

# Algorithmen

## 1) Einführung

- Algorithmus := Eine eindeutige Handlungsvorschrift zur Lösung eines Problems oder einer Klasse von Problemen. Algorithmen bestehen aus endl. vielen wohldef. Einzelschritten

- Fibonacci Zahlen

$$F(n) := \begin{cases} 0, & n=0 \\ 1, & n=1 \\ F(n-1) + F(n-2), & n>2 \end{cases}$$

- Für  $n \geq 6$ :  $2^{\frac{n}{2}} \leq F(n) \leq 2^n$

- Algo: nicht-rekursive einfache Schleife

INPUT:  $n \in \mathbb{N}$

OUTPUT:  $F_n$

ALGORITHM: FibLoop( $n$ )

```

1 Allokiere F als ein Array der Länge  $n+1$ ;  $\mathcal{O}(n)$ 
2  $F[0] \leftarrow 0;$  }  $\mathcal{O}(1)$ 
3  $F[1] \leftarrow 1;$  }  $\mathcal{O}(1)$ 
4 for  $i \in \{2, \dots, n\}$  do }  $\mathcal{O}((n-1) \cdot n)$ 
5   |  $F[i] \leftarrow F[i-1] + F[i-2];$ 
6 end
7 return  $F[n]$ 
```

$$\Rightarrow \mathcal{O}(n+2 + (n-1) \cdot n) = \mathcal{O}(n^2)$$

- Algo: Rekursion

INPUT:  $n \in \mathbb{N}$

OUTPUT:  $F_n$

ALGORITHM: FibRekursive( $n$ )

```

1 if  $n=0$  then }  $\mathcal{O}(1)$ 
2   | return 0
3 end
4 if  $n=1$  then }  $\mathcal{O}(1)$ 
5   | return 1
6 end
7 return FibRekursive( $n-1$ ) + FibRekursive( $n-2$ )
 $\Rightarrow T(n) \geq 2^{\frac{n}{2}}$ 
```

$$- Oder: \begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$$

mit  $\mathcal{O}(8 \cdot \log(n) \cdot l^{2,59})$

l = Bit-Zahlen

## Pseudocode-Konventionen:

- Name: Algo(input)
- INPUT
- OUTPUT
- Standardkonstrukte:

- if (...) then  
| ...  
end
- else  
| return ...  
end
- foreach (...) do  
| ...  
end

- while (...) and (...) do  
| :  
end
- $i \in \{1, \dots, n\}$
- $i \leftarrow i+1;$

## 2) $\mathcal{O}$ -Notation

Orientiert sich an größtem Term

$f \in O(g) \Leftrightarrow f$  ist von Ordnung höchstens  $g$   
 $\Leftrightarrow \exists c > 0 \ \exists n_0 > 0 \ \forall n > n_0 : 0 < f(n) \leq c \cdot g(n)$   
 $\Leftrightarrow \limsup_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty$

$f \in o(g) \Leftrightarrow f$  ist von Ordnung echt kleiner als  $g$   
 $\Leftrightarrow \forall c > 0 \ \exists n_0 > 0 \ \forall n > n_0 : 0 < f(n) < c \cdot g(n)$   
 $\Leftrightarrow \limsup_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$

$f \in \Omega(g) \Leftrightarrow f$  ist von Ordnung mind.  $g$   
 $\Leftrightarrow \exists c > 0 \ \exists n_0 > 0 \ \forall n > n_0 : f(n) \geq c \cdot g(n) > 0$   
 $\Leftrightarrow 0 < \liminf_{n \rightarrow \infty} \frac{f(n)}{g(n)} \leq \infty$   
 $\Leftrightarrow g \in O(f)$

$f \in \omega(g) \Leftrightarrow f$  ist von Ordnung echt größer als  $g$   
 $\Leftrightarrow \forall c > 0 \ \exists n_0 > 0 \ \forall n > n_0 : f(n) > c \cdot g(n) > 0$   
 $\Leftrightarrow \liminf_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$   
 $\Leftrightarrow g \in O(f)$

$f \in \Theta(g) \Leftrightarrow f$  ist von selber Ordnung wie  $g$   
 $\Leftrightarrow \exists c_1, c_2 > 0 \ \exists n_0 > 0 \ \forall n > n_0 : c_1 g(n) \leq f(n) \leq c_2 g(n)$   
 $\Leftrightarrow 0 < \limsup_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty$   
 $\Leftrightarrow f \in O(g) \text{ und } f \in \Omega(g)$

## Landau Notation. Regeln aus der Übung:

- $f = o(g)$  und  $g = O(h) \Rightarrow f = O(h)$
- $f = O(g)$  und  $g = o(h) \Rightarrow f = O(h)$
- ~~$f = \Theta(g) \Leftrightarrow g = \omega(f)$~~
- $f = \Omega(g)$  und  $g = \Omega(h) \Rightarrow f = \Omega(h)$

## asymptotisches Wachstum

$$f < g \Leftrightarrow f(n) \in O(g(n))$$

$$\log(n) < 2^{\sqrt{\log n}} < 42n < \log(n!) < (\log n)^{\log n} < (\log n)^{\sqrt{n}} < \log(\log n)$$

$$\begin{aligned} 1 &< \log \log n < \log n < \sqrt[n]{n} < n < n \cdot \log n \\ &< n^2 < 2^n < n! < n^n \end{aligned}$$

## Amortisierte Analyse

= Aufwiegeln von teuren und günstigen Operationen

• z.B.:  $f(n) = n^2 - n + 3$

$f \in O(n^2)$ ,  $f \in \Theta(n^3)$ ,  $f \in \Omega(n^2)$ ,  $f \in \omega(n)$ ,  $f \in \Theta(n^2)$

-  $f(n) = n^2 + 3n + 7$

$f \in O(n^2)$ ,  $f \in \Theta(n^3)$ ,  $f \in \omega(\log n)$ ,  $f \in \Theta(n^2)$

-  $f(n) = n^2 + 10n + 5$

$f \in O(n^2)$ ,  $f \notin \Theta(n^2)$ ,  $f \in \Omega(n^2)$ ,  $f \notin \omega(n^2)$

-  $n! = \Theta(n^n)$

## Rechenregeln:

-  $f$  Polynom mit Grad d.

