

# Algorithmen und Datenstrukturen SS09

## Foliensatz 17

Michael Brinkmeier

Technische Universität Ilmenau  
Institut für Theoretische Informatik

Sommersemester 2009

## Breitensuche

# Graphtraverse

Wie bei Bäumen, ist es in verschiedenen Situationen notwendig, alle Knoten des Graphen systematisch zu Durchlaufen.

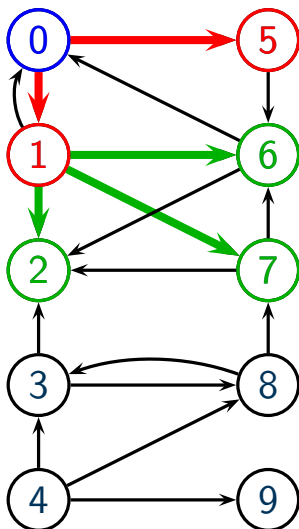
Da Graphen – im Gegensatz zu Bäumen – aber nicht zusammenhängend sein müssen, muss das Problem etwas anders formuliert werden.

Durchlaufe alle Knoten, die von einem Knoten  $v_0$  aus erreichbar sind.

Ziel ist es dabei, bei jedem Knoten einmal während dieses Durchlaufes eine Aktion vorzunehmen.

## Breitensuche

Bei der **Breitensuche** besucht man die Knoten in aufsteigender **Distanz** zu dem Ausgangsknoten  $v_0$ .



Von 0 aus besucht man also ...  
...zuerst **1,5**  
...dann **2,6,7** ...

Die Breitensuche entspricht dem **Levelorder-Durchlauf** von Bäumen.

# Breitensuche

Die Knoten werden zuerst **entdeckt** und später **bearbeitet**.

Während der **Entdeckung** werden die Knoten nummeriert, als **entdeckt** markiert und mit der Distanz zu  $v_0$  versehen.

Während der **Bearbeitung** wird die Adjazenzliste eines Knotens durchlaufen um neue Knoten zu **entdecken**.

Um Knoten in der Reihenfolge ihrer **Entdeckung** zu bearbeiten, werden sie in eine **Queue** eingefügt.

Anfangs enthält die Queue nur den Knoten  $v_0$ .

# Breitensuche

Für jeden Knoten  $v$  speichern wir die folgenden Werte:

- die **Breitensuchnummer**  $\text{bfs\_num}[v]$ , d.h. als wievielter Knoten  $v$  **entdeckt** wurde; anfänglich mit 0 initialisiert
- den **Level**  $\text{level}[v]$ , d.h. die Distanz von  $v$  zum Ausgangsknoten  $v_0$ .
- den **Vorgänger**  $\text{p}[v]$  von  $v$  auf einem kürzesten Weg von  $v_0$  nach  $v$ .

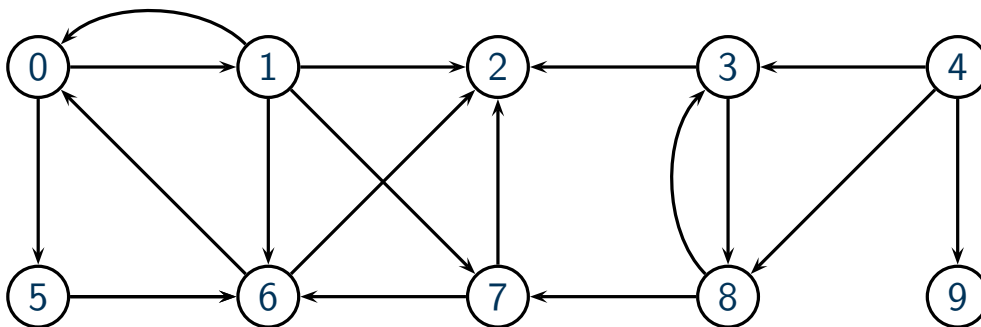
Zusätzlich benötigen wir einen globalen Zähler **bfs\_count**, der die Anzahl der bisher **entdeckten** Knoten enthält.

# Breitensuche



## BFS( $v, bfs\_count$ )

```
bfs_count ++; bfs_num[v0] = bfs_count;
level[v0] = 0; p[v0] = v0;
Q.enqueue(v0);
solange not Q.isempty() tue
  v = Q.dequeue();
  für alle Nachfolger w von v tue
    wenn bfs_num[w] = 0 dann
      bfs_count ++; bfs_num[w] = bfs_count;
      level[w] = level[v] + 1; p[w] = v;
      Q.enqueue(w);
  Ende
Ende
Ende
```

# Breitensuche

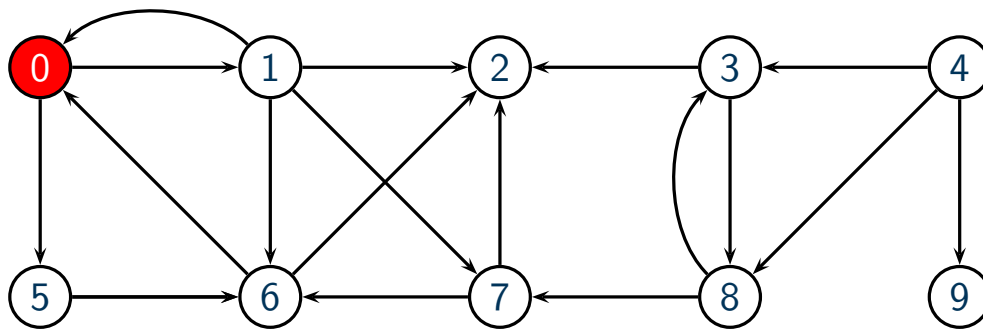


Die Adjazenzlisten sind nach den Zielknoten sortiert.

-  entdeckt
-  bearbeitet

Queue
entdeckte Knoten
Level
Vorgänger

# Breitensuche

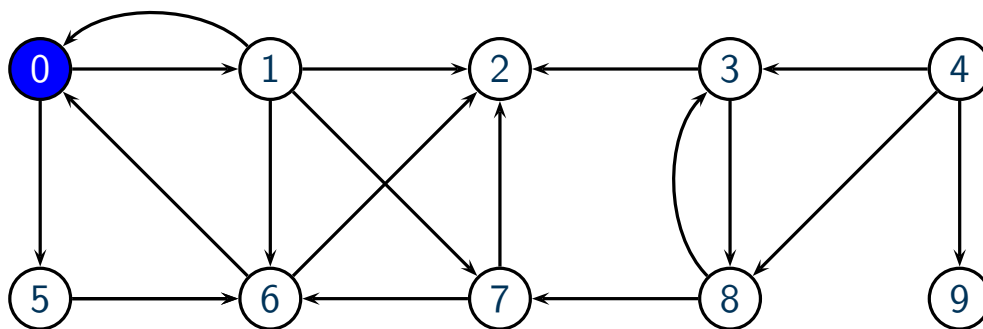


Die Adjazenzlisten sind nach den Zielknoten sortiert.

- A entdeckt
- B bearbeitet

Queue	0
entdeckte Knoten	
Level	0
Vorgänger	0

# Breitensuche

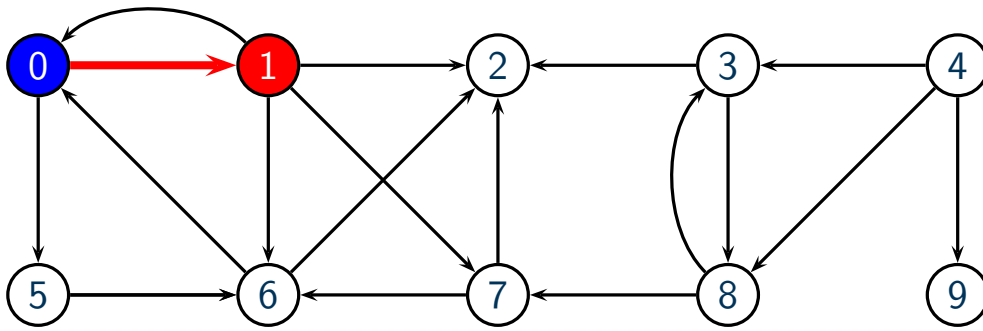


Die Adjazenzlisten sind nach den Zielknoten sortiert.

