

SS 2021

Algorithmen und Datenstrukturen

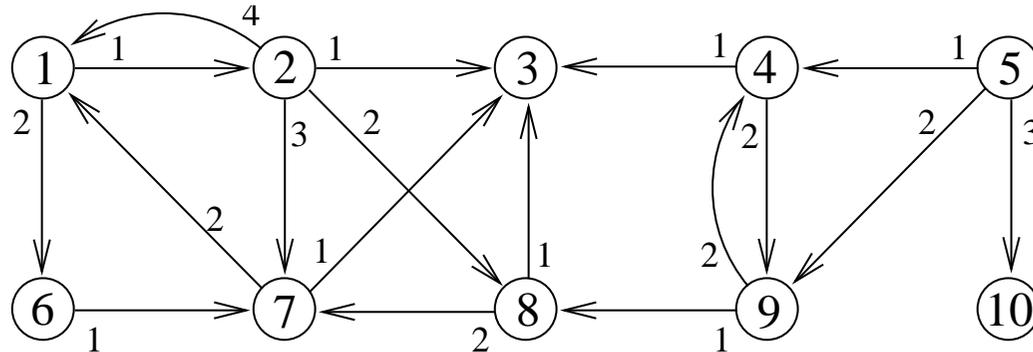
8. Kapitel

Tiefensuche

Martin Dietzfelbinger

Juni 2021

8.1 Einfache Tiefensuche in Digraphen



Eingabeformat:

Digraph $G = (V, E)$, Knotenmenge $V = \{1, \dots, n\}$ (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)

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 $w_1, \dots, w_{\text{outdeg}(v)}$ 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.

Dazu: „Statusinformation“, in Array `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:

`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).

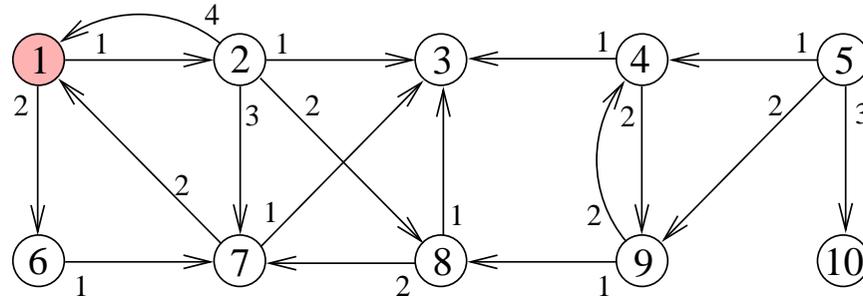
Prozedur $\text{dfs}(v)$ // Tiefensuche an v , **rekursiv**,
// nur für v mit $\text{status}[v] = \boxed{\text{neu}}$ aufzurufen

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

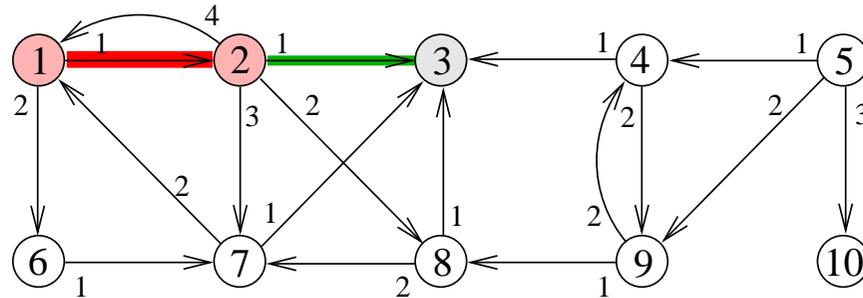
Tiefensuche von v_0 aus:

Initialisiere alle Knoten als „ $\boxed{\text{neu}}$ “; setze $\text{dfs_count} \leftarrow 0$.

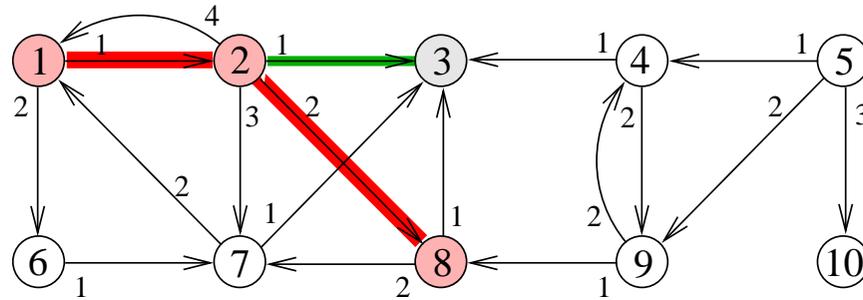
Rufe $\text{dfs}(v_0)$ auf.



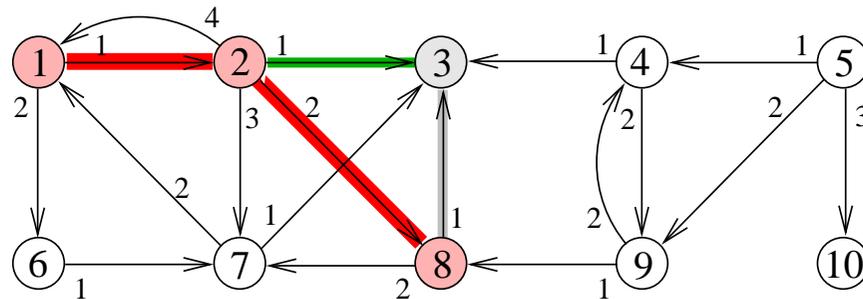
Start von $\text{dfs}(1)$, „Roter Weg“: **(1)**.



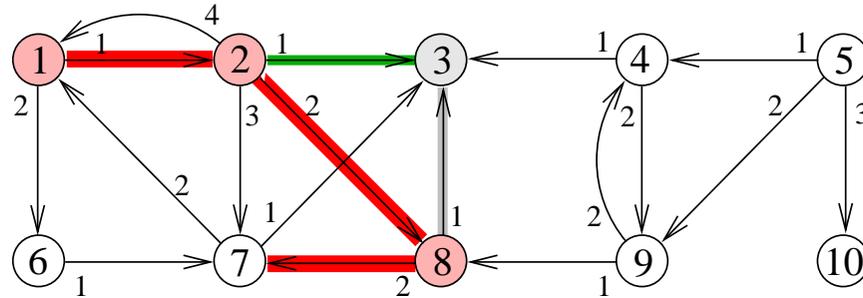
Ende von $\text{dfs}(3)$, „Roter Weg“: **(1, 2)**.



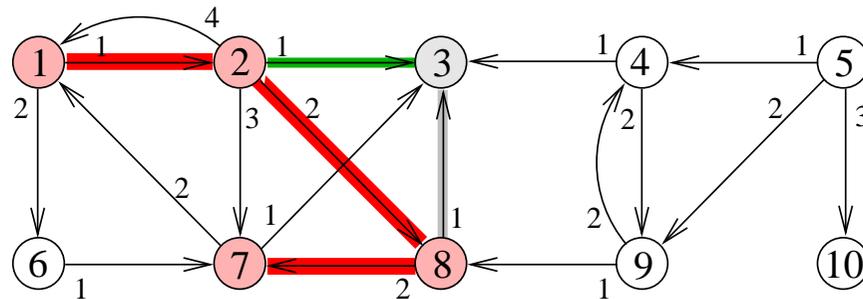
Start von $\text{dfs}(8)$, „Roter Weg“: **(1, 2, 8)**.



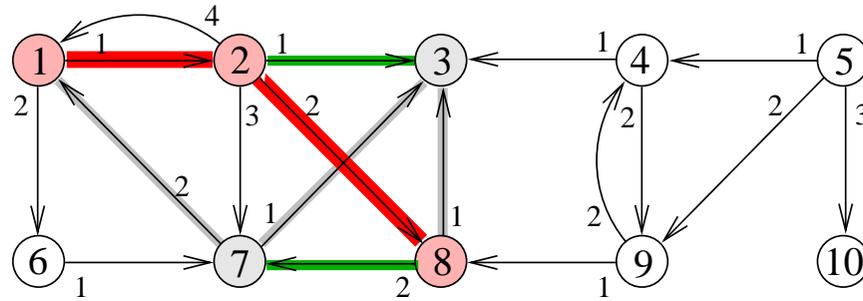
Kante (8, 3) wird angesehen, Knoten 3 ist **fertig** .



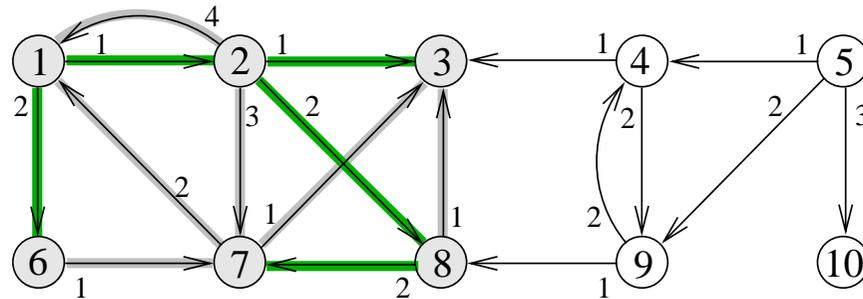
Kante (8, 7) wird angesehen, Knoten 7 ist **neu** .



Start von $\text{dfs}(7)$, „Roter Weg“: **(1, 2, 8, 7)**.



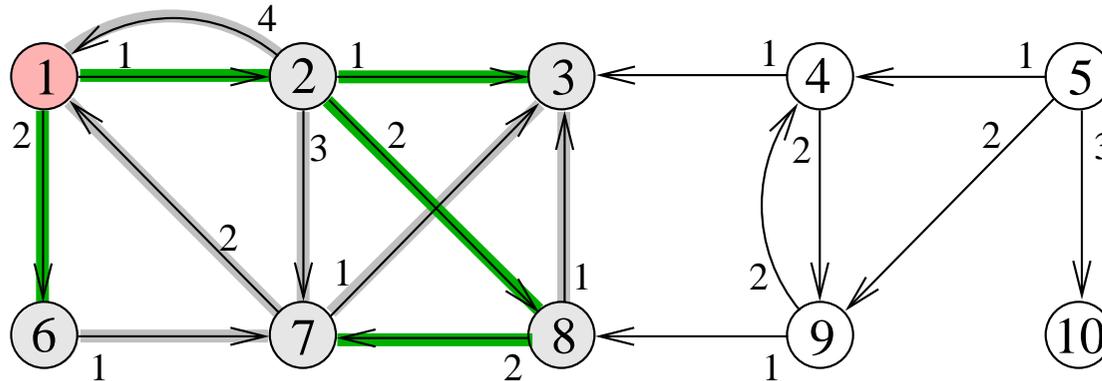
dfs(7) endet, „Roter Weg“: **(1, 2, 8)**.