$\Rightarrow f \in \Theta(n^d)$ ,  $f \in O(\exp(n))$ ,  $f \in \omega(\log n)$

- bei Log. Basis egal

$\Rightarrow \log_2(n) \in \Theta(\log_b(n))$

-  $f$  Konstant

$\Rightarrow f \in O(1)$

-  $f \in O(g_1 + g_2)$  und  $g_1 \in O(g_2)$   $\Rightarrow f \in O(g_2)$

-  $f_1 \in O(g_1)$  und  $f_2 \in O(g_2)$   $\Rightarrow f_1 + f_2 \in O(g_1 + g_2)$

-  $f \in g_1 \cdot O(g_2)$   $\Rightarrow f \in O(g_1 \cdot g_2)$

-  $c > 0$  konst.:  $O(cg) = O(g)$

-  $f \in O(g_1)$ ,  $g_1 \in O(g_2)$   $\Rightarrow f \in O(g_2)$

- schreibe auch  $f = O(g)$

## z.B. Kosten

- Addition zweier standard Integer  $O(1)$

- Addition zweier Zahlen der Länge n  $O(n)$

- Multiplikation zweier standard Integer  $O(1)$

- - - - - Zahlen der Länge n und m  $O(n \cdot m)$

- Einlesen von n Strings der Länge 10  $O(n)$

- a+b+c mit a aus  $n^2$ , b aus  $\log n$ , c aus n Bits  $O(n^2 + \log n \cdot n) = O(n^2)$

## Logarithmen-Regeln

-  $\log(a \cdot b) = \log(a) + \log(b)$

-  $\log(a^b) = b \cdot \log(a)$

-  $\log_a b = \frac{\log b}{\log a}$

-  $b^{\log_a b} = a$

- z.B.  $2^{\log_4 n} = 2^{\frac{\log_2 n}{\log_2 4}} = n^{\frac{1}{\log_2 4}} = n^{1/2} = \sqrt{n}$

-  $\log_a a = 1$

-  $\log_a 1 = 0$

-  $\log_a a^k = k$

-  $2^x = 5$

$\Rightarrow \log_2 5 = x$

$$\cdot \left(\frac{n}{2}\right)^{\frac{n}{2}} \leq n! \leq n^n$$

### • Summen

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

$$-\sum_{i=0}^{\infty} x^i = \frac{1}{1-x} \quad \text{für } |x| < 1$$

$$-\sum_{i=1}^n \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \dots = \ln n + O(1) \quad \text{n-te harmon. Zahl}$$

• l'Hospital: Wenn  $\lim_{n \rightarrow \dots} \frac{g(x)}{f(x)} = \frac{0}{0}$  oder  $\frac{\infty}{\infty}$   
dann  $\dots = \lim_{n \rightarrow \dots} \frac{g'(x)}{f'(x)}$

$$-\sum_{i=0}^n x^i = \frac{x^{n+1} - 1}{x - 1} \quad \text{geom. Reihe}$$

$$-\sum_{i=0}^{\infty} i \cdot x^i = \frac{x}{(1-x)^2} \quad \text{für } |x| < 1$$

## 3) Rekursion

Rekursionen sind Algorithmen, die ein Problem in kleinere Teilprobleme zerlegen, von denen wir annehmen, dass diese - weil kleiner - einfacher zu lösen sind. Wir versuchen, die Teilprobleme so weit zu verkleinern, bis wir beim trivialen Fall sind. Sie haben häufig folgende Strukturen:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

$$T(n) = a \cdot T(n-b) + f(n)$$

mit  $n = 2^k \quad k \in \mathbb{N}$

Dabei stehen:

- a = Anzahl der Teilprobleme

- b = Größe der Teilprobleme

- f(n) = Kosten des Zusammenfügens der Teilergebnisse zum Gesamtergebnis.

• Laufzeit:

### 1) Master-theorem

Seien  $a, b \geq 1$  konstant,  $f: \mathbb{N} \rightarrow \mathbb{N}$ ,  $T(n) \geq 0$  mit

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

Dann gilt:

$n^{\log_b a} > f(n)$  i)  $f(n) = O(n^{\log_b a - \varepsilon})$  für  $\varepsilon > 0$

$n^{\log_b a} = f(n)$  ii)  $f(n) = \Theta(n^{\log_b a})$

$n^{\log_b a} < f(n)$  iii)  $f(n) = \Omega(n^{\log_b a + \varepsilon})$  mit  $\varepsilon > 0$

und  $\exists c < 1$  mit  $a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n)$

$$\Rightarrow T(n) = \Theta(n^{\log_b a})$$

$$\begin{aligned} \Rightarrow T(n) &= \Theta(n^{\log_b a} \cdot \log_b n) \\ &= \Theta(n^{\log_b a} \cdot \log_b n) \end{aligned}$$

$$\Rightarrow T(n) = \Theta(f(n))$$

$$2) \text{ geschlossene Form } T(n) = a \cdot T(n-b) + f(n)$$

- i) Rekursives Formeleinsetzen um Struktur zu erkennen; Substitution  $n=2^k$
- ii) Bew. durch Induktion über  $n$ ; mit  $n \mapsto 2n$ ;  $3^k = 3^{\log_2 n}$ ; d.h.  $k = \log_2 n$
- IV: Für eine Zuliepotenz  $n > 0$  sei  $T(n) = \dots$