- A entdeckt
- B bearbeitet

Queue	
entdeckte Knoten	0
Level	0
Vorgänger	0

# Breitensuche

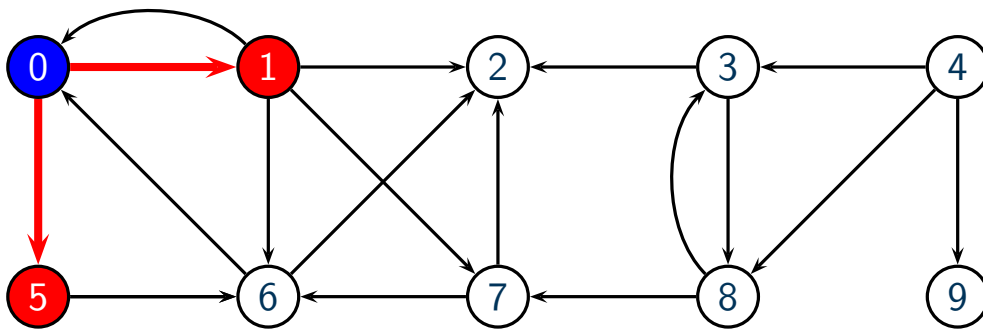


Die Adjazenzlisten sind nach den Zielknoten sortiert.

- A entdeckt
- B bearbeitet

Queue		1
entdeckte Knoten	0	
Level	0	1
Vorgänger	0	0

# Breitensuche

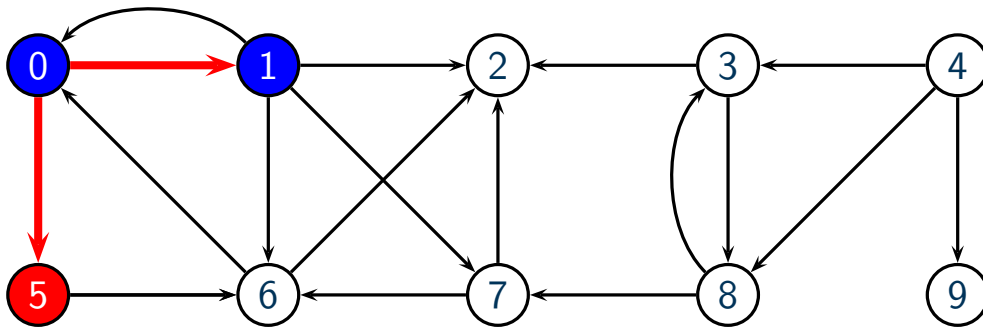


Die Adjazenzlisten sind nach den Zielknoten sortiert.

- A entdeckt
- B bearbeitet

Queue		1	5
entdeckte Knoten	0		
Level	0	1	1
Vorgänger	0	0	0

# Breitensuche

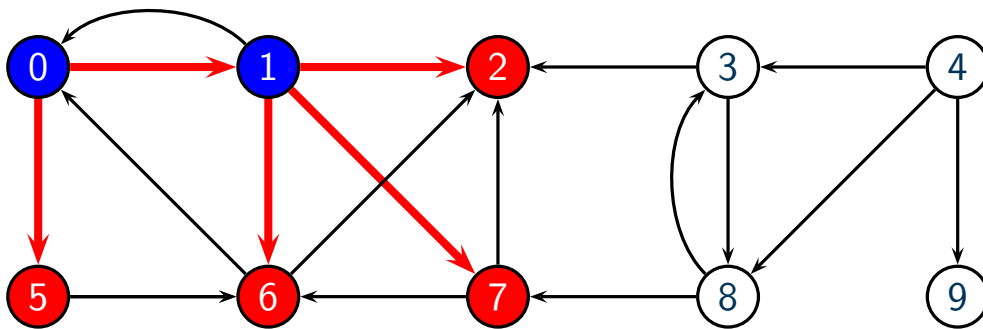


Die Adjazenzlisten sind nach den Zielknoten sortiert.

- A entdeckt
- B bearbeitet

Queue			5
entdeckte Knoten	0	1	
Level	0	1	1
Vorgänger	0	0	0

# Breitensuche

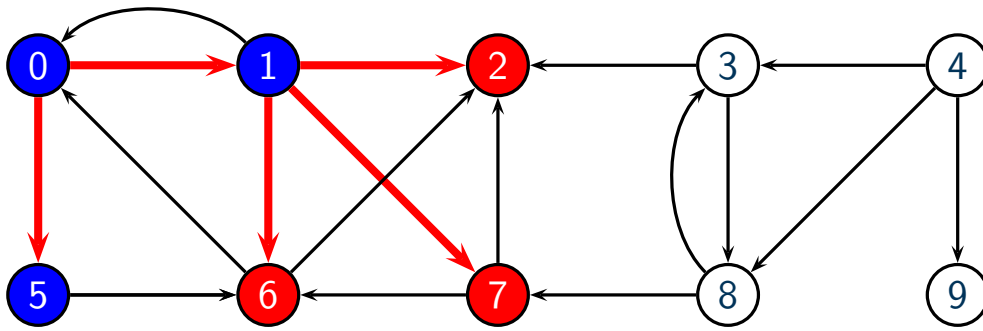


Die Adjazenzlisten sind nach den Zielknoten sortiert.

- A entdeckt
- B bearbeitet

Queue			5	2	6	7
entdeckte Knoten	0	1				
Level	0	1	1	2	2	2
Vorgänger	0	0	0	1	1	1

# Breitensuche

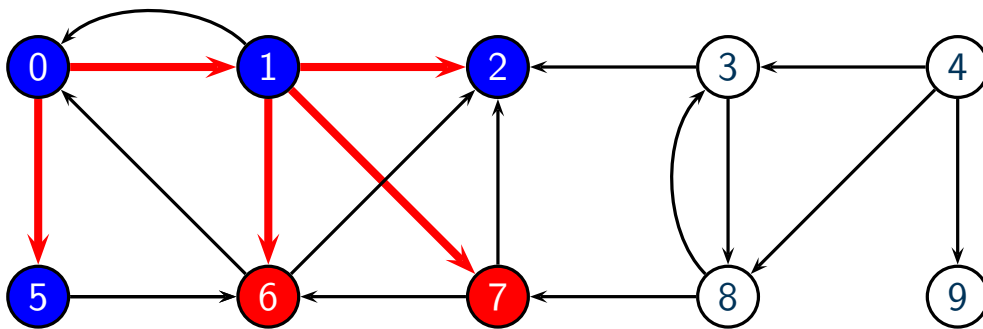


Die Adjazenzlisten sind nach den Zielknoten sortiert.

- A entdeckt
- B bearbeitet

Queue				2	6	7
entdeckte Knoten	0	1	5			
Level	0	1	1	2	2	2
Vorgänger	0	0	0	1	1	1

# Breitensuche

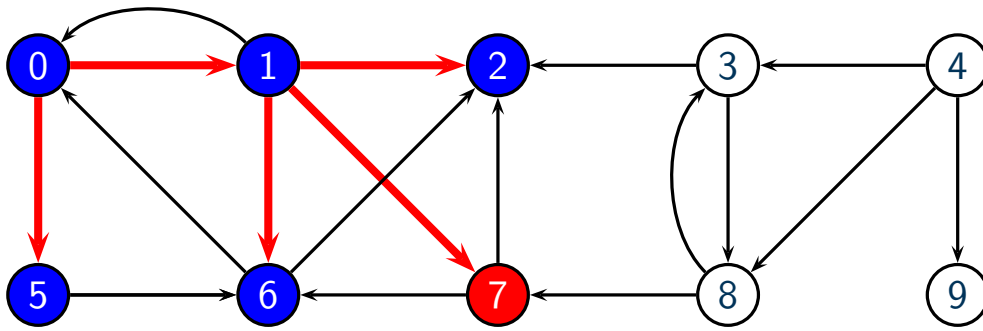


Die Adjazenzlisten sind nach den Zielknoten sortiert.