Fertig! Erreichbare Knoten und Kanten, **DFS-Baum** zu Startknoten 1.

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

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



Satz 8.1.1

Bei Tiefensuche von v_0 aus wird für jeden Knoten $v \in R_{v_0}$ $\text{dfs}(v)$ (genau einmal) aufgerufen, für Knoten $v \notin R_{v_0}$ wird $\text{dfs}(v)$ 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 $\text{dfs}(v)$ nur aufgerufen werden, wenn $v_0 \rightsquigarrow v$ gilt.

(2) Sobald $\text{dfs}(v)$ aufgerufen wird, wird $\text{status}[v]$ 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 $(v_0, v_1, \dots, v_t = v)$ ein Weg von v_0 nach v .

Annahme: Es gibt $i > 0$, für das $\text{dfs}(v_i)$ nicht aufgerufen wird.

Betrachte das kleinste solche i . Dann wird $\text{dfs}(v_{i-1})$ aufgerufen.

Dabei wird die Kante (v_{i-1}, v_i) untersucht und festgestellt, dass v_i „neu“ ist. Also wird $\text{dfs}(v_i)$ aufgerufen, **Widerspruch**.

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

Beobachtung 1:

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

Der **Zeitaufwand** für den gesamten Aufruf $\text{dfs}(v_0)$ ist **linear**, nämlich

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

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

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

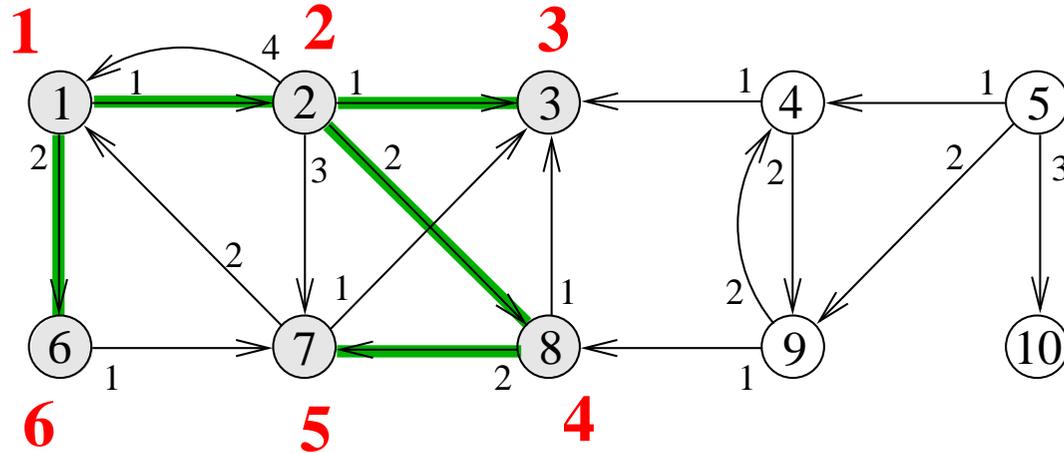
Die (**reflexive** und) **transitive Hülle** eines Digraphen G ist der Digraph **TH**(G) := (V, E^*) mit Kantenmenge

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

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

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

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

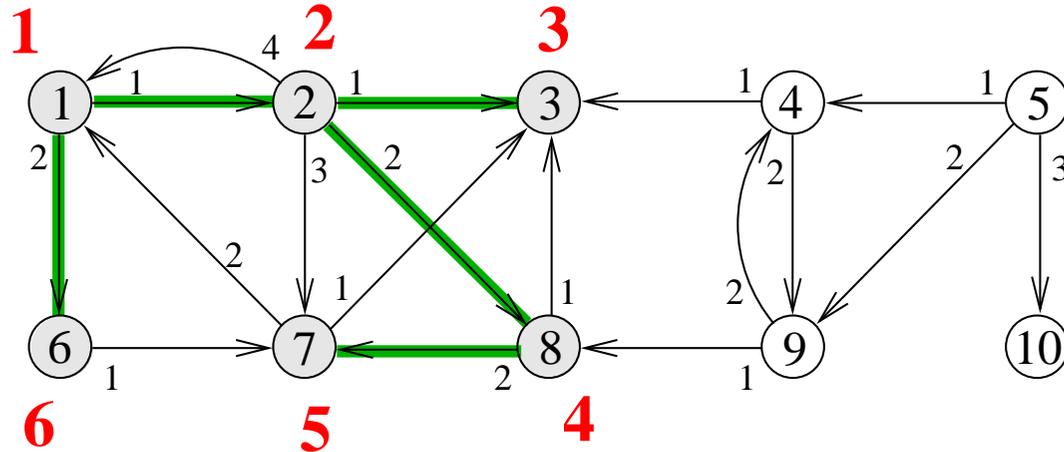


Beobachtung 2:

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 .

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

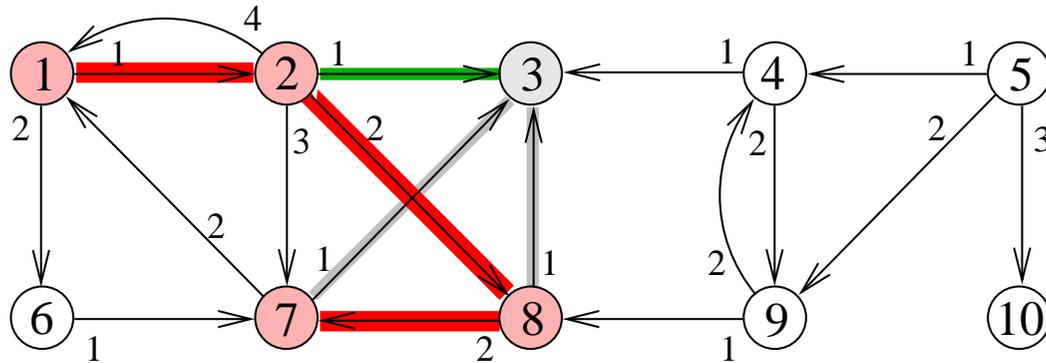
Man beachte: Aufruf „dfs(v)“ **beginnt nach** und **endet vor** Aufruf „dfs($p(v)$)“.



Die Kanten $(p(v), v)$ ($\hat{=}$ direkter Aufruf $\text{dfs}(v)$ aus $\text{dfs}(p(v))$) 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.



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

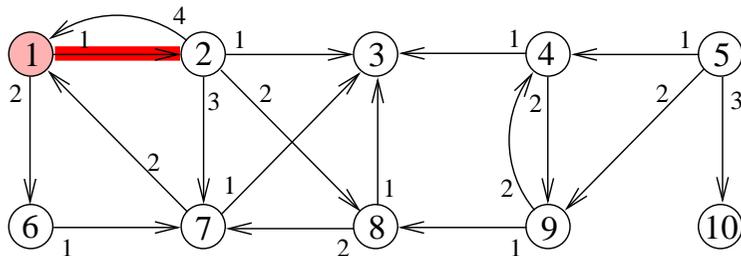
Startpunkt ist v_0 , Endpunkt ist der Knoten v , in dessen „dfs(v)“-Aufruf eben gearbeitet wird. (Hier: Knoten 7.)

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

Wie entscheidet sich, ob w in $T_{\text{dfs}}(v_0)$ Nachfahr von v ist? D.h.: Wie kann man sehen, ob $\text{dfs}(w)$ 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 $\text{dfs}(v)$ aufgerufen wird, existiert ein Weg von v nach w
 aus Knoten, die alle **neu** („weiß“) sind.



Beispiel: $\text{dfs}(2)$ startet.

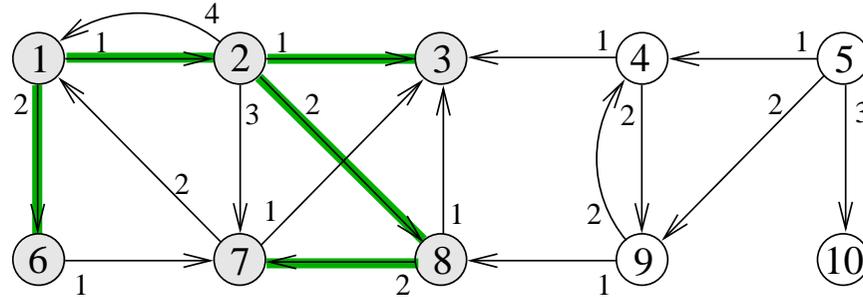
Knoten 3, 7, 8 sind auf „weißen Wegen“ von 2 aus erreichbar

\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: „ \Rightarrow “:



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: $(v = u_0, u_1, u_2, \dots, u_t = w)$.

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

Nach Beobachtung 2 wird erst $\text{dfs}(v)$ aufgerufen, dann $\text{dfs}(u_1)$, dann $\text{dfs}(u_2)$, usw.; also sind in dem Moment, in dem der Aufruf $\text{dfs}(v)$ erfolgt, alle Knoten $v = u_0, u_1, u_2, \dots, u_t = w$ noch **neu**.

Beweis von Satz 8.1.3: „ \Leftarrow “:

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

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

I.A.: Die Beh. ist für $i = 0$ klar.

