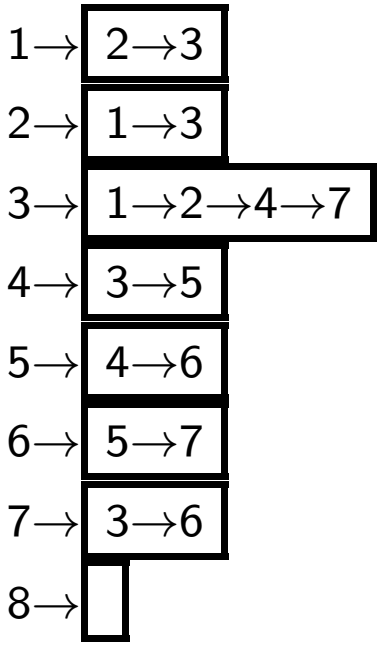
$u \in V$:

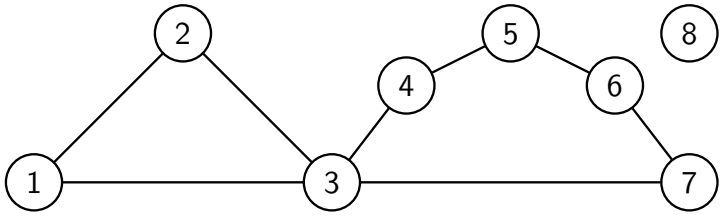
$\mathrm{Adj}[u] = [v_1, v_2, \ldots, v_k]$, falls $\{v_1, v_2, \ldots, v_k\} = \Gamma(u)$ (bzw. $\Gamma^+(u)$)

Es gilt

$$\sum_{u \in V} |\mathrm{Adj}[u]| = \sum_{u \in V} d(u) = 2|E| \text{ bzw. } \sum_{u \in V} |\mathrm{Adj}[u]| = \sum_{u \in V} d^+(u) = |E|.$$

Ad Beispiel:





Grundlegende Graphenalgorithmen

Gewichtete Graphen: $G = (V, E, w)$ mit $w : E \to \mathbb{R}$

- Adjazenzmatrix $A = (w(v_i v_j))_{i,j=1\ldots,|V|}$ bzw. $A = (w(v_i, v_j))_{i,j=1,\ldots,|V|}$
- entsprechend erweiterte Adjazenzlisten

Vor- und Nachteile:

- Adjazenzmatrix:
  - einfach zu implementieren;
  - Suche nach $uv \in E$ mittels Direktzugriff;
  - hoher Speicherplatzbedarf: $\Theta(|V|^2)$.
- Adjazenzlisten:
  - niedriger Speicherplatzbedarf, falls G dünn: $\Theta(|V| + |E|)$
  - Suche nach $uv \in E$ erfordert Suche in Adj$[u]$

Breitensuche (breadth-first-search, BFS)

Algorithmus zum Durchsuchen eines Graphen

Eingabe: Graph $G = (V, E)$, Startknoten $s \in V$
Verfahren:

- Knoten von G ausgehend von s erforschen.
- Distanz zu s berechnen, wobei

$$d(s, v) = \begin{cases} \min\limits_{W(s \to v)} \#\text{ Kanten von } W, & \text{falls es solche Wege gibt,} \\ \infty & \text{sonst.} \end{cases}$$

- Erzeugen eines Breitensuchbaums T. Die Wurzel von T ist s. Jeder Weg $s \to v$ in T entspricht kürzestem Weg $s \to v$ in G.

Status eines Knotens: durch Farbe angezeigt,

- Weiß: unentdeckt;
- Grau: entdeckt (und es gibt möglicherweise weiße Nachbarn);
- Schwarz: Suche für u abgeschlossen (keine weißen Nachbarn).

