

Kap. 5: Graphalgorithmen

Graphen

Traversieren von Graphen

Topologisches Sortieren

Starke Zusammenhangskomponenten

Spannbäume

Kürzeste Pfade

Weiterführende Graphalgorithmen im Überblick

5.1 Graphen

- Graph: $G = (V, E)$

- V = Menge der **Knoten**, E = Menge der **Kanten**, $|V| = N$, $|E| = M$

- **ungerichtet**: $E \subseteq \{\{x, y\} \mid x, y \in V, x \neq y\}$ keine Schleifen

- **gerichtet**: $E \subseteq \{(x, y) \mid x, y \in V\}$ Schleifen

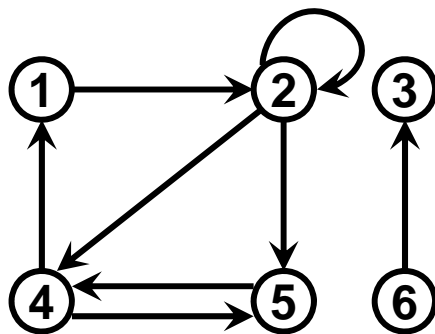
- **Multigraph**: ungerichtet mit multiplen Kanten + Schleifen

- **Hypergraph**: $E \subseteq 2^V$

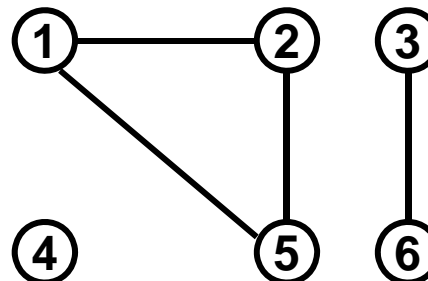
- Knoten w ist **adjacent** zu Knoten v falls $\{v, w\} \in E$, bzw. falls $(v, w) \in E$

- Kante e ist **inzident** mit Knoten v : $v \in e$

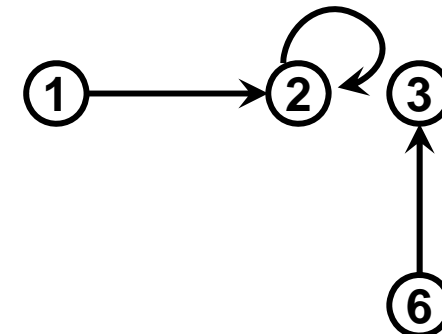
- ◆ gerichteter Graph: $e = (v, w)$, e ist **inzident von** v und **inzident nach** w



(a)



(b)



(c)

Graphen

- **Grad** eines Knotens: $\deg(v) = |\{e \in E \mid v \in e\}|$
 - gerichtet: $\text{indeg}(v) = |\{e \in E \mid e = (x,v)\}|$,
 $\text{outdeg}(v) = |\{e \in E \mid e = (v,x)\}|$, $\deg(v) = \text{indeg}(v) + \text{outdeg}(v)$
- **Pfad** der Länge k : (v_0, v_1, \dots, v_k) , $v_i \in V$, $(v_{i-1}, v_i) \in E$ für alle $1 \leq i \leq k$
 - einfacher Pfad: $v_i \neq v_j$ für alle $i \neq j$, $1 \leq i, j \leq k$
 - Teilpfad: (v_i, \dots, v_j) mit $1 \leq i < j \leq k$
 - Kreis / Zyklus: $v_0 = v_k$
 - Graph ohne Kreise: azyklisch
- **Erreichbarkeit**:
 - v ist **erreichbar** von w , g.d.w. es existiert ein Pfad von w nach v
 - ungerichteter Graph $G=(V,E)$ ist **verbunden** (connected), g.d.w. $\forall v,w \in V \exists$ Pfad von v nach w
 - gerichteter Graph $G=(V,E)$ ist **stark verbunden** (strongly connected), g.d.w. $\forall v,w \in V \exists$ Pfad von v nach w und von w nach v

Graphen

■ Subgraph:

- $G'=(V',E')$ ist ein **Subgraph** von $G=(V,E)$, falls $V' \subseteq V$, $E' \subseteq E$
- $G'=(V', E')$ ist der durch V' induzierte Subgraph, falls
$$E' = \{(u,v) \in E \mid u, v \in V'\}$$

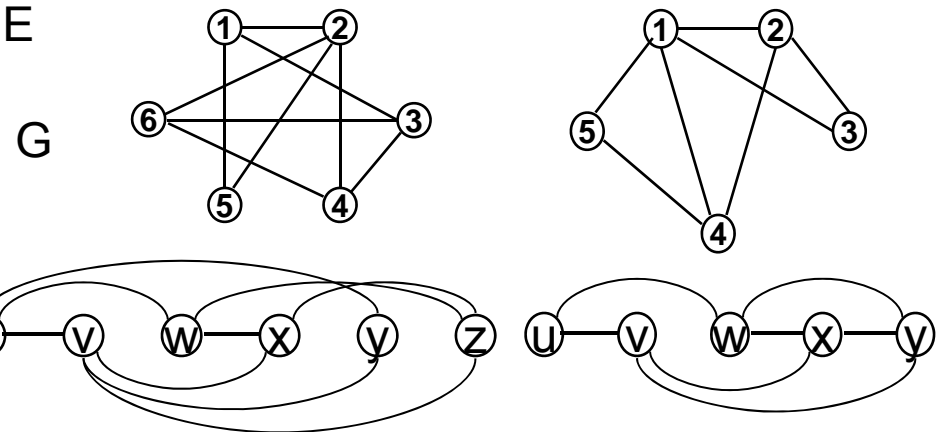
■ Zusammenhang:

- **Zusammenhangskomponente** (connected component): maximal zusammenhängender Subgraphen eines ungerichteten Graphen
- **starke Zusammenhangskomponente** (strongly connected component): analog für gerichtete Graphen

■ Isomorphie:

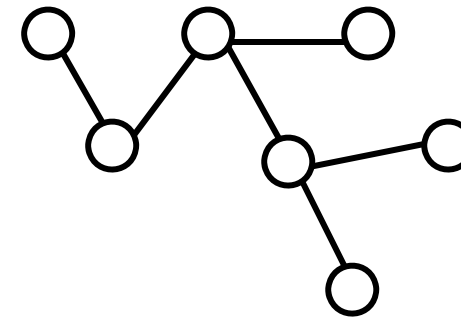
- G und G' sind **isomorph**, falls es eine bijektive Abbildung $f: V \rightarrow V'$ gibt mit $(f(v),f(w)) \in E'$ g.d.w. $(v,w) \in E$

- a) isomorph
($1=u, 2=v, \dots, 6=z$)
- b) nicht isomorph

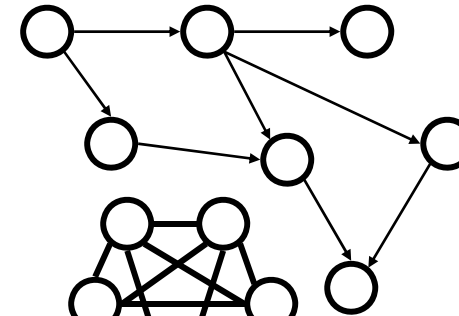


Spezielle Graphen

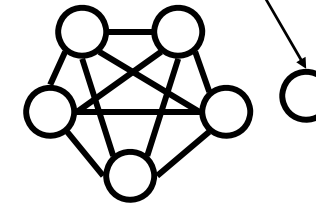
- **Baum (tree):**
 - ungerichteter, zusammenhängender Graph ohne Kreise
 - gewurzelt (rooted): ausgezeichnete Wurzelknoten
- **Wald (forest):**
 - ungerichteter Graph ohne Kreise
- **dag (directed acyclic graph):**
 - gerichteter Graph ohne Kreise
- **vollständiger Graph (complete graph):**
 - ungerichteter Graph, in dem alle Paare von Knoten adjazent sind
- **bipartiter Graph:**
 - Knotenmenge lässt sich in zwei disjunkte Teilmengen U, W aufteilen, $\forall (v, w) \in E : v \in U \text{ und } w \in W \text{ oder } v \in W \text{ und } w \in U$
- **planarer Graph:**
 - Graph lässt sich ohne Kantenüberschneidungen zeichnen



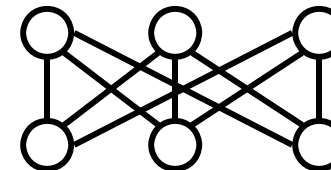
tree



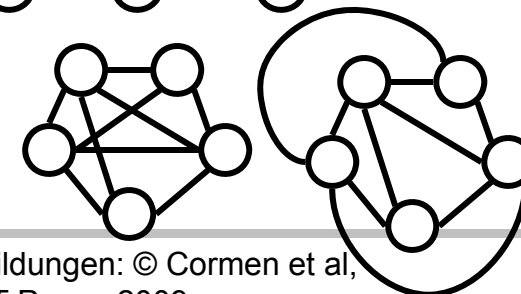
dag



vollst.
Graph



bipartiter
Graph

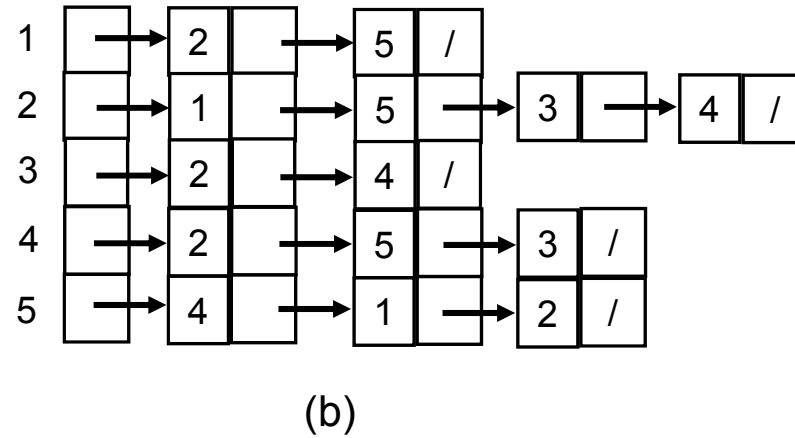
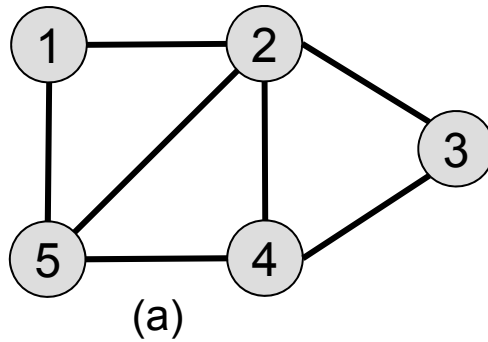


planarer
Graph

Repräsentation von Graphen

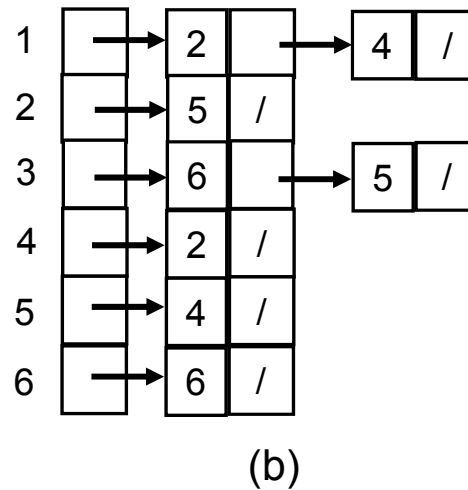
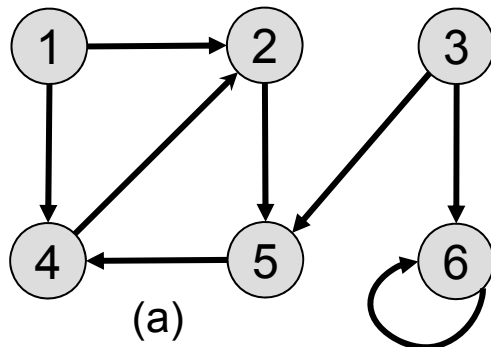
- Knoten: Array oder Liste, nummeriert von 1 bis N
- Kanten:
 - **Adjazenzmatrix:** $N \times N$ Matrix
 - ◆ $A[i,j] = 1$ falls $(i,j) \in E$, bzw $\{i,j\} \in E$; $A[i,j] = 0$ sonst
 - ◆ Speicherbedarf $\Theta(N^2)$
 - ◆ ungerichtete Graphen: $A[i,j] = A[j,i]$ (Matrix ist symmetrisch)
 - ◆ gerichtete Graphen: $A^T = ([a_{ij}])^T = [a_{ji}]$ ist die Adjazenzmatrix des transponierten Graphen (Umkehr aller Kanten)
 - ◆ Effizienter Test auf das Vorhandensein einer Kante
 - **Adjazenzlisten:**
 - ◆ Pro Knoten v eine Liste mit Nummern der zu v adjazenten Knoten:
$$v.\text{Adj} = \{ w \in V \mid \{v,w\} \in E, \text{ bzw. } (v,w) \in E \}$$
 - ◆ Speicherbedarf $\Theta(N+M)$
 - ◆ genauer: $2M$ falls G ungerichtet, M falls G gerichtet ist
 - ◆ Effizientes Durchlaufen aller zu einem Knoten v adjazenten Knoten w
- Kanten und Knoten tragen anwendungsspezifische Information
 - **gewichtete Graphen:** $w: E \rightarrow \mathbb{R}$ (jede Kante hat ein Gewicht)
 - **knotengewichtete Graphen:** $w: V \rightarrow \mathbb{R}$

Repräsentation von Graphen



Adjacency matrix representation (c) for graph (a):

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0



Adjacency matrix representation (c) for graph (a):

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

5.2 Traversieren von Graphen

- Ziel:
 - besuche alle Knoten eines Graphen in einer systematischer Reihenfolge
 - Grundgerüst für spätere Graphalgorithmen
- **Breitensuche** (breadth first search):
 - Alle Knoten werden zunächst als weiß markiert (noch nicht besucht)
 - Zum Zeitpunkt der ersten Entdeckung werden diese grau gefärbt.
 - Ein Knoten wird schwarz, wenn alle adjazenten Knoten bereits entdeckt wurden (d.h. nicht mehr weiß sind).
 - Start an einem beliebigen Knoten s
 - Besuche iterativ die noch nicht besuchten Nachbarn von s , dann die Nachbarn der Nachbarn von s , usw.
 - Realisierung durch Warteschlange
 - Vorgänger / Vater von u : wird u erstmals über die Kante (v,u) besucht, wird v als der Vorgänger von u bezeichnet
 - Die Eigenschaft ‚Vorgänger von‘ beschreibt einen Baum (BFS-Baum) mit Wurzel s
 - Distanz von u : Pfadlänge von der Wurzel s bis u im BFS-Baum

Breitensuche (breadth-first-search)

■ BFS(G, s)

```
1  for each  $u \in G.V \setminus \{s\}$ 
2     $u.color = WEISS$ 
3     $d[u] = \infty; \pi[u] = NIL$ 
5   $s.color = GRAU$ 
6   $d[s] = 0; \pi[s] = NIL$ 
8   $Q = \emptyset; ENQUEUE(Q, s)$ 
10 while  $Q \neq \emptyset$  do
11    $u = DEQUEUE(Q)$ 
12   for each  $v \in u.Adj$ 
13     if ( $v.color == WEISS$ )
14        $v.color = GRAU$ 
15        $d[v] = d[u] + 1; \pi[v] = u$ 
17        $ENQUEUE(Q, v)$ 
18    $u.color = SCHWARZ$ 
```

■ $u.color$:

- weiß: noch nicht besucht
- grau: besucht, hat noch nicht besuchte Nachbarn
- schwarz: besucht, alle Nachbarn besucht

■ $d[u]$:

- Pfadlänge von u zu s

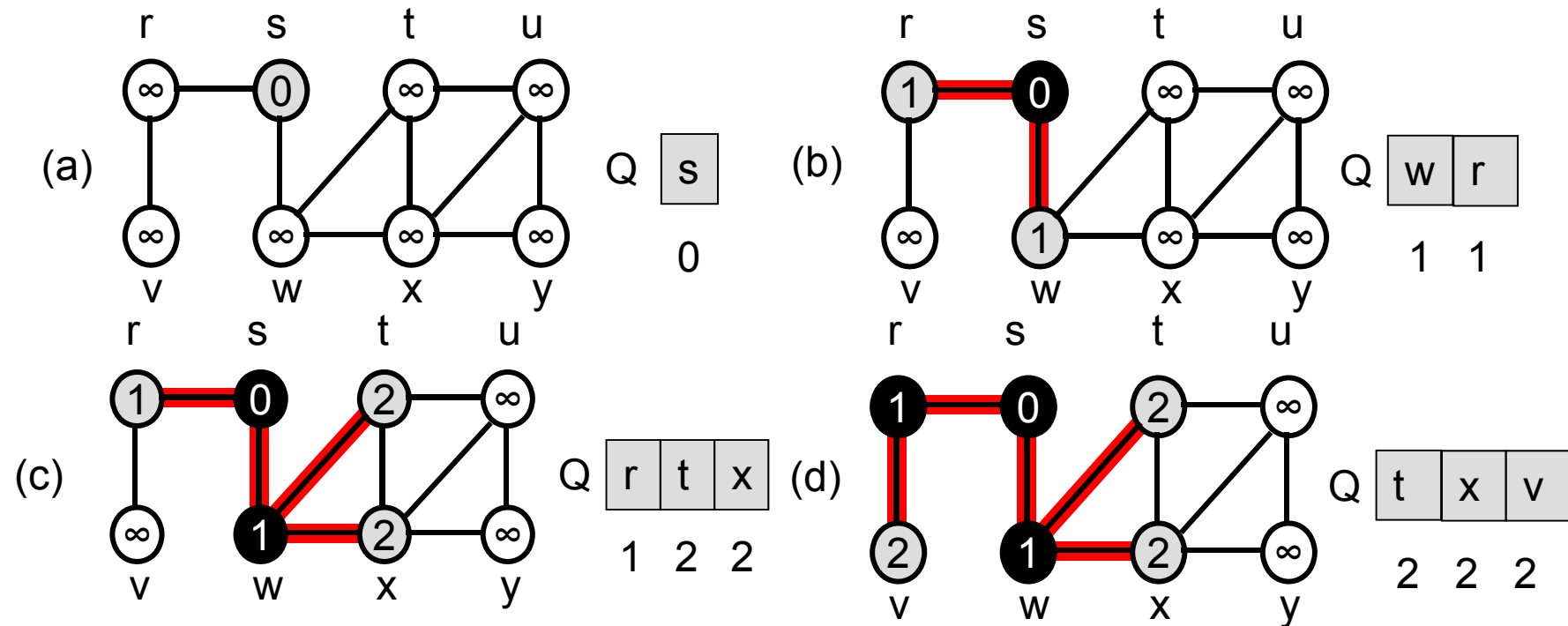
■ $\pi[u]$:

- Vorgänger von u

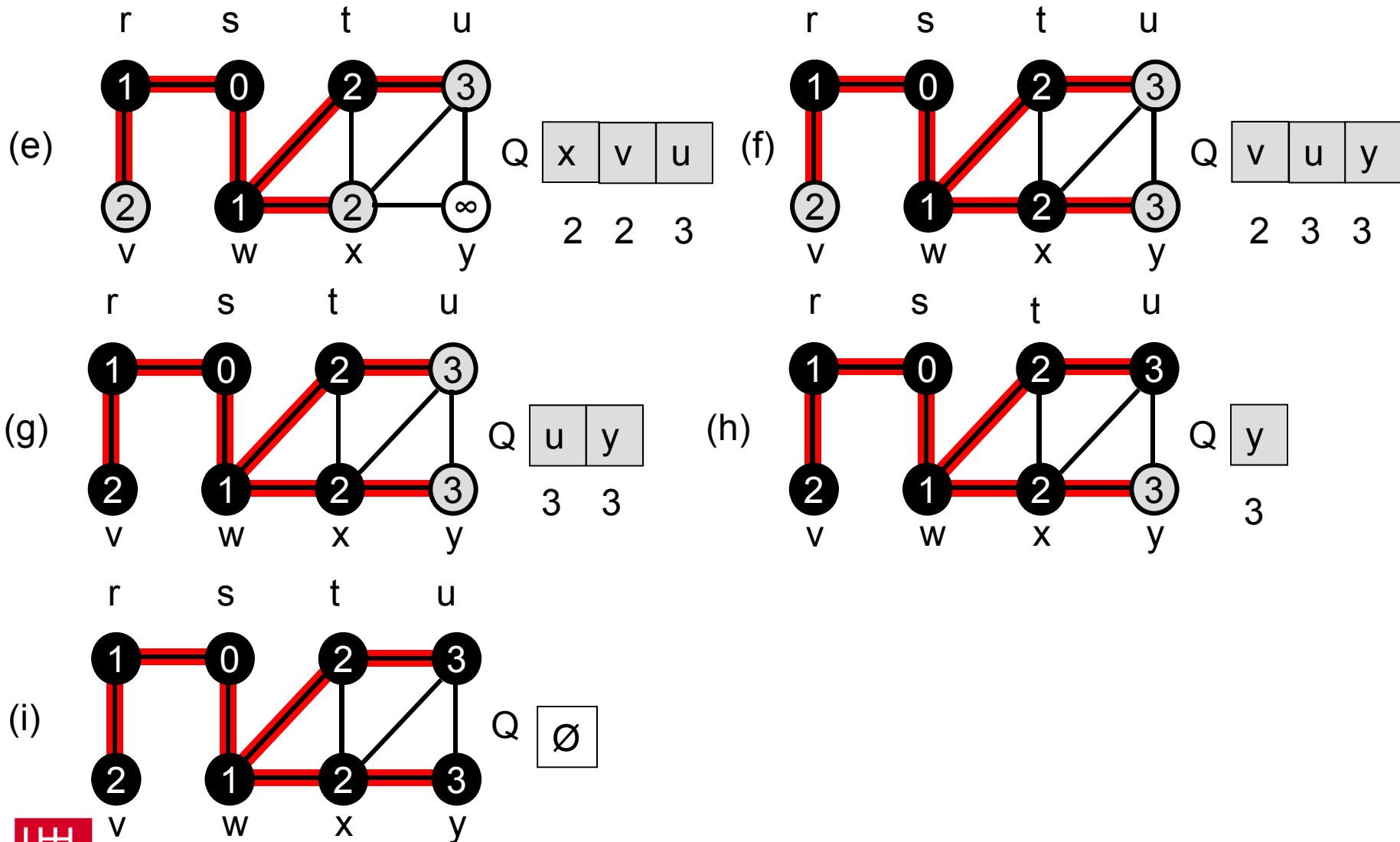
■ Warteschlange Q :

- alle grauen Knoten, Suchfront

Breitensuche: Beispiel



Breitensuche: Beispiel



■ BFS-Baum: (Vorgängerteilgraph)

- Kanten $E_\pi = \{(\pi(u), u) \mid u \in V, \pi(u) \in V\}$ bezeichnen die Kanten, über die die Knoten erstmalig besucht werden.
- Sei $V_\pi = \{u \in V \mid \pi(u) \neq \text{NIL}\} \cup \{s\}$ die während des BFS besuchten Knoten
- Graph $G_\pi = (V_\pi, E_\pi)$ beschreibt einen Baum mit Wurzel s :
 - ◆ G_π ist zusammenhängend: Für alle Knoten v existiert ein Pfad von v zur Wurzel über die $(u, \pi(u))$ -Kanten
 - ◆ G_π enthält keine Kreise: jeder Knoten wird nur einmalig grau gefärbt, zu diesem Zeitpunkt wird eine Kante zu einem Vorgänger eingefügt.
- Die asymptotische Laufzeit von $\text{BFS}()$ ist $O(N+M)$.
 - Initialisierung: $O(|V|)$
 - Jeder Knoten wird einmalig in Q eingefügt und wieder entnommen: $O(|V|)$
 - In der while-Schleife, wird jede Adjazenzliste einmal durchlaufen:

$$T_{\text{BFS}}(V, E) = c|V| + \sum_{v \in V} \sum_{(v, u) \in E} c = c|V| + 2c|E| = O(|V| + |E|)$$

Breitensuche: Bestimmung kürzester Pfade

- Sei $\delta(s,u)$ die minimale Anzahl Kanten eines Pfades von s nach u . Existiert kein solcher Pfad sei $\delta(s,u)=\infty$. $\delta(s,u)$ wird als der **kürzeste Abstand**, der Pfad von s nach u als **kürzester Pfad** bezeichnet.
- **Lemma 22.1:** (Struktur kürzester Pfade)
 - Sei $G = (V,E)$ ein Graph, $s \in V$. Für alle Kanten $e = (u,v) \in E$ gilt:
 $\delta(s,v) \leq \delta(s,u) + 1$.
 - Beweis:
 - Fall 1: $\delta(s,u) = \infty$
Dann ist u nicht erreichbar von s und somit auch v nicht, d.h. $\delta(s,v) = \infty$
 - Fall 2: $\delta(s,u) < \infty$
Es gibt einen Pfad von s zu u und dann über Kante e zu v . Da $\delta(s,v)$ den kürzesten Abstand beschreibt, gilt $\delta(s,v) \leq \delta(s,u) + 1$
- **Lemma 22.2:** (BFS beschränkt $\delta(s,u)$ von oben)
 - Sei $G=(V,E)$ ein Graph, $s \in V$, $d[\cdot]$ durch BFS berechnet, dann gilt $d[v] \geq \delta(s,v)$.

Breitensuche: Bestimmung kürzester Pfade

- Beweis: (durch Induktion über Einfüge-Reihenfolge in Q)

Annahme: $d[v] \geq \delta(s,v) \forall v \in V$

Induktionsanfang: $d[s] = 0 = \delta(s,s)$ und $d[v] = \infty \geq \delta(s,v)$.

Induktionsschritt: Betrachte den Zeitpunkt, in den v über Kante (u,v) in Q eingefügt wird:

$$\begin{aligned} d[v] &= d[u] + 1 && \text{(Zeile 15 des BFS)} \\ &\geq \delta(s,u) + 1 && \text{(Induktionsannahme)} \\ &\geq \delta(s,v) && \text{(Lemma 22.1)} \end{aligned}$$

- **Lemma 22.3:** Struktur von Q

Sei $G(V,E)$ ein Graph, $s \in V$, $d[]$ durch BFS berechnet, $Q = (v_1, \dots, v_r)$ die Warteschlange im BFS. Zu jedem Zeitpunkt gilt $d[v_1] \leq d[v_2] \leq \dots \leq d[v_i] \leq d[v_{i+1}] \leq \dots \leq d[v_r] \leq d[v_1] + 1$.

- Beweis: (durch Induktion über die Operationen auf Q)

Induktionsanfang: $Q = (s)$ mit $d[s] = 0$.

Breitensuche: Bestimmung kürzester Pfade

■ Beweis: (Fortsetzung)

Induktionsschritt:

Fall 1: ausgeführte Operation war DEQUEUE()

- ◆ v_2 wird neuer Kopf der Liste. Da $d[v_2] \geq d[v_1]$, folgt mit der Induktionsannahme das Lemma.

Fall 2: ausgeführte Operation war ENQUEUE()

- ◆ Sei v der neu eingefügte Knoten v_{r+1} , u der zuvor aus Q entnommene Knoten. Dann gilt $d[v] = d[u] + 1$ (Zeile 15).
- ◆ Nach Induktionsannahme gilt $d[u] \leq d[v_1]$. Es folgt $d[v_{r+1}] = d[v] = d[u] + 1 \leq d[v_1] + 1$. Zudem gilt $d[v_r] \leq d[u] + 1 = d[v_{r+1}]$.

■ Struktur von Q :

- Die $d[]$ -Werte der in Q gespeicherten Knoten sind monoton steigend.
- Da die $d[]$ -Werte diskret sind, gibt es unter den gespeicherten Knoten maximal zwei mögliche $d[]$ -Werte: $d[\text{HEAD}(Q)]$ und $d[\text{HEAD}(Q)] + 1$.

Breitensuche: Bestimmung kürzester Pfade

■ **Theorem 22.5:** Sei $G=(V,E)$ ein Graph, $s \in V$. Nach Durchführung von $\text{BFS}()$ gilt für alle Knoten v , die von s aus erreichbar sind:

1. $v.\text{color} = \text{SCHWARZ}$
2. $d[v] = \delta(s,v)$ (d.h. $d[v]$ entspricht dem kürzesten Abstand)
3. Es gibt einen kürzesten Pfad von s nach v , der mit der Kante $(\pi[v],v)$ endet.

■ Beweis (Eigenschaft 2 durch Widerspruch)

Aus Lemma 22.1 folgt bereits $d[v] \geq \delta(s,v)$.

Annahme: Es gibt einen Knoten v mit $d[v] > \delta(s,v)$.

Offensichtlich gilt:

◆ $v \neq s$, da $d[s] = 0 = \delta(s,v)$

◆ v ist von s aus erreichbar, da sonst $d[v] = \infty = \delta(s,v)$ gilt.

Sei v gewählt mit $d[v] > \delta(s,v)$ und $\delta(s,v)$ minimal.

Sei u der Vorgänger von v auf dem kürzesten Pfad, d.h. $\delta(s,v) = \delta(s,u) + 1$.

Aufgrund der Wahl von v (Minimalität) gilt für u : $d[u] = \delta(s,u)$. Es folgt:

$$d[v] > \delta(s,v) = \delta(s,u) + 1 = d[u] + 1$$

Breitensuche: Bestimmung kürzester Pfade

- Beweis von Theorem 22.5 (Fortsetzung)

Betrachte den Zeitpunkt der Entnahme von u aus Q :


Fall 1: $v.color = WEISS$

$d[v] = d[u] + 1$ (lt. Zeile 15 des `BFS()`). 

Fall 2: $v.color = SCHWARZ$

v wurde bereits aus Q entfernt, nach Lemma 22.2 folgt $d[v] \leq d[u]$ 

Fall 3: $v.color = GRAU$

Sei w der Knoten, bei dessen Entnahme v grau gefärbt wurde. Es gilt $d[w] \leq d[u]$ (wg. Lemma 22.2; w wurde vor u entfernt) und zudem $d[v] = d[w] + 1 \leq d[u] + 1$. 

- Eigenschaft 1: folgt direkt aus Eigenschaft 2, da $d[v] = \infty$, falls v nicht bearbeitet wurde

- Eigenschaft 3: folgt direkt aus $d[v] = d[\pi(v)] + 1$

Tiefensuche (depth-first-search)

■ Strategie:

- Suche in die Tiefe, d.h. für jeden Knoten, arbeite zuerst den ersten Nachfolger vollständig ab, dann den zweiten, u.s.w
- Statt einer Schlange wird ein Stapel verwendet (durch Rekursion)
- Farben wie bei BFS:
 - ◆ weiß: noch nicht besucht
 - ◆ grau: besucht, Adjazenzliste noch nicht abgearbeitet
 - ◆ schwarz: vollständig abgearbeitet
- Diskrete Zeitstempel:
 - ◆ $d[u]$: Zeitpunkt des erstmaligen Eintrags (weiß \rightarrow grau, discovery time)
 - ◆ $f[u]$: Zeitpunkt der vollst. Abarbeitung (grau \rightarrow schwarz, finishing time)
 - ◆ Hinweise:
 - ◆ Offensichtlich gilt $d[u] < f[u]$
 - ◆ $d[u]$ beschreibt nun nicht mehr den Abstand zu s !
- DFS-Wald (Tiefensuchwald):
 - ◆ Kanten die bei erstmaligen Besuch durchlaufen werden, beschreiben eine Menge von Bäumen
 - ◆ $\pi[u]$: Vorgänger im DFS-Wald

Tiefensuche

■ DFS(G)

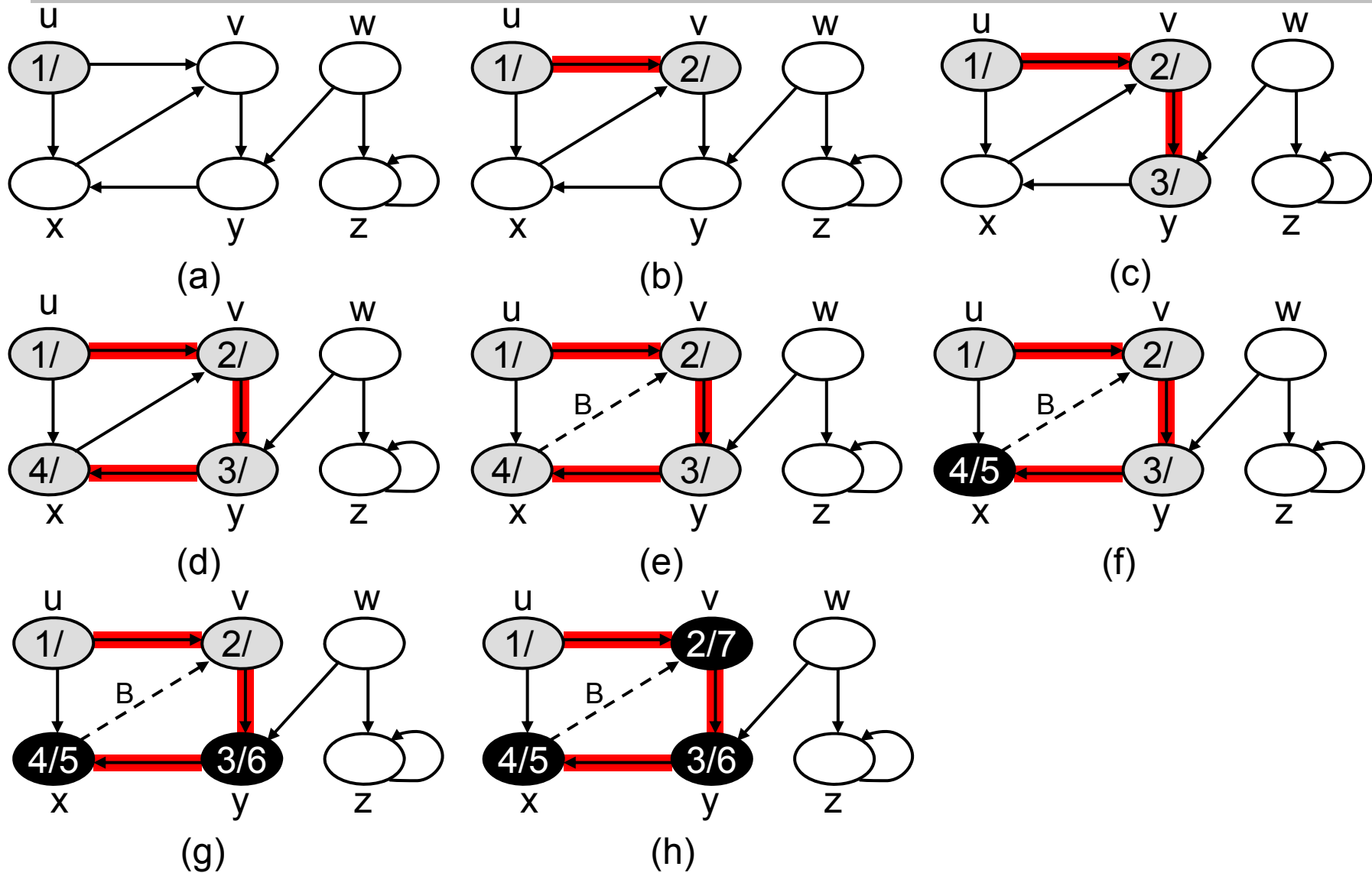
Laufzeit: $O(N+M)$

```
1 for each  $u \in G.V$ 
2    $u.color = WEISS; \pi[u] = NIL$ 
4  $time = 0$ 
5 for each  $u \in G.V$ 
6   if(  $u.color == WEISS$  ) DFS-VISIT( $u$ )
```

