Karlsruher Institut für Technologie

5. Zusatztutorium
Tiefensuche (DFS), Dynamische Programmierung (DP)

Algorithmen I, Zusatztutorium

Carina, Henriette, Jonas, Tobias | 29. August 2022

FAKULTÄT FÜR INFORMATIK

Roadmap

Roadmap

Tiefensuche Motivation

Wir wollen einen ungewichteten Graph erkunden.

Tiefensuche Motivation

Wir wollen einen ungewichteten Graph erkunden.

Wir haben Breitensuche (BFS) letzte Woche besprochen. Mit BFS durchsuchen wir vom Startknoten aus das nahe Umfeld und suchen danach immer weiter weg vom Startpunkt (Layer by Layer).

Tiefensuche Motivation

Wir wollen einen ungewichteten Graph erkunden.

Wir haben Breitensuche (BFS) letzte Woche besprochen. Mit BFS durchsuchen wir vom Startknoten aus das nahe Umfeld und suchen danach immer weiter weg vom Startpunkt (Layer by Layer).
Bei BFS müssen wir somit um einen Knoten v zu finden, alle Knoten mit kleinerer Distanz zu s vorher besuchen.

Tiefensuche Motivation

Wir wollen einen ungewichteten Graph erkunden.

Wir haben Breitensuche (BFS) letzte Woche besprochen. Mit BFS durchsuchen wir vom Startknoten aus das nahe Umfeld und suchen danach immer weiter weg vom Startpunkt (Layer by Layer).
Bei BFS müssen wir somit um einen Knoten v zu finden, alle Knoten mit kleinerer Distanz zu s vorher besuchen.

Was wollen wir?

Wir wollen ein Algorithmus, der schneller größere Teilbereiche des Graphens sieht.
Somit brauchen wir eine andere Besuchsreihenfolge der Knoten bei der es «egal», ist wie weit ein Knoten von Startknoten entfernt ist.

Tiefensuche (depth-first search)

Grundidee:

- Erkunde den Graphen «Strang für Strang»
  - Der Startknoten `s` bildet den Startpunkt
  - Die Nachbarn von `s` beginnen jeweils einen Strang
  - Der Algorithmus geht einen Strang entlang, kommt zurück, geht den nächsten Strang entlang
  - etc…
- Merke dir für jeden Knoten `v`, der erreicht wird:
  - seinen Vorgängerknoten `parent(v)`
  - seine Entfernung zum Startknoten `d(v)`
- Der Algorithmus hört auf, in die Tiefe zu gehen:
  - wenn der aktuelle Strang auf einen markierten Knoten trifft
  - wenn am aktuellen Knoten keine ausgehende Kante mehr existiert
- Dann geht der Algorithmus den Strang zurück und nimmt den nächsten Strang

Tiefensuche

Vereinfachter Algorithmus:

- Markiere den aktuellen Knoten v.
  - $\forall u \in N(v)$: Wenn u noch nicht markiert: Tiefensuche(u)
- Zusätzlich speichert der «richtige» Algorithmus noch Zusatzinformationen, dazu später mehr.

Tiefensuche

Vereinfachter Algorithmus:

- Markiere den aktuellen Knoten v.
  - $\forall u \in N(v)$: Wenn u noch nicht markiert: Tiefensuche(u)
- Zusätzlich speichert der «richtige» Algorithmus noch Zusatzinformationen, dazu später mehr.

In Pseudocode:

```
1: DFS(G = (V, E): Graph, v : Node)
2:     mark v
3:     for u ∈ N(v) do
4:         if not marked u then
5:             DFS(G, u)
```

Motivation iterative DFS

Rekursion hat verschiedene Vor- und Nachteile:

Vorteile

- simpel zu implementieren

Motivation iterative DFS

Rekursion hat verschiedene Vor- und Nachteile:

Vorteile

- simpel zu implementieren

Nachteile

- Rekursionstack hat auf realen PCs endliche Größe
- Funktionsaufrufe brauchen Zeit
- wir brauchen globale Variabelen (doof bei mehreren Funktionsaufrufen, vorallem wenn parallel)

Motivation iterative DFS

Rekursion hat verschiedene Vor- und Nachteile:

Vorteile

- simpel zu implementieren

Nachteile

- Rekursionstack hat auf realen PCs endliche Größe
- Funktionsaufrufe brauchen Zeit
- wir brauchen globale Variabelen (doof bei mehreren Funktionsaufrufen, vorallem wenn parallel)

Iterative DFS löst diese Probleme.
Beobachtudng: DFS ist relativ nah an BFS nur statts einer Queue brauchen wir einen Stack.

Iterative DFS

```
1: ITER-DFS(G = (V, E) : Graph, s : ℕ)
2:     par : [ℕ; n] = ⟨∞, ..., ∞⟩
3:     todo, done : Stack⟨ℕ⟩ = ⟨⟩
4:     todo.push(s); done.push(∞)
```

Iterative DFS

```
1: ITER-DFS(G = (V, E) : Graph, s : ℕ)
2:     par : [ℕ; n] = ⟨∞, ..., ∞⟩
3:     todo, done : Stack⟨ℕ⟩ = ⟨⟩
4:     todo.push(s); done.push(∞)
5:     while ¬todo.empty() do
6:         v : ℕ = todo.pop()
7:         if marked v then contiune
```

Iterative DFS

```
1: ITER-DFS(G = (V, E) : Graph, s : ℕ)
2:     par : [ℕ; n] = ⟨∞, ..., ∞⟩
3:     todo, done : Stack⟨ℕ⟩ = ⟨⟩
4:     todo.push(s); done.push(∞)
5:     while ¬todo.empty() do
6:         v : ℕ = todo.pop()
7:         if marked v then contiune
8:         mark v
9:         while par[v] ≠ done.top() do done.pop()
```

Iterative DFS

```
1:  ITER-DFS(G = (V, E) : Graph, s : ℕ)
2:      par : [ℕ; n] = ⟨∞, ..., ∞⟩
3:      todo, done : Stack⟨ℕ⟩ = ⟨⟩
4:      todo.push(s); done.push(∞)
5:      while ¬todo.empty() do
6:          v : ℕ = todo.pop()
7:          if marked v then contiune
8:          mark v
9:          while par[v] ≠ done.top() do done.pop()
10:         done.push(v)
11:         for u ∈ N(v) do
12:             if not marked u then
13:                 todo.push(u)
14:                 par[u] = v
```

Iterative DFS

```
1:  ITER-DFS(G = (V, E) : Graph, s : ℕ)
2:      par : [ℕ; n] = ⟨∞, ..., ∞⟩
3:      todo, done : Stack⟨ℕ⟩ = ⟨⟩
4:      todo.push(s); done.push(∞)
5:      while ¬todo.empty() do
6:          v : ℕ = todo.pop()
7:          if marked v then contiune
8:          mark v
9:          while par[v] ≠ done.top() do done.pop()
10:         done.push(v)
11:         for u ∈ N(v) do
12:             if not marked u then
13:                 todo.push(u)
14:                 par[u] = v
15:     while ¬done.empty() do done.pop()
```

Pingo

- Wie viele Kantentypen hat der DFS-Baum für gerichtete Graphen?

Pingo

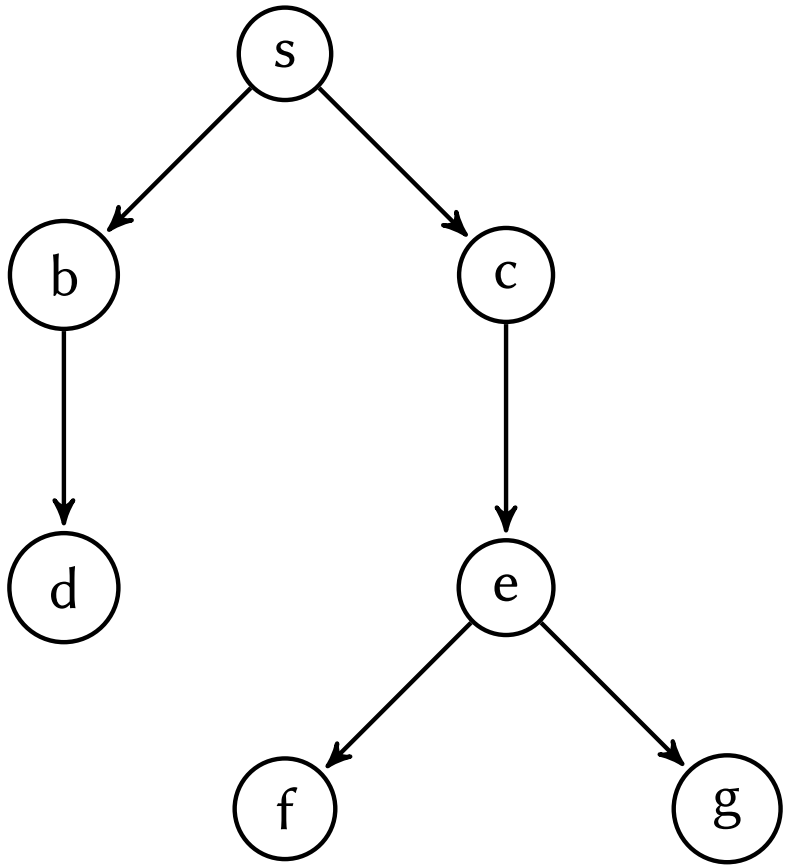
- Wie viele Kantentypen hat der DFS-Baum für gerichtete Graphen?
  4
- Wie viele Kantentypen hat der DFS-Baum für **ungerichtete** Graphen?

Pingo

- Wie viele Kantentypen hat der DFS-Baum für gerichtete Graphen?
  4
- Wie viele Kantentypen hat der DFS-Baum für **ungerichtete** Graphen?
  3

Kantentypen

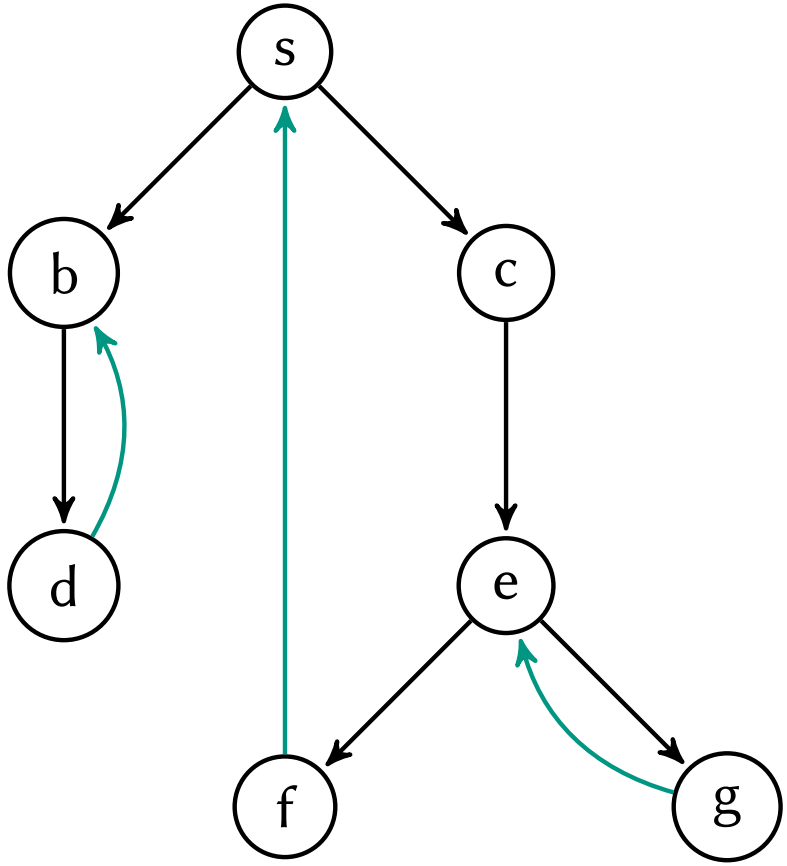
- Tree-Edge



Tiefensuchreihenfolge:
s, b, d, c, e, f, g

Kantentypen

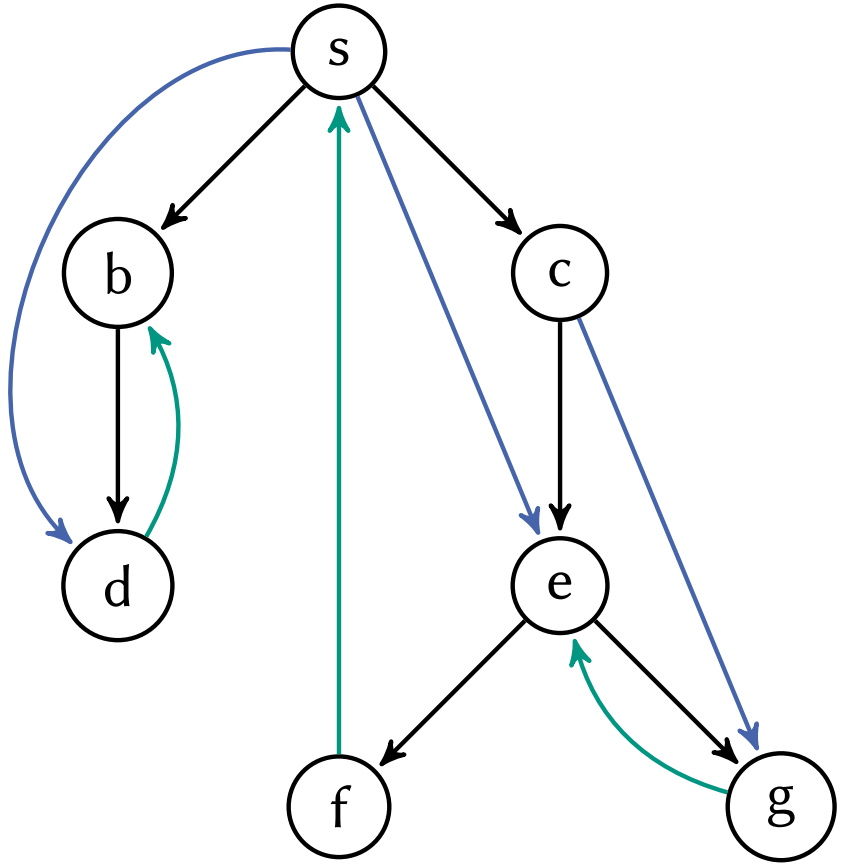
- Tree-Edge
- Backward-Edges



Tiefensuchreihenfolge:
s, b, d, c, e, f, g

Kantentypen

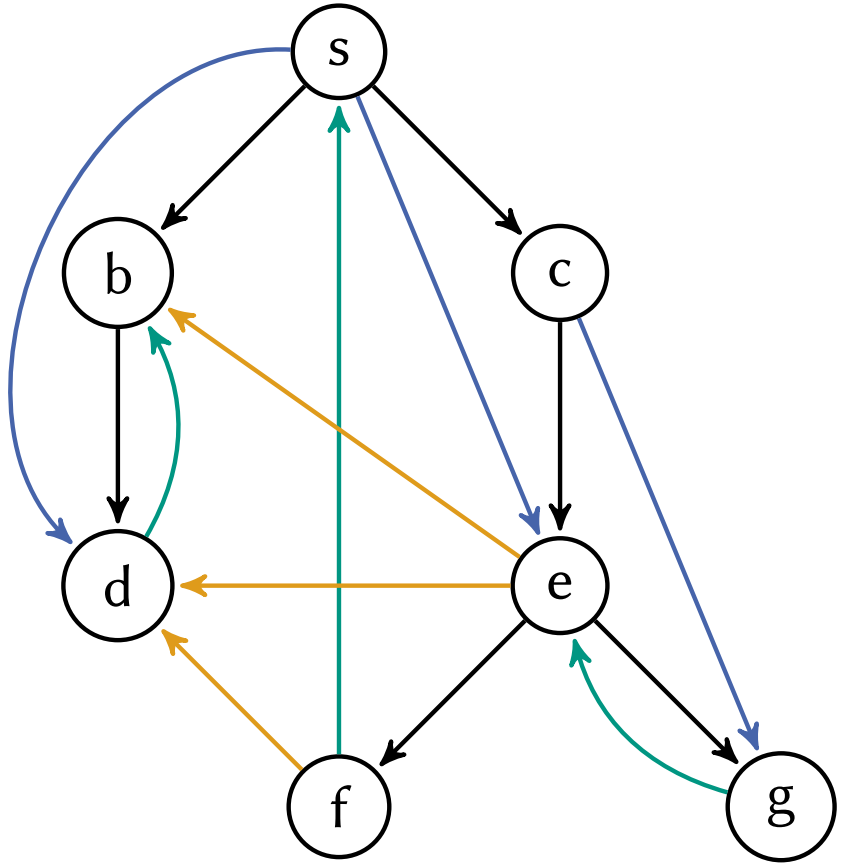
- Tree-Edge
- Backward-Edges
- Forward-Edges



Tiefensuchreihenfolge:
s, b, d, c, e, f, g

Kantentypen

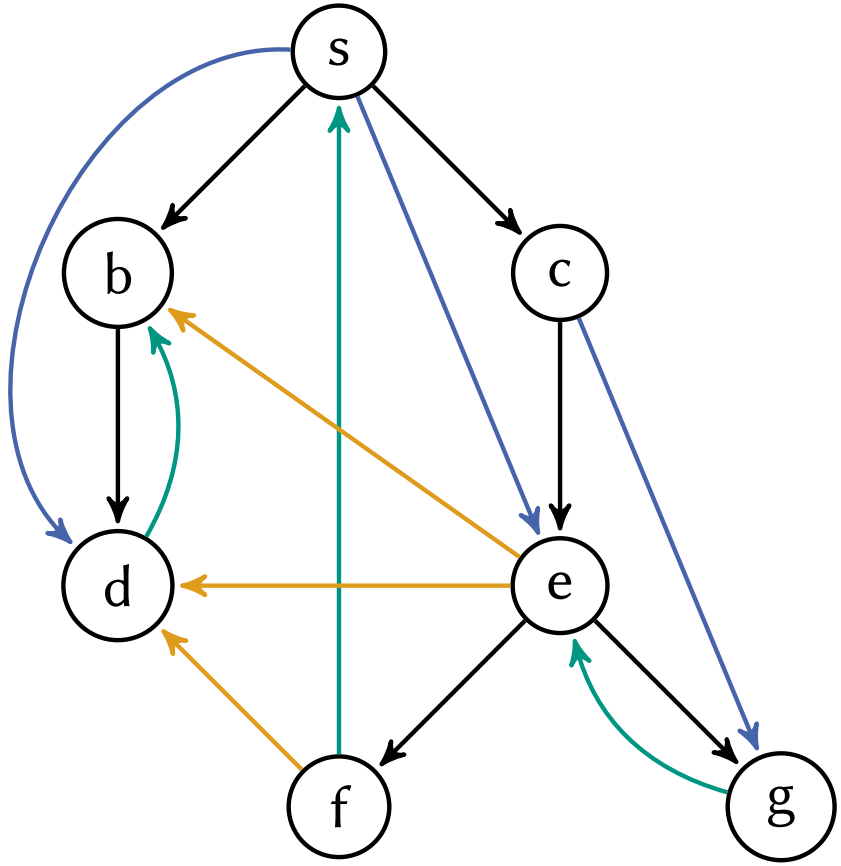
- Tree-Edge
- Backward-Edges
- Forward-Edges
- Cross-Edges



Tiefensuchreihenfolge:
s, b, d, c, e, f, g

Kantentypen

- Tree-Edge
- Backward-Edges
- Forward-Edges
- Cross-Edges



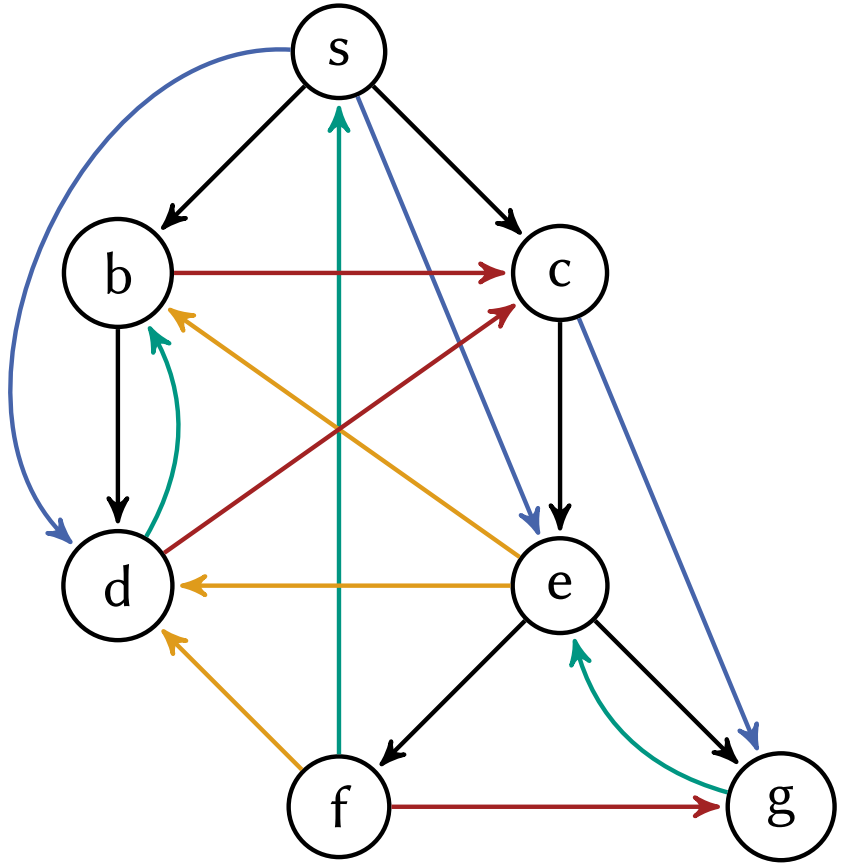
Tiefensuchreihenfolge:
s, b, d, c, e, f, g

Cross-Kanten

Wir können nur Cross-Kante vom neuen «Strang» zu bereits besuchten «Strängen» haben.

Kantentypen

- Tree-Edge
- Backward-Edges
- Forward-Edges
- Cross-Edges
- Illegal-Edges



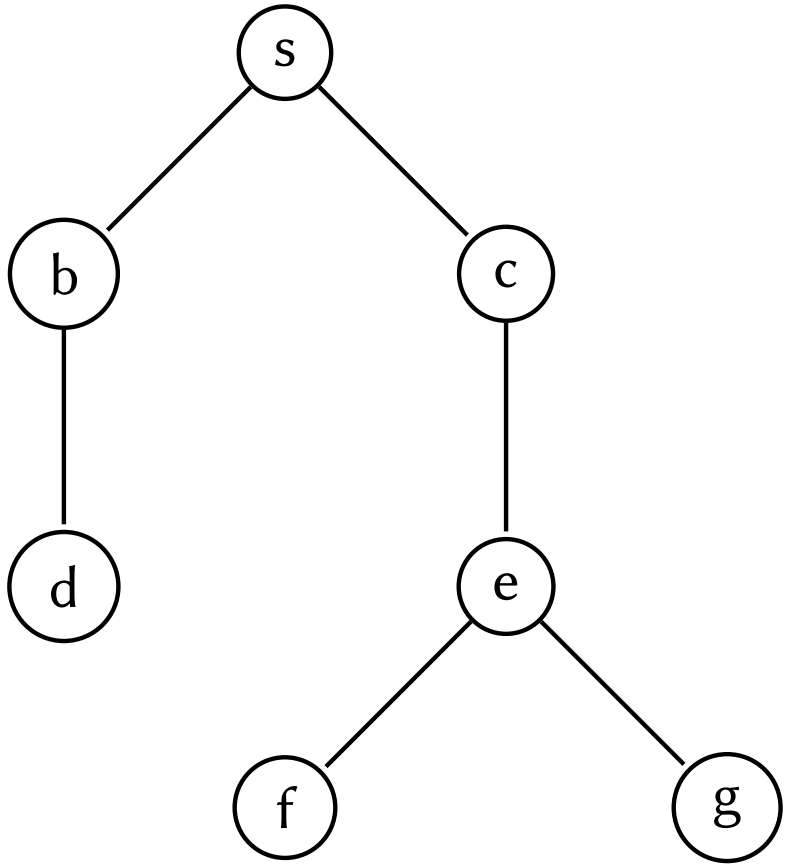
Tiefensuchreihenfolge:
s, b, d, c, e, f, g

Cross-Kanten

Wir können nur Cross-Kante vom neuen «Strang» zu bereits besuchten «Strängen» haben.

Kantentypen ungerichtete Graphen

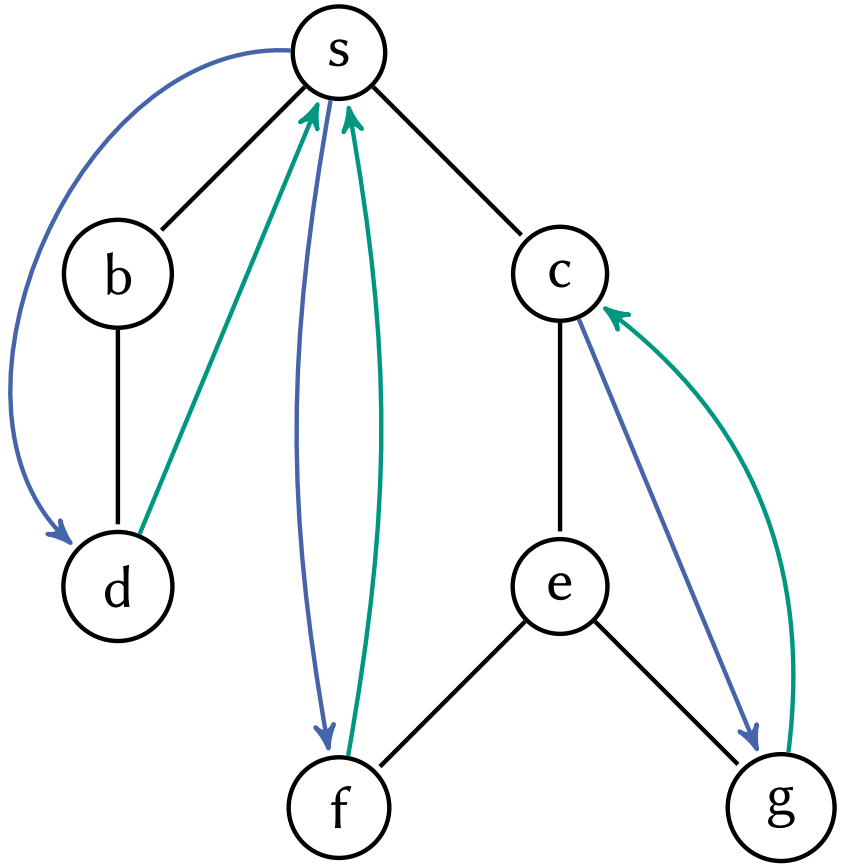
- Tree-Edges



Tiefensuchreihenfolge:
s, b, d, c, e, f, g

Kantentypen ungerichtete Graphen

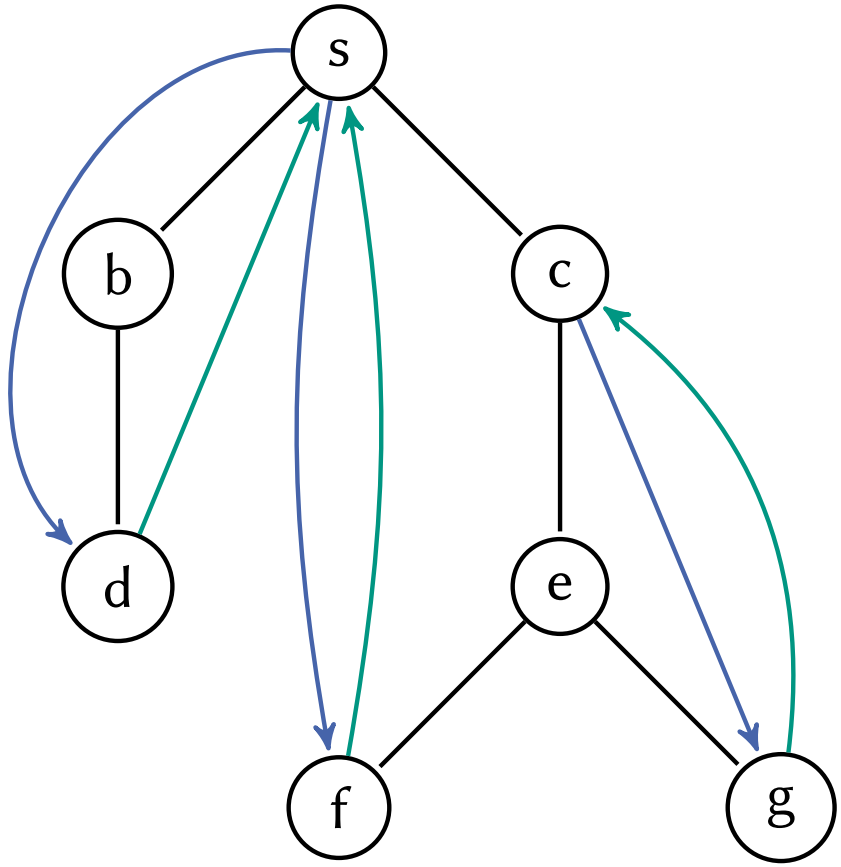
- Tree-Edges
- Backward-Edges
- Forward-Edges



Tiefensuchreihenfolge:
s, b, d, c, e, f, g

Kantentypen ungerichtete Graphen

- Tree-Edges
- Backward-Edges
- Forward-Edges



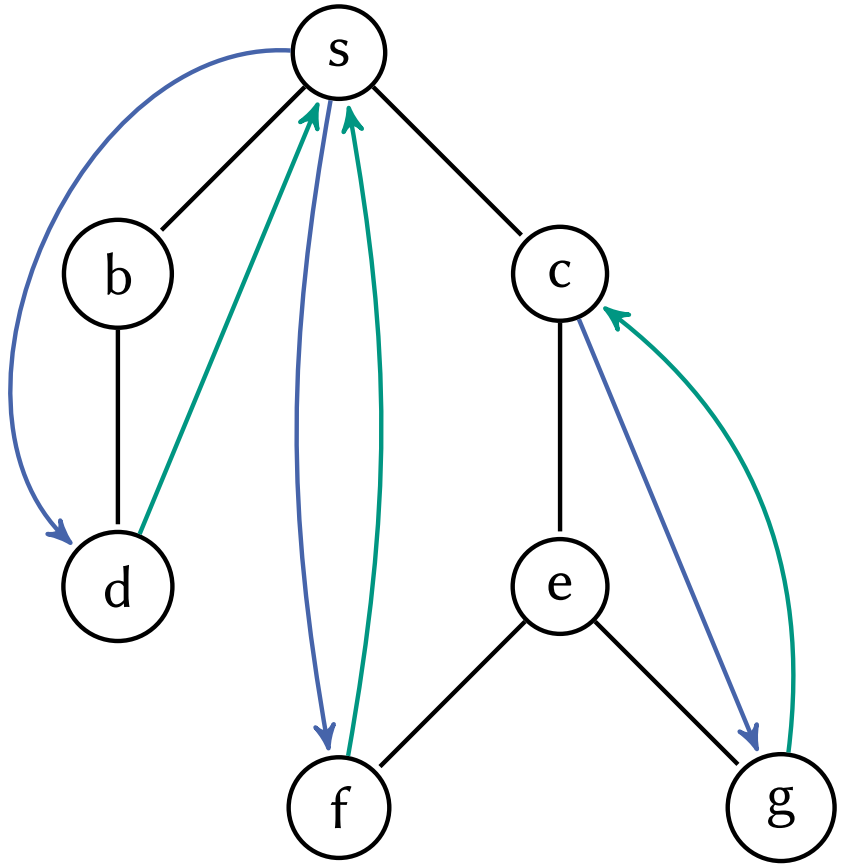
Tiefensuchreihenfolge:
s, b, d, c, e, f, g

Forward/Backward-Edge

Immer wenn eine Backward-Edge gefunden wird wird die Kante nochmal als Forward-Edge gefunden.

Kantentypen ungerichtete Graphen

- Tree-Edges
- Backward-Edges
- Forward-Edges
- Cross-Edges



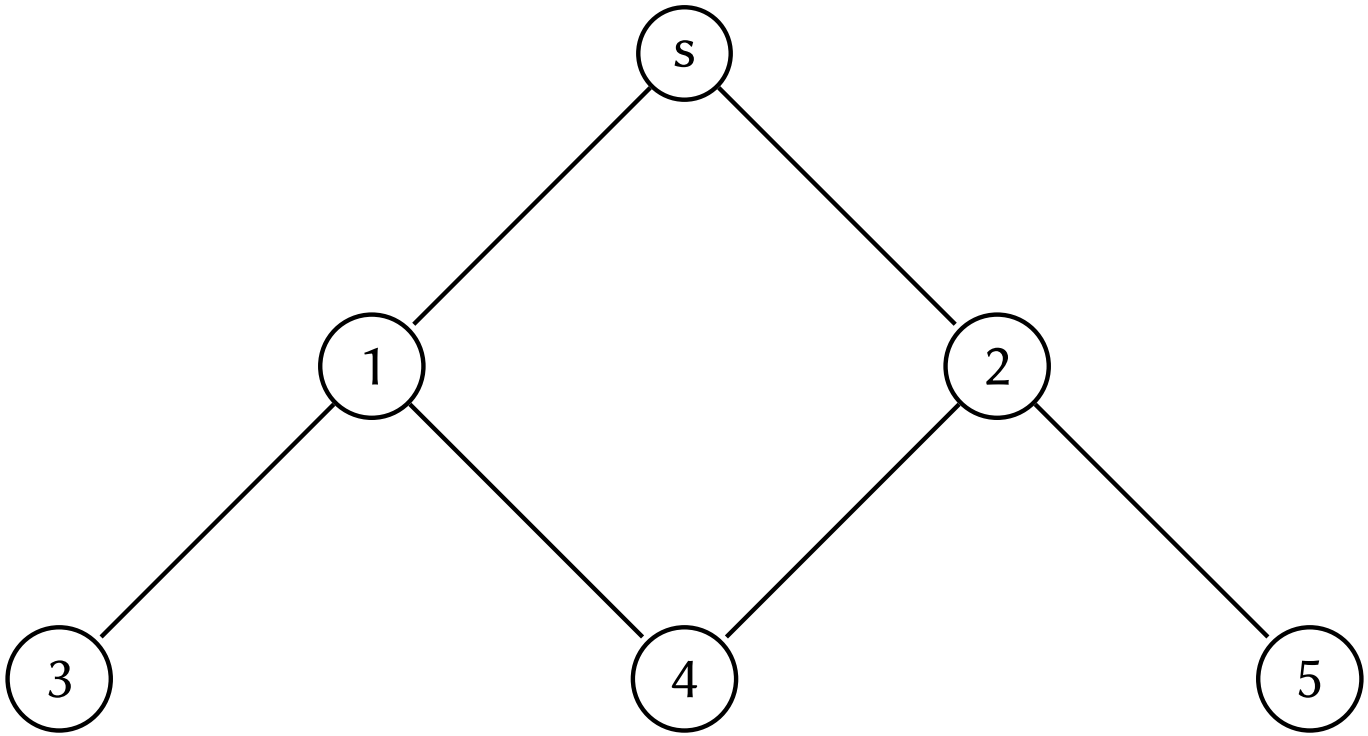
Tiefensuchreihenfolge:
s, b, d, c, e, f, g

Cross-Edge

Bei ungerichteten Graphen können keine Cross-Kanten auftreten.

Pingo

Wie viele verschiedene DFS-Bäume hat der Graph unterhalb?



Knoteninformation

Während der Tiefensuche berechnen wir Information über den Knoten, anhand dessen wir Probleme lösen können:

- Parent
- Distanz zu Startknoten im DFS-Baum

Knoteninformation

Während der Tiefensuche berechnen wir Information über den Knoten, anhand dessen wir Probleme lösen können:

- Parent
- Distanz zu Startknoten im DFS-Baum
- DFS-Number
- FIN-Number
- LOW-Number

Knoteninformation

Während der Tiefensuche berechnen wir Information über den Knoten, anhand dessen wir Probleme lösen können:

- Parent
- Distanz zu Startknoten im DFS-Baum
- DFS-Number
- FIN-Number
- LOW-Number

In den folgenden Folien gehen wir **immer** davon aus, dass alle Werte vor dem Start der Tiefensuche mit Null initalisiert sind (auch globale Zähler).

Parent/Distanz

Parent (`par`) und Distanz (`dist`) kennen wir bereits von anderen Algorithmen. Hier ist die Distanz aber im **DFS-Baum** (tree-edges), somit **nicht** die Distanz die BFS berechnen würde.

Dementsprechend speichert DFS für Knoten u Knoten v als Parent, wenn wir von Knoten v aus u besuchen. Somit laufen wir Kante (v, u) «entlang».

Distanze von u ist somit Distanz von v plus Eins.

Parent/Distanz

Parent (`par`) und Distanz (`dist`) kennen wir bereits von anderen Algorithmen. Hier ist die Distanz aber im **DFS-Baum** (tree-edges), somit **nicht** die Distanz die BFS berechnen würde.

Dementsprechend speichert DFS für Knoten u Knoten v als Parent, wenn wir von Knoten v aus u besuchen. Somit laufen wir Kante (v, u) «entlang».

Distanze von u ist somit Distanz von v plus Eins.

```
1: DFS(G = (V, E): Graph, v : Node)
2:     mark v
3:     for u ∈ N(v) do
4:         if not marked u then
5:             dist[u] := dist[v] + 1
6:             par[u] := v
7:             DFS(G, u)
```

DFS-Number

Die DFS-Number `dfsNum` beschreibt den Zeitpunkt, zu dem der Knoten das erste Mal besucht wird. Hierzu verwenden wir einen globalen Zähler. Unsere Zeitpunkte sind die Natürliche Zahlen startend bei Null.

Alle Nachfolger vom aktuellen Knoten haben eine größere DFS-Number, wenn sie noch nicht besucht wurden.

DFS-Number

Die DFS-Number `dfsNum` beschreibt den Zeitpunkt, zu dem der Knoten das erste Mal besucht wird. Hierzu verwenden wir einen globalen Zähler. Unsere Zeitpunkte sind die Natürliche Zahlen startend bei Null.

Alle Nachfolger vom aktuellen Knoten haben eine größere DFS-Number, wenn sie noch nicht besucht wurden.

```
1: DFS(G = (V, E): Graph, v : Node)
2:     mark v
3:     dfsNum[v] := dfsCounter
4:     dfsCounter := dfsCounter + 1
5:     for u ∈ N(v) do
6:         if not marked u then
7:             DFS(G, u)
```

FIN-Number

Die FIN-Number `finNum` beschreibt den Zeitpunkt, an dem der Knoten das letzte Mal besucht wird. Zu diesem Zeitpunkt wurden alle Nachfolger, Nachfolger von Nachfolger, ... abgeschlossen.

FIN-Number

Die FIN-Number `finNum` beschreibt den Zeitpunkt, an dem der Knoten das letzte Mal besucht wird. Zu diesem Zeitpunkt wurden alle Nachfolger, Nachfolger von Nachfolger, ... abgeschlossen.

Alle Nachfolger vom aktuellen Knoten mit einer Höheren DFS-Number sind zum Zeitpunkt des Setzens der FIN-Number des aktuellen Knoten bereits gesetzt. Es gilt somit für Knoten $v \in V$ $u \in V$:
$dfsNum[v] < dfsNum[u] \wedge finNum[v] > finNum[u] \Rightarrow u$ ist Kind von v.

FIN-Number

Die FIN-Number `finNum` beschreibt den Zeitpunkt, an dem der Knoten das letzte Mal besucht wird. Zu diesem Zeitpunkt wurden alle Nachfolger, Nachfolger von Nachfolger, ... abgeschlossen.

Alle Nachfolger vom aktuellen Knoten mit einer Höheren DFS-Number sind zum Zeitpunkt des Setzens der FIN-Number des aktuellen Knoten bereits gesetzt. Es gilt somit für Knoten $v \in V$ $u \in V$:
$dfsNum[v] < dfsNum[u] \wedge finNum[v] > finNum[u] \Rightarrow u$ ist Kind von v.

```
1: DFS(G = (V, E): Graph, v : Node)
2:     mark v
3:     for u ∈ N(v) do
4:         if not marked u then
5:             DFS(G, u)
6:     finNum[v] := finCounter
7:     finCounter := finCounter + 1
```

LOW-Number

Die LOW-Number `lowNum` gibt die niedrigste DFS-Number an, die von einem Kind vom aktuellen Knoten erreicht werden kann.

