SS 2021

Algorithmen und Datenstrukturen

8. Kapitel

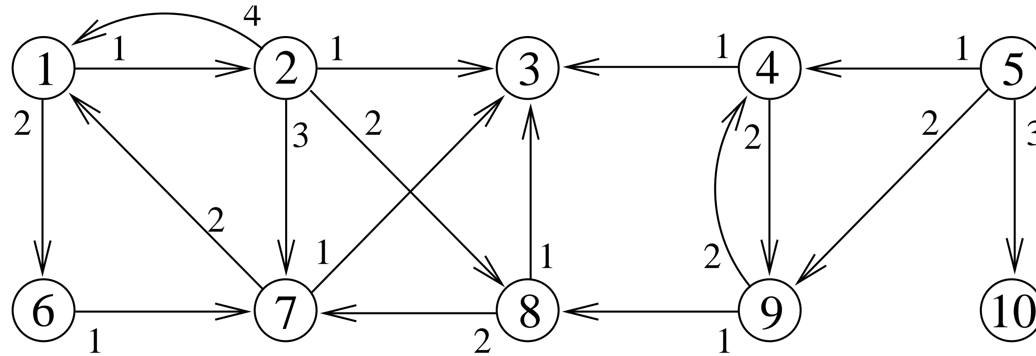
Tiefensuche

Martin Dietzfelbinger

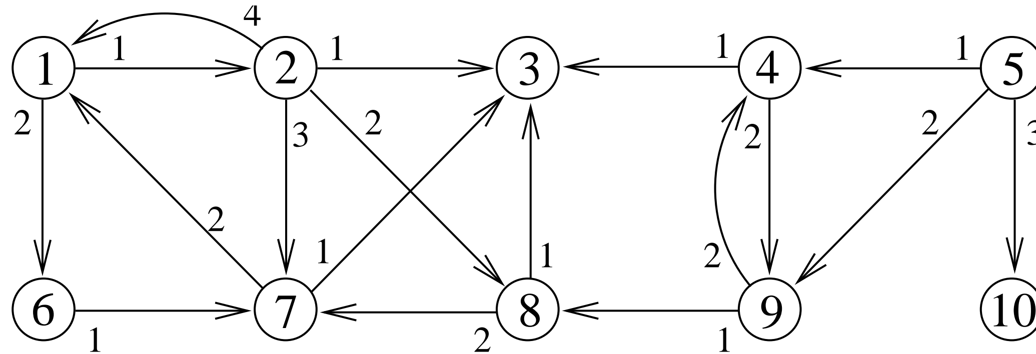
Juni 2021

8.1 Einfache Tiefensuche in Digraphen

8.1 Einfache Tiefensuche in Digraphen



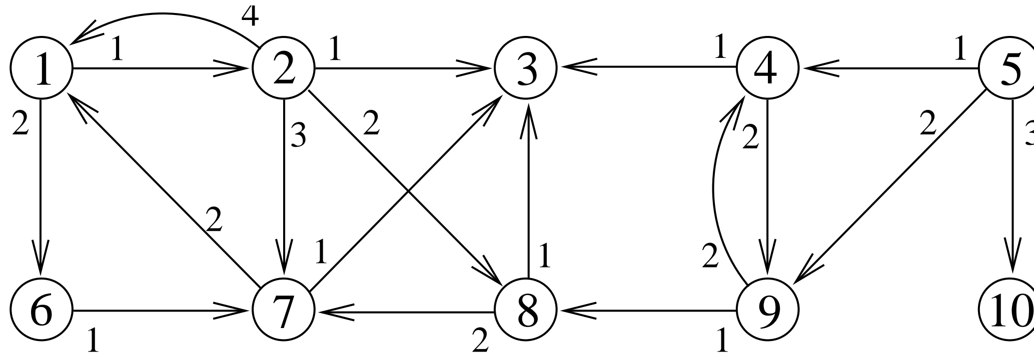
8.1 Einfache Tiefensuche in Digraphen



Eingabeformat:

Digraph $G = (V, E)$, Knotenmenge $V = \{1, \ldots, n\}$ (o. B. d. A.),

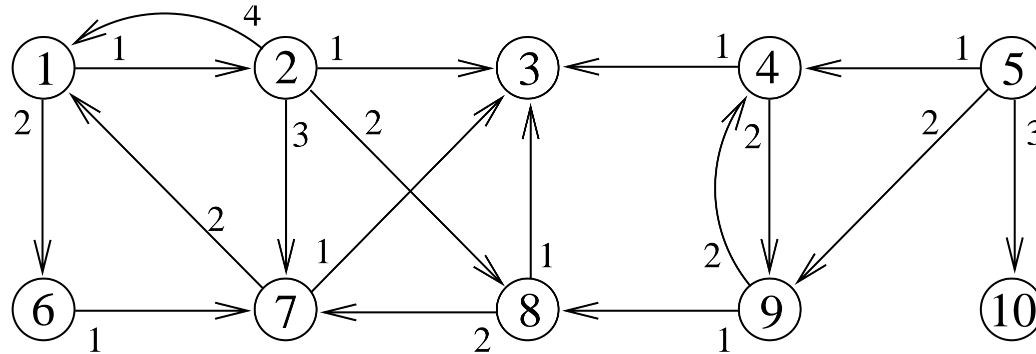
8.1 Einfache Tiefensuche in Digraphen



Eingabeformat:

Digraph $G = (V, E)$, Knotenmenge $V = \{1, \ldots, n\}$ (o. B. d. A.),
in Adjazenzlisten-/array- oder Adjazenzmatrixdarstellung,
aus v ausgehende Kanten sind (implizit) angeordnet.

8.1 Einfache Tiefensuche in Digraphen

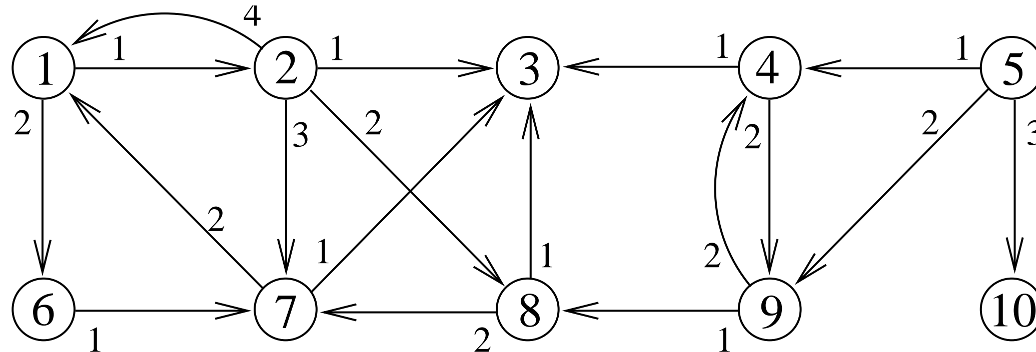


Eingabeformat:
Digraph $G = (V, E)$, Knotenmenge $V = \{1, \ldots, n\}$ (o. B. d. A.), in Adjazenzlisten-/array- oder Adjazenzmatrixdarstellung, aus v ausgehende Kanten sind (implizit) angeordnet.

Ziele:

- Besuche **alle** Knoten und Kanten.

8.1 Einfache Tiefensuche in Digraphen



Eingabeformat:
Digraph $G = (V, E)$, Knotenmenge $V = \{1, \ldots, n\}$ (o. B. d. A.), in Adjazenzlisten-/array- oder Adjazenzmatrixdarstellung, aus v ausgehende Kanten sind (implizit) angeordnet.

Ziele:

- Besuche **alle** Knoten und Kanten.
- Sammle einfache **Strukturinformation**.

Tiefensuche (Depth-First-Search)

Tiefensuche (Depth-First-Search)

Zunächst: Gehe von Knoten v_0 aus, finde alle auf Wegen von v_0 aus erreichbaren Knoten und Kanten.

Vorwärtsgehen hat **Vorrang**!

Tiefensuche (Depth-First-Search)

Zunächst: Gehe von Knoten v_0 aus, finde alle auf Wegen von v_0 aus erreichbaren Knoten und Kanten.

Vorwärtsgehen hat **Vorrang**! **Effekt:**
Die Folge der entdeckten Knoten geht immer so weit wie möglich „in die Tiefe“.

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

Zunächst: Gehe von Knoten v_0 aus, finde alle auf Wegen von v_0 aus erreichbaren Knoten und Kanten.

Vorwärtsgehen hat **Vorrang**! **Effekt:**
Die Folge der entdeckten Knoten geht immer so weit wie möglich „in die Tiefe“.

Realisierung: **Rekursive** Prozedur „**dfs**(v)“:

Tiefensuche (Depth-First-Search)

Zunächst: Gehe von Knoten v_0 aus, finde alle auf Wegen von v_0 aus erreichbaren Knoten und Kanten.

Vorwärtsgehen hat **Vorrang**! **Effekt:**
Die Folge der entdeckten Knoten geht immer so weit wie möglich „in die Tiefe“.

Realisierung: **Rekursive** Prozedur „**dfs**(v)“:

- „Besuche“ Knoten v (Aktion an diesem Knoten).

Tiefensuche (Depth-First-Search)

Zunächst: Gehe von Knoten v_0 aus, finde alle auf Wegen von v_0 aus erreichbaren Knoten und Kanten.

Vorwärtsgehen hat **Vorrang**! **Effekt:**
Die Folge der entdeckten Knoten geht immer so weit wie möglich „in die Tiefe“.

Realisierung: **Rekursive** Prozedur „**dfs(v)**“:

- „Besuche“ Knoten v (Aktion an diesem Knoten).
- Dann betrachte die Nachfolger $w_1, \ldots, w_{\text{outdeg}(v)}$ von v nacheinander, aber:

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

(Weitere Nachfolger von v werden später betrachtet.)

Man muss verhindern, dass Knoten v mehrfach besucht wird.

Man muss verhindern, dass Knoten v mehrfach besucht wird.

Dazu: „Statusinformation“, in Array `status`$[1..n]$ (oder im `nodes`Array):

Man muss verhindern, dass Knoten v mehrfach besucht wird.

Dazu: „Statusinformation", in Array `status`$[1 \ldots n]$ (oder im `nodes`Array):

v **neu** – noch nie gesehen

Man muss verhindern, dass Knoten v mehrfach besucht wird.

Dazu: „Statusinformation“, in Array $\texttt{status}[1..n]$ (oder im `nodes`Array):

v **neu** – noch nie gesehen

v **aktiv** – dfs(v) gestartet, noch nicht beendet

Man muss verhindern, dass Knoten v mehrfach besucht wird.

Dazu: „Statusinformation“, in Array `status`$[1..n]$ (oder im `nodes`Array):

v **neu** – noch nie gesehen

v **aktiv** – dfs(v) gestartet, noch nicht beendet

v **fertig** – dfs(v) beendet

Man muss verhindern, dass Knoten v mehrfach besucht wird.

Dazu: „Statusinformation“, in Array `status` $[1..n]$ (oder im `nodes`Array):

v **neu** – noch nie gesehen

v **aktiv** – dfs(v) gestartet, noch nicht beendet

v **fertig** – dfs(v) beendet

Initialisierung: Alle Knoten sind **neu**.

Man muss verhindern, dass Knoten v mehrfach besucht wird.

Dazu: „Statusinformation“, in Array `status`$[1..n]$ (oder im `nodes`Array):

v **neu** – noch nie gesehen

v **aktiv** – dfs(v) gestartet, noch nicht beendet

v **fertig** – dfs(v) beendet

Initialisierung: Alle Knoten sind **neu**.

Einfachversion:

Man muss verhindern, dass Knoten v mehrfach besucht wird.

Dazu: „Statusinformation“, in Array $\texttt{status}[1..n]$ (oder im `nodesArray`):

v **neu** – noch nie gesehen

v **aktiv** – dfs(v) gestartet, noch nicht beendet

v **fertig** – dfs(v) beendet

Initialisierung: Alle Knoten sind **neu**.

Einfachversion:

Knoten werden in der Reihenfolge der Entdeckung durchnummeriert:

Man muss verhindern, dass Knoten v mehrfach besucht wird.

Dazu: „Statusinformation“, in Array `status`$[1..n]$ (oder im `nodes`Array):

v **neu** – noch nie gesehen

v **aktiv** – dfs(v) gestartet, noch nicht beendet

v **fertig** – dfs(v) beendet

Initialisierung: Alle Knoten sind **neu**.

Einfachversion:

Knoten werden in der Reihenfolge der Entdeckung durchnummeriert:

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

Man muss verhindern, dass Knoten v mehrfach besucht wird.

Dazu: „Statusinformation“, in Array `status`$[1..n]$ (oder im `nodes`Array):

v **neu** – noch nie gesehen

v **aktiv** – dfs(v) gestartet, noch nicht beendet

v **fertig** – dfs(v) beendet

Initialisierung: Alle Knoten sind **neu**.

Einfachversion:

Knoten werden in der Reihenfolge der Entdeckung durchnummeriert:

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

Mitzählen in `dfs_count` (globale Variable), mit 0 initialisiert.

Man muss verhindern, dass Knoten v mehrfach besucht wird.

Dazu: „Statusinformation“, in Array `status`$[1\,..\,n]$ (oder im `nodes`Array):

v **neu** – noch nie gesehen

v **aktiv** – dfs(v) gestartet, noch nicht beendet

v **fertig** – dfs(v) beendet

Initialisierung: Alle Knoten sind **neu**.

Einfachversion:

Knoten werden in der Reihenfolge der Entdeckung durchnummeriert:

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

Mitzählen in `dfs_count` (globale Variable), mit 0 initialisiert.

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

Prozedur dfs(v) // Tiefensuche an v, **rekursiv**,

// nur für v mit `status`$[v]$ = **neu** aufzurufen

Prozedur dfs(v) // Tiefensuche an v, **rekursiv**,
// nur für v mit `status[`v`]` = **neu** aufzurufen

(1) `dfs_count++;`

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

Prozedur dfs(v) // Tiefensuche an v, **rekursiv**,

// nur für v mit `status[`v`]` $=$ **neu** aufzurufen

(1) `dfs_count++;`

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

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

Prozedur dfs(v) // Tiefensuche an v, **rekursiv**,
// nur für v mit `status`$[v]$ = **neu** aufzurufen

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

Prozedur dfs(v) // Tiefensuche an v, **rekursiv**,

// nur für v mit `status[v]` = **neu** aufzurufen

(1) `dfs_count++;`

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

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

(4) dfs-visit(v); // Aktion an v bei Erstbesuch

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

Prozedur dfs(v) // Tiefensuche an v, **rekursiv**,

// nur für v mit `status[v]` $=$ **neu** aufzurufen

(1) `dfs_count++;`

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

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

(4) dfs-visit(v); // Aktion an v bei Erstbesuch

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

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

Prozedur dfs(v) // Tiefensuche an v, **rekursiv**,
// nur für v mit status[v] $=$ **neu** aufzurufen

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

Prozedur dfs(v) // Tiefensuche an v, **rekursiv**,
// nur für v mit status[v] = **neu** aufzurufen

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

Prozedur dfs(v) // Tiefensuche an v, **rekursiv**,
// nur für v mit $\texttt{status}[v] =$ **neu** aufzurufen

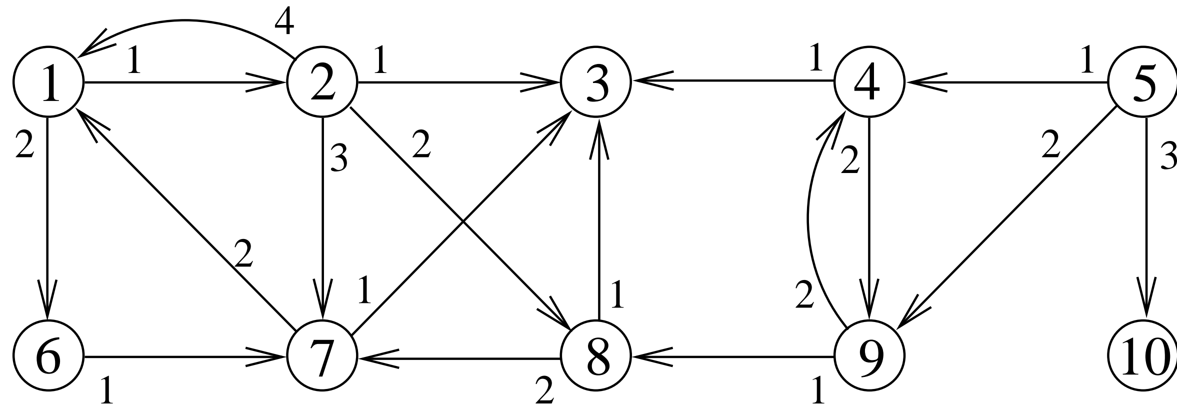
(1) $\texttt{dfs_count++}$;
(2) $\texttt{dfs_num}[v] \leftarrow \texttt{dfs_count}$;
(3) $\texttt{status}[v] \leftarrow$ **aktiv** ;
(4) dfs-visit(v); // Aktion an v bei Erstbesuch
(5) für jeden Nachfolger $\texttt{w}$ von v (Adjazenzliste!) tue:
(6) **if** $\texttt{status}[\texttt{w}] =$ **neu** **then**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

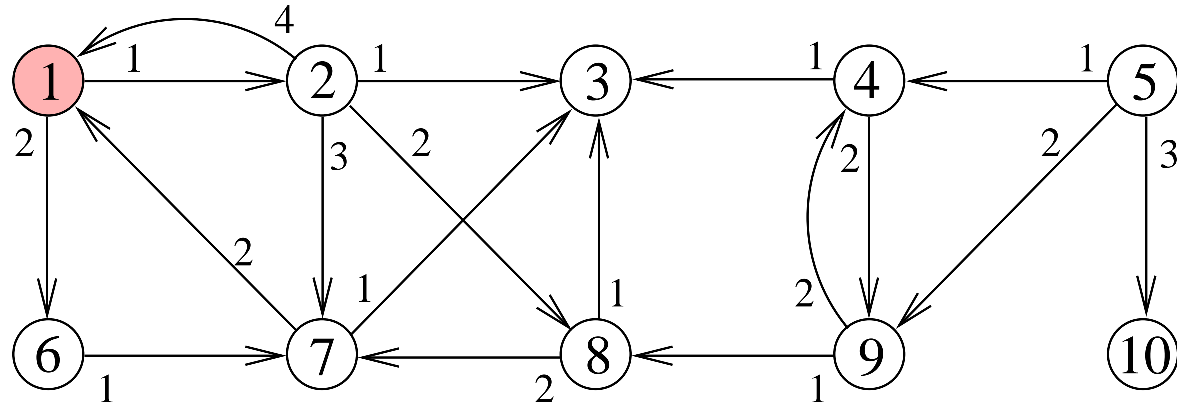
Tiefensuche von v_0 aus:
Initialisiere alle Knoten als „**neu**“; setze $\texttt{dfs_count} \leftarrow 0$.

Prozedur dfs(v) // Tiefensuche an v, **rekursiv**,
// nur für v mit `status[v]` $=$ **neu** aufzurufen

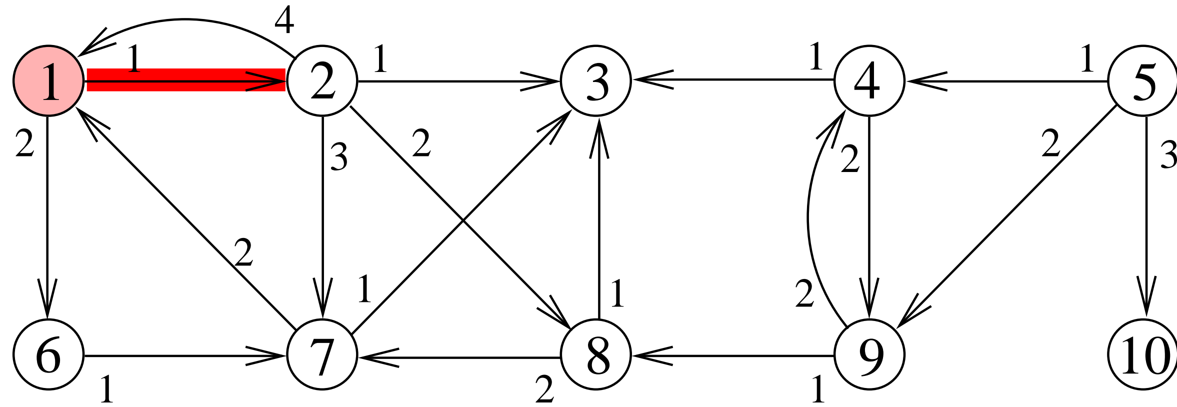
(1) `dfs_count++;`
(2) `dfs_num[v]` $\leftarrow$ `dfs_count;`
(3) `status[v]` $\leftarrow$ **aktiv** ;
(4) dfs-visit(v); // Aktion an v bei Erstbesuch
(5) für jeden Nachfolger `w` von v (Adjazenzliste!) tue:
(6) **if** `status[w]` $=$ **neu** **then** // `w` wird entdeckt!
(7) dfs(`w`);
(8) `status[v]`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

Tiefensuche von v_0 aus:
Initialisiere alle Knoten als „**neu**“; setze `dfs_count` $\leftarrow 0$.
Rufe dfs(v_0) auf.

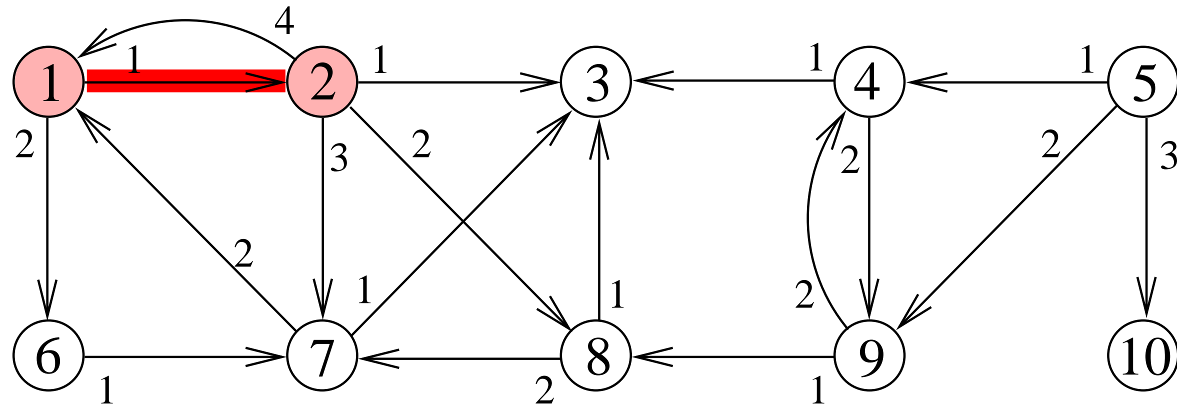

1
2
3
4
5
6
7
8
9
10



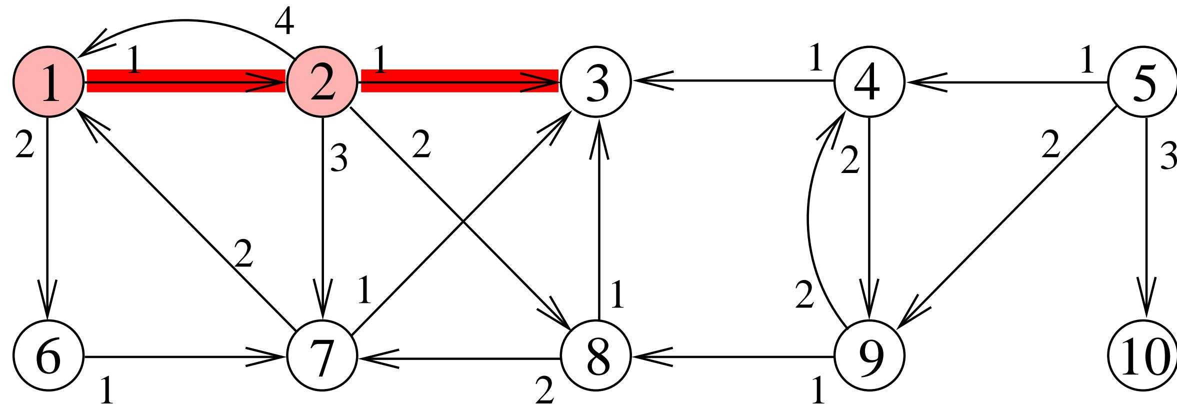
„Roter Weg“: **(1)**



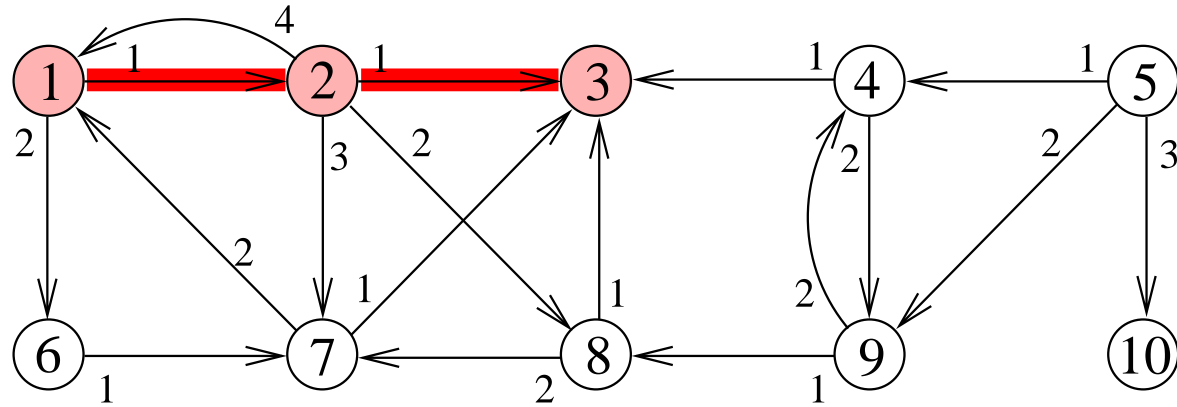
„Roter Weg“: **(1)**



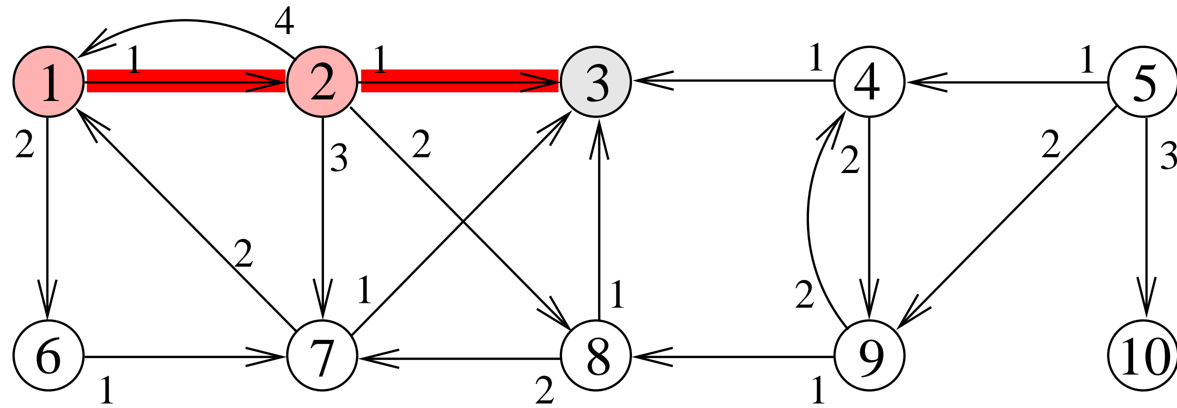
„Roter Weg“: $\mathbf{(1,2)}$



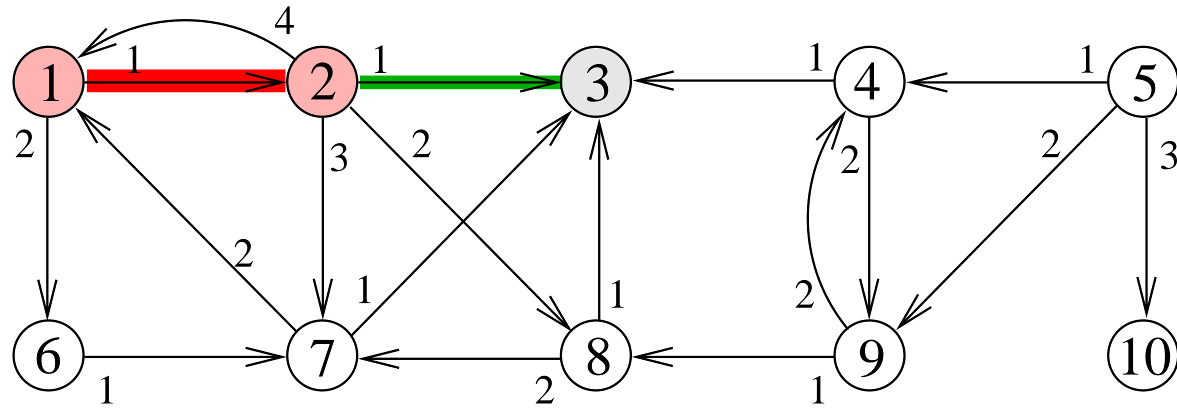
„Roter Weg“: $\mathbf{(1, 2)}$



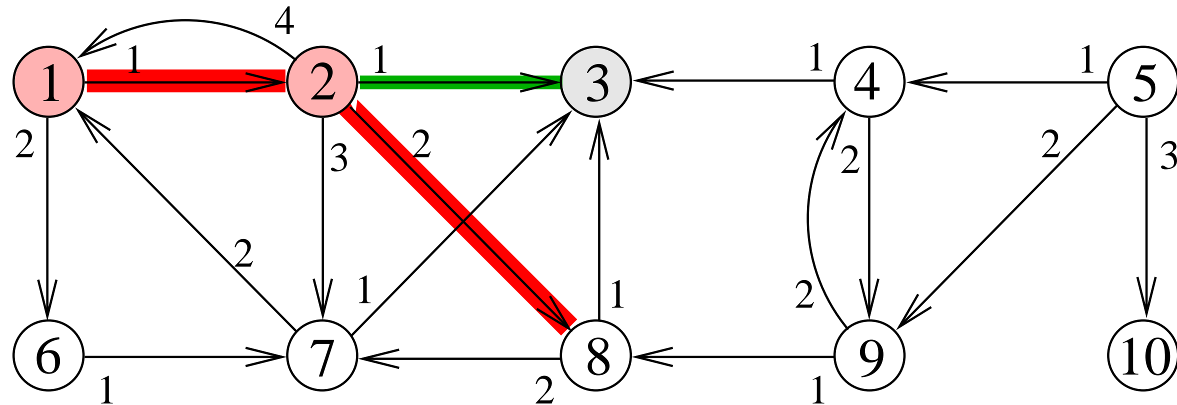
„Roter Weg“: $\mathbf{(1, 2, 3)}$



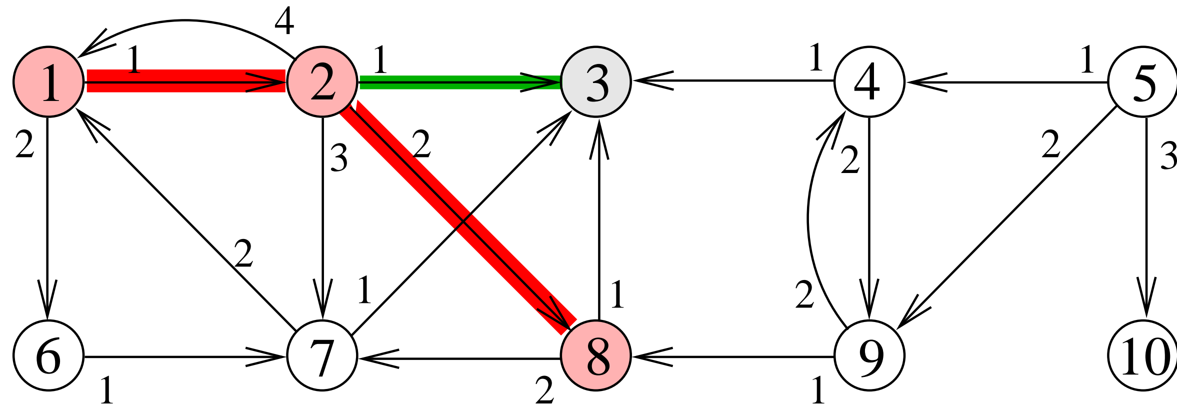
„Roter Weg“: $\mathbf{(1, 2)}$



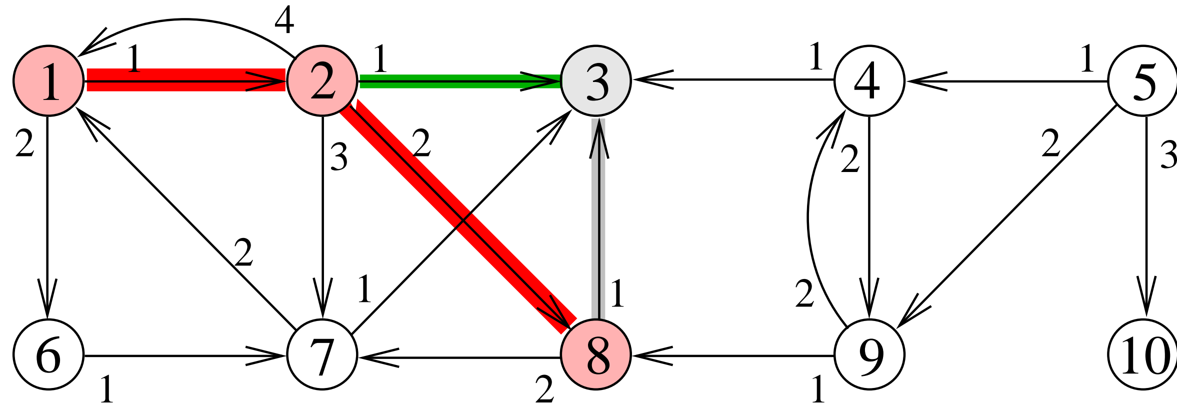
„Roter Weg“: $\mathbf{(1, 2)}$



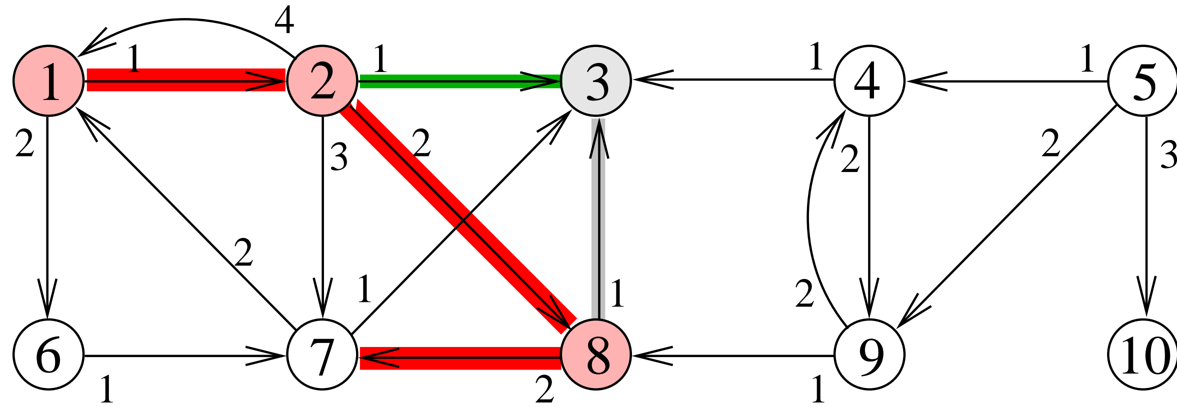
„Roter Weg“: $\mathbf{(1, 2)}$



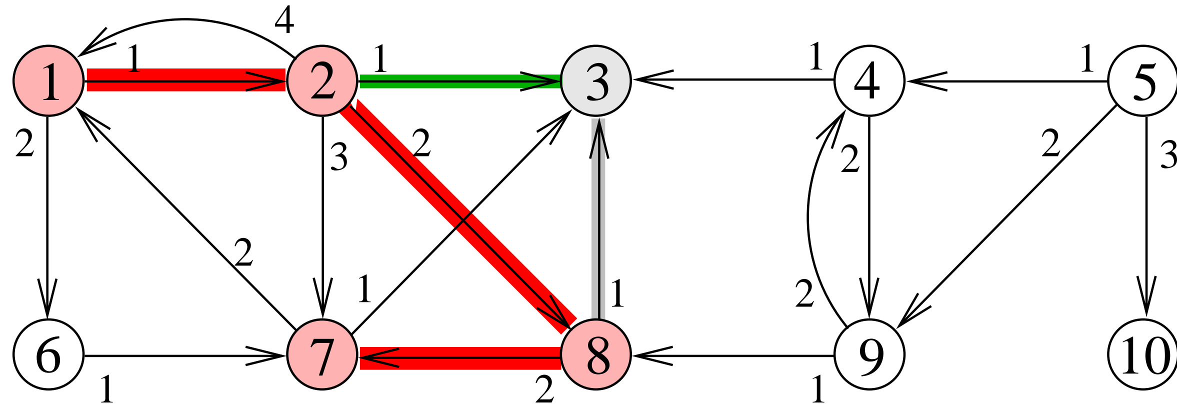
„Roter Weg“: $\mathbf{(1, 2, 8)}$



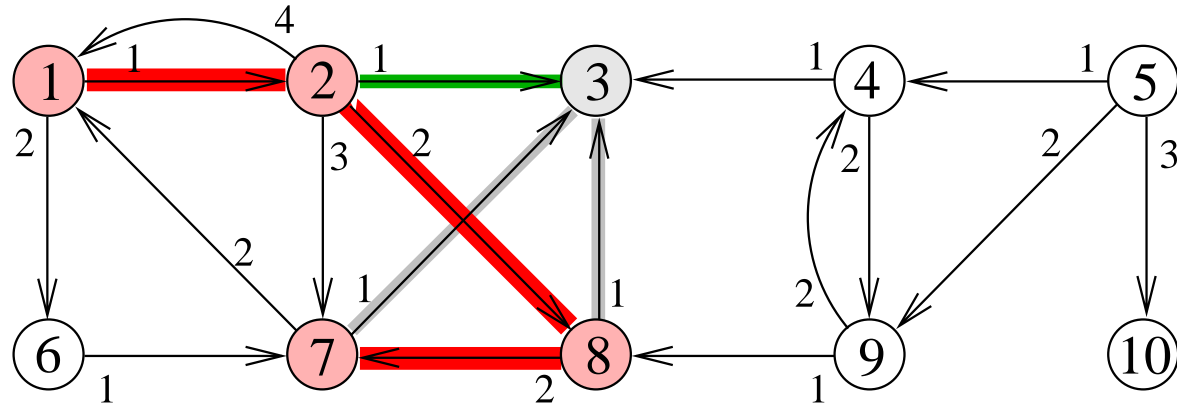
„Roter Weg“: $\mathbf{(1, 2, 8)}$



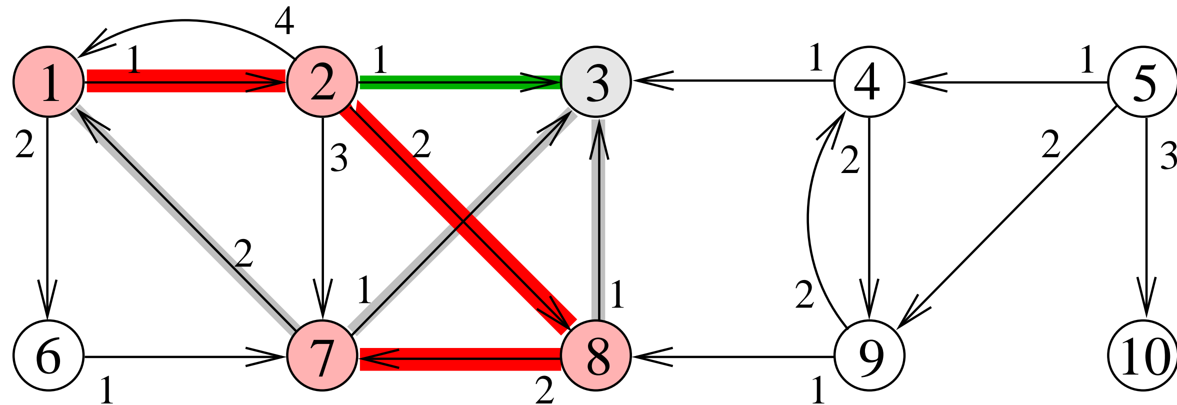
„Roter Weg“: $\mathbf{(1, 2, 8)}$



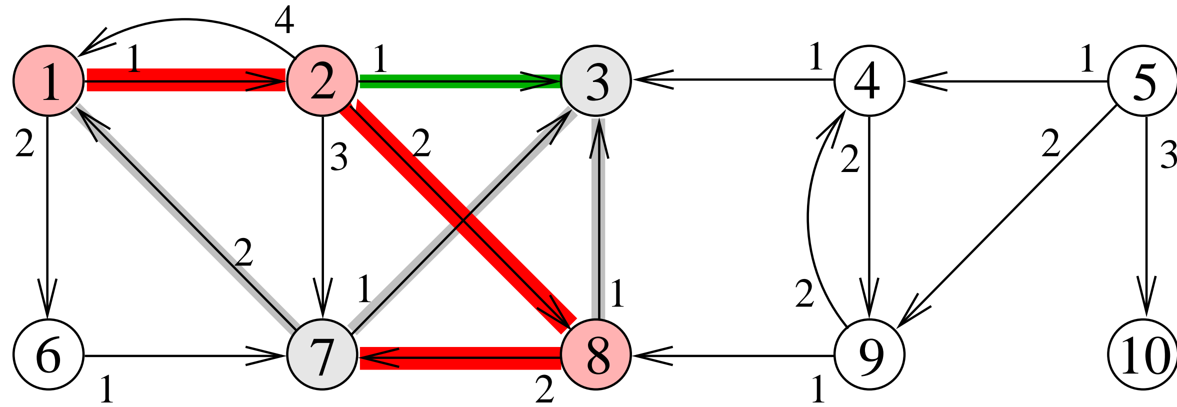
„Roter Weg": $\mathbf{(1, 2, 8, 7)}$



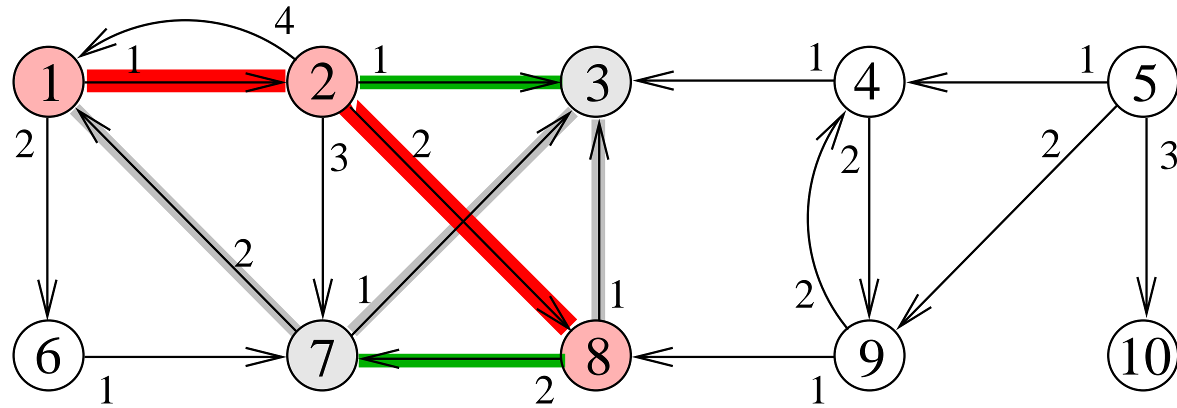
„Roter Weg“: $\mathbf{(1, 2, 8, 7)}$



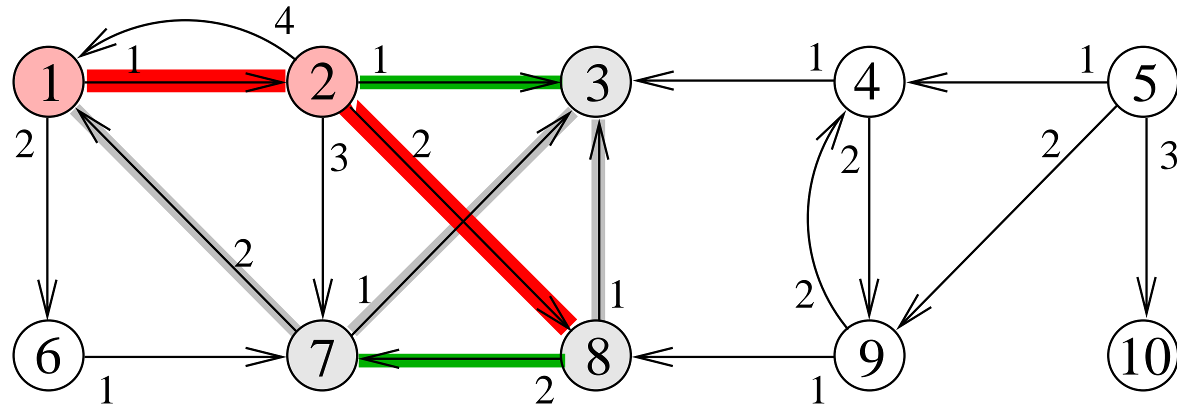
„Roter Weg": $\mathbf{(1, 2, 8, 7)}$



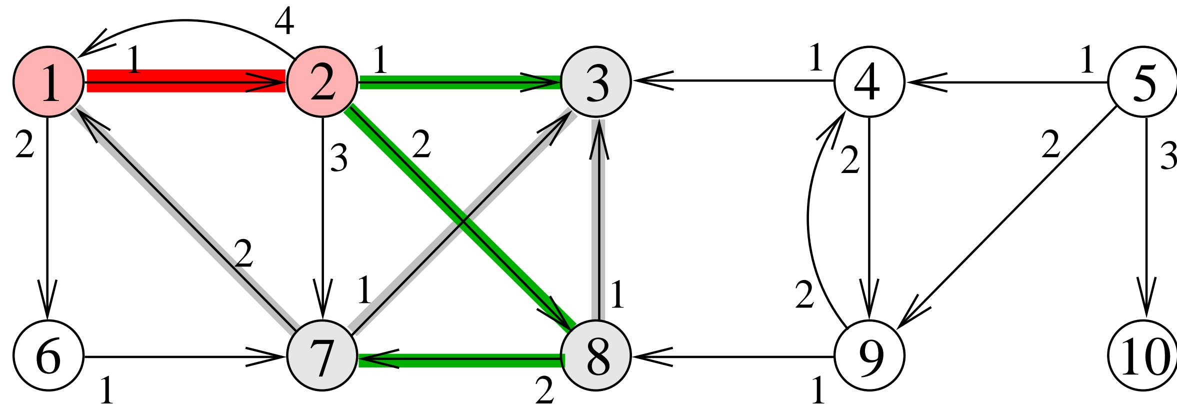
„Roter Weg“: $\mathbf{(1, 2, 8)}$



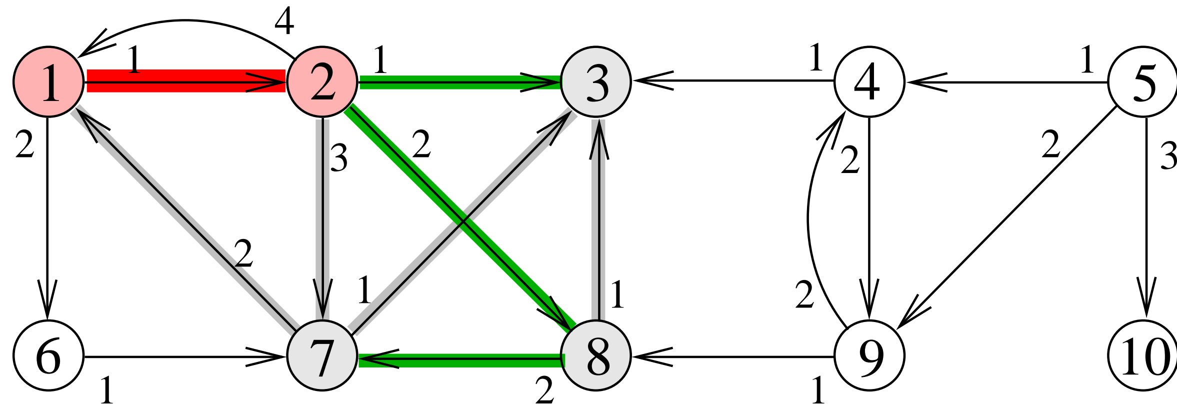
„Roter Weg“: $\mathbf{(1, 2, 8)}$



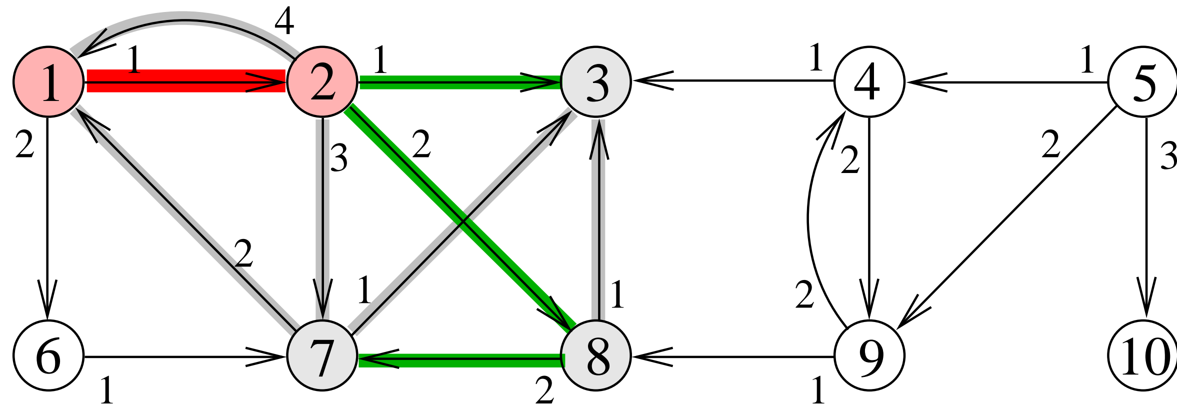
„Roter Weg“: $\mathbf{(1, 2)}$



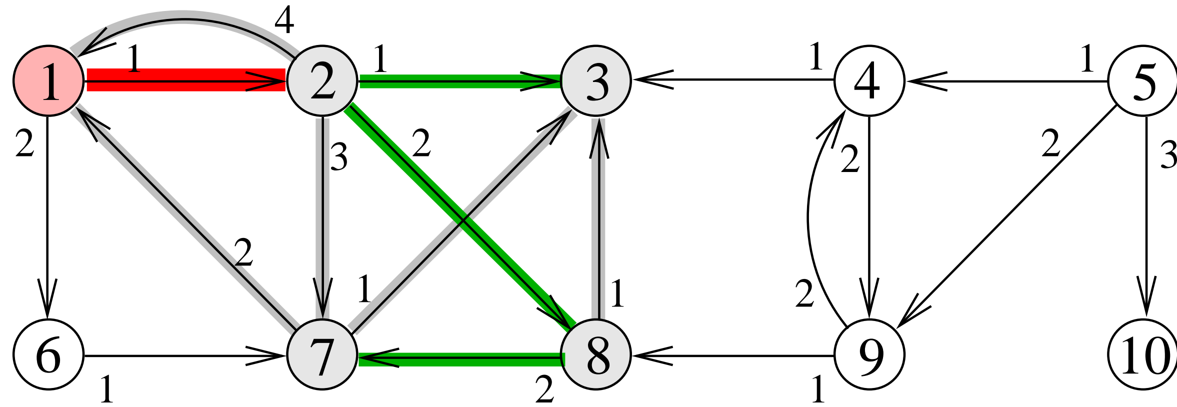
„Roter Weg“: $\mathbf{(1,2)}$



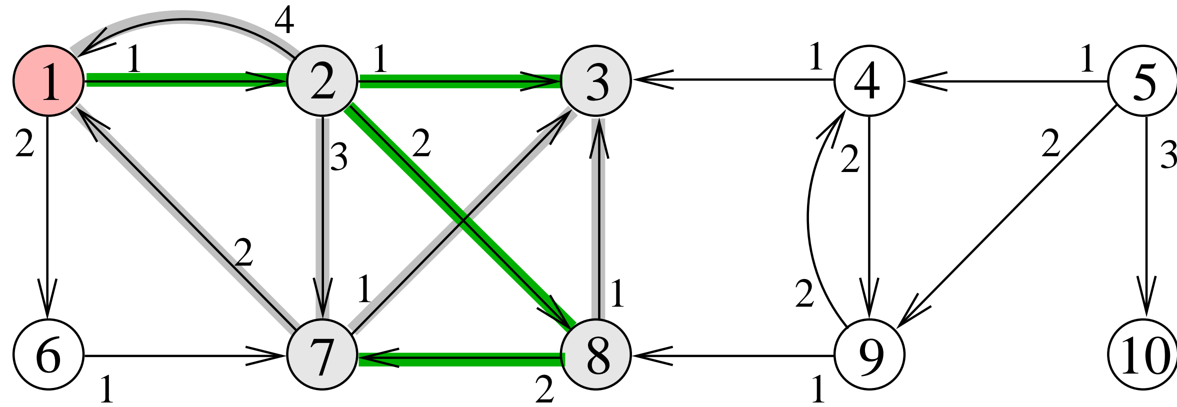
„Roter Weg“: $\mathbf{(1, 2)}$



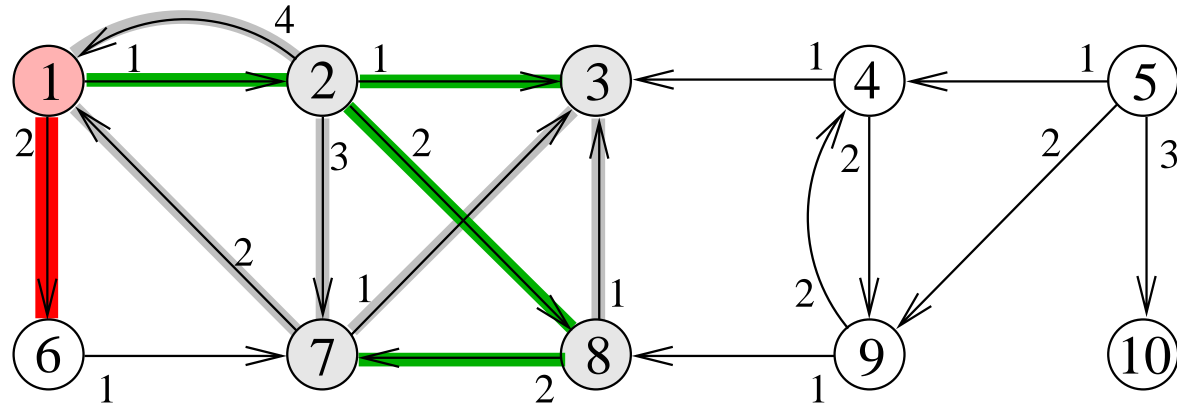
„Roter Weg“: $\mathbf{(1,2)}$



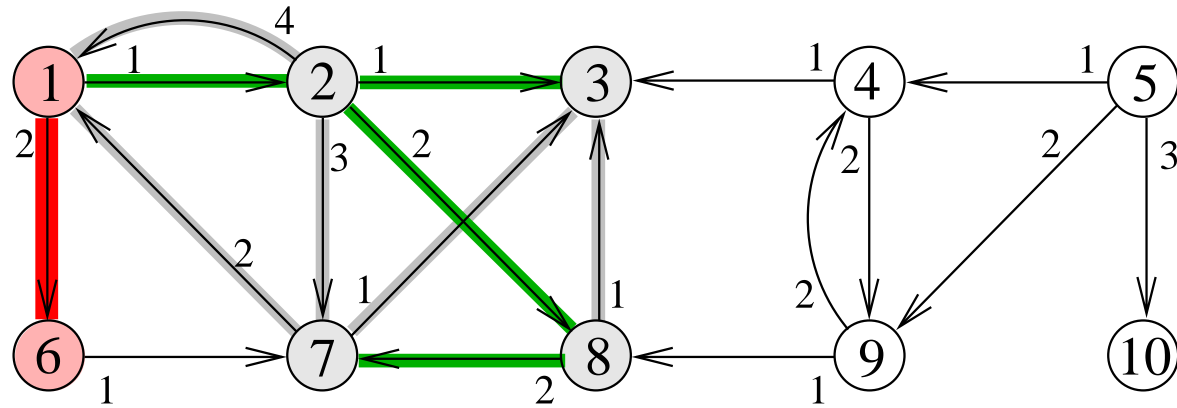
„Roter Weg“: **(1)**



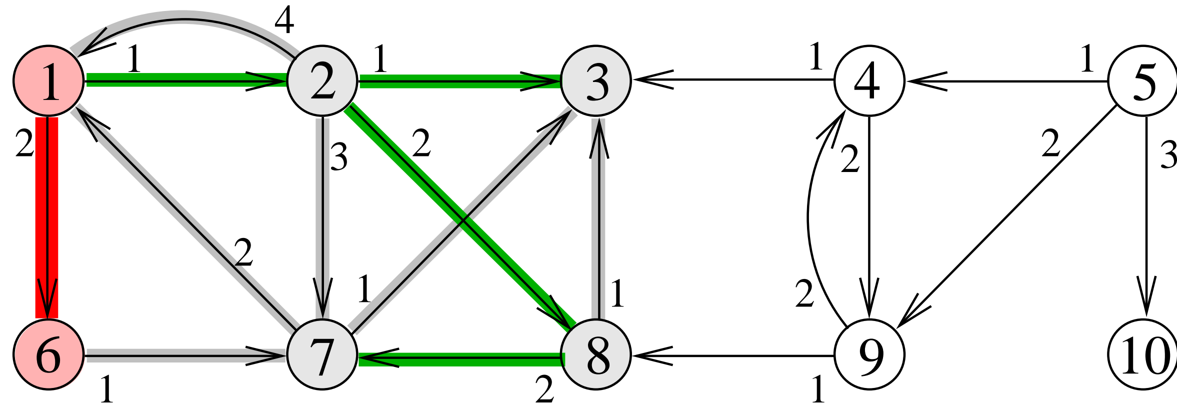
„Roter Weg“: **(1)**



„Roter Weg“: (1)

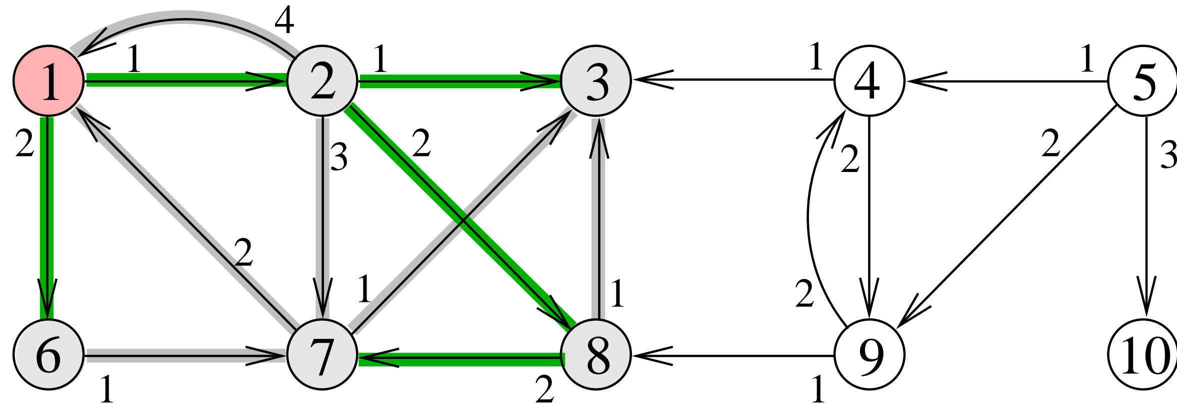


„Roter Weg“: $\mathbf{(1,6)}$

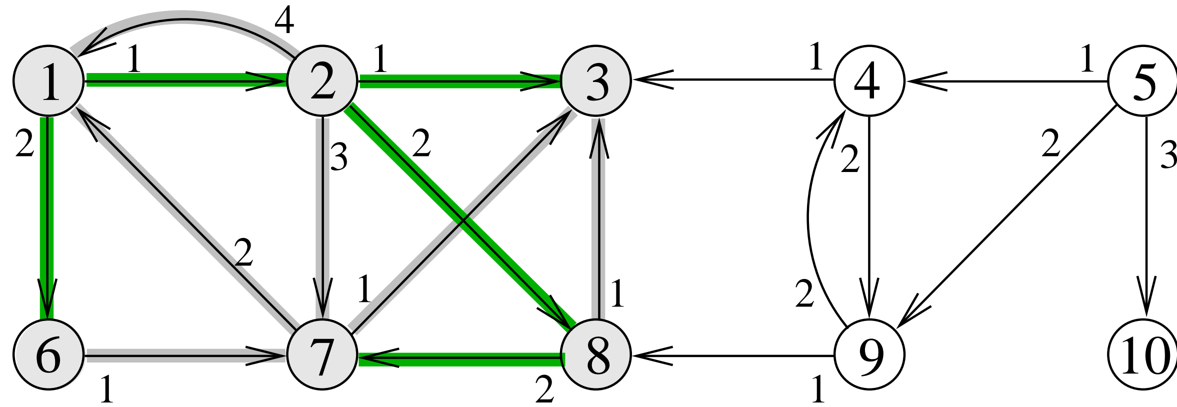


„Roter Weg“: $\mathbf{(1, 6)}$

„Roter Weg“: (1)



„Roter Weg“: **(1)**



„Roter Weg“: leer.