- A entdeckt
- B bearbeitet

Queue					6	7
entdeckte Knoten	0	1	5	2		
Level	0	1	1	2	2	2
Vorgänger	0	0	0	1	1	1

# Breitensuche

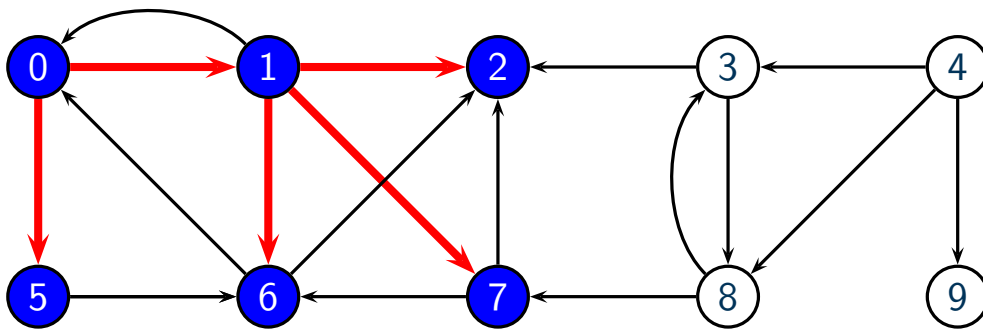


Die Adjazenzlisten sind nach den Zielknoten sortiert.

- A entdeckt
- B bearbeitet

Queue						7
entdeckte Knoten	0	1	5	2	6	
Level	0	1	1	2	2	2
Vorgänger	0	0	0	1	1	1

# Breitensuche



Die Adjazenzlisten sind nach den Zielknoten sortiert.

- A entdeckt
- B bearbeitet

Queue							7
entdeckte Knoten	0	1	5	2	6	7	
Level	0	1	1	2	2	2	2
Vorgänger	0	0	0	1	1	1	1

# Die Breitensuche

## Lemma

Wird die Breitensuche in Knoten  $v_0$  gestartet, werden genau die Knoten besucht, die von  $v_0$  aus erreichbar sind.

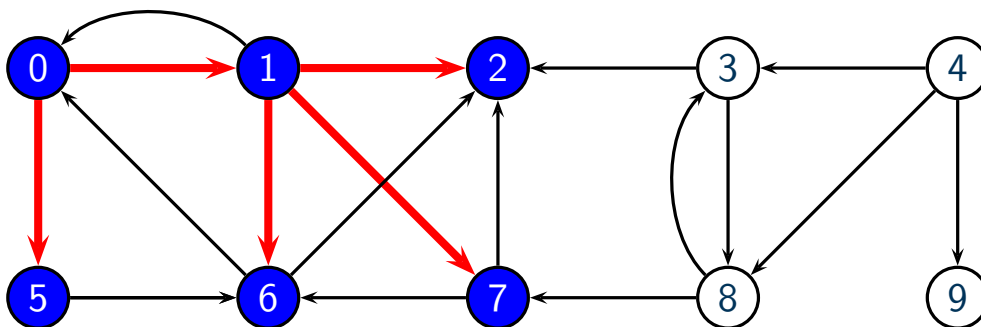
Sei  $R_v := \{w \in V \mid v \rightsquigarrow w\}$  die Menge der von  $v$  erreichbaren Knoten.

- 1  $\text{bfs}(v_0)$  benötigt Zeit  $\mathcal{O}(|R_{v_0}| + |E_{v_0}|)$ .
- 2 Der Speicherbedarf der Queue ist  $\mathcal{O}(|R_{v_0}|)$ .
- 3 Für  $v \neq v_0$  ist  $p(v)$  der Knoten, von dem aus  $v$  entdeckt wird. Weiter gilt  $\text{level}(p(v)) + 1 = \text{level}(v)$ .

## Der Breitensuchbaum

### Fazit

Die entdeckten Knoten mit den Kanten  $(p(v), v)$  bilden einen Baum, den **Breitensuchbaum**.



# Der Breitensuchbaum

## Satz

Nach Aufruf von  $\text{bfs}(v_0)$  ist  $\text{level}(v)$  die Länge eines kürzesten Weges von  $v_0$  nach  $v$  für jedes  $v \in R_{v_0}$ .

Der Weg  $(v_0, v_1, \dots, v_l = v)$  mit  $v_{i-1} = p(v_i)$  Für  $1 \leq i \leq l$  ist ein kürzester Weg von  $v_0$  nach  $v_l = v$ .

Der Beweis verwendet ähnliche Argumente, wie der für den Levelorder-Durchlauf.

## Die globale Breitensuche

### Frage

Wie werden die restlichen Knoten erreicht?

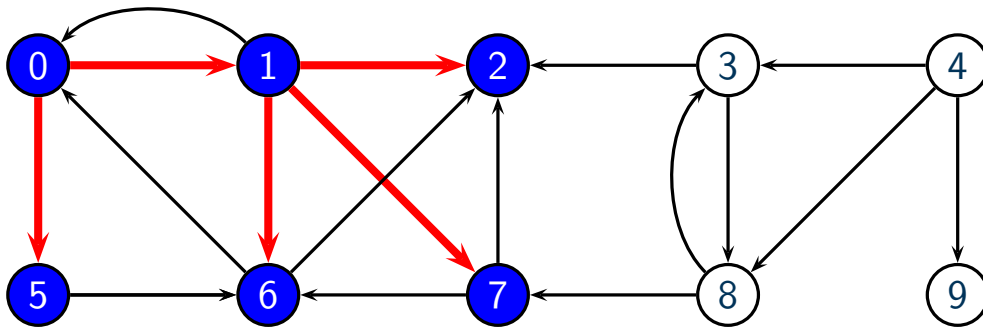
### Antwort

Man startet eine Breitensuche bei jedem Knoten, der nicht in einer vorhergegangenen Breitensuche gefunden wurde.

### BFS( $G$ )

```
bfs_count = 0;  
für alle  $v \in V$  tue bfs_num( $v$ ) = 0;  
für alle  $v \in V$  tue  
    wenn bfs_num( $v$ ) == 0 dann  
        bfs( $v$ , bfs_count);  
    Ende  
Ende
```

# Breitensuche

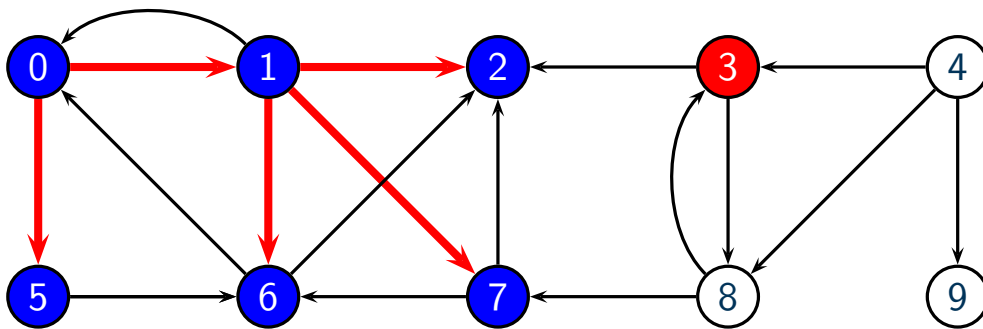


Die Adjazenzlisten sind nach den Zielknoten sortiert.

- A entdeckt
- B bearbeitet

Queue						
entdeckte Knoten	0	1	5	2	6	7
Level	0	1	1	2	2	2
Vorgänger	0	0	0	1	1	1

# Breitensuche

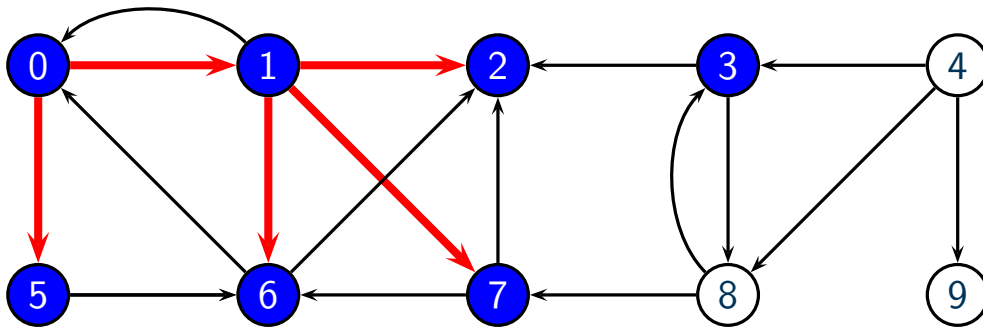


Die Adjazenzlisten sind nach den Zielknoten sortiert.