LOW-Number

Die LOW-Number `lowNum` gibt die niedrigste DFS-Number an, die von einem Kind vom aktuellen Knoten erreicht werden kann.

```
1: DFS(G = (V, E): Graph, v : Node)
2:     mark v
3:     lowNum[v] := dfsNum[v]
4:     for u ∈ N(v) do
5:         if not marked u then                          // tree-edge
6:             DFS(G, u)
7:             lowNum[v] := min(lowNum[v], lowNum[u])
8:         else                                          // non-tree-edge
9:             lowNum[v] := min(lowNum[v], dfsNum[u])
```

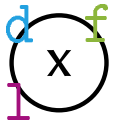
DFS mit allen Berechnungen

```
1:  DFS(G := (V, E): Graph, v : Node)
2:      mark v
3:      dfsNum[v] := dfsCounter
4:      dfsCounter := dfsCounter + 1
5:      lowNum[v] := dfsNum[v]
6:      for u ∈ N(v) do
7:          if not marked u then                          // tree-edge
8:              dist[u] := dist[v] + 1
9:              par[v] := v
10:             DFS(G, u)
11:             lowNum[v] := min(lowNum[v], lowNum[u])
12:         else                                          // non-tree-edge
13:             lowNum[v] := min(lowNum[v], dfsNum[u])
14:     finNum[v] := finCounter
15:     finCounter := finCounter + 1
```

DFS Beispiel

skip animation

Knoten:



dfsNum

finNum

lowNum

Kanten:

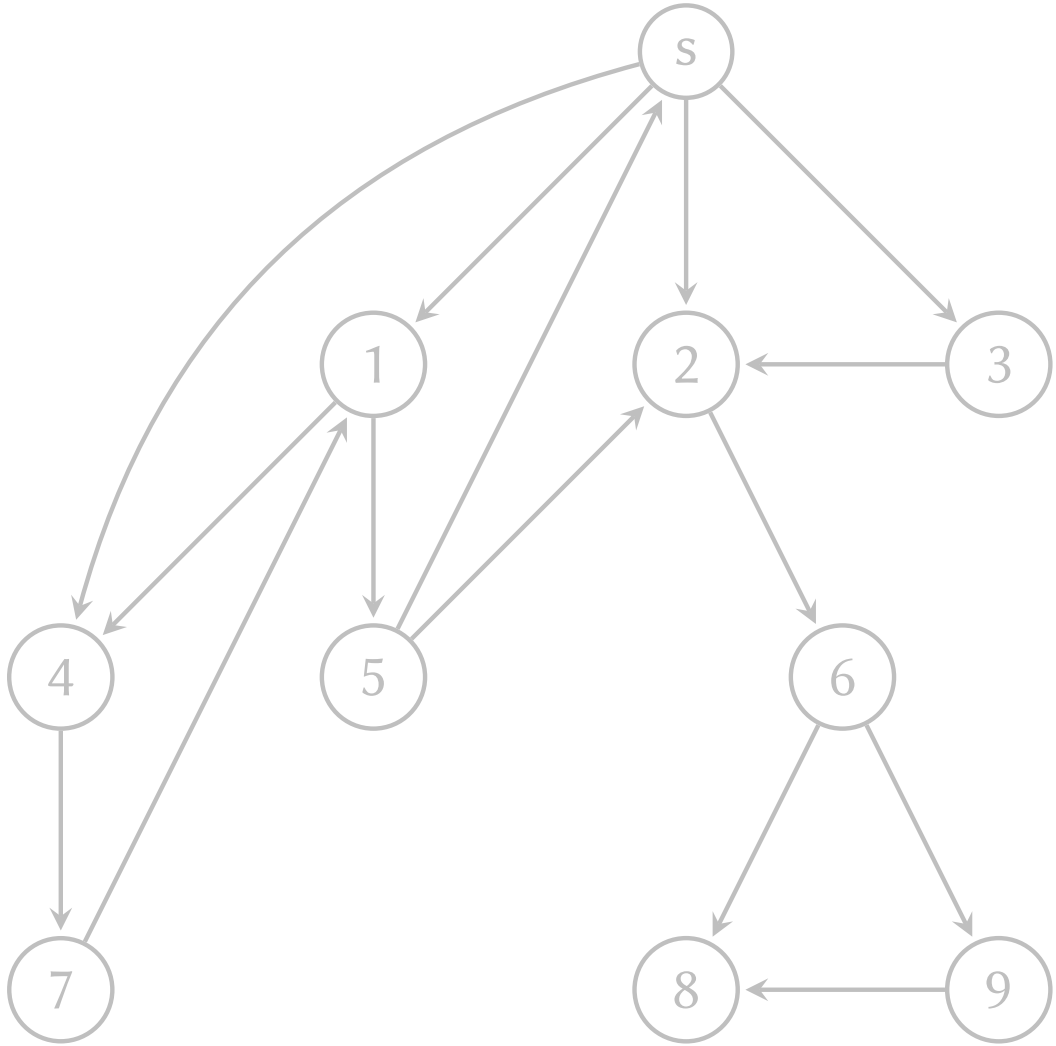
← unknown

← tree

← backward

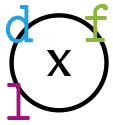
← forward

← cross



DFS Beispiel

Knoten:



dfsNum

finNum

lowNum

Kanten:

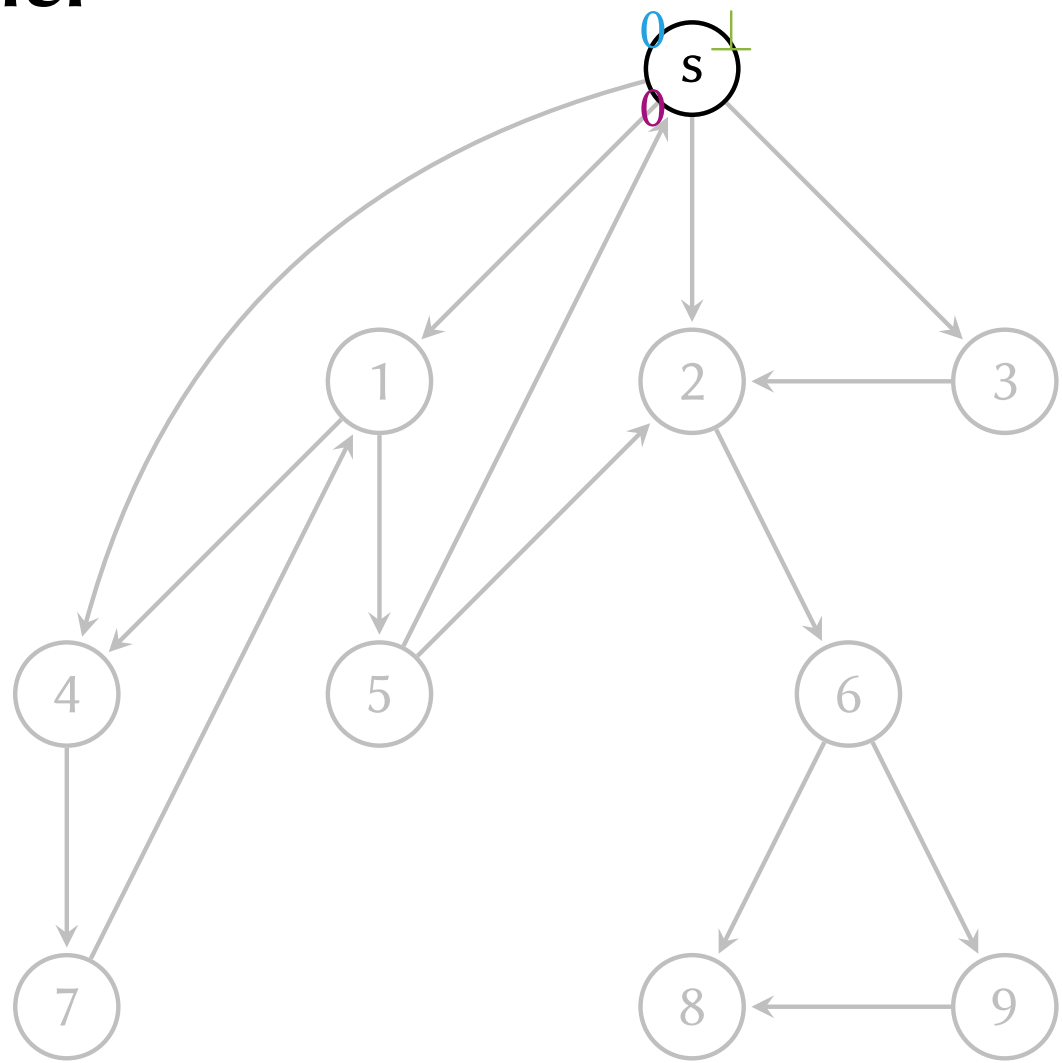
unknown

tree

backward

forward

cross

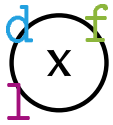


Starte bei Startknoten s

skip animation

DFS Beispiel

Knoten:



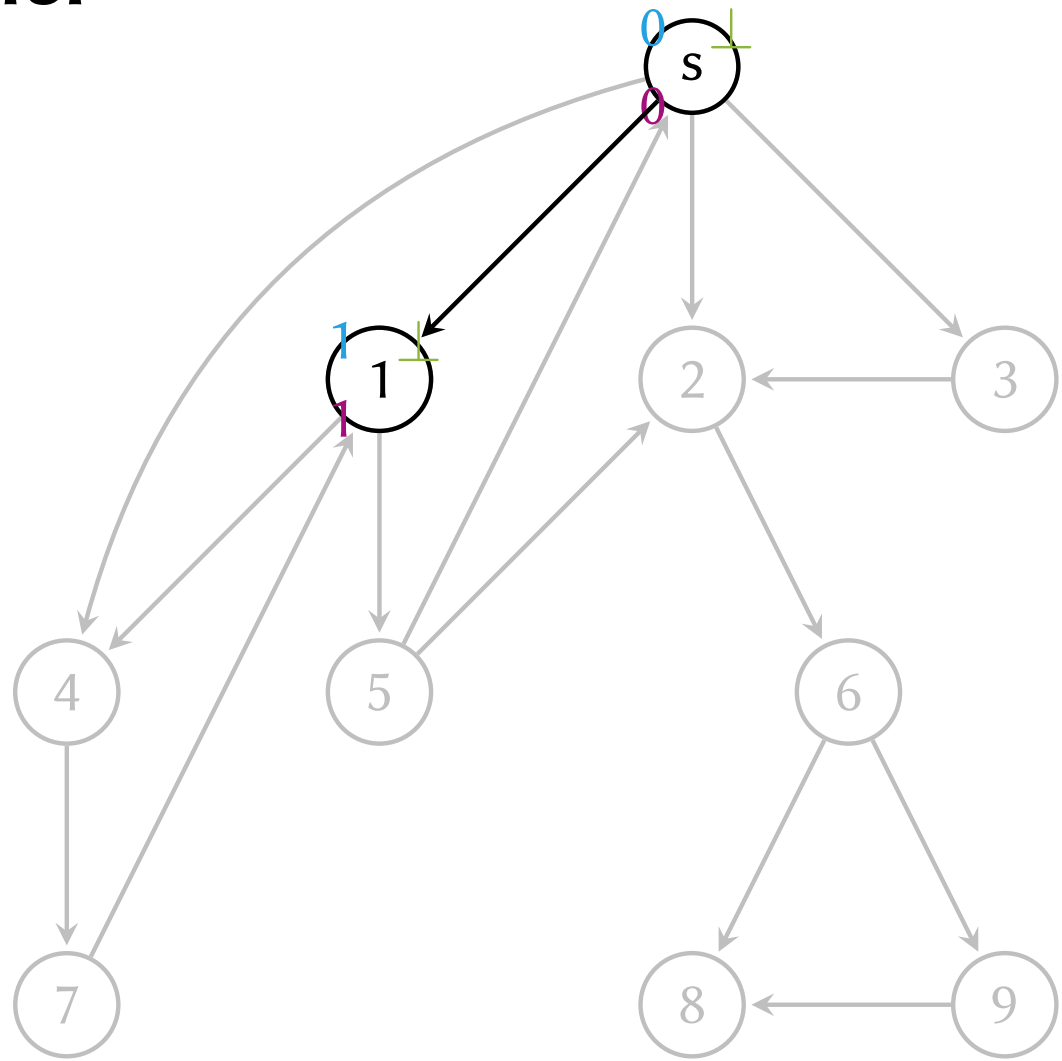
dfsNum

finNum

lowNum

Kanten:

- unknown
- tree
- backward
- forward
- cross



skip animation

verfolge Baum-Kante $(s, 1)$

DFS Beispiel

Knoten:

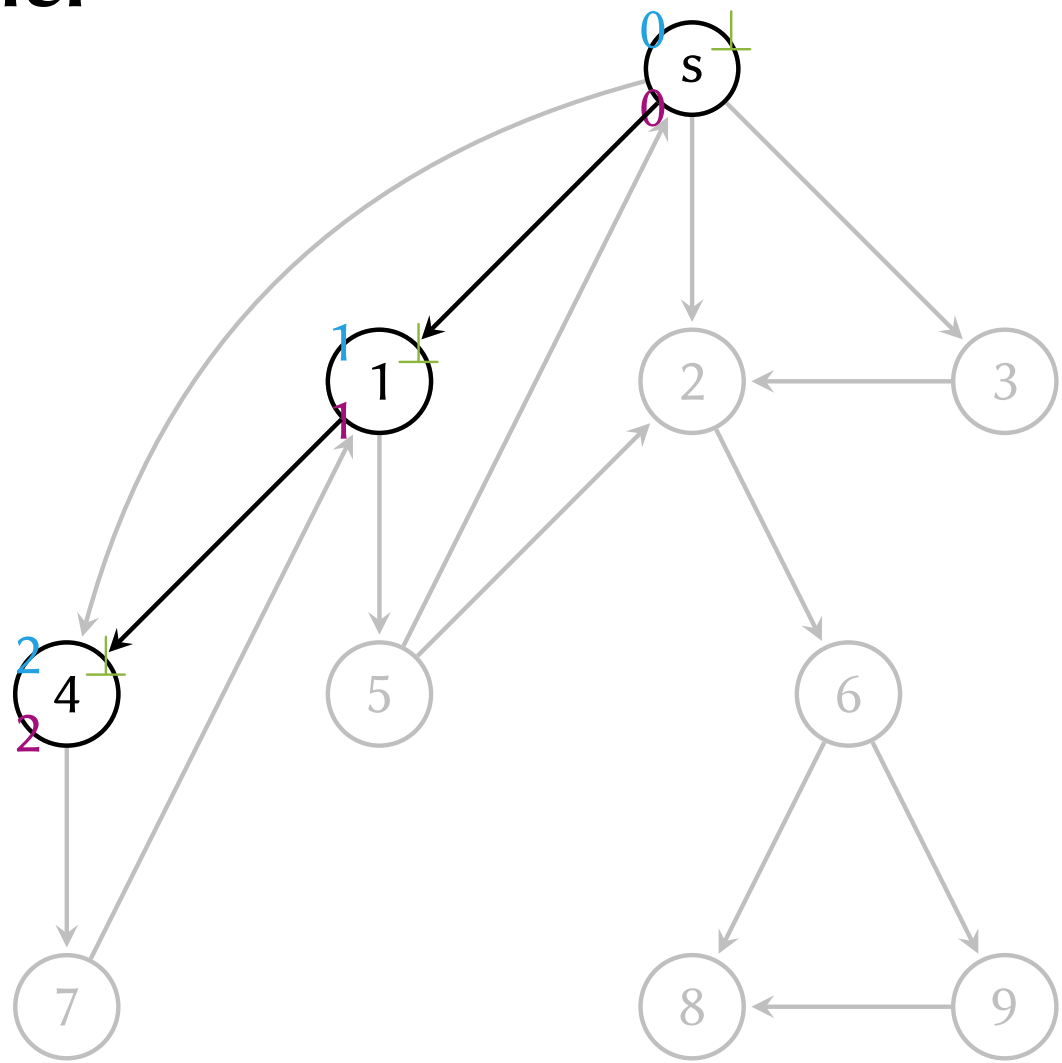
dfsNum

finNum

lowNum

Kanten:

- unknown
- tree
- backward
- forward
- cross



skip animation

verfolge Baum-Kante $(1, 4)$

DFS Beispiel

skip animation

Knoten:

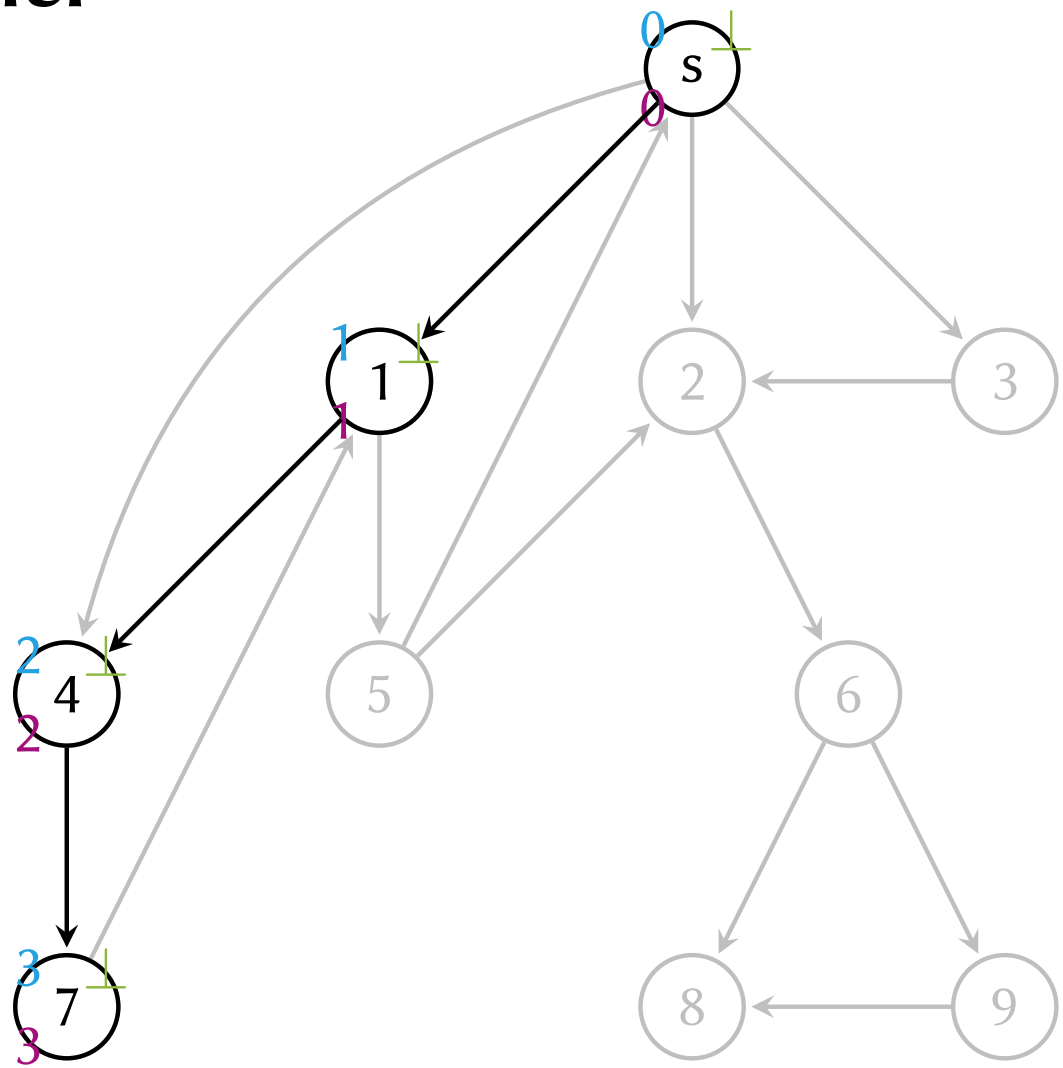
dfsNum

finNum

lowNum

Kanten:

- unknown
- tree
- backward
- forward
- cross

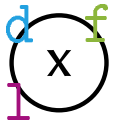


verfolge Baum-Kante (4, 7)

DFS Beispiel

skip animation

Knoten:



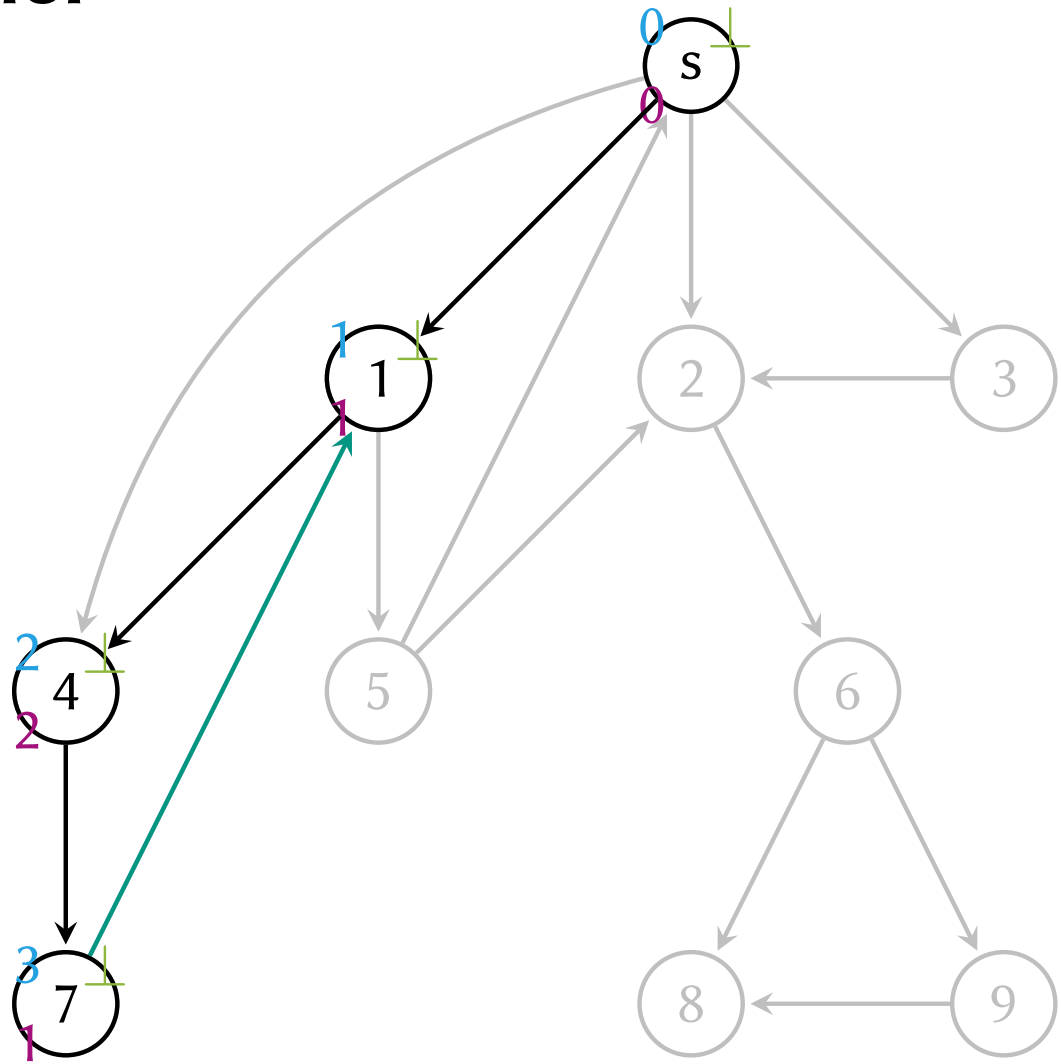
`dfsNum`

`finNum`

`lowNum`

Kanten:

- unknown
- tree
- backward
- forward
- cross

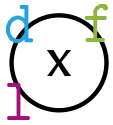


entdecke Rückwärts-Kante (7, 1), aktualisiere `lowNum`[7] zu 1

DFS Beispiel

skip animation

Knoten:



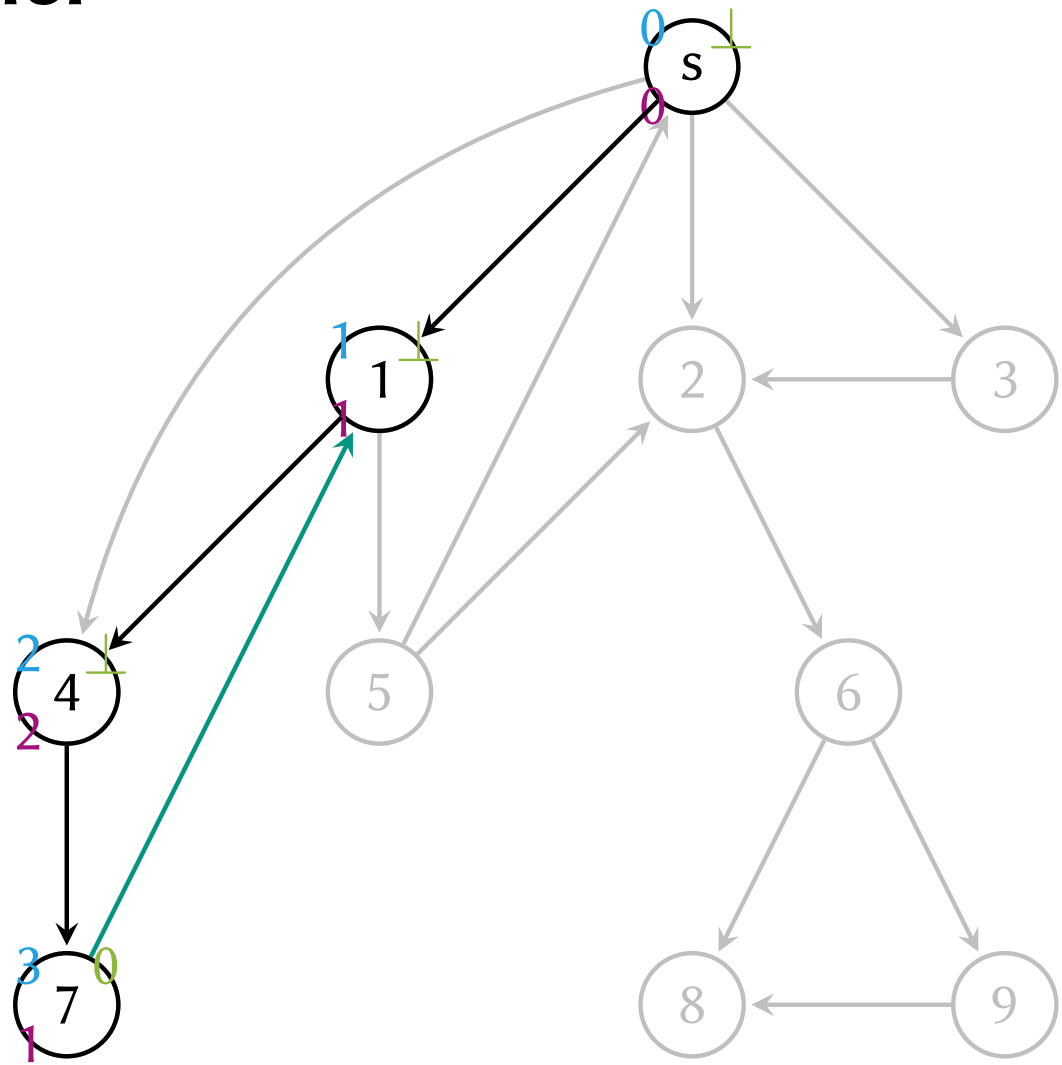
`dfsNum`

`finNum`

`lowNum`

Kanten:

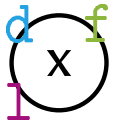
- unknown
- tree
- backward
- forward
- cross



finalisiere 7, gehe zu Parent 4, aktualisiere `finNum[7]` zu 0

DFS Beispiel

Knoten:



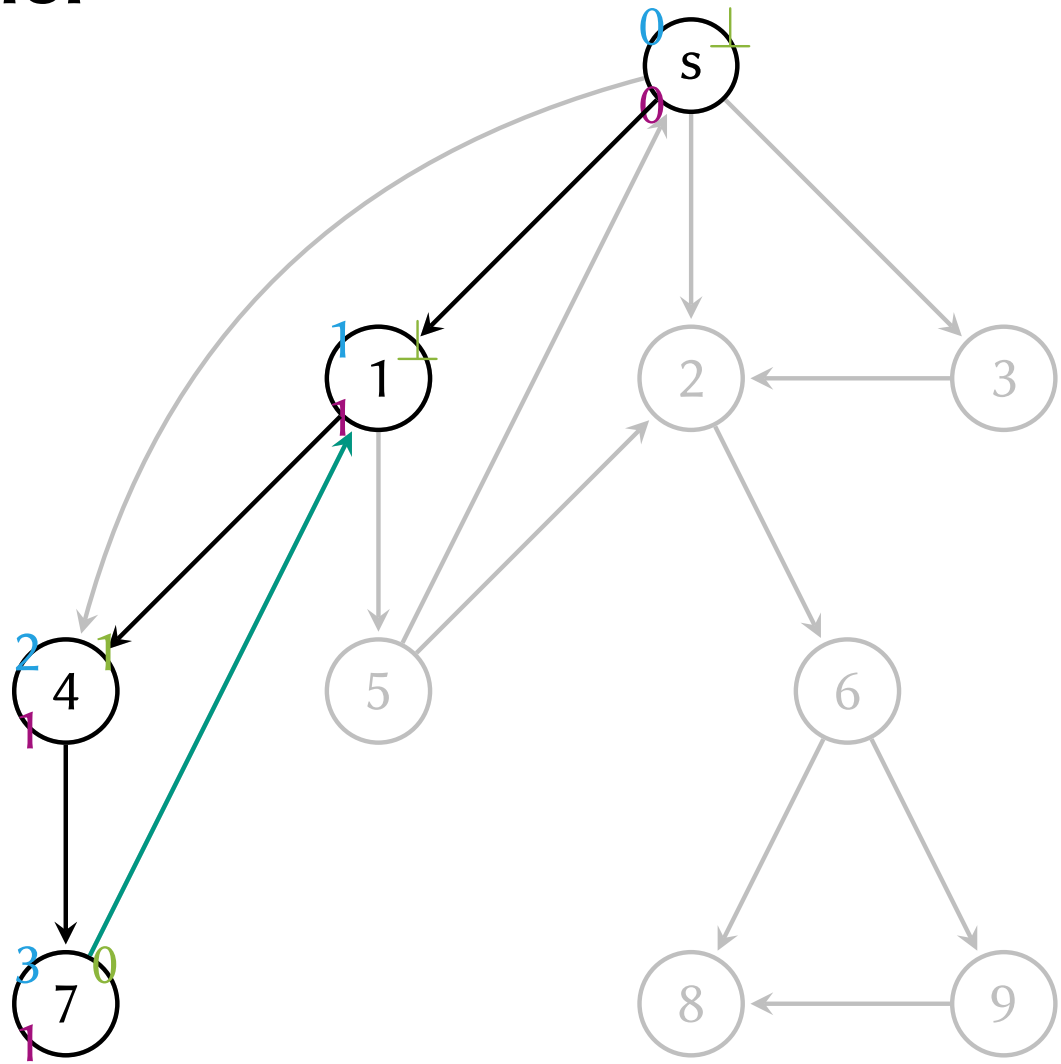
`dfsNum`

`finNum`

`lowNum`

Kanten:

- unknown
- tree
- backward
- forward
- cross

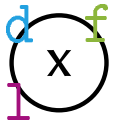


skip animation

finalisiere 4, gehe zu Parent 1, aktualisiere `finNum[4]` zu 1

DFS Beispiel

Knoten:



dfsNum

finNum

lowNum

Kanten:

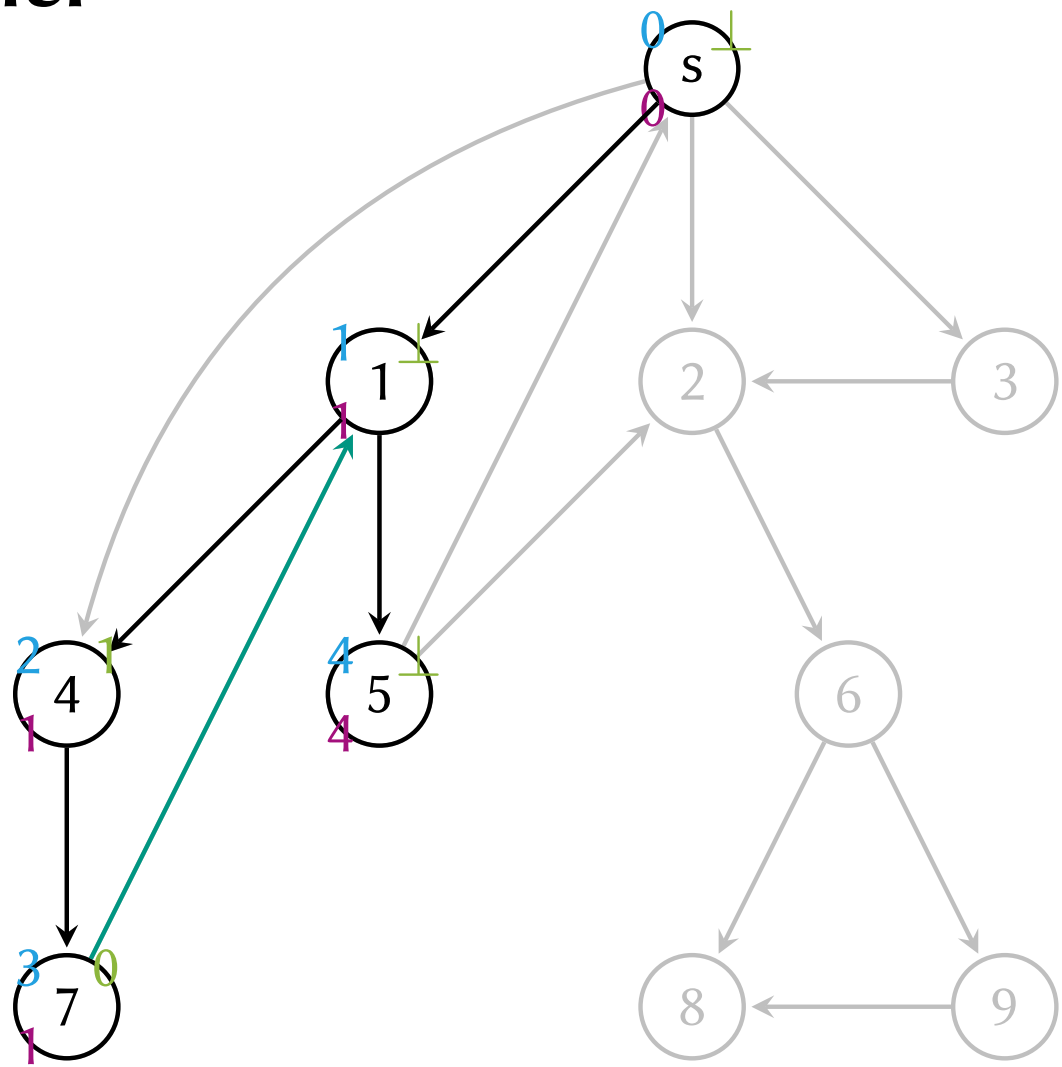
unknown

tree

backward

forward

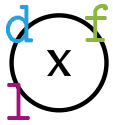
cross



verfolge Baum-Kante $(1, 5)$

DFS Beispiel

Knoten:



dfsNum

finNum

lowNum

Kanten:

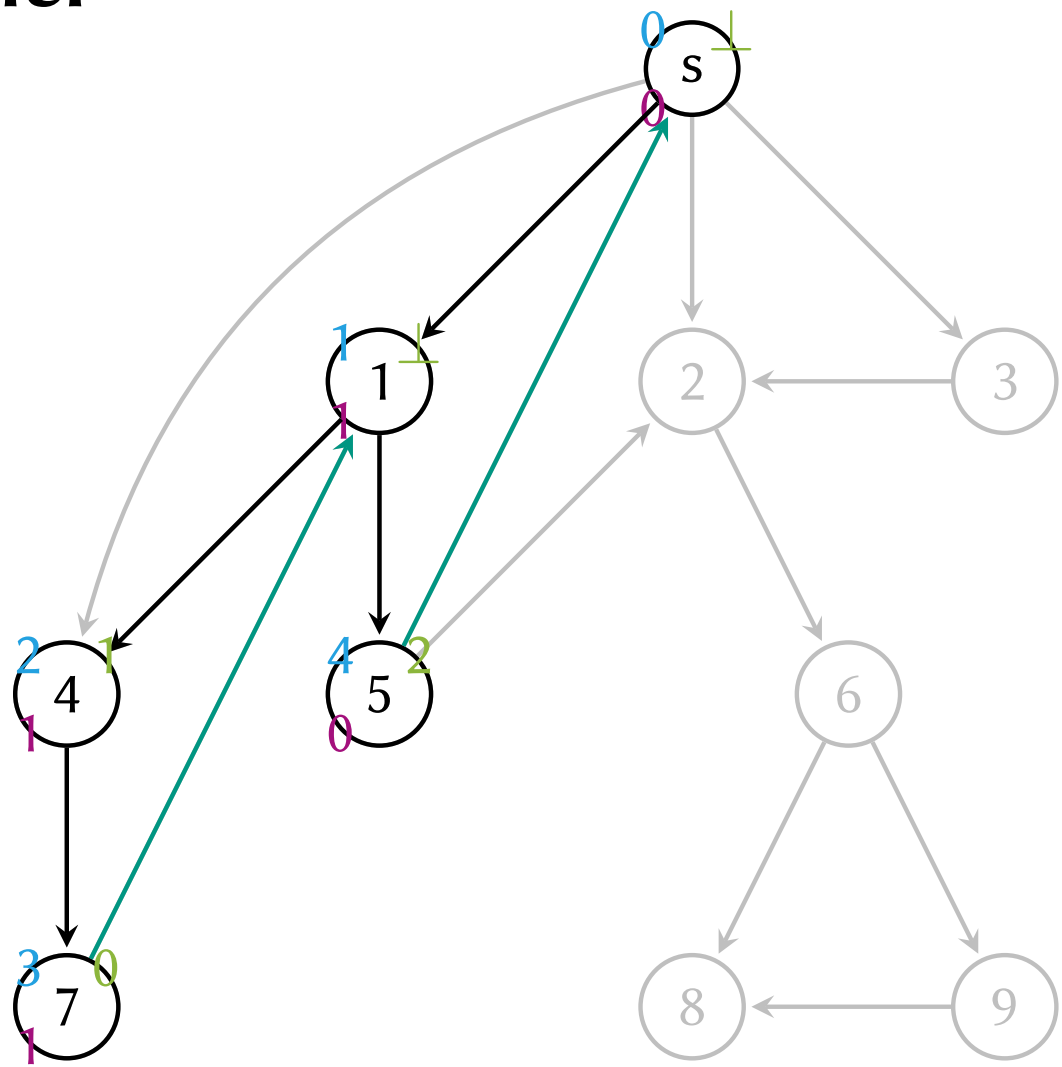
unknown

tree

backward

forward

cross



skip animation

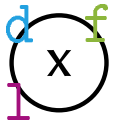
entdecke Rückwärts-Kante $(5, s)$, aktualisiere `lowNum`$[5]$ zu 0

DFS Beispiel

Karlsruher Institut für Technologie

skip animation

Knoten:



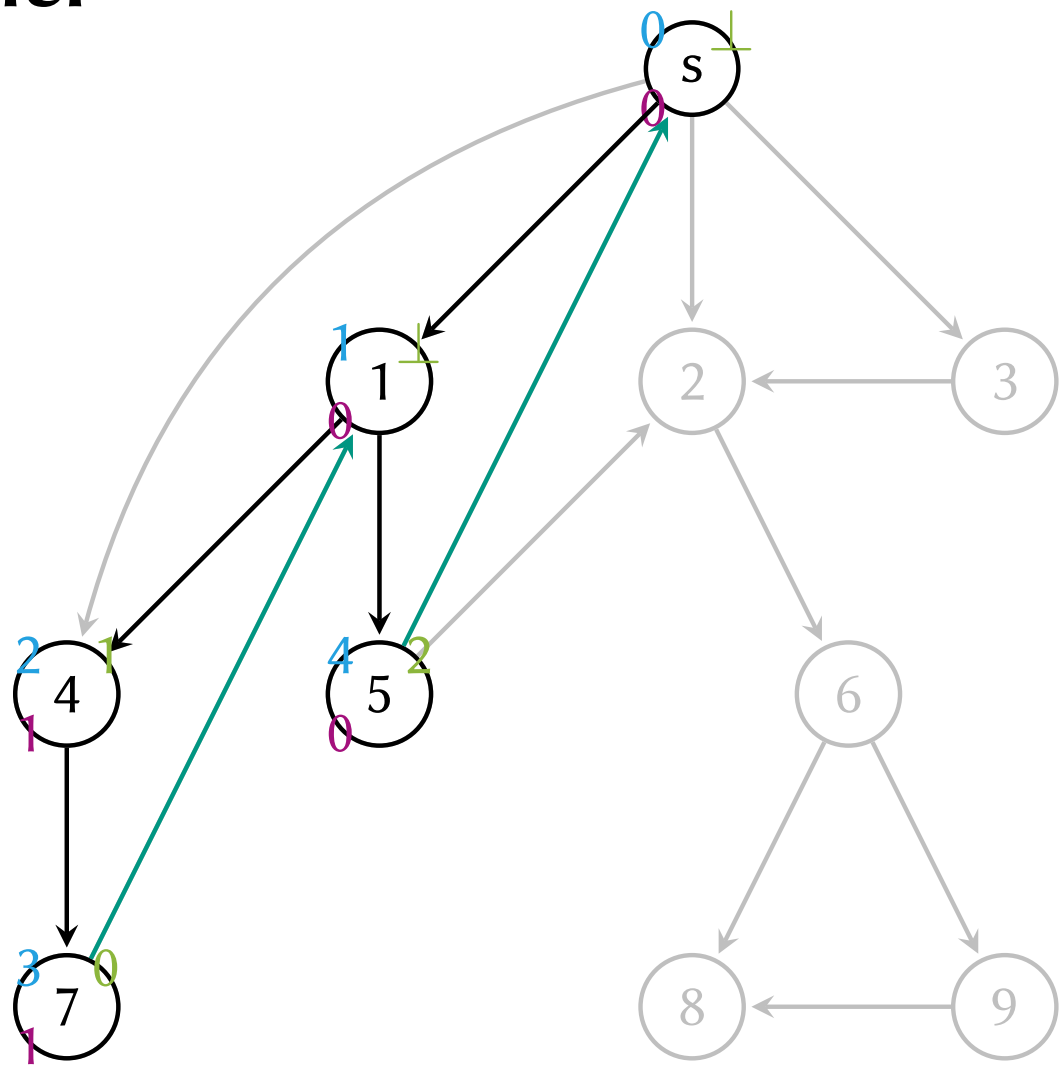
dfsNum

finNum

lowNum

Kanten:

- unknown
- tree
- backward
- forward
- cross

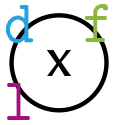


finalisiere 5, gehe zu Parent 1, aktualisiere `finNum[5]` zu 2, `lowNum[1]` zu 0

DFS Beispiel

skip animation

Knoten:



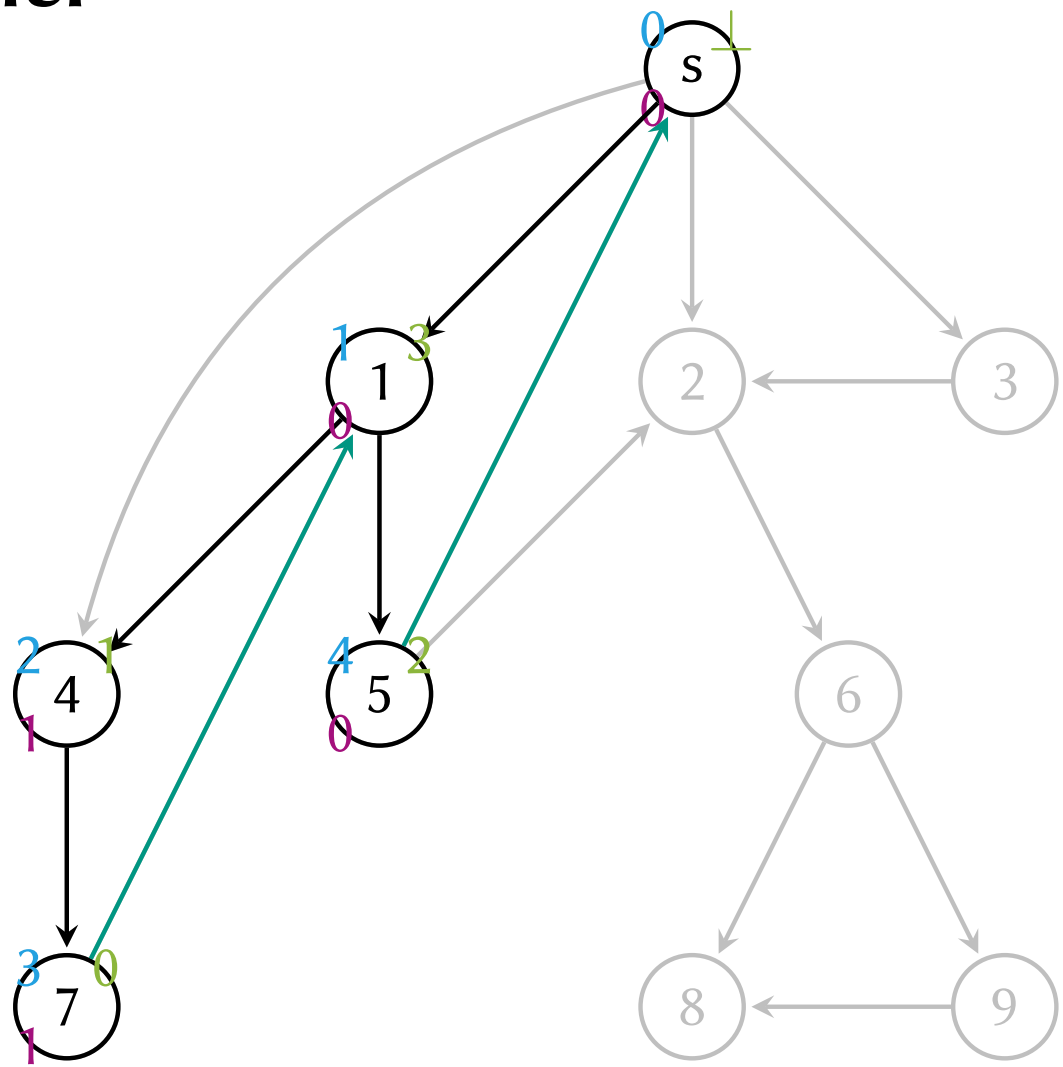
dfsNum

finNum

lowNum

Kanten:

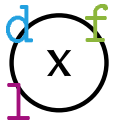
- unknown
- tree
- backward
- forward
- cross



finalisiere 1, gehe zu Parent *s*, aktualisiere `finNum[1]` zu 3

DFS Beispiel

Knoten:



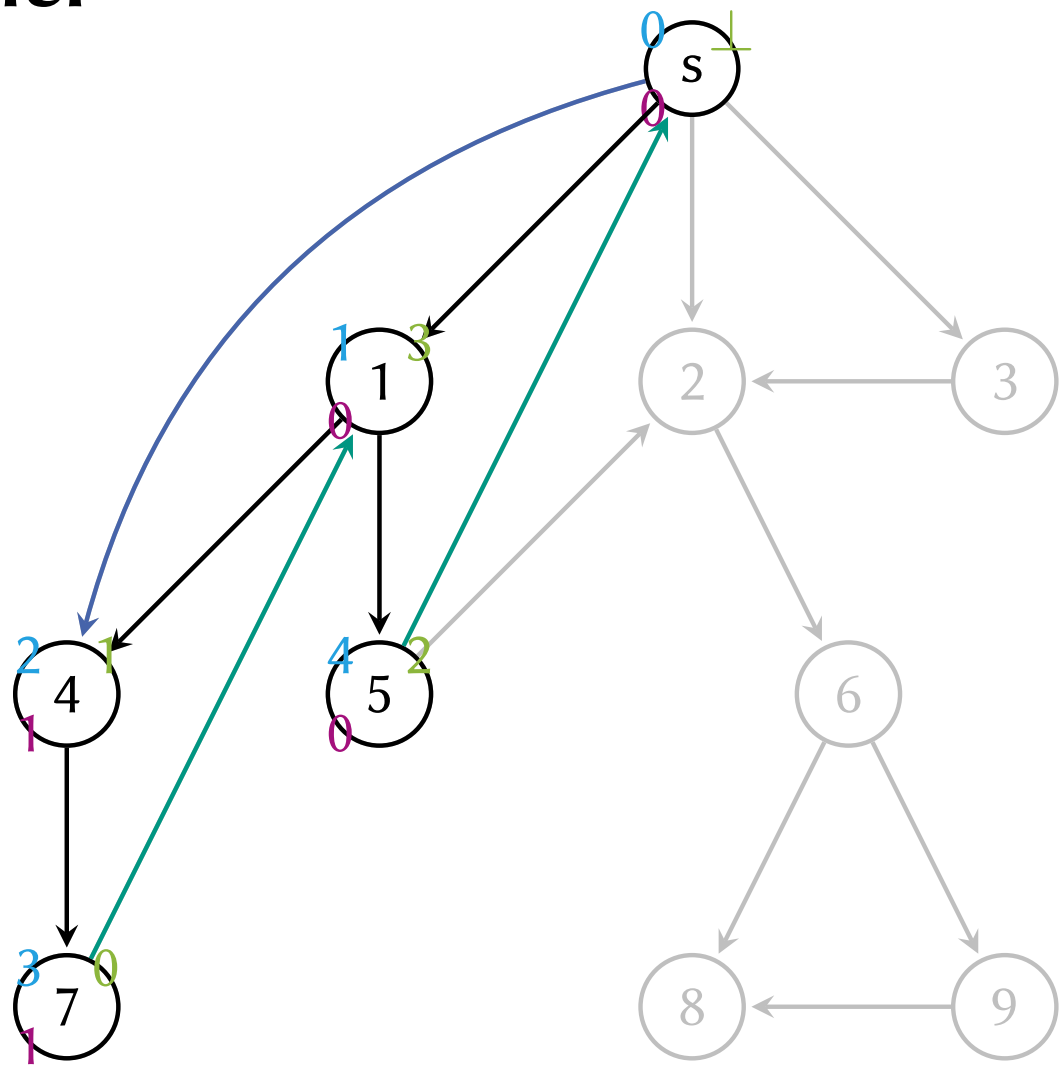
dfsNum

finNum

lowNum

Kanten:

- unknown
- tree
- backward
- forward
- cross



skip animation

entdecke Vorwärts-Kante $(s, 4)$

DFS Beispiel

Knoten:

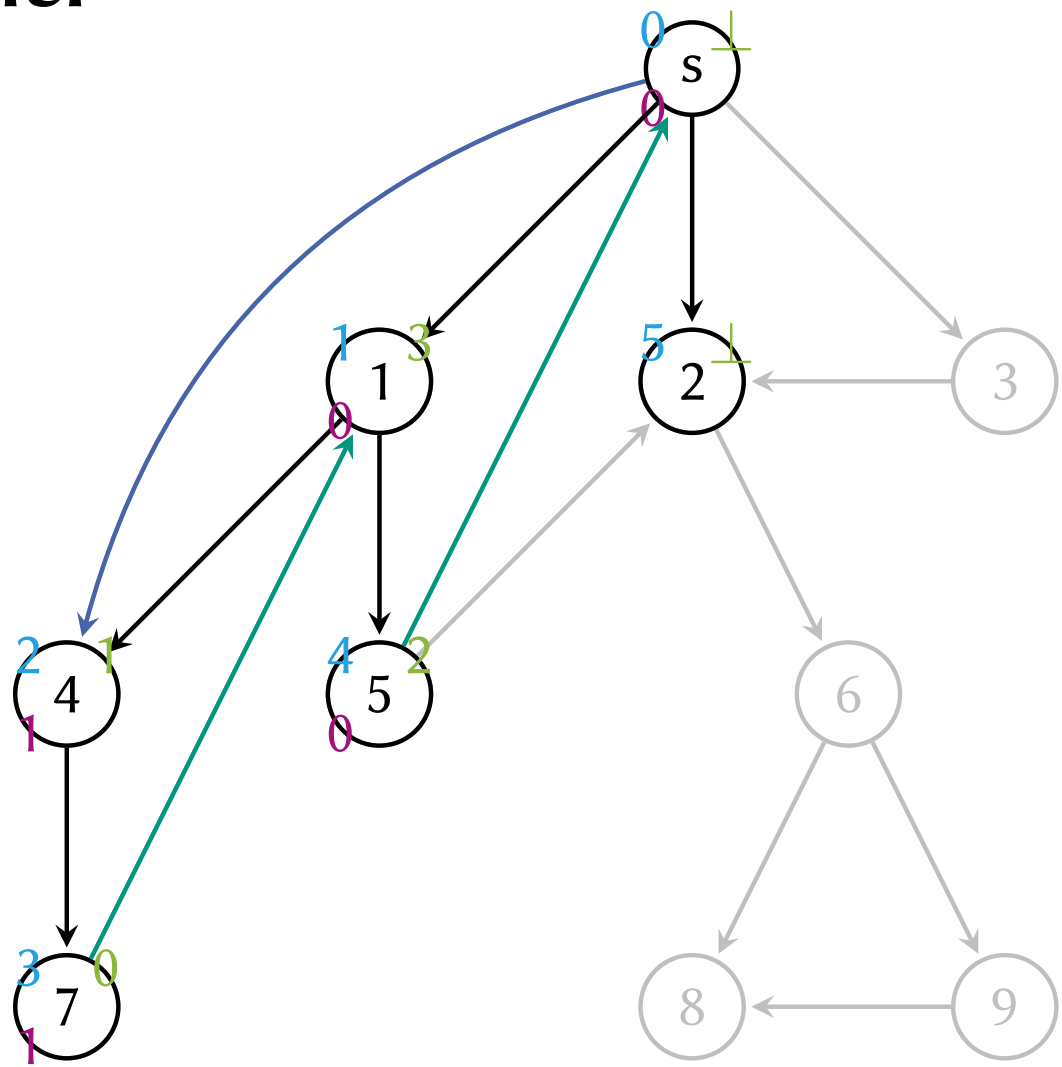
dfsNum

finNum

lowNum

Kanten:

- unknown
- tree
- backward
- forward
- cross



skip animation

verfolge Baum-Kante $(s, 2)$

DFS Beispiel

Knoten:

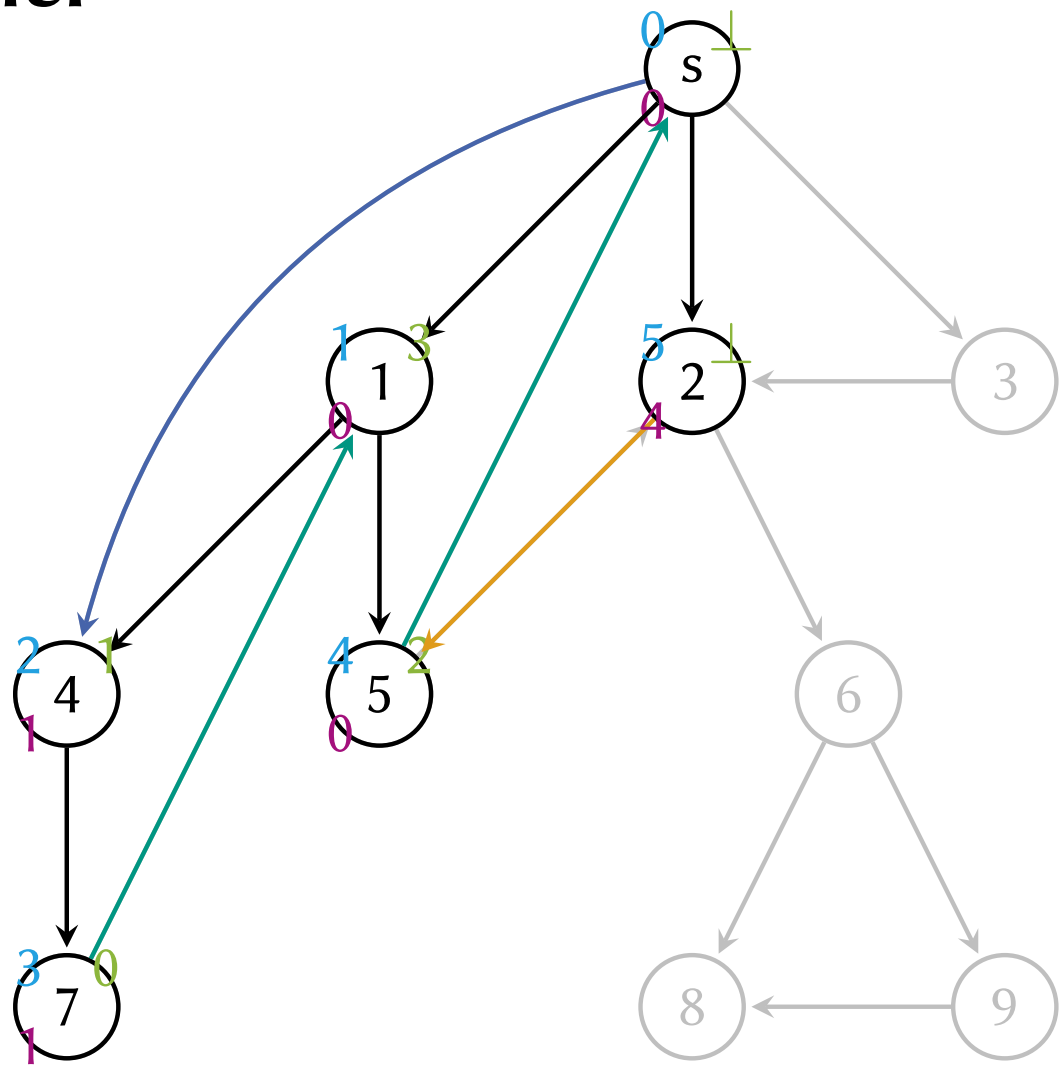
`dfsNum`

`finNum`

`lowNum`

Kanten:

- unknown
- tree
- backward
- forward
- cross



skip animation

entdecke Quer-Kante $(2, 5)$, aktualisiere `lowNum`$[2]$ zu 4

DFS Beispiel

Knoten:

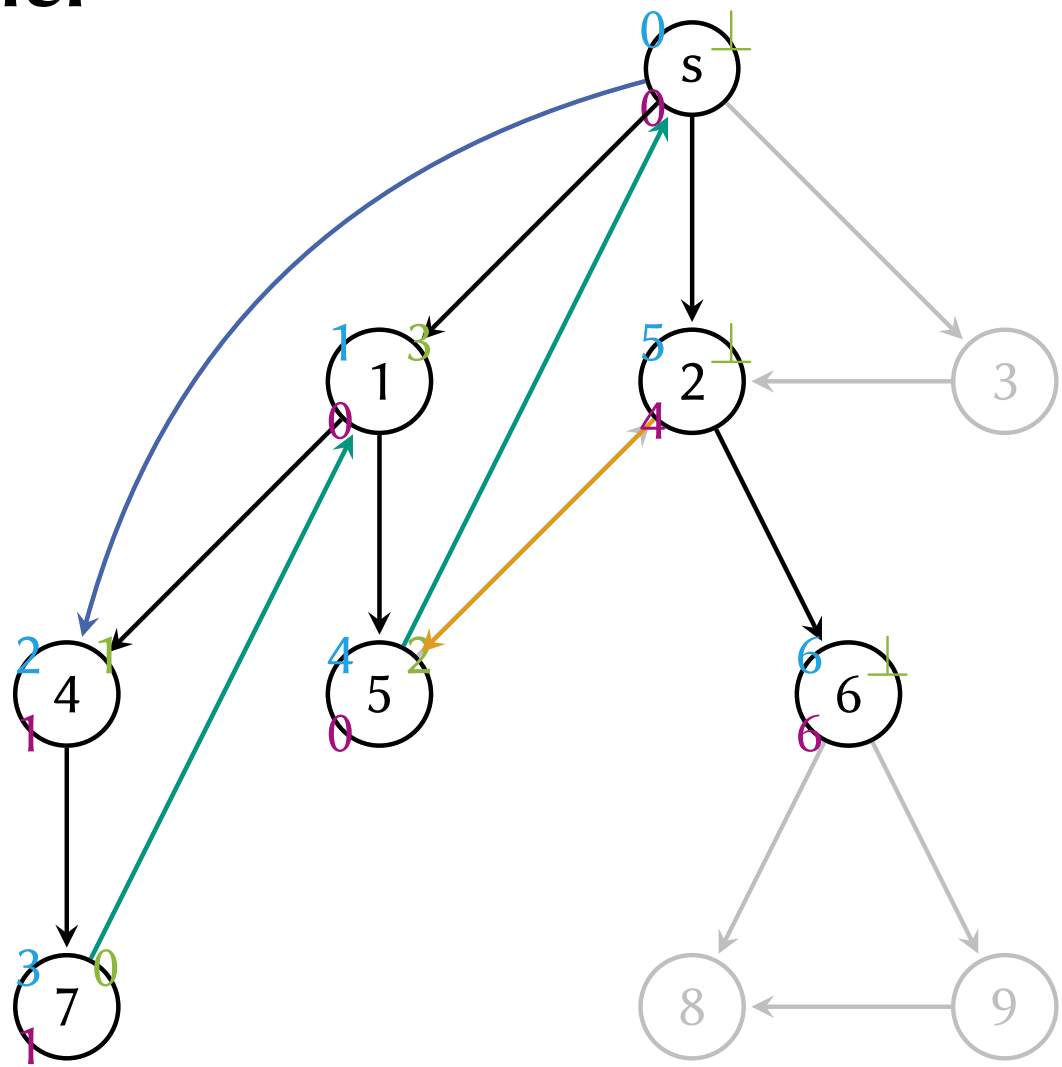
dfsNum

finNum

lowNum

Kanten:

- unknown
- tree
- backward
- forward
- cross



verfolge Baum-Kante $(2, 6)$

DFS Beispiel

Knoten:

dfsNum

finNum

lowNum

Kanten:

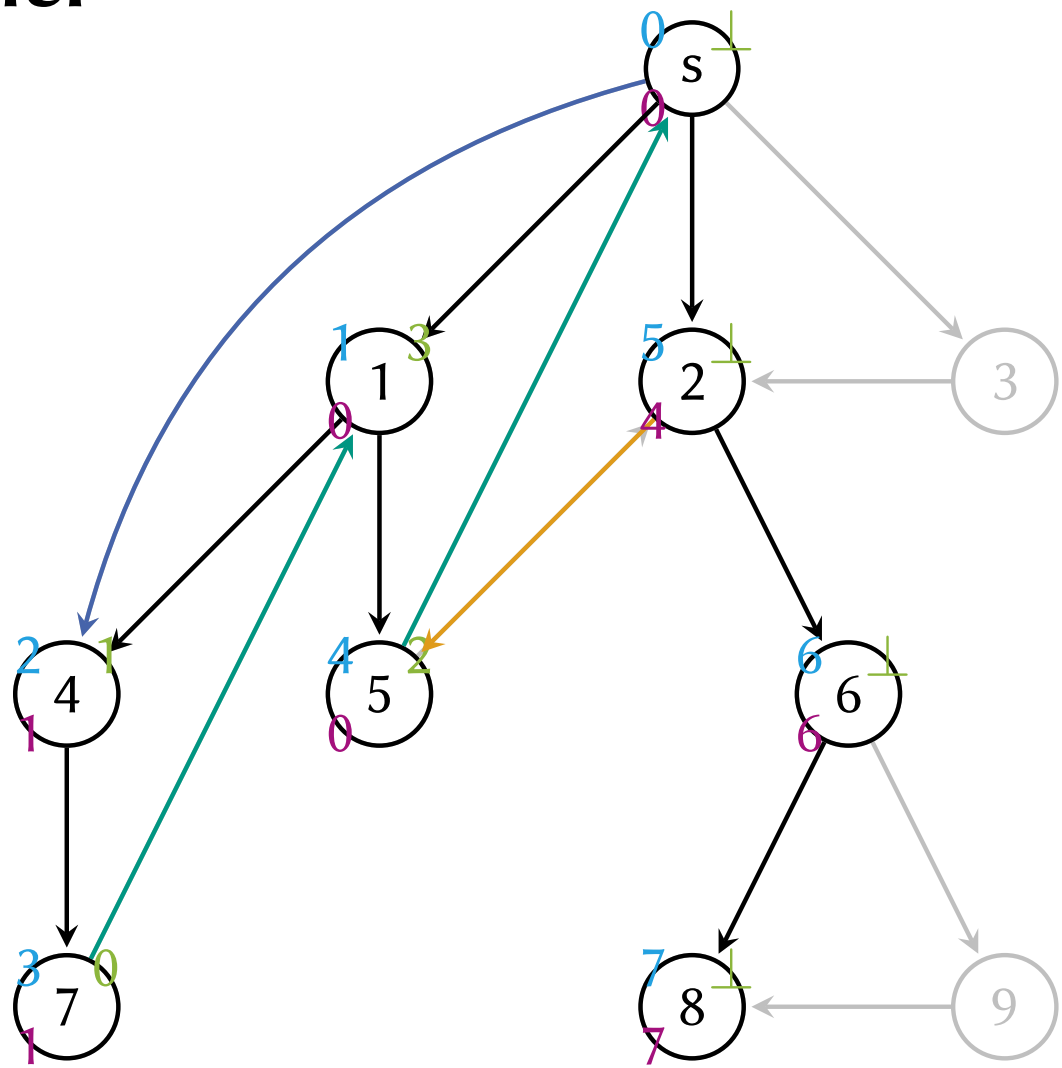
unknown

tree

backward

forward

cross



verfolge Baum-Kante $(6, 8)$

DFS Beispiel

Knoten:

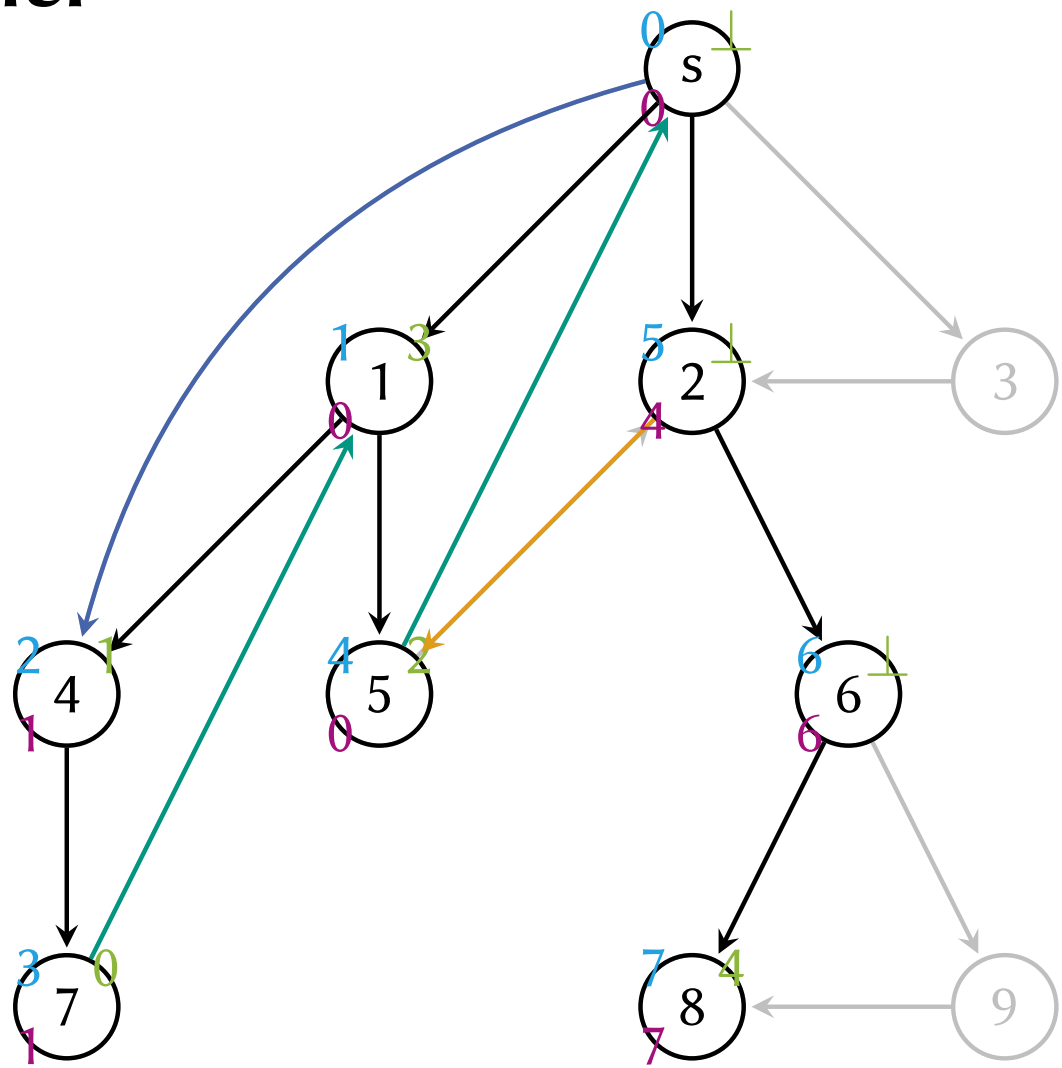
`dfsNum`

`finNum`

`lowNum`

Kanten:

- unknown
- tree
- backward
- forward
- cross



skip animation

finalisiere 8, gehe zu Parent 6, aktualisiere `finNum`[8] zu 4

DFS Beispiel

Knoten:

dfsNum

finNum

lowNum

Kanten:

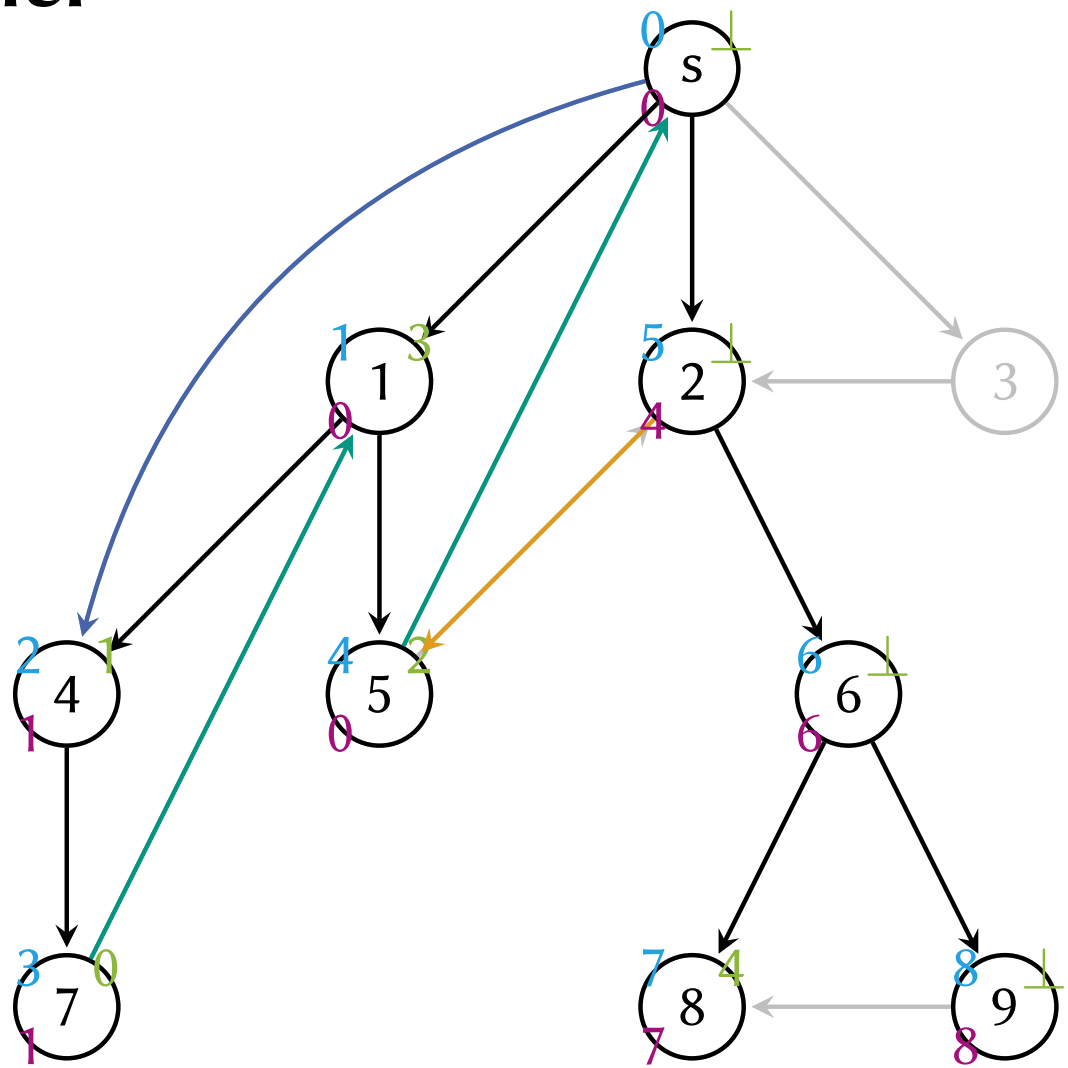
unknown

tree

backward

forward

cross



verfolge Baum-Kante $(6, 9)$

DFS Beispiel

Knoten:

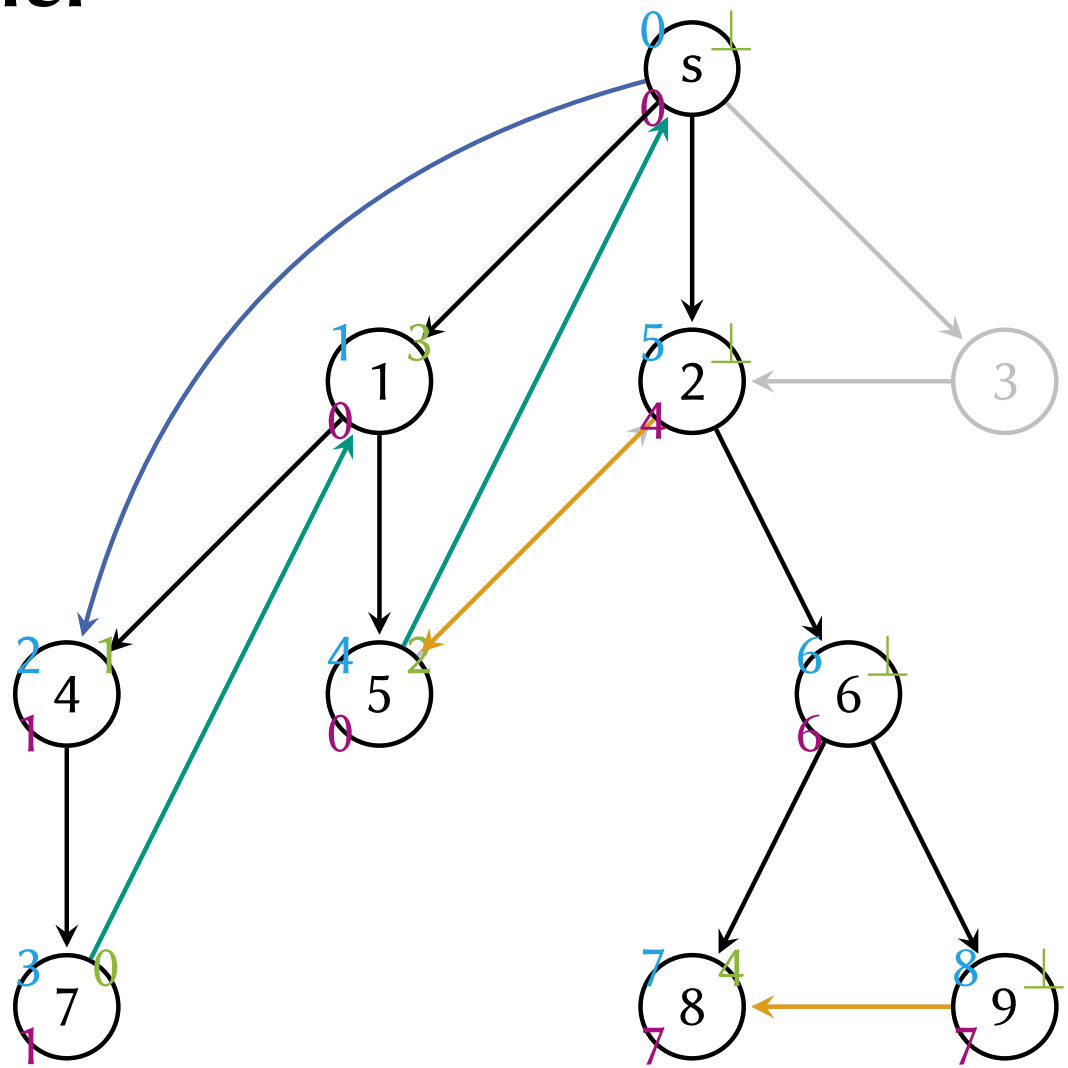
dfsNum

finNum

lowNum

Kanten:

- unknown
- tree
- backward
- forward
- cross



skip animation

entdecke Quer-Kante $(9, 8)$, aktualisiere `lowNum`$[9]$ zu 7

DFS Beispiel

skip animation

Knoten:

dfsNum

finNum

lowNum

Kanten:

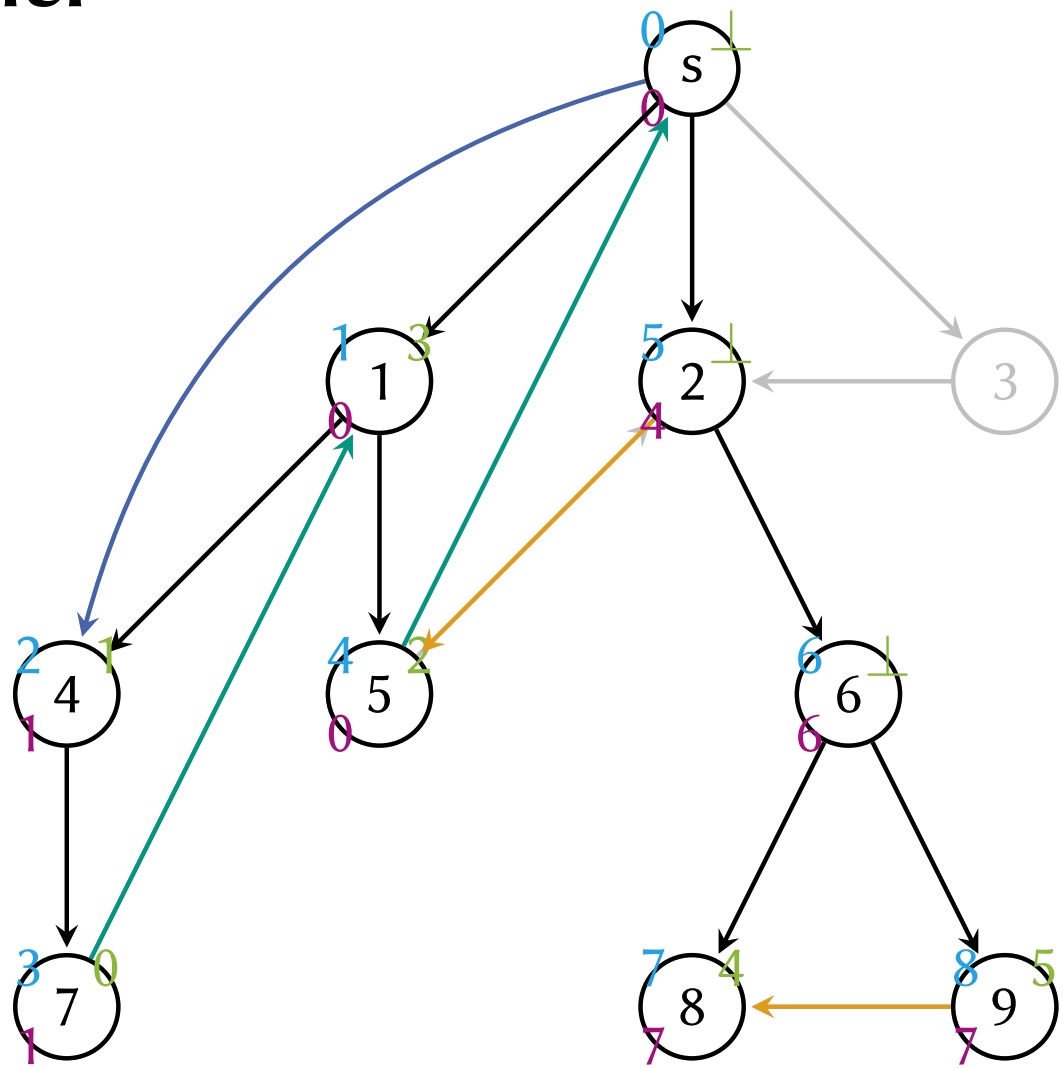
unknown

tree

backward

forward

cross



finalisiere 9, gehe zu Parent 6, aktualisiere `finNum`[9] zu 5

DFS Beispiel

Knoten:

dfsNum

finNum

lowNum

Kanten:

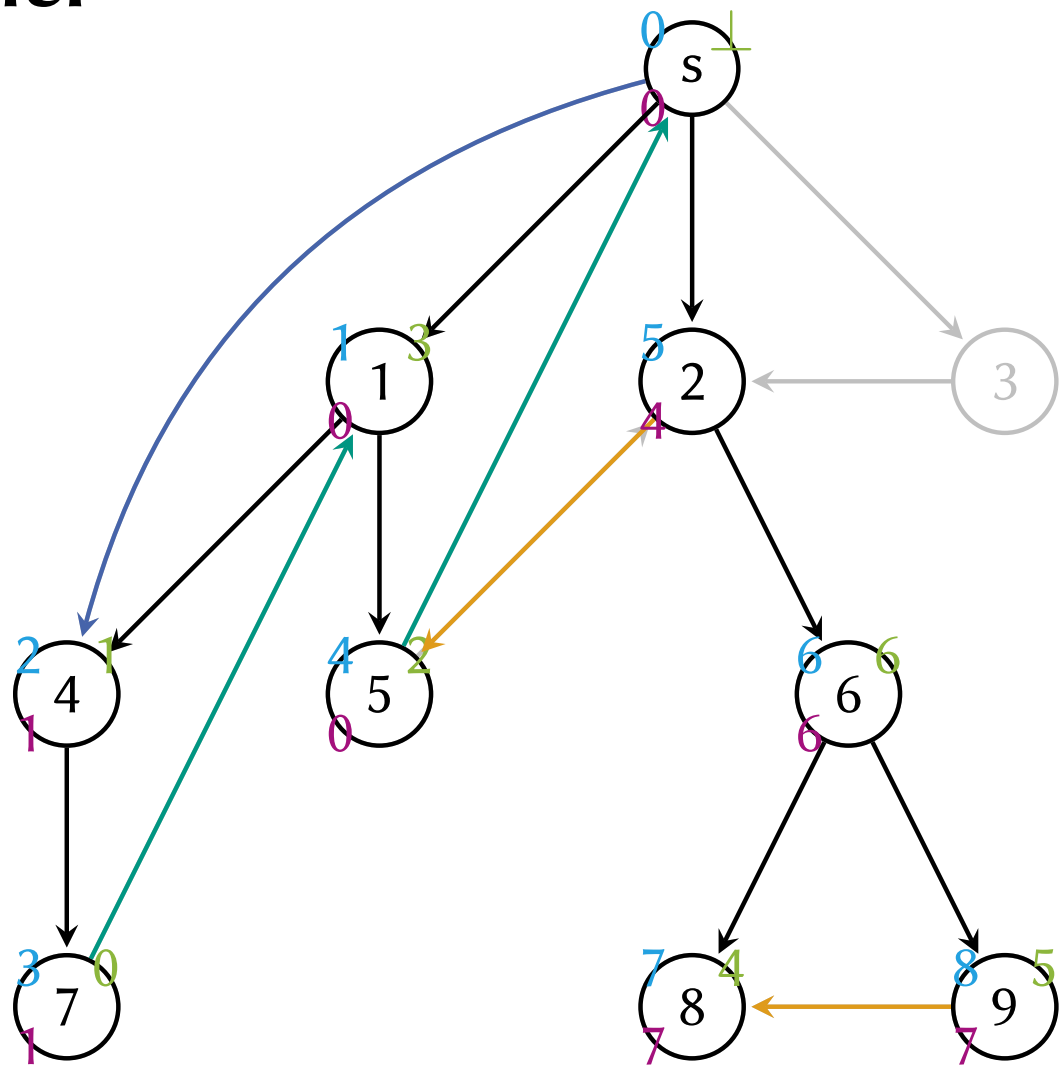
unknown

tree

backward

forward

cross



skip animation

finalisiere 6, gehe zu Parent 2, aktualisiere `finNum`[6] zu 6

DFS Beispiel

Knoten:

dfsNum

finNum

lowNum

Kanten:

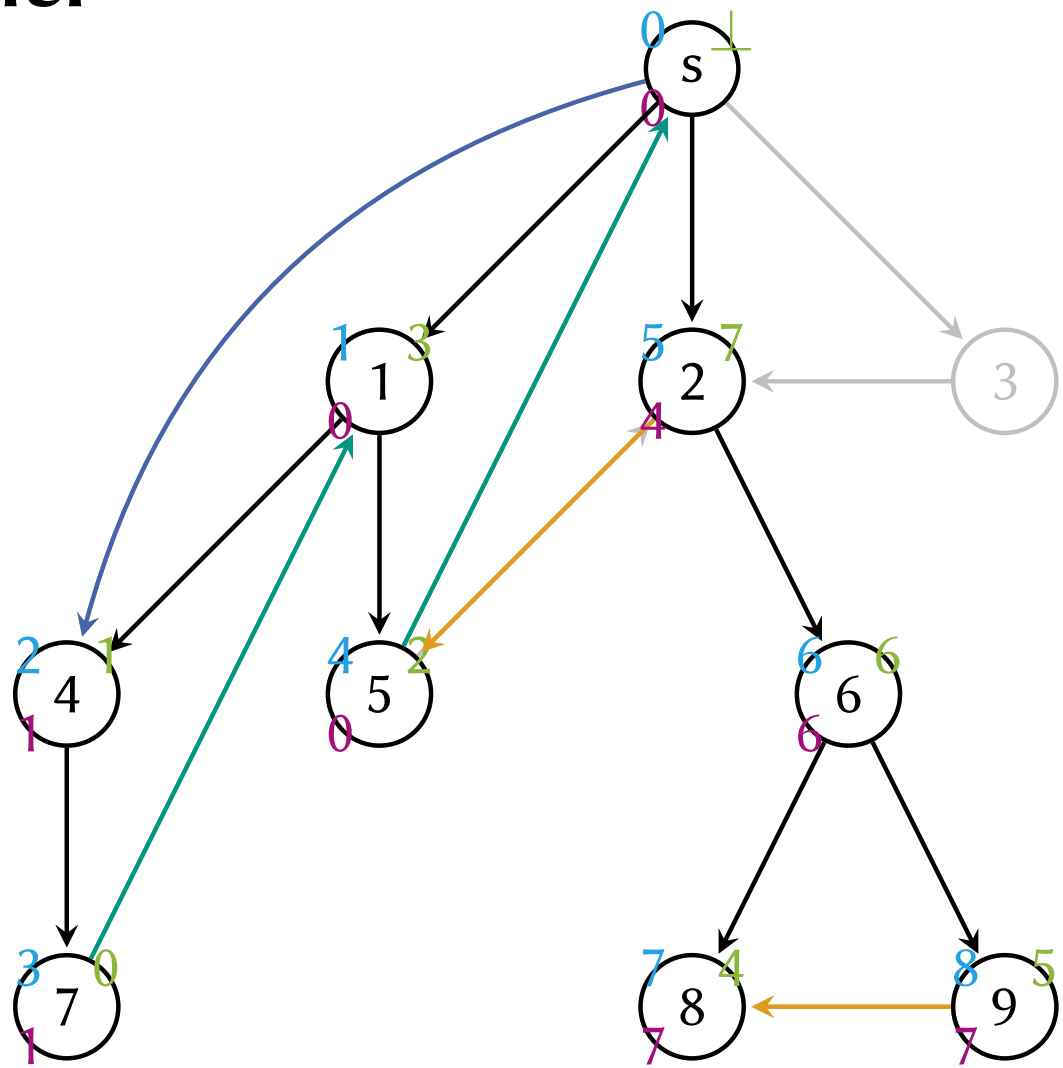
unknown

tree

backward

forward

cross



skip animation

finalisiere 2, gehe zu Parent *s*, aktualisiere `finNum[2]` zu 7

DFS Beispiel

Knoten:

dfsNum

finNum

lowNum

Kanten:

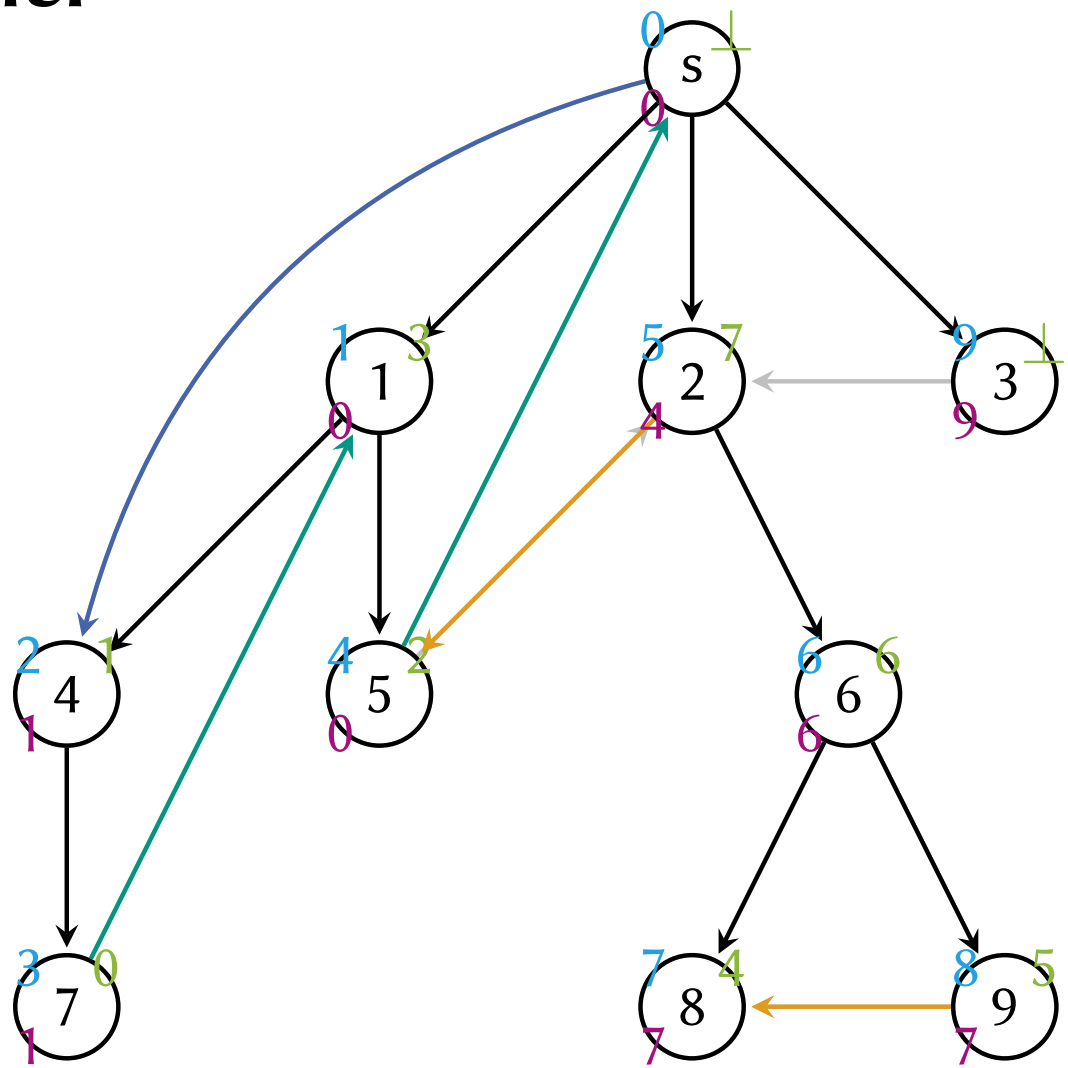
unknown

tree

backward

forward

cross



verfolge Baum-Kante $(s, 3)$

DFS Beispiel

Knoten:

dfsNum

finNum

lowNum

Kanten:

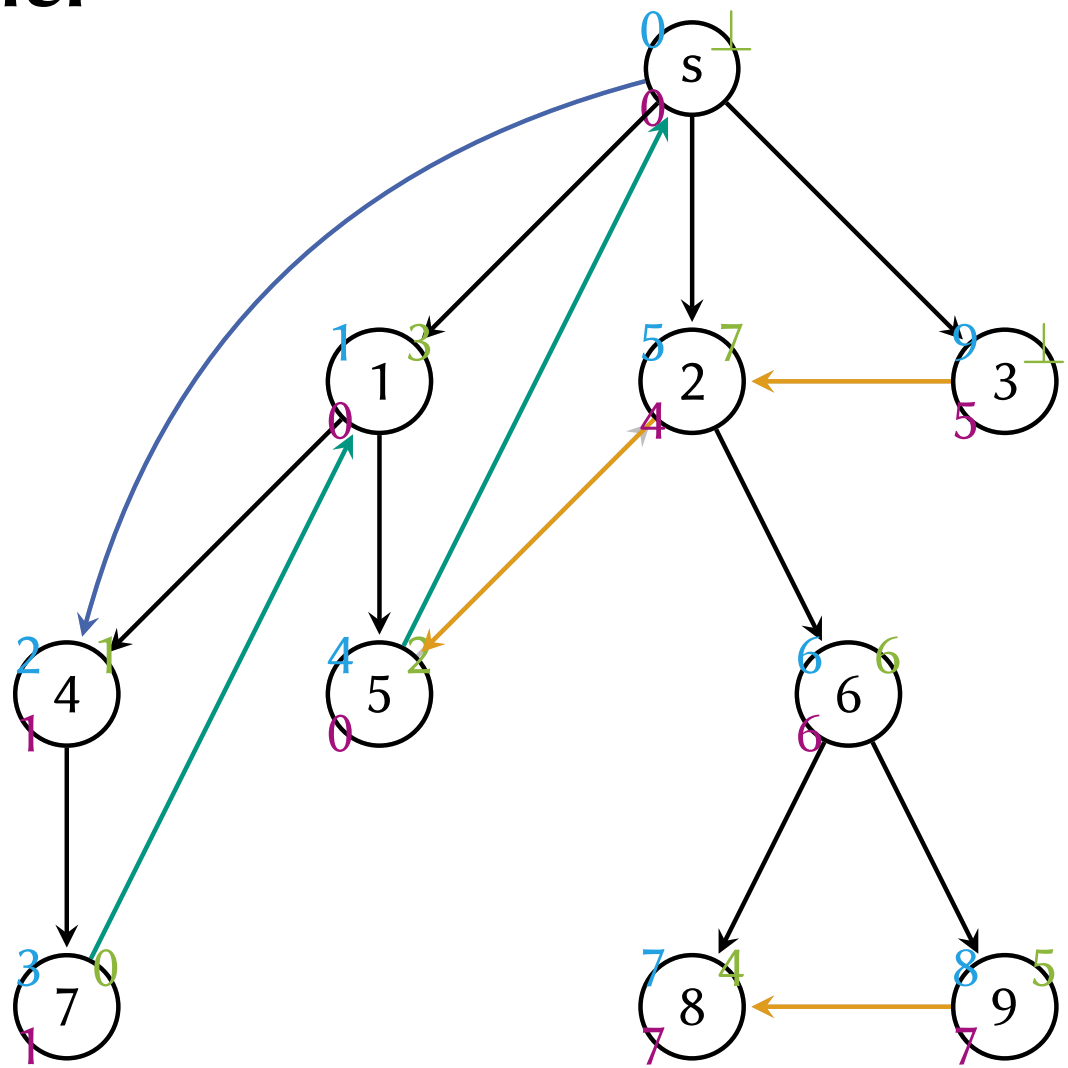
unknown

tree

backward

forward

cross



entdecke Quer-Kante $(3, 6)$, aktualisiere `lowNum`$[3]$ zu 5

DFS Beispiel

skip animation

Knoten:

dfsNum

finNum

lowNum

Kanten:

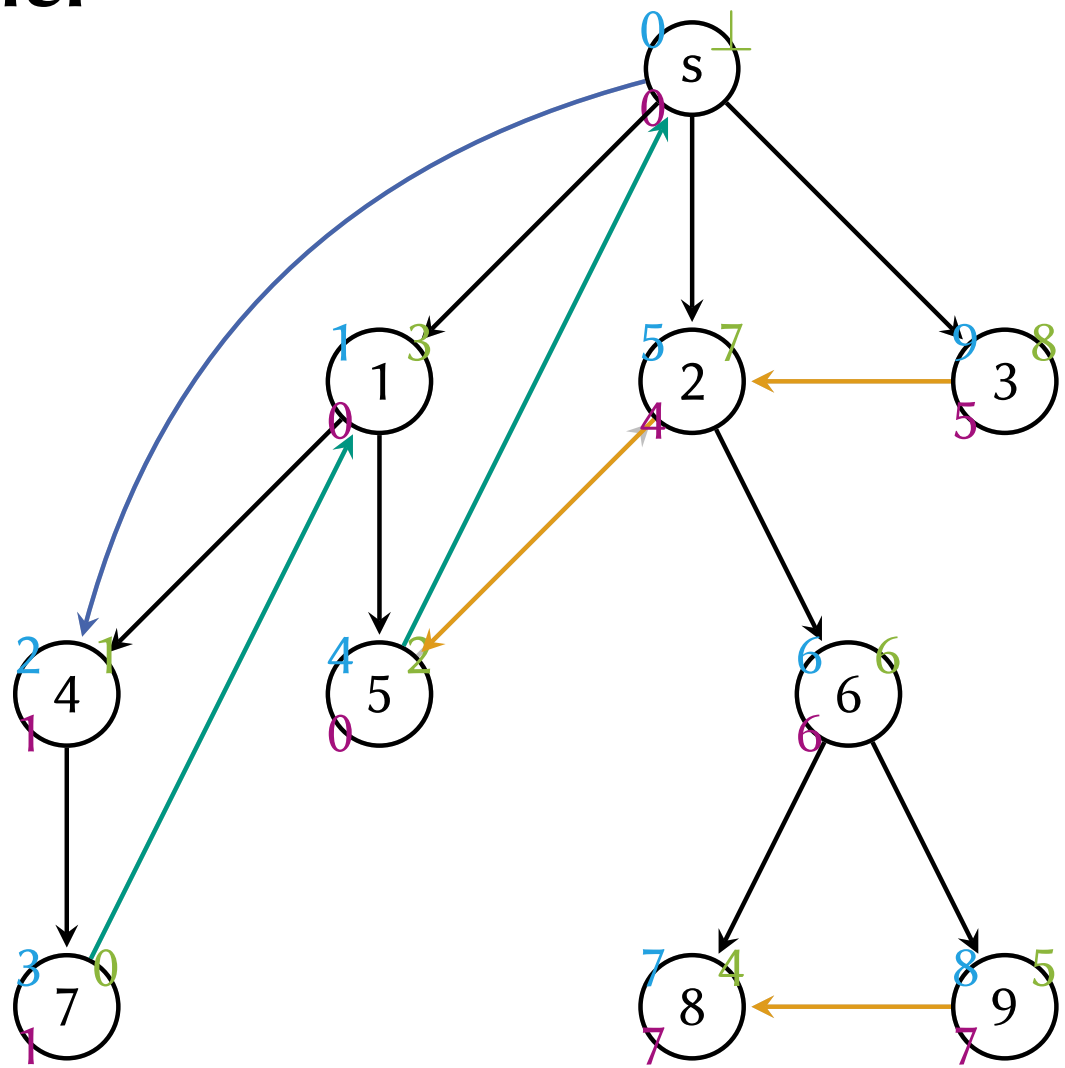
unknown

tree

backward

forward

cross



finalisiere 3, gehe zu Parent *s*, aktualisiere `finNum`[3] zu 8

DFS Beispiel

Knoten:

dfsNum

finNum

lowNum

Kanten:

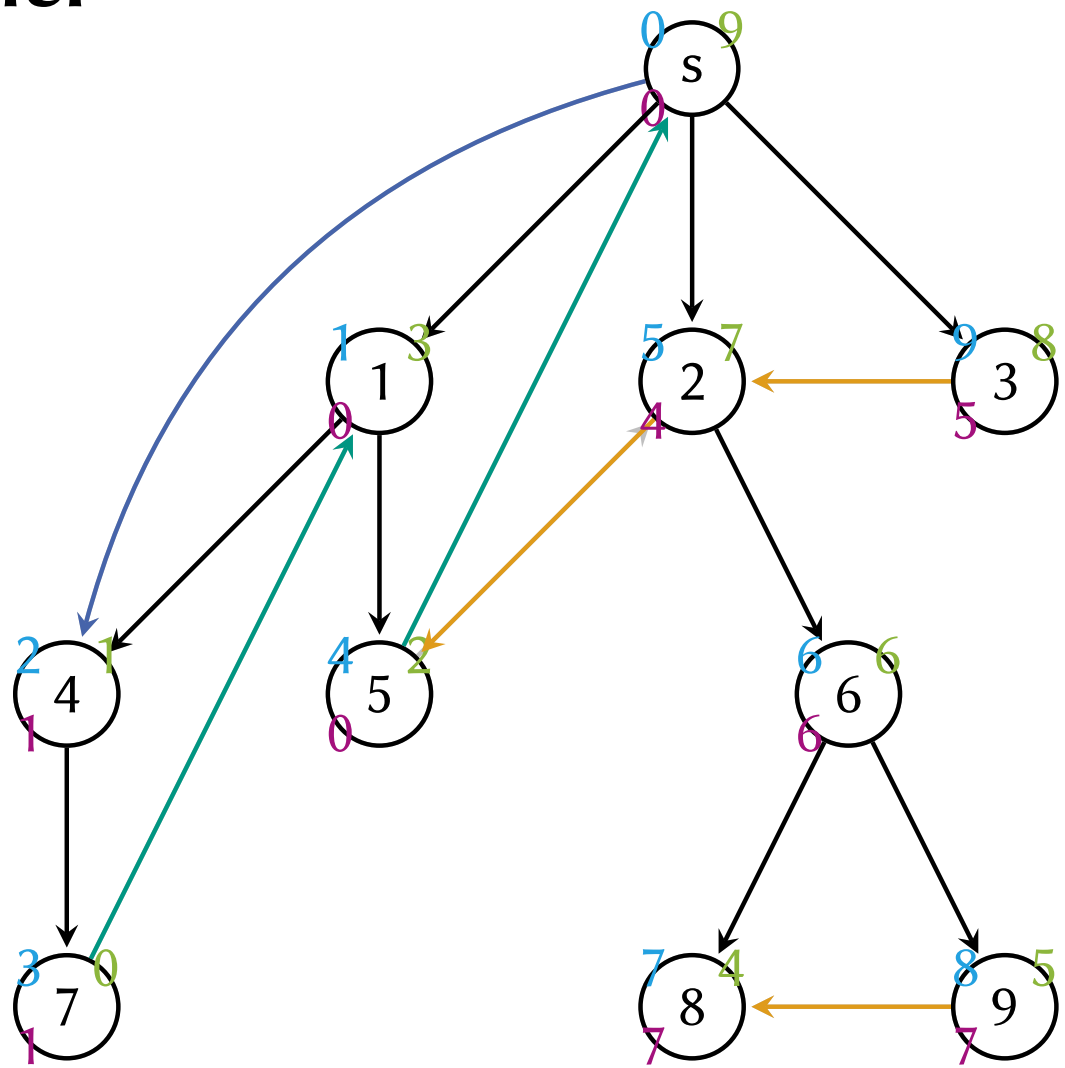
unknown

tree

backward

forward

cross



finalisiere *s*

Edgetypes mit Knoteninformation

Sei Kante $(v, u) \in E$ gegeben, wie bekommen wir den Kantentyp?

Edgetypes mit Knoteninformation

Sei Kante $(v, u) \in E$ gegeben, wie bekommen wir den Kantentyp?

	DFS-Number	FIN-Number	Beziehung
Tree-Edge	klein $\rightarrow$ groß	groß $\rightarrow$ klein	$\texttt{par}[u] = v$
Forward-Edge	klein $\rightarrow$ groß	groß $\rightarrow$ klein	$\texttt{par}[u] \neq v$
Backward-Edge	groß $\rightarrow$ klein	groß $\rightarrow$ klein	—
Cross-Edge	groß $\rightarrow$ klein	klein $\rightarrow$ groß	—

Edgetypes mit Knoteninformation

Sei Kante $(v, u) \in E$ gegeben, wie bekommen wir den Kantentyp?

	DFS-Number	FIN-Number	Beziehung
Tree-Edge	klein $\rightarrow$ groß	groß $\rightarrow$ klein	$\texttt{par}[u] = v$
Forward-Edge	klein $\rightarrow$ groß	groß $\rightarrow$ klein	$\texttt{par}[u] \neq v$
Backward-Edge	groß $\rightarrow$ klein	groß $\rightarrow$ klein	—
Cross-Edge	groß $\rightarrow$ klein	klein $\rightarrow$ groß	—

Hinweis

Die Knoteninformationen können nicht nur zum Erkennen von Kanten verwendet werden. Wir können dadurch auch Relationen zwischen Knoten erkennen.

Knoten in Relation (gerichtet)

Im Folgenden betrachten wir zwei unterschiedliche Knoten $u, v \in V$ aus einem beliebigen Graphen. Es gilt für den DFS-Baum des Graphen:

	Beziehung
u wurde vor v entdeckt	$\texttt{dfsNum}[u] < \texttt{dfsNum}[v]$
u wurde vor v finalisiert	$\texttt{finNum}[u] < \texttt{finNum}[v]$
u ist Kind von v	$\texttt{dfsNum}[v] < \texttt{dfsNum}[u] \wedge \texttt{finNum}[u] < \texttt{finNum}[v]$
u liegt auf einem Kreis	$\texttt{lowNum}[u] = \texttt{dfsNum}[v] \wedge u$ ist Kind von v

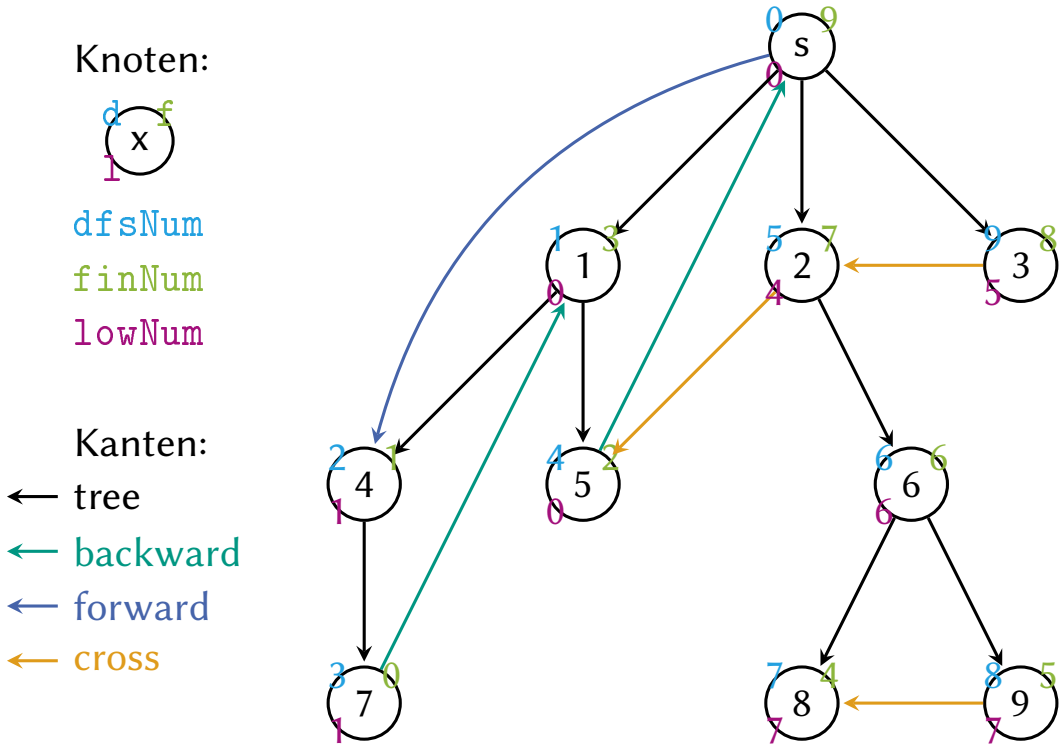
u ist Kind von *v*

Bedingung:

$$\texttt{dfsNum}[v] < \texttt{dfsNum}[u]$$
$$\wedge\ \texttt{finNum}[u] < \texttt{finNum}[v]$$

Zum Beispiel 8 ist Kind von 2:

$$\texttt{dfsNum}[2] = 5 < 7 = \texttt{dfsNum}[8] \wedge \texttt{finNum}[8] = 4 < 7 = \texttt{finNum}[2].$$

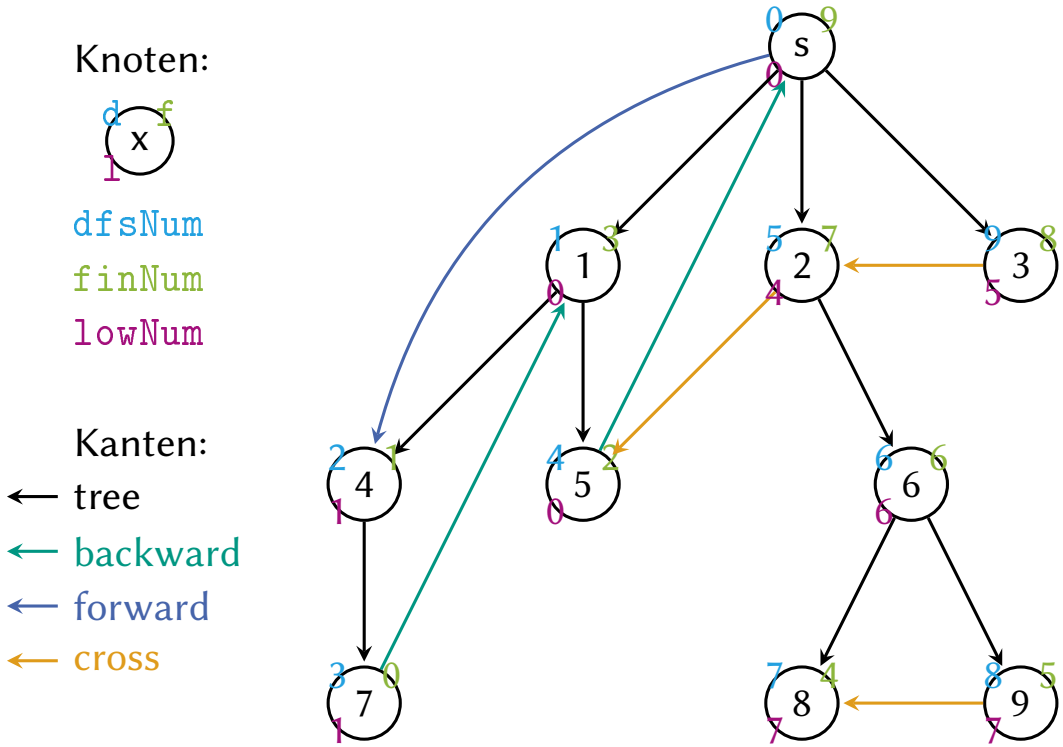


u liegt auf einem Kreis

Bedingung:

$$\texttt{lowNum}[u] = \texttt{dfsNum}[v]$$
$$\wedge \quad u \text{ ist Kind von } v$$

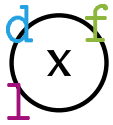
Zum Beispiel liegt 7 auf einem Kreis mit 1: $\texttt{lowNum}[7] = 1 = \texttt{dfsNum}[1]$
$\wedge\, \texttt{dfsNum}[1] = 1 < 3 = \texttt{dfsNum}[7] \wedge \texttt{finNum}[7] = 0 < 3 = \texttt{finNum}[1]$.



DFS Beispiel

skip animation

Knoten:



dfsNum

finNum

lowNum

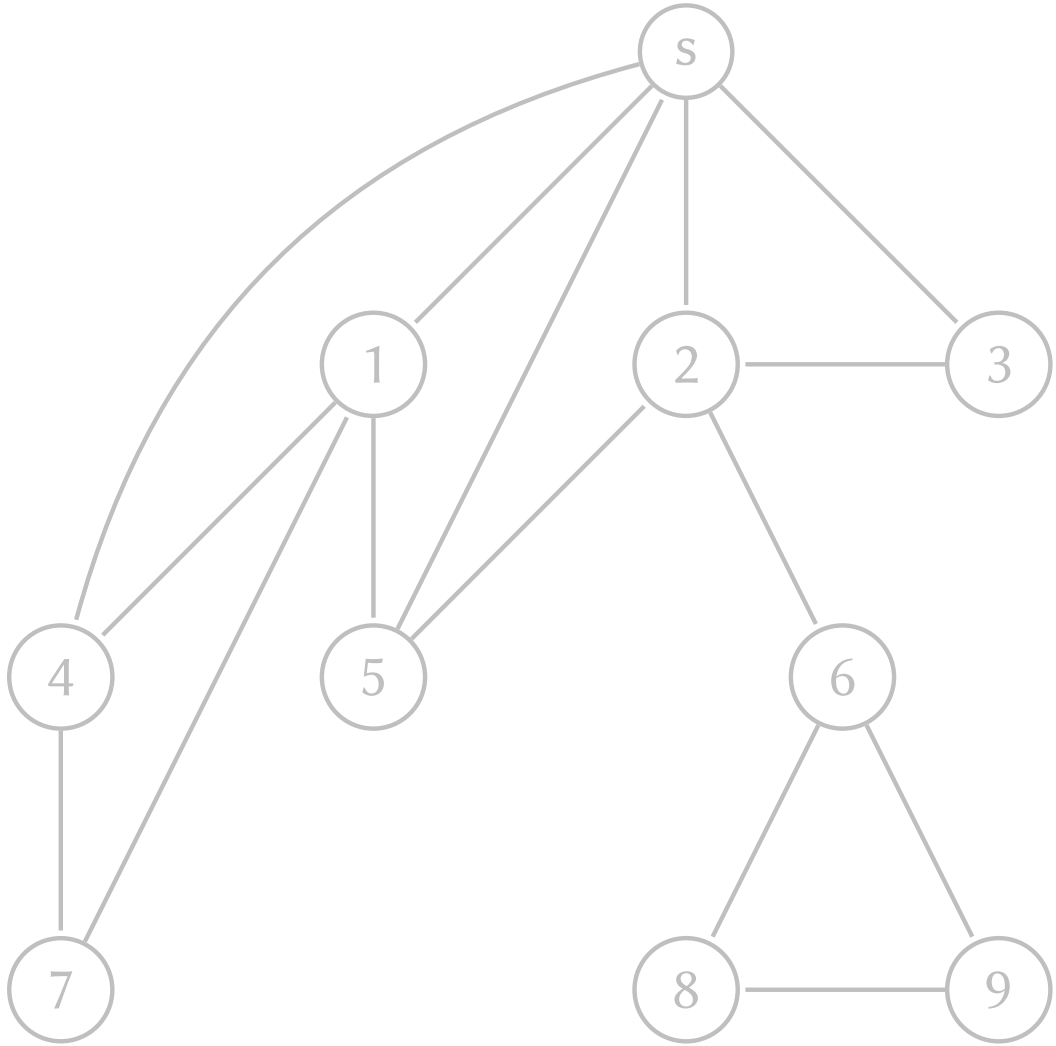
Kanten:

← unknown

← tree

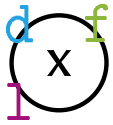
← backward

← forward



DFS Beispiel

Knoten:



dfsNum

finNum

lowNum

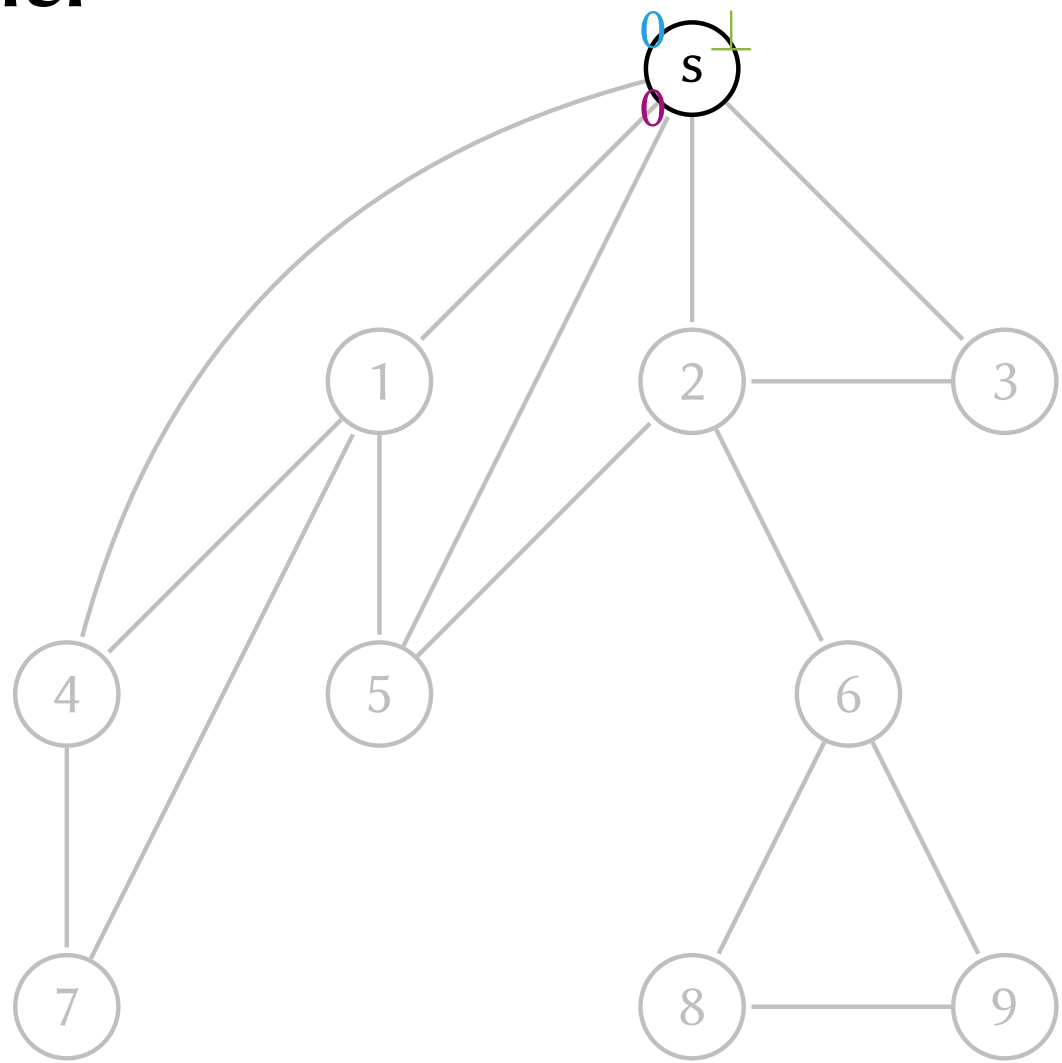
Kanten:

← unknown

← tree

← backward

← forward

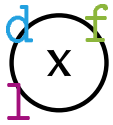


skip animation

Starte bei Startknoten s

DFS Beispiel

Knoten:



dfsNum

finNum

lowNum

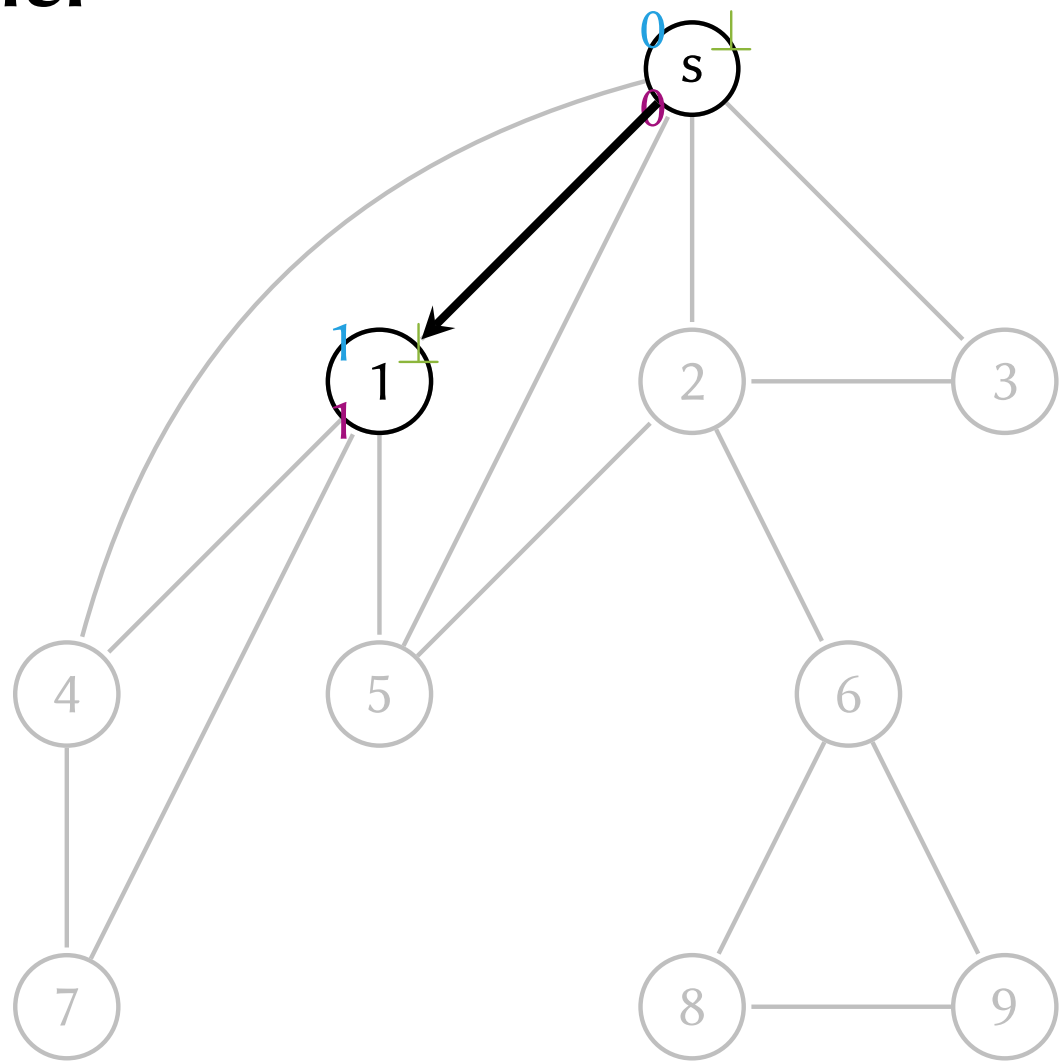
Kanten:

unknown

tree

backward

forward



skip animation

verfolge Baum-Kante $(s, 1)$

DFS Beispiel

Karlsruher Institut für Technologie

skip animation

Knoten:

dfsNum

finNum

lowNum

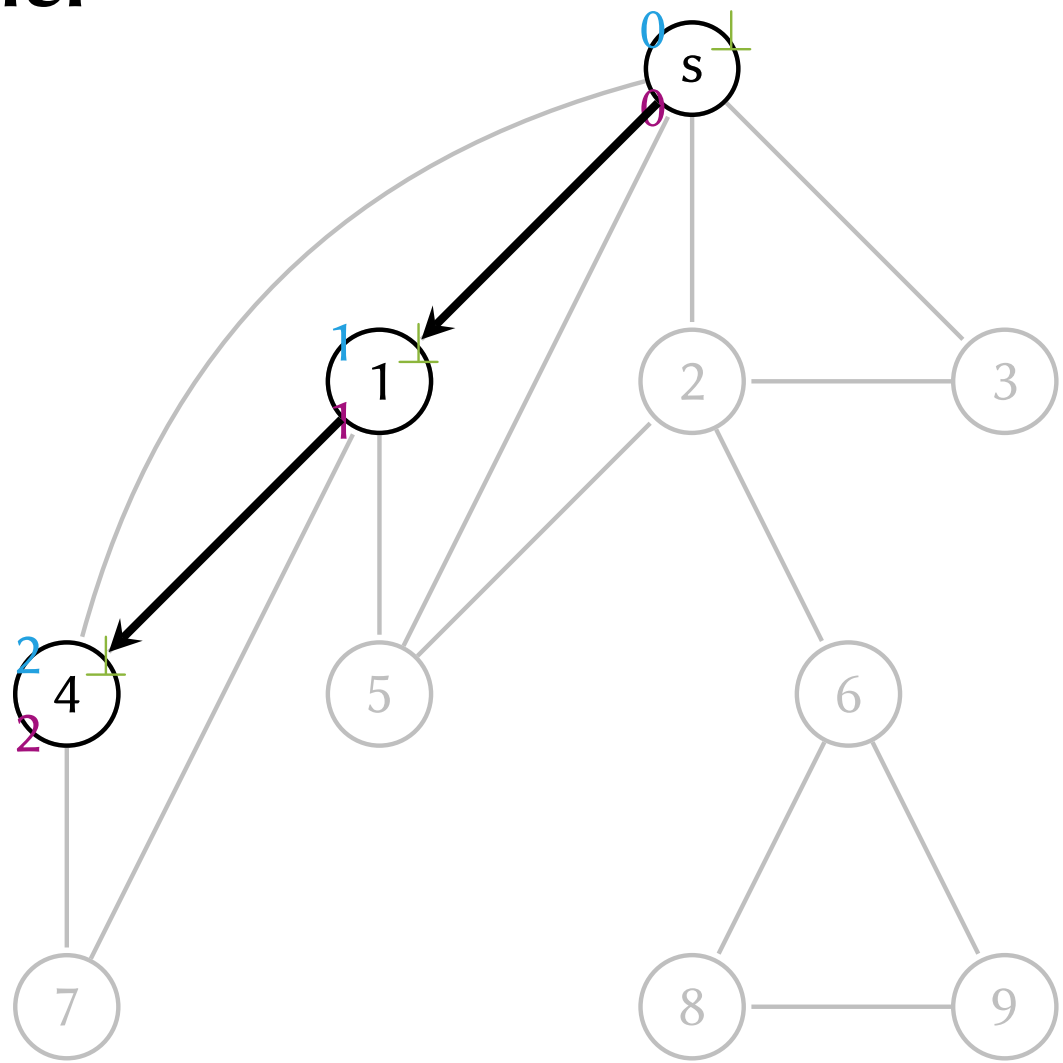
Kanten:

unknown

tree

backward

forward



verfolge Baum-Kante $(1, 4)$

DFS Beispiel

skip animation

Knoten:

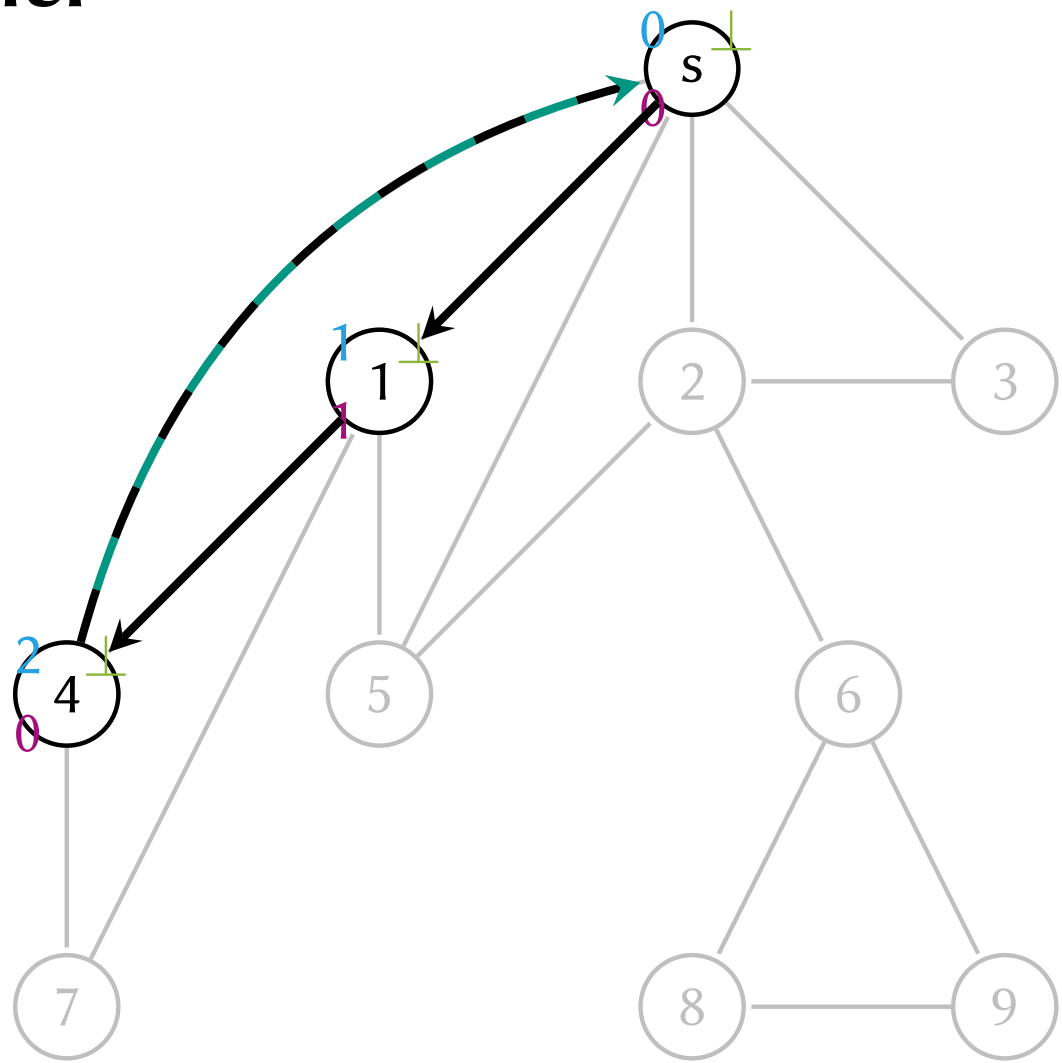
`dfsNum`

`finNum`

`lowNum`

Kanten:

- unknown
- tree
- backward
- forward



entdecke Rückwärts-Kante $(4, s)$, aktualisiere `lowNum`$[4]$ zu 0

DFS Beispiel

Karlsruher Institut für Technologie

skip animation

Knoten:

dfsNum

finNum

lowNum

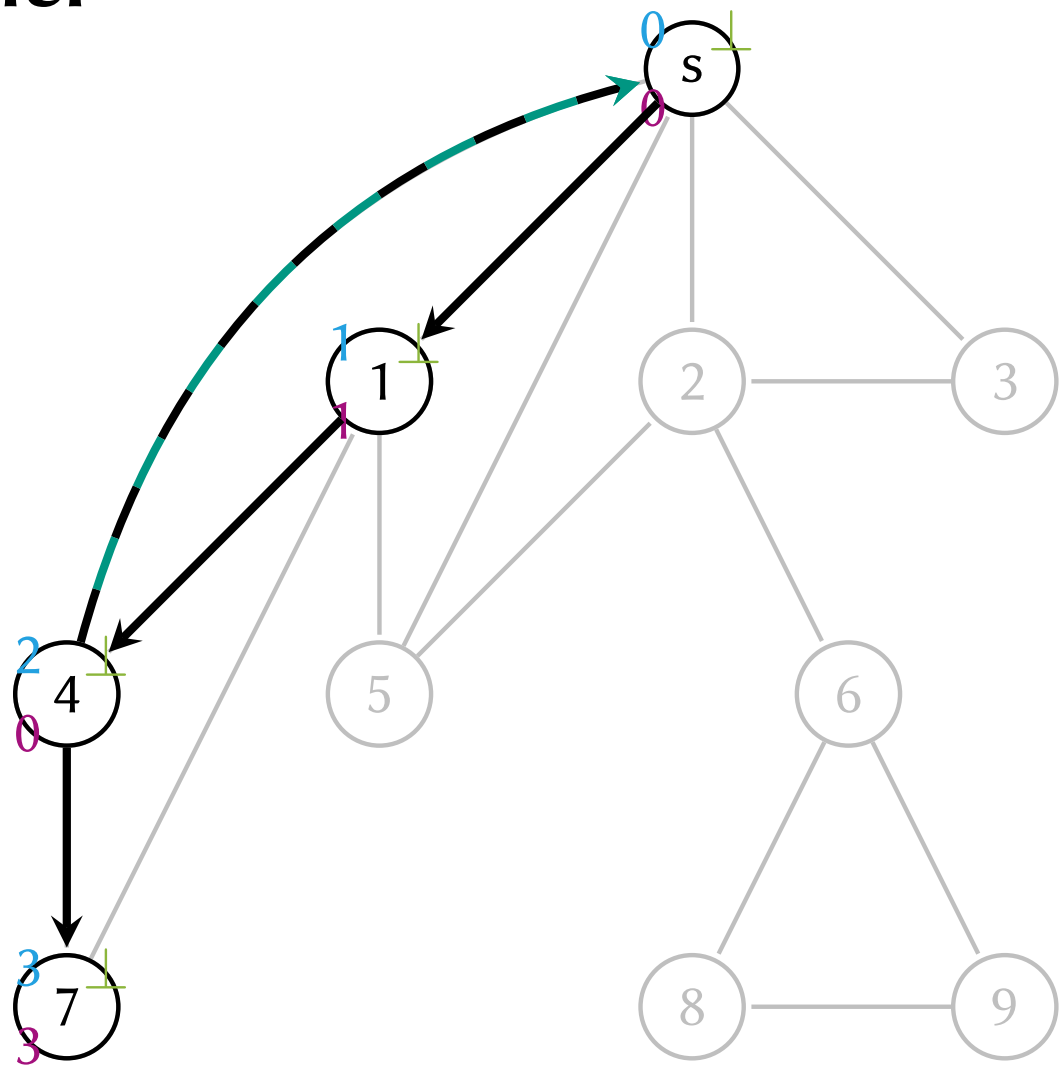
Kanten:

unknown

tree

backward

forward



verfolge Baum-Kante (4, 7)

DFS Beispiel

skip animation

Knoten:

dfsNum

finNum

lowNum

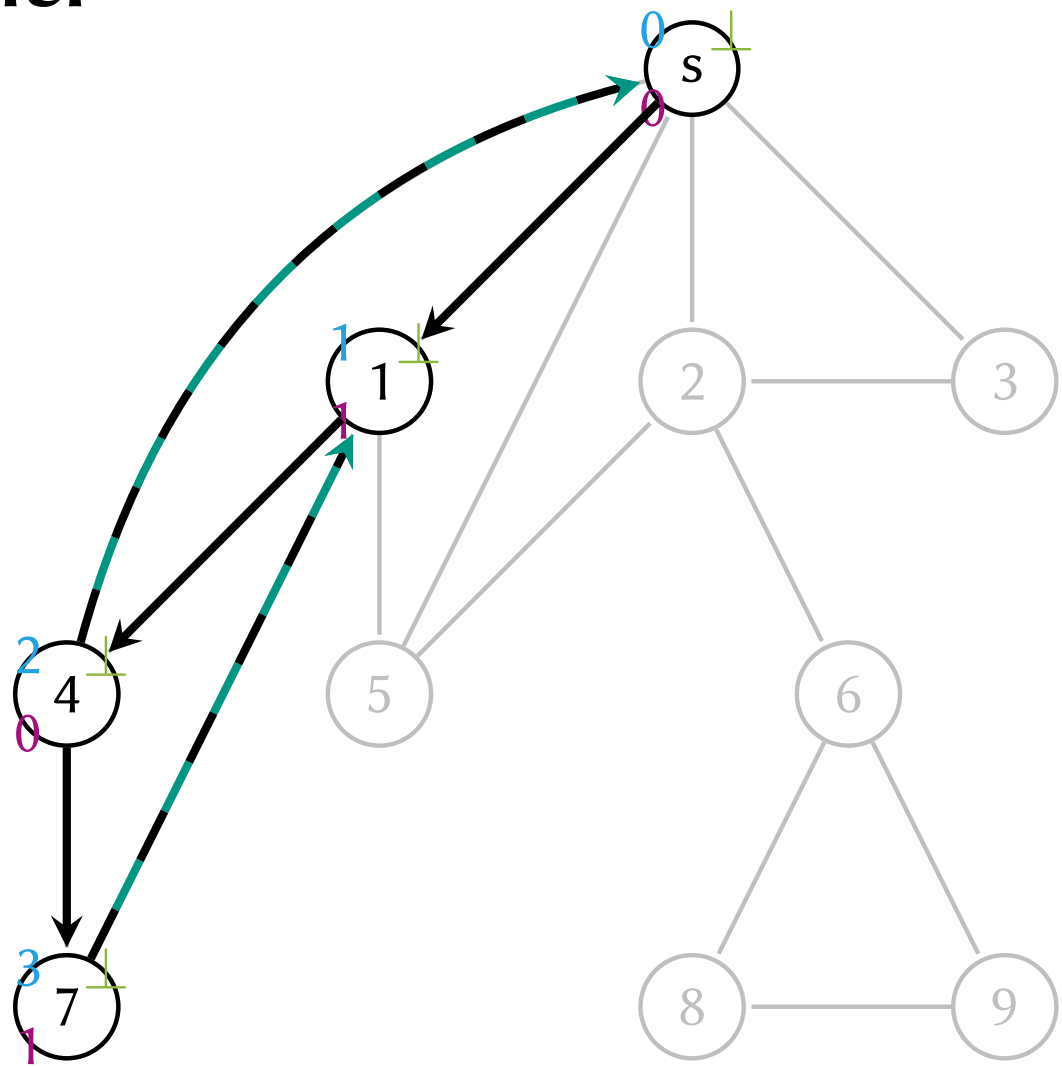
Kanten:

unknown

tree

backward

forward



entdecke Rückwärts-Kante (7, 1), aktualisiere `lowNum`[7] zu 1

DFS Beispiel

skip animation

Knoten:

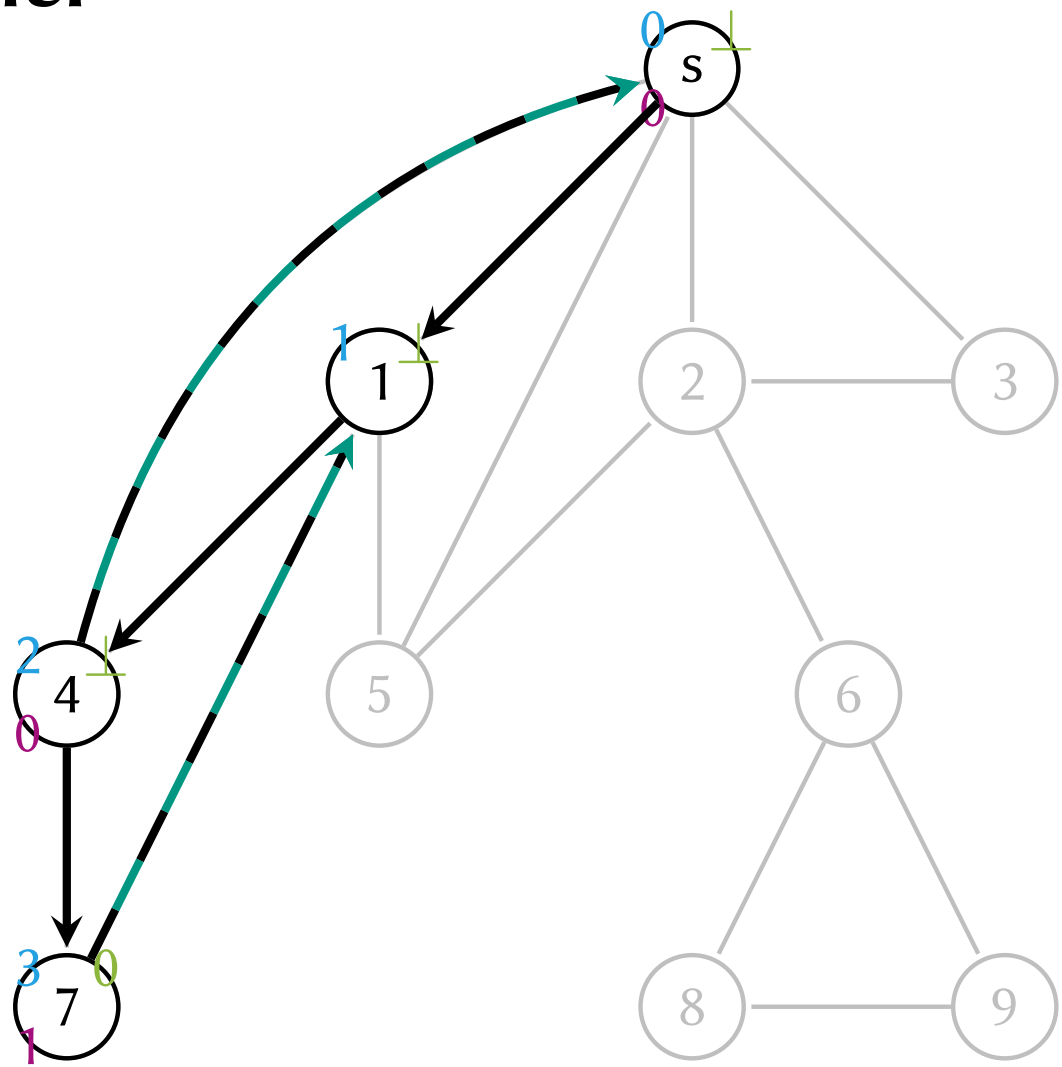
`dfsNum`

`finNum`

`lowNum`

Kanten:

- unknown
- tree
- backward
- forward

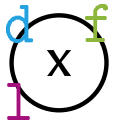


finalisiere 7, gehe zu Parent 4, aktualisiere `finNum[7]` zu 0

DFS Beispiel

skip animation

Knoten:



dfsNum

finNum

lowNum

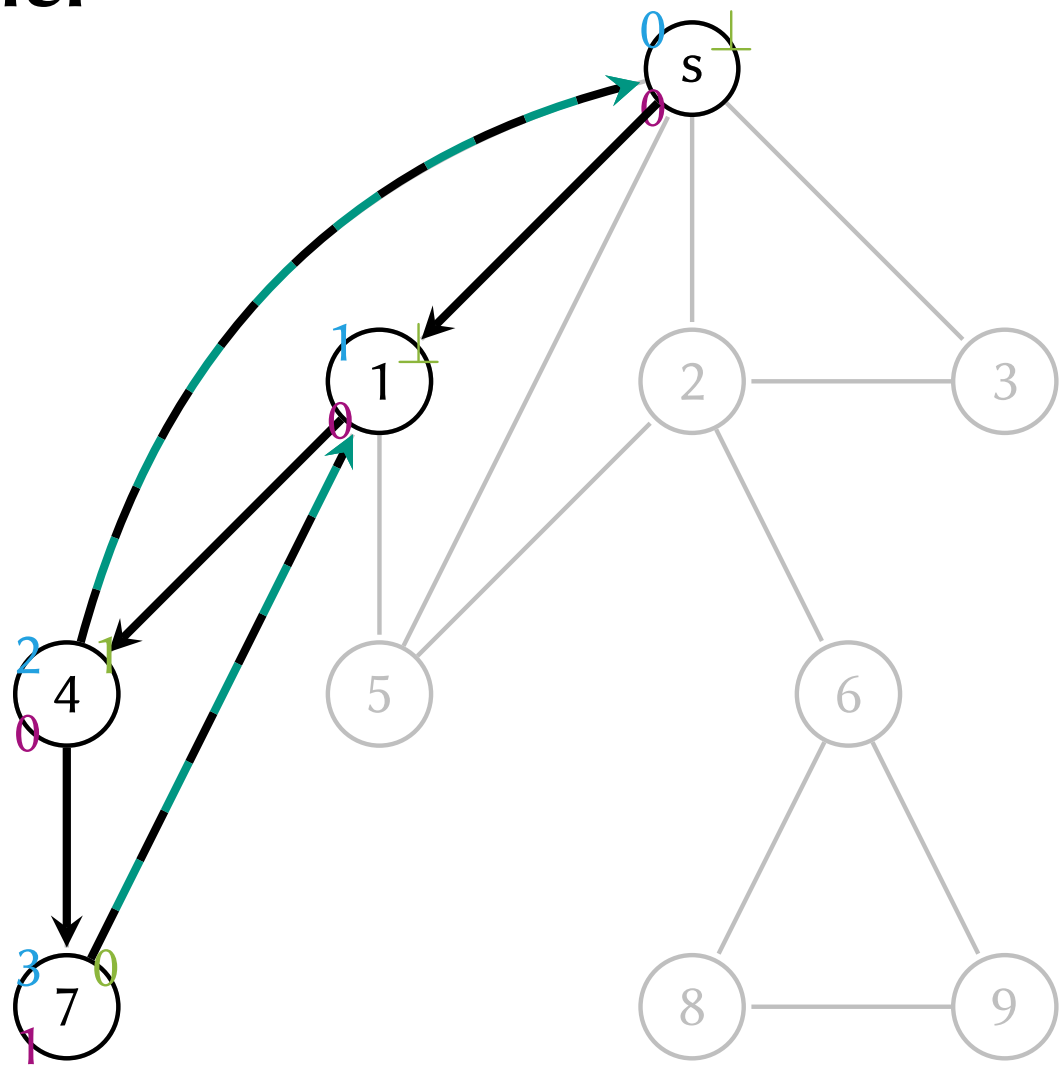
Kanten:

← unknown

← tree

← backward

← forward



finalisiere 4, gehe zu Parent 1, aktualisiere `finNum[4]` zu 1

DFS Beispiel

Karlsruher Institut für Technologie

skip animation

Knoten:

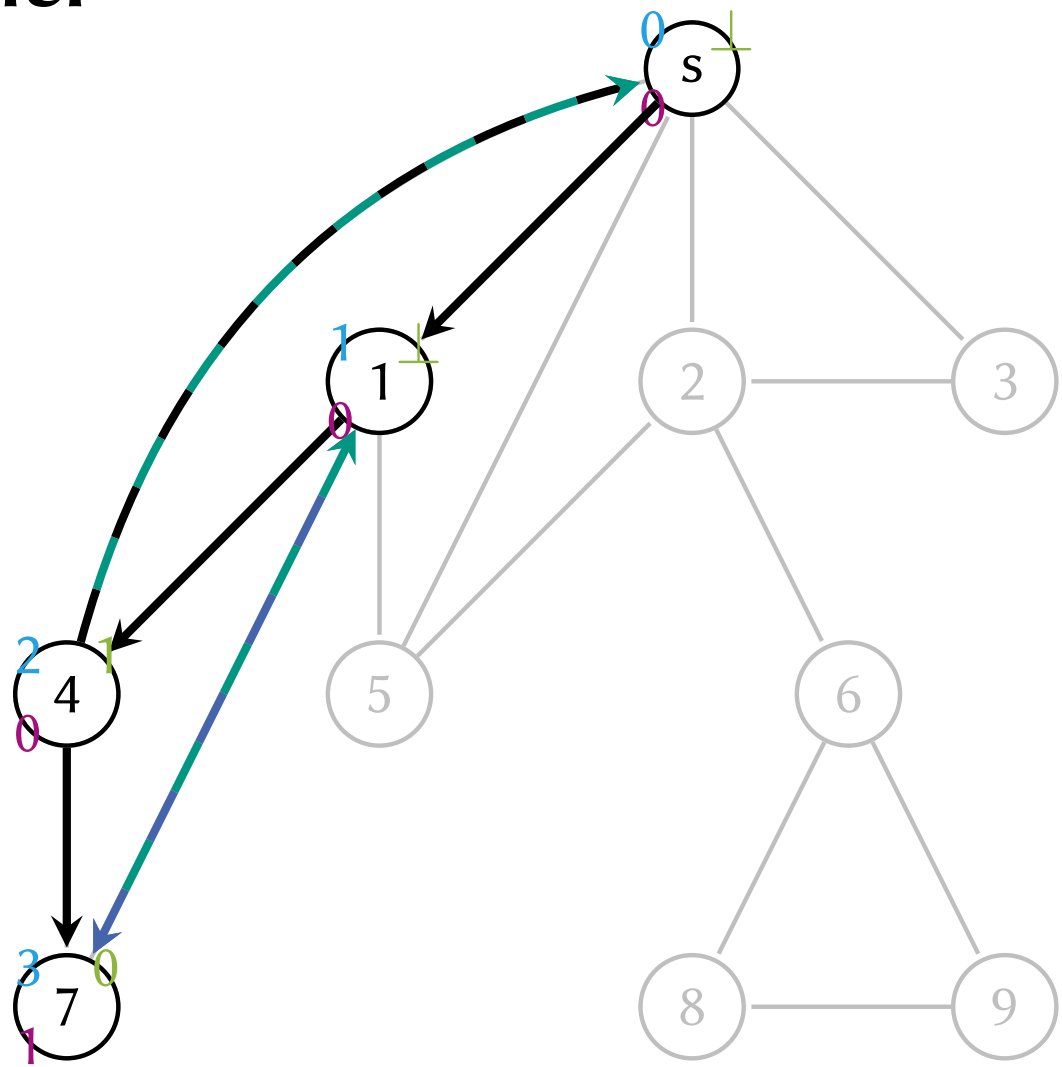
dfsNum

finNum

lowNum

Kanten:

- unknown
- tree
- backward
- forward



entdecke Vorwärts-Kante $(1, 7)$

DFS Beispiel

skip animation

Knoten:

dfsNum

finNum

lowNum

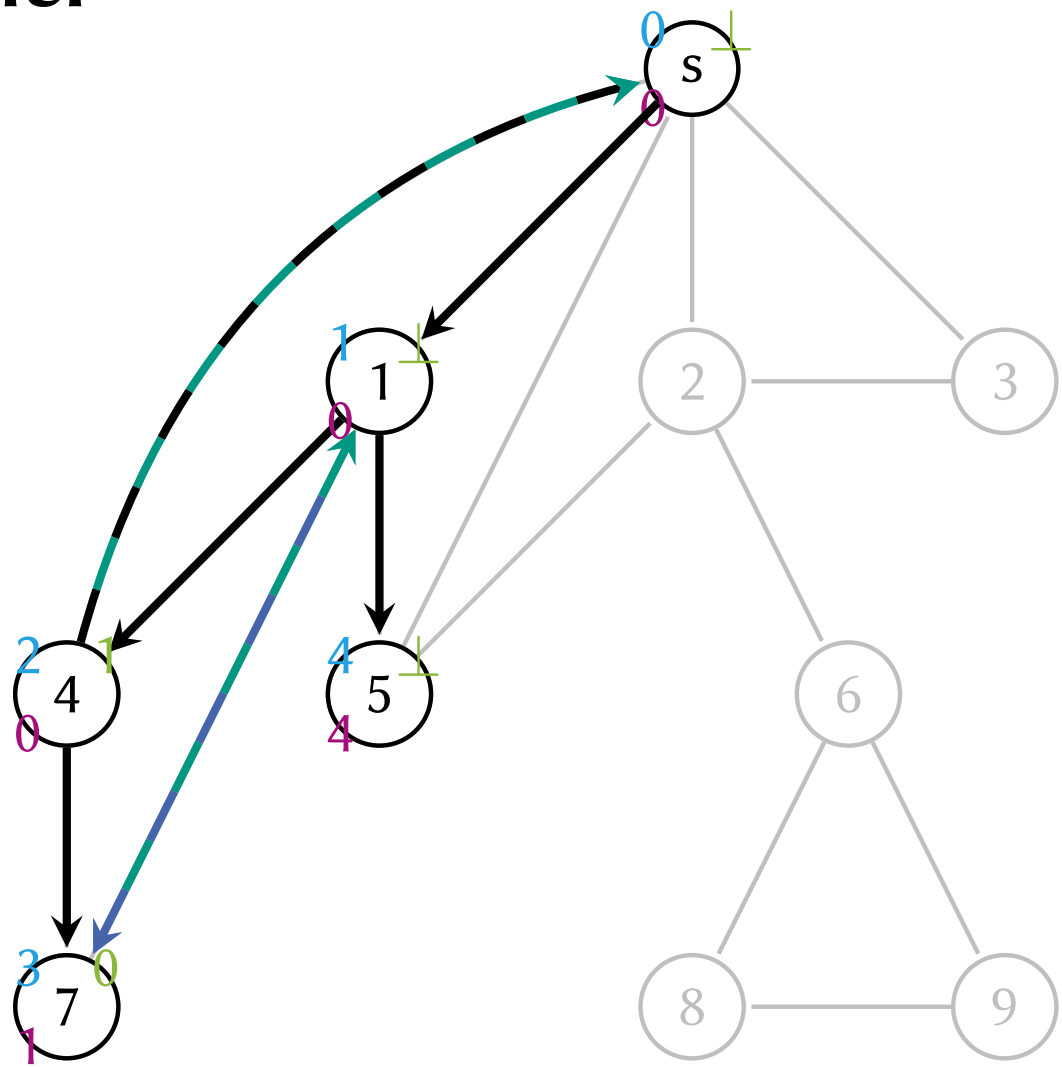
Kanten:

unknown

tree

backward

forward



verfolge Baum-Kante $(1, 5)$

DFS Beispiel

skip animation

Knoten:

dfsNum

finNum

lowNum

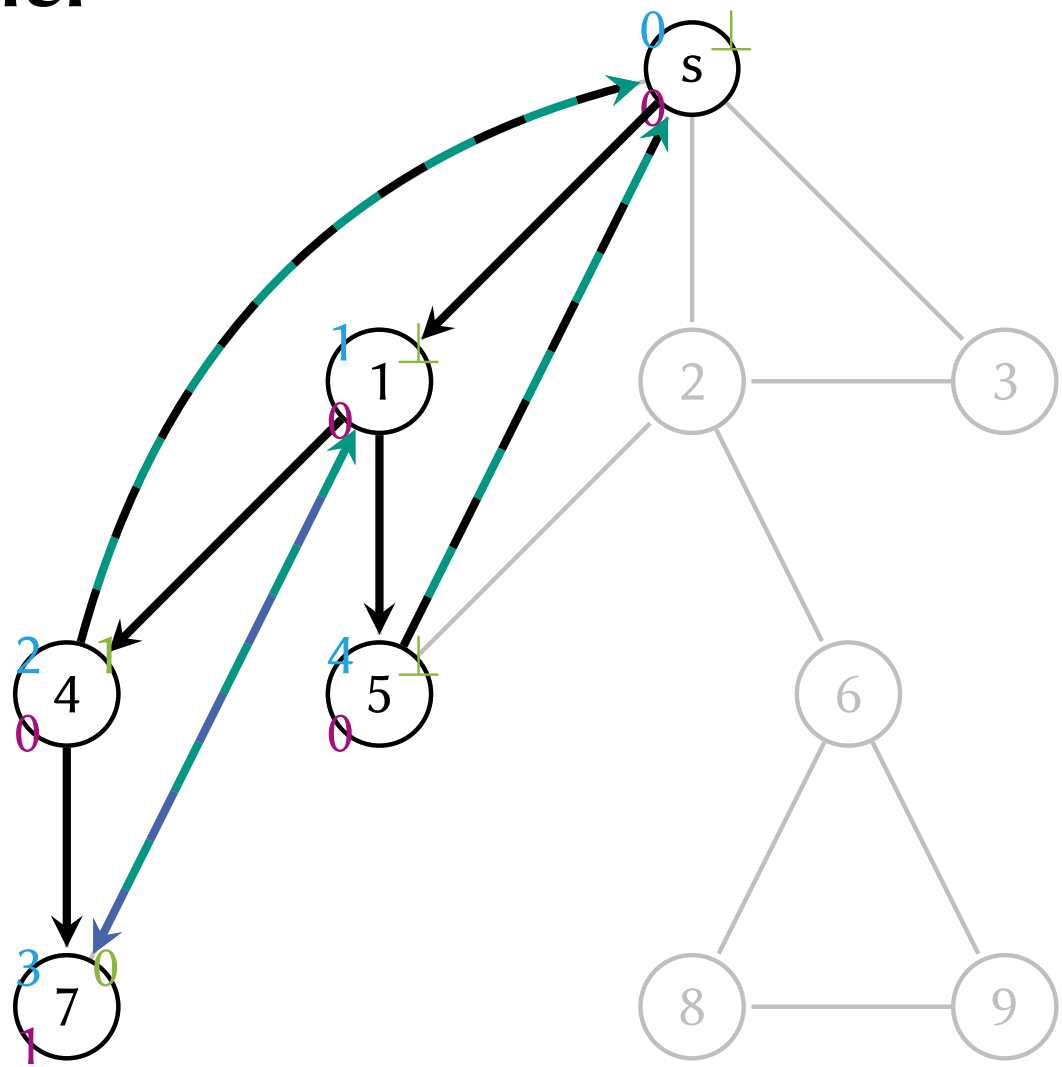
Kanten:

unknown

tree

backward

forward



entdecke Rückwärts-Kante $(5, s)$, aktualisiere `lowNum`$[5]$ zu 0

DFS Beispiel

Knoten:

dfsNum

finNum

lowNum

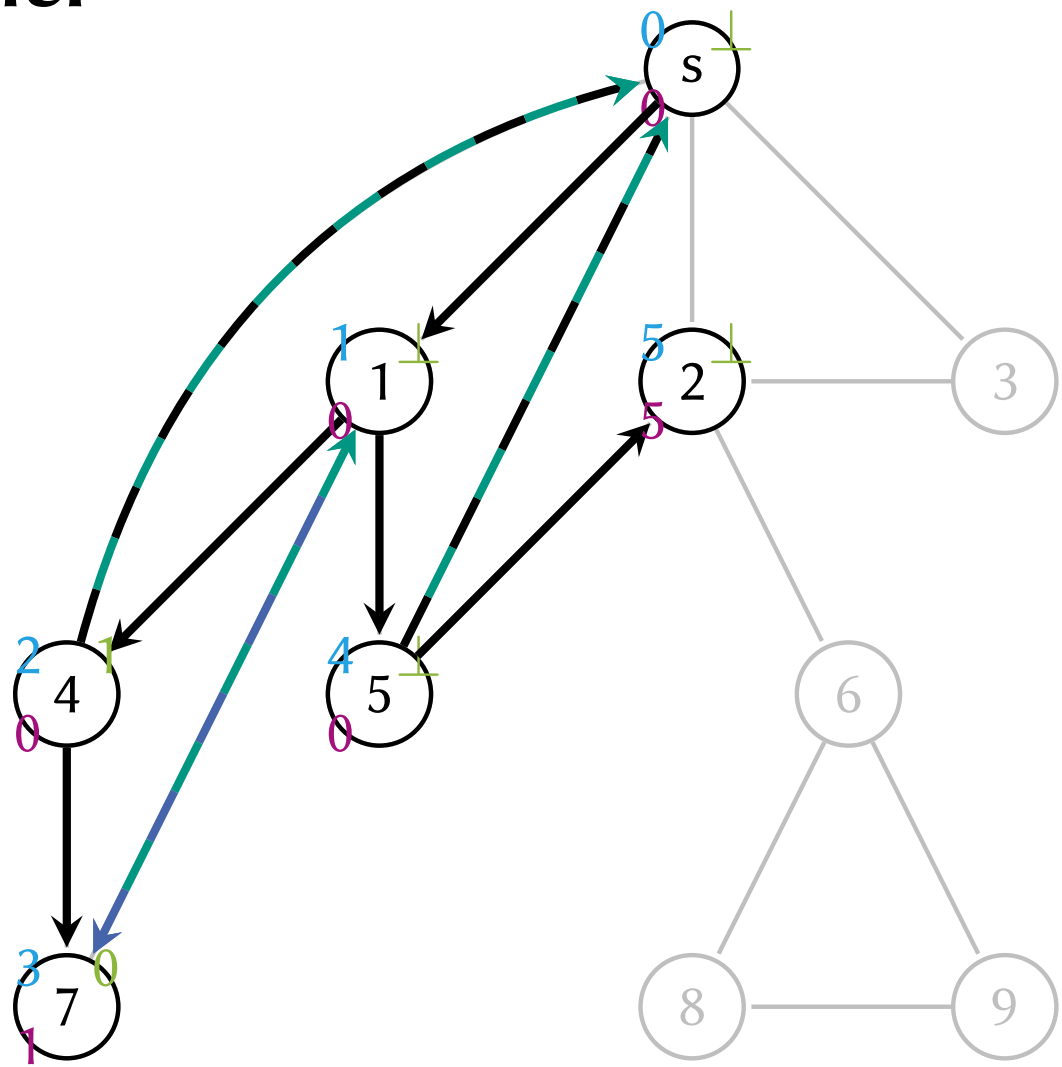
Kanten:

unknown

tree

backward

forward



skip animation

verfolge Baum-Kante (5, 2)

DFS Beispiel

Knoten:

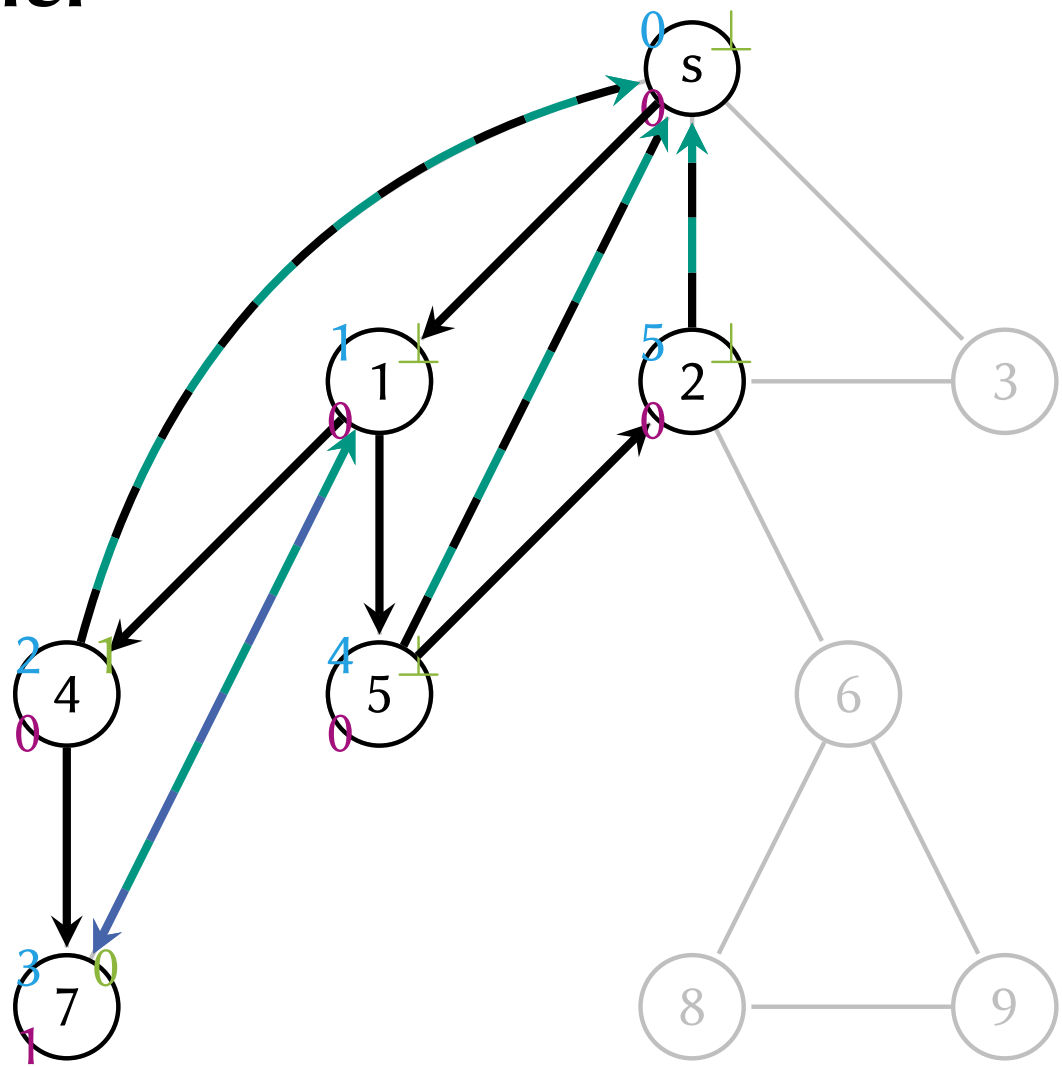
`dfsNum`

`finNum`

`lowNum`

Kanten:

- unknown
- tree
- backward
- forward



skip animation

entdecke Rückwärts-Kante $(2, s)$, aktualisiere `lowNum`[2] zu 0

DFS Beispiel

skip animation

Knoten:

dfsNum

finNum

lowNum

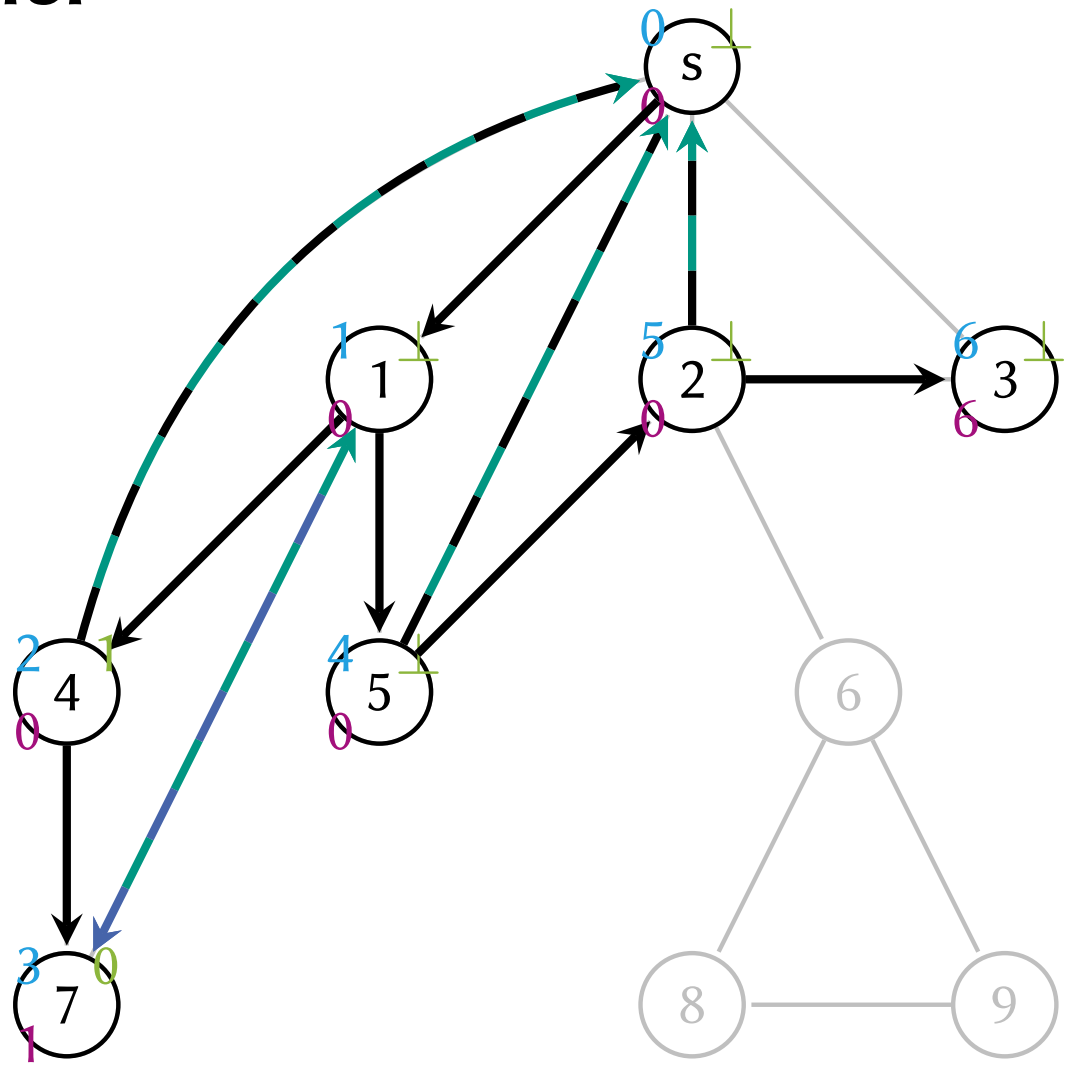
Kanten:

unknown

tree

backward

forward



verfolge Baum-Kante (2, 3)

DFS Beispiel

Karlsruher Institut für Technologie

skip animation

Knoten:

dfsNum

finNum

lowNum

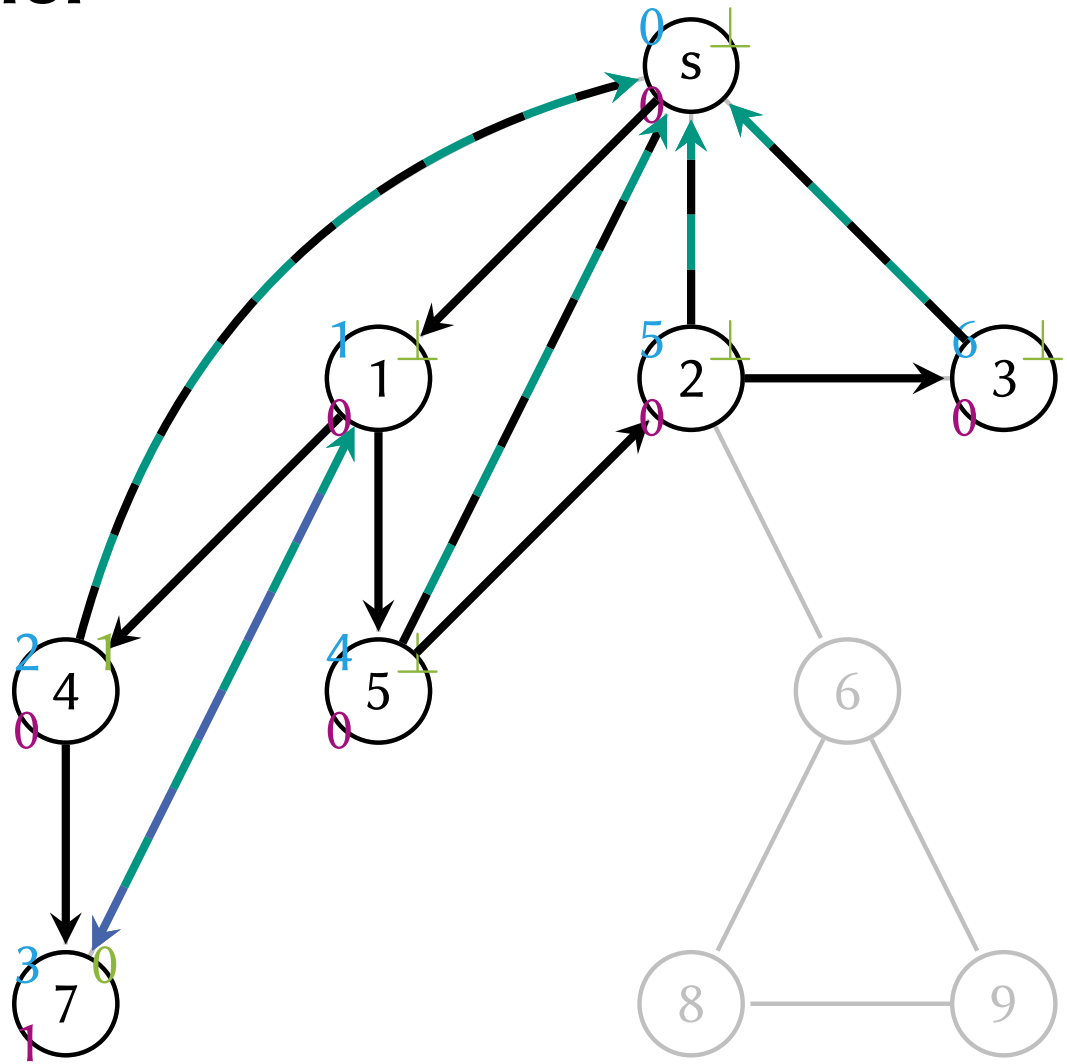
Kanten:

unknown

tree

backward

forward



entdecke Rückwärts-Kante $(3, s)$, aktualisiere `lowNum`$[3]$ zu 0

DFS Beispiel

skip animation

Knoten:

dfsNum

finNum

lowNum

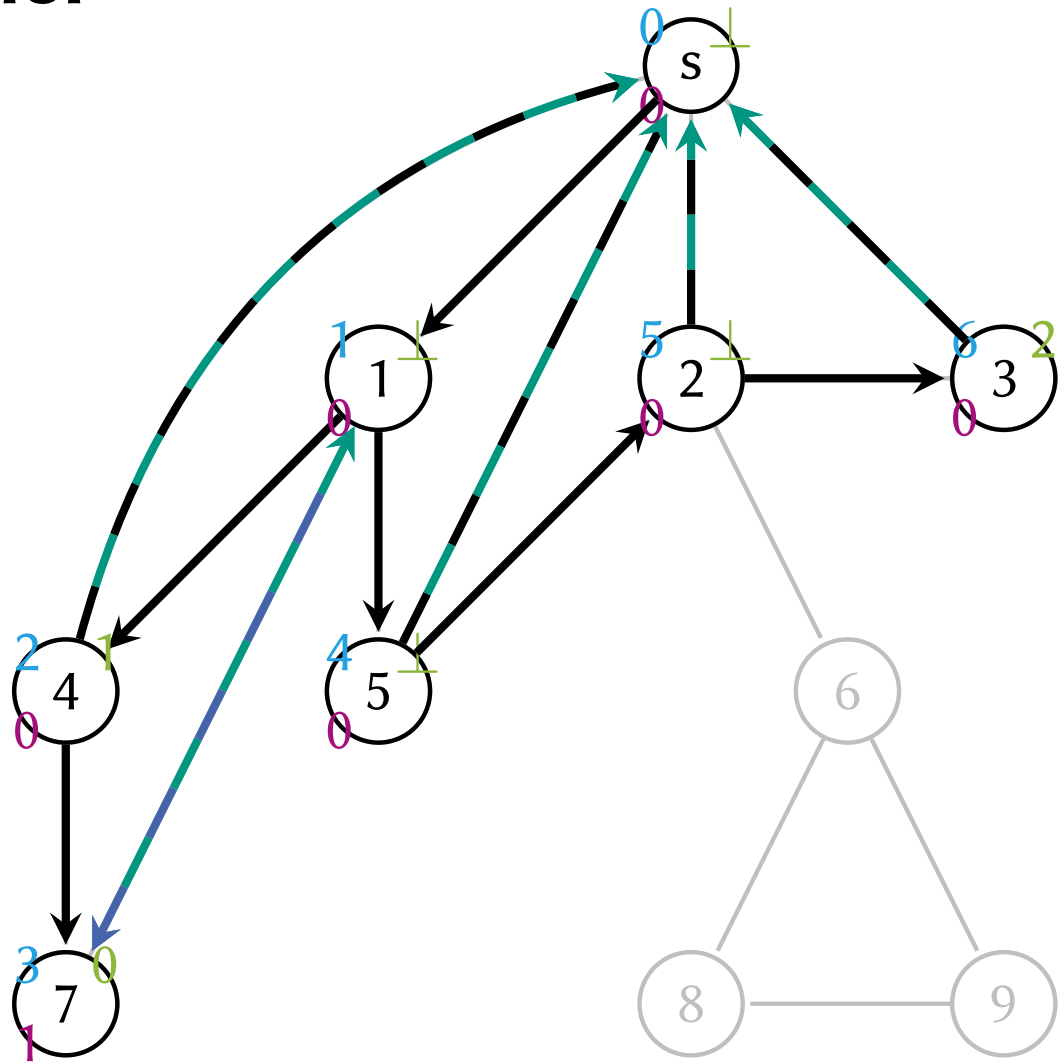
Kanten:

unknown

tree

backward

forward



finalisiere 3, gehe zu Parent 2, aktualisiere `finNum[3]` zu 2

DFS Beispiel

Karlsruher Institut für Technologie

skip animation

Knoten:

dfsNum

finNum

lowNum

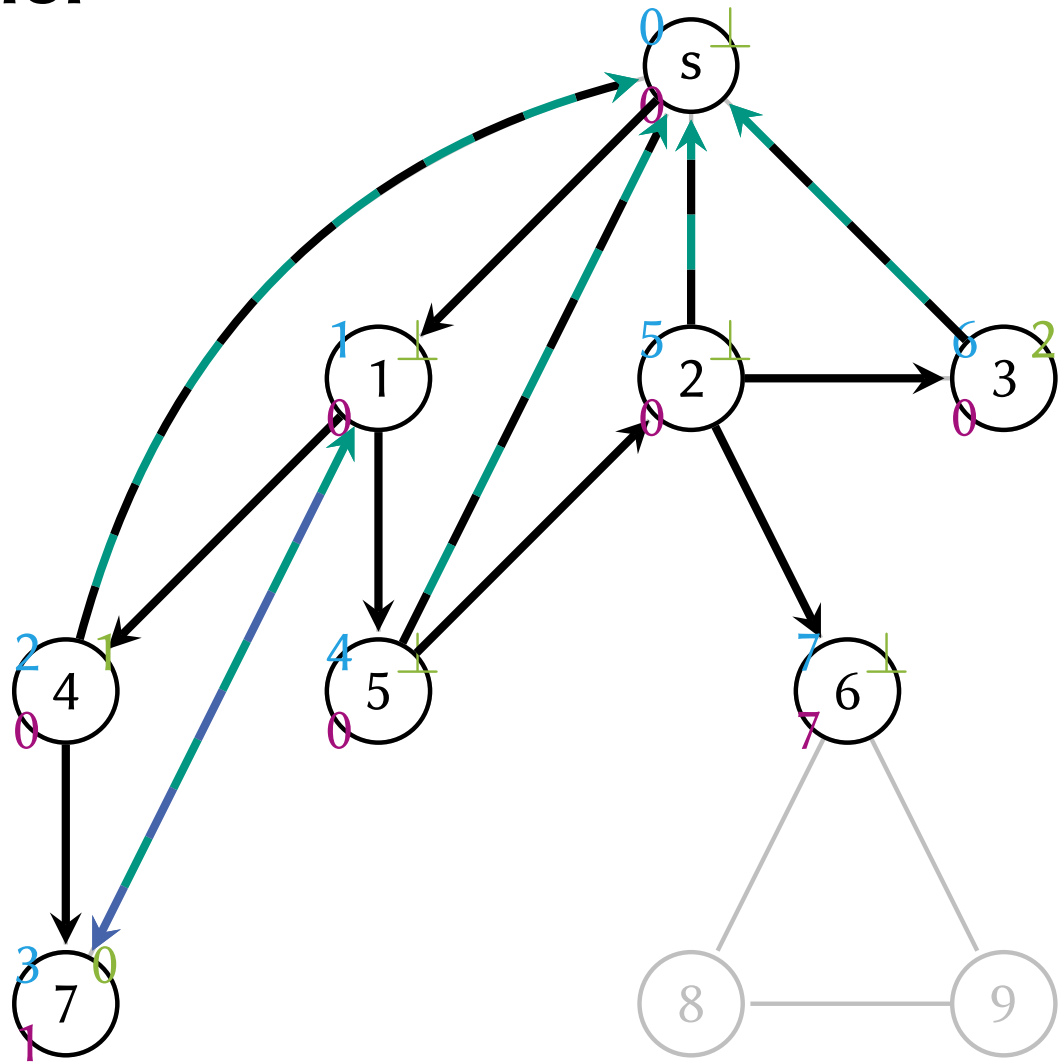
Kanten:

unknown

tree

backward

forward



verfolge Baum-Kante $(2, 6)$

DFS Beispiel

Knoten:

dfsNum

finNum

lowNum

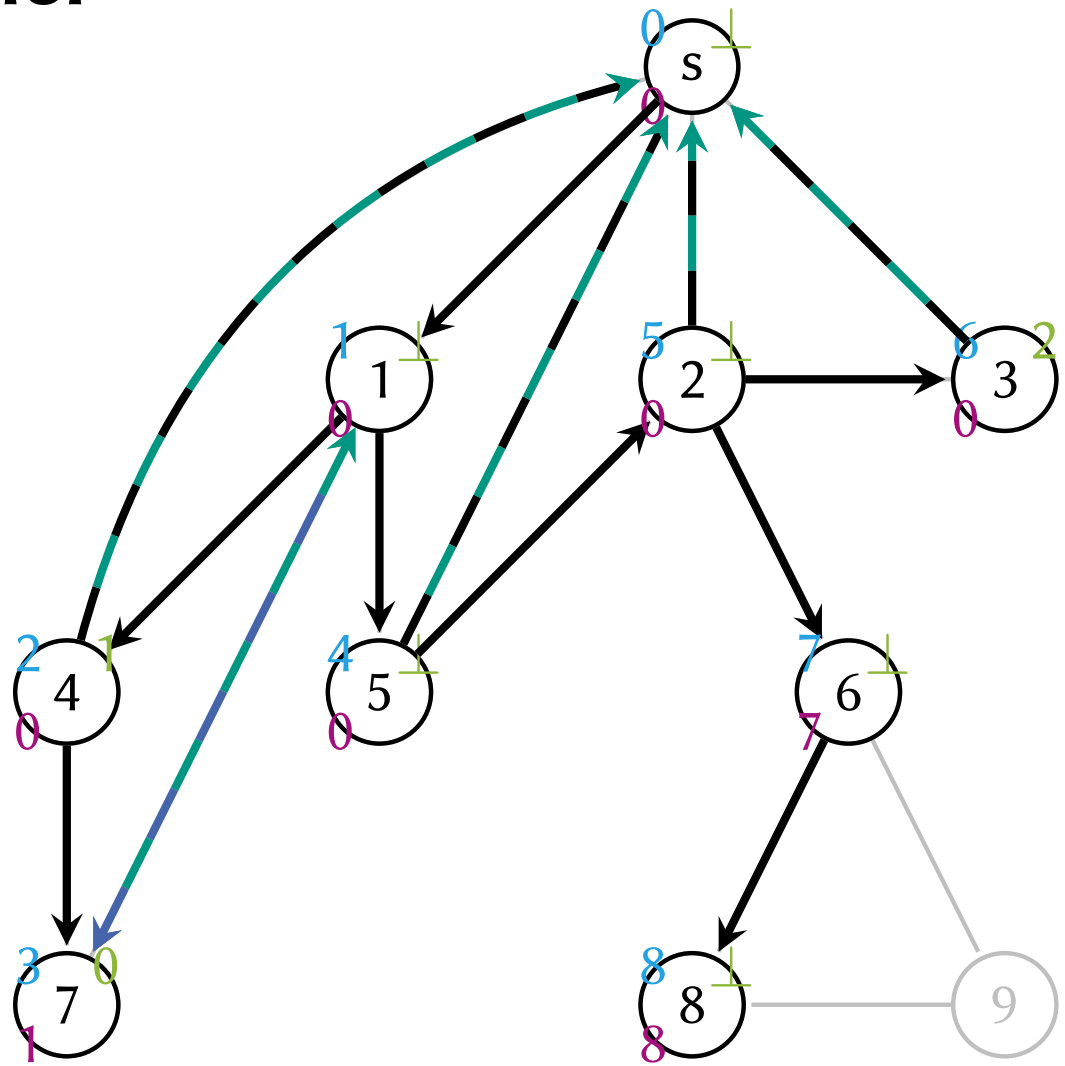
Kanten:

unknown

tree

backward

forward



verfolge Baum-Kante $(6, 8)$

DFS Beispiel

Knoten:

dfsNum

finNum

lowNum

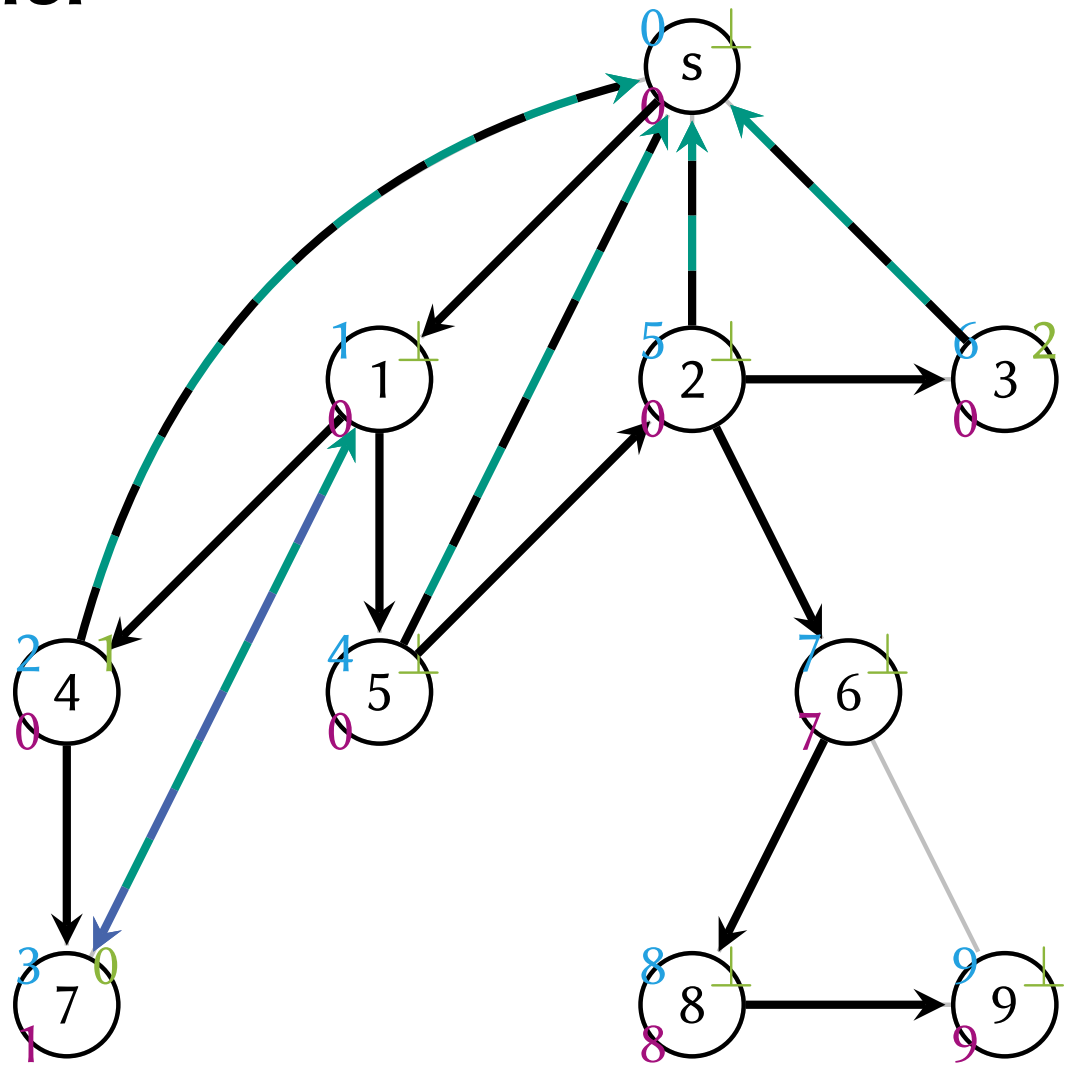
Kanten:

unknown

tree

backward

forward



verfolge Baum-Kante $(8, 9)$

DFS Beispiel

skip animation

Knoten:

dfsNum

finNum

lowNum

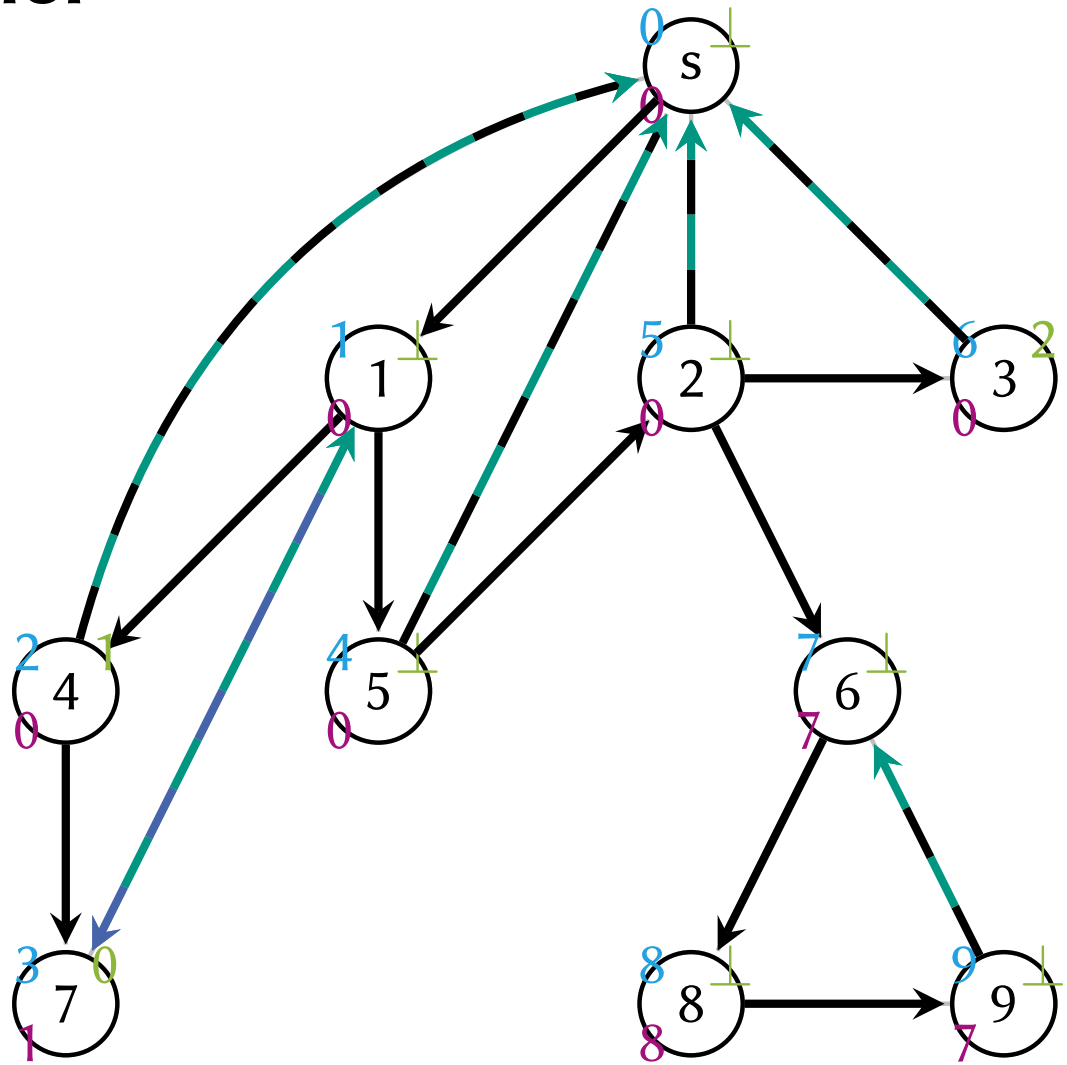
Kanten:

← unknown

← tree

← backward

← forward



entdecke Rückwärts-Kante $(9, 6)$, aktualisiere `lowNum`$[9]$ zu 7

DFS Beispiel

skip animation

Knoten:

dfsNum

finNum

lowNum

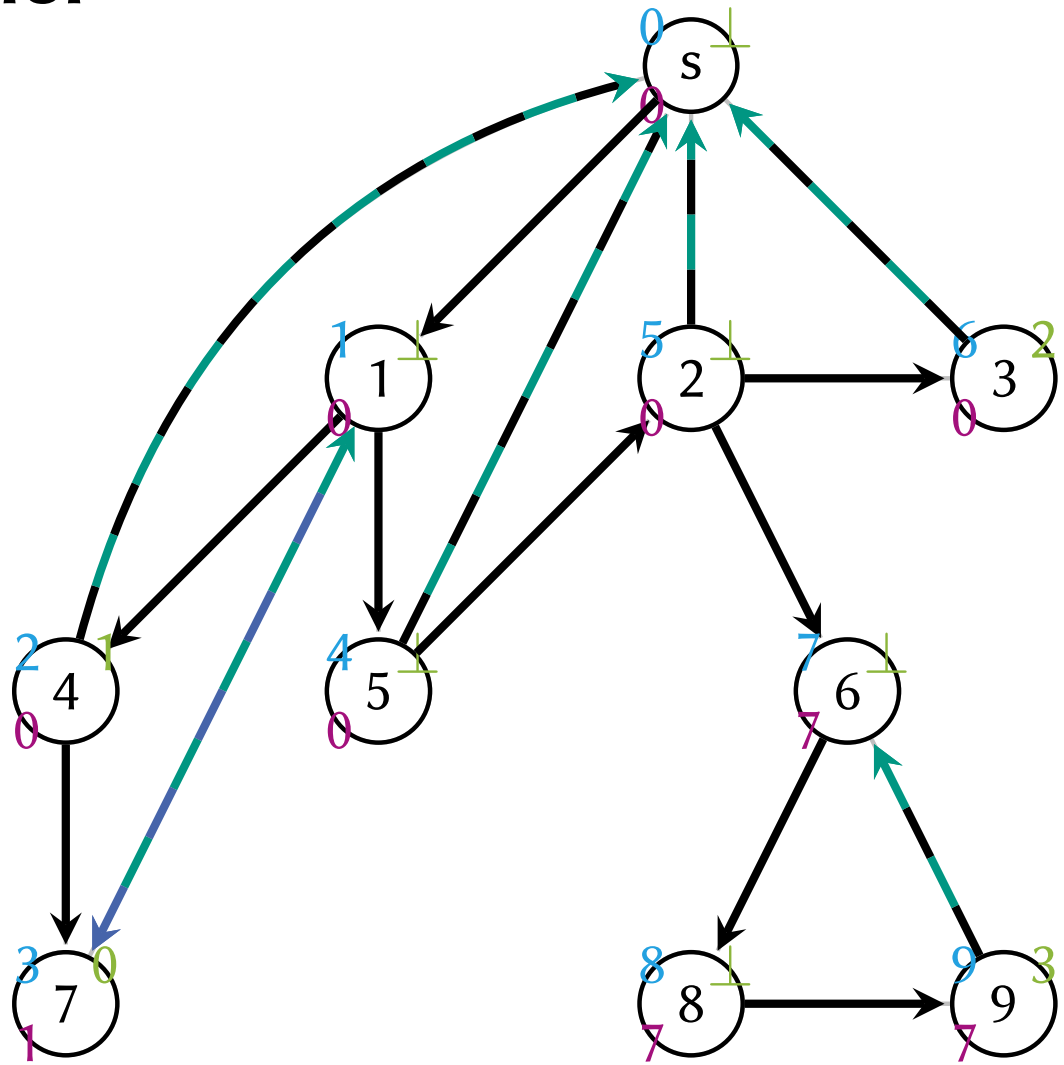
Kanten:

← unknown

← tree

← backward

← forward



finalisiere 9, gehe zu Parent 8, aktualisiere `finNum`[9] zu 3, `lowNum`[8] zu 7

DFS Beispiel

skip animation

Knoten:

dfsNum

finNum

lowNum

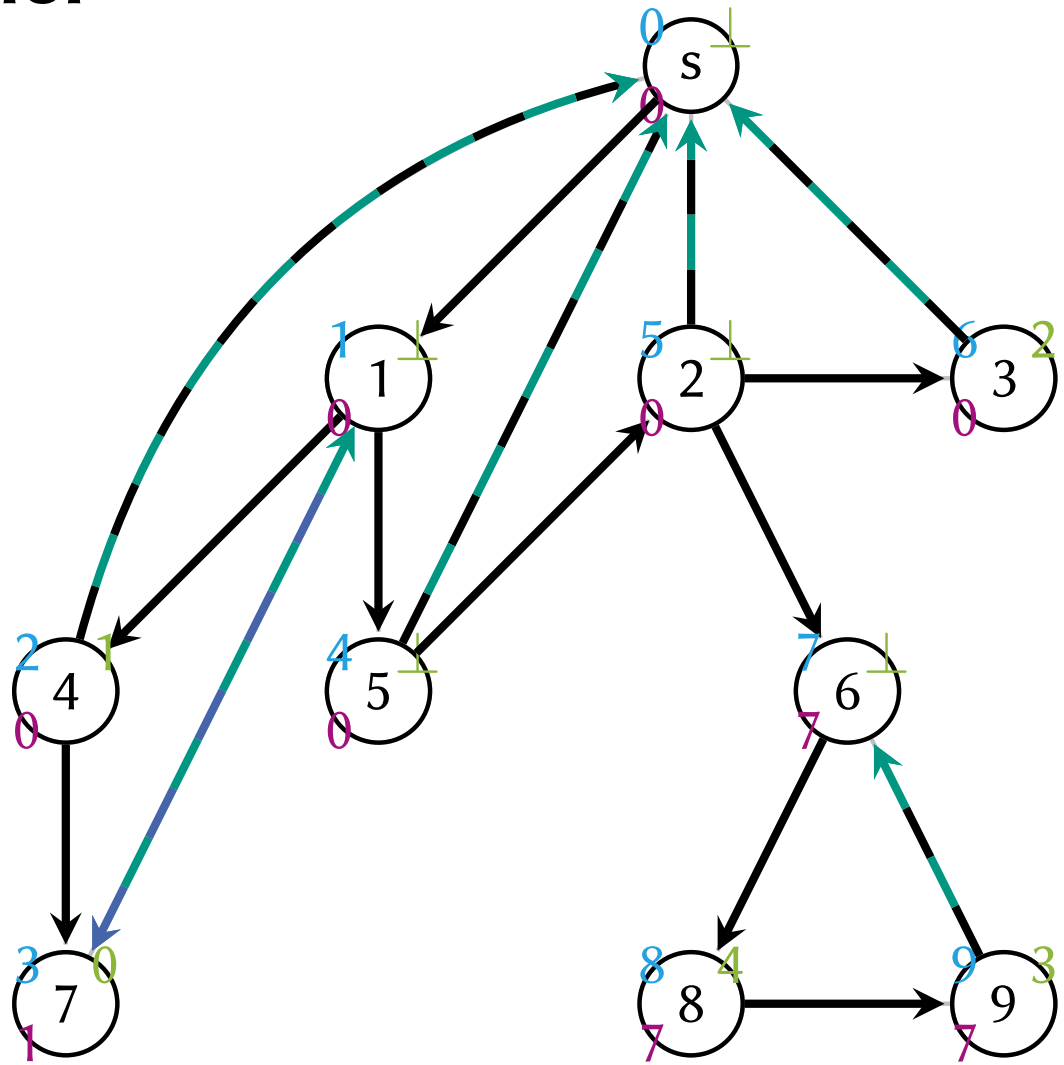
Kanten:

← unknown

← tree

← backward

← forward



finalisiere 8, gehe zu Parent 6, aktualisiere `finNum`[8] zu 4

DFS Beispiel

Knoten:

dfsNum

finNum

lowNum

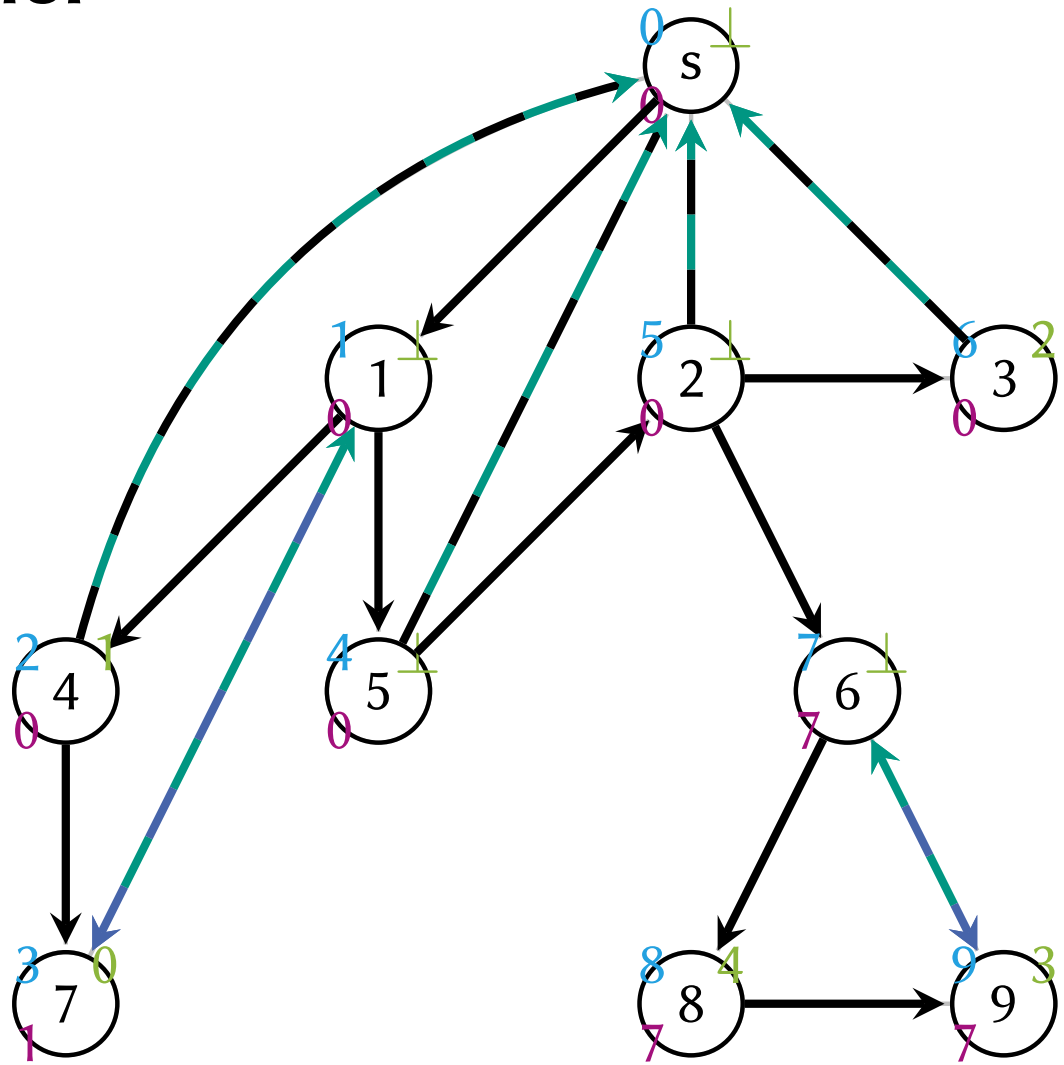
Kanten:

← unknown

← tree

← backward

← forward



skip animation

entdecke Vorwärts-Kante $(6, 9)$

DFS Beispiel

skip animation

Knoten:

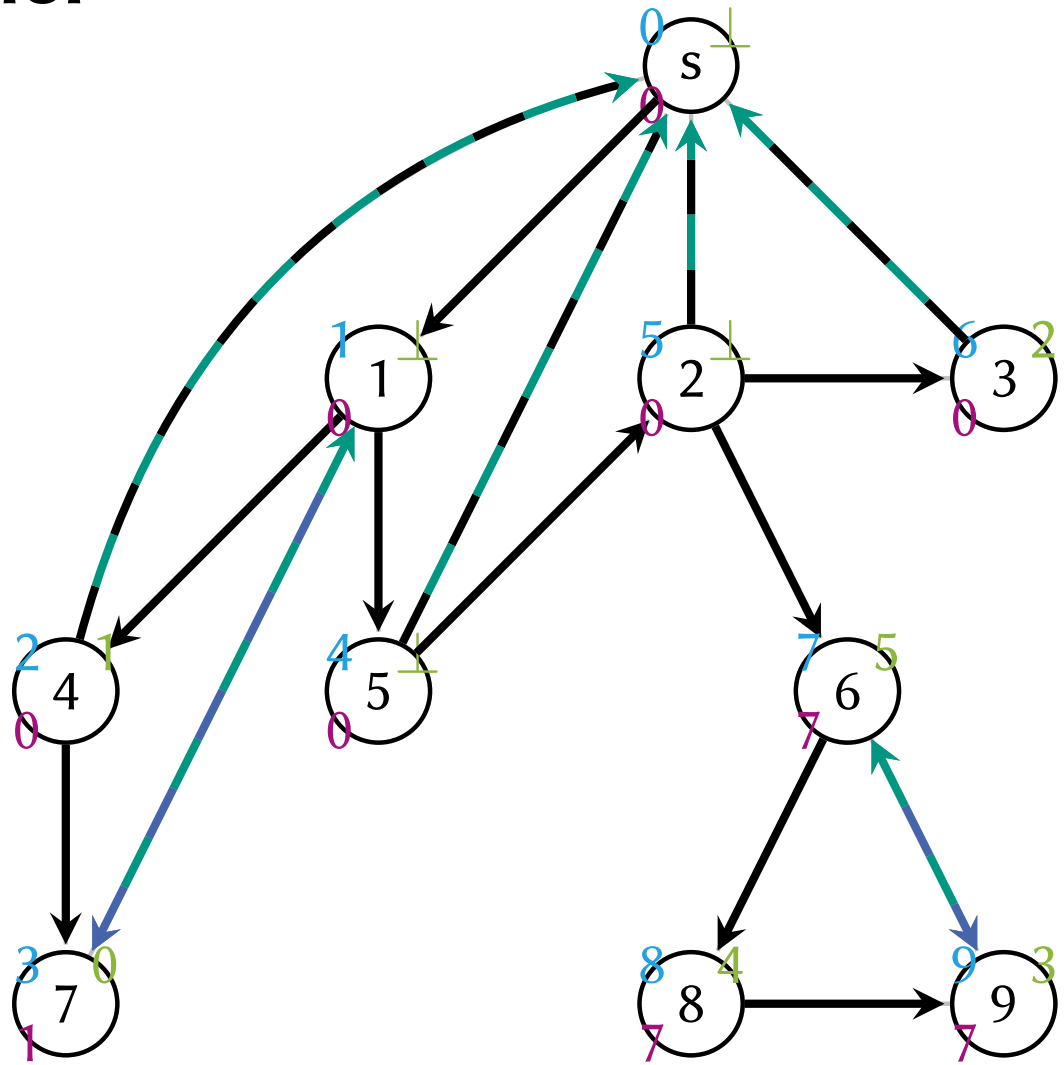
`dfsNum`

`finNum`

`lowNum`

Kanten:

- unknown
- tree
- backward
- forward



finalisiere 6, gehe zu Parent 2, aktualisiere `finNum`[6] zu 5

DFS Beispiel

skip animation

Knoten:

dfsNum

finNum

lowNum

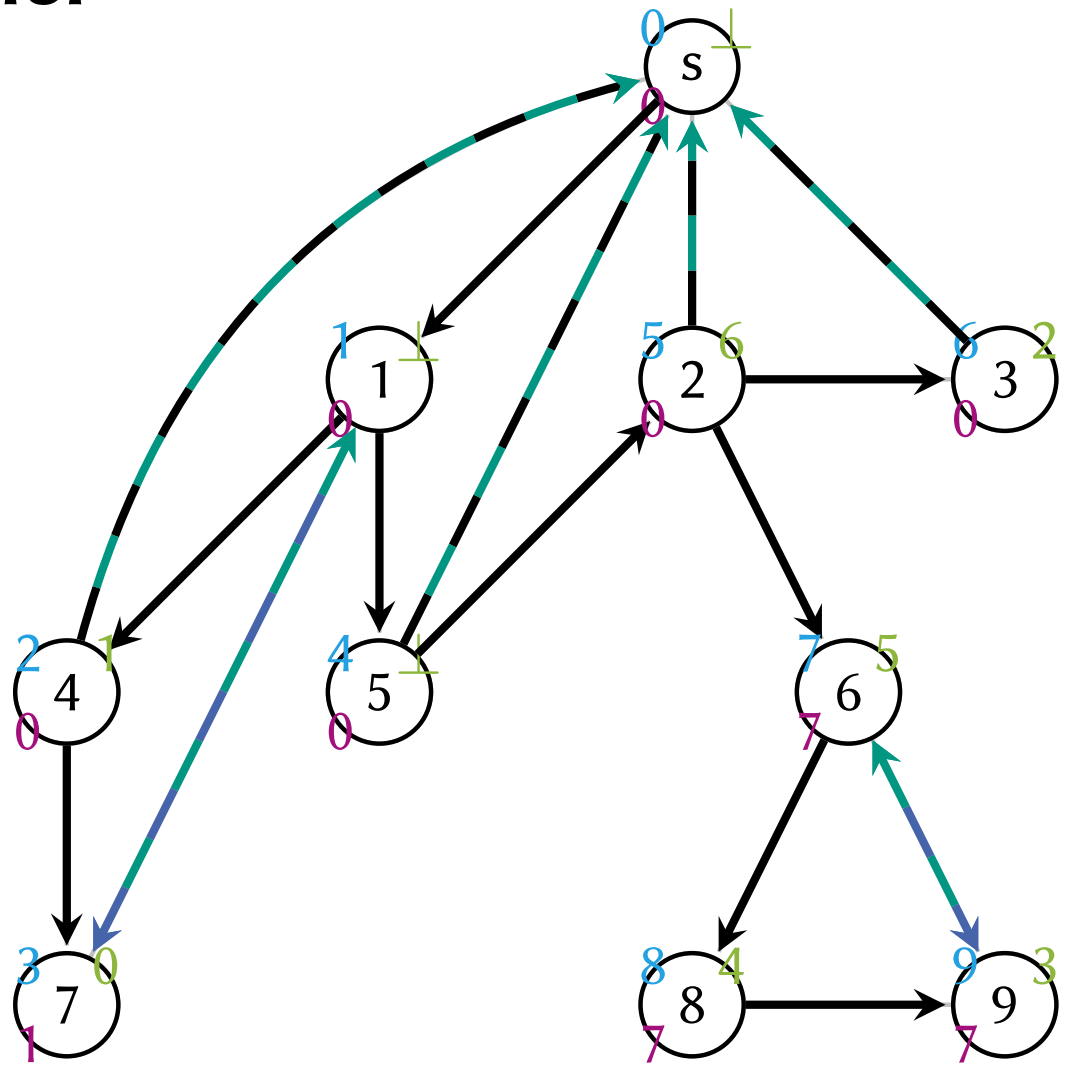
Kanten:

← unknown

← tree

← backward

← forward



finalisiere 5, gehe zu Parent 2, aktualisiere `finNum`[5] zu 6

DFS Beispiel

Karlsruher Institut für Technologie

skip animation

Knoten:

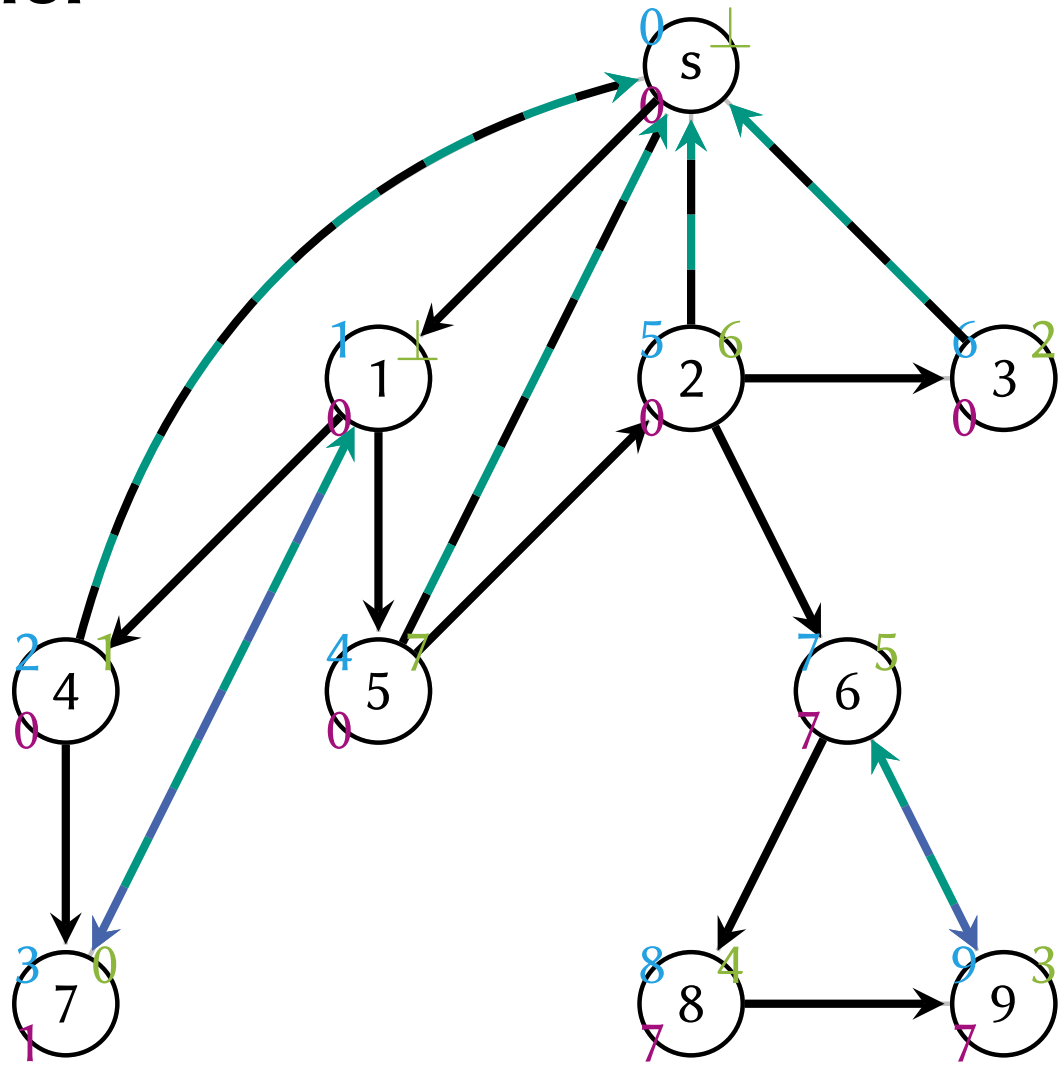
`dfsNum`

`finNum`

`lowNum`

Kanten:

- unknown
- tree
- backward
- forward



finalisiere 5, gehe zu Parent 1, aktualisiere `finNum`[5] zu 7

DFS Beispiel

Karlsruher Institut für Technologie

skip animation

Knoten:

dfsNum

finNum

lowNum

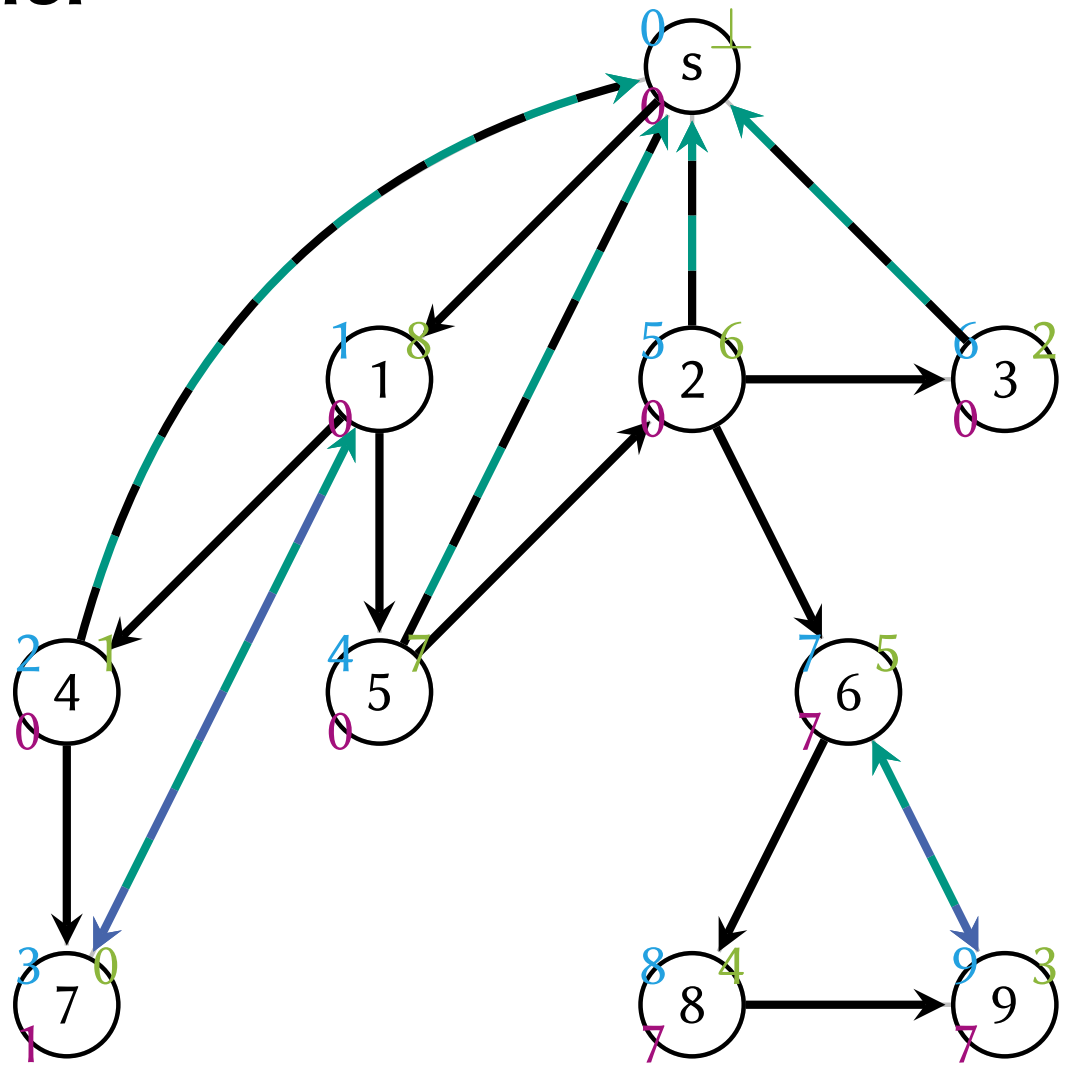
Kanten:

← unknown

← tree

← backward

← forward



finalisiere 1, gehe zu Parent *s*, aktualisiere `finNum[1]` zu 8

DFS Beispiel

Knoten:

dfsNum

finNum

lowNum

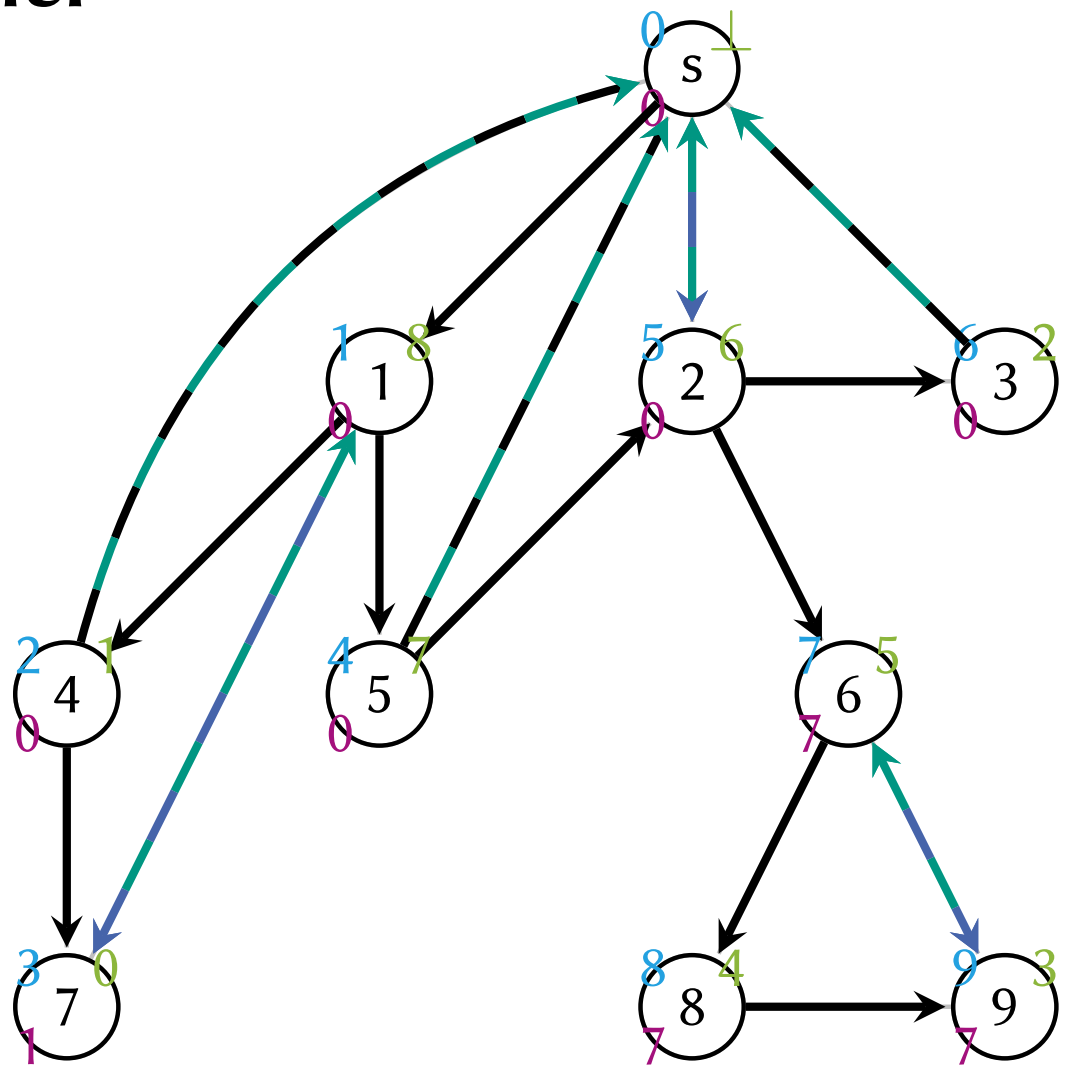
Kanten:

← unknown

← tree

← backward

← forward



entdecke Vorwärts-Kante $(s, 2)$

DFS Beispiel

Knoten:

dfsNum

finNum

lowNum

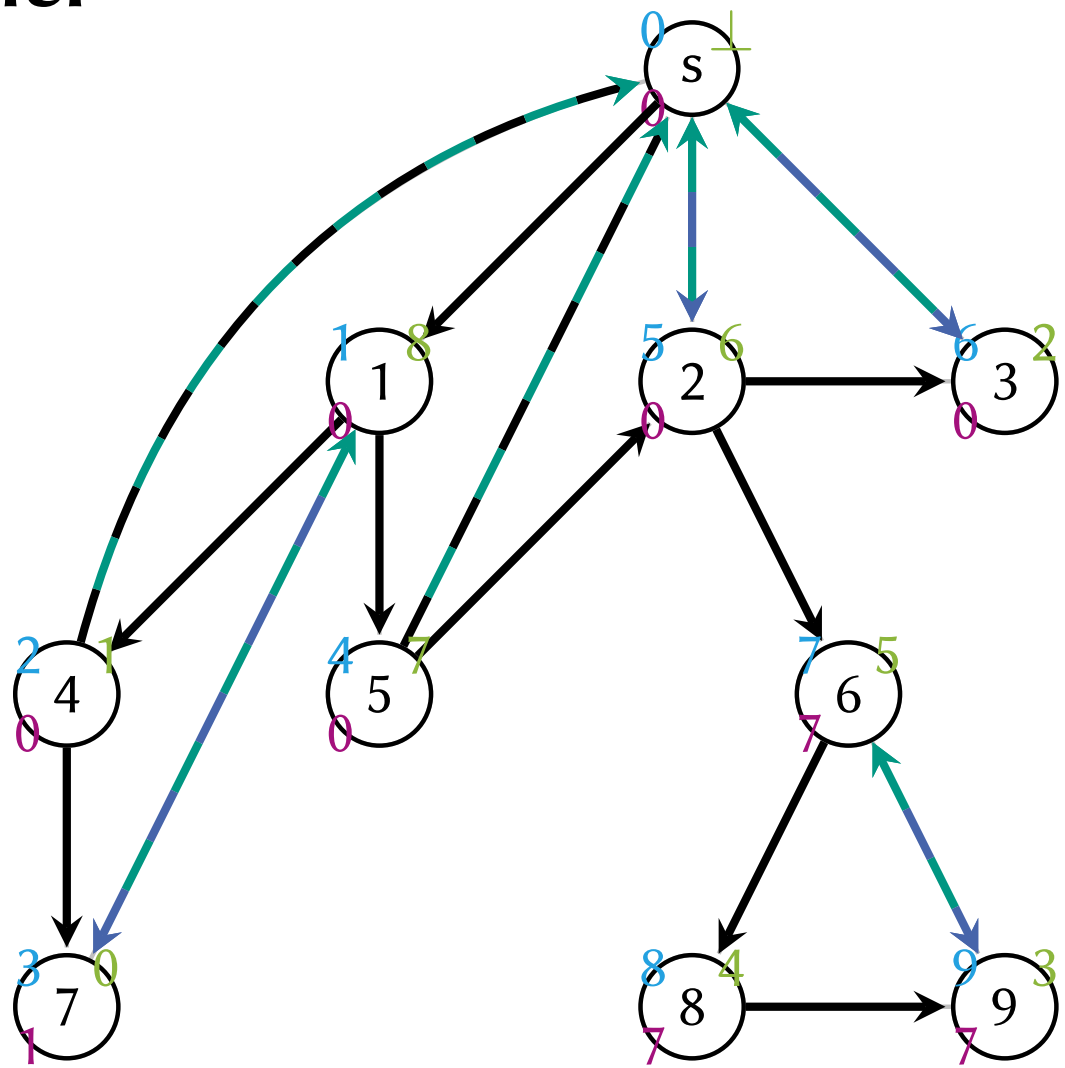
Kanten:

← unknown

← tree

← backward

← forward



skip animation

entdecke Vorwärts-Kante $(s, 3)$

DFS Beispiel

Knoten:

dfsNum

finNum

lowNum

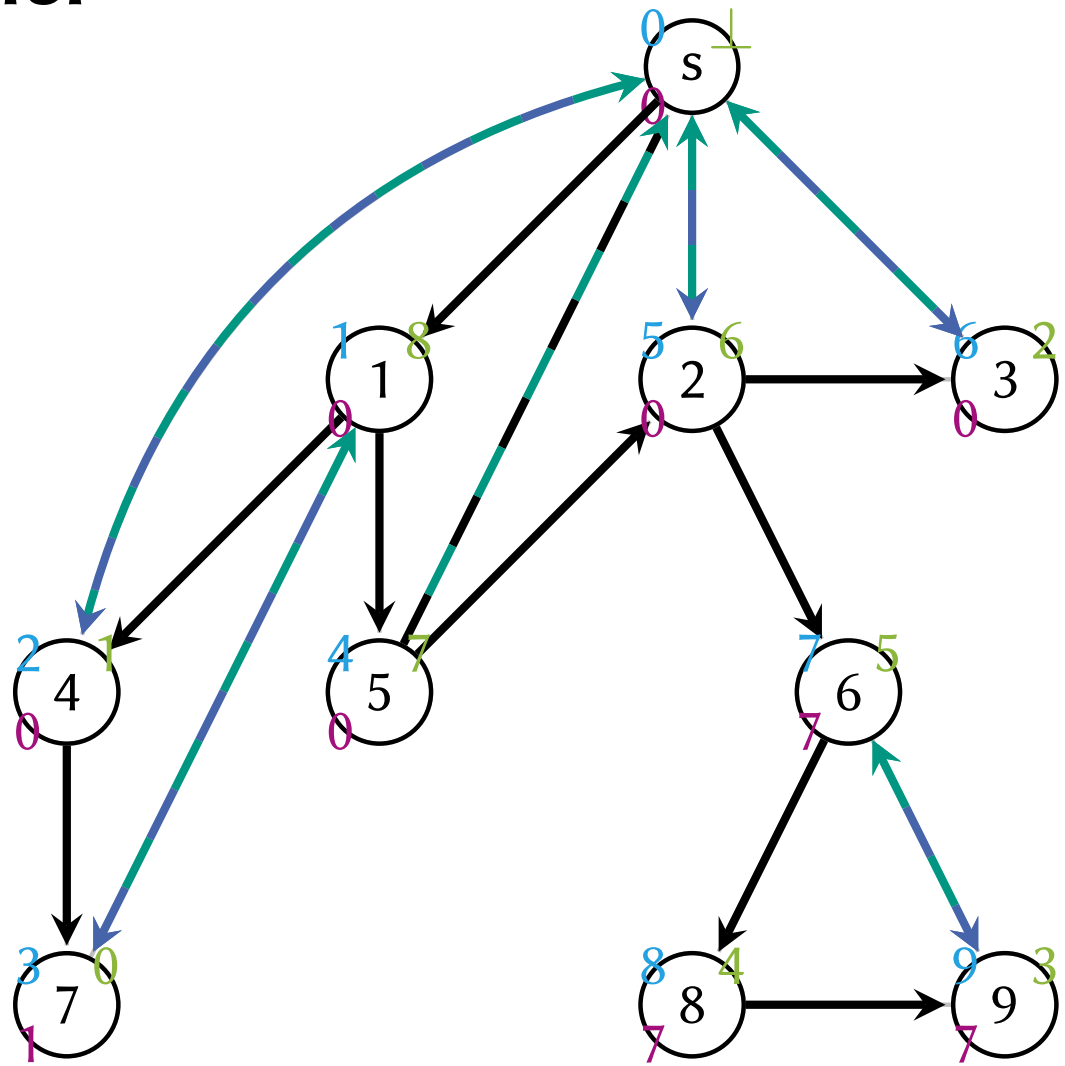
Kanten:

← unknown

← tree

← backward

← forward



entdecke Vorwärts-Kante $(s, 4)$

DFS Beispiel

Knoten:

dfsNum

finNum

lowNum

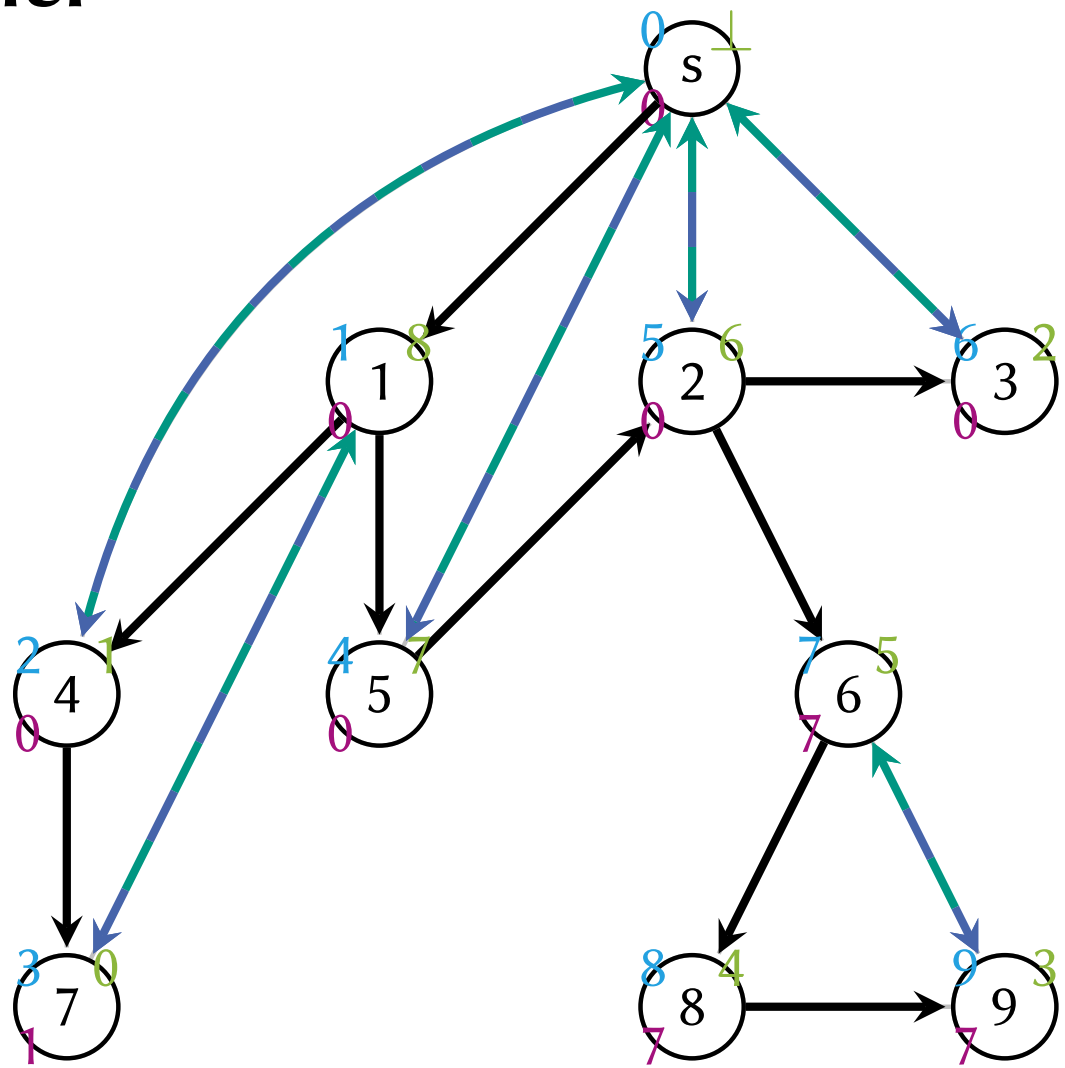
Kanten:

← unknown

← tree

← backward

← forward



entdecke Vorwärts-Kante $(s, 5)$

DFS Beispiel

Knoten:

dfsNum

finNum

lowNum

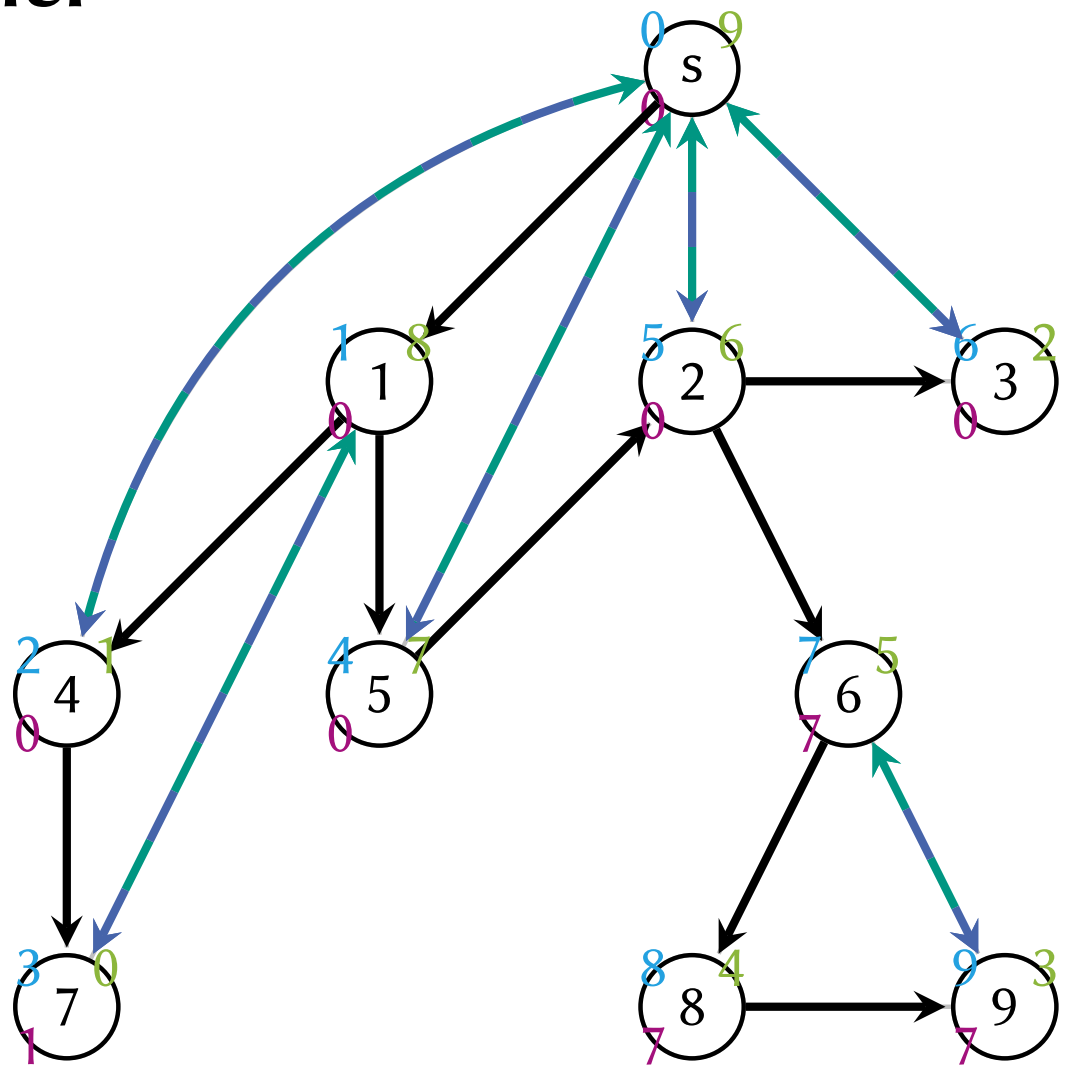
Kanten:

← unknown

← tree

← backward

← forward



finalisiere *s*

Knoten in Relation (ungerichtet)

Im Folgenden betrachten wir zwei unterschiedliche Knoten $u, v \in V$ aus einem beliebigen Graphen. Es gilt für den DFS-Baum des Graphen:

	Beziehung
v ist cut-vertex $\star$	$v = par[u] \wedge \texttt{lowNum}[u] >= \texttt{dfsNum}[v]$
(v, u) ist Brücke	$\texttt{lowNum}[u] > \texttt{dfsNum}[v]$

$\star$

Wenn v Startknoten, dann muss geprüft werden, ob v mehr als ein Kind im DFS-Baum hat.

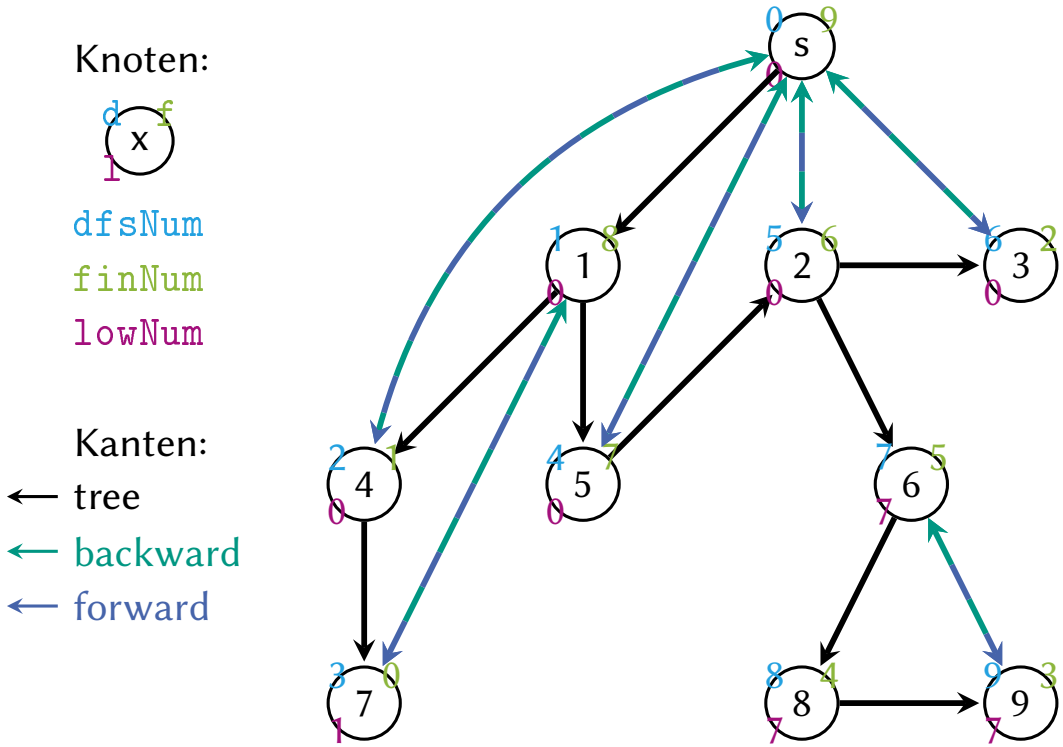
v ist cut-vertex

Bedingung:

$$v = par[u]$$
$$\wedge \ \texttt{lowNum}[u] >= \texttt{dfsNum}[v]$$

Zum Beispiel 6 ist cur-vertex:

$6 = par[8] \wedge \texttt{lowNum}[8] = 7 >= 7 = \texttt{dfsNum}[6]$.

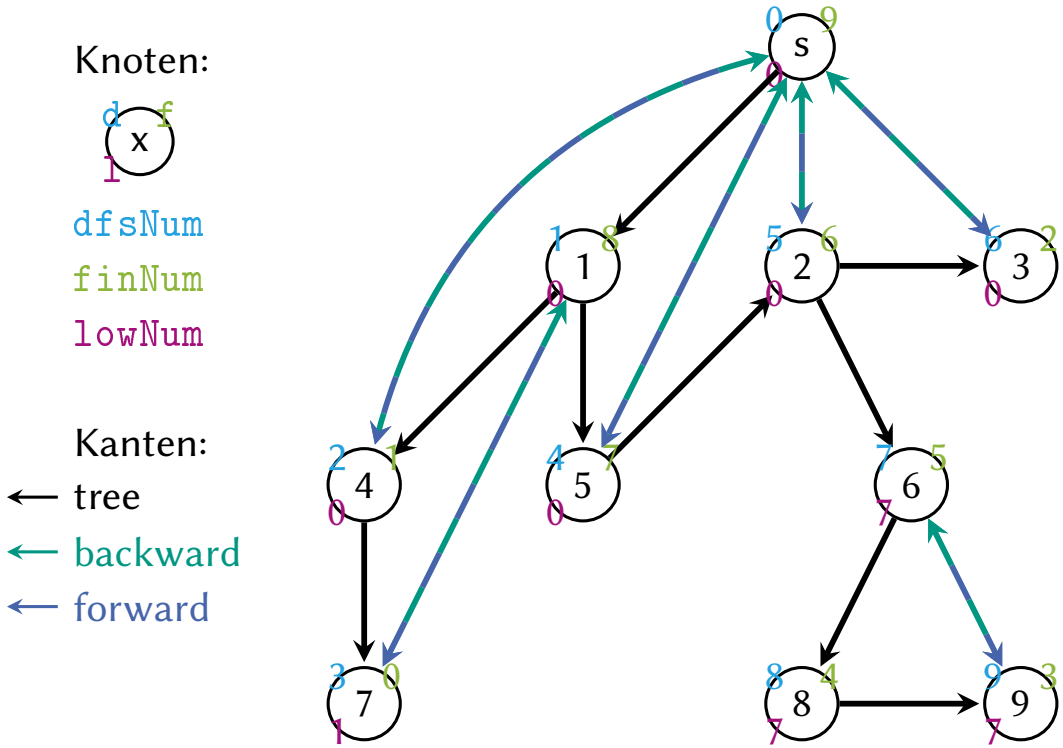


(v, u) ist Brücke

Bedingung:

$$\texttt{lowNum}[u] > \texttt{dfsNum}[v]$$

Zum Beispiel ist (v, u) Brücke: $\texttt{lowNum}[6] = 7 > 5 = \texttt{dfsNum}[2]$.



Topologische Sortierung

Topologische Sortierung

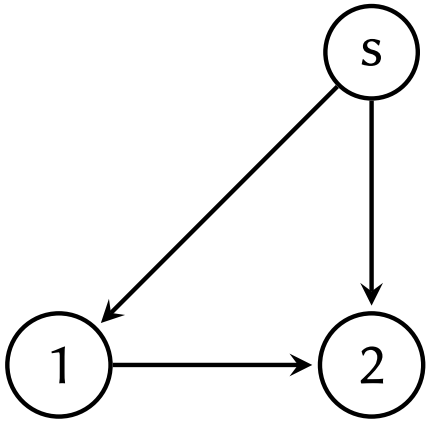
Eine Topologische Sortierung ist die Ordnund bestimmter Objekte basierend auf einer vorgegebenen Abhängigkeits-Beziehung.
Wir betrachten in der Vorlesung Topologische Sortierungen auf Knoten.

Topologische Sortierung auf DAGs

Sei $G = (V, E)$ ein DAG, für Knoten $u, v \in V$ gilt $u <_T v$, wenn es einen Pfad von v nach u in G gibt.

Pingo

Was ist die Topologische Sortierung von unteren Graph?



TopoSort Pseudocode DAGs

```
1: TopoSort(G = (V, E): DAG) : [Node; n]
2:     topo: [ℕ; n] = ⟨0, ..., 0⟩
3:     for v ∈ V do
4:         if not marked v then
5:             DFS(G, v)
6:     return topo
```

TopoSort Pseudocode DAGs

```
1:  TopoSort(G = (V, E): DAG) : [Node; n]
2:      topo: [ℕ; n] = ⟨0, ..., 0⟩
3:      for v ∈ V do
4:          if not marked v then
5:              DFS(G, v)
6:      return topo
7:  DFS(G = (V, E): Graph, v : Node)
8:      mark v
9:      for u ∈ N(v) do
10:         if not marked u then
11:             DFS(G, u)
12:     finNum[v] := finCounter
13:     topo[finCounter] := v
14:     finCounter := finCounter + 1
```

Topologische Sortierung auf DAGs

Knoten:

dfsNum

finNum

lowNum

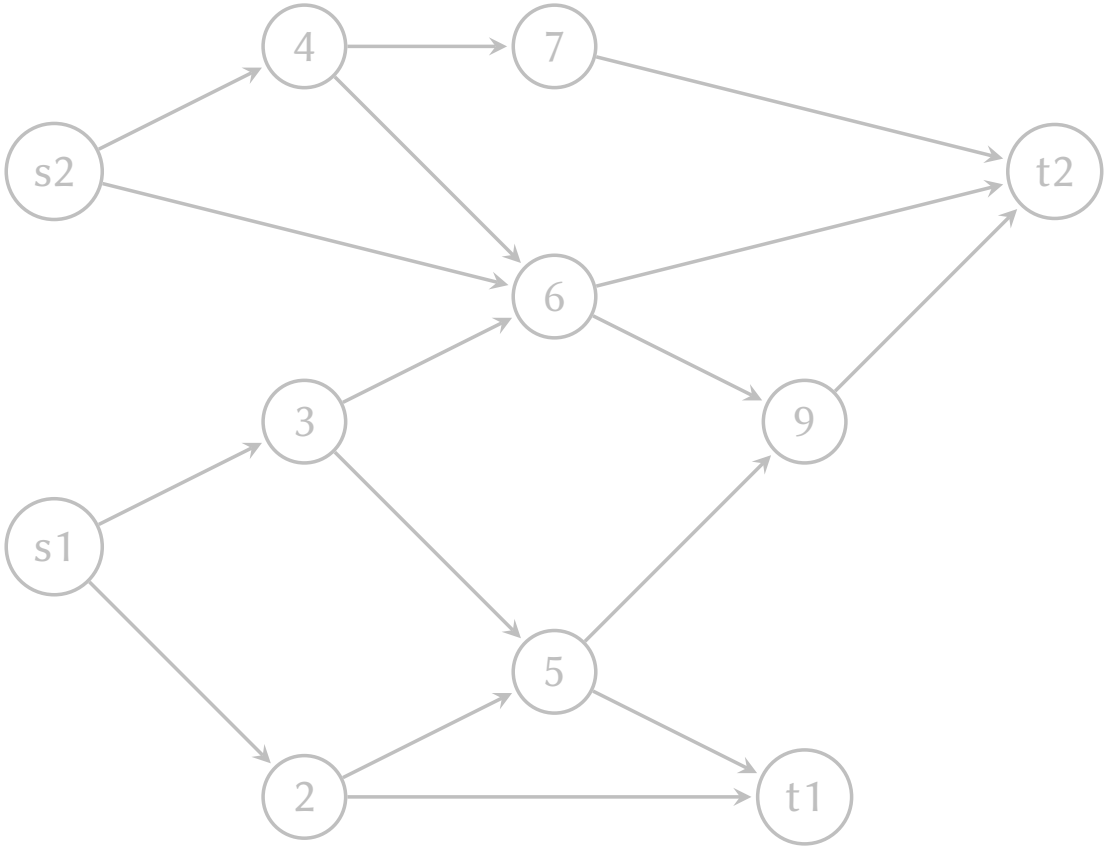
Kanten:

tree

backward

forward

cross



Topologische Sortierung:

Topologische Sortierung auf DAGs

Knoten:

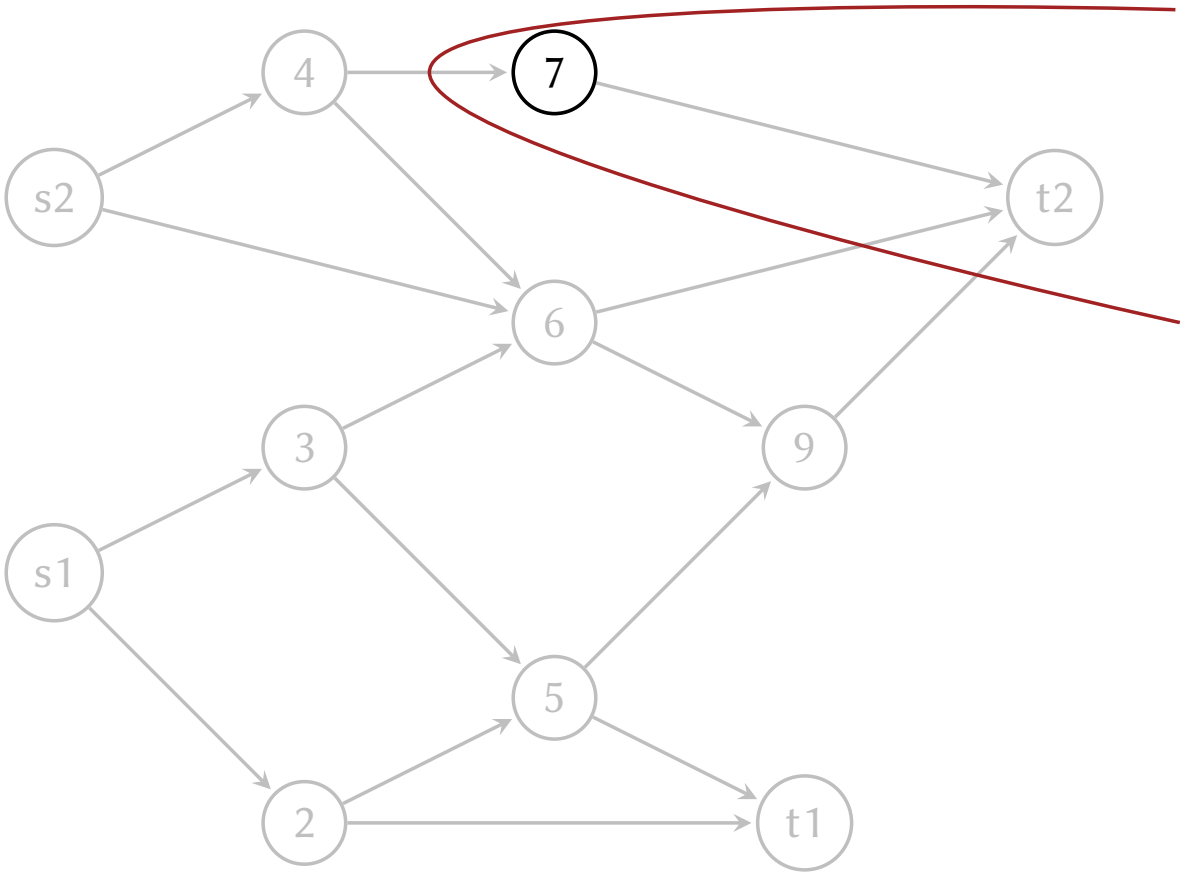
dfsNum

finNum

lowNum

Kanten:

- tree
- backward
- forward
- cross



Topologische Sortierung:

Topologische Sortierung auf DAGs

Knoten:

dfsNum

finNum

lowNum

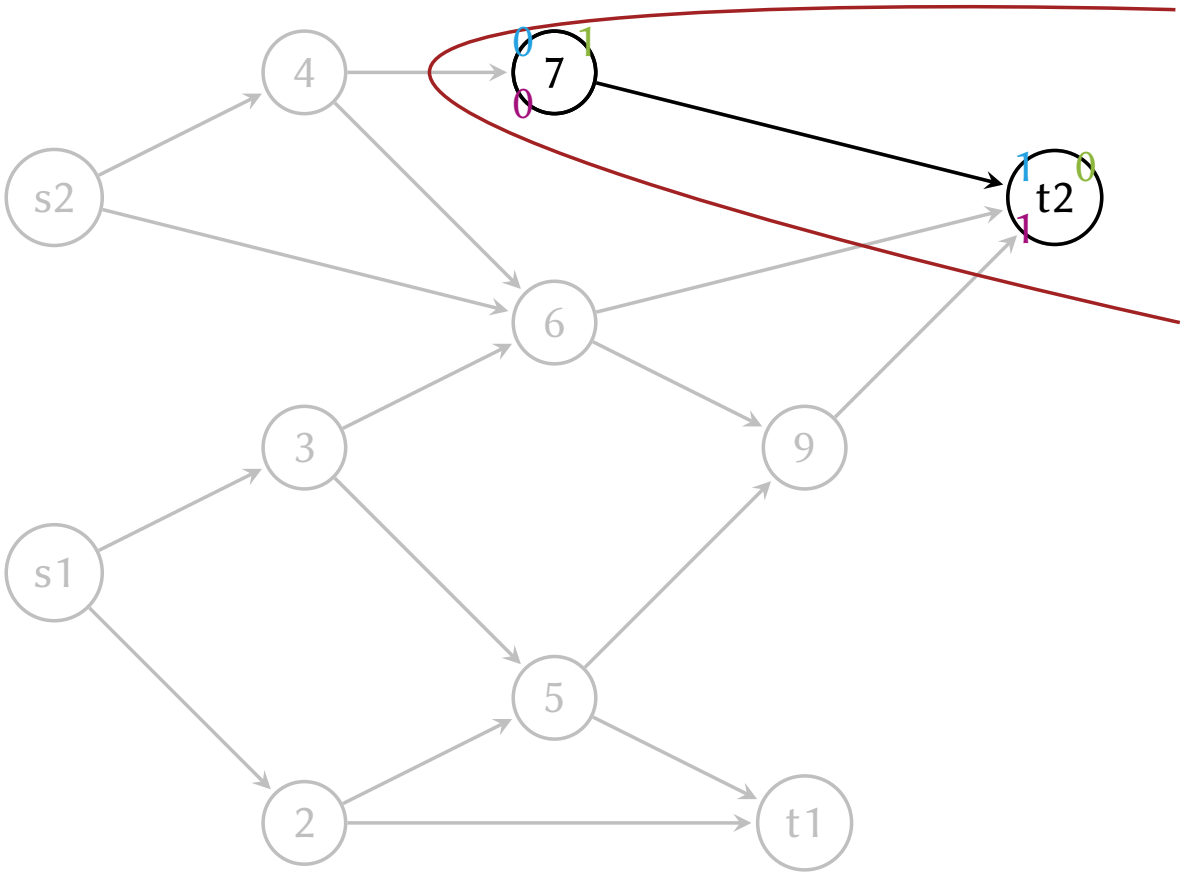
Kanten:

← tree

← backward

← forward

← cross



Topologische Sortierung:

t2, 7

Topologische Sortierung auf DAGs

Knoten:

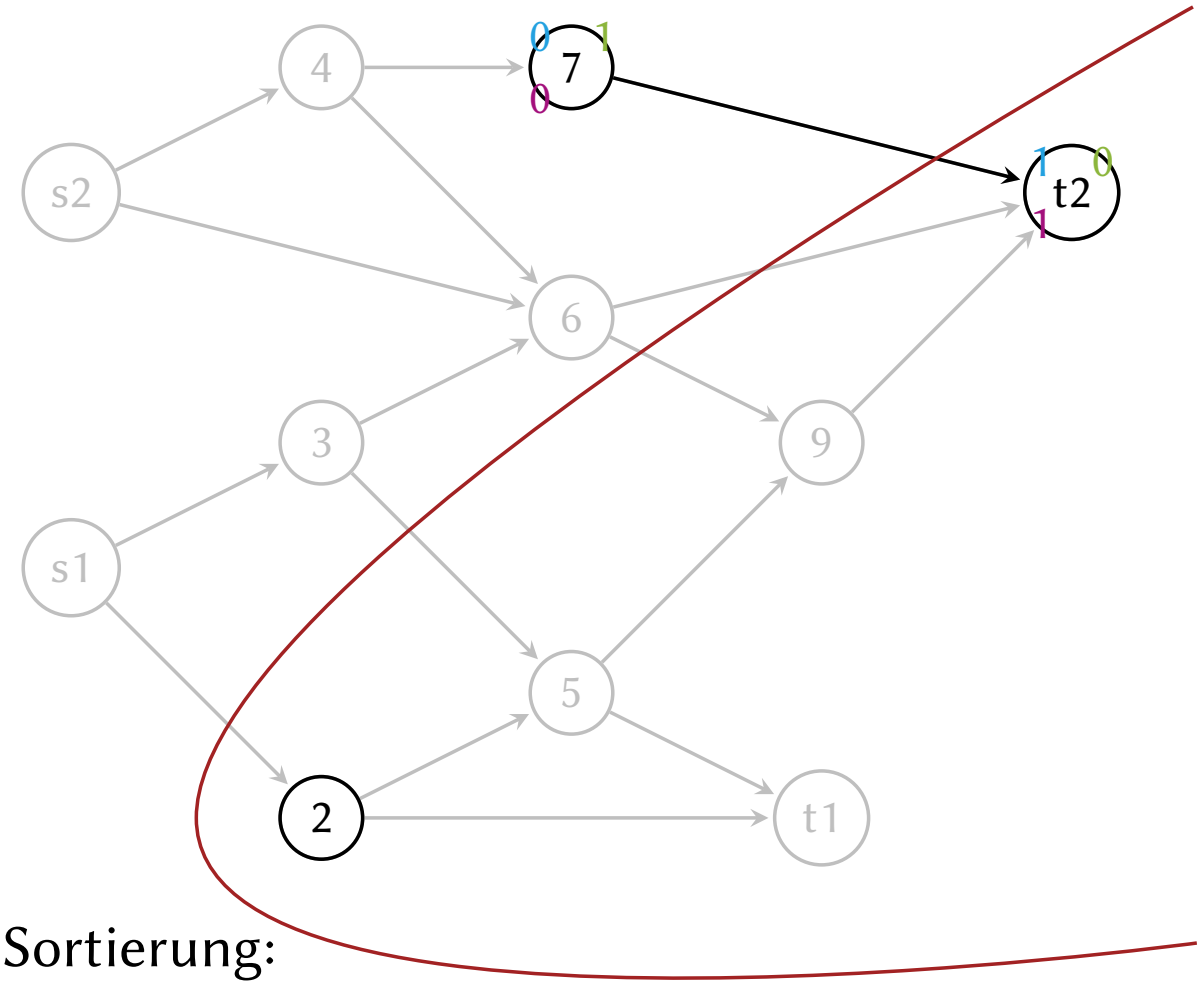
`dfsNum`

`finNum`

`lowNum`

Kanten:

- tree
- backward
- forward
- cross



Topologische Sortierung:
t2, 7

Topologische Sortierung auf DAGs

Knoten:

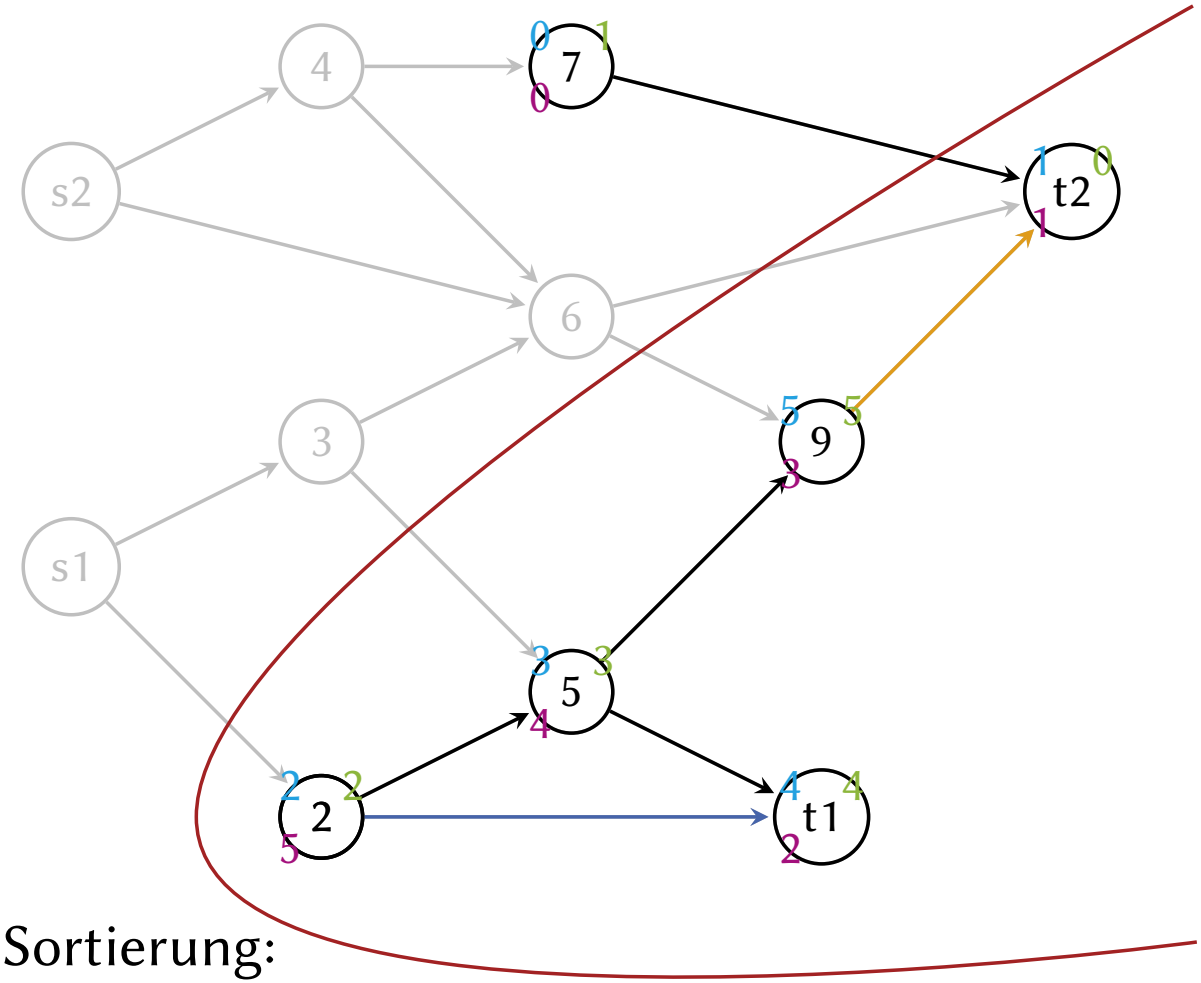
dfsNum

finNum

lowNum

Kanten:

- tree
- backward
- forward
- cross



Topologische Sortierung:

t2, 7, t1, 9, 5, 2

Topologische Sortierung auf DAGs

Knoten:

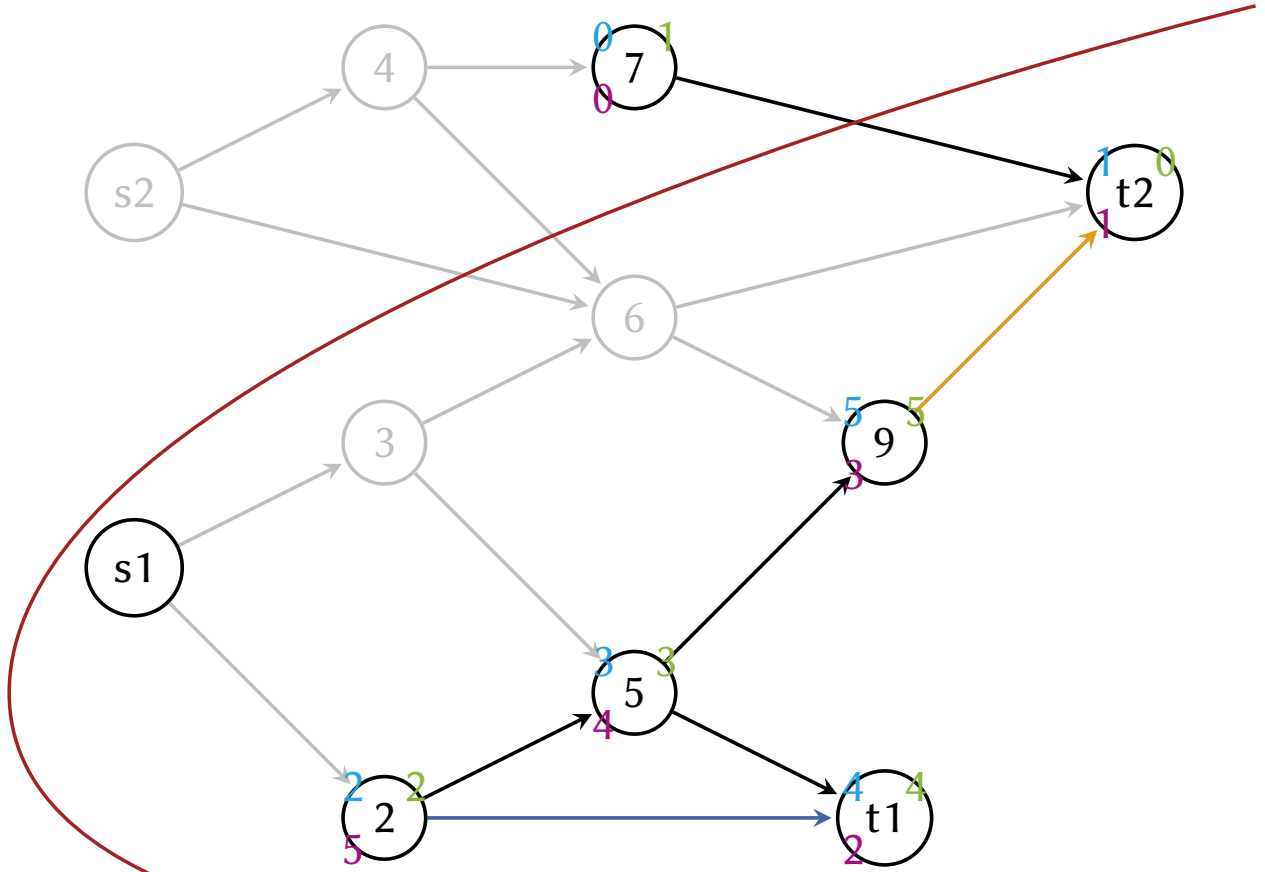
`dfsNum`

`finNum`

`lowNum`

Kanten:

- tree
- backward
- forward
- cross



Topologische Sortierung:

t2, 7, t1, 9, 5, 2

Topologische Sortierung auf DAGs

Knoten:

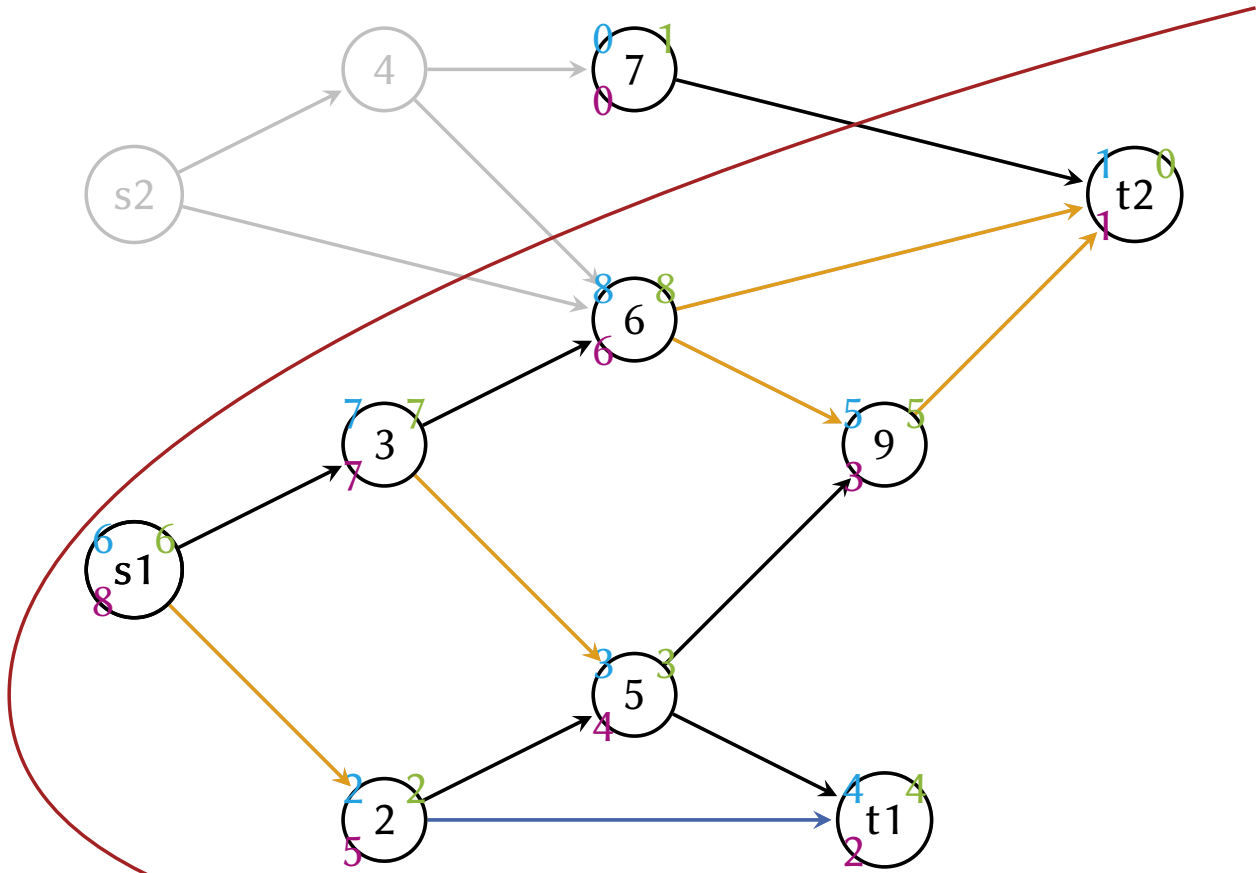
`dfsNum`

`finNum`

`lowNum`

Kanten:

- tree
- backward
- forward
- cross



Topologische Sortierung:

t2, 7, t1, 9, 5, 2, 6, 3, s1

Topologische Sortierung auf DAGs

Knoten:

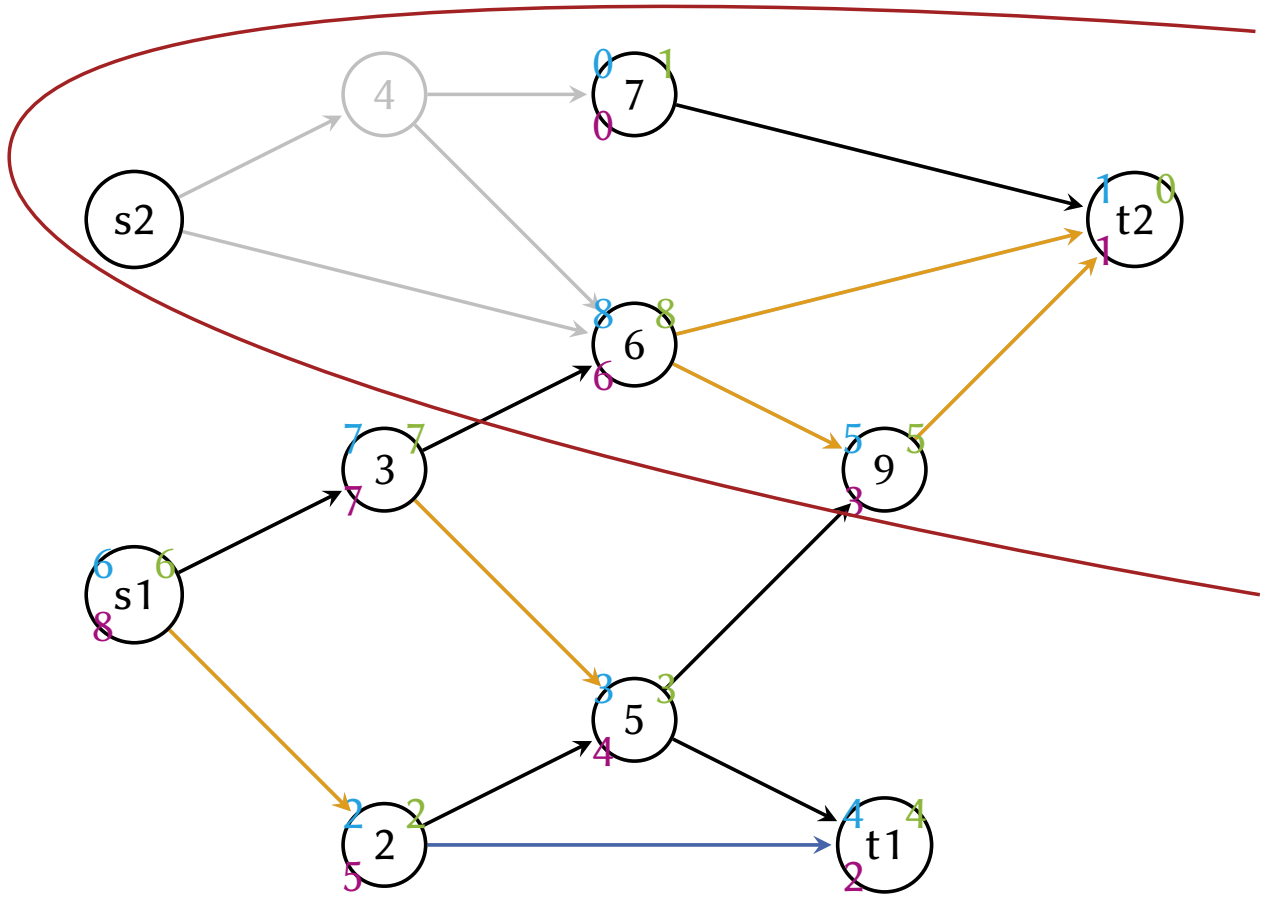
dfsNum

finNum

lowNum

Kanten:

- tree
- backward
- forward
- cross



Topologische Sortierung:

t2, 7, t1, 9, 5, 2, 6, 3, s1

Topologische Sortierung auf DAGs

Knoten:

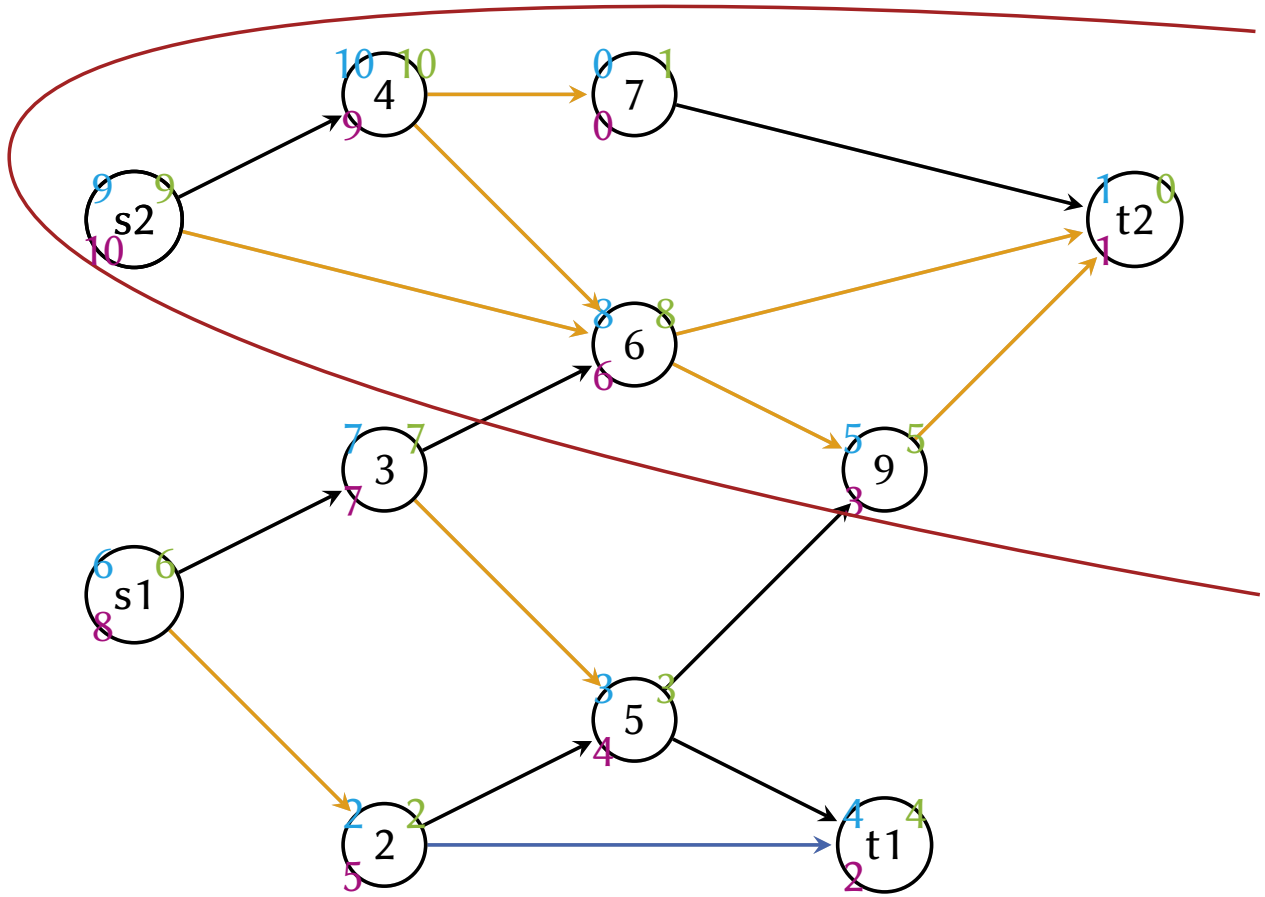
dfsNum

finNum

lowNum

Kanten:

- tree
- backward
- forward
- cross



Topologische Sortierung:

t2, 7, t1, 9, 5, 2, 6, 3, s1, 4, s2

DFS-Baum TopoSort

Achtung!

- Der TopoSort-Algorithmus baut keinen korrekten DFS-Baum auf. Wir haben vielmehr einen Wald der stark von der Wahl der Knotenreihenfolge abhängt.

DFS-Baum TopoSort

Achtung!

- Der TopoSort-Algorithmus baut keinen korrekten DFS-Baum auf. Wir haben vielmehr einen Wald der stark von der Wahl der Knotenreihenfolge abhängt.
- Zudem können keine Rückwärtskanten auftreten. Wenn Rückwärtskanten auftritt, dann ist der Graph kein DAG und hat mindestens einen Kreis.

Roadmap

Dynamische Programmierung

Dynamische Programmierung

Dynamische Programmierung beschreibt einen Ansatz zur Lösung von Problemen, deren **optimale Lösung** sich **aus optimalen Lösungen von Teilproblemen** zusammensetzt. Man errechnet zunächst die optimalen Lösungen der Teilprobleme, um daraus eine Lösung des Gesamtproblems zu berechnen.

- Man verwendet auf diese Weise eine Rekurrenz, um eine Tabelle von Teillösungen nach und nach zu füllen, bis man die Gesamtlösung gefunden hat.
- Zum Finden einer sinnvollen Formulierung von Teilproblemen ist Kreativität gefragt.
- Es muss gezeigt werden, dass optimale Lösungen der Teilprobleme auch zu einer optimalen Lösung des zusammengesetzten Problems führen.

DP - MST

Bekanntes Beispiel

Wir kennen bereits einen Algorithmus, der dynamische Programmierung verwendet: Der Kruskal-Algorithmus.

- Hier bestehen die Teilprobleme im Finden eines MST auf Teilgraphen.
- Als Rekursionsanfang oder kleinstes Teilproblem wählt man Teilgraphen mit nur einem Knoten, da sie ihr eigener MST sind.
- In der VL wurde gezeigt, dass sich ein MST aus den MSTs der Teilgraphen zusammensetzt. Es wird bei Kruskal somit eine optimale Lösung des Gesamtproblems aus optimalen Lösungen von Teilproblemen zusammengesetzt.

Einschub: Greedy-Algorithmus

Def.: Greedy-Algorithmus

Ein **Greedy-Algorithmus** ist ein Algorithmus, der **lokal optimale Schritte** macht. Es wird schrittweise der Folgezustand ausgewählt, der zum Zeitpunkt der Wahl das beste Ergebnis verspricht (Bewertung der möglichen Teillösugen durch eine Bewertungsfunktion).
Greedy-Algorithmen sind schnell, lösen Probleme i.A. aber nicht optimal.

Einschub: Greedy-Algorithmus

Def.: Greedy-Algorithmus

Ein **Greedy-Algorithmus** ist ein Algorithmus, der **lokal optimale Schritte** macht. Es wird schrittweise der Folgezustand ausgewählt, der zum Zeitpunkt der Wahl das beste Ergebnis verspricht (Bewertung der möglichen Teillösugen durch eine Bewertungsfunktion).
Greedy-Algorithmen sind schnell, lösen Probleme i.A. aber nicht optimal.

Kruskal ist ein Greedy-Algorithmus und löst das MST Problem optimal, das gilt aber nicht für alle Probleme, weshalb wir Dynamische Programmierung brauchen, um optimale Lösungen zu finden.

Einschub: Greedy-Algorithmus

Def.: Greedy-Algorithmus

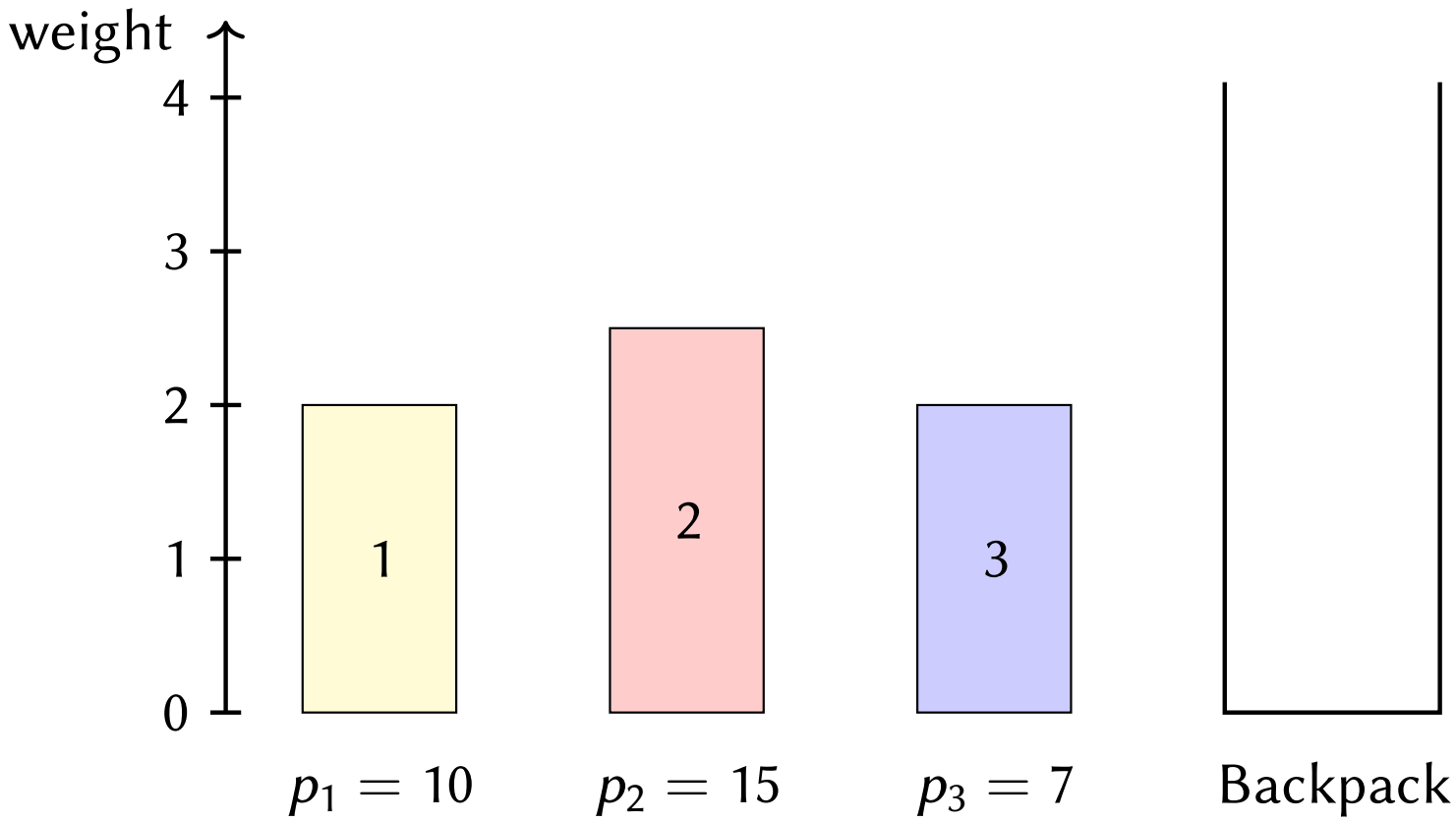
Ein **Greedy-Algorithmus** ist ein Algorithmus, der **lokal optimale Schritte** macht. Es wird schrittweise der Folgezustand ausgewählt, der zum Zeitpunkt der Wahl das beste Ergebnis verspricht (Bewertung der möglichen Teillösugen durch eine Bewertungsfunktion).
Greedy-Algorithmen sind schnell, lösen Probleme i.A. aber nicht optimal.

Kruskal ist ein Greedy-Algorithmus und löst das MST Problem optimal, das gilt aber nicht für alle Probleme, weshalb wir Dynamische Programmierung brauchen, um optimale Lösungen zu finden.

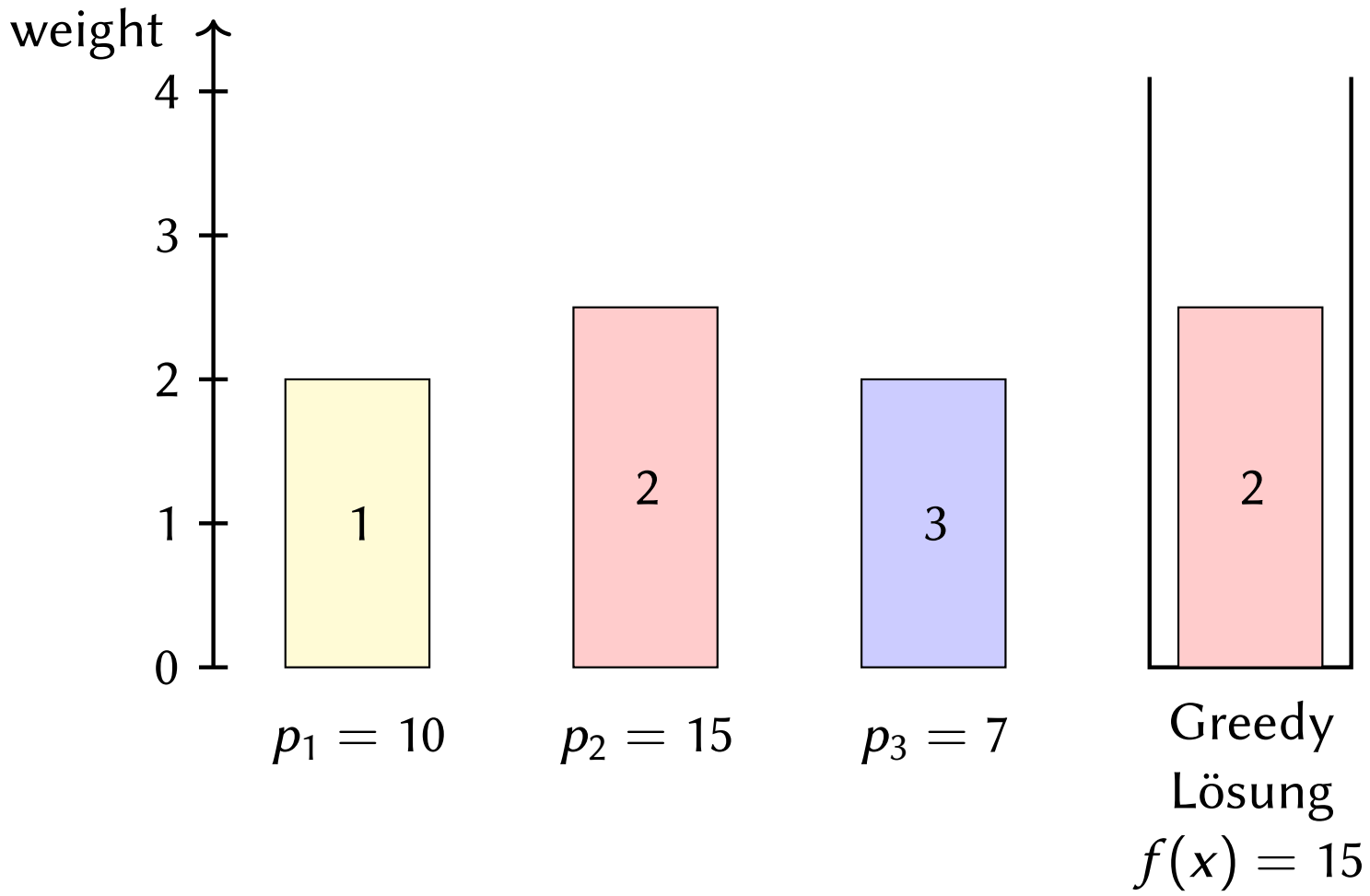
Deshalb: Beweis bei DP notwendig

Da wir zeigen wollen, das unser DP-Algorithmus korrekte optimale Lösungen liefert, müssen wir daher beiweisen, dass das zusammensetzten der Teilprobleme immer optimale Lösungen liefert.

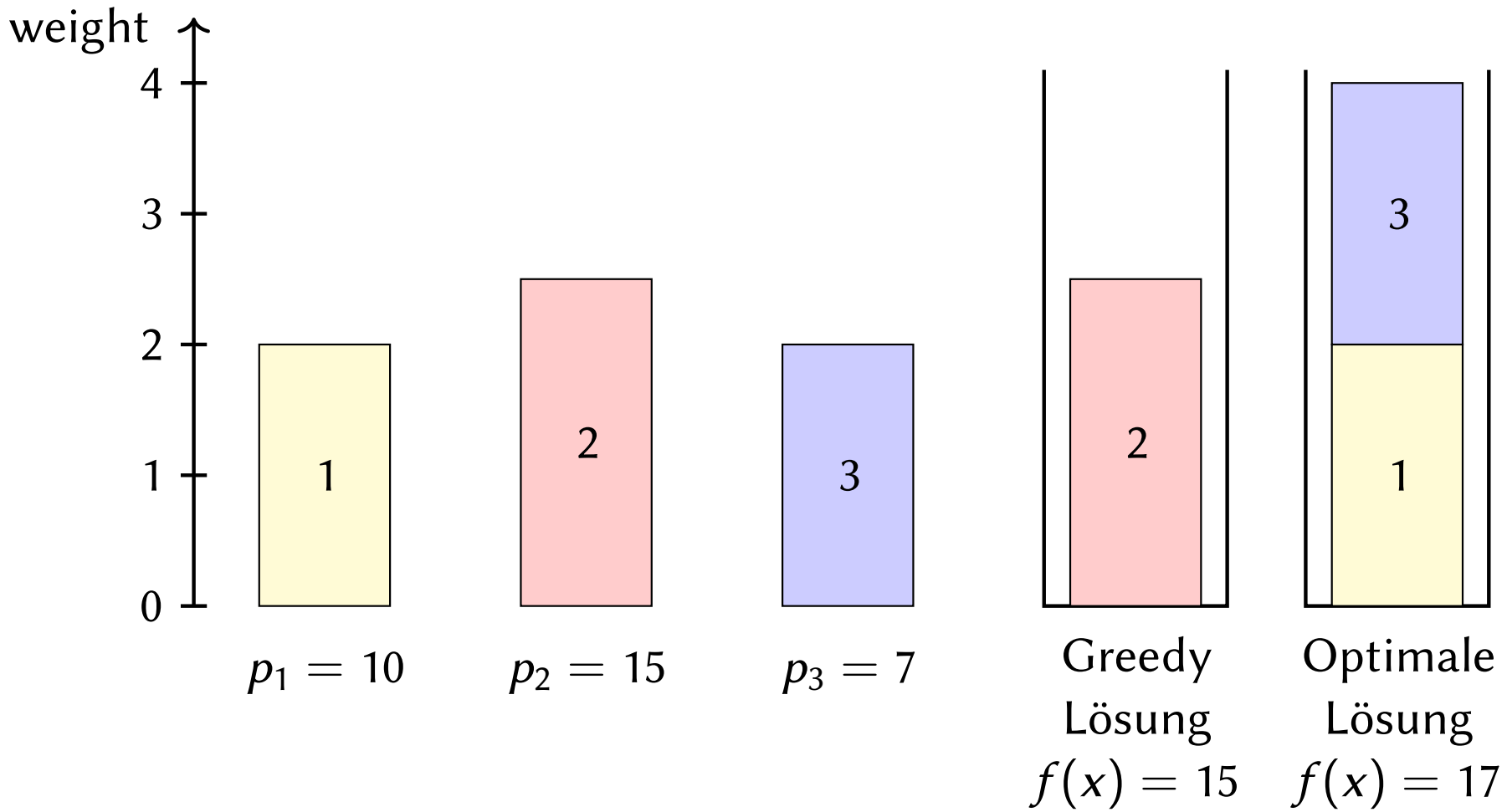
Rucksackproblem



Rucksackproblem



Rucksackproblem



DP - Rucksackproblem

Auch das Rucksackproblem kann durch dynamische Programmierung gelöst werden. Die Formulierung eines Teilproblems lautet:

Betrachte das Rucksackproblem mit einem Teilrucksack der Größe $C \leq M$ und nur den Elementen $\{1..i\}$ für $i \leq n$.

Lemma zu Teilproblemen des Rucksackproblems

Für den optimalen Profit $P(i, C)$ eines Rucksacks der Kapazität C mit den Elementen $\{1..i\}$ gilt:

$$P(i, C) = \max(\underbrace{P(i-1, C)}_{\substack{\text{wenn } i \text{ nicht} \\ \text{gewählt wird}}}, \underbrace{P(i-1, C-w_i) + p_i}_{\text{wenn } i \text{ gewählt wird}})$$

DP - Rucksackproblem

- Wir setzen also als Rekursionsanfang/kleinstes Teilproblem $P(0, C) = 0$.
- Es gilt dann außerdem offensichtlich $P(i, 0) = 0$ für $i = 1, \ldots, n$.
- Von hier aus können wir das Lemma mit den Informationen über Gewicht/Profit der Gegenstände verwenden, um eine Tabelle mit $i \times C$ zu füllen.
- Wir sind fertig, wenn $P(n, M)$ berechnet wurde.

DP - Rucksackproblem Algo

```
1:  dynamicKnapsack(p: [ℕ; n], w: [ℕ; n], M: ℕ) : ℕ
2:      P : [[ℕ_0;M + 1];n + 1] = ⟨⟨0, ..., 0⟩, ..., ⟨0, ..., 0⟩⟩
3:      decision : [[Bool;M + 1];n + 1]= ⟨⟨f, ..., f⟩, ..., ⟨f, ..., f⟩⟩
4:      for i ∈ {1, ..., n − 1} do
5:          for C ∈ {0, ..., M} do
6:              if C ≥ w_i and P[i − 1][C − w_i] + p_i > P[i − 1][C] then
7:                  P[i][C] := P[i − 1][C − w_i] + p_i
8:                  decision[i, C] := w
9:              else
10:                 P[i][C] := P[i − 1][C]
11:                 decision[i, C] := f
12: return P[n][M]
```

Laufzeit?

DP - Rucksackproblem Algo

```
1:  dynamicKnapsack(p: [ℕ; n], w: [ℕ; n], M: ℕ) : ℕ
2:      P : [[ℕ_0;M + 1];n + 1] = ⟨⟨0, ..., 0⟩, ..., ⟨0, ..., 0⟩⟩
3:      decision : [[Bool;M + 1];n + 1]= ⟨⟨f, ..., f⟩, ..., ⟨f, ..., f⟩⟩
4:      for i ∈ {1, ..., n − 1} do
5:          for C ∈ {0, ..., M} do
6:              if C ≥ w_i and P[i − 1][C − w_i] + p_i > P[i − 1][C] then
7:                  P[i][C] := P[i − 1][C − w_i] + p_i
8:                  decision[i, C] := w
9:              else
10:                 P[i][C] := P[i − 1][C]
11:                 decision[i, C] := f
12: return P[n][M]
```

Laufzeit? $\mathcal{O}(nM) \Rightarrow$ *pseudopolynomiell*

DP - Rucksackproblem Beispiel

Es sei ein Rucksack der Kapazität $M = 7$, der Gewichtsvektor $w = (1, 3, 5, 4)$ und der Profitvektor $p = (10, 15, 30, 20)$ gegeben.

Wir haben eine Lösungstabelle inder der Algorithmus $P = (i, C)$ und in Klammern $decision[i][C]$ in jeder Zelle einträgt.

i \ C	0	1	2	3	4	5	6	7
0	0 (**f**)	0 (**f**)	0 (**f**)	0 (**f**)	0 (**f**)	0 (**f**)	0 (**f**)	0 (**f**)
1	0 (**f**)							
2	0 (**f**)							
3	0 (**f**)							
4	0 (**f**)							

DP - Rucksackproblem Lösung

Lösung

i \ C	0	1	2	3	4	5	6	7
0	0 (**f**)	0 (**f**)	0 (**f**)	0 (**f**)	0 (**f**)	0 (**f**)	0 (**f**)	0 (**f**)
1	0 (**f**)	10 (**w**)	10 (**w**)	10 (**w**)	10 (**w**)	10 (**w**)	10 (**w**)	10 (**w**)
2	0 (**f**)	10 (**f**)	10 (**f**)	15 (**w**)	25 (**w**)	25 (**w**)	25 (**w**)	25 (**w**)
3	0 (**f**)	10 (**f**)	10 (**f**)	15 (**f**)	25 (**f**)	30 (**w**)	40 (**w**)	40 (**w**)
4	0 (**f**)	10 (**f**)	10 (**f**)	15 (**f**)	25 (**f**)	30 (**f**)	40 (**f**)	40 (**f**)

```
1: getSolution: {0,1}^n                          // Rekonstruiere Lösung
2:     C := M
3:     x : [Bool;n+1] = ⟨f, ..., f⟩              // Index 0 in x wird ignoriert
4:     for i ∈ {n, ..., 1} do
5:         x[i] := decision[i, C]
6:         if x[i] == w then C := C − w_i
7:     return x
```

Lösen von Problemen mit DP

Schritt 1 - Kreativität:

Wie kann das Probelem in Teilprobleme zerlegt werden, sodass alle Lösungen der Teilprobleme eine Gesamtlösung ergeben?

Lösen von Problemen mit DP

Schritt 1 - Kreativität:

Wie kann das Probelem in Teilprobleme zerlegt werden, sodass alle Lösungen der Teilprobleme eine Gesamtlösung ergeben?

Schritt 2 - Rekurenz:

Wie können wir rekursive aus den Teilproblemen eine Gesamtlösung berechnen?

Deim Zusammensetzen muss bewiesen werden, das die neue Lösung ein korrekte und falls notwendig optimales Ergebnis ist.

Lösen von Problemen mit DP

Schritt 1 - Kreativität:

Wie kann das Probelem in Teilprobleme zerlegt werden, sodass alle Lösungen der Teilprobleme eine Gesamtlösung ergeben?

Schritt 2 - Rekurenz:

Wie können wir rekursive aus den Teilproblemen eine Gesamtlösung berechnen?

Deim Zusammensetzen muss bewiesen werden, das die neue Lösung ein korrekte und falls notwendig optimales Ergebnis ist.

Schritt 3 - Algorithmenkonstruktion:

Wie kann die rekursive Formel iterativ berechnet werden?

In welcher Reihenfolge werden die Lösungen berechnet um keine Abhängigkeitsprobleme zu bekommen?

Schritt 1 - Kreativität

Zentrale Frage: Wie kann das Problem zerlegt werden?

$n \rightsquigarrow n - 1$

$n \rightsquigarrow n/2$

Baum $\rightsquigarrow$ Teilbäume

String $\rightsquigarrow$ Substring(s)

Array $\rightsquigarrow$ TeilArray(s)

Mengen von Elemente $\rightsquigarrow$ Teilemenge(n) der Elemente

...

Schritt 1 - Kreativität

Zentrale Frage: Wie kann das Problem zerlegt werden?

$n \rightsquigarrow n - 1$

$n \rightsquigarrow n/2$

Baum $\rightsquigarrow$ Teilbäume

String $\rightsquigarrow$ Substring(s)

Array $\rightsquigarrow$ TeilArray(s)

Mengen von Elemente $\rightsquigarrow$ Teilemenge(n) der Elemente

...

Wichtig dabei; Teillösungen müssen zu Gesamtlösung zusammengebaut werden können.

Schritt 2 - Rekurrenz

Nachdem wir eine Idee haben wie wir das Problem zerlegen können, geht es darum eine rekursive Formel zu finden, inder alle Fälle inbegriffen sind, die auftreten können um eine optimale Lösung aus Teillösungen zu erzeugen.

Für das Rucksackproblem gilt:

$$P(i, C) = \max(\underbrace{P(i-1, C)}_{\text{wenn } i \text{ nicht gewählt wird}}, \underbrace{P(i-1, C-w_i) + p_i}_{\text{wenn } i \text{ gewählt wird}})$$

In diesem Fall haben wir 2 Möglichkeiten: wir nehmen das aktuelle Element i oder wir nehmen das aktuelle Element i nicht.

Schritt 2 - Rekurenz 2

In diesem Beispiel ist relativ schnell ersichtlich, warum wir aus den Teilproblemen eine optimale Lösung erzeugen:
Wir betrachten die Elemente nacheinander für jede Kapazität. Dadurch entscheiden wir jeweils lokal, ob es für eine bestimmte Kapazität und eine ausgewählten Teilmenge an Elementen Sinn macht das aktuell betrachtete Element mitzunehmen.
konkret:
Für Menge $M = \{e_1, \ldots, e_n\}$ haben wir im Schritt i, e_i aus Menge $M_i = \{e_1, \ldots, e_i\}$ betrachtet und das für alle validen Kapazitäten. Somit haben wir alle möglichen Kombinationen mindestens einmal durchprobiert.

Schritt 3 - Algorithmenkonstruktion

Im Schritt 3 ist es das Ziel einen iterativen Algorithmus zu schreiben der die Rekursionsformel umsetzt.

Bei vielen DP Problemen muss oft das gleiche Ergebnis mehrfach berechnet werden. Durch iterativen Algorithmenentwurf können Zwischenergebnisse gespeichert werden, dadurch müssen Zwischenergebnisse nur ein mal berechnet werden. Und die Laufzeit ist polynomiell in der Eingabegröße.

Simples Beispiel für Exponentielle Laufzeit wenn man die Rekurenzformel einfach nur implementiert sind die FIbonacci Zahlen: $F(0) = 0, F(1) = 1, F(n) = F(n-1) + F(n-2)$. Der Rekursionsbaum wird exponentiell groß.

Levenshtein Distanz

Levenshtein Distanz

Gegeben zwei Strings S_1, S_2 betrachte folgende Operationen auf S_1

- insert(i, c) Füge in S_1 an Stelle i Zeichen c ein, die nachfolgenden Zeichen rutschen nach hinten.
- delete(i) Entferne Zeichen i aus S_1, die nachfolgenden Zeichen rutschen nach.
- replace(i, c) Ersetze in S_1 Zeichen i durch Zeichen c, nachfolgende Zeichen bleiben unverändert.

Die Levenshtein Distanz gibt die minimale Anzahl an Operationen an, um S_1 in S_2 zu transformieren.

Beispiel Levenshtein Distanz

S_1 `levenshtein`

S_2 `meilenstein`

Transformation

- `levenshtein`

Beispiel Levenshtein Distanz

S_1 `levenshtein`

S_2 `meilenstein`

Transformation

- `levenshtein`
- `mevenshtein` replace(0, m)

Beispiel Levenshtein Distanz

S_1 `levenshtein`

S_2 `meilenstein`

Transformation

- `levenshtein`
- `mevenshtein` replace(0, m)
- `meivenshtein` insert(2, i)

Beispiel Levenshtein Distanz

S_1 `levenshtein`

S_2 `meilenstein`

Transformation

- `levenshtein`
- `mevenshtein` replace(0, m)
- `meivenshtein` insert(2, i)
- `meilenshtein` replace(3, l)

Beispiel Levenshtein Distanz

S_1 `levenshtein`

S_2 `meilenstein`

Transformation

- `levenshtein`
- `mevenshtein` replace(0, m)
- `meivenshtein` insert(2, i)
- `meilenshtein` replace(3, l)
- `meilenstein` delete(7)

Beispiel Levenshtein Distanz

S_1 `levenshtein`

S_2 `meilenstein`

Transformation

- `levenshtein`
- `mevenshtein` replace(0, m)
- `meivenshtein` insert(2, i)
- `meilenshtein` replace(3, l)
- `meilenstein` delete(7)

$\rightarrow$ Levenshtein Distanz: 4

Pingo

Was ist die Levenshtein Distanz von `Tier` und `Tor`?

Schritt 1 - Levenshtein Distanz - DP

Idee: auf kürzeren Wörtern ist die Levenshtein Distanz leichter zu berechnen(weniger Kombinationen), d.h. wir unterteilen das Wort aber wie?

halbieren Problem: Wo unterteilen wir? Wir müssten S_1 und S_2 so teilen, dass die Schnitte in den Zeichenketten nach den Operationen gleich sind. Dementsprechend müssten wir wissen wie viele Einfüge- und Löschoperationen wir auf einem der beiden Teilstrings machen.

Schritt 1 - Levenshtein Distanz - DP

Idee: auf kürzeren Wörtern ist die Levenshtein Distanz leichter zu berechnen(weniger Kombinationen), d.h. wir unterteilen das Wort aber wie?

halbieren Problem: Wo unterteilen wir? Wir müssten S_1 und S_2 so teilen, dass die Schnitte in den Zeichenketten nach den Operationen gleich sind. Dementsprechend müssten wir wissen wie viele Einfüge- und Löschoperationen wir auf einem der beiden Teilstrings machen.

verkürzen Wenn wir die Strings entsprechend der Operation um 1 verkürzen, fangen wir bei einem Zeichen an und erweitern so unsere Lösung bis wir beide Wörter vollständig gleichgemacht haben.

Schritt 2 - Levenshtein Distanz - DP

Notation: $S_k = c_k^0 c_k^1 \dots c_k^{n_k - 1}$ es sei $S_k^l = c_k^0 c_k^1 \dots c_k^l$ der Teilstring von S_k mit den ersten l Zeichen.

$$lev(S_1^{m_1}, S_2^{m_2}) = \begin{cases} m_1 & m_2 = 0 \\ m_2 & m_1 = 0 \\ \\ \\ \\ \\ \end{cases}$$

Schritt 2 - Levenshtein Distanz - DP

Notation: $S_k = c_k^0 c_k^1 \dots c_k^{n_k-1}$ es sei $S_k^l = c_k^0 c_k^1 \dots c_k^l$ der Teilstring von S_k mit den ersten l Zeichen.

$$lev(S_1^{m_1}, S_2^{m_2}) = \begin{cases} m_1 & m_2 = 0 \\ m_2 & m_1 = 0 \\ lev(S_1^{m_1-1}, S_2^{m_2-1}) & c_1^{m_1} = c_2^{m_2} \end{cases}$$

Schritt 2 - Levenshtein Distanz - DP

Notation: $S_k = c_k^0 c_k^1 \dots c_k^{n_k-1}$ es sei $S_k^l = c_k^0 c_k^1 \dots c_k^l$ der Teilstring von S_k mit den ersten l Zeichen.

$$lev(S_1^{m_1}, S_2^{m_2}) = \begin{cases} m_1 & m_2 = 0 \\ m_2 & m_1 = 0 \\ lev(S_1^{m_1-1}, S_2^{m_2-1}) & c_1^{m_1} = c_2^{m_2} \\ 1 + \min \begin{cases} lev(S_1^{m_1}, S_2^{m_2-1}) & \text{insert} \\ \\ \end{cases} \end{cases}$$

Schritt 2 - Levenshtein Distanz - DP

Notation: $S_k = c_k^0 c_k^1 \dots c_k^{n_k-1}$ es sei $S_k^l = c_k^0 c_k^1 \dots c_k^l$ der Teilstring von S_k mit den ersten l Zeichen.

$$lev(S_1^{m_1}, S_2^{m_2}) = \begin{cases} m_1 & m_2 = 0 \\ m_2 & m_1 = 0 \\ lev(S_1^{m_1-1}, S_2^{m_2-1}) & c_1^{m_1} = c_2^{m_2} \\ 1 + \min \begin{cases} lev(S_1^{m_1}, S_2^{m_2-1}) & \text{insert} \\ lev(S_1^{m_1-1}, S_2^{m_2}) & \text{delete} \end{cases} & \end{cases}$$

Schritt 2 - Levenshtein Distanz - DP

Notation: $S_k = c_k^0 c_k^1 \dots c_k^{n_k-1}$ es sei $S_k^l = c_k^0 c_k^1 \dots c_k^l$ der Teilstring von S_k mit den ersten l Zeichen.

$$lev(S_1^{m_1}, S_2^{m_2}) = \begin{cases} m_1 & m_2 = 0 \\ m_2 & m_1 = 0 \\ lev(S_1^{m_1-1}, S_2^{m_2-1}) & c_1^{m_1} = c_2^{m_2} \\ 1 + \min \begin{cases} lev(S_1^{m_1}, S_2^{m_2-1}) & \text{insert} \\ lev(S_1^{m_1-1}, S_2^{m_2}) & \text{delete} \\ lev(S_1^{m_1-1}, S_2^{m_2-1}) & \text{replace} \end{cases} & \text{otherwise.} \end{cases}$$

Schritt 2 - Levenshtein Distanz - DP

Warum ist die Rekurrenz korrekt?

Fall 1: m_1, when $m_2 = 0$
Wir brauchen m_1 delete-Operationen um den Unterschied auszugleichen.

Fall 2: m_1, when $m_2 = 0$
Wir brauchen m_1 insert-Operationen um den Unterschied auszugleichen.

Fall 3: $lev(S_1^{m_1-1}, S_2^{m_2-1})$, when $c_1^{m_1} = c_2^{m_2}$
Wenn die letzten beiden Buchstaben($c_1^{m_1}$, $c_2^{m_2}$) gleich sind, dann ändert sich die Distanz nicht.

Schritt 2 - Levenshtein Distanz - DP

Warum ist die Rekurrenz korrekt?

Fall 4:
Tritt keiner der vorherigen Fälle ein müssen wir der String verändern und somit wird die Distanz um 1 größer. Die Frage ist aber wie sich die zwei Teilsrings $S_1^{m_1}$ und $S_2^{m_2}$ verändern.

insert Wenn eine Insert-Operation gemacht wird, ist die Distanz die Gleiche wie für ein Zeichen von S_2 weniger. Dies liegt daran, dass das letzte Zeichen($c_2^{m_2}$) von S_2^{m-1} an $S_1^{m_1}$ gehängt wird.

delete Wenn eine Delete-Operation gemacht wird ist das äquivalent zu insert, wir machen aber $S_1^{m_1}$ kürzer.

replace Da der Wert $c_1^{m_1}$ in S_1 durch $c_2^{m_2}$ ersetzt wird, schauen wir uns die Levenshtein Distanz der kürzeren Strings an, da der aktuelle Wert abgeglichen wurde.

Schritt 3 - Levenshtein Distanz - DP

In der Rekursion von $lev(S_1^{m_1}, S_1^{m_2})$ haben wir folgende Zwischenergebnisse:

- $lev(S_1^{m_1}, S_2^{m_2-1})$
- $lev(S_1^{m_1-1}, S_2^{m_2})$
- $lev(S_1^{m_1-1}, S_2^{m_2-1})$

Diese müssen also vor $lev(S_1^{m_1}, S_2^{m_2})$ berechnet werden.

Zwei Teilstrings von $(S_1, S_2) \rightsquigarrow$ 2-dimensionalem Array.

Schritt 3 - Levenshtein Distanz - DP

Beispiel:

S_1 `verstecken`

S_2 `erschrecken`

S_1 \ S_2	-	e	r	s	c	h	r	e	c	k	e	n
-	0	1	2	3	4	5	6	7	8	9	10	11
v	1											
e	2											
r	3											
s	4											
t	5											
e	6											
c	7											
k	8											
e	9											
n	10											

Schritt 3 - Levenshtein Distanz - DP

$\textsc{lev}(S_1 : [char; m_1], S_2 : [char; m_1]) : \mathbb{N}$

$\quad lev : [[char; m_2]m_1] = \langle\langle 0, \ldots, 0\rangle, \ldots, \langle 0, \ldots, 0\rangle\rangle$

$\quad$ **for** $k = 0$ to m_1 **do**

$\quad\quad lev[k][0] = k$

$\quad$ **for** $l = 0$ to m_2 **do**

$\quad\quad lev[0][l] = l$

Schritt 3 - Levenshtein Distanz - DP

```
LEV(S_1 : [char; m_1], S_2 : [char; m_1]) : ℕ
    lev : [[char; m_2]m_1] = ⟨⟨0, ..., 0⟩, ..., ⟨0, ..., 0⟩⟩
    for k = 0 to m_1 do
        lev[k][0] = k
    for l = 0 to m_2 do
        lev[0][l] = l
    for k = 1 to m_1 do
        for l = 1 to m_2 do
            if S_1^{k-1} = S_2^{l-1} then
                lev[k][l] = lev[k − 1][l − 1]
```

Schritt 3 - Levenshtein Distanz - DP

```
LEV(S_1 : [char; m_1], S_2 : [char; m_1]) : ℕ
    lev : [[char; m_2]m_1] = ⟨⟨0, ..., 0⟩, ..., ⟨0, ..., 0⟩⟩
    for k = 0 to m_1 do
        lev[k][0] = k
    for l = 0 to m_2 do
        lev[0][l] = l
    for k = 1 to m_1 do
        for l = 1 to m_2 do
            if S_1^{k-1} = S_2^{l-1} then
                lev[k][l] = lev[k − 1][l − 1]
            else
                lev[k][l] = 1+
                min(lev[k − 1][l − 1], lev[1][l − 1], lev[k − 1][l])
    return lev[m_1][m_2]
```

Schritt 3 - Levenshtein Distanz - DP

S_1 \ S_2	-	e	r	s	c	h	r	e	c	k	e	n
-	0	1	2	3	4	5	6	7	8	9	10	11
v	1	0	1	2	3	4	5	6	7	8	9	10
e	2	1	1	2	3	4	5	5	6	7	8	9
r	3	2	1	2	3	4	4	5	6	7	8	9
s	4	3	2	1	2	3	4	5	6	7	8	9
t	5	4	3	2	2	3	4	5	6	7	8	9
e	6	5	4	3	3	3	4	4	5	6	7	8
c	7	6	5	4	3	4	4	5	4	5	6	7
k	8	7	6	5	4	4	5	5	5	4	5	6
e	9	8	7	6	5	5	5	5	6	5	4	5
n	10	9	8	7	6	6	6	6	6	6	5	4

Alles klar soweit?

Viel Erfolg in der Klaussur :)

Karlsruher Institut für Technologie

Fragen?