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

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

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

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

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

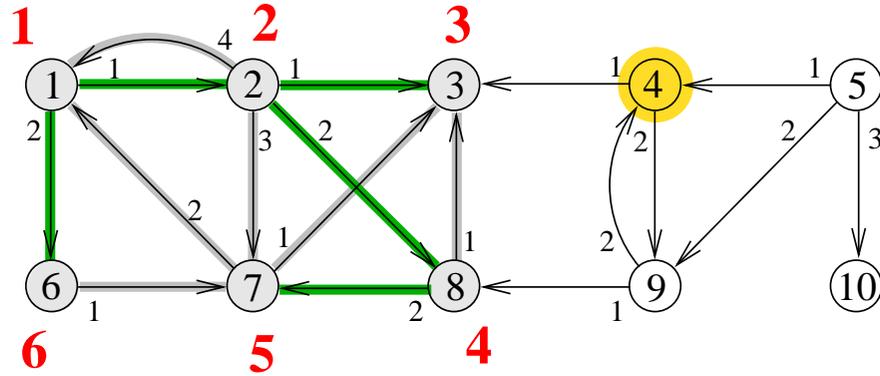
Demnach wird $\text{dfs}(w)$ (direkt oder indirekt) aus $\text{dfs}(v)$ aufgerufen, also ist w Nachfahr von v im DFS-Baum. □

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

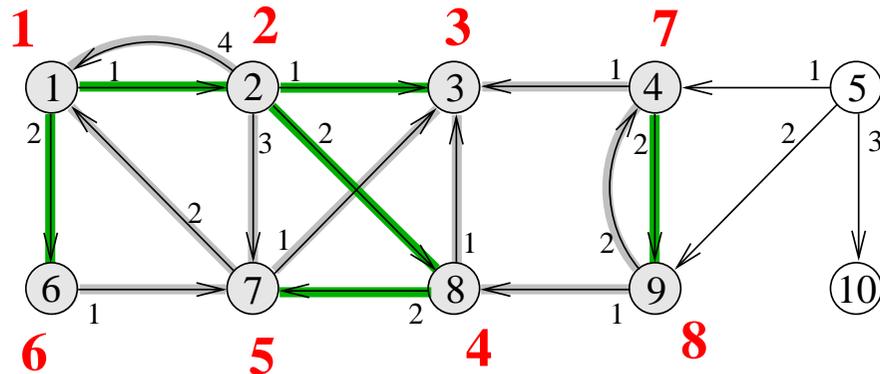
Algorithmus DFS(G) // Tiefensuche in $G = (V, E)$ mit $V = \{1, \dots, n\}$

- (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

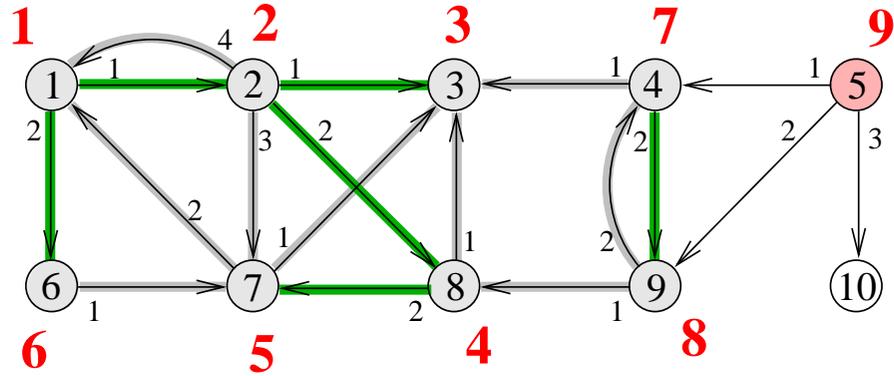
Zu Zeilen (2), (4): Die `status`[v]-Werte werden von den `dfs`-Aufrufen in Zeile (5) verändert; sie haben also beim Wieder-Lesen in der zweiten Schleife nicht unbedingt immer noch den Initialisierungswert **neu**.



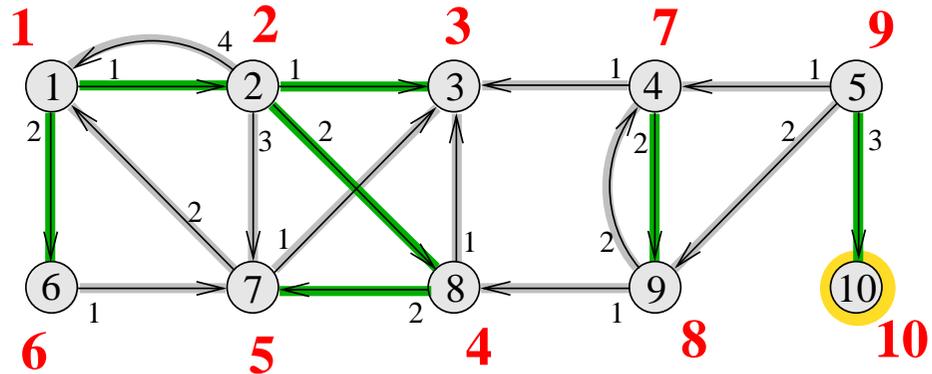
Knoten 4 ist neu , also starte $\text{dfs}(4)$.



$\text{dfs}(4)$ endet.



dfs(5) startet.



Letzter Test: Ist Knoten **10** neu ? Nein, also fertig.

DFS(G) erzwingt, dass für **jeden** Knoten v in V irgendwann einmal $\text{dfs}(v)$ 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:

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

Satz 8.1.4

Der Zeitaufwand von $\text{DFS}(G)$ ist $O(|V| + |E|)$.

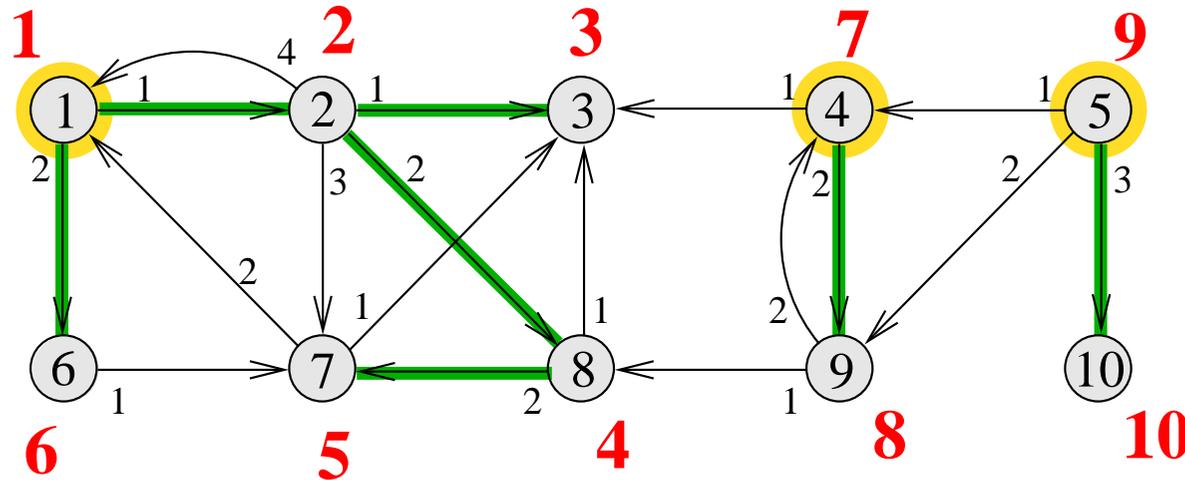
(Sogar: $\Theta(|V| + |E|)$, da jeder Knoten und jede Kante betrachtet wird.)

Bemerkung:

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

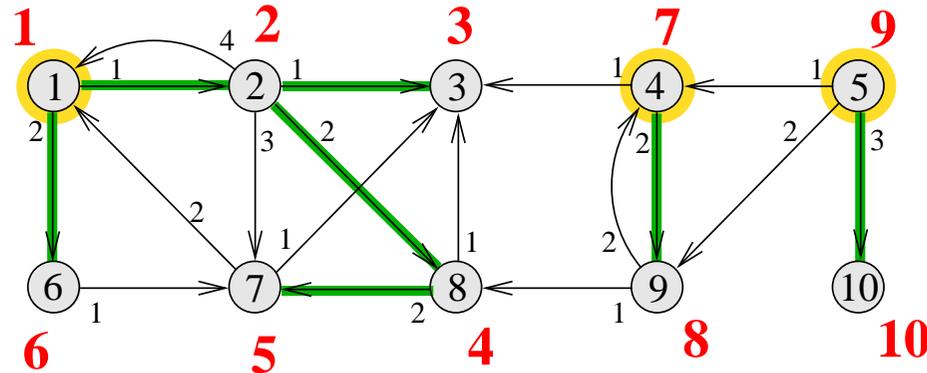
(Kante (w, v) entspricht dem Aufruf $\text{dfs}(v)$ direkt aus $\text{dfs}(w)$.)

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 $5 \rightsquigarrow 1$, 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.



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

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 $\Leftrightarrow v \notin W_{v-1}$.

(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.)

Beweis: (a) „ \Leftarrow “: Sei $v \notin W_{v-1}$. v kann nicht in einem Baum mit einer Wurzel $w < v$ sitzen, weil es überhaupt keinen Weg von w nach v gibt.

Also ist v selbst eine Wurzel.

„ \Rightarrow “: Sei $v \in W_{v-1}$.

Betrachte einen Weg $(w = v_0, v_1, \dots, v_t = v)$ mit $w < v$.

Der Aufruf $\text{dfs}(w)$ erfolgt spätestens, wenn w in der äußeren Schleife von $\text{DFS}(G)$ getestet wird, also bevor v getestet wird.

Betrachte den Knoten v_i auf dem Weg, dessen Aufruf $\text{dfs}(v_i)$ **zuerst** erfolgt.

Wenn dieser Aufruf erfolgt, sind v_i, v_{i+1}, \dots, v_t noch neu.

Nach Satz 8.1.3, dem „Satz vom weißen Weg“, wird v Nachfahr von v_i , also ist v keine Wurzel.

(b) Sei v Wurzel und sei $v = v_0, v_1, \dots, v_t = w$ ein Weg von v zu $w \in R_v - W_{v-1}$. Dann sind offenbar alle Knoten v_i auf dem Weg in $R_v - W_{v-1}$.

Wenn also der Aufruf $\text{dfs}(v)$ erfolgt, sind alle Knoten $v = v_0, v_1, \dots, v_t = w$ neu.

Nach dem Satz vom weißen Weg wird w Nachfahr von v . □

8.2 Volle Tiefensuche in Digraphen

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

- T („Tree“): **Baumkanten**, die Kanten des Tiefensuchwaldes.

Kriterium: Wenn $\text{dfs}(v)$ die Kante (v, w) betrachtet, ist w noch **neu**.