- A entdeckt
- B bearbeitet

Queue							3
entdeckte Knoten	0	1	5	2	6	7	
Level	0	1	1	2	2	2	0
Vorgänger	0	0	0	1	1	1	3

# Breitensuche

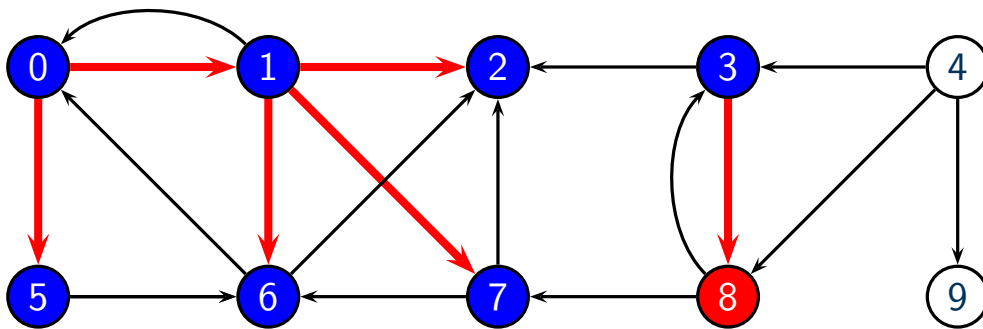


Die Adjazenzlisten sind nach den Zielknoten sortiert.

- A entdeckt
- B bearbeitet

Queue							
entdeckte Knoten	0	1	5	2	6	7	3
Level	0	1	1	2	2	2	0
Vorgänger	0	0	0	1	1	1	3

# Breitensuche

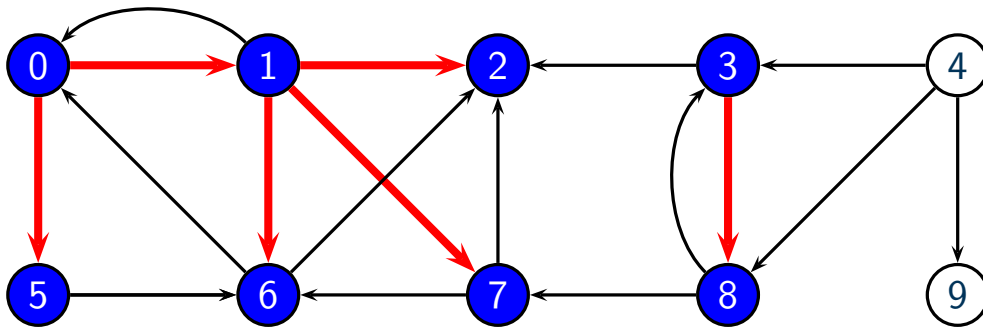


Die Adjazenzlisten sind nach den Zielknoten sortiert.

- A entdeckt
- B bearbeitet

Queue								8
entdeckte Knoten	0	1	5	2	6	7	3	
Level	0	1	1	2	2	2	0	1
Vorgänger	0	0	0	1	1	1	3	3

# Breitensuche

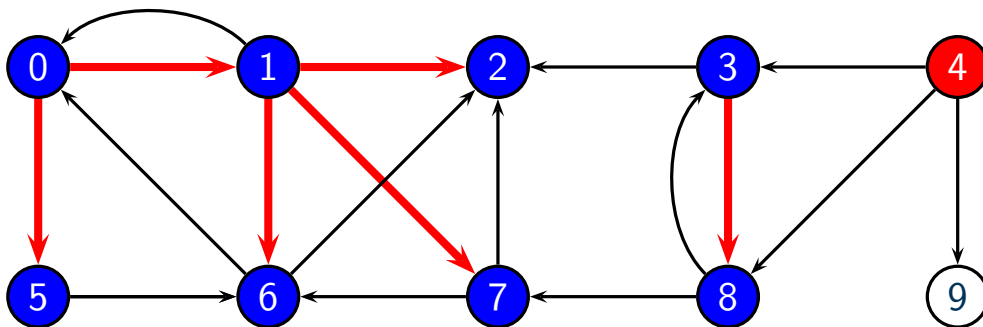


Die Adjazenzlisten sind nach den Zielknoten sortiert.

- A entdeckt
- B bearbeitet

Queue								
entdeckte Knoten	0	1	5	2	6	7	3	8
Level	0	1	1	2	2	2	0	1
Vorgänger	0	0	0	1	1	1	3	3

# Breitensuche

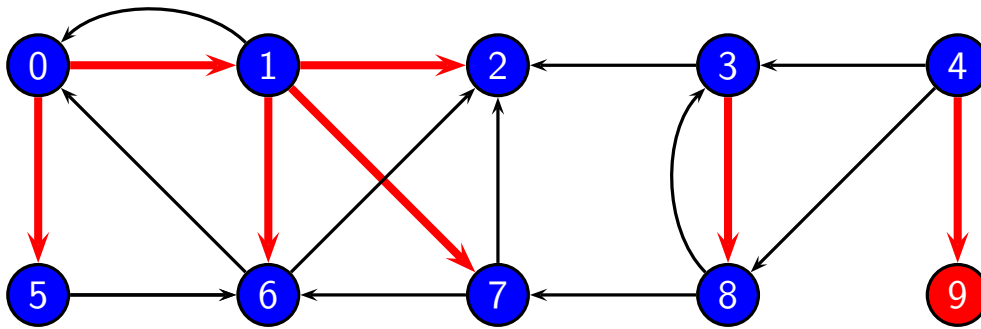


Die Adjazenzlisten sind nach den Zielknoten sortiert.

- A entdeckt
- B bearbeitet

Queue									4
entdeckte Knoten	0	1	5	2	6	7	3	8	
Level	0	1	1	2	2	2	0	1	0
Vorgänger	0	0	0	1	1	1	3	3	4

# Breitensuche

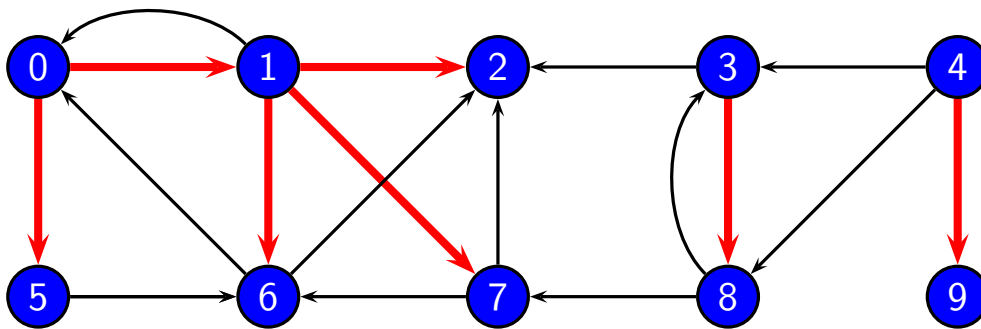


Die Adjazenzlisten sind nach den Zielknoten sortiert.

- A entdeckt
- B bearbeitet

Queue										9
entdeckte Knoten	0	1	5	2	6	7	3	8	4	
Level	0	1	1	2	2	2	0	1	0	
Vorgänger	0	0	0	1	1	1	3	3	4	

# Breitensuche



Die Adjazenzlisten sind nach den Zielknoten sortiert.

- A entdeckt
- B bearbeitet

Queue										
entdeckte Knoten	0	1	5	2	6	7	3	8	4	9
Level	0	1	1	2	2	2	0	1	0	1
Vorgänger	0	0	0	1	1	1	3	3	4	4

# Die globale Breitensuche

## Achtung!

Im Baum mit Wurzel  $v$  stehen nicht alle von  $v$  aus erreichbaren Knoten!

Er enthält nur alle Knoten aus  $R_v$ , die nicht bei der Breitensuche von einem vorherigen Knoten aus besucht wurden.