• z.B.: Merge Sort :  $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n$

Binäre Suche:  $T(n) = T\left(\frac{n}{2}\right) + 1 \Rightarrow \mathcal{O}(\log_2 n) = \mathcal{O}(\log n)$

## 4) Arrays und Listen

### • Arrays

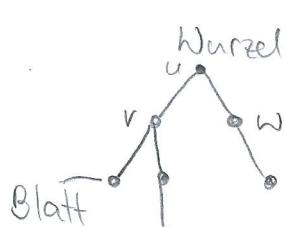
- Array der Länge  $n :=$  Anordnung von  $n$  Objekten des selben Typs in fortlaufender Adressen im Speicher.
- Zugriff in  $\mathcal{O}(1)$  über Index
- feste Größe

### • Liste

- Können Größe ändern  $\rightarrow$  dynamisch
- Element besteht aus Schlüssel und Zeiger next auf nächstes El. Zusätzlich gibt es einen Zeiger auf erstes El der Liste
- Bei doppelter Verkettung: zusätzlicher Zeiger prev auf Vorgängerel
- Operationen (doppelte Verkettung)
  - Einfügen eines El./einer Liste:  $\mathcal{O}(1)$
  - Löschen eines El./einer Liste:  $\mathcal{O}(1)$
  - Suche eines El.:  $\mathcal{O}(n)$

(schlechter wenn einfach verkettet)

## 5) Bäume



- Kinder in Liste organisiert. Knoten hat Zeiger auf linkstes Kind.
- w ist Kind von u
- u ist Vater von w
- Jeder Knoten hat Zeiger auf rechten Geschwisterknoten.

- Sei  $v \in T$  Knoten im Baum  $T$ :

$$\begin{aligned} \text{- Tiefe}(v) = & \begin{cases} 0 & v \text{ Wurzel} \\ \text{Tiefe}(\text{Vater}(v)) + 1 & \text{sonst} \end{cases} \end{aligned}$$

$$\begin{aligned} \text{- Höhe}(v) = & \begin{cases} 0 & v \text{ Blatt} \\ 1 + \max\{\text{Höhe}(w), w \text{ Kind von } v\} & \text{sonst} \end{cases} \end{aligned}$$

- Binärbaum - Knoten hat Schlüsselwert, Zeiger auf l.&r. Kind und Vater.

- Jeder Knoten hat max 2 Kinder

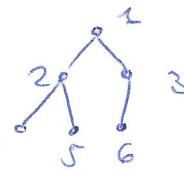
- voll:  $\Leftrightarrow$  Alle Level gefüllt

- vollständig:  $\Leftrightarrow$  Alle Level außer unterstes voll. Unten Knoten möglichst links. 5

### • Implementierung: als Array

Knoten mit Nr.  $i$ : - Vater hat Nr.  $\lfloor \frac{i}{2} \rfloor$

- Kinder haben Nr.  $2i, 2i+1$



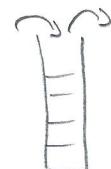
### • Höhe / Tiefe

- voller Binärbaum mit  $n$  Knoten:  $\log_2(n+1) - 1 \in \Theta(\log n)$

- vollständiger Binärbaum:  $\lceil \log_2(n+1) - 1 \rceil \in \Theta(\log n)$

## 6) Stacks und Queues

### • Keller (Stacks):



- dynamisch

- oberstes Element zugreifbar

- **Operationen:** • Push( $x$ ): Fügt El.  $x$  oben auf den Keller ein  
• Pop( $x$ ): Löscht

- Last in First out

- **Implementierung:** • Array mit Zeiger auf Top-Element  
• Operationen in  $O(1)$   
• doppelte Verkettung: Zeiger am Ende der Liste  
• einfache " " : El. vorne einfügen.

### • Warteschlangen (Queues)

- dynamisch



- **Operationen:** • Einfügen( $x$ ) (Enqueue): fügt El.  $x$  ans Ende der Queue  
• Löschen: Löscht das El. der Queue

- First in First out

- **Implementierung:** • Array mit Zeiger auf End- und Top-Element  
• Operationen in  $O(1)$

## 7) Heaps

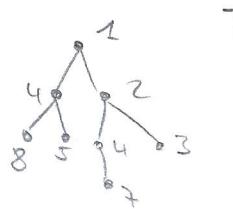
• Einfache baumartige Datenstruktur zur Verwaltung einer Menge mit Ordnungsrelation.

• Heapeigenschaft: Vater  $\leq$  Kind

• **Implementierung:** vollständiger Binärbaum

→ Baum der Höhe  $k$  hat mind.  $2^k$  Knoten

→ bzw. Höhe =  $O(\log n)$



## \* Operationen:

- ExtractMin: 1) Min ist in Wurzel  $\rightarrow O(1)$

2) - Entferne Min

- Repariere Heap: Blatt in Wurzel einfügen  
Nach unten sinken lassen }  
(Vgl. mit kleinstem Kind) }  $\rightarrow O(\log n)$

## - Initialisierung:

1) Element  $v$  "in der untersten Reihe links" einfügen.  
Heap-eigenschaft wiederherstellen. Nächstes El.