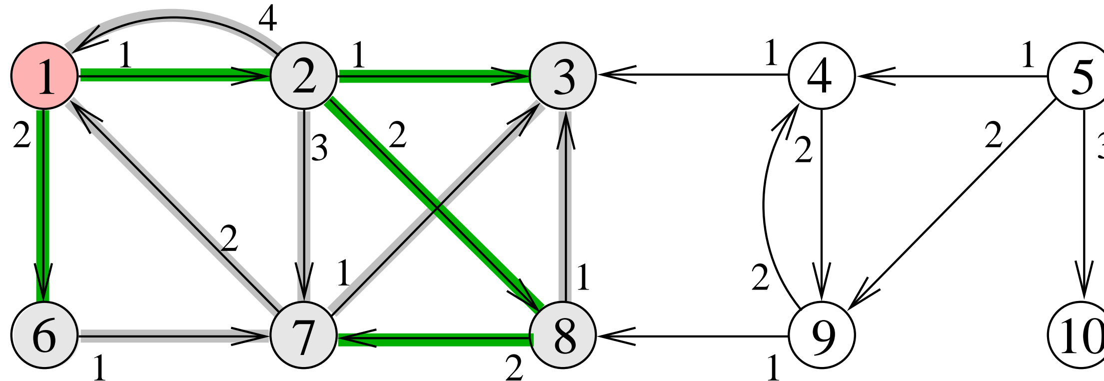
$\boldsymbol{R_{v_0}} = \{v \in V \mid v_0 \rightsquigarrow v\}$. (Von v_0 aus erreichbare **Knoten**.)

$\boldsymbol{R_{v_0}} = \{v \in V \mid v_0 \leadsto v\}$. (Von v_0 aus erreichbare **Knoten**.)

$\boldsymbol{E_{v_0}} = \{(v, w) \in E \mid v_0 \leadsto v\}$. (Von v_0 aus erreichbare **Kanten**.)

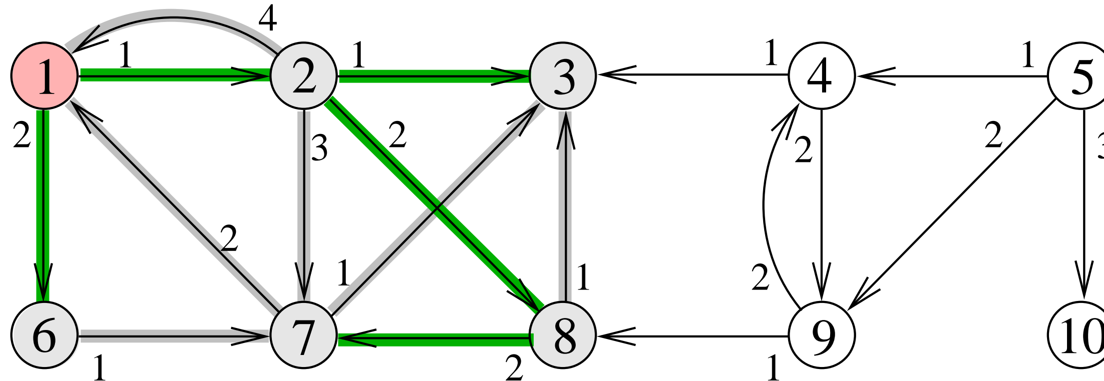
$\boldsymbol{R_{v_0}} = \{v \in V \mid v_0 \rightsquigarrow v\}$. (Von v_0 aus erreichbare **Knoten**.)

$\boldsymbol{E_{v_0}} = \{(v, w) \in E \mid v_0 \rightsquigarrow v\}$. (Von v_0 aus erreichbare **Kanten**.)



$\boldsymbol{R_{v_0}} = \{v \in V \mid v_0 \rightsquigarrow v\}$. (Von v_0 aus erreichbare **Knoten**.)

$\boldsymbol{E_{v_0}} = \{(v,w) \in E \mid v_0 \rightsquigarrow v\}$. (Von v_0 aus erreichbare **Kanten**.)



Satz 8.1.1

Bei Tiefensuche von v_0 aus wird für jeden Knoten $v \in R_{v_0}$ dfs(v) (genau einmal) aufgerufen, für Knoten $v \notin R_{v_0}$ wird dfs(v) nicht aufgerufen. Jede Kante $(v,w) \in E_{v_0}$ wird genau einmal betrachtet, andere Kanten nicht.

$\boldsymbol{R_{v_0}} = \{v \in V \mid v_0 \leadsto v\}$. (Von v_0 aus erreichbare **Knoten**.)

$\boldsymbol{E_{v_0}} = \{(v, w) \in E \mid v_0 \leadsto v\}$. (Von v_0 aus erreichbare **Kanten**.)

Satz 8.1.1

Bei Tiefensuche von v_0 aus wird für jeden Knoten $v \in R_{v_0}$ dfs(v) (genau einmal) aufgerufen, für Knoten $v \notin R_{v_0}$ wird dfs(v) nicht aufgerufen. Jede Kante $(v, w) \in E_{v_0}$ wird genau einmal betrachtet, andere Kanten nicht.

Anm.: Auch Breitensuche von v_0 aus besucht genau diese Knoten/Kanten.

Beweis:

(1) Da Aufrufe nur entlang von Kanten von G erfolgen,
kann dfs(v) nur aufgerufen werden, wenn $v_0 \leadsto v$ gilt.

Beweis:

(1) Da Aufrufe nur entlang von Kanten von G erfolgen,
kann dfs(v) nur aufgerufen werden, wenn $v_0 \rightsquigarrow v$ gilt.

(2) Sobald dfs(v) aufgerufen wird, wird `status[`v`]` auf „aktiv“ gesetzt, später auf „fertig“, aber nie mehr zurück auf „neu“. Ein Aufruf erfolgt nur für „neue“ Knoten. Daher: **maximal** ein Aufruf für v.

Beweis:

(1) Da Aufrufe nur entlang von Kanten von G erfolgen,
kann dfs(v) nur aufgerufen werden, wenn $v_0 \rightsquigarrow v$ gilt.

(2) Sobald dfs(v) aufgerufen wird, wird `status[`v`]` auf „aktiv“ gesetzt, später auf „fertig“, aber nie mehr zurück auf „neu“. Ein Aufruf erfolgt nur für „neue“ Knoten. Daher: **maximal** ein Aufruf für v.

(3) Sei $(v_0, v_1, \ldots, v_t = v)$ ein Weg von v_0 nach v.

Beweis:

(1) Da Aufrufe nur entlang von Kanten von G erfolgen,
kann dfs(v) nur aufgerufen werden, wenn $v_0 \rightsquigarrow v$ gilt.

(2) Sobald dfs(v) aufgerufen wird, wird `status`$[v]$ auf „aktiv“ gesetzt, später auf „fertig“, aber nie mehr zurück auf „neu“. Ein Aufruf erfolgt nur für „neue“ Knoten. Daher: **maximal** ein Aufruf für v.

(3) Sei $(v_0, v_1, \ldots, v_t = v)$ ein Weg von v_0 nach v.
Annahme: Es gibt $i > 0$, für das dfs(v_i) nicht aufgerufen wird.

Beweis:

(1) Da Aufrufe nur entlang von Kanten von G erfolgen,
kann dfs(v) nur aufgerufen werden, wenn $v_0 \rightsquigarrow v$ gilt.

(2) Sobald dfs(v) aufgerufen wird, wird `status`$[v]$ auf „aktiv“ gesetzt, später auf „fertig“, aber nie mehr zurück auf „neu“. Ein Aufruf erfolgt nur für „neue“ Knoten. Daher: **maximal** ein Aufruf für v.

(3) Sei $(v_0, v_1, \ldots, v_t = v)$ ein Weg von v_0 nach v.
Annahme: Es gibt $i > 0$, für das dfs(v_i) nicht aufgerufen wird.
Betrachte das kleinste solche i.

Beweis:

(1) Da Aufrufe nur entlang von Kanten von G erfolgen,
kann dfs(v) nur aufgerufen werden, wenn $v_0 \rightsquigarrow v$ gilt.

(2) Sobald dfs(v) aufgerufen wird, wird `status[`v`]` auf „aktiv“ gesetzt, später auf „fertig“, aber nie mehr zurück auf „neu“. Ein Aufruf erfolgt nur für „neue“ Knoten. Daher: **maximal** ein Aufruf für v.

(3) Sei $(v_0, v_1, \ldots, v_t = v)$ ein Weg von v_0 nach v.
Annahme: Es gibt $i > 0$, für das dfs(v_i) nicht aufgerufen wird.
Betrachte das kleinste solche i. Dann wird dfs(v_{i-1}) aufgerufen.

Beweis:

(1) Da Aufrufe nur entlang von Kanten von G erfolgen,
kann dfs(v) nur aufgerufen werden, wenn $v_0 \rightsquigarrow v$ gilt.

(2) Sobald dfs(v) aufgerufen wird, wird `status[`v`]` auf „aktiv" gesetzt, später auf „fertig", aber nie mehr zurück auf „neu". Ein Aufruf erfolgt nur für „neue" Knoten. Daher: **maximal** ein Aufruf für v.

(3) Sei $(v_0, v_1, \ldots, v_t = v)$ ein Weg von v_0 nach v.
Annahme: Es gibt $i > 0$, für das dfs(v_i) nicht aufgerufen wird.
Betrachte das kleinste solche i. Dann wird dfs(v_{i-1}) aufgerufen.
Dabei wird die Kante (v_{i-1}, v_i) untersucht und festgestellt, dass v_i „neu" ist.

Beweis:

(1) Da Aufrufe nur entlang von Kanten von G erfolgen,
kann dfs(v) nur aufgerufen werden, wenn $v_0 \rightsquigarrow v$ gilt.

(2) Sobald dfs(v) aufgerufen wird, wird `status[`v`]` auf „aktiv“ gesetzt, später auf „fertig“, aber nie mehr zurück auf „neu“. Ein Aufruf erfolgt nur für „neue“ Knoten. Daher: **maximal** ein Aufruf für v.

(3) Sei $(v_0, v_1, \ldots, v_t = v)$ ein Weg von v_0 nach v.
Annahme: Es gibt $i > 0$, für das dfs(v_i) nicht aufgerufen wird.
Betrachte das kleinste solche i. Dann wird dfs(v_{i-1}) aufgerufen.
Dabei wird die Kante (v_{i-1}, v_i) untersucht und festgestellt, dass v_i „neu“ ist. Also wird dfs(v_i) aufgerufen,

Beweis:

(1) Da Aufrufe nur entlang von Kanten von G erfolgen,
kann dfs(v) nur aufgerufen werden, wenn $v_0 \leadsto v$ gilt.

(2) Sobald dfs(v) aufgerufen wird, wird `status[`v`]` auf „aktiv“ gesetzt, später auf „fertig“, aber nie mehr zurück auf „neu“. Ein Aufruf erfolgt nur für „neue“ Knoten. Daher: **maximal** ein Aufruf für v.

(3) Sei $(v_0, v_1, \ldots, v_t = v)$ ein Weg von v_0 nach v.
Annahme: Es gibt $i > 0$, für das dfs(v_i) nicht aufgerufen wird.
Betrachte das kleinste solche i. Dann wird dfs(v_{i-1}) aufgerufen.
Dabei wird die Kante (v_{i-1}, v_i) untersucht und festgestellt, dass v_i „neu“ ist. Also wird dfs(v_i) aufgerufen, **Widerspruch**.

Beweis:

(1) Da Aufrufe nur entlang von Kanten von G erfolgen,
kann dfs(v) nur aufgerufen werden, wenn $v_0 \rightsquigarrow v$ gilt.

(2) Sobald dfs(v) aufgerufen wird, wird `status`$[v]$ auf „aktiv“ gesetzt, später auf „fertig“, aber nie mehr zurück auf „neu“. Ein Aufruf erfolgt nur für „neue“ Knoten. Daher: **maximal** ein Aufruf für v.

(3) Sei $(v_0, v_1, \ldots, v_t = v)$ ein Weg von v_0 nach v.
Annahme: Es gibt $i > 0$, für das dfs(v_i) nicht aufgerufen wird.
Betrachte das kleinste solche i. Dann wird dfs(v_{i-1}) aufgerufen.
Dabei wird die Kante (v_{i-1}, v_i) untersucht und festgestellt, dass v_i „neu“ ist. Also wird dfs(v_i) aufgerufen, **Widerspruch**.

(4) Eine Kante $(v, w) \in E_{v_0}$ wird im Aufruf dfs(v) untersucht (und in keinem anderen Aufruf).

Beweis:

(1) Da Aufrufe nur entlang von Kanten von G erfolgen,
kann dfs(v) nur aufgerufen werden, wenn $v_0 \rightsquigarrow v$ gilt.

(2) Sobald dfs(v) aufgerufen wird, wird `status[`v`]` auf „aktiv" gesetzt, später auf „fertig", aber nie mehr zurück auf „neu". Ein Aufruf erfolgt nur für „neue" Knoten. Daher: **maximal** ein Aufruf für v.

(3) Sei $(v_0, v_1, \ldots, v_t = v)$ ein Weg von v_0 nach v.
Annahme: Es gibt $i > 0$, für das dfs(v_i) nicht aufgerufen wird.
Betrachte das kleinste solche i. Dann wird dfs(v_{i-1}) aufgerufen.
Dabei wird die Kante (v_{i-1}, v_i) untersucht und festgestellt, dass v_i „neu" ist. Also wird dfs(v_i) aufgerufen, **Widerspruch**.

(4) Eine Kante $(v, w) \in E_{v_0}$ wird im Aufruf dfs(v) untersucht (und in keinem anderen Aufruf).
Kanten (v, w) mit $v \notin R_{v_0}$ werden nie betrachtet. □

Beobachtung 1:

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

Beobachtung 1:

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

Der **Zeitaufwand** für den gesamten Aufruf dfs(v_0) ist **linear**, nämlich

$$O\Big(\sum_{v \in R_{v_0}} (1 + \text{outdeg}(v))\Big) = O(|E_{v_0}|).$$

Beobachtung 1:

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

Der **Zeitaufwand** für den gesamten Aufruf dfs(v_0) ist **linear**, nämlich

$$O\Big(\sum_{v \in R_{v_0}} (1 + \text{outdeg}(v))\Big) = O(|E_{v_0}|).$$

(Beachte: $|R_{v_0}| \leq |E_{v_0}| + 1$.)

Beobachtung 1:

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

Der **Zeitaufwand** für den gesamten Aufruf dfs(v_0) ist **linear**, nämlich

$$O\Big(\sum_{v \in R_{v_0}} (1 + \mathrm{outdeg}(v))\Big) = O(|E_{v_0}|).$$

(Beachte: $|R_{v_0}| \le |E_{v_0}| + 1$.)

Der Zeitaufwand für die Initialisierung ist $O(n)$. – Also:

Mit Tiefensuche in G von v_0 aus lassen sich in Zeit $O(|V| + |E_{v_0}|)$
die Mengen R_{v_0} und E_{v_0} ermitteln.

Definition 8.1.2

Definition 8.1.2

Die (**reflexive** und) **transitive Hülle**

Definition 8.1.2

Die (**reflexive** und) **transitive Hülle** eines Digraphen G ist der Digraph **TH**$(G) := (V, E^*)$ mit Kantenmenge

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

Definition 8.1.2

Die (**reflexive** und) **transitive Hülle** eines Digraphen G ist der Digraph **TH(G)** $:= (V, E^*)$ mit Kantenmenge

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

Tiefensuche in G, gestartet von jedem Knoten v nacheinander, ermittelt alle Mengen R_v, $v \in V$, also **TH**$(G) = (V, E^*)$, in Zeit

$$O\Big(\sum_{v \in V} |E_{R_v}|\Big) = O(|E^*|) = O(|V| \cdot |E|).$$

Definition 8.1.2

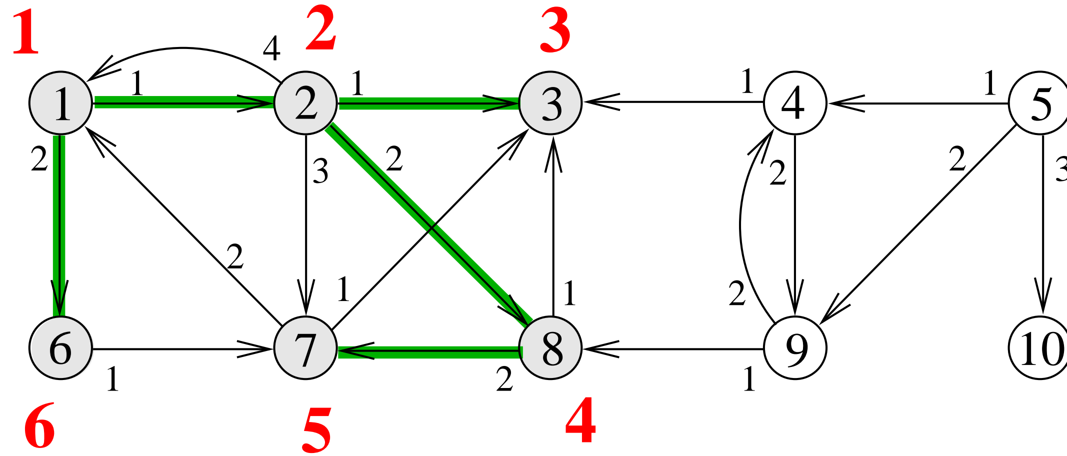
Die (**reflexive** und) **transitive Hülle** eines Digraphen G ist der Digraph **TH**$(G) := (V, E^*)$ mit Kantenmenge

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

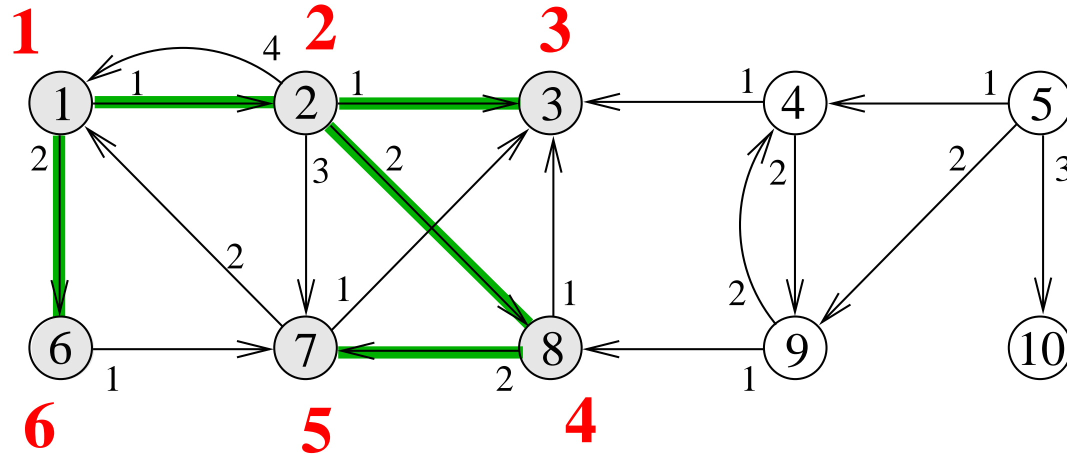
Tiefensuche in G, gestartet von jedem Knoten v nacheinander, ermittelt alle Mengen R_v, $v \in V$, also **TH**$(G) = (V, E^*)$, in Zeit

$$O\Big(\sum_{v \in V} |E_{R_v}|\Big) = O(|E^*|) = O(|V| \cdot |E|).$$

(Nach jedem Aufruf „dfs(v)“ wird mit Hilfe von R_v in Zeit $O(|R_v|)$ das `status`-Array auf „**neu**“ zurückgesetzt.)

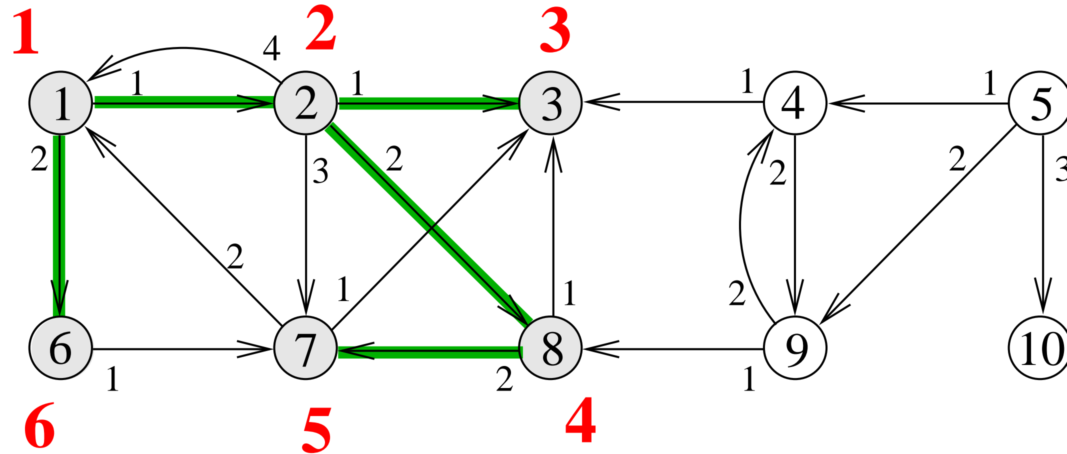


Beobachtung 2:



Beobachtung 2:

Jeder Knoten $v \in R_{v_0} - \{v_0\}$ wird von einem **eindeutig bestimmten** Knoten w aus „entdeckt" (Inspektion der Kante (w, v) zeigt v als **neu**);
dieses w nennen wir $p(v)$, den **Vorgänger** von v.

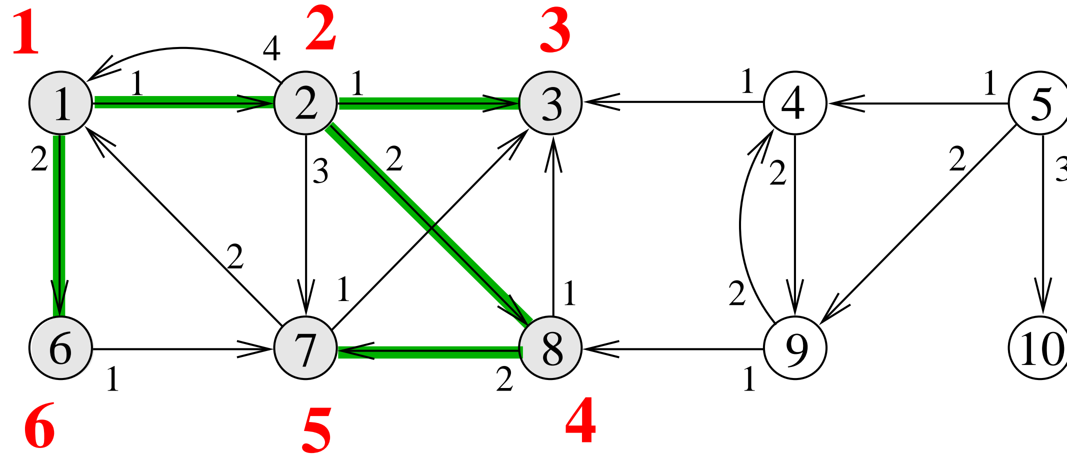


Beobachtung 2:

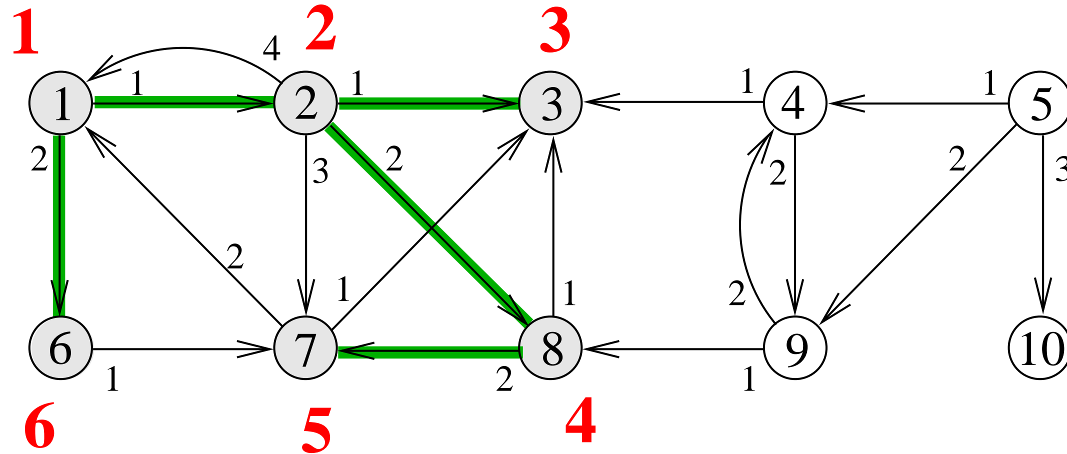
Jeder Knoten $v \in R_{v_0} - \{v_0\}$ wird von einem **eindeutig bestimmten** Knoten w aus „entdeckt“ (Inspektion der Kante (w, v) zeigt v als **neu**); dieses w nennen wir $\boldsymbol{p(v)}$, den **Vorgänger** von v.

Kanten von $p(v)$ zu v: „Entdeckungsrichtung“.

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

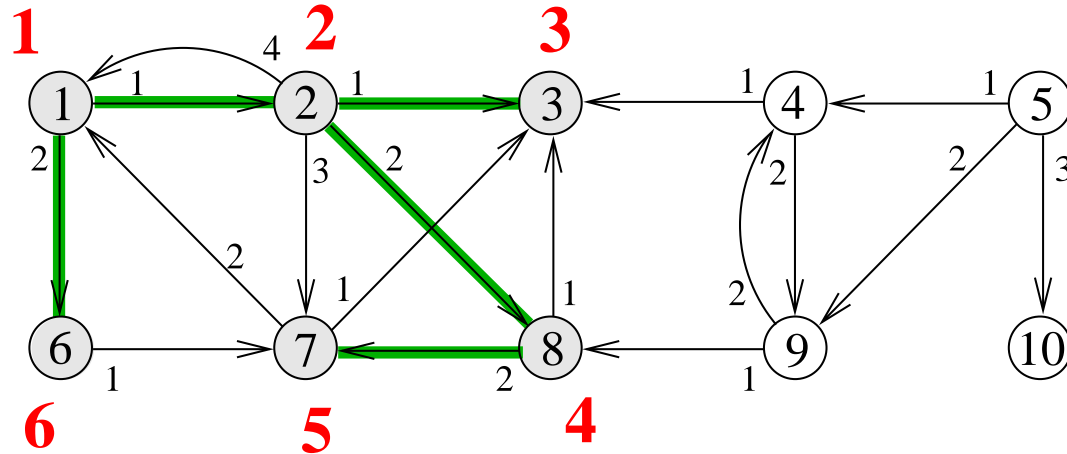


Die Kanten $(p(v), v)$ ($\widehat{=}$ direkter Aufruf dfs(v) aus dfs$(p(v))$) bilden einen **(gerichteten)** **Baum** mit Wurzel v_0 und Knotenmenge R_{v_0}, den



Die Kanten $(p(v), v)$ ($\widehat{=}$ direkter Aufruf dfs(v) aus dfs$(p(v))$) bilden einen **(gerichteten)** **Baum** mit Wurzel v_0 und Knotenmenge R_{v_0}, den

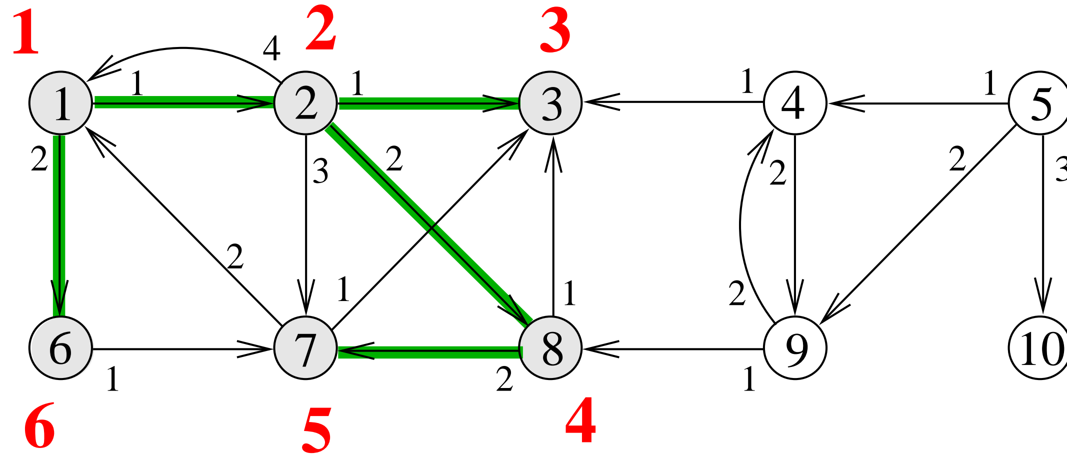
Tiefensuch-Baum/DFS-Baum $T_{\mathsf{dfs}}(v_0)$.



Die Kanten $(p(v), v)$ ($\widehat{=}$ direkter Aufruf dfs(v) aus dfs$(p(v))$) bilden einen **(gerichteten)** **Baum** mit Wurzel v_0 und Knotenmenge R_{v_0}, den

Tiefensuch-Baum/DFS-Baum $T_{\text{dfs}}(v_0)$.

Ein **Weg** im DFS-Baum entspricht einer indirekten Aufrufbeziehung.

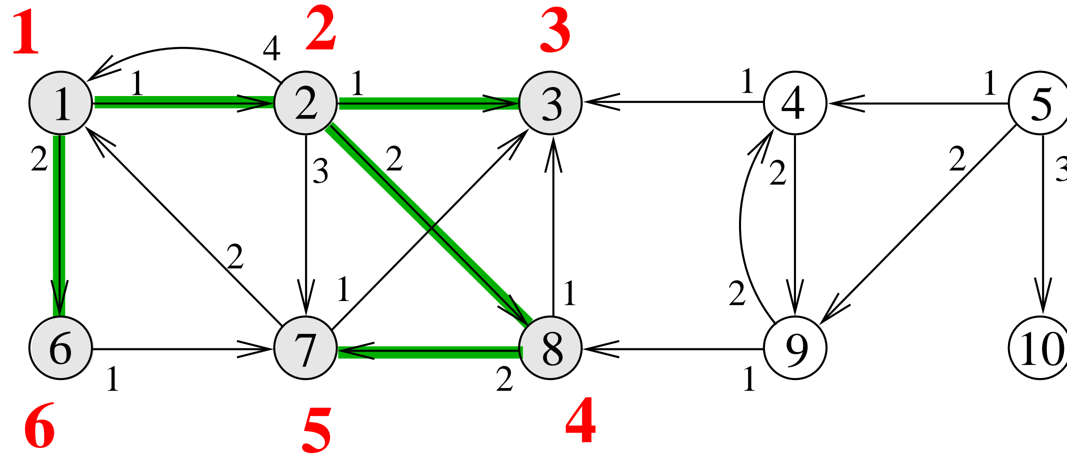


Die Kanten $(p(v), v)$ ($\widehat{=}$ direkter Aufruf dfs(v) aus dfs$(p(v))$) bilden einen **(gerichteten)** **Baum** mit Wurzel v_0 und Knotenmenge R_{v_0}, den

Tiefensuch-Baum/DFS-Baum $T_{\text{dfs}}(v_0)$.

Ein **Weg** im DFS-Baum entspricht einer indirekten Aufrufbeziehung.

Die DFS-Nummern **(rot)** zählen die Knoten in $T_{\text{dfs}}(v_0)$ in **Präorder**-Reihenfolge auf.

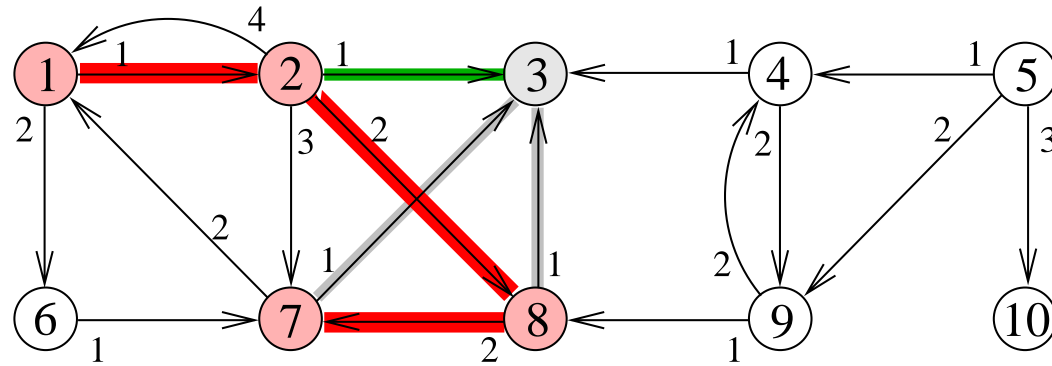


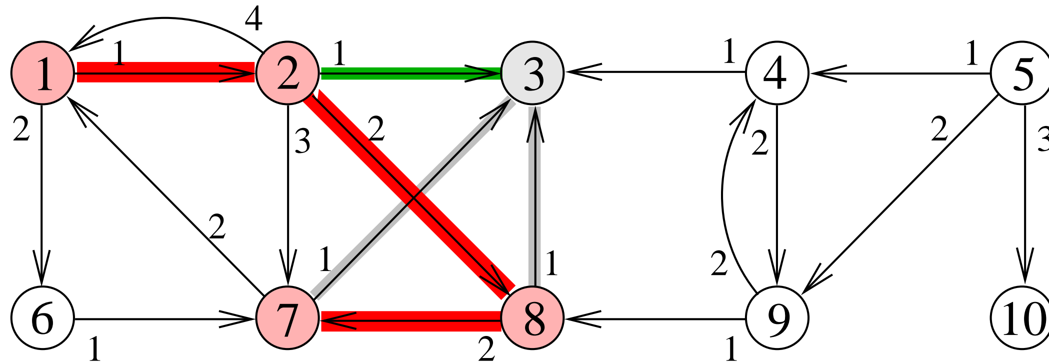
Die Kanten $(p(v), v)$ ($\widehat{=}$ direkter Aufruf dfs(v) aus dfs$(p(v))$) bilden einen **(gerichteten)** **Baum** mit Wurzel v_0 und Knotenmenge R_{v_0}, den

Tiefensuch-Baum/DFS-Baum $T_{\text{dfs}}(v_0)$.

Ein **Weg** im DFS-Baum entspricht einer indirekten Aufrufbeziehung.

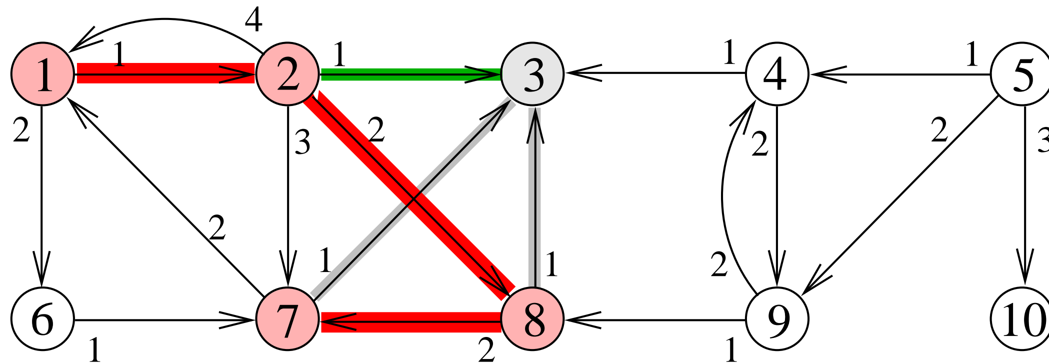
Die DFS-Nummern **(rot)** zählen die Knoten in $T_{\text{dfs}}(v_0)$ in **Präorder**-Reihenfolge auf. Die Anordnung der Kinder eines Knotens wird durch die Anordnung der Nachfolger in den Adjazenzlisten bestimmt.


1
2
3
4
5
6
7
8
9
10



Zu jedem Zeitpunkt bilden die **aktiven** Knoten einen Weg (den **„roten Weg“**) im DFS-Baum.

Startpunkt ist v_0, Endpunkt ist der Knoten v, in dessen „dfs(v)“-Aufruf eben gearbeitet wird. (Hier: Knoten 7.)



Zu jedem Zeitpunkt bilden die **aktiven** Knoten einen Weg (den **„roten Weg“**) im DFS-Baum.

Startpunkt ist v_0, Endpunkt ist der Knoten v, in dessen „dfs(v)“-Aufruf eben gearbeitet wird. (Hier: Knoten 7.)

Die Kanten des roten Weges sind Kanten im DFS-Baum.

PAUSE

Es folgt: Satz vom weißen Weg,
Erforschung des gesamten Digraphen

Wie entscheidet sich, ob w in $T_{\mathrm{dfs}}(v_0)$ Nachfahr von v ist?

Wie entscheidet sich, ob w in $T_{\mathrm{dfs}}(v_0)$ Nachfahr von v ist? D.h.: Wie kann man sehen, ob dfs(w) aufgerufen wird, während v **aktiv** ist?

Wie entscheidet sich, ob w in $T_{\text{dfs}}(v_0)$ Nachfahr von v ist? D.h.: Wie kann man sehen, ob dfs(w) aufgerufen wird, während v **aktiv** ist?

Satz 8.1.3 („Satz vom weißen Weg“)

Wie entscheidet sich, ob w in $T_{\text{dfs}}(v_0)$ Nachfahr von v ist? D.h.: Wie kann man sehen, ob dfs(w) aufgerufen wird, während v **aktiv** ist?

Satz 8.1.3 („Satz vom weißen Weg")

w ist ein Nachfahr von v im Tiefensuchbaum $T_{\text{dfs}}(v_0)$ $\Leftrightarrow$

Wie entscheidet sich, ob w in $T_{\mathrm{dfs}}(v_0)$ Nachfahr von v ist? D.h.: Wie kann man sehen, ob dfs(w) aufgerufen wird, während v **aktiv** ist?

Satz 8.1.3 („Satz vom weißen Weg“)

w ist ein Nachfahr von v im Tiefensuchbaum $T_{\mathrm{dfs}}(v_0)$ $\Leftrightarrow$

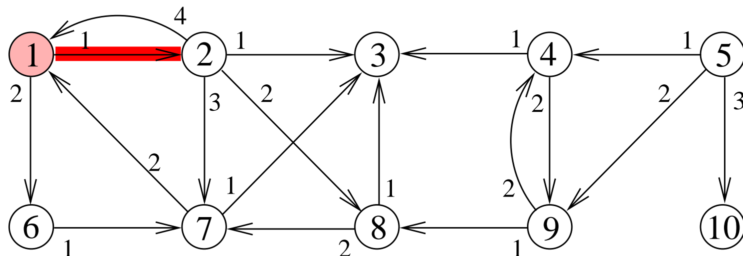
in dem Moment, in dem dfs(v) aufgerufen wird, existiert ein Weg von v nach w aus Knoten, die alle **neu** („weiß“) sind.

Wie entscheidet sich, ob w in $T_{\text{dfs}}(v_0)$ Nachfahr von v ist? D.h.: Wie kann man sehen, ob dfs(w) aufgerufen wird, während v **aktiv** ist?

Satz 8.1.3 („Satz vom weißen Weg“)

w ist ein Nachfahr von v im Tiefensuchbaum $T_{\text{dfs}}(v_0)$ $\Leftrightarrow$

in dem Moment, in dem dfs(v) aufgerufen wird, existiert ein Weg von v nach w aus Knoten, die alle **neu** („weiß“) sind.

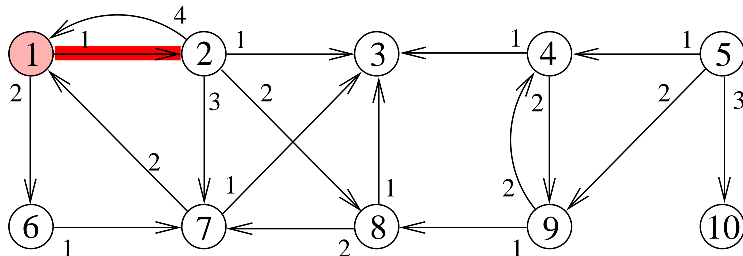


Wie entscheidet sich, ob w in $T_{\mathrm{dfs}}(v_0)$ Nachfahr von v ist? D.h.: Wie kann man sehen, ob dfs(w) aufgerufen wird, während v **aktiv** ist?

Satz 8.1.3 („Satz vom weißen Weg“)

w ist ein Nachfahr von v im Tiefensuchbaum $T_{\mathrm{dfs}}(v_0)$ $\Leftrightarrow$

in dem Moment, in dem dfs(v) aufgerufen wird, existiert ein Weg von v nach w aus Knoten, die alle **neu** („weiß“) sind.



Beispiel: dfs(2) startet.

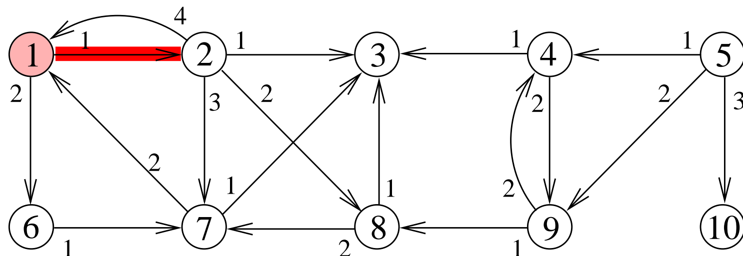
Knoten 3, 7, 8 sind auf „weißen Wegen“ von 2 aus erreichbar

Wie entscheidet sich, ob w in $T_{\text{dfs}}(v_0)$ Nachfahr von v ist? D.h.: Wie kann man sehen, ob dfs(w) aufgerufen wird, während v **aktiv** ist?

Satz 8.1.3 („Satz vom weißen Weg“)

w ist ein Nachfahr von v im Tiefensuchbaum $T_{\text{dfs}}(v_0)$ $\Leftrightarrow$

in dem Moment, in dem dfs(v) aufgerufen wird, existiert ein Weg von v nach w aus Knoten, die alle **neu** („weiß“) sind.



Beispiel: dfs(2) startet.

Knoten 3, 7, 8 sind auf „weißen Wegen“ von 2 aus erreichbar

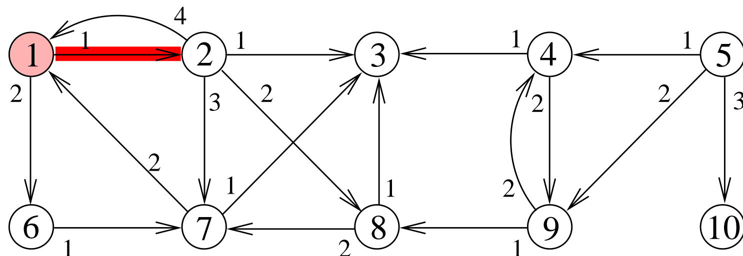
$\Rightarrow$ diese Knoten sind im DFS-Baum Nachfahren von 2.

Wie entscheidet sich, ob w in $T_{\text{dfs}}(v_0)$ Nachfahr von v ist? D.h.: Wie kann man sehen, ob dfs(w) aufgerufen wird, während v **aktiv** ist?

Satz 8.1.3 („Satz vom weißen Weg")

w ist ein Nachfahr von v im Tiefensuchbaum $T_{\text{dfs}}(v_0)$ $\Leftrightarrow$

in dem Moment, in dem dfs(v) aufgerufen wird, existiert ein Weg von v nach w aus Knoten, die alle **neu** („weiß") sind.



Beispiel: dfs(2) startet.

Knoten 3, 7, 8 sind auf „weißen Wegen" von 2 aus erreichbar

$\Rightarrow$ diese Knoten sind im DFS-Baum Nachfahren von 2.

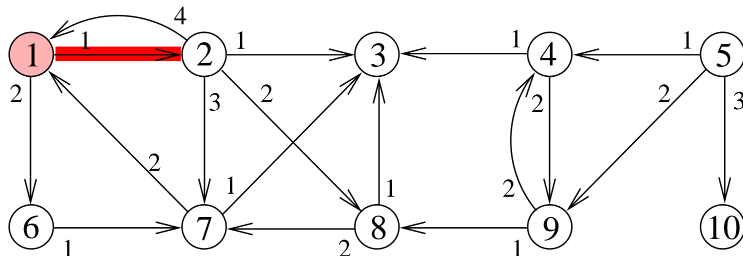
Knoten 6 ist von 2 aus erreichbar, aber nicht auf einem „weißen Weg"

Wie entscheidet sich, ob w in $T_{\mathrm{dfs}}(v_0)$ Nachfahr von v ist? D.h.: Wie kann man sehen, ob dfs(w) aufgerufen wird, während v **aktiv** ist?

Satz 8.1.3 („Satz vom weißen Weg“)

w ist ein Nachfahr von v im Tiefensuchbaum $T_{\mathrm{dfs}}(v_0)$ $\Leftrightarrow$

in dem Moment, in dem dfs(v) aufgerufen wird, existiert ein Weg von v nach w aus Knoten, die alle **neu** („weiß“) sind.



Beispiel: dfs(2) startet.

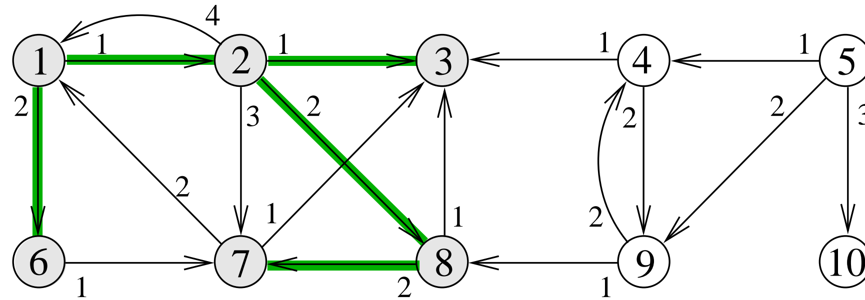
Knoten 3, 7, 8 sind auf „weißen Wegen“ von 2 aus erreichbar

$\Rightarrow$ diese Knoten sind im DFS-Baum Nachfahren von 2.

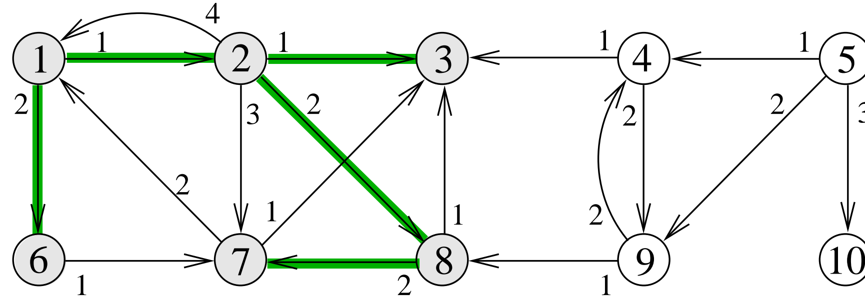
Knoten 6 ist von 2 aus erreichbar, aber nicht auf einem „weißen Weg“

$\Rightarrow$ Knoten 6 ist im DFS-Baum nicht Nachfahr von 2.

Beweis von Satz 8.1.3: „⇒“:

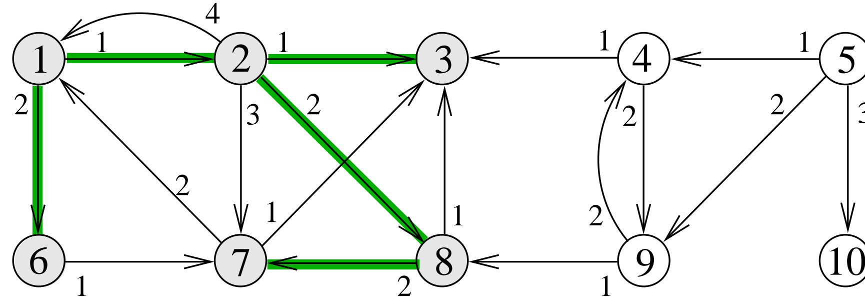


Beweis von Satz 8.1.3: „⇒“:



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

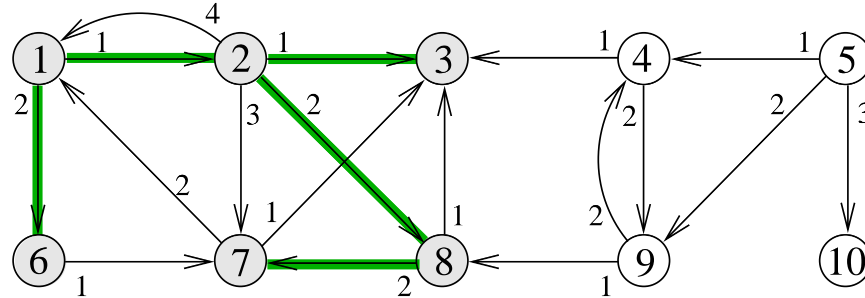
Beweis von Satz 8.1.3: „⇒“:



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

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

Beweis von Satz 8.1.3: „⇒“:

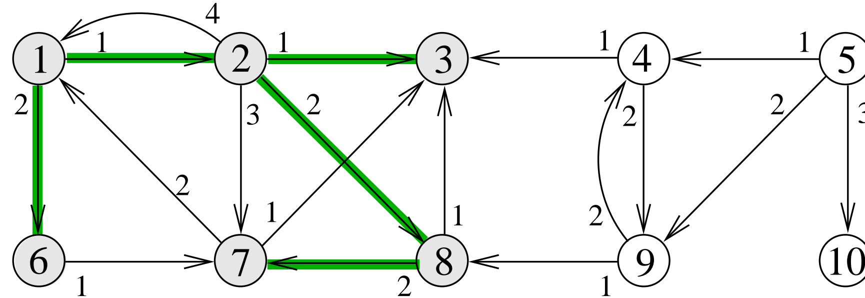


Sei w in $T_{\mathrm{dfs}}(v_0)$ Nachfahr von v.

Betrachte 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 im Bild: Weg $(2, 8, 7)$.)

Beweis von Satz 8.1.3: „⇒“:



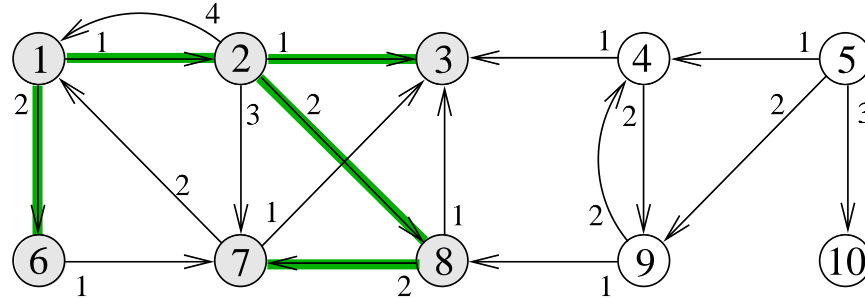
Sei w in $T_{\text{dfs}}(v_0)$ Nachfahr von v.

Betrachte den Weg, der in $T_{\text{dfs}}(v_0)$ von v nach w führt: $(v = u_0, u_1, u_2, \ldots, u_t = w)$.

(Beispiel im Bild: Weg $(2, 8, 7)$.)

Nach Beobachtung 2 wird erst dfs(v) aufgerufen, dann dfs(u_1), dann dfs(u_2), usw.;

Beweis von Satz 8.1.3: „⇒“:



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

Betrachte den Weg, der in $T_{\rm dfs}(v_0)$ von v nach w führt: $(v = u_0, u_1, u_2, \ldots, u_t = w)$.

(Beispiel im Bild: Weg $(2, 8, 7)$.)

Nach Beobachtung 2 wird erst dfs(v) aufgerufen, dann dfs(u_1), dann dfs(u_2), usw.;

also sind in dem Moment, in dem der Aufruf dfs(v) erfolgt,
alle Knoten $v = u_0, u_1, u_2, \ldots, u_t = w$ noch **neu** .

Beweis von Satz 8.1.3: „$\Leftarrow$“:

Beweis von Satz 8.1.3: „$\Leftarrow$":

Sei $(v = u_0, u_1, u_2, \ldots, u_t = w)$, mit $t > 0$, ein Weg aus Knoten, die im Moment des Aufrufs dfs(v) alle **neu** sind.

Beweis von Satz 8.1.3: „$\Leftarrow$“:

Sei $(v = u_0, u_1, u_2, \ldots, u_t = w)$, mit $t > 0$, ein Weg aus Knoten, die im Moment des Aufrufs dfs(v) alle **neu** sind.

Durch Induktion über i zeigen wir, dass für jedes i, $0 \le i \le t$, der Aufruf dfs(u_i) erfolgt, bevor dfs(v) endet.

Beweis von Satz 8.1.3: „$\Leftarrow$“:

Sei $(v = u_0, u_1, u_2, \ldots, u_t = w)$, mit $t > 0$, ein Weg aus Knoten, die im Moment des Aufrufs dfs(v) alle **neu** sind.

Durch Induktion über i zeigen wir, dass für jedes i, $0 \leq i \leq t$, der Aufruf dfs(u_i) erfolgt, bevor dfs(v) endet.

I.A.: Die Beh. ist für $i = 0$ klar.

Beweis von Satz 8.1.3: „$\Leftarrow$“:

Sei $(v = u_0, u_1, u_2, \ldots, u_t = w)$, mit $t > 0$, ein Weg aus Knoten, die im Moment des Aufrufs dfs(v) alle **neu** sind.

Durch Induktion über i zeigen wir, dass für jedes i, $0 \leq i \leq t$, der Aufruf dfs(u_i) erfolgt, bevor dfs(v) endet.

I.A.: Die Beh. ist für $i = 0$ klar.

I.Schritt: Sei nun $1 \leq i \leq t$.

Beweis von Satz 8.1.3: „$\Leftarrow$“:

Sei $(v = u_0, u_1, u_2, \ldots, u_t = w)$, mit $t > 0$, ein Weg aus Knoten, die im Moment des Aufrufs dfs(v) alle **neu** sind.

Durch Induktion über i zeigen wir, dass für jedes i, $0 \leq i \leq t$, der Aufruf dfs(u_i) erfolgt, bevor dfs(v) endet.

I.A.: Die Beh. ist für $i = 0$ klar.

I.Schritt: Sei nun $1 \leq i \leq t$.

Nach I.V. beginnt dfs(u_{i-1}), bevor dfs(v) endet.

Beweis von Satz 8.1.3: „$\Leftarrow$“:

Sei $(v = u_0, u_1, u_2, \ldots, u_t = w)$, mit $t > 0$, ein Weg aus Knoten, die im Moment des Aufrufs dfs(v) alle **neu** sind.

Durch Induktion über i zeigen wir, dass für jedes i, $0 \le i \le t$, der Aufruf dfs(u_i) erfolgt, bevor dfs(v) endet.

I.A.: Die Beh. ist für $i = 0$ klar.

I.Schritt: Sei nun $1 \le i \le t$.

Nach I.V. beginnt dfs(u_{i-1}), bevor dfs(v) endet.

Während des Aufrufs dfs(u_{i-1}) wird die Kante (u_{i-1}, u_i) inspiziert.

Beweis von Satz 8.1.3: „$\Leftarrow$“:

Sei $(v = u_0, u_1, u_2, \ldots, u_t = w)$, mit $t > 0$, ein Weg aus Knoten, die im Moment des Aufrufs dfs(v) alle **neu** sind.

Durch Induktion über i zeigen wir, dass für jedes i, $0 \le i \le t$, der Aufruf dfs(u_i) erfolgt, bevor dfs(v) endet.

I.A.: Die Beh. ist für $i = 0$ klar.

I.Schritt: Sei nun $1 \le i \le t$.

Nach I.V. beginnt dfs(u_{i-1}), bevor dfs(v) endet.

Während des Aufrufs dfs(u_{i-1}) wird die Kante (u_{i-1}, u_i) inspiziert.

1. Fall: In diesem Moment ist u_i noch **neu**. Dann erfolgt der Aufruf dfs(u_i) jetzt.

Beweis von Satz 8.1.3: „$\Leftarrow$“:

Sei $(v = u_0, u_1, u_2, \ldots, u_t = w)$, mit $t > 0$, ein Weg aus Knoten, die im Moment des Aufrufs dfs(v) alle **neu** sind.

Durch Induktion über i zeigen wir, dass für jedes i, $0 \le i \le t$, der Aufruf dfs(u_i) erfolgt, bevor dfs(v) endet.

I.A.: Die Beh. ist für $i = 0$ klar.

I.Schritt: Sei nun $1 \le i \le t$.

Nach I.V. beginnt dfs(u_{i-1}), bevor dfs(v) endet.

Während des Aufrufs dfs(u_{i-1}) wird die Kante (u_{i-1}, u_i) inspiziert.

1. Fall: In diesem Moment ist u_i noch **neu**. Dann erfolgt der Aufruf dfs(u_i) jetzt.

2. Fall: In diesem Moment ist u_i **aktiv** oder **fertig**. Dann ist der Aufruf dfs(u_i) schon vorher passiert.

Beweis von Satz 8.1.3: „$\Leftarrow$“:

Sei $(v = u_0, u_1, u_2, \ldots, u_t = w)$, mit $t > 0$, ein Weg aus Knoten, die im Moment des Aufrufs dfs(v) alle **neu** sind.

Durch Induktion über i zeigen wir, dass für jedes i, $0 \le i \le t$, der Aufruf dfs(u_i) erfolgt, bevor dfs(v) endet.

I.A.: Die Beh. ist für $i = 0$ klar.

I.Schritt: Sei nun $1 \le i \le t$.

Nach I.V. beginnt dfs(u_{i-1}), bevor dfs(v) endet.

Während des Aufrufs dfs(u_{i-1}) wird die Kante (u_{i-1}, u_i) inspiziert.

1. Fall: In diesem Moment ist u_i noch **neu**. Dann erfolgt der Aufruf dfs(u_i) jetzt.

2. Fall: In diesem Moment ist u_i **aktiv** oder **fertig**. Dann ist der Aufruf dfs(u_i) schon vorher passiert.

Demnach wird dfs(w) (direkt oder indirekt) aus dfs(v) aufgerufen,
also ist w Nachfahr von v im DFS-Baum. $\square$

Globale Tiefensuche in Digraphen: Alle Knoten und Kanten werden besucht.

Globale Tiefensuche in Digraphen: Alle Knoten und Kanten werden besucht.

Algorithmus DFS(G) // Tiefensuche in $G = (V, E)$ mit $V = \{1, \ldots, n\}$

Globale Tiefensuche in Digraphen: Alle Knoten und Kanten werden besucht.

Algorithmus DFS(G) // Tiefensuche in $G = (V, E)$ mit $V = \{1, \ldots, n\}$

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

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

Globale Tiefensuche in Digraphen: Alle Knoten und Kanten werden besucht.

