SS09

# Effiziente Algorithmen

# 2. Kapitel: Graphdurchläufe

Martin Dietzfelbinger

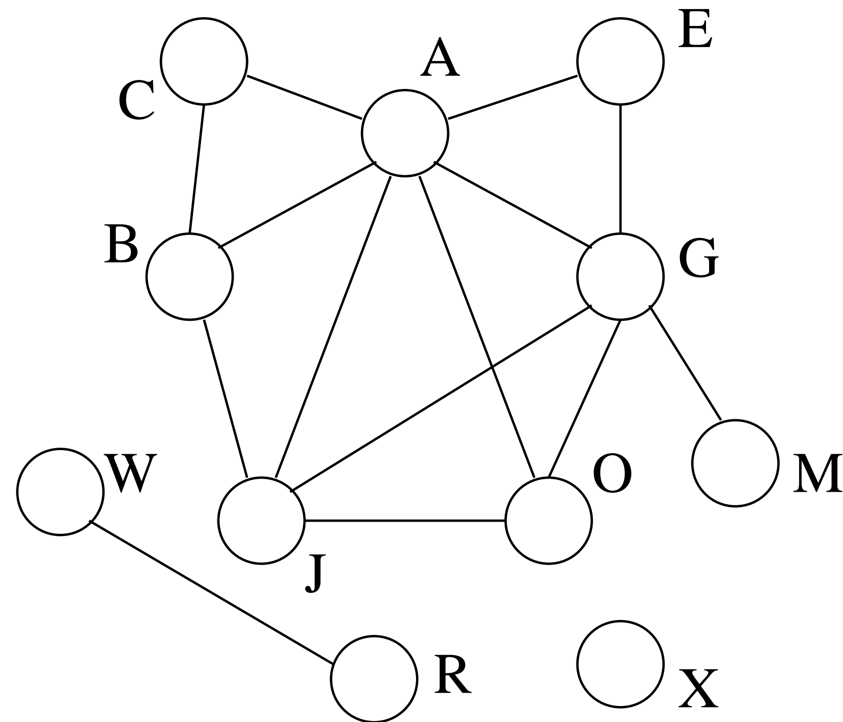
April/Mai 2009

# Kapitel 2 Durchsuchen und Strukturanalyse von Graphen

## (Ungerichteter) Graph

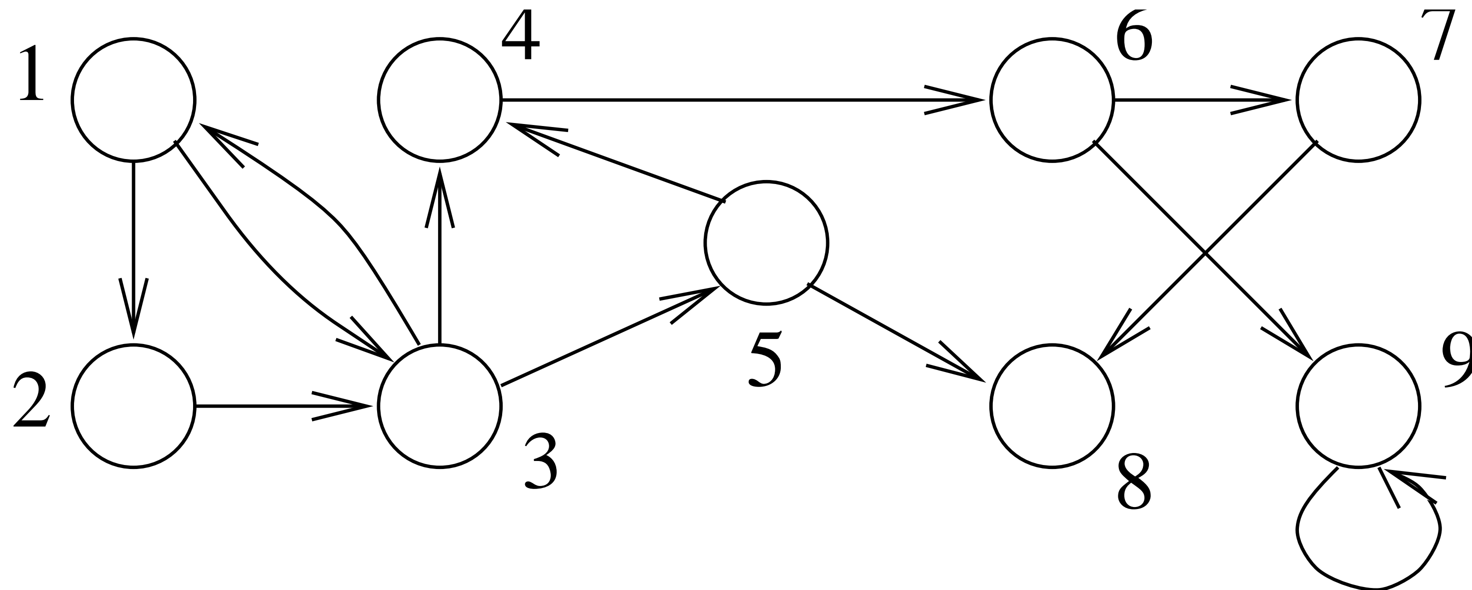
# Kapitel 2 Durchsuchen und Strukturanalyse von Graphen

## (Ungerichteter) Graph



# Kapitel 2 Durchsuchen und Strukturanalyse von Graphen

**Gerichteter Graph**



# Kapitel 2 Durchsuchen und Strukturanalyse von Graphen

**Erinnerung** (siehe

http://www.tu-ilmenau.de/fakia/fileadmin/template/startIA/ktea/Lehre/a_d/ss08/AuD-080630-4auf1.pdf

- Graphen, gerichtete Graphen (Digraphen)

# Kapitel 2 Durchsuchen und Strukturanalyse von Graphen

**Erinnerung** (siehe

http://www.tu-ilmenau.de/fakia/fileadmin/template/startIA/ktea/
Lehre/a_d/ss08/AuD-080630-4auf1.pdf

- Graphen, gerichtete Graphen (Digraphen)
- Grad, Ingrad, Ausgrad

# Kapitel 2 Durchsuchen und Strukturanalyse von Graphen

**Erinnerung** (siehe
http://www.tu-ilmenau.de/fakia/fileadmin/template/startIA/ktea/
Lehre/a_d/ss08/AuD-080630-4auf1.pdf

- Graphen, gerichtete Graphen (Digraphen)
- Grad, Ingrad, Ausgrad
- Wege (gerichtet/ungerichtet), einfache Wege

# Kapitel 2 Durchsuchen und Strukturanalyse von Graphen

**Erinnerung** (siehe

`http://www.tu-ilmenau.de/fakia/fileadmin/template/startIA/ktea/Lehre/a_d/ss08/AuD-080630-4auf1.pdf`

- Graphen, gerichtete Graphen (Digraphen)
- Grad, Ingrad, Ausgrad
- Wege (gerichtet/ungerichtet), einfache Wege
- Kreise (oder: Zyklen) (gerichtet/ungerichtet)

# Kapitel 2 Durchsuchen und Strukturanalyse von Graphen

**Erinnerung** (siehe

http://www.tu-ilmenau.de/fakia/fileadmin/template/startIA/ktea/Lehre/a_d/ss08/AuD-080630-4auf1.pdf

- Graphen, gerichtete Graphen (Digraphen)
- Grad, Ingrad, Ausgrad
- Wege (gerichtet/ungerichtet), einfache Wege
- Kreise (oder: Zyklen) (gerichtet/ungerichtet)
- azyklische Digraphen

# Kapitel 2 Durchsuchen und Strukturanalyse von Graphen

**Erinnerung** (siehe

`http://www.tu-ilmenau.de/fakia/fileadmin/template/startIA/ktea/Lehre/a_d/ss08/AuD-080630-4auf1.pdf`

- Graphen, gerichtete Graphen (Digraphen)
- Grad, Ingrad, Ausgrad
- Wege (gerichtet/ungerichtet), einfache Wege
- Kreise (oder: Zyklen) (gerichtet/ungerichtet)
- azyklische Digraphen
- kreisfreie ungerichtete Graphen: Bäume und Wälder

# Kapitel 2 Durchsuchen und Strukturanalyse von Graphen

## Erinnerung

- Zusammenhängende Graphen

# Kapitel 2 Durchsuchen und Strukturanalyse von Graphen

## Erinnerung

- Zusammenhängende Graphen
- Zusammenhangskomponenten

# Kapitel 2 Durchsuchen und Strukturanalyse von Graphen

## Erinnerung

- Zusammenhängende Graphen
- Zusammenhangskomponenten
- Bäume, Wälder (kreisfreie ungerichtete Graphen)

# Kapitel 2 Durchsuchen und Strukturanalyse von Graphen

## Erinnerung

- Zusammenhängende Graphen
- Zusammenhangskomponenten
- Bäume, Wälder (kreisfreie ungerichtete Graphen)
- Gerichtete Bäume und Wälder

# Kapitel 2 Durchsuchen und Strukturanalyse von Graphen

## Erinnerung

- Zusammenhängende Graphen
- Zusammenhangskomponenten
- Bäume, Wälder (kreisfreie ungerichtete Graphen)
- Gerichtete Bäume und Wälder
- Darstellung über Adjazenzmatrix

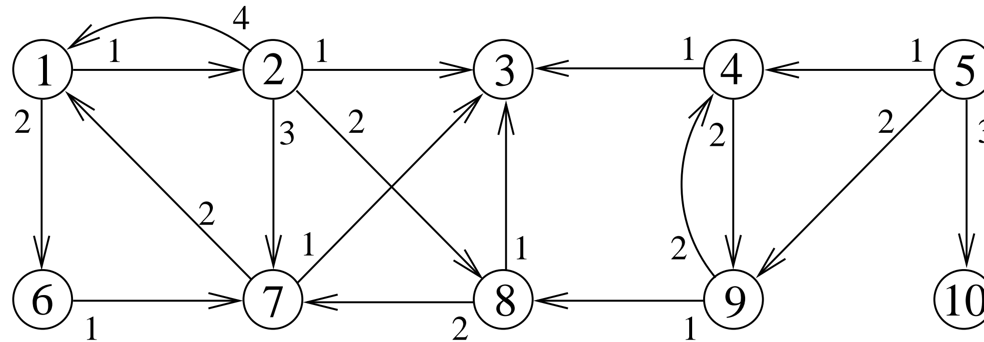
# Kapitel 2 Durchsuchen und Strukturanalyse von Graphen

## Erinnerung

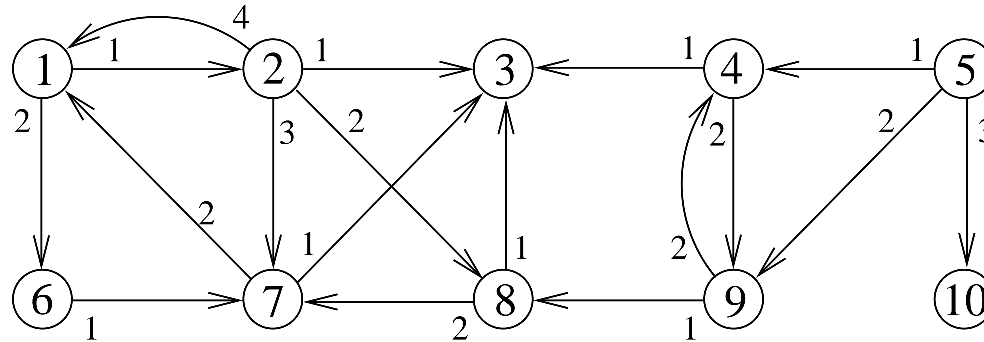
- Zusammenhängende Graphen
- Zusammenhangskomponenten
- Bäume, Wälder (kreisfreie ungerichtete Graphen)
- Gerichtete Bäume und Wälder
- Darstellung über Adjazenzmatrix
- Darstellung über Adjazenzlisten
  (einfach oder doppelt verkettet)

# Tiefensuche in Graphen/Digraphen

# Tiefensuche in Graphen/Digraphen



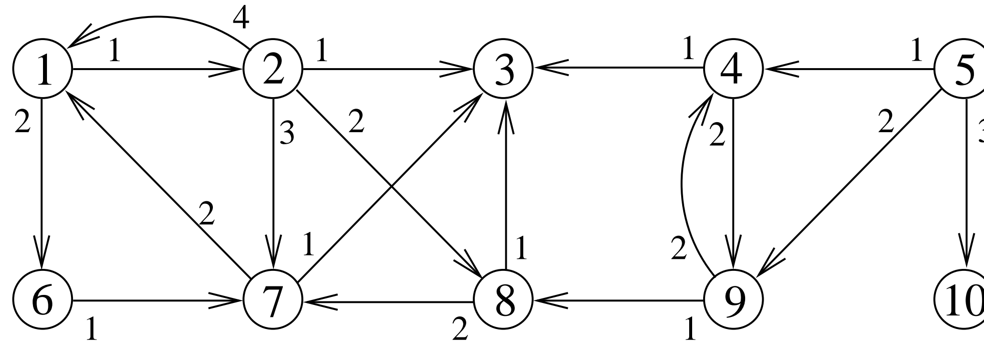
# Tiefensuche in Graphen/Digraphen




**Eingabeformat:**

Graph/Digraph $G = (V, E)$, Adjazenzlisten- oder Adjazenzmatrixformat, Knotenmenge $V = \{1, \ldots, n\}$,

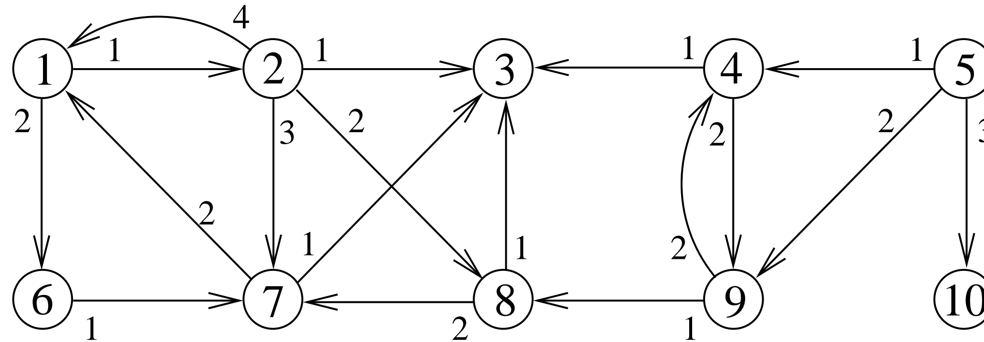
# Tiefensuche in Graphen/Digraphen




**Eingabeformat:**

Graph/Digraph $G = (V, E)$, Adjazenzlisten- oder Adjazenz-matrixformat, Knotenmenge $V = \{1, \ldots, n\}$, geordnete Kantenlisten.

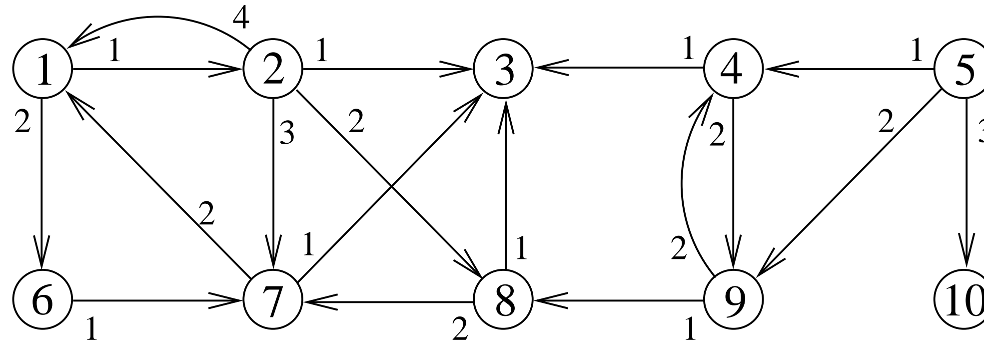
# Tiefensuche in Graphen/Digraphen




**Eingabeformat:**
Graph/Digraph $G = (V, E)$, Adjazenzlisten- oder Adjazenzmatrixformat, Knotenmenge $V = \{1, \ldots, n\}$, geordnete Kantenlisten. — **Ziele:**

- Besuche alle Knoten und Kanten.

# Tiefensuche in Graphen/Digraphen




**Eingabeformat:**

Graph/Digraph $G = (V, E)$, Adjazenzlisten- oder Adjazenzmatrixformat, Knotenmenge $V = \{1, \ldots, n\}$, geordnete Kantenlisten. — **Ziele:**

- Besuche alle Knoten und Kanten.
- Sammle Strukturinformation

## **Tiefensuche** (Depth-First-Search)

von Knoten $v$ aus: „**dfs**$(v)$“

## **Tiefensuche** (Depth-First-Search)

von Knoten $v$ aus: „**dfs**$(v)$“

Grundansatz, rekursiv: Besuche Knoten $v$.

**Tiefensuche** (**D**epth-**F**irst-**S**earch)

von Knoten $v$ aus: „**dfs**$(v)$“

Grundansatz, rekursiv: Besuche Knoten $v$.

Bearbeite die Nachbarn $w_1, \ldots, w_{\text{outdeg}(v)}$ von $v$ nacheinander, aber:

## **Tiefensuche** (Depth-First-Search)

von Knoten $v$ aus: „**dfs**$(v)$“

Grundansatz, rekursiv: Besuche Knoten $v$.

Bearbeite die Nachbarn $w_1, \ldots, w_{\mathrm{outdeg}(v)}$ von $v$ nacheinander, aber:

Sobald neuer Knoten $w$ „entdeckt“ wird, starte **sofort dfs**$(w)$.

## **Tiefensuche** (Depth-First-Search)

von Knoten $v$ aus: „**dfs**$(v)$“

Grundansatz, rekursiv: Besuche Knoten $v$.

Bearbeite die Nachbarn $w_1, \ldots, w_{\mathrm{outdeg}(v)}$ von $v$ nacheinander, aber:

Sobald neuer Knoten $w$ „entdeckt“ wird, starte **sofort dfs**$(w)$.

(Weitere Nachbarn von $v$ kommen später dran.)

**Tiefensuche** (**D**epth-**F**irst-**S**earch)

von Knoten $v$ aus: „**dfs**$(v)$"

Grundansatz, rekursiv: Besuche Knoten $v$.

Bearbeite die Nachbarn $w_1, \ldots, w_{\mathrm{outdeg}(v)}$ von $v$ nacheinander, aber:

Sobald neuer Knoten $w$ „entdeckt" wird, starte **sofort dfs**$(w)$.

(Weitere Nachbarn von $v$ kommen später dran.)

**Vorwärtsgehen** hat **Vorrang**!

**Tiefensuche** (**D**epth-**F**irst-**S**earch)

von Knoten $v$ aus: „**dfs**$(v)$“

Grundansatz, rekursiv: Besuche Knoten $v$.

Bearbeite die Nachbarn $w_1, \ldots, w_{\mathrm{outdeg}(v)}$ von $v$ nacheinander, aber:

Sobald neuer Knoten $w$ „entdeckt“ wird, starte **sofort dfs**$(w)$.

(Weitere Nachbarn von $v$ kommen später dran.)

**Vorwärtsgehen** hat **Vorrang**!

**Effekt:** Die Folge der entdeckten Knoten geht vorrangig „in die Tiefe“.

Man muss verhindern, dass Knoten $v$ mehrfach entdeckt wird.

Man muss verhindern, dass Knoten $v$ mehrfach entdeckt wird.

Dazu: „Statusinformation“ *neu*, *aktiv*, *fertig*.

$v$ *neu*: weiß — noch nie gesehen

Man muss verhindern, dass Knoten $v$ mehrfach entdeckt wird.

Dazu: „Statusinformation“ *neu*, *aktiv*, *fertig*.

$v$ *neu*: weiß — noch nie gesehen
$v$ *aktiv*: rot — **dfs**$(v)$ gestartet, noch nicht beendet

Man muss verhindern, dass Knoten $v$ mehrfach entdeckt wird.

Dazu: „Statusinformation“ *neu*, *aktiv*, *fertig*.

$v$ *neu*: weiß — noch nie gesehen

$v$ *aktiv*: rot — **dfs**$(v)$ gestartet, noch nicht beendet

$v$ *fertig*: grau — **dfs**$(v)$ beendet

Man muss verhindern, dass Knoten $v$ mehrfach entdeckt wird.

Dazu: „Statusinformation“ *neu*, *aktiv*, *fertig*.

$v$ *neu*: weiß — noch nie gesehen
$v$ *aktiv*: rot — **dfs**$(v)$ gestartet, noch nicht beendet
$v$ *fertig*: grau — **dfs**$(v)$ beendet

Status wird in einem Array $\texttt{status}[1..n]$ gehalten —
ein Eintrag pro Knoten.

Man muss verhindern, dass Knoten $v$ mehrfach entdeckt wird.

Dazu: „Statusinformation“ *neu*, *aktiv*, *fertig*.

$v$ *neu*: weiß — noch nie gesehen
$v$ *aktiv*: rot — **dfs**$(v)$ gestartet, noch nicht beendet
$v$ *fertig*: grau — **dfs**$(v)$ beendet

Status wird in einem Array `status`$[1..n]$ gehalten — ein Eintrag pro Knoten.

Parallel zu (oder Teil von) Knotenarray `nodes`$[1..n]$

Man muss verhindern, dass Knoten $v$ mehrfach entdeckt wird.

Dazu: „Statusinformation“ *neu*, *aktiv*, *fertig*.

$v$ *neu*: weiß — noch nie gesehen
$v$ *aktiv*: rot — **dfs**$(v)$ gestartet, noch nicht beendet
$v$ *fertig*: grau — **dfs**$(v)$ beendet

Status wird in einem Array `status`$[1\,..\,n]$ gehalten — ein Eintrag pro Knoten.

Parallel zu (oder Teil von) Knotenarray `nodes`$[1\,..\,n]$

Initialisierung: Alle Knoten sind „*neu*“.

Knoten werden in der Reihenfolge der Entdeckung durchnummeriert:

Knoten werden in der Reihenfolge der Entdeckung durchnummeriert:

`dfs_num`$[1 \mathinner{\ldotp\ldotp} n]$.

Knoten werden in der Reihenfolge der Entdeckung durchnummeriert:

`dfs_num`$[1 \,.\,.\, n]$. **Tiefensuch-Nummerierung**.

Knoten werden in der Reihenfolge der Entdeckung durchnummeriert:

`dfs_num`$[1\,..\,n]$. **Tiefensuch-Nummerierung**.

Mitzählen: in `dfs_count`, mit $0$ initialisiert.

Knoten werden in der Reihenfolge der Entdeckung durchnummeriert:

`dfs_num`$[1\,..\,n]$. **Tiefensuch-Nummerierung**.

Mitzählen: in `dfs_count`, mit $0$ initialisiert.

**dfs-visit**$(v)$: Aktion an $v$ bei Entdeckung, anwendungsabhängig.

Knoten werden in der Reihenfolge der Entdeckung durchnummeriert:

`dfs_num[1..n]`, $\texttt{dfs_num}[1\,..\,n]$. **Tiefensuch-Nummerierung**.

Mitzählen: in `dfs_count`, mit $0$ initialisiert.

**dfs-visit**$(v)$: Aktion an $v$ bei Entdeckung, anwendungsabhängig.

Zunächst: Tiefensuche in **gerichteten Graphen/Digraphen**.

**Algorithmus dfs**$(v)$ $(*$ Tiefensuche von $v$ aus, rekursiv $*)$

**Algorithmus dfs**$(v)$ $(*$ Tiefensuche von $v$ aus, rekursiv $*)$
$(*$ darf nur für $v$ mit $\texttt{status}[v] =$ *neu* aufgerufen werden $*)$

**Algorithmus dfs**$(v)$ (∗ Tiefensuche von $v$ aus, rekursiv ∗)

(∗ darf nur für $v$ mit `status`$[v]$ = *neu* aufgerufen werden ∗)

(1) `dfs_count++`;

(2) `dfs_num`$[v] \leftarrow$ `dfs_count`;

**Algorithmus dfs**$(v)$ $(*$ Tiefensuche von $v$ aus, rekursiv $*)$

$(*$ darf nur für $v$ mit `status`$[v]$ $=$ *neu* aufgerufen werden $*)$

(1) `dfs_count++`;

(2) `dfs_num`$[v] \leftarrow$ `dfs_count`;

(3) **dfs-visit**$(v)$; $(*$ Aktion an $v$ bei Erstbesuch $*)$

**Algorithmus dfs**$(v)$ (∗ Tiefensuche von $v$ aus, rekursiv ∗)

(∗ darf nur für $v$ mit `status[`$v$`]` $=$ *neu* aufgerufen werden ∗)

(1) `dfs_count++`;

(2) `dfs_num[`$v$`]` $\leftarrow$ `dfs_count`;

(3) **dfs-visit**$(v)$; (∗ Aktion an $v$ bei Erstbesuch ∗)

(4) `status[`$v$`]` $\leftarrow$ **aktiv** ;

**Algorithmus dfs**$(v)$ (* Tiefensuche von $v$ aus, rekursiv *)
(* darf nur für $v$ mit `status[`$v$`]` $=$ *neu* aufgerufen werden *)

(1) `dfs_count++;`

(2) `dfs_num[`$v$`]` $\leftarrow$ `dfs_count;`

(3) **dfs-visit**$(v)$; (* Aktion an $v$ bei Erstbesuch *)

(4) `status[`$v$`]` $\leftarrow$ **aktiv** ;

(5) für jeden Nachfolger `w` von $v$ (Adjazenzliste!) tue:

**Algorithmus dfs**$(v)$ (* Tiefensuche von $v$ aus, rekursiv *)

(* darf nur für $v$ mit status[$v$] $=$ *neu* aufgerufen werden *)

(1) dfs_count++;

(2) dfs_num[$v$] $\leftarrow$ dfs_count;

(3) **dfs-visit**$(v)$; (* Aktion an $v$ bei Erstbesuch *)

(4) status[$v$] $\leftarrow$ **aktiv** ;

(5) für jeden Nachfolger w von $v$ (Adjazenzliste!) tue:

(6) **if** status[w] $=$ **neu** (* w wird entdeckt! *) **then**

**Algorithmus dfs**$(v)$ (∗ Tiefensuche von $v$ aus, rekursiv ∗)

(∗ darf nur für $v$ mit `status[`$v$`]` = *neu* aufgerufen werden ∗)

(1) `dfs_count++;`

(2) `dfs_num[`$v$`]` ← `dfs_count;`

(3) **dfs-visit**$(v)$; (∗ Aktion an $v$ bei Erstbesuch ∗)

(4) `status[`$v$`]` ← **aktiv** ;

(5) für jeden Nachfolger `w` von $v$ (Adjazenzliste!) tue:

(6) **if** `status[w]` = **neu** (∗ `w` wird entdeckt! ∗) **then**

(7) **dfs**(`w`);

**Algorithmus dfs**$(v)$ (∗ Tiefensuche von $v$ aus, rekursiv ∗)

(∗ darf nur für $v$ mit `status[`$v$`]` = *neu* aufgerufen werden ∗)

(1) `dfs_count++`;

(2) `dfs_num[`$v$`]` ← `dfs_count`;

(3) **dfs-visit**$(v)$; (∗ Aktion an $v$ bei Erstbesuch ∗)

(4) `status[`$v$`]` ← **aktiv** ;

(5) für jeden Nachfolger `w` von $v$ (Adjazenzliste!) tue:

(6) **if** `status[w]` = **neu** (∗ `w` wird entdeckt! ∗) **then**

(7) **dfs**(`w`);

(8) `status[`$v$`]` ← **fertig** .

**Algorithmus dfs**$(v)$ (∗ Tiefensuche von $v$ aus, rekursiv ∗)
(∗ darf nur für $v$ mit `status[`$v$`]` $=$ *neu* aufgerufen werden ∗)

(1) `dfs_count++;`
(2) `dfs_num[`$v$`]` $\leftarrow$ `dfs_count;`
(3) **dfs-visit**$(v)$; (∗ Aktion an $v$ bei Erstbesuch ∗)
(4) `status[`$v$`]` $\leftarrow$ **aktiv** ;
(5) für jeden Nachfolger `w` von $v$ (Adjazenzliste!) tue:
(6) **if** `status[w]` $=$ **neu** (∗ `w` wird entdeckt! ∗) **then**
(7) **dfs**(`w`);
(8) `status[`$v$`]` $\leftarrow$ **fertig** .

Tiefensuche von $v_0$ aus:
Initialisiere `dfs_count` $\leftarrow 0$.

**Algorithmus dfs**$(v)$ (∗ Tiefensuche von $v$ aus, rekursiv ∗)
(∗ darf nur für $v$ mit `status[`$v$`]` = *neu* aufgerufen werden ∗)

(1) `dfs_count++;`
(2) `dfs_num[`$v$`]` ← `dfs_count;`
(3) **dfs-visit**$(v)$; (∗ Aktion an $v$ bei Erstbesuch ∗)
(4) `status[`$v$`]` ← **aktiv** ;
(5) für jeden Nachfolger `w` von $v$ (Adjazenzliste!) tue:
(6) **if** `status[w]` = **neu** (∗ `w` wird entdeckt! ∗) **then**
(7) **dfs**(`w`);
(8) `status[`$v$`]` ← **fertig** .

Tiefensuche von $v_0$ aus:
Initialisiere `dfs_count` ← 0. Alle Knoten sind „ **neu** “.