Die Reihenfolge der Knoten bei der globalen Breitensuche sei durch  $<$  gegeben. Dann enthält der Baum mit Wurzel  $v$  genau die Knoten

$$R_v \setminus \bigcup_{u < v} R_u.$$

## Tiefensuche

# Tiefensuche

Während die Breitensuche im Wesentlichen dem Levelorder-Durchlauf eines Baumes entspricht und zur Berechnung der Distanz vom Ausgangsknoten dienen kann, versucht die **Tiefensuche** Strukturinformationen zu Sammeln (Kreisfreiheit, Erreichbarkeit etc.)

## Grundidee der Tiefensuche

Während der **Tiefensuche** werden die Knoten rekursiv besucht.

Wird gerade  $v$  besucht, so werden alle seine Nachfolger  $u_1, \dots, u_l$  nacheinander bearbeitet.

Ist  $u_i$  bereits früher **entdeckt** worden, dann wird er ignoriert.

Ist  $u_i$  **neu**, dann wird **sofort** eine Tiefensuche auf  $u_i$  gestartet. Die übrigen Nachbarn werden erst später bearbeitet.

# Tiefensuche

## Effekt

Vorwärtsgehen hat Vorrang vor dem Entdecken von **Peers**, d.h. Knoten mit gleichem Vorgänger.

Die Tiefensuche geht vorrangig **in die Tiefe**

# Die Realisierung

Um eine Tiefensuche zu realisieren, werden zwei Knoten-Zustände benötigt:

- **neu** – der Knoten wurde noch nicht entdeckt
- **entdeckt** – der Knoten wurde bereits entdeckt

Da es häufig von Nutzen ist zu wissen, welche Knoten bereits vollständig bearbeitet wurden und bei welchen Knoten noch Nachfolger bearbeitet werden müssen, wird der Zustand **entdeckt** häufig in zwei Zustände aufgeteilt:

- **aktiv** – entdeckt, aber es sind noch nicht alle Nachfolger bearbeitet worden
- **fertig** – die Nachfolger wurden vollständig bearbeitet

Der Zustand eines Knotens  $v$  wird in **status**[ $v$ ] gespeichert.

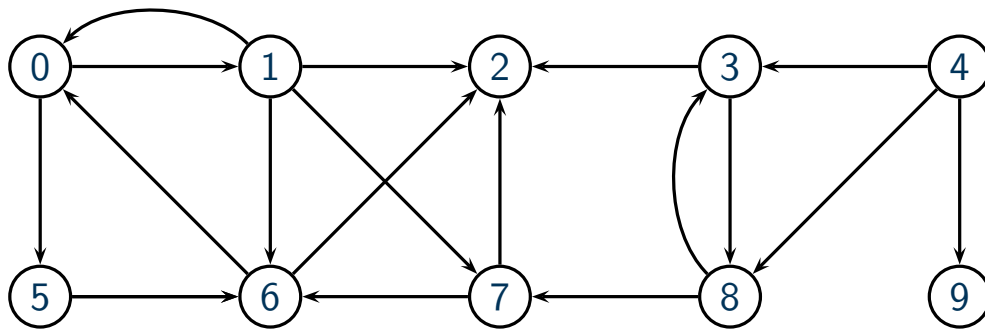
## Tiefensuche

```
dfs( $v$ , dfs_count)
```

```
dfs_count ++;  
dfs_num[ $v$ ] = dfs_count;  
dfs_visit( $v$ ); // Aktion bei Entdeckung  
status[ $v$ ] = aktiv;  
für alle Nachfolger  $u$  von  $v$  tue  
  wenn status[ $u$ ] = neu dann  
    dfs( $u$ , dfs_count);  
Ende  
Ende  
status[ $v$ ] = fertig;
```

Zu Beginn gilt für alle Knoten **status**[ $v$ ] = **neu**.

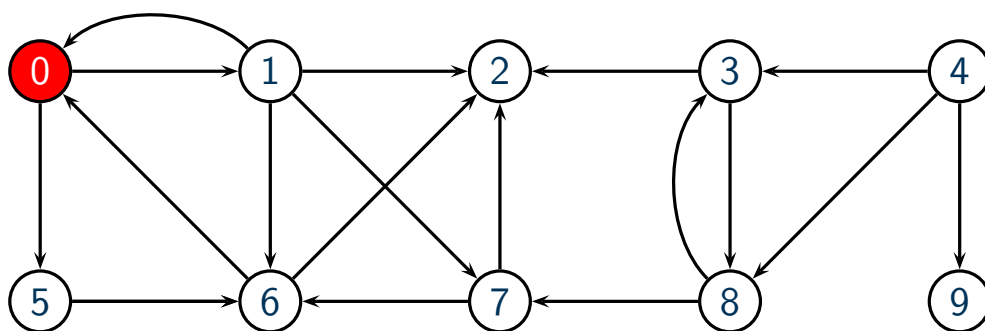
# Tiefensuche



Die Adjazenzlisten sind nach den Zielknoten sortiert.

- A aktiv
- B fertig

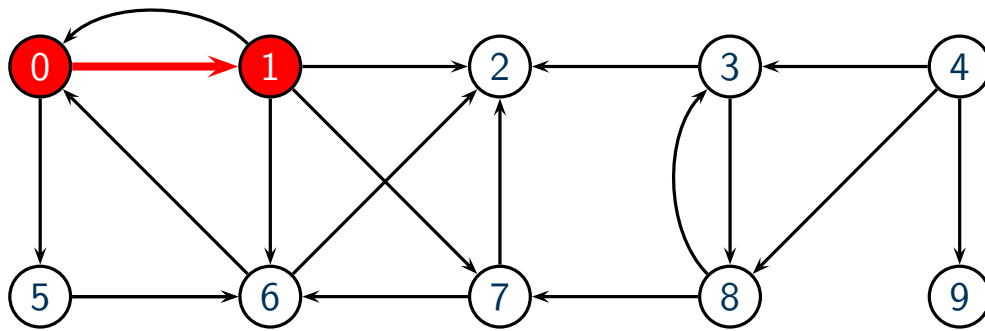
# Tiefensuche



Die Adjazenzlisten sind nach den Zielknoten sortiert.

- A aktiv
- B fertig

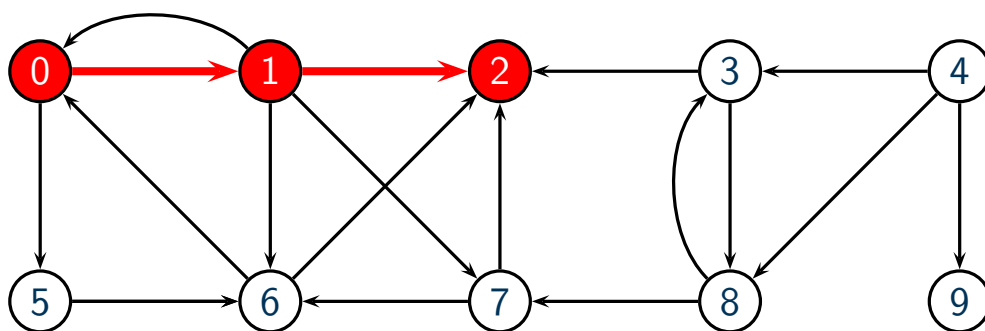
# Tiefensuche



Die Adjazenzlisten sind nach den Zielknoten sortiert.

- A** aktiv
- B** fertig

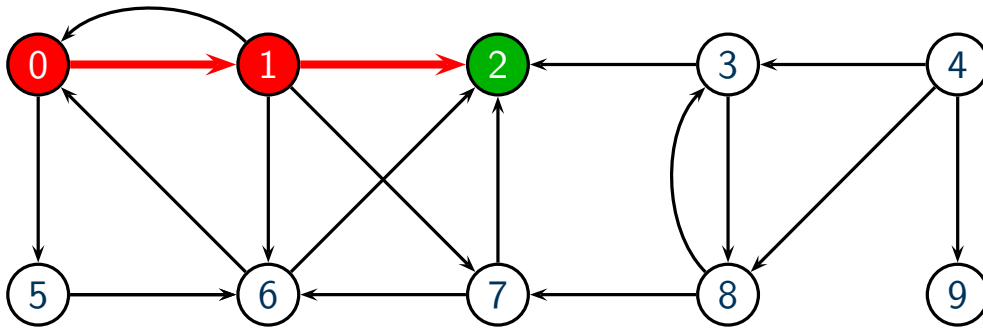
# Tiefensuche



Die Adjazenzlisten sind nach den Zielknoten sortiert.

