Grundlagen der Algorithmen und Datenstrukturen

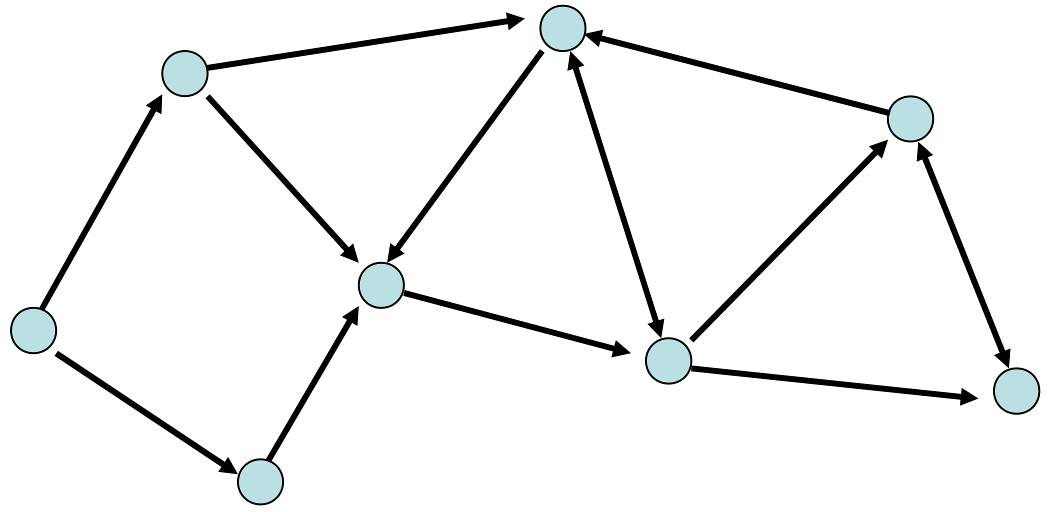
Kapitel 9

Christian Scheideler + Helmut Seidl

SS 2009

Graphdurchlauf

Zentrale Frage: Wie können wir die Knoten eines Graphen durchlaufen, so dass jeder Knoten mindestens einmal besucht wird?



Graphdurchlauf

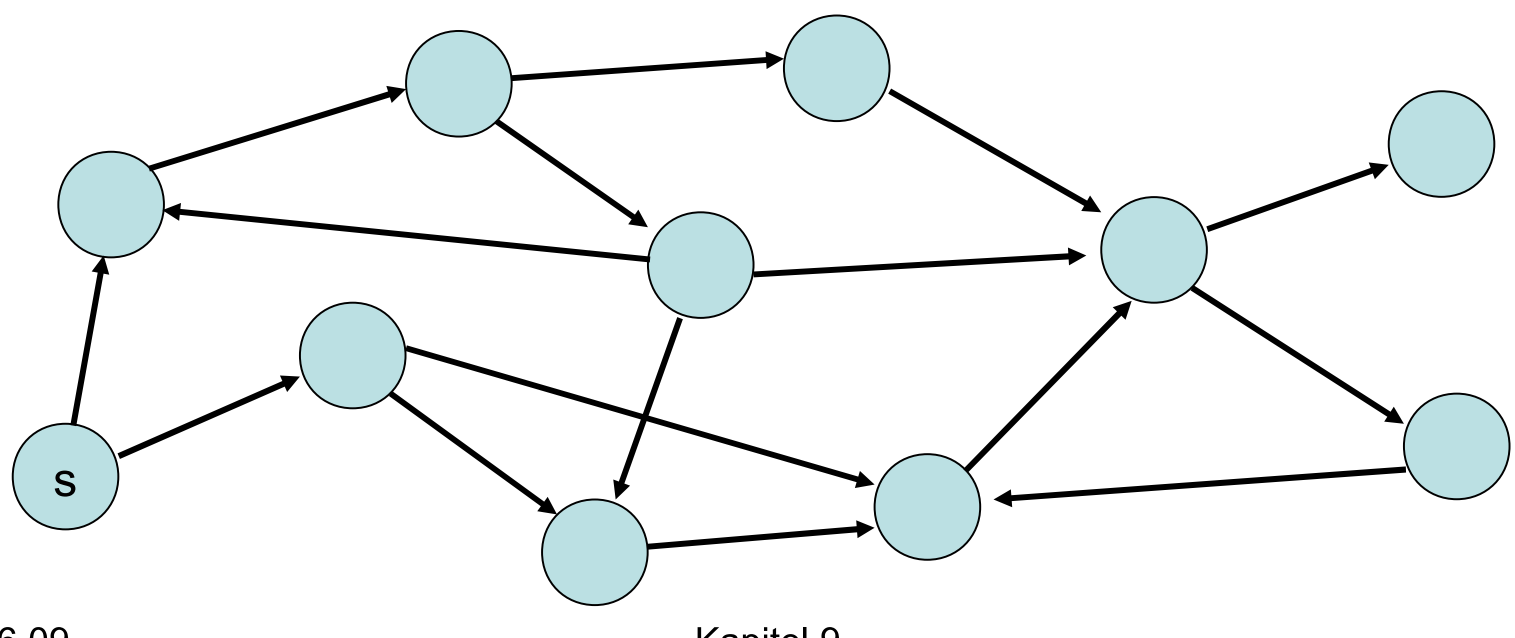
Zentrale Frage: Wie können wir die Knoten eines Graphen durchlaufen, so dass jeder Knoten mindestens einmal besucht wird?

Grundlegende Strategien:

- Breitensuche
- Tiefensuche

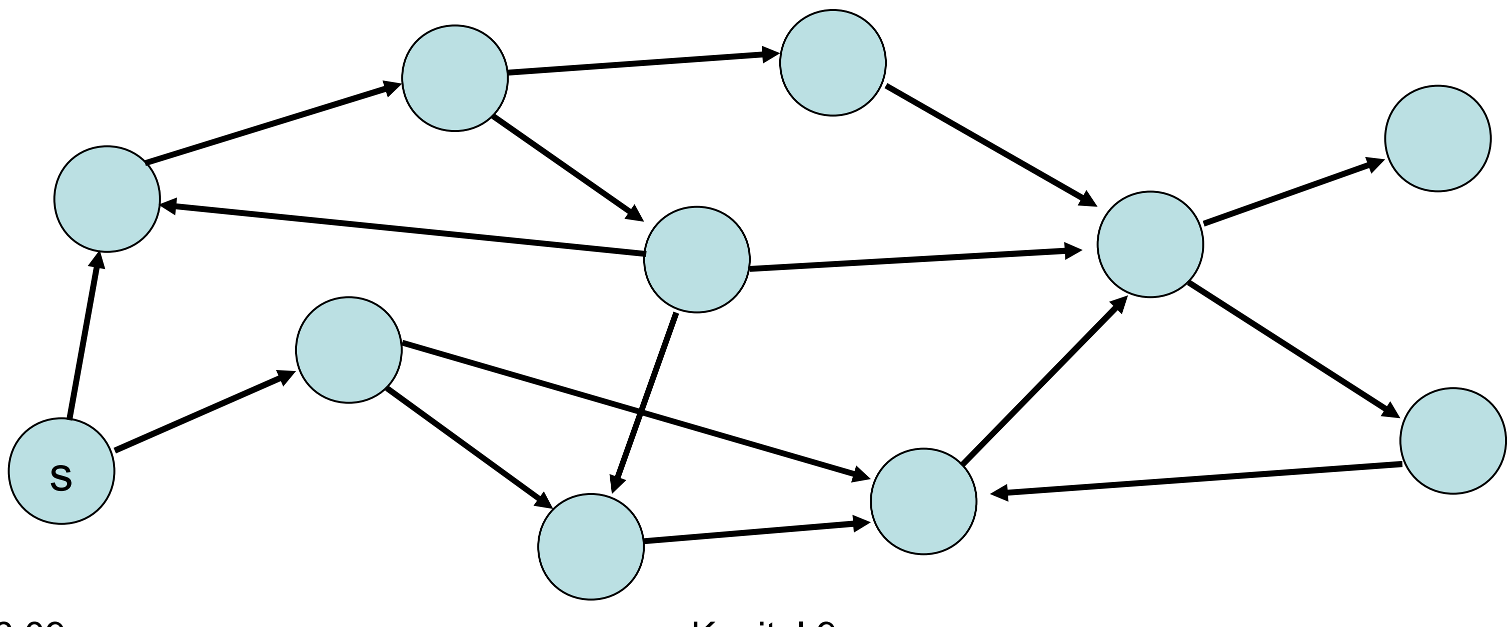
Breitensuche

- Starte von einem Knoten s
- Exploriere Graph Distanz für Distanz



Tiefensuche

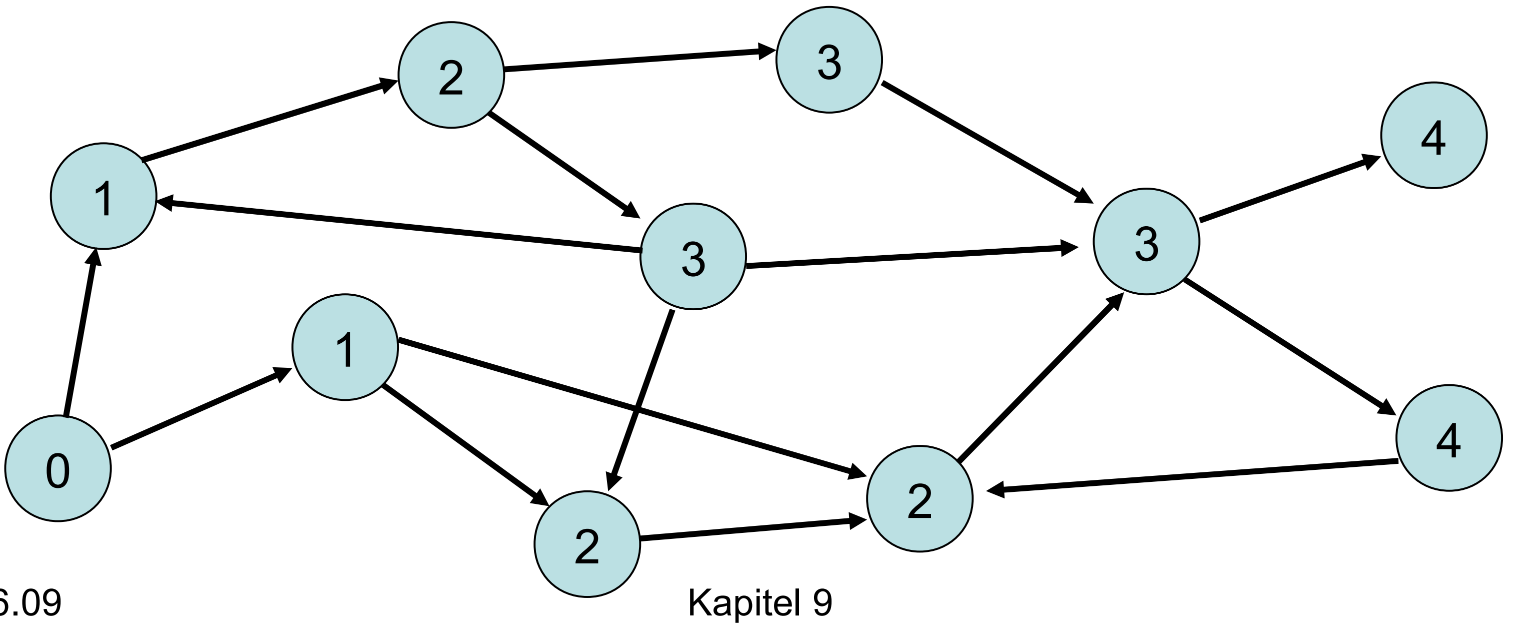
- Starte von einem Knoten s
- Exploriere Graph in die Tiefe
  (: aktuell, : noch aktiv, : fertig)



Breitensuche

- d(v): Distanz von Knoten v zu s (d(s)==0)
- parent(v): Knoten, von dem v besucht

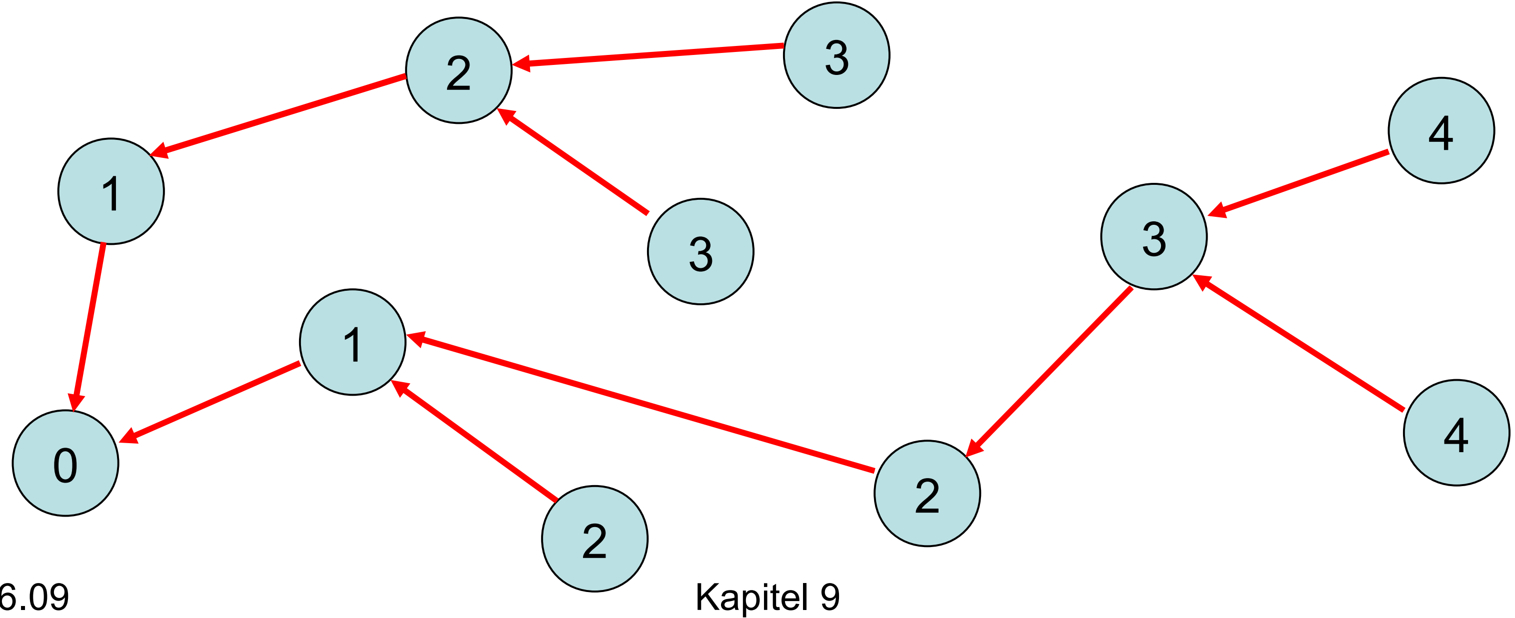
Distanzen:



Breitensuche

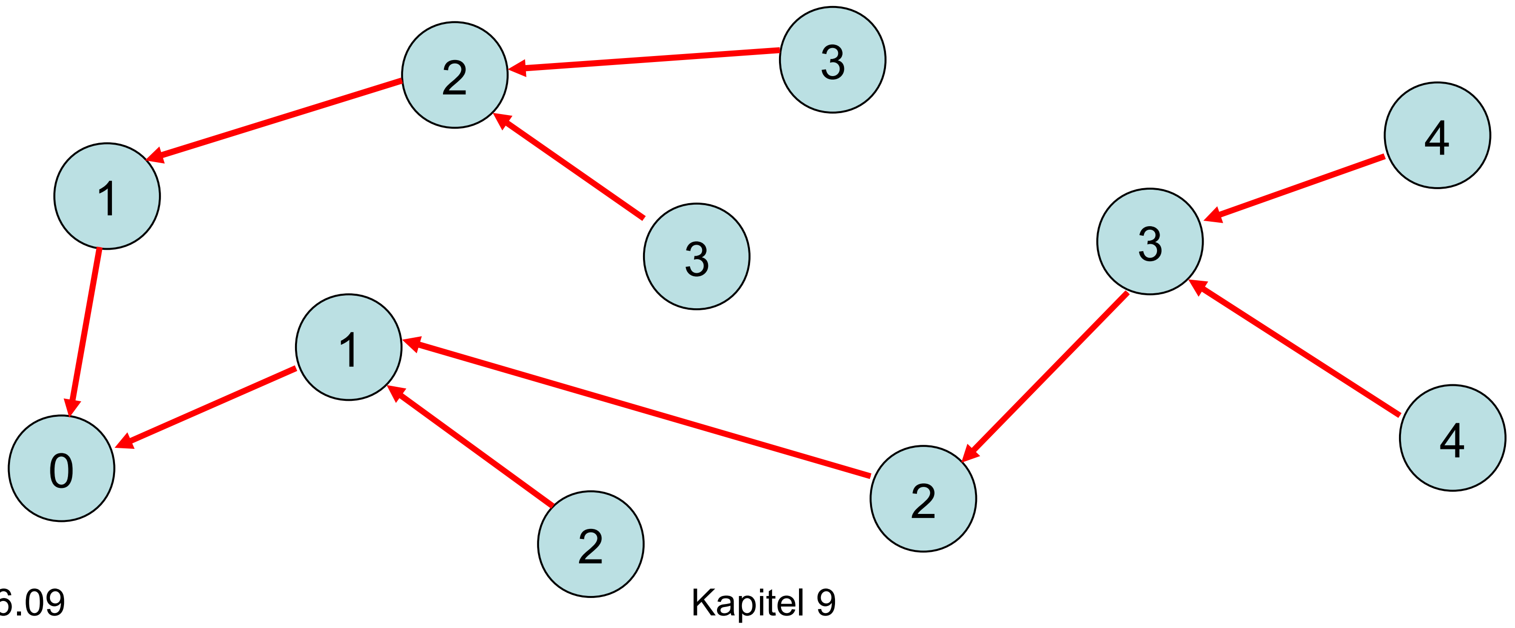
- d(v): Distanz von Knoten v zu s (d(s)=0)
- parent(v): Knoten, von dem v besucht

Mögliche Parent-Beziehungen in rot:



Breitensuche

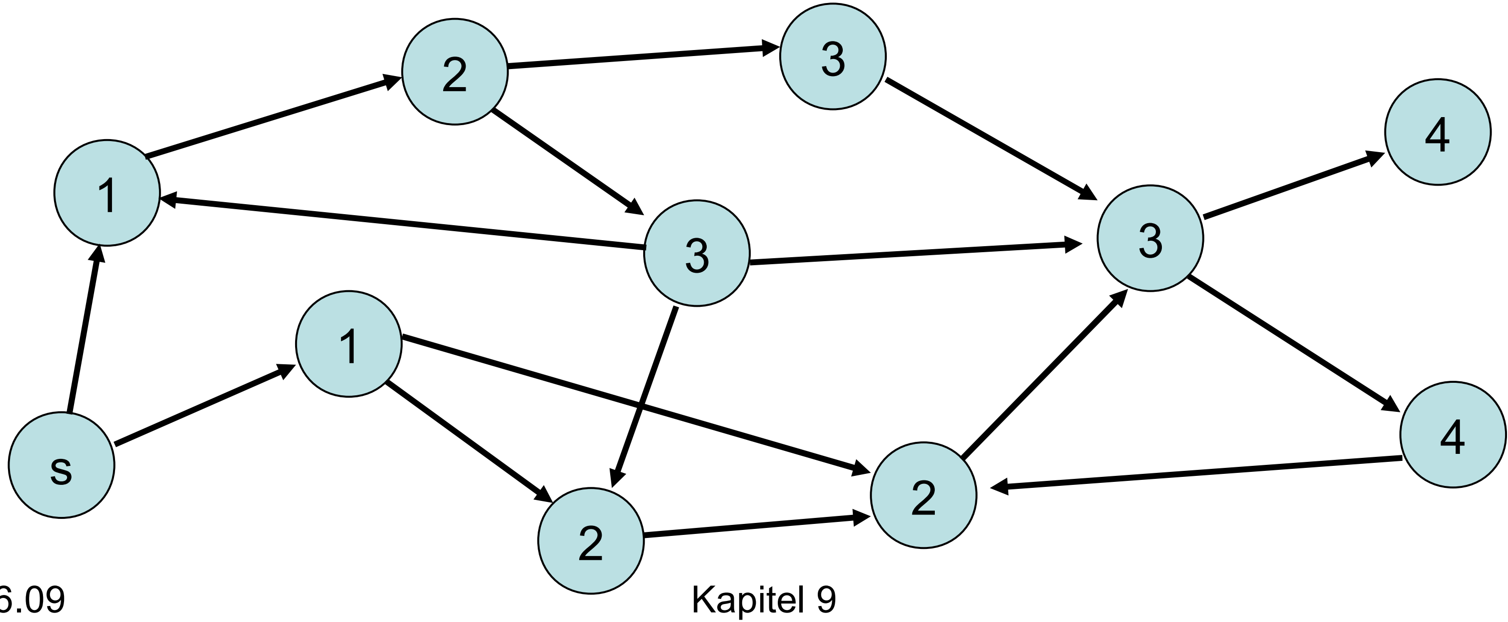
Parent-Beziehung eindeutig: wenn Knoten v zum erstenmal besucht wird, wird parent(v) gesetzt und v markiert, so dass v nicht nochmal besucht wird



Breitensuche

Kantentypen:

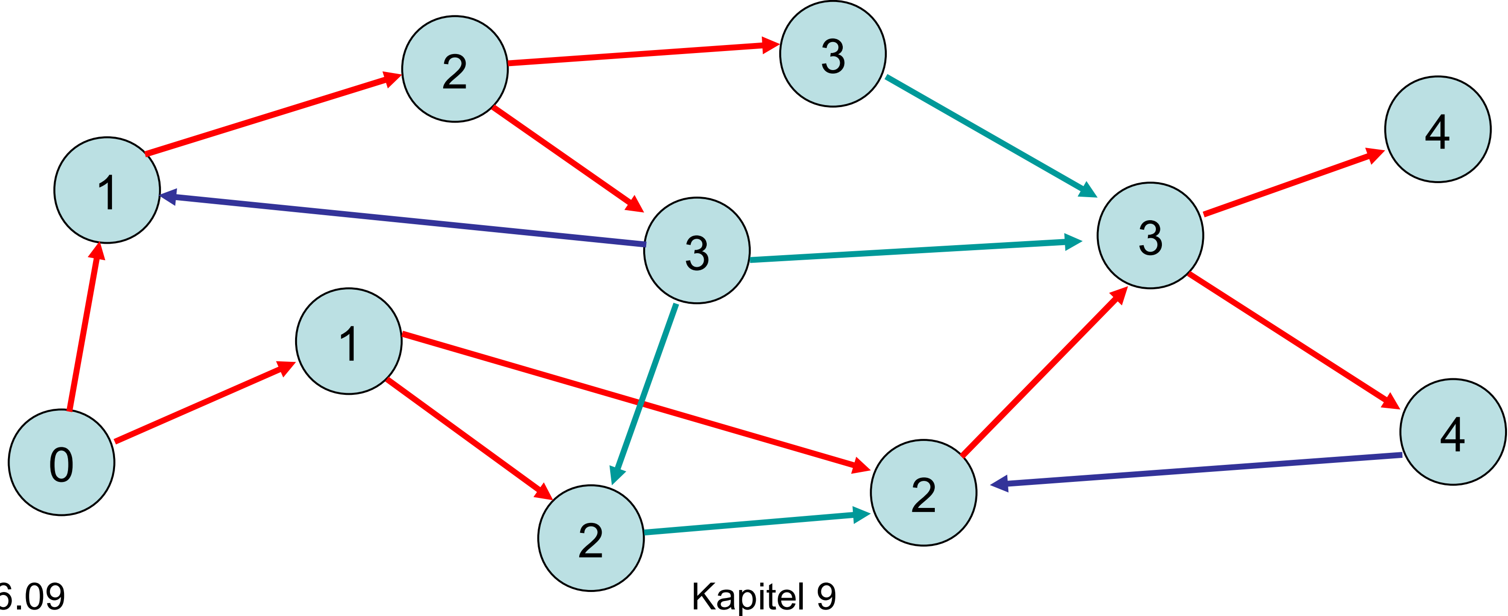
- Baumkante: zum Kind
- Rückwärtskante: zu einem Vorfahr
- Kreuzkante: alle sonstige Kanten



Breitensuche

Kantentypen:

- Baumkante: zum Kind
- Rückwärtskante: zu einem Vorfahr
- Kreuzkante: alle sonstige Kanten



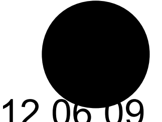
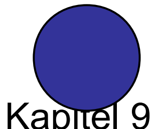
Breitensuche

```
int [] d = new int [n];
Node [] parent = new Node [n];
void BFS(Node s) {
  d[s] = 0;          // s hat Distanz 0 zu sich
  parent[s] = s;     // s ist sein eigener Vater
  List<Node> q = <s>;  // q:Queue zu besuchender Knoten
  while (q != <>) {      // solange q nicht leer
    u = q.popFront();  // nimm Knoten nach FIFO-Regel
    foreach ((u,v) ∈ E)
      if (parent[v]==null) {  // v schon besucht?
        q.pushBack(v);  // nein, dann in q hinten einfügen
        d[v] = d[u]+1;
        parent[v] = u;
      }
}}
```

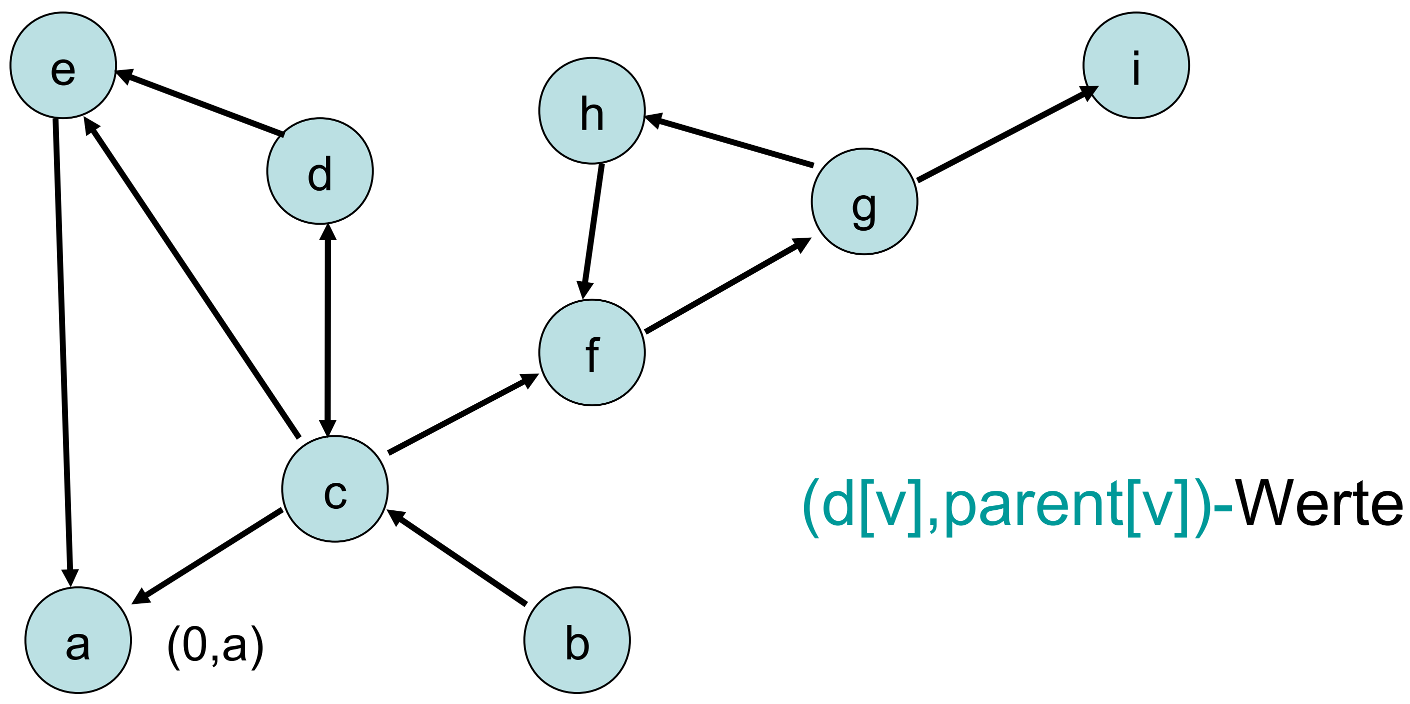
BFS(b)

(2,c) e
(2,c) d
h (4,g)
i (4,g)
g (3,f)
f (2,c)
c (1,b)
(2,c) a
b (0,b)

(d[v],parent[v])-Werte

 : besucht, noch in q  : besucht, nicht mehr in q

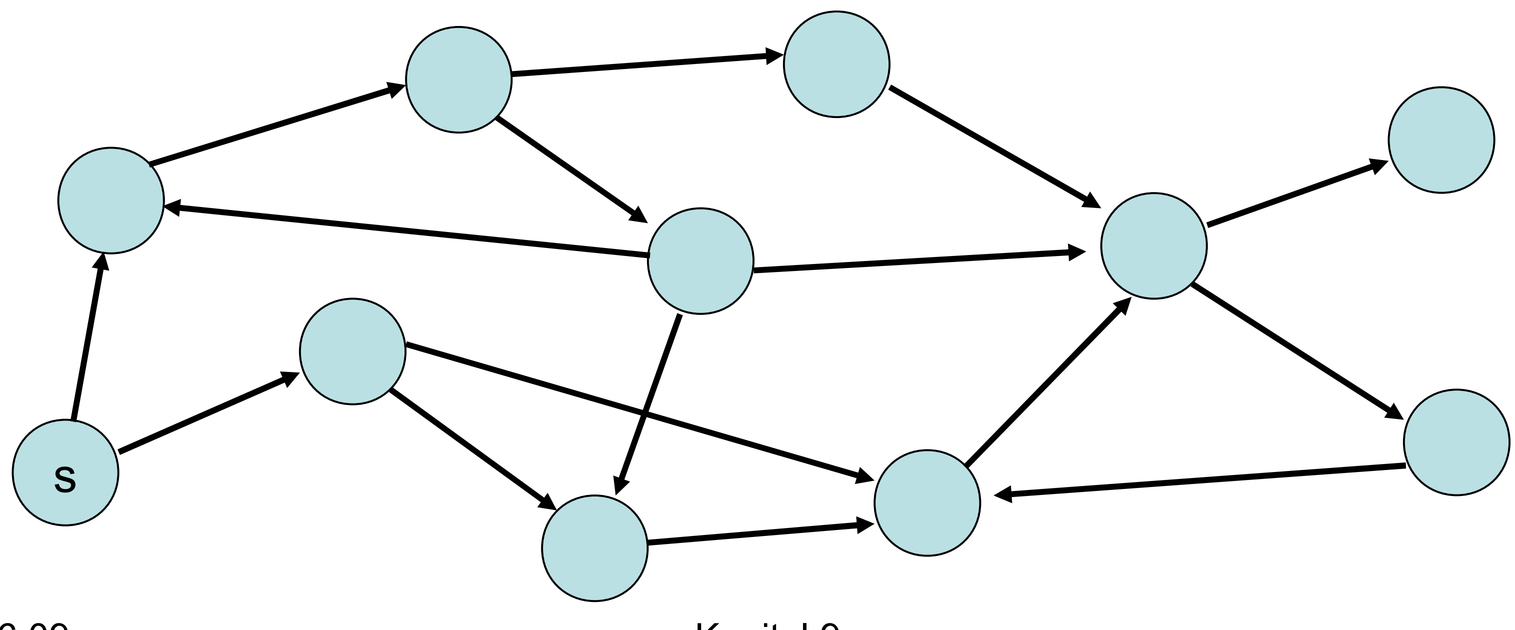
BFS(a)



Von a kein anderer Knoten erreichbar.