**Algorithmus dfs**$(v)$ (∗ Tiefensuche von $v$ aus, rekursiv ∗)

(∗ darf nur für $v$ mit `status[`$v$`]` = *neu* aufgerufen werden ∗)

(1) `dfs_count++`;

(2) `dfs_num[`$v$`]` ← `dfs_count`;

(3) **dfs-visit**$(v)$; (∗ Aktion an $v$ bei Erstbesuch ∗)

(4) `status[`$v$`]` ← **aktiv** ;

(5) für jeden Nachfolger `w` von $v$ (Adjazenzliste!) tue:

(6) **if** `status[w]` = **neu** (∗ `w` wird entdeckt! ∗) **then**

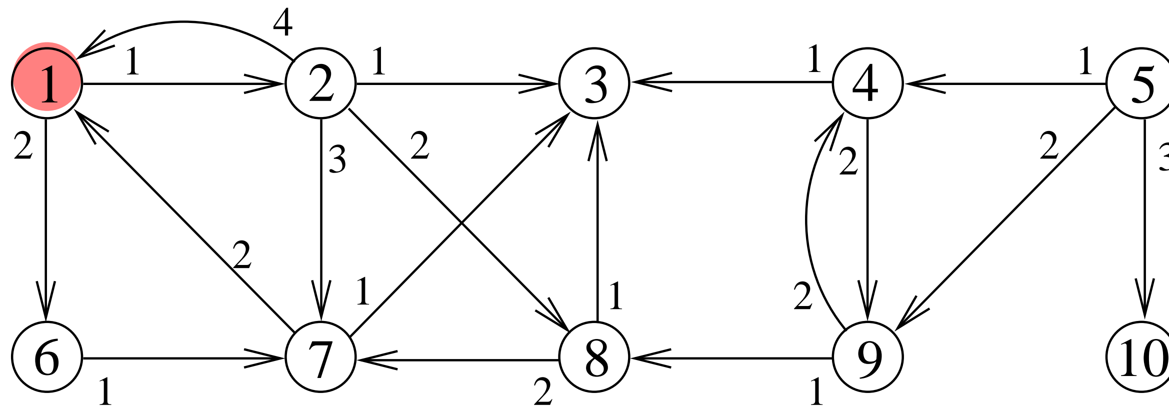
(7) **dfs**(`w`);

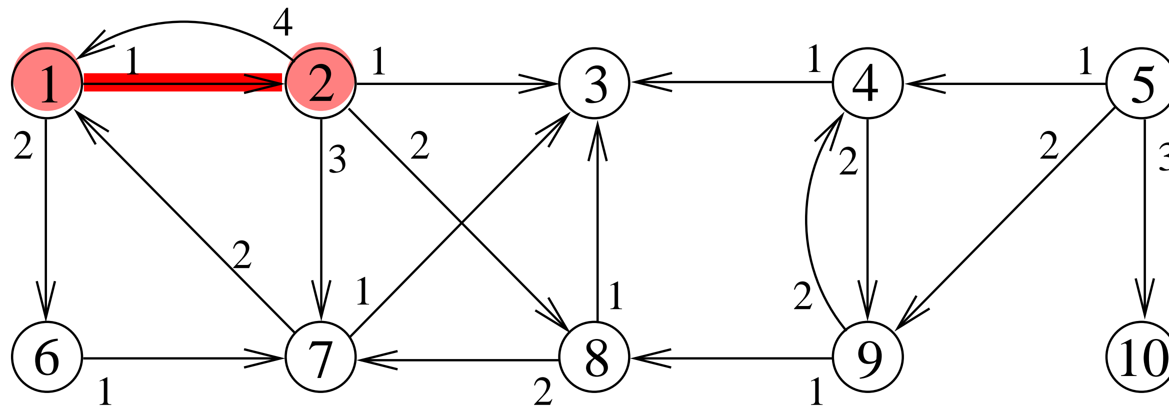
(8) `status[`$v$`]` ← **fertig** .

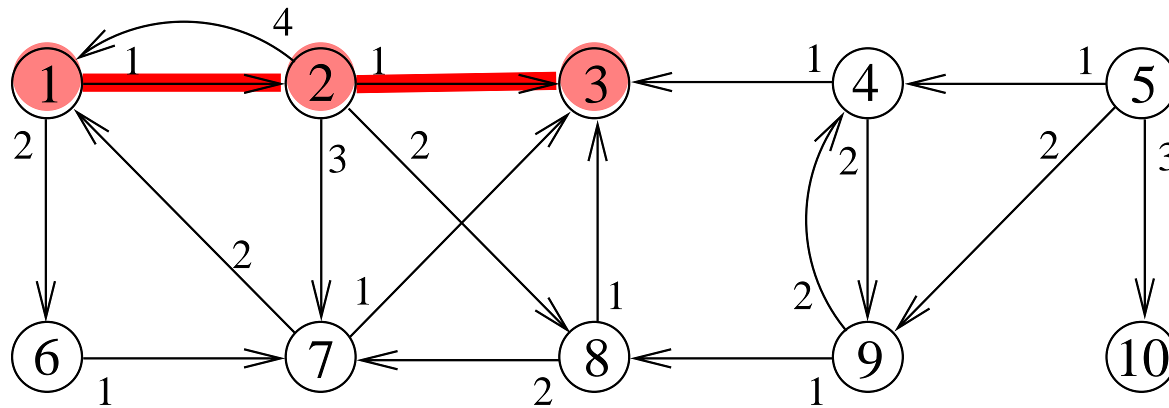
Tiefensuche von $v_0$ aus:

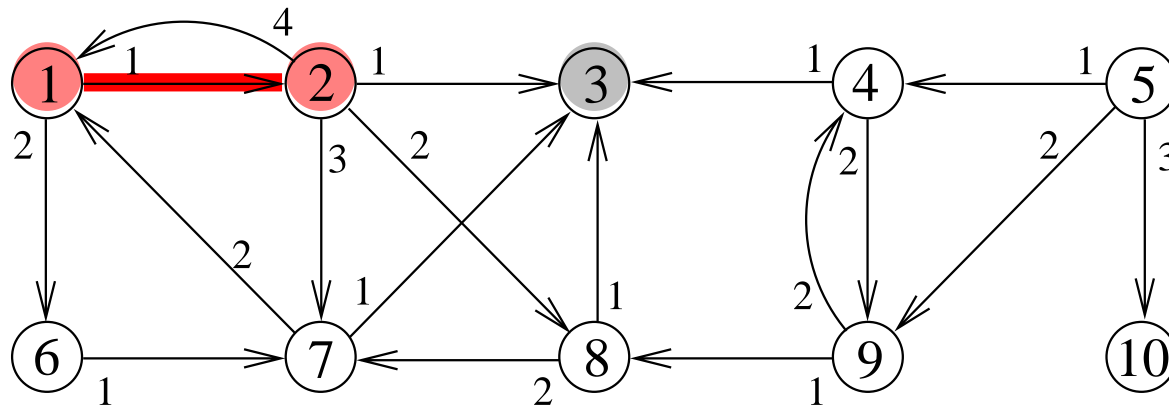
Initialisiere `dfs_count` ← 0. Alle Knoten sind „ **neu** “.

Rufe **dfs**$(v_0)$ auf.

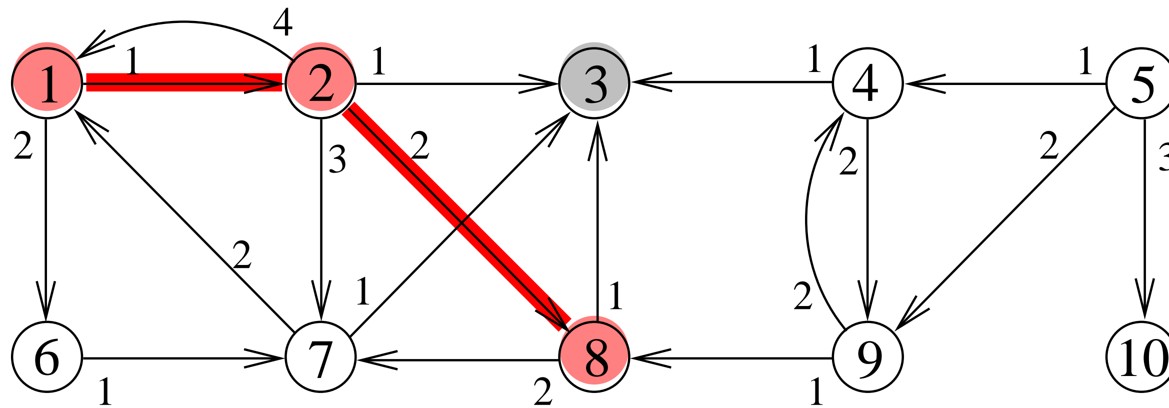

1
2
3
4
5
6
7
8
9
10
4
1
1
1
1
2
3
2
2
2
2
3
2
1
1
2
1
2
1

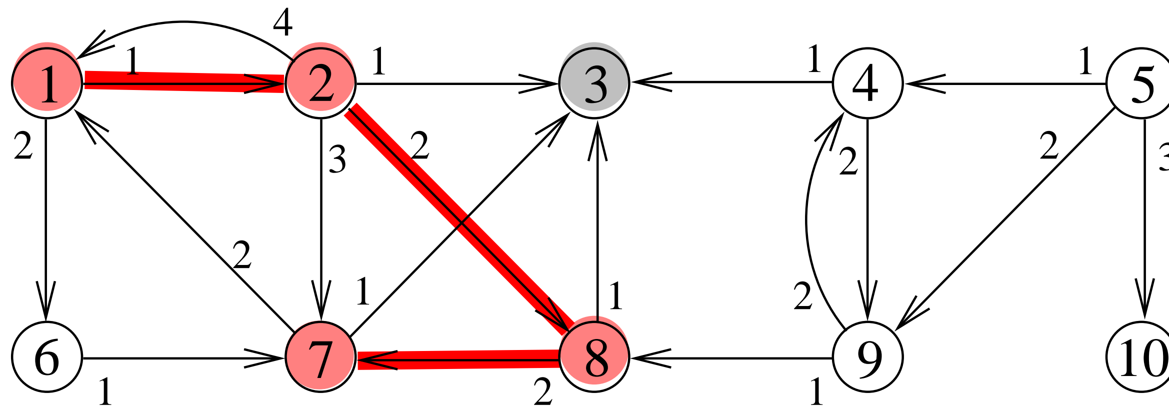

1
2
3
4
5
6
7
8
9
10
4
1
1
1
1
2
3
2
2
1
1
2
2
2
3
2
1
2
1


1
2
3
4
5
6
7
8
9
10

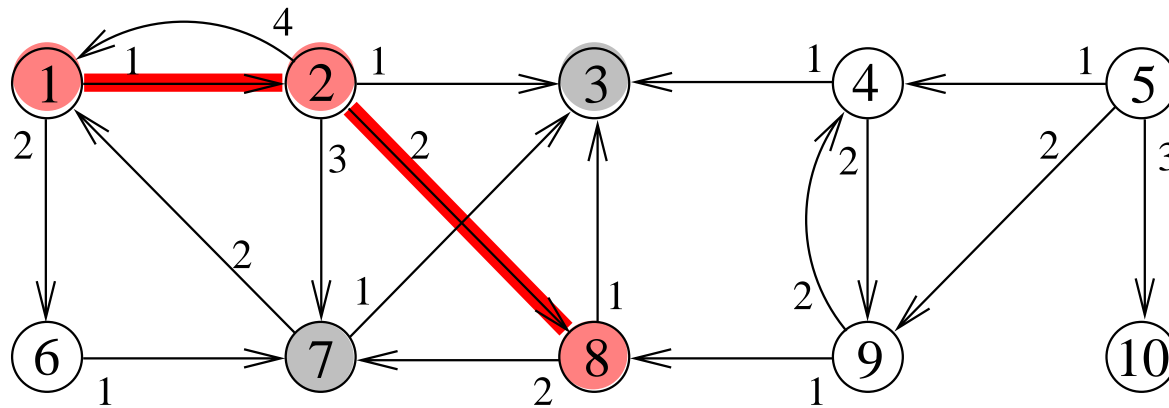


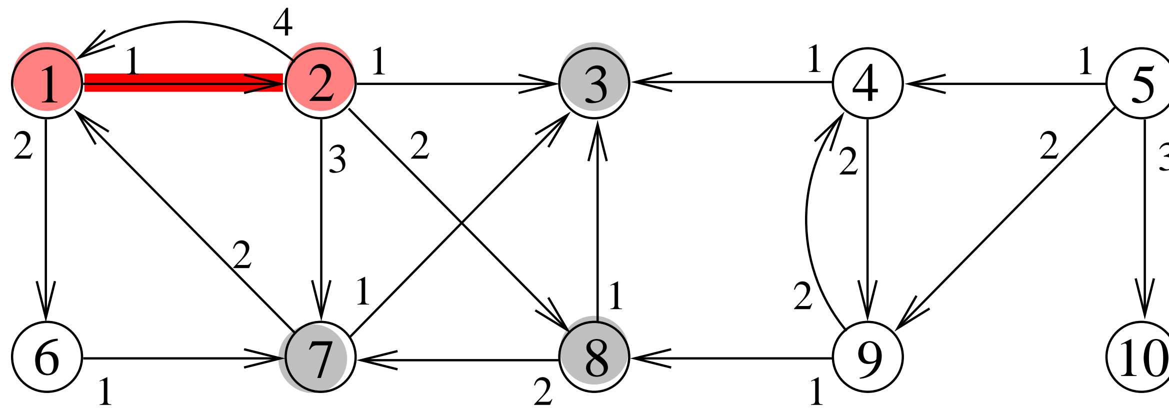
1
2
3
4
5
6
7
8
9
10

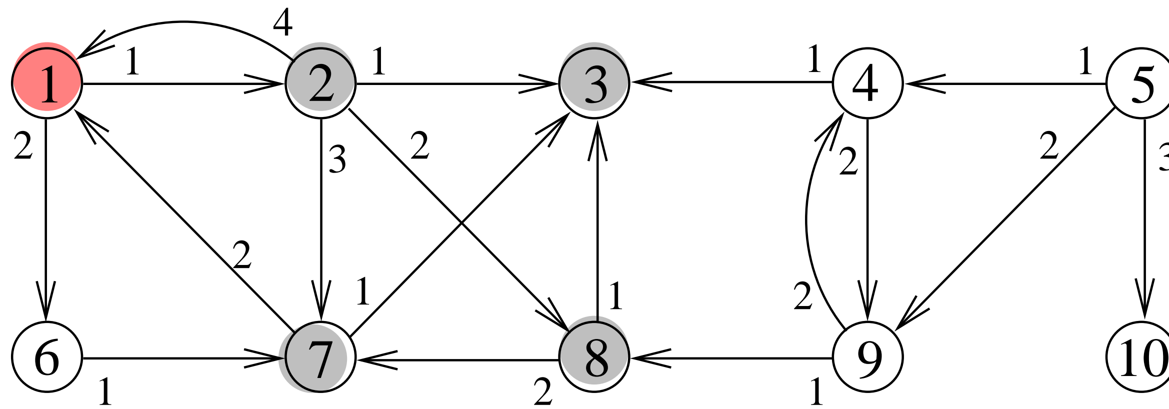

1
2
3
4
5
6
7
8
9
10

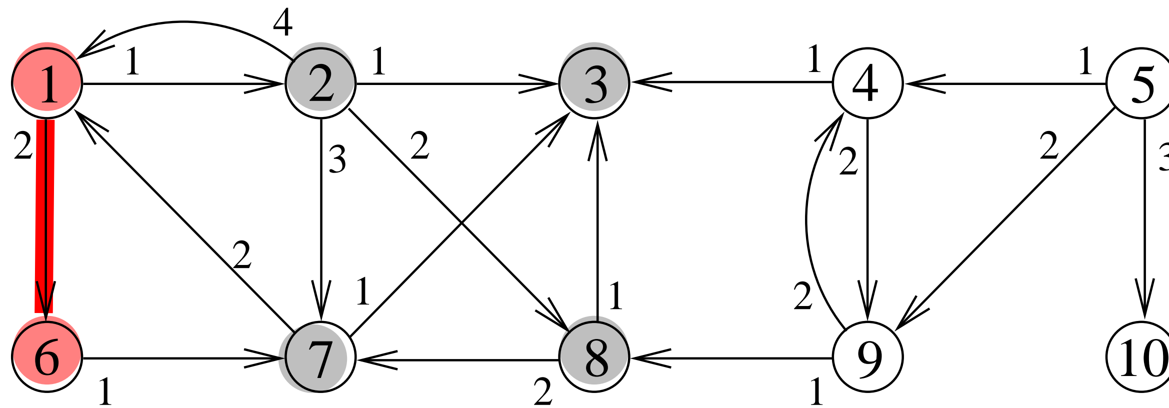


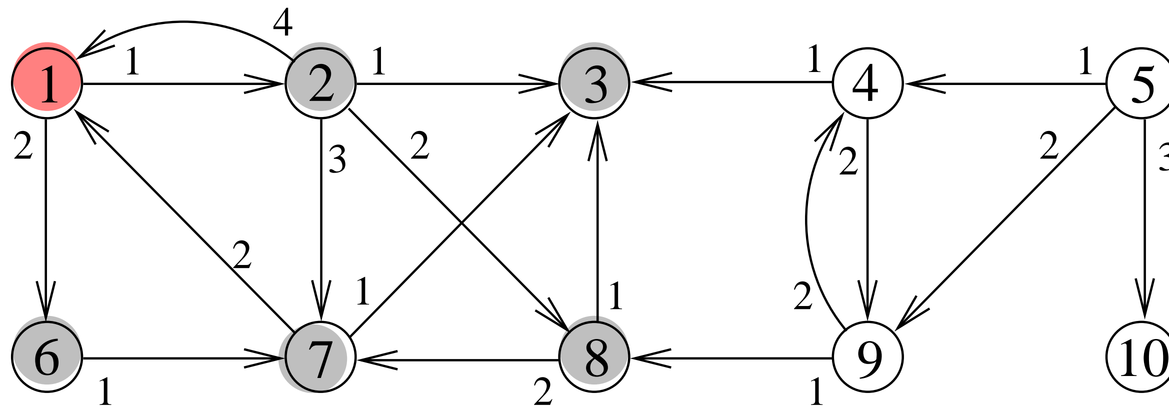
1
2
3
4
5
6
7
8
9
10

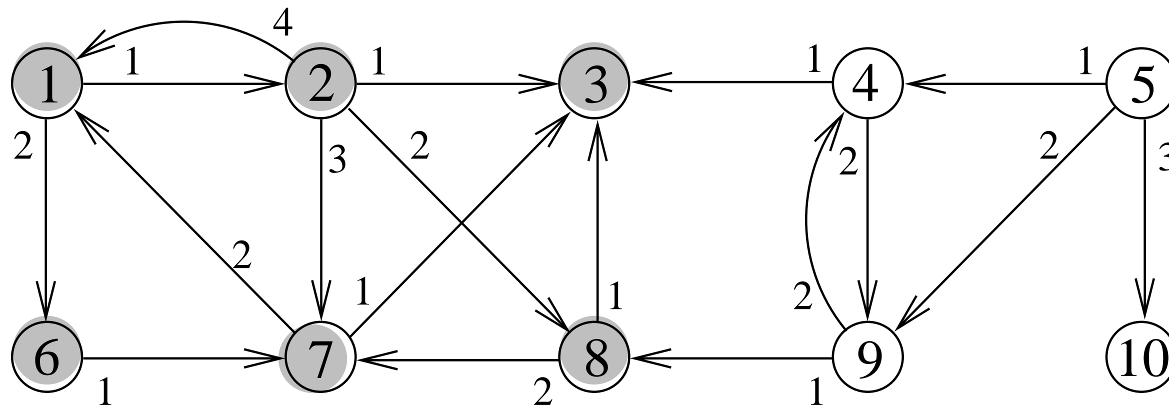

1
2
3
4
5
6
7
8
9
10




1
2
3
4
5
6
7
8
9
10
4
1
1
1
1
2
3
2
2
2
2
3
2
1
1
2
1
2
1


1
2
3
4
5
6
7
8
9
10


1
2
3
4
5
6
7
8
9
10
4
1
1
1
1
2
3
2
2
2
2
3
2
1
1
2
1
1
2
1


1
2
3
4
5
6
7
8
9
10

**Beobachtung 1** über „**dfs**$(v_0)$“:

Für jeden von $v_0$ erreichbaren Knoten $v$ wird **dfs**$(v)$ genau einmal aufgerufen.

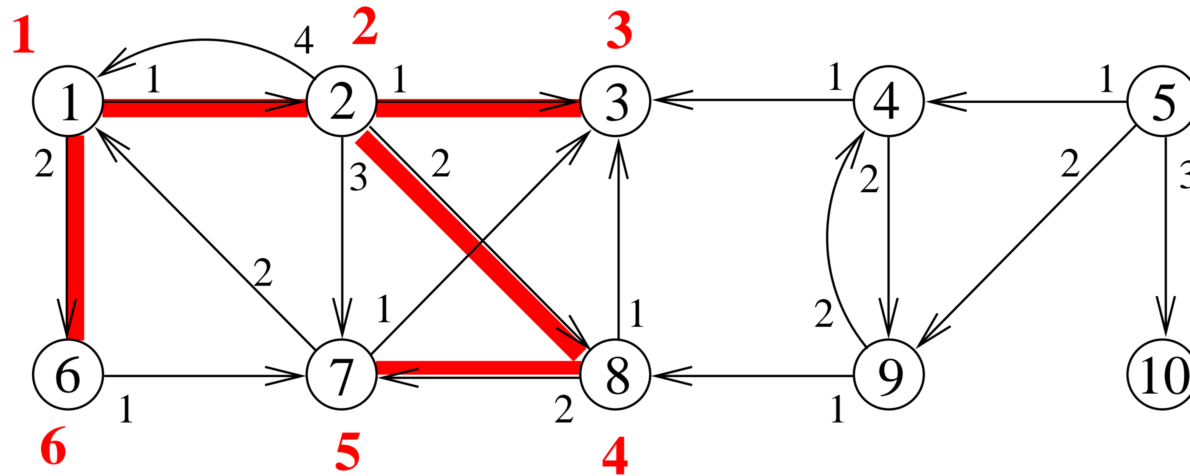
**Beobachtung 2** über „**dfs**$(v_0)$“:

Sei $V_0 = \{v \mid v_0 \rightsquigarrow v\}$. (Von $v_0$ erreichbar.)

Jeder Knoten $v \in V_0 - \{v_0\}$ wird von einem eindeutig bestimmten Knoten $w$ aus entdeckt; diesen nennen wir $p(v)$.

Der Aufruf „**dfs**$(v)$“ erfolgt während der Abarbeitung des Aufrufs „**dfs**$(p(v))$“.

Insbesondere: „**dfs**$(v)$“ beginnt nach und endet vor „**dfs**$(p(v))$“.




**Rot:** Pfeile von $p(v)$ zu $v$: „Entdeckungsrichtung".

Wegen der Eindeutigkeit des Vorgängers: Baumstruktur.

**Beobachtung 3** über „**dfs**$(v_0)$“:

Der Zeitaufwand für die Bearbeitung eines Aufrufs **dfs**$(v)$ ist – ohne die rekursiven Aufrufe – $O(1) + O(\text{outdeg}(v))$.

Der **Zeitaufwand** für „**dfs**$(v_0)$“ ist **linear**,

**Beobachtung 3** über „**dfs**$(v_0)$“:

Der Zeitaufwand für die Bearbeitung eines Aufrufs **dfs**$(v)$ ist – ohne die rekursiven Aufrufe – $O(1) + O(\mathrm{outdeg}(v))$.

Der **Zeitaufwand** für „**dfs**$(v_0)$“ ist **linear**, nämlich

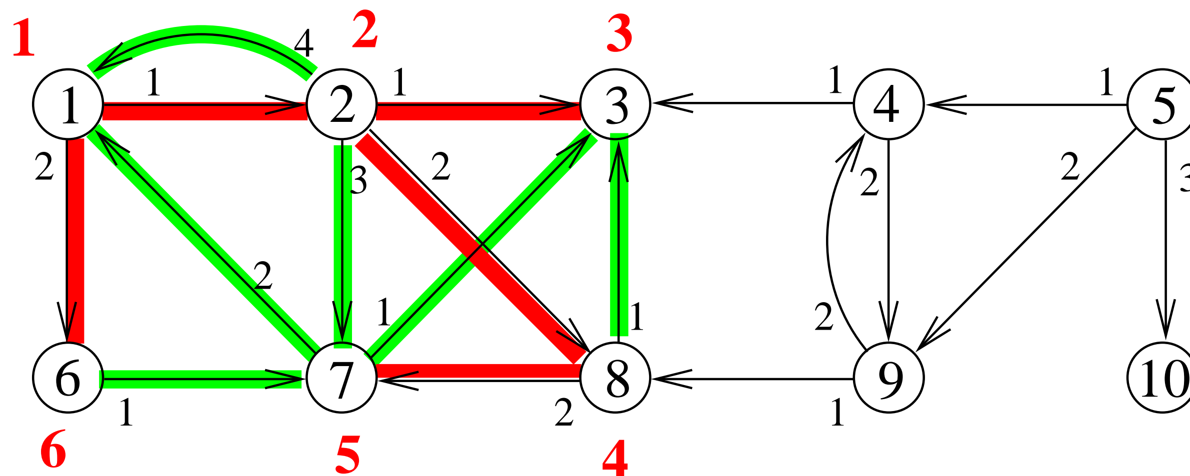
$$O\Big(\sum_{v \in V_0} (1 + \mathrm{outdeg}(v))\Big) = O(|E_0|),$$

wo $V_0$ die Menge der von $v_0$ aus erreichbaren Knoten,
$E_0$ die Menge der von $v_0$ aus erreichbaren Kanten ist.
(Beachte: $|V_0| \leq |E_0| + 1$.)

## Strukturinformation 1:

Mit Tiefensuche in $G$ lässt sich in Zeit $O(|V| + |E_0|)$ die Menge $V_0$ der von $v_0$ aus erreichbaren Knoten ermitteln.

($O(|V|)$ wegen der Initialisierungskosten.)

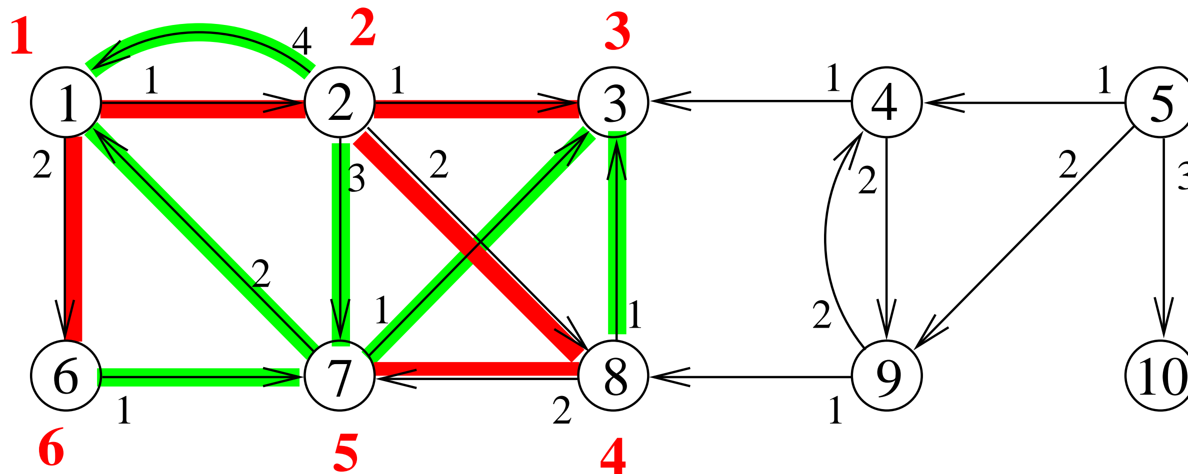


## Strukturinformation 1:

Mit Tiefensuche in $G$ lässt sich in Zeit $O(|V| + |E_0|)$ die Menge $V_0$ der von $v_0$ aus erreichbaren Knoten ermitteln.

($O(|V|)$ wegen der Initialisierungskosten.)

Für $v \in V$ definieren wir: $R_v = \{w \in V \mid v \leadsto w\}$.



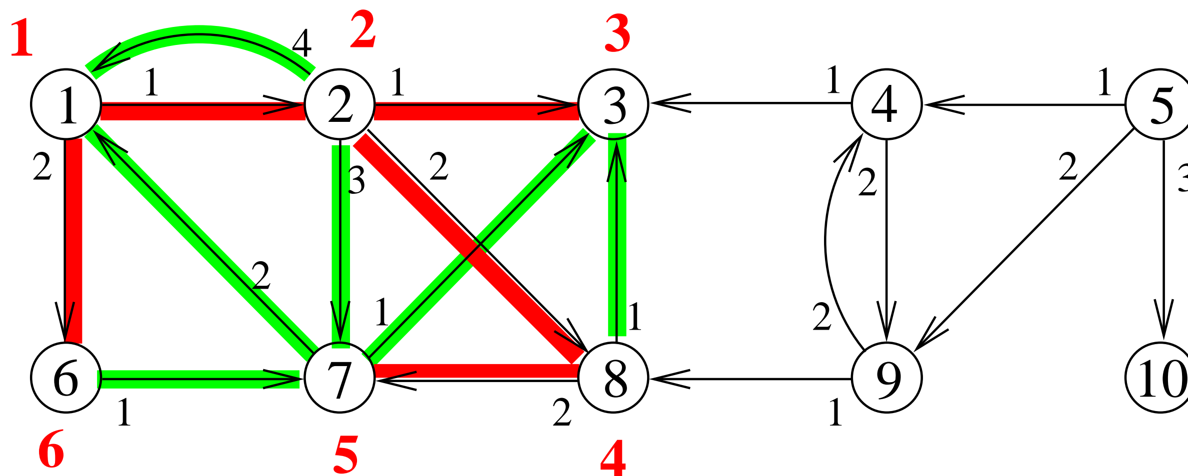
## Strukturinformation 1:

Mit Tiefensuche in $G$ lässt sich in Zeit $O(|V| + |E_0|)$ die Menge $V_0$ der von $v_0$ aus erreichbaren Knoten ermitteln.

($O(|V|)$ wegen der Initialisierungskosten.)

Für $v \in V$ definieren wir: $R_v = \{w \in V \mid v \rightsquigarrow w\}$.

$E_{R_v}$ sei die Menge der Kanten, die von Knoten in $R_v$ ausgehen.



Mit Tiefensuche in $G$, gestartet von jedem Knoten nacheinander, lassen sich in Zeit $O(\sum_{v \in V} |E_{R_v}|) = O(|V| \cdot |E|)$ alle Mengen $R_v$, $v \in V$, ermitteln.

Mit Tiefensuche in $G$, gestartet von jedem Knoten nacheinander, lassen sich in Zeit $O(\sum_{v \in V} |E_{R_v}|) = O(|V| \cdot |E|)$ alle Mengen $R_v$, $v \in V$, ermitteln.

(Nach Aufruf **dfs**$(v)$ kann man mit Hilfe der Liste für $R_v$ in Zeit $O(|R_v|)$ das `status`-Array wieder auf „*neu*“ setzen.)

Mit Tiefensuche in $G$, gestartet von jedem Knoten nacheinander, lassen sich in Zeit $O(\sum_{v \in V} |E_{R_v}|) = O(|V| \cdot |E|)$ alle Mengen $R_v$, $v \in V$, ermitteln.

(Nach Aufruf **dfs**$(v)$ kann man mit Hilfe der Liste für $R_v$ in Zeit $O(|R_v|)$ das `status`-Array wieder auf „*neu*“ setzen.)

**Definition**

Mit Tiefensuche in $G$, gestartet von jedem Knoten nacheinander, lassen sich in Zeit $O(\sum_{v \in V} |E_{R_v}|) = O(|V| \cdot |E|)$ alle Mengen $R_v$, $v \in V$, ermitteln.

(Nach Aufruf **dfs**$(v)$ kann man mit Hilfe der Liste für $R_v$ in Zeit $O(|R_v|)$ das `status`-Array wieder auf „*neu*" setzen.)

**Definition**

Die **transitive Hülle**

Mit Tiefensuche in $G$, gestartet von jedem Knoten nacheinander, lassen sich in Zeit $O(\sum_{v \in V} |E_{R_v}|) = O(|V| \cdot |E|)$ alle Mengen $R_v$, $v \in V$, ermitteln.

(Nach Aufruf **dfs**$(v)$ kann man mit Hilfe der Liste für $R_v$ in Zeit $O(|R_v|)$ das `status`-Array wieder auf „*neu*“ setzen.)