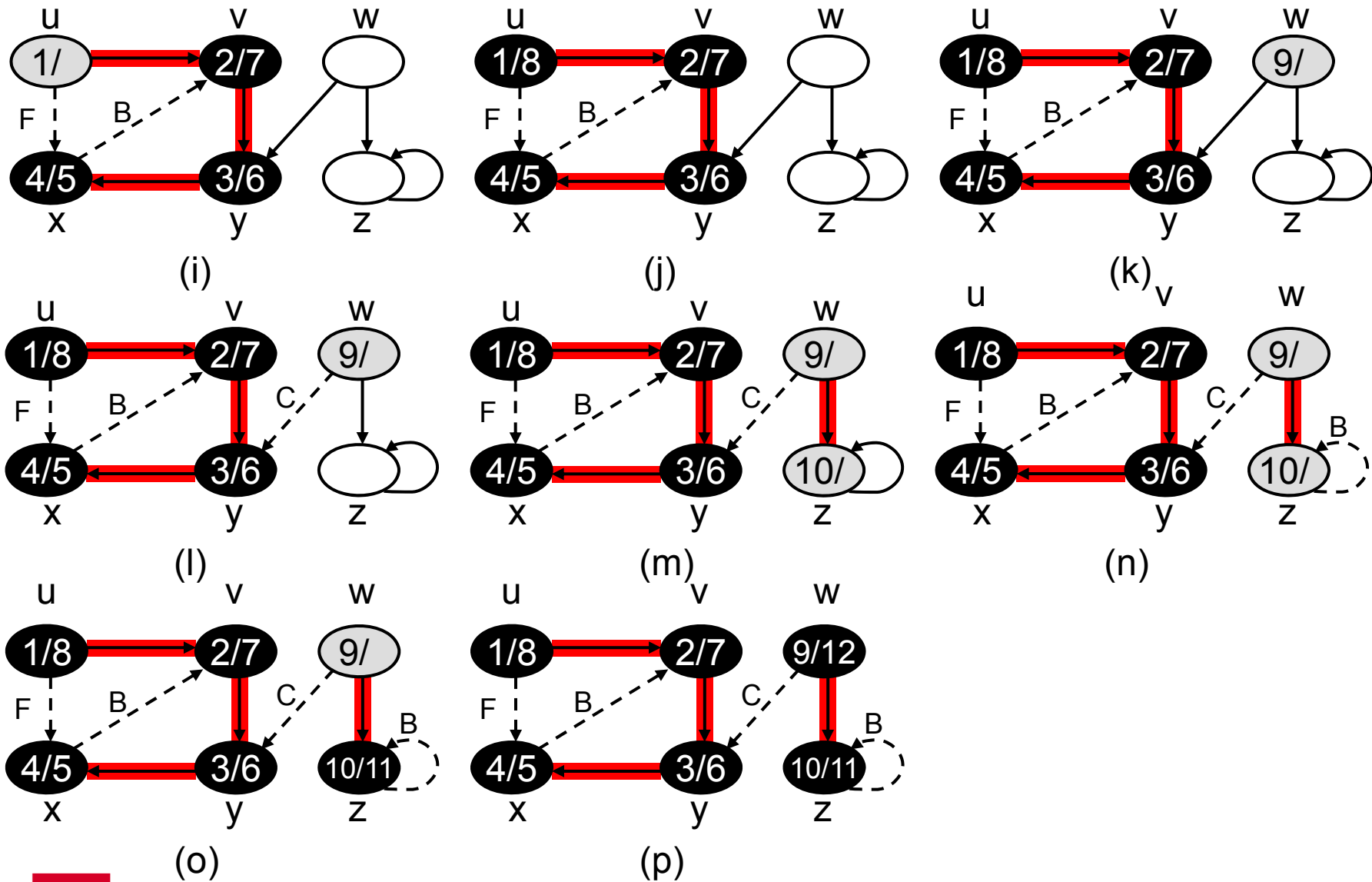
■ DFS-VISIT(u)

```
1  $u.color = GRAU$                                 // Entdeckung von  $u$ 
2  $time = time + 1; d[u] = time$ 
4 for each  $v \in u.Adj$                              // Sondierung der Adjazenzliste
5   if(  $v.color == WEISS$  )
6      $\pi[v] = u$ 
7     DFS-VISIT( $v$ )
8  $u.color = SCHWARZ$ 
9  $time = time + 1; f[u] = time$                     // Vollständige Abarbeitung von  $u$ 
```

Tiefensuche: Ein Beispiel



Tiefensuche: Ein Beispiel



■ **Theorem 22.7** (Klammerungstheorem):

Nach Durchführung von DFS gelten für die Entdeckungs- und Endzeiten $d[]$ und $f[]$ für je zwei Knoten u und v , sei $L(u) = [d[u], f[u]]$ das Zeitintervall zwischen Entdeckungs- und Endzeit von u :

1. $L(u) \cap L(v) = \emptyset$ und u ist im DFS-Wald weder Vorfahre noch Nachfahre von v
2. $L(u) \subset L(v)$ und u ist im DFS-Wald Nachfahre von v
3. $L(u) \supset L(v)$ und u ist im DFS-Wald Vorfahre von v

■ Beweis: O.B.d.A. nehmen wir $d[u] < d[v]$ an.

Fall 1: $d[v] < f[u]$: Zum Zeitpunkt $d[v]$ galt $u.\text{color} = \text{GRAU}$

=> u ist ein Vorfahre von v , da u noch nicht abgearbeitet ist.

=> alle Kanten von v werden sondiert, bevor u abgearbeitet wird, d.h.
 $f[v] < f[u]$

=> Fall 3 des Theorems.

Fall 2: $d[v] > f[u]$: Zum Zeitpunkt $d[v]$ galt $u.\text{color} = \text{SCHWARZ}$

=> u ist bereits abgearbeitet, somit ist u kein Vorfahre von v

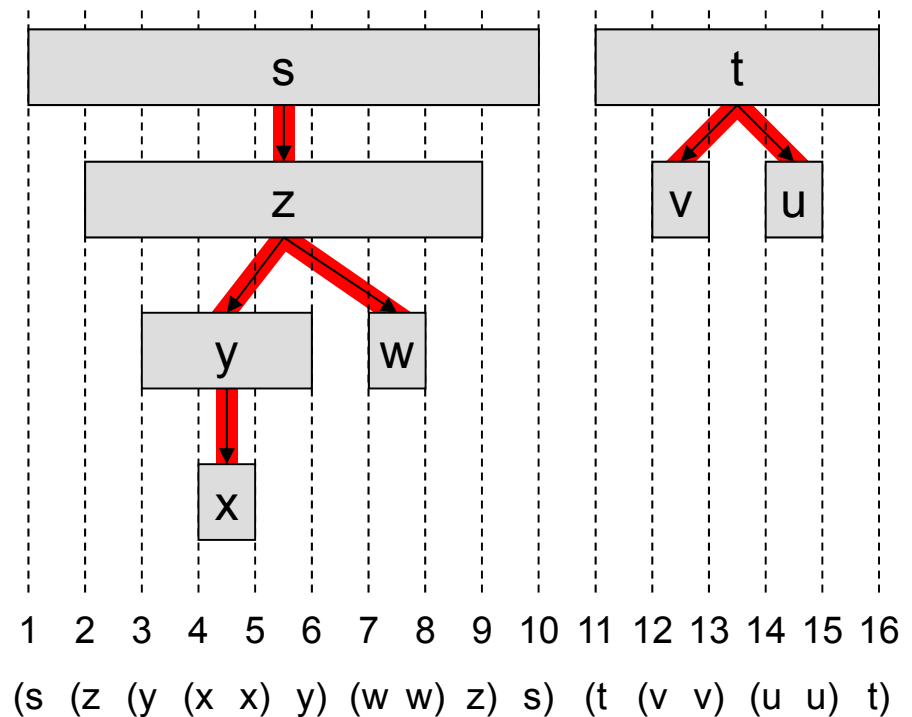
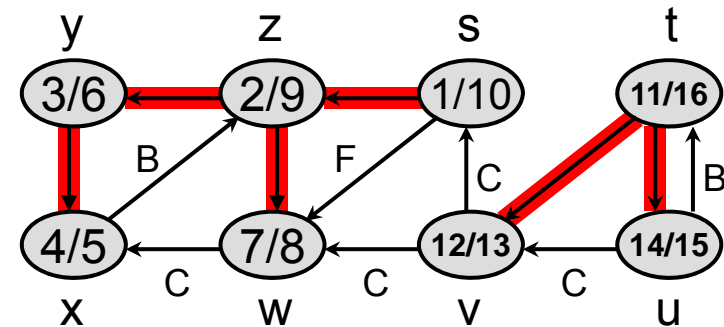
=> v wird nach u entdeckt, somit ist u kein Nachfahre von v

=> da $d[v] < f[v]$ und $d[u] < f[u] < d[v] < f[v]$

=> Fall 1 des Theorems.

Tiefensuche: Bedeutung des Klammerungstheorems

- Beispiel:
 - Knoten enthalten $d[v]/f[v]$
 - Baumkanten sind rot unterlegt
 - Startknoten ist s (dann t)
- Knoten können durch Intervalle auf der Zeitachse dargestellt werden.
- Intervalle sind entweder disjunkt oder vollständig enthalten.
- DFS-Wald lässt sich durch Intervallstruktur ableiten.
- ACHTUNG: der DFS-Wald ist nicht eindeutig:
 - Wahl des Startknotens
 - Reihenfolge der Kanten in den Adjazenzlisten



- **Theorem 22.9** (Weiße Pfade): In einem DFS-Wald ist v ein Nachfahre von u genau dann wenn zum Zeitpunkt $d[u]$ ein Pfad $(u, w_1, \dots, w_n=v)$ von u zu v existiert mit $w_i.\text{color} = \text{WEISS}$ für $i=1, \dots, n$.

- Beweis: Teil 1 (\Rightarrow) : v ist Nachfahre von u

Für alle Nachfahren w von u gilt: $d[w] > d[u]$ (Theorem 22.7) und somit $w.\text{color} = \text{WEISS}$. Damit existiert ein weißer Pfad (im DFS-Wald).

Teil 2 (\Leftarrow): Es existiert ein weißer Pfad $(u, w_1, \dots, w_{n-1}=w, w_n=v)$.

Annahme: v ist kein Nachfahre von u

Offensichtlich gilt $d[v] > d[u]$ (v ist weiß zum Zeitpunkt $d[u]$).

Sei o.B.d.A v entlang des Pfades der erste Knoten, der kein Nachfahre von u ist, d.h. w ist Nachfahre von u .

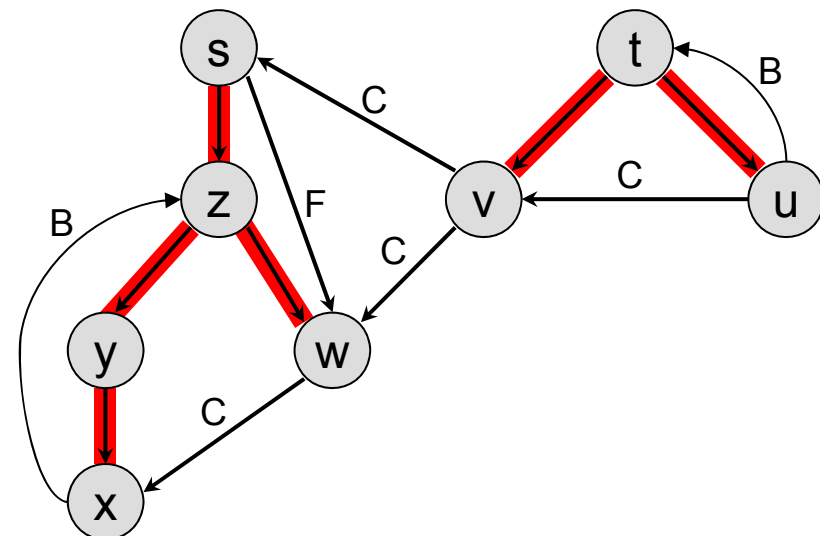
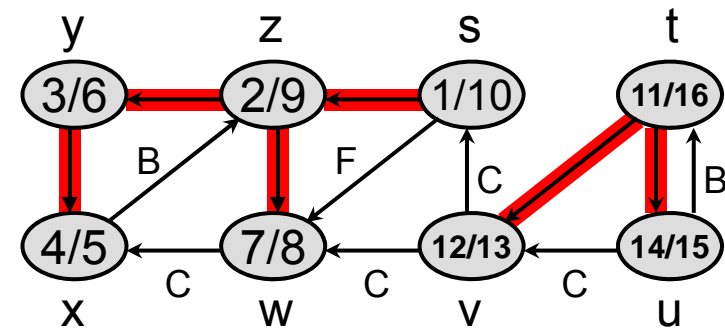
- ◆ $f[w] \leq f[u]$ (w ist Nachfahre von u und nach Theorem 22.7 Fall 2)
 - ◆ $d[v] < f[w]$ (wg. Kante (w,v) wird v entdeckt bevor w abgearbeitet ist)
- $\Rightarrow d[u] < d[v] < f[w] \leq f[u] \Rightarrow [d[u], f[u]] \supset [d[v], f[v]]$, bzw. $L(u) \supset L(v)$
- $\Rightarrow u$ ist Vorfahre von v , d.h. v ist Nachfahre von u . ⚡

Tiefensuche: Kantenklassifikation

■ Klassifikation von Kanten $e = (u,v)$:

1. **Baumkante**: wird durchlaufen beim ersten Besuch von v
[dicke rote Kanten]
2. **Rückkante**: zeigt auf einen Vorgänger im DFS-Baum
[Typ B]
3. **Vorwärtskante**: zeigt auf einen Nachfolger im DFS-Baum
[Typ F]
4. **Querkante**: nicht 1-3
[Typ C]

- Die Zuordnung der Kantentypen in einem DFS-Wald ist eindeutig, die Kantentypen sind jedoch NICHT eindeutig für den Graph G .



Tiefensuche: Kantenklassifikation

■ Klassifikation von Kanten $e = (u, v)$:

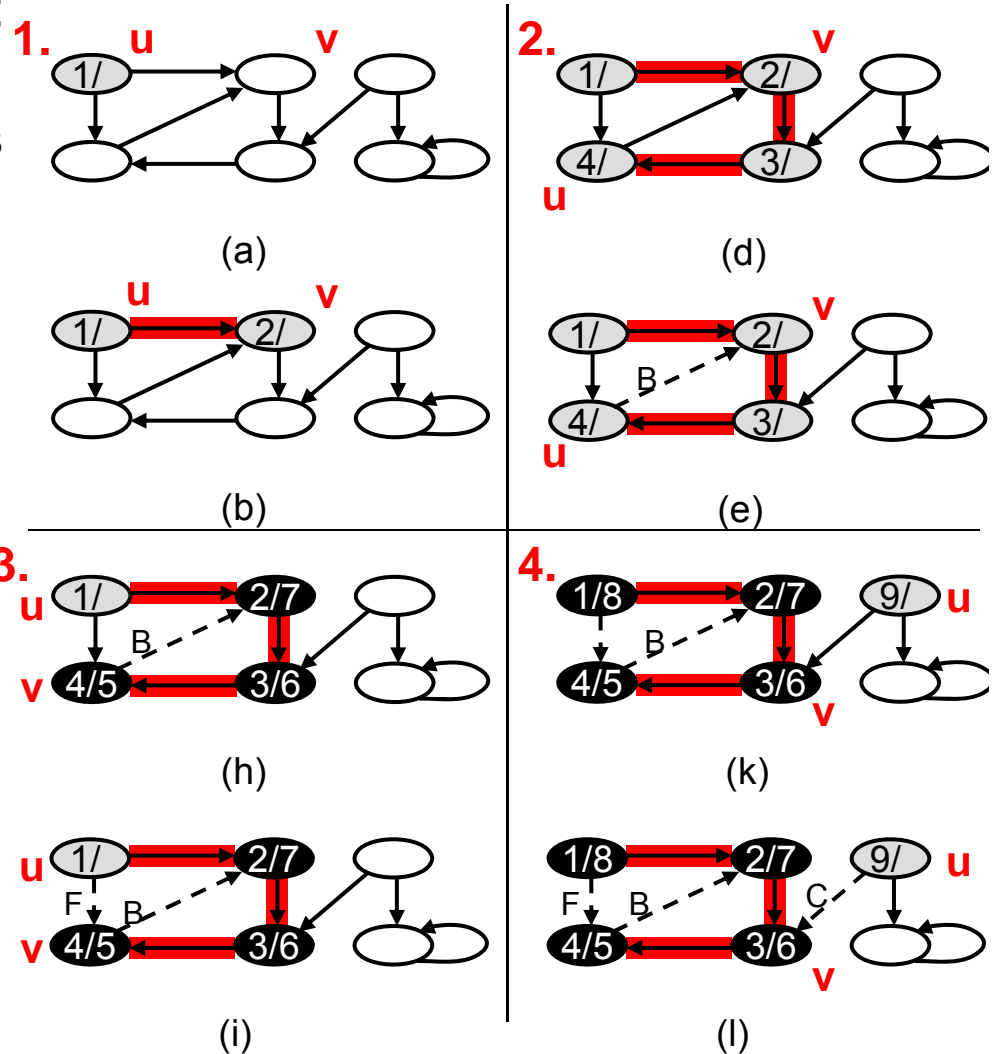
- Zum Zeitpunkt der Sondierung der Kante $e = (u, v)$ gilt zum einen, dass $u.color = \text{GRAU}$ ist, zum anderen:

1. $v.color = \text{WEISS} \Rightarrow$ **Baumkante**
2. $v.color = \text{GRAU} \Rightarrow$ **Rückkante**
3. $v.color = \text{SCHWARZ}$, $d[u] < d[v] \Rightarrow$ **Vorwärtskante**

[da: $d[u] < d[v] < f[v] < f[u]$ gilt, ist nach Theorem 22.7 Fall 2, u ist ein Vorfahre von v]

4. $v.color = \text{SCHWARZ}$, $d[u] > d[v] \Rightarrow$ **Querkante**

[da $d[v] < d[u]$ und $f[v] < f[u]$ gilt nach Theorem 22.7 Fall 1, dass u weder Vorfahre noch Nachfahre von v ist]



Tiefensuche mit Kantenklassifikation

■ DFS(G)


```
1 for each  $u \in G.V$ 
2    $u.color = WEISS; \pi[u] = NIL$ 
4  $time = 0$ 
5 for each  $u \in G.V$ 
6   if(  $u.color == WEISS$  ) DFS-VISIT( $u$ )
```

Laufzeit: $O(N+M)$

■ DFS-VISIT(u)