Eigenschaften:

- Algorithmus besucht alle $u \in V$ mit $d(s,u) = k$, bevor irgendein $w \in V$ mit $d(s,w) = k+1$ besucht wird.
- Breitensuchbaum:
  - besteht anfangs nur aus s,
  - dann werden für jeden aktuellen Knoten u weiße Knoten $v \in \text{Adj}[u]$ an u angehängt.
  - u wird damit Vorgänger von v in T.
- Jeder Knoten speichert Attribute:
  - Farbe $f : V \to \{\text{WEISS, GRAU, SCHWARZ}\}$,
  - Vorgänger $\pi : V \to V \cup \{\text{NIL}\}$,
  - Abstand von der Wurzel $D : V \to \mathbb{N}$.
- Algorithmus benutzt FIFO-Warteschlange Q (first-in-first-out). Operationen:
  - ENQUEUE(Q, x): $x \in V$ an Q anfügen, $(Q \leftarrow x)$
  - DEQUEUE(Q): Erstes Element entfernen und ausgeben $(x \leftarrow Q)$.

Algorithm BFS(G, s)

```
 1: f(s) :=GRAU
 2: D(s) := 0
 3: π(s) :=NIL
 4: ENQUEUE(Q, s)
 5: for u ∈ V \ {s} do
 6:     f(u) :=WEISS
 7:     D(u) := ∞
 8:     π(u) :=NIL
 9: end for
10: while Q ≠ [] do
11:     u :=DEQUEUE(Q)
12:     for v ∈ Adj[u] do
13:         if f(v) =WEISS then
14:             f(v) :=GRAU
15:             D(v) := D(u) + 1
16:             π(v) := u
17:             ENQUEUE(Q, v)
18:         end if
19:     end for
20:     f(u) :=SCHWARZ
21: end while
```

Laufzeit

Eingabe: $G = (V, E)$

Eigenschaften:

- Jeder Knoten der selben Komponente wie s wird genau einmal an Q angehängt.
  Kosten: je $\mathcal{O}(1)$, insgesamt also $\mathcal{O}(|V|)$.
- Jede Adjazenzliste wird einmal durchsucht (Adj[u], nachdem $u \leftarrow Q$). Wegen

$$\sum_{u \in V} |\text{Adj}[u]| = \sum_{u \in V} d(u) = 2|E|.$$

  haben wir insgesamt $\mathcal{O}(|E|)$
- Initialisierung: $\mathcal{O}(|V|)$

Alles zusammen ergibt Kosten von $\mathcal{O}(|V| + |E|)$.

Korrektheit

Gegeben: $G = (V, E)$ und $s \in V$ (O.B.d.A. G zusammenhängend)

$$d(s,v) = \begin{cases} \min\limits_{W(s \to v)} \text{Länge von } W, & \text{falls es solche Wege gibt,} \\ \infty & \text{sonst.} \end{cases}$$

Lemma

$uv \in E \Rightarrow d(s,v) \leq d(s,u) + d(u,v) = d(s,u) + 1$

Beweis: Dreiecksungleichung.

Lemma

Annahme: $\mathrm{BFS}(G, s)$ *ausgeführt. Dann gilt* $D(v) \geq d(s, v)$ *für alle* $v \in V$.

Beweis: Induktion nach Anzahl der ENQUEUE-Aufrufe.
Nach dem ersten Aufruf: $D(s) = 0$, auch $d(s, s) = 0$;
$v \neq s$: $D(v) = \infty \geq d(s, v)$

Sei $Q \leftarrow u$ der zuletzt durchgeführte Aufruf von ENQUEUE.
Nach der Induktionsvoraussetzung gilt für alle $v \in V$:
$D(v) \geq d(s, v)$.
ENQUEUE(v) passiert, wenn v (von einem Knoten w aus) entdeckt also grau wird. Daraus folgt $v \sim w$ und daher

$$D(v) := D(w) + 1 \geq d(s, w) + 1 \geq d(s, v).$$

Die D-Werte der anderen Knoten werden nicht verändert. □

Lemma

$\mathrm{BFS}(G, s)$ *werde gerade ausgeführt, zum aktuellen Zeitpunkt gelte* $Q = [v_1, v_2, \ldots, v_r]$. *Dann gilt* $D(v_r) \leq D(v_1) + 1$ *und* $D(v_i) \leq D(v_{i+1})$ *für* $1 \leq i \leq r - 1$.