Algorithmus DFS(G) // Tiefensuche in $G = (V, E)$ mit $V = \{1, \ldots, n\}$

(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**

Globale Tiefensuche in Digraphen: Alle Knoten und Kanten werden besucht.

Algorithmus DFS(G) // Tiefensuche in $G = (V, E)$ mit $V = \{1, \ldots, n\}$

(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**

Globale Tiefensuche in Digraphen: Alle Knoten und Kanten werden besucht.

Algorithmus DFS(G) // Tiefensuche in $G = (V, E)$ mit $V = \{1, \ldots, n\}$

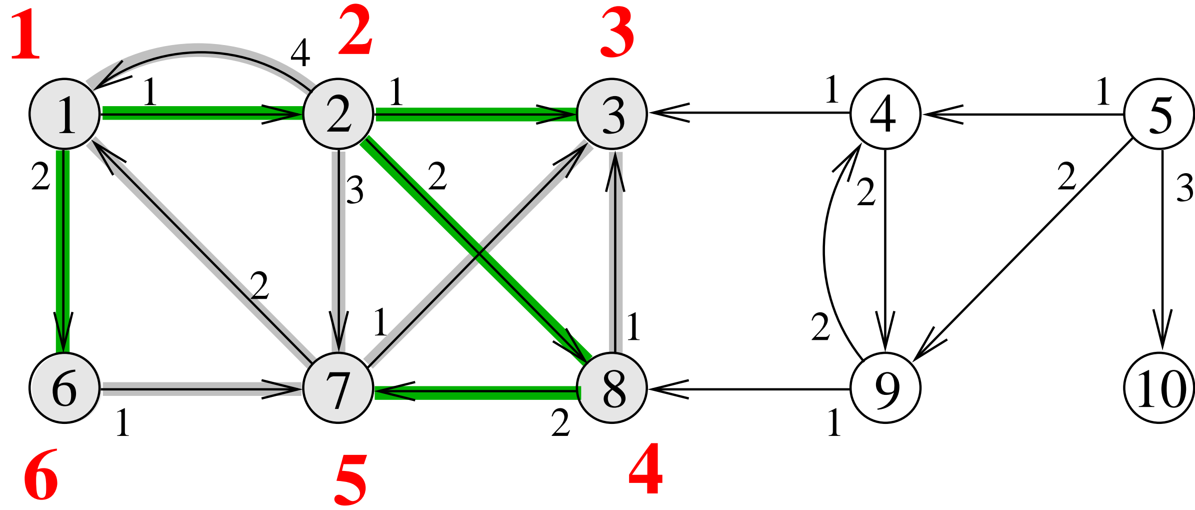
(1) `dfs_count` $\leftarrow 0$;
(2) **for** `v` **from** 1 **to** n **do** `status[v]` $\leftarrow$ **neu** ;
(3) **for** `v` **from** 1 **to** n **do**
(4) **if** `status[v]` $=$ **neu** **then**
(5) dfs(`v`); // starte Tiefensuche von `v` aus

Globale Tiefensuche in Digraphen: Alle Knoten und Kanten werden besucht.

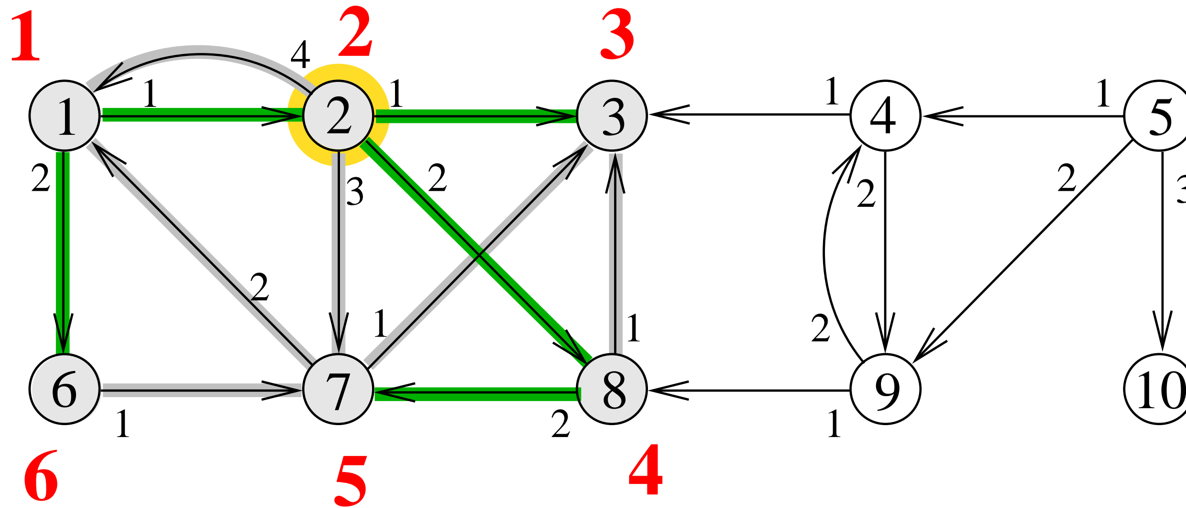
Algorithmus DFS(G) // Tiefensuche in $G = (V, E)$ mit $V = \{1, \ldots, n\}$

(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

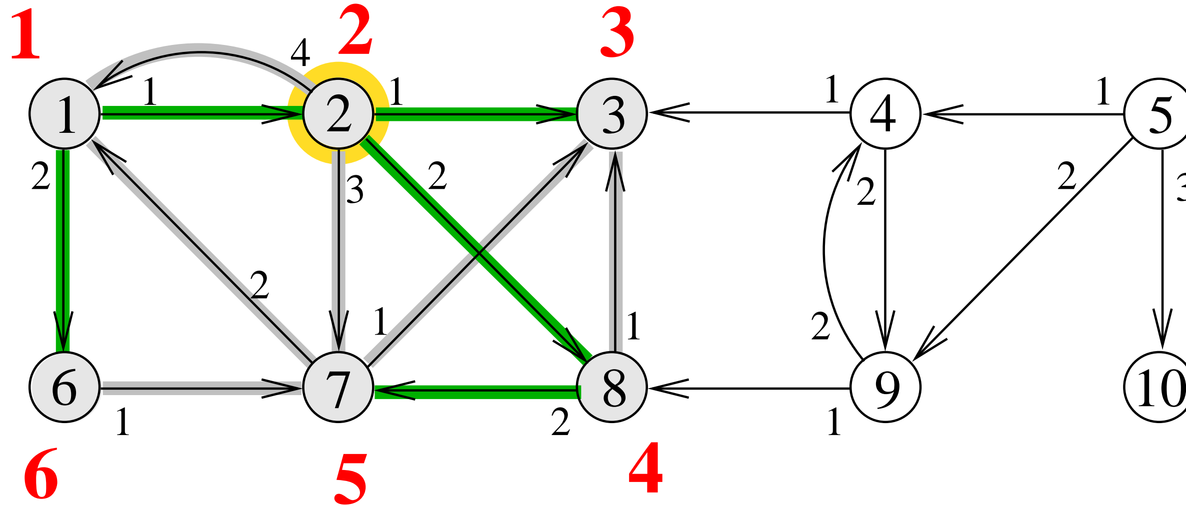
Zu Zeilen (2), (4): Die `status`$[v]$-Werte werden von den dfs-Aufrufen in Zeile (5) verändert;
sie haben also beim Wieder-Lesen in der zweiten Schleife nicht unbedingt immer noch den Initialisierungs-Wert **neu**.



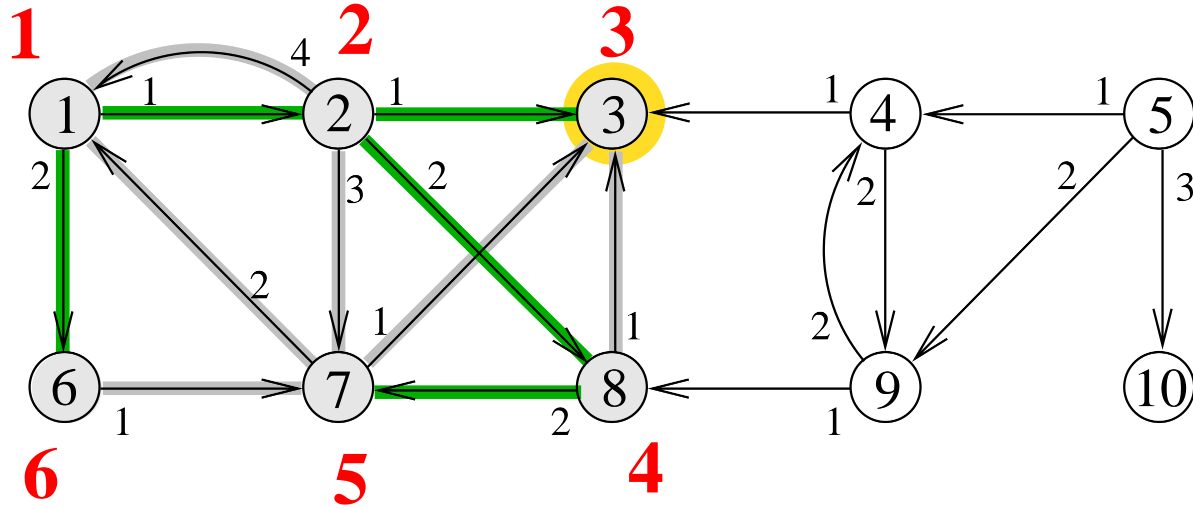
Nach dfs(1).



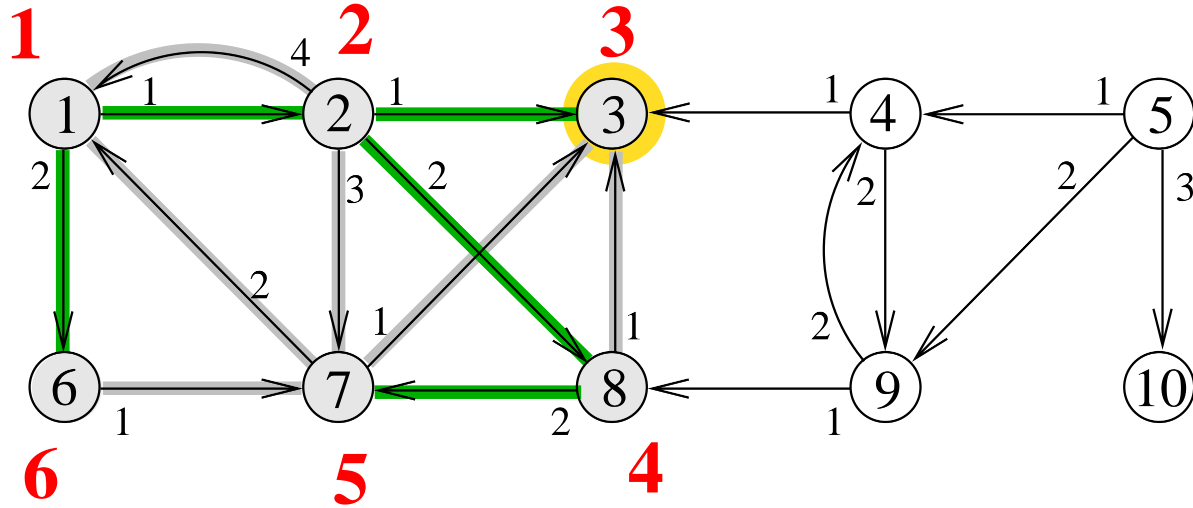
Ist Knoten 2 **neu** ?



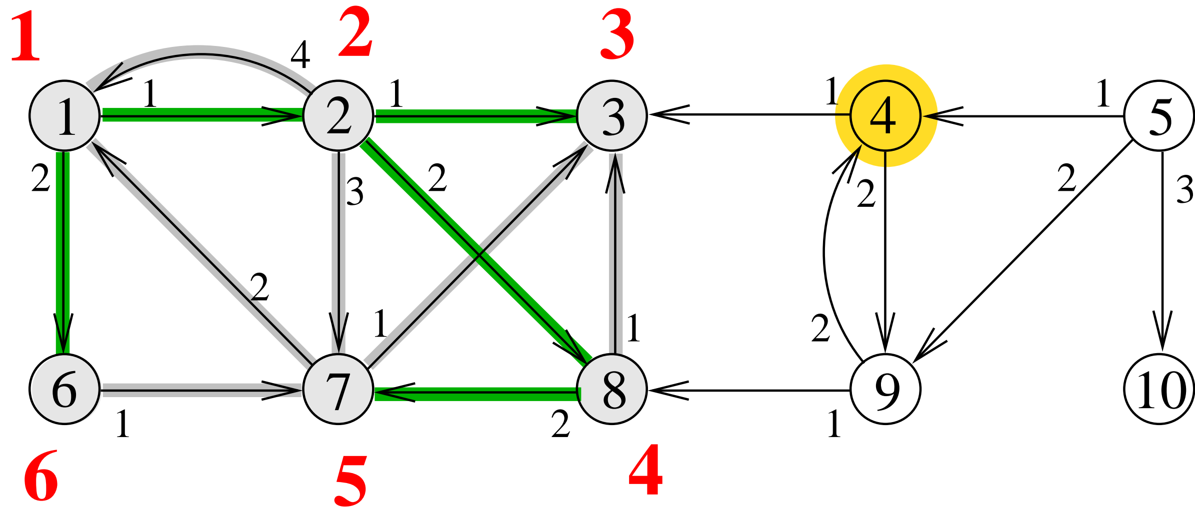
Ist Knoten 2 **neu** ? Nein.



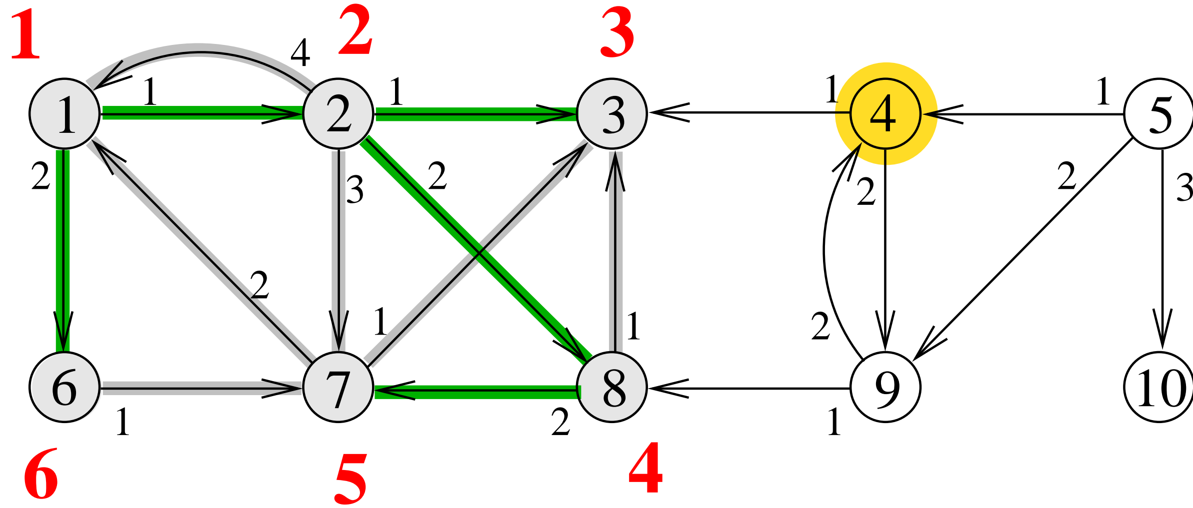
Ist Knoten **3** **neu** ?



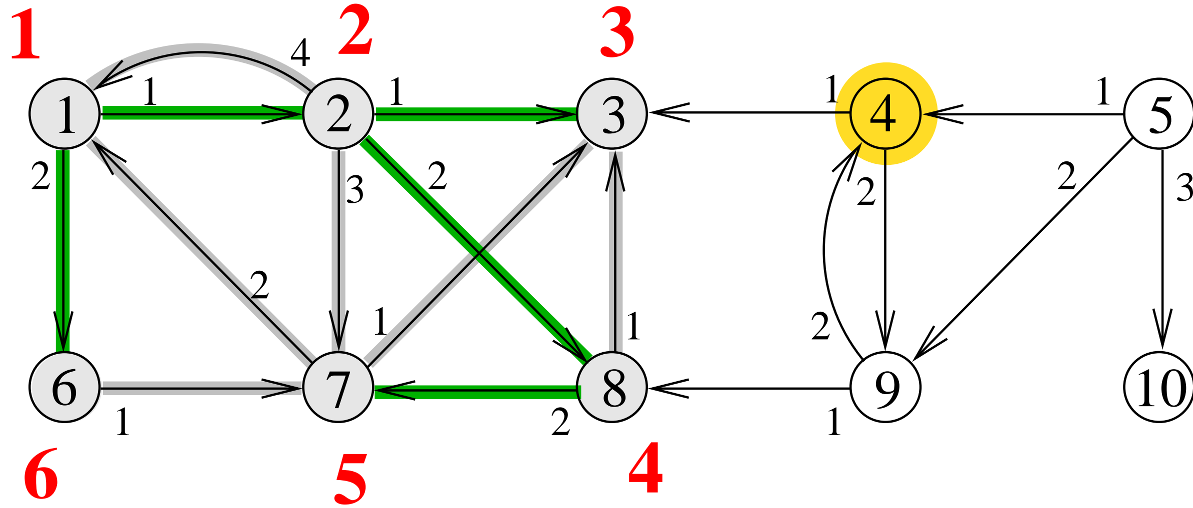
Ist Knoten 3 **neu** ? Nein.



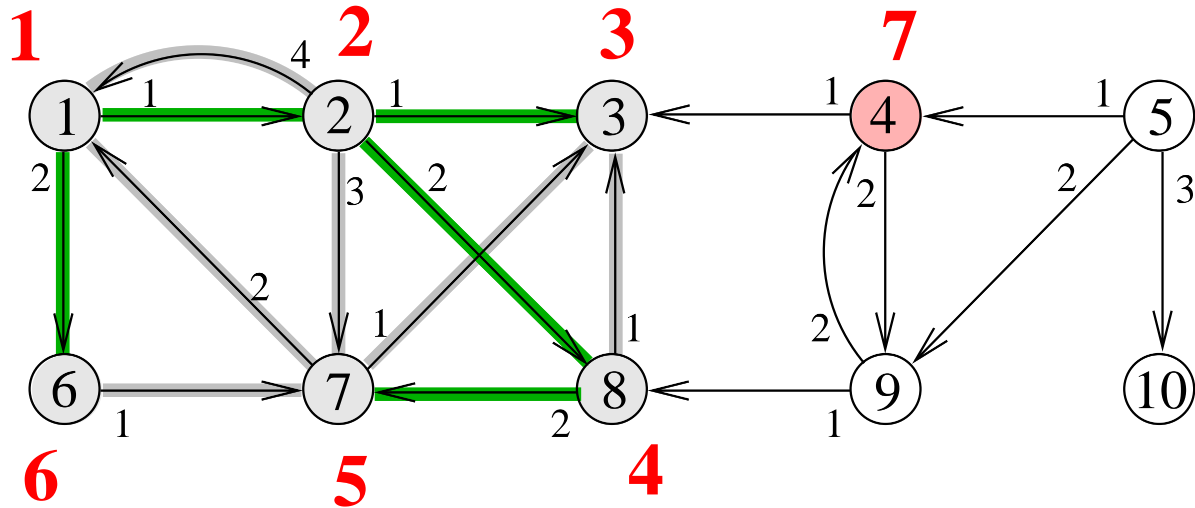
Ist Knoten 4 **neu** ?



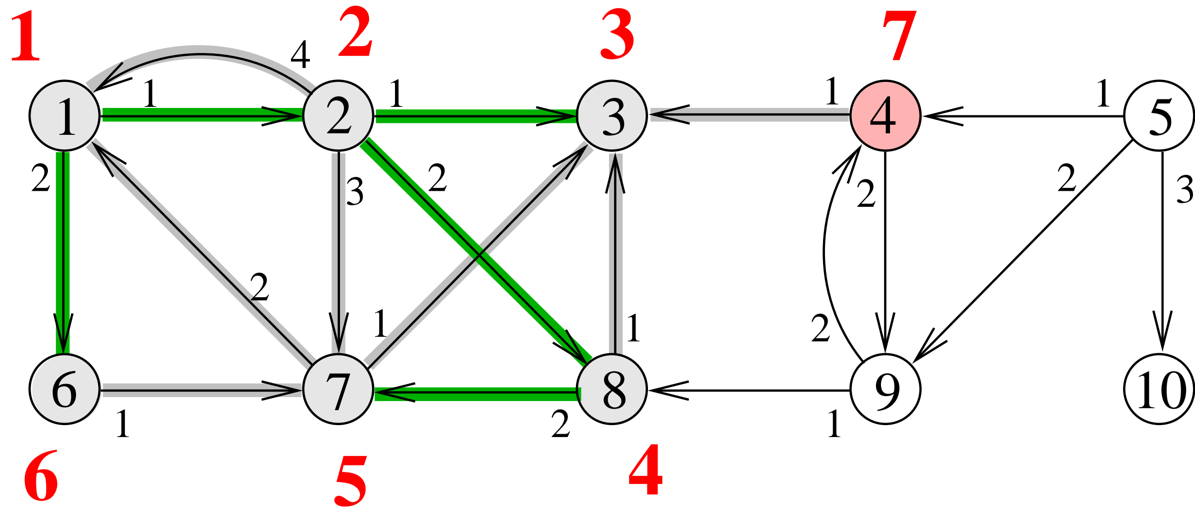
Ist Knoten 4 **neu** ? Ja.



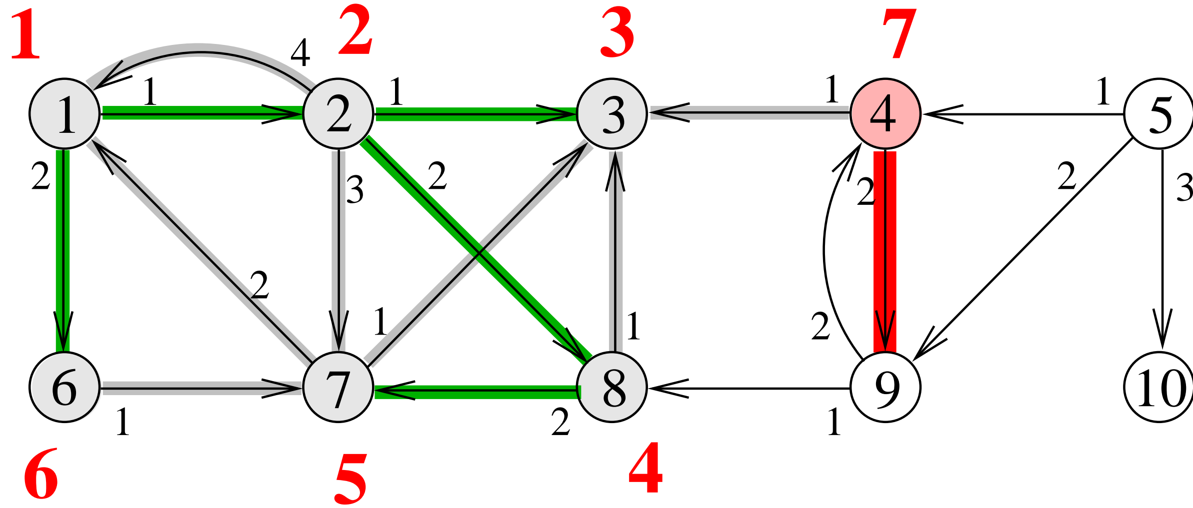
Ist Knoten 4 **neu** ? Ja. Aufruf dfs(4) erfolgt.



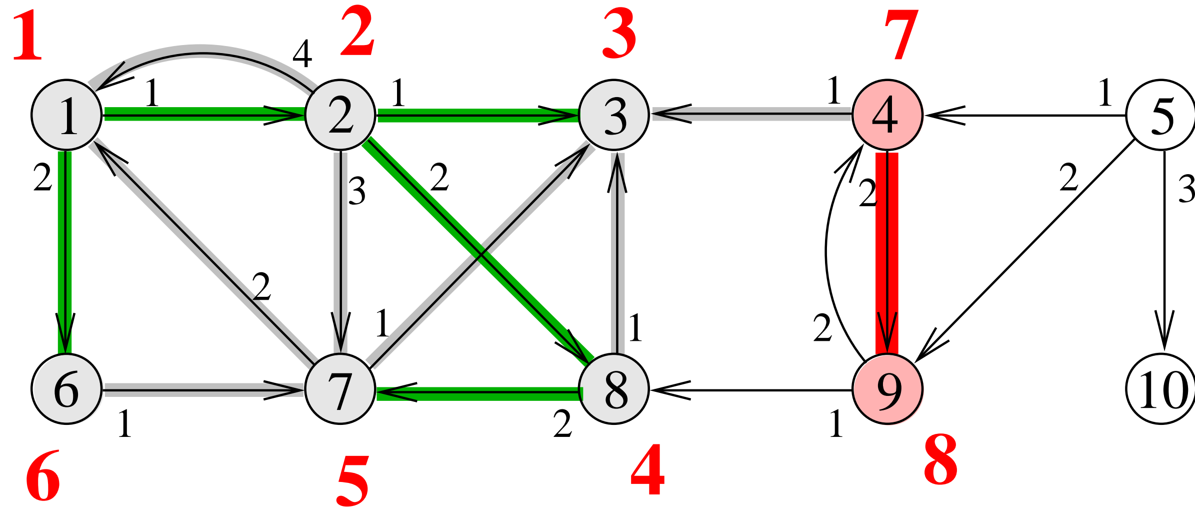
„Roter Weg“: (4)



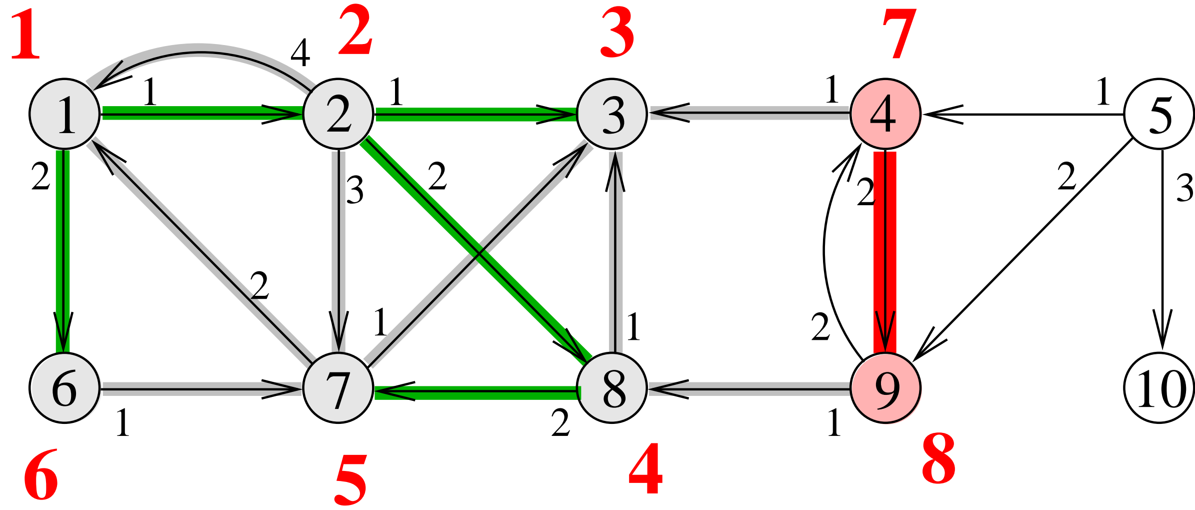
„Roter Weg“: (4)



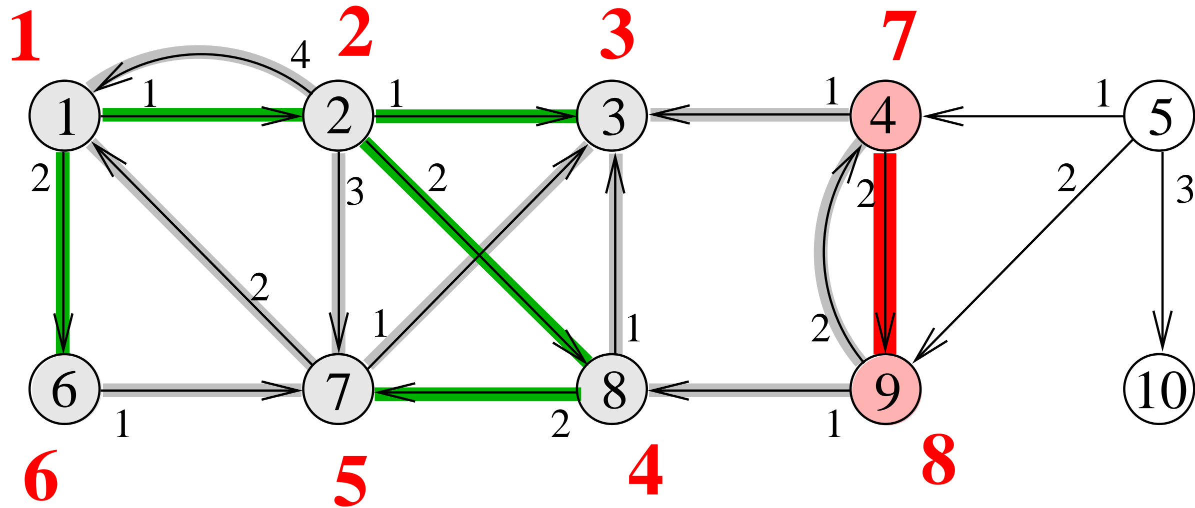
„Roter Weg“: (4)



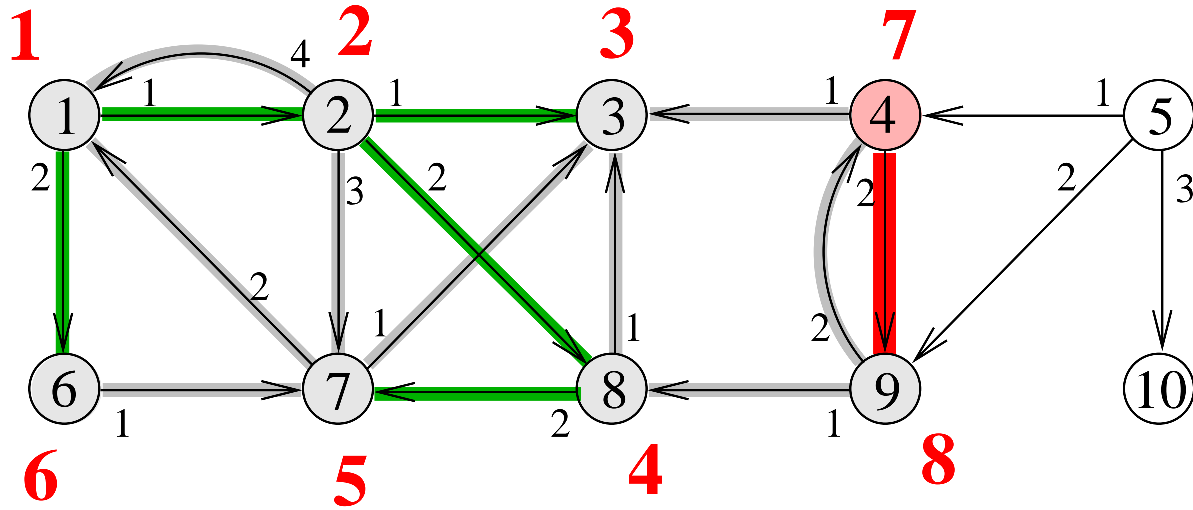
„Roter Weg“: $\mathbf{(4,9)}$



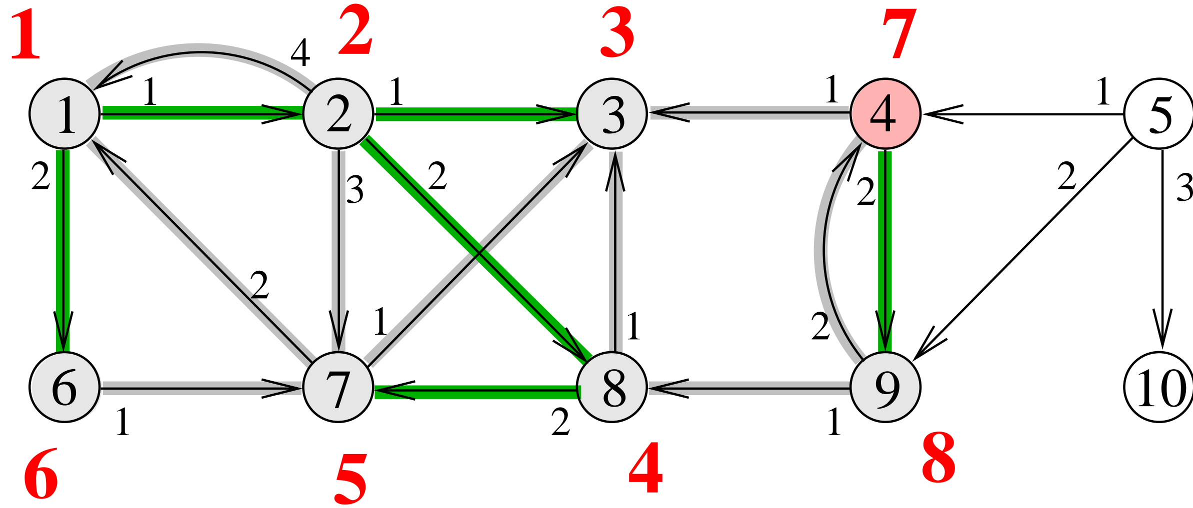
„Roter Weg“: $\mathbf{(4, 9)}$



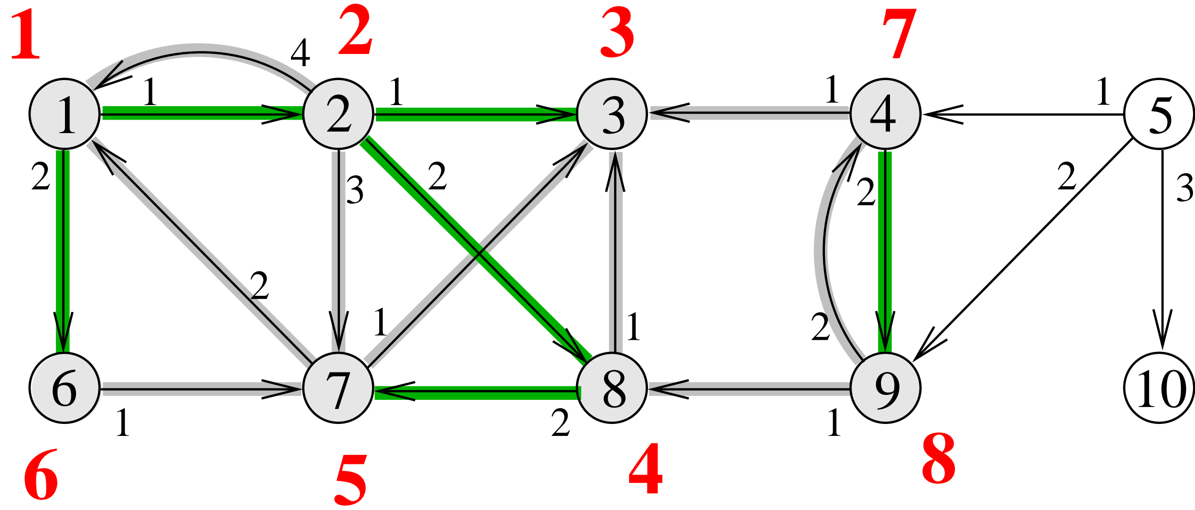
„Roter Weg“: $\mathbf{(4, 9)}$



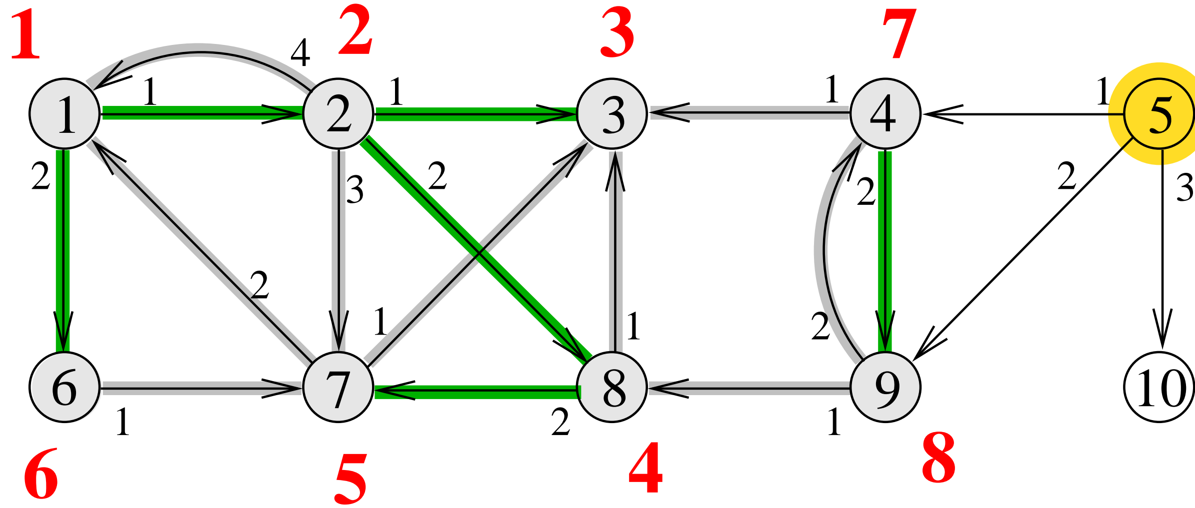
„Roter Weg“: (4)



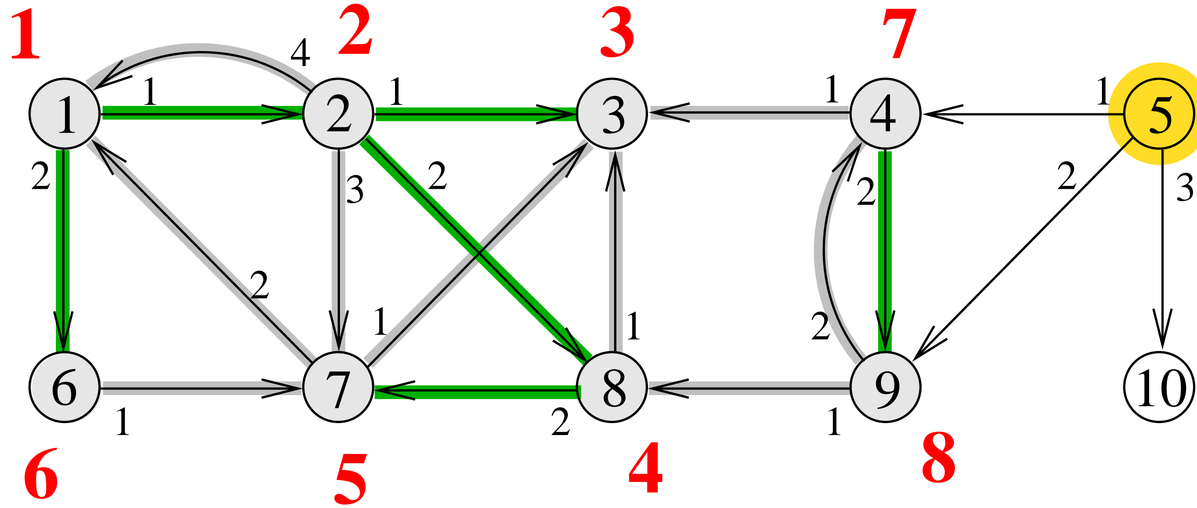
„Roter Weg“: (4)



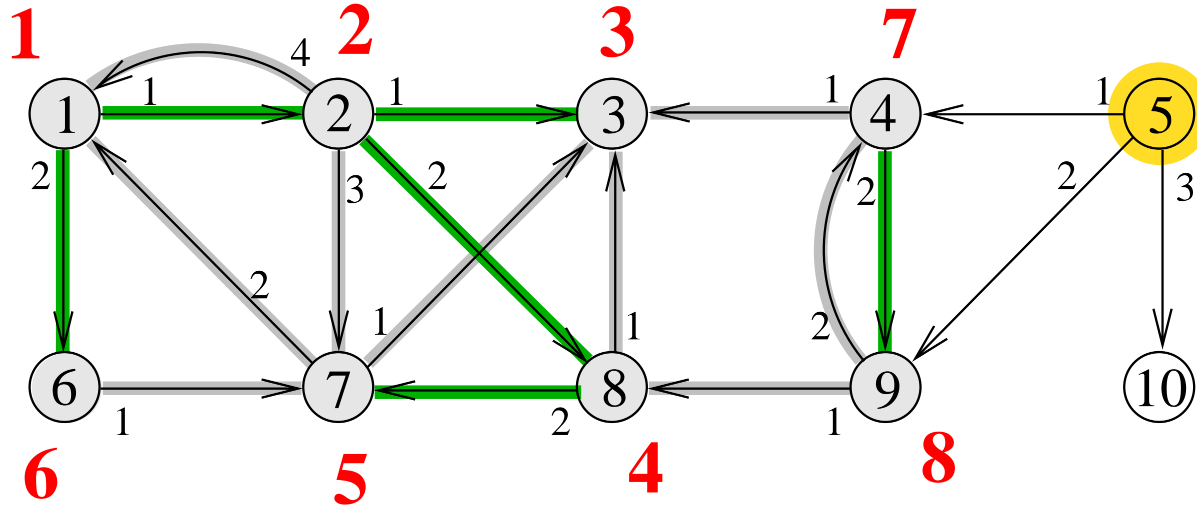
„Roter Weg“: leer.



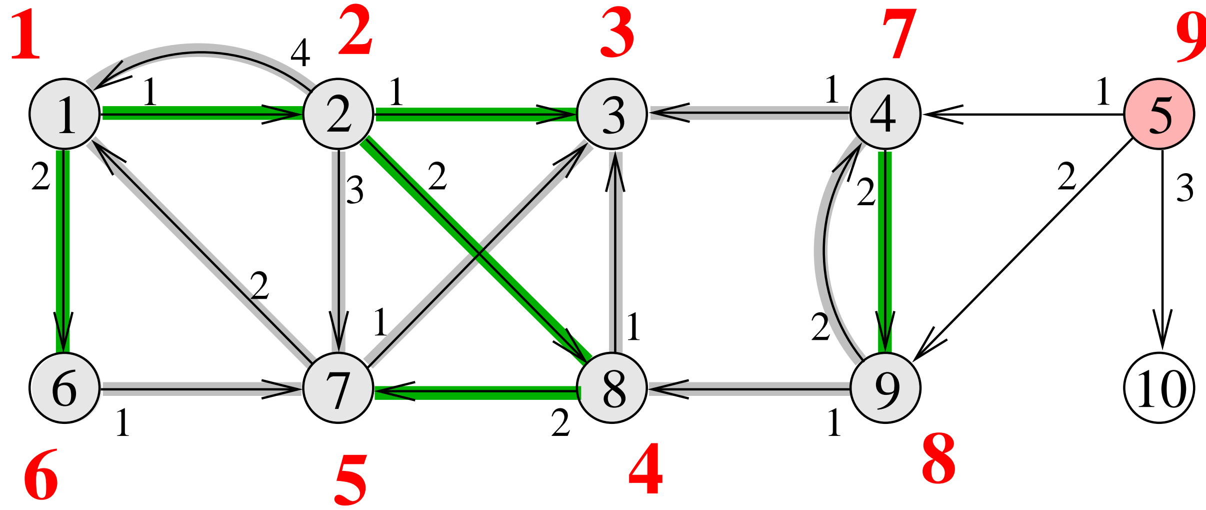
Ist Knoten 5 **neu** ?



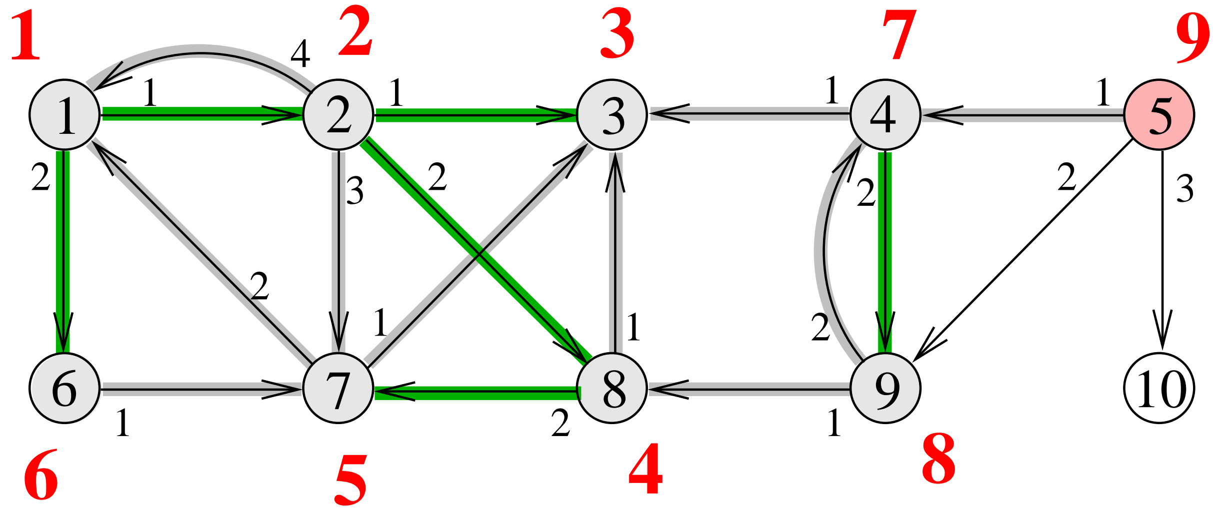
Ist Knoten 5 **neu** ? Ja.



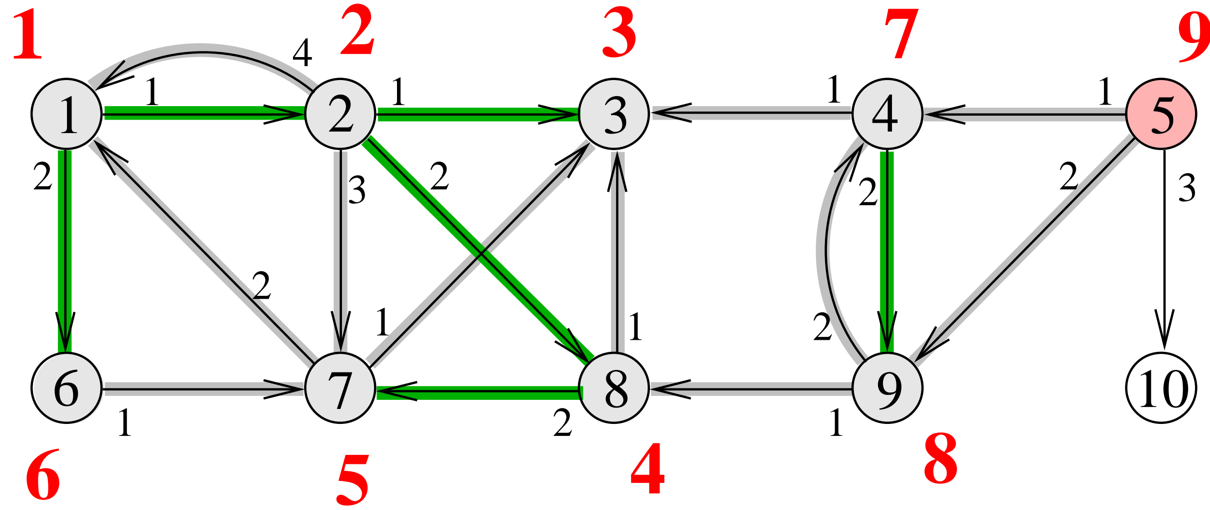
Ist Knoten $\mathbf{5}$ **neu** ? Ja. Aufruf dfs(5) erfolgt.



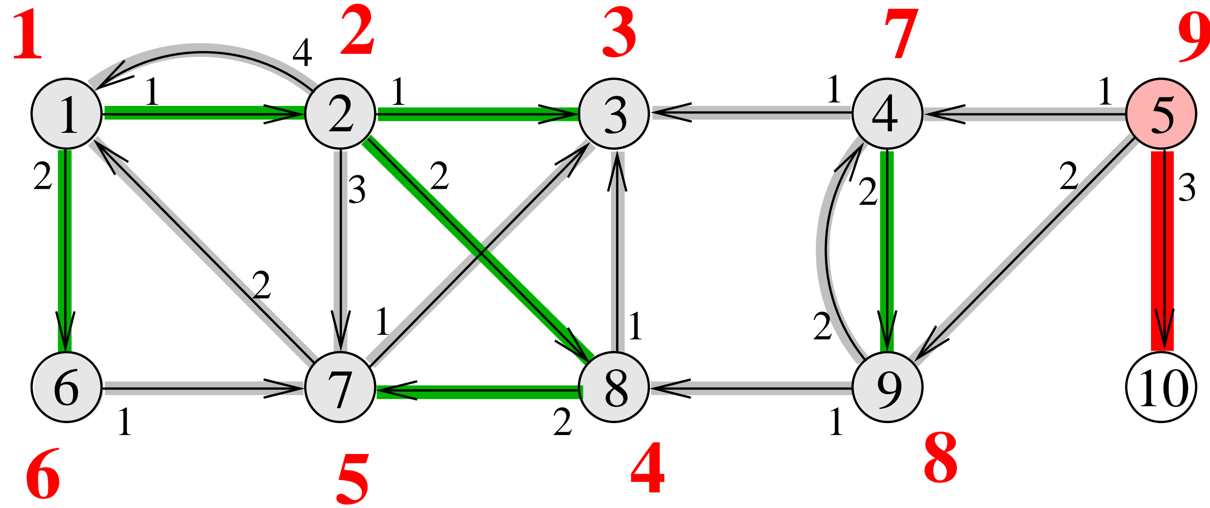
„Roter Weg“: **(5)**



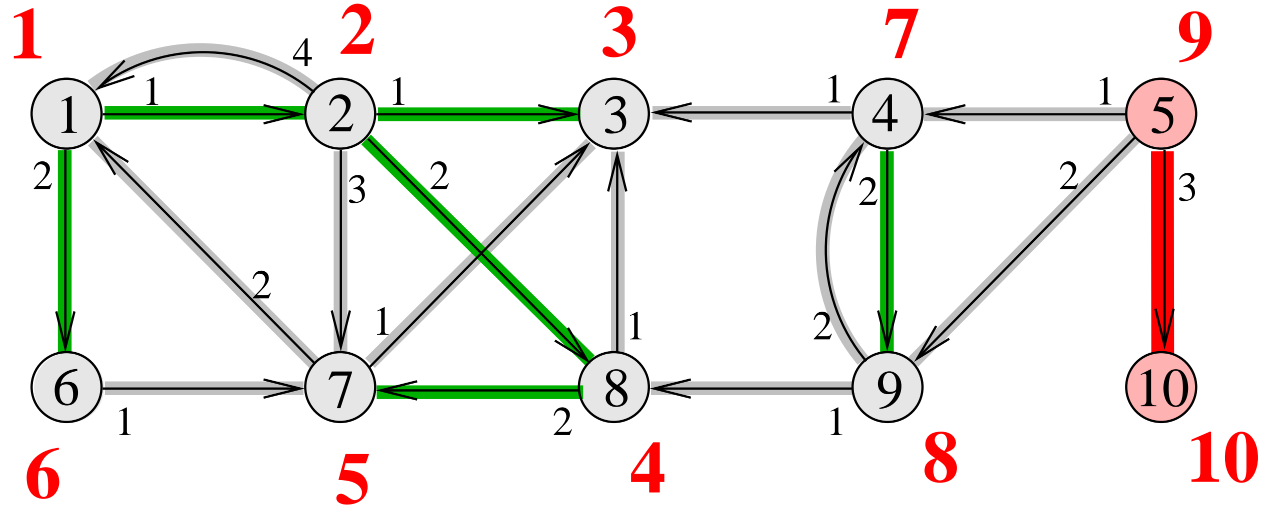
„Roter Weg": **(5)**



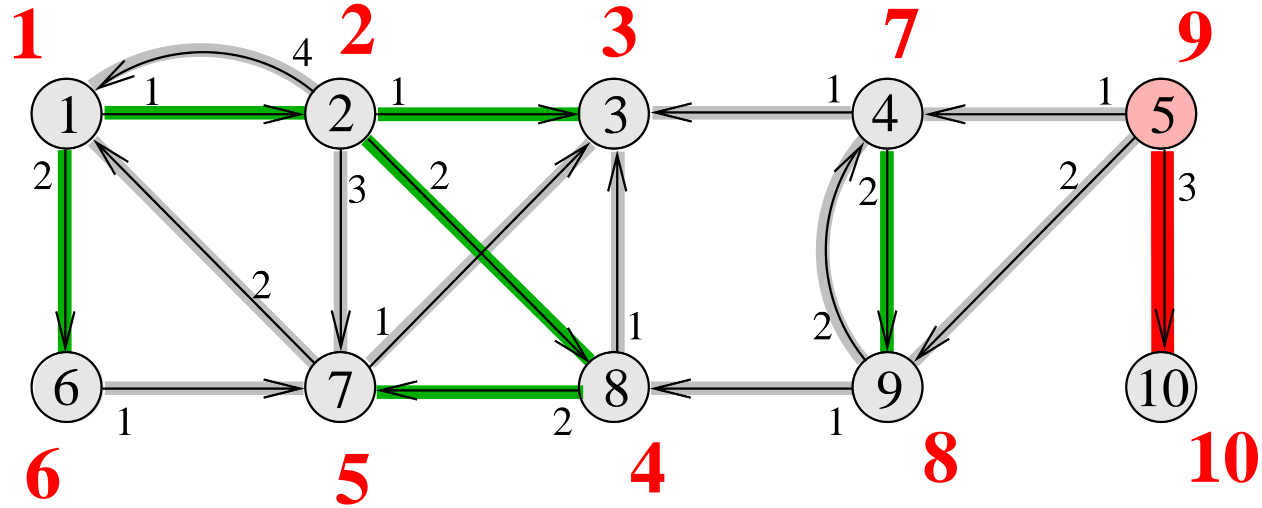
„Roter Weg“: **(5)**



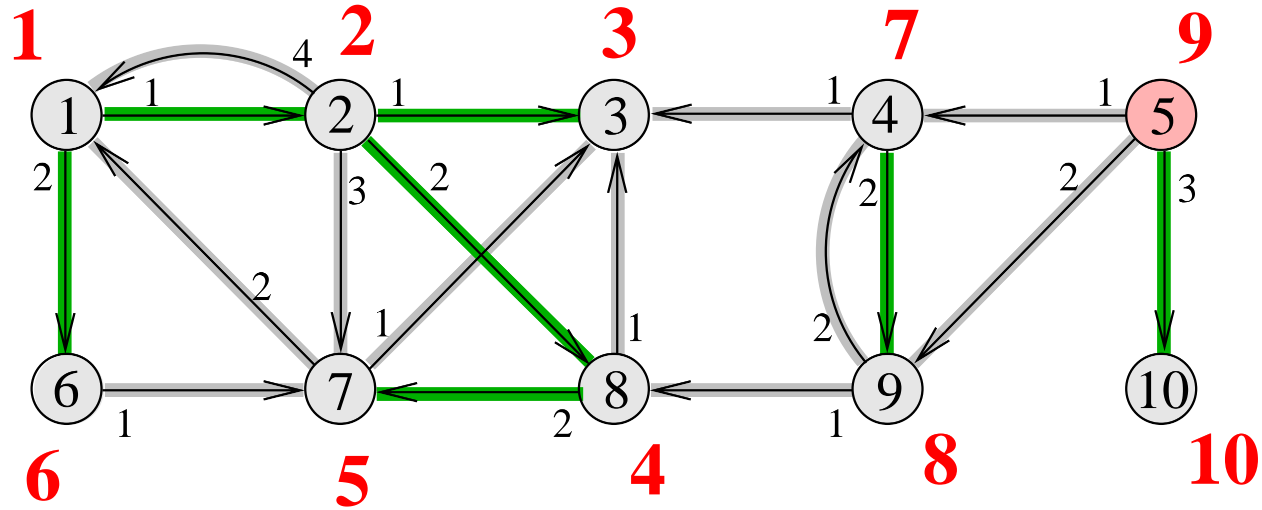
„Roter Weg“: **(5)**



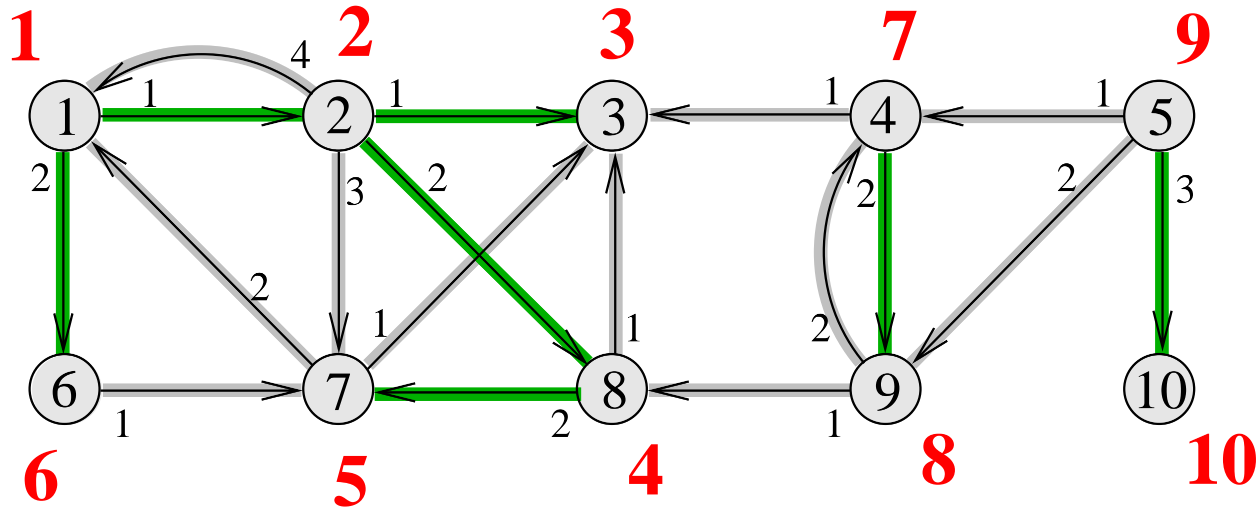
„Roter Weg“: $\mathbf{(5, 10)}$



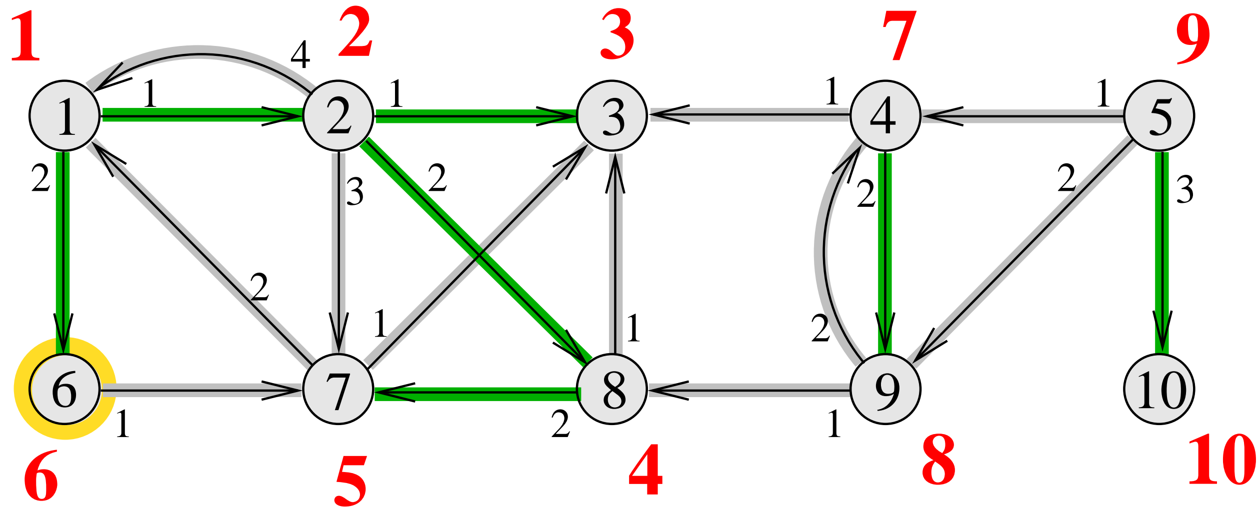
„Roter Weg": **(5)**



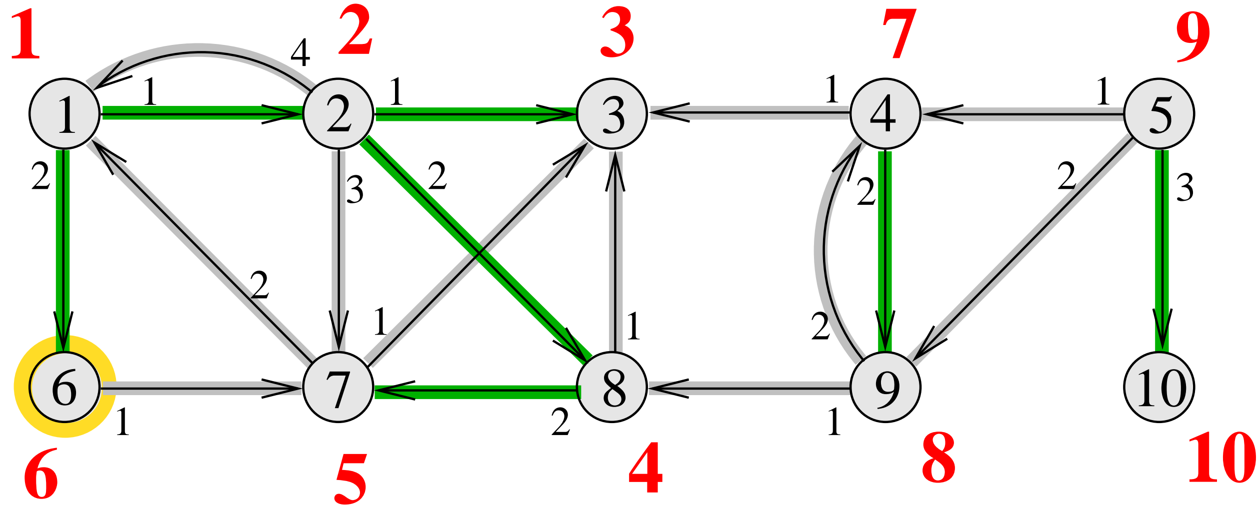
„Roter Weg“: **(5)**



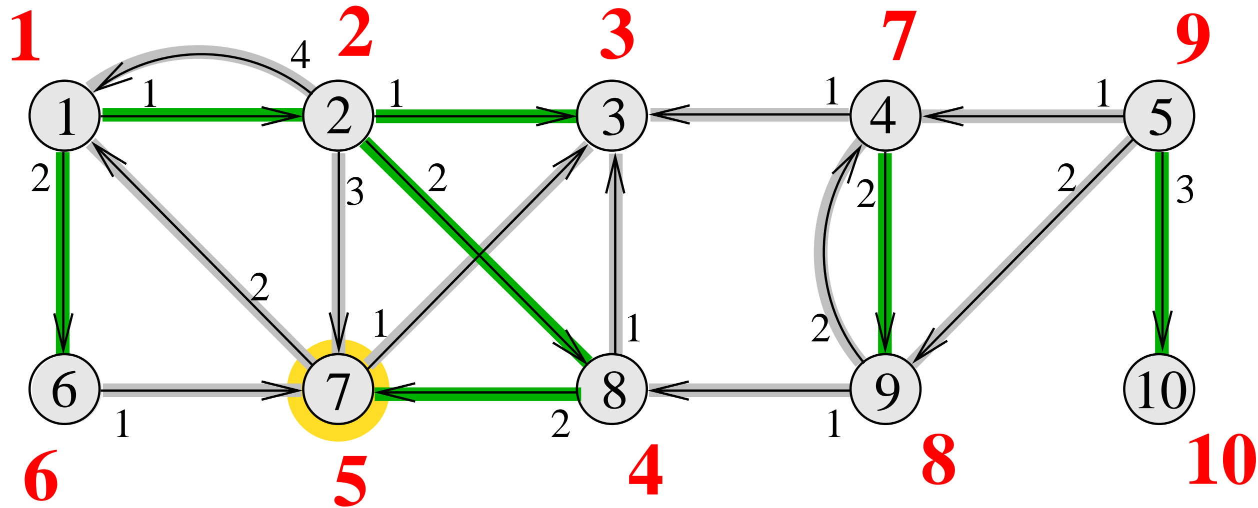
„Roter Weg“: leer.