**Definition**

Die **transitive Hülle** eines Digraphs $G$ ist der Graph **TH**$(G) = (V, E^*)$ mit Kantenmenge

Mit Tiefensuche in $G$, gestartet von jedem Knoten nacheinander, lassen sich in Zeit $O(\sum_{v \in V} |E_{R_v}|) = O(|V| \cdot |E|)$ alle Mengen $R_v$, $v \in V$, ermitteln.

(Nach Aufruf **dfs**$(v)$ kann man mit Hilfe der Liste für $R_v$ in Zeit $O(|R_v|)$ das `status`-Array wieder auf „*neu*“ setzen.)

**Definition**

Die **transitive Hülle** eines Digraphs $G$ ist der Graph **TH**$(G) = (V, E^*)$ mit Kantenmenge

$$E^* = \{(v, w) \mid v, w \in V, v \rightsquigarrow_G w\} = \bigcup_{v \in V} (\{v\} \times R_v).$$

Mit Tiefensuche in $G$, gestartet von jedem Knoten nacheinander, lassen sich in Zeit $O(\sum_{v \in V} |E_{R_v}|) = O(|V| \cdot |E|)$ alle Mengen $R_v$, $v \in V$, ermitteln.

(Nach Aufruf **dfs**$(v)$ kann man mit Hilfe der Liste für $R_v$ in Zeit $O(|R_v|)$ das `status`-Array wieder auf „*neu*" setzen.)

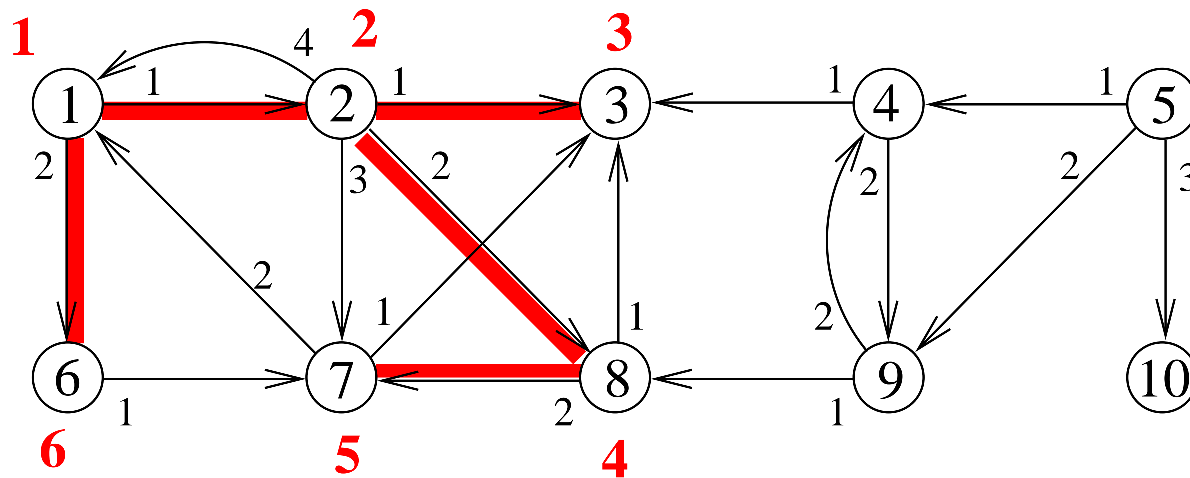
## Definition

Die **transitive Hülle** eines Digraphs $G$ ist der Graph **TH**$(G) = (V, E^*)$ mit Kantenmenge

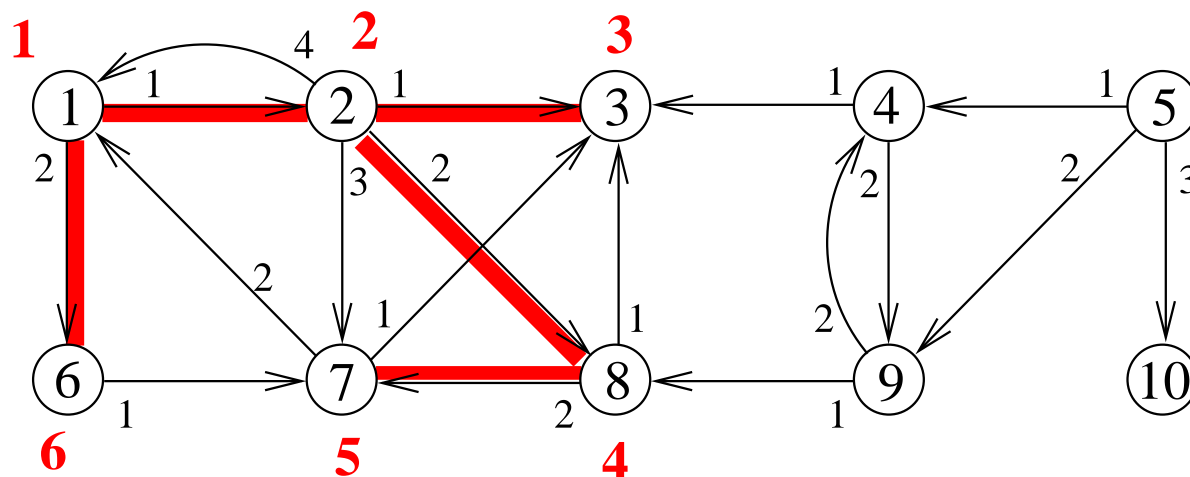
$$E^* = \{(v, w) \mid v, w \in V, v \rightsquigarrow_G w\} = \bigcup_{v \in V} (\{v\} \times R_v).$$

Obige Beobachtung besagt, dass man die Kantenmenge $E^*$ in Zeit $O(|V| \cdot |E|)$ berechnen kann.

# Strukturinformation 2:

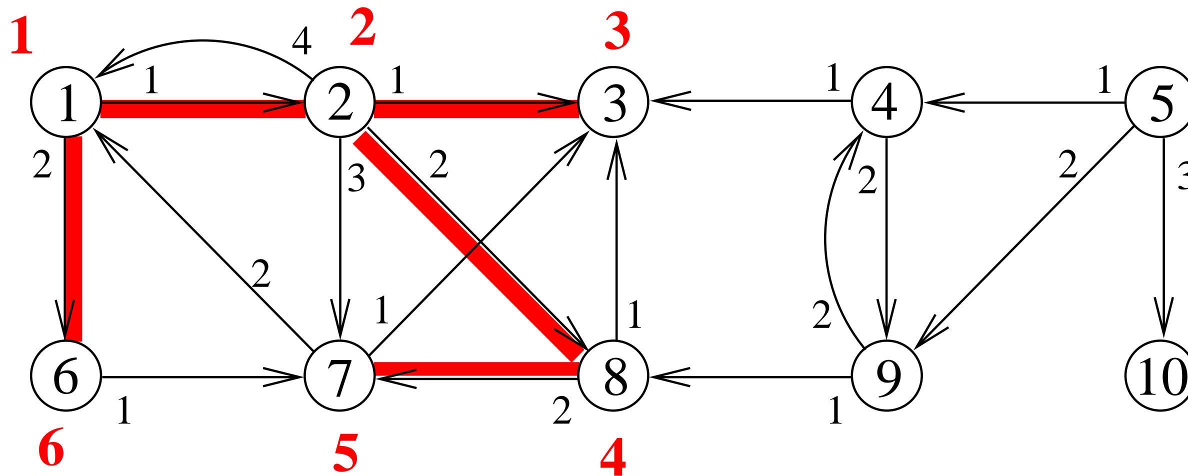


## Strukturinformation 2:




Die Kanten $(p(v), v)$ bilden einen **(gerichteten) Baum** mit Wurzel $v_0$ und Knotenmenge $V_0$.

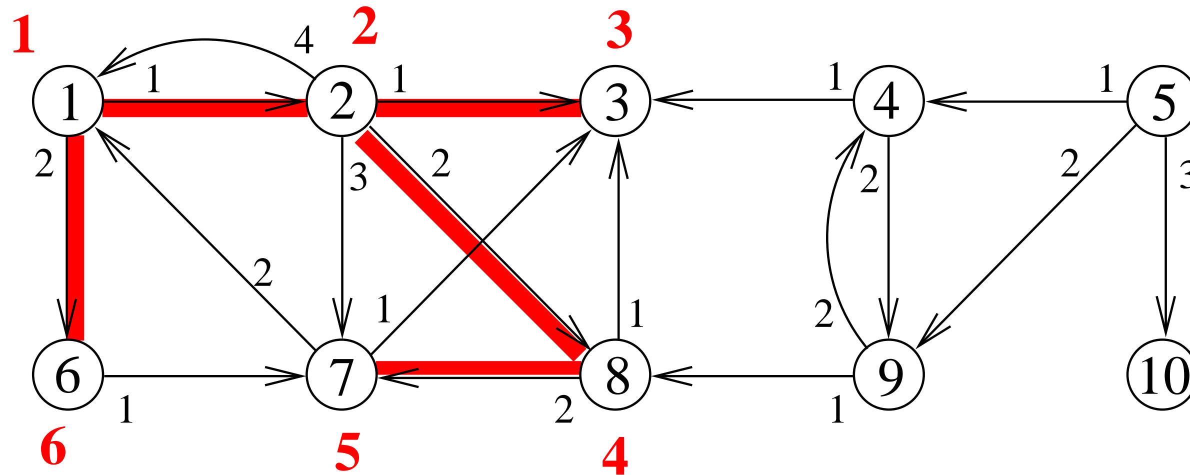
## Strukturinformation 2:




Die Kanten $(p(v), v)$ bilden einen **(gerichteten) Baum** mit Wurzel $v_0$ und Knotenmenge $V_0$.

**Tiefensuch-Baum** $\boldsymbol{T_{\mathrm{dfs}}(v_0)}$

## Strukturinformation 2:

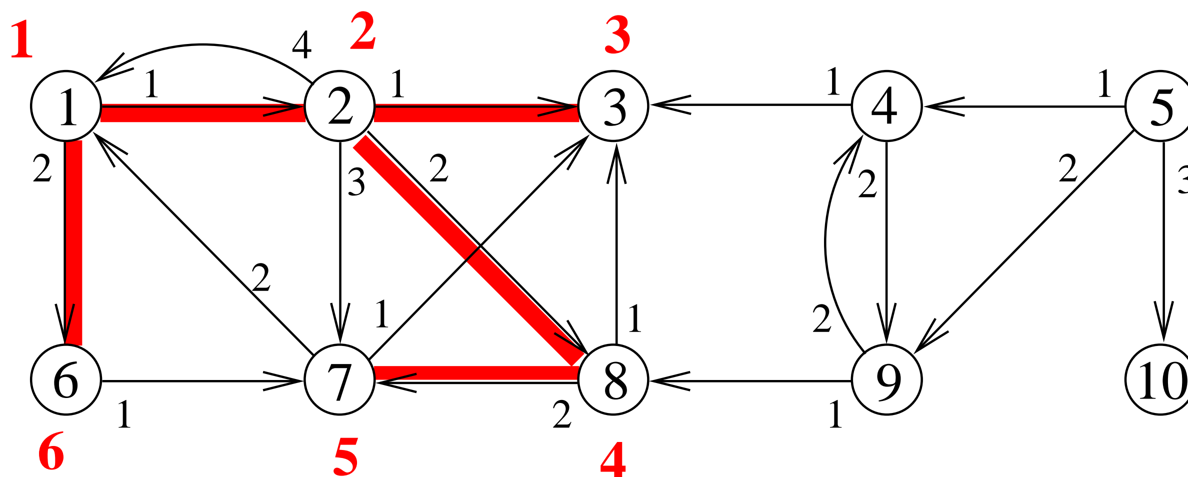



Die Kanten $(p(v), v)$ bilden einen **(gerichteten) Baum** mit Wurzel $v_0$ und Knotenmenge $V_0$.

**Tiefensuch-Baum** $\boldsymbol{T_{\mathrm{dfs}}(v_0)}$

Kante $(w, v)$ in $T_{\mathrm{dfs}}(v_0)$ $\Leftrightarrow$ **dfs**$(v)$ direkt aus **dfs**$(w)$ aufgerufen.

## Strukturinformation 2:

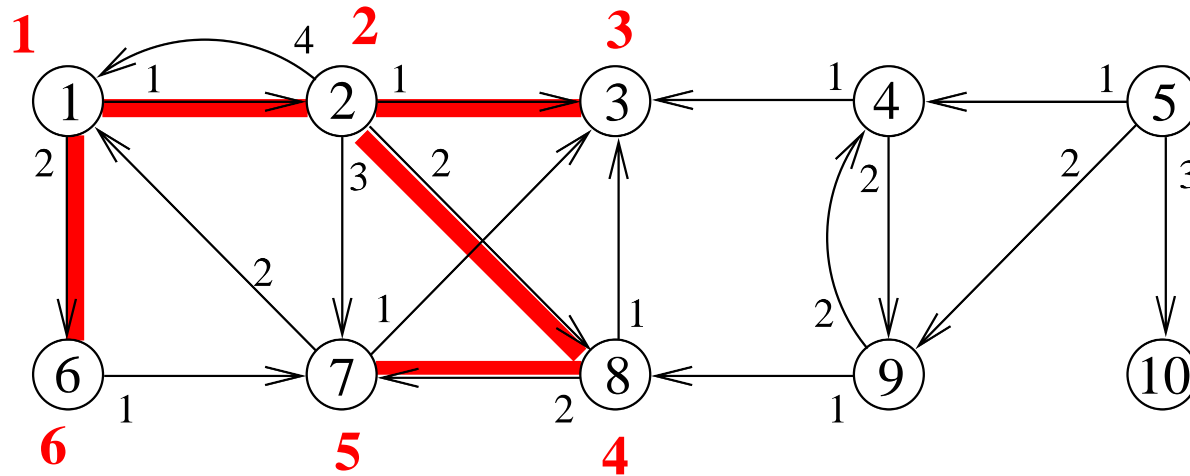



Die Kanten $(p(v), v)$ bilden einen **(gerichteten) Baum** mit Wurzel $v_0$ und Knotenmenge $V_0$.

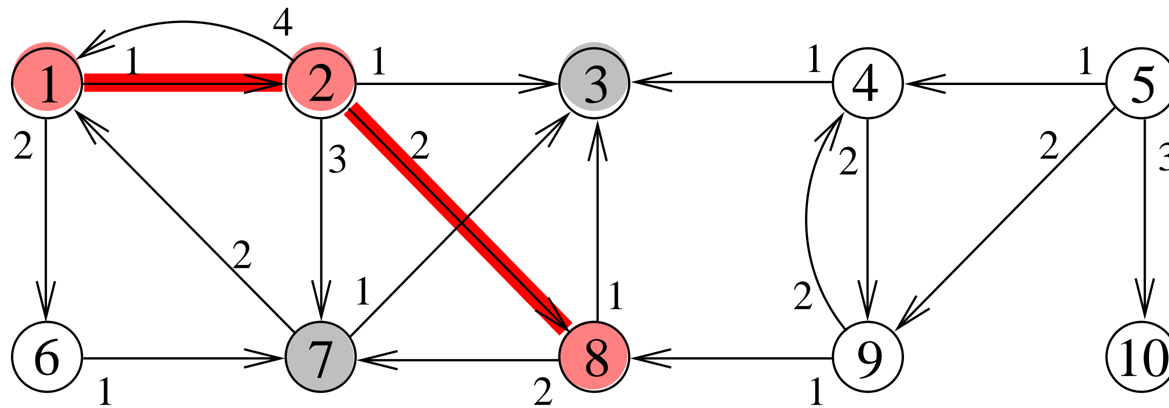
**Tiefensuch-Baum** $\boldsymbol{T_{\mathrm{dfs}}(v_0)}$

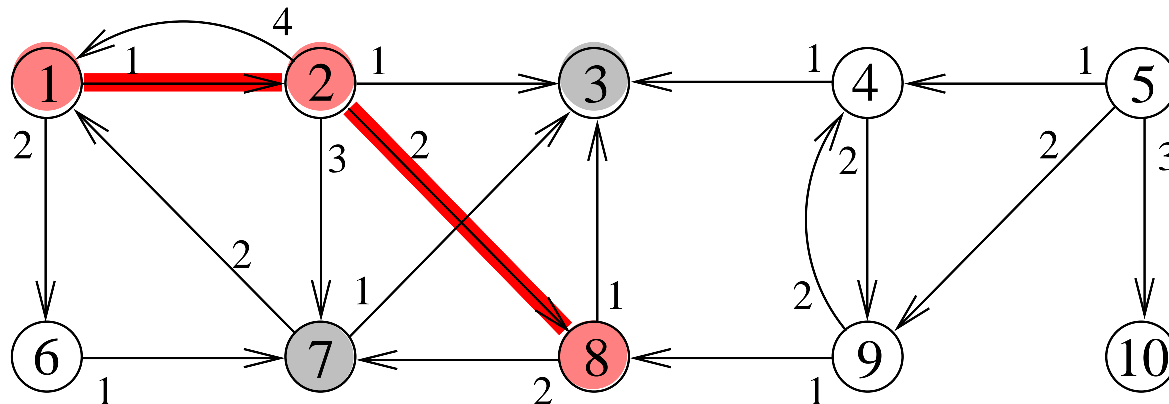
Kante $(w, v)$ in $T_{\mathrm{dfs}}(v_0)$ $\Leftrightarrow$ **dfs**$(v)$ direkt aus **dfs**$(w)$ aufgerufen.

Weg im Tiefensuch-Baum entspricht indirekter Aufrufbeziehung.




Die Tiefensuch-Nummern nummerieren die Knoten im Baum in Präorder-Art: erst die Wurzel, dann nacheinander rekursiv die Unterbäume.


1
2
3
4
5
6
7
8
9
10
4
1
1
1
1
2
3
2
2
1
1
2
2
2
3
2
1
2
1




Zu jedem Zeitpunkt bilden die **aktiven** Knoten einen Weg im Tiefensuch-Baum, Startpunkt $v_0$, Endpunkt der Knoten, in dessen „**dfs**$(v)$“-Aufruf eben gearbeitet wird.

**???** Wann sitzt $w$ in $T_{\rm dfs}(v_0)$ im Unterbaum mit Wurzel $v$ ?

**???** Wann sitzt $w$ in $T_{\mathrm{dfs}}(v_0)$ im Unterbaum mit Wurzel $v$ ?

**???** Wann wird **dfs**$(w)$ aufgerufen, während $v$ *aktiv* ist?

**???** Wann sitzt $w$ in $T_{\mathrm{dfs}}(v_0)$ im Unterbaum mit Wurzel $v$ ?

**???** Wann wird **dfs**$(w)$ aufgerufen, während $v$ *aktiv* ist?

**Satz** („Satz vom weißen Weg")

**???** Wann sitzt $w$ in $T_{\text{dfs}}(v_0)$ im Unterbaum mit Wurzel $v$ ?

**???** Wann wird **dfs**$(w)$ aufgerufen, während $v$ *aktiv* ist?

**Satz** („Satz vom weißen Weg“)

$w$ ist ein (direkter oder indirekter) Nachfolger von $v$
im Tiefensuchbaum $T_{\text{dfs}}(v_0)$,

$\Leftrightarrow$

**???** Wann sitzt $w$ in $T_{\mathrm{dfs}}(v_0)$ im Unterbaum mit Wurzel $v$ ?

**???** Wann wird **dfs**$(w)$ aufgerufen, während $v$ *aktiv* ist?

**Satz** („Satz vom weißen Weg“)

$w$ ist ein (direkter oder indirekter) Nachfolger von $v$
im Tiefensuchbaum $T_{\mathrm{dfs}}(v_0)$,

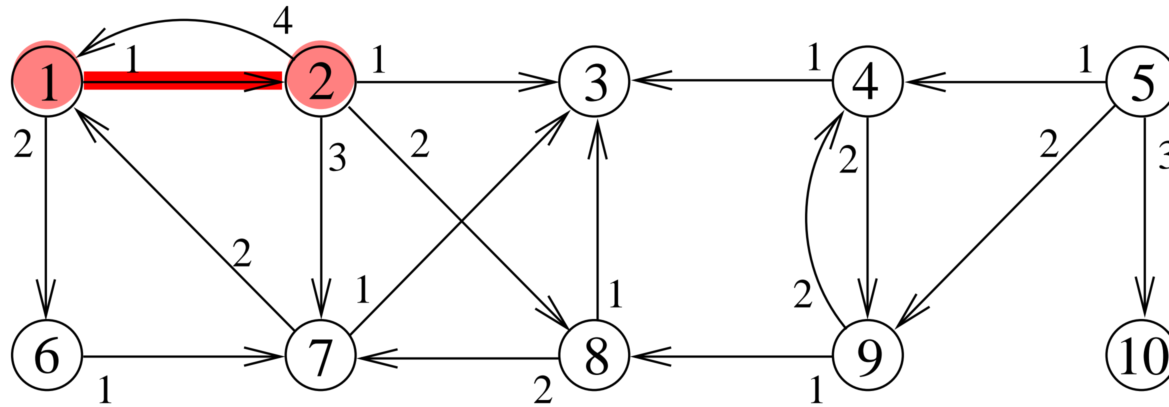
$\Leftrightarrow$

in dem Moment, in dem **dfs**$(v)$ aufgerufen wird, existiert ein Weg

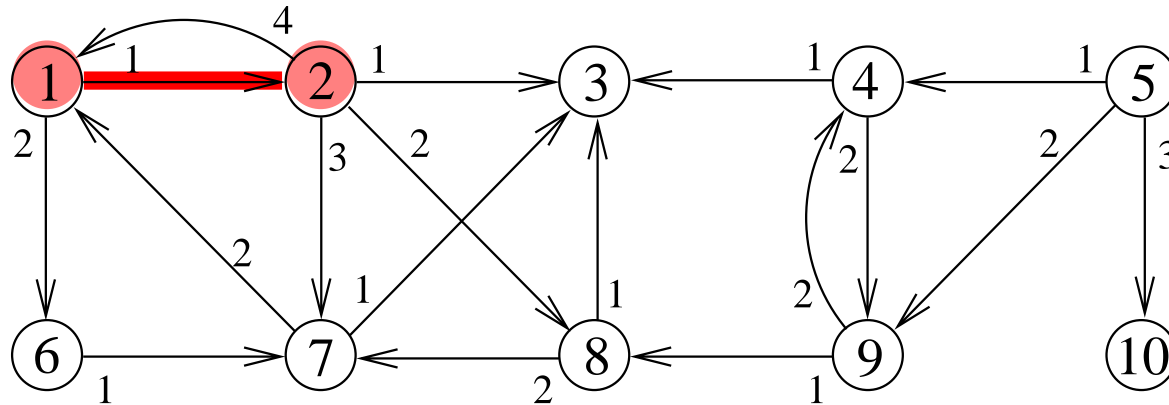
$v = u_0, u_1, u_2, \ldots, u_t = w$

aus lauter neuen Knoten
(ein Weg von $v$ nach $w$ aus „weißen Knoten“).

Beispiel: **dfs**$(2)$ startet.

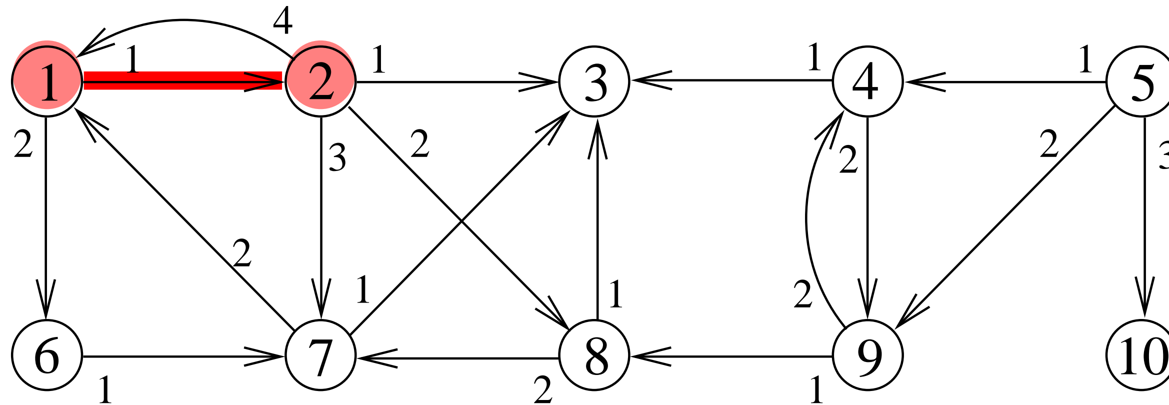


Beispiel: **dfs**$(2)$ startet.




Knoten $6$ ist von $2$ aus erreichbar, aber nicht auf einem weißen Weg

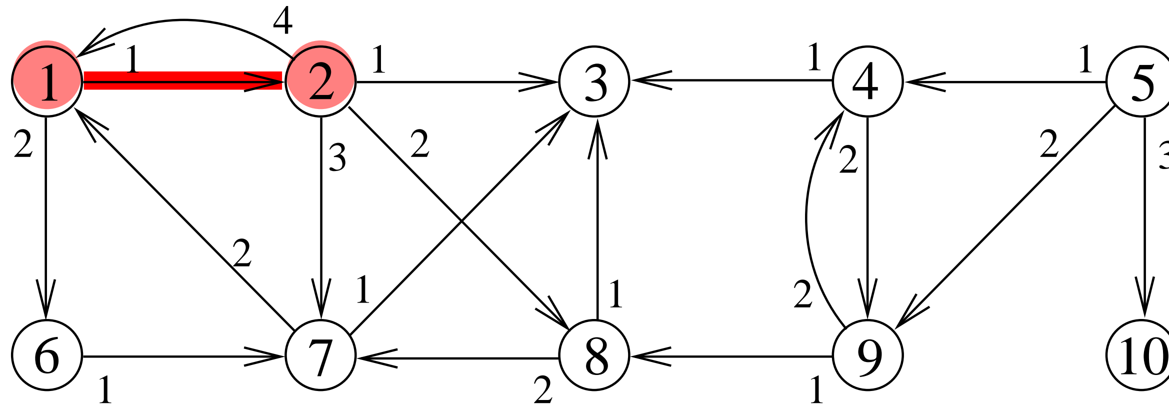
Beispiel: **dfs**$(2)$ startet.




Knoten $6$ ist von $2$ aus erreichbar, aber nicht auf einem weißen Weg

$\Rightarrow$ Knoten $6$ nicht in Tiefensuchbaum unter $2$.

Beispiel: **dfs**$(2)$ startet.

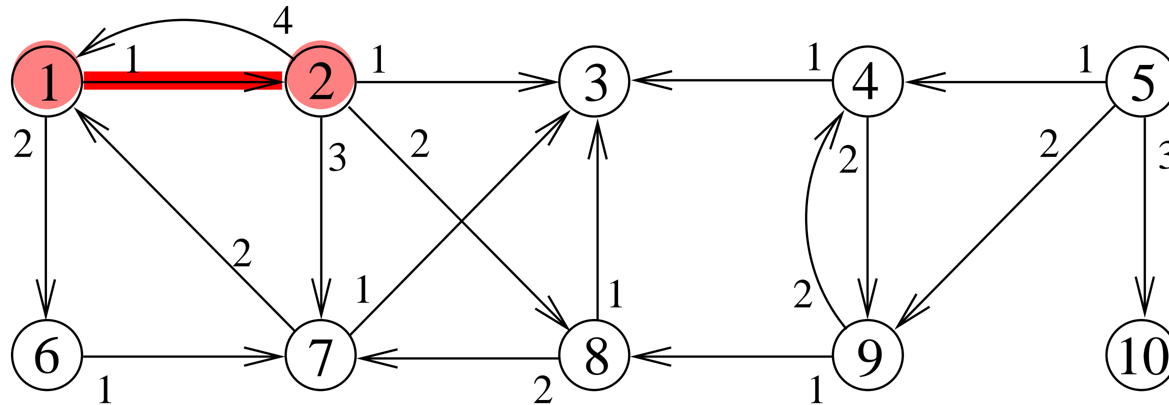



Knoten $6$ ist von $2$ aus erreichbar, aber nicht auf einem weißen Weg

$\Rightarrow$ Knoten $6$ nicht in Tiefensuchbaum unter $2$.

Knoten $3$, $7$, $8$ sind auf weißen Wegen erreichbar

Beispiel: **dfs**$(2)$ startet.

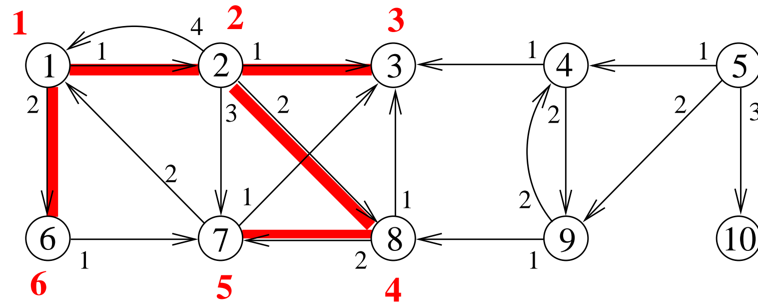



Knoten $6$ ist von $2$ aus erreichbar, aber nicht auf einem weißen Weg

$\Rightarrow$ Knoten $6$ nicht in Tiefensuchbaum unter $2$.

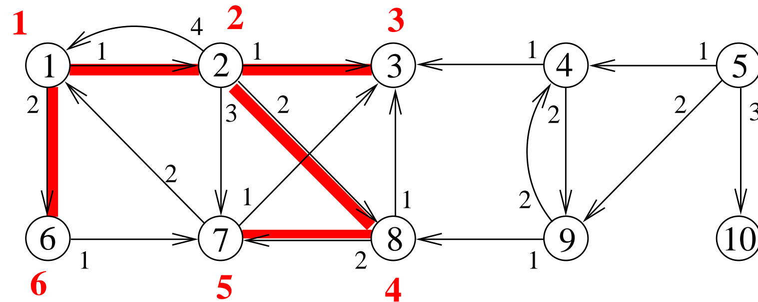
Knoten $3$, $7$, $8$ sind auf weißen Wegen erreichbar

$\Rightarrow$ diese Knoten sitzen im Tiefensuchbaum unter $2$.




*Beweis*: „⇒“:

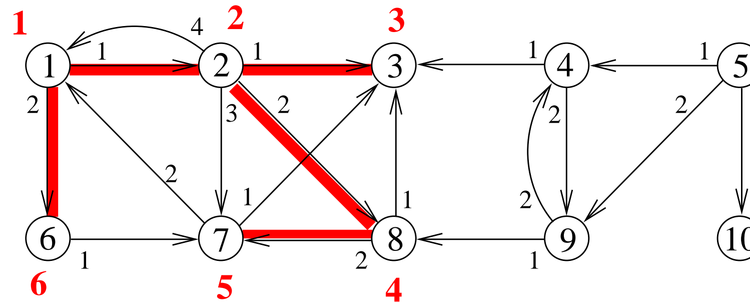
Sei $w$ in $T_{\mathrm{dfs}}(v_0)$ Nachfolger von $v$.