Beweis: Induktion nach Anzahl der Warteschlangenoperationen (ENQUEUE,DEQUEUE).
Induktionsanfang: $Q = [s]$ ✓

Induktionsschritt:
DEQUEUE: $\rightsquigarrow Q = [v_2, \ldots, v_r]$. Laut IVS gilt:
$D(v_1) \leq D(v_2)$ und $D(v_r) \leq D(v_1) + 1$ und somit
$D(v_r) \leq D(v_2) + 1$.

ENQUEUE: $\rightsquigarrow Q = [v_1, v_2, \ldots, v_r, v_{r+1}]$ mit $v_{r+1} \in \mathrm{Adj}[u]$, wobei $u \leftarrow Q$ unmittelbar vorher.
Daher $D(u) \leq D(v_1)$ und folglich
$D(v_{r+1}) := D(u) + 1 \leq D(v_1) + 1$.
Die IVS liefert dann $D(v_r) \leq D(u) + 1 = D(v_{r+1})$. □

Folgerung

Wenn $Q \leftarrow v_i$ vor $Q \leftarrow v_j$ ausgeführt wird, dann gilt $D(v_i) \leq D(v_j)$.

Satz

BFS *ist korrekt, d.h., nach der Ausführung erfüllen alle Knoten $D(v) = d(s, v)$. Es gibt einen kürzesten Weg $W(s \to v)$ von der Form $W'(s \to \pi(v)) \cup \{\pi(v)v\}$, wobei $W'(s \to \pi(v))$ ein kürzester Weg von s nach $\pi(v)$ ist.*

Beweis: 1. Teil: Annahme, es gebe $v \in V$ mit $D(v) > d(s, v)$.
Sei $d(s, v)$ minimal unter all jenen v.
(insbesondere gilt dann $v \neq s$ und $d(s, v) < \infty$)

Sei $s - v_1 - v_2 - \cdots - v_k - u - v$ ein kürzester Weg.
Dann gilt $d(s, v) = d(s, u) + 1$ und $D(u) = d(s, u)$ und daher

$$D(v) > d(s, v) = d(s, u) + 1 = D(u) + 1.$$

Wir haben also $s - v_1 - v_2 - \cdots - v_k - u - v$ und

$$D(v) > D(u) + 1. \qquad (*)$$

Während der Ausführung von BFS wird irgendwann $u :=$DEQUEUE(Q) aufgerufen. Damit ist u der aktuelle Knoten.

1. **Fall:** v weiß.
   Dann wird später $D(v) := D(u) + 1$ ausgeführt. $\lightning$ zu $(*)$
2. **Fall:** v schwarz.
   Dann war $v :=$DEQUEUE(Q) bevor $u :=$DEQUEUE(Q).
   Dann aber auch $Q \leftarrow v$ vor $Q \leftarrow u$.
   Aufgrund der Folgerung ist dann $D(v) \leq D(u)$ $\lightning$ zu $(*)$
3. **Fall:** v grau.
   Dann wurde v nach einem Aufruf $w :=$DEQUEUE(Q) gefärbt.
   Daher muss $Q \leftarrow w$ vor $Q \leftarrow u$ erfolgen, woraus
   $D(w) \leq D(u)$ folgt. Aber nach $w :=$DEQUEUE(Q) erfolgt
   $D(v) := D(w) + 1 \leq D(u) + 1$ $\lightning$ zu $(*)$

2. Teil: Es gibt einen kürzesten Weg $W(s \to v)$ von der Form $W'(s \to \pi(v)) \cup \{\pi(v)v\}$, wobei $W'(s \to \pi(v))$ ein kürzester Weg von s nach $\pi(v)$ ist.

Wenn $\pi(v) = u$, dann erfolgt irgendwann die Zuweisung $\pi(v) := u$ und unmittelbar davor $D(v) := D(u) + 1$.