D.h.  $\text{parent}(v) > v$ ? ja  $\Rightarrow$  tausche und vgl. mit neuem parent.

$\rightarrow O(n \cdot \text{Höhe}(T)) = O(n \cdot \log n)$  [Pseudocode existent]  
Folie 75

2) Alle Schlüssel einfügen. Mache levelweise alle Unterbäume zu Heaps.  $\rightarrow O(n)$

## - IncreaseKey:

Schlüssel eines bel. Knoten  $v$  wird erhöht.  
Lasse  $v$  nach unten sinken:

Tausche Platz mit kleinerem Kind, bis  $v < \text{child}(v)$

$\rightarrow O(\log n)$  Höhe des Heaps

## - DecreaseKey

Schlüssel eines bel. Knoten  $v$  wird erniedrigt.

Tausche  $v \leftrightarrow \text{parent}(v)$  so lange, bis  $v > \text{parent}(v)$

$\rightarrow O(\log(n))$  Höhe des Heaps.

# 8) Priority Queues (Prioritätswarteschlangen)

\* Priority Queue := Datenstruktur um eine dynamische Menge von Elementen mit Prioritätswerten zu verwalten. immer "mitteltener"

## \* Operationen:

|                                     | ungeordnetes Array/Liste | geordnetes Array/Liste        | Heap                             |
|-------------------------------------|--------------------------|-------------------------------|----------------------------------|
| Erstellen                           | /                        | Sortieren $O(n \cdot \log n)$ | Aufbau $O(n)$                    |
| Einfügen                            | $O(1)$ ans Ende          | Sortieren $O(n)$              | $O(\log n)$                      |
| Löschen des Max-El.                 | Durchlaufen $O(n)$       | $O(1)$                        | $O(1)$<br>Repariere: $O(\log n)$ |
| Decrease / Increase Key (Priorwert) | $O(1)$                   | $O(n)$                        | $O(\log n)$                      |

# 9) Hashing

## Idee:

- Universum  $U = [0, \dots, N-1]$
- zu verwaltende Menge  $S \subseteq U$ ,  $|S| = n$
- Hashtafel  $T[0, \dots, m-1]$  der Größe  $m$
- Hashfunktion  $h: U \rightarrow T[0, \dots, m-1]$   
 $a \mapsto T[h(a)]$
- $\beta = \frac{n}{m}$  Belegungsfaktor

- **Operationen:** Zugriff( $a, S$ ), Einfügen( $a, S$ ), Streichen( $a, S$ )  $\rightarrow$  wollen  $O(1)$

wir bilden die El. aus  $S$  über eine Hashfkt.  $h$  in ein Array (Hashtafel  $T$ ) ab.

## • Hashing mit Verkettung

- Idee: Sei  $i \in [0, \dots, m-1]$ . Jedes El.  $T[i]$  der Hashtafel besteht aus einer Liste, die alle  $x \in S$  mit  $h(x) = i$  enthält.
- Worst Case  $O(n)$  (alle  $x \in S$  in einer Liste)
- $\delta_h(x, y) = \begin{cases} 1 & h(x) = h(y) \text{ mit } x \neq y \\ 0 & \text{sonst} \end{cases}$

Anzahl der El. in der Liste bei  $h(x)$

$$= \delta_h(x, S) = \sum_{y \in S} \delta_h(x, y)$$

$\rightarrow$  Kosten der Operationen  $(x, S) = O(1 + \delta_h(x, S))$

$\rightarrow$  Bei gleichmäßiger Verteilung der El. erwarten wir Kosten Operation  $(x, S) = O(1 + \beta)$

- Erwartungswert der längsten Liste  $L = O(\frac{\log n}{\log \log n})$

## • Wahl von $\beta = \frac{n}{m}$

- $\beta \leq 1 \rightarrow$  gute Laufzeit
- $\beta \geq \frac{1}{4} \rightarrow$  Platzausnutzung gut

$\left. \begin{array}{l} \rightarrow \text{sonst eins von} \\ \text{beidem schlecht} \end{array} \right\}$

## • Rehashing $\rightarrow$ teuer!

- Benutze Folge von Hashtafeln  $T_0, \dots$ , der Größe  $m, 2m, 4m, \dots$
- $\beta = 1$

bei Tafel  $T_i$  der Größe  $2^i m$ , kopiere El. in Tafel  $T_{i+1}$  doppelter Gr.

$$\rightarrow \beta \text{ nun} = \frac{1}{2}$$

$$\beta \leq \frac{1}{4}$$

kopiere El. von  $T_i$  nach  $T_{i-1}$

$$\rightarrow \beta \text{ nun} = \frac{1}{2}$$

### • Offene Adressierung

- Alle Tafelinträge werden max. einmal belegt.  
Ist ein Feld belegt nehme ein anderes.

→ Wir brauchen eine Folge von Hashfkt.en.

- Linear Probing:

Probiere das nächste Feld.  $h_i(x) = (h(x) + i) \bmod m$

- Quadratic Probing:

$$h_i(x) = (h(x) + c_1 \cdot i + c_2 \cdot i^2) \bmod m$$

- Double Hashing

$$h_i(x) = (h_1(x) + i \cdot h_2(x)) \bmod m$$

- Laufzeit: Einfügen / Streichen

Wenn alle Hashwerte gleichwahrscheinlich vorkommen

$$O\left(\frac{1}{1-\beta}\right)$$

- gut wenn Hashtafel leer

- Linear Probing

Invariante: Fall  $k$  auf Feld  $h(k)+x$  gespeichert ist,  
so sind Felder  $h(k), h(k)+1, \dots, h(k)+x$  belegt.

auf  $k$

Zugriff: Beginne bei  $h(k)$  und suche bis  $k$  gefunden  
oder freies Feld gefunden.