- A** aktiv
- B** fertig

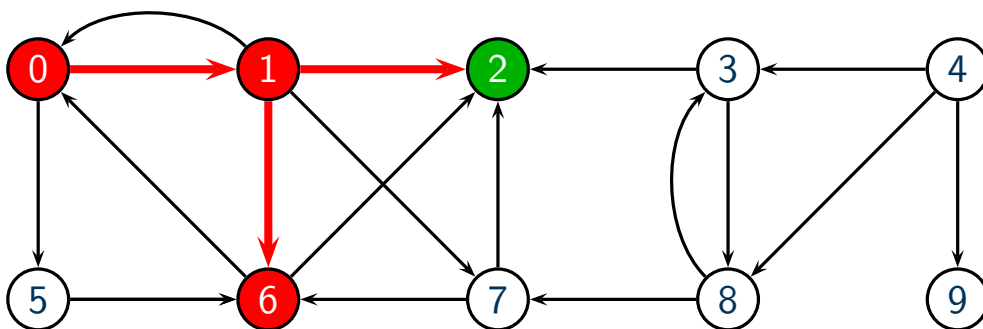
# Tiefensuche



Die Adjazenzlisten sind nach den Zielknoten sortiert.

- A aktiv
- B fertig

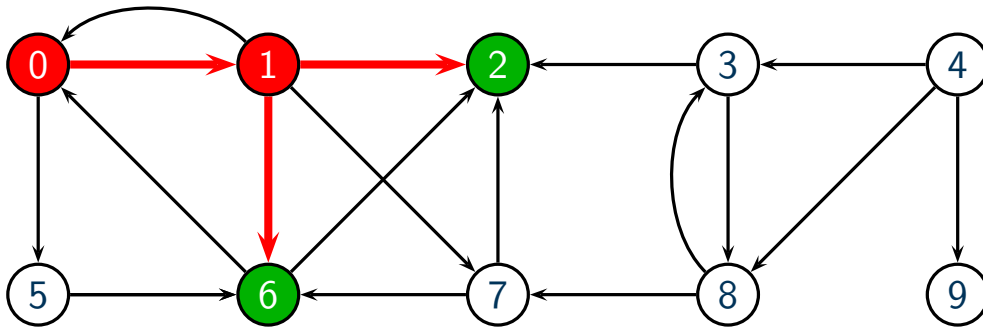
# Tiefensuche



Die Adjazenzlisten sind nach den Zielknoten sortiert.

- A aktiv
- B fertig

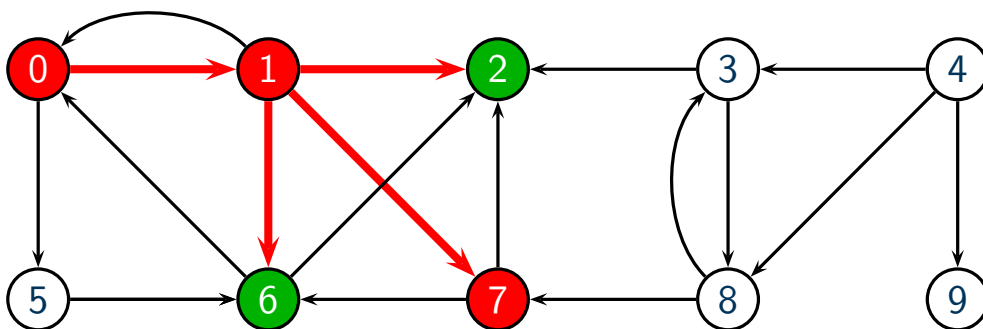
# Tiefensuche



Die Adjazenzlisten sind nach den Zielknoten sortiert.

- A** aktiv
- B** fertig

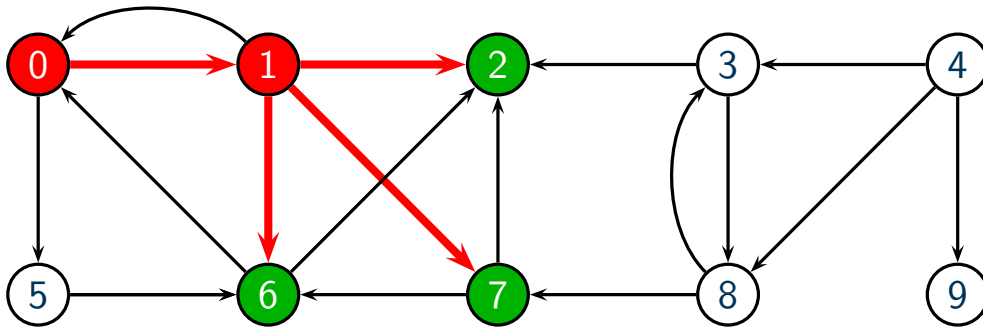
# Tiefensuche



Die Adjazenzlisten sind nach den Zielknoten sortiert.

- A** aktiv
- B** fertig

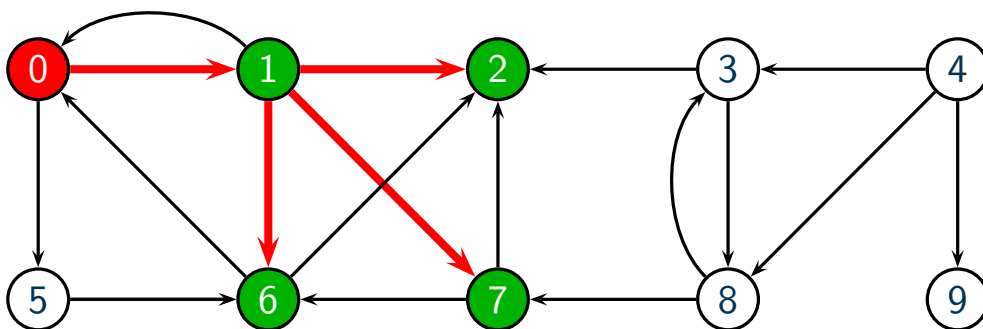
# Tiefensuche



Die Adjazenzlisten sind nach den Zielknoten sortiert.

- A** aktiv
- B** fertig

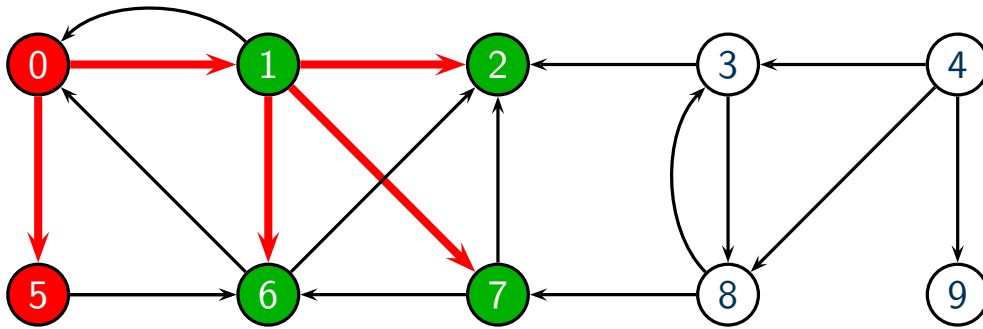
# Tiefensuche



Die Adjazenzlisten sind nach den Zielknoten sortiert.