- B („Back“): **Rückwärtskanten**.

(v, w) ist Rückwärtskante, wenn w Vorfahr von v im Tiefensuchwald ist.

Kriterium: Wenn $\text{dfs}(v)$ 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.

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

- **C : Querkanten** („Cross“).

(v, w) ist Querkante, wenn der gesamte Unterbaum von w schon fertig abgearbeitet ist, wenn $\text{dfs}(v)$ aufgerufen wird.

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

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

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

Prozedur $\text{dfs}(v)$ // „volle“ Tiefensuche in v , nur für v **neu** erlaubt

- (1) $\text{dfs_count}++;$
- (2) $\text{dfs_num}[v] \leftarrow \text{dfs_count};$
- (3) $\text{status}[v] \leftarrow \text{aktiv};$
- (4) **dfs-visit**(v); // Aktion an v bei Entdeckung
- (5) für jeden Nachfolger w von v (Adjazenzliste!) tue:
 - (6) 1. Fall: $\text{status}[w] = \text{neu}$: $T \leftarrow T \cup \{(v, w)\}; \text{dfs}(w);$
 - (7) 2. Fall: $\text{status}[w] = \text{aktiv}$: $B \leftarrow B \cup \{(v, w)\};$
 - (8) 3. Fall: $\text{status}[w] = \text{fertig} \wedge \text{dfs_num}[v] < \text{dfs_num}[w]$:
 $F \leftarrow F \cup \{(v, w)\};$
 - (9) 4. Fall: $\text{status}[w] = \text{fertig} \wedge \text{dfs_num}[v] > \text{dfs_num}[w]$:
 $C \leftarrow C \cup \{(v, w)\};$
- (10) $\text{f_count}++;$
- (11) $\text{f_num}[v] \leftarrow \text{f_count};$
- (12) **fin-visit**(v); // Aktion an v bei Abschluss
- (13) $\text{status}[v] \leftarrow \text{fertig}.$

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

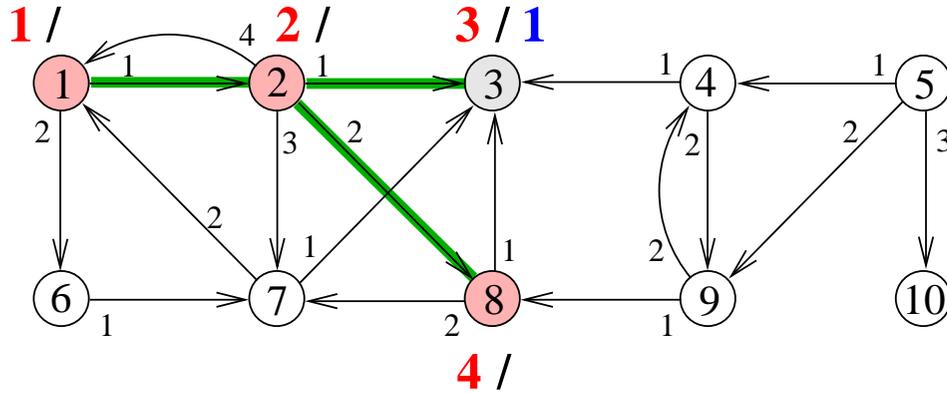
Alle Knoten sind **neu** ; Mengen T , B , F , C sind leer.

Tiefensuche von v_0 aus („dfs(v_0)“) entdeckt R_{v_0} .

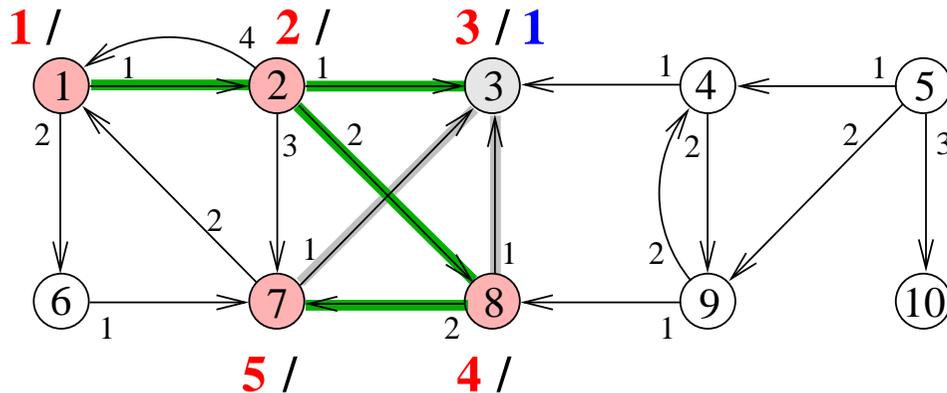
Tiefensuche **für den ganzen Graphen**: DFS(G) wie vorher, nur mit der voll ausgebauten dfs-Prozedur, und der Initialisierung von f_count und T , B , F , C .

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

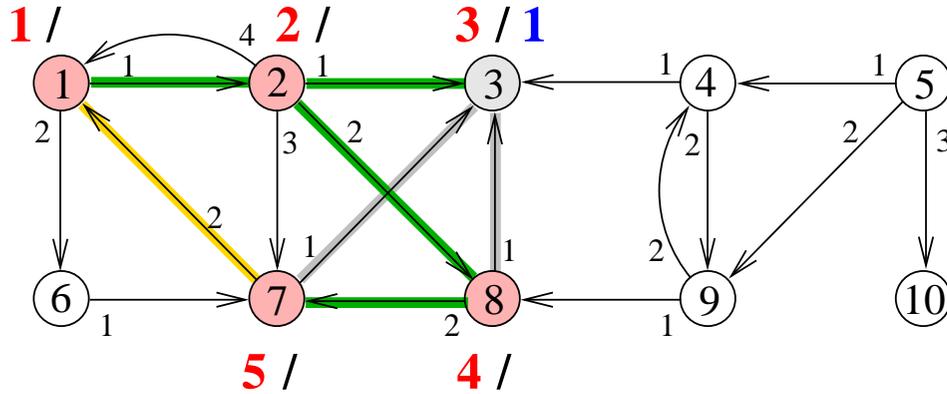
- (1) $\text{dfs_count} \leftarrow 0$;
- (2) $\text{f_count} \leftarrow 0$;
- (3) $T \leftarrow \emptyset$; $B \leftarrow \emptyset$; $F \leftarrow \emptyset$; $C \leftarrow \emptyset$;
- (3) **for** v **from** 1 **to** n **do** $\text{status}[v] \leftarrow \text{neu}$;
- (4) **for** v **from** 1 **to**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



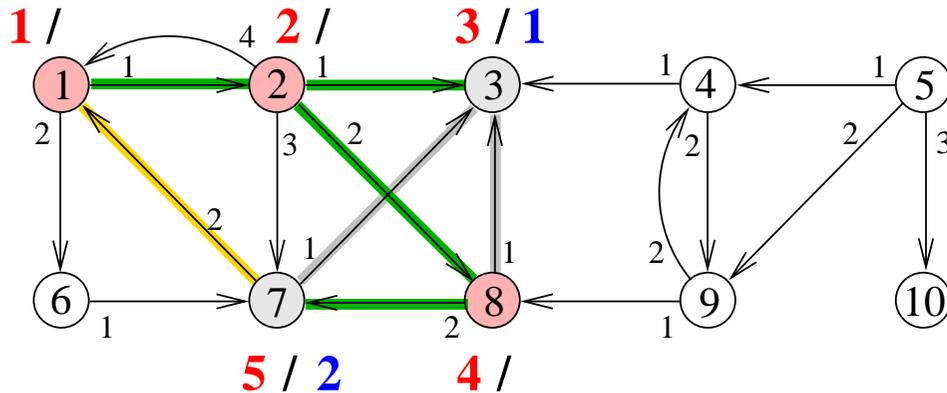
dfs(8) startet, dfs-Nummer **4**,
 (2, 8) wird Baumkante.



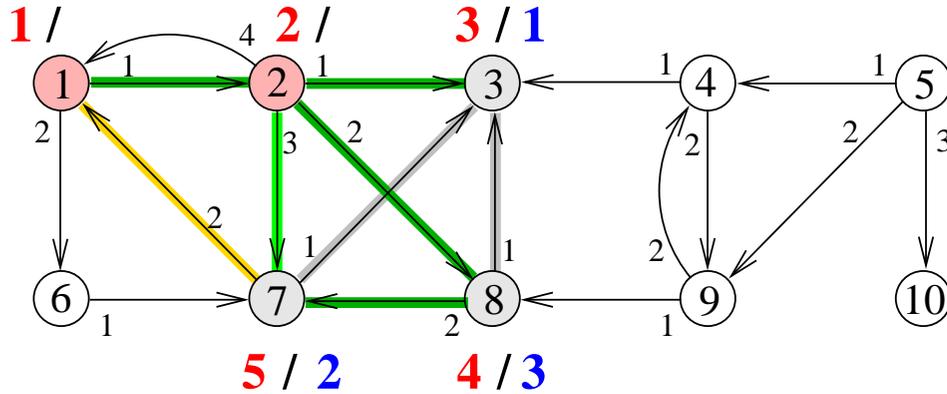
(7, 3) wird **Querkante**,
 weil 3 **fertig** ist. und
5 > 3 gilt.



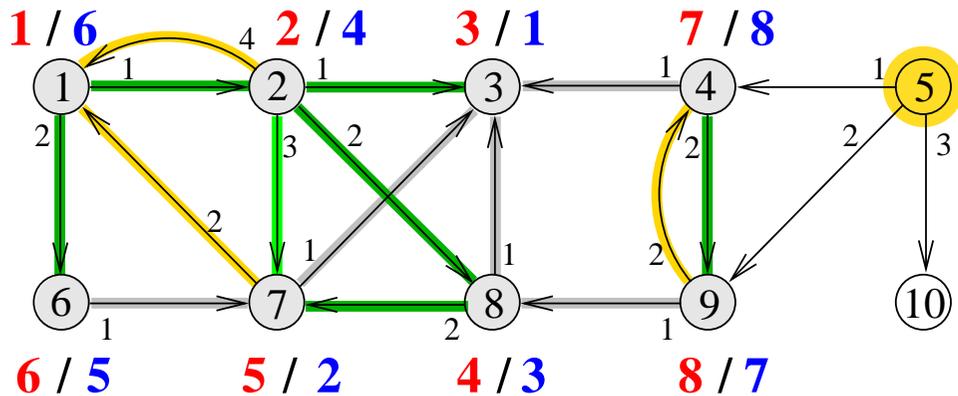
(7, 1) wird **Rückwärtskante**,
weil 1 **aktiv** ist.