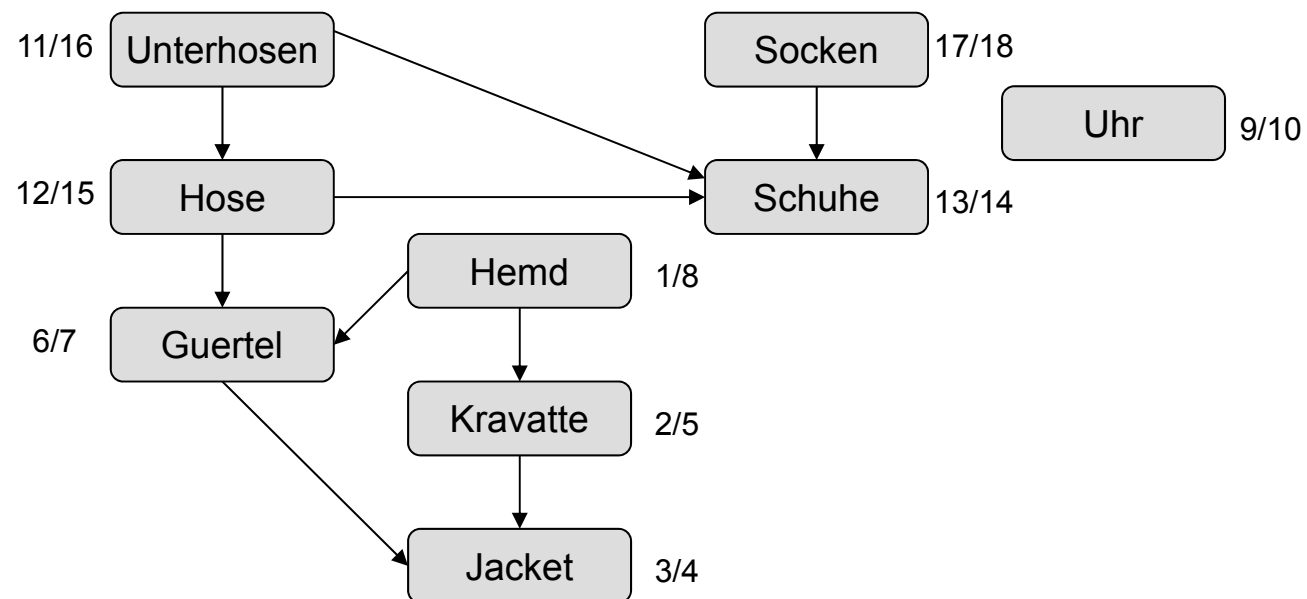
```
1  $u.color = GRAU$  // Entdeckung von  $u$ 
2  $time = time + 1; d[u] = time$ 
4 for each  $v \in u.Adj$  // Sondierung der Adjazenzliste
5   if(  $v.color == WEISS$  )
6      $type[(u,v)] = TREE; \pi[v] = u$ 
7     DFS-VISIT( $v$ ) // ACHTUNG: color[v] ändert sich!
8   else if(  $v.color == GRAU$  )  $type[(u,v)] = BACK$ 
9   else if(  $d[u] < d[v]$  )  $type[(u,v)] = FORWARD$ 
10  else  $type[(u,v)] = CROSS$ 
11  $u.color = SCHWARZ$ 
12  $time = time + 1; f[u] = time$  // Vollständige Abarbeitung von  $u$ 
```

Tiefensuche: Kantenklassifikation in ungerichteten Graphen

- Im Falle von ungerichteten Graphen wird jede Kante zwei mal betrachtet. Die Klassifikation wird bei der **ersten Sondierung** vorgenommen.
- Angenommen ein ungerichteter Graph $G=(V,E_u)$ wird durch einen gerichteten Graphen (V, E) mit $E = \{(u,v) \in V \times V \mid \{u,v\} \in E_u\}$ repräsentiert:
- Angenommen, (u,v) wird vor (v,u) sondiert, dann gilt:
 1. $\text{type}[(u,v)] = \text{TREE}$ $\Rightarrow \text{type}[(v,u)] = \text{BACK}$
 2. $\text{type}[(u,v)] = \text{BACK}$ $\Rightarrow \text{type}[(v,u)] = \text{FORWARD}$
 3. $\text{type}[(u,v)] = \text{FORWARD}$ oder $\text{type}[(u,v)] = \text{CROSS}$
 $\Rightarrow v.\text{color} = \text{SCHWARZ}$
 \Rightarrow alle Kanten von v wurden bereits sondiert
 $\Rightarrow (v,u)$ wurde vor (u,v) sondiert. 
- **Theorem 22.10:**
Ist G ein ungerichteter Graph, so ist jede Kante in G entweder eine Baum- oder Rückkante.


5.3 Topologisches Sortieren

- DAG (directed acyclic graph): gerichteter Graph ohne Kreise
- **Topologische Sortierung** der Knotenmenge V :
 - Sei $G=(V,E)$ ein gerichteter Graph. Eine bijektive Funktion $\pi: V \rightarrow \{1, \dots, |V|\}$ heißt topologische Sortierung, genau dann wenn für alle $e = (v,w) \in E$ gilt: $\pi(v) < \pi(w)$
- Anwendung der topologischen Sortierung:
 - Sortierung bei partieller Ordnung
 - Priorisierungs- und Schedulingprobleme:



Topologisches Sortieren: Existenz in DAGS

- **Lemma 22.x:** Sei $G=(V,E)$ ein gerichteter Graph. Für G existiert eine topologische Sortierung, genau dann wenn G keine Kreise enthält (d.h. ein DAG ist).

- Beweis: Teil 1 (\Rightarrow): Sei π eine topologische Sortierung für G . Angenommen $(v_1, v_2, \dots, v_r=v_1)$ sei ein Kreis in G . Es gilt $\pi(v_1) < \pi(v_2) < \dots < \pi(v_r) = \pi(v_1)$. 
- Teil 2 (\Leftarrow): Sei G ein DAG. Wir zeigen zunächst, dass ein Knoten $u \in V$ mit $\text{indeg}(u) = 0$ existiert: Sei $G^T = (V, E^T)$ der zu G transponierte Graph (d.h. alle Kanten werden umgekehrt). Betrachte den folgenden Algorithmus:

SEARCH-SOURCE(V)

$v = \text{select from } V \text{ arbitrarily}$

while $v.\text{Adj}^T \neq \emptyset$

$(v, w) = \text{HEAD}(v.\text{Adj}^T)$

$v = w$

SEARCH-SOURCE() terminiert, da jeder Knoten aus V maximal einmal besucht wird (beim zweiten Besuch hätten wir einen Kreis detektiert).

Topologisches Sortieren: Existenz in DAGs

■ Beweis: Teil 2 (\leq): (Fortsetzung)

Beweis durch Induktion über die Anzahl der Knoten $|V|$:

Induktionsanfang: Falls $|V| = 1$, ist $\pi(v) = 1$ eine gültige topologische Sortierung.

Induktionsschritt: Sei $u \in V$ mit $\text{indeg}(u) = 0$, $V' = V \setminus \{u\}$ und $G' = (V', E')$ der durch V' induzierte Subgraph. G' ist offensichtlich ein DAG mit $|V|-1$ Knoten. Nach Induktionsvoraussetzung existiert eine top. Sortierung π' .

Sei $\pi: V \rightarrow \{1, \dots, |V|\}$ definiert als:

$$\pi(v) = \begin{cases} 1 & : v = u \\ 1 + \pi'(v) & : v \in V' \end{cases}$$

- ◆ Für $e = (u, v) \in E$ gilt: $\pi(u) = 1 < 1 + \pi'(v) = \pi(v)$.
- ◆ Für $e = (v, w) \in E$ mit $v \neq u$ gilt: $\pi(v) = 1 + \pi'(v) < 1 + \pi'(w) = \pi(w)$, da $(v, w) \in E'$ und π' eine top. Sortierung von G' ist.
- ◆ Kanten $e = (v, u)$ existieren nicht, da $\text{indeg}(u) = 0$.

Folglich ist π eine top. Sortierung für G .

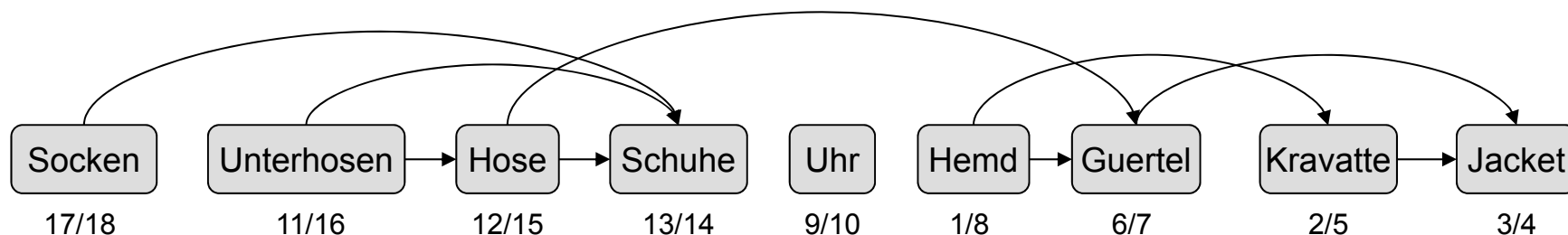
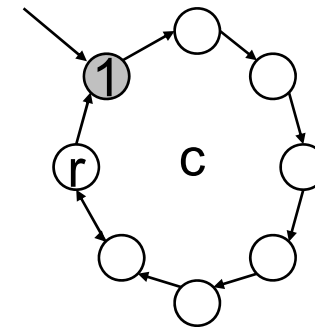
- In einem DAG werden Knoten mit $\text{indeg}()=0$ **Quellen** und mit $\text{outdeg}()=0$ **Senken** genannt.

Topologisches Sortieren: Erkennung von DAGs

■ **Lemma 22.11:** Ein ungerichteter Graph $G=(V,E)$ ist ein DAG, genau dann wenn ein DFS-Durchlauf keine Rückkante detektiert.

■ Beweis: Teil 1 (\Rightarrow): Sei G ein DAG. Angenommen es gibt eine Kante $e = (u,v)$ mit $\text{type}[(u,v)] = \text{BACK}$. v ist ein Vorgänger von u , daher existiert ein Pfad von v nach u und mit e ein Zyklus. ⚡

■ Teil 2 (\Leftarrow): Sei $c = (v_1, \dots, v_r, v_{r+1})$ ein Zyklus und v_1 der Knoten, der zuerst in einem DFS-Durchlauf entdeckt wird. Zum Zeitpunkt $d[v_1]$ gilt $v_i.\text{color} = \text{WEISS}$ für $i=2, \dots, r$. Nach dem Theorem der weißen Pfade sind v_2, \dots, v_r Nachfolger von v_1 . Wenn v_r abgearbeitet wird, gilt $v_1.\text{color} = \text{GRAU}$, somit ist (v_r, v_1) eine Rückkante.



Topologisches Sortieren: Berechnung mittels DFS()

■ **TOPOLOGICAL-SORT**(G)

```
1 for each  $u \in G.V$ 
2    $u.color = WEISS$ ;  $\pi[u] = NIL$ 
4  $time = 0$ ;  $L = LIST-INIT()$ 
5 for each  $u \in G.V$ 
6   if(  $u.color == WEISS$  )  $DFS-VISIT(u, L)$ 
7 return  $L$ 
```

■ **DFS-VISIT**(u, L)

```
1  $u.color = GRAU$  // Entdeckung von u
2  $time = time + 1$ ;  $d[u] = time$ 
4 for each  $v \in u.Adj$  // Sondierung der Adjazenzliste
5   if(  $v.color == WEISS$  )
6      $\pi[v] \leftarrow u$ 
7      $DFS-VISIT(v, L)$ 
8   else if(  $v.color == GRAU$  )  $ERROR$  „G is not a DAG!“
9    $u.color = SCHWARZ$ ;  $LIST-INSERT(L, u)$ 
10  $time = time + 1$ ;  $f[u] = time$  // Vollständige Abarbeitung von u
```

■ **Theorem 22.12:** Korrektheit von TOPOLOGICAL-SORT()

Sei $G=(V,E)$ ein DAG. TOPOLOGICAL-SORT(G) berechnet in L eine topologische Sortierung der Knoten in V .

- Beweis: L enthält alle Knoten $u \in V$ in der Reihenfolge fallender Endzeiten $f[u]$ (Knoten werden nach Bearbeitung jeweils vorne in L eingefügt).

Sei $e = (u,v) \in E$ eine beliebige Kante. Wir zeigen $f[u] > f[v]$ mit dem Klammerungstheorem (Theorem 22.7):

- ◆ Fall 1: $\text{type}[(u,v)] = \text{TREE}$
 $d[u] < d[v] < f[u] \Rightarrow f[u] > f[v]$.
- ◆ Fall 2: $\text{type}[(u,v)] = \text{FORWARD}$ oder $\text{type}[(u,v)] = \text{CROSS}$
Zum Zeitpunkt der Sondierung von e ist $v.\text{color} = \text{SCHWARZ}$ und $f[u]$ noch nicht gesetzt $\Rightarrow f[u] > f[v]$
- ◆ Fall 3: $\text{type}[(u,v)] = \text{BACK}$

Dieser Fall tritt nicht auf, da G ein DAG ist (Lemma 22.11).

Bzgl. der Ordnung π in L gilt somit für jede Kante $(u,v) \in E$: $\pi(u) < \pi(v)$.

5.4 Starke Zusammenhangskomponenten

■ Definition:

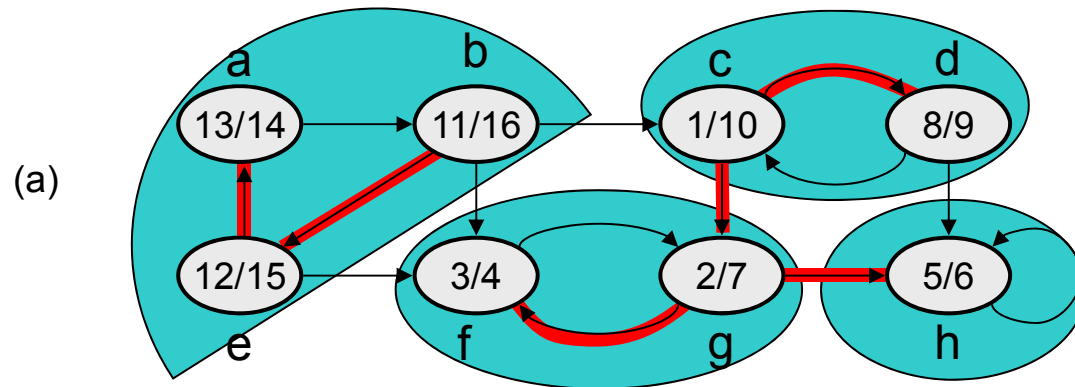
$u \rightsquigarrow v$ beschreibt die Relation ‚Es existiert ein Pfad von u nach v .‘.

Sei $G=(V,E)$ ein gerichteter Graph. Eine maximale Menge $C \subseteq V$, in der $\forall u,v \in C: u \rightsquigarrow v$ und $v \rightsquigarrow u$ gilt, heißt **starke Zusammenhangskomponente** (strongly connected component, SCC).

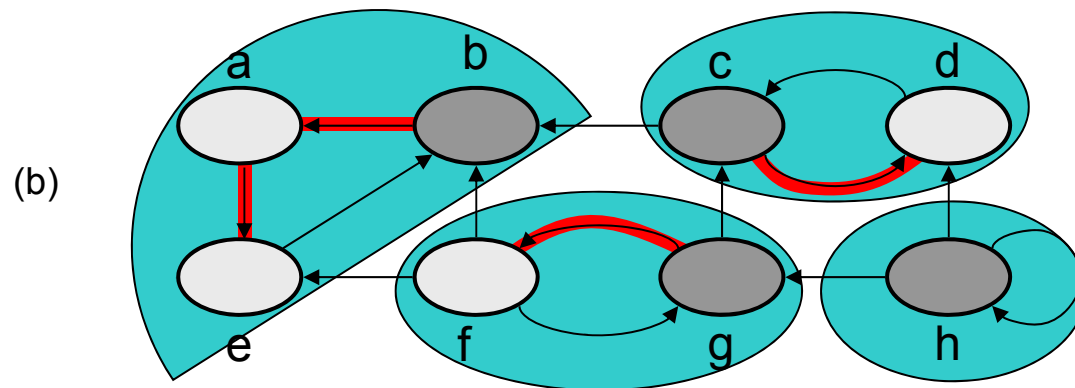
■ Lemma 22.x/22.13: Struktur von SCCs

1. Die Knotenmenge V zerfällt in eine Menge disjunkter starker Zusammenhangskomponenten.
 - ◆ Beweis: Seien C_1, C_2 SCCs mit $C_1 \cap C_2 \neq \emptyset, v \in C_1 \cap C_2$. Sei $u \in C_1$ und $w \in C_2$ beliebig gewählt. Dann gilt $u \rightsquigarrow v \rightsquigarrow w$ und $w \rightsquigarrow v \rightsquigarrow u$ und somit $C_1 = C_2$.
2. Die Graphen $G=(V,E)$ und $G^T = (V, E^T)$ mit $E^T = \{(u,v) : (v,u) \in E\}$ haben die gleichen starken Zusammenhangskomponenten.
3. Der **Komponentengraph** $G^{SCC}=(V^{SCC}, E^{SCC})$ ist definiert als $V^{SCC} =$ Menge der SCCs, $E^{SCC} = \{(C,D) \mid \exists u \in C \exists v \in D: (u,v) \in E\}$.
 G^{SCC} ist ein DAG.
 - ◆ Beweis: analog zu 1.

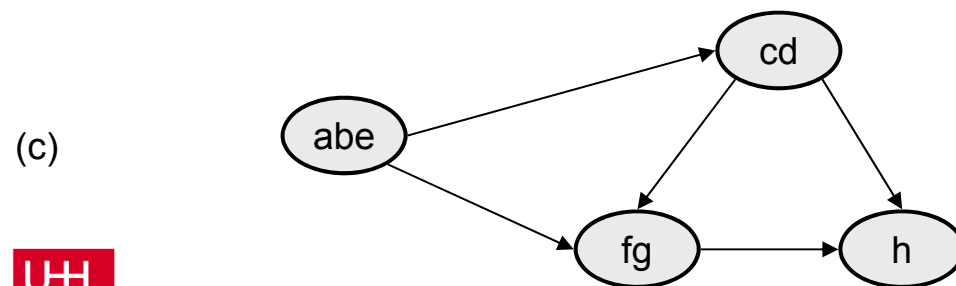
Starke Zusammenhangskomponenten: Ein Beispiel



Eingabegraph G mit grau unterlegten SCCs (Knoten enthalten discovery/finishing-Zeiten, Baumkanten rot unterlegt)



transponierter Graph G^T



Komponentengraph G^{SCC}

Starke Zusammenhangskomponenten: Berechnung

■ STRONGLY-CONNECTED-COMPONENTS(G)

```
1 for( each  $u \in G.V$  )  $u.color = WEISS$ 
2  $time = 0$ 
3 for( each  $u \in G.V$  )
4   if(  $u.color == WEISS$  ) DFS-VISIT(  $u, fv, NIL, 0$  )

5 INVERT-EDGES( $G$ ) // berechnet  $E^T$  aus  $E$ 

6 for( each  $u \in V[G]$  )  $u.color = WEISS$ 
7  $scc\_no = 0$ 
8 for(  $i = |G.V|$  down to 1 )
9   if(  $fv[i].color == WEISS$  )
10      $scc\_no = scc\_no + 1$ 
11     DFS-VISIT(  $fv[i], NIL, scc, scc\_no$  )
12 return(  $scc$  )
```

DFS-Lauf 1: Ordne Knoten nach Endzeit $f[u] \rightarrow fv[]$

DFS-Lauf 2 (in *absteigender* Reihenfolge $f[u]$): Bestimmung der SCCs

Starke Zusammenhangskomponenten: Berechnung

■ DFS-VISIT(u , fv , scc , scc_no)

```
1  $u.color = GRAU$ 
2 for( each  $v \in u.Adj$  )
3   if(  $v.color == WEISS$  ) DFS-VISIT( $v$ ,  $fv$ ,  $scc$ ,  $scc\_no$ )
4  $u.color = SCHWARZ$ 
5 if(  $fv \neq NIL$  )  $time = time+1; fv[time]$ 
6 else  $scc[u] = scc\_no$ 
```

DFS-Lauf 1: Ordne Knoten nach Endzeit $f[u]$

DFS-Lauf 2 (in Reihenfolge $f[u]$): Bestimmung der SCCs

■ INVERT-EDGES(G)

```
1 for( each  $v \in G.V$  )
2   for( each  $u \in v.Adj$  ) LIST-INSERT(  $u.AdjT$ ,  $v$  )
3 for( each  $v \in V$  )  $v.Adj = v.AdjT$ 
```

■ STRONGLY-CONNECTED-COMPONENTS(G) benötigt asymptotisch $O(N+M)$ Zeit.