• Löschen:

1) Ersetze Element durch Spezialsymbol (# o. ä.)

Problem: Hashtafel wird voll.

Lösung: Reorganisieren

2) Einträge nach gelöschten El. testen.

Falls nicht korrekt schließe 1 nach links

Abbruch: •  $k'$  steht an Stelle  $h(k')$

• leeres Feld gefunden / alle Einträge getestet

### • Universelles Hashing

Für gegebene Datensetze sind "die meisten" Hashfkt. gut.

→ Wähle beliebige dieser Fkt.en

## • Perfektes Hashing

Für feste Menge von Schlüsseln wird kollisionsfreie (injektive) Hash-Fkt. konstruiert

## 10) Graphen

- Graph  $G = (V, E)$  besteht aus einer Menge von Knoten  $V$  und einer Menge von Kanten  $E \subseteq V \times V$ .  $|V|=n$ ,  $|E|=m$   
Notation Kanten:  $\{u, v\}$  oder  $(u, v)$
  - $u$  heißt adjazent (benachbart) zu  $v$   $\Leftrightarrow \{u, v\} \in E$
  - $(u, v) \in E$  ist incident zu  $u$  und  $v$ .
  - Gewicht:  $w(u, v)$
  - Kosten:  $c(u, v)$
  - ungewichtete Graphen können als speziell gewichtete Kanten gesehen werden:  $w(u, v) = 0 \Leftrightarrow (u, v) \notin E$   
 $w(u, v) = 1 \Leftrightarrow (u, v) \in E$
  - $(u, v) \in E$  heißt (Selbst)schleife  $\Leftrightarrow u = v$
  - Ausgangsgrad eines Knoten  $v$  ist die Zahl seiner ausgehenden Kanten.  
 $\text{outdeg}(v) = |\{w \in V \mid (v, w) \in E\}|$
  - Eingangsgrad  $\text{indeg}(v) = |\{w \in V \mid (w, v) \in E\}|$
  - $\deg(v) = \text{indeg}(v) + \text{outdeg}(v) = \text{Grad von } v$
  - Quelle := Knoten mit nur ausgehenden Kanten
  - Senke := — — — eingehenden — — —
  - Eine Folge  $(v_0, v_1, \dots, v_k)$  heißt Pfad  
 $\Leftrightarrow \forall 0 \leq i \leq k-1$  existieren Kanten  $(v_i, v_{i+1})$
  - Ein Pfad  $(v_0, \dots, v_k)$  heißt Zykel  $\Leftrightarrow v_k = v_0$
  - $G$  heißt Baum:  $\Leftrightarrow$ 
    - 1)  $\exists! v_0 \in V : \text{indeg}(v_0) = 0$
    - 2)  $\forall v \in V \setminus \{v_0\} : \text{indeg}(v) = 1$
    - 3)  $G$  ist azyklisch
- $\Rightarrow |E| = m = n - 1 = O(n)$

• Vergleich : Adjazenz -

|                          | Matrix                               | Liste  |
|--------------------------|--------------------------------------|--|
| G hat Kante von i nach j | $A[i,j] = 1$                         | $j \in L_i$  |
| G ist ungerichtet        | $\forall i,j \in V: A[i,j] = A[j,i]$ | $\forall i,j \in V: j \in L_i \Leftrightarrow i \in L_j$ |
| G ist schleifenfrei      | $\forall i \in V: A[i,i] = 0$        | $\forall i \in V: i \notin L_i$                          |

• Netzwerk  $(V, E, c)$  besteht aus Graph  $G = (V, E)$  und Kantenkosten  $c: E \rightarrow \mathbb{R}$ .

Kosten des Pfades  $c(v_0, v_1, \dots, v_k) := \sum_{i=1}^k c(v_{i-1}, v_i)$

- $G$  heißt Wald  $\Leftrightarrow G = G_1 \cup \dots \cup G_k$  mit  $G_i$ : Baum
- $G$  heißt vollständig  $\Leftrightarrow \forall u, v \in V : u \neq v \Rightarrow (u, v) \in E$   
 $\Rightarrow G$  hat  $\frac{n \cdot (n-1)}{2} = O(n^2)$  Kanten
- $G$  heißt bipartit  $\Leftrightarrow V = A \cup B$ ,  $A \neq \emptyset \neq B$  und  $\forall (u, v) \in E : u \in A \Leftrightarrow v \in B$   
 $\rightarrow$  vollst. bipartit hat  $n \cdot n$  Kanten
- $G' = (V', E') \subset G$  heißt Teilgraph  $\Leftrightarrow$  induzierter
  - $V' \subseteq V$
  - $\forall (u, v) \in E$  mit  $u, v \in V' : (u, v) \in E'$
- Ein ungerichteter Graph heißt zusammenhängend  
 $\Leftrightarrow \forall u, v \in V : \exists$  Pfad von  $u$  nach  $v$ .
- Eine Zusammenhangskomponente (ZK) eines ungerichteten Graphen  $G$  ist ein maximaler zusammenhängender Teilgraph von  $G$ .
- Ein gerichteter Graph  $G$  heißt stark zusammenhängend  
 $\Leftrightarrow \forall u, v \in V : u \neq v \Rightarrow \exists$  Pfad von  $u$  nach  $v$  und  
 $\exists$  Pfad von  $v$  nach  $u$
- Eine starke Zusammenhangskomponente (SZK) ist ein maximaler stark zusammenhängender Teilgraph eines gerichteten Graphen  $G$ .