Aus dem Teil 1 folgt nun $d(s, v) = d(s, u) + 1$. □

Exkurs: weitere Grundbegriffe der Graphentheorie

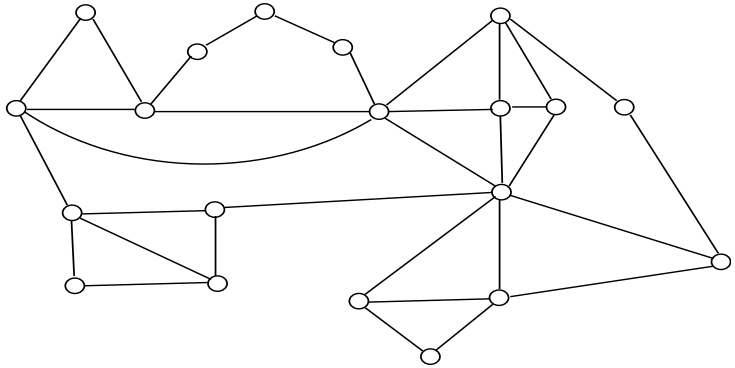
Definition (Erreichbarkeitsrelation)

$G = (V, E)$, $v, w \in V$. Wir definieren: $v \approx w$ genau dann, wenn eine (möglicherweise leere) Kantenfolge von v nach w existiert.

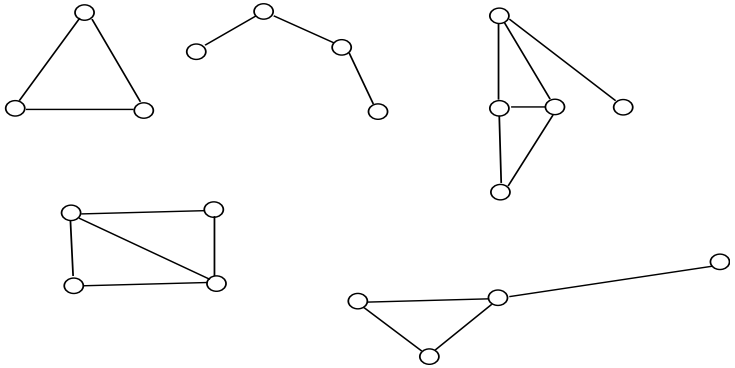
Satz

Die Erreichbarkeitsrelation ist eine Äquivalenzrelation auf V. Ihre Äquivalenzklassen heißen Zusammenhangskomponenten von G.