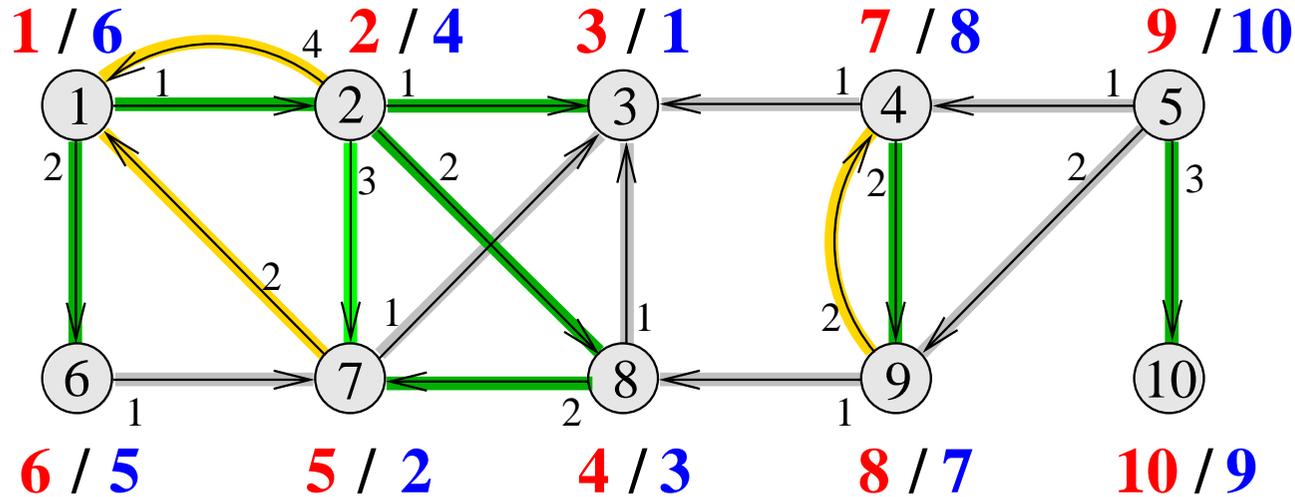
dfs(7) endet,
f-Nummer **2**.



(2, 7) wird **Vorwärtskante**,
weil 7 **fertig** ist und $2 < 5$ gilt.



5 wird getestet,
ist noch **neu**,
dfs(5) wird gestartet.



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) $\text{dfs}(w)$ beginnt nach $\text{dfs}(v)$ und endet vor $\text{dfs}(v)$
(d. h. $\text{dfs}(w)$ wird (direkt oder indirekt) von $\text{dfs}(v)$ aus aufgerufen);
- (b) $\text{dfs_num}[v] < \text{dfs_num}[w] \wedge \text{f_num}[v] > \text{f_num}[w]$;
- (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 $\text{DFS}(G)$ klassifiziert die Kanten korrekt.

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

1. Fall: $\text{dfs}(w)$ wird direkt aus $\text{dfs}(v)$ aufgerufen.

Dann ist (v, w) **Baumkante** und wird richtig als **T** klassifiziert.

2. Fall: Die Untersuchung der Kante (v, w) in $\text{dfs}(v)$ 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 **B** klassifiziert.

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

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

Dann ist w nach Beobachtung 3 Nachfahr von v im Tiefensuchwald, also ist (v, w) **Vorwärtskante** und wird richtig als **F** klassifiziert.

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

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

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 **C** klassifiziert.

8.3 Tiefensuche in ungerichteten Graphen

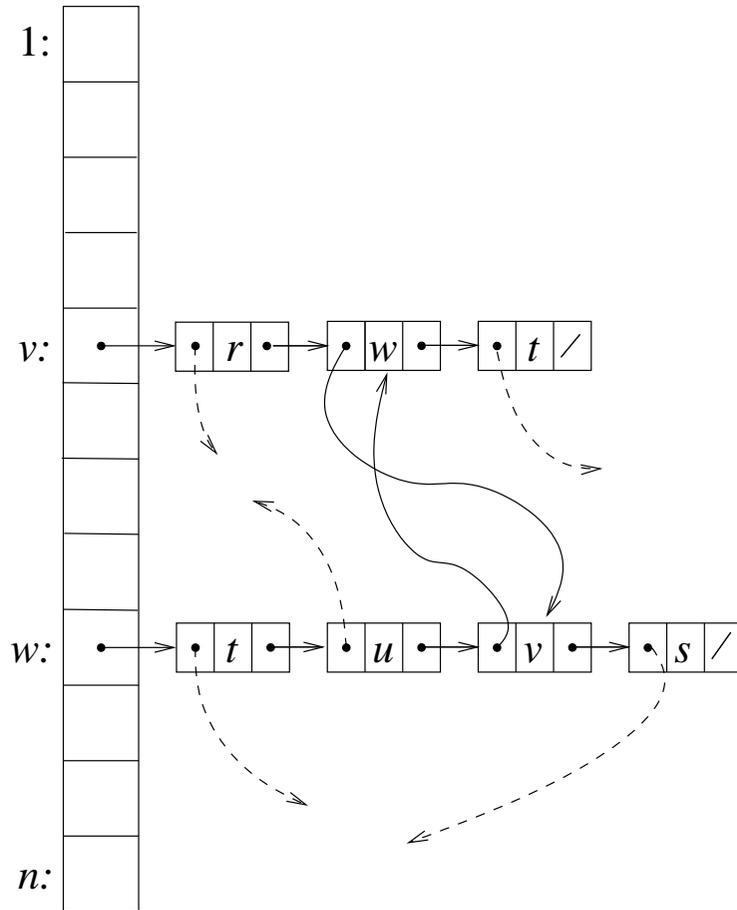
Einfacher als Tiefensuche in gerichteten Graphen.

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: 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.

Klassifizierung in T -Kanten und B -Kanten.

Algorithmus $\text{udfs}(v)$ // Tiefensuche von v aus, rekursiv

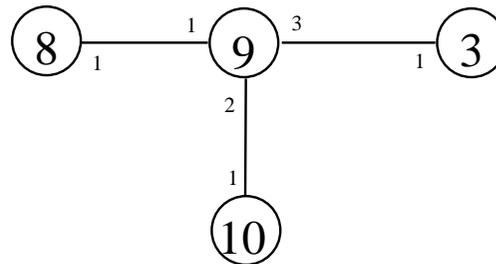
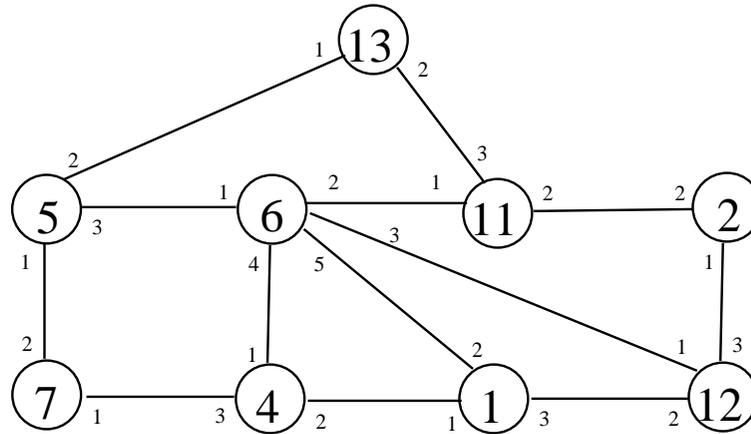
// nur für v mit $\text{status}[v] = \text{neu}$ aufrufen

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

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.

Satz 8.3.1 $\mathbf{udfs}(v_0)$ werde aufgerufen. Dann gilt:

- (a) $\mathbf{udfs}(v_0)$ entdeckt genau die Knoten in der Zusammenhangskomponente von v_0 .
- (b) Die Kanten (v, w) , für die $\mathbf{udfs}(w)$ unmittelbar aus $\mathbf{udfs}(v)$ 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 $\mathbf{udfs}(v)$ 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.

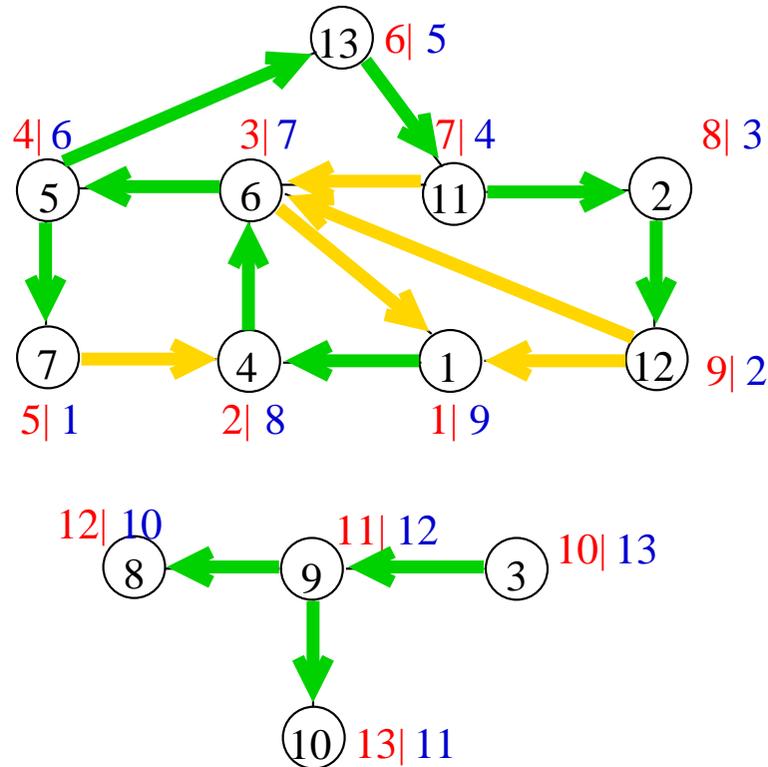
Beweis: Ähnlich zum gerichteten Fall.