- Adjazenzmatrix  $A = (a_{i,j})$  mit  $a_{i,j} = \begin{cases} 1 & (i, j) \in E \\ 0 & \text{sonst} \end{cases}$   $i \downarrow j \rightarrow$ 
  - $G$  ungerichtet  $\Rightarrow A$  sym.
  - Platz  $O(n^2)$
  - Zugriff  $O(1)$
  - Bei dünnen Graphen ( $m = O(n)$ ) nicht effizient.  
 $A^k[i, j] = \text{Anzahl verschiedener Pfade von } i \text{ nach } j$
- Adjazenzliste: Speichert für jeden Knoten seine Nachbarknoten
  - $G$  gerichtet:
    - InAdj(v) =  $\{w \in V \mid (w, v) \in E\}$
    - OutAdj(v) =  $\{w \in V \mid (v, w) \in E\}$
  - $G$  ungerichtet:
    - Adj(v) =  $\{w \in V \mid (v, w) \in E\}$
    - Platz  $O(n+m)$
    - Zugriff  $O(\text{outdeg}(v))$
    - Besonders bei dünnen Graphen effizient. Zugriff schlechter als bei A.

- DAG = gerichtetes azyklischer Graph
- Sei  $G$  gerichtet, num:  $V \rightarrow \{1, 2, \dots, n\}$   
heißt topologische Sortierung  
 $\Leftrightarrow \forall (v, w) \in E : \text{num}(v) < \text{num}(w)$
- $G$  besitzt top. Sortierung  $\Leftrightarrow G$  azyklisch
- Algo:  
 $\text{TopSort}((V, E), i)$ : (rekursiv)  
 1 if ( $|V|=1$ )  
 2 | num(v)  $\leftarrow i$  für  $v \in V$ ;  
 3 end  
 4 if ( $|V|>1$ )  
 5 |  $v \leftarrow$  Knoten aus  $V$  mit indegree 0;  
 6 | num(v)  $\leftarrow i$ ;  
 7 |  $\text{TopSort}((V \setminus \{v\}, E \setminus \{(v, w) \mid (v, w) \in E\}), i+1)$ ;  
 8 end

z.B.:



### • Iterativ:

$\text{TopSort}((V, E))$ :

```

1 count  $\leftarrow 0$ ;  

2 while ( $\exists v \in V$  mit  $\text{indeg}(v) = 0$ )  

3   | count ++;  

4   | num(v)  $\leftarrow$  count;  

5   | streiche v mit ausgehenden Kanten;  

6 end  

7 if (count  $< |V|$ )  

8   | return 'G zyklisch';  

9 end  

10 return num;
  
```

- Sei  $G$  azyklisch.  $C \subseteq V$  heißt Vertex-cover  
 $\Leftrightarrow \forall (u, v) \in E : u \in C \text{ oder } v \in C$
- $C^* \subseteq V$  heißt Minimum-Vertex-cover  
 $\Leftrightarrow |C^*| \leq |C|$  für alle  $C$  Vertex-Cover

# 11) Single Source Shortest Path (SSSP)

•  $s \in V$  Startpunkt der Pfade

$\forall u \in V \quad P(s, u)$  Menge aller Pfade von  $s$  nach  $u$ .

$$d(u) := \begin{cases} \infty & P(s, u) = \emptyset \\ \inf \{c(p) \mid p \in P(s, u)\} & \text{sonst} \end{cases}$$

zyklischer Pfad  $p$  mit  $c(p) < 0$  heißt negativer Zykel

•  $u \in V$ :

-  $d(u) = -\infty \Leftrightarrow u$  erreichbar von negativem Zykel, der von  $s$  aus erreichbar ist.

-  $d(u) \in \mathbb{R}$   $\Rightarrow$   $\exists$  billigster Weg von  $s$  nach  $u$  mit Kosten  $d(u)$

~ Sei  $G$  DAG. azyklisches Netzwerk

Algo: SSSP( $G, s$ )

1 Distanzen  $d(s) \leftarrow 0$

2 Pfad( $s$ )  $\leftarrow s$

3 for all  $(v \in V \setminus \{s\})$

4 |  $d(v) \leftarrow \infty$

5 end

6 for ( $v \leftarrow s+1$  bis  $n$ )

Vorgängerknoten

7 |  $d(v) \leftarrow \min_{u \geq s} d(u) + c(u, v) \mid \stackrel{\uparrow}{(u, v)} \in E \}$

Kosten kürzester Pfad

8 | Pfad( $v$ )  $\leftarrow \text{concat}(Pfad(u), v)$

kürzester Pfad

9 end return  $d, \text{Pfad};$

Laufzeit  $\mathcal{O}(n+m)$

• Dijkstra

-  $s$  Startknoten

-  $S = \{v \in V \mid d(v) = f(v)\}$

-  $S' = \{v \in V \setminus S \mid v \text{ hat Nachbar in } S\}$

- Vergrößere  $S$  bis  $S = V$

Dijkstra(G, s) mit  $G: c(v) > 0 \forall v \in V$

```
1  $S \leftarrow \{s\}$ 
2  $d(s) \leftarrow 0$ 
3  $d'(s) \leftarrow 0$ 
4  $S' \leftarrow \text{OutAdj}(s)$ 
5 for all  $(u \in S')$ 
6   |  $d'(u) \leftarrow c(s, u)$ 
7 end
8 for all  $(u \in V \setminus (S' \cup \{s\}))$ 
9   |  $d(u) \leftarrow \infty$ 
10 end
11 while  $(S \neq V)$ 
12   | wähle  $w \in S'$  geeignet // findmin
13   |  $d(w) \leftarrow d'(w)$ 
14   |  $S \leftarrow S \cup \{w\}$ 
15   |  $S' \leftarrow S' \setminus \{w\}$ 
16   | for all  $(u \in \text{OutAdj}(w))$ 
17     |   | if  $(u \notin (S' \cup S))$ 
18     |   |   |  $S' \leftarrow S' \cup \{u\}$ 
19     |   | end
20     |   |  $d'(u) \leftarrow \min\{d'(u), d'(w) + c(w, u)\}$ 
21   | end
22 end
```