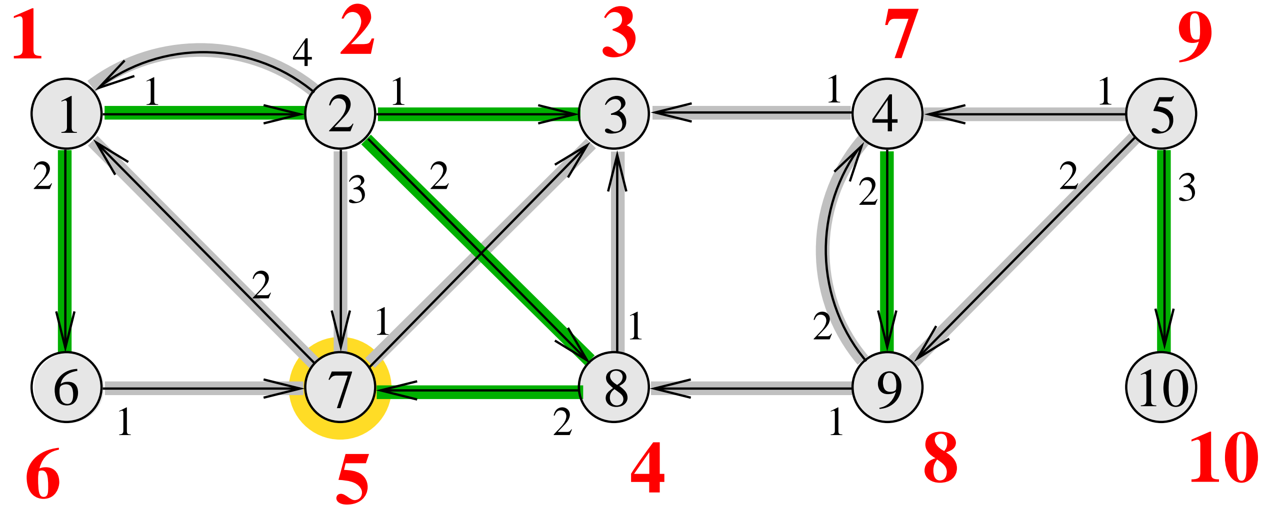
Ist Knoten 6 **neu** ?



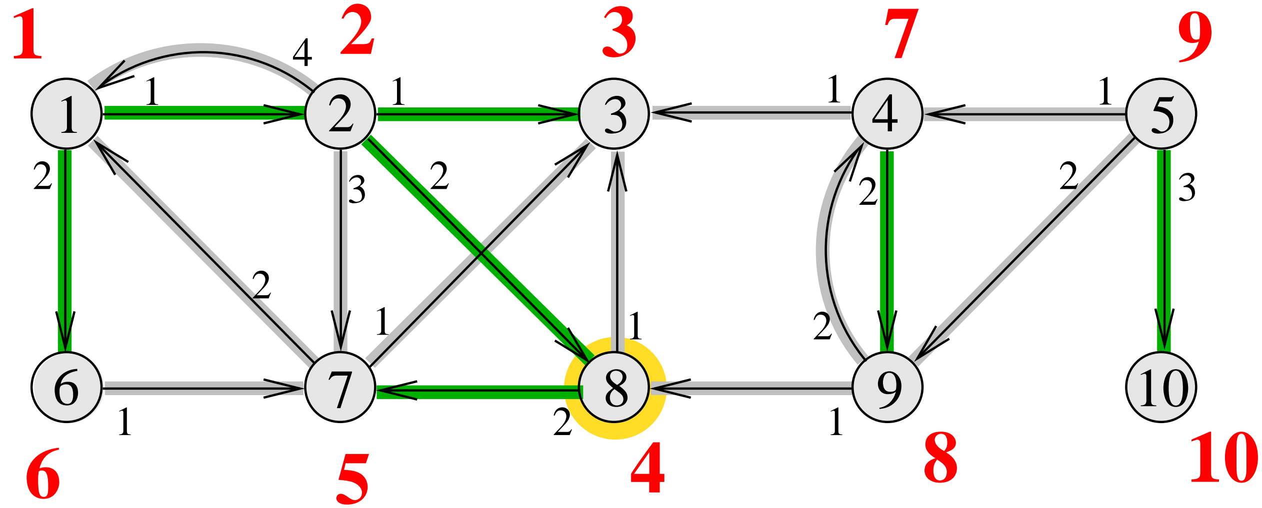
Ist Knoten 6 **neu** ? Nein.



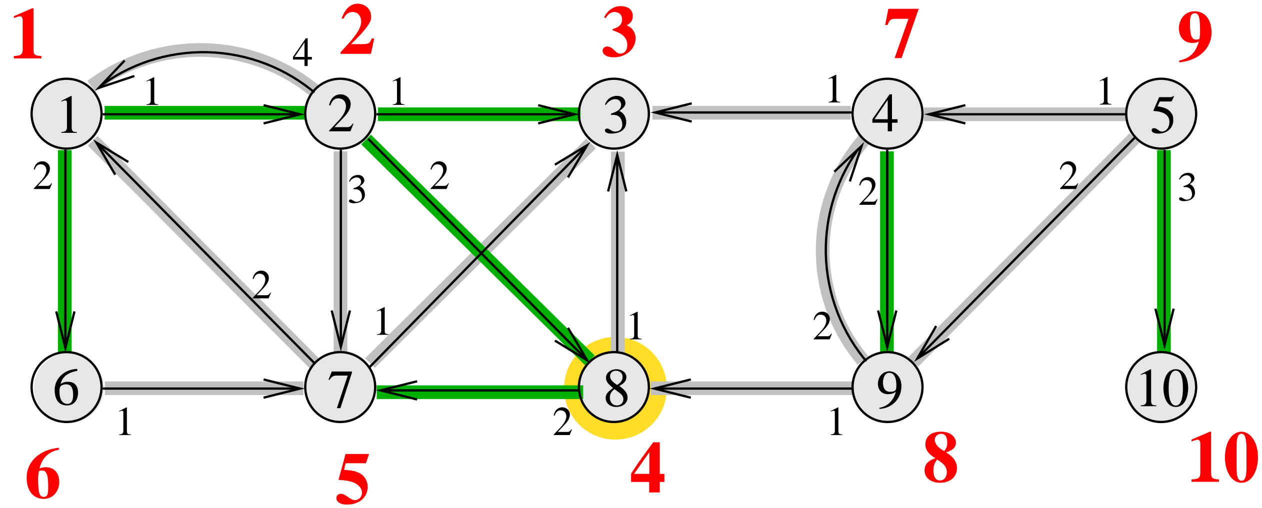
Ist Knoten 7 **neu** ?



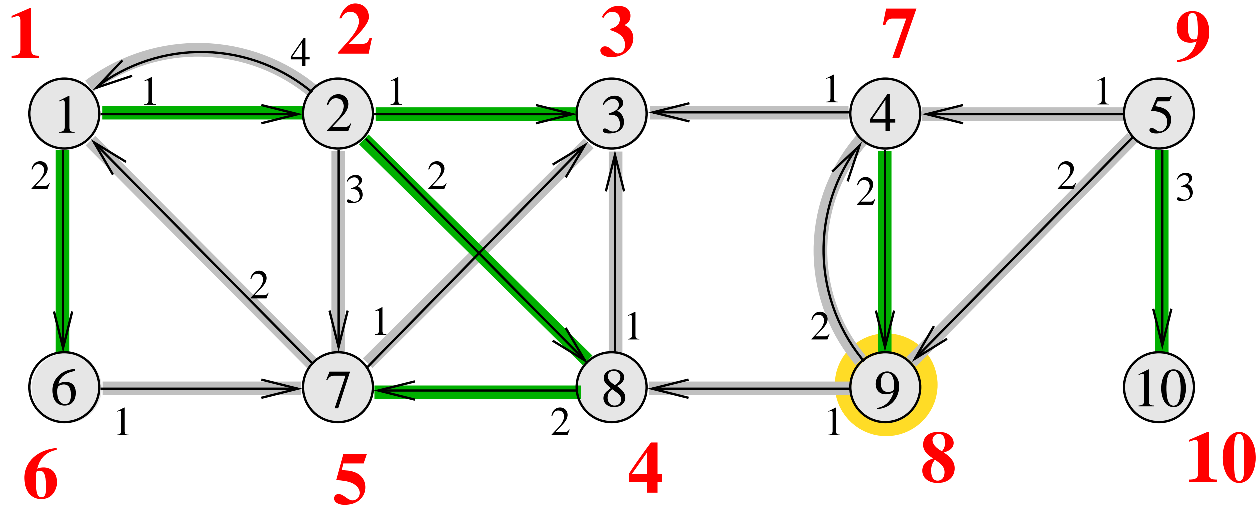
Ist Knoten 7 **neu** ? Nein.



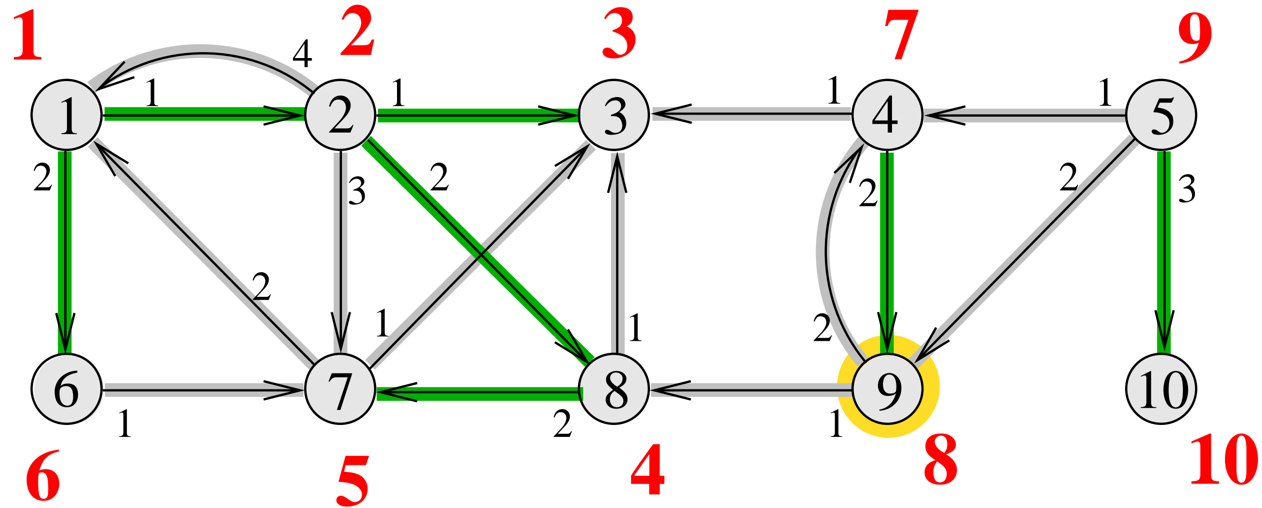
Ist Knoten 8 **neu** ?



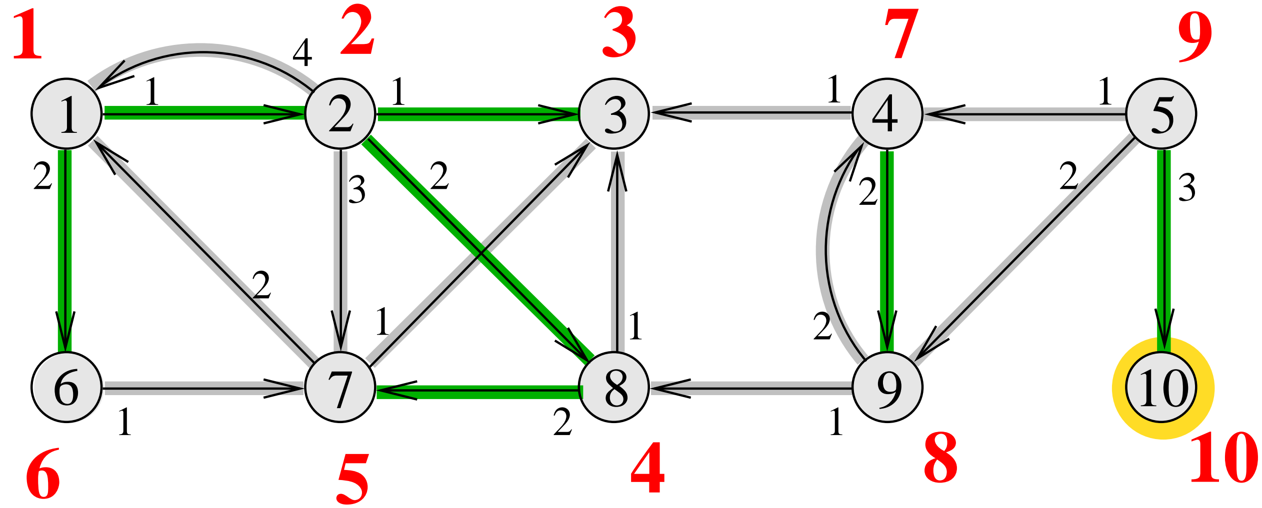
Ist Knoten 8 **neu** ? Nein.



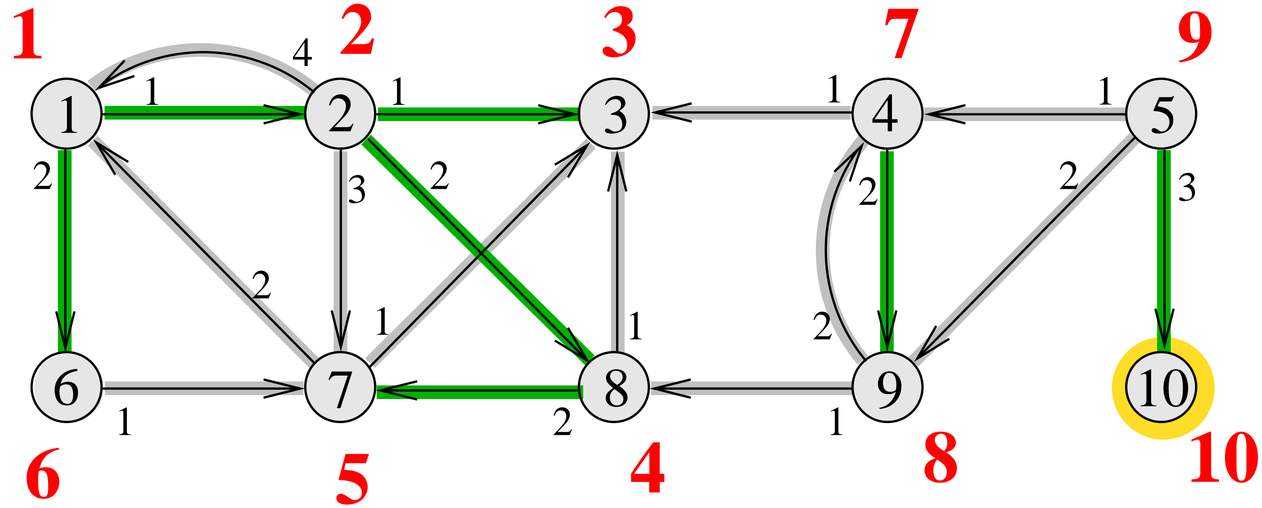
Ist Knoten 9 **neu** ?



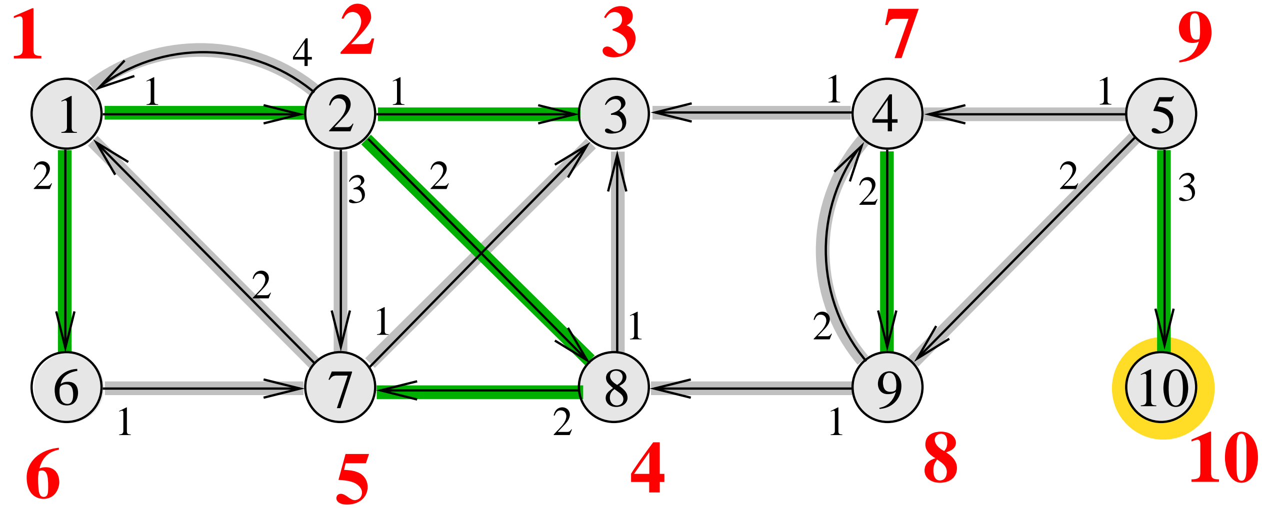
Ist Knoten 9 **neu** ? Nein.



Ist Knoten 10 **neu** ?



Ist Knoten 10 **neu** ? Nein.



Ist Knoten 10 **neu** ? Nein.

Fertig.

DFS(G) erzwingt, dass für **jeden** Knoten v in V irgendwann einmal dfs(v) aufgerufen wird. (Dass es höchstens einen Aufruf für v gibt, sieht man wie vorher.)

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

DFS(G) erzwingt, dass für **jeden** Knoten v in V irgendwann einmal dfs(v) aufgerufen wird. (Dass es höchstens einen Aufruf für v gibt, sieht man wie vorher.)

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

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

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

DFS(G) erzwingt, dass für **jeden** Knoten v in V irgendwann einmal dfs(v) aufgerufen wird. (Dass es höchstens einen Aufruf für v gibt, sieht man wie vorher.)

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

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

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

Satz 8.1.4

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

DFS(G) erzwingt, dass für **jeden** Knoten v in V irgendwann einmal dfs(v) aufgerufen wird. (Dass es höchstens einen Aufruf für v gibt, sieht man wie vorher.)

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 8.1.4

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

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

Bemerkung:
Bei DFS entstehen mehrere Bäume, die zusammen den **„Tiefensuchwald“** bilden.

Bemerkung:

Bei DFS entstehen mehrere Bäume, die zusammen den **„Tiefensuchwald“** bilden.

(Kante (w, v) entspricht dem Aufruf dfs(v) direkt aus dfs(w).)

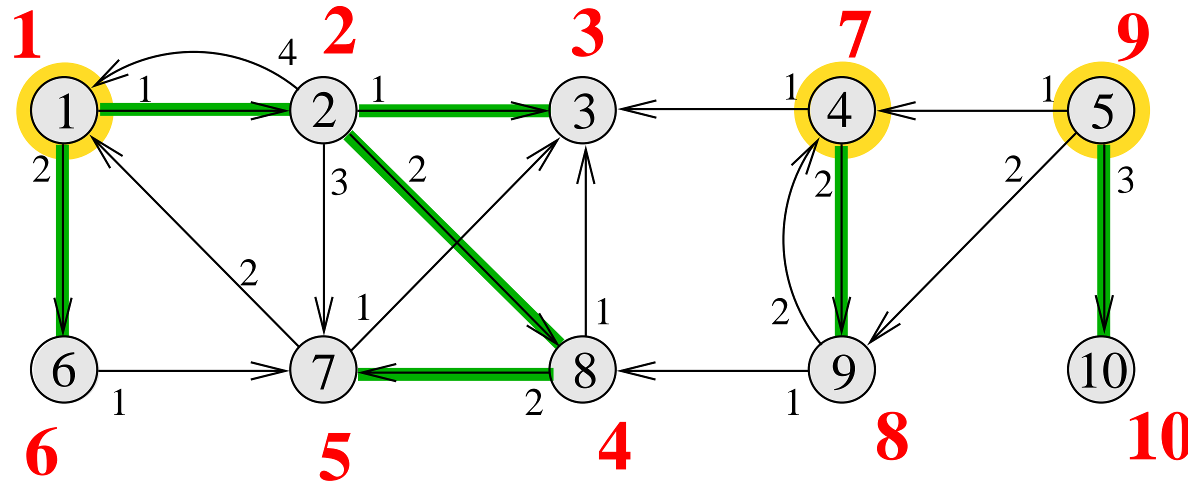
Vorsicht: Wenn ein Baum im Tiefensuchwald Wurzel v hat, so ist nicht gesagt, dass dieser Baum alle von v aus erreichbaren Knoten enthält. (Dies gilt nur für den **ersten** der Bäume.)

Bemerkung:

Bei DFS entstehen mehrere Bäume, die zusammen den **„Tiefensuchwald“** bilden.

(Kante (w, v) entspricht dem Aufruf dfs(v) direkt aus dfs(w).)

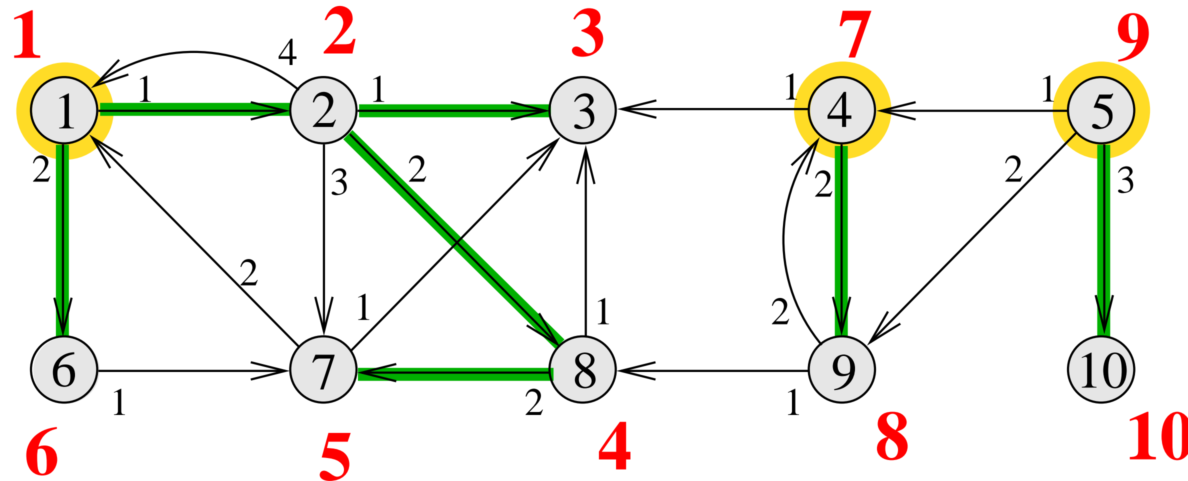
Vorsicht: Wenn ein Baum im Tiefensuchwald Wurzel v hat, so ist nicht gesagt, dass dieser Baum alle von v aus erreichbaren Knoten enthält. (Dies gilt nur für den **ersten** der Bäume.)



Bemerkung:
Bei DFS entstehen mehrere Bäume, die zusammen den **„Tiefensuchwald“** bilden.

(Kante (w, v) entspricht dem Aufruf dfs(v) direkt aus dfs(w).)

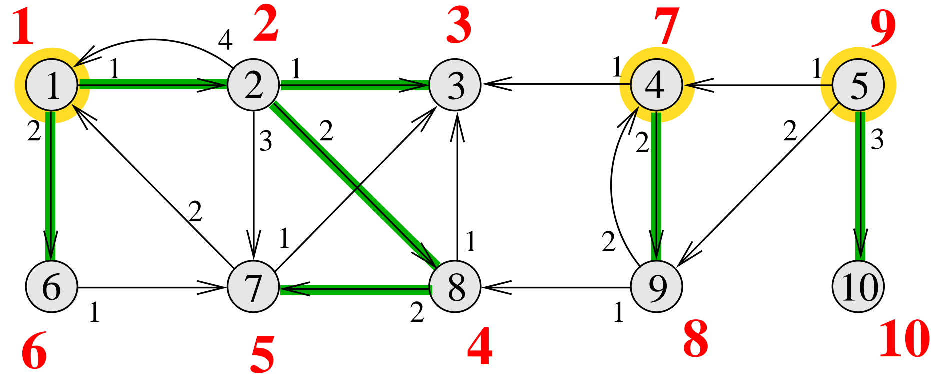
Vorsicht: Wenn ein Baum im Tiefensuchwald Wurzel v hat, so ist nicht gesagt, dass dieser Baum alle von v aus erreichbaren Knoten enthält. (Dies gilt nur für den **ersten** der Bäume.)



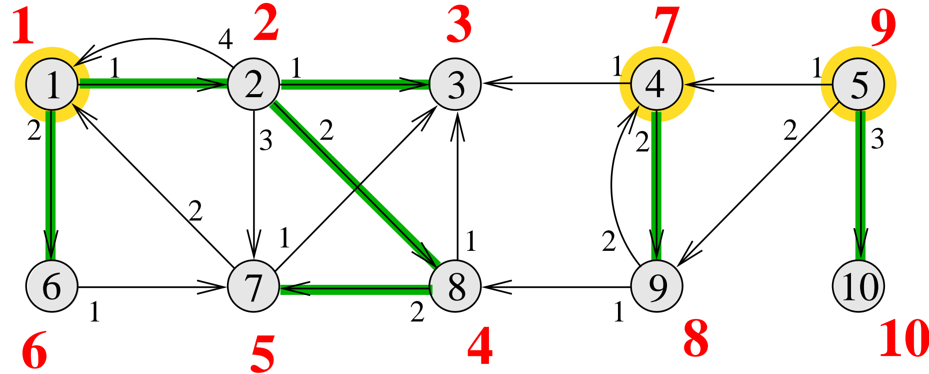
Beispiel: Es gilt $5 \leadsto 1$, aber 1 ist nicht im Baum mit Wurzel 5.

Satz 8.1.3, der „Satz vom weißen Weg“, gilt auch für den Tiefensuchwald, mit identischem Beweis.

Satz 8.1.3, der „Satz vom weißen Weg“, gilt auch für den Tiefensuchwald, mit identischem Beweis.

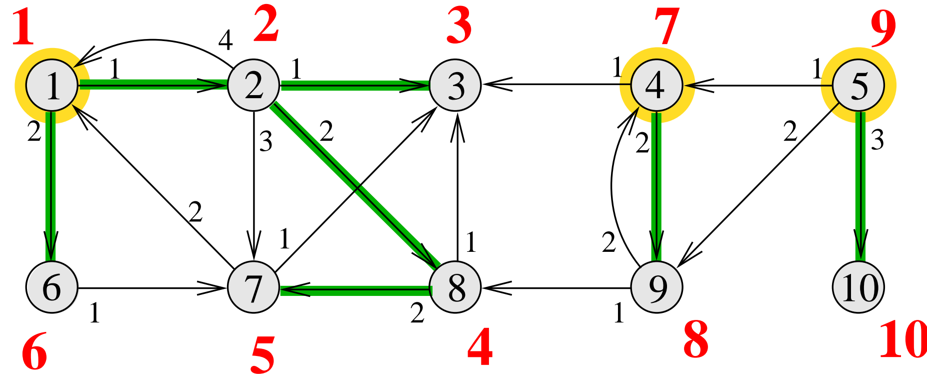


Satz 8.1.3, der „Satz vom weißen Weg“, gilt auch für den Tiefensuchwald, mit identischem Beweis.



Für $v \in V = \{1, \ldots, n\}$: $\boldsymbol{W_v} := \{u \in V \mid \exists w \leq v\colon w \leadsto u\}$.

Satz 8.1.3, der „Satz vom weißen Weg“, gilt auch für den Tiefensuchwald, mit identischem Beweis.



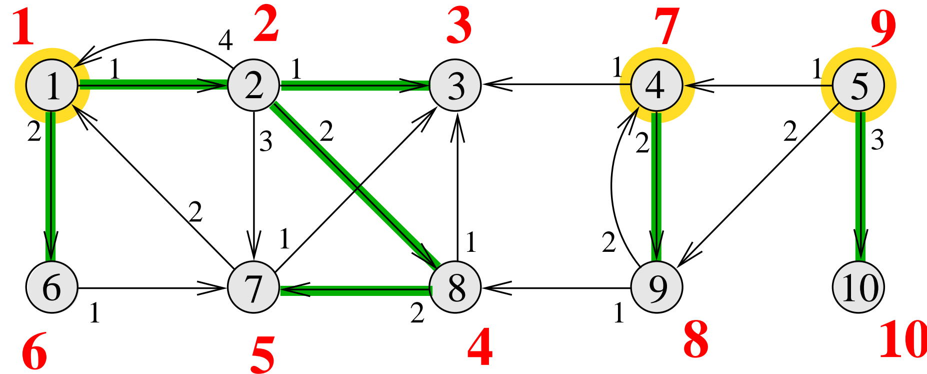
Für $v \in V = \{1, \ldots, n\}$: $\boldsymbol{W_v} := \{u \in V \mid \exists w \leq v \colon w \rightsquigarrow u\}$.

Beispiel: $W_1 = W_2 = W_3 = \{1, 2, 3, 6, 7, 8\}$, $W_4 = \{1, 2, 3, 4, 6, 7, 8, 9\}$, also $4 \notin W_3$ und $W_4 - W_3 = \{4, 9\}$.

Satz 8.1.5 (vgl. Beh. 7.3.5 zu BFS-Bäumen)

(a) v wird Wurzel eines Baums im Tiefensuch-Wald $\Leftrightarrow v \notin W_{v-1}$.

Satz 8.1.3, der „Satz vom weißen Weg“, gilt auch für den Tiefensuchwald, mit identischem Beweis.



Für $v \in V = \{1, \ldots, n\}$: $\boldsymbol{W_v} := \{u \in V \mid \exists w \leq v \colon w \leadsto u\}$.

Beispiel: $W_1 = W_2 = W_3 = \{1, 2, 3, 6, 7, 8\}$, $W_4 = \{1, 2, 3, 4, 6, 7, 8, 9\}$, also $4 \notin W_3$ und $W_4 - W_3 = \{4, 9\}$.

Satz 8.1.5 (vgl. Beh. 7.3.5 zu BFS-Bäumen)

(a) v wird Wurzel eines Baums im Tiefensuch-Wald $\Leftrightarrow v \notin W_{v-1}$.

(b) Wenn v Wurzel ist, dann hat der Baum mit Wurzel v Knotenmenge $W_v - W_{v-1}$ (erreichbar von v aus, aber von keinem vorherigen Knoten). (Beweis auf Druckfolien.)

PAUSE

Es folgt: „Volle“ Tiefensuche: Noch mehr Information!

8.2 Volle Tiefensuche in Digraphen

8.2 Volle Tiefensuche in Digraphen

Wir bauen dfs(v) und DFS(G) aus, so dass die Kanten (v, w) von G in vier Klassen $\boldsymbol{T}$, $\boldsymbol{B}$, $\boldsymbol{F}$, C eingeteilt werden:

8.2 Volle Tiefensuche in Digraphen

Wir bauen dfs(v) und DFS(G) aus, so dass die Kanten (v, w) von G in vier Klassen $\boldsymbol{T}$, $\boldsymbol{B}$, $\boldsymbol{F}$, C eingeteilt werden:

- $\boldsymbol{T}$ **(„Tree“): Baumkanten**, die Kanten des Tiefensuchwaldes.

8.2 Volle Tiefensuche in Digraphen

Wir bauen dfs(v) und DFS(G) aus, so dass die Kanten (v, w) von G in vier Klassen $\boldsymbol{T}$, $\boldsymbol{B}$, $\boldsymbol{F}$, C eingeteilt werden:

- $\boldsymbol{T}$ **(„Tree“): Baumkanten**, die Kanten des Tiefensuchwaldes.

  Kriterium: Wenn dfs(v) die Kante (v, w) betrachtet, ist w noch **neu** .

8.2 Volle Tiefensuche in Digraphen

Wir bauen dfs(v) und DFS(G) aus, so dass die Kanten (v, w) von G in vier Klassen $\boldsymbol{T}$, $\boldsymbol{B}$, $\boldsymbol{F}$, C eingeteilt werden:

- $\boldsymbol{T}$ **(„Tree"): Baumkanten**, die Kanten des Tiefensuchwaldes.

  Kriterium: Wenn dfs(v) die Kante (v, w) betrachtet, ist w noch **neu** .

- B **(„Back"): Rückwärtskanten**.
  (v, w) ist Rückwärtskante, wenn w Vorfahr von v im Tiefensuchwald ist.

8.2 Volle Tiefensuche in Digraphen

Wir bauen dfs(v) und DFS(G) aus, so dass die Kanten (v, w) von G in vier Klassen $\boldsymbol{T}$, $\boldsymbol{B}$, $\boldsymbol{F}$, C eingeteilt werden:

- $\boldsymbol{T}$ **(„Tree“): Baumkanten**, die Kanten des Tiefensuchwaldes.

  Kriterium: Wenn dfs(v) die Kante (v, w) betrachtet, ist w noch **neu** .

- $\boldsymbol{B}$ **(„Back“): Rückwärtskanten**.
  (v, w) ist Rückwärtskante, wenn w Vorfahr von v im Tiefensuchwald ist.

  Kriterium: Wenn dfs(v) die Kante (v, w) betrachtet, ist w **aktiv** .

- ***F*: Vorwärtskanten** („Forward“).
  (v, w) ist Vorwärtskante, wenn w Nachfahr von v im Tiefensuchwald ist, und (v, w) keine Baumkante ist.

- ***F*: Vorwärtskanten** („Forward").
  (v, w) ist Vorwärtskante, wenn w Nachfahr von v im Tiefensuchwald ist, und (v, w) keine Baumkante ist.

  Kriterium: Wenn dfs(v) die Kante (v, w) betrachtet, ist w **fertig** und *dfs_num*$(v) <$ *dfs_num*(w).

- F: **Vorwärtskanten** („Forward").
  (v, w) ist Vorwärtskante, wenn w Nachfahr von v im Tiefensuchwald ist, und (v, w) keine Baumkante ist.

  Kriterium: Wenn dfs(v) die Kante (v, w) betrachtet, ist w **fertig** und *dfs_num*$(v) <$ *dfs_num*(w).

- C: **Querkanten** („Cross").
  (v, w) ist Querkante, wenn der gesamte Unterbaum von w schon fertig abgearbeitet ist, wenn dfs(v) aufgerufen wird.

- ***F*****: Vorwärtskanten** („Forward“).
  (v, w) ist Vorwärtskante, wenn w Nachfahr von v im Tiefensuchwald ist, und (v, w) keine Baumkante ist.

  Kriterium: Wenn dfs(v) die Kante (v, w) betrachtet, ist w **fertig** und *dfs_num*$(v) <$ *dfs_num*(w).

- ***C*****: Querkanten** („Cross“).
  (v, w) ist Querkante, wenn der gesamte Unterbaum von w schon fertig abgearbeitet ist, wenn dfs(v) aufgerufen wird.

  Kriterium: Wenn dfs(v) die Kante (v, w) betrachtet, ist w **fertig** und *dfs_num*$(v) >$ *dfs_num*(w).

- ***F*: Vorwärtskanten** („Forward").
  (v, w) ist Vorwärtskante, wenn w Nachfahr von v im Tiefensuchwald ist, und (v, w) keine Baumkante ist.

  Kriterium: Wenn dfs(v) die Kante (v, w) betrachtet, ist w **fertig** und *dfs_num*$(v) <$ *dfs_num*(w).

- ***C*: Querkanten** („Cross").
  (v, w) ist Querkante, wenn der gesamte Unterbaum von w schon fertig abgearbeitet ist, wenn dfs(v) aufgerufen wird.

  Kriterium: Wenn dfs(v) die Kante (v, w) betrachtet, ist w **fertig** und *dfs_num*$(v) >$ *dfs_num*(w).

Außerdem nummerieren wir die Knoten $v \in V$ auch **in der Reihenfolge** durch, in der die dfs(v)-Aufrufe **beendet** werden:

- F: **Vorwärtskanten** („Forward“).
  (v, w) ist Vorwärtskante, wenn w Nachfahr von v im Tiefensuchwald ist, und (v, w) keine Baumkante ist.

  Kriterium: Wenn dfs(v) die Kante (v, w) betrachtet, ist w **fertig** und *dfs_num*$(v) <$ *dfs_num*(w).

- C: **Querkanten** („Cross“).
  (v, w) ist Querkante, wenn der gesamte Unterbaum von w schon fertig abgearbeitet ist, wenn dfs(v) aufgerufen wird.

  Kriterium: Wenn dfs(v) die Kante (v, w) betrachtet, ist w **fertig** und *dfs_num*$(v) >$ *dfs_num*(w).

Außerdem nummerieren wir die Knoten $v \in V$ auch **in der Reihenfolge** durch, in der die dfs(v)-Aufrufe **beendet** werden:

f(in)-Nummern $\texttt{f_num}(v)$, $v \in \{1, \ldots, n\}$, mit Wertebereich $\{1, 2, \ldots, |V|\}$.

Prozedur dfs(v) // „volle“ Tiefensuche in v, nur für v **neu** erlaubt

Prozedur dfs(v) // „volle“ Tiefensuche in v, nur für v **neu** erlaubt

(1) `dfs_count++;`

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

Prozedur dfs(v) // „volle“ Tiefensuche in v, nur für v **neu** erlaubt

(1) `dfs_count++;`

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

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

Prozedur dfs(v) // „volle“ Tiefensuche in v, nur für v **neu** erlaubt

(1) `dfs_count++;`

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

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

(4) **dfs-visit**(v); // Aktion an v bei Entdeckung

Prozedur dfs(v) // „volle“ Tiefensuche in v, nur für v **neu** erlaubt

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

Prozedur dfs(v) // „volle“ Tiefensuche in v, nur für v **neu** erlaubt

(1) $\texttt{dfs_count++}$;
(2) $\texttt{dfs_num}[v] \leftarrow \texttt{dfs_count}$;
(3) $\texttt{status}[v] \leftarrow$ **aktiv** ;
(4) **dfs-visit**(v); // Aktion an v bei Entdeckung
(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$(\texttt{w})$;

Prozedur dfs(v) // „volle“ Tiefensuche in v, nur für v **neu** erlaubt

(1) `dfs_count++;`
(2) `dfs_num[`v`]` $\leftarrow$ `dfs_count;`
(3) `status[`v`]` $\leftarrow$ **aktiv** ;
(4) **dfs-visit**(v); // Aktion an v bei Entdeckung
(5) für jeden Nachfolger `w` von v (Adjazenzliste!) tue:
(6) **1. Fall:** `status[w]` $=$ **neu** : $\boldsymbol{T} \leftarrow \boldsymbol{T} \cup \{(v, \mathtt{w})\}$; dfs$(\mathtt{w})$;
(7) **2. Fall:** `status[w]` $=$ **aktiv** : $\boldsymbol{B} \leftarrow \boldsymbol{B} \cup \{(v, \mathtt{w})\}$;

Prozedur dfs(v) // „volle“ Tiefensuche in v, nur für v **neu** erlaubt

(1) `dfs_count++`;
(2) `dfs_num[`v`]` $\leftarrow$ `dfs_count`;
(3) `status[`v`]` $\leftarrow$ **aktiv** ;
(4) **dfs-visit**(v); // Aktion an v bei Entdeckung
(5) für jeden Nachfolger `w` von v (Adjazenzliste!) tue:
(6) **1. Fall:** `status[w]` $=$ **neu** : $\boldsymbol{T} \leftarrow \boldsymbol{T} \cup \{(v, \mathtt{w})\}$; dfs$(\mathtt{w})$;
(7) **2. Fall:** `status[w]` $=$ **aktiv** : $\boldsymbol{B} \leftarrow \boldsymbol{B} \cup \{(v, \mathtt{w})\}$;
(8) **3. Fall:** `status[w]` $=$ **fertig** $\wedge$ `dfs_num[`v`]` $<$ `dfs_num[w]`:
$\boldsymbol{F} \leftarrow \boldsymbol{F} \cup \{(v, \mathtt{w})\}$;

Prozedur dfs(v) // „volle“ Tiefensuche in v, nur für v **neu** erlaubt

(1) `dfs_count++;`

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

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

(4) **dfs-visit**(v); // Aktion an v bei Entdeckung

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

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

(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]`:
$C \leftarrow C \cup \{(v, \mathtt{w})\}$;

Prozedur dfs(v) // „volle“ Tiefensuche in v, nur für v **neu** erlaubt

(1) `dfs_count++;`

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

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

(4) **dfs-visit**(v); // Aktion an v bei Entdeckung

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

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

(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++;`

Prozedur dfs(v) // „volle" Tiefensuche in v, nur für v **neu** erlaubt

```
(1)   dfs_count++;
(2)   dfs_num[v] ← dfs_count;
(3)   status[v] ← aktiv ;
(4)   dfs-visit(v); // Aktion an v bei Entdeckung
(5)       für jeden Nachfolger w von v (Adjazenzliste!) tue:
(6)           1. Fall: status[w] = neu :     T ← T ∪ {(v, w)}; dfs(w);
(7)           2. Fall: status[w] = aktiv :   B ← B ∪ {(v, w)};
(8)           3. Fall: status[w] = fertig ∧ dfs_num[v] < dfs_num[w]:
                                              F ← F ∪ {(v, w)};
(9)           4. Fall: status[w] = fertig ∧ dfs_num[v] > dfs_num[w]:
                                              C ← C ∪ {(v, w)};
(10)  f_count++;
(11)  f_num[v] ← f_count;
```

Prozedur dfs(v) // „volle“ Tiefensuche in v, nur für v **neu** erlaubt

(1) `dfs_count++;`
(2) `dfs_num[`v`]` $\leftarrow$ `dfs_count;`
(3) `status[`v`]` $\leftarrow$ **aktiv** ;
(4) **dfs-visit**(v); // Aktion an v bei Entdeckung
(5) für jeden Nachfolger `w` von v (Adjazenzliste!) tue:
(6) **1. Fall:** `status[w]` $=$ **neu** : $T \leftarrow T \cup \{(v, \mathtt{w})\}$; dfs$(\mathtt{w})$;
(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 Abschluss

Prozedur dfs(v) // „volle“ Tiefensuche in v, nur für v **neu** erlaubt

(1) `dfs_count++;`
(2) `dfs_num[`v`]` $\leftarrow$ `dfs_count;`
(3) `status[`v`]` $\leftarrow$ **aktiv** ;
(4) **dfs-visit**(v); // Aktion an v bei Entdeckung
(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 Abschluss
(13) `status[`v`]` $\leftarrow$ **fertig** .

Prozedur dfs(v) // „volle“ Tiefensuche in v, nur für v **neu** erlaubt

Prozedur dfs(v) // „volle“ Tiefensuche in v, nur für v **neu** erlaubt

(1) `dfs_count++;`

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

Prozedur dfs(v) // „volle“ Tiefensuche in v, nur für v **neu** erlaubt

(1) `dfs_count++;`

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

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

Prozedur dfs(v) // „volle“ Tiefensuche in v, nur für v **neu** erlaubt

(1) `dfs_count++;`

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

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

(4) **dfs-visit**(v); // Aktion an v bei Entdeckung

Prozedur dfs(v) // „volle“ Tiefensuche in v, nur für v **neu** erlaubt

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

Prozedur dfs(v) // „volle“ Tiefensuche in v, nur für v **neu** erlaubt

(1) `dfs_count++;`

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

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

(4) **dfs-visit**(v); // Aktion an v bei Entdeckung

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

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

Prozedur dfs(v) // „volle“ Tiefensuche in v, nur für v **neu** erlaubt

(1) `dfs_count++;`
(2) `dfs_num[`v`]` $\leftarrow$ `dfs_count;`
(3) `status[`v`]` $\leftarrow$ **aktiv** ;
(4) **dfs-visit**(v); // Aktion an v bei Entdeckung
(5) für jeden Nachfolger `w` von v (Adjazenzliste!) tue:
(6) **1. Fall:** `status[w]` $=$ **neu** : $\boldsymbol{T} \leftarrow \boldsymbol{T} \cup \{(v, \mathtt{w})\}$; dfs$(\mathtt{w})$;
(7) **2. Fall:** `status[w]` $=$ **aktiv** : $\boldsymbol{B} \leftarrow \boldsymbol{B} \cup \{(v, \mathtt{w})\}$;

Prozedur dfs(v) // „volle“ Tiefensuche in v, nur für v **neu** erlaubt

(1) `dfs_count++;`

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

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

(4) **dfs-visit**(v); // Aktion an v bei Entdeckung

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

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

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

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

Prozedur dfs(v) // „volle“ Tiefensuche in v, nur für v **neu** erlaubt

(1) `dfs_count++;`

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

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

(4) **dfs-visit**(v); // Aktion an v bei Entdeckung

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

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

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

(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]`:
$C \leftarrow C \cup \{(v, \mathtt{w})\}$;

Prozedur dfs(v) // „volle“ Tiefensuche in v, nur für v **neu** erlaubt

(1) `dfs_count++;`

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

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

(4) **dfs-visit**(v); // Aktion an v bei Entdeckung

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

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

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

(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++;`

Prozedur dfs(v) // „volle“ Tiefensuche in v, nur für v **neu** erlaubt

```
(1)   dfs_count++;
(2)   dfs_num[v] ← dfs_count;
(3)   status[v] ← aktiv ;
(4)   dfs-visit(v); // Aktion an v bei Entdeckung
(5)       für jeden Nachfolger w von v (Adjazenzliste!) tue:
(6)           1. Fall: status[w] = neu :      T ← T ∪ {(v, w)}; dfs(w);
(7)           2. Fall: status[w] = aktiv :    B ← B ∪ {(v, w)};
(8)           3. Fall: status[w] = fertig ∧ dfs_num[v] < dfs_num[w]:
                                              F ← F ∪ {(v, w)};
(9)           4. Fall: status[w] = fertig ∧ dfs_num[v] > dfs_num[w]:
                                              C ← C ∪ {(v, w)};
(10)  f_count++;
(11)  f_num[v] ← f_count;
```

Prozedur dfs(v) // „volle“ Tiefensuche in v, nur für v **neu** erlaubt

(1) `dfs_count++;`

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

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

(4) **dfs-visit**(v); // Aktion an v bei Entdeckung

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

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

(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 Abschluss

Prozedur dfs(v) // „volle“ Tiefensuche in v, nur für v **neu** erlaubt

(1) `dfs_count++;`
(2) `dfs_num[`v`]` $\leftarrow$ `dfs_count;`
(3) `status[`v`]` $\leftarrow$ **aktiv** ;
(4) **dfs-visit**(v); // Aktion an v bei Entdeckung
(5) für jeden Nachfolger `w` von v (Adjazenzliste!) tue:
(6) **1. Fall:** `status[w]` $=$ **neu** : $T \leftarrow T \cup \{(v, \mathtt{w})\}$; dfs$(\mathtt{w})$;
(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 Abschluss
(13) `status[`v`]` $\leftarrow$ **fertig** .

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

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

Alle Knoten sind **neu** ; Mengen $\boldsymbol{T}$, $\boldsymbol{B}$, $\boldsymbol{F}$, C sind leer.

Initialisierung: `dfs_count` $\leftarrow 0$; `f_count` $\leftarrow 0$.
Alle Knoten sind **neu** ; Mengen $\boldsymbol{T}$, $\boldsymbol{B}$, $\boldsymbol{F}$, C sind leer.
Tiefensuche von v_0 aus („dfs(v_0)“) entdeckt R_{v_0}.

Initialisierung: `dfs_count` $\leftarrow 0$; `f_count` $\leftarrow 0$.
Alle Knoten sind **neu** ; Mengen $\boldsymbol{T}$, $\boldsymbol{B}$, $\boldsymbol{F}$, C sind leer.
Tiefensuche von v_0 aus („dfs(v_0)“) entdeckt R_{v_0}.

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

Initialisierung: `dfs_count` $\leftarrow 0$; `f_count` $\leftarrow 0$.
Alle Knoten sind **neu** ; Mengen $\boldsymbol{T}$, $\boldsymbol{B}$, $\boldsymbol{F}$, C sind leer.
Tiefensuche von v_0 aus („dfs(v_0)“) entdeckt R_{v_0}.

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

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

Initialisierung: `dfs_count` $\leftarrow 0$; `f_count` $\leftarrow 0$.
Alle Knoten sind **neu** ; Mengen $\boldsymbol{T}$, $\boldsymbol{B}$, $\boldsymbol{F}$, C sind leer.
Tiefensuche von v_0 aus („dfs(v_0)“) entdeckt R_{v_0}.

Tiefensuche **für den ganzen Graphen**: DFS(G) wie vorher, nur mit der voll ausgebauten dfs-Prozedur, und der Initialisierung von `f_count` und $\boldsymbol{T}$, $\boldsymbol{B}$, $\boldsymbol{F}$, 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 (volle) Tiefensuche von `v` aus

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

Initialisierung: `dfs_count` $\leftarrow 0$; `f_count` $\leftarrow 0$.
Alle Knoten sind **neu** ; Mengen T, B, F, C sind leer.

Initialisierung: `dfs_count` $\leftarrow 0$; `f_count` $\leftarrow 0$.
Alle Knoten sind **neu** ; Mengen $\boldsymbol{T}$, $\boldsymbol{B}$, $\boldsymbol{F}$, $\boldsymbol{C}$ sind leer.
Tiefensuche von v_0 aus („dfs(v_0)“) entdeckt R_{v_0}.

Initialisierung: $\texttt{dfs_count} \leftarrow 0$; $\texttt{f_count} \leftarrow 0$.
Alle Knoten sind **neu** ; Mengen $\boldsymbol{T}$, $\boldsymbol{B}$, $\boldsymbol{F}$, $\boldsymbol{C}$ sind leer.
Tiefensuche von v_0 aus („dfs(v_0)") entdeckt R_{v_0}.

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

Initialisierung: `dfs_count` $\leftarrow 0$; `f_count` $\leftarrow 0$.
Alle Knoten sind **neu** ; Mengen $\boldsymbol{T}$, $\boldsymbol{B}$, $\boldsymbol{F}$, $\boldsymbol{C}$ sind leer.
Tiefensuche von v_0 aus („dfs(v_0)“) entdeckt R_{v_0}.

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

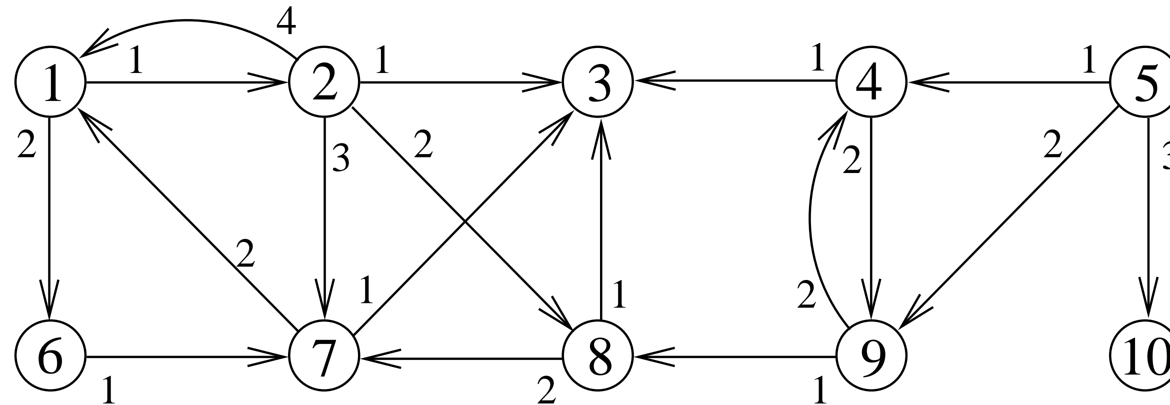
Algorithmus DFS(G) // Tiefensuche in $G = (V, E)$, voll ausgebaut

Initialisierung: $\texttt{dfs_count} \leftarrow 0$; $\texttt{f_count} \leftarrow 0$.
Alle Knoten sind **neu** ; Mengen T, B, F, C sind leer.
Tiefensuche von v_0 aus („dfs(v_0)") entdeckt R_{v_0}.

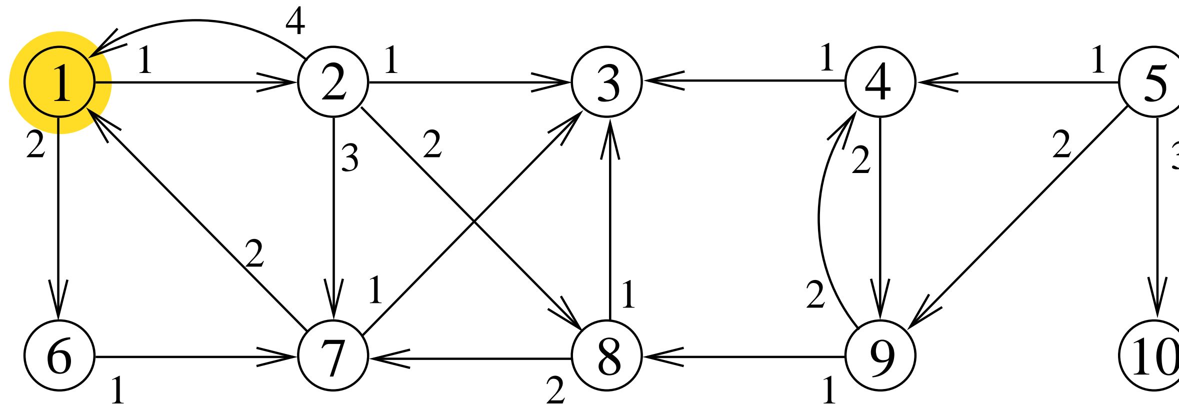
Tiefensuche **für den ganzen Graphen**: DFS(G) wie vorher, nur mit der voll ausgebauten dfs-Prozedur, und der Initialisierung von $\texttt{f_count}$ und T, B, F, C.