*Beweis*: „$\Rightarrow$“:

Sei $w$ in $T_{\rm dfs}(v_0)$ Nachfolger von $v$.

Wir nehmen den Weg, der in $T_{\rm dfs}(v_0)$ von $v$ nach $w$ führt:



*Beweis*: „⇒“:

Sei $w$ in $T_{\mathrm{dfs}}(v_0)$ Nachfolger von $v$.

Wir nehmen den Weg, der in $T_{\mathrm{dfs}}(v_0)$ von $v$ nach $w$ führt:

$v = u_0, u_1, u_2, \ldots, u_t = w$. (Beispiel: $(2, 8, 7)$.)

Dann wird **dfs**$(u_1)$ nach **dfs**$(\underbrace{u_0}_{v})$ aufgerufen, **dfs**$(u_2)$ nach **dfs**$(u_1)$, usw.

also sind im Moment des Aufrufs **dfs**$(v)$ alle Knoten $v = u_0, u_1, u_2, \ldots, u_t = w$ noch „neu“.

*Beweis* des Satzes: „$\Leftarrow$“:

*Beweis* des Satzes: „$\Leftarrow$“:

Sei zum Zeitpunkt des Aufrufs **dfs**$(v)$

$v = u_0, u_1, u_2, \ldots, u_t = w$, $t \geq 1$,

ein einfacher Weg aus neuen („weißen“) Knoten.

*Beweis* des Satzes: „$\Leftarrow$“:

Sei zum Zeitpunkt des Aufrufs **dfs**$(v)$

$v = u_0, u_1, u_2, \ldots, u_t = w$, $t \geq 1$,

ein einfacher Weg aus neuen („weißen“) Knoten.
(Beispiel: $(2, 7)$.)

*Beweis* des Satzes: „$\Leftarrow$“:

Sei zum Zeitpunkt des Aufrufs **dfs**$(v)$

$v = u_0, u_1, u_2, \ldots, u_t = w$, $t \geq 1$,

ein einfacher Weg aus neuen („weißen“) Knoten.
(Beispiel: $(2, 7)$.)

**Achtung:** Es ist **nicht** gesagt, dass genau dieser Weg im Baum $T_{\text{dfs}}(v_0)$ auftaucht.

*Beweis* des Satzes: „$\Leftarrow$“:

Sei zum Zeitpunkt des Aufrufs **dfs**$(v)$

$v = u_0, u_1, u_2, \ldots, u_t = w$, $t \geq 1$,

ein einfacher Weg aus neuen („weißen“) Knoten.
(Beispiel: $(2,7)$.)

**Achtung:** Es ist **nicht** gesagt, dass genau dieser Weg im Baum $T_{\mathrm{dfs}}(v_0)$ auftaucht.

O.B.d.A.: $i > 0$. (Sonst Aussage trivial.)

*Beweis* des Satzes: „$\Leftarrow$“:

Sei zum Zeitpunkt des Aufrufs **dfs**$(v)$

$v = u_0, u_1, u_2, \ldots, u_t = w$, $t \geq 1$,

ein einfacher Weg aus neuen („weißen“) Knoten.
(Beispiel: $(2, 7)$.)

**Achtung:** Es ist **nicht** gesagt, dass genau dieser Weg im Baum $T_{\text{dfs}}(v_0)$ auftaucht.

O.B.d.A.: $i > 0$. (Sonst Aussage trivial.)

Wir zeigen durch Induktion über $i = 0, 1, \ldots, t$:

*Beweis* des Satzes: „$\Leftarrow$“:

Sei zum Zeitpunkt des Aufrufs **dfs**$(v)$

$v = u_0, u_1, u_2, \ldots, u_t = w$, $t \geq 1$,

ein einfacher Weg aus neuen („weißen“) Knoten.
(Beispiel: $(2, 7)$.)

**Achtung:** Es ist **nicht** gesagt, dass genau dieser Weg im Baum $T_{\text{dfs}}(v_0)$ auftaucht.

O.B.d.A.: $i > 0$. (Sonst Aussage trivial.)

Wir zeigen durch Induktion über $i = 0, 1, \ldots, t$:

**Behauptung:** Der Aufruf **dfs**$(u_i)$ beginnt, bevor **dfs**$(v)$ endet.

**I.Beh.:** Der Aufruf **dfs**$(u_i)$ beginnt, bevor **dfs**$(v)$ endet.

**I.Beh.:** Der Aufruf **dfs**$(u_i)$ beginnt, bevor **dfs**$(v)$ endet.

**I.A.:** $i = 0$: $u_0 = v$; **dfs**$(v)$ beginnt, bevor **dfs**$(v)$ endet.

**I.Beh.:** Der Aufruf **dfs**$(u_i)$ beginnt, bevor **dfs**$(v)$ endet.

**I.A.:** $i = 0$: $u_0 = v$; **dfs**$(v)$ beginnt, bevor **dfs**$(v)$ endet.

**I.V.:** $i \geq 1$, und **dfs**$(u_{i-1})$ beginnt, bevor **dfs**$(v)$ endet.

**I.Beh.:** Der Aufruf **dfs**$(u_i)$ beginnt, bevor **dfs**$(v)$ endet.

**I.A.:** $i = 0$: $u_0 = v$; **dfs**$(v)$ beginnt, bevor **dfs**$(v)$ endet.

**I.V.:** $i \geq 1$, und **dfs**$(u_{i-1})$ beginnt, bevor **dfs**$(v)$ endet.

**I.S.:** Aus I.V. folgt:
**dfs**$(u_{i-1})$ endet auch spätestens wenn **dfs**$(v)$ endet. $(*)$
(Grundprinzip bei geschachtelten Prozeduraufrufen.)

**I.Beh.:** Der Aufruf **dfs**$(u_i)$ beginnt, bevor **dfs**$(v)$ endet.

**I.A.:** $i = 0$: $u_0 = v$; **dfs**$(v)$ beginnt, bevor **dfs**$(v)$ endet.

**I.V.:** $i \geq 1$, und **dfs**$(u_{i-1})$ beginnt, bevor **dfs**$(v)$ endet.

**I.S.:** Aus I.V. folgt:
**dfs**$(u_{i-1})$ endet auch spätestens wenn **dfs**$(v)$ endet. $(*)$
(Grundprinzip bei geschachtelten Prozeduraufrufen.)

Im Aufruf **dfs**$(u_{i-1})$

**I.Beh.:** Der Aufruf **dfs**$(u_i)$ beginnt, bevor **dfs**$(v)$ endet.

**I.A.:** $i = 0$: $u_0 = v$; **dfs**$(v)$ beginnt, bevor **dfs**$(v)$ endet.

**I.V.:** $i \geq 1$, und **dfs**$(u_{i-1})$ beginnt, bevor **dfs**$(v)$ endet.

**I.S.:** Aus I.V. folgt:
**dfs**$(u_{i-1})$ endet auch spätestens wenn **dfs**$(v)$ endet. $(*)$
(Grundprinzip bei geschachtelten Prozeduraufrufen.)

Im Aufruf **dfs**$(u_{i-1})$ wird die Kante $(u_{i-1}, u_i)$ angesehen.

**I.Beh.:** Der Aufruf **dfs**$(u_i)$ beginnt, bevor **dfs**$(v)$ endet.

**I.A.:** $i = 0$: $u_0 = v$; **dfs**$(v)$ beginnt, bevor **dfs**$(v)$ endet.

**I.V.:** $i \geq 1$, und **dfs**$(u_{i-1})$ beginnt, bevor **dfs**$(v)$ endet.

**I.S.:** Aus I.V. folgt:
**dfs**$(u_{i-1})$ endet auch spätestens wenn **dfs**$(v)$ endet. $(*)$
(Grundprinzip bei geschachtelten Prozeduraufrufen.)

Im Aufruf **dfs**$(u_{i-1})$ wird die Kante $(u_{i-1}, u_i)$ angesehen.

**1. Fall:** $u_i$ ist *aktiv* oder *fertig*.
Wegen $(*)$: **dfs**$(u_i)$ hat begonnen, bevor **dfs**$(v)$ endet.

**I.Beh.:** Der Aufruf **dfs**$(u_i)$ beginnt, bevor **dfs**$(v)$ endet.

**I.A.:** $i = 0$: $u_0 = v$; **dfs**$(v)$ beginnt, bevor **dfs**$(v)$ endet.

**I.V.:** $i \geq 1$, und **dfs**$(u_{i-1})$ beginnt, bevor **dfs**$(v)$ endet.

**I.S.:** Aus I.V. folgt:
**dfs**$(u_{i-1})$ endet auch spätestens wenn **dfs**$(v)$ endet. $(*)$
(Grundprinzip bei geschachtelten Prozeduraufrufen.)

Im Aufruf **dfs**$(u_{i-1})$ wird die Kante $(u_{i-1}, u_i)$ angesehen.

**1. Fall:** $u_i$ ist *aktiv* oder *fertig*.
Wegen $(*)$: **dfs**$(u_i)$ hat begonnen, bevor **dfs**$(v)$ endet.

**2. Fall:** $u_i$ ist *neu*.
Dann wird **dfs**$(u_i)$ direkt aus **dfs**$(u_{i-1})$ aufgerufen.

**I.Beh.:** Der Aufruf **dfs**$(u_i)$ beginnt, bevor **dfs**$(v)$ endet.

**I.A.:** $i = 0$: $u_0 = v$; **dfs**$(v)$ beginnt, bevor **dfs**$(v)$ endet.

**I.V.:** $i \geq 1$, und **dfs**$(u_{i-1})$ beginnt, bevor **dfs**$(v)$ endet.

**I.S.:** Aus I.V. folgt:
**dfs**$(u_{i-1})$ endet auch spätestens wenn **dfs**$(v)$ endet. $(*)$
(Grundprinzip bei geschachtelten Prozeduraufrufen.)

Im Aufruf **dfs**$(u_{i-1})$ wird die Kante $(u_{i-1}, u_i)$ angesehen.

**1. Fall:** $u_i$ ist *aktiv* oder *fertig*.
Wegen $(*)$: **dfs**$(u_i)$ hat begonnen, bevor **dfs**$(v)$ endet.

**2. Fall:** $u_i$ ist *neu*.
Dann wird **dfs**$(u_i)$ direkt aus **dfs**$(u_{i-1})$ aufgerufen.
Wegen $(*)$: **dfs**$(u_i)$ beginnt, bevor **dfs**$(v)$ endet.

Wenn wir die **Behauptung** für $i = t$ anwenden, folgt:

Der Aufruf **dfs**$(w)$ beginnt, bevor **dfs**$(v)$ endet.

Wenn wir die **Behauptung** für $i = t$ anwenden, folgt:

Der Aufruf **dfs**$(w)$ beginnt, bevor **dfs**$(v)$ endet.

Weil $w$ beim Start des Aufrufs **dfs**$(v)$ neu war und $w \neq v$:
**dfs**$(w)$ beginnt nach **dfs**$(v)$ und endet, bevor **dfs**$(v)$ endet.

Wenn wir die **Behauptung** für $i = t$ anwenden, folgt:

Der Aufruf **dfs**$(w)$ beginnt, bevor **dfs**$(v)$ endet.

Weil $w$ beim Start des Aufrufs **dfs**$(v)$ neu war und $w \neq v$:
**dfs**$(w)$ beginnt nach **dfs**$(v)$ und endet, bevor **dfs**$(v)$ endet.

Daher: Die Folge $w, p(w), p(p(w)), p(p(p(w))), \ldots$ erreicht $v$.
Umgedreht gelesen, ist dies ein Weg von $v$ nach $w$ in $T_{\mathrm{dfs}}(v_0)$.

## **Globale Tiefensuche in Digraph $G$:** Alle Knoten besuchen.

**Globale Tiefensuche in Digraph $G$:** Alle Knoten besuchen.

**Algorithmus DFS**$(G)$ $(*$ Tiefensuche in $G = (V, E)$ $*)$

**Globale Tiefensuche in Digraph $G$:** Alle Knoten besuchen.

**Algorithmus DFS**$(G)$ $(*$ Tiefensuche in $G = (V, E)$ $*)$

(1) `dfs_count` $\leftarrow 0$;
(2) **for** `v` **from** $1$ **to** $n$ **do** `status[v]` $\leftarrow$ *neu*;
(3) **for** `v` **from** $1$ **to** $n$ **do**
(4)     **if** `status[v]` $=$ *neu* **then**
(5)         **dfs**(`v`); $(*$ starte Tiefensuche von `v` aus $*)$

**Globale Tiefensuche in Digraph $G$:** Alle Knoten besuchen.

**Algorithmus DFS**$(G)$ $(*$ Tiefensuche in $G = (V, E)$ $*)$

```
(1)   dfs_count ← 0;
(2)   for v from 1 to n do status[v] ← neu;
(3)   for v from 1 to n do
(4)       if status[v] = neu then
(5)           dfs(v); (* starte Tiefensuche von v aus *)
```

Zu Zeilen (1)/(5): `dfs_count` wird weitergezählt.

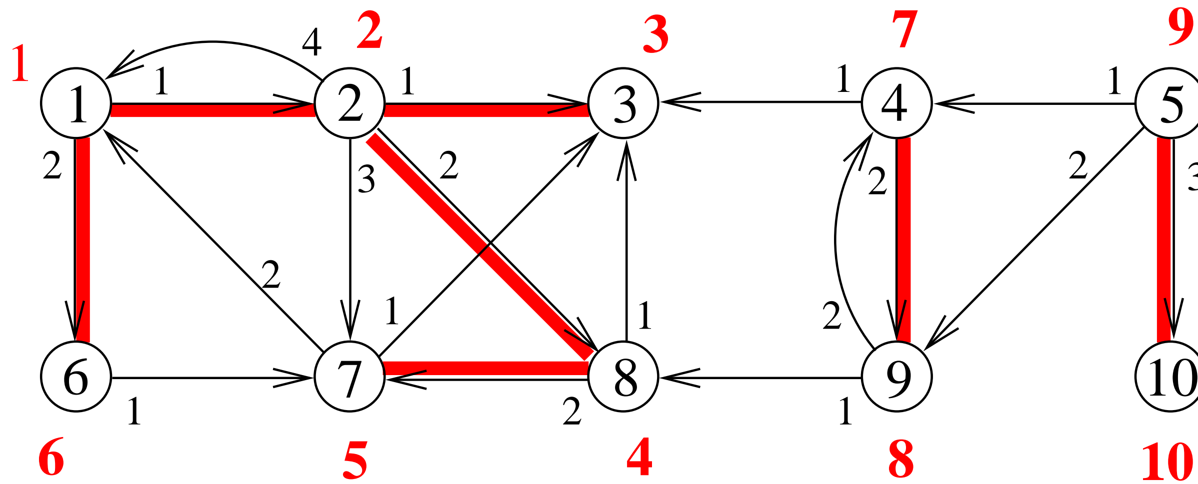
**Globale Tiefensuche in Digraph $G$:** Alle Knoten besuchen.

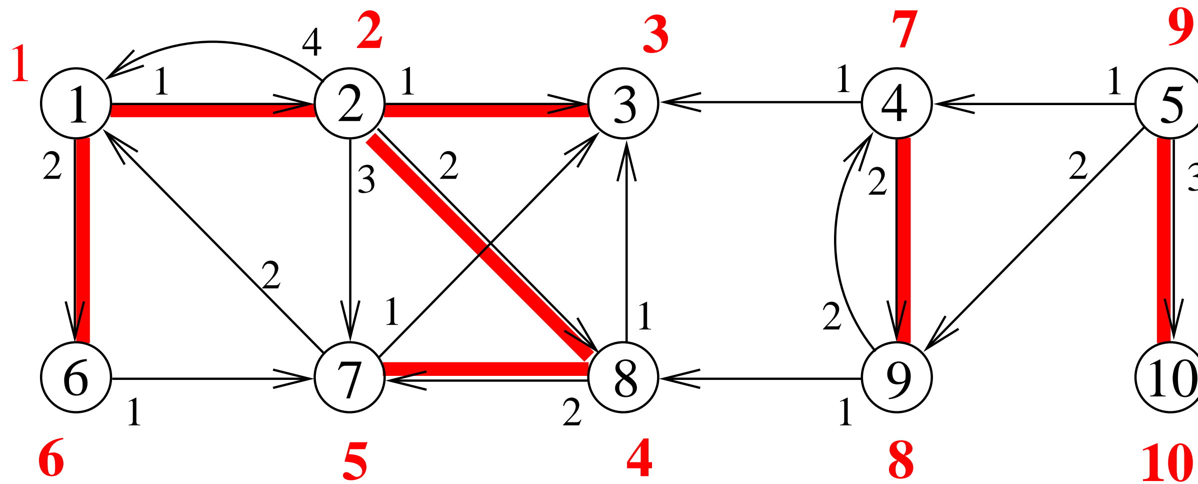
**Algorithmus DFS**$(G)$ $(*$ Tiefensuche in $G = (V, E)$ $*)$

(1) `dfs_count` $\leftarrow 0$;
(2) **for** `v` **from** $1$ **to** $n$ **do** `status[v]` $\leftarrow$ *neu*;
(3) **for** `v` **from** $1$ **to** $n$ **do**
(4)     **if** `status[v]` $=$ *neu* **then**
(5)         **dfs**(`v`); $(*$ starte Tiefensuche von `v` aus $*)$

Zu Zeilen (1)/(5): `dfs_count` wird weitergezählt.

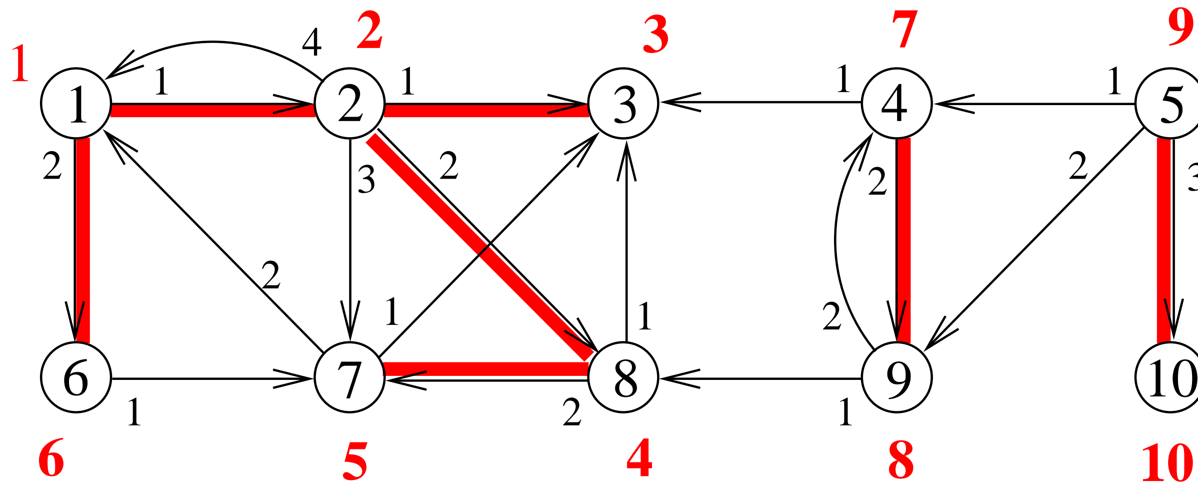
Zu Zeilen (2), (4):
Die `status[`$v$`]`-Werte werden von den **dfs**-Aufrufen in Zeile (5) verändert;
sie haben also beim Wieder-Lesen in der zweiten Schleife nicht unbedingt immer noch den Initialisierungs-Wert *neu*.


1
2
3
7
9
1
2
3
4
5
6
7
8
9
10
6
5
4
8
10
4
1
1
1
1
2
3
2
2
2
3
2
1
1
2
1
2
1




Tiefensuchwald mit dfs-Nummern.

Pfeile von $p(v)$ zu $v$:




Tiefensuchwald mit dfs-Nummern.

Pfeile von $p(v)$ zu $v$: „Entdeckungsrichtung“

DFS($G$) erzwingt, dass für **jeden** Knoten $v$ in $G$ irgendwann einmal dfs($v$) aufgerufen wird.

Daher werden auch alle Kanten, die aus $v$ herausführen, angesehen.

(Sogar: $\Theta(|V| + |E|)$, da jeder Knoten und jede Kante betrachtet wird.)

DFS($G$) erzwingt, dass für **jeden** Knoten $v$ in $G$ irgendwann einmal dfs($v$) aufgerufen wird.

Daher werden auch alle Kanten, die aus $v$ herausführen, angesehen.

**Gesamt-(Zeit-)Aufwand:** $O(1 + \mathrm{outdeg}(v))$ für Knoten $v$. Insgesamt:

$$O\Big(\sum_{v \in V} (1 + \mathrm{outdeg}(v))\Big) = O(|V| + |E|) = O(n + m).$$

(Sogar: $\Theta(|V| + |E|)$, da jeder Knoten und jede Kante betrachtet wird.)

DFS$(G)$ erzwingt, dass für **jeden** Knoten $v$ in $G$ irgendwann einmal dfs$(v)$ aufgerufen wird.

Daher werden auch alle Kanten, die aus $v$ herausführen, angesehen.

**Gesamt-(Zeit-)Aufwand:** $O(1 + \text{outdeg}(v))$ für Knoten $v$. Insgesamt:

$$O\Big(\sum_{v \in V} (1 + \text{outdeg}(v))\Big) = O(|V| + |E|) = O(n + m).$$

**Satz** Der Zeitaufwand von DFS$(G)$ ist $O(|V| + |E|)$.

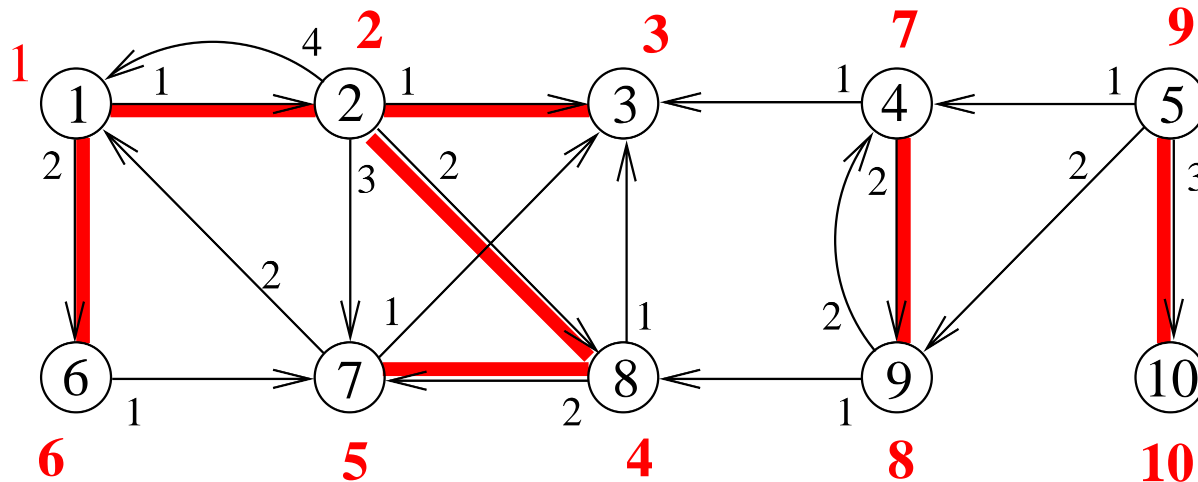
(Sogar: $\Theta(|V| + |E|)$, da jeder Knoten und jede Kante betrachtet wird.)

## Bemerkung:

**Bemerkung:** Bei DFS entstehen mehrere Bäume, die zusammen den Tiefensuch-Wald bilden.

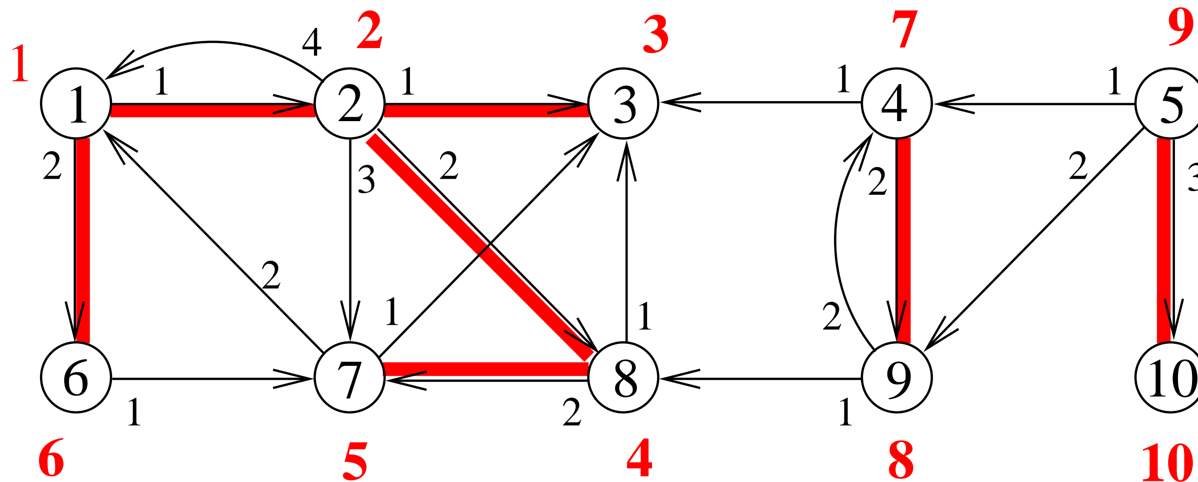
**Bemerkung:** Bei DFS entstehen mehrere Bäume, die zusammen den Tiefensuch-Wald bilden.

**Vorsicht:** Wenn ein Baum im Tiefensuchwald Wurzel $v$ hat, so ist nicht gesagt, dass dieser Baum alle von $v$ aus erreichbaren Knoten enthält. (Dies gilt nur für den ersten der Bäume.)

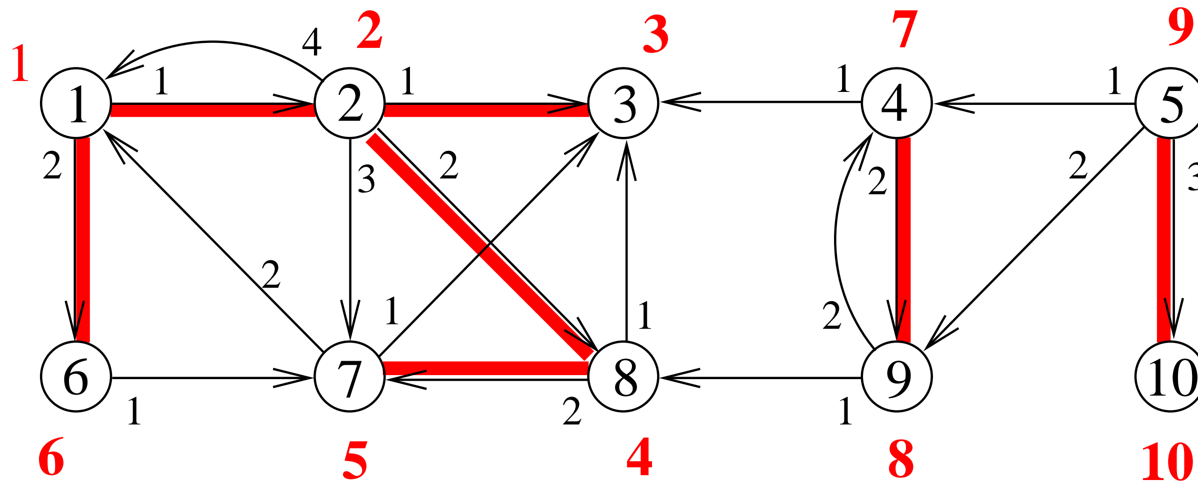


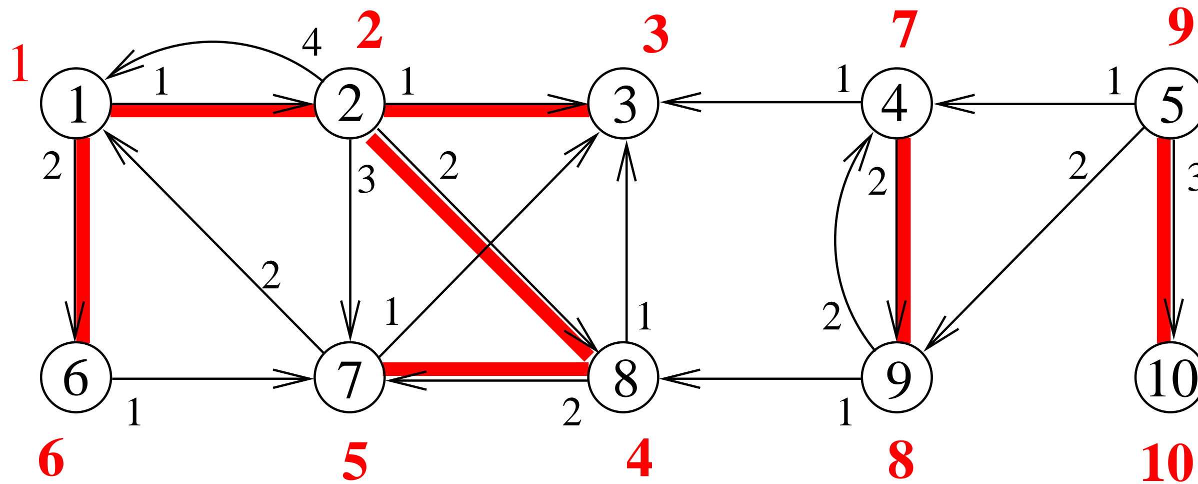
**Bemerkung:** Bei DFS entstehen mehrere Bäume, die zusammen den Tiefensuch-Wald bilden.

**Vorsicht:** Wenn ein Baum im Tiefensuchwald Wurzel $v$ hat, so ist nicht gesagt, dass dieser Baum alle von $v$ aus erreichbaren Knoten enthält. (Dies gilt nur für den ersten der Bäume.)