Starke Zusammenhangskomponenten: Korrektheit

■ Im folgenden:

- beziehen sich $d[]$ und $f[]$ auf die Entdeckungs- und Endzeiten des ersten DFS-Laufs.

- sei für $U \subseteq V$ definiert: $d(U) = \min_{u \in U} d[u]$ und $f(U) = \max_{u \in U} f[u]$

■ **Lemma 22.14:** (SCCs und Endzeiten $f[]$)

Seien C, C' zwei SCCs des Graphen $G=(V,E)$ mit $C \neq C'$.
Für alle Kanten $(u,v) \in E$ mit $u \in C$ und $v \in C'$
gilt: $f(C) > f(C')$.

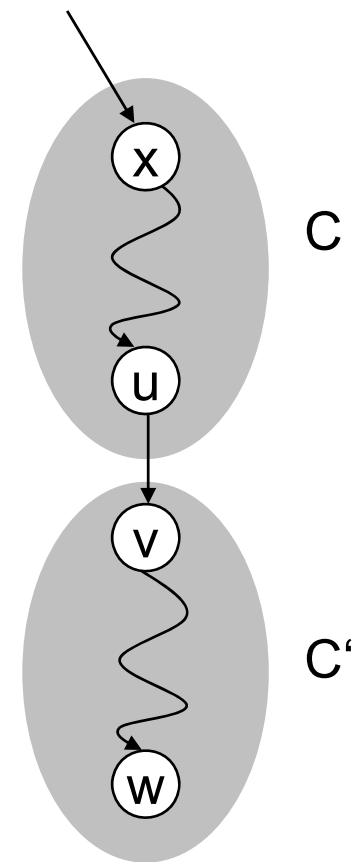
- Beweis: Fallunterscheidung nach Entdeckungsreihenfolge

Fall 1: $d(C) < d(C')$:

Sei $x \in C$ mit $d[x] = d(C)$. Zum Zeitpunkt $d[x]$ gilt: Für jeden Knoten $w \in C'$ existiert ein weißer Pfad $x \rightsquigarrow u \rightarrow v \rightsquigarrow w$.

=> w ist Nachfahre von x (Theorem der weißen Pfade)

=> $f[x] = f(C) > f(C')$ (Klammerungstheorem)



Starke Zusammenhangskomponenten: Korrektheit

■ Beweis von Lemma 22.14 (Fortsetzung)

Fall 2: $d(C) > d(C')$

Sei $y \in C'$ mit $d[y] = d(C')$. Zum Zeitpunkt $d[y]$ gilt: Für jeden Knoten $w' \in C'$ existiert ein weißer Pfad $y \rightsquigarrow w'$.

$\Rightarrow w'$ ist Nachfahre von y (Theorem der weißen Pfade)

und $f(C') = f[y]$ (Klammerungstheorem)

Da $d(C) > d(C')$, gilt zum Zeitpunkt $d[y]$:

$\forall w \in C: w.\text{color} = \text{WEISS}$.

G^{SCC} ist ein DAG und $(C, C') \in E^{\text{SCC}}$

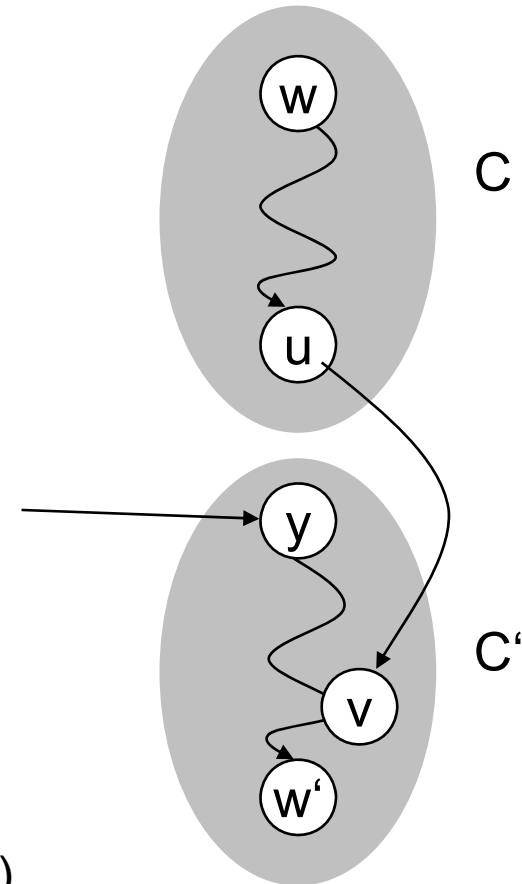
$\Rightarrow (C', C) \notin E^{\text{SCC}}$, d.h. $\forall r \in C', s \in C:$
 $r \rightsquigarrow s$ ist falsch.

\Rightarrow Zum Zeitpunkt $f[y]$ gilt:

$\forall w \in C: w.\text{color} = \text{WEISS}$

$\Rightarrow f(C) > d(C) > f[y] = f(C')$ (Klammerungstheorem)

■ **Korollar 22.15:** Für Kanten $(u, v) \in E^T$ mit $u \in C$,
 $v \in C'$ gilt $f(C) < f(C')$



Starke Zusammenhangskomponenten: Korrektheit

■ Theorem 22.16: Korrektheit des SCC-Algorithmus

Sei $G=(V,E)$ ein gerichteter Graph, dann bestimmt der Algorithmus `STRONGLY-CONNECTED-COMPONENTS()` die starken Zusammenhangskomponenten von G .

- Beweis: (durch vollständige Induktion über Schleife Zeile 8-11)

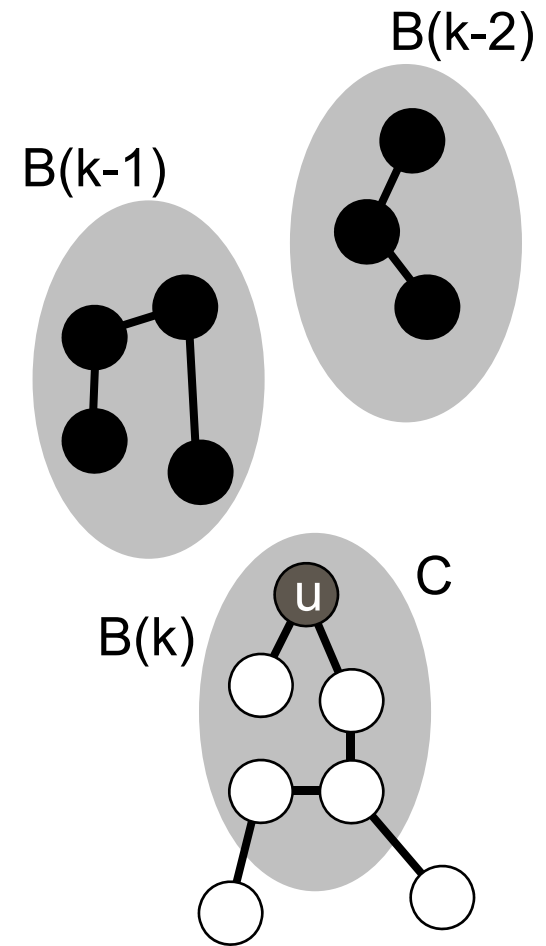
Annahme: der im k -ten Aufruf von `DFS-VISIT()` erzeugte DFS-Baum $B(k)$ ist eine SCC in G .

Induktionsschritt: Sei u die Wurzel des k -ten DFS-Baums $B(k)$, sei C eine SCC in G mit $u \in C$ und sei X der Zeitpunkt, in dem in der Schleife <Zeile 8-11> u gewählt wird.

Teil 1: $C \subseteq B(k)$: Zum Zeitpunkt X existieren keine grauen Knoten. Alle schwarzen Knoten gehören nach Induktionsannahme zu anderen SCCs.

$\Rightarrow \forall w \in C: w.\text{color} = \text{WEISS}$ und es existiert $u \rightsquigarrow w$ in C , da C eine SCC ist.

$\Rightarrow w \in B(k)$ (Theorem der weißen Pfade)



Starke Zusammenhangskomponenten: Korrektheit

■ Beweis Theorem 22.16: (Fortsetzung)

Teil 2: $C \supseteq B(k)$

Sei $w \in B(k)$. Angenommen, es sei $w \in C'$, $C' \neq C$.

Sei w so gewählt, dass $\delta(u, w)$ minimal ist.

◆ $\forall i \in \{1, \dots, k-1\}$: Zum Zeitpunkt X sind alle Knoten aus $B(i)$ bereits schwarz, w jedoch weiß.

$\Rightarrow C' \neq B(i) \quad \forall i \in \{1, \dots, k-1\}$

◆ w ist Nachfahre von u , da $w \in B(k)$ ist.

\Rightarrow Zum Zeitpunkt X existiert ein weißer Pfad von u nach w . Sei $e=(v, w)$ die letzte Kante des Pfades. Aufgrund der Wahl von w ist $v \in C$.

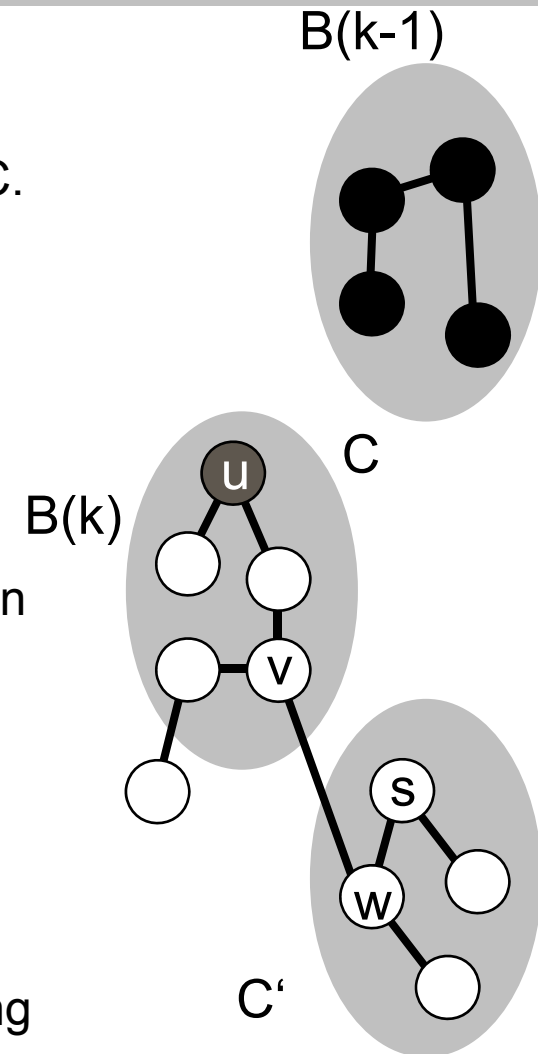
$(v, w) \in E^T \Rightarrow f(C) < f(C')$ (Korollar 22.15)

\Rightarrow Sei $s \in C'$ mit $f[s] = f(C')$.

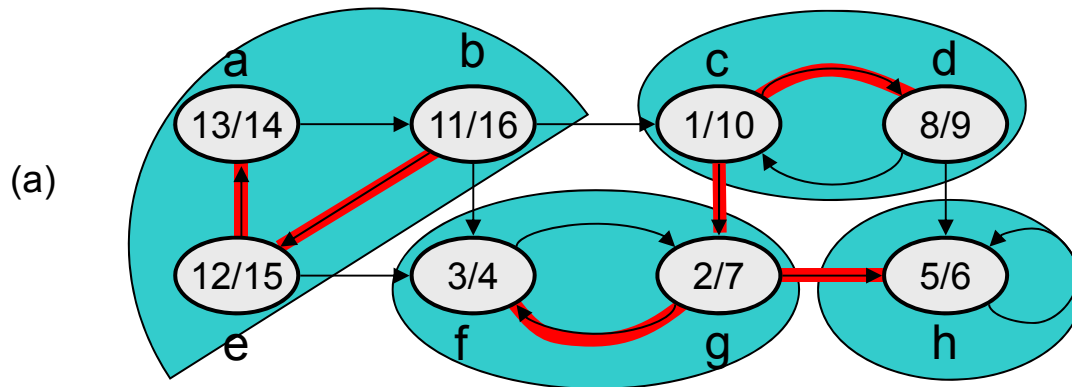
Dann folgt $f[u] \leq f(C) < f(C') = f[s]$. Zudem gilt $s.\text{color} = \text{WEISS}$, da $C' \neq B(i) \quad \forall i \in \{1, \dots, k-1\}$

\Rightarrow Zum Zeitpunkt X wird aufgrund der Sortierung s statt u gewählt.

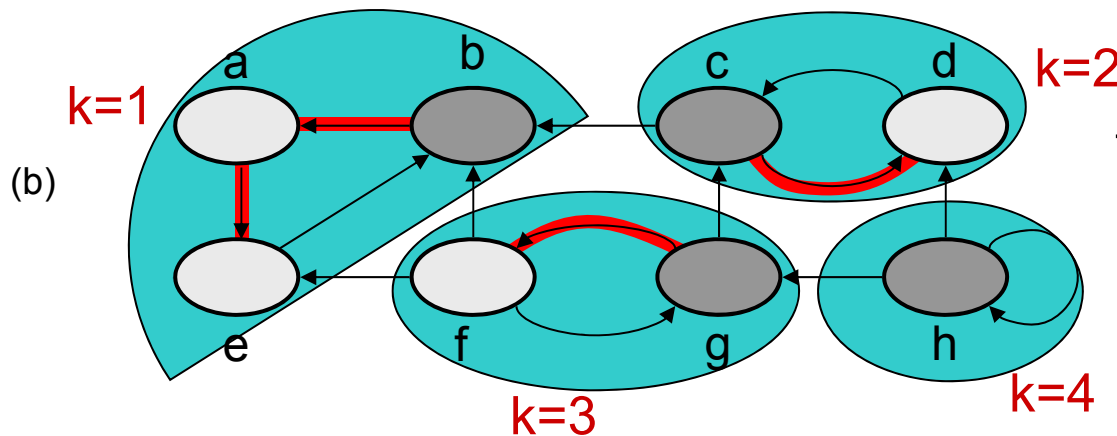
$\Rightarrow w$ existiert nicht, d.h. $C \supseteq B(k)$



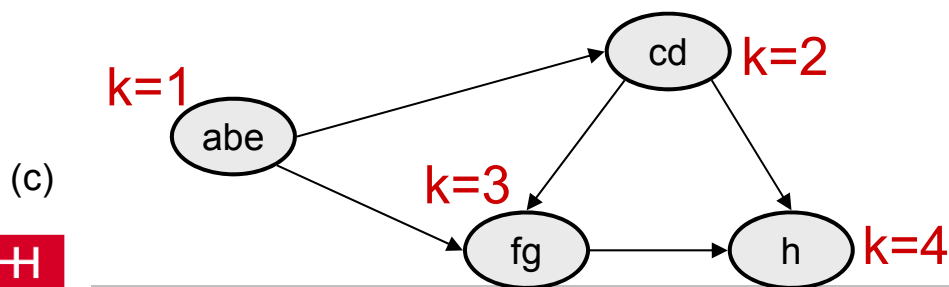
Starke Zusammenhangskomponenten: Ein Beispiel



Eingabegraph G mit grau unterlegten SCCs



transponierter Graph G^T

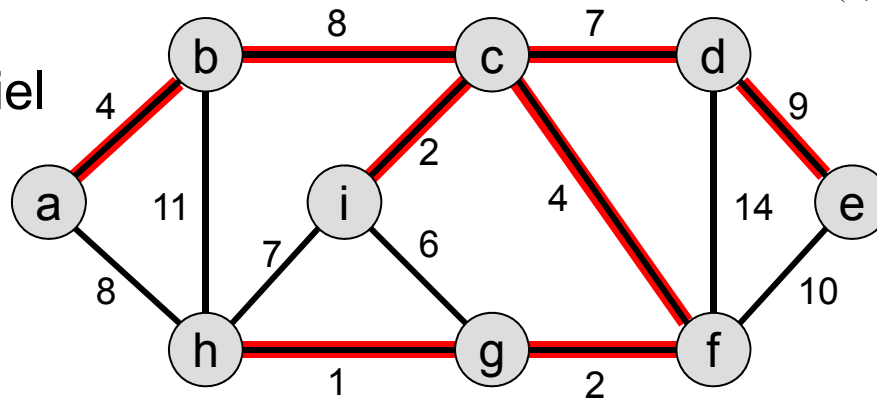


Komponentengraph G^{SCC}

5.5 Minimale Spannbäume

- MST-Problem (minimal spanning tree):
 - geg: (ungerichteter) Graph $G=(V,E)$ mit Kantengewichten $w:E \rightarrow \mathbb{R}$
 - ges: Baum (V,T) mit $T \subseteq E$ und $w(T) = \sum_{(u,v) \in T} w((u,v))$ minimal

- Beispiel



- (V,T) wird als **(minimaler) Spannbaum** bezeichnet.
- Das MST-Problem ist ein **Optimierungsproblem**: Aus einer Menge gültiger Lösungen wird die mit optimalen Kosten gesucht.
- **Greedy-Algorithmen**:
 - Lösung wird durch Einzelentscheidungen aufgebaut, die – einmal getroffen – nie wieder zurückgenommen werden.
 - häufig keine Optimalitätsgarantie

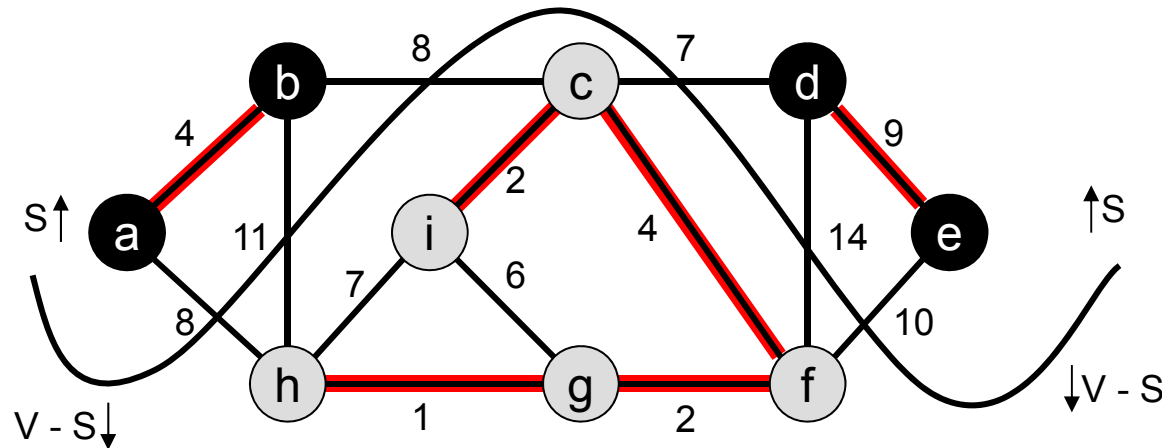
Greedy-Algorithmen für das MST-Problem

- (Greedy-)Strategie:
 - Aufbau des MST durch sukzessives Hinzunehmen von Kanten
- Definition (sichere Kante):
 - Sei A eine Teilmenge eines MST. Die Kante (u,v) ist **für A sicher**, g.d.w. $A \cup \{(u,v)\}$ ist eine Teilmenge eines MST
- Anzahl Kanten:
 - Ein Spannbaum über $G=(V,E)$ hat $|V|-1$ Kanten.

```
■ GENERIC-MST( $G, w$ )  
   $A = \emptyset$   
  while(  $|A| < |V|-1$  )  
     $e = \text{FIND-SAVE-EDGE}(G, A)$   
     $A = A \cup \{e\}$   
  return  $A$ 
```

- Korrektheit von GENERIC-MST:
 - folgt direkt aus der Definition der sicheren Kanten.