$\Rightarrow O(n+m \cdot \log n)$

- $\text{dist}(u, v) = \text{Länge des kürzesten Pfades zw. } u, v$
- $\text{diam}(G) = \max_{(u, v) \in V \times V} \{ \text{dist}(u, v) \}$   
= größter Abstand, den zwei Knoten in einem Graph haben
- $\arg \min_x f(x) = \{x \mid f(x) = \min_y f(y)\}$   
=  $x$  welches  $f$  minimiert
- $\text{zentrum}(G) = \arg \min_{u \in V} \{ \max_{v \in V} \{ \text{dist}(u, v) \} \}$   
= Knoten, der von allen anderen aus am günstigsten zu erreichen ist.

### Bellman - Ford

Sei  $G$  azyklisches Netzwerk (neg. Kantenkosten, erlaubt), Kantenreihenfolge gegeben

#### Bellman Ford ( $G, s$ )

```

1  $d(s) \leftarrow 0$ 
2 for all  $(v \neq s)$ 
3   |  $d(v) \leftarrow \infty$ 
4 end
5 for  $(i \in \{1, \dots, n-1\})$ 
6   | for all  $((v, w) \in E)$ 
7     |  $| d(w) \leftarrow \min \{d(w), d(v) + c(v, w)\}$ 
8   | end
9 end

```

$\rightarrow \mathcal{O}(n \cdot m)$

- negative Zykel bemerken:

-  $d$  nach letzter Iteration merken.

-  $n$  weitere Iterationen von z. 6-8. erhalte  $d'$

- Vergleiche  $d$  und  $d'$ .  $\forall v$ :

-  $d'(w) < d(w) \Rightarrow w$  liegt in neg. Zykel

-  $d'(w) = d(w) \Rightarrow$  Es kein neg. Zykel über  $w$

## 12) All Pairs Shortest Paths (APSP)

### Floyd - Warshall

- Sei  $V = \{1, \dots, n\}$ ,  $(V, E, c)$  Netzwerk ohne neg. Zykel

- Seien  $i, j \in V$ ,  $0 \leq k \leq n$ .

$f_k(i, j)$  - Kosten des billigsten Weges von  $i$  nach  $j$  mit inneren Knoten zwischen 0 und  $k$ .

$$= \min \{f_{k-1}(i, j), f_{k-1}(i, k) + f_{k-1}(k, j)\}$$

mit  $f_0(i, j) := \begin{cases} c(i, j) & (i, j) \in E \\ 0 & i = j \\ \infty & \text{sonst} \end{cases}$

## - Floyd / Warshall (G)

```
1 for all ( $i \neq j \in V$ )
2   if ( $(i,j) \in E$ )
3      $\delta_0(i,j) \leftarrow c(i,j)$ 
4   else
5      $\delta_0(i,j) \leftarrow \infty$ 
6 for all ( $i=j \in V$ )
7    $\delta_0(i,j) \leftarrow 0$ 
8 for ( $k \in \{1, \dots, n\}$ )
9   for all ( $i, j \in V$ )
10     $\delta_k(i,j) = \min \{\delta_{k-1}(i,j), \delta_{k-1}(i,k) + \delta_{k-1}(k,j)\}$ 
```

$\rightarrow O(n^3)$

### • Alternativen:

-  $n$  mal Bellman Ford (jeder Knoten ist 1 mal Quelle)  
 $\rightarrow O(n^2 \cdot m)$

-  $n$  mal Dijkstra (jeder Knoten ist 1x Quelle, Kantenkosten pos!)  
 $\rightarrow O(n^2 \log n + n \cdot m)$

$n \times$  Dijkstra

$1 \times$  Bellman Ford für nichtneg. Kanten

nicht neg

## 13) Graphendurchmusterung

### - Durchmusterung ( $G, s$ ) $G = (V, E)$ , $s$ Startknoten, $S = \{b\} \text{ bekannte Knoten}\}$

```
1  $S \leftarrow \{s\}$  +  $S$  boolesches Array
2 markiere alle Kanten als unbesucht  $\nwarrow$  Adjazenzlisten
3 while ( $\exists v \in S$  mit unbewerteter Kante  $(v,w) \in E$ )
4   sei  $e = (v,w)$  eine solche Kante  $\nwarrow$  über Menge  $S' \subseteq S$  zu
5   markiere  $e$  als besucht  $\downarrow$  finden
6    $S \leftarrow S \cup \{w\}$  als Queue oder Stack
```

- Invariante: Alle Knoten in  $S$  sind über besuchte Kanten von  $s$  aus erreichbar.