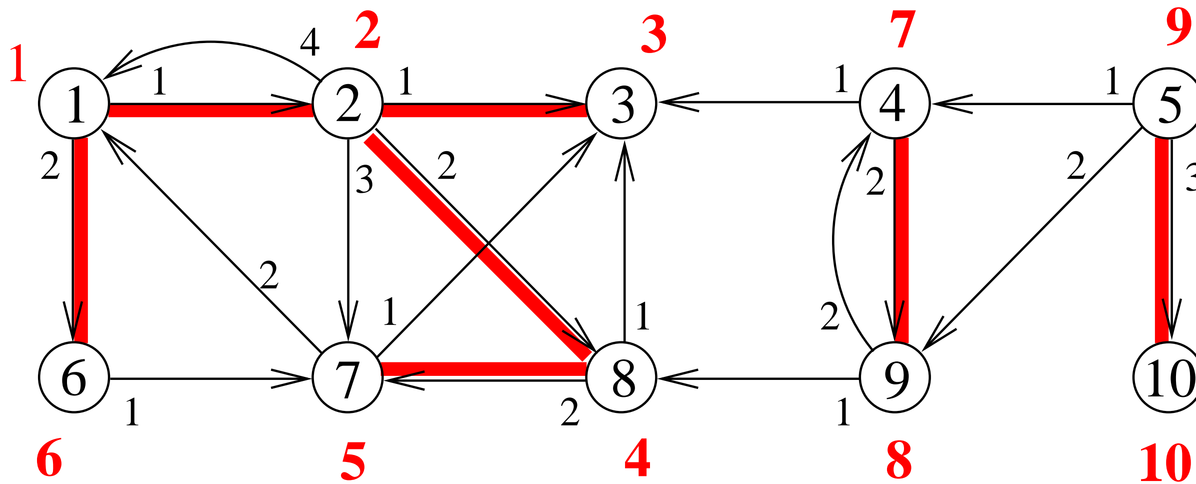



*Beispiel*: Es gilt $5 \leadsto 1$, aber $1$ ist nicht im Baum unter $5$.


1
2
3
7
9
1
2
3
4
5
6
7
8
9
10
4
1
1
1
1
2
3
2
2
2
2
3
2
1
1
1
2
2
1
6
5
4
8
10

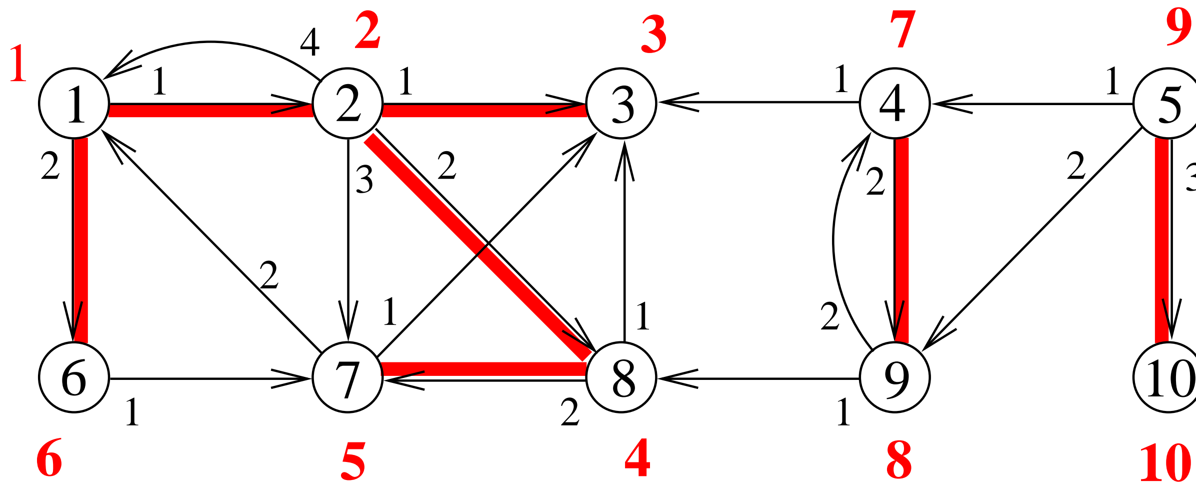



Der **Satz vom weißen Weg** gilt auch für den Tiefensuchwald.




Der **Satz vom weißen Weg** gilt auch für den Tiefensuchwald.

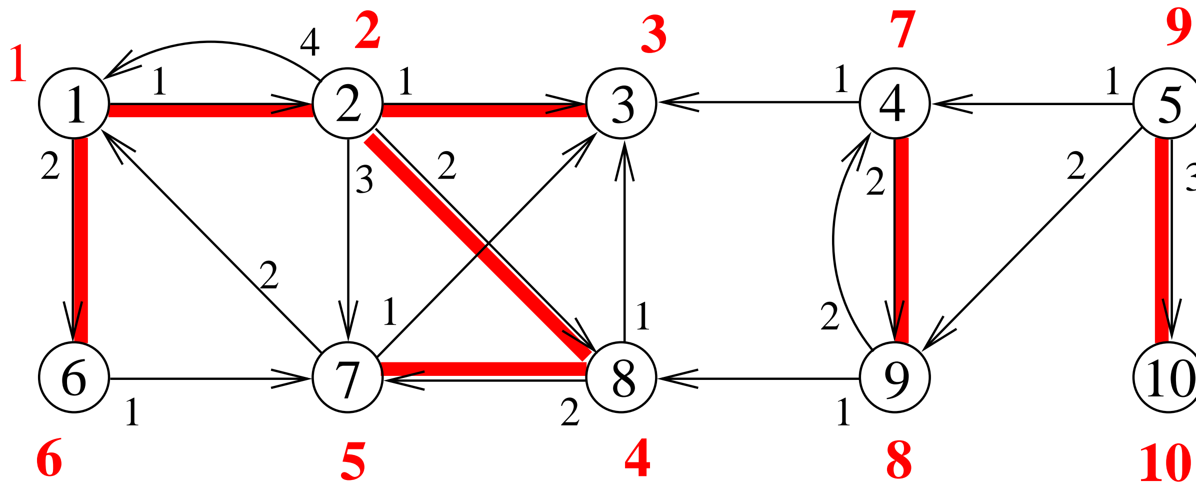
Wir definieren: $W_v = \{u \in \{1, \ldots, n\} \mid \exists w \leq v \colon w \rightsquigarrow u\}$




Der **Satz vom weißen Weg** gilt auch für den Tiefensuchwald.

Wir definieren: $W_v = \{u \in \{1,\ldots,n\} \mid \exists w \le v\colon w \rightsquigarrow u\}$

**Behauptung:** $v$ wird Wurzel eines Baums im Tiefensuch-Wald genau dann wenn $v \notin W_{v-1}$.




Der **Satz vom weißen Weg** gilt auch für den Tiefensuchwald.

Wir definieren: $W_v = \{u \in \{1, \ldots, n\} \mid \exists w \leq v \colon w \rightsquigarrow u\}$

**Behauptung:** $v$ wird Wurzel eines Baums im Tiefensuch-Wald genau dann wenn $v \notin W_{v-1}$.

In diesem Fall: Baum mit Wurzel $v$ hat als Knoten die Elemente der Menge $W_v - W_{v-1}$.

*Beweis* der Behauptung: „$\Leftarrow$“: Sei $v$ keine Wurzel eines Tiefensuchbaums. (Beispiel: Knoten $9$.)

Beim Test `status[`$v$`]=`*`neu`* in Zeile (4) ist $v$ nicht neu,

also ist $v$ in einem Tiefensuchbaum mit einer Wurzel $w < v$.

Die Kanten im Tiefensuchbaum sind auch in $E$ vorhanden, also ist $v$ von $w < v$ aus in $G$ erreichbar.

„$\Rightarrow$“: Sei $v$ von $w < v$ aus erreichbar.

Wir wählen $w \in 1, \ldots, n$ minimal mit dieser Eigenschaft, und wählen einen Weg $w = u_0, \ldots, u_t = v$.
Dann sind alle $u_i$, $1 \leq i \leq t$, größer als $w$ (wegen der Minimalität), und keines der $u_i$, $0 \leq i \leq t$, ist von einem Knoten $x < w$ aus erreichbar (wegen der Minimalität).
Daher kann keines der $u_i$ (und auch $w$ nicht) in einem Tiefensuchbaum sein, der eine Wurzel $x < w$ hat.
Daher: Wenn in Zeile (4) „`status`$[w]$ = *neu*?“ getestet wird, ist die Antwort *true*, also wird dfs$(w)$ aufgerufen.
In diesem Moment sind alle Knoten auf dem Weg $w = u_0, \ldots, u_t = v$ noch „neu“. Nach dem Satz vom weißen Weg wird $v$ Nachfahr von $w$ im Tiefensuch-Wald, also ist $v$ keine Wurzel.

## Strukturinformation 3:

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

## Strukturinformation 3:

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

Besitzt $G(V, E)$ einen Kreis, der von $v_0$ aus erreichbar ist?

**Strukturinformation 3:**
Besitzt $G = (V, E)$ einen Kreis?
Besitzt $G(V, E)$ einen Kreis, der von $v_0$ aus erreichbar ist?

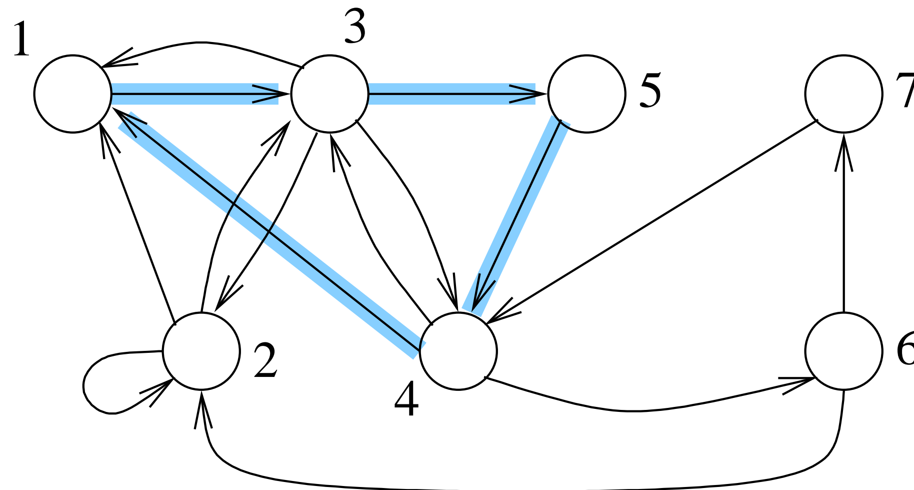
Erinnerung: (Gerichteter) Kreis in Digraphen $G$.

## Strukturinformation 3:

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

Besitzt $G(V, E)$ einen Kreis, der von $v_0$ aus erreichbar ist?

Erinnerung: (Gerichteter) Kreis in Digraphen $G$.

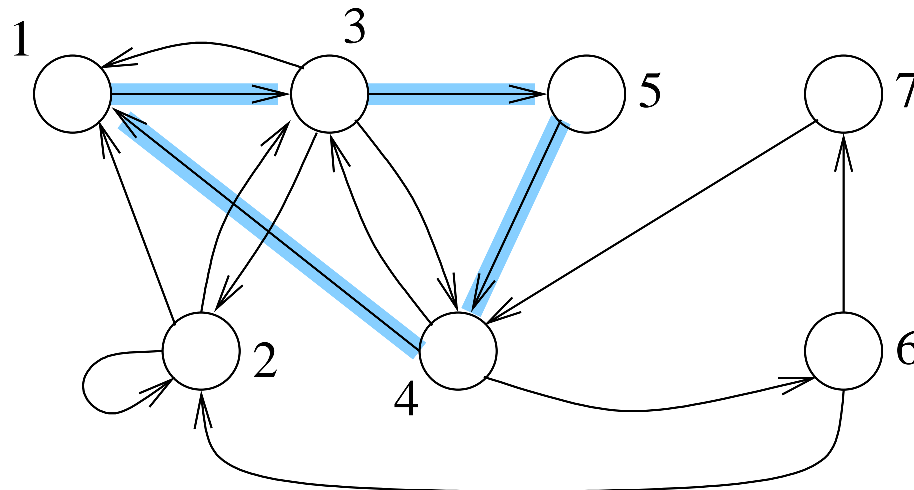


**Strukturinformation 3:**

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

Besitzt $G(V, E)$ einen Kreis, der von $v_0$ aus erreichbar ist?

Erinnerung: (Gerichteter) Kreis in Digraphen $G$.




**Anwendung:** Finden von Verklemmungen in Systemen von Prozessen, die gemeinsame Betriebsmittel benutzen.

Wir bauen dfs$(v)$ und DFS$(G)$ aus, so dass die Kanten von $G$ in vier Teilmengen $T$, $B$, $F$, $C$ eingeteilt werden:

Wir bauen dfs$(v)$ und DFS$(G)$ aus, so dass die Kanten von $G$ in vier Teilmengen $T$, $B$, $F$, $C$ eingeteilt werden:

- $\boldsymbol{T}$: **Baumkanten** – die Kanten des Tiefensuchwaldes;

Wir bauen dfs$(v)$ und DFS$(G)$ aus, so dass die Kanten von $G$ in vier Teilmengen $T$, $B$, $F$, $C$ eingeteilt werden:

- $\boldsymbol{T}$: **Baumkanten** – die Kanten des Tiefensuchwaldes;
- $\boldsymbol{B}$: **Rückwärtskanten** – $(v, w)$ ist Rückwärtskante, wenn $v$ Nachfahr von $w$ im Tiefensuchwald ist;

Wir bauen dfs$(v)$ und DFS$(G)$ aus, so dass die Kanten von $G$ in vier Teilmengen $T$, $B$, $F$, $C$ eingeteilt werden:

- $\boldsymbol{T}$: **Baumkanten** – die Kanten des Tiefensuchwaldes;
- $\boldsymbol{B}$: **Rückwärtskanten** – $(v, w)$ ist Rückwärtskante, wenn $v$ Nachfahr von $w$ im Tiefensuchwald ist;
- $\boldsymbol{F}$: **Vorwärtskanten** – $(v, w)$ ist Vorwärtskante, wenn $v$ Vorfahr von $w$ im Tiefensuchwald ist;

Wir bauen dfs$(v)$ und DFS$(G)$ aus, so dass die Kanten von $G$ in vier Teilmengen $T$, $B$, $F$, $C$ eingeteilt werden:

- $\boldsymbol{T}$: **Baumkanten** – die Kanten des Tiefensuchwaldes;
- $\boldsymbol{B}$: **Rückwärtskanten** – $(v, w)$ ist Rückwärtskante, wenn $v$ Nachfahr von $w$ im Tiefensuchwald ist;
- $\boldsymbol{F}$: **Vorwärtskanten** – $(v, w)$ ist Vorwärtskante, wenn $v$ Vorfahr von $w$ im Tiefensuchwald ist;
- $\boldsymbol{C}$: **Querkanten** – $(v, w)$ ist Querkante, wenn der gesamte Unterbaum von $w$ fertig abgearbeitet ist, wenn die Kante $(v, w)$ getestet wird
  ($w$ liegt im Tiefensuchwald gemäß Präorder- und gemäß Postorder-Reihenfolge vor $v$).

Zudem nummerieren wir (in Gedanken) die Knoten $v \in V$ **in der Reihenfolge** durch,

Zudem nummerieren wir (in Gedanken) die Knoten $v \in V$ **in der Reihenfolge** durch, in der die **dfs**$(v)$-Aufrufe **beendet** werden.

Zudem nummerieren wir (in Gedanken) die Knoten $v \in V$ **in der Reihenfolge** durch, in der die **dfs**$(v)$-Aufrufe **beendet** werden.

Diese Nummern heißen **fin-Nummern** oder **f-Nummern**: `f_num`$(v)$.

Zudem nummerieren wir (in Gedanken) die Knoten $v \in V$ **in der Reihenfolge** durch, in der die **dfs**$(v)$-Aufrufe **beendet** werden.

Diese Nummern heißen **fin-Nummern** oder **f-Nummern**: $\texttt{f_num}(v)$.

Der Wertebereich der f-Nummern ist $\{1, 2, \ldots, |V|\}$.

**Algorithmus dfs**$(v)$ $(*$ Tiefensuche von $v$ aus, voll ausgebaut $*)$

**Algorithmus dfs**$(v)$ (∗ Tiefensuche von $v$ aus, voll ausgebaut ∗)
(∗ darf nur für $v$ mit `status[`$v$`]` $=$ *neu* aufgerufen werden ∗)

**Algorithmus dfs**$(v)$ (∗ Tiefensuche von $v$ aus, voll ausgebaut ∗)
(∗ darf nur für $v$ mit status[$v$] = *neu* aufgerufen werden ∗)

(1) dfs_count++;

(2) dfs_num[$v$] ← dfs_count;

**Algorithmus dfs**$(v)$ $(*$ Tiefensuche von $v$ aus, voll ausgebaut $*)$

$(*$ darf nur für $v$ mit `status`$[v] =$ *neu* aufgerufen werden $*)$

(1) `dfs_count++`;

(2) `dfs_num`$[v] \leftarrow$ `dfs_count`;

(3) **dfs-visit**$(v)$; $(*$ Aktion an $v$ bei Erstbesuch $*)$

**Algorithmus dfs**$(v)$ $(*$ Tiefensuche von $v$ aus, voll ausgebaut $*)$
$(*$ darf nur für $v$ mit `status`$[v]$ $=$ *neu* aufgerufen werden $*)$

(1) `dfs_count++`;
(2) `dfs_num`$[v] \leftarrow$ `dfs_count`;
(3) **dfs-visit**$(v)$; $(*$ Aktion an $v$ bei Erstbesuch $*)$
(4) `status`$[v] \leftarrow$ **aktiv** ;

**Algorithmus dfs**$(v)$ (* Tiefensuche von $v$ aus, voll ausgebaut *)
(* darf nur für $v$ mit `status`$[v]$ = *neu* aufgerufen werden *)

(1) `dfs_count++`;
(2) `dfs_num`$[v] \leftarrow$ `dfs_count`;
(3) **dfs-visit**$(v)$; (* Aktion an $v$ bei Erstbesuch *)
(4) `status`$[v] \leftarrow$ **aktiv** ;
(5) für jeden Nachfolger `w` von $v$ (Adjazenzliste!) tue:

**Algorithmus dfs**$(v)$ (* Tiefensuche von $v$ aus, voll ausgebaut *)

(* darf nur für $v$ mit $\texttt{status}[v]$ = *neu* aufgerufen werden *)

(1) $\texttt{dfs_count++}$;

(2) $\texttt{dfs_num}[v] \leftarrow \texttt{dfs_count}$;

(3) **dfs-visit**$(v)$; (* Aktion an $v$ bei Erstbesuch *)

(4) $\texttt{status}[v] \leftarrow$ **aktiv** ;

(5) für jeden Nachfolger $\texttt{w}$ von $v$ (Adjazenzliste!) tue:

(6) **1. Fall:** $\texttt{status}[\texttt{w}] =$ **neu**: $T \leftarrow T \cup \{(v, \texttt{w})\}$; **dfs**$(\texttt{w})$;

**Algorithmus dfs**$(v)$ (* Tiefensuche von $v$ aus, voll ausgebaut *)

(* darf nur für $v$ mit `status`$[v]$ = *neu* aufgerufen werden *)

(1) `dfs_count++`;

(2) `dfs_num`$[v] \leftarrow$ `dfs_count`;

(3) **dfs-visit**$(v)$; (* Aktion an $v$ bei Erstbesuch *)

(4) `status`$[v] \leftarrow$ **aktiv** ;

(5) für jeden Nachfolger `w` von $v$ (Adjazenzliste!) tue:

(6) **1. Fall:** `status[w]` = **neu**: $T \leftarrow T \cup \{(v, \mathtt{w})\}$; **dfs**(`w`);

(7) **2. Fall:** `status[w]` = **aktiv**: $B \leftarrow B \cup \{(v, \mathtt{w})\}$;

(8) **3. Fall:** `status[w]` = **fertig** $\wedge$ `dfs_num`$[v] <$ `dfs_num[w]`:
$F \leftarrow F \cup \{(v, \mathtt{w})\}$;

(9) **4. Fall:** `status[w]` = **fertig** $\wedge$ `dfs_num`$[v] >$ `dfs_num[w]`:
$C \leftarrow C \cup \{(v, \mathtt{w})\}$;

(10) `f_count++`;

**Algorithmus dfs**$(v)$ $(*$ Tiefensuche von $v$ aus, voll ausgebaut $*)$

$(*$ darf nur für $v$ mit $\texttt{status}[v] =$ *neu* aufgerufen werden $*)$

(1) `dfs_count++;`

(2) $\texttt{dfs_num}[v] \leftarrow \texttt{dfs_count}$;

(3) **dfs-visit**$(v)$; $(*$ Aktion an $v$ bei Erstbesuch $*)$

(4) $\texttt{status}[v] \leftarrow$ **aktiv** ;

(5) für jeden Nachfolger `w` von $v$ (Adjazenzliste!) tue:

(6) **1. Fall:** $\texttt{status}[\texttt{w}] =$ **neu**: $T \leftarrow T \cup \{(v, \texttt{w})\}$; **dfs**$(\texttt{w})$;

(7) **2. Fall:** $\texttt{status}[\texttt{w}] =$ **aktiv**: $B \leftarrow B \cup \{(v, \texttt{w})\}$;

(8) **3. Fall:** $\texttt{status}[\texttt{w}] =$ **fertig** $\wedge\ \texttt{dfs_num}[v] < \texttt{dfs_num}[\texttt{w}]$:
$F \leftarrow F \cup \{(v, \texttt{w})\}$;

(9) **4. Fall:** $\texttt{status}[\texttt{w}] =$ **fertig** $\wedge\ \texttt{dfs_num}[v] > \texttt{dfs_num}[\texttt{w}]$:
$C \leftarrow C \cup \{(v, \texttt{w})\}$;

(10) `f_count++;`

(11) $\texttt{f_num}[v] \leftarrow \texttt{f_count}$;

**Algorithmus dfs**$(v)$ (* Tiefensuche von $v$ aus, voll ausgebaut *)

(* darf nur für $v$ mit `status[`$v$`]` = *neu* aufgerufen werden *)

(1) `dfs_count++`;

(2) `dfs_num[`$v$`]` $\leftarrow$ `dfs_count`;

(3) **dfs-visit**$(v)$; (* Aktion an $v$ bei Erstbesuch *)

(4) `status[`$v$`]` $\leftarrow$ **aktiv** ;

(5) für jeden Nachfolger `w` von $v$ (Adjazenzliste!) tue:

(6) **1. Fall:** `status[w]` = **neu**: $T \leftarrow T \cup \{(v, \mathtt{w})\}$; **dfs**(`w`);

(7) **2. Fall:** `status[w]` = **aktiv**: $B \leftarrow B \cup \{(v, \mathtt{w})\}$;

(8) **3. Fall:** `status[w]` = **fertig** $\wedge$ `dfs_num[`$v$`]` $<$ `dfs_num[w]`:
$F \leftarrow F \cup \{(v, \mathtt{w})\}$;

(9) **4. Fall:** `status[w]` = **fertig** $\wedge$ `dfs_num[`$v$`]` $>$ `dfs_num[w]`:
$C \leftarrow C \cup \{(v, \mathtt{w})\}$;

(10) `f_count++`;

(11) `f_num[`$v$`]` $\leftarrow$ `f_count`;

(12) **fin-visit**$(v)$; (* Aktion an $v$ bei letztem Besuch *)

**Algorithmus dfs**$(v)$ (∗ Tiefensuche von $v$ aus, voll ausgebaut ∗)

(∗ darf nur für $v$ mit $\texttt{status}[v] =$ *neu* aufgerufen werden ∗)

(1) `dfs_count++;`

(2) $\texttt{dfs_num}[v] \leftarrow \texttt{dfs_count}$;

(3) **dfs-visit**$(v)$; (∗ Aktion an $v$ bei Erstbesuch ∗)

(4) $\texttt{status}[v] \leftarrow$ **aktiv** ;

(5) für jeden Nachfolger `w` von $v$ (Adjazenzliste!) tue:

(6) **1. Fall:** $\texttt{status}[\texttt{w}] =$ **neu**: $T \leftarrow T \cup \{(v, \texttt{w})\}$; **dfs**$(\texttt{w})$;

(7) **2. Fall:** $\texttt{status}[\texttt{w}] =$ **aktiv**: $B \leftarrow B \cup \{(v, \texttt{w})\}$;

(8) **3. Fall:** $\texttt{status}[\texttt{w}] =$ **fertig** $\wedge\ \texttt{dfs_num}[v] < \texttt{dfs_num}[\texttt{w}]$:
$F \leftarrow F \cup \{(v, \texttt{w})\}$;

(9) **4. Fall:** $\texttt{status}[\texttt{w}] =$ **fertig** $\wedge\ \texttt{dfs_num}[v] > \texttt{dfs_num}[\texttt{w}]$:
$C \leftarrow C \cup \{(v, \texttt{w})\}$;

(10) `f_count++;`

(11) $\texttt{f_num}[v] \leftarrow \texttt{f_count}$;

(12) **fin-visit**$(v)$; (∗ Aktion an $v$ bei letztem Besuch ∗)

(13) $\texttt{status}[v] \leftarrow$ **fertig**.

Tiefensuche von $v_0$ aus, entdeckt den von $v_0$ aus erreichbaren Teil des Graphen:

Tiefensuche von $v_0$ aus, entdeckt den von $v_0$ aus erreichbaren Teil des Graphen:

Initialisierung: `dfs_count` $\leftarrow 0$; `f_count` $\leftarrow 0$.

Tiefensuche von $v_0$ aus, entdeckt den von $v_0$ aus erreichbaren Teil des Graphen:

Initialisierung: `dfs_count` $\leftarrow 0$; `f_count` $\leftarrow 0$.

Alle Knoten sind „*neu*“; Mengen $T, B, F, C$ sind leer.

Tiefensuche von $v_0$ aus, entdeckt den von $v_0$ aus erreichbaren Teil des Graphen:

Initialisierung: `dfs_count` $\leftarrow 0$; `f_count` $\leftarrow 0$.

Alle Knoten sind „*neu*“; Mengen $T, B, F, C$ sind leer.

Rufe **dfs**$(v_0)$ auf.

Tiefensuche von $v_0$ aus, entdeckt den von $v_0$ aus erreichbaren Teil des Graphen:
Initialisierung: `dfs_count` $\leftarrow 0$; `f_count` $\leftarrow 0$.
Alle Knoten sind „*neu*“; Mengen $T, B, F, C$ sind leer.
Rufe **dfs**$(v_0)$ auf.

Tiefensuche für den ganzen Graphen: **DFS**$(G)$ wie vorher, nur mit der „voll ausgebauten“ **dfs**-Prozedur, und der Initialisierung von `f_count` und $T, B, F, C$.

Tiefensuche von $v_0$ aus, entdeckt den von $v_0$ aus erreichbaren Teil des Graphen:

Initialisierung: $\texttt{dfs_count} \leftarrow 0$; $\texttt{f_count} \leftarrow 0$.

Alle Knoten sind „*neu*“; Mengen $T, B, F, C$ sind leer.

Rufe **dfs**$(v_0)$ auf.

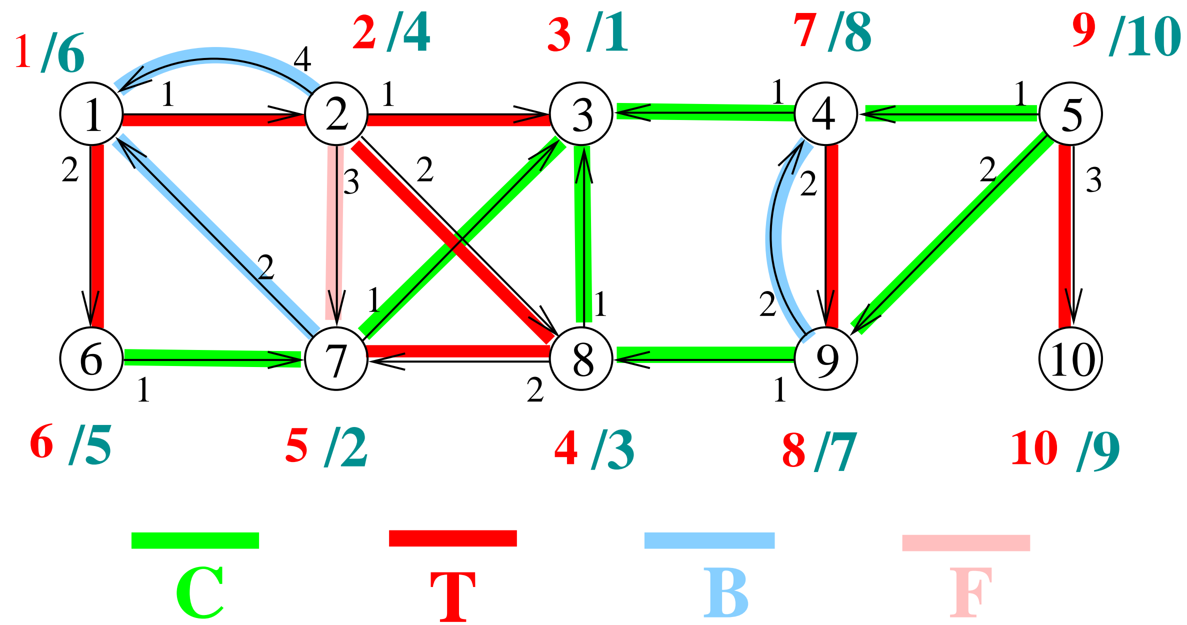
Tiefensuche für den ganzen Graphen: **DFS**$(G)$ wie vorher, nur mit der „voll ausgebauten“ **dfs**-Prozedur, und der Initialisierung von $\texttt{f_count}$ und $T, B, F, C$.

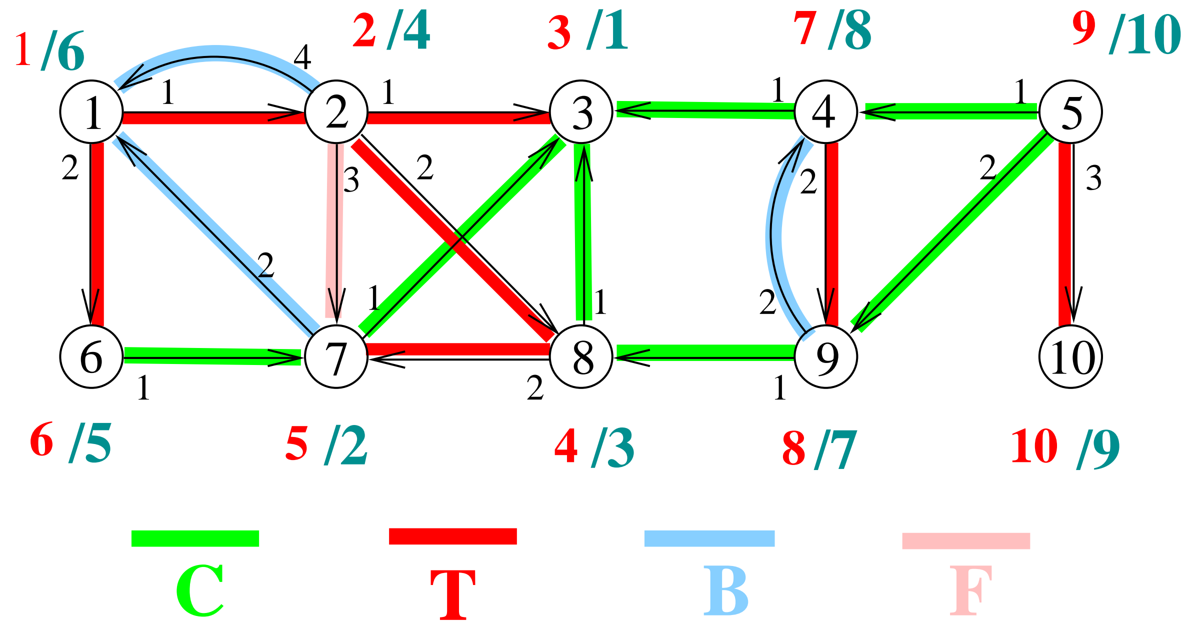
**Algorithmus DFS**$(G)$ $(*$ Tiefensuche in $G = (V, E)$, voll ausgebaut $*)$

Tiefensuche von $v_0$ aus, entdeckt den von $v_0$ aus erreichbaren Teil des Graphen:
Initialisierung: `dfs_count` $\leftarrow 0$; `f_count` $\leftarrow 0$.
Alle Knoten sind „*neu*“; Mengen $T, B, F, C$ sind leer.
Rufe **dfs**$(v_0)$ auf.