- A** aktiv
- B** fertig

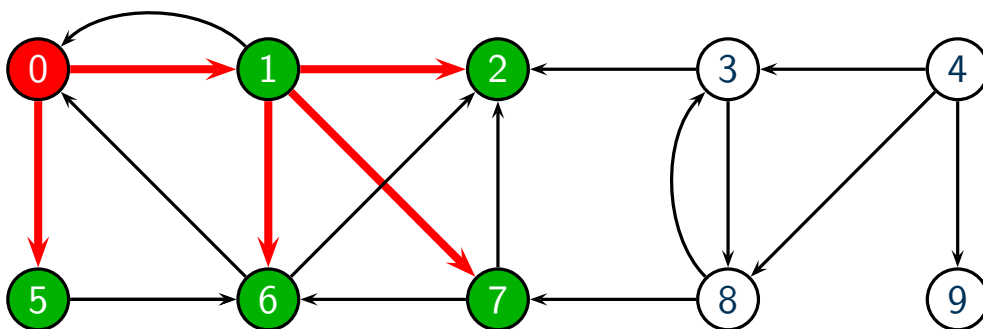
# Tiefensuche



Die Adjazenzlisten sind nach den Zielknoten sortiert.

- A aktiv
- B fertig

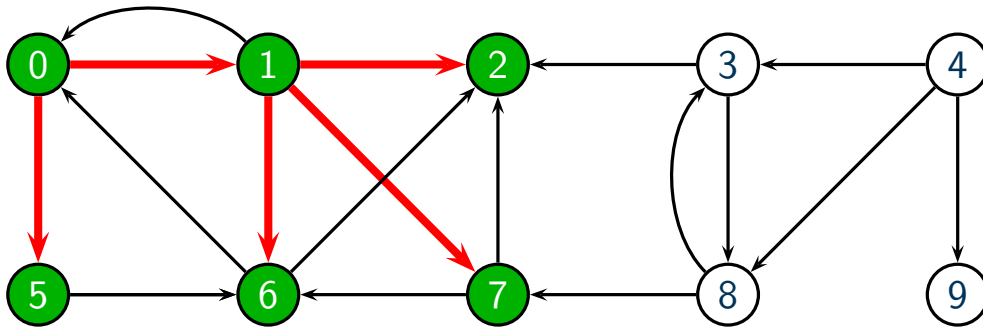
# Tiefensuche



Die Adjazenzlisten sind nach den Zielknoten sortiert.

- A aktiv
- B fertig

# Tiefensuche



Die Adjazenzlisten sind nach den Zielknoten sortiert.



# Tiefensuche

## Beobachtung

$v_0$  sei der Startknoten der Tiefensuche.

- Für jeden von  $v_0$  aus erreichbaren Knoten wird dfs genau einmal aufgerufen.
- Jeder von  $v_0$  aus erreichbare Knoten wird von einem eindeutig bestimmten Knoten aus **entdeckt**. Diesen nennen wir  $p[V]$ .
- Der Aufruf von  $\text{dfs}(v)$  erfolgt während des Aufrufens von  $\text{dfs}(p(v))$ .
- **Insbesondere** beginnt  $\text{dfs}(v)$  nach und endet vor  $\text{dfs}(p(v))$ .

Damit bilden die Kanten  $(p(v), v)$  wieder einen Baum mit Wurzel  $v_0$ , den **Tiefensuchbaum**.

# Tiefensuche

Der Zeitaufwand für die Bearbeitung eines Aufrufes von  $\text{dfs}(v)$  ist – ohne die rekursiven Aufrufe

$$\mathcal{O}(1) + \mathcal{O}(\text{outdeg}(v)).$$

## Lemma

Der Zeitaufwand für  $\text{dfs}(v_0)$  ist

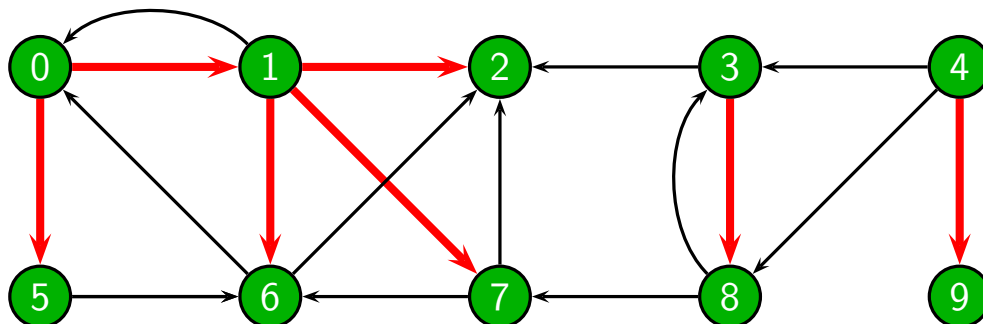
$$\mathcal{O}\left(\sum_{v \in R_{v_0}} (1 + \text{outdeg}(v))\right) = \mathcal{O}(|E_0|)$$

wobei  $E_0$  die Menge aller von  $v_0$  erreichbaren Kanten ist.

## Globale Tiefensuche

Analog zur Breitensuche, findet die Tiefensuche von einem Knoten ausgehend nur einen Teil des Graphen.

Eine globale Version kann analog zur Breitensuche realisiert werden.



## Satz

Die Globale Tiefensuche  $\text{DFS}(G)$  benötigt Zeit

$$\mathcal{O}\left(\sum_{v \in V} (1 + \text{outdeg}(v))\right) = \mathcal{O}(|V| + |E|).$$

# Test auf Kreisfreiheit

## Test auf Kreisfreiheit

### Frage

Besitzt ein gerichteter Graph  $G = (V, E)$  einen Kreis?

Besitzt  $G$  einen Kreis, der von  $v_0$  aus erreichbar ist?

# Test auf Kreisfreiheit

Um einen Kreis zu entdecken, müssen wir lediglich die Behandlung von bereits **entdeckten** Knoten anpassen:

## Veränderte Behandlung bereits entdeckter Knoten

[...]

für alle Nachfolger  $u$  von  $v$  tue

wenn  $\text{status}[u] = \text{aktiv}$  dann

$\text{KreisGesehen} = \text{true};$

Ende

sonst wenn  $\text{status}[u] = \text{neu}$  dann

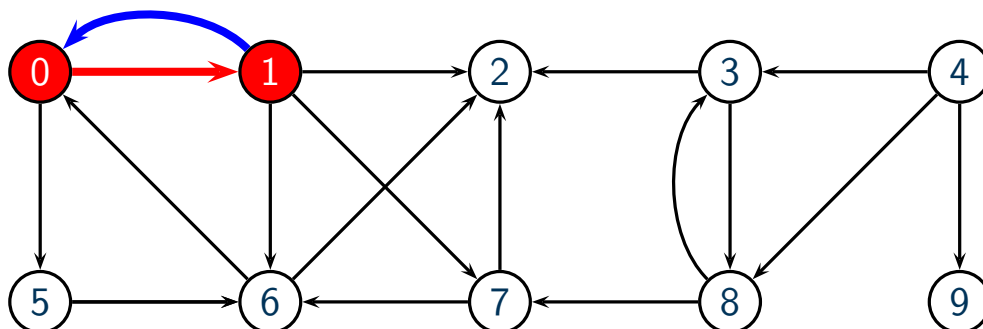
$\text{dfs}(u);$

Ende

Ende

[...]

# Test auf Kreisfreiheit



Die Adjazenzlisten sind nach den Zielknoten sortiert.

-  aktiv
-  fertig

Während der Ausführung von  $\text{dfs}(2)$  wird die Kante  $(2, 1)$  überprüft. Dabei wird der **aktive** Knoten 1 wieder „**entdeckt**“, und somit ein Kreis gefunden.

# Test auf Kreisfreiheit

## Satz

Der von  $v_0$  aus erreichbare Teil  $R_{v_0}$  von  $G$  enthält genau dann einen Kreis, wenn nach dem Aufruf von  $\text{dfs}(v_0)$  das Flag  $\text{KReisGesehen}$  den Wert `true` enthält.

## Beobachtung

Die **aktiven** Knoten bilden stets einen Weg von  $v_0$  zu dem aktuell bearbeiteten Knoten  $u$ .

Dieser Weg entspricht dem Weg von  $v_0$  zu  $u$  im resultierenden Tiefensuchbaum.

