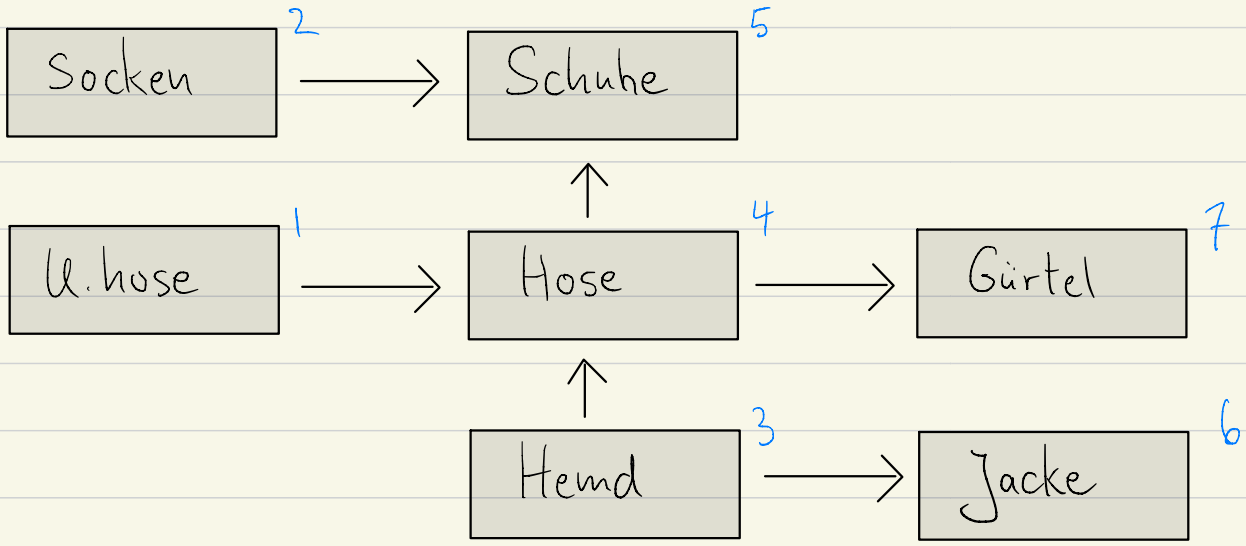
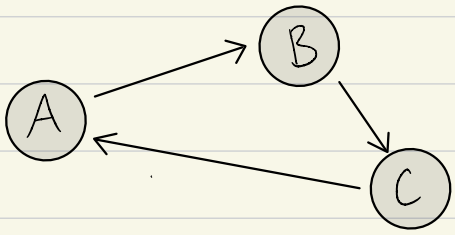
Abhängige Aufgaben planen

Beispiel: Kleidungsstücke anziehen



topologische Reihenfolge: alle Abhängigkeiten erfüllt

immer möglich?



jeder Knoten hat Nachfolger

$\rightsquigarrow$ keine topologische Sortierung möglich

gerichteter Zyklus (Länge 3)

topologisch letzter Knoten darf keine Nachfolger haben

Behauptung: $\exists$ topo. Sort. $\Leftrightarrow \nexists$ ger. Zyklus

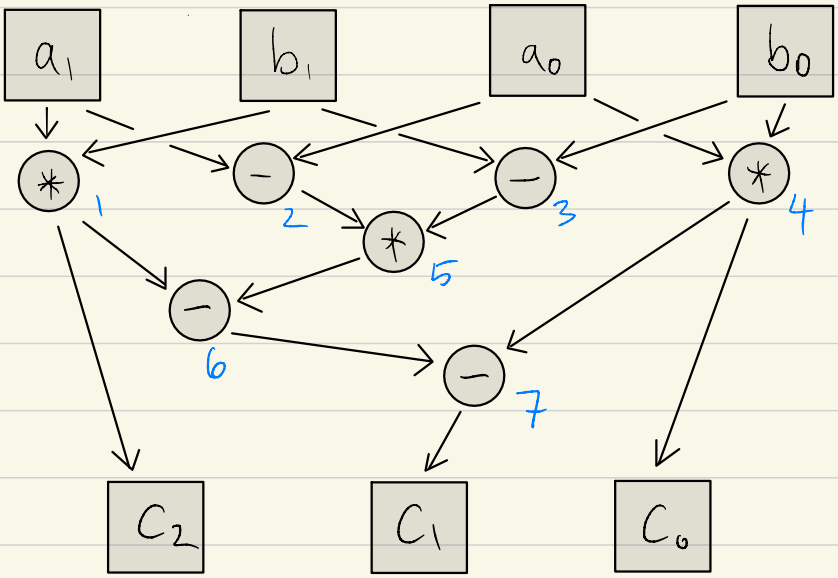
Beweis:

"$\Rightarrow$" s. oben

"$\Leftarrow$" per Alg. (diese Vorlesung)

weiteres Beispiel: Reihenfolge für Berechnungen

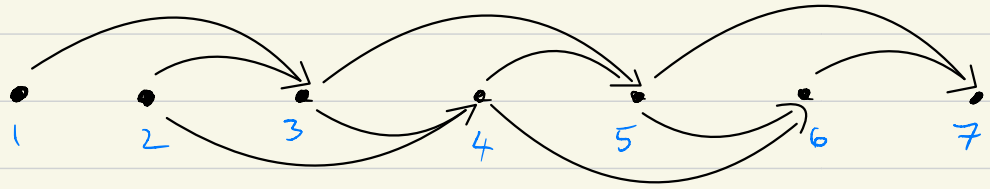
Karatsubas Algorithmus



Rechenoperationen

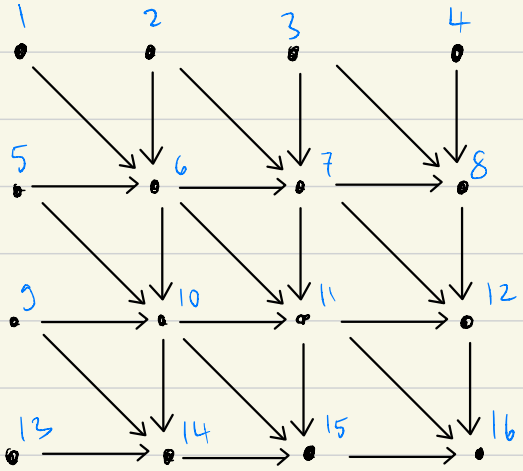
dynamische Programmierung

Fibonacci



$$F_n = F_{n-1} + F_{n-2}$$

längste gemeinsame Teilfolge

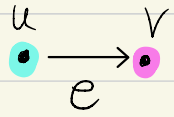


$$T(i,j) = \max\{\, T(i-1,j),\ T(i,j-1),\ T(i-1,j-1) + c_{ij} \,\}$$

Definition: gerichteter Graph $G = (V, E)$

fast wie ungerichteter Graph

ausser: Kanten sind **geordnete Paare**

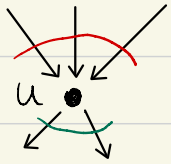


entspricht $e = (u, v) \in E$

Begriffe

v ist Nachfolger von u

u ist Vorgänger von v



$\deg_{in}(u)$ = Eingangsgrad

$\deg_{out}(u)$ = Ausgangsgrad

Quelle u: $\deg_{in}(u) = 0$

Senke v: $\deg_{out}(v) = 0$

gerichteter Weg: $v_0 \longrightarrow v_1 \longrightarrow \cdots \longrightarrow v_{\ell-1} \longrightarrow v_\ell$

——"—— Zyklus: $v_0 = v_\ell$

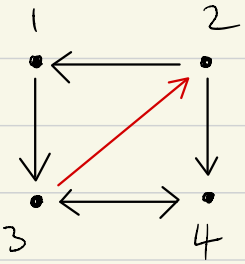
——"—— Pfad: kein Knoten wiederholt

$V = \{1, \dots, n\}$, $G = (V, E)$, $m = |E|$

Adjazenzmatrix: $A = (A_{uv})$

$$A_{uv} = \begin{cases} 1 & \text{falls } (u,v) \in E \\ 0 & \text{sonst} \end{cases}$$