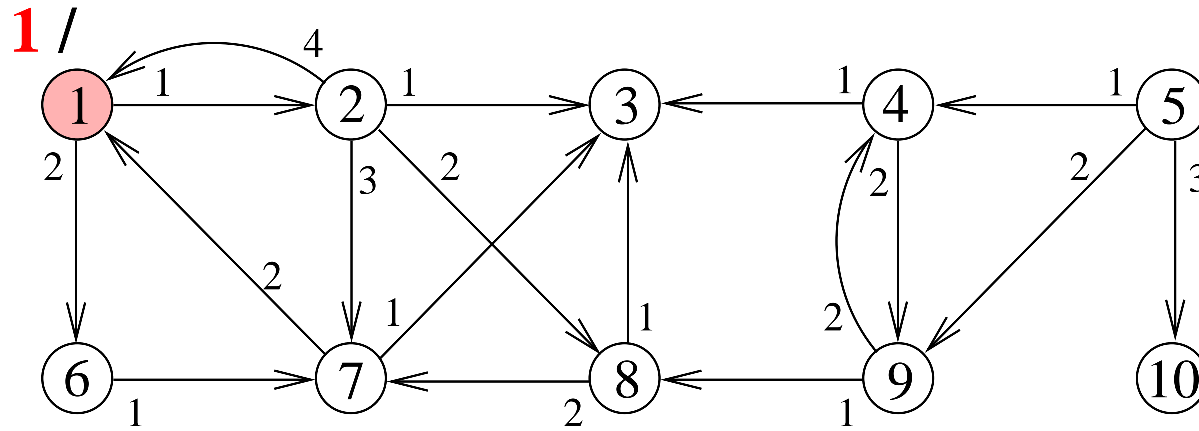
Algorithmus DFS(G) // Tiefensuche in $G = (V, E)$, voll ausgebaut

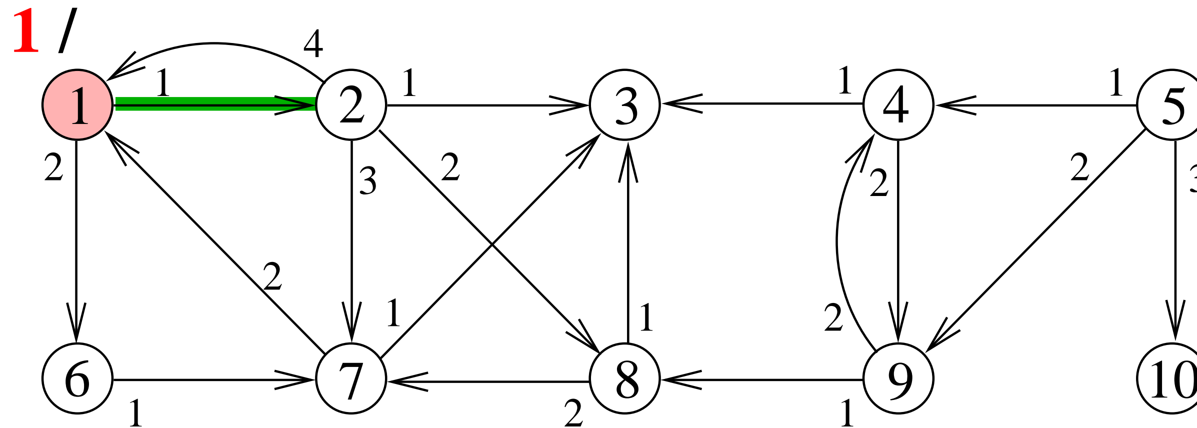
(1) $\texttt{dfs_count} \leftarrow 0$;
(2) $\texttt{f_count} \leftarrow 0$;
(3) $T \leftarrow \emptyset$; $B \leftarrow \emptyset$; $F \leftarrow \emptyset$; $C \leftarrow \emptyset$;
(3) **for** v **from** 1 **to** n **do** status[v] $\leftarrow$ **neu** ;
(4) **for** v **from** 1 **to** n **do**
(5)   **if** status[v] = **neu** **then**
(6)   dfs(v); // starte (volle) Tiefensuche von v aus



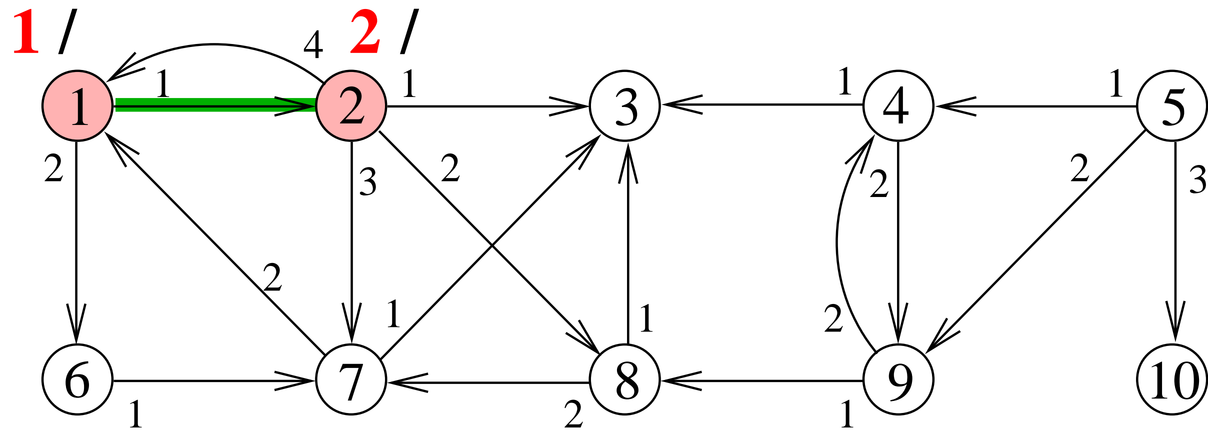
1
2
3
4
5
6
7
8
9
10
4
1
1
1
1
2
3
2
2
2
3
2
1
1
2
1
2
1

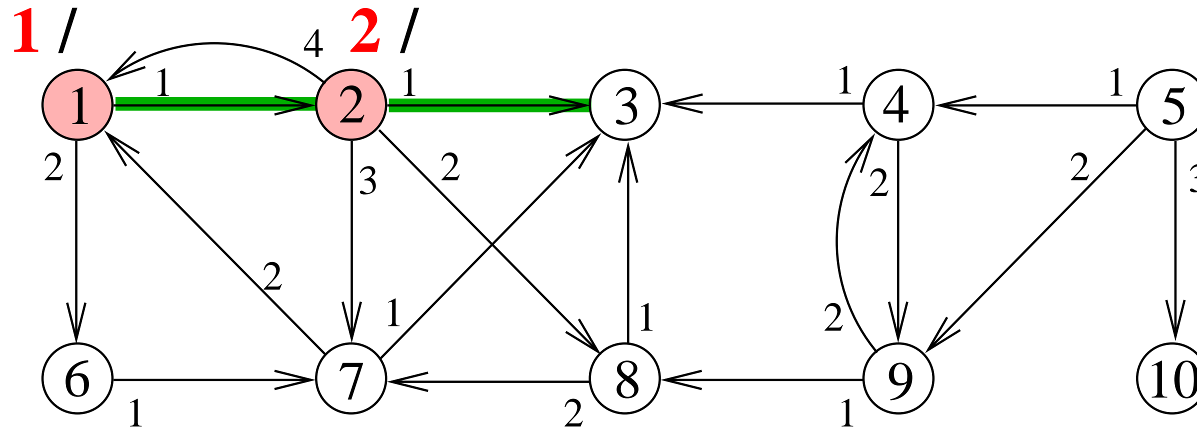

1
2
3
4
5
6
7
8
9
10
4
1
1
1
1
2
3
2
2
2
2
3
2
1
1
2
1
2
1


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

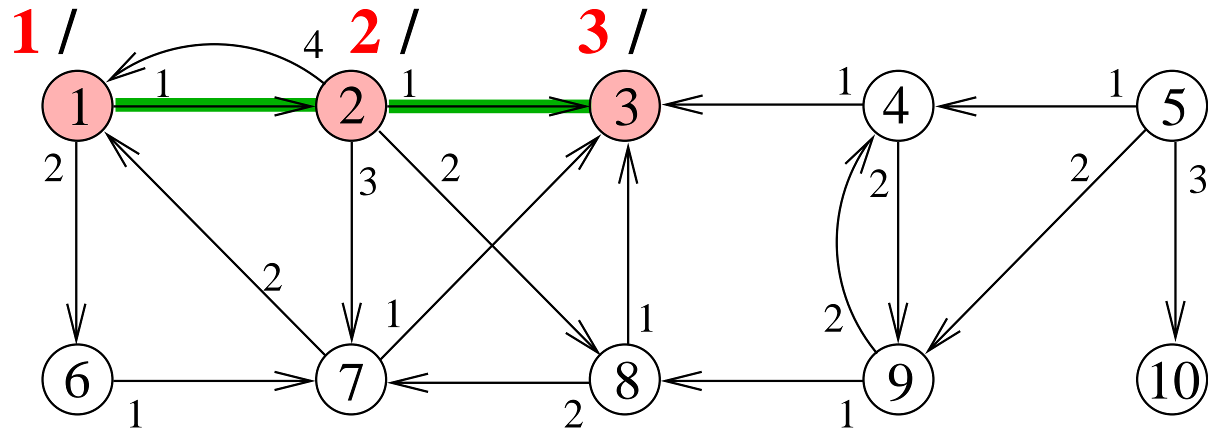


Baumkante, weil Knoten 2 **neu** ist.

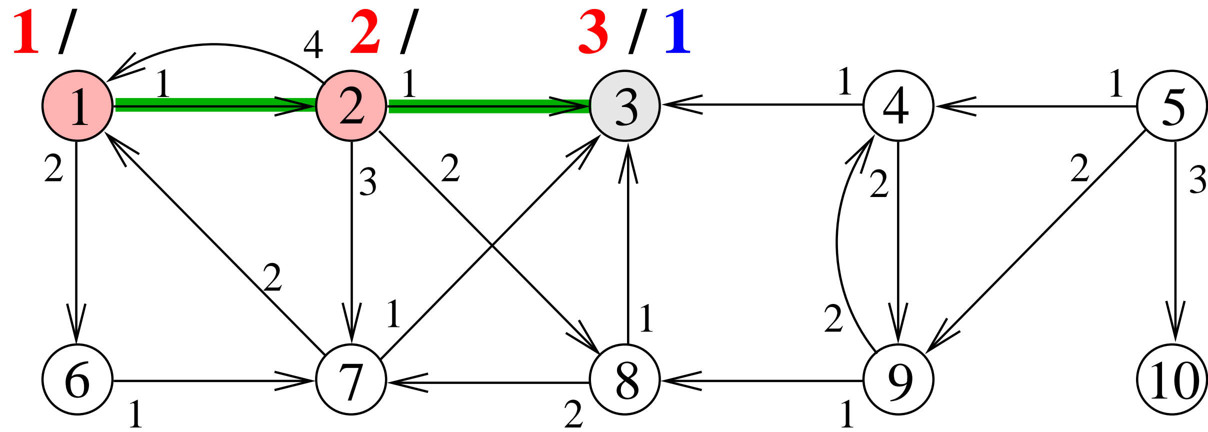

1 /
2 /
1
2
3
4
5
6
7
8
9
10
4
1
1
1
1
2
3
2
2
2
2
3
2
1
1
2
1
1
2
1

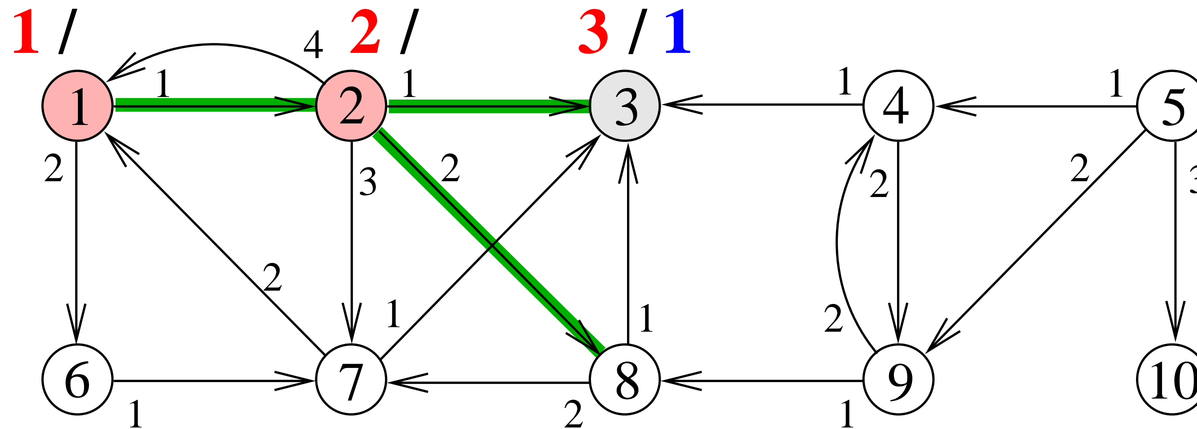


Baumkante, weil Knoten 3 **neu** ist.

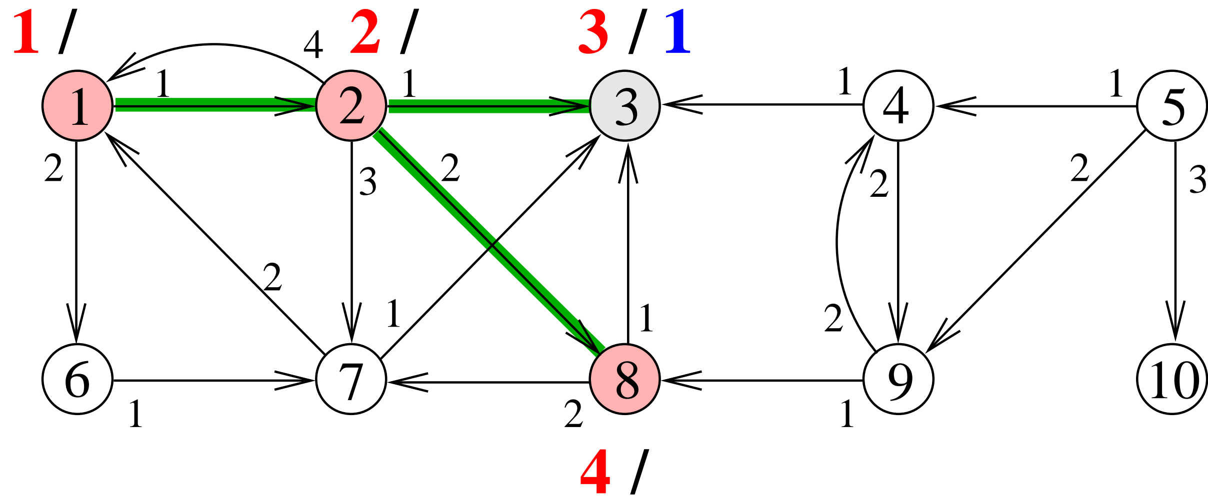


1 /
2 /
3 /
1
2
3
4
5
6
7
8
9
10

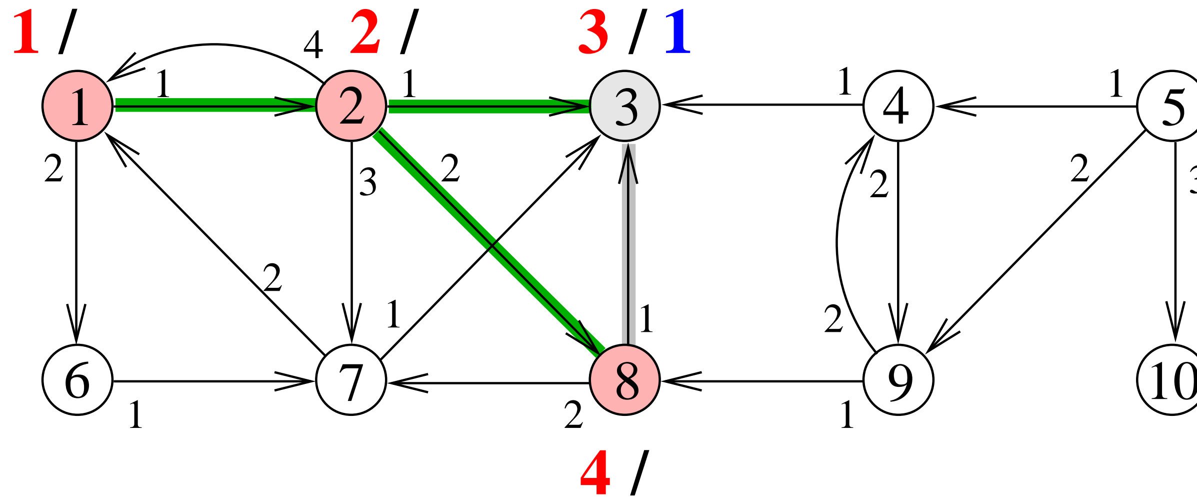

1 /
2 /
3 / 1
1
2
3
4
5
6
7
8
9
10



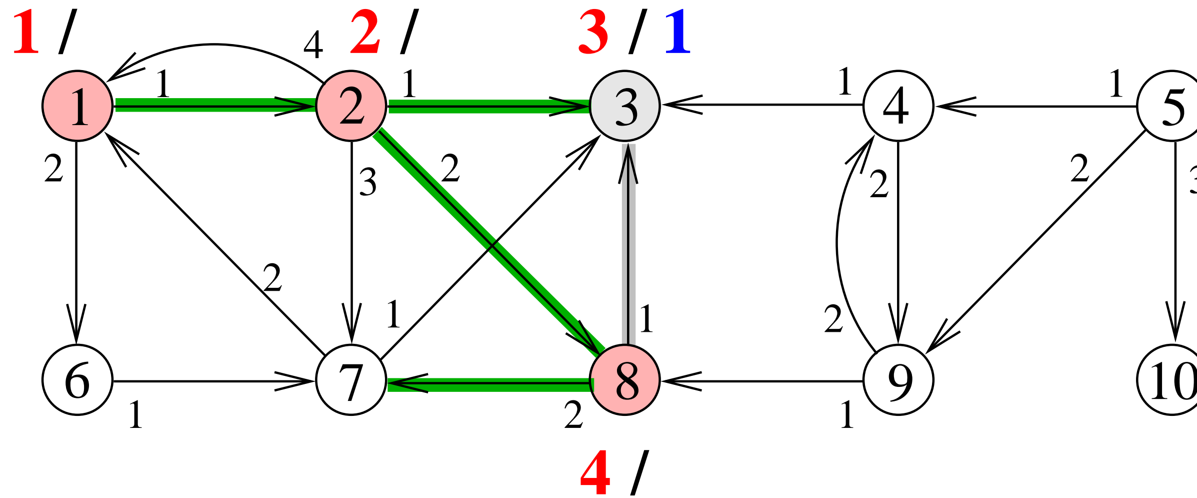
Baumkante, weil Knoten 8 **neu** ist.



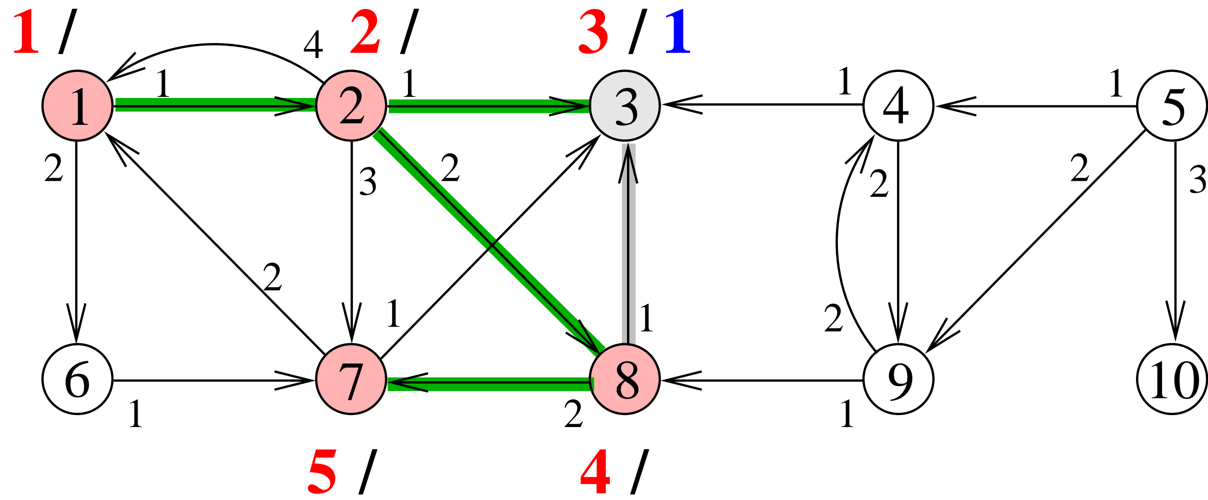
1 /
2 /
3 / 1
4 /
1
2
3
4
5
6
7
8
9
10

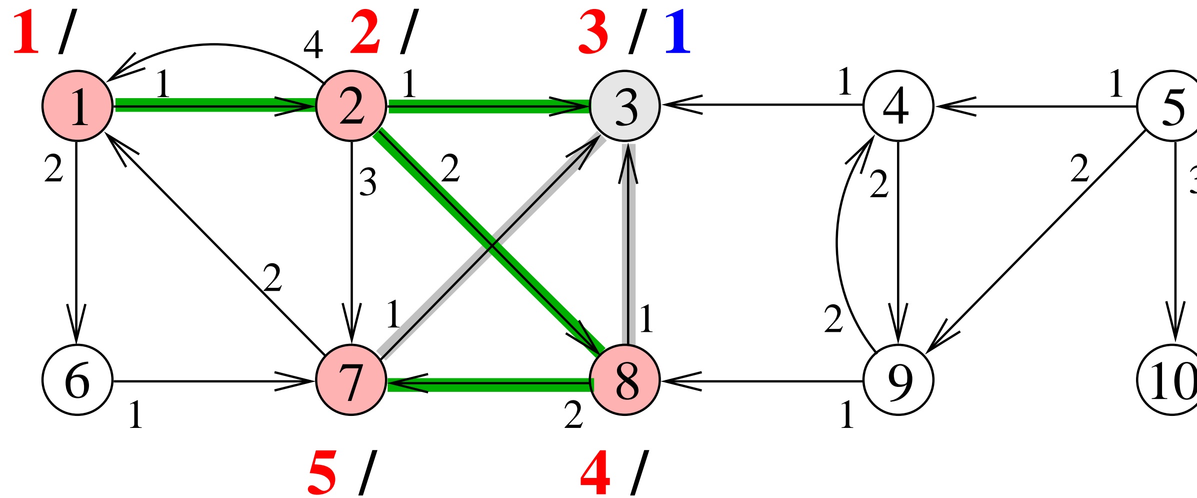


Querkante, weil Knoten 3 **fertig** ist und für die dfs-Nummern $\mathbf{4} > \mathbf{3}$ gilt.

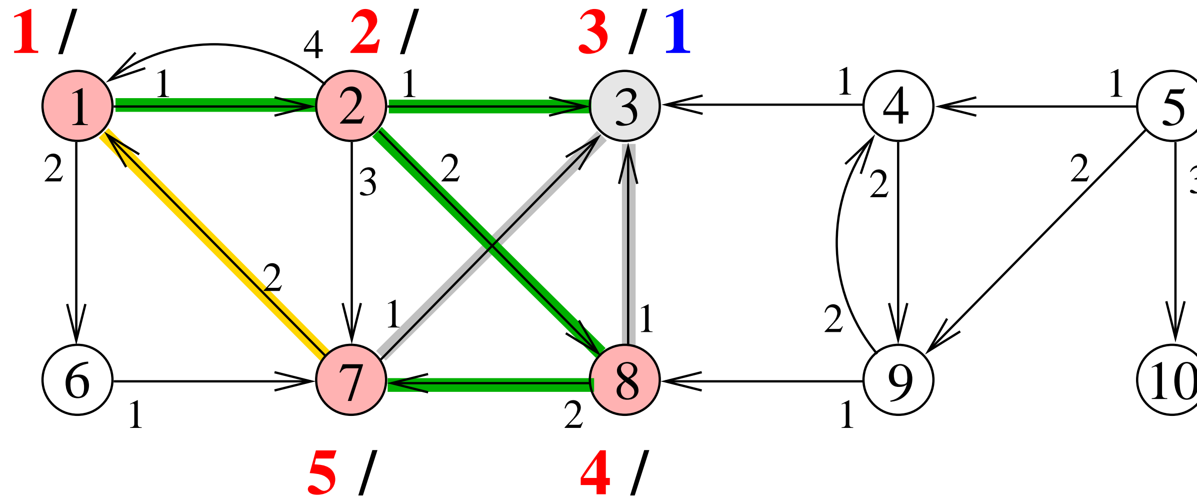


Baumkante, weil Knoten 7 **neu** ist.


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

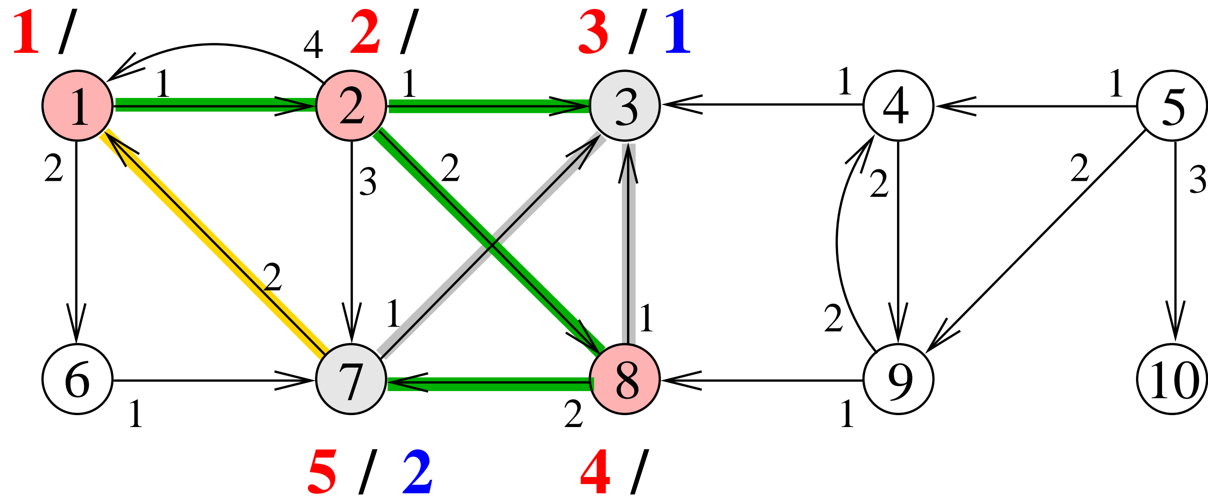


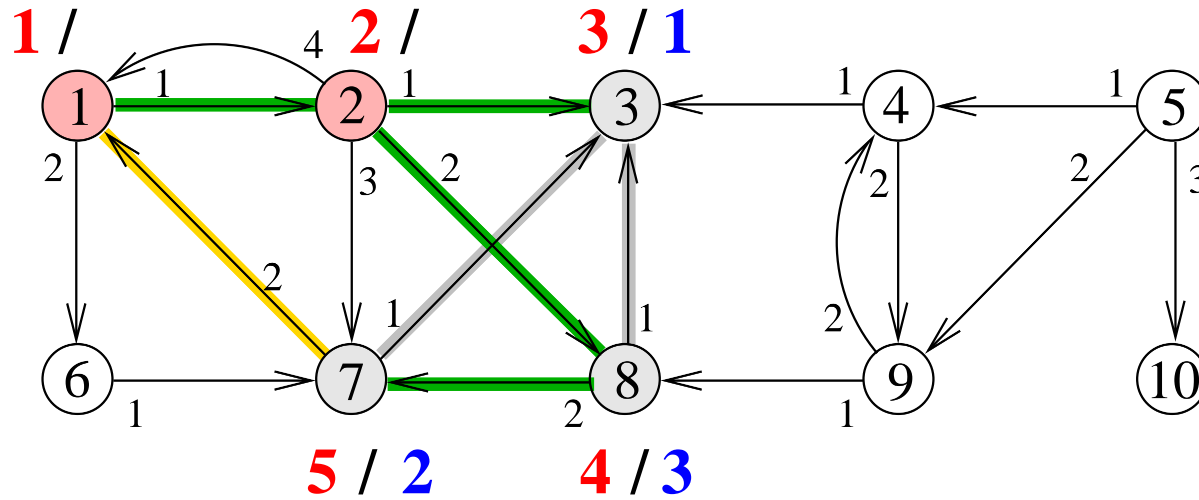
Querkante, weil Knoten 3 **fertig** ist und für die dfs-Nummern $\mathbf{5} > \mathbf{3}$ gilt.

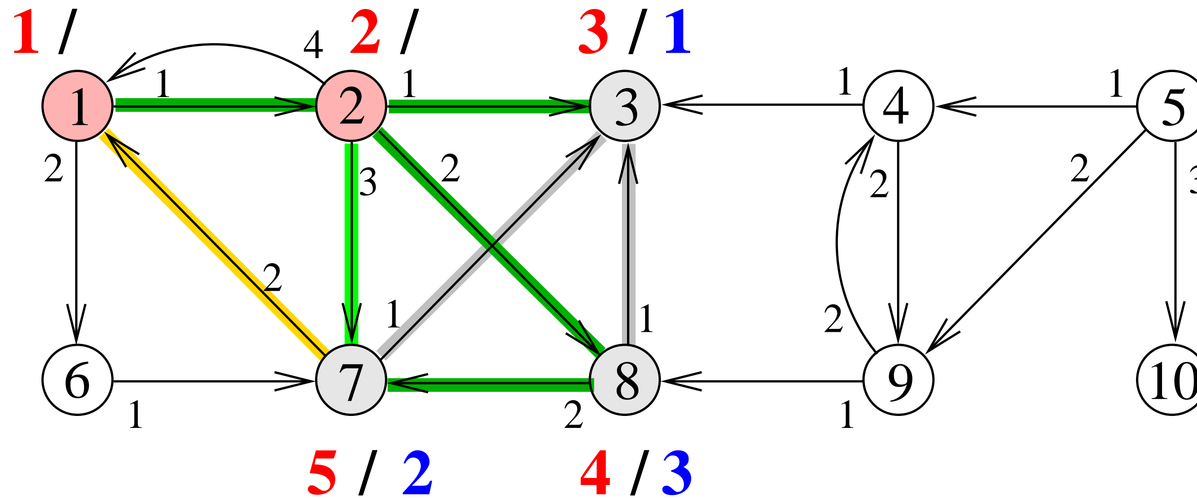


Rückwärtskante, weil Knoten 1 **aktiv** ist.

Knoten 1 ist Vorfahr von Knoten 7.

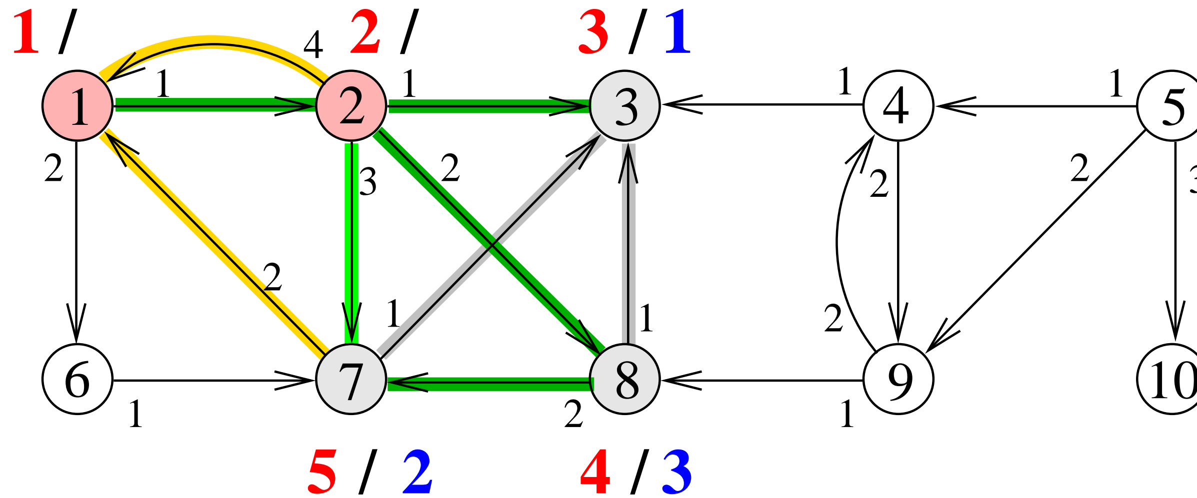

1 /
2 /
3 / 1
5 / 2
4 /
1
2
3
4
5
6
7
8
9
10


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

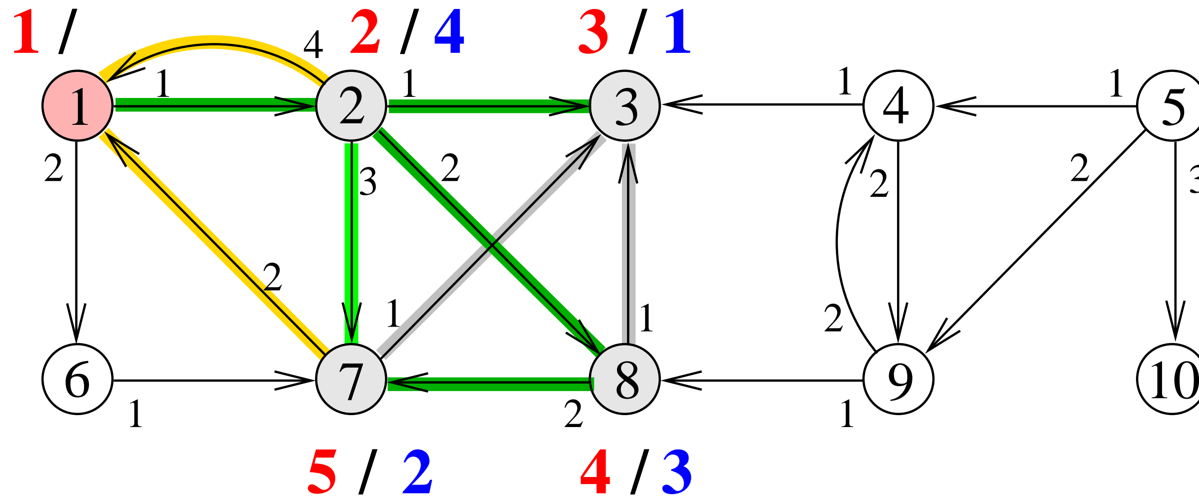


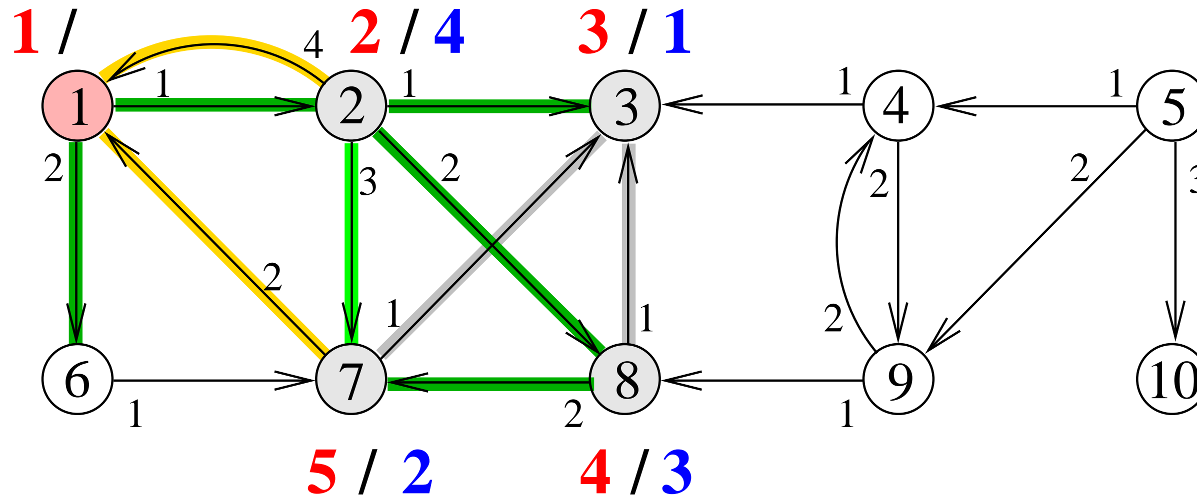
Vorwärtskante, weil Knoten 7 **fertig** ist und für die dfs-Nummern $\mathbf{2} < \mathbf{5}$ gilt.

Knoten 7 ist Nachfahr von Knoten 2.

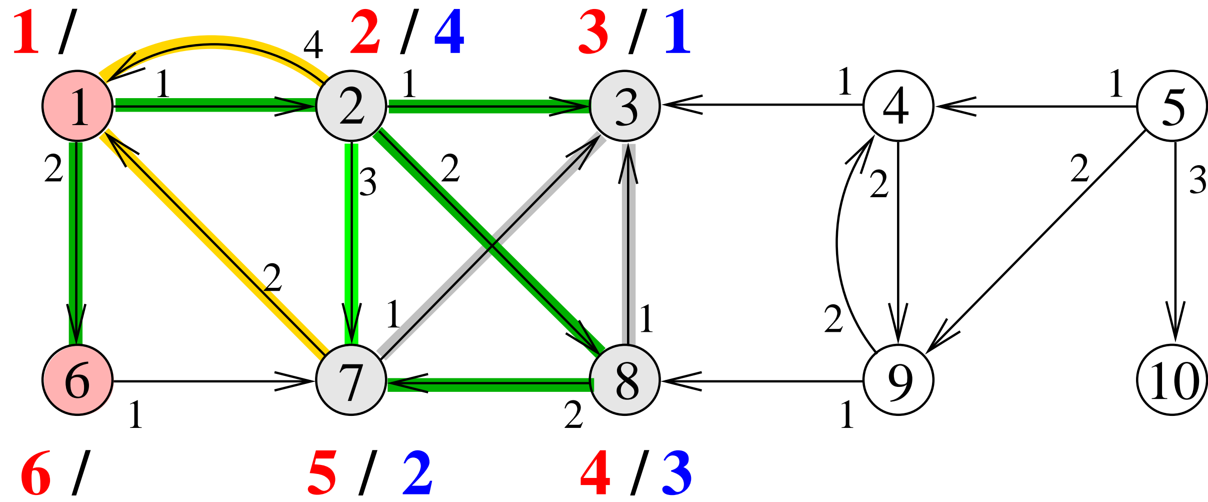


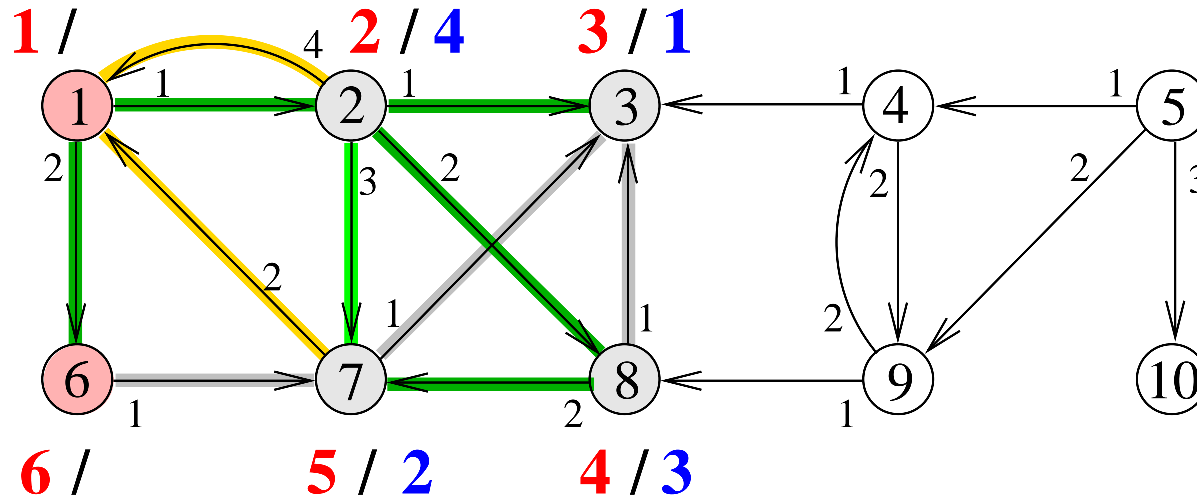
Rückwärtskante, weil Knoten 1 **aktiv** ist.


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

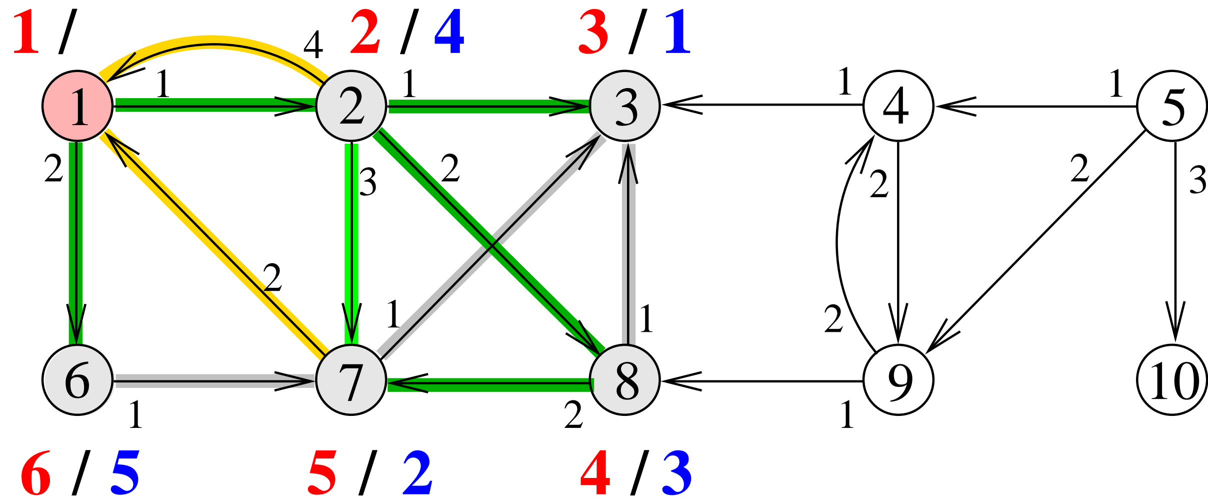


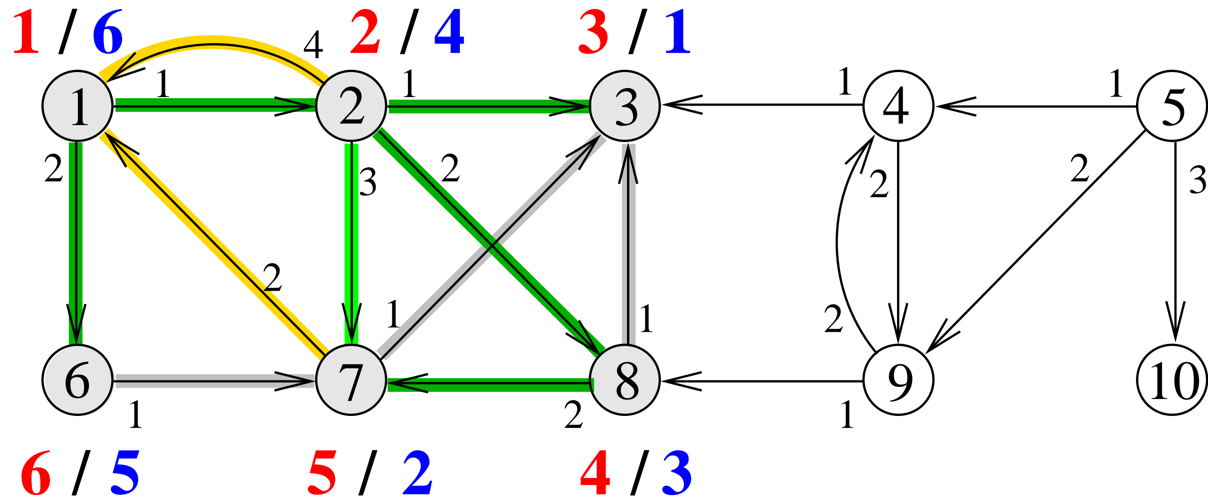
Baumkante, weil Knoten 6 **neu** ist.

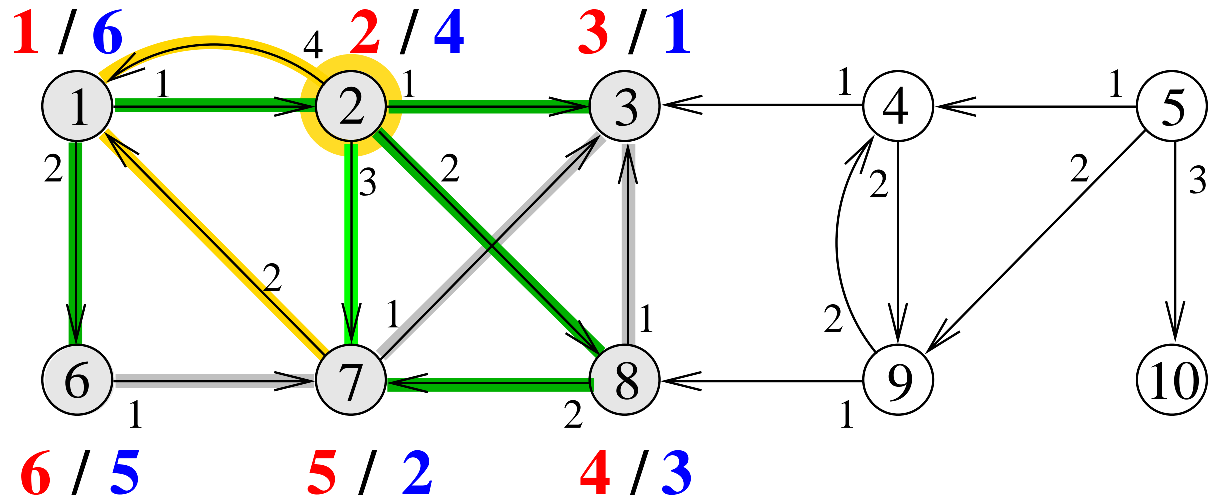

1 /
2 / 4
3 / 1
6 /
5 / 2
4 / 3
1
2
3
4
5
6
7
8
9
10

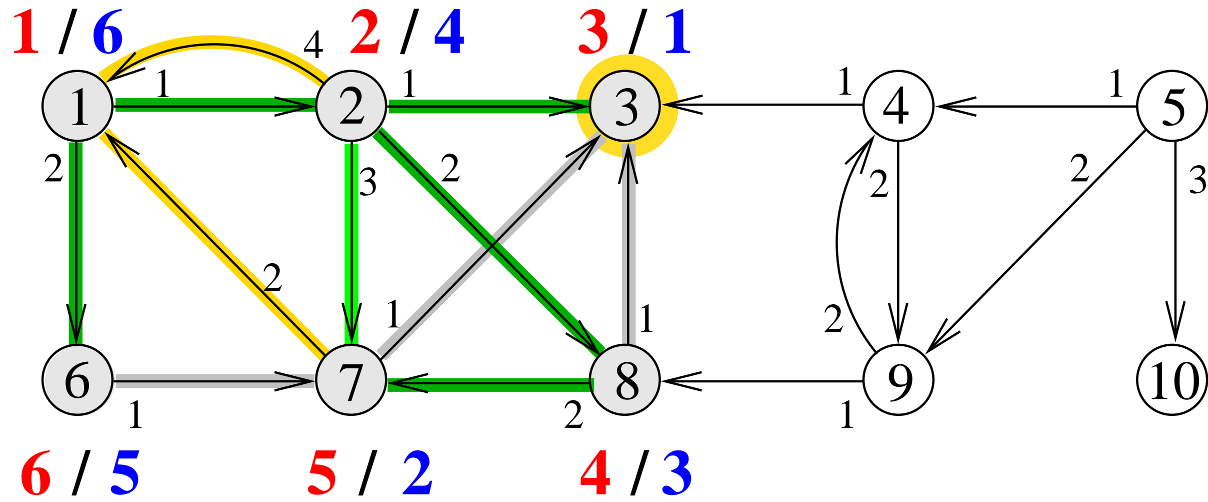


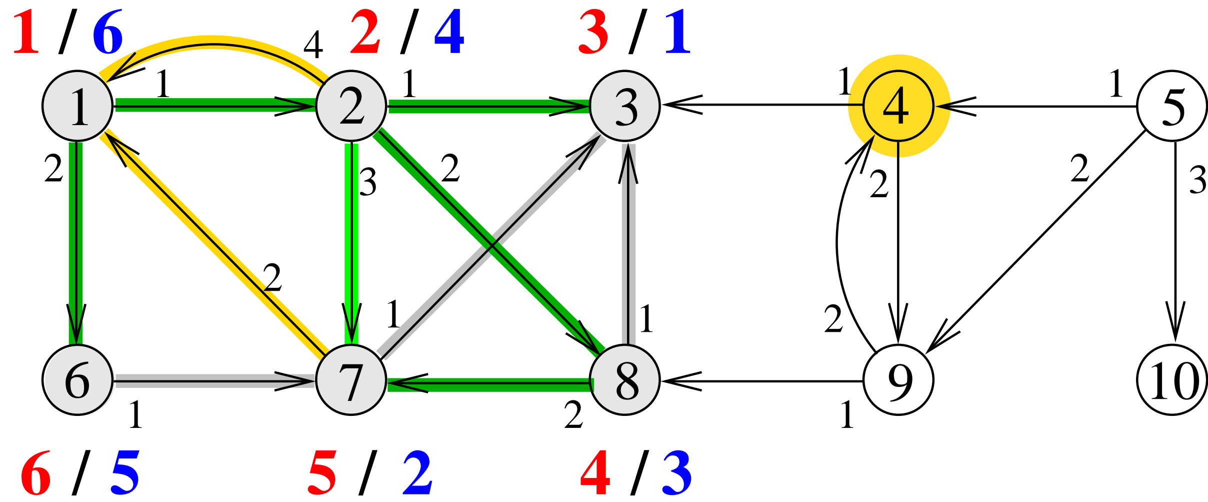
Querkante, weil Knoten 7 **fertig** ist und für die dfs-Nummern $6 > 5$ gilt.

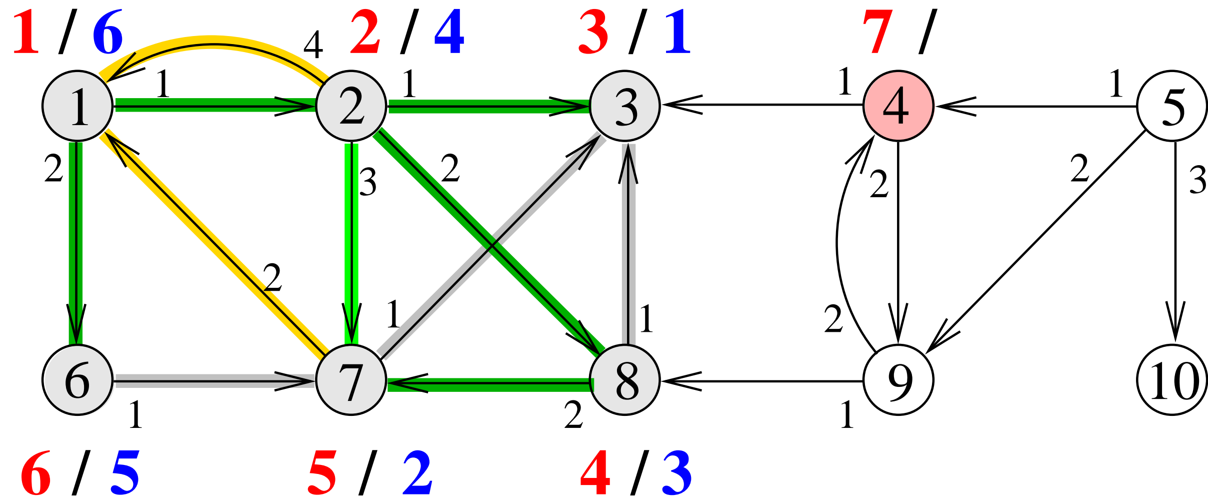

1 /
2 / 4
3 / 1
6 / 5
5 / 2
4 / 3
1
2
3
4
5
6
7
8
9
10

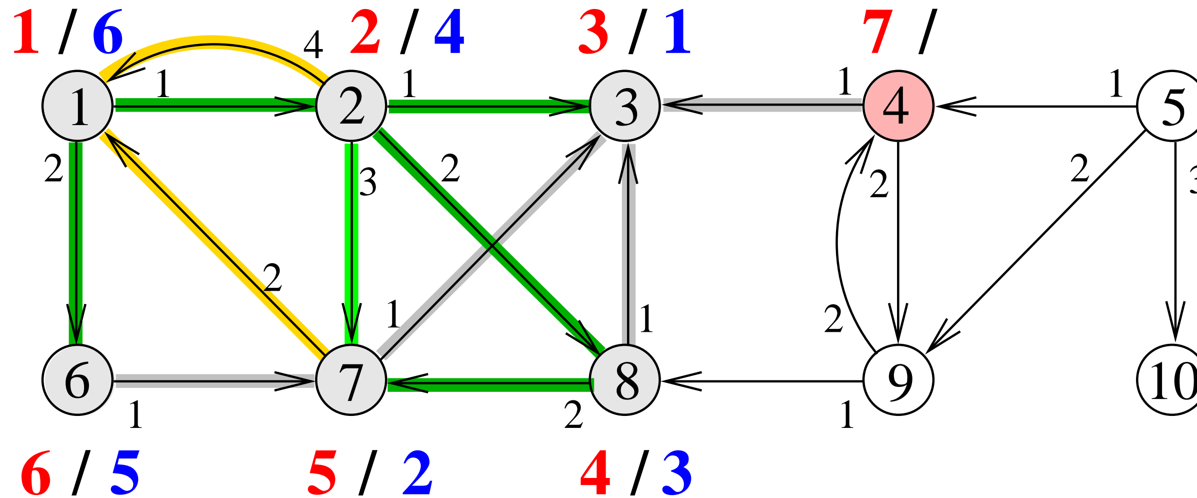

1 / 6
2 / 4
3 / 1
6 / 5
5 / 2
4 / 3
1
2
3
4
5
6
7
8
9
10


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

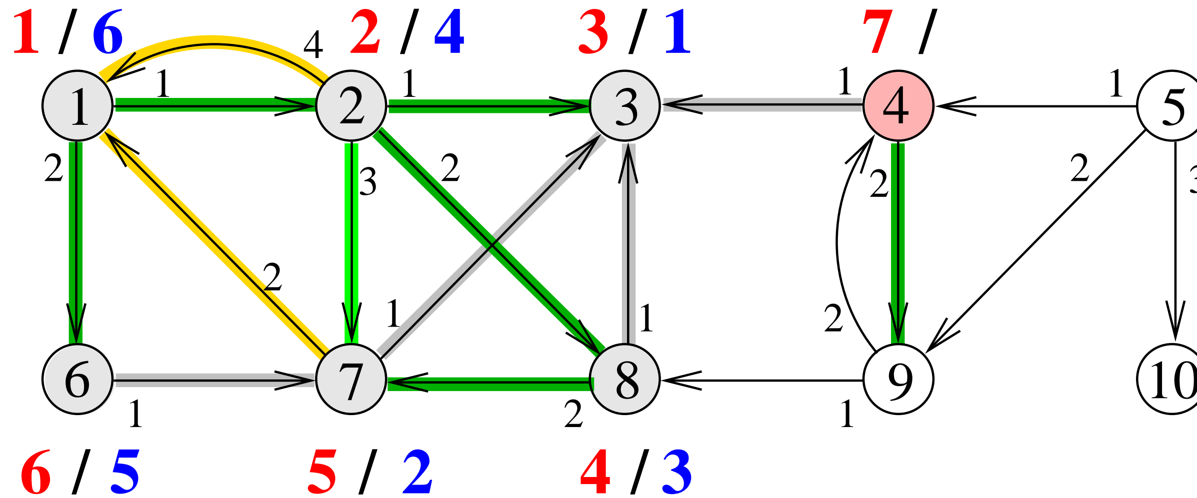

1 / 6
2 / 4
3 / 1
6 / 5
5 / 2
4 / 3
1
2
3
4
5
6
7
8
9
10


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

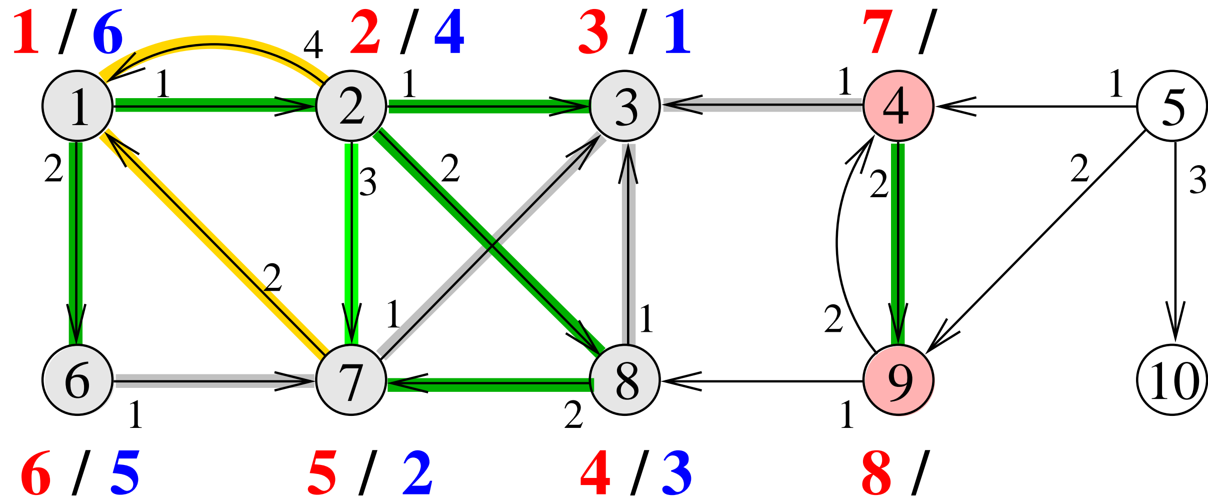

1 / 6
2 / 4
3 / 1
7 /
6 / 5
5 / 2
4 / 3
1
2
3
4
5
6
7
8
9
10



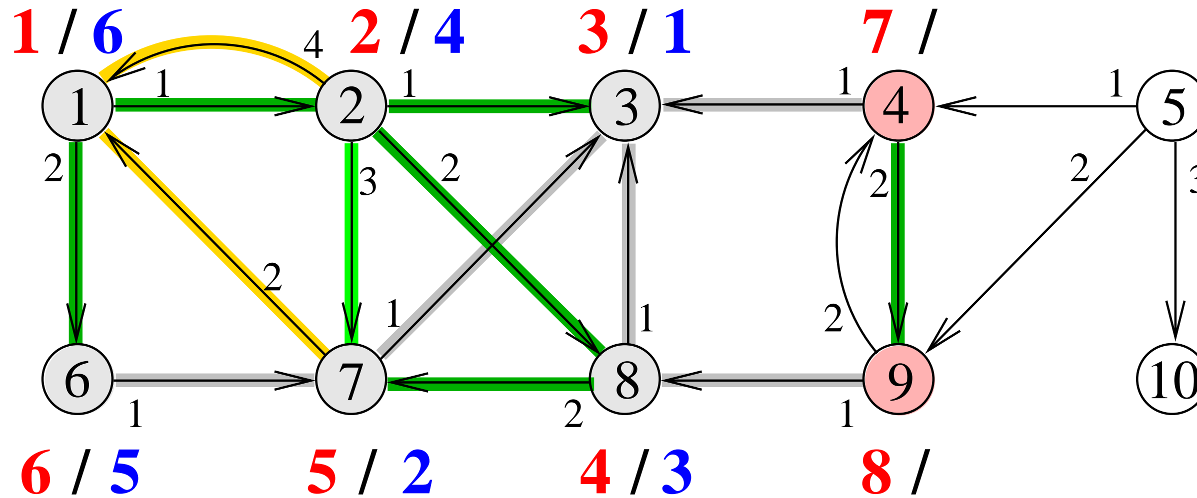
Querkante, weil Knoten 3 **fertig** ist und für die dfs-Nummern $7 > 3$ gilt.



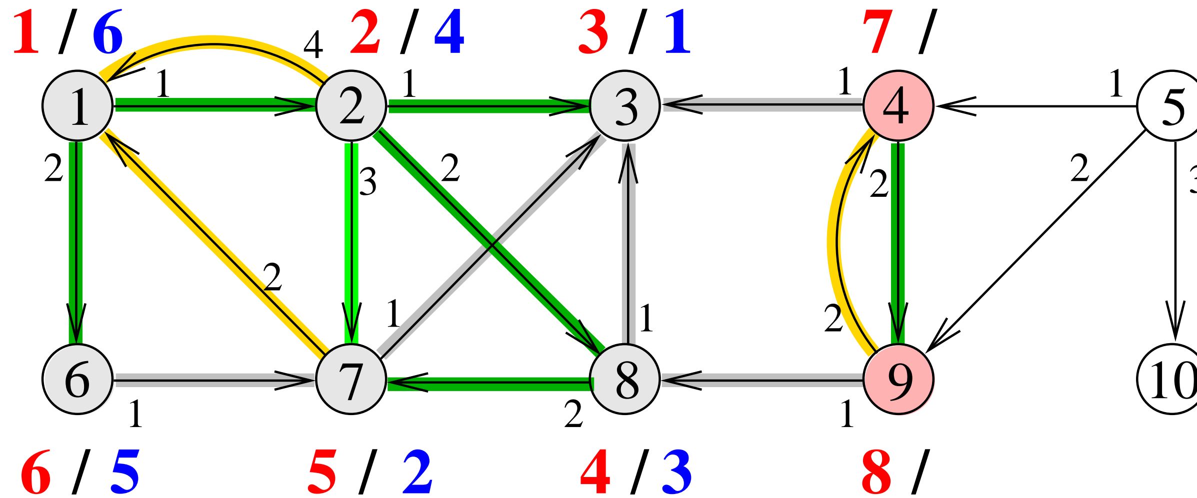
Baumkante, weil Knoten 9 **neu** ist.