Tiefensuche für den ganzen Graphen: **DFS**$(G)$ wie vorher, nur mit der „voll ausgebauten“ **dfs**-Prozedur, und der Initialisierung von `f_count` und $T, B, F, C$.

**Algorithmus DFS**$(G)$ (∗ Tiefensuche in $G = (V, E)$, voll ausgebaut ∗)
(1) `dfs_count` $\leftarrow 0$;
(2) `f_count` $\leftarrow 0$;
(3) $T \leftarrow \emptyset$; $B \leftarrow \emptyset$; $F \leftarrow \emptyset$; $C \leftarrow \emptyset$;
(3) **for** v **from** $1$ **to** $n$ **do** `status[v]` $\leftarrow$ *neu*;
(4) **for** v **from** $1$ **to** $n$ **do**
(5)     **if** `status[v]` $=$ *neu* **then**
(6)         **dfs**(v); (∗ starte Tiefensuche von v aus ∗)


1 /6
2 /4
3 /1
7 /8
9 /10
6 /5
5 /2
4 /3
8 /7
10 /9
C
T
B
F




Ergebnis der Kantenklassifikation am Beispiel-Digraphen.

**Klar:** Alle Laufzeitaussagen für die einfache dfs-Prozedur behalten weiter Gültigkeit.

**Beobachtung 1:** Jede Kante $(v, w)$ wird genau einmal betrachtet und daher genau in eine der Mengen $T$, $B$, $F$ und $C$ eingeordnet.

**Beobachtung 1:** Jede Kante $(v, w)$ wird genau einmal betrachtet und daher genau in eine der Mengen $T$, $B$, $F$ und $C$ eingeordnet.

**Beobachtung 2:** Die dfs-Nummern entsprechen einem Präorder-Durchlauf, die f-Nummern einem Postorder-Durchlauf durch die Bäume des Tiefensuch-Waldes.

**Beobachtung 3:** Wenn **dfs**$(w)$ nach **dfs**$(v)$ beginnt und vor **dfs**$(v)$ endet, also **dfs**$(w)$ (indirekt) von **dfs**$(v)$ aus aufgerufen wird, dann gilt:

(a) $\texttt{dfs_num}[v] < \texttt{dfs_num}[w]$ und $\texttt{f_num}[v] > \texttt{f_num}[w]$;

(b) $v$ ist Vorfahr von $w$ im Tiefensuch-Wald.

**Beobachtung 3:** Wenn **dfs**$(w)$ nach **dfs**$(v)$ beginnt und vor **dfs**$(v)$ endet, also **dfs**$(w)$ (indirekt) von **dfs**$(v)$ aus aufgerufen wird, dann gilt:

(a) $\texttt{dfs_num}[v] < \texttt{dfs_num}[w]$ und $\texttt{f_num}[v] > \texttt{f_num}[w]$;

(b) $v$ ist Vorfahr von $w$ im Tiefensuch-Wald.

**Konsequenz:** Die **Vorfahr-Nachfahr-Relation** im Tiefensuch-Wald lässt sich in Zeit $O(1)$ testen, wenn die dfs- und die f-Nummern bekannt sind.

## Satz

Die Tiefensuch-Prozedur klassifiziert die Kanten korrekt.

*Beweis*:

## Satz

Die Tiefensuch-Prozedur klassifiziert die Kanten korrekt.

*Beweis*:

**1. Fall: dfs**$(w)$ wird direkt aus **dfs**$(v)$ aufgerufen. – Dann ist $(v, w)$ Baumkante und wird richtig klassifiziert.

## Satz

Die Tiefensuch-Prozedur klassifiziert die Kanten korrekt.

*Beweis*:

**1. Fall: dfs**$(w)$ wird direkt aus **dfs**$(v)$ aufgerufen. – Dann ist $(v, w)$ Baumkante und wird richtig klassifiziert.

**2. Fall:** In **dfs**$(v)$ wird die Kante $(v, w)$ untersucht und es stellt sich heraus, dass $w$ *aktiv* ist. – Dann liegt $w$ auf dem Weg von der Wurzel des Tiefensuchbaums zu $v$, ist also Vorfahr von $v$ (oder gleich $v$ – Schleifen sind nicht verboten). Daher ist $(v, w)$ Rückwärtskante und wird richtig klassifiziert.

**3. Fall:** In **dfs**$(v)$ wird die Kante $(v, w)$ untersucht und es stellt sich heraus, dass $w$ *fertig* ist und dass $\texttt{dfs_num}[v] < \texttt{dfs_num}[w]$ ist. –

Dann wird zwangsläufig $\texttt{f_num}[v] > \texttt{f_num}[w]$, also ist nach Beobachtung 3 $w$ Nachfahr von $v$ im Tiefensuchwald. Daher ist $(v, w)$ Vorwärtskante und wird richtig klassifiziert.

**3. Fall:** In **dfs**$(v)$ wird die Kante $(v, w)$ untersucht und es stellt sich heraus, dass $w$ *fertig* ist und dass $\texttt{dfs_num}[v] < \texttt{dfs_num}[w]$ ist. –

Dann wird zwangsläufig $\texttt{f_num}[v] > \texttt{f_num}[w]$, also ist nach Beobachtung 3 $w$ Nachfahr von $v$ im Tiefensuchwald. Daher ist $(v, w)$ Vorwärtskante und wird richtig klassifiziert.

**4. Fall:** In **dfs**$(v)$ wird die Kante $(v, w)$ untersucht und es stellt sich heraus, dass $w$ *fertig* ist und dass $\texttt{dfs_num}[v] > \texttt{dfs_num}[w]$ ist. –

Dann ist die Untersuchung von $w$ und all seinen Nachfolgern im Wald abgeschlossen, bevor $v$ begonnen wird. Daher ist $(v, w)$ Querkante und wird richtig klassifiziert.

# Kreisfreiheitstest

## Satz

(a) $G$ enthält einen Kreis $\Leftrightarrow$

# Kreisfreiheitstest

## Satz

(a) $G$ enthält einen Kreis $\Leftrightarrow$ $B \neq \emptyset$.

# Kreisfreiheitstest

## Satz

(a) $G$ enthält einen Kreis $\Leftrightarrow$ $B \neq \emptyset$.

(Nach Aufruf **DFS**$(G)$.)

(b) Der von $v$ aus erreichbare Teil $R_v = \{w \in V \mid v \rightsquigarrow w\}$ von $G$ (mit den darin verlaufenden Kanten aus $E$) enthält einen Kreis

$\Leftrightarrow$

# Kreisfreiheitstest

## Satz

(a) $G$ enthält einen Kreis $\Leftrightarrow$ $B \neq \emptyset$.

(Nach Aufruf **DFS**$(G)$.)

(b) Der von $v$ aus erreichbare Teil $R_v = \{w \in V \mid v \rightsquigarrow w\}$ von $G$ (mit den darin verlaufenden Kanten aus $E$) enthält einen Kreis

$\Leftrightarrow$ Aufruf **dfs**$(v)$ liefert $B \neq \emptyset$.

*Beweis* des Satzes:

*Beweis* des Satzes: Nur Teil (a).

*Beweis* des Satzes: Nur Teil (a).

„$\Leftarrow$“:

*Beweis* des Satzes: Nur Teil (a).

„$\Leftarrow$“: Wenn DFS$(G)$ eine Rückwärtskante $(v, w)$ gefunden hat, ergibt sich folgende Situation:

*Beweis* des Satzes: Nur Teil (a).

„$\Leftarrow$“: Wenn DFS$(G)$ eine Rückwärtskante $(v, w)$ gefunden hat, ergibt sich folgende Situation:

$w$ ist Vorfahr von $v$, also gibt es einen Weg (aus Baumkanten)

*Beweis* des Satzes: Nur Teil (a).

„$\Leftarrow$“: Wenn DFS$(G)$ eine Rückwärtskante $(v, w)$ gefunden hat, ergibt sich folgende Situation:

$w$ ist Vorfahr von $v$, also gibt es einen Weg (aus Baumkanten)

$w = u_0, u_1, \ldots, u_t = v$ in $G$, $t \geq 0$.

*Beweis* des Satzes: Nur Teil (a).

„$\Leftarrow$“: Wenn DFS($G$) eine Rückwärtskante $(v, w)$ gefunden hat, ergibt sich folgende Situation:

$w$ ist Vorfahr von $v$, also gibt es einen Weg (aus Baumkanten)

$w = u_0, u_1, \ldots, u_t = v$ in $G$, $t \geq 0$.

Zusammen mit der Kante $(v, w)$ erhalten wir einen Kreis.

*Beweis* des Satzes: Nur Teil (a).

„$\Leftarrow$“: Wenn DFS$(G)$ eine Rückwärtskante $(v, w)$ gefunden hat, ergibt sich folgende Situation:

$w$ ist Vorfahr von $v$, also gibt es einen Weg (aus Baumkanten)

$w = u_0, u_1, \ldots, u_t = v$ in $G$, $t \geq 0$.

Zusammen mit der Kante $(v, w)$ erhalten wir einen Kreis.

**Spezialfall:** $v = w$, $t = 0$: **Schleife.** Wird als Kreis erkannt.

*Beweis* des Satzes: Nur Teil (a).

„$\Rightarrow$“:

*Beweis* des Satzes: Nur Teil (a).
„$\Rightarrow$“: Nun sei $B = \emptyset$.

*Beweis* des Satzes: Nur Teil (a).

„$\Rightarrow$“: Nun sei $B = \emptyset$.

Für eine Kante $(v, w)$ gibt es dann nur noch drei Fälle:

*Beweis* des Satzes: Nur Teil (a).

„$\Rightarrow$“: Nun sei $B = \emptyset$.

Für eine Kante $(v, w)$ gibt es dann nur noch drei Fälle:

**1. Fall:** $(v, w)$ ist Baumkante.

*Beweis* des Satzes: Nur Teil (a).

„$\Rightarrow$“: Nun sei $B = \emptyset$.

Für eine Kante $(v, w)$ gibt es dann nur noch drei Fälle:

**1. Fall:** $(v, w)$ ist Baumkante. – Dann ist `f_num[`$v$`]` $>$ `f_num[`$w$`]`.

*Beweis* des Satzes: Nur Teil (a).

„$\Rightarrow$“: Nun sei $B = \emptyset$.

Für eine Kante $(v, w)$ gibt es dann nur noch drei Fälle:

**1. Fall:** $(v, w)$ ist Baumkante. – Dann ist $\texttt{f_num}[v] > \texttt{f_num}[w]$.

**2. Fall:** $(v, w)$ ist Vorwärtskante.

*Beweis* des Satzes: Nur Teil (a).

„$\Rightarrow$“: Nun sei $B = \emptyset$.

Für eine Kante $(v, w)$ gibt es dann nur noch drei Fälle:

**1. Fall:** $(v, w)$ ist Baumkante. – Dann ist f_num[$v$] $>$ f_num[$w$].

**2. Fall:** $(v, w)$ ist Vorwärtskante. – Dann ist $w$ „fertig“, während an $v$ noch gearbeitet wird, also ist f_num[$v$] $>$ f_num[$w$].

*Beweis* des Satzes: Nur Teil (a).

„$\Rightarrow$“: Nun sei $B = \emptyset$.

Für eine Kante $(v, w)$ gibt es dann nur noch drei Fälle:

**1. Fall:** $(v, w)$ ist Baumkante. – Dann ist `f_num[`$v$`]` $>$ `f_num[`$w$`]`.

**2. Fall:** $(v, w)$ ist Vorwärtskante. – Dann ist $w$ „fertig“, während an $v$ noch gearbeitet wird, also ist `f_num[`$v$`]` $>$ `f_num[`$w$`]`.

**3. Fall:** $(v, w)$ ist Querkante.

*Beweis* des Satzes: Nur Teil (a).

„$\Rightarrow$": Nun sei $B = \emptyset$.

Für eine Kante $(v, w)$ gibt es dann nur noch drei Fälle:

**1. Fall:** $(v, w)$ ist Baumkante. – Dann ist `f_num[`$v$`]` $>$ `f_num[`$w$`]`.

**2. Fall:** $(v, w)$ ist Vorwärtskante. – Dann ist $w$ „fertig", während an $v$ noch gearbeitet wird, also ist `f_num[`$v$`]` $>$ `f_num[`$w$`]`.

**3. Fall:** $(v, w)$ ist Querkante. – Wie 2. Fall.

*Beweis* des Satzes: Nur Teil (a).

„$\Rightarrow$“: Nun sei $B = \emptyset$.

Für eine Kante $(v, w)$ gibt es dann nur noch drei Fälle:

**1. Fall:** $(v, w)$ ist Baumkante. – Dann ist f_num[$v$] $>$ f_num[$w$].

**2. Fall:** $(v, w)$ ist Vorwärtskante. – Dann ist $w$ „fertig“, während an $v$ noch gearbeitet wird, also ist f_num[$v$] $>$ f_num[$w$].

**3. Fall:** $(v, w)$ ist Querkante. – Wie 2. Fall.

Wenn also $u_0, \ldots, u_t$ irgendein Weg in $G$ der Länge $t \geq 1$ ist, so nehmen entlang dieses Weges die f-Nummern strikt ab.

*Beweis* des Satzes: Nur Teil (a).

„$\Rightarrow$“: Nun sei $B = \emptyset$.

Für eine Kante $(v, w)$ gibt es dann nur noch drei Fälle:

**1. Fall:** $(v, w)$ ist Baumkante. – Dann ist `f_num[`$v$`]` $>$ `f_num[`$w$`]`.

**2. Fall:** $(v, w)$ ist Vorwärtskante. – Dann ist $w$ „fertig“, während an $v$ noch gearbeitet wird, also ist `f_num[`$v$`]` $>$ `f_num[`$w$`]`.

**3. Fall:** $(v, w)$ ist Querkante. – Wie 2. Fall.

Wenn also $u_0, \ldots, u_t$ irgendein Weg in $G$ der Länge $t \geq 1$ ist, so nehmen entlang dieses Weges die f-Nummern strikt ab. Daher kann auf keinen Fall $u_0 = u_t$ sein.

*Beweis* des Satzes: Nur Teil (a).

„$\Rightarrow$“: Nun sei $B = \emptyset$.

Für eine Kante $(v, w)$ gibt es dann nur noch drei Fälle:

**1. Fall:** $(v, w)$ ist Baumkante. – Dann ist f_num[$v$] $>$ f_num[$w$].

**2. Fall:** $(v, w)$ ist Vorwärtskante. – Dann ist $w$ „fertig“, während an $v$ noch gearbeitet wird, also ist f_num[$v$] $>$ f_num[$w$].

**3. Fall:** $(v, w)$ ist Querkante. – Wie 2. Fall.

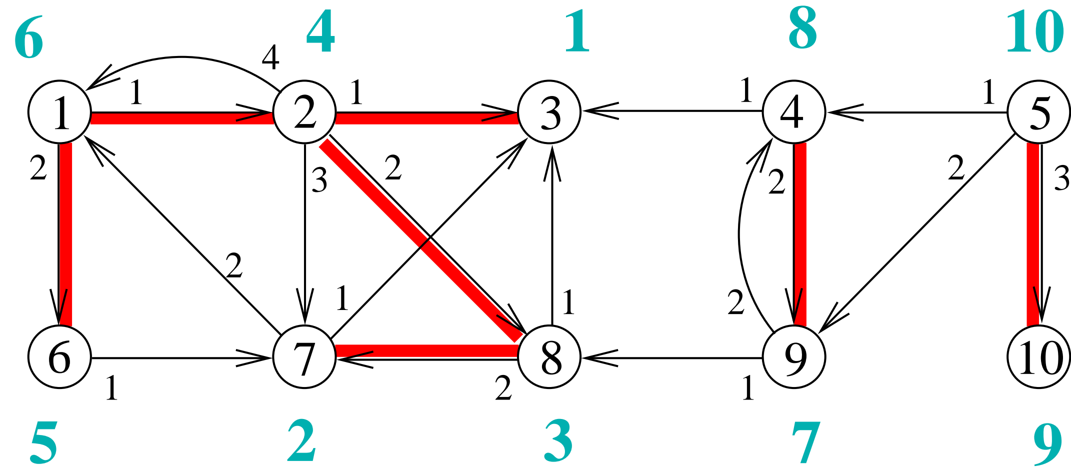
Wenn also $u_0, \ldots, u_t$ irgendein Weg in $G$ der Länge $t \geq 1$ ist, so nehmen entlang dieses Weges die f-Nummern strikt ab.

Daher kann auf keinen Fall $u_0 = u_t$ sein.

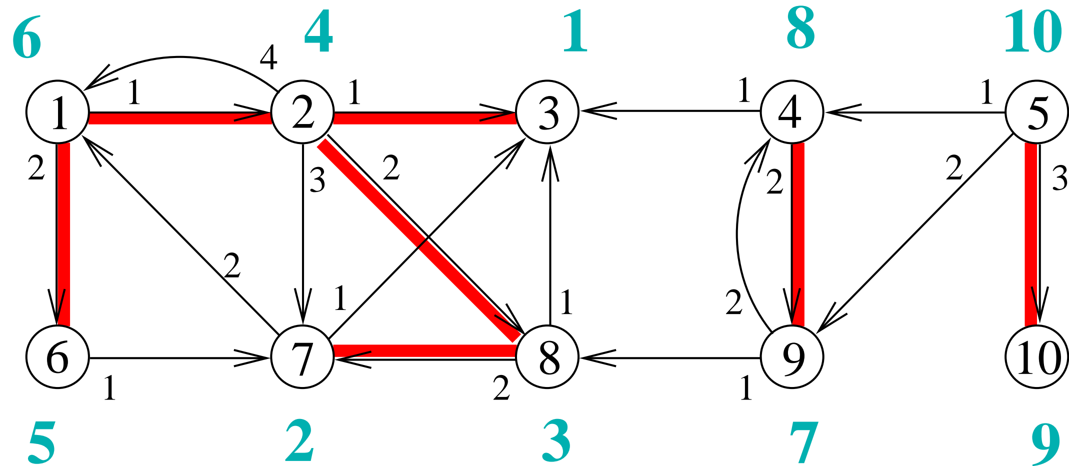
Also enthält $G$ keinen Kreis.

# Ein (zyklischer) Digraph mit f-Nummern:

# Ein (zyklischer) Digraph mit f-Nummern:

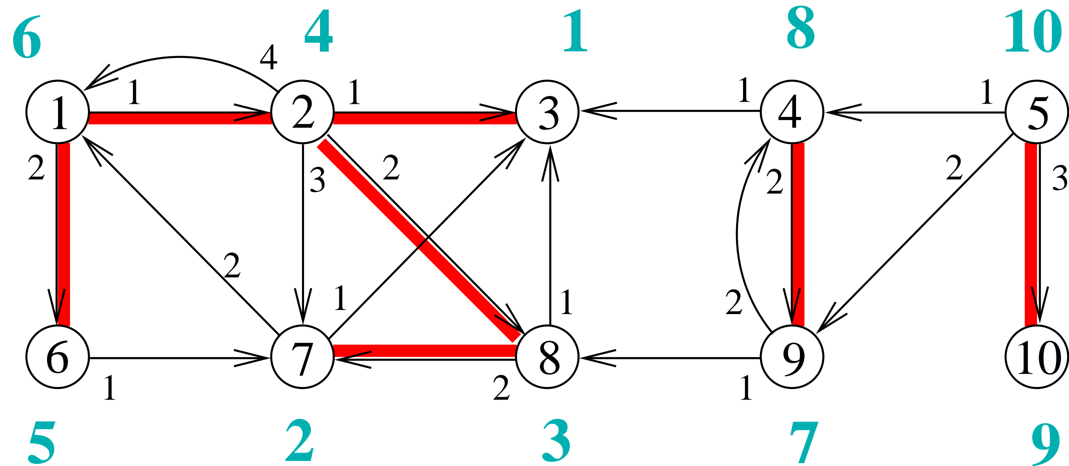


Ein (zyklischer) Digraph mit f-Nummern:

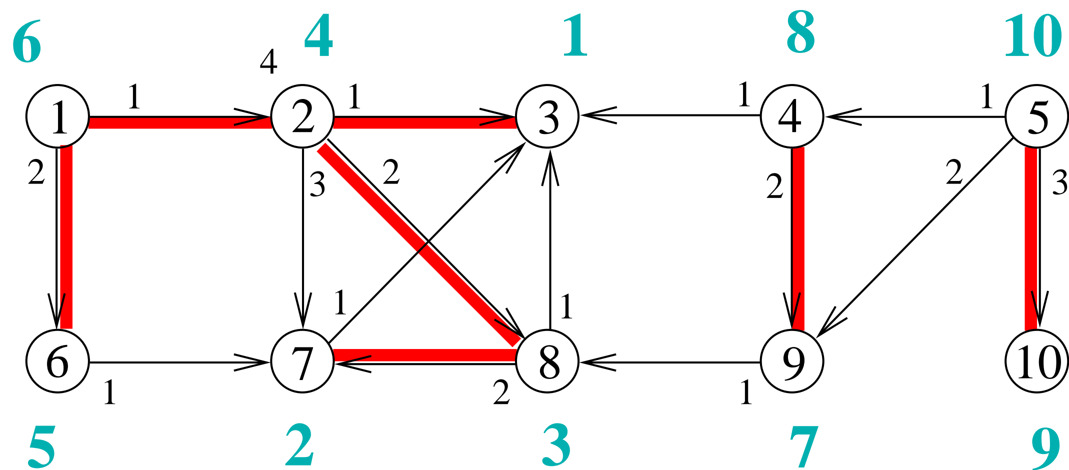



Ein (azyklischer) Digraph mit f-Nummern:

## Ein (zyklischer) Digraph mit f-Nummern:




## Ein (azyklischer) Digraph mit f-Nummern:



**Definition** Ein gerichteter Graph (Digraph), der keine Kreise hat, heißt ein **gerichteter azyklischer Graph** („**directed acyclic graph**" – **dag**).

Bei einem azyklischen Digraphen kann man die Knoten in eine Reihenfolge bringen, so dass die Kanten nur „von links nach rechts" laufen.

**Definition** Ein gerichteter Graph (Digraph), der keine Kreise hat, heißt ein **gerichteter azyklischer Graph** („**directed acyclic graph**“ – **dag**).

Bei einem azyklischen Digraphen kann man die Knoten in eine Reihenfolge bringen, so dass die Kanten nur „von links nach rechts“ laufen.

**Definition** Sei $G = (V, E)$ ein azyklischer Digraph.

**Definition** Ein gerichteter Graph (Digraph), der keine Kreise hat, heißt ein **gerichteter azyklischer Graph** („**directed acyclic graph**" – **dag**).

Bei einem azyklischen Digraphen kann man die Knoten in eine Reihenfolge bringen, so dass die Kanten nur „von links nach rechts" laufen.

**Definition** Sei $G = (V, E)$ ein azyklischer Digraph.

Eine **topologische Sortierung** (oder topologische Anordnung) der Knoten in $V$ ist eine Bijektion $\pi\colon V \to \{1, \ldots, n\}$, so dass $(v, w) \in E \Rightarrow \pi(v) < \pi(w)$ gilt.

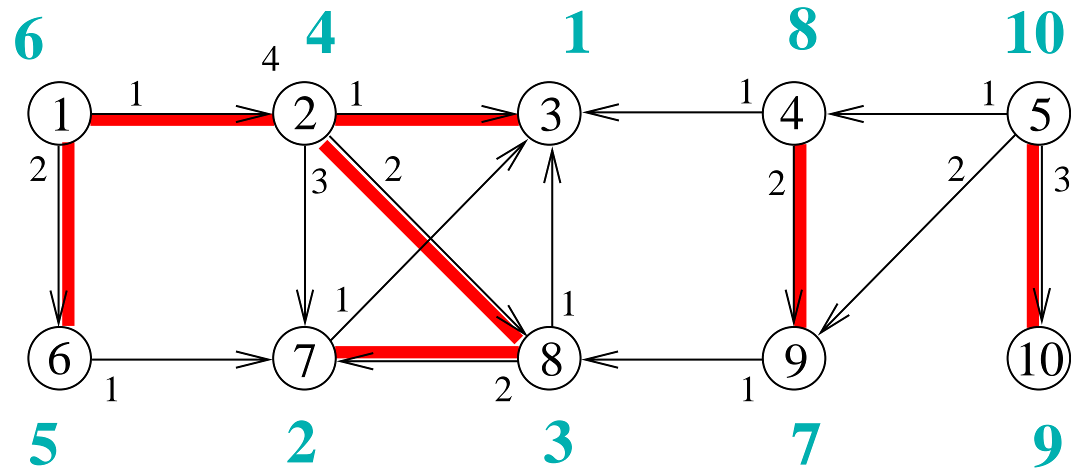
**Definition** Ein gerichteter Graph (Digraph), der keine Kreise hat, heißt ein **gerichteter azyklischer Graph** („**directed acyclic graph**" – **dag**).

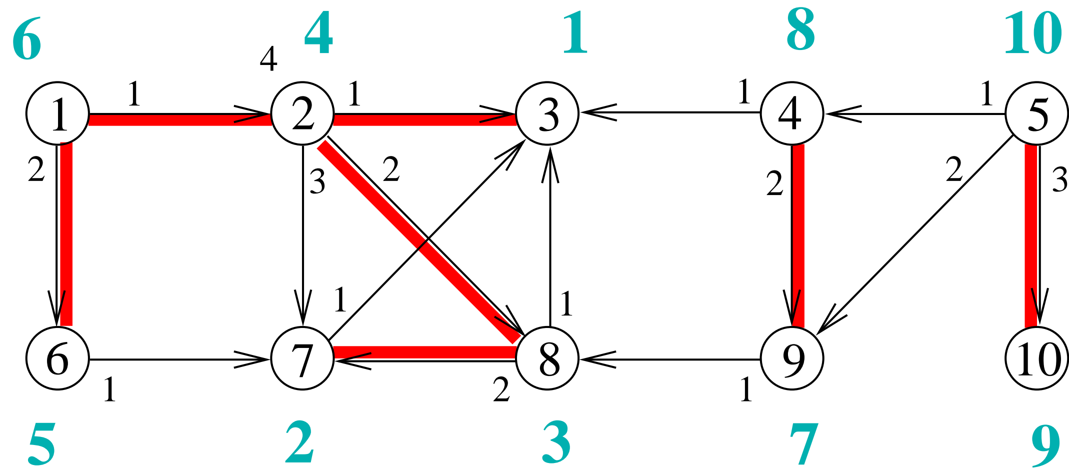
Bei einem azyklischen Digraphen kann man die Knoten in eine Reihenfolge bringen, so dass die Kanten nur „von links nach rechts" laufen.

**Definition** Sei $G = (V, E)$ ein azyklischer Digraph.

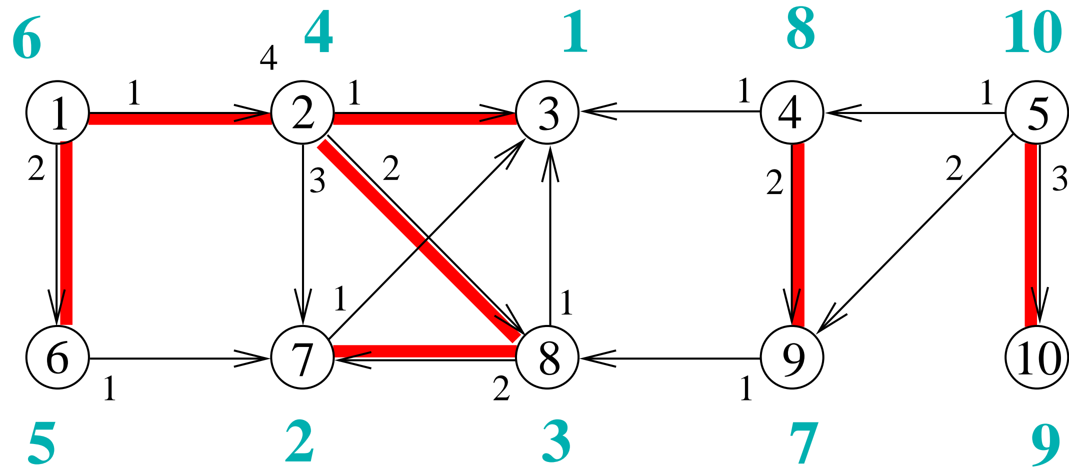
Eine **topologische Sortierung** (oder topologische Anordnung) der Knoten in $V$ ist eine Bijektion $\pi \colon V \to \{1, \ldots, n\}$, so dass $(v, w) \in E \Rightarrow \pi(v) < \pi(w)$ gilt.

Man stellt sich die Knoten in der Reihenfolge $\pi^{-1}(1), \pi^{-1}(2), \ldots, \pi^{-1}(n)$ vor; dann laufen die Kanten alle von links nach rechts.