im Speicher als zweidimensionales Array



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

generell: ineffizient
ausser wenn $m \geq \Omega(n^2)$

Adjazenzliste: Array von Listen

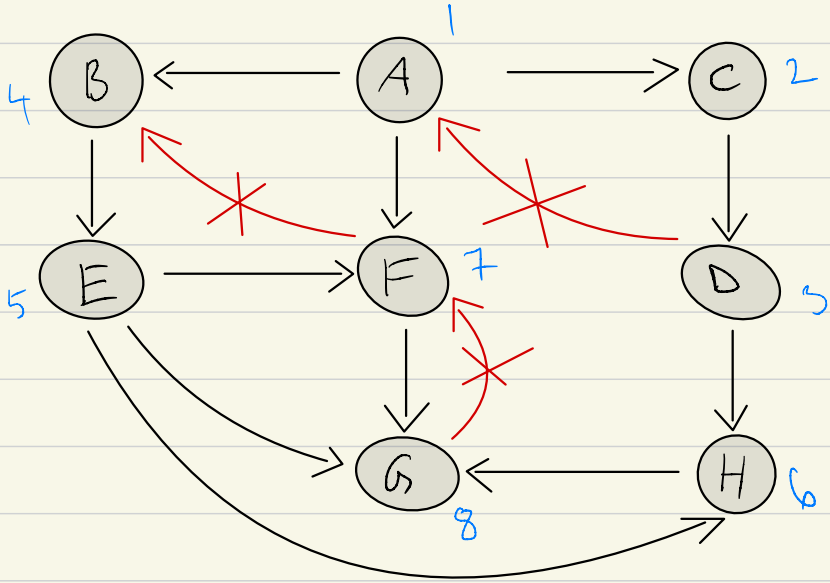
Adj[u] = Liste aller Nachfolger von u Reihenfolge beliebig

u	Adj[u]
1	3
2	1 → 4
3	4 → 2
4	3

im Speicher als verknüpfte Liste

Operation	Laufzeit A.-matrix	Laufzeit A.-liste
teste ob $(u,v) \in E$	$O(1)$	$O(1 + \deg_{out}(u))$
zähle alle Nachfolger von u auf	$O(n)$	$O(1 + \deg_{out}(u))$

Topologisch Sortieren



Ansatz:

- finde Senke v
- setze v an letzte Stelle
- entferne v und löse Rest rekursiv

Wie?

Was wenn ∄ Senke?

werden zeigen (per Alg.):

∄ Zyklus ⟹ ∃ Senke

Folglich: Ansatz funktioniert

(auch in rekursiven Instanzen keine Zyklen)

Path(u): (finde langen Pfad von u, unmarkiert)

markiere u

if ∃ Nachfolger v, unmarkiert

Path(v)

Beispiel: Path(A): A B E F G

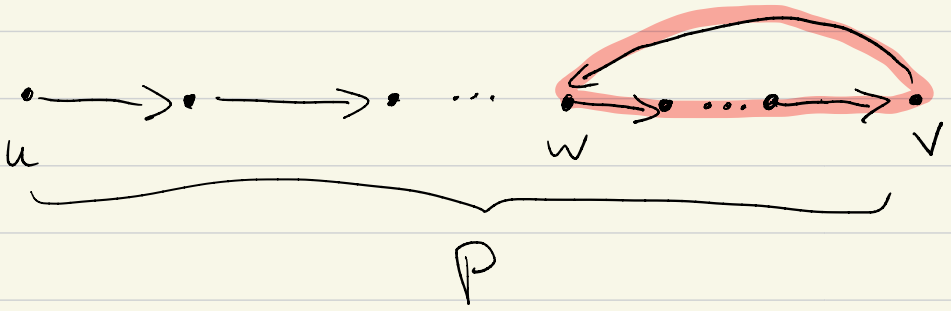
Eigenschaften

(1) Path(u) markiert Pfad P mit Start u

(2) Endknoten v von P hat alle Nachfolger markiert

Behauptung: ∄ ger. Zyklus ⇒ v ist Senke

Beweis (indirekt / kontrapositiv):