Tiefensuche

- Starte von einem Knoten s
- Exploriere Graph in die Tiefe
  (: aktuell, : noch aktiv, : fertig)



Tiefensuche - Schema

Übergeordnete Methode:

```
unmark_all_nodes();
init();
foreach (s∈V) {  // stelle sicher, dass alle Knoten besucht werden
   if (!is_marked(s)) {
      mark(s);
      root(s);
      DFS(s,s);    // s: Startknoten
}
void DFS(Node u, Node v) {  // u: Vater von v
   foreach ((v,w)∈E)
      if (is_marked(w)) traverseNonTreeEdge(v,w);
      else { traverseTreeEdge(v,w);
               mark(w);
               DFS(v,w);
       }
   backtrack(u,v);
}
```

Methoden in rot: noch zu spezifizieren

DFS-Nummerierung

Variablen:

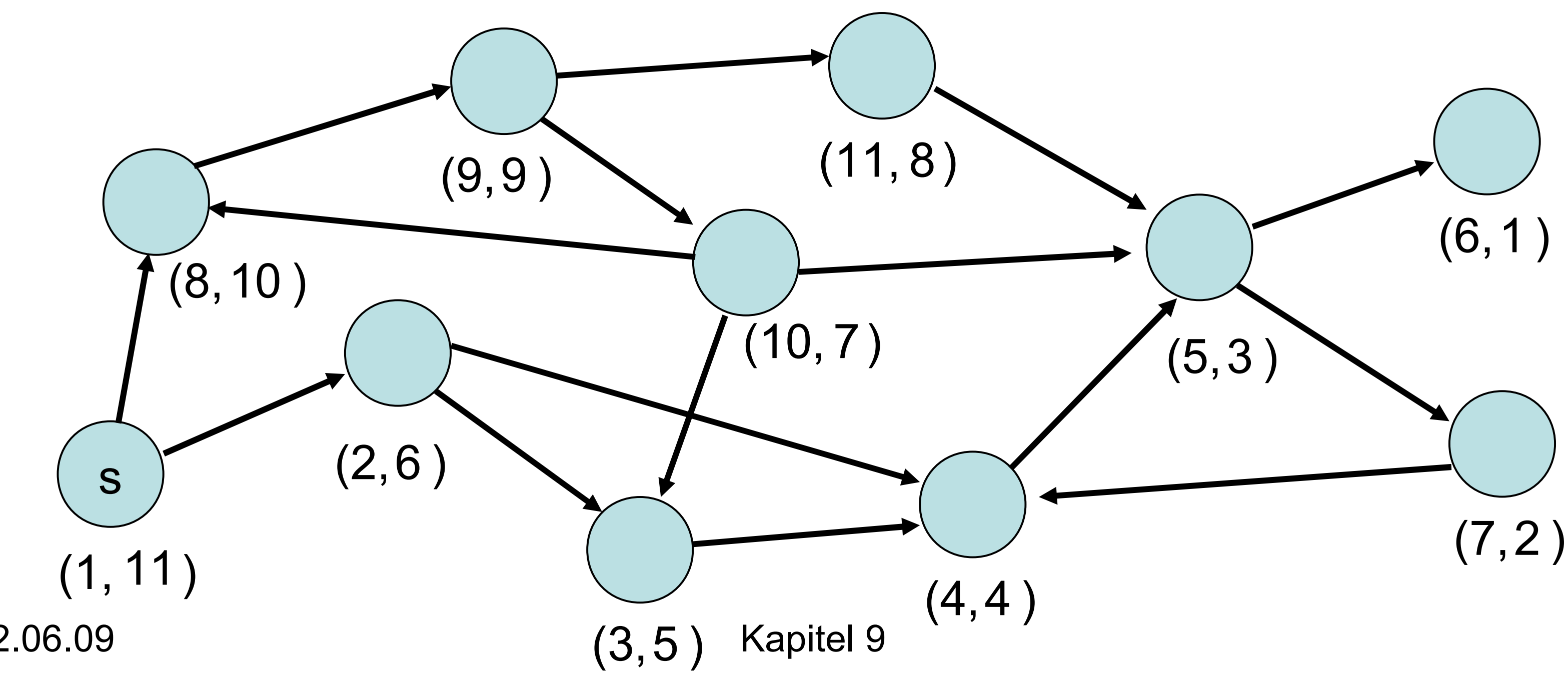
- int [] dfsNum; // Zeitpunkt wenn Knoten erstmals
- int [] finishTime; // Zeitpunkt wenn Knoten →
- int dfsPos, finishingTime; // Zähler

Methoden:

- void init() {
  dfsPos = 1; finishingTime = 1;}
- void root(Node s) {
  dfsNum[s] = dfsPos; dfsPos = dfsPos+1;}
- void traverseTreeEdge(Node v, Node w) {
  dfsNum[w]:=dfsPos; dfsPos = dfsPos+1;}
- void traverseNonTreeEdge(Node v, Node w) { }
- backtrack(Node u, Node v) {
  finishTime[v] = finishingTime; finishingTime =finishingTime+1;}

DFS-Nummerierung

- Exploriere Graph in die Tiefe
  (●: aktuell, ●: noch aktiv, ●: fertig)
- Paare (i,j): i: dfsNum, j: finishTime



DFS-Nummerierung

Ordnung $<$ auf den Knoten:

$u<v \Leftrightarrow dfsNum[u]<dfsNum[v]$

Lemma 9.1: Die Knoten im DFS-Rekursionsstack (alle ● Knoten) sind sortiert bezüglich $<$.

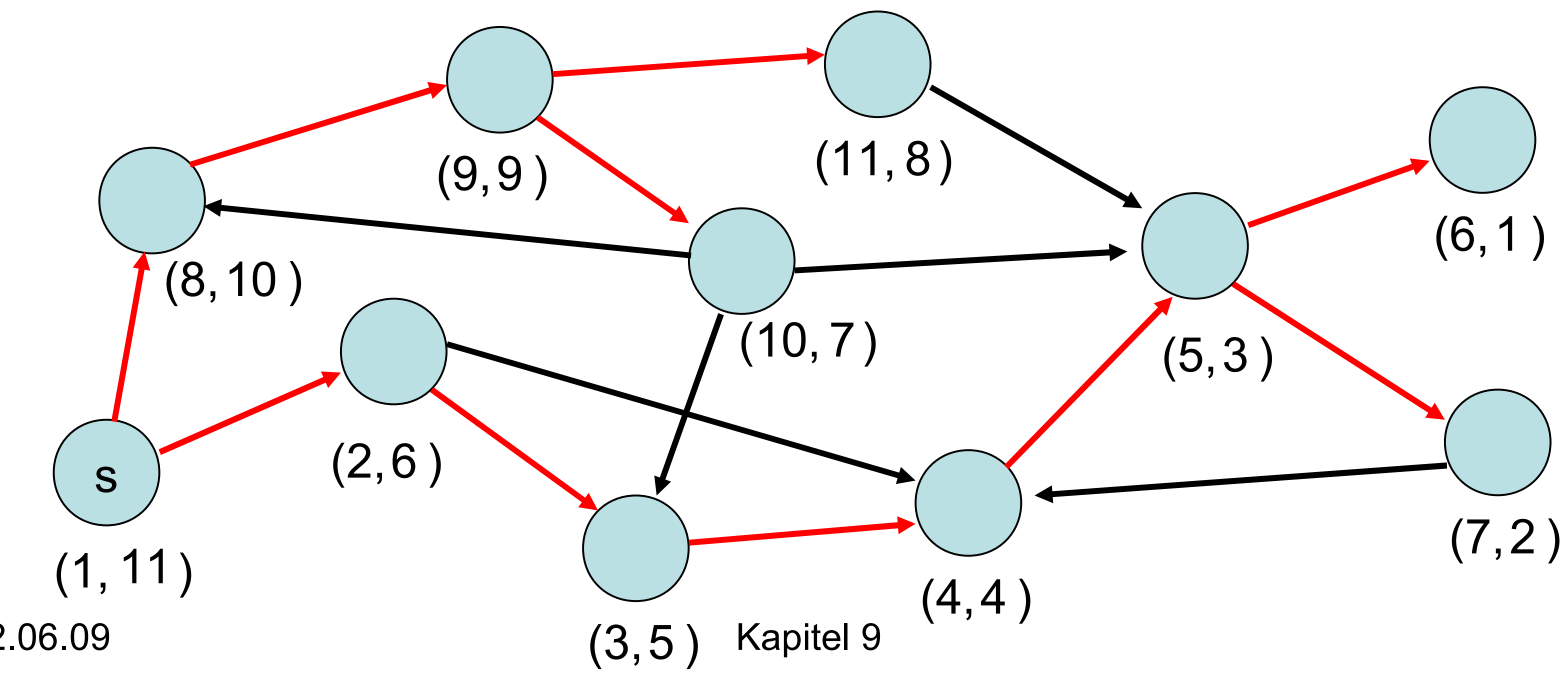
Beweis:

dfsPos wird nach jeder Zuweisung von dfsNum erhöht. Jeder neue aktive Knoten hat also immer die höchste dfsNum.

DFS-Nummerierung

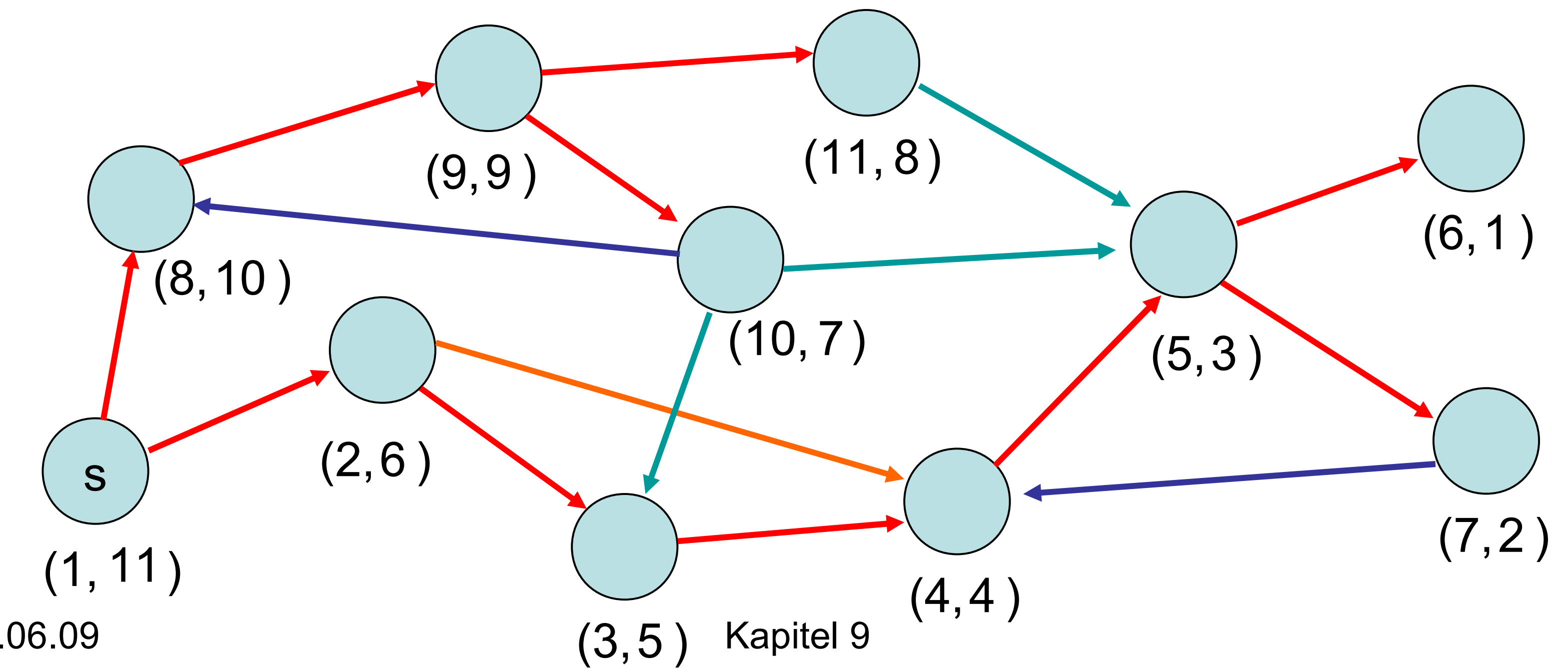
Überprüfung von Lemma 9.1:

- Rekursionsstack: roter Pfad von s
- Paare (i,j): i: dfsNum, j: finishTime



DFS-Nummerierung

- Baumkante: zum Kind
- Vorwärtskante: zu einem Nachkommen
- Rückwärtskante: zu einem Vorfahr
- Kreuzkante: alle sonstigen Kanten