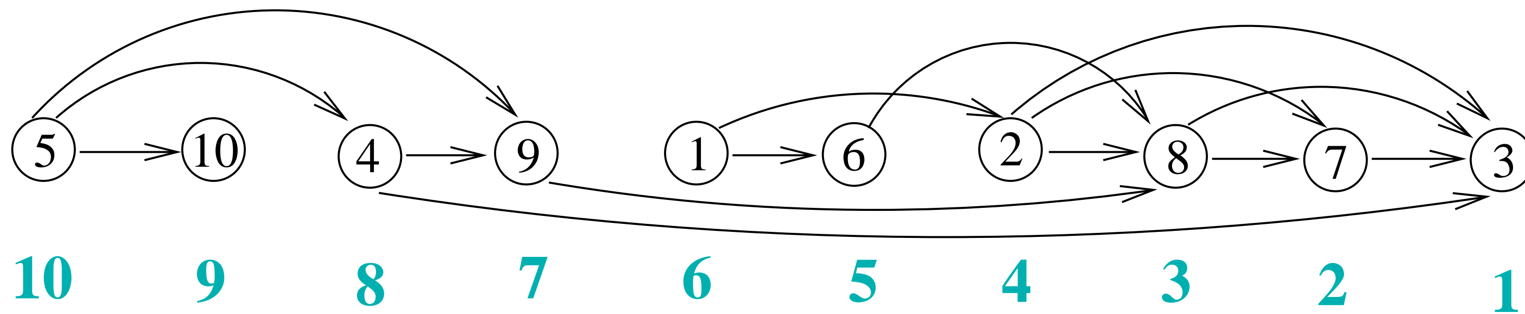

6
4
1
8
10
4
1
1
1
1
1
2
3
4
5
2
3
2
2
2
3
1
1
6
7
8
9
10
1
2
1
5
2
3
7
9

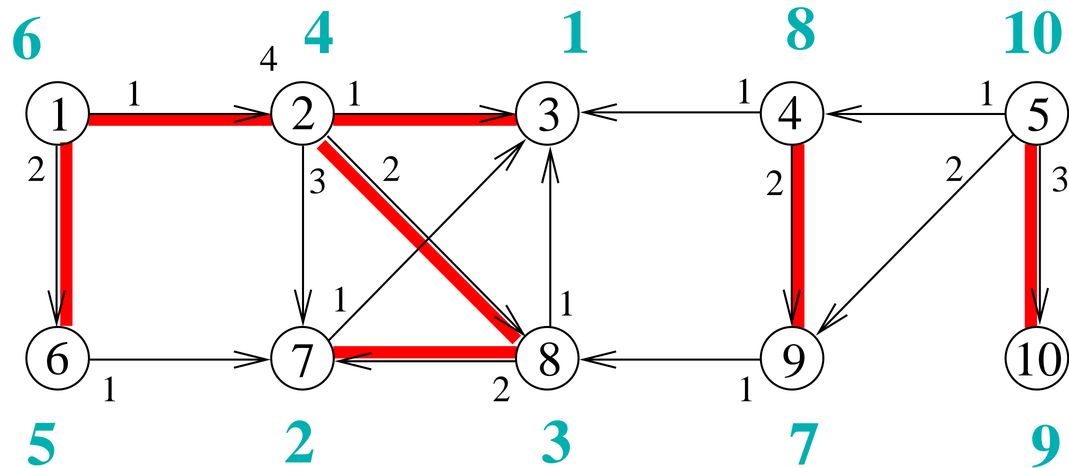



Ein **azyklischer** Digraph mit f-Nummern.

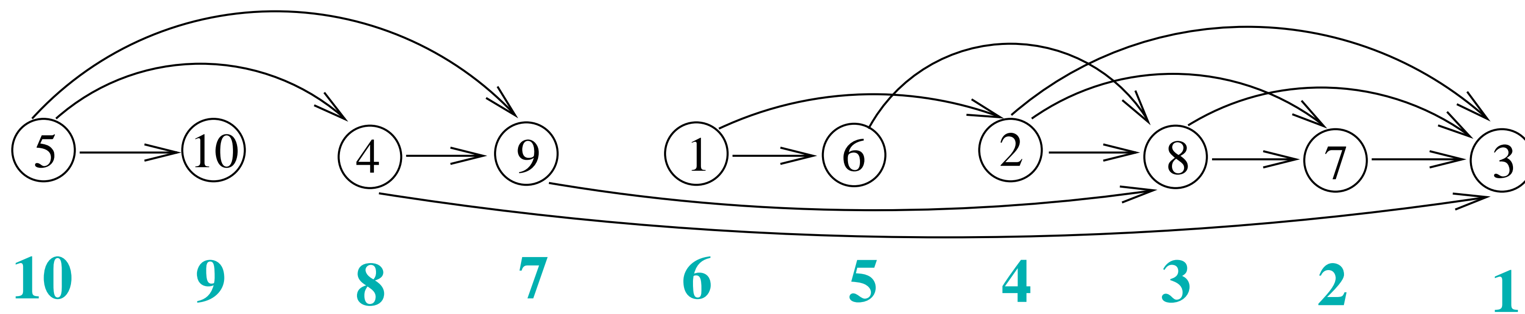



Ein **azyklischer** Digraph mit f-Nummern.






Ein **azyklischer** Digraph mit f-Nummern.




Derselbe Graph, linear angeordnet.

## **DFS** findet unmittelbar eine topologische Sortierung!

**DFS** findet unmittelbar eine topologische Sortierung!

(Natürlich nur in **azyklischen** Digraphen.)

**DFS** findet unmittelbar eine topologische Sortierung!

(Natürlich nur in **azyklischen** Digraphen.)

**Idee:** Wir haben oben gesehen, dass die f-Nummern entlang jeder Kante $(v, w)$ **strikt fallen**; außerdem sind die f-Nummern eine **Bijektion**.

**DFS** findet unmittelbar eine topologische Sortierung!

(Natürlich nur in **azyklischen** Digraphen.)

**Idee:** Wir haben oben gesehen, dass die f-Nummern entlang jeder Kante $(v, w)$ **strikt fallen**; außerdem sind die f-Nummern eine **Bijektion**.

Definiere:

$\pi(v) := (n + 1) - \mathit{f_num}(v)$.

**DFS** findet unmittelbar eine topologische Sortierung!

(Natürlich nur in **azyklischen** Digraphen.)

**Idee:** Wir haben oben gesehen, dass die f-Nummern entlang jeder Kante $(v, w)$ **strikt fallen**; außerdem sind die f-Nummern eine **Bijektion**.

Definiere:

$\pi(v) := (n + 1) - \textit{f_num}(v)$.

Laufzeit/Kosten zur Ermittlung einer topologischen Sortierung: $O(|V| + |E|)$, **linear**.

**DFS** findet unmittelbar eine topologische Sortierung!

(Natürlich nur in **azyklischen** Digraphen.)

**Idee:** Wir haben oben gesehen, dass die f-Nummern entlang jeder Kante $(v, w)$ **strikt fallen**; außerdem sind die f-Nummern eine **Bijektion**.

Definiere:

$\pi(v) := (n + 1) - \textit{f_num}(v)$.

Laufzeit/Kosten zur Ermittlung einer topologischen Sortierung: $O(|V| + |E|)$, **linear**.

## Satz

DFS($G$) (in der ausführlichen Version) testet Digraphen auf Azyklizität und findet gegebenenfalls eine topologische Sortierung von $G$, in Zeit $O(|V| + |E|)$.

# Starke Zusammenhangskomponenten in Digraphen

# Starke Zusammenhangskomponenten in Digraphen

## Definition

# Starke Zusammenhangskomponenten in Digraphen

**Definition** Zwei Knoten $v$ und $w$ in einem Digraphen $G$ heißen **äquivalent**, wenn $v \leadsto w$ und $w \leadsto v$ gilt. Notation: $v \leftrightsquigarrow w$.

# Starke Zusammenhangskomponenten in Digraphen

**Definition** Zwei Knoten $v$ und $w$ in einem Digraphen $G$ heißen **äquivalent**, wenn $v \leadsto w$ und $w \leadsto v$ gilt. Notation: $v \leftrightsquigarrow w$. – Überlege:

(a) Die so definierte Relation ist reflexiv, transitiv, und symmetrisch, also eine **Äquivalenzrelation**.

(b) Zwei Knoten $v$ und $w$ sind äquivalent genau dann wenn sie identisch sind **oder** es in $G$ einen (gerichteten) Kreis gibt, auf dem beide Knoten liegen.

**Definition**

# Starke Zusammenhangskomponenten in Digraphen

**Definition** Zwei Knoten $v$ und $w$ in einem Digraphen $G$ heißen **äquivalent**, wenn $v \leadsto w$ und $w \leadsto v$ gilt. Notation: $v \leftrightsquigarrow w$. – Überlege:

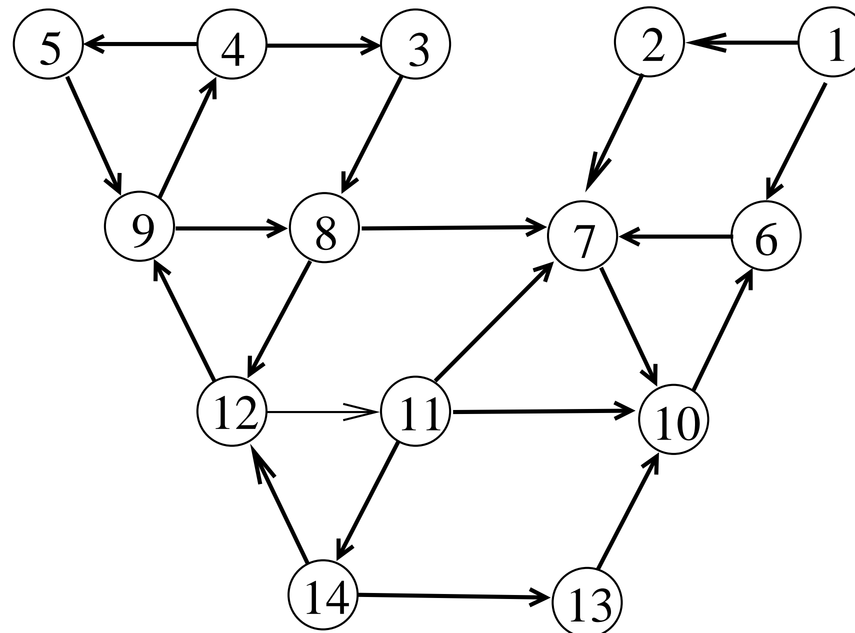
(a) Die so definierte Relation ist reflexiv, transitiv, und symmetrisch, also eine **Äquivalenzrelation**.

(b) Zwei Knoten $v$ und $w$ sind äquivalent genau dann wenn sie identisch sind **oder** es in $G$ einen (gerichteten) Kreis gibt, auf dem beide Knoten liegen.

**Definition** Die Äquivalenzklassen zur Relation $\leftrightsquigarrow$ heißen **starke Zusammenhangskomponenten** von $G$.

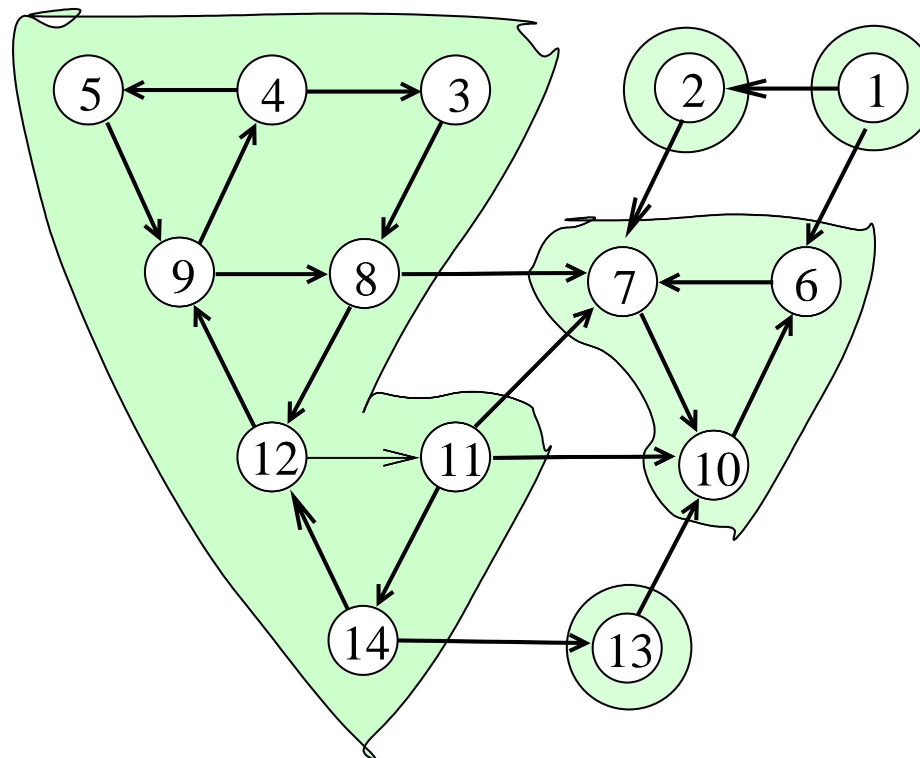
# Starke Zusammenhangskomponenten in Digraphen

*Beispiel*:



# Starke Zusammenhangskomponenten in Digraphen

*Beispiel*:



## Lemma

Bei der Ausführung von **DFS**$(G)$ gilt: Wenn $C \subseteq V$ eine starke Zusammenhangskomponente ist, dann liegen die Knoten von $C$ alle in einem Tiefensuchbaum.

## Lemma

Bei der Ausführung von **DFS**$(G)$ gilt: Wenn $C \subseteq V$ eine starke Zusammenhangskomponente ist, dann liegen die Knoten von $C$ alle in einem Tiefensuchbaum.

M.a.W.: Jeder Tiefensuchbaum ist eine starke Zsh.-Komponente oder lässt sich in mehrere starke Zsh.-Komponenten zerlegen.

## Lemma

Bei der Ausführung von **DFS**$(G)$ gilt: Wenn $C \subseteq V$ eine starke Zusammenhangskomponente ist, dann liegen die Knoten von $C$ alle in einem Tiefensuchbaum.

M.a.W.: Jeder Tiefensuchbaum ist eine starke Zsh.-Komponente oder lässt sich in mehrere starke Zsh.-Komponenten zerlegen.

*Beweis*: Betrachte den ersten Knoten $v \in C$, für den **dfs**$(v)$ aufgerufen wird.

## Lemma

Bei der Ausführung von **DFS**$(G)$ gilt: Wenn $C \subseteq V$ eine starke Zusammenhangskomponente ist, dann liegen die Knoten von $C$ alle in einem Tiefensuchbaum.

M.a.W.: Jeder Tiefensuchbaum ist eine starke Zsh.-Komponente oder lässt sich in mehrere starke Zsh.-Komponenten zerlegen.

*Beweis*: Betrachte den ersten Knoten $v \in C$, für den **dfs**$(v)$ aufgerufen wird.

Von $v$ aus sind alle $w \in C$ erreichbar (Definition s.Z.K.), und alle diese $w$ sind noch „neu“.

## Lemma

Bei der Ausführung von **DFS**$(G)$ gilt: Wenn $C \subseteq V$ eine starke Zusammenhangskomponente ist, dann liegen die Knoten von $C$ alle in einem Tiefensuchbaum.

M.a.W.: Jeder Tiefensuchbaum ist eine starke Zsh.-Komponente oder lässt sich in mehrere starke Zsh.-Komponenten zerlegen.

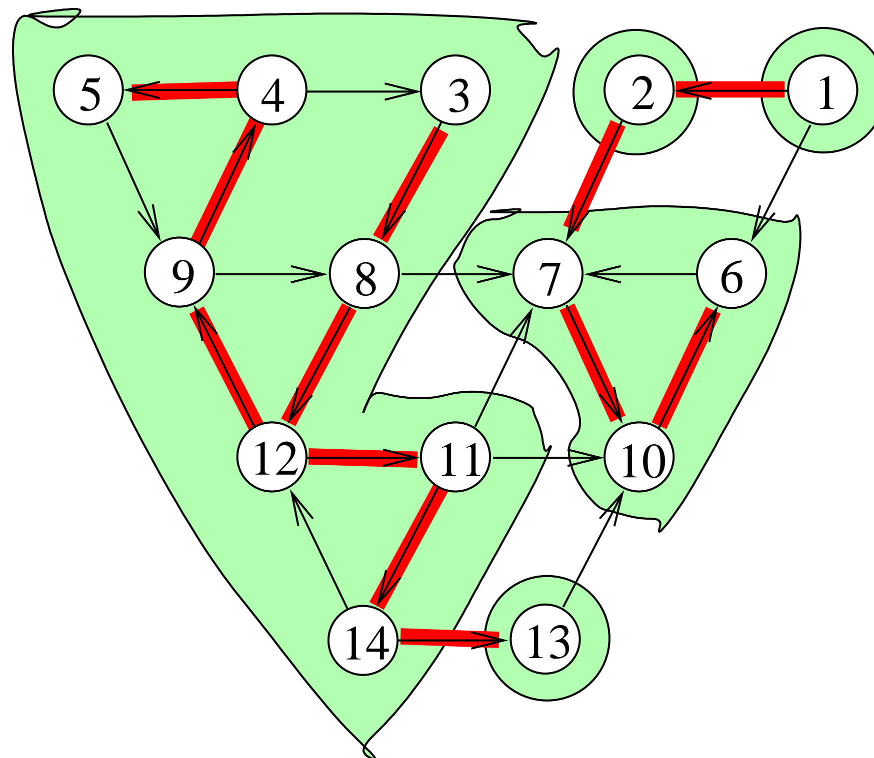
*Beweis*: Betrachte den ersten Knoten $v \in C$, für den **dfs**$(v)$ aufgerufen wird.

Von $v$ aus sind alle $w \in C$ erreichbar (Definition s.Z.K.), und alle diese $w$ sind noch „neu".

$\Rightarrow$ (nach dem „Satz vom weißen Weg") Alle Knoten von $C$ werden Nachfahren von $v$ im Tiefensuchbaum.

# Starke Zusammenhangskomponenten in Digraphen

*Beispiel*: 5 Komponenten, 2 Tiefensuchbäume.



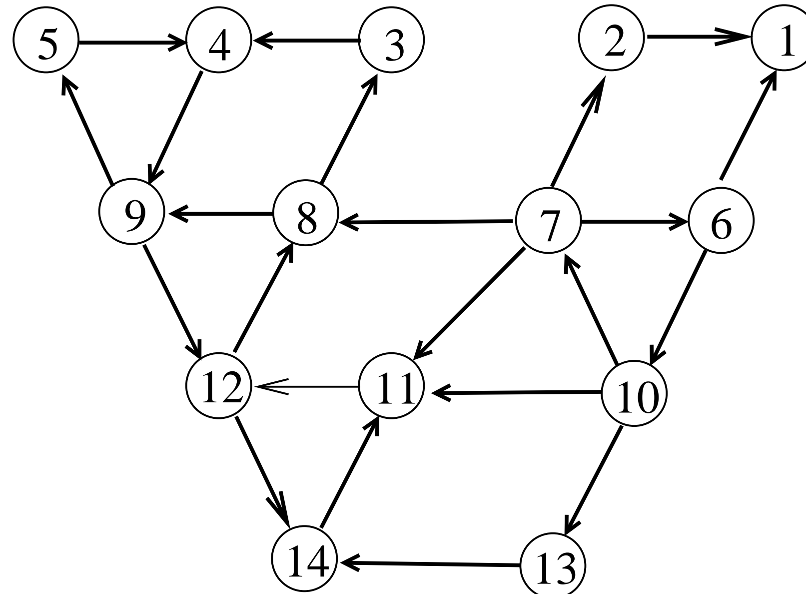
Um mit Tiefensuche die starken Zusammenhangskomponenten zu finden, muss man verhindern, dass in einem Baum mehrere solche Komponenten sitzen.

Um mit Tiefensuche die starken Zusammenhangskomponenten zu finden, muss man verhindern, dass in einem Baum mehrere solche Komponenten sitzen.

**Trick:** Benutze „Umkehrgraphen“ $G^R$, der durch Umkehren aller Kanten in $G$ entsteht.

Um mit Tiefensuche die starken Zusammenhangskomponenten zu finden, muss man verhindern, dass in einem Baum mehrere solche Komponenten sitzen.

**Trick:** Benutze „Umkehrgraphen“ $G^R$, der durch Umkehren aller Kanten in $G$ entsteht.



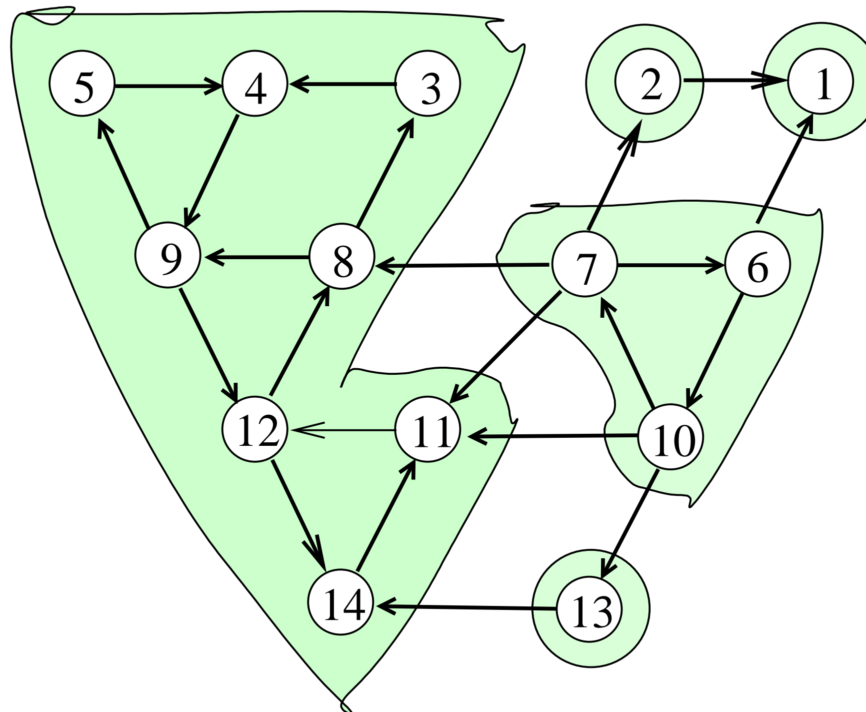
**Beobachtung:** $G^R$ hat dieselben starken Zusammenhangs-komponenten wie $G$.

**Beobachtung:** $G^R$ hat dieselben starken Zusammenhangskomponenten wie $G$.

(Die Kreise in den beiden Graphen sind dieselben, nur mit umgekehrter Durchlaufreihenfolge.)

**Beobachtung:** $G^R$ hat dieselben starken Zusammenhangskomponenten wie $G$.

(Die Kreise in den beiden Graphen sind dieselben, nur mit umgekehrter Durchlaufreihenfolge.)



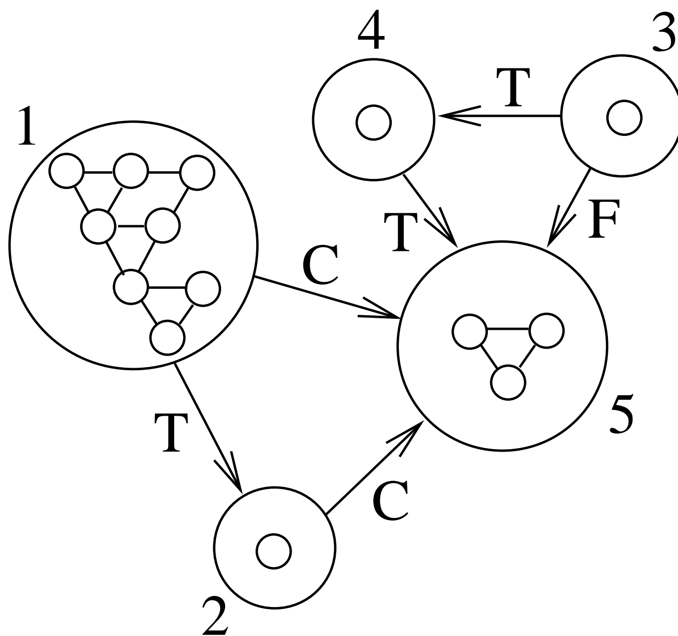
Eine günstige Wahl der Reihenfolge der Startpunkte für die Tiefensuche in $G^R$ verhindert, dass „versehentlich“ Knoten einer anderen starken Zusammenhangskomponente erreicht werden.

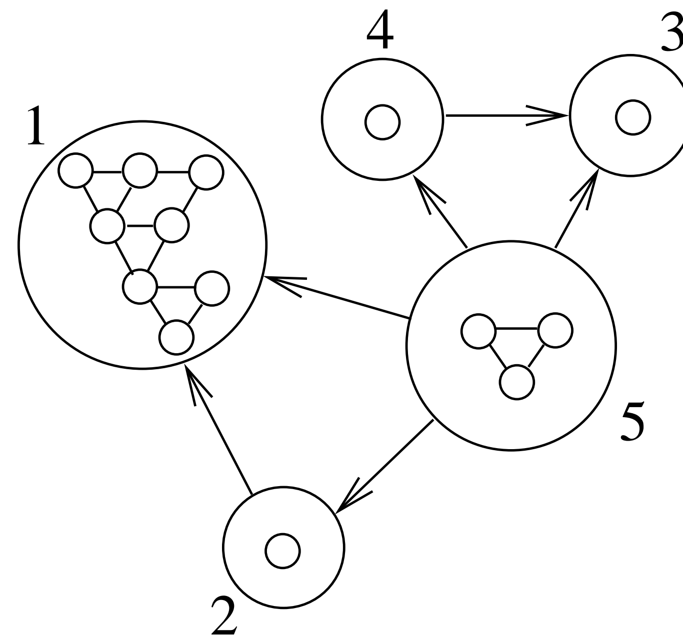
Ziehe (konzeptuell) starke Zusammenhangskomponenten auf einen Knoten zusammen.

Ziehe (konzeptuell) starke Zusammenhangskomponenten auf einen Knoten zusammen.

Kontrahierte Knoten, topologische Anordnung:

In $G$: In $G^R$:



Der kontrahierte Graph hat nur T-, C- und F-Kanten, ist also kreisfrei und hat daher eine topologische Sortierung. Dies gilt für die $G$- **und** die $G^R$-Version.

Arbeite den kontrahierten $G^R$-Graphen
(der kreisfrei ist, wie man leicht sieht), in der Reihenfolge einer **umgekehrten** topologischen Sortierung der kontrahierten Version von $G^R$ ab.

Dann kann keine Kante zu einer noch nicht abgearbeiteten starken Zusammenhangskomponente führen.

Der kontrahierte Graph hat nur T-, C- und F-Kanten, ist also kreisfrei und hat daher eine topologische Sortierung. Dies gilt für die $G$- **und** die $G^R$-Version.

Arbeite den kontrahierten $G^R$-Graphen
(der kreisfrei ist, wie man leicht sieht), in der Reihenfolge einer **umgekehrten** topologischen Sortierung der kontrahierten Version von $G^R$ ab.

Dann kann keine Kante zu einer noch nicht abgearbeiteten starken Zusammenhangskomponente führen.

Woher bekommt man umgekehrte topologische Sortierung der kontrahierten Version von $G^R$?

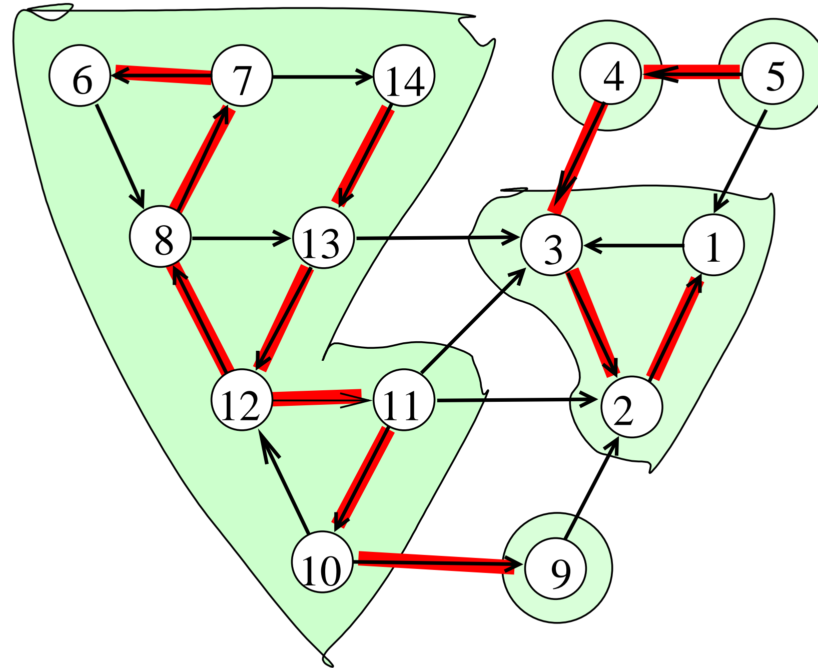
Der kontrahierte Graph hat nur T-, C- und F-Kanten, ist also kreisfrei und hat daher eine topologische Sortierung. Dies gilt für die $G$- **und** die $G^R$-Version.

Arbeite den kontrahierten $G^R$-Graphen
(der kreisfrei ist, wie man leicht sieht), in der Reihenfolge einer **umgekehrten** topologischen Sortierung der kontrahierten Version von $G^R$ ab.

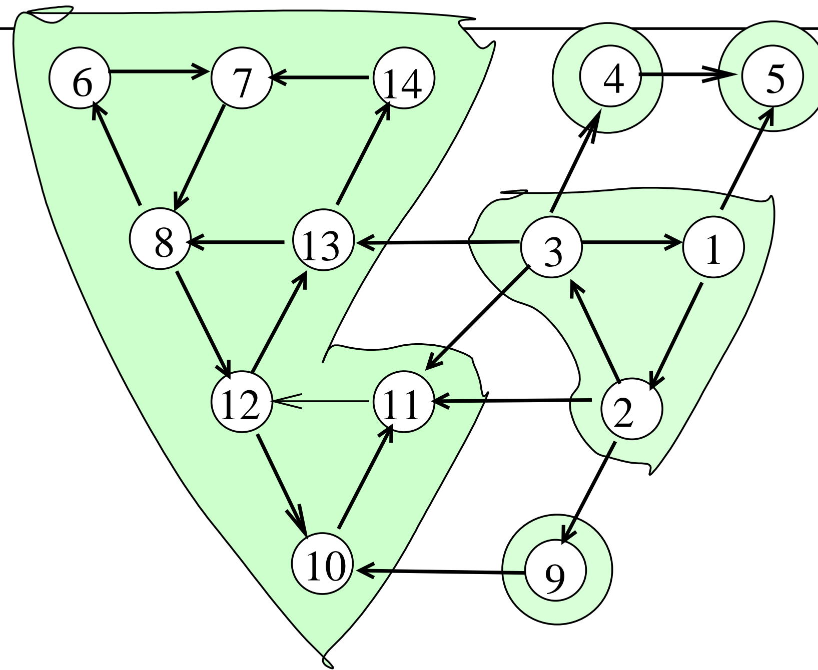
Dann kann keine Kante zu einer noch nicht abgearbeiteten starken Zusammenhangskomponente führen.

Woher bekommt man umgekehrte topologische Sortierung der kontrahierten Version von $G^R$?

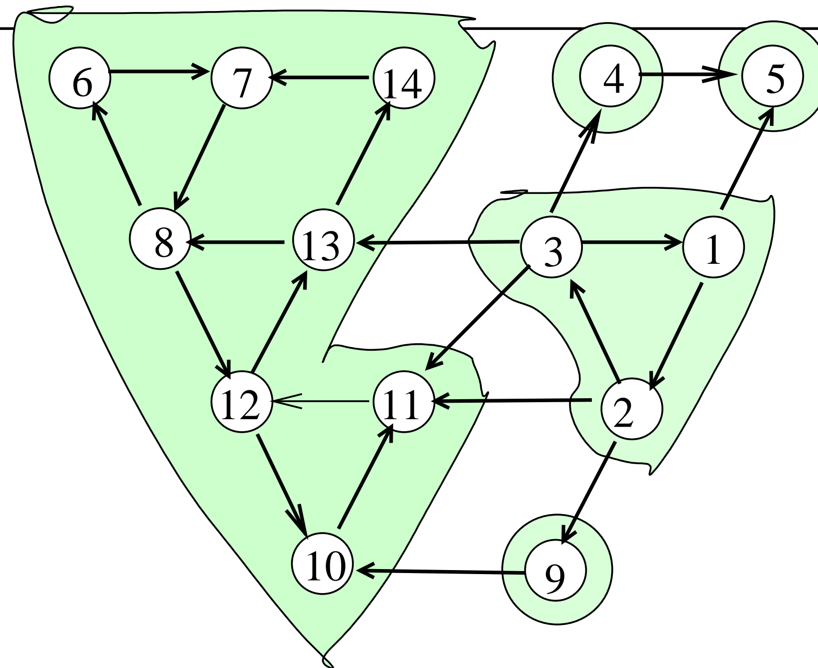
Verwende topologische Sortierung von $G$!