angenommen: v nicht Senke

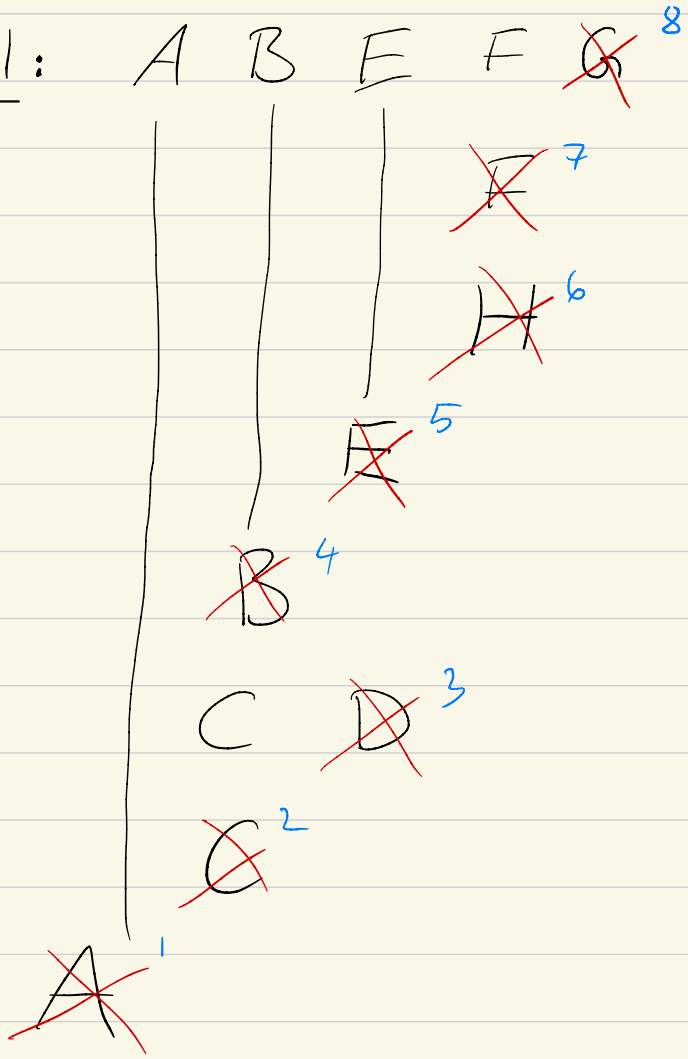
(2) ⇝ v hat markierten Nachfolger w

⇝ ∃ ger. Zyklus ⊠

Schnelle Laufzeit

Idee: anstatt Suche neu zu starten, Pfad wiederverwenden soweit möglich

Beispiel:



rekursive Umsetzung

Visit(u):

markiere u

For Nachfolger v, unmarkiert:

| Visit(v)

Füge u zur top. Sort. hinzu

DFS(G): (Tiefensuche, Rahmenprogramm)

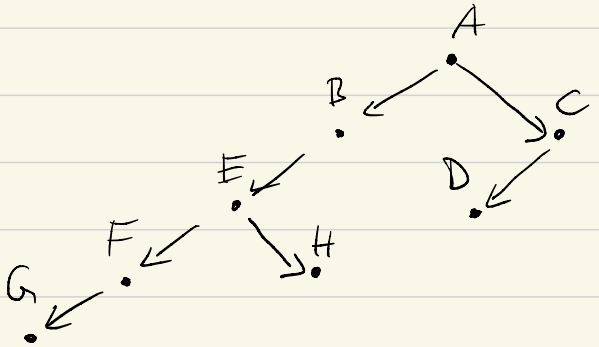
alle Knoten unmarkiert

For $u_0 \in V$, unmarkiert:

Visit(u_0)

Beispiel:

Rekursionsbaum (oft: Tiefensuchbaum/-wald)



Laufzeit analyse

Adjazenzmatrix

Anzahl Visit Aufrufe: n — einen Aufruf pro Knoten

Zeit für Visit(u): $O(n)$ + Zeit in rekursiven Aufrufen — laufe über alle Nachfolger von u

Total: $n \cdot O(n) = O(n^2)$

Adjazenzliste

Zeit für Visit(u): $O(1 + \deg_{out}(u)) +$ Rekursion

Total:

$$\sum_{u \in V} (1 + \deg_{out}(u)) = |V| + \underbrace{\sum_{u \in V} \deg_{out}(u)}_{|E|}$$

Handschlaglemma

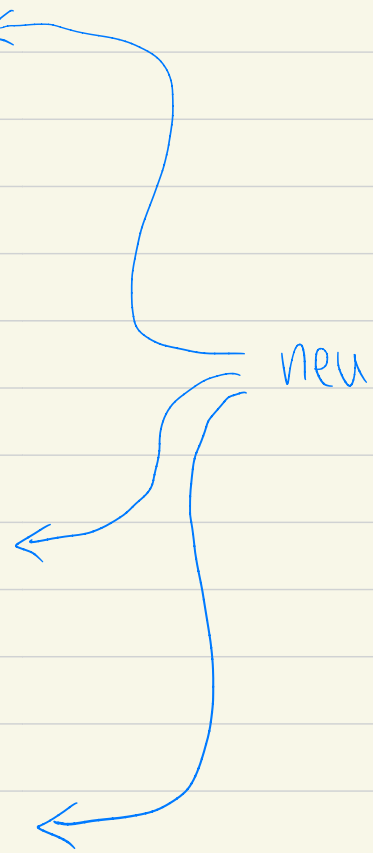
$\rightsquigarrow$ Tiefensuche hat Laufzeit $O(|V| + |E|)$ für Adjazenzliste