Satz 8.3.2

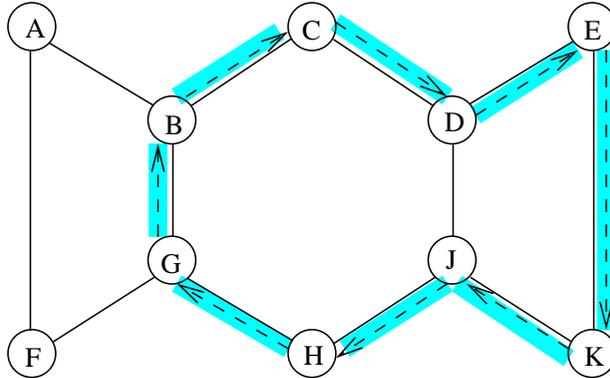
UDFS(G) werde aufgerufen. Dann gilt:

- (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 (T) oder Rückwärtskante (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, \dots, 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, \dots, v_{k-1} verschieden sind.

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. \square

Bemerkung

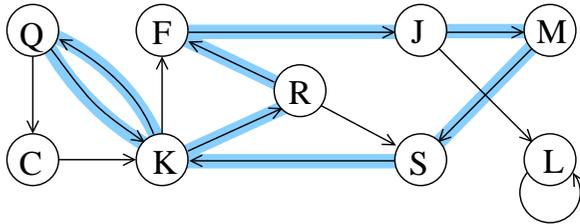
UDFS auf einen ungerichteten Wald (Vereinigung disjunkter Bäume) angewendet gibt jeder Komponente eine Wurzel und richtet die Kanten von dieser Wurzel weg.

Dies ist eine sehr einfache und schnelle Methode (Linearzeit!), einen ungerichteten Wald zu „orientieren“.

(Dies ist natürlich auch mit Breitensuche zu realisieren.)

8.4 Kreisfreiheitstest und topologische Sortierung

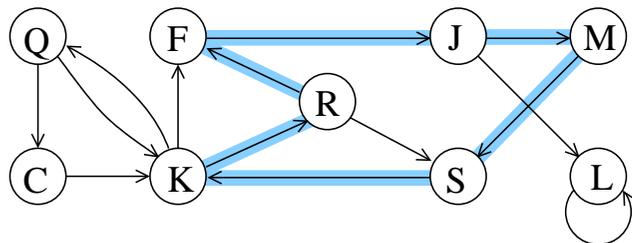
(Erinnerung an **Definition 7.1.4**)



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

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

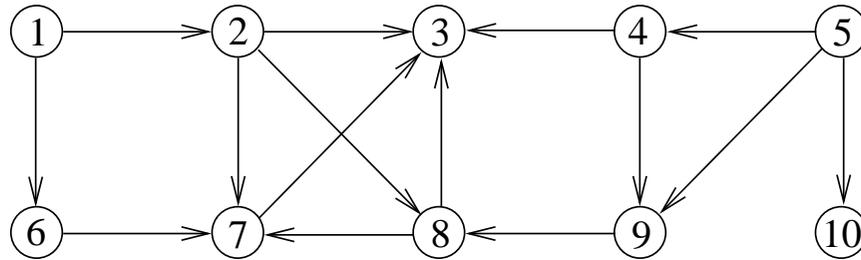
Kreise, die durch einen zyklischen Shift auseinander hervorgehen, werden als gleich betrachtet.



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

Beispiel: (J, M, S, K, R, F, J) ist ein einfacher Kreis.

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



(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 **DAGs**.

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

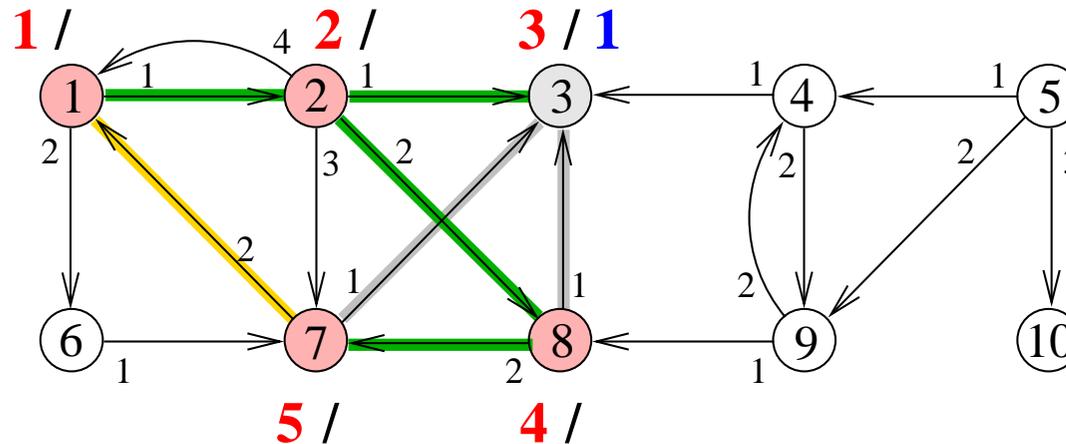
Satz 8.4.1

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

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: **(1, 2, 8, 7)**; Rückwärtskante **(7, 1)** schließt den Kreis.

„ \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 $f_num[v] > f_num[w]$.

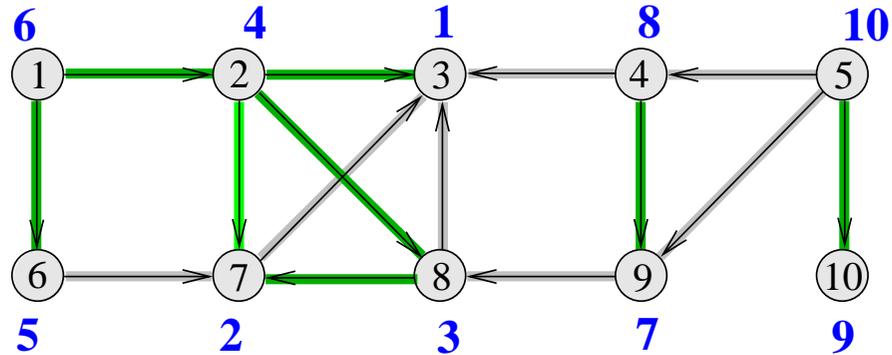
Für jede Kante (v, w) in E ist $f_num[v]$ größer als $f_num[w]$.

Wenn (u_0, \dots, 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:



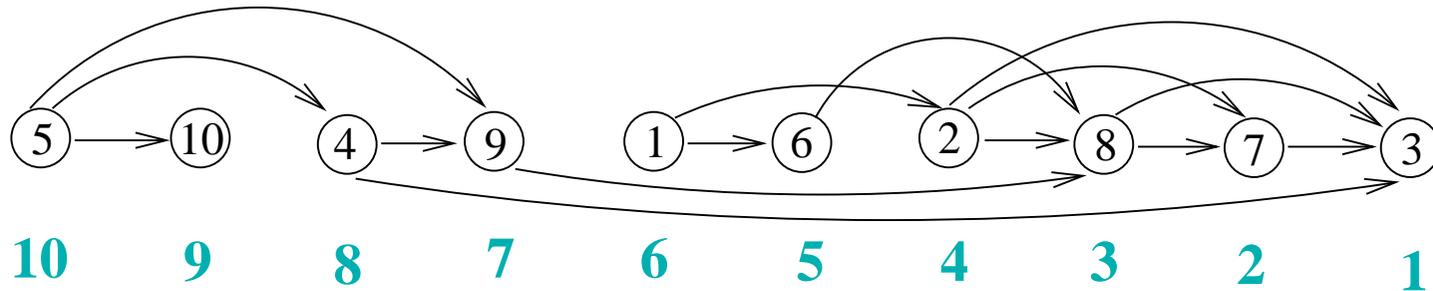
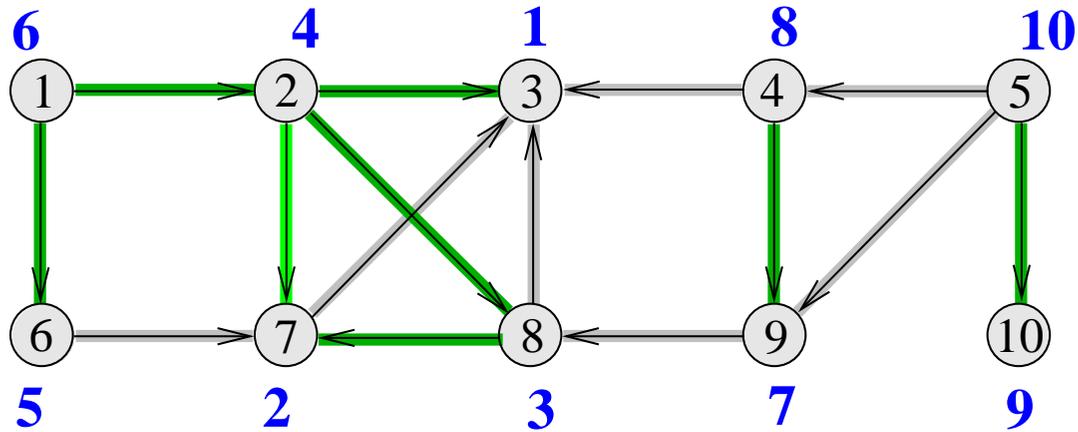
Wie im Beweis von Satz 8.4.1: f-Nummern fallen entlang jeder Kante!

Definition 8.4.2

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

Eine **topologische Sortierung** (oder topologische Anordnung) der Knoten in V ist eine Bijektion $\pi: V \rightarrow \{1, \dots, n\}$, 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“.



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!

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

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 . . .
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!