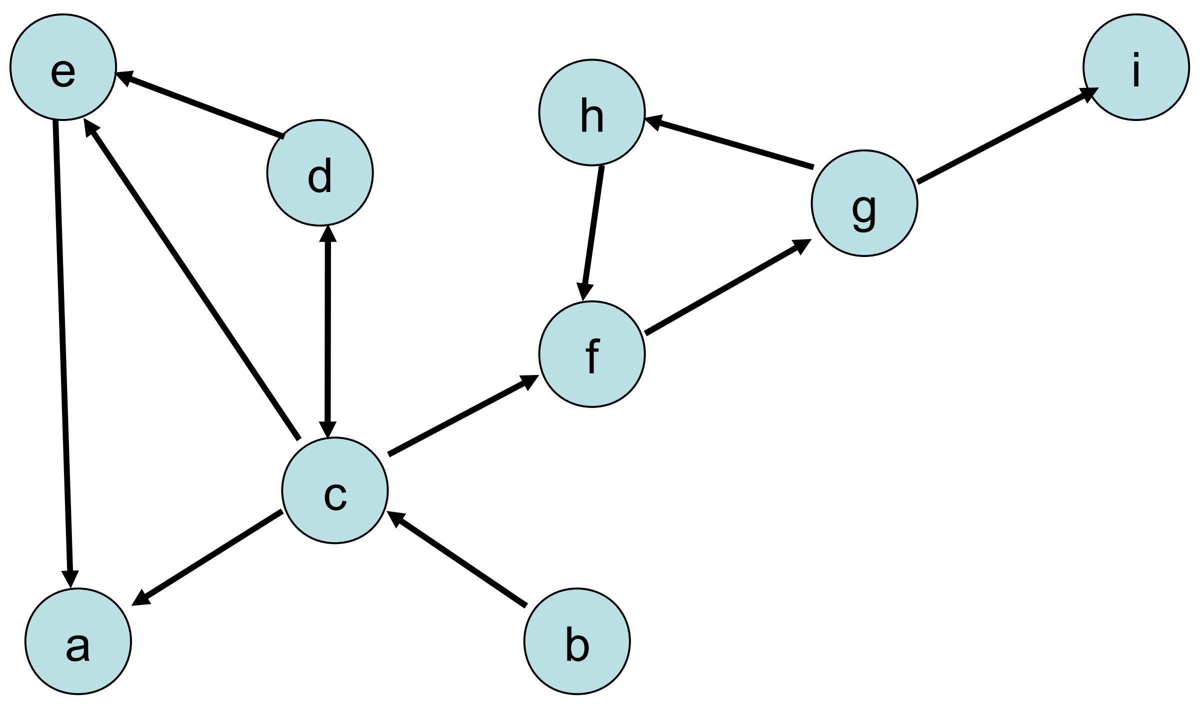
DFS-Nummerierung

Beobachtung für Kante (v,w):

Kantentyp	$dfsNum[v] < dfsNum[w]$	$finishTime[v] > finishTime[w]$
Baum & Vorwärts	Ja	Ja
Rückwärts	Nein	Nein
Kreuz	Nein	Ja

DFS-Nummerierung

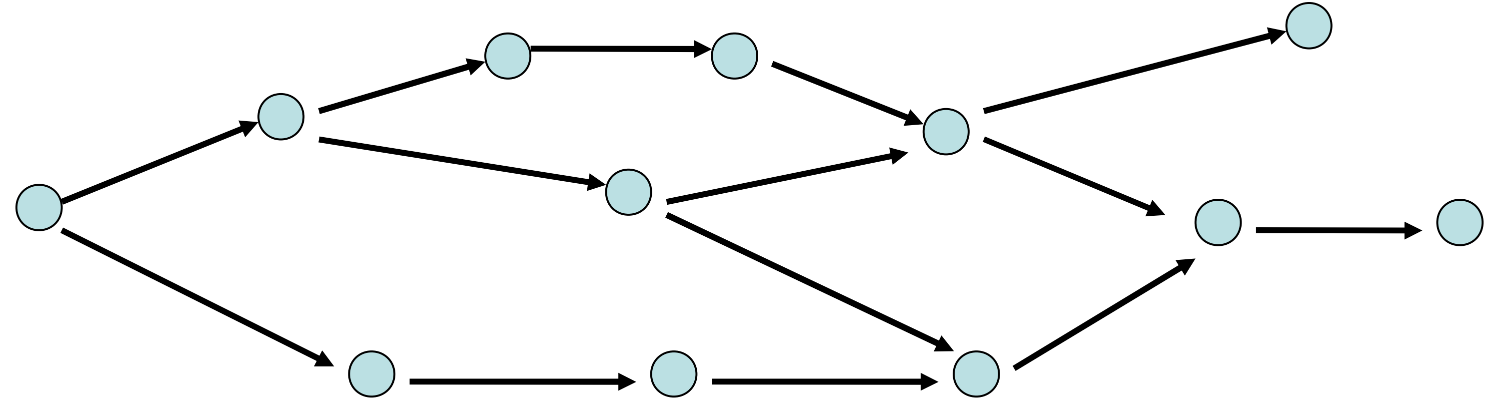
Beispiel auf der Folie:



DFS-Nummerierung

Anwendung:

- Erkennung eines azyklischen gerichteten Graphen (engl. DAG)



Merkmal: keine gerichteten Kreise

DFS-Nummerierung

Lemma 9.3: Das Folgende ist äquivalent:

1. G ist ein DAG
2. DFS enthält keine Rückwärtskante
3. $\forall$ (v,w)$\in$E: finishTime[v]>finishTime[w]

Beweis:

2. $\Leftrightarrow$ 3.: folgt aus Tabelle

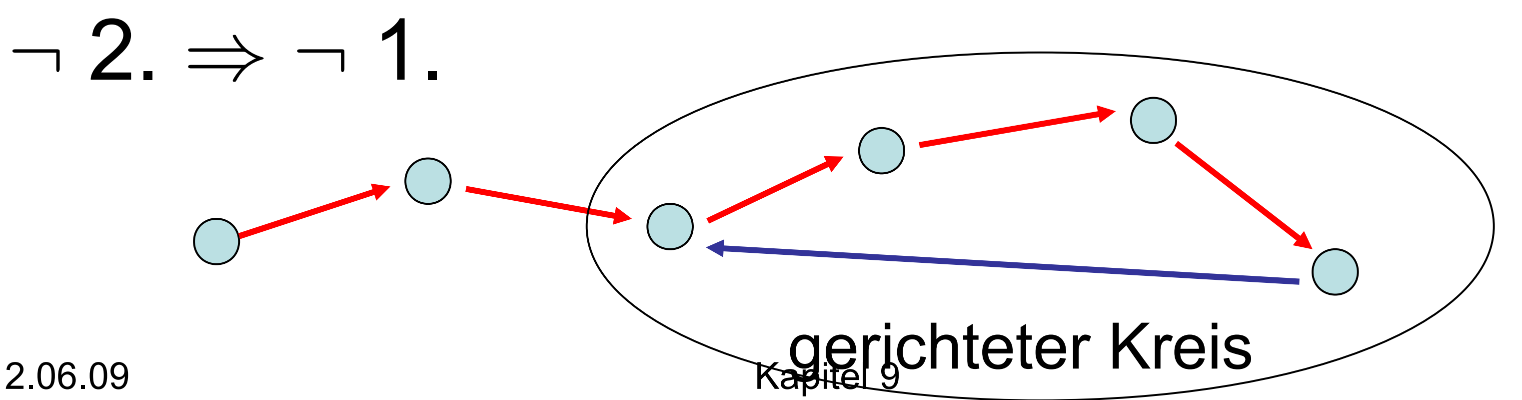
DFS-Nummerierung

Lemma 9.3: Das Folgende ist äquivalent:

1. G ist ein DAG
2. DFS enthält keine Rückwärtskante
3. $\forall$ (v,w)$\in$E: finishTime[v]>finishTime[w]

Beweis:

$\neg$ 2. $\Rightarrow$ $\neg$ 1.



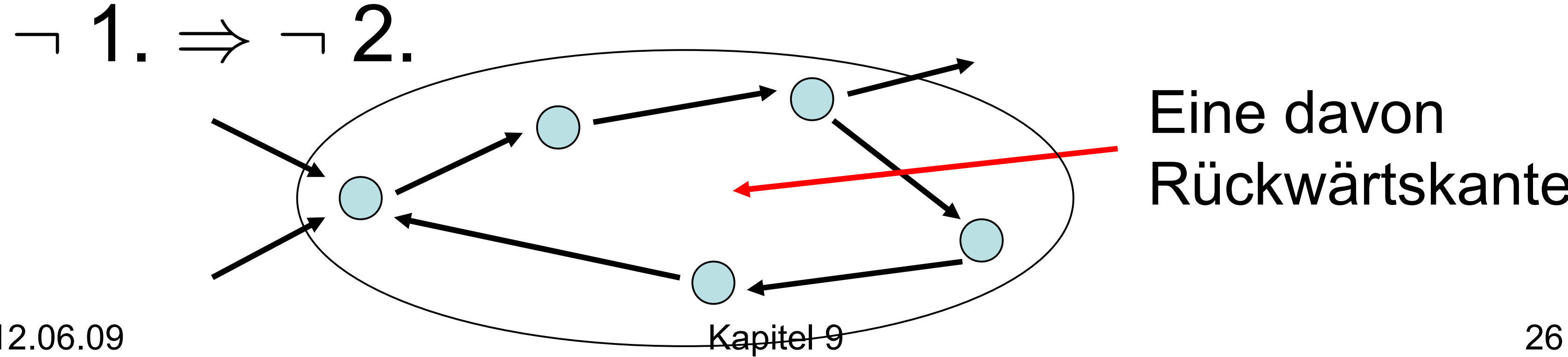
DFS-Nummerierung

Lemma 9.3: Das Folgende ist äquivalent:

1. G ist ein DAG
2. DFS enthält keine Rückwärtskante
3. $\forall (v,w) \in E$: finishTime[v]>finishTime[w]

Beweis:

$\neg$ 1. $\Rightarrow$ $\neg$ 2.



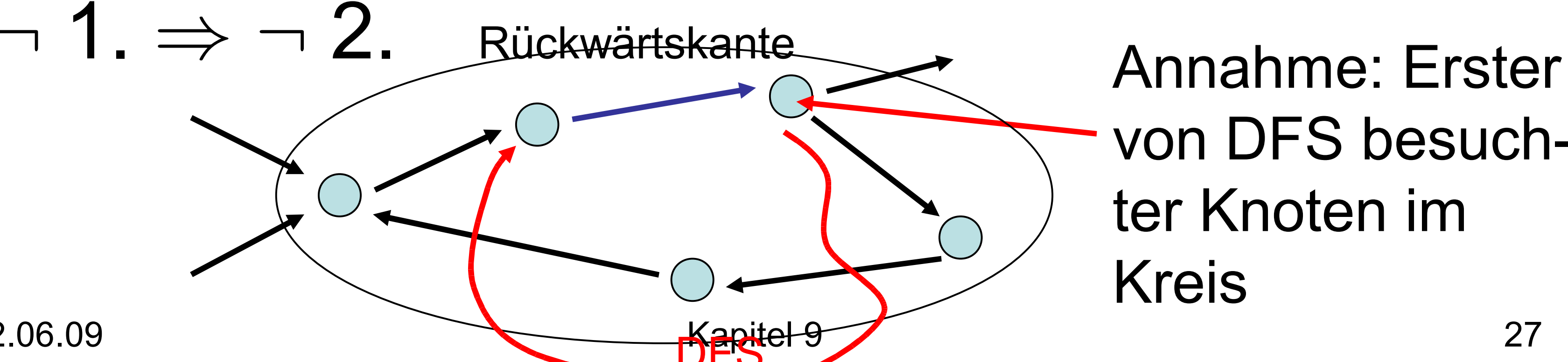
DFS-Nummerierung

Lemma 9.3: Das Folgende ist äquivalent:

1. G ist ein DAG
2. DFS enthält keine Rückwärtskante
3. $\forall$ (v,w)$\in$E: finishTime[v]>finishTime[w]

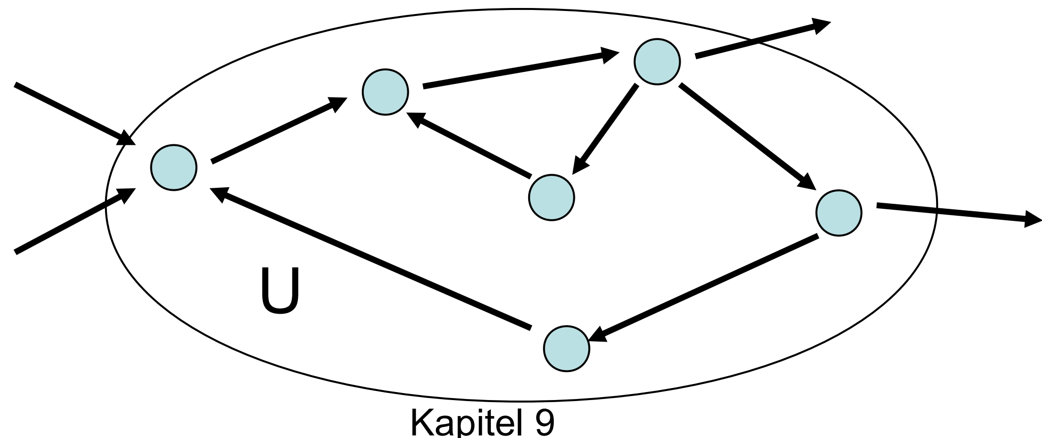
Beweis:

$\neg$ 1. $\Rightarrow$ $\neg$ 2.



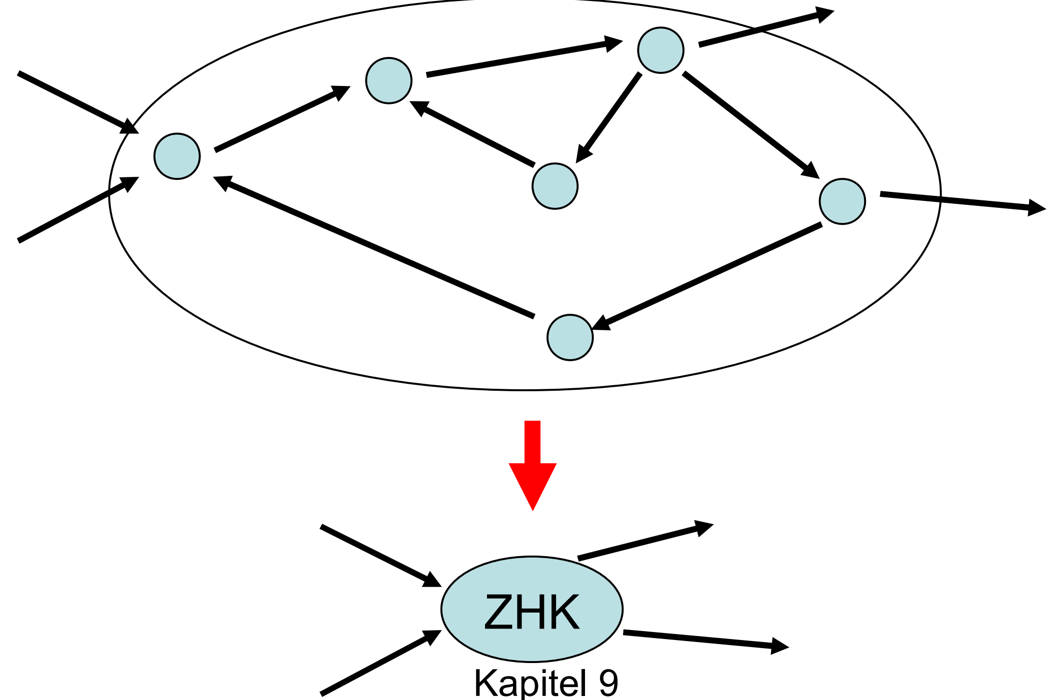
Starke ZHKs

Definition: Sei $G=(V,E)$ ein gerichteter Graph. $U \subset V$ ist eine **starke Zusammenhangskomponente** (ZHK) von V $\Leftrightarrow$ für alle $u,v \in U$ gibt es einen gerichteten Weg von u nach v in G und U maximal