Tiefensuche tiefer verstehen

depth-first search (DFS)

Erinnerung

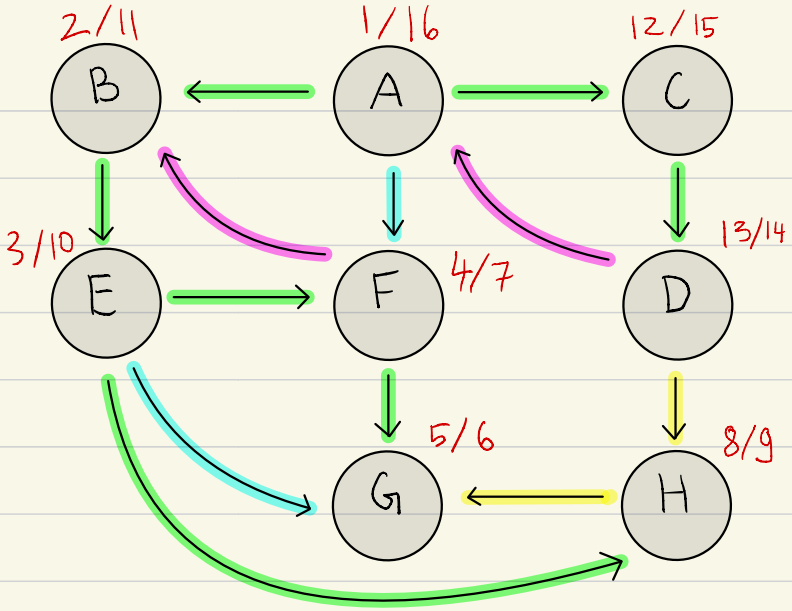
```
Visit(u):
    pre[u] ← T; T ← T+1
    markiere u
    FOR Nachfolger v, unmarkiert
        Visit(u)
    post[u] ← T; T ← T+1

DFS(G):
    T ← 1
    alle Knoten unmarkiert
    FOR u₀ ∈ V, unmarkiert
        Visit(u₀)
```



Neu: merke Start- und Endzeitpunkte aller Visit-Aufrufe

$\rightsquigarrow$ pre / post Zahlen

Beispiel: pre/post



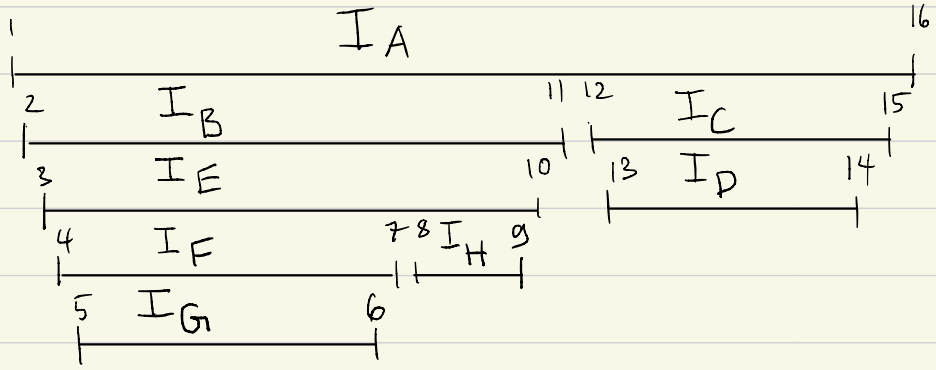
pre-order: ABEFGHCD

post-order: GFHEBDCB

vorherige Vorlesung:

falls G azyklisch, dann ist umgekehrte post-order eine topologische Sortierung

Bildlich: Intervall $I_u = \{ pre[u], \dots, post[u] \}$

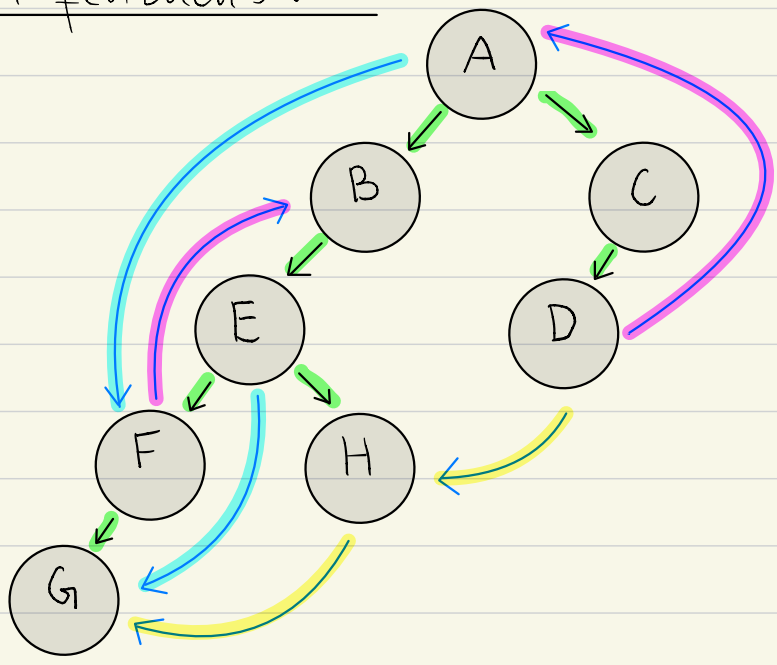


unmassstäblich

Verschachtelung entspricht Rekursionsbaum

Tiefensuchbaum

eigentlich: Tiefensuchwald

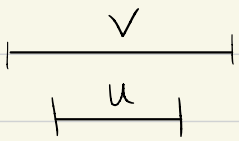


Kanten klassifizieren

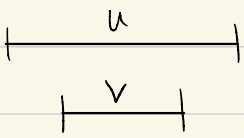
im Tiefensuchbaum: tree

jede andere Kante $(u,v) \in E$: anhand I_u und I_v

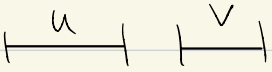
(pre/post Zahlen von u und v)



back z.B.: (F,B), (D,A)

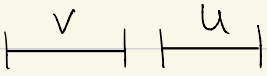


forward (wenn nicht tree) z.B.: (A,F) (E,G)

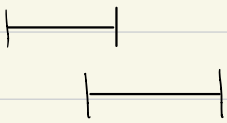


nicht möglich!

Visit(u) würde Visit(v) aufrufen da v unmarkierter Nachfolger von u



cross z.B.: (H,G), (D,H)



nicht möglich!

Aufruf endet erst wenn alle rekursiven Aufrufe beendet

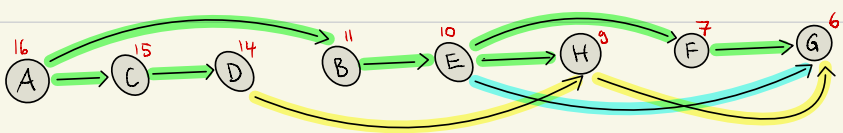
Beobachtungen

(1) $\exists$ back Kante $\Rightarrow$ $\exists$ gerichteter Zyklus

(2) $\forall$ nicht-back $(u,v) \in E$. post(u) > post(v)

(1),(2) $\rightsquigarrow$ $\nexists$ gerichteter Zyklus $\overset{(1)}{\Rightarrow}$ $\nexists$ back Kante

$\overset{(2)}{\Rightarrow}$ umgekehrte post-order ist topologische Sortierung



Kurzer Korrektheitsbeweis für DFS zum topologischen Sortieren

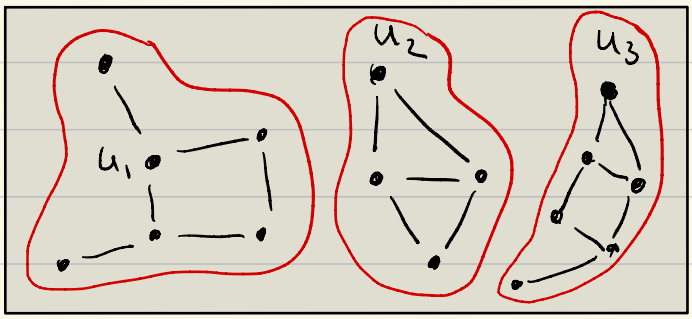
DFS auf ungerichteten Graphen

back = forward Kanten (umgekehrt durchlaufen)

cross Kanten nicht möglich

Zusammenhangskomponenten

G



DFS Wald