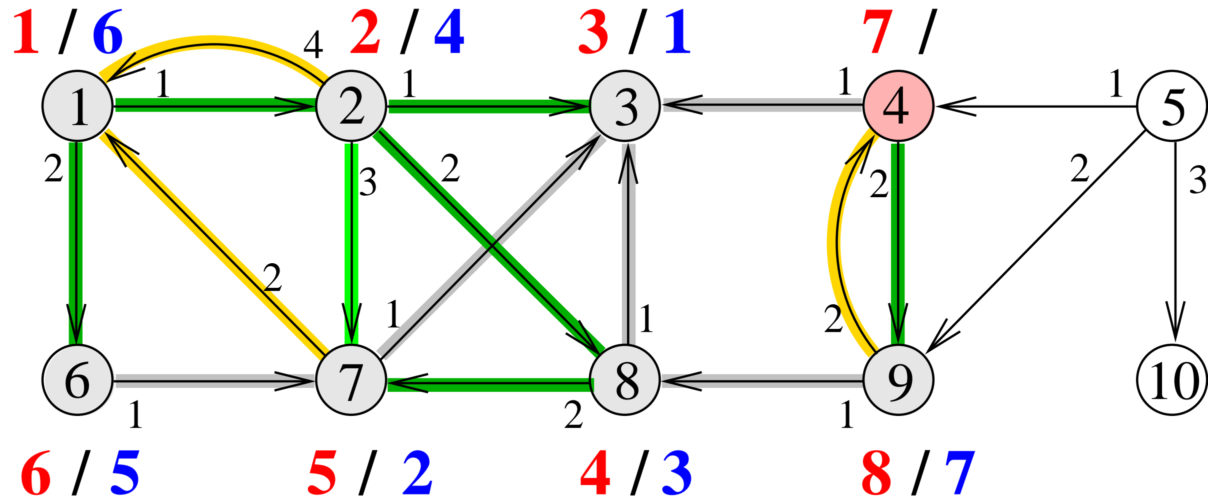
1 / 6
2 / 4
3 / 1
7 /
1
2
3
4
5
6
7
8
9
10
6 / 5
5 / 2
4 / 3
8 /

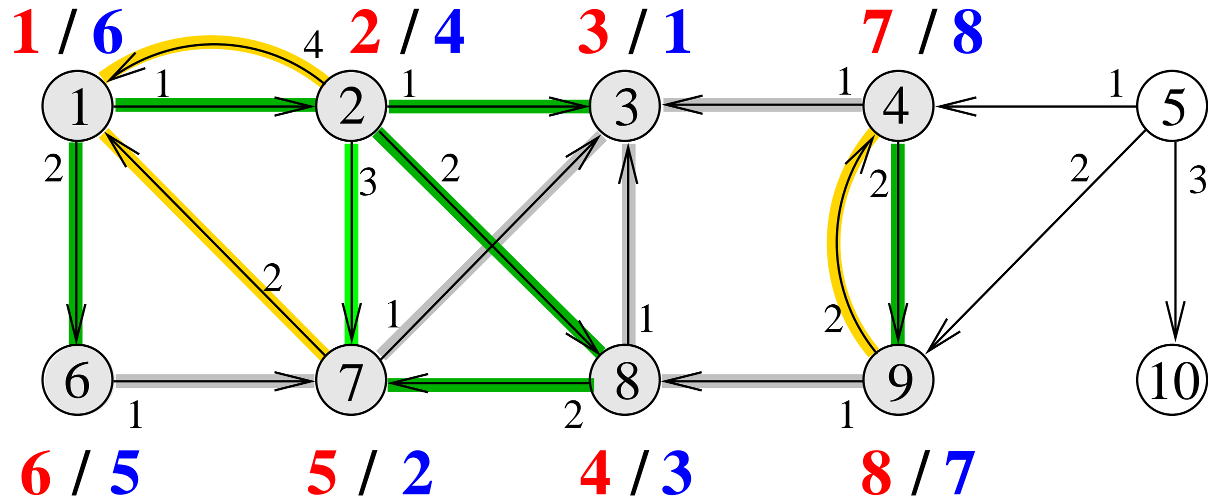


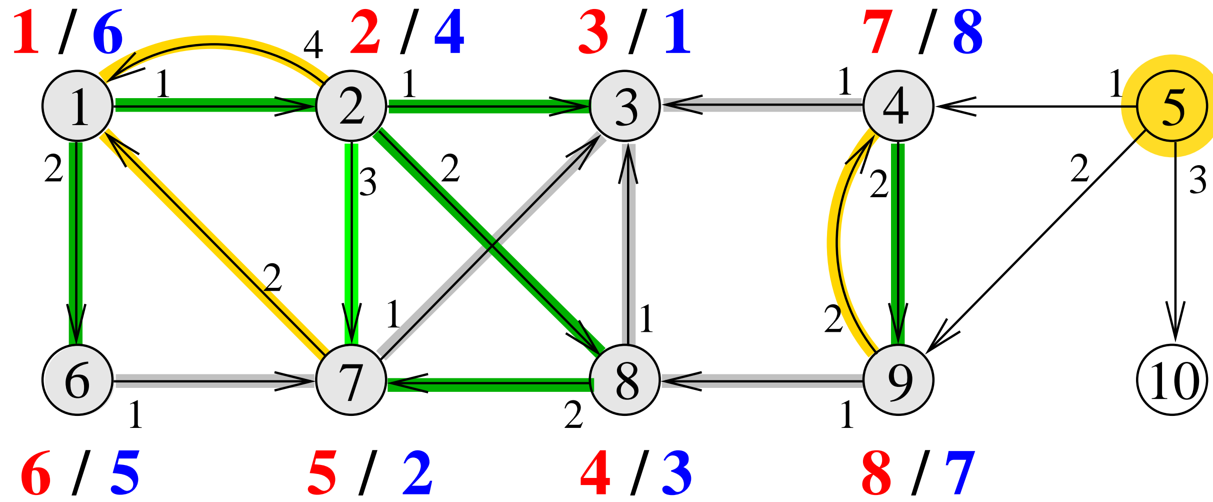
Querkante, weil Knoten 8 **fertig** ist und für die dfs-Nummern $8 > 4$ gilt.

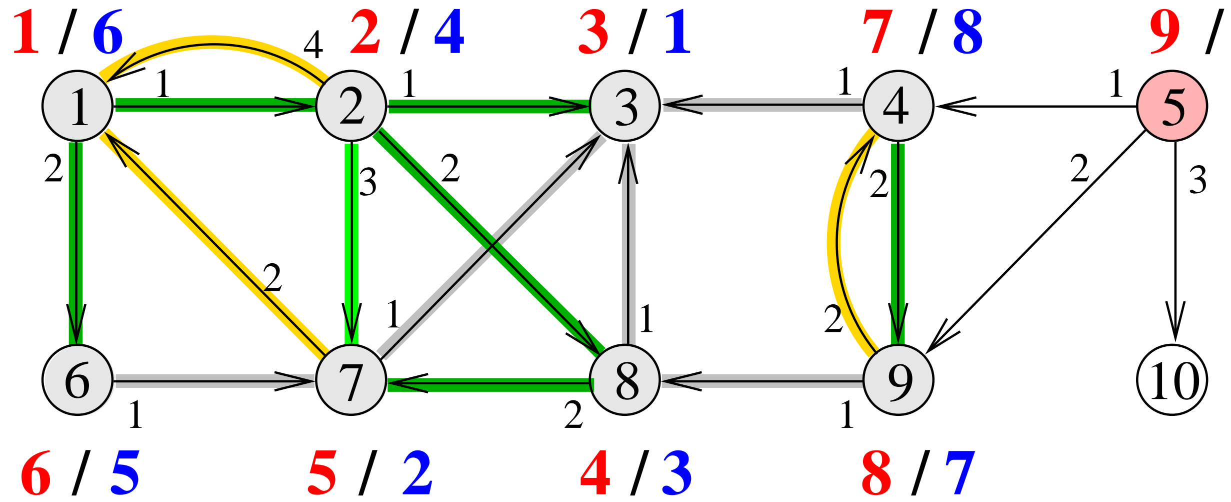


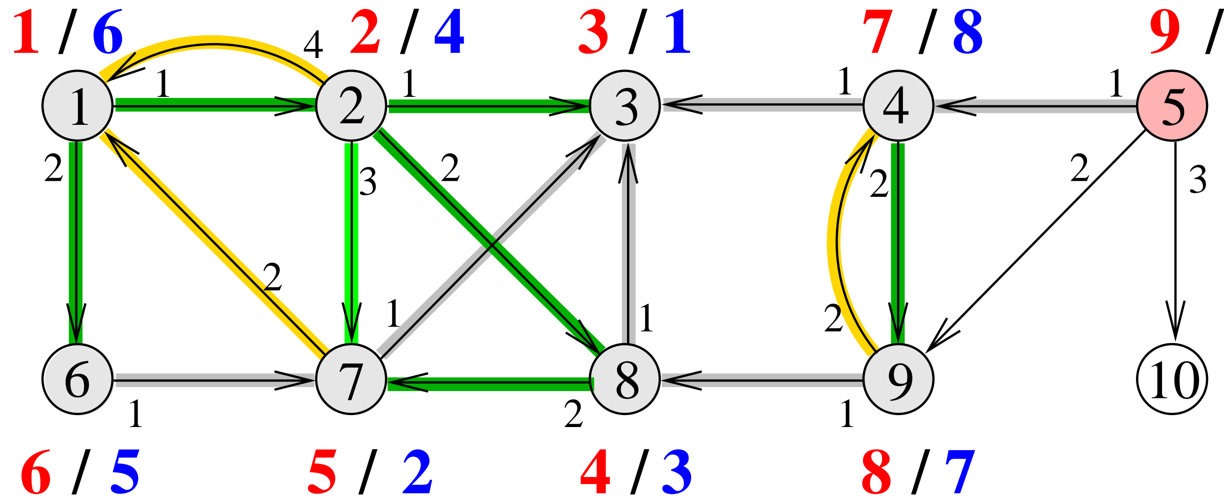
Rückwärtskante, weil Knoten 4 **aktiv** ist.


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

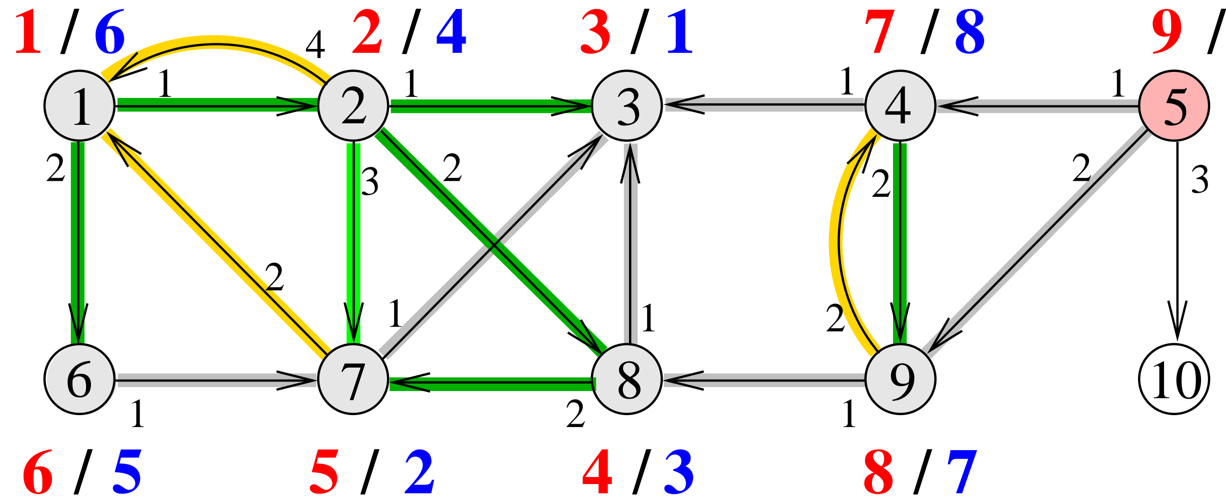

1 / 6
2 / 4
3 / 1
7 / 8
6 / 5
5 / 2
4 / 3
8 / 7
1
2
3
4
5
6
7
8
9
10


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

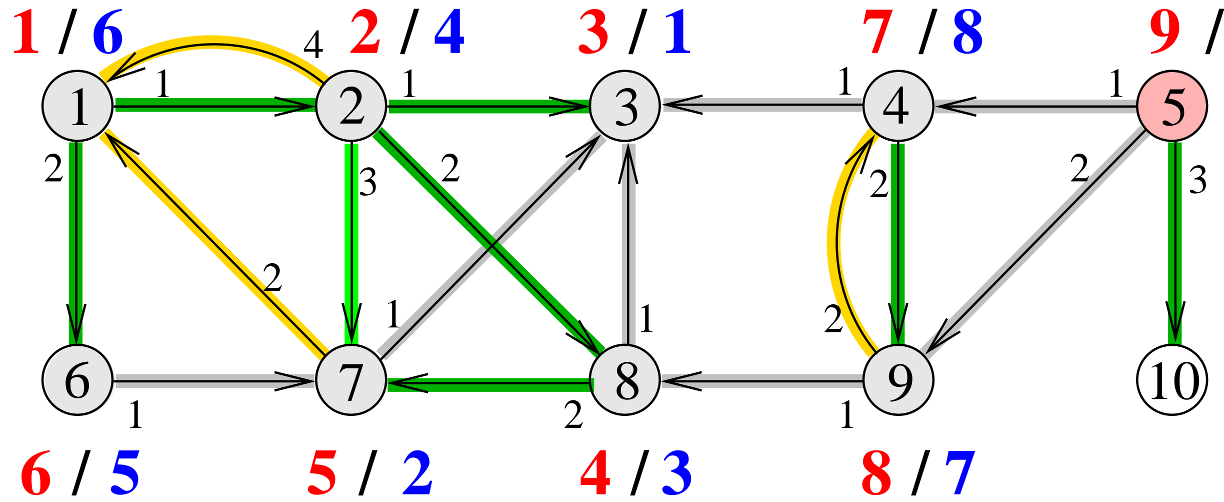

1 / 6
2 / 4
3 / 1
7 / 8
9 /
6 / 5
5 / 2
4 / 3
8 / 7
1
2
3
4
5
6
7
8
9
10



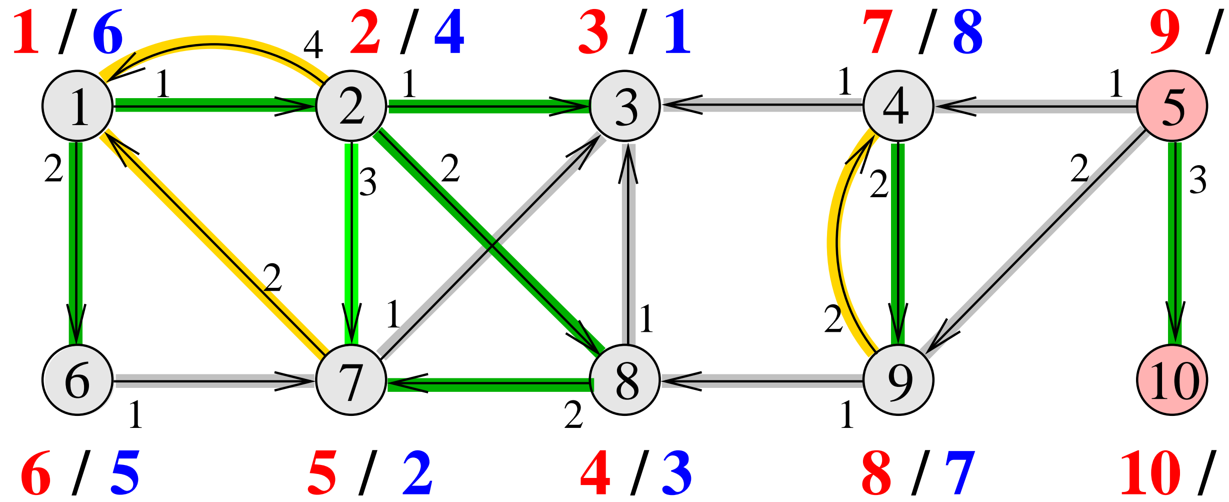
Querkante, weil Knoten 4 **fertig** ist und für die dfs-Nummern $9 > 7$ gilt.

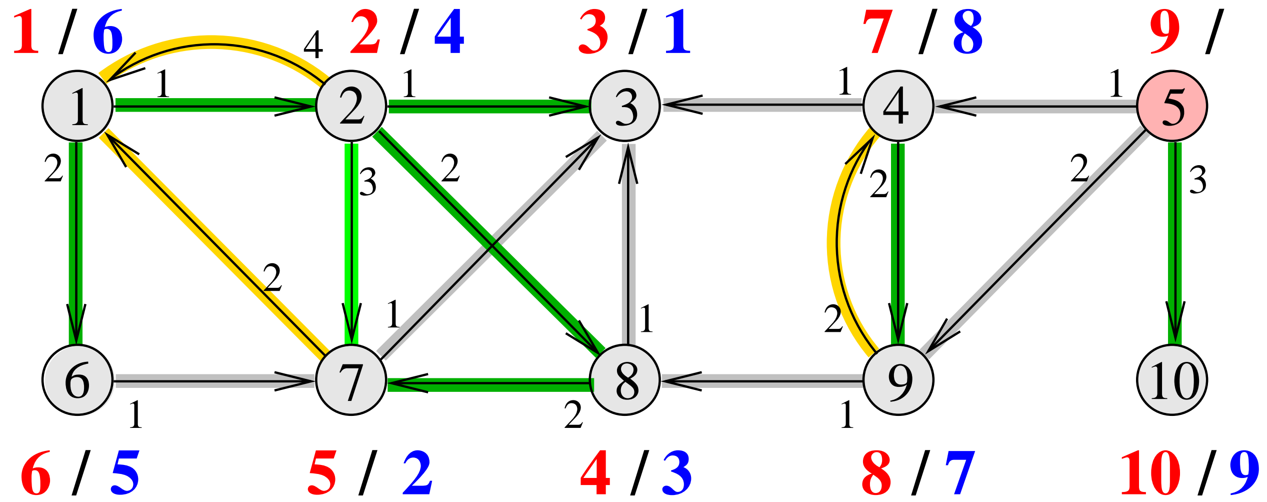


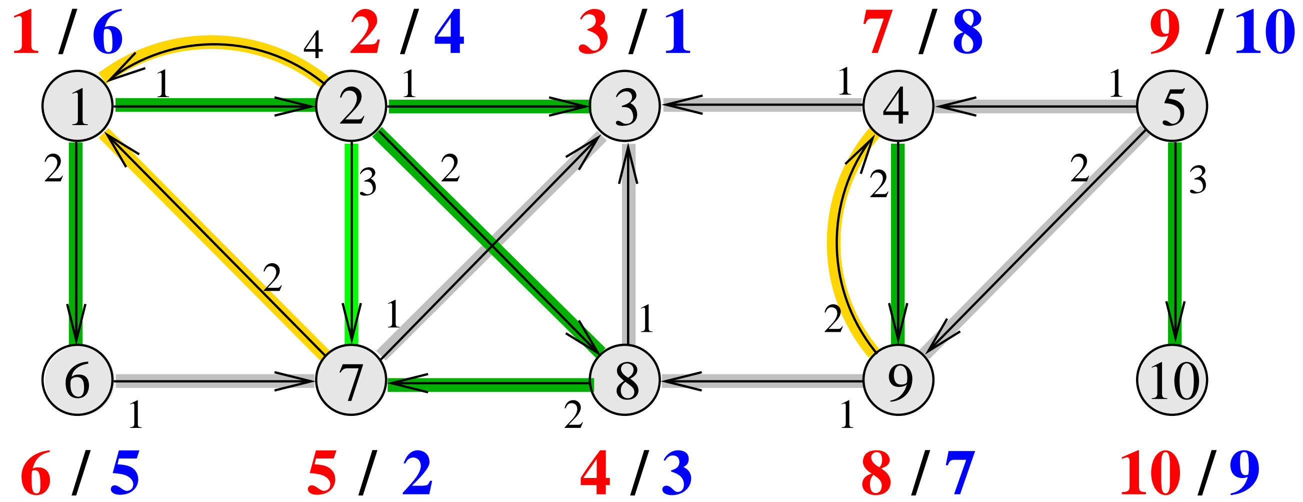
Querkante, weil Knoten 9 **fertig** ist und für die dfs-Nummern $9 > 8$ gilt.

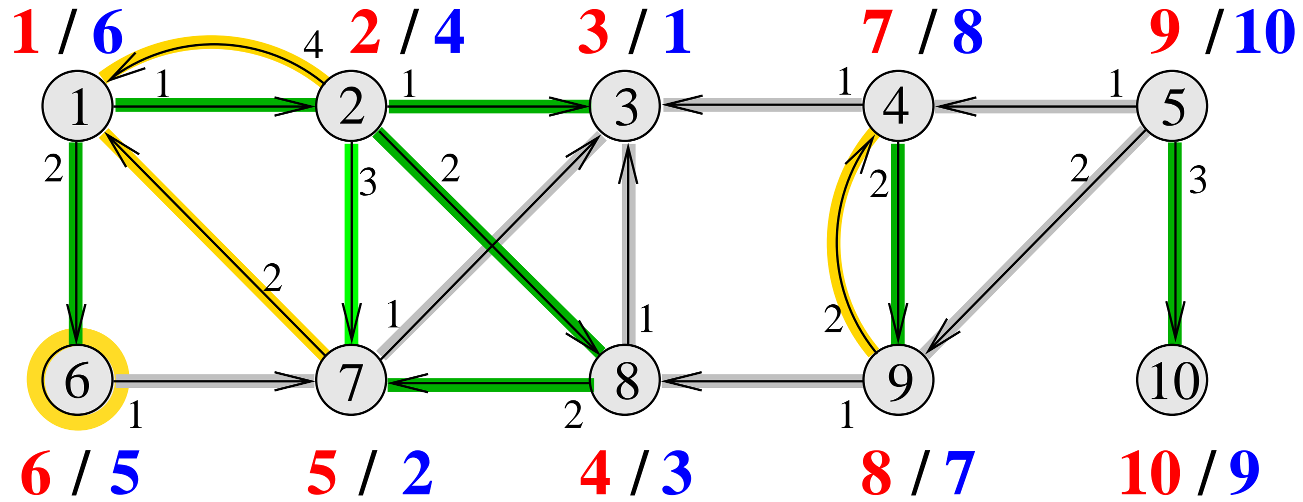


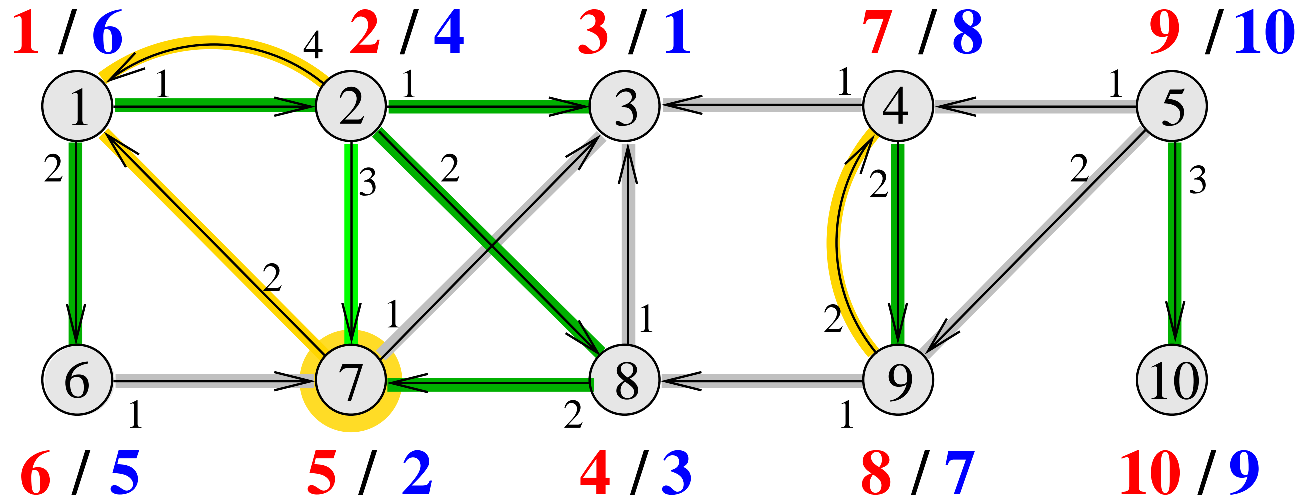
Baumkante, weil Knoten 10 **neu** ist.

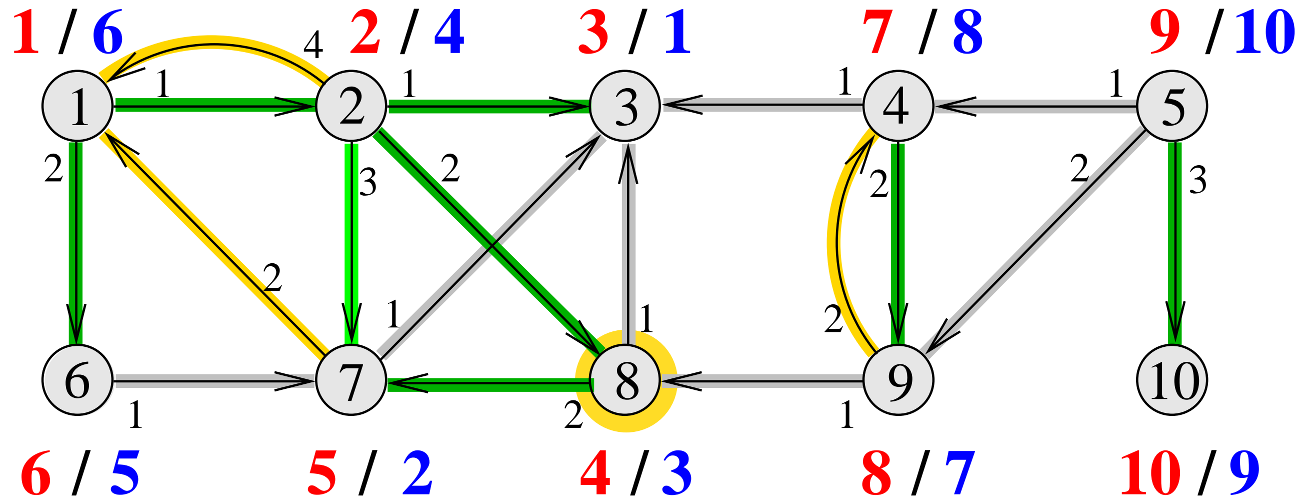

1 / 6
2 / 4
3 / 1
7 / 8
9 /
1
2
3
4
5
6
7
8
9
10
6 / 5
5 / 2
4 / 3
8 / 7
10 /

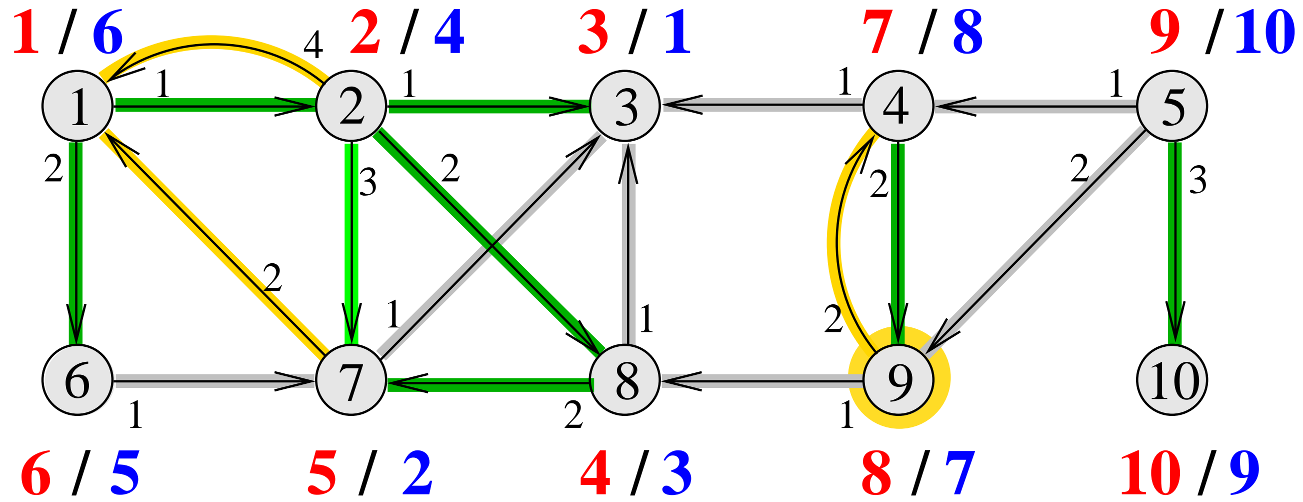

1 / 6
2 / 4
3 / 1
7 / 8
9 /
1
2
3
4
5
6
7
8
9
10
6 / 5
5 / 2
4 / 3
8 / 7
10 / 9


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

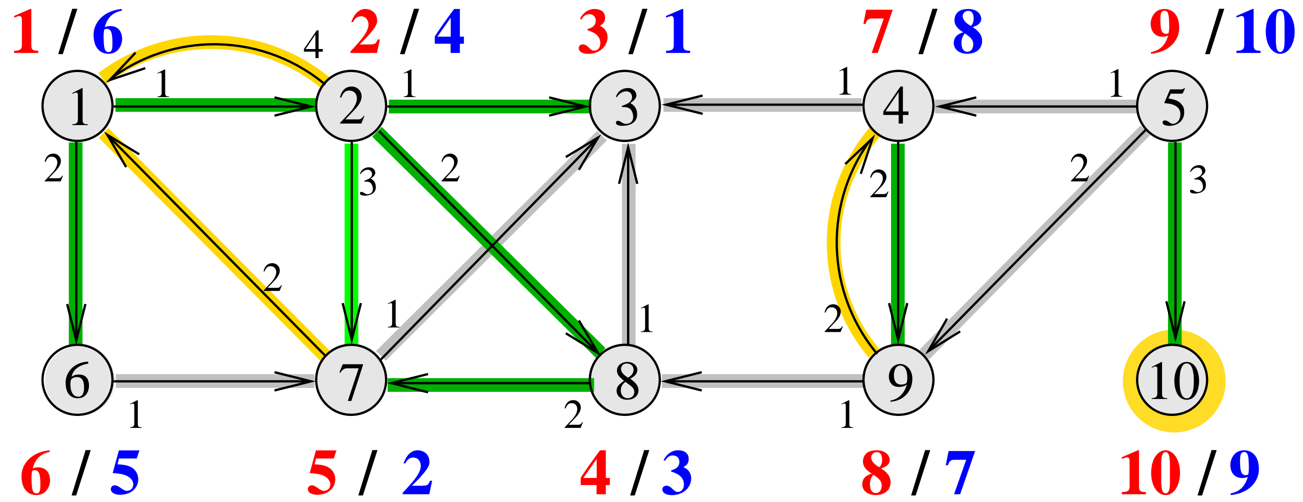

1 / 6
2 / 4
3 / 1
7 / 8
9 / 10
1
2
3
4
5
6
7
8
9
10
6 / 5
5 / 2
4 / 3
8 / 7
10 / 9

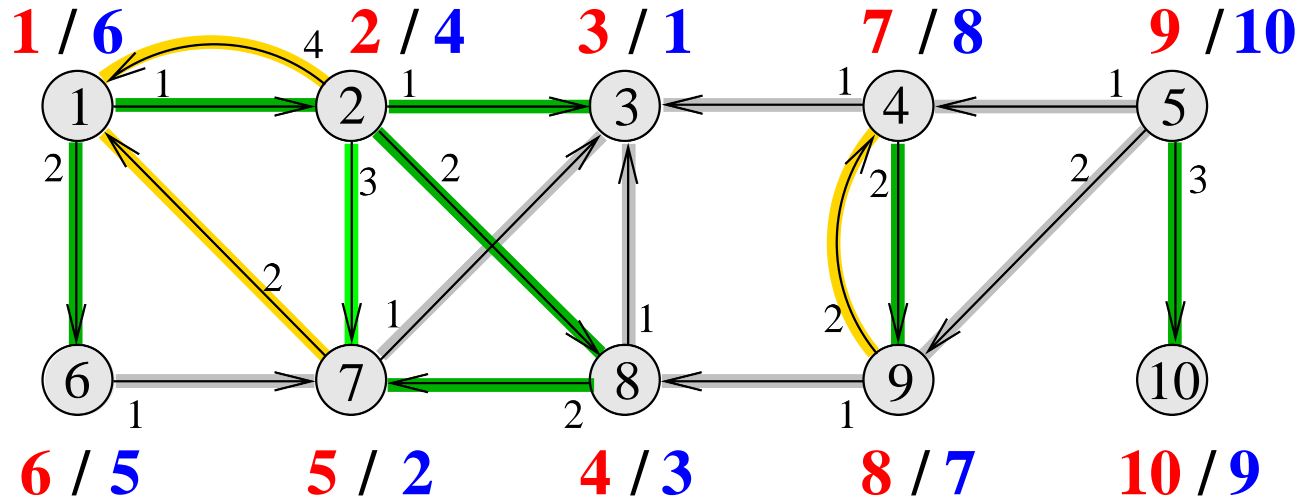

1 / 6
2 / 4
3 / 1
7 / 8
9 / 10
1
2
3
4
5
6
7
8
9
10
6 / 5
5 / 2
4 / 3
8 / 7
10 / 9

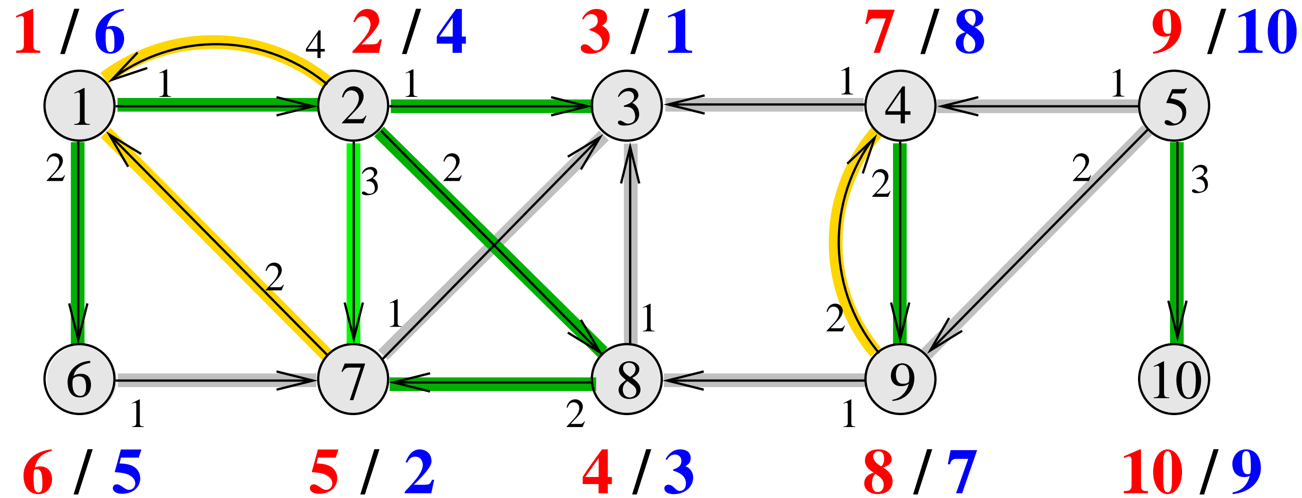

1 / 6
2 / 4
3 / 1
7 / 8
9 / 10
1
2
3
4
5
6
7
8
9
10
6 / 5
5 / 2
4 / 3
8 / 7
10 / 9



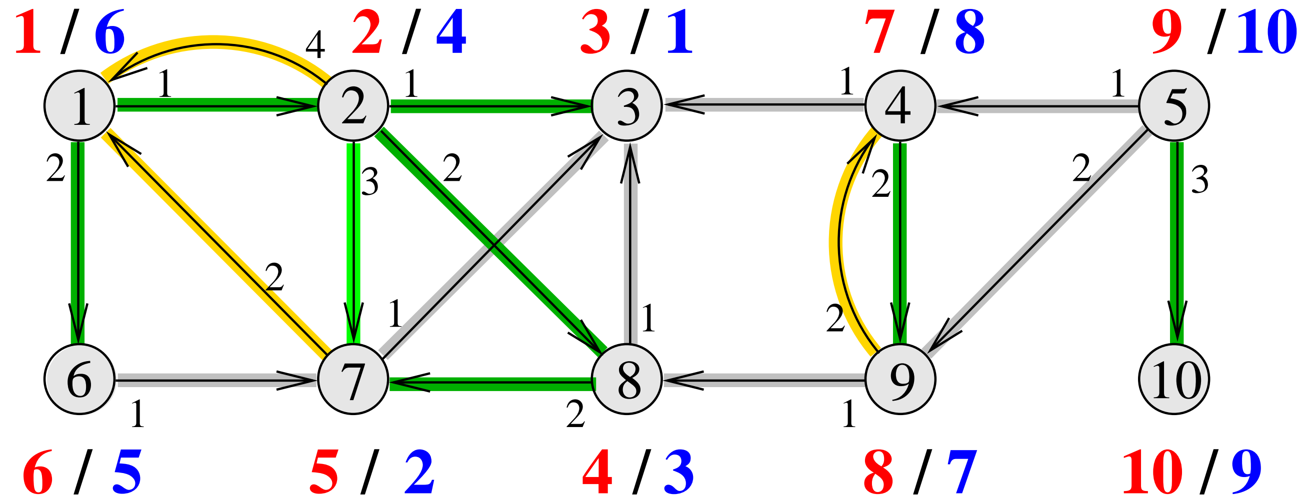
1 / 6
2 / 4
3 / 1
7 / 8
9 / 10
1
2
3
4
5
6
7
8
9
10
6 / 5
5 / 2
4 / 3
8 / 7
10 / 9


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


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

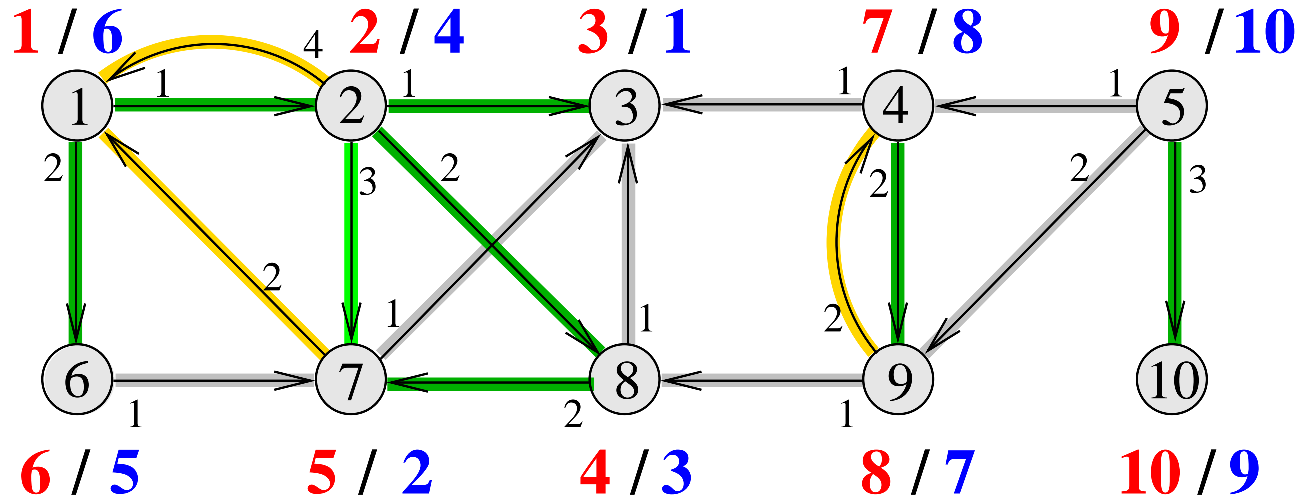


Klar: Alle Aussagen für die einfache dfs-Prozedur bleiben gültig.



Klar: Alle Aussagen für die einfache dfs-Prozedur bleiben gültig.

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



Klar: Alle Aussagen für die einfache dfs-Prozedur bleiben gültig.

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: Folgende Aussagen sind äquivalent:

(a) dfs(w) beginnt nach dfs(v) und endet vor dfs(v)
 (d. h. dfs(w) wird (direkt oder indirekt) von dfs(v) aus aufgerufen);

Beobachtung 3: Folgende Aussagen sind äquivalent:

(a) dfs(w) beginnt nach dfs(v) und endet vor dfs(v)
(d. h. dfs(w) wird (direkt oder indirekt) von dfs(v) aus aufgerufen);

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

Beobachtung 3: Folgende Aussagen sind äquivalent:

(a) dfs(w) beginnt nach dfs(v) und endet vor dfs(v)
(d. h. dfs(w) wird (direkt oder indirekt) von dfs(v) aus aufgerufen);

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

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

Beobachtung 3: Folgende Aussagen sind äquivalent:

(a) dfs(w) beginnt nach dfs(v) und endet vor dfs(v)
(d. h. dfs(w) wird (direkt oder indirekt) von dfs(v) aus aufgerufen);

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

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

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

Satz 8.2.1

Der Algorithmus DFS(G) klassifiziert die Kanten korrekt.

Satz 8.2.1

Der Algorithmus DFS(G) klassifiziert die Kanten korrekt.

Beweis: Sei (v, w) eine beliebige Kante in G.

Satz 8.2.1

Der Algorithmus DFS(G) klassifiziert die Kanten korrekt.

Beweis: Sei (v, w) eine beliebige Kante in G.

1. Fall: dfs(w) wird direkt aus dfs(v) aufgerufen.

Dann ist (v, w) **Baumkante** und wird richtig als $\boldsymbol{T}$ klassifiziert.

Satz 8.2.1

Der Algorithmus DFS(G) klassifiziert die Kanten korrekt.

Beweis: Sei (v, w) eine beliebige Kante in G.

1. Fall: dfs(w) wird direkt aus dfs(v) aufgerufen.

Dann ist (v, w) **Baumkante** und wird richtig als $\boldsymbol{T}$ klassifiziert.

2. Fall: Die Untersuchung der Kante (v, w) in dfs(v) ergibt, dass w **aktiv** ist.

Satz 8.2.1

Der Algorithmus DFS(G) klassifiziert die Kanten korrekt.

Beweis: Sei (v, w) eine beliebige Kante in G.

1. Fall: dfs(w) wird direkt aus dfs(v) aufgerufen.

Dann ist (v, w) **Baumkante** und wird richtig als $\boldsymbol{T}$ klassifiziert.

2. Fall: Die Untersuchung der Kante (v, w) in dfs(v) ergibt, dass w **aktiv** ist.

Dann liegt w auf dem **roten Weg** von der Wurzel des Tiefensuchbaums zu v, ist also Vorfahr von v (oder gleich v – Schleifen sind ja nicht verboten).

Satz 8.2.1

Der Algorithmus DFS(G) klassifiziert die Kanten korrekt.

Beweis: Sei (v, w) eine beliebige Kante in G.

1. Fall: dfs(w) wird direkt aus dfs(v) aufgerufen.

Dann ist (v, w) **Baumkante** und wird richtig als $\boldsymbol{T}$ klassifiziert.

2. Fall: Die Untersuchung der Kante (v, w) in dfs(v) ergibt, dass w **aktiv** ist.

Dann liegt w auf dem **roten Weg** von der Wurzel des Tiefensuchbaums zu v, ist also Vorfahr von v (oder gleich v – Schleifen sind ja nicht verboten).
Daher ist (v, w) **Rückwärtskante** und wird richtig als $\boldsymbol{B}$ klassifiziert.

3. Fall: Die Untersuchung der Kante (v, w) in dfs(v) ergibt, dass w **fertig** ist und $\texttt{dfs_num}[v] < \texttt{dfs_num}[w]$ gilt.

3. Fall: Die Untersuchung der Kante (v, w) in dfs(v) ergibt, dass w **fertig** ist und $\texttt{dfs_num}[v] < \texttt{dfs_num}[w]$ gilt.

Weil dfs(v) noch aktiv ist, wird $\texttt{f_num}[v] > \texttt{f_num}[w]$.

3. Fall: Die Untersuchung der Kante (v, w) in dfs(v) ergibt, dass w **fertig** ist und $\texttt{dfs_num}[v] < \texttt{dfs_num}[w]$ gilt.

Weil dfs(v) noch aktiv ist, wird $\texttt{f_num}[v] > \texttt{f_num}[w]$.
Dann ist w nach Beobachtung 3 Nachfahr von v im Tiefensuchwald,
also ist (v, w) **Vorwärtskante** und wird richtig als $\boldsymbol{F}$ klassifiziert.

3. Fall: Die Untersuchung der Kante (v, w) in dfs(v) ergibt, dass w **fertig** ist und $\texttt{dfs_num}[v] < \texttt{dfs_num}[w]$ gilt.

Weil dfs(v) noch aktiv ist, wird $\texttt{f_num}[v] > \texttt{f_num}[w]$.
Dann ist w nach Beobachtung 3 Nachfahr von v im Tiefensuchwald, also ist (v, w) **Vorwärtskante** und wird richtig als $\boldsymbol{F}$ klassifiziert.

4. Fall: Die Untersuchung der Kante (v, w) in dfs(v) ergibt, dass w **fertig** ist und $\texttt{dfs_num}[v] > \texttt{dfs_num}[w]$ gilt.

3. Fall: Die Untersuchung der Kante (v, w) in dfs(v) ergibt, dass w **fertig** ist und $\texttt{dfs_num}[v] < \texttt{dfs_num}[w]$ gilt.

Weil dfs(v) noch aktiv ist, wird $\texttt{f_num}[v] > \texttt{f_num}[w]$.
Dann ist w nach Beobachtung 3 Nachfahr von v im Tiefensuchwald,
also ist (v, w) **Vorwärtskante** und wird richtig als $\boldsymbol{F}$ klassifiziert.

4. Fall: Die Untersuchung der Kante (v, w) in dfs(v) ergibt, dass w **fertig** ist und $\texttt{dfs_num}[v] > \texttt{dfs_num}[w]$ gilt.

Weil dfs(v) noch aktiv ist, wird $\texttt{f_num}[v] > \texttt{f_num}[w]$.

3. Fall: Die Untersuchung der Kante (v, w) in dfs(v) ergibt, dass w **fertig** ist und $\texttt{dfs_num}[v] < \texttt{dfs_num}[w]$ gilt.

Weil dfs(v) noch aktiv ist, wird $\texttt{f_num}[v] > \texttt{f_num}[w]$.
Dann ist w nach Beobachtung 3 Nachfahr von v im Tiefensuchwald,
also ist (v, w) **Vorwärtskante** und wird richtig als $\boldsymbol{F}$ klassifiziert.

4. Fall: Die Untersuchung der Kante (v, w) in dfs(v) ergibt, dass w **fertig** ist und $\texttt{dfs_num}[v] > \texttt{dfs_num}[w]$ gilt.

Weil dfs(v) noch aktiv ist, wird $\texttt{f_num}[v] > \texttt{f_num}[w]$.
Nach Beobachtung 3 ist w weder Nachfahr noch Vorfahr von v im Tiefensuchwald.
(Bei der Auffassung der dfs-Aufrufe als Präorder- oder Postorderdurchlauf im Tiefensuchwald ist die Bearbeitung des Unterbaums unter w komplett abgeschlossen, bevor v erreicht wird.)

3. Fall: Die Untersuchung der Kante (v, w) in dfs(v) ergibt,
dass w **fertig** ist und $\texttt{dfs_num}[v] < \texttt{dfs_num}[w]$ gilt.

Weil dfs(v) noch aktiv ist, wird $\texttt{f_num}[v] > \texttt{f_num}[w]$.
Dann ist w nach Beobachtung 3 Nachfahr von v im Tiefensuchwald,
also ist (v, w) **Vorwärtskante** und wird richtig als $\boldsymbol{F}$ klassifiziert.

4. Fall: Die Untersuchung der Kante (v, w) in dfs(v) ergibt,
dass w **fertig** ist und $\texttt{dfs_num}[v] > \texttt{dfs_num}[w]$ gilt.

Weil dfs(v) noch aktiv ist, wird $\texttt{f_num}[v] > \texttt{f_num}[w]$.
Nach Beobachtung 3 ist w weder Nachfahr noch Vorfahr von v im Tiefensuchwald.
(Bei der Auffassung der dfs-Aufrufe als Präorder- oder Postorderdurchlauf im Tiefensuchwald ist die Bearbeitung des Unterbaums unter w komplett abgeschlossen, bevor v erreicht wird.)
Daher ist (v, w) **Querkante** und wird richtig als $\boldsymbol{C}$ klassifiziert.

PAUSE

Es folgt: Bemerkungen zu DFS in ungerichteten Graphen

8.3 Tiefensuche in ungerichteten Graphen

8.3 Tiefensuche in ungerichteten Graphen

Einfacher als Tiefensuche in gerichteten Graphen.

8.3 Tiefensuche in ungerichteten Graphen

Einfacher als Tiefensuche in gerichteten Graphen.

Zweck:

- Zusammenhangskomponenten finden

8.3 Tiefensuche in ungerichteten Graphen

Einfacher als Tiefensuche in gerichteten Graphen.

Zweck:

- Zusammenhangskomponenten finden
- Spannbaum für jede Zusammenhangskomponente finden

8.3 Tiefensuche in ungerichteten Graphen

Einfacher als Tiefensuche in gerichteten Graphen.

Zweck:

- Zusammenhangskomponenten finden
- Spannbaum für jede Zusammenhangskomponente finden
- Kreisfreiheitstest

8.3 Tiefensuche in ungerichteten Graphen

Einfacher als Tiefensuche in gerichteten Graphen.

Zweck:

- Zusammenhangskomponenten finden
- Spannbaum für jede Zusammenhangskomponente finden
- Kreisfreiheitstest

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

8.3 Tiefensuche in ungerichteten Graphen

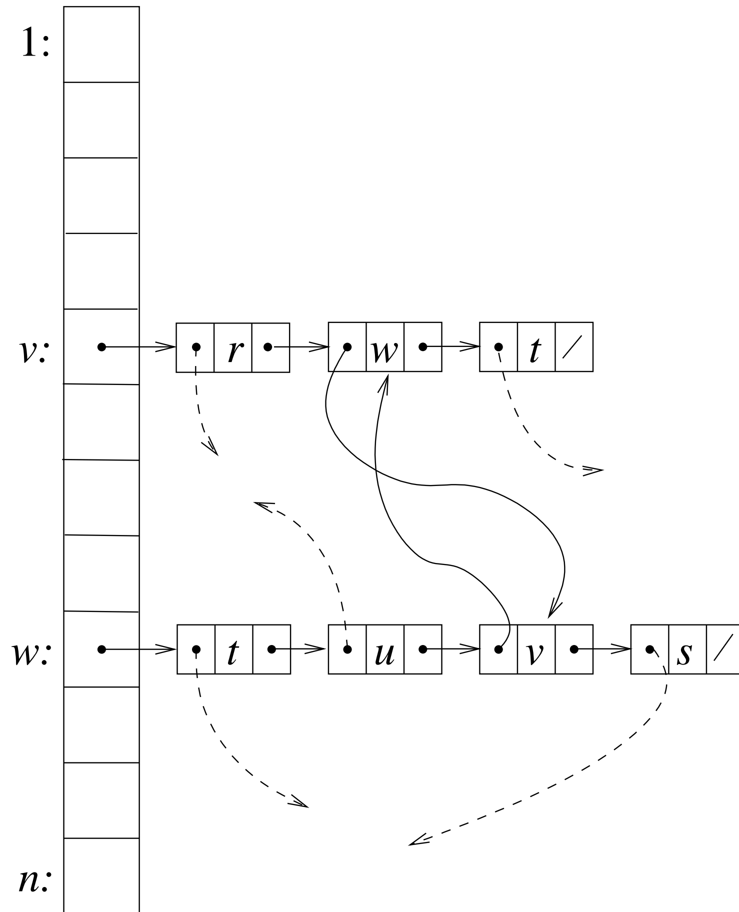
Einfacher als Tiefensuche in gerichteten Graphen.

Zweck:

- Zusammenhangskomponenten finden
- Spannbaum für jede Zusammenhangskomponente finden
- Kreisfreiheitstest

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

Komplexere Erweiterungen, z. B. „Zweifach-Zusammenhangskomponenten“, **erfordern** DFS. (Literatur z. B.: [Sedgewick, Algorithms, Part 5: Graph Algorithms].)



Adjazenzlistendarstellung mit Querverweisen.

Erinnerung: **Adjazenzlistendarstellung** mit Querverweisen.

Knotenarray: $\texttt{nodes[1..}n\texttt{]}$.

Zu Knoten v gibt es eine lineare Liste L_v mit einem Eintrag für jede Kante (v, w).

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

Erinnerung: **Adjazenzlistendarstellung** mit Querverweisen.

Knotenarray: `nodes[1..`n`]`.

Zu Knoten v gibt es eine lineare Liste L_v mit einem Eintrag für jede Kante (v, w).

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

Wir können an Kanten Markierungen anbringen.

DFS(G): Anfangs markieren wir beide Richtungen aller Kanten mit „neu“.

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

Dank der Gegenkanten ist dies in Zeit $O(1)$ möglich.

Erinnerung: **Adjazenzlistendarstellung** mit Querverweisen.

Knotenarray: $\texttt{nodes[1..}n\texttt{]}$.

Zu Knoten v gibt es eine lineare Liste L_v mit einem Eintrag für jede Kante (v, w).

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

Wir können an Kanten Markierungen anbringen.

DFS(G): Anfangs markieren wir beide Richtungen aller Kanten mit „neu“.

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

Dank der Gegenkanten ist dies in Zeit $O(1)$ möglich.

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 `status`$[v]$ = **neu** aufrufen

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`;

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

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

```
(1) dfs_count++;
(2) dfs_num[v] ← dfs_count;
(3) dfs-visit(v); // Aktion an v bei Erstbesuch
(4) status[v] ← aktiv ;
```

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** ;

(5) für jede **„neue“** Kante $(v, \texttt{w})$ (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 $(v, \mathtt{w})$ (Adjazenzliste!) tue:

(6) setze Gegenkante $(\mathtt{w}, v)$ auf **„gesehen“**;

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

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 $(v, \mathtt{w})$ (Adjazenzliste!) tue:

(6) setze Gegenkante $(\mathtt{w}, v)$ auf „gesehen“;

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

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

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 $(v, \mathtt{w})$ (Adjazenzliste!) tue:

(6) setze Gegenkante $(\mathtt{w}, v)$ auf **„gesehen“**;

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

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

(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 $(v, \mathtt{w})$ (Adjazenzliste!) tue:

(6) setze Gegenkante $(\mathtt{w}, v)$ auf **„gesehen“**;

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

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

(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 $(v, \mathtt{w})$ (Adjazenzliste!) tue:

(6) setze Gegenkante $(\mathtt{w}, v)$ auf **„gesehen“**;

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

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

(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 $(v, \mathtt{w})$ (Adjazenzliste!) tue:

(6) setze Gegenkante $(\mathtt{w}, v)$ auf „gesehen“;

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

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

(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:

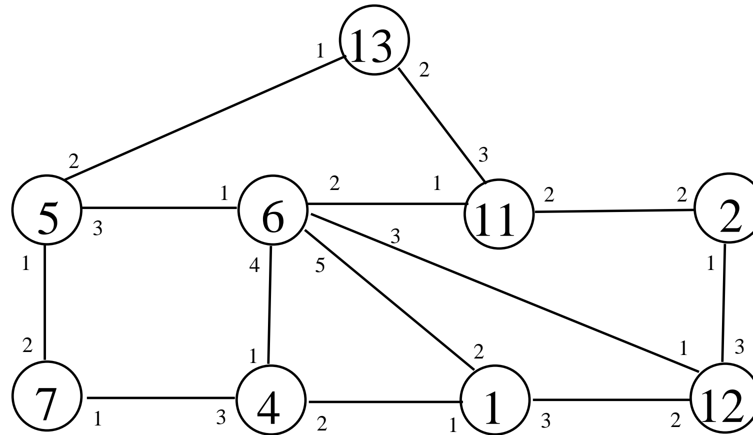
Globale Tiefensuche in (ungerichtetem) Graphen G:

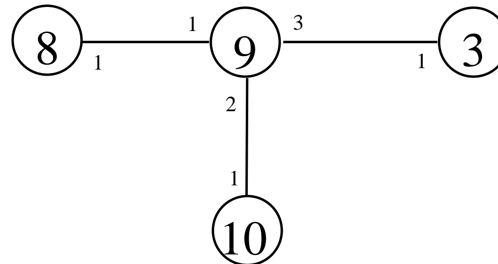
Algorithmus UDFS(G) // Tiefensuche in $G = (V, E)$

Globale Tiefensuche in (ungerichtetem) Graphen G:

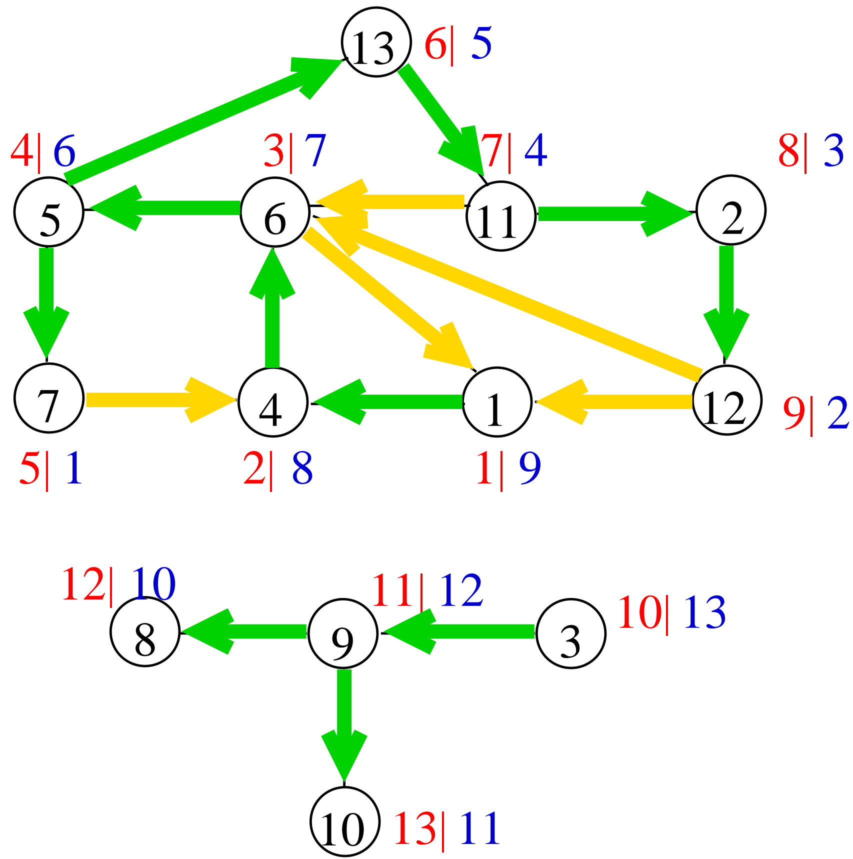
Algorithmus UDFS(G) // Tiefensuche in $G = (V, E)$

(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 dfs- und f-Nummern.

Satz 8.3.1 **udfs**(v_0) werde aufgerufen. Dann gilt:

Satz 8.3.1 **udfs**(v_0) werde aufgerufen. Dann gilt:

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

Satz 8.3.1 **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), für die **udfs**(w) unmittelbar aus **udfs**(v) aufgerufen wird, bilden einen gerichteten Baum mit Wurzel v_0 **(„Tiefensuchbaum“)**.