Theorem der sicheren Kanten



■ Definition (Schnitt, etc.):

- Ein **Schnitt** $(S, V - S)$ ist eine Partition der Knotenmenge, $S \subseteq V$.
- Eine Kante $e=(u,v)$ **kreuzt** den Schnitt, falls $u \in S, v \in V - S$ gilt,
- sonst **respektiert** die Kante $e=(u,v)$ den Schnitt.
- Eine S kreuzende Kante e heißt **leichte Kante**, g.d.w. $w(e)$ unter allen S kreuzenden Kanten minimal ist.

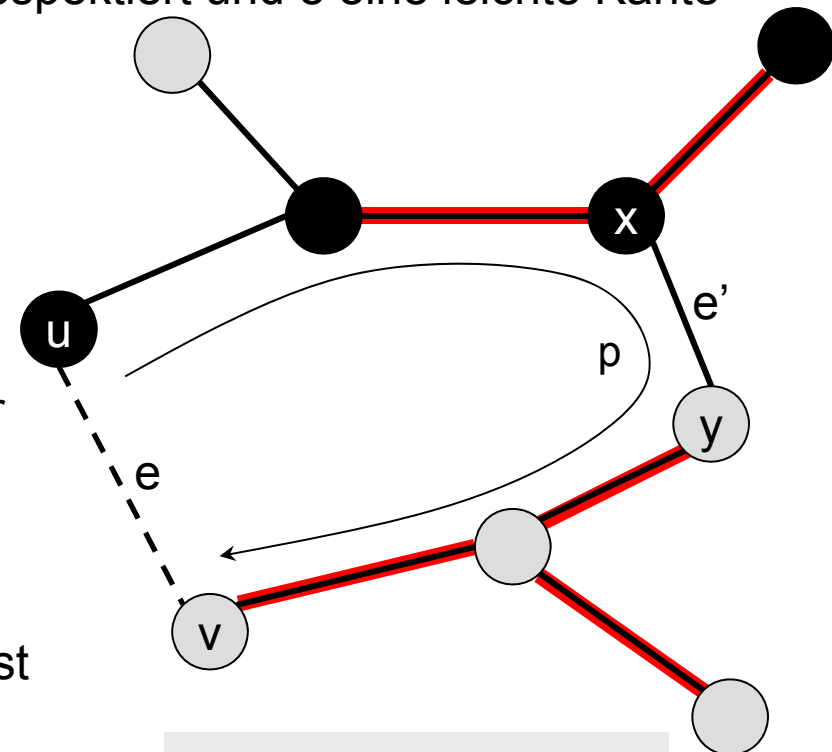
Theorem der sicheren Kanten

■ Theorem 23.1 (sichere Kanten):

Sei $G=(V,E)$ ein zusammenhängender, ungerichteter Graph, $w: E \rightarrow \mathbb{R}$ eine Gewichtsfunktion, sei A Teilmenge eines MST. Sei $(S,V-S)$ ein Schnitt, der von (allen Kanten in) A respektiert und e eine leichte Kante die $(S,V-S)$ kreuzt. Dann ist e eine sichere Kante.

■ Beweis:

- Sei T ein MST, mit $A \subseteq T$ und $(S,V-S)$ und $e=(u,v)$ wie im Theorem gefordert. Angenommen, $e \notin T$:
- Es gibt einen eindeutigen Zyklus, der außer e nur Kanten aus T enthält.
- Es gibt eine Kante $e'=(x,y) \in T$, die $(S,V-S)$ kreuzt; $e' \notin A$
- Sei $T' = T \setminus \{e'\} \cup \{e\}$. Offensichtlich ist T' ein Spannbaum
- $w(T') = w(T) - w(e') + w(e) \leq w(T)$ da $w(e) \leq w(e')$ (e ist leichte Kante)
- $A \subseteq T'$ und $e \in T' \Rightarrow e$ ist für A sicher.



rot: Kanten der Menge A
schwarz: Knoten von S
grau: Knoten von V-S

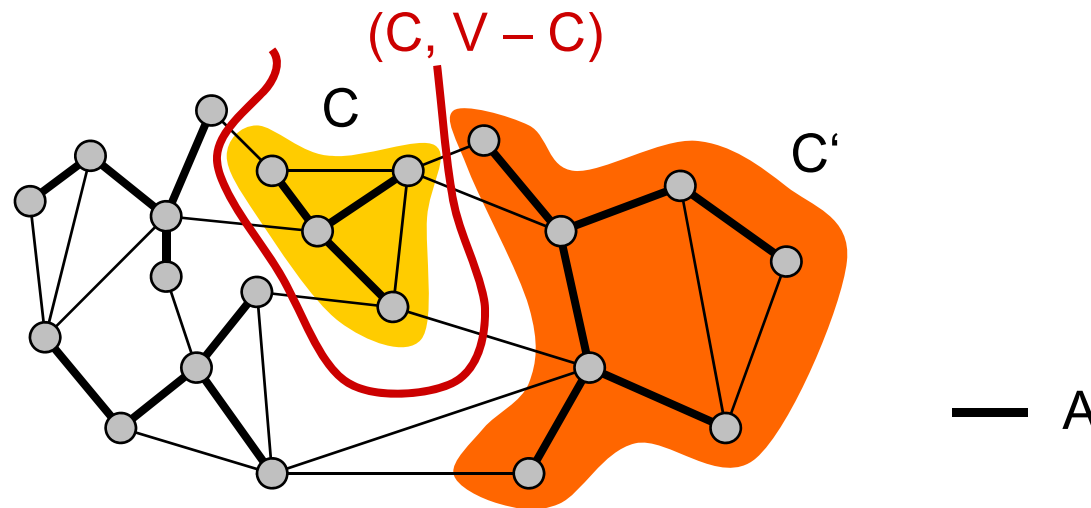
Theorem der sicheren Kanten

- Aus dem Theorem der sicheren Kanten folgt die Korrektheit der Greedy-Strategie:

- **Korollar 23.2:**

Sei $G=(V,E)$ ein ungerichteter, zusammenhängender Graph, $w: E \rightarrow \mathbb{R}$ eine Gewichtsfunktion, sei A Teilmenge eines MST. A definiert den Wald $G_A = (V,A)$. Sei C eine Zusammenhangskomponente in G_A und e eine leichte Kante, die C mit einer anderen Komponente C' verbindet. Dann ist e für A sicher.

- Beweis: Der Schnitt $(C, V - C)$ respektiert A und e ist eine leichte Kante für $(C, V - C)$. Somit ist e für A sicher.



Kruskals Algorithmus

- Idee:

- A beschreibt einen Wald, mit jeder Kante werden zwei Bäume zu einem verschmolzen.
- Zu Beginn haben wir $|V|$ Bäume mit je einem Knoten.
- Kanten werden mit aufsteigendem Gewicht eingefügt.
- Schnitt trennt jeweils einen Baum vom Rest des Graphen.

- Kruskals Algorithmus benötigt eine Datenstruktur, die für je zwei Knoten effizient entscheidet, ob diese zum gleichen Baum gehören oder nicht:

- **Disjoint-Set** Datenstruktur:

- MAKE-SET: erstellt eine Menge
- UNION: vereinigt zwei disjunkte Mengen
- FIND-SET: liefert ein repräsentatives Element einer Menge

Kruskals Algorithmus

MST-KRUSKAL(G, w)

$A = \emptyset$

for(each $v \in G.V$)

 MAKE-SET(v)

SORT-EDGES-BY-INCREASING-WEIGHT($G.E$)

for(each $(u,v) \in E[G]$ (ordered by increasing weight))

if(FIND-SET(u) \neq FIND-SET(v))

$A = A \cup \{(u,v)\}$

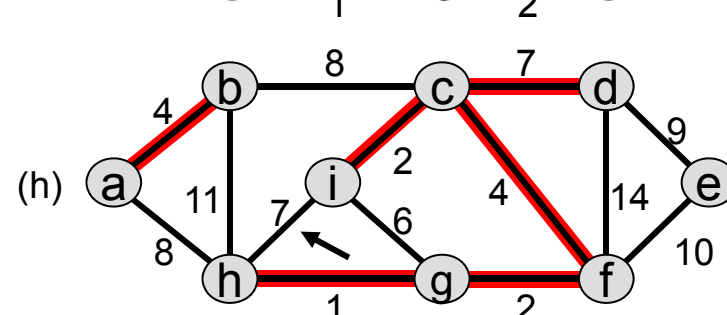
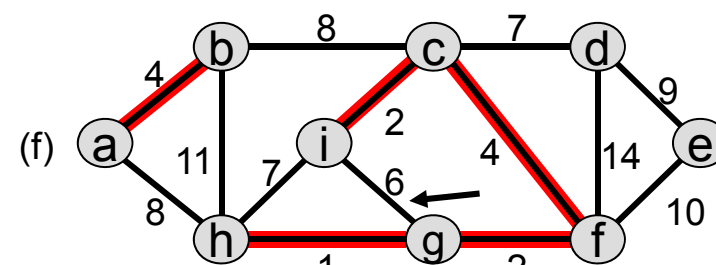
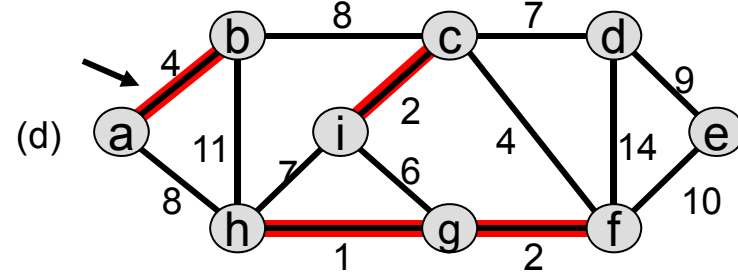
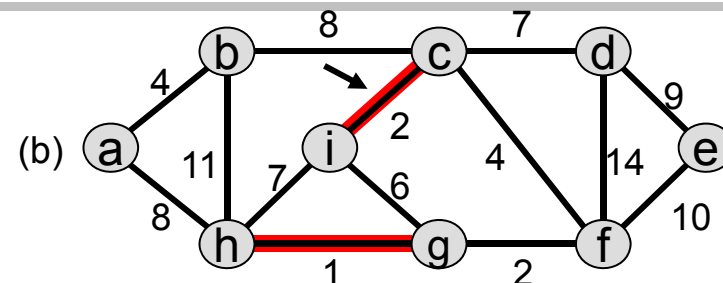
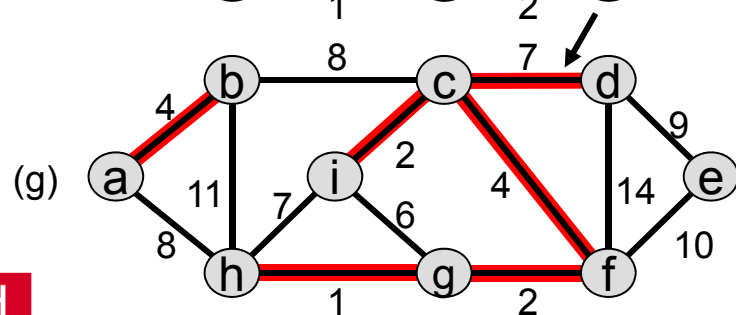
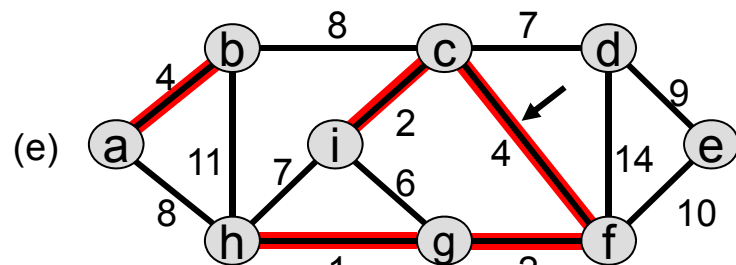
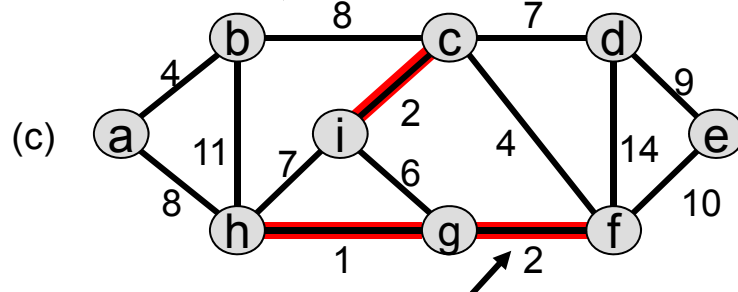
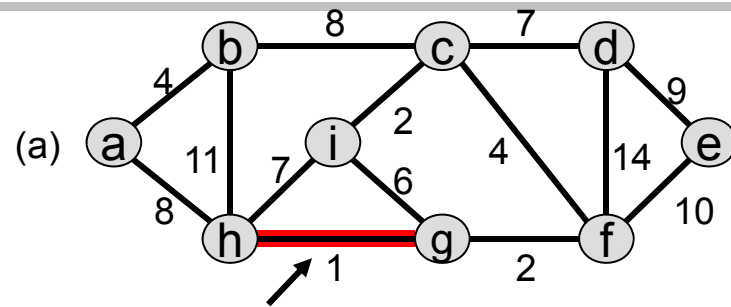
 UNION(u,v)

return A

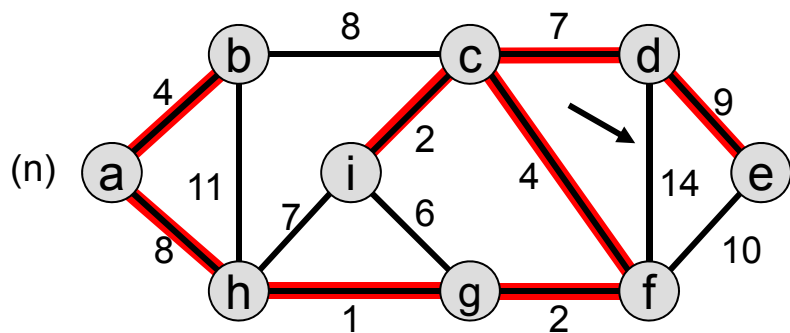
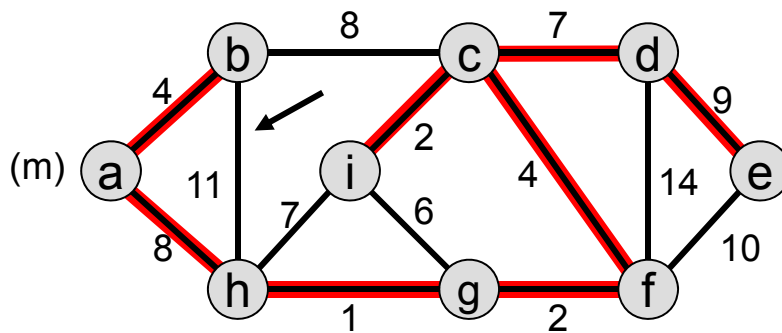
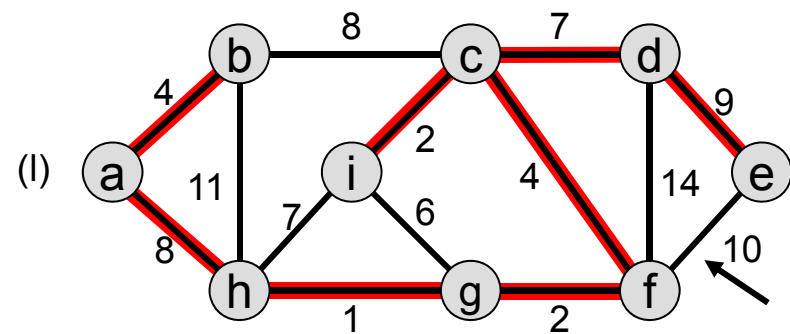
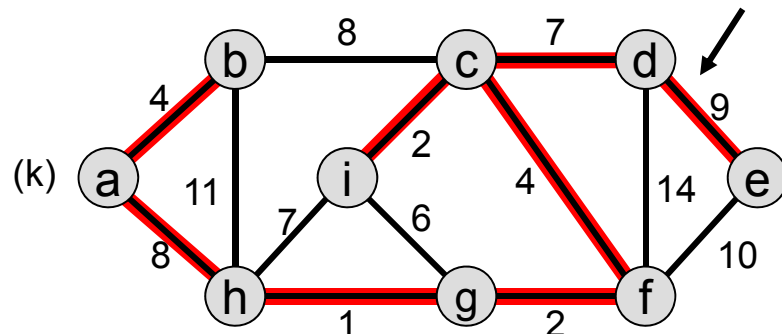
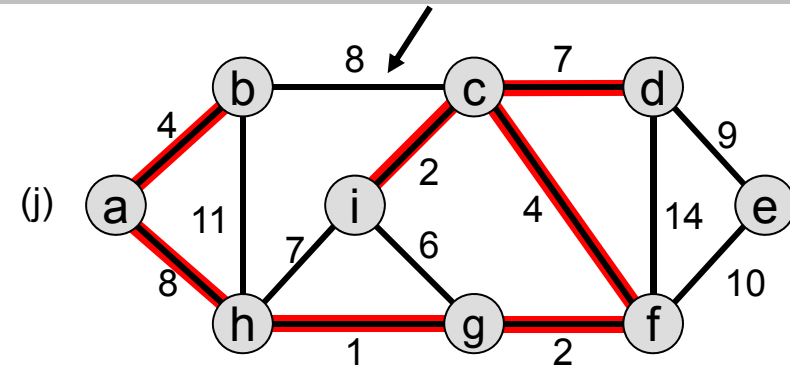
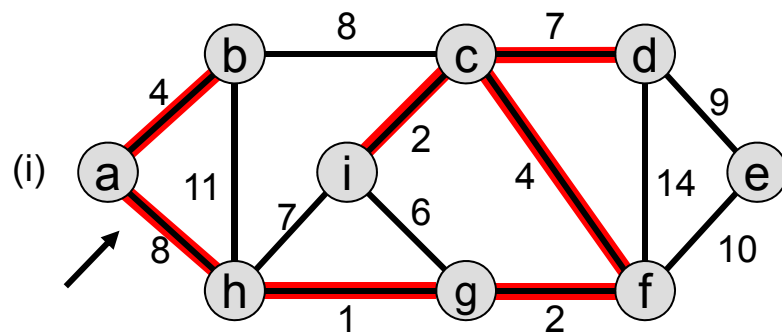
- Korrektheit: Aufgrund der aufsteigenden Sortierung ist (u,v) eine leichte Kante. Die Korrektheit folgt somit direkt aus Korollar 23.2.
- Laufzeit: Seien $T_{\text{make-set}}, T_{\text{find-set}}, T_{\text{union}}$ die Laufzeiten der Operationen der Disjoint-Set-Datenstruktur, dann gilt:

$$T_{\text{Kruskal}}(N,M) = N * T_{\text{make-set}} + O(M \log N) + 2M * T_{\text{find-set}} + (N-1) * T_{\text{union}}$$