- Stack Implementierung heißt Tiefensuche (DFS) 

- Queue -  - Breitensuche (BFS)

- Sei  $G = (V, E)$ ,  $s$  Start,  $S$  Menge bekannter Knoten,  $S' = \{ \text{knoten mit unbewerteten Kanten} \}$

### Explorefrom ( $s$ )

```

1   $S \leftarrow \{s\}$ 
2   $S' \leftarrow \{s\}$ 
3  for all  $(v \in V)$ 
4     $p(v) \leftarrow \text{adjlistenstart}(v)$            - PFS
5  while  $(S' \neq \emptyset)$ 
6    sei  $v \in S'$  bel.
7    if  $(p(v)$  nicht am Listenende)
8       $w \leftarrow p(v)$ 
9      verschiebe  $p(v)$    ← Pointer 1 nach vorne schieben
10     if  $(w \notin S)$ 
11       einfige  $(w, S)$ 
12       einfige  $(w, S')$ 
13     else
14       streiche  $(v, S')$ 

```

$\rightarrow \mathcal{O}(n+m)$  mit  $G_s$  von  $V_s$  induzierter Teilgraph von  $G$

### Berechnung von Zusammenhangskomponenten

1 Initialisierung wie bisher  $S \leftarrow \emptyset$

2 for all  $(s \in V)$

3 | if  $(s \notin S)$

4 | | Explorefrom( $s$ )

Für  $G$  ungerichtet  $\mathcal{O}(n+m)$ ,

$\rightarrow$  Adj. matr.  $\mathcal{O}(n^2)$

### DFS rekursiv

-  $S'$  implizit als Aufrufkeller der Rekursion,  $S'$  als Stack

- dflsnum, compnum Aufruf-/Abschlussreihenfolge

### DFS( $v$ ):

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

2 | if  $(w \notin S)$

3 | |  $S \leftarrow S \cup \{w\}$  füge  $(v, w)$  zu  $T$  hinzu

4 | | dflsnum( $w$ )  $\leftarrow$  count + 1

5 | | count + 1 ++

```

6   |   DFS(w)
7   |   componum(w) ← count2
8   |   count2 ++
9   |   füge (v,w) zu T hinzu
10  |   elseif (v →* w) Pfad über bereits entdeckte
11  |       |   füge (v,w) zu F hinzu Baumkanten
12  |   elseif (w →* v)
13  |       |   füge (v,w) zu B hinzu
14  |   else
15  |       |   füge (v,w) zu C hinzu

```

### - Main

```

1 S ← ∅
2 count1 ← 1
3 count2 ← 2
4 for all (v ∈ S)
5     |   S ← S ∪ {v}
6     |   dfsonum(v) ← count1
7     |   count1 ++
8     |   DFS(v)
9     |   componum(v) ← count2
10    |   count2 ++

```

↗ gerichtet  
 =>  $O(n+m)$

- dfsonum = Reihenfolge, in der die Knoten entdeckt wurden

- componum = Abschlussreihenfolge

- T - Baumkanten = Kanter, über die man durch die Tiefe -  
suche neue Knoten entdeckt hat  
= Aufbaum der Rekursion
- F = Vorrückskanten (v,w) = der Knoten w kann von  
v aus auch über Baumkanten erreicht werden
- B = Rückwärtskanten (v,w) = der Knoten v kann von  
w aus auch über Baumkanten erreicht werden
- C = Querkanter (v,w) = über Baumkanten existiert weder  
ein Pfad von v nach w, noch einer von w nach v

- Für alle  $(v, w) \in E$ :
  - $(v, w) \in T \cup F \Leftrightarrow \text{dfsnum}(v) < \text{dfsnum}(w)$
  - $(v, w) \in B \Leftrightarrow \text{dfsnum}(v) > \text{dfsnum}(w)$   
und  $\text{compnum}(w) > \text{compnum}(v)$
  - $(v, w) \in C \Leftrightarrow \text{dfsnum}(v) > \text{dfsnum}(w)$   
und  $\text{compnum}(w) < \text{compnum}(v)$
- $G$  acyklisch und gerichtet  $\Rightarrow$  Rückwärtskanten
- $n+1 - \text{compnum}(v)$  ist topologische Sortierung

### BFS (\$\$)

```

1  $ \leftarrow \emptyset
2  $' \leftarrow \emptyset
3  for all  $(v \in V)$ 
4  | p(v) \leftarrow \text{adj.listenerstart}(v)
5  while  $(S \neq \emptyset)$ 
6    | sei  $v \in S$  bel.
7    | if (p(v) nicht am Listenende)
8    |   | w \leftarrow p(v)
9    |   | verschiebe p(v)
10   |   | if ( $w \notin S$ )
11   |   |     | einfüge  $(w, S)$ 
12   |   |     | einfüge  $(w, S')$ 
13   |   | else
14   |   |     | streiche  $(v, S')$ 

```

•  $S'$  als Queue

$\rightarrow O(n+m)$

## 14) Starke Zusammenhangskomponenten

- $G = (V, E)$  gerichtet heißt stark zusammenhängend  
 $\Leftrightarrow \forall v, w \in V: \exists \text{path}(v, w)$
- $G'$  heißt SZK (starke Zusammenhangskomponente)  
 $\Leftrightarrow G'$  ist maximaler stark zusammenhängender Teilgraph von  $G$  19

- Sei  $G' = (V', E')$  STK von  $G$ .  
 $v \in V'$  heißt Wurzel : $\Leftrightarrow$   
 $v$  hat minimale dflsnum  $\forall v \in V'$   
 $\Rightarrow$  alle Knoten von  $G'$  sind von der Wurzel aus über Baumkanten erreichbar
- STK heißt abgeschlossen : $\Leftrightarrow$   $\text{dfs}(v) \forall v \in V'$  abgeschlossen