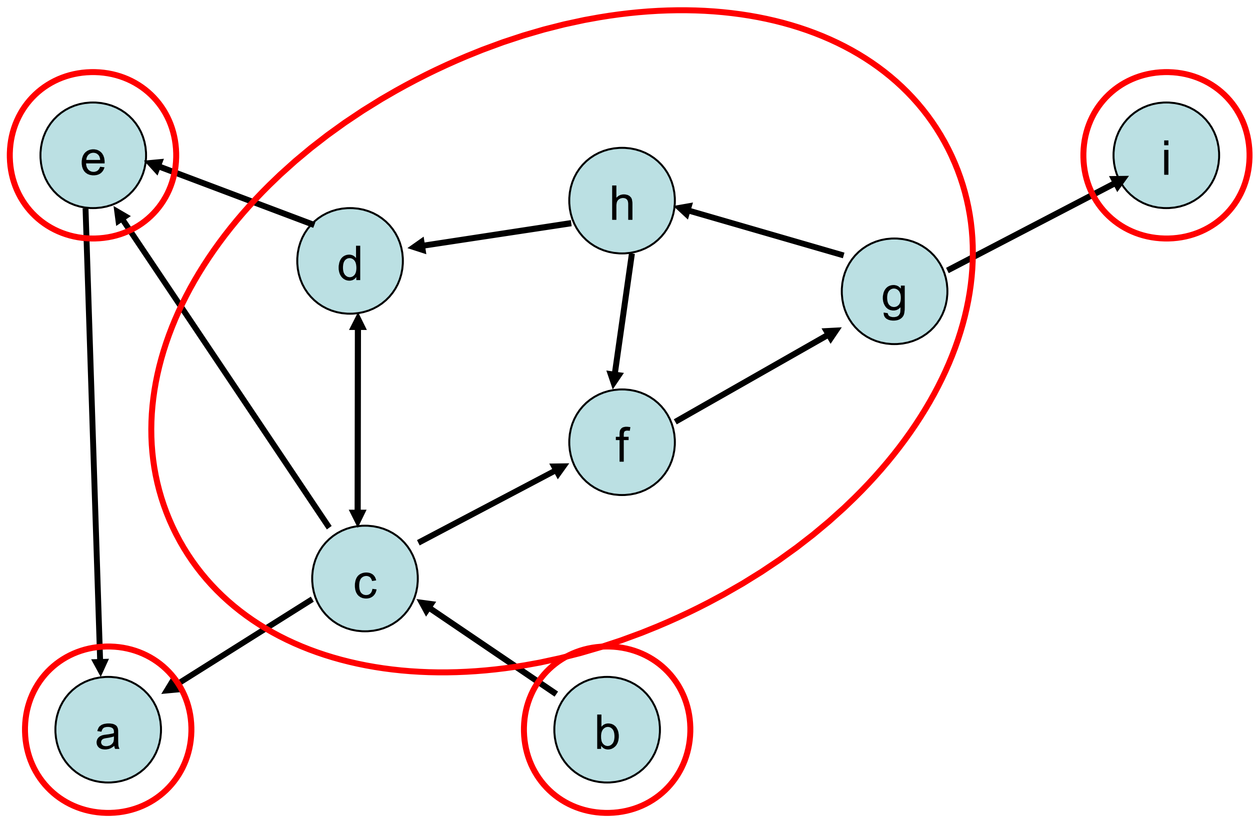


Starke ZHKs

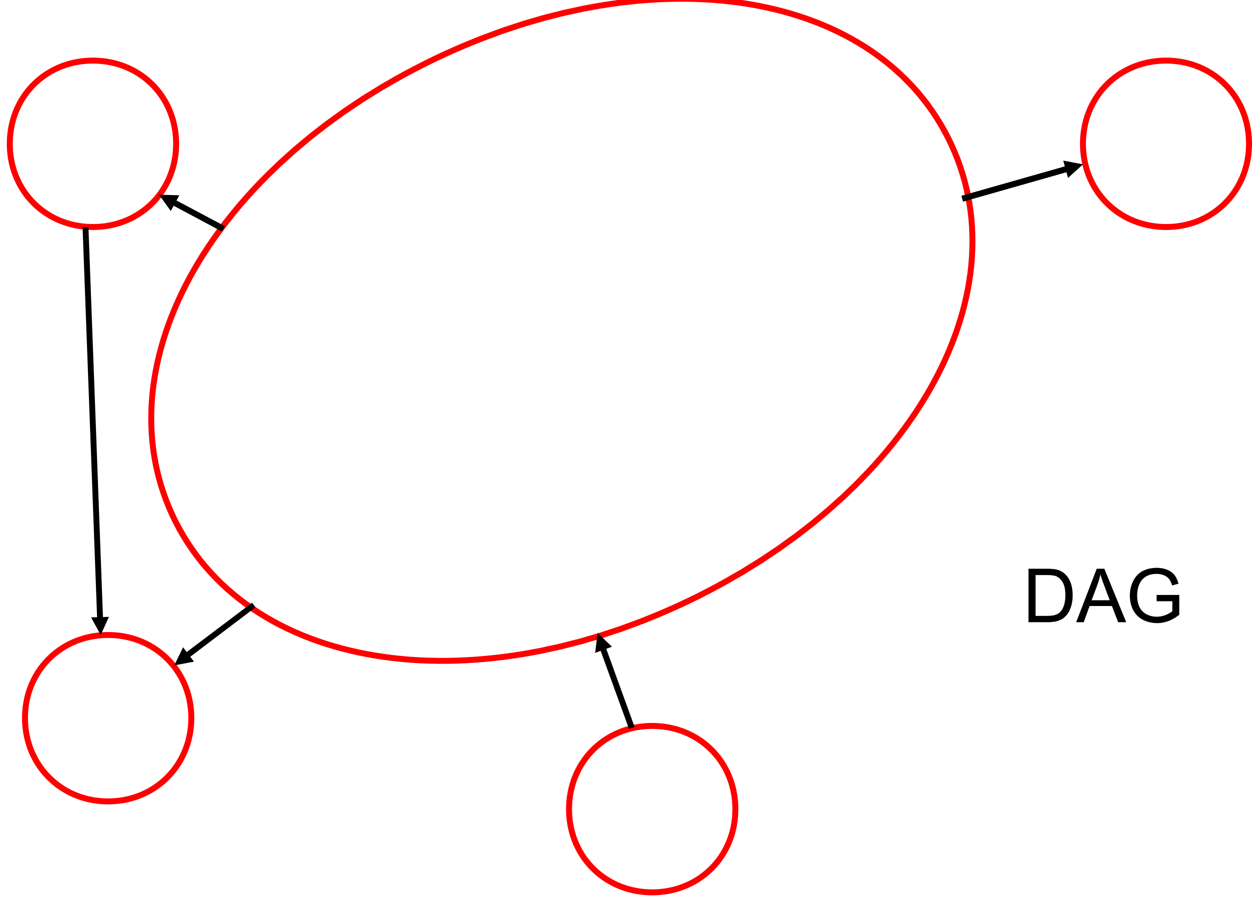
Beobachtung: Schrumpft man starke ZHKs zu einzelnen Knoten, dann ergibt sich DAG.



Starke ZHKs - Beispiel



Starke ZHKs - Beispiel



Starke ZHKs

Ziel: Finde alle starken ZHKs im Graphen in $O(n+m)$ Zeit (n: #Knoten, m: #Kanten)

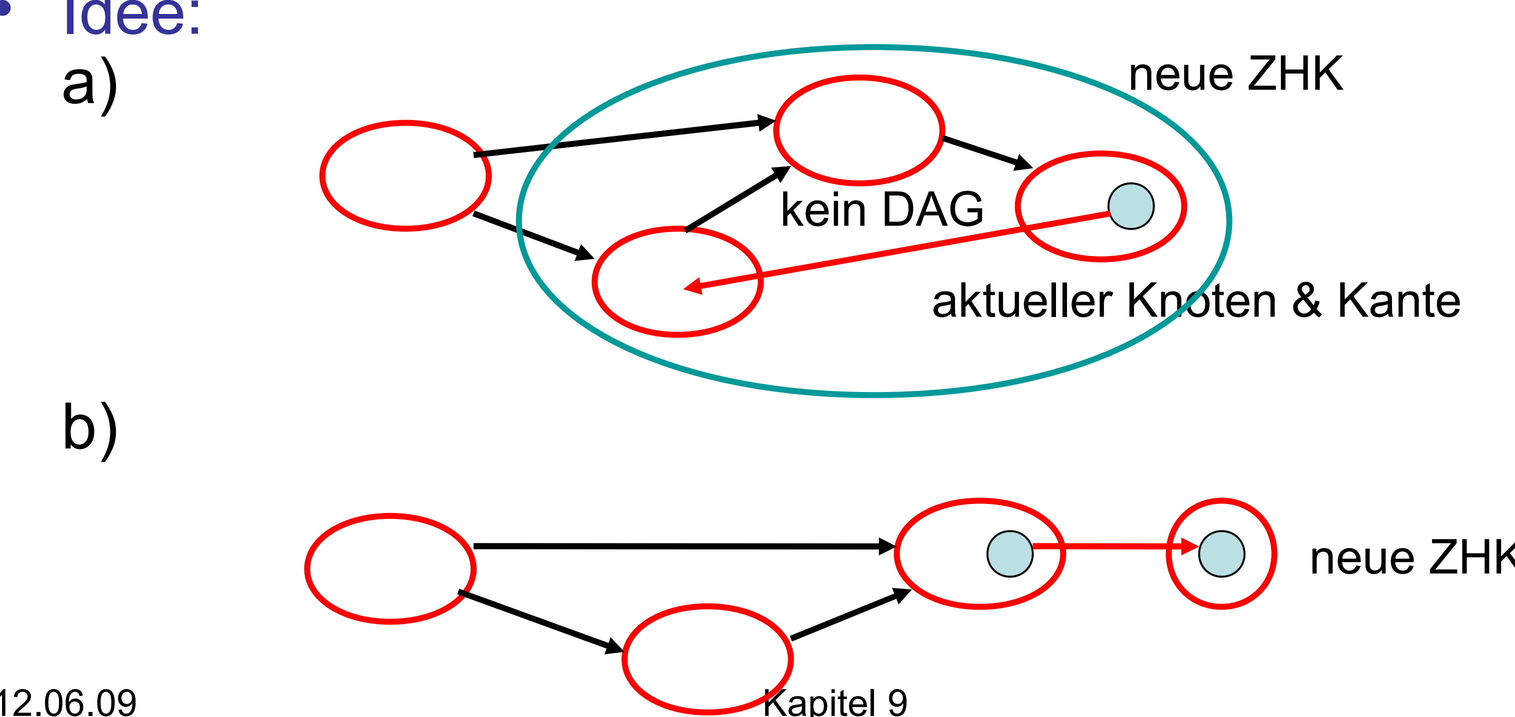
Strategie: Verwende DFS-Verfahren mit

int [] component;

Am Ende: component[v]==component[w] $\Leftrightarrow$ v und w sind in derselben starken ZHK

Starke ZHKs

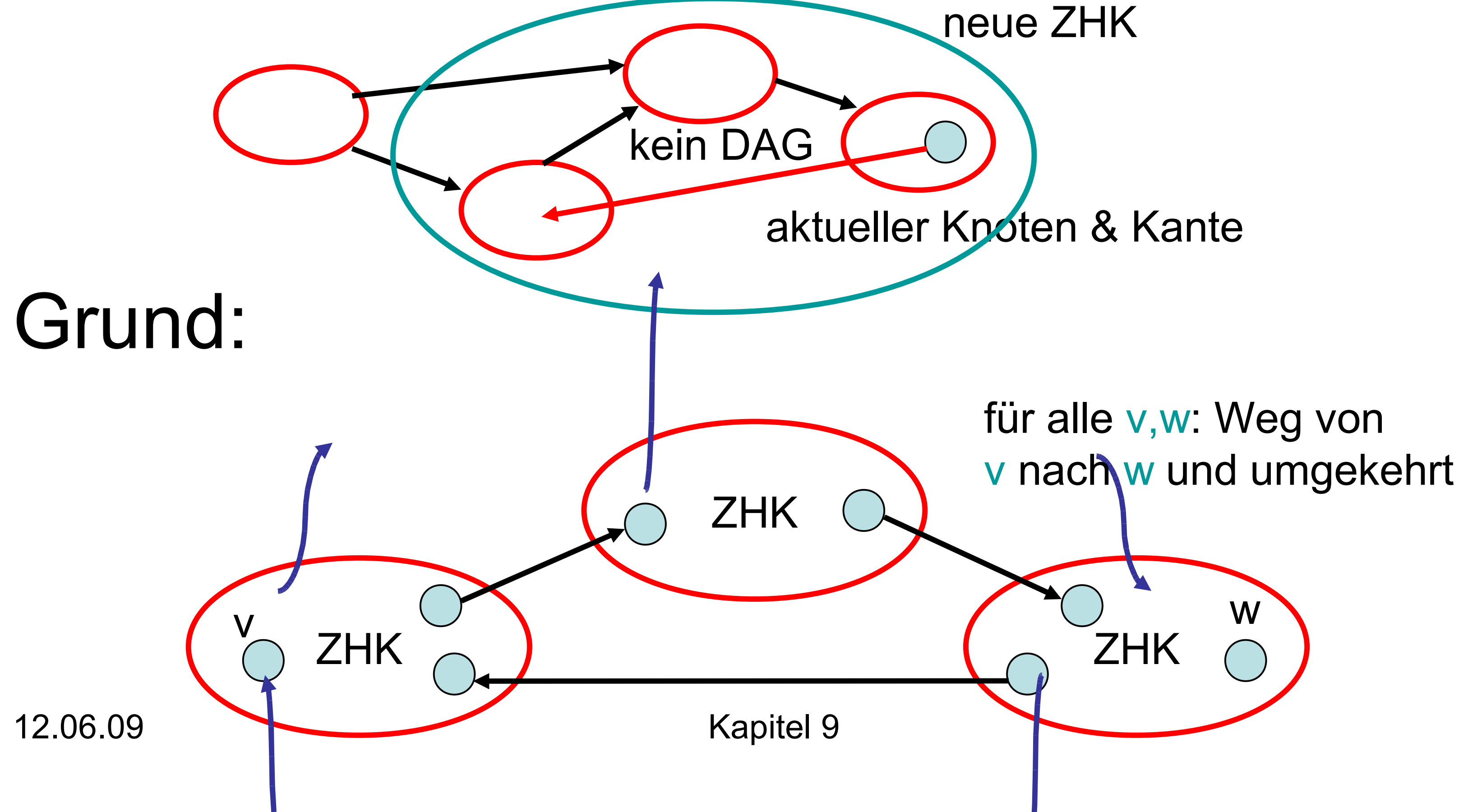
- Betrachte DFS auf $G=(V,E)$
- Sei $G_c=(V_c,E_c)$ bereits besuchter Teilgraph von G
- Ziel: bewahre starke ZHKs in G_c
- Idee:



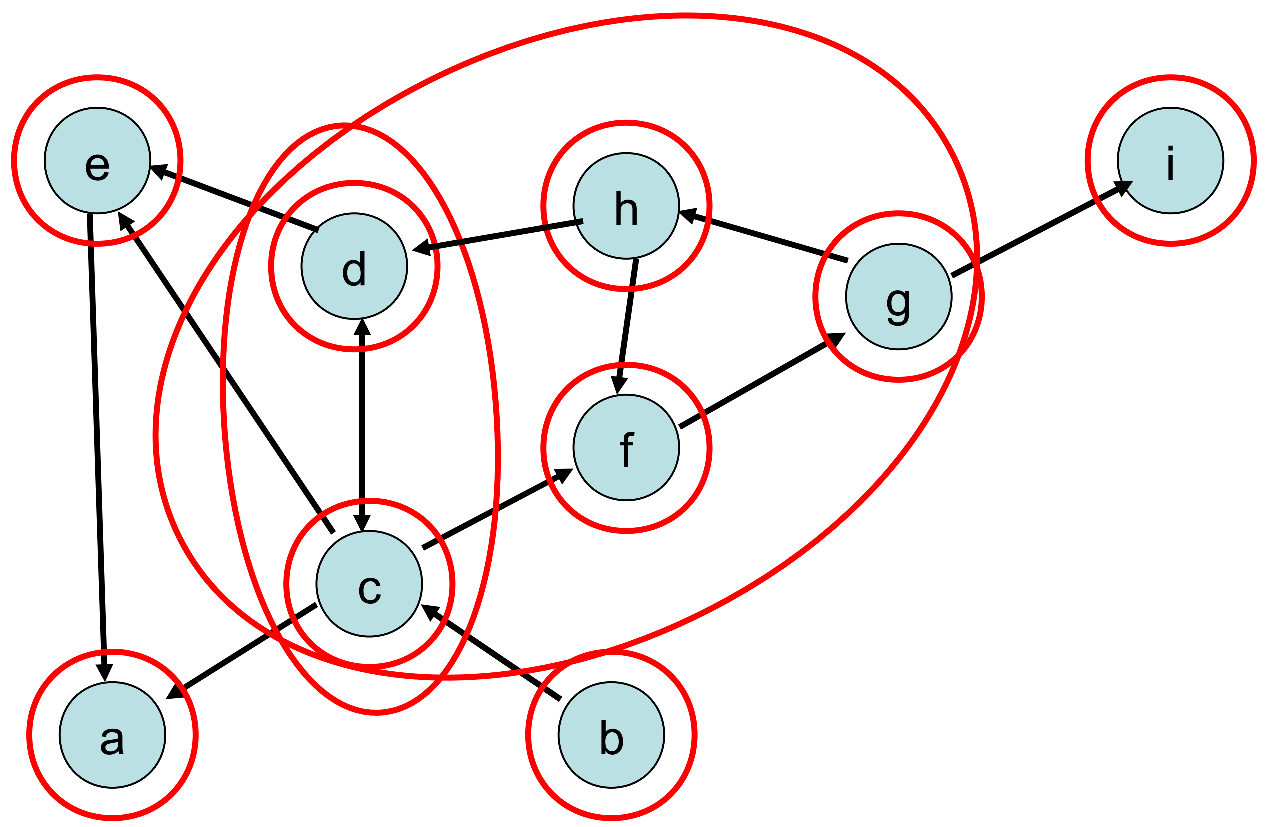
Starke ZHKs

Warum ZHKs zu einer zusammenfassbar?

Grund:



Starke SHKs - Beispiel



Problem: wie fasst man ZHKs effizient zusammen?

Starke ZHKs

Definition:

- ● ● : unfertiger Knoten
- ● : fertiger Knoten
- Eine ZHK in G heißt offen, falls sie noch unfertige Knoten enthält. Sonst heißt sie (und ihre Knoten) geschlossen.
- Repräsentant einer ZHK: Knoten mit kleinster dfsNum.

Starke ZHKs

Beobachtungen:

1. Alle Pfade aus geschlossenen Repräsentanten führen zu geschlossenen Knoten.
2. Der Pfad zum aktuellen Knoten enthält die Repräsentanten aller offenen ZHKs.
3. Betrachte die Knoten in offenen ZHKs sortiert nach DFS-Nummern. Die Repräsentanten partitionieren diese Folge in die offenen ZHKs.

Starke ZHKs

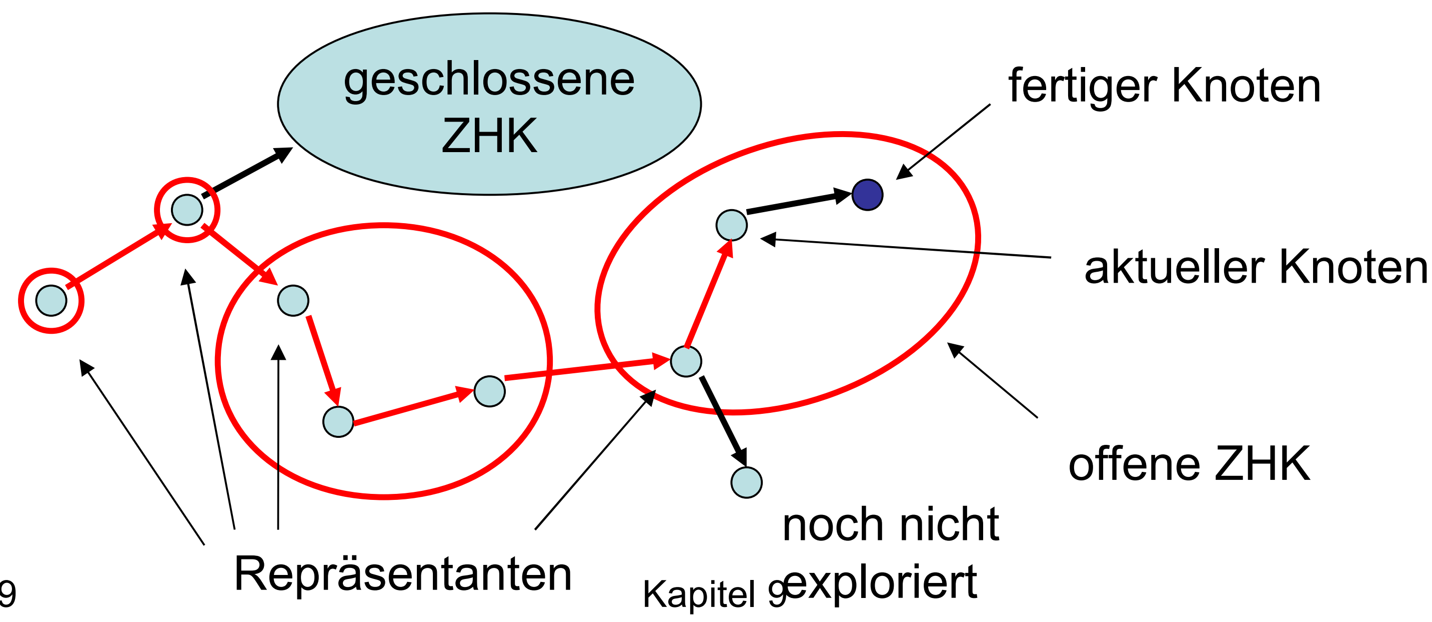
Beobachtungen sind Invarianten:

1. Alle Pfade aus geschlossenen Repräsentanten führen zu geschlossenen Knoten.
2. Der Pfad zum aktuellen Knoten enthält die Repräsentanten aller offenen ZHKs.
3. Betrachte die Knoten in offenen ZHKs sortiert nach DFS-Nummern. Die Repräsentanten partitionieren diese Folge in die offenen ZHKs.

Starke ZHKs

Beweis über vollständige Induktion.

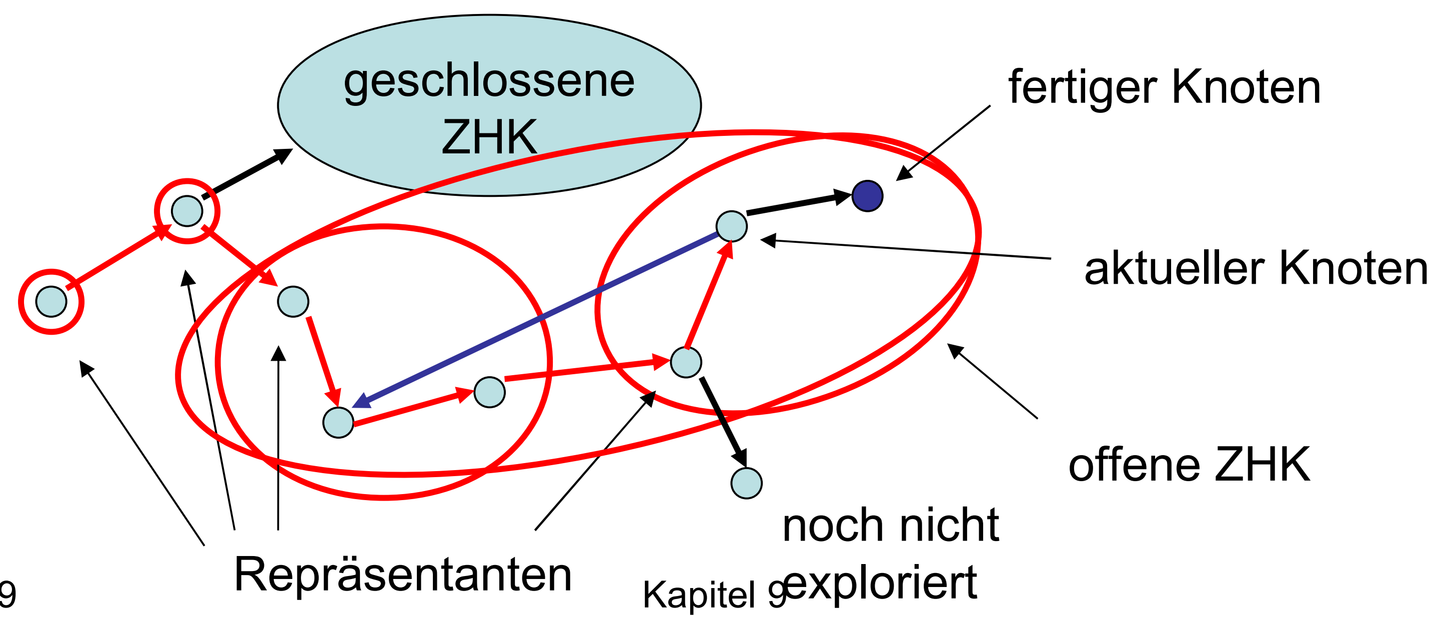
- Anfangs gelten alle Invarianten
- Wir betrachten verschiedene Fälle



Starke ZHKs

Beweis über vollständige Induktion.

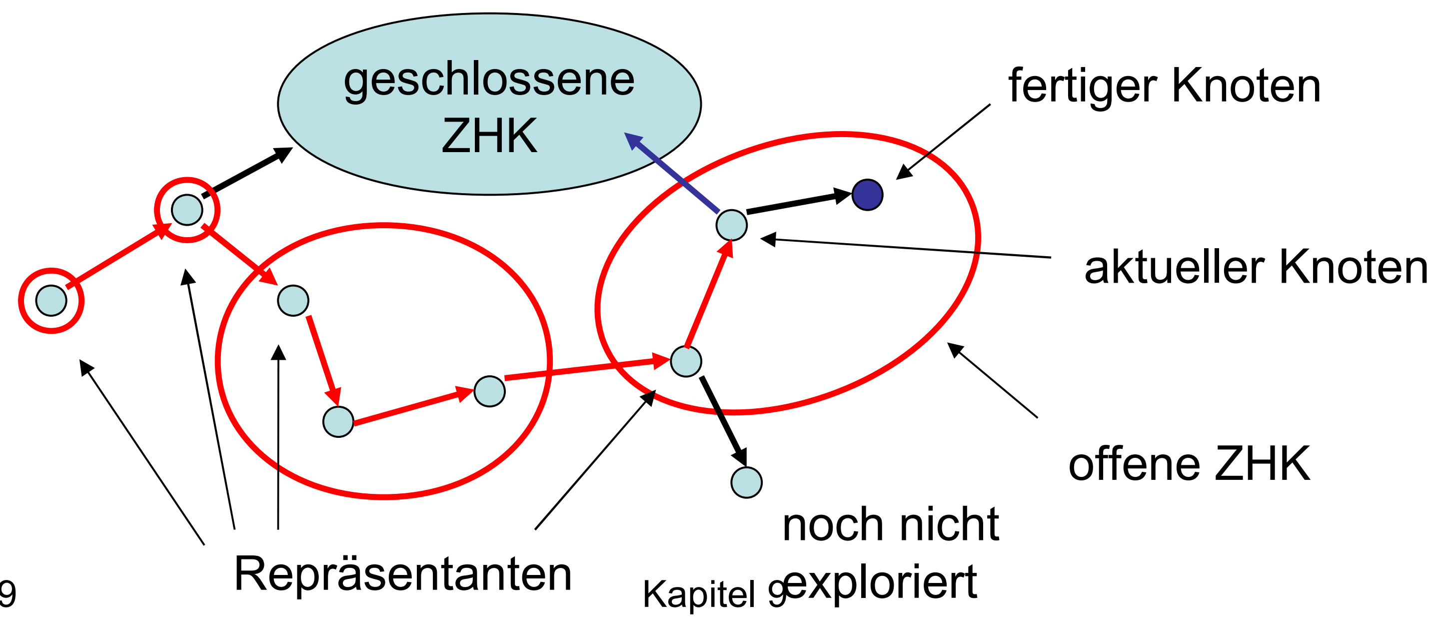
- Anfangs gelten alle Invarianten
- Fall 1: Kante zu unfertigem Knoten



Starke ZHKs

Beweis über vollständige Induktion.

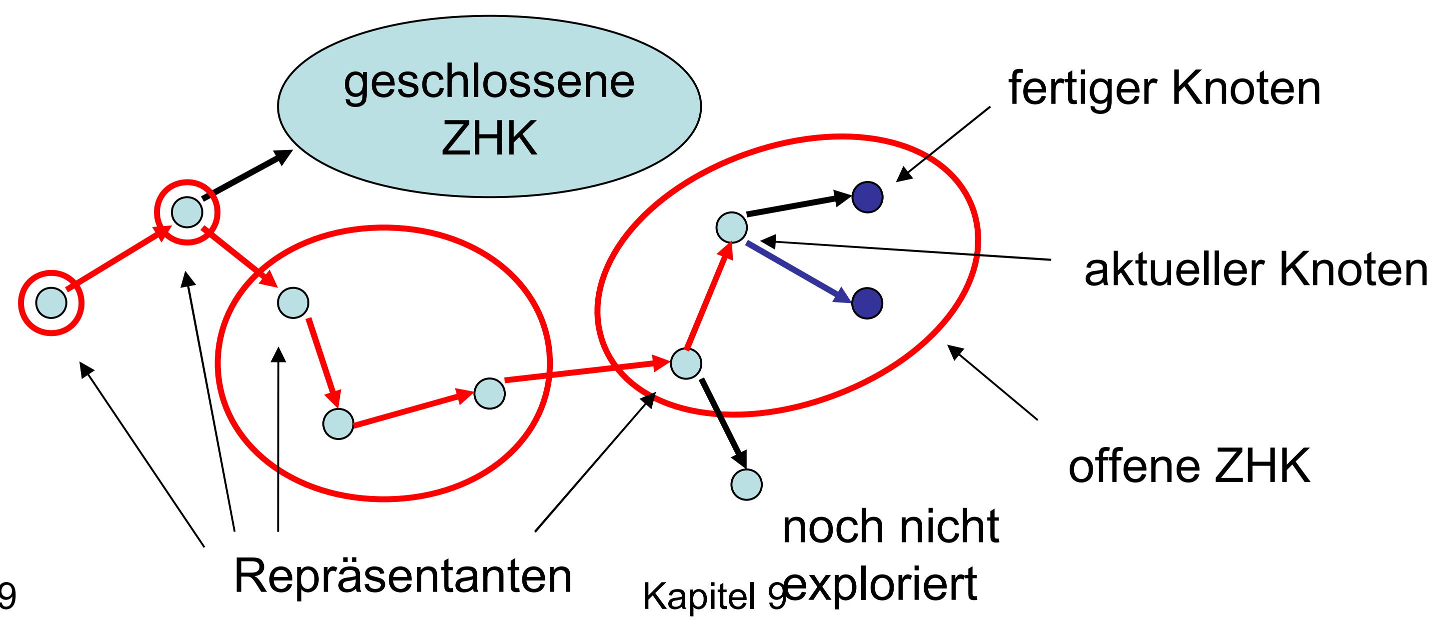
- Anfangs gelten alle Invarianten
- Fall 2: Kante zu geschlossenem Knoten



Starke ZHKs

Beweis über vollständige Induktion.