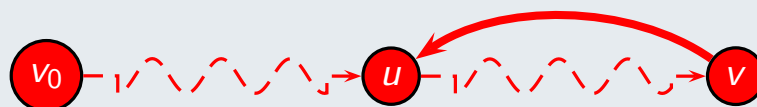
# Test auf Kreisfreiheit

## Beweis des Satzes

„ $\Leftarrow$ “

Wenn am Ende von  $\text{dfs}(v_0)$  das Flag  $\text{KReisGesehen}$  gesetzt ist (d.h. den Wert `true` hat), dann trat irgendwann einmal die folgende Situation auf:

$\text{dfs}(v)$  wird aktuell bearbeitet. Dabei wird die Kante  $(v, u)$  zu einem **aktiven** Knoten  $u$  betrachtet.



Da  $u$  **aktiv** ist, liegt er auf dem Weg von  $v_0$  zu  $v$  und somit existiert ein Kreis, auf dem  $v$  und  $u$  liegen.

...

# Test auf Kreisfreiheit

## Beweis des Satzes (Fortsetzung)

„ $\Rightarrow$ “

Wir gehen davon aus, dass für keinen Knoten  $v$  ein **aktiver** Nachfolger gefunden wird. D.h. `KreisGesehen` hat am Ende den Wert `false`.

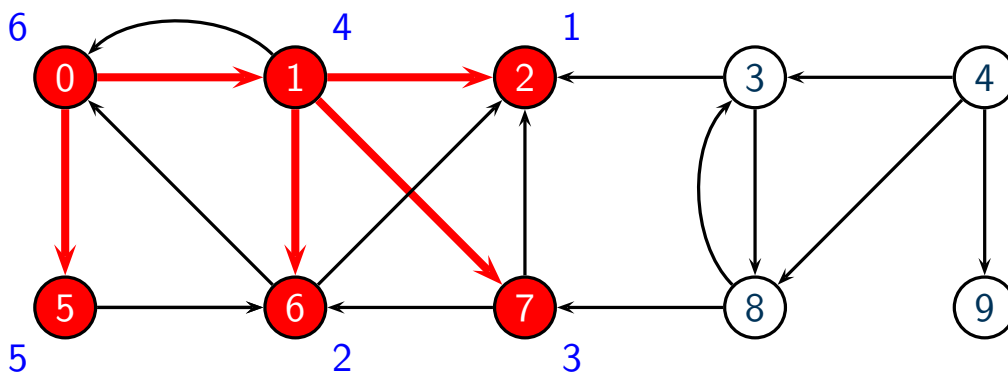
Wir werden im Folgenden zeigen, dass  $G$  dann keinen Kreis hat.

Dazu nummerieren wir die Knoten in  $R_{v_0}$  in der Reihenfolge durch, in der die `dfs(v)`-Aufrufe **beendet** werden.

Diese Nummern nennen wir **f-Nummern**:  $f\_num(v) \in \{1, \dots, |V|\}$ .

...

# Test auf Kreisfreiheit



Die Adjazenzlisten sind nach den Zielknoten sortiert.

-  aktiv
-  fertig

Ein zyklischer Graph mit f-Nummern.

# Test auf Kreisfreiheit

## Behauptung

Wenn  $(v, u) \in E$ , dann ist  $f\_num(v) > f\_num(u)$ .

## Beweis der Behauptung

Die Kante  $(v, u)$  wird während der Ausführung von  $dfs(v)$  gemustert.

**1.Fall:**  $status[u] = \text{neu}$

Dann wird  $dfs(u)$  rekursiv aus  $dfs(v)$  aufgerufen und endet somit vorher, was die Behauptung impliziert.

**2.Fall:**  $status[u] \neq \text{neu}$

Damit muss  $status[v] = \text{fertig}$  sein, denn  $\text{KreisGesehen}$  wurde nicht gesetzt.

Damit ist  $dfs(u)$  bereits beendet, wenn  $dfs(v)$  aufgerufen wird. □

# Test auf Kreisfreiheit

## Beweis des Satzes (Fortsetzung)

Der Behauptung zu Folge gilt

$$(v, u) \in E \Rightarrow f\_num(v) > f\_num(u),$$

und somit fallen die f-Nummern entlang eines Kreises streng monoton.

Das impliziert, dass es keinen Kreis geben kann. □

## Satz

Ein gerichteter Graph  $G = (V, E)$  kann in Zeit  $\mathcal{O}(|V| + |E|)$  auf Kreisfreiheit getestet werden.

## Topologische Sortierung

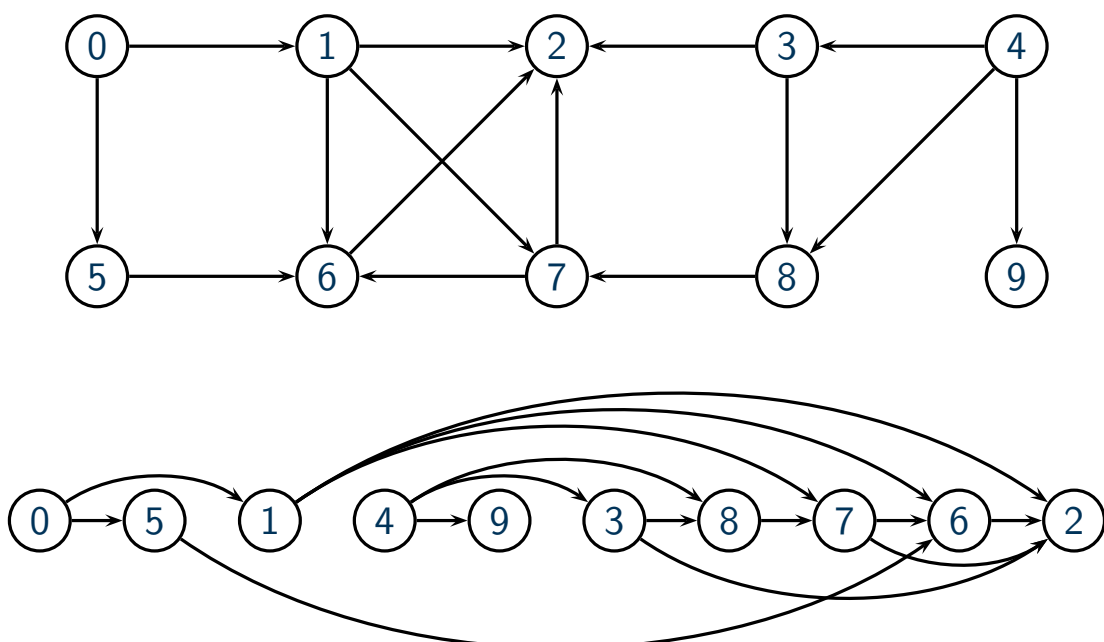
# Topologische Sortierung

Wie die f-Nummern gezeigt haben, ist es möglich, in einem **DAG** (**gerichteten azyklischen Graphen**) die Knoten so zu ordnen, dass jede Kante nur von „links nach rechts“ verläuft. Oder etwas präziser:

## Definition (Topologische Sortierung)

Eine **topologische Sortierung** eines DAG  $G = (V, E)$  ist eine Bijektion  $\pi: V \rightarrow \{1, \dots, |V|\}$ , so dass  $\pi(v) < \pi(u)$  für jede Kante  $(v, u)$  gilt.

# Topologische Sortierung



## Die f-Nummern

Wie wir bereits gesehen haben, fallen die f-Nummern in einem DAG entlang einer Kante strikt.

Außerdem stellen die f-Nummern eine Bijektion  $f\_num: V \rightarrow \{1, \dots, |V|\}$  dar.

Damit wäre  $\pi(v) := (|V| + 1) - f\_num(v)$  eine **topologische Sortierung**.

Um die f-Nummern zu ermitteln, müssen wir die Knoten nur in der Reihenfolge durchnummerieren, in der ihr Status auf **fertig** gesetzt wird.

D.h. wir ergänzen dfs direkt um einen Zähler `f_count`, der anfänglich mit 0 initialisiert wird, und die folgende Zeilen:

```
...
status[v] = fertig; // alte Zeile
f_count ++;
f_num[v] = f_count;
```

## Topologische Sortierung

### Satz

Ein gerichteter Graph  $G = (V, E)$  kann in Zeit  $\mathcal{O}(|V| + |E|)$  auf Kreisfreiheit getestet werden. Gleichzeitig kann eine Topologische Sortierung berechnet werden, wenn  $G$  azyklisch ist.