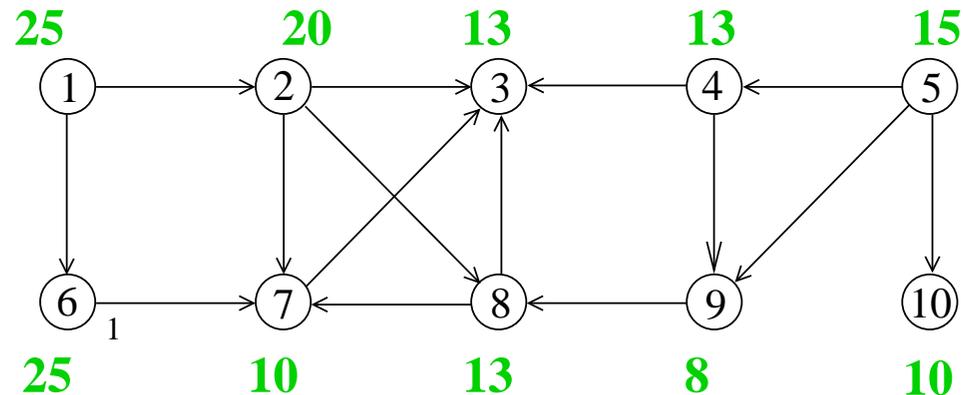
Anwendung 2:

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

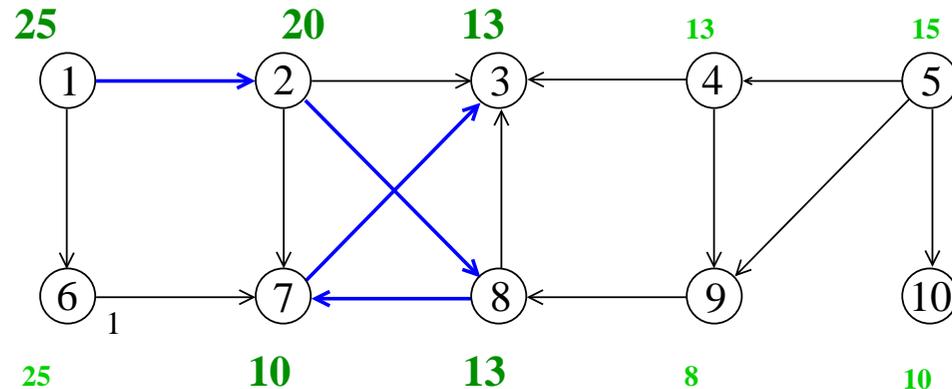
Frage: Wie lange dauert die Ausführung aller Tasks (mindestens) bei optimaler Planung?
Weitergehend: Finde einen optimalen Plan.



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

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

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

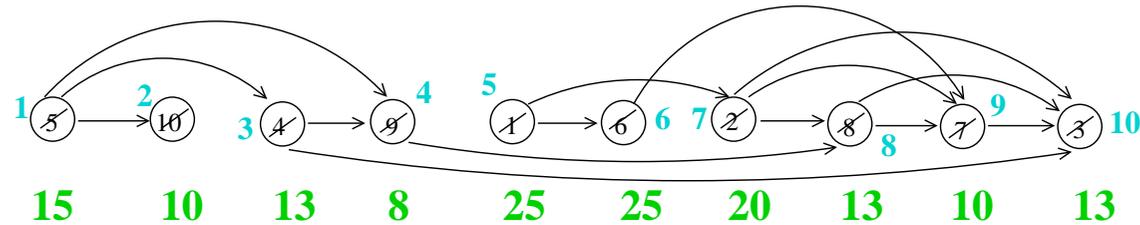


Ein kritischer Pfad mit Kosten $25 + 20 + 13 + 10 + 13 = 81$.

Aber wie berechnen?

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

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



Definiere, für $1 \leq v \leq n$:

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

$s(v)$:= erster Knoten nach v auf einem solchen Pfad bzw. undefiniert, wenn v in G keinen Nachfolger hat.

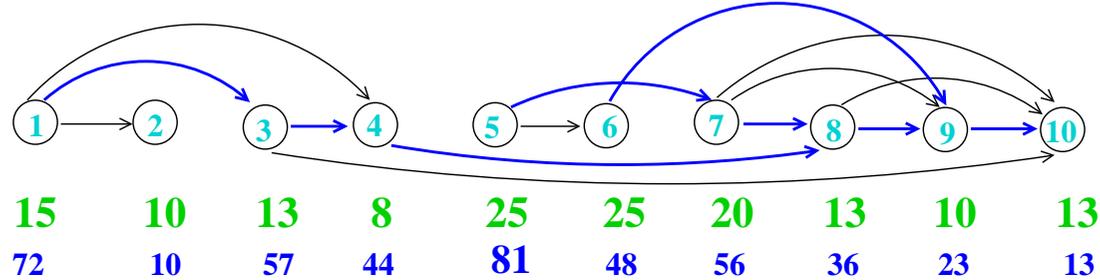
Berechne $\bar{c}(v)$ und $s(v)$ iterativ, für $v = n, \dots, 1$, wie folgt.

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:

$$\bar{c}(v) = \begin{cases} c(v) + \max\{\bar{c}(w) \mid (v, w) \in E\} & , \text{ falls eine Kante } (v, w) \text{ existiert } (!) v < w, \\ c(v) & , \text{ sonst.} \end{cases}$$

Ergebnis der Berechnung:



Ausgabe: Suche v^* mit maximalem $\bar{c}(v^*)$ (hier: $v^* = 5$ mit $\bar{c}(v^*) = 81$).

Kritischer Pfad: $(v^*, s(v^*), s(s(v^*)), \dots)$ (bis zum ersten Knoten ohne Nachfolger).

Hier: $(5, 7, 8, 9, 10)$.

Alternative: Berechnung in der Reihenfolge $1, \dots, n$. Nachteil: Benötigt Umkehrgraphen G^R .

Eine simple Erweiterung liefert sogar einen Ablaufplan, der optimale Zeit $\bar{c}(v^*)$ braucht:

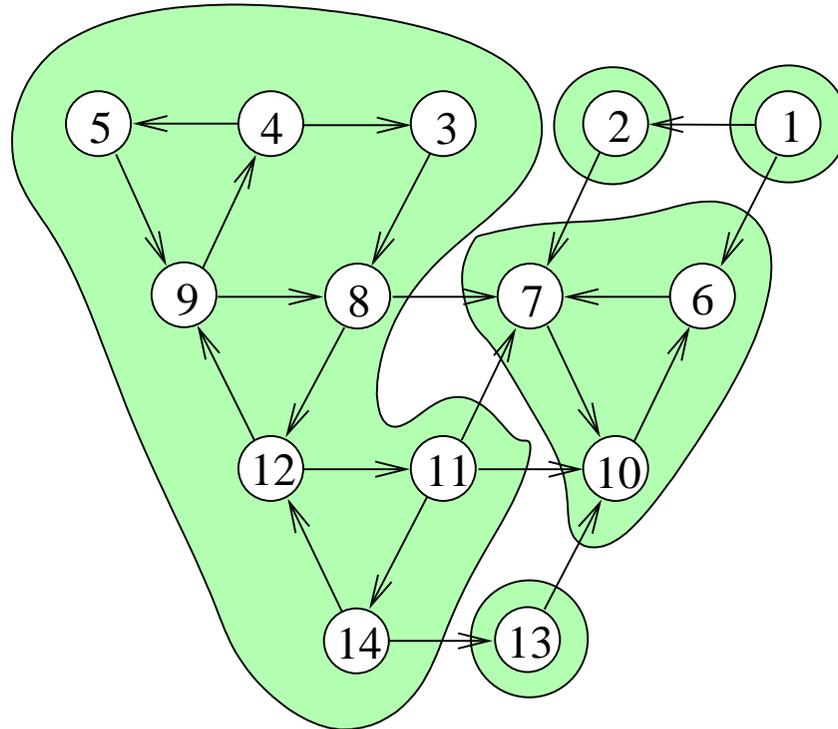
Starte Task v zum Zeitpunkt $t(v) := \bar{c}(v^*) - \bar{c}(v)$.

(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 .

8.5 Starke Zusammenhangskomponenten in Digraphen

Beispiel: 5 starke Zusammenhangskomponenten



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: $v \leftrightarrow w$.

Überlege:

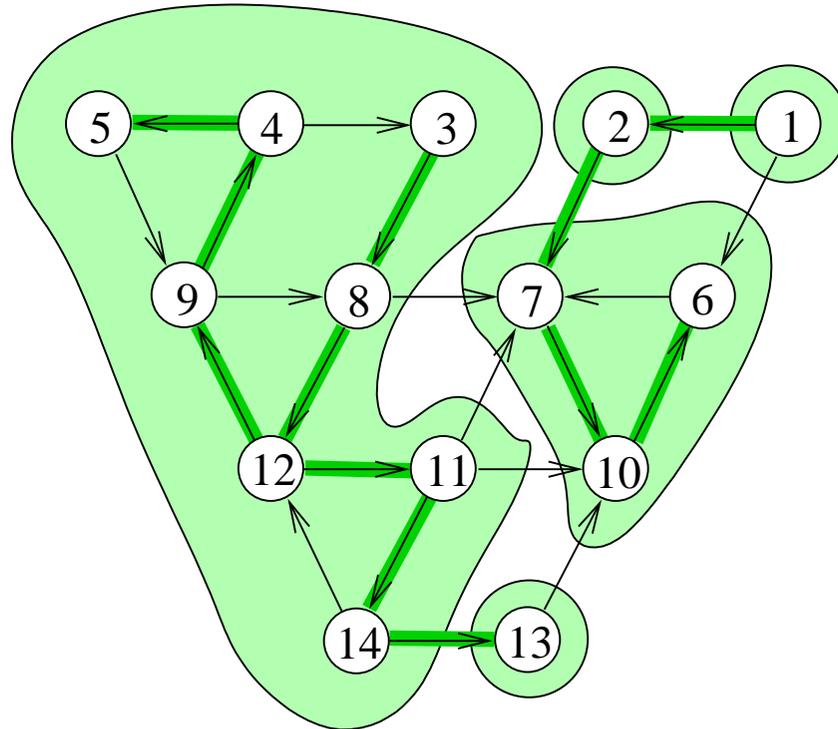
- (a) Die so definierte Relation ist reflexiv, transitiv und symmetrisch, also eine **Äquivalenzrelation**.
- (b) Zwei Knoten v und w sind äquivalent genau dann wenn sie identisch sind **oder** es in G einen (gerichteten) Kreis gibt, auf dem beide Knoten liegen.

Definition 8.5.2

Die Äquivalenzklassen zur Relation \leftrightarrow heißen **starke Zusammenhangskomponenten** von G .