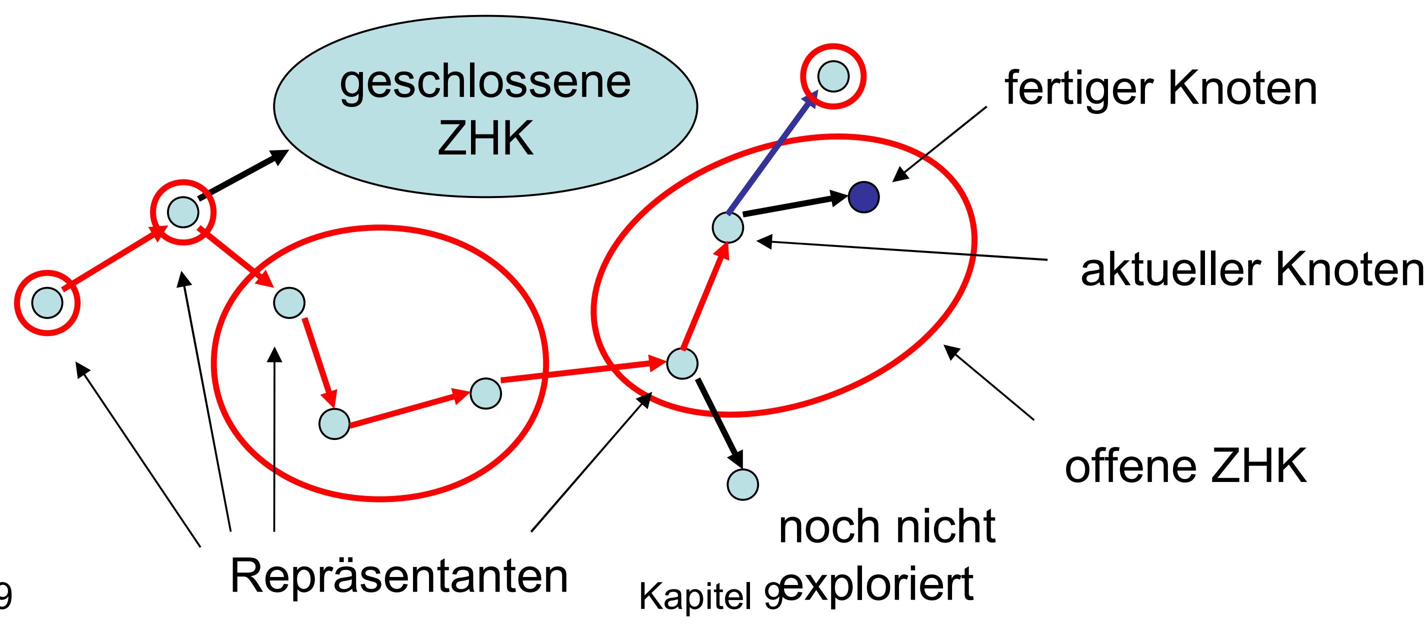
- Anfangs gelten alle Invarianten
- Fall 3: Kante zu fertigem Knoten



Starke ZHKs

Beweis über vollständige Induktion.

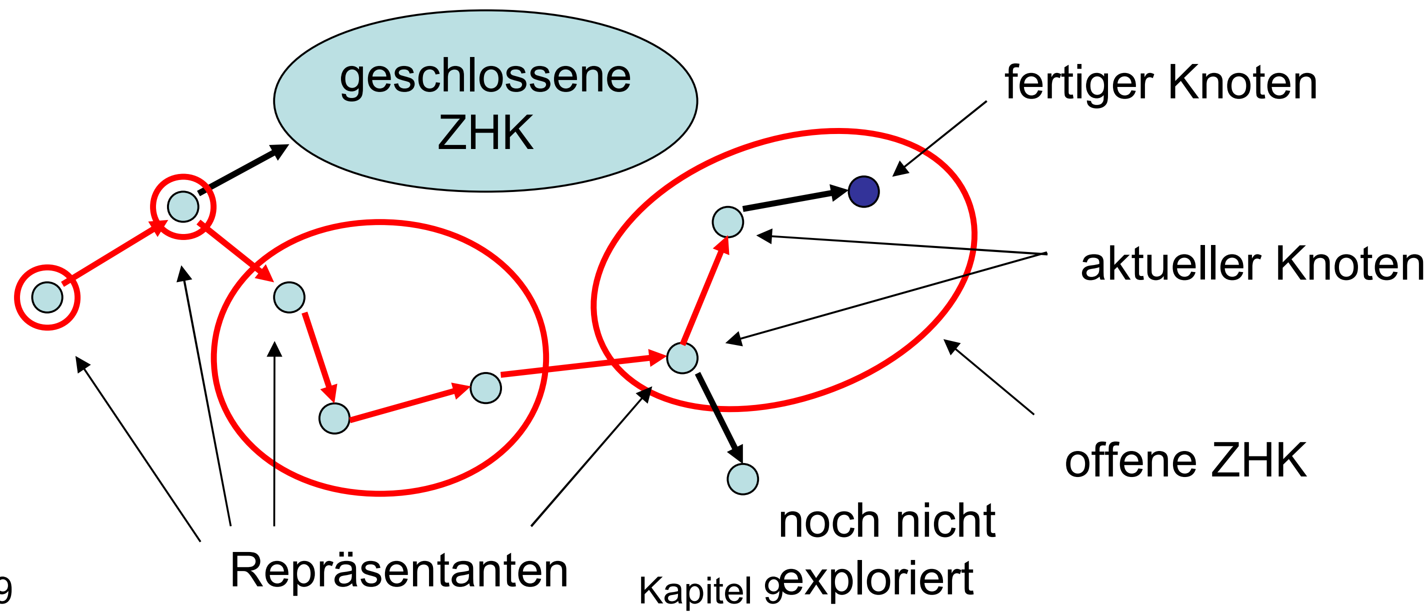
- Anfangs gelten alle Invarianten
- Fall 4: Kante zu nicht exploriertem Knoten



Starke ZHKs

Beweis über vollständige Induktion.

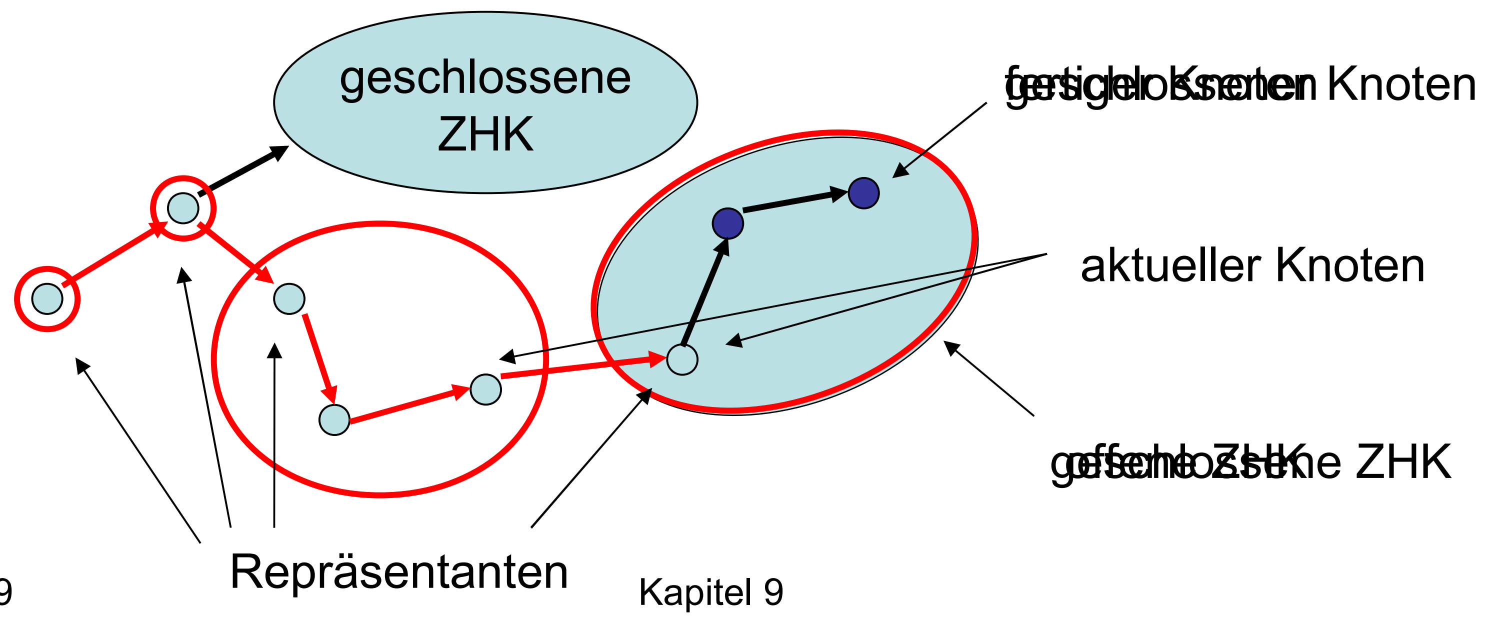
- Anfangs gelten alle Invarianten
- Fall 5: Knoten exploriert



Starke ZHKs

Beweis über vollständige Induktion.

- Anfangs gelten alle Invarianten
- Fall 5: Knoten exploriert



Starke ZHKs

Lemma 9.4: Eine geschlossene ZHK in G_c ist eine ZHK in G.

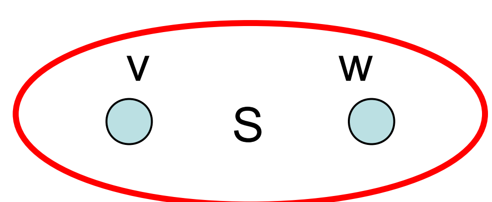
Beweis:

- v: geschlossener Repräsentant
- S: ZHK in G, die v enthält
- S_c: ZHK in G_c, die v enthält
- Es gilt: $S_c \subseteq S$

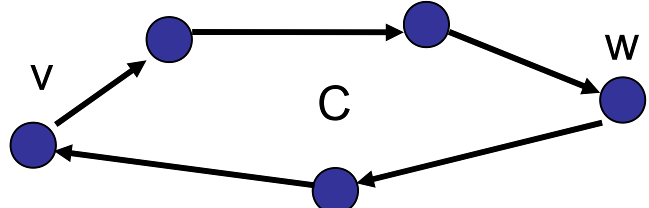
Zu zeigen: $S \subseteq S_c$

Starke ZHKs

Beweis von Lemma 9.4:



- w: beliebiger Knoten in S
- Es gibt gerichteten Kreis C durch v und w
- Invariante 1: alle Knoten in C geschlossen

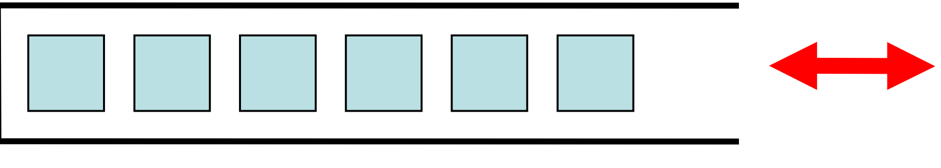


- Da alle Kanten geschlossener Knoten exploriert worden sind, ist C in G_c und daher $w \in S_c$

Starke ZHKs

Invarianten 2 und 3: einfache Methode, um offene ZHKs in G_c zu repräsentieren:

- Wir verwalten Folge oNodes aller offenen (nicht geschl.) Knoten in steigender DFS-Nummer und eine Teilfolge oReps aller offenen ZHK-Repräsentanten
- Stack ausreichend für beide Folgen



Starke ZHKs

```
init() {
   component = new int [n];
   oReps = <>;
   oNodes = <>;
   dfsPos = 1;
   }
void root(Node w) / traverseTreeEdge(Node v, Node
   w) {
   oReps.push(w);     // neue ZHK
   oNodes.push(w);   // neuer offener Knoten
   dfsNum[w] = dfsPos; dfsPos = dfsPos+1;
}
```

Starke ZHKs

```
void traverseNonTreeEdge(Node v, Node w) {
  if (w ∈ oNodes)   // kombiniere ZHKs
    while (dfsNum[w] < dfsNum[oReps.top()])
      oReps.pop();
}
void backtrack(Node u, Node v) {
  if (v == oReps.top()) {  // v Repräsentant?
    oReps.pop();  // ja: entferne v
    do {           // und offene Knoten bis v
      w = oNodes.pop();
      component[w] = v;
    while (w != v);
}
```

Starke ZHKs

Theorem 9.5: Der DFS-basierte Algorithmus für starke ZHKs benötigt O(n+m) Zeit.