Satz 8.3.1 **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), für die **udfs**(w) unmittelbar aus **udfs**(v) aufgerufen wird, bilden einen gerichteten Baum mit Wurzel v_0 **(„Tiefensuchbaum“)**.

(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 8.3.1 **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), für die **udfs**(w) unmittelbar aus **udfs**(v) aufgerufen wird, bilden einen gerichteten Baum mit Wurzel v_0 **(„Tiefensuchbaum“)**.

(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.

Rechenzeit (mit Initialisierung): $O(|V| + |E_{v_0}|)$, wo E_{v_0} die Menge der Kanten in der Zusammenhangskomponente von v_0 ist.

Satz 8.3.1 **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), für die **udfs**(w) unmittelbar aus **udfs**(v) aufgerufen wird, bilden einen gerichteten Baum mit Wurzel v_0 **(„Tiefensuchbaum“)**.

(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.

Rechenzeit (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 8.3.2

UDFS(G) werde aufgerufen. Dann gilt:

Satz 8.3.2

UDFS(G) werde aufgerufen. Dann gilt:

(a) Die T-Kanten bilden eine Reihe von Bäumen (die Tiefensuch-Bäume).

Satz 8.3.2

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 8.3.2

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 Zusammenhangs-komponente in der Reihenfolge, die in Zeile (2) benutzt wird.

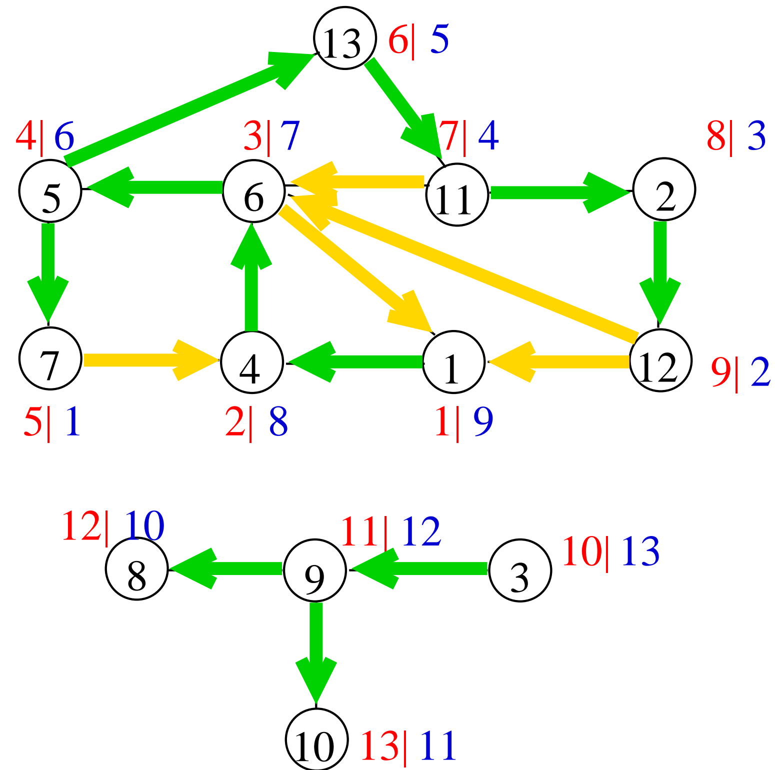
Satz 8.3.2

UDFS(G) werde aufgerufen. Dann gilt:

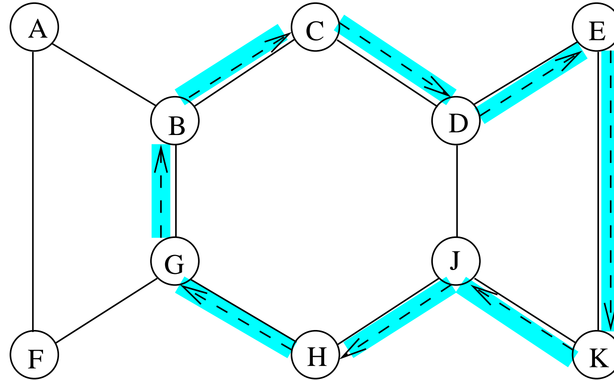
(a) Die T-Kanten bilden eine Reihe von Bäumen (die Tiefensuch-Bäume).
Die Knotenmengen dieser Bäume sind die Zusammenhangskomponenten von G.

(b) Die Wurzeln der Bäume sind die jeweils ersten Knoten einer Zusammenhangskomponente in der Reihenfolge, die in Zeile (2) benutzt wird.

Gesamtrechenzeit: $O(|V| + |E|)$: linear!



Beobachtung: Jeder Knoten wird besucht; jede Kante wird in genau einer Richtung betrachtet und als Baumkante ($\boldsymbol{T}$) oder Rückwärtskante ($\boldsymbol{B}$) klassifiziert; die jeweiligen Gegenkanten werden nicht betrachtet (weil sie auf „gesehen“ gesetzt werden).



Einfache Kreise: (B,C,D,E,K,J,H,G,B) und (K,J,D,E,K)

(Erinnerung an Definition 7.1.14)

Ein Kantenzug $(v_0, v_1, \ldots, v_k)$ mit $k \geq 3$ in einem (ungerichteten) Graphen G heißt ein **(einfacher) Kreis**, wenn $v_0 = v_k$ gilt und $v_0, v_1, \ldots, v_{k-1}$ verschieden sind.

Kreisfreiheitstest in Graphen und Finden eines Kreises, wenn es einen gibt.

Kreisfreiheitstest in Graphen und Finden eines Kreises, wenn es einen gibt.

Satz 8.3.3

Nach Ausführung von **UDFS**(G) gilt: $B \neq \emptyset \;\Leftrightarrow\; G$ enthält einen Kreis.

Kreisfreiheitstest in Graphen und Finden eines Kreises, wenn es einen gibt.

Satz 8.3.3

Nach Ausführung von **UDFS**(G) gilt: $B \neq \emptyset \;\Leftrightarrow\; G$ enthält einen Kreis.

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 ≥ 3.

Kreisfreiheitstest in Graphen und Finden eines Kreises, wenn es einen gibt.

Satz 8.3.3

Nach Ausführung von **UDFS**(G) gilt: $B \neq \emptyset \;\Leftrightarrow\; G$ enthält einen Kreis.

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 ≥ 3.
(Es kann nicht sein, dass (w, v) eine Baumkante ist, da sonst (v, w) den Status „gesehen“ hätte.)

Kreisfreiheitstest in Graphen und Finden eines Kreises, wenn es einen gibt.

Satz 8.3.3

Nach Ausführung von **UDFS**(G) gilt: $B \neq \emptyset \;\Leftrightarrow\; G$ enthält einen Kreis.

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 ≥ 3.
(Es kann nicht sein, dass (w, v) eine Baumkante ist, da sonst (v, w) den Status „**gesehen**“ hätte.)

In diesem Moment kann man auch leicht diesen Kreis ausgeben.

Kreisfreiheitstest in Graphen und Finden eines Kreises, wenn es einen gibt.

Satz 8.3.3

Nach Ausführung von **UDFS**(G) gilt: $B \neq \emptyset \;\Leftrightarrow\; G$ enthält einen Kreis.

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 ≥ 3.
(Es kann nicht sein, dass (w, v) eine Baumkante ist, da sonst (v, w) den Status „gesehen“ hätte.)

In diesem Moment kann man auch leicht diesen Kreis ausgeben.

„$\Leftarrow$“: Wenn $B = \emptyset$ ist, dann wird jede Kante von G in einer Richtung Baumkante.

Kreisfreiheitstest in Graphen und Finden eines Kreises, wenn es einen gibt.

Satz 8.3.3

Nach Ausführung von **UDFS**(G) gilt: $B \neq \emptyset \;\Leftrightarrow\; G$ enthält einen Kreis.

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 ≥ 3.
(Es kann nicht sein, dass (w, v) eine Baumkante ist, da sonst (v, w) den Status „**gesehen**“ hätte.)

In diesem Moment kann man auch leicht diesen Kreis ausgeben.

„$\Leftarrow$“: 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“.

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“.

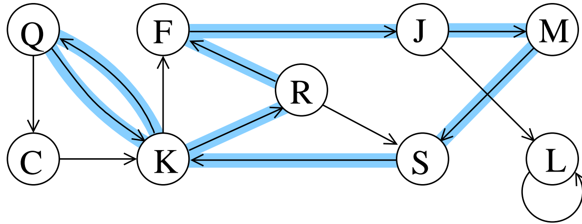
(Dies ist natürlich auch mit Breitensuche zu realisieren.)

PAUSE

Es folgt: Kreisfreie Digraphen, Topologische Sortierung

8.4 Kreisfreiheitstest und topologische Sortierung

8.4 Kreisfreiheitstest und topologische Sortierung



(Erinnerung an Definition 7.1.4)

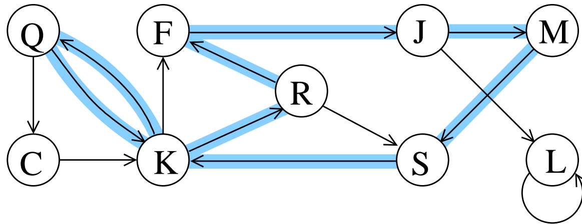
(a) Ein Kantenzug $(v_0, v_1, \ldots, v_k)$ in einem Digraphen G heißt ein **Kreis** oder **Zyklus**, wenn $k \geq 1$ und $v_0 = v_k$.

Beispiel: $(\mathrm{Q}, \mathrm{K}, \mathrm{R}, \mathrm{F}, \mathrm{J}, \mathrm{M}, \mathrm{S}, \mathrm{K}, \mathrm{Q})$ ist ein Kreis.

Kreise, die durch einen zyklischen Shift auseinander hervorgehen, werden als gleich betrachtet.

8.4 Kreisfreiheitstest und topologische Sortierung

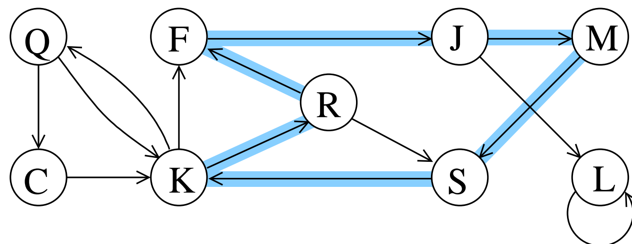
(Erinnerung an Definition 7.1.4)



(a) Ein Kantenzug $(v_0, v_1, \ldots, v_k)$ in einem Digraphen G heißt ein **Kreis** oder **Zyklus**, wenn $k \geq 1$ und $v_0 = v_k$.

Beispiel: $(\mathrm{Q}, \mathrm{K}, \mathrm{R}, \mathrm{F}, \mathrm{J}, \mathrm{M}, \mathrm{S}, \mathrm{K}, \mathrm{Q})$ ist ein Kreis.

Kreise, die durch einen zyklischen Shift auseinander hervorgehen, werden als gleich betrachtet.

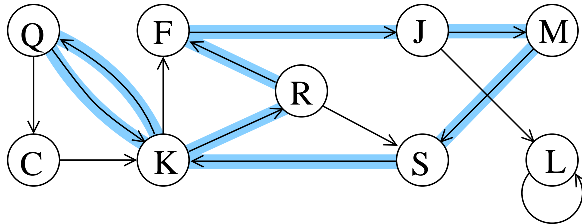


(b) Ein Kreis $(v_0, v_1, \ldots, v_{k-1}, v_0)$ heißt **einfach**, wenn $v_0, \ldots, v_{k-1}$ verschieden sind.

Beispiel: $(\mathrm{J}, \mathrm{M}, \mathrm{S}, \mathrm{K}, \mathrm{R}, \mathrm{F}, \mathrm{J})$ ist ein einfacher Kreis.

8.4 Kreisfreiheitstest und topologische Sortierung

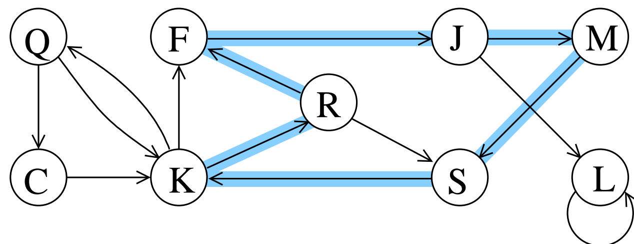
(Erinnerung an Definition 7.1.4)



(a) Ein Kantenzug $(v_0, v_1, \ldots, v_k)$ in einem Digraphen G heißt ein **Kreis** oder **Zyklus**, wenn $k \geq 1$ und $v_0 = v_k$.

Beispiel: $(\mathrm{Q}, \mathrm{K}, \mathrm{R}, \mathrm{F}, \mathrm{J}, \mathrm{M}, \mathrm{S}, \mathrm{K}, \mathrm{Q})$ ist ein Kreis.

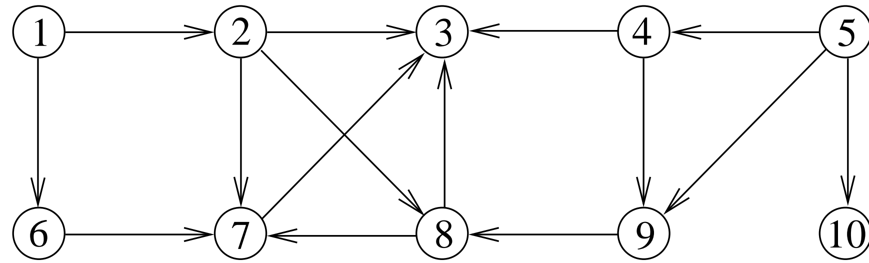
Kreise, die durch einen zyklischen Shift auseinander hervorgehen, werden als gleich betrachtet.



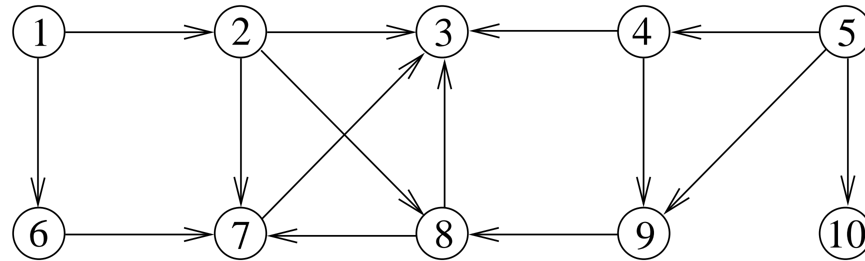
(b) Ein Kreis $(v_0, v_1, \ldots, v_{k-1}, v_0)$ heißt **einfach**, wenn $v_0, \ldots, v_{k-1}$ verschieden sind.

Beispiel: $(\mathrm{J}, \mathrm{M}, \mathrm{S}, \mathrm{K}, \mathrm{R}, \mathrm{F}, \mathrm{J})$ ist ein einfacher Kreis.

Wir wissen: Wenn G einen Kreis enthält, dann auch einen einfachen Kreis.

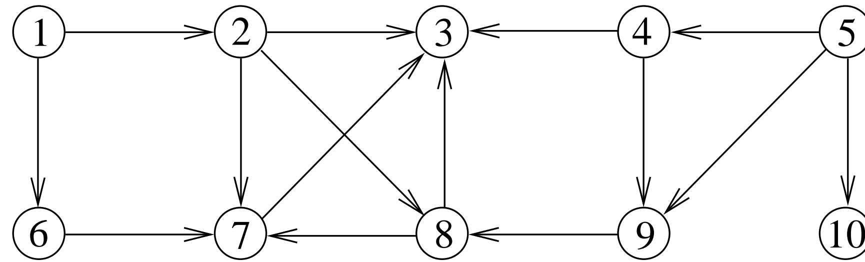


1
2
3
4
5
6
7
8
9
10



(Erinnerung an Definition 7.1.5)

Ein Digraph G heißt **kreisfrei** oder **azyklisch**, wenn es in G keinen Kreis gibt, sonst heißt G **zyklisch**.

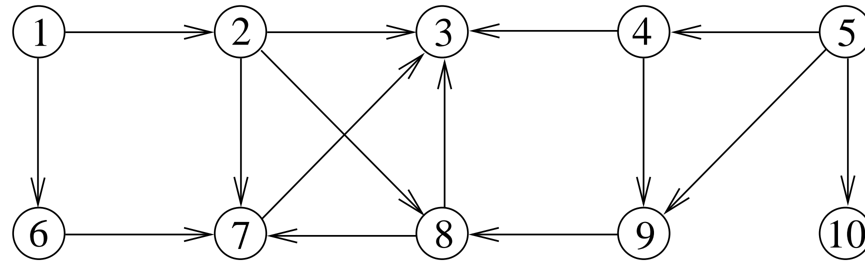


(Erinnerung an Definition 7.1.5)

Ein Digraph G heißt **kreisfrei** oder **azyklisch**, wenn es in G keinen Kreis gibt, sonst heißt G **zyklisch**.

Azyklische gerichtete Graphen: „**Directed Acyclic Graphs**“, daher **DAG**s.

Ob ein Digraph zyklisch ist oder nicht, kann man nach Durchführung einer Tiefensuche sofort ablesen.



(Erinnerung an Definition 7.1.5)

Ein Digraph G heißt **kreisfrei** oder **azyklisch**, wenn es in G keinen Kreis gibt, sonst heißt G **zyklisch**.

Azyklische gerichtete Graphen: „**D**irected **A**cyclic **G**raphs“, daher **DAG**s.

Ob ein Digraph zyklisch ist oder nicht, kann man nach Durchführung einer Tiefensuche sofort ablesen.

Satz 8.4.1

Nach Aufruf von DFS(G) gilt: G enthält einen Kreis $\Leftrightarrow$ $B \neq \emptyset$.

Beweis: „$\Leftarrow$“:

Beweis: „$\Leftarrow$“: Wenn DFS(G) eine B-Kante (v, w) findet, dann ist in diesem Moment v der vorderste Knoten des **roten Weges**, und w liegt irgendwo auf dem roten Weg. Also gibt es in G einen Weg von w nach v aus Baumkanten. Zusammen mit der Kante (v, w) ergibt sich ein Kreis.

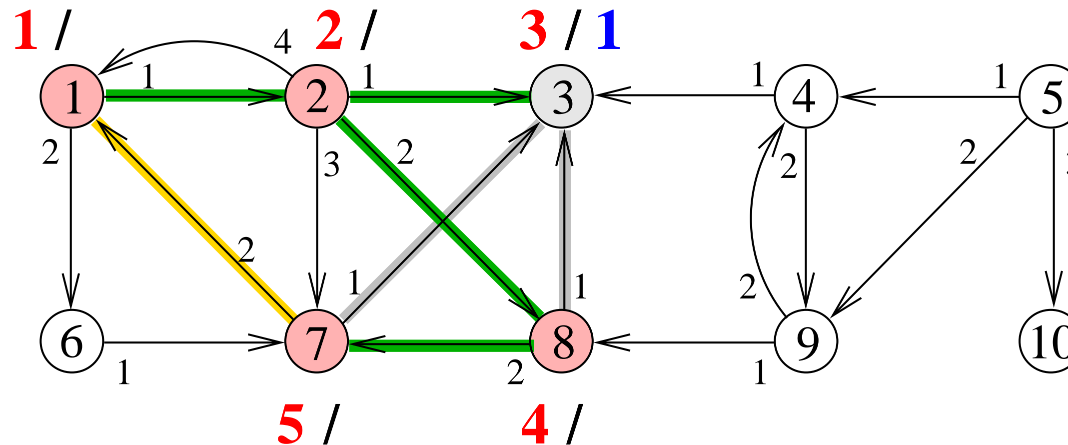
Beweis: „$\Leftarrow$“: Wenn DFS(G) eine B-Kante (v, w) findet, dann ist in diesem Moment v der vorderste Knoten des **roten Weges**, und w liegt irgendwo auf dem roten Weg. Also gibt es in G einen Weg von w nach v aus Baumkanten. Zusammen mit der Kante (v, w) ergibt sich ein Kreis.

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

Beweis: „$\Leftarrow$“: Wenn DFS(G) eine B-Kante (v, w) findet, dann ist in diesem Moment v der vorderste Knoten des **roten Weges**, und w liegt irgendwo auf dem roten Weg. Also gibt es in G einen Weg von w nach v aus Baumkanten. Zusammen mit der Kante (v, w) ergibt sich ein Kreis.

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

Beispiel:



Roter Weg: $(\mathbf{1, 2, 8, 7})$; Rückwärtskante $(\mathbf{7, 1})$ schließt den Kreis.

„$\Rightarrow$“:

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

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

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

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

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

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

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

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

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

„$\Rightarrow$“: 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./3. Fall: (v, w) ist Vorwärtskante oder Querkante.

„⇒“: 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./3. Fall: (v, w) ist Vorwärtskante oder Querkante. – Dann ist w „fertig“, während an v noch gearbeitet wird, also ist wieder $\texttt{f_num}[v] > \texttt{f_num}[w]$.

„$\Rightarrow$“: 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./3. Fall: (v, w) ist Vorwärtskante oder Querkante. – Dann ist w „fertig“, während an v noch gearbeitet wird, also ist wieder $\texttt{f_num}[v] > \texttt{f_num}[w]$.

Für jede Kante (v, w) in E ist $\texttt{f_num}[v]$ größer als $\texttt{f_num}[w]$.

„$\Rightarrow$“: 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./3. Fall: (v, w) ist Vorwärtskante oder Querkante. – Dann ist w „fertig“, während an v noch gearbeitet wird, also ist wieder f_num[v] $>$ f_num[w].

Für jede Kante (v, w) in E ist f_num[v] größer als f_num[w].

Wenn $(u_0, \ldots, u_t)$ ein Weg in G ist, so **nehmen** entlang dieses Weges **die f-Nummern strikt ab**.

„$\Rightarrow$“: 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./3. Fall: (v, w) ist Vorwärtskante oder Querkante. – Dann ist w „fertig“, während an v noch gearbeitet wird, also ist wieder $\texttt{f_num}[v] > \texttt{f_num}[w]$.

Für jede Kante (v, w) in E ist $\texttt{f_num}[v]$ größer als $\texttt{f_num}[w]$.

Wenn $(u_0, \ldots, u_t)$ ein Weg in G ist, so **nehmen** entlang dieses Weges **die f-Nummern strikt ab**.

Daher kann auf keinen Fall $t \geq 1$ und $u_0 = u_t$ sein.

„$\Rightarrow$“: 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./3. Fall: (v, w) ist Vorwärtskante oder Querkante. – Dann ist w „fertig“, während an v noch gearbeitet wird, also ist wieder `f_num`[v] $>$ `f_num`[w].

Für jede Kante (v, w) in E ist `f_num`[v] größer als `f_num`[w].

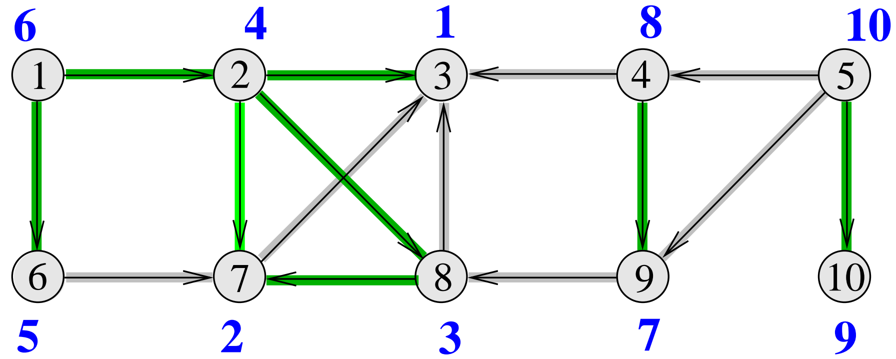
Wenn $(u_0, \ldots, u_t)$ ein Weg in G ist, so **nehmen** entlang dieses Weges **die f-Nummern strikt ab**.

Daher kann auf keinen Fall $t \geq 1$ und $u_0 = u_t$ sein.

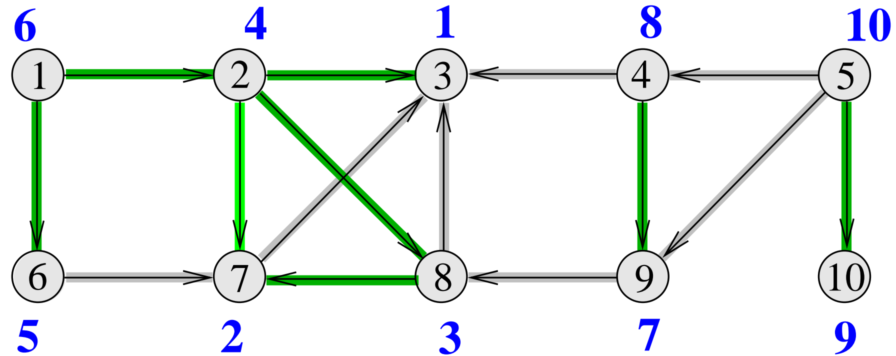
Also enthält G keinen Kreis. □

Beispiel: (Azyklischer) Digraph G mit f-Nummern:

Beispiel: (Azyklischer) Digraph G mit f-Nummern:

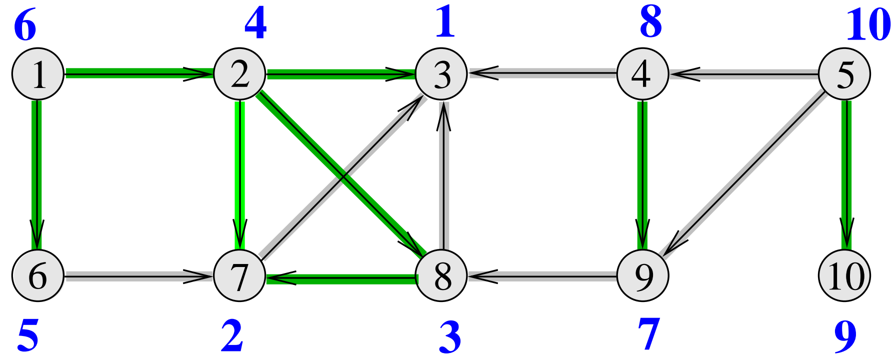


Beispiel: (Azyklischer) Digraph G mit f-Nummern:



Wie im Beweis von Satz 8.4.1: f-Nummern fallen entlang jeder Kante!

Beispiel: (Azyklischer) Digraph G mit f-Nummern:



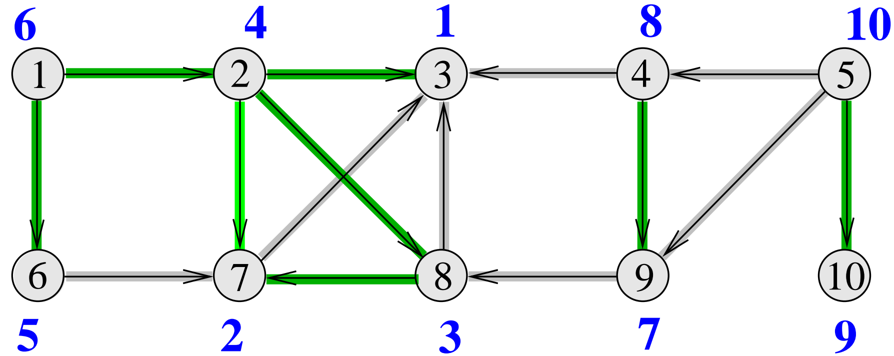
Wie im Beweis von Satz 8.4.1: f-Nummern fallen entlang jeder Kante!

Definition 8.4.2

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 für alle Kanten $(v, w) \in E$ die Beziehung $\pi(v) < \pi(w)$ gilt.

Beispiel: (Azyklischer) Digraph G mit f-Nummern:



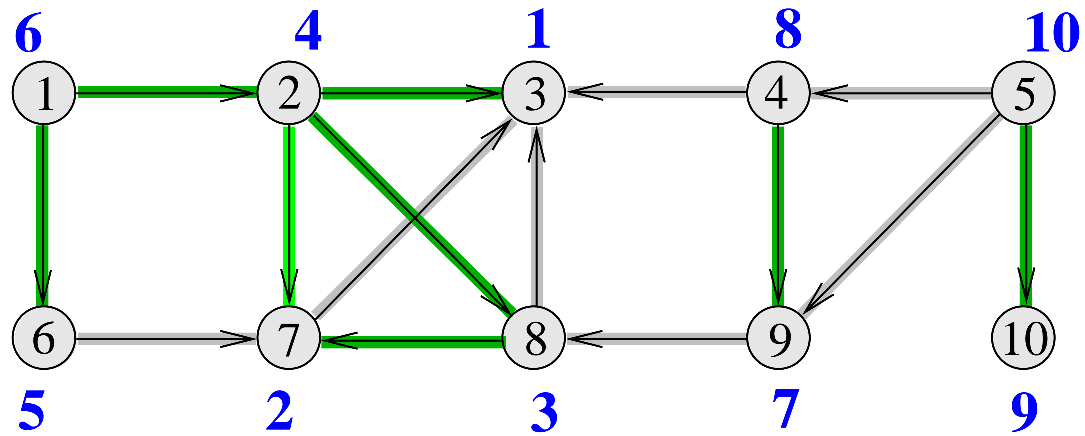
Wie im Beweis von Satz 8.4.1: f-Nummern fallen entlang jeder Kante!

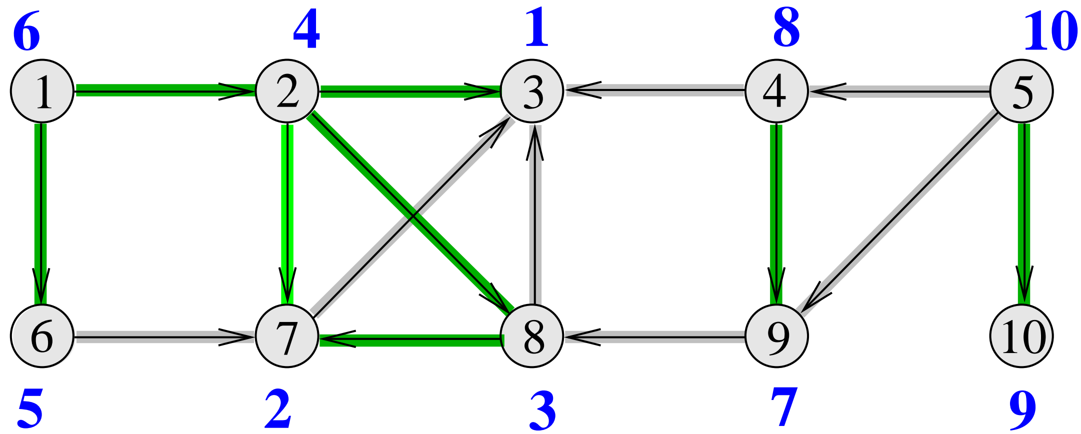
Definition 8.4.2

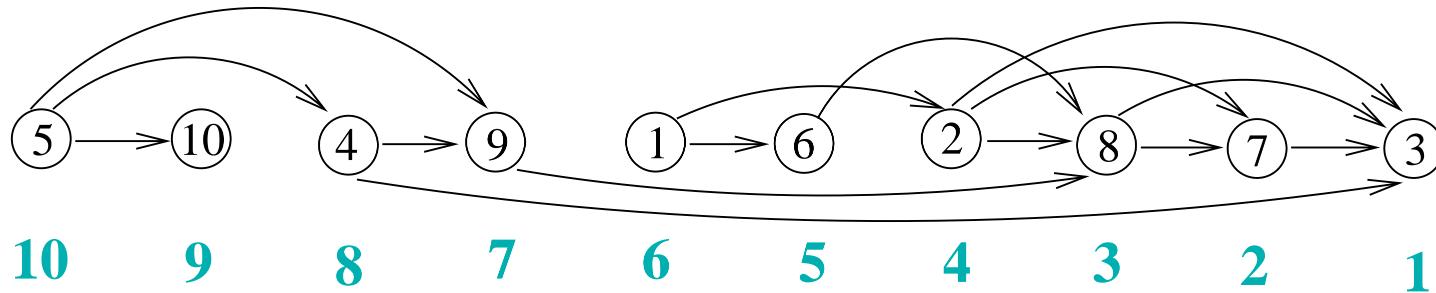
Sei $G = (V, E)$ ein azyklischer Digraph.

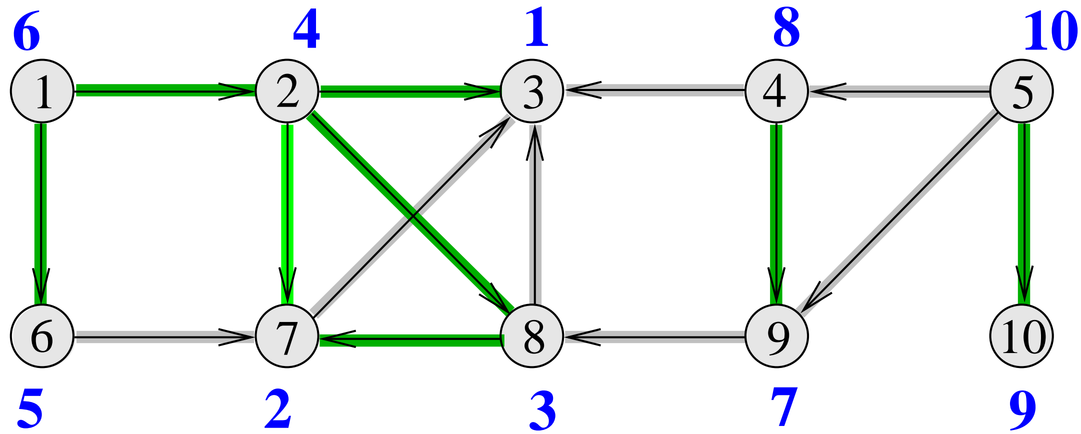
Eine **topologische Sortierung** (oder topologische Anordnung) der Knoten in V ist eine Bijektion $\pi \colon V \to \{1, \ldots, n\}$, so dass für alle Kanten $(v, w) \in E$ die Beziehung $\pi(v) < \pi(w)$ gilt.

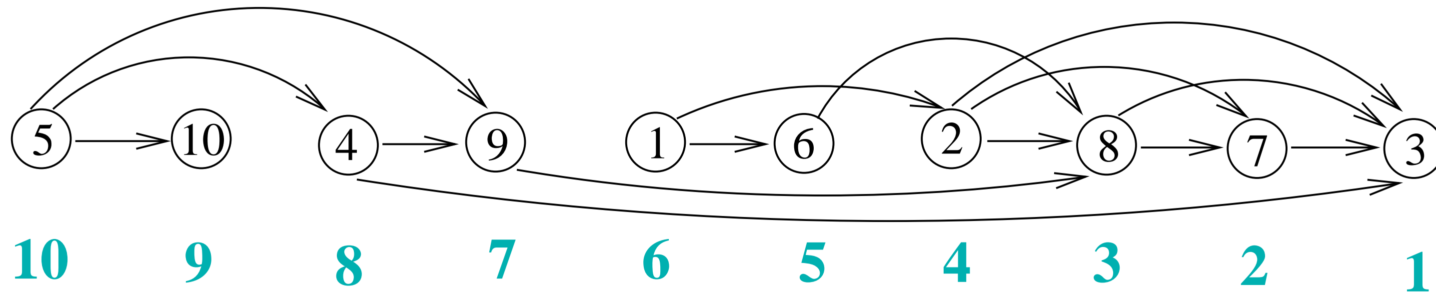
Anschauung: Man setzt Knoten v an Position $\pi(v)$; dann laufen alle Kanten „von links nach rechts“.


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

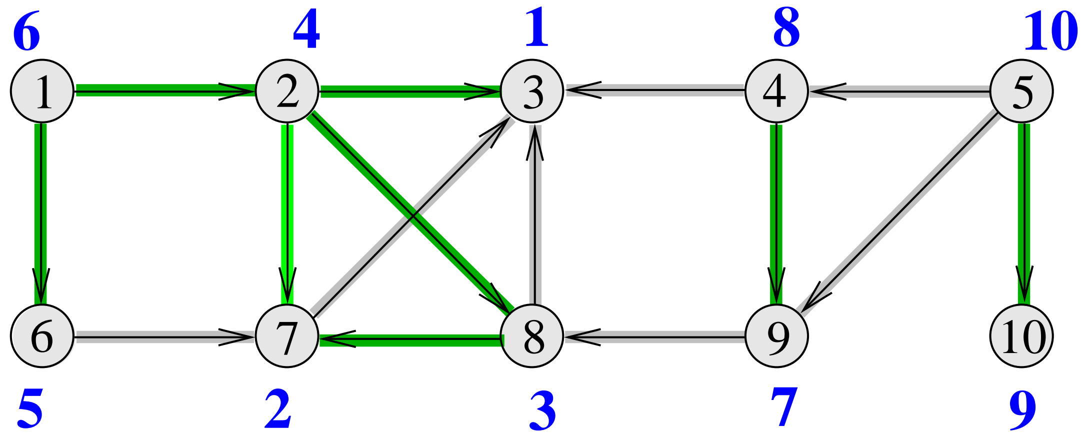

6
4
1
8
10
1
2
3
4
5
6
7
8
9
10
5
2
3
7
9

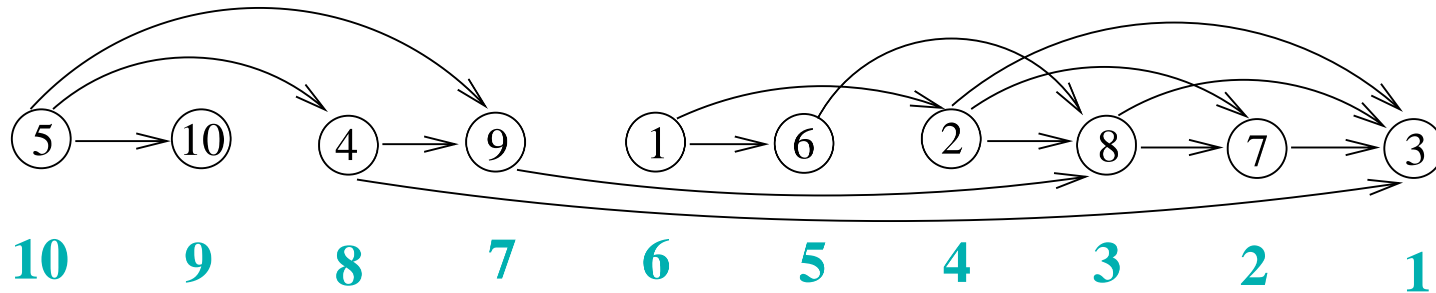

5
10
4
9
1
6
2
8
7
3
10
9
8
7
6
5
4
3
2
1





Derselbe Graph, linear angeordnet.





Derselbe Graph, linear angeordnet. – Permutation π dazu:

v	1	2	3	4	5	6	7	8	9	10
$\pi(v)$	5	7	10	3	1	6	9	8	4	2

DFS(G) findet sofort eine topologische Sortierung!

DFS(G) findet sofort eine topologische Sortierung!

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

DFS(G) findet sofort eine topologische Sortierung!

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

Idee: Wir haben im Beweis von 8.4.1 gesehen, dass die f-Nummern entlang jeder Kante (v, w) **strikt fallen**; außerdem sind die f-Nummern eine **Bijektion**.

DFS(G) findet sofort eine topologische Sortierung!
(Natürlich nur in **azyklischen** Digraphen.)

Idee: Wir haben im Beweis von 8.4.1 gesehen, dass die f-Nummern entlang jeder Kante (v, w) **strikt fallen**; außerdem sind die f-Nummern eine **Bijektion**.

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

DFS(G) findet sofort eine topologische Sortierung!
(Natürlich nur in **azyklischen** Digraphen.)

Idee: Wir haben im Beweis von 8.4.1 gesehen, dass die f-Nummern entlang jeder Kante (v, w) **strikt fallen**; außerdem sind die f-Nummern eine **Bijektion**.

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

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

DFS(G) findet sofort eine topologische Sortierung!

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

Idee: Wir haben im Beweis von 8.4.1 gesehen, dass die f-Nummern entlang jeder Kante (v, w) **strikt fallen**; außerdem sind die f-Nummern eine **Bijektion**.

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

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

Satz 8.4.3

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

Anwendung 1:

V: Menge von Tasks (für einen Computer, für eine Maschine).

$E \subseteq V \times V$: Zeitliche Abhängigkeiten.

$(v, w) \in E$ heißt: Task v muss beendet sein, bevor Task w beginnt.

Der resultierende Graph $G = (V, E)$ sollte azyklisch sein, sonst . . .

Anwendung 1:

V: Menge von Tasks (für einen Computer, für eine Maschine).

$E \subseteq V \times V$: Zeitliche Abhängigkeiten.

$(v, w) \in E$ heißt: Task v muss beendet sein, bevor Task w beginnt.

Der resultierende Graph $G = (V, E)$ sollte azyklisch sein, sonst . . .
ist die Aufgabe überhaupt nicht durchführbar!

Anwendung 1:

V: Menge von Tasks (für einen Computer, für eine Maschine).

$E \subseteq V \times V$: Zeitliche Abhängigkeiten.

$(v, w) \in E$ heißt: Task v muss beendet sein, bevor Task w beginnt.

Der resultierende Graph $G = (V, E)$ sollte azyklisch sein, sonst . . .
ist die Aufgabe überhaupt nicht durchführbar!

Eine topologische Sortierung ist dann eine mögliche Ausführungsreihenfolge für die Tasks auf der Maschine.

Anwendung 1:

V: Menge von Tasks (für einen Computer, für eine Maschine).

$E \subseteq V \times V$: Zeitliche Abhängigkeiten.

$(v, w) \in E$ heißt: Task v muss beendet sein, bevor Task w beginnt.

Der resultierende Graph $G = (V, E)$ sollte azyklisch sein, sonst . . .
ist die Aufgabe überhaupt nicht durchführbar!

Eine topologische Sortierung ist dann eine mögliche Ausführungsreihenfolge für die Tasks auf der Maschine.

Bei Parallelverarbeitung: „Scheduling“ ist eine viel komplexere Aufgabe!

Anwendung 2:

V: Menge von Tasks (für einen Computer, für eine Maschine),
mit **Verarbeitungszeiten** $\boldsymbol{c(v)}$, für $v \in V$.

$E \subseteq V \times V$: Zeitliche Abhängigkeiten wie vorher, azyklisch.

Anwendung 2:

V: Menge von Tasks (für einen Computer, für eine Maschine),
mit **Verarbeitungszeiten** $\boldsymbol{c(v)}$, für $v \in V$.

$E \subseteq V \times V$: Zeitliche Abhängigkeiten wie vorher, azyklisch.

Es stehen genügend Maschinen zur Verfügung, um beliebig viele Tasks parallel auszuführen.

Anwendung 2:

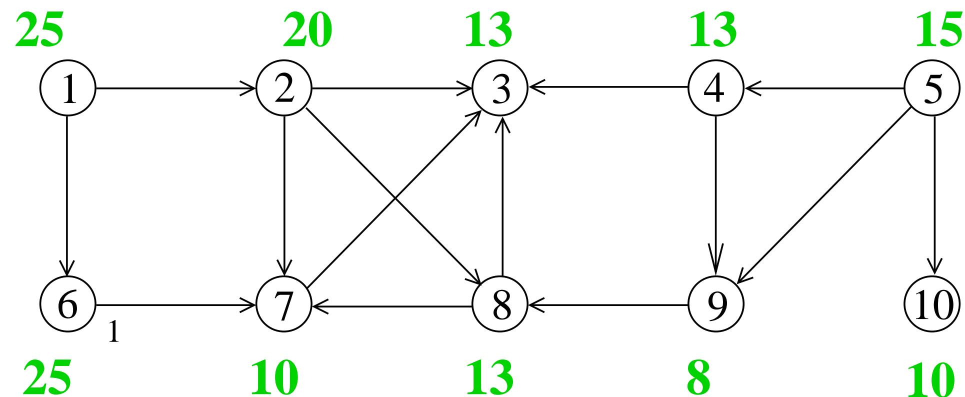
V: Menge von Tasks (für einen Computer, für eine Maschine), mit **Verarbeitungszeiten** $\boldsymbol{c(v)}$, für $v \in V$.

$E \subseteq V \times V$: Zeitliche Abhängigkeiten wie vorher, azyklisch.

Es stehen genügend Maschinen zur Verfügung, um beliebig viele Tasks parallel auszuführen.

Frage: Wie lange dauert die Ausführung aller Tasks (mindestens) bei optimaler Planung?

Weitergehend: Finde einen optimalen Plan.



Definition: Ein **„kritischer Pfad“** in $G = (V, E)$ ist eine Folge $p = (v_0, \ldots, v_r)$ von $r \geq 1$ verschiedenen Tasks mit $(v_0, v_1), \ldots, (v_{r-1}, v_r) \in E$ und maximalen „Kosten“ $c(p) := c(v_0) + \cdots + c(v_r)$.

Klar: Die Gesamt-Ausführungszeit ist mindestens $c(p)$. (Man kann sogar Gleichheit zeigen.)

Definition: Ein **„kritischer Pfad“** in $G = (V, E)$ ist eine Folge $p = (v_0, \ldots, v_r)$ von $r \geq 1$ verschiedenen Tasks mit $(v_0, v_1), \ldots, (v_{r-1}, v_r) \in E$ und maximalen „Kosten“ $c(p) := c(v_0) + \cdots + c(v_r)$.

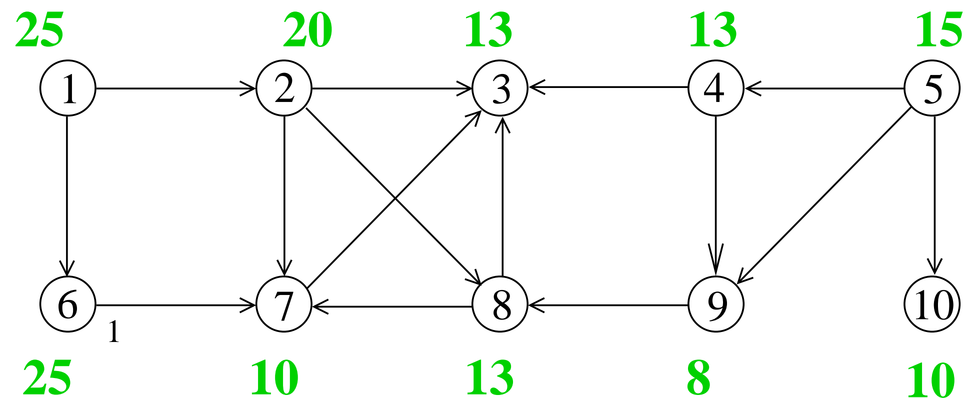
Klar: Die Gesamt-Ausführungszeit ist mindestens $c(p)$. (Man kann sogar Gleichheit zeigen.)

Aufgabe: Gegeben DAG $G = (V, E)$, bestimme kritischen Pfad, in Zeit $O(|V|+|E|)$.

Definition: Ein **„kritischer Pfad“** in $G = (V, E)$ ist eine Folge $p = (v_0, \ldots, v_r)$ von $r \geq 1$ verschiedenen Tasks mit $(v_0, v_1), \ldots, (v_{r-1}, v_r) \in E$ und maximalen „Kosten“ $c(p) := c(v_0) + \cdots + c(v_r)$.

Klar: Die Gesamt-Ausführungszeit ist mindestens $c(p)$. (Man kann sogar Gleichheit zeigen.)

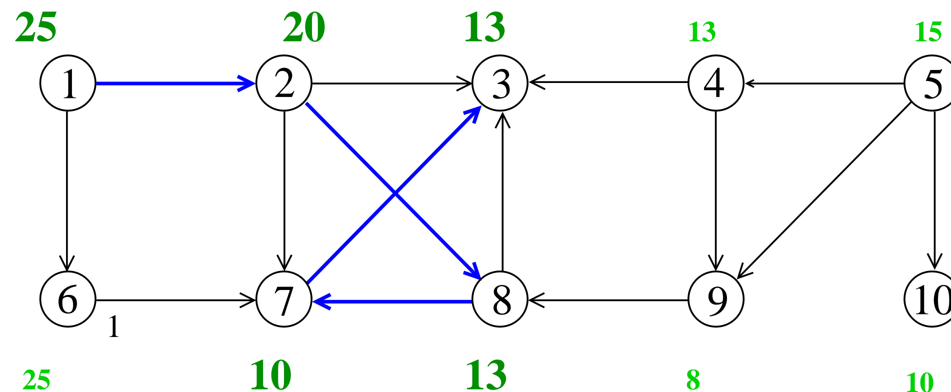
Aufgabe: Gegeben DAG $G = (V, E)$, bestimme kritischen Pfad, in Zeit $O(|V|+|E|)$.



Definition: Ein **„kritischer Pfad“** in $G = (V, E)$ ist eine Folge $p = (v_0, \ldots, v_r)$ von $r \geq 1$ verschiedenen Tasks mit $(v_0, v_1), \ldots, (v_{r-1}, v_r) \in E$ und maximalen „Kosten“ $c(p) := c(v_0) + \cdots + c(v_r)$.

Klar: Die Gesamt-Ausführungszeit ist mindestens $c(p)$. (Man kann sogar Gleichheit zeigen.)

Aufgabe: Gegeben DAG $G = (V, E)$, bestimme kritischen Pfad, in Zeit $O(|V|+|E|)$.

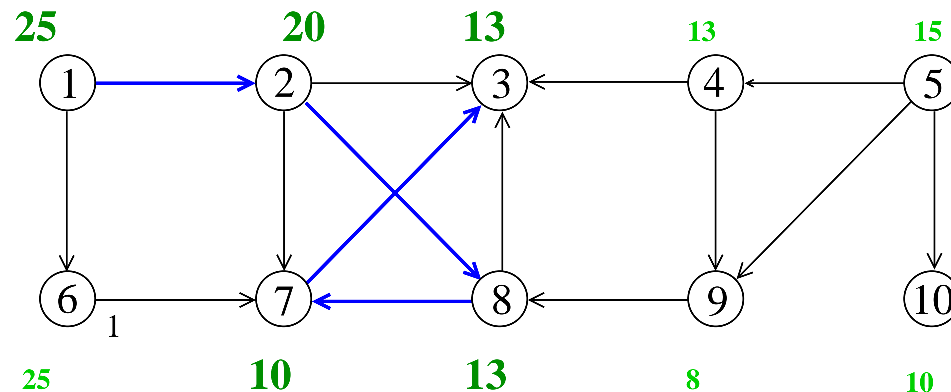


Ein kritischer Pfad mit Kosten $25 + 20 + 13 + 10 + 13 = 81$.

Definition: Ein **„kritischer Pfad“** in $G = (V, E)$ ist eine Folge $p = (v_0, \ldots, v_r)$ von $r \geq 1$ verschiedenen Tasks mit $(v_0, v_1), \ldots, (v_{r-1}, v_r) \in E$ und maximalen „Kosten“ $c(p) := c(v_0) + \cdots + c(v_r)$.

Klar: Die Gesamt-Ausführungszeit ist mindestens $c(p)$. (Man kann sogar Gleichheit zeigen.)

Aufgabe: Gegeben DAG $G = (V, E)$, bestimme kritischen Pfad, in Zeit $O(|V|+|E|)$.

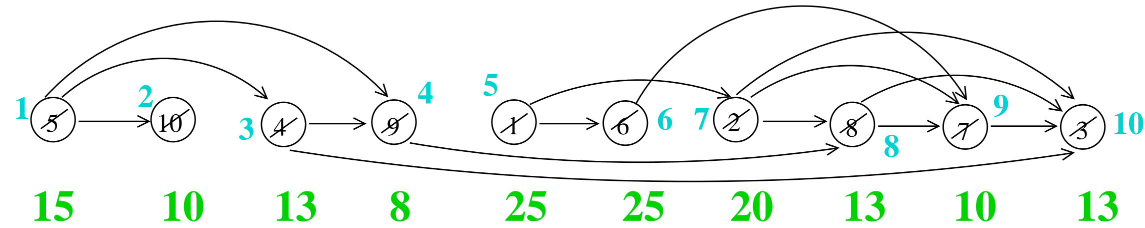


Ein kritischer Pfad mit Kosten $25 + 20 + 13 + 10 + 13 = 81$.

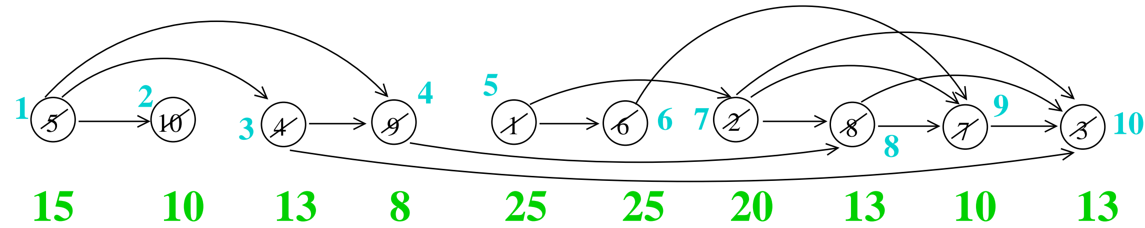
Aber wie berechnen?

Lösung: Ermittle zunächst eine topologische Sortierung von $G = (V, E)$.
Dann benenne Knoten um, so dass („o.B.d.A.“) gilt: $(v, w) \in E \Rightarrow v < w$.

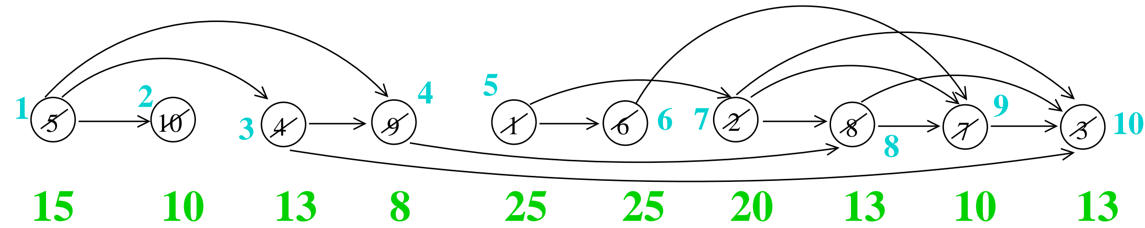
Lösung: Ermittle zunächst eine topologische Sortierung von $G = (V, E)$.
Dann benenne Knoten um, so dass („o.B.d.A.") gilt: $(v, w) \in E \Rightarrow v < w$.



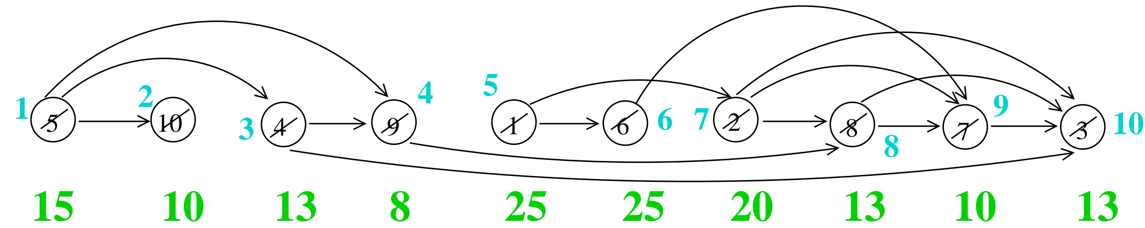
Lösung: Ermittle zunächst eine topologische Sortierung von $G = (V, E)$.
Dann benenne Knoten um, so dass („o.B.d.A.“) gilt: $(v, w) \in E \Rightarrow v < w$.



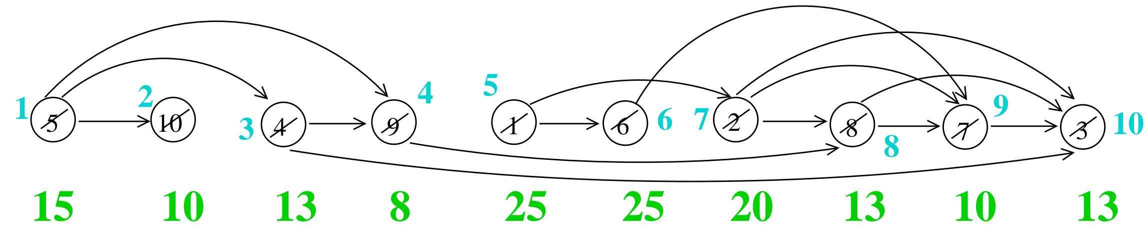
Lösung: Ermittle zunächst eine topologische Sortierung von $G = (V, E)$.
Dann benenne Knoten um, so dass („o.B.d.A.“) gilt: $(v, w) \in E \Rightarrow v < w$.



Lösung: Ermittle zunächst eine topologische Sortierung von $G = (V, E)$.
Dann benenne Knoten um, so dass („o.B.d.A.“) gilt: $(v, w) \in E \Rightarrow v < w$.

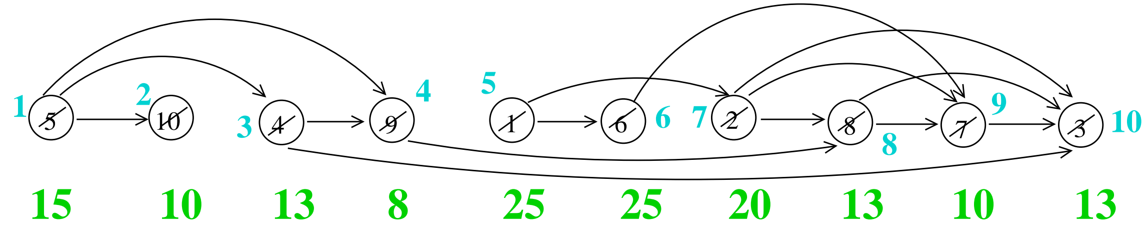


Lösung: Ermittle zunächst eine topologische Sortierung von $G = (V, E)$.
Dann benenne Knoten um, so dass („o.B.d.A.“) gilt: $(v, w) \in E \Rightarrow v < w$.



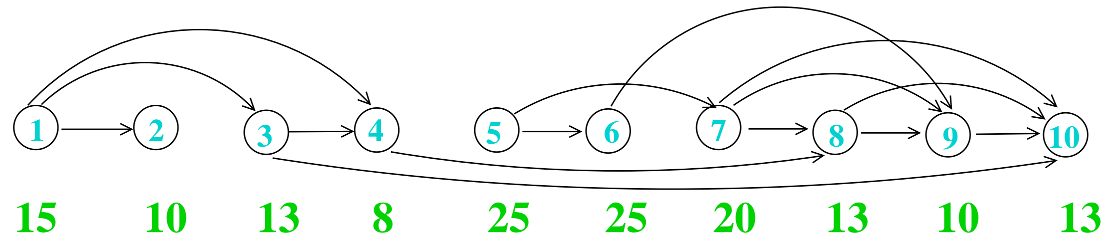
Lösung: Ermittle zunächst eine topologische Sortierung von $G = (V, E)$.

Dann benenne Knoten um, so dass („o.B.d.A.“) gilt: $(v, w) \in E \Rightarrow v < w$.

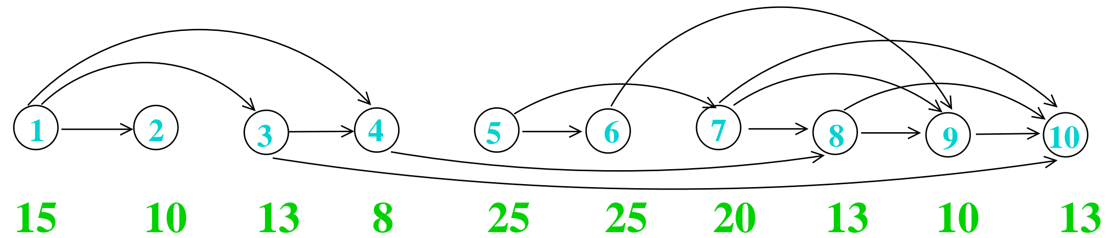


Lösung: Ermittle zunächst eine topologische Sortierung von $G = (V, E)$.
Dann benenne Knoten um, so dass („o.B.d.A.“) gilt: $(v, w) \in E \Rightarrow v < w$.

Lösung: Ermittle zunächst eine topologische Sortierung von $G = (V, E)$.
Dann benenne Knoten um, so dass („o.B.d.A.“) gilt: $(v, w) \in E \Rightarrow v < w$.