Graph $G$ mit DFS-Wald und f-Nummern.




Graph $G^R$ mit f-Nummern der $G$-Tiefensuche.




Graph $G^R$ mit f-Nummern der $G$-Tiefensuche.

Komponenten:

$\{6, 7, \mathbf{14}, 8, 13, 12, 11, 10\}$, $\{\mathbf{9}\}$,
$\{\mathbf{5}\}$, $\{\mathbf{4}\}$, $\{\mathbf{3}, 1, 2\}$

Graph $G^R$ mit f-Nummern der $G$-Tiefensuche.

Komponenten:

$\{6, 7, \mathbf{14}, 8, 13, 12, 11, 10\}$, $\{\mathbf{9}\}$,
$\{\mathbf{5}\}$, $\{\mathbf{4}\}$, $\{\mathbf{3}, 1, 2\}$

Die **größten** f-Nummern in den Komponenten, fallend gelesen, sortieren diese umgekehrt topologisch in $G^R$.

# Algorithmus Strong Components$(G)$

**Algorithmus Strong Components**$(G)$

**Eingabe:** Digraph $G = (V, E)$

**Ausgabe:** starke Zusammenhangskomponenten von $G$

(1) Berechne die f-Nummern mittels $\texttt{DFS}(G)$;

(2) Bilde „Umkehrgraphen“ $G^R$ durch Umkehren aller Kanten in $G$;

(3) Führe $\texttt{DFS}(G^R)$ durch;
    Knotenreihenfolge: f-Nummern aus (1) absteigend
    (topologische Sortierung)

(4) Ausgabe: Die Knotenmengen der Tiefensuchbäume aus (3).

**Algorithmus Strong Components**$(G)$

**Eingabe:** Digraph $G = (V, E)$

**Ausgabe:** starke Zusammenhangskomponenten von $G$

(1) Berechne die f-Nummern mittels $\texttt{DFS}(G)$;

(2) Bilde „Umkehrgraphen" $G^R$ durch Umkehren aller Kanten in $G$;

(3) Führe $\texttt{DFS}(G^R)$ durch;
    Knotenreihenfolge: f-Nummern aus (1) absteigend
    (topologische Sortierung)

(4) Ausgabe: Die Knotenmengen der Tiefensuchbäume aus (3).

## Satz

(a) Der Algorithmus **Strong Components** gibt die starken Zusammenhangskomponenten von $G$ aus.

(b) Die Laufzeit ist $O(|V| + |E|)$.

*Beweis* von Teil (a):

Wir wissen schon, dass jede starke Zsh.-Komponente $C$ komplett in einem DFS-Baum der Tiefensuche in $G^R$ enthalten ist.

Zu zeigen: Kein DFS-Baum der $G^R$-Tiefensuche enthält mehrere Komponenten.

Es sei $C \subseteq V$ eine beliebige starke Zusammenhangskomponente von $G$ bzw. von $G^R$.

$v \in C$ sei **der Knoten mit maximaler f-Nummer** (aus der $G$-Tiefensuche).

**Behauptung:** In $G^R$ ist $v$ von keinem Knoten mit größerer f-Nummer aus erreichbar.

[Dann folgt aus dem Struktursatz für die Knotenmengen der DFS-Bäume und der besonderen Durchlaufreihenfolge – fallende f-Nummern – , dass im zweiten DFS-Durchlauf $v$ Wurzel eines DFS-Baums wird. Da $C$ beliebig war, enthält also jede starke Zusammenhangskomponente eine DFS-Baum-Wurzel, also müssen diese Komponenten jede für sich einen Baum bilden.]

**Behauptung:** In $G^R$ ist $v$ von keinem Knoten $w$ mit größerer f-Nummer aus erreichbar.

**Äquivalent:** Wenn es in $G^R$ einen Weg von $w$ nach $v$ gibt, dann gilt *f-num*$(w) \leq$ *f-num*$(v)$.

*Beweis* dieser Formulierung:
Sei $w = v_t, v_{t-1}, v_{t-2}, \ldots, v_1, v_0 = v$ ein Weg in $G^R$.
O.B.d.A.: $v \neq w$.

Das heißt: $\underbrace{v = v_0, v_1, \ldots, v_{t-1}, v_t = w}_{=p}$ ist ein Weg in $G$.

Wir betrachten nun die (erste) Tiefensuche in $G$.

**1. Fall:** Wenn dfs$(v)$ startet, ist der gesamte Weg $p$ weiß.
Dann wird $w$ Nachfahr von $v$ im DFS-Baum, also ist *f-num*$(w)$ $<$ *f-num*$(v)$.

**2. Fall:** Wenn dfs($v$) startet, liegt auf dem Weg $p$ ein roter Knoten $v_i$.

Dann ist $v_i$ Vorfahr von $v$ im DFS-Baum, und von $v$ zu $v_i$ führt der erste Teil des Weges $p$: also liegen $v$ und $v_i$ auf einem Kreis. Also ist $v \leftrightsquigarrow v_i$, also $v_i \in C$.

Andererseits ist *f-num*$(v_i) >$ *f-num*$(v)$, im **Widerspruch** zur Wahl von $v$. Der 2. Fall kann also gar nicht eintreten.

**3. Fall:** Wenn dfs($v$) startet, liegt auf dem Weg $p$ ein grauer Knoten $v_i \neq v$.

Nach der dfs-Prozedur und Fall 2 gilt: Wenn $v_i$ grau ist und $i < t$, dann ist auch $v_{i+1}$ grau. Also ist $w$ grau, wenn dfs($v$) startet. Daraus: *f-num*$(w) <$ *f-num*$(v)$. $\square$

# Tiefensuche in ungerichteten Graphen

# Tiefensuche in ungerichteten Graphen

Einfacher als Tiefensuche in gerichteten Graphen.

# Tiefensuche in ungerichteten Graphen

Einfacher als Tiefensuche in gerichteten Graphen.

**Zweck:**

- Zusammenhangskomponenten finden

# Tiefensuche in ungerichteten Graphen

Einfacher als Tiefensuche in gerichteten Graphen.

**Zweck:**

- Zusammenhangskomponenten finden
- Spannbaum für jede Zusammenhangskomponente finden

# Tiefensuche in ungerichteten Graphen

Einfacher als Tiefensuche in gerichteten Graphen.

**Zweck:**

- Zusammenhangskomponenten finden
- Spannbaum für jede Zusammenhangskomponente finden
- Kreisfreiheitstest

# Tiefensuche in ungerichteten Graphen

Einfacher als Tiefensuche in gerichteten Graphen.

**Zweck:**

- Zusammenhangskomponenten finden
- Spannbaum für jede Zusammenhangskomponente finden
- Kreisfreiheitstest

(Diese einfachen Aufgaben wären auch auch von Breitensuche zu erledigen.)

# Tiefensuche in ungerichteten Graphen

Einfacher als Tiefensuche in gerichteten Graphen.

**Zweck:**

- Zusammenhangskomponenten finden
- Spannbaum für jede Zusammenhangskomponente finden
- Kreisfreiheitstest

(Diese einfachen Aufgaben wären auch auch von Breitensuche zu erledigen.)

**Komplexere Erweiterungen**, z. B. „Zweifach-Zusammenhangs-komponenten", **erfordern** DFS.

Darstellung von ungerichteten Graphen:

**Adjazenzlistendarstellung** mit Querverweisen.

Zu Knoten $v$ gibt es eine Liste $L_v$, die für jede Kante $(v, w)$ einen Eintrag hat.

Von Eintrag für Kante $(v, w)$ führt ein Verweis auf die „Gegenkante“ $(w, v)$ in der Adjazenzliste $L_w$ (und natürlich auch umgekehrt).

Darstellung von ungerichteten Graphen:

**Adjazenzlistendarstellung** mit Querverweisen.

Zu Knoten $v$ gibt es eine Liste $L_v$, die für jede Kante $(v, w)$ einen Eintrag hat.

Von Eintrag für Kante $(v, w)$ führt ein Verweis auf die „Gegenkante“ $(w, v)$ in der Adjazenzliste $L_w$ (und natürlich auch umgekehrt).

Wir können an Kanten Markierungen anbringen.

Wir stellen sicher, dass nach Benutzung der Kante $(v, w)$ in Richtung von $v$ nach $w$ die umgekehrte Richtung nicht mehr benutzt wird **(„gesehen“)**.

Darstellung von ungerichteten Graphen:

**Adjazenzlistendarstellung** mit Querverweisen.

Zu Knoten $v$ gibt es eine Liste $L_v$, die für jede Kante $(v, w)$ einen Eintrag hat.

Von Eintrag für Kante $(v, w)$ führt ein Verweis auf die „Gegenkante“ $(w, v)$ in der Adjazenzliste $L_w$ (und natürlich auch umgekehrt).

Wir können an Kanten Markierungen anbringen.

Wir stellen sicher, dass nach Benutzung der Kante $(v, w)$ in Richtung von $v$ nach $w$ die umgekehrte Richtung nicht mehr benutzt wird **(„gesehen“)**.

**Klassifizierung** in $T$-Kanten und $B$-Kanten.

## **Algorithmus udfs**($v$) ($*$ Tiefensuche von $v$ aus, rekursiv $*$)

**Algorithmus udfs**$(v)$ (∗ Tiefensuche von $v$ aus, rekursiv ∗)
(∗ nur für $v$ mit $\texttt{status}[v]$ = *neu* aufrufen ∗)

**Algorithmus udfs**$(v)$ (∗ Tiefensuche von $v$ aus, rekursiv ∗)

(∗ nur für $v$ mit $\texttt{status}[v] = $ *neu* aufrufen ∗)

(1) `dfs_count++`;

(2) $\texttt{dfs_num}[v] \leftarrow \texttt{dfs_count}$;

**Algorithmus udfs**$(v)$ (* Tiefensuche von $v$ aus, rekursiv *)

(* nur für $v$ mit $\texttt{status}[v] = \textit{neu}$ aufrufen *)

(1) $\texttt{dfs_count++}$;

(2) $\texttt{dfs_num}[v] \leftarrow \texttt{dfs_count}$;

(3) **dfs-visit**$(v)$; (* Aktion an $v$ bei Erstbesuch *)

**Algorithmus udfs**$(v)$ (∗ Tiefensuche von $v$ aus, rekursiv ∗)
(∗ nur für $v$ mit $\texttt{status}[v] =$ *neu* aufrufen ∗)

(1) `dfs_count++`;

(2) $\texttt{dfs_num}[v] \leftarrow \texttt{dfs_count}$;

(3) **dfs-visit**$(v)$; (∗ Aktion an $v$ bei Erstbesuch ∗)

(4) $\texttt{status}[v] \leftarrow$ *aktiv* ;

**Algorithmus udfs**$(v)$ (* Tiefensuche von $v$ aus, rekursiv *)

(* nur für $v$ mit `status[`$v$`]` $=$ *neu* aufrufen *)

(1) `dfs_count++`;

(2) `dfs_num[`$v$`]` $\leftarrow$ `dfs_count`;

(3) **dfs-visit**$(v)$; (* Aktion an $v$ bei Erstbesuch *)

(4) `status[`$v$`]` $\leftarrow$ *aktiv* ;

(5) für jede **„neue“** Kante $(v, w)$ (Adjazenzliste!) tue:

**Algorithmus udfs**$(v)$ (∗ Tiefensuche von $v$ aus, rekursiv ∗)

(∗ nur für $v$ mit `status[`$v$`]` $=$ *neu* aufrufen ∗)

(1) `dfs_count++`;

(2) `dfs_num[`$v$`]` $\leftarrow$ `dfs_count`;

(3) **dfs-visit**$(v)$; (∗ Aktion an $v$ bei Erstbesuch ∗)

(4) `status[`$v$`]` $\leftarrow$ *aktiv* ;

(5) für jede **„neue“** Kante $(v, w)$ (Adjazenzliste!) tue:

(6) setze Gegenkante $(w, v)$ auf **„gesehen“**;

(7) **1. Fall:** `status[w]` $=$ *neu*: $T \leftarrow T \cup \{(v, w)\}$; **udfs**(`w`);

**Algorithmus udfs**$(v)$ (∗ Tiefensuche von $v$ aus, rekursiv ∗)

(∗ nur für $v$ mit `status[`$v$`]` $=$ *neu* aufrufen ∗)

(1) `dfs_count++`;

(2) `dfs_num[`$v$`]` $\leftarrow$ `dfs_count`;

(3) **dfs-visit**$(v)$; (∗ Aktion an $v$ bei Erstbesuch ∗)

(4) `status[`$v$`]` $\leftarrow$ *aktiv* ;

(5) für jede **„neue“** Kante $(v, w)$ (Adjazenzliste!) tue:

(6) setze Gegenkante $(w, v)$ auf **„gesehen“**;

(7) **1. Fall:** `status[w]` $=$ *neu*: $T \leftarrow T \cup \{(v, w)\}$; **udfs**(`w`);

(8) **2. Fall:** `status[w]` $=$ *aktiv*: $B \leftarrow B \cup \{(v, w)\}$;

**Algorithmus udfs**$(v)$ (∗ Tiefensuche von $v$ aus, rekursiv ∗)

(∗ nur für $v$ mit `status[`$v$`]` $=$ *neu* aufrufen ∗)

(1) `dfs_count++`;

(2) `dfs_num[`$v$`]` $\leftarrow$ `dfs_count`;

(3) **dfs-visit**$(v)$; (∗ Aktion an $v$ bei Erstbesuch ∗)

(4) `status[`$v$`]` $\leftarrow$ *aktiv* ;

(5) für jede **„neue“** Kante $(v, w)$ (Adjazenzliste!) tue:

(6) setze Gegenkante $(w, v)$ auf **„gesehen“**;

(7) **1. Fall:** `status[w]` $=$ *neu*: $T \leftarrow T \cup \{(v, w)\}$; **udfs**(`w`);

(8) **2. Fall:** `status[w]` $=$ *aktiv*: $B \leftarrow B \cup \{(v, w)\}$;

(9) `f_count++`;

**Algorithmus udfs**$(v)$ (* Tiefensuche von $v$ aus, rekursiv *)
(* nur für $v$ mit status[$v$] = *neu* aufrufen *)

(1) dfs_count++;
(2) dfs_num[$v$] $\leftarrow$ dfs_count;
(3) **dfs-visit**$(v)$; (* Aktion an $v$ bei Erstbesuch *)
(4) status[$v$] $\leftarrow$ *aktiv* ;
(5) für jede **„neue“** Kante $(v, w)$ (Adjazenzliste!) tue:
(6) setze Gegenkante $(w, v)$ auf **„gesehen“**;
(7) **1. Fall:** status[w] = *neu*: $T \leftarrow T \cup \{(v, w)\}$; **udfs**(w);
(8) **2. Fall:** status[w] = *aktiv*: $B \leftarrow B \cup \{(v, w)\}$;
(9) f_count++;
(10) f_num[$v$] $\leftarrow$ f_count;

**Algorithmus udfs**$(v)$ (∗ Tiefensuche von $v$ aus, rekursiv ∗)

(∗ nur für $v$ mit `status`$[v]$ = *neu* aufrufen ∗)

(1) `dfs_count++`;

(2) `dfs_num`$[v] \leftarrow$ `dfs_count`;

(3) **dfs-visit**$(v)$; (∗ Aktion an $v$ bei Erstbesuch ∗)

(4) `status`$[v] \leftarrow$ *aktiv* ;

(5) für jede **„neue“** Kante $(v, w)$ (Adjazenzliste!) tue:

(6) setze Gegenkante $(w, v)$ auf **„gesehen“**;

(7) **1. Fall:** `status[w]` = *neu*: $T \leftarrow T \cup \{(v, w)\}$; **udfs**(`w`);

(8) **2. Fall:** `status[w]` = *aktiv*: $B \leftarrow B \cup \{(v, w)\}$;

(9) `f_count++`;

(10) `f_num`$[v] \leftarrow$ `f_count`;

(11) **fin-visit**$(v)$; (∗ Aktion an $v$ bei letztem Besuch ∗)

**Algorithmus udfs**$(v)$ (∗ Tiefensuche von $v$ aus, rekursiv ∗)

(∗ nur für $v$ mit `status[`$v$`]` $=$ *neu* aufrufen ∗)

(1) `dfs_count++`;

(2) `dfs_num[`$v$`]` $\leftarrow$ `dfs_count`;

(3) **dfs-visit**$(v)$; (∗ Aktion an $v$ bei Erstbesuch ∗)

(4) `status[`$v$`]` $\leftarrow$ *aktiv* ;

(5) für jede **„neue“** Kante $(v, w)$ (Adjazenzliste!) tue:

(6) setze Gegenkante $(w, v)$ auf **„gesehen“**;

(7) **1. Fall:** `status[w]` $=$ *neu*: $T \leftarrow T \cup \{(v, w)\}$; **udfs**(`w`);

(8) **2. Fall:** `status[w]` $=$ *aktiv*: $B \leftarrow B \cup \{(v, w)\}$;

(9) `f_count++`;

(10) `f_num[`$v$`]` $\leftarrow$ `f_count`;

(11) **fin-visit**$(v)$; (∗ Aktion an $v$ bei letztem Besuch ∗)

(12) `status[`$v$`]` $\leftarrow$ **fertig**.

# Globale Tiefensuche in (ungerichtetem) Graphen $G$:

## Globale Tiefensuche in (ungerichtetem) Graphen $G$:

**Algorithmus UDFS**$(G)$ $(*$ Tiefensuche in $G = (V, E)$ $*)$

## Globale Tiefensuche in (ungerichtetem) Graphen $G$:

**Algorithmus UDFS**$(G)$ $(*$ Tiefensuche in $G = (V, E)$ $*)$

(1) `dfs_count` $\leftarrow 0$;

(2) `f_count` $\leftarrow 0$;

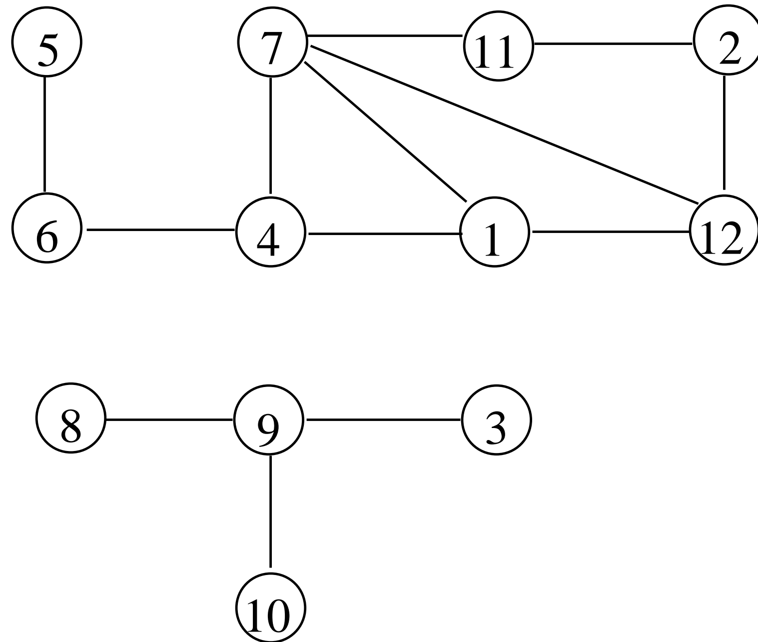
(3) **for** `v` **from** $1$ **to** $n$ **do** `status[v]` $\leftarrow$ *neu*;

(4) $T \leftarrow \emptyset$; $B \leftarrow \emptyset$;

(5) **for** `v` **from** $1$ **to** $n$ **do**

(6)     **if** `status[v]` $=$ *neu* **then**

(7)         **udfs**(`v`); $(*$ starte Tiefensuche von `v` aus $*)$




Ungerichteter Graph.

Ungerichteter Graph mit Baum- und Rückwärtskanten sowie der dfs-Nummerierung.

**Satz** **udfs**$(v_0)$ werde aufgerufen. Dann gilt:

**Satz** **udfs**$(v_0)$ werde aufgerufen. Dann gilt:

(a) **udfs**$(v_0)$ entdeckt genau die Knoten in der Zusammenhangskomponente von $v_0$.

**Satz** **udfs**$(v_0)$ werde aufgerufen. Dann gilt:

(a) **udfs**$(v_0)$ entdeckt genau die Knoten in der Zusammenhangskomponente von $v_0$.

(b) Die Kanten $(v, w)$, wo **udfs**$(w)$ unmittelbar aus **udfs**$(v)$ aufgerufen wird, bilden einen (gerichteten) Baum mit Wurzel $v_0$.

**Satz** **udfs**$(v_0)$ werde aufgerufen. Dann gilt:

(a) **udfs**$(v_0)$ entdeckt genau die Knoten in der Zusammenhangskomponente von $v_0$.

(b) Die Kanten $(v, w)$, wo **udfs**$(w)$ unmittelbar aus **udfs**$(v)$ aufgerufen wird, bilden einen (gerichteten) Baum mit Wurzel $v_0$. **(Spannbaum)**

**Satz** **udfs**$(v_0)$ werde aufgerufen. Dann gilt:

(a) **udfs**$(v_0)$ entdeckt genau die Knoten in der Zusammenhangskomponente von $v_0$.

(b) Die Kanten $(v, w)$, wo **udfs**$(w)$ unmittelbar aus **udfs**$(v)$ aufgerufen wird, bilden einen (gerichteten) Baum mit Wurzel $v_0$. **(Spannbaum)**

(c) (Satz vom weißen Weg) $u$ ist im DFS-Baum ein Nachfahr von $v$ genau dann wenn zum Zeitpunkt des Aufrufs **udfs**$(v)$ ein Weg von $v$ nach $u$ existiert, der nur neue (weiße) Knoten enthält.

**Satz** **udfs**$(v_0)$ werde aufgerufen. Dann gilt:

(a) **udfs**$(v_0)$ entdeckt genau die Knoten in der Zusammenhangskomponente von $v_0$.

(b) Die Kanten $(v, w)$, wo **udfs**$(w)$ unmittelbar aus **udfs**$(v)$ aufgerufen wird, bilden einen (gerichteten) Baum mit Wurzel $v_0$. **(Spannbaum)**

(c) (Satz vom weißen Weg) $u$ ist im DFS-Baum ein Nachfahr von $v$ genau dann wenn zum Zeitpunkt des Aufrufs **udfs**$(v)$ ein Weg von $v$ nach $u$ existiert, der nur neue (weiße) Knoten enthält.

Laufzeit (mit Initialisierung): $O(|V|+|E_{v_0}|)$, wo $E_{v_0}$ die Menge der Kanten in der Zusammenhangskomponente von $v_0$ ist.

**Satz** **udfs**$(v_0)$ werde aufgerufen. Dann gilt:

(a) **udfs**$(v_0)$ entdeckt genau die Knoten in der Zusammenhangskomponente von $v_0$.

(b) Die Kanten $(v, w)$, wo **udfs**$(w)$ unmittelbar aus **udfs**$(v)$ aufgerufen wird, bilden einen (gerichteten) Baum mit Wurzel $v_0$. **(Spannbaum)**

(c) (Satz vom weißen Weg) $u$ ist im DFS-Baum ein Nachfahr von $v$ genau dann wenn zum Zeitpunkt des Aufrufs **udfs**$(v)$ ein Weg von $v$ nach $u$ existiert, der nur neue (weiße) Knoten enthält.

Laufzeit (mit Initialisierung): $O(|V|+|E_{v_0}|)$, wo $E_{v_0}$ die Menge der Kanten in der Zusammenhangskomponente von $v_0$ ist.

*Beweis*: Ähnlich zum gerichteten Fall.

## Satz

**UDFS**$(G)$ werde aufgerufen. Dann gilt:

## Satz

**UDFS**$(G)$ werde aufgerufen. Dann gilt:

(a) Die $T$-Kanten bilden eine Reihe von Bäumen (die Tiefensuch-Bäume).

## Satz

**UDFS**$(G)$ werde aufgerufen. Dann gilt:

(a) Die $T$-Kanten bilden eine Reihe von Bäumen (die Tiefensuch-Bäume). Die Knotenmengen dieser Bäume sind die Zusammenhangskomponenten von $G$.

## Satz

**UDFS**$(G)$ werde aufgerufen. Dann gilt:

(a) Die $T$-Kanten bilden eine Reihe von Bäumen (die Tiefensuch-Bäume). Die Knotenmengen dieser Bäume sind die Zusammenhangskomponenten von $G$.

(b) Die Wurzeln der Bäume sind die jeweils ersten Knoten einer Zusammenhangskomponente in der Reihenfolge, die in Zeile (2) benutzt wird.

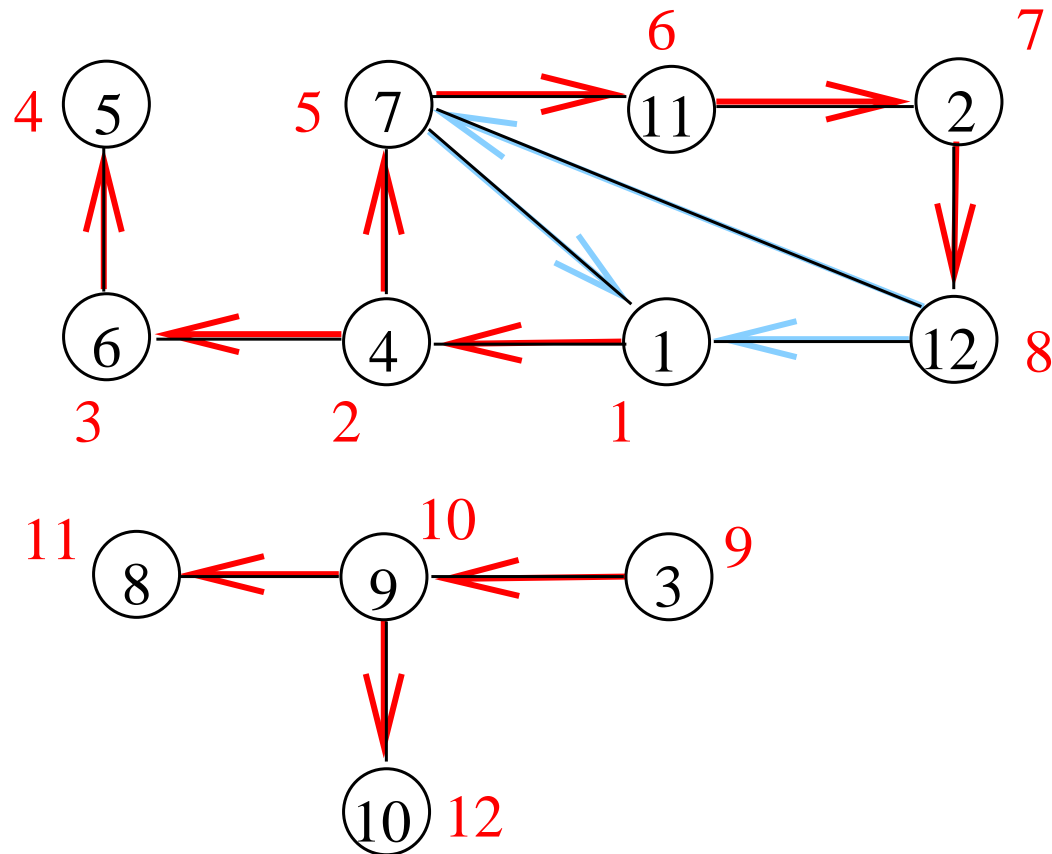
## Satz

**UDFS**$(G)$ werde aufgerufen. Dann gilt:

(a) Die $T$-Kanten bilden eine Reihe von Bäumen (die Tiefensuch-Bäume). Die Knotenmengen dieser Bäume sind die Zusammenhangskomponenten von $G$.

(b) Die Wurzeln der Bäume sind die jeweils ersten Knoten einer Zusammenhangskomponente in der Reihenfolge, die in Zeile (2) benutzt wird.

Gesamt-Laufzeit: $O(|V| + |E|)$: linear!




**Beobachtung:** Jeder Knoten wird besucht; jede Kante wird in genau einer Richtung betrachtet und als Baumkante ($T$) oder Rückwärtskante ($B$) klassifiziert; die jeweiligen Gegenkanten werden nicht betrachtet (weil sie auf „gesehen“ gesetzt werden).

## **Kreisfreiheitstest** in ungerichteten Graphen.

**Kreisfreiheitstest** in ungerichteten Graphen.

Zudem: Finden eines Kreises, wenn es einen gibt.

**Kreisfreiheitstest** in ungerichteten Graphen.

Zudem: Finden eines Kreises, wenn es einen gibt.

## Satz

Nach **UDFS**$(G)$ gilt $B \neq \emptyset$ genau dann wenn $G$ einen Kreis enthält.

*Beweis*:

„$\Rightarrow$“: Wenn in Zeile (8) eine „neue“ Kante $(v, w)$ gefunden wird, die zu einem aktiven (roten) Knoten $w$ führt, dann bildet diese Kante mit dem Abschnitt des roten Weges von $w$ nach $v$ zusammen einen Kreis der Länge $\geq 3$.
(Es kann nicht sein, dass $(w, v)$ eine Baumkante ist, da sonst $(v, w)$ den Status „gesehen“ hätte.)

*Beweis*:

„$\Rightarrow$“: Wenn in Zeile (8) eine „neue“ Kante $(v, w)$ gefunden wird, die zu einem aktiven (roten) Knoten $w$ führt, dann bildet diese Kante mit dem Abschnitt des roten Weges von $w$ nach $v$ zusammen einen Kreis der Länge $\geq 3$.
(Es kann nicht sein, dass $(w, v)$ eine Baumkante ist, da sonst $(v, w)$ den Status „gesehen“ hätte.)

„$\Leftarrow$“: Wenn $B = \emptyset$ ist, dann wird jede Kante von $G$ in einer Richtung Baumkante.
Die Kanten der UDFS-Bäume bilden ungerichtete Bäume und enthalten alle Kanten von $G$, also kann $G$ keinen Kreis haben.

□

## Bemerkung

UDFS auf einen ungerichteten Wald (Vereinigung disjunkter Bäume) angewendet gibt jeder Komponente eine Wurzel und richtet die Kanten von dieser Wurzel weg.

Dies ist eine sehr einfache und schnelle Methode (Linearzeit!), einen ungerichteten Wald zu „orientieren“.