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 $\text{DFS}(G)$ 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 $(v_C = v_0, v_1, \dots, v_t = v)$ 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 $v_i \leftrightarrow v_C$, also $v_i \in C$.)

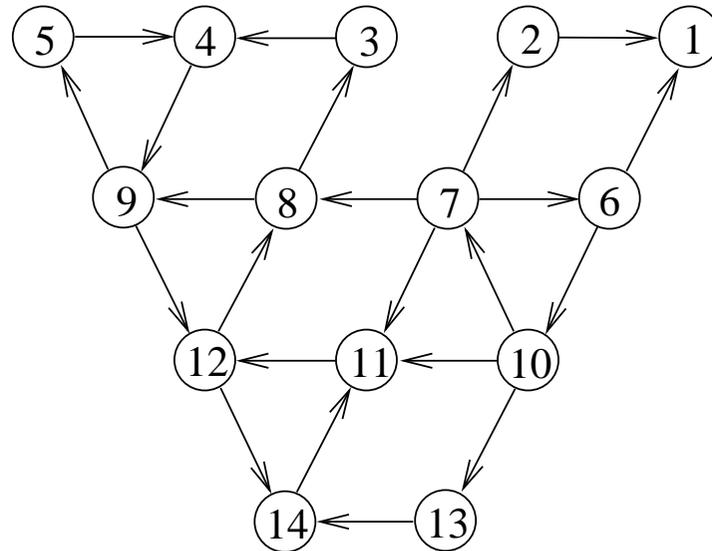
\Rightarrow (nach Beh. und Wahl von $v_0 = v_C$) Zu dem Zeitpunkt, zu dem $\text{dfs}(v_C)$ aufgerufen wird, sind die Knoten auf dem Weg $(v_0, v_1, \dots, v_t = v)$ alle noch neu

\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 .

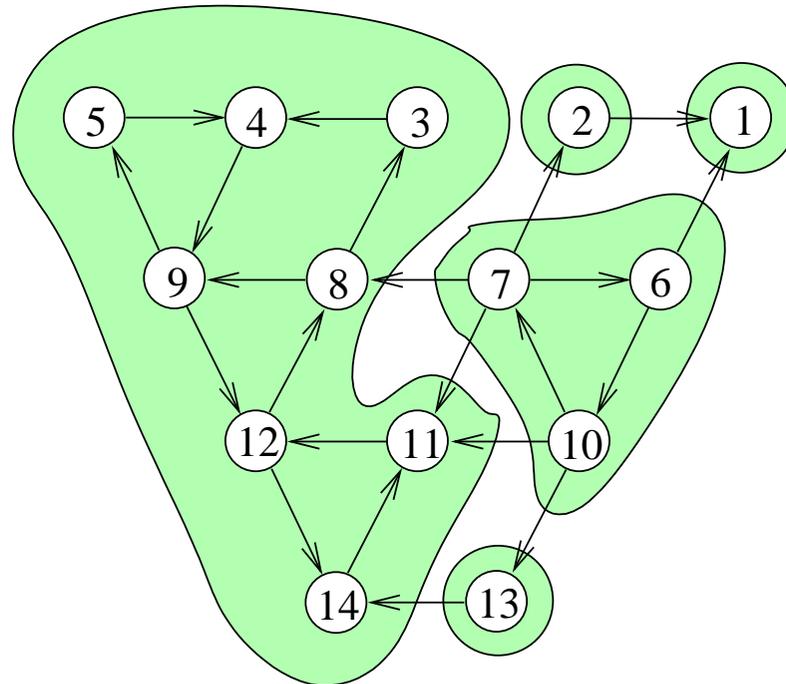
Um mit Tiefensuche die starken Zusammenhangskomponenten zu finden, muss man verhindern, dass in einem Baum mehrere solche Komponenten sitzen (wie im Bild auf Folie 58).

Trick 1: Führe DFS im „Umkehrgraphen“ G^R durch, der durch Umkehren aller Kanten in G entsteht.

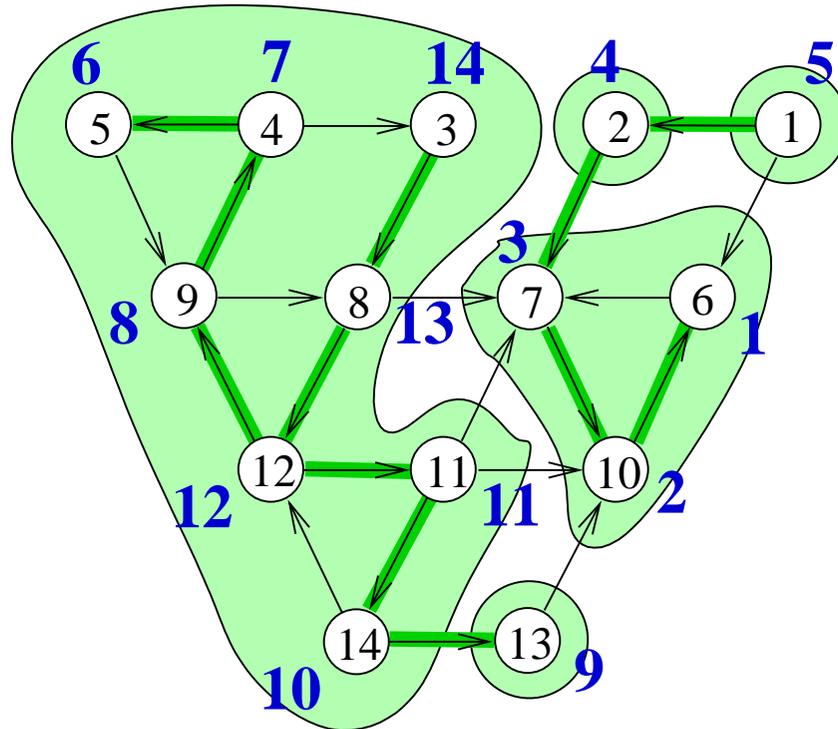


Umkehrgraph G^R von G .

Beobachtung: G^R hat dieselben starken Zusammenhangskomponenten wie G .
(Die Kreise in den beiden Graphen sind dieselben, nur mit umgekehrter Durchlaufreihenfolge.)

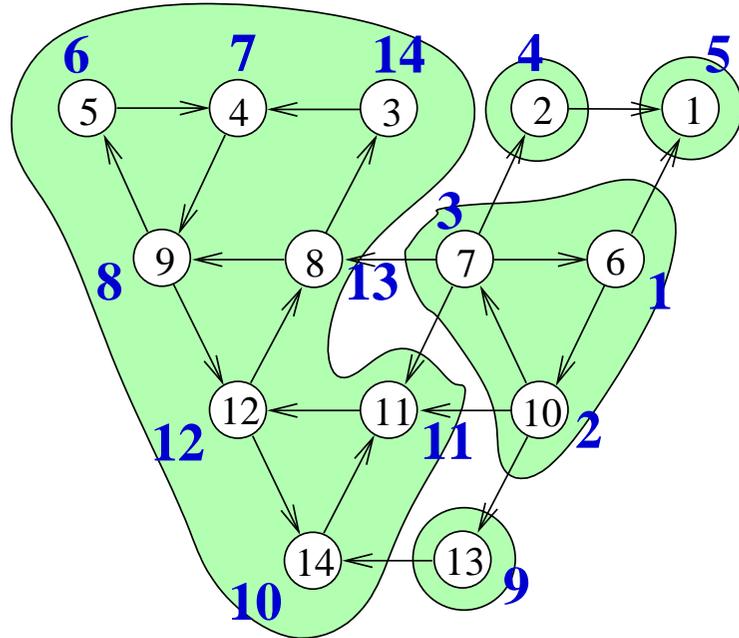


Trick 2: Verwende die **f-Nummern** einer ersten DFS in G , um zu bestimmen, in welcher Reihenfolge die Knoten in der Hauptschleife der zweiten DFS (in G^R) betrachtet werden.



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“.

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)

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 (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 .

$v_C \in C$ 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.

(D. h.: In $\text{DFS}(G^R)$ 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 $f\text{-num}(v) > f\text{-num}(v_C)$ und $v \rightsquigarrow_{G^R} v_C$

\Leftrightarrow es gibt kein v mit $f\text{-num}(v) > f\text{-num}(v_C)$ und $v_C \rightsquigarrow_G v$

\Leftrightarrow für alle v mit $v_C \rightsquigarrow_G v$ gilt $f\text{-num}(v) \leq f\text{-num}(v_C)$.

Es genügt also, Folgendes zu zeigen:

Lemma 8.5.5

Aus $v_C \rightsquigarrow_G v$ folgt $f\text{-num}(v) \leq f\text{-num}(v_C)$ (in DFS in Teil (1)!).

Lemma 8.5.5 Aus $v_C \rightsquigarrow_G v$ folgt $f\text{-num}(v) \leq f\text{-num}(v_C)$.

Beweis: O. B. d. A. gilt $v_C \neq v$.

Sei $v_C = v_0, v_1, \dots, v_{t-1}, v_t = v$ ein Weg in G .
 $\underbrace{\hspace{10em}}_{=: p}$

1. Fall: Wenn $\text{dfs}(v_C)$ 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 $f\text{-num}(v) < f\text{-num}(v_C)$.

2. Fall: Wenn $\text{dfs}(v_C)$ 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: $v_C \rightsquigarrow v_i$, also $v_i \in C$.

Andererseits ist $f\text{-num}(v_i) > f\text{-num}(v_C)$, im **Widerspruch** zur Wahl von v_C .

Der 2. Fall kann also gar nicht eintreten.

3. Fall: Wenn $\text{dfs}(v_C)$ startet, ist ein Knoten v_i , $0 < i \leq t$, **fertig**.

Wegen des Vorgehens der dfs-Prozedur und weil Fall 2 nicht eintritt, gilt für $i \leq j < t$, zu dem Zeitpunkt, an dem $\text{dfs}(v_C)$ startet:

Wenn v_j „fertig“ ist, dann ist auch v_{j+1} „fertig“.

(Bevor $\text{dfs}(v_j)$ endete, wurde Kante (v_j, v_{j+1}) betrachtet; also kann v_{j+1} nicht „neu“ sein.)

Also ist v „fertig“, wenn $\text{dfs}(v_C)$ startet.

Daraus folgt: $f\text{-num}(v) < f\text{-num}(v_C)$. □