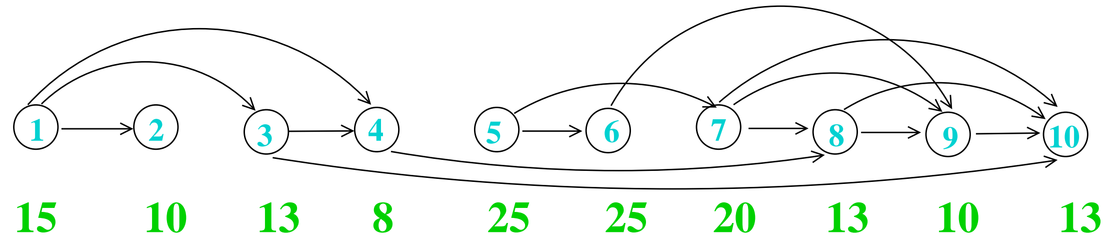
Lösung: Ermittle zunächst eine topologische Sortierung von $G = (V, E)$.
Dann benenne Knoten um, so dass („o.B.d.A.") gilt: $(v, w) \in E \Rightarrow v < w$.



Definiere, für $1 \le v \le n$:

$\bar{c}(v)$:= Kosten eines teuersten Pfades in $\{v, \ldots, n\}$, der mit v beginnt.

Lösung: Ermittle zunächst eine topologische Sortierung von $G = (V, E)$.
Dann benenne Knoten um, so dass („o.B.d.A.“) gilt: $(v, w) \in E \Rightarrow v < w$.

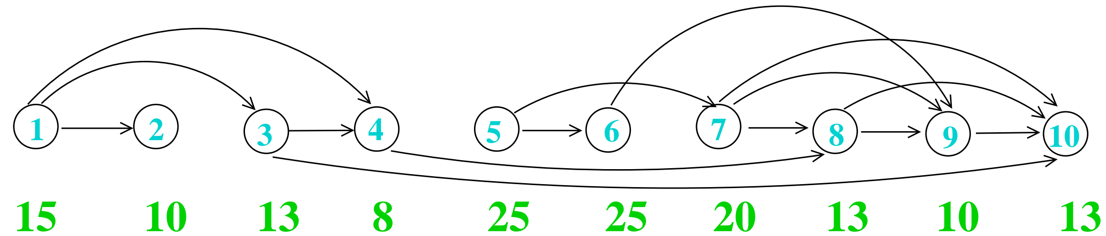


Definiere, für $1 \leq v \leq n$:

$\bar{c}(v)$:= Kosten eines teuersten Pfades in $\{v, \ldots, n\}$, der mit v beginnt.

$s(v)$:= erster Knoten nach v auf einem solchen Pfad bzw. undefiniert, wenn v in G keinen Nachfolger hat.

Lösung: Ermittle zunächst eine topologische Sortierung von $G = (V, E)$.
Dann benenne Knoten um, so dass („o.B.d.A.“) gilt: $(v, w) \in E \Rightarrow v < w$.



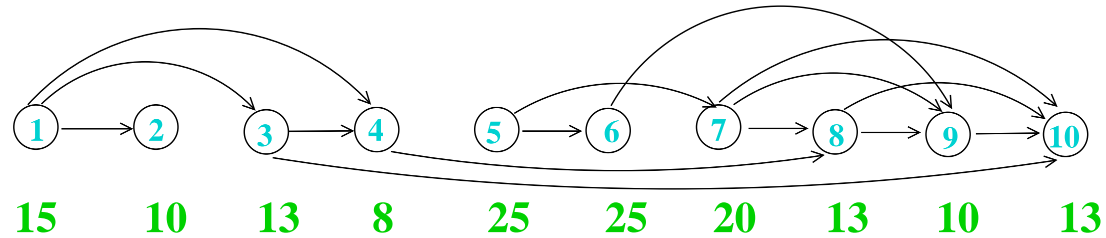
Definiere, für $1 \leq v \leq n$:

$\bar{c}(v)$:= Kosten eines teuersten Pfades in $\{v, \ldots, n\}$, der mit v beginnt.

$s(v)$:= erster Knoten nach v auf einem solchen Pfad bzw. undefiniert, wenn v in G keinen Nachfolger hat.

Berechne $\bar{c}(v)$ und $s(v)$ iterativ, für $v = n, \ldots, 1$, wie folgt.

Lösung: Ermittle zunächst eine topologische Sortierung von $G = (V, E)$.
Dann benenne Knoten um, so dass („o.B.d.A.“) gilt: $(v, w) \in E \Rightarrow v < w$.



Definiere, für $1 \leq v \leq n$:

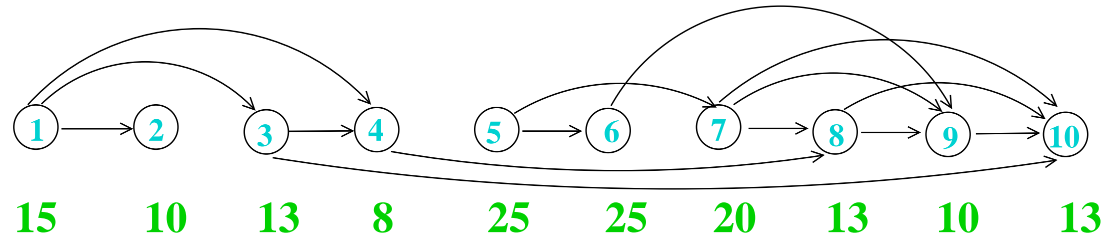
$\bar{c}(v)$:= Kosten eines teuersten Pfades in $\{v, \ldots, n\}$, der mit v beginnt.

$s(v)$:= erster Knoten nach v auf einem solchen Pfad bzw. undefiniert,
wenn v in G keinen Nachfolger hat.

Berechne $\bar{c}(v)$ und $s(v)$ iterativ, für $v = n, \ldots, 1$, wie folgt.

Weil n keinen Nachfolger hat, gilt $\bar{c}(n) = c(n)$, und $s(n)$ ist undefiniert.

Lösung: Ermittle zunächst eine topologische Sortierung von $G = (V, E)$.
Dann benenne Knoten um, so dass („o.B.d.A.“) gilt: $(v, w) \in E \Rightarrow v < w$.



Definiere, für $1 \leq v \leq n$:

$\bar{c}(v)$:= Kosten eines teuersten Pfades in $\{v, \ldots, n\}$, der mit v beginnt.

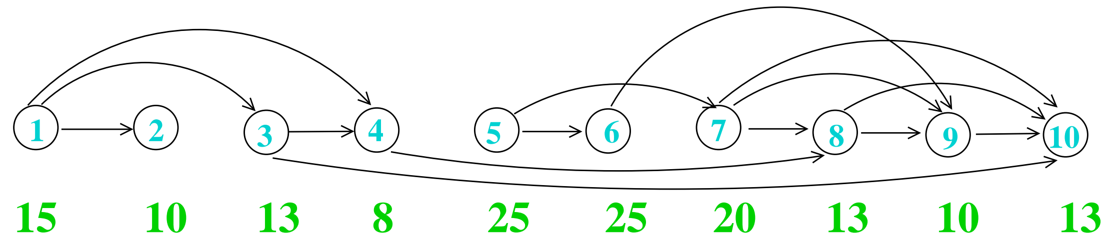
$s(v)$:= erster Knoten nach v auf einem solchen Pfad bzw. undefiniert, wenn v in G keinen Nachfolger hat.

Berechne $\bar{c}(v)$ und $s(v)$ iterativ, für $v = n, \ldots, 1$, wie folgt.

Weil n keinen Nachfolger hat, gilt $\bar{c}(n) = c(n)$, und $s(n)$ ist undefiniert.

Nun sei $v < n$ und $t(w)$ für $v < w \leq n$ schon definiert.

Lösung: Ermittle zunächst eine topologische Sortierung von $G = (V, E)$.
Dann benenne Knoten um, so dass („o.B.d.A.") gilt: $(v, w) \in E \Rightarrow v < w$.



Definiere, für $1 \leq v \leq n$:

$\bar{c}(v)$:= Kosten eines teuersten Pfades in $\{v, \ldots, n\}$, der mit v beginnt.

$s(v)$:= erster Knoten nach v auf einem solchen Pfad bzw. undefiniert, wenn v in G keinen Nachfolger hat.

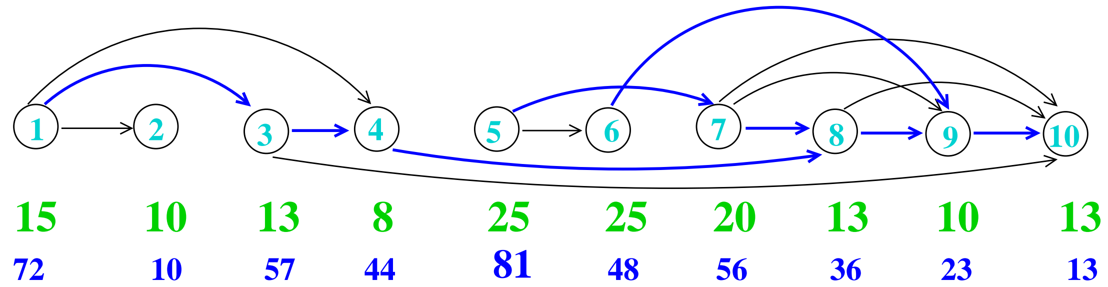
Berechne $\bar{c}(v)$ und $s(v)$ iterativ, für $v = n, \ldots, 1$, wie folgt.

Weil n keinen Nachfolger hat, gilt $\bar{c}(n) = c(n)$, und $s(n)$ ist undefiniert.

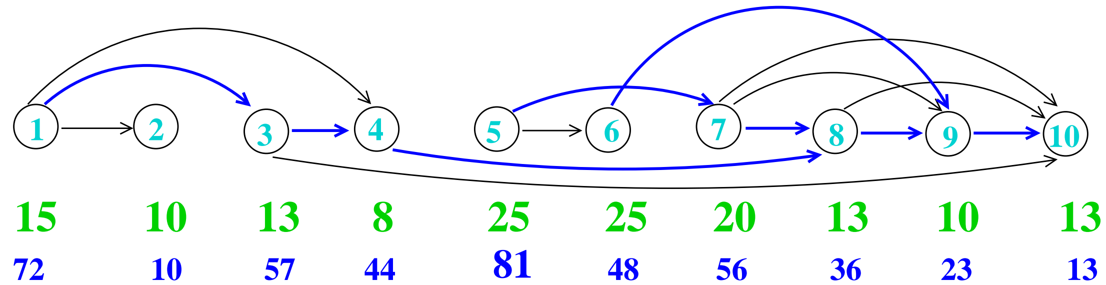
Nun sei $v < n$ und $t(w)$ für $v < w \leq n$ schon definiert. Dann gilt:

$$\bar{c}(v) = \begin{cases} c(v) + \max\{\bar{c}(w) \mid (v, w) \in E\} & \text{, falls eine Kante } (v, w) \text{ existiert } \textbf{(!)}\ v < w), \\ c(v) & \text{, sonst.} \end{cases}$$

Ergebnis der Berechnung:



Ergebnis der Berechnung:

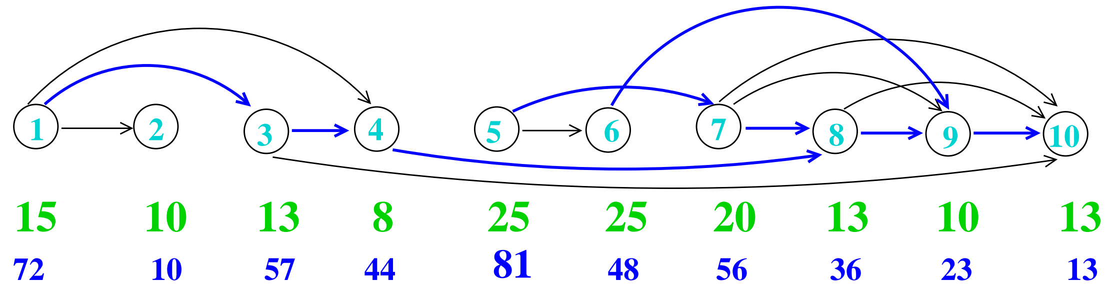


Ausgabe: Suche v^* mit maximalem $\bar{c}(v^*)$ (hier: $v^* = 5$ mit $\bar{c}(v^*) = 81$).

Kritischer Pfad: $(v^*, s(v^*), s(s(v^*)), \ldots)$ (bis zum ersten Knoten ohne Nachfolger).

Hier: $(5, 7, 8, 9, 10)$.

Ergebnis der Berechnung:



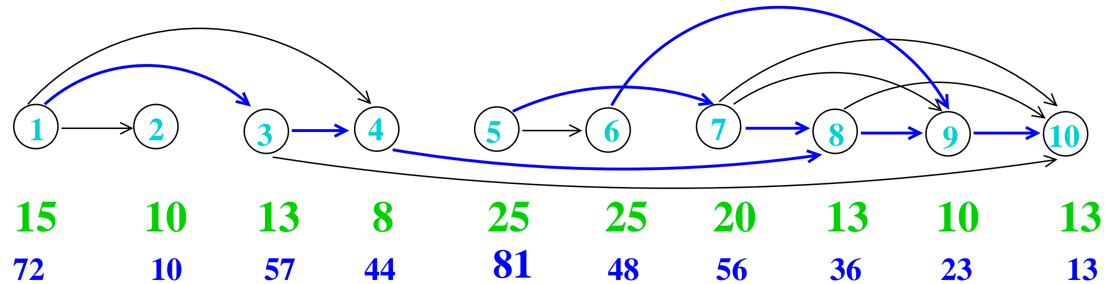
Ausgabe: Suche v^* mit maximalem $\bar{c}(v^*)$ (hier: $v^* = 5$ mit $\bar{c}(v^*) = 81$).

Kritischer Pfad: $(v^*, s(v^*), s(s(v^*)), \ldots)$ (bis zum ersten Knoten ohne Nachfolger).

Hier: $(5, 7, 8, 9, 10)$.

Alternative: Berechnung in der Reihenfolge $1, \ldots, n$. Nachteil: Benötigt Umkehrgraphen G^{R}.

Ergebnis der Berechnung:



Ausgabe: Suche v^* mit maximalem $\bar{c}(v^*)$ (hier: $v^* = 5$ mit $\bar{c}(v^*) = 81$).

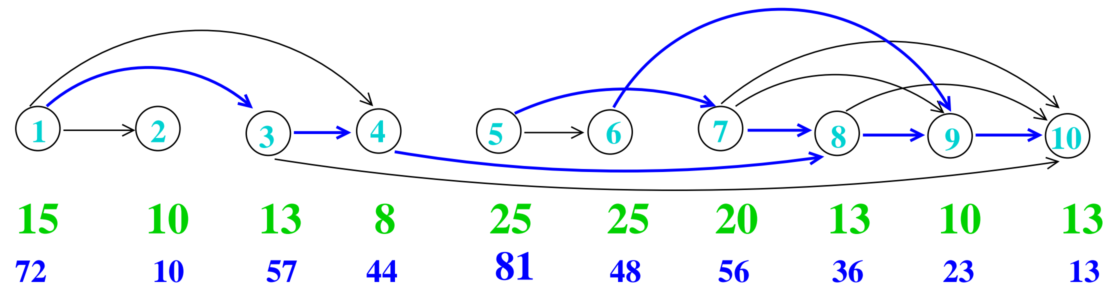
Kritischer Pfad: $(v^*, s(v^*), s(s(v^*)), \ldots)$ (bis zum ersten Knoten ohne Nachfolger).

Hier: $(5, 7, 8, 9, 10)$.

Alternative: Berechnung in der Reihenfolge $1, \ldots, n$. Nachteil: Benötigt Umkehrgraphen G^{R}.

Eine simple Erweiterung liefert sogar einen Ablaufplan, der optimale Zeit $\bar{c}(v^*)$ braucht:

Ergebnis der Berechnung:



Ausgabe: Suche v^* mit maximalem $\bar{c}(v^*)$ (hier: $v^* = 5$ mit $\bar{c}(v^*) = 81$).

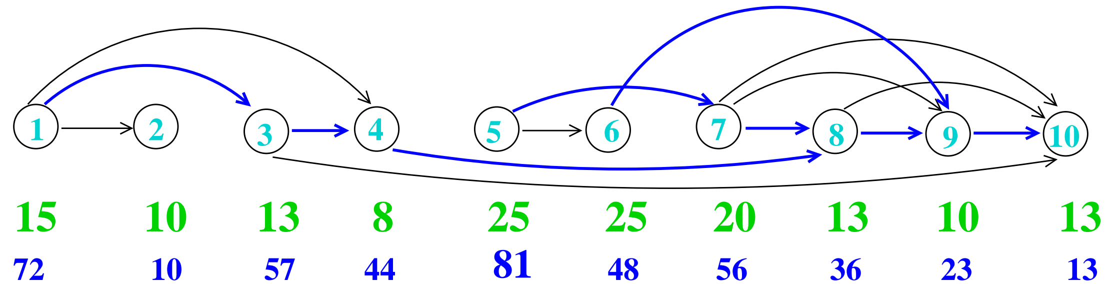
Kritischer Pfad: $(v^*, s(v^*), s(s(v^*)), \ldots)$ (bis zum ersten Knoten ohne Nachfolger).

Hier: $(5, 7, 8, 9, 10)$.

Alternative: Berechnung in der Reihenfolge $1, \ldots, n$. Nachteil: Benötigt Umkehrgraphen G^{R}.

Eine simple Erweiterung liefert sogar einen Ablaufplan, der optimale Zeit $\bar{c}(v^*)$ braucht:
Starte Task v zum Zeitpunkt $t(v) := \bar{c}(v^*) - \bar{c}(v)$.
(Das ist der „Aufschieberitis-Ablaufplan“: Alle Tasks werden so spät wie nur irgend möglich gestartet, ohne die bestmögliche Zeitschranke $\bar{c}(v^*)$ zu gefährden.)

Ergebnis der Berechnung:



Ausgabe: Suche v^* mit maximalem $\bar{c}(v^*)$ (hier: $v^* = 5$ mit $\bar{c}(v^*) = 81$).

Kritischer Pfad: $(v^*, s(v^*), s(s(v^*)), \ldots)$ (bis zum ersten Knoten ohne Nachfolger).

Hier: $(5, 7, 8, 9, 10)$.

Alternative: Berechnung in der Reihenfolge $1, \ldots, n$. Nachteil: Benötigt Umkehrgraphen G^{R}.

Eine simple Erweiterung liefert sogar einen Ablaufplan, der optimale Zeit $\bar{c}(v^*)$ braucht:
Starte Task v zum Zeitpunkt $t(v) := \bar{c}(v^*) - \bar{c}(v)$.
(Das ist der „Aufschieberitis-Ablaufplan“: Alle Tasks werden so spät wie nur irgend möglich gestartet, ohne die bestmögliche Zeitschranke $\bar{c}(v^*)$ zu gefährden.)
Wenn (v, w) eine Kante ist, gilt $t(w) - t(v) = \bar{c}(v) - \bar{c}(w) \geq c(v)$ nach Konstruktion, also reicht die Zeit zwischen $t(v)$ und $t(w)$ für die Ausführung von Task v.

PAUSE

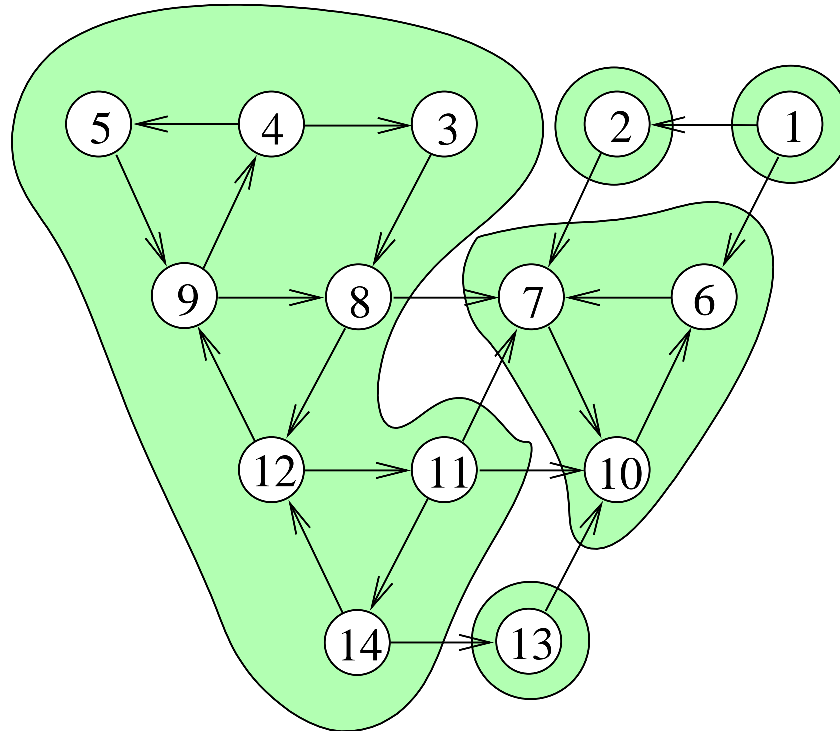
Es folgt: Starke Zusammenhangskomponenten

8.5 Starke Zusammenhangskomponenten in Digraphen

Beispiel:

8.5 Starke Zusammenhangskomponenten in Digraphen

Beispiel: 5 starke Zusammenhangskomponenten



8.5 Starke Zusammenhangskomponenten in Digraphen

Definition 8.5.1

8.5 Starke Zusammenhangskomponenten in Digraphen

Definition 8.5.1

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

8.5 Starke Zusammenhangskomponenten in Digraphen

Definition 8.5.1

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**.

8.5 Starke Zusammenhangskomponenten in Digraphen

Definition 8.5.1

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.

8.5 Starke Zusammenhangskomponenten in Digraphen

Definition 8.5.1

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**.

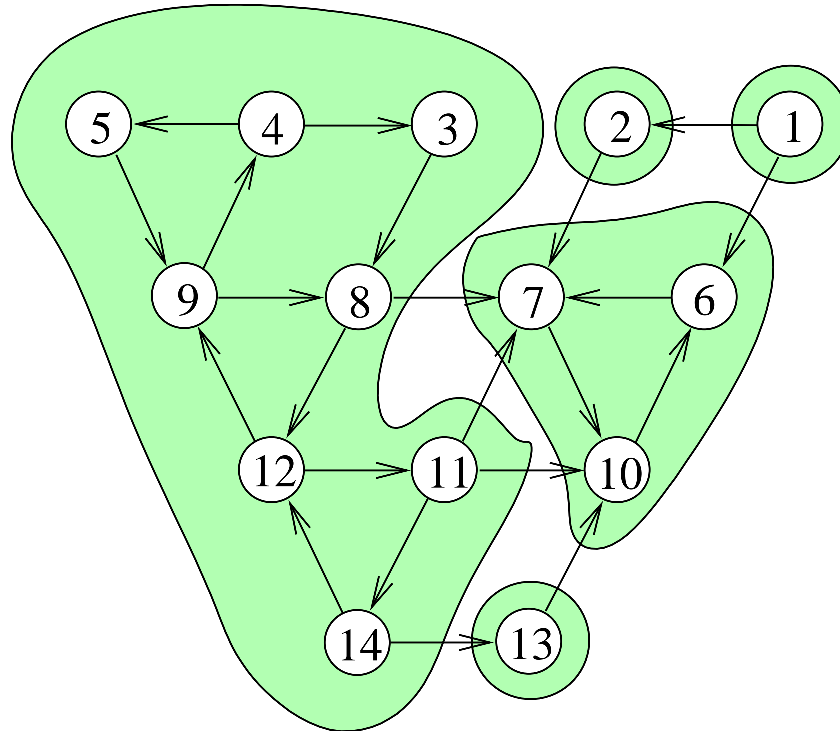
(b) Zwei Knoten v und w sind äquivalent genau dann wenn sie identisch sind **oder** es in G einen (gerichteten) Kreis gibt, auf dem beide Knoten liegen.

Definition 8.5.2

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

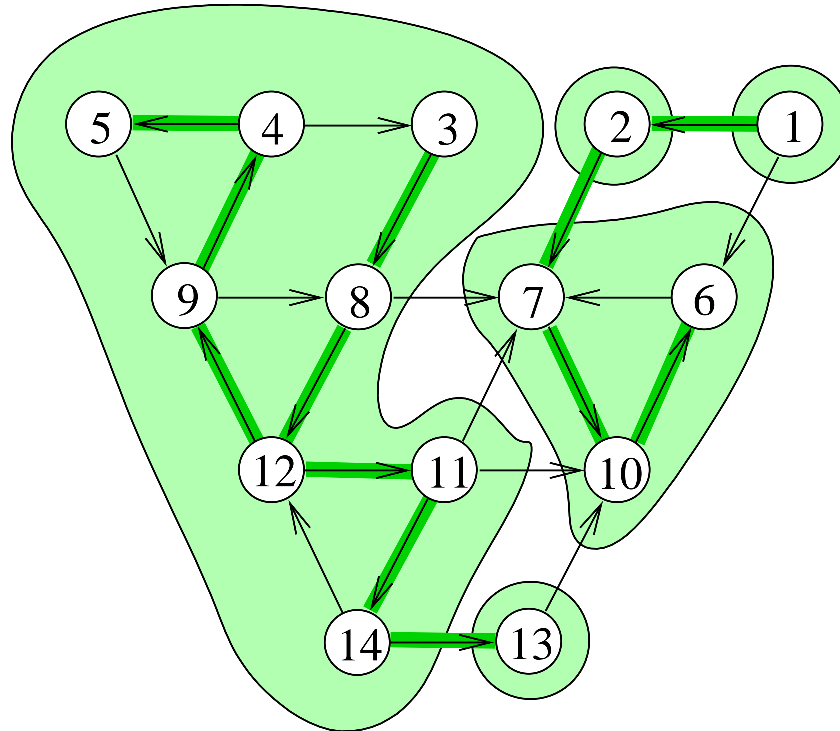
Starke Zusammenhangskomponenten in Digraphen

Beispiel: 5 starke Zusammenhangskomponenten



Starke Zusammenhangskomponenten in Digraphen

Beispiel: 5 starke Zusammenhangskomponenten, 2 DFS-Bäume.



Lemma 8.5.3

Für jede starke Zusammenhangskomponente („**Komponente**") $C \subseteq V$ gilt: Wenn DFS(G) ausgeführt wird, liegen die Knoten von C alle in einem Tiefensuchbaum.

Lemma 8.5.3

Für jede starke Zusammenhangskomponente („**Komponente**") $C \subseteq V$ gilt: Wenn DFS(G) ausgeführt wird, liegen die Knoten von C alle in einem Tiefensuchbaum.

Mit anderen Worten: Jeder Tiefensuchbaum ist selbst eine Komponente oder lässt sich (disjunkt) in mehrere Komponenten zerlegen.

Lemma 8.5.3

Für jede starke Zusammenhangskomponente („**Komponente**“) $C \subseteq V$ gilt: Wenn DFS(G) ausgeführt wird, liegen die Knoten von C alle in einem Tiefensuchbaum.

Mit anderen Worten: Jeder Tiefensuchbaum ist selbst eine Komponente oder lässt sich (disjunkt) in mehrere Komponenten zerlegen.

Beweis: Es sei v_C der erste Knoten in C, für den dfs aufgerufen wird.

Lemma 8.5.3

Für jede starke Zusammenhangskomponente („**Komponente**“) $C \subseteq V$ gilt: Wenn DFS(G) ausgeführt wird, liegen die Knoten von C alle in einem Tiefensuchbaum.

Mit anderen Worten: Jeder Tiefensuchbaum ist selbst eine Komponente oder lässt sich (disjunkt) in mehrere Komponenten zerlegen.

Beweis: Es sei v_C der erste Knoten in C, für den dfs aufgerufen wird. Sei $v \in C$ beliebig.

Lemma 8.5.3

Für jede starke Zusammenhangskomponente („**Komponente**“) $C \subseteq V$ gilt: Wenn DFS(G) ausgeführt wird, liegen die Knoten von C alle in einem Tiefensuchbaum.

Mit anderen Worten: Jeder Tiefensuchbaum ist selbst eine Komponente oder lässt sich (disjunkt) in mehrere Komponenten zerlegen.

Beweis: Es sei v_C der erste Knoten in C, für den dfs aufgerufen wird. Sei $v \in C$ beliebig.
Dann ist v von v_C aus auf einem Weg $(v_C = v_0, v_1, \ldots, v_t = v)$ **in $\boldsymbol{G}$** erreichbar.

Lemma 8.5.3

Für jede starke Zusammenhangskomponente („**Komponente**") $C \subseteq V$ gilt: Wenn DFS(G) ausgeführt wird, liegen die Knoten von C alle in einem Tiefensuchbaum.

Mit anderen Worten: Jeder Tiefensuchbaum ist selbst eine Komponente oder lässt sich (disjunkt) in mehrere Komponenten zerlegen.

Beweis: Es sei v_C der erste Knoten in C, für den dfs aufgerufen wird. Sei $v \in C$ beliebig.
Dann ist v von v_C aus auf einem Weg $(v_C = v_0, v_1, \ldots, v_t = v)$ **in G** erreichbar.

Behauptung: Alle Knoten v_i dieses Weges liegen sogar **in C**.

Lemma 8.5.3

Für jede starke Zusammenhangskomponente („**Komponente**“) $C \subseteq V$ gilt: Wenn DFS(G) ausgeführt wird, liegen die Knoten von C alle in einem Tiefensuchbaum.

Mit anderen Worten: Jeder Tiefensuchbaum ist selbst eine Komponente oder lässt sich (disjunkt) in mehrere Komponenten zerlegen.

Beweis: Es sei v_C der erste Knoten in C, für den dfs aufgerufen wird. Sei $v \in C$ beliebig.
Dann ist v von v_C aus auf einem Weg $(v_C = v_0, v_1, \ldots, v_t = v)$ **in G** erreichbar.

Behauptung: Alle Knoten v_i dieses Weges liegen sogar **in C**.

(*Beweis* hierfür: Weil C Komponente ist, gilt auch $v \leadsto v_C$. Zusammen mit $v_C \leadsto v_i$ und $v_i \leadsto v$ ergibt sich $v_i \leftrightsquigarrow v_C$, also $v_i \in C$.)

Lemma 8.5.3

Für jede starke Zusammenhangskomponente („**Komponente**“) $C \subseteq V$ gilt: Wenn DFS(G) ausgeführt wird, liegen die Knoten von C alle in einem Tiefensuchbaum.

Mit anderen Worten: Jeder Tiefensuchbaum ist selbst eine Komponente oder lässt sich (disjunkt) in mehrere Komponenten zerlegen.

Beweis: Es sei v_C der erste Knoten in C, für den dfs aufgerufen wird. Sei $v \in C$ beliebig.
Dann ist v von v_C aus auf einem Weg $(v_C = v_0, v_1, \ldots, v_t = v)$ **in G** erreichbar.

Behauptung: Alle Knoten v_i dieses Weges liegen sogar **in C**.

(*Beweis* hierfür: Weil C Komponente ist, gilt auch $v \rightsquigarrow v_C$. Zusammen mit $v_C \rightsquigarrow v_i$ und $v_i \rightsquigarrow v$ ergibt sich $v_i \leftrightsquigarrow v_C$, also $v_i \in C$.)

$\Rightarrow$ (nach Beh. und Wahl von $v_0 = v_C$) Zu dem Zeitpunkt, zu dem dfs(v_C) aufgerufen wird, sind die Knoten auf dem Weg $(v_0, v_1, \ldots, v_t = v)$ alle noch neu

Lemma 8.5.3

Für jede starke Zusammenhangskomponente („**Komponente**“) $C \subseteq V$ gilt: Wenn DFS(G) ausgeführt wird, liegen die Knoten von C alle in einem Tiefensuchbaum.

Mit anderen Worten: Jeder Tiefensuchbaum ist selbst eine Komponente oder lässt sich (disjunkt) in mehrere Komponenten zerlegen.

Beweis: Es sei v_C der erste Knoten in C, für den dfs aufgerufen wird. Sei $v \in C$ beliebig.
Dann ist v von v_C aus auf einem Weg $(v_C = v_0, v_1, \ldots, v_t = v)$ **in G** erreichbar.

Behauptung: Alle Knoten v_i dieses Weges liegen sogar **in C**.

(*Beweis* hierfür: Weil C Komponente ist, gilt auch $v \rightsquigarrow v_C$. Zusammen mit $v_C \rightsquigarrow v_i$ und $v_i \rightsquigarrow v$ ergibt sich $v_i \leftrightsquigarrow v_C$, also $v_i \in C$.)

$\Rightarrow$ (nach Beh. und Wahl von $v_0 = v_C$) Zu dem Zeitpunkt, zu dem dfs(v_C) aufgerufen wird, sind die Knoten auf dem Weg $(v_0, v_1, \ldots, v_t = v)$ alle noch neu

$\Rightarrow$ (nach Satz 8.1.3, dem „Satz vom weißen Weg“) v wird Nachfahr von v_C im Tiefensuchbaum. $\square$

Lemma 8.5.3

Für jede starke Zusammenhangskomponente („**Komponente**“) $C \subseteq V$ gilt: Wenn DFS(G) ausgeführt wird, liegen die Knoten von C alle in einem Tiefensuchbaum.

Mit anderen Worten: Jeder Tiefensuchbaum ist selbst eine Komponente oder lässt sich (disjunkt) in mehrere Komponenten zerlegen.

Beweis: Es sei v_C der erste Knoten in C, für den dfs aufgerufen wird. Sei $v \in C$ beliebig.
Dann ist v von v_C aus auf einem Weg $(v_C = v_0, v_1, \ldots, v_t = v)$ **in G** erreichbar.

Behauptung: Alle Knoten v_i dieses Weges liegen sogar **in C**.

(*Beweis* hierfür: Weil C Komponente ist, gilt auch $v \rightsquigarrow v_C$. Zusammen mit $v_C \rightsquigarrow v_i$ und $v_i \rightsquigarrow v$ ergibt sich $v_i \leftrightsquigarrow v_C$, also $v_i \in C$.)

$\Rightarrow$ (nach Beh. und Wahl von $v_0 = v_C$) Zu dem Zeitpunkt, zu dem dfs(v_C) aufgerufen wird, sind die Knoten auf dem Weg $(v_0, v_1, \ldots, v_t = v)$ alle noch neu

$\Rightarrow$ (nach Satz 8.1.3, dem „Satz vom weißen Weg“) v wird Nachfahr von v_C im Tiefensuchbaum. $\square$

Folgerung (aus dem Beweis): v_C hat die kleinste dfs-Nummer und die größte f-Nummer in C.

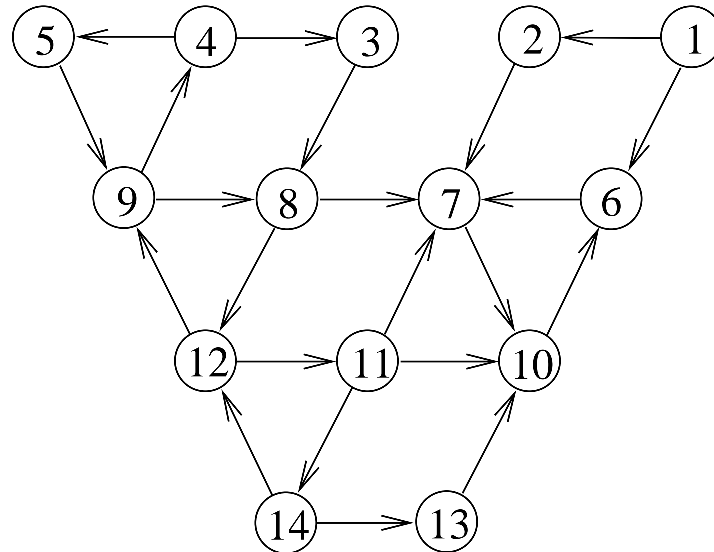
Um mit Tiefensuche die starken Zusammenhangskomponenten zu finden, muss man verhindern, dass in einem Baum mehrere solche Komponenten sitzen (wie im Bild auf Folie 57).

Um mit Tiefensuche die starken Zusammenhangskomponenten zu finden, muss man verhindern, dass in einem Baum mehrere solche Komponenten sitzen (wie im Bild auf Folie 57).

Trick 1: Führe DFS im „Umkehrgraphen" G^{R} durch, 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 (wie im Bild auf Folie 57).

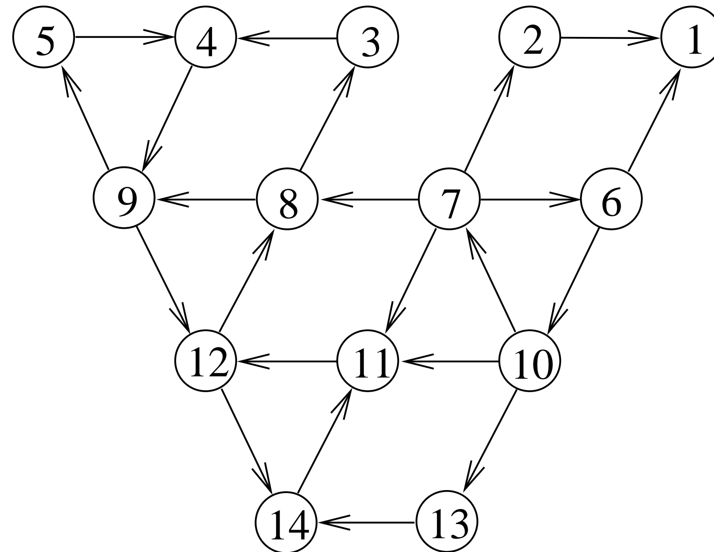
Trick 1: Führe DFS im „Umkehrgraphen“ G^{R} durch, der durch Umkehren aller Kanten in G entsteht.



G im Original.

Um mit Tiefensuche die starken Zusammenhangskomponenten zu finden, muss man verhindern, dass in einem Baum mehrere solche Komponenten sitzen (wie im Bild auf Folie 57).

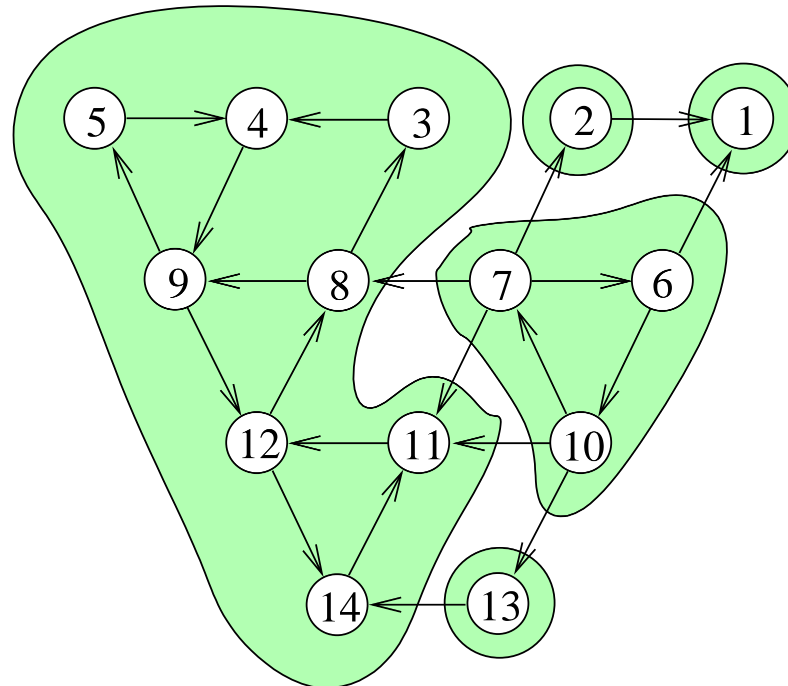
Trick 1: Führe DFS im „Umkehrgraphen“ G^{R} durch, der durch Umkehren aller Kanten in G entsteht.



Umkehrgraph G^{R} von G.

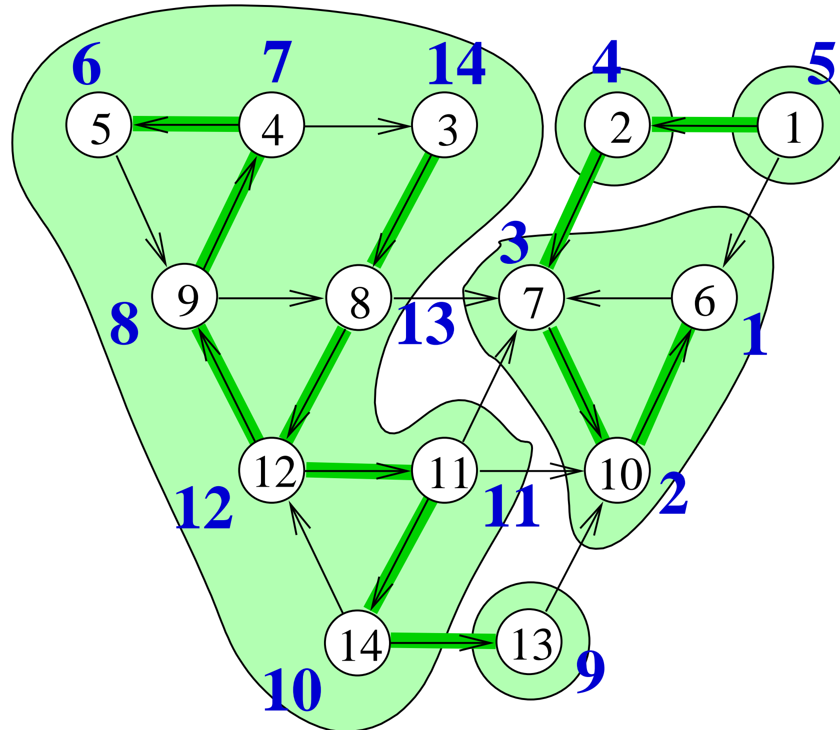
Beobachtung: G^{R} hat dieselben starken Zusammenhangskomponenten wie G.

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



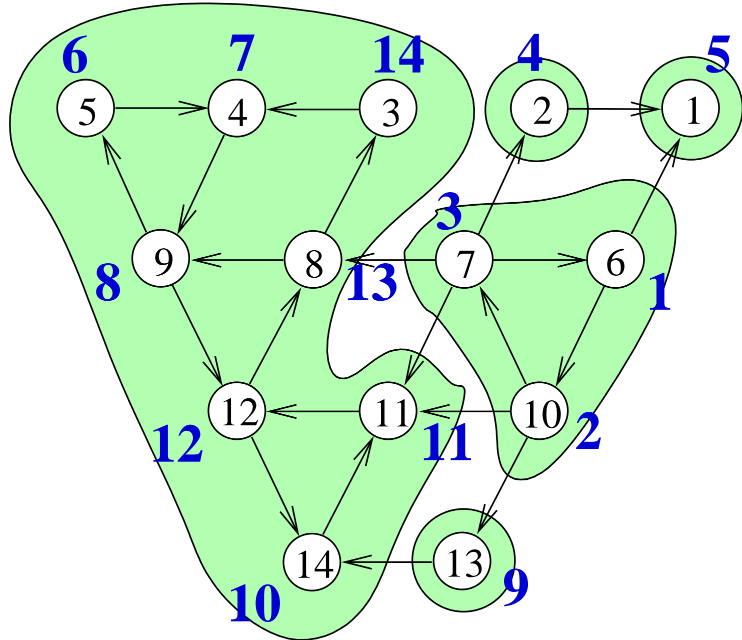
Trick 2: Verwende die **f-Nummern** einer ersten DFS in G, um zu bestimmen, in welcher Reihenfolge die Knoten in der Hauptschleife der zweiten DFS (in G^{R}) betrachtet werden.

Trick 2: Verwende die **f-Nummern** einer ersten DFS in G, um zu bestimmen, in welcher Reihenfolge die Knoten in der Hauptschleife der zweiten DFS (in G^{R}) betrachtet werden.



Graph G mit starken Zusammenhangskomponenten, Tiefensuchwald und f-Nummern aus einem ersten DFS-Aufruf.

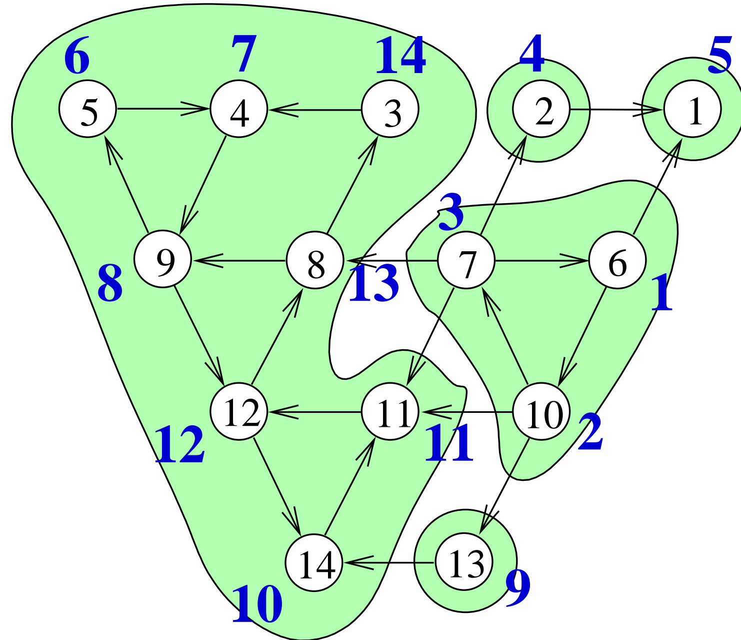
Beispiel:



Graph G^{R} mit Komponenten und f-Nummern für G.

Führe nun DFS in G^{R} durch, untersuche dabei Knoten gemäß **fallenden f-Nummern** auf Eigenschaft „neu“.

Beispiel:



Graph G^{R} mit Komponenten und f-Nummern für G.

Führe nun DFS in G^{R} durch, untersuche dabei Knoten gemäß **fallenden f-Nummern** auf Eigenschaft „neu“.

Im Beispiel: Von 3 **(14)** aus wird eine Komponente erreicht, von 13 **(9)** aus die nächste, von (1) **(5)** aus die nächste, von (2) **(4)** aus die nächste, von 7 **(3)** aus die letzte.

Keine Komponente wird „versehentlich“ betreten.

Algorithmus Strong Components(G) (von **S. R. Kosaraju**, 1978)

Algorithmus Strong Components(G) (von **S. R. Kosaraju**, 1978)

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

Ausgabe: Starke Zusammenhangskomponenten von G

Algorithmus Strong Components(G) (von **S. R. Kosaraju**, 1978)

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

Ausgabe: Starke Zusammenhangskomponenten von G

(1) Berechne die f-Nummern mittels DFS(G);

Algorithmus Strong Components(G) (von **S. R. Kosaraju**, 1978)

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

Ausgabe: Starke Zusammenhangskomponenten von G

(1) Berechne die f-Nummern mittels DFS(G);
(2) Bilde „Umkehrgraphen“ G^{R} durch Umkehren aller Kanten in G;

Algorithmus Strong Components(G) (von **S. R. Kosaraju**, 1978)

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

Ausgabe: Starke Zusammenhangskomponenten von G

(1) Berechne die f-Nummern mittels DFS(G);
(2) Bilde „Umkehrgraphen“ G^{R} durch Umkehren aller Kanten in G;
(3) Führe DFS(G^{R}) durch; Knotenreihenfolge: f-Nummern aus (1) absteigend

Algorithmus Strong Components(G) (von **S. R. Kosaraju**, 1978)

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

Ausgabe: Starke Zusammenhangskomponenten von G

(1) Berechne die f-Nummern mittels DFS(G);
(2) Bilde „Umkehrgraphen“ G^{R} durch Umkehren aller Kanten in G;
(3) Führe DFS(G^{R}) durch; Knotenreihenfolge: f-Nummern aus (1) absteigend
(4) **Ausgabe:** Die Knotenmengen der Tiefensuchbäume aus (3).

Algorithmus Strong Components(G) (von **S. R. Kosaraju**, 1978)

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

Ausgabe: Starke Zusammenhangskomponenten von G

(1) Berechne die f-Nummern mittels DFS(G);
(2) Bilde „Umkehrgraphen" G^{R} durch Umkehren aller Kanten in G;
(3) Führe DFS(G^{R}) durch; Knotenreihenfolge: f-Nummern aus (1) absteigend
(4) **Ausgabe:** Die Knotenmengen der Tiefensuchbäume aus (3).

Satz 8.5.4

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

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

Beweis von Teil (b): Die DFS-Aufrufe haben lineare Laufzeit, und auch der Umkehrgraph lässt sich in Zeit $O(|V|+|E|)$ berechnen.

Beweis von Teil (b): Die DFS-Aufrufe haben lineare Laufzeit, und auch der Umkehrgraph lässt sich in Zeit $O(|V| + |E|)$ berechnen.

Beweis von Teil (a): Nach Lemma 8.5.3 wissen wir, dass jeder DFS-Baum der Tiefensuche in G^{R} Vereinigung von Komponenten (von G^{R}, also von G) ist.

Beweis von Teil (b): Die DFS-Aufrufe haben lineare Laufzeit, und auch der Umkehrgraph lässt sich in Zeit $O(|V| + |E|)$ berechnen.

Beweis von Teil (a): Nach Lemma 8.5.3 wissen wir, dass jeder DFS-Baum der Tiefensuche in G^{R} Vereinigung von Komponenten (von G^{R}, also von G) ist.

Wir zeigen im Folgenden, dass jeder solche Baum nur eine Komponente enthält.

Betrachte dazu eine beliebige starke Zusammenhangskomponente C von G.

Beweis von Teil (b): Die DFS-Aufrufe haben lineare Laufzeit, und auch der Umkehrgraph lässt sich in Zeit $O(|V| + |E|)$ berechnen.

Beweis von Teil (a): Nach Lemma 8.5.3 wissen wir, dass jeder DFS-Baum der Tiefensuche in G^{R} Vereinigung von Komponenten (von G^{R}, also von G) ist.

Wir zeigen im Folgenden, dass jeder solche Baum nur eine Komponente enthält.

Betrachte dazu eine beliebige starke Zusammenhangskomponente C von G.
$\boldsymbol{v_C} \in C$ sei **der Knoten mit maximaler f-Nummer** in C.

(Aus „Folgerung“ nach Lemma 8.5.3: In der ersten Tiefensuche in G ist v_C der erste Knoten von C, der besucht wird, also auch der mit minimaler DFS-Nummer in C.)

Wir wollen zeigen, dass in der Tiefensuche in G^{R} (Teil (3))
Knoten v_C Wurzel eines DFS-Baums wird.

Wir wollen zeigen, dass in der Tiefensuche in G^{R} (Teil (3))
Knoten v_C Wurzel eines DFS-Baums wird.

(D. h.: In DFS(G^{R}) in Teil (3) enthält jede Komponente eine Wurzel eines DFS-Baums.

Wir wollen zeigen, dass in der Tiefensuche in G^{R} (Teil (3))
Knoten v_C Wurzel eines DFS-Baums wird.

(D. h.: In DFS(G^{R}) in Teil (3) enthält jede Komponente eine Wurzel eines DFS-Baums.
Mit Lemma 8.5.3 folgt: Jeder solche DFS-Baum enthält genau eine Komponente, und wir sind fertig.)

Wir wollen zeigen, dass in der Tiefensuche in G^{R} (Teil (3))
Knoten v_C Wurzel eines DFS-Baums wird.

(D. h.: In DFS(G^{R}) in Teil (3) enthält jede Komponente eine Wurzel eines DFS-Baums.
Mit Lemma 8.5.3 folgt: Jeder solche DFS-Baum enthält genau eine Komponente, und wir sind fertig.)

Wegen der besonderen Reihenfolge der Knoten in der äußeren Schleife in Teil (3) gilt nach Satz 8.1.5:

In Teil (3) wird v_C Wurzel eines DFS-Baums

Wir wollen zeigen, dass in der Tiefensuche in G^{R} (Teil (3))
Knoten v_C Wurzel eines DFS-Baums wird.

(D. h.: In DFS(G^{R}) in Teil (3) enthält jede Komponente eine Wurzel eines DFS-Baums.
Mit Lemma 8.5.3 folgt: Jeder solche DFS-Baum enthält genau eine Komponente, und wir sind fertig.)

Wegen der besonderen Reihenfolge der Knoten in der äußeren Schleife in Teil (3) gilt nach Satz 8.1.5:

In Teil (3) wird v_C Wurzel eines DFS-Baums

$\Leftrightarrow$ es gibt kein v mit *f-num*$(v) >$ *f-num*(v_C) und $v \leadsto_{G^{\mathsf{R}}} v_C$

Wir wollen zeigen, dass in der Tiefensuche in G^{R} (Teil (3)) Knoten v_C Wurzel eines DFS-Baums wird.

(D. h.: In DFS(G^{R}) in Teil (3) enthält jede Komponente eine Wurzel eines DFS-Baums. Mit Lemma 8.5.3 folgt: Jeder solche DFS-Baum enthält genau eine Komponente, und wir sind fertig.)

Wegen der besonderen Reihenfolge der Knoten in der äußeren Schleife in Teil (3) gilt nach Satz 8.1.5:

In Teil (3) wird v_C Wurzel eines DFS-Baums

$\Leftrightarrow$ es gibt kein v mit *f-num*$(v) >$ *f-num*(v_C) und $v \leadsto_{G^{\mathsf{R}}} v_C$

$\Leftrightarrow$ es gibt kein v mit *f-num*$(v) >$ *f-num*(v_C) und $v_C \leadsto_G v$

Wir wollen zeigen, dass in der Tiefensuche in G^{R} (Teil (3))
Knoten v_C Wurzel eines DFS-Baums wird.

(D. h.: In DFS(G^{R}) in Teil (3) enthält jede Komponente eine Wurzel eines DFS-Baums.
Mit Lemma 8.5.3 folgt: Jeder solche DFS-Baum enthält genau eine Komponente, und wir sind fertig.)

Wegen der besonderen Reihenfolge der Knoten in der äußeren Schleife in Teil (3) gilt nach Satz 8.1.5:

In Teil (3) wird v_C Wurzel eines DFS-Baums

$\Leftrightarrow$ es gibt kein v mit *f-num*$(v) >$ *f-num*(v_C) und $v \rightsquigarrow_{G^{\mathsf{R}}} v_C$

$\Leftrightarrow$ es gibt kein v mit *f-num*$(v) >$ *f-num*(v_C) und $v_C \rightsquigarrow_G v$

$\Leftrightarrow$ für alle v mit $v_C \rightsquigarrow_G v$ gilt *f-num*$(v) \leq$ *f-num*(v_C).

Wir wollen zeigen, dass in der Tiefensuche in G^{R} (Teil (3))
Knoten v_C Wurzel eines DFS-Baums wird.

(D. h.: In DFS(G^{R}) in Teil (3) enthält jede Komponente eine Wurzel eines DFS-Baums.
Mit Lemma 8.5.3 folgt: Jeder solche DFS-Baum enthält genau eine Komponente, und wir sind fertig.)

Wegen der besonderen Reihenfolge der Knoten in der äußeren Schleife in Teil (3) gilt nach Satz 8.1.5:

In Teil (3) wird v_C Wurzel eines DFS-Baums

$\Leftrightarrow$ es gibt kein v mit $\mathit{f\text{-}num}(v) > \mathit{f\text{-}num}(v_C)$ und $v \rightsquigarrow_{G^{\mathsf{R}}} v_C$

$\Leftrightarrow$ es gibt kein v mit $\mathit{f\text{-}num}(v) > \mathit{f\text{-}num}(v_C)$ und $v_C \rightsquigarrow_G v$

$\Leftrightarrow$ für alle v mit $v_C \rightsquigarrow_G v$ gilt $\mathit{f\text{-}num}(v) \leq \mathit{f\text{-}num}(v_C)$.

Es genügt also, Folgendes zu zeigen:

Lemma 8.5.5

Aus $v_C \rightsquigarrow_G v$ folgt $\mathit{f\text{-}num}(v) \leq \mathit{f\text{-}num}(v_C)$ (in DFS in Teil (1)!).

Lemma 8.5.5 Aus $v_C \rightsquigarrow_G v$ folgt $\textit{f-num}(v) \le \textit{f-num}(v_C)$.

Lemma 8.5.5 Aus $v_C \leadsto_G v$ folgt $\textit{f-num}(v) \le \textit{f-num}(v_C)$.

Beweis: O. B. d. A. gilt $v_C \neq v$.

Lemma 8.5.5 Aus $v_C \rightsquigarrow_G v$ folgt $\textit{f-num}(v) \leq \textit{f-num}(v_C)$.

Beweis: O. B. d. A. gilt $v_C \neq v$.

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

Lemma 8.5.5 Aus $v_C \rightsquigarrow_G v$ folgt $\textit{f-num}(v) \leq \textit{f-num}(v_C)$.

Beweis: O. B. d. A. gilt $v_C \neq v$.

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

1. Fall: Wenn dfs(v_C) startet, sind alle Knoten auf p $\boxed{\textbf{neu}}$.

Lemma 8.5.5 Aus $v_C \rightsquigarrow_G v$ folgt *f-num*$(v) \leq$ *f-num*(v_C).

Beweis: O. B. d. A. gilt $v_C \neq v$.

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

1. Fall: Wenn dfs(v_C) startet, sind alle Knoten auf p **neu** .
Nach Satz 8.1.3 („Satz vom weißen Weg“) wird v Nachfahr von v_C im DFS-Baum, also ist *f-num*$(v) <$ *f-num*(v_C).

Lemma 8.5.5 Aus $v_C \rightsquigarrow_G v$ folgt *f-num*$(v) \le$ *f-num*(v_C).

Beweis: O. B. d. A. gilt $v_C \neq v$.

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

1. Fall: Wenn dfs(v_C) startet, sind alle Knoten auf p **neu** .
Nach Satz 8.1.3 („Satz vom weißen Weg“) wird v Nachfahr von v_C im DFS-Baum, also ist *f-num*$(v) <$ *f-num*(v_C).

2. Fall: Wenn dfs(v_C) startet, ist ein Knoten v_i, $0 < i \le t$, **aktiv** .
Dann führt ein Schluss-Stück des roten Weges (Folie 12) von v_i zu v_C.
Von v_C nach v_i führt der erste Teil des Weges p.
Daraus: $v_C \leftrightsquigarrow v_i$, also $v_i \in C$.

Lemma 8.5.5 Aus $v_C \rightsquigarrow_G v$ folgt $\textit{f-num}(v) \leq \textit{f-num}(v_C)$.

Beweis: O. B. d. A. gilt $v_C \neq v$.

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

1. Fall: Wenn dfs(v_C) startet, sind alle Knoten auf p **neu** .
Nach Satz 8.1.3 („Satz vom weißen Weg“) wird v Nachfahr von v_C im DFS-Baum, also ist $\textit{f-num}(v) < \textit{f-num}(v_C)$.

2. Fall: Wenn dfs(v_C) startet, ist ein Knoten v_i, $0 < i \leq t$, **aktiv** .
Dann führt ein Schluss-Stück des roten Weges (Folie 12) von v_i zu v_C.
Von v_C nach v_i führt der erste Teil des Weges p.
Daraus: $v_C \leftrightsquigarrow v_i$, also $v_i \in C$.
Andererseits ist $\textit{f-num}(v_i) > \textit{f-num}(v_C)$, im **Widerspruch** zur Wahl von v_C.
Der 2. Fall kann also gar nicht eintreten.

3. Fall: Wenn dfs(v_C) startet, ist ein Knoten v_i, $0 < i \leq t$, **fertig**.

3. Fall: Wenn dfs(v_C) startet, ist ein Knoten v_i, $0 < i \le t$, **fertig**.

Wegen des Vorgehens der dfs-Prozedur und weil Fall 2 nicht eintritt, gilt für $i \le j < t$, zu dem Zeitpunkt, an dem dfs(v_C) startet:

Wenn v_j „fertig" ist, dann ist auch v_{j+1} „fertig".
(Bevor dfs(v_j) endete, wurde Kante (v_j, v_{j+1}) betrachtet; also kann v_{j+1} nicht „neu" sein.)

Also ist v „fertig", wenn dfs(v_C) startet.

Daraus folgt: *f-num*$(v) <$ *f-num*(v_C). □

PAUSE

Ende des 8. Kapitels