zusammenhängender Graph

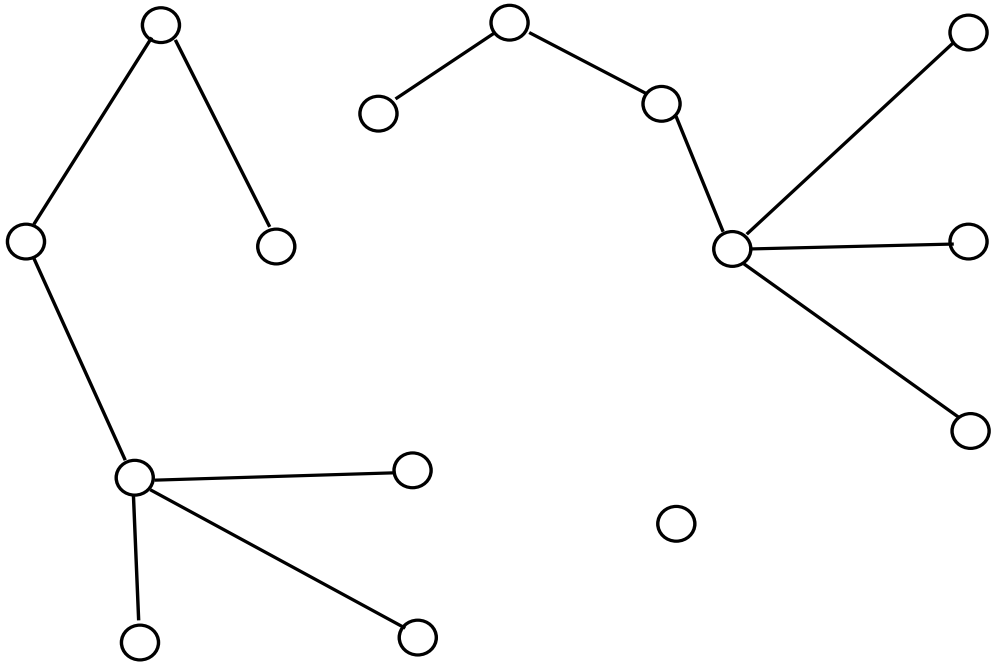


nicht zusammenhängender Graph



Definition

Ein schlichter ungerichteter Graph, der keine Kreise positiver Länge besitzt, heißt Wald.
Ein zusammenhängender Wald heißt Baum



Satz

Sei $G = (V, E)$ ein Graph. Die folgenden Aussagen sind äquivalent:

1. *G ist ein Baum.*
2. *Für alle $v, w \in V$ existiert genau ein Weg $W(v \to w)$.*
3. *G ist zusammenhängend und $|V| = |E| + 1$.*
4. *G ist minimal zusammenhängend, d.h. Entfernen einer Kante zerstört den Zusammenhang.*
5. *G ist maximal azyklisch, d.h. G hat keine Kreise und Hinzufügen einer Kante erzeugt einen Kreis.*

Wir beweisen exemplarisch die Äquivalenz (1) $\Longleftrightarrow$ (3).

Definition

Ein Knoten v eines Graphen $G = (V, E)$ heißt Endknoten, wenn $d(v) = 1$.

Lemma

Ein Baum mit mindestens zwei Knoten besitzt mindestens zwei Endknoten.

Beweis: Man betrachte einen längsten Weg

$$v - v_1 - v_2 - v_3 - \cdots\cdots - v_k - w.$$

Dann müssen v und w Endknoten sein. □

(1) $\Longrightarrow$ (3):
Ein Baum ist zusammenhängend und erfüllt $|V| = |E| + 1$.

Dass G zusammenhängend ist, folgt daraus, dass G ein Baum ist.

$|V| = |E| + 1$: Induktion nach $n = |V|$.

$n = 1$ ✓

Sei G ein Baum mit $n + 1$ Knoten.
Dann gibt es einen Endknoten v und eine mit v inzidente Kante e.
Wende nun die IVS auf $G' = (V \setminus \{v\}, E \setminus \{e\})$ an.
(Beachte: G' ist zusammenhängend, also auch ein Baum)

$$\alpha_0(G') = \alpha_1(G') + 1.$$

$\alpha_0(G') = |V| - 1$, $\alpha_1(G') = |E| - 1$ leisten das Gewünschte. □

(3) $\Longrightarrow$ (1):
Ein zusammenhängender Graph mit $|V| = |E| + 1$ ist ein Baum.

G ist zusammenhängend, also muss nur Kreisfreiheit gezeigt werden.

Annahme, es gebe einen Kreis (V_k, E_k) mit $V = \{v_1, \ldots, v_k\}$.
Dann gilt $|V_k| = |E_k| = k$ und $k < |V|$.

Dann gibt es $v_{k+1} \in V \setminus V_k$ und ein i mit $e = v_i v_{k+1} \in E$.
Sei $G_{k+1} = (V_k \cup \{v_{k+1}\}, E_k \cup \{e_{k+1}\})$.
Es gilt $|V_{k+1}| = |E_{k+1}| = k + 1$.

Falls $k + 1 < |V|$, dann gibt es v_{k+1} und e_{k+1} analog zu vorhin.

$\rightsquigarrow G_{k+2}, G_{k+3}, \ldots, G_n = (V, E_n)$ mit $n = |V| = |E_n| > |E|$ $\lightning$ $\square$