Beispiel: Kruskals Algorithmus



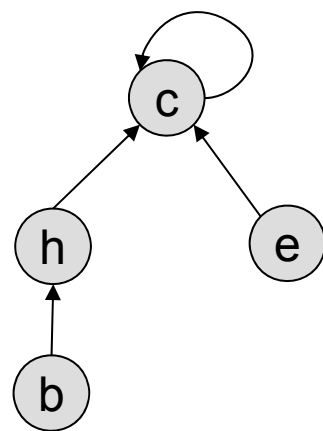
Beispiel: Kruskals Algorithmus



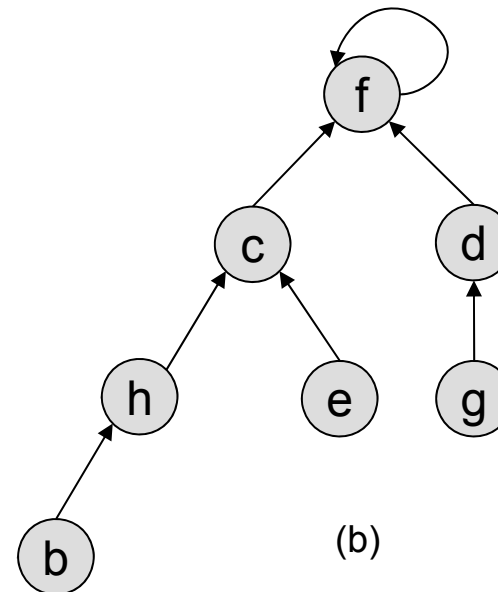
Eine Datenstruktur für disjunkte Mengen (Union-Find Datenstruktur)

■ Idee:

- Menge wird durch einen gewurzelten Baum repräsentiert
- Repräsentant ist die Wurzel des Baums
- FIND-SET: gibt die Wurzel des Baums aus
- UNION(c,f): hängt einen Baum c unter einen anderen Baum f



(a)



(b)

Die Union-Find Datenstruktur

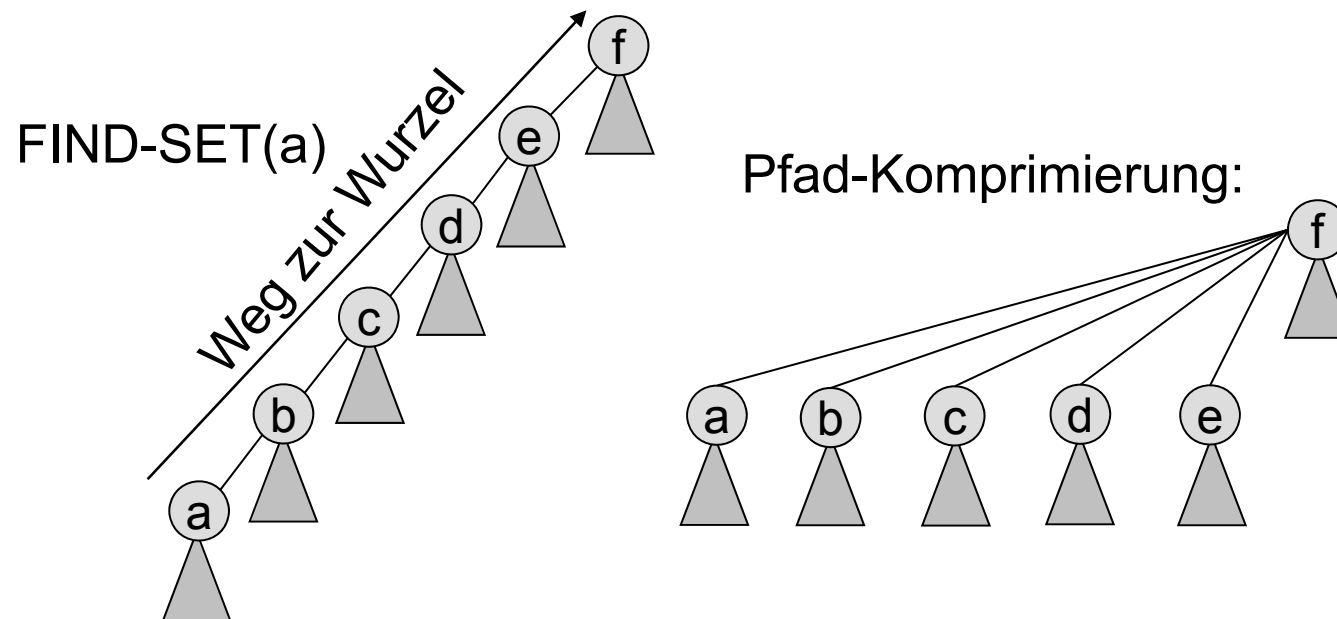
■ Heuristiken

■ **union-by-rank:** (UNION-Operation)

- ◆ hänge den flacheren Baum unter den tieferen

■ **path-compression:** (FIND-SET-Operation)

- ◆ hänge alle Teilbäume auf dem Weg zur Wurzel unter die Wurzel



Die Union-Find Datenstruktur

■ Datenstruktur:

- `x.p`: Vorgänger / Elter-Knoten
- `x.rank`: Obere Schranke für die Höhe des Teilbaums unter `x`

■ `MAKE-SET(x)` // Initialisierung

```
x.p = x  
x.rank = 0
```

■ `LINK(x, y)` // hängt flacheren Teilbaum unter den anderen

```
if( x.rank > y.rank ) y.p = x else x.p = y  
if( x.rank == y.rank ) y.rank = y.rank +1
```

■ `UNION(x, y)` // Vereinigung zweier Mengen, die durch je // einen Repräsentanten gegeben sind

```
LINK( FIND-SET(x), FIND-SET(y) )
```

Die Union-Find Datenstruktur

- Rekursives FIND-SET:

- `FIND-SET(x)`

```
if( x ≠ x.p )  
    x.p = FIND-SET(x.p)  
return x.p
```

- Iteratives FIND-SET:

- `FIND-SET(x)`

```
r = x           // 1. Suche die Wurzel  
while( r ≠ r.p ) r = r.p  
while( x ≠ r ) // 2. Pfad-Kompression  
    px = x.p; x.p = r; x = px  
return r
```

- Achtung: FIND-SET mit Pfad-Kompression korrigiert $\text{rank}[x]$ nicht!

- Amortisierte Laufzeit für

- N MAKE-SET-Operationen

- N-1 UNION-Operationen

- M FIND-SET-Operationen

} $T_{\text{UNION-FIND}}(N,M) = O(M \alpha(N))$
 $\alpha()$ ist die Inverse der Ackermann-Funktion, $\alpha(x) \leq 4 \quad \forall x \leq 10^{80}$

- Laufzeit Kruskals MST-Algorithmus:

$$\begin{aligned} T_{\text{Kruskal}}(N,M) &= N * T_{\text{make-set}} + O(M \log N) + 2M * T_{\text{find-set}} + (N-1) * T_{\text{union}} \\ &= O(M \log N) + O(M \alpha(N)) = O(M \log N) \end{aligned}$$

Prims Algorithmus

■ Idee:

- A ist zunächst leer
- Der Algorithmus beginnt an einem beliebig wählbaren Startknoten r
- In jeder Iteration wird ein zusammenhängender minimaler (Teil-) Spannbaum um eine Kante erweitert.
- Eine Prioritätswarteschlange Q speichert alle Knoten, die noch nicht durch A verbunden sind mit Priorität nach min. Kantengewicht, um sie zu verbinden
- π speichert eine Vorgänger-Relation (wie bei der Breitensuche)
- Die Menge A ist implizit definiert: $A = \{ (v, \pi(v)) \mid v \in V - \{r\} - Q \}$

Prims Algorithmus

■ **MST-PRIM**(G, w, r)

for(each $u \in G.V$)

$key[u] = \infty$

$\pi[u] = \text{NIL}$

$key[r] = 0$

$Q = G.V$

while($Q \neq \emptyset$)

$u = \text{EXTRACT-MIN}(Q)$

for(each $v \in u.\text{Adj}$)

if($v \in Q$ **and** $w(u,v) < key[v]$)

$\pi[v] = u$

$key[v] = w(u,v)$

DECREASE-KEY($Q, v, w(u,v)$)

■ **key**[u]:

■ Minimales Gewicht einer Kante von u zum Baum A , falls u eine Kante von A entfernt ist, ∞ sonst

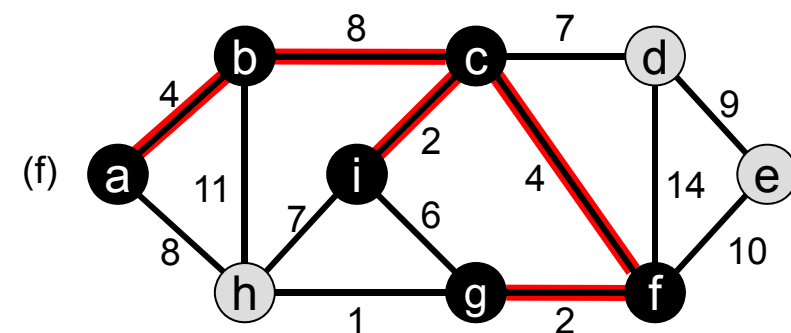
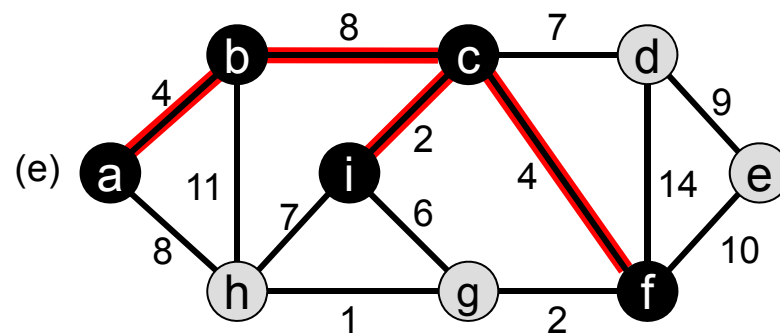
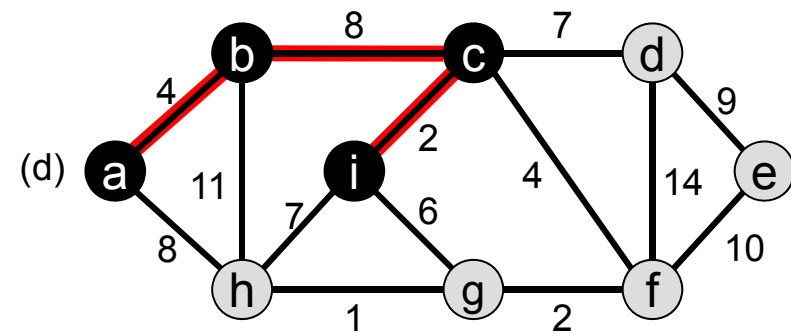
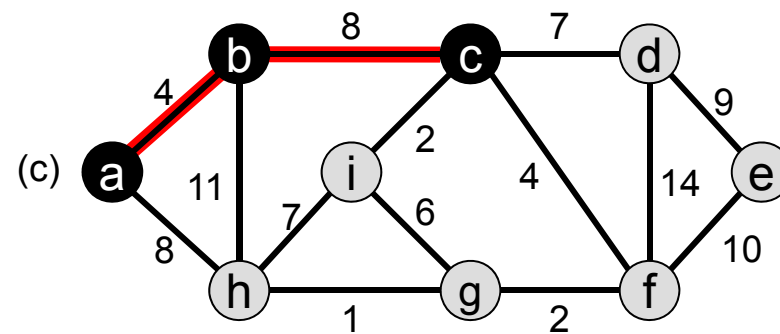
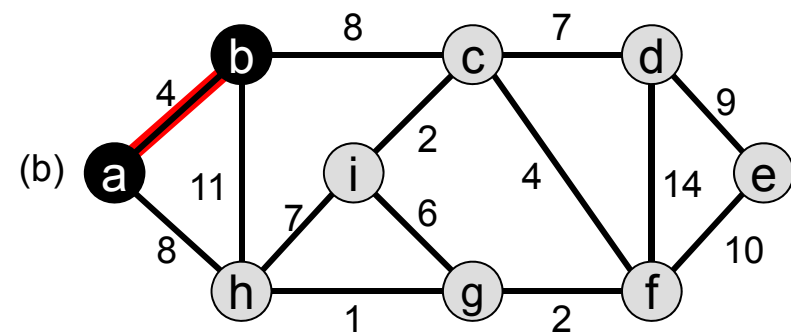
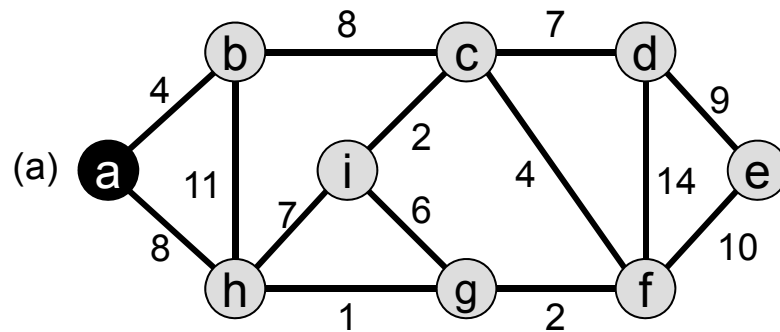
■ **π** [u]:

■ Vater von u im MST

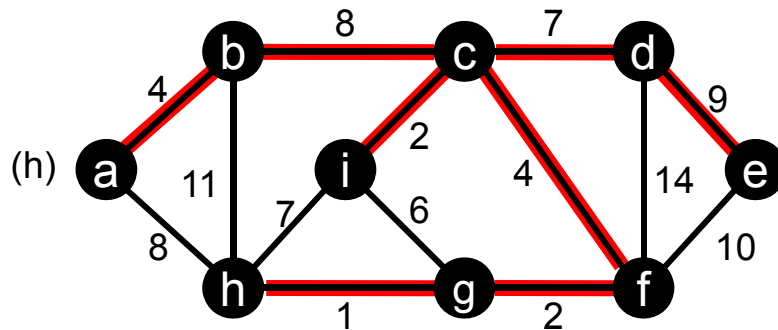
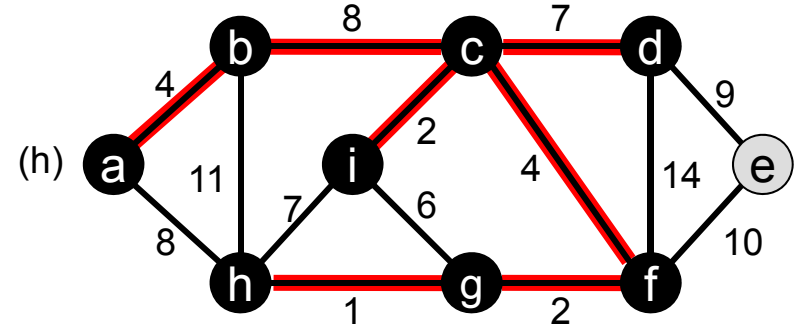
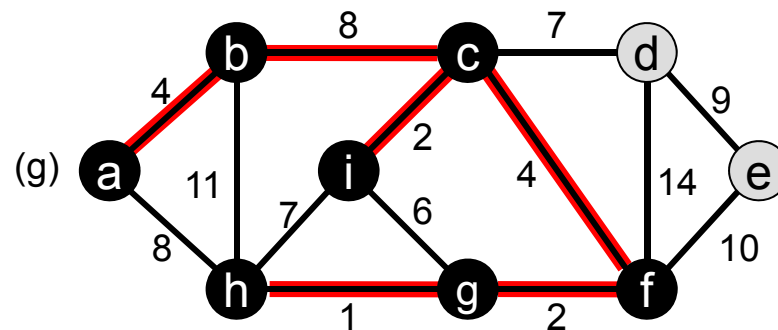
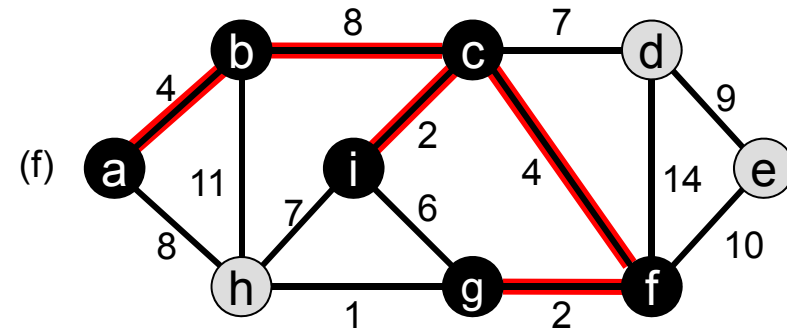
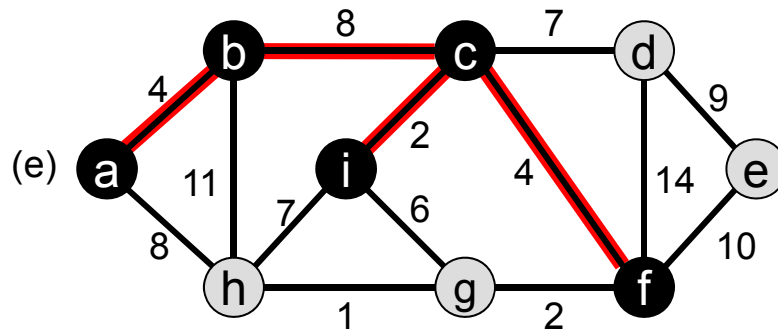
■ $A = \{ (u, \pi[u]) \mid u \in V - \{r\} \}$

ACHTUNG: In Cormen et al wird ein implizites **DECREASE** angenommen.