Beweis:

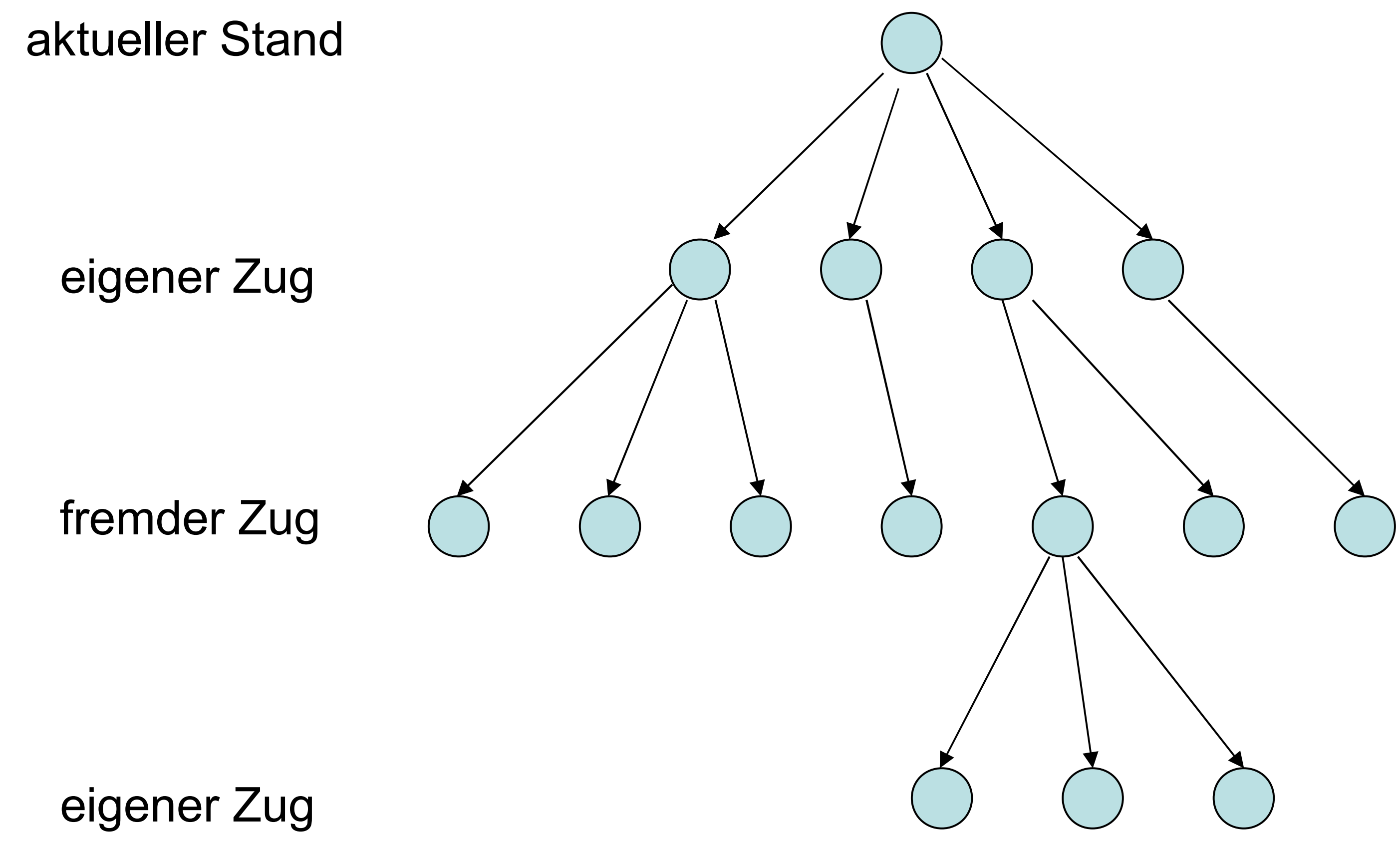
- init, root, traverseTreeEdge: Zeit O(1)
- Backtrack, traverseNonTreeEdge: da jeder Knoten nur höchstens einmal in oReps und oNodes landet, insgesamt Zeit O(n+m)
- DFS-Gerüst: Zeit O(n+m)

Anwendungen

- BFS:
  Spiele (Exploration des Spielbaums)

- DFS:
  Suche des Ausgangs in Labyrinth

Exploration des Spielbaums

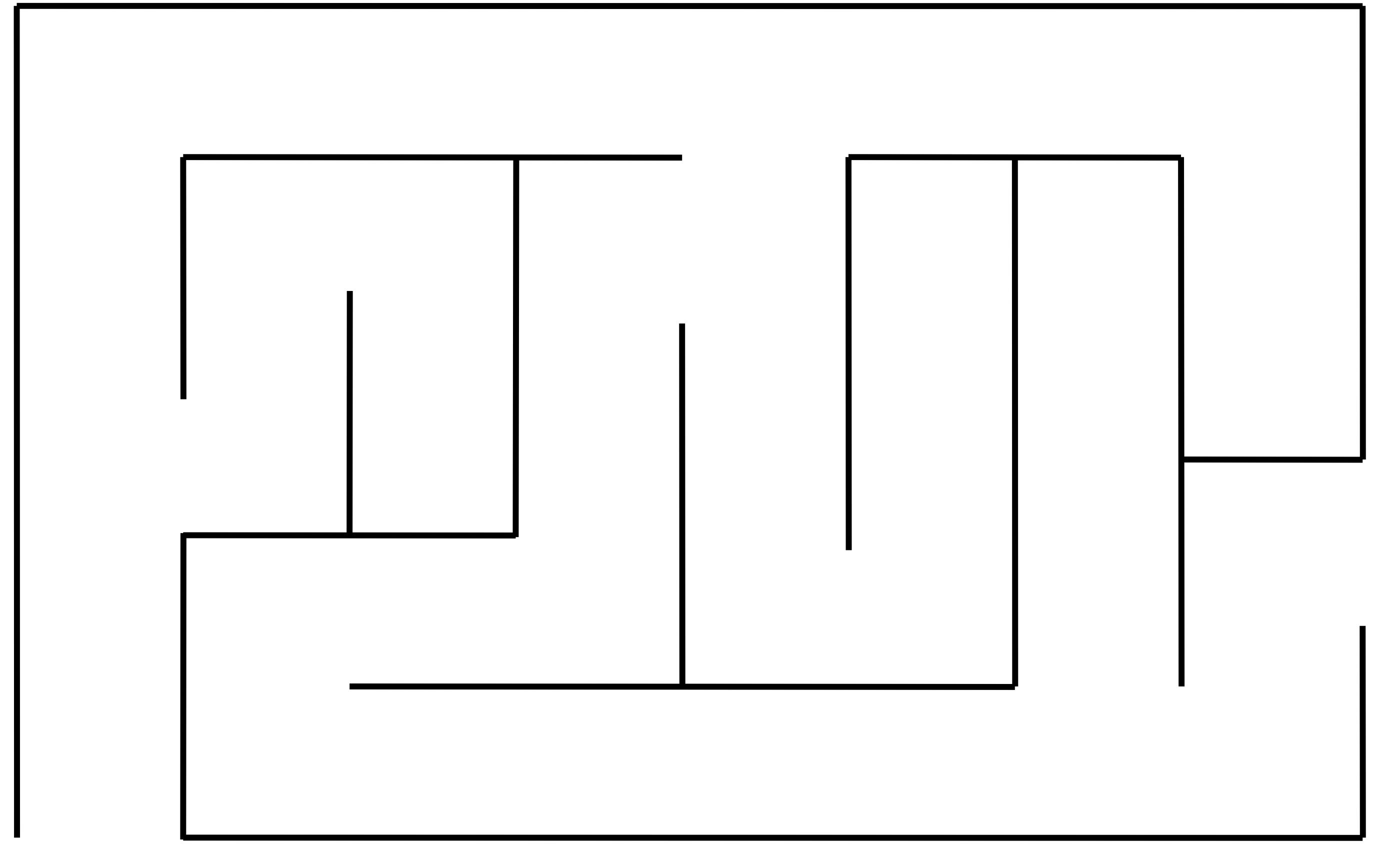


Problem: halte Aufwand zur Suche eines guten Zuges in Grenzen

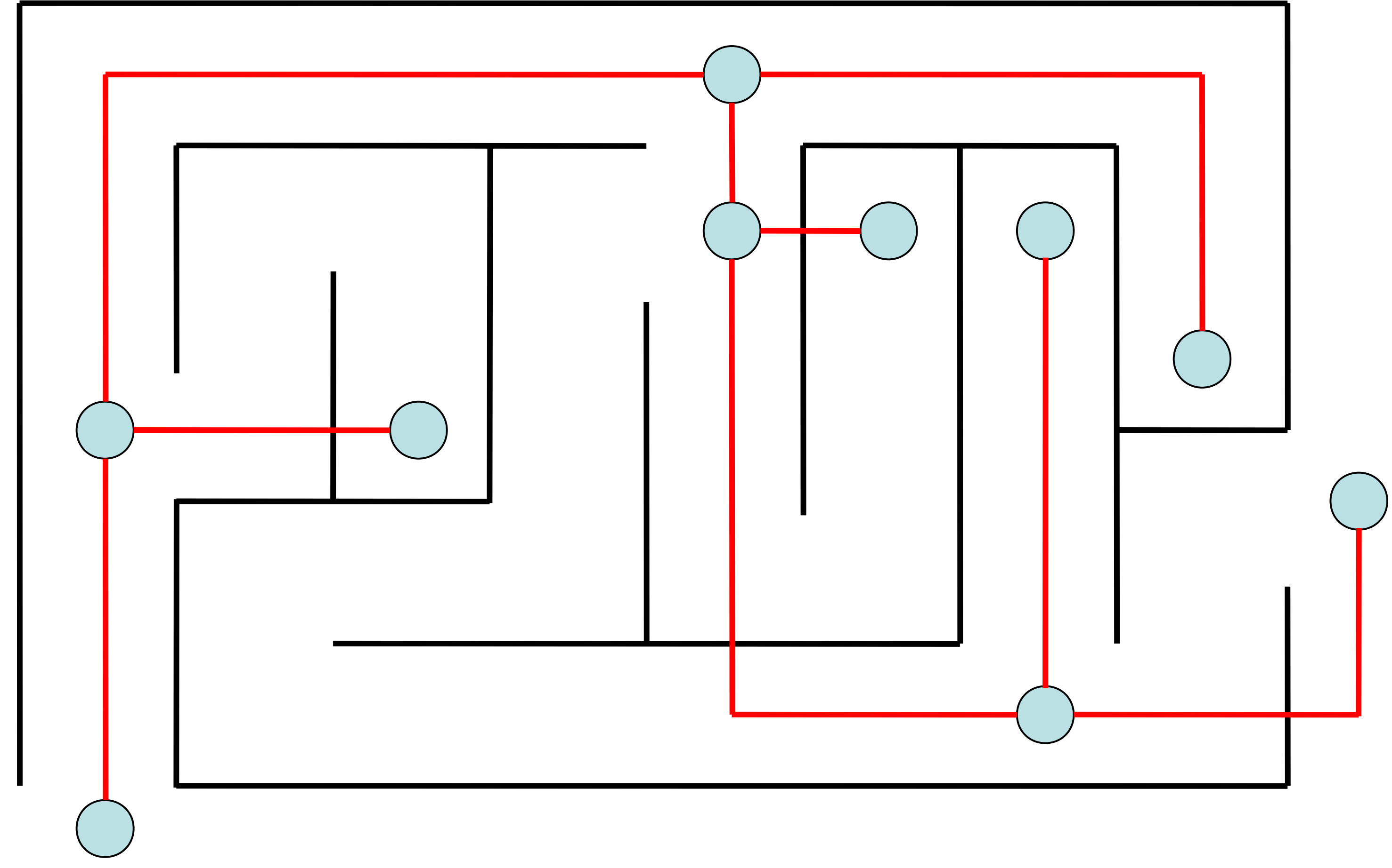
Exploration des Spielbaums

- Standard-BFS: verwendet FIFO-Queue (ebenenweise Erkundung), aber zu teuer!!
- Besserer BFS: verwende Priority Queue (z.B. realisiert durch binären Heap), wobei Priorität eines Knoten durch eine Güte-Heuristik des Spielzustands, den er repräsentiert, gegeben ist
- Dieser bessere BFS heißt auch Best-First-Search

Labyrinth

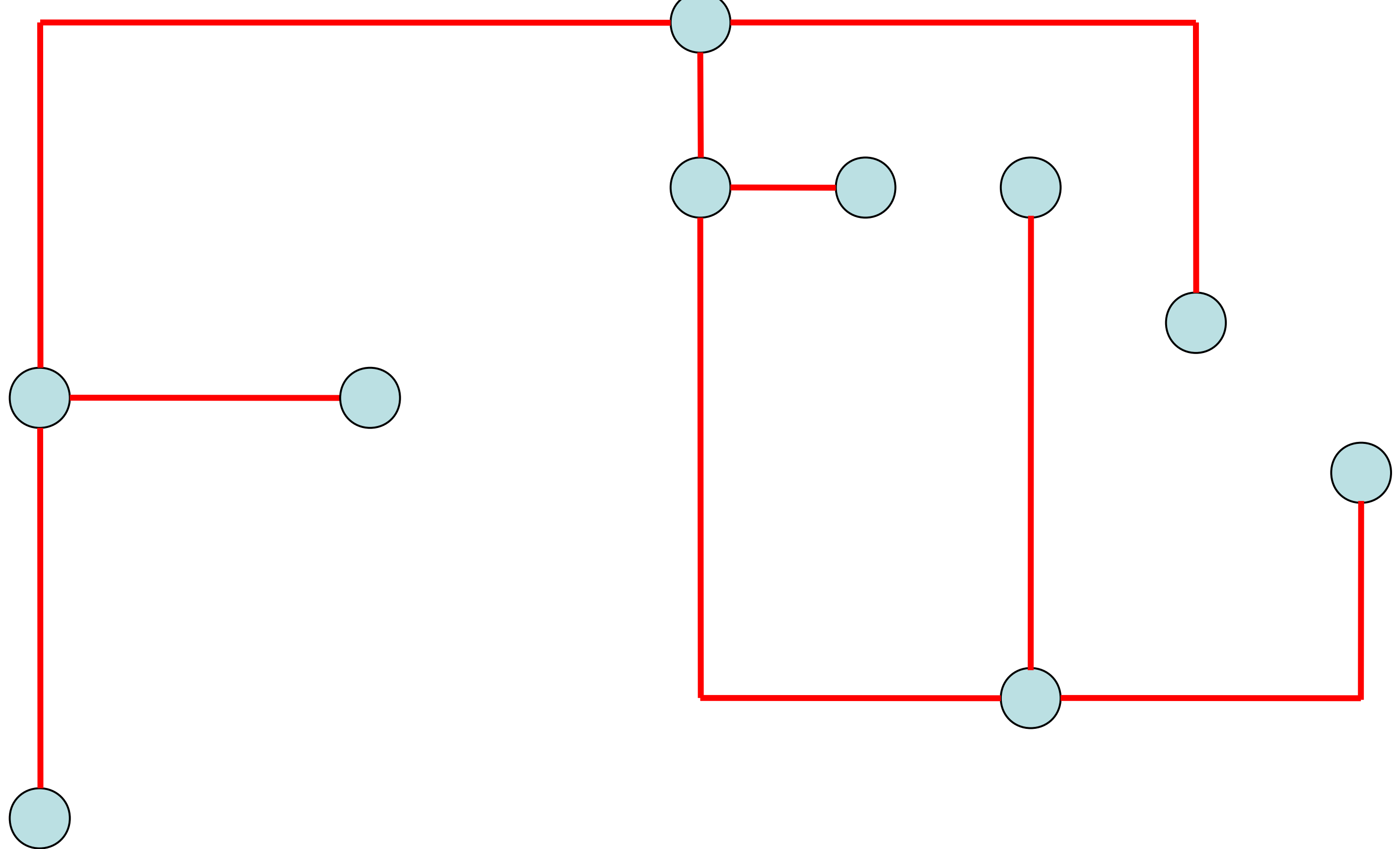


Labyrinth als Graph



Labyrinth als Graph

Tiefensuche findet in Linearzeit Weg zum Ziel



Suche in Labyrinth

In speziellen Fällen (planare Labyrinthe mit rechteckigen Wänden und dem Eingang und Ausgang an der Seite) sind auch Verfahren bekannt, die sich nicht merken müssen, welcher Knotenpfad zur Zeit exploriert wird.

Siehe Pledge Algorithmus