Beispiel: Prim's Algorithmus



Beispiel: Prim's Algorithmus



Prims Algorithmus: Performanz

■ MST-PRIM(G, w, r)

for(each $u \in G.V$)

$key[u] = \infty$

$\pi[u] = NIL$

$key[r] = 0$

$Q = G.V$

while($Q \neq \emptyset$)

$u = \text{EXTRACT-MIN}(Q)$

for(each $v \in u.Adj$)

if($v \in Q$ **and** $w(u,v) < key[v]$)

$\pi[v] = u$

$key[v] = w(u,v)$

$\text{DECREASE-KEY}(Q, v, w(u,v))$

■ Laufzeitanalyse ($N = |V|, M = |E|$)

■ Initialisierung (1-5): $O(N)$

■ Schleife 6-12: N Durchläufe

■ Schleife 8-12: insgesamt $2M$ Durchläufe

■ Priority-Queue: binärer MIN-Heap mit zusätzlichem Zeigerfeld zum Auffinden von v in Q :

EXTRACT-MIN: $O(\log N)$

DECREASE-KEY: $O(\log N)$

$$T_{\text{PRIM}}(N,M) = O(M \log N)$$

Korrektheit:

Wir betrachten den Schnitt $(Q, V - Q)$. Wenn u aus Q entfernt wird, ist $(u, \pi[u])$ eine leichte Kante. Die Korrektheit folgt dann aus Korollar 23.2.

5.6 Kürzeste Pfade

■ Problem:

■ geg: Graph $G=(V,E)$, Kantengewichte $w:E \rightarrow \mathbb{R}$

■ ges: kürzeste Pfade (bzw. Wege) zwischen Knoten aus V ,

◆ Länge oder Gewicht eines Pfades $(v_0, v_1, \dots, v_k): w(p) = \sum_{i=1}^k w(v_{i-1}, v_i)$

◆ Gewicht des kürzesten Pfades:

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \xrightarrow{p} v\} & \text{falls ein Pfad von } u \text{ nach } v \text{ existiert} \\ \infty & \text{sonst} \end{cases}$$

■ Kantengewichte:

■ Distanzen, Zeit, Kosten, Erfolgswahrscheinlichkeiten, Verlust

■ Problemvarianten:

■ single-source: kürzeste Pfade von einem Knoten zu allen anderen

■ single-pair: kürzester Pfad zw. zwei ausgewählten Knoten

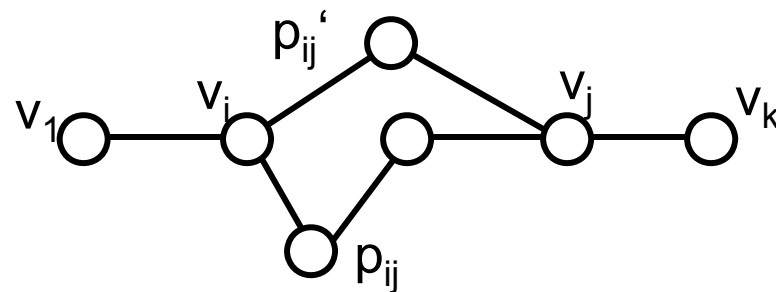
■ all-pairs: kürzeste Pfade zwischen allen Knotenpaaren

Eigenschaften kürzester Pfade

■ Lemma 24.1 (Optimale Teilstruktur):

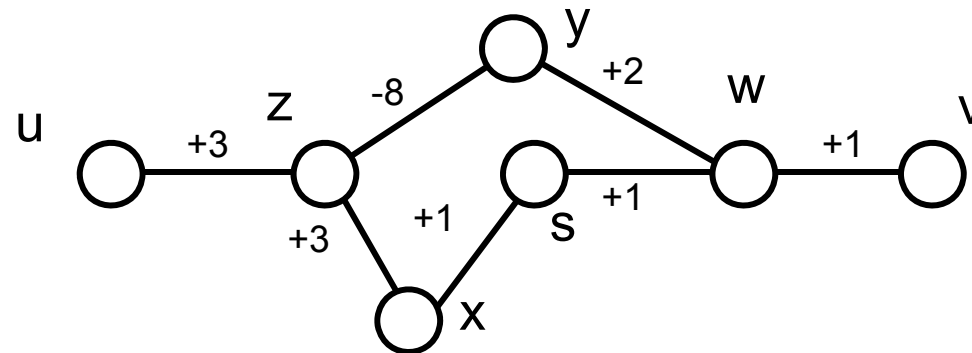
Sei $G = (V, E)$ ein gewichteter, gerichteter Graph, $w: E \rightarrow \mathbb{R}$ eine Gewichtsfunktion. Sei $p = (v_1, v_2, \dots, v_k)$ ein kürzester Pfad von v_1 zu v_k . Für alle $1 \leq i \leq j \leq k$ gilt: $p_{ij} = (v_i, v_{i+1}, \dots, v_j)$ ist ein kürzester Pfad von v_i nach v_j .

■ Beweis: Angenommen es existiert ein Pfad p_{ij}' von p_i nach p_j mit $p_{ij}' \neq p_{ij}$ und $w(p_{ij}') < w(p_{ij})$. Dann gilt für $p' = (v_1, \dots, v_{i-1}, p_{ij}', v_{j+1}, \dots, v_k)$: $w(p') < w(p)$. Somit ist p kein kürzester Pfad. ⚡



Eigenschaften kürzester Pfade

- Was geschieht bei Kanten mit negativen Kantengewichten?
 - $\delta(u,v)$ ist im Prinzip auch für negative Kantengewichte definiert
 - Angenommen, es gibt auf dem Pfad von u nach v einen Zyklus mit negativer Gesamtlänge:



$$w((u,z,x,s,w,v)) = 9 \quad w((u,z,y,w,v)) = -2$$

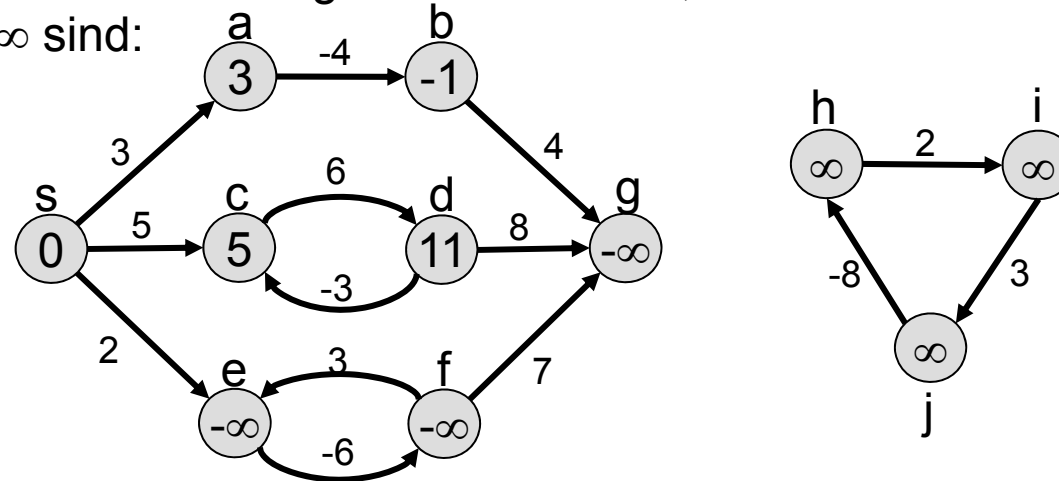
$$w((u,z,y,w,s,x,z,y,w,v)) = -3$$

Mit jedem Durchlauf des negativen Zyklus verkürzt sich der Pfad!

- per Definition: $\delta(u,v) = -\infty$, falls es auf einen Pfad von u nach v einen Zyklus negativer Gesamtlänge gibt.

Eigenschaften kürzester Pfade

- Ein Zyklus negativer Gesamtlänge bedeutet nicht, dass alle Distanzen in einem Graphen $-\infty$ sind:



- Falls $|\delta(u,v)| < \infty$, existiert ein kürzester Pfad mit höchstens $|V|-1$ Kanten.
oder: Falls $|\delta(u,v)| < \infty$, existiert ein kürzester Pfad ohne Zyklen.
 - Annahme: ein kürzester Pfad p von u nach v enthält $|V|$ Kanten
 - => p enthält $|V|+1$ Knoten
 - => in p kommt mindestens ein Knoten doppelt vor
 - => p enthält einen Zyklus c:
 - $w(c) < 0$: $\delta(u,v) = -\infty$
 - $w(c) = 0$: c kann aus p gelöscht werden, ohne dass sich $w(p)$ verändert
 - $w(c) > 0$: durch Löschen von c entsteht ein Pfad p' mit $w(p') < w(p)$

Repräsentation kürzester Pfade

Ziel:

- Speicherung eines kürzesten Pfades von einem Startknoten s zu allen anderen Knoten
- Speicherung aller kürzesten Pfadlängen

Methode:

- bzgl. eines Zielknotens s , haben alle Knoten v einen Vorgänger $\pi[v]$ auf dem kürzesten Pfad zu s (Lemma 24.1):

Vorgängerteilgraph $G_\pi = (V_\pi, E_\pi)$ mit:

$$V_\pi = \{v \in V \mid \pi[v] \neq NIL\} \cup \{s\}$$
$$E_\pi = \{(\pi[v], v) \in E \mid v \in V_\pi - \{s\}\}$$

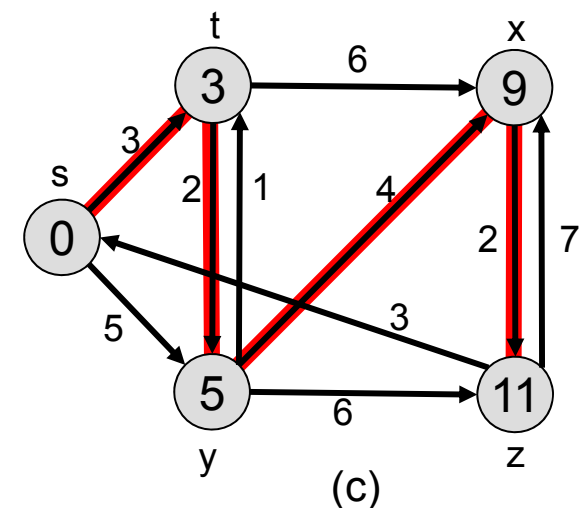
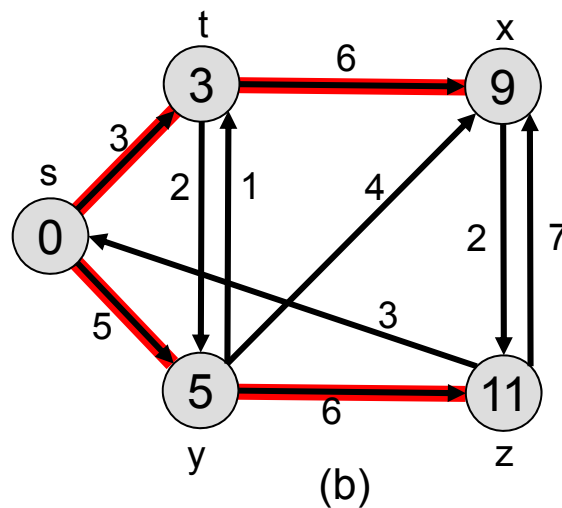
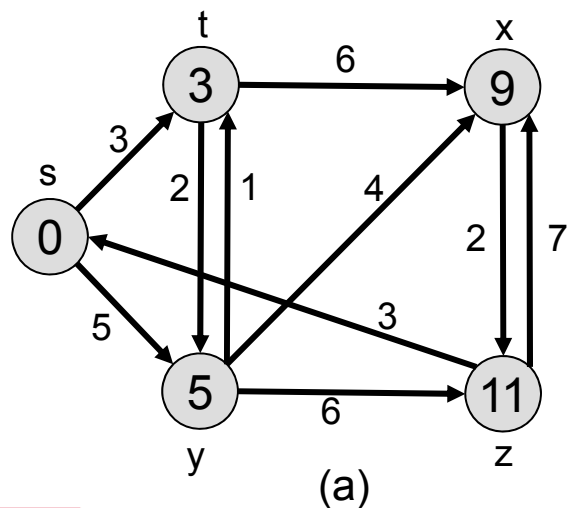
=> Jeder Knoten hat bzgl. eines kürzesten Pfades maximal einen Vorgänger, d.h. kürzesten Pfade können als Baumstruktur gespeichert werden (Shortest-Paths Tree)

- ◆ $d[v]$: Obere Schranke für $\delta(s, v)$
- ◆ $\delta(s, v)$: Länge des kürzesten Pfades von v zu s
- ◆ $\pi[v]$: Vorgänger von v auf dem kürzesten Pfad zu s

Repräsentation kürzester Pfade

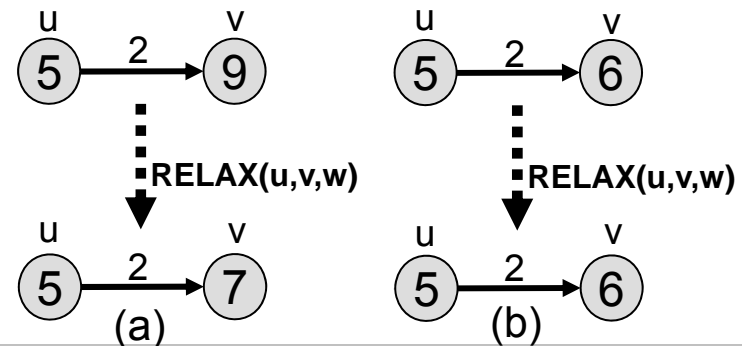
Der Baum kürzester Pfade $G' = (V', E')$ ist ein gerichteter Teilgraph des Eingabegraphen $G=(V,E)$ mit

- V' ist die Menge der in G von s aus erreichbaren Knoten
- G' bildet einen gerichteten Baum mit der Wurzel s
- Für alle $v \in V'$ ist der eindeutige, einfache Pfad von s nach v in G' ein kürzester Pfad von s nach v in G
- Achtung: Weder der Baum kürzester Pfade noch kürzeste Pfade selbst sind eindeutig.



Relaxation

- Berechnung kürzester Pfade mittels Relaxation:
 - Attribut $d[v]$ ist eine obere Schranke für die Länge eines kürzesten Pfades von s nach v : $\delta(s,v)$
 - $d[v]$ wird zunächst mit ∞ initialisiert und anschließend von oben an $\delta(s,v)$ angenähert
 - Annäherung erfolgt durch sukzessives **Relaxieren** von Kanten
- INITIALIZE-SINGLE-SOURCE(G, s)
 - for**(each $v \in G.V$)
 - $d[v] = \infty$
 - $\pi[v] = \text{NIL}$
 - $d[s] = 0$
- Relaxation einer Kante $e = (u,v)$:
 - Test, ob der bisher gefundene kürzeste Pfad zu v verbessern können, in dem wir u als Vorgänger von v (auf dem kürzesten Pfad) betrachten
 - Falls ja, werden $d[v]$ und $\pi[v]$ angepasst:
- RELAX(u, v, w)
 - if**($d[v] > d[u] + w(u,v)$)
 - $d[v] = d[u] + w(u,v)$
 - $\pi[v] = u$



Eigenschaften kürzester Pfade und der Relaxation

■ Dreiecksungleichung:

$$\forall e=(u,v) \in E: \delta(s,v) \leq \delta(s,u) + w(u,v)$$

■ Pfad-Relaxations-Eigenschaft:

Sei $p=(s=v_0, v_1, v_2, \dots, v_k=v)$ ein kürzester Pfad von s zu v . Werden alle Kanten (v_i, v_{i+1}) in der Reihenfolge von s zu v relaxiert, so gilt $d[v] = \delta(s,v)$.

■ Allgemeine Eigenschaften der Kürzeste-Pfade Algorithmen:

■ Eigenschaft der oberen Schranke (Upper-bound property):

$d[v] \geq \delta(s,v)$ und fällt monoton

■ Kein-Pfad-Eigenschaft (No-path property):

Gibt es keinen Pfad von s zu v , so gilt $d[v] = \infty$

■ Konvergenzeigenschaft (Convergence property):

Sei u Vorgänger von v auf einem kürzesten Pfad von s zu v . Gilt $d[u] = \delta(s,u)$ zu irgend einem Zeitpunkt vor der Relaxation der Kante (u,v) , so gilt zu jedem Zeitpunkt nach der Relaxation von (u,v) $d[v] = \delta(s,v)$

■ Vorgängerteilgraph-Eigenschaft (Shortest-path-tree property):

Wenn $d[v] = \delta(s,v)$ gilt, beschreibt der Vorgängerteilgraph einen Baum kürzester Pfade mit Wurzel s

Achtung: Fehler
im Cormen S. 590



■ Beweis der **Dreiecksungleichung**:

- Fall 1: Es gibt einen kürzesten Pfad von s nach v

Der Pfad $s \rightsquigarrow u \rightarrow v$ existiert und ist mindestens so lang wie der kürzeste Pfad von s nach v .

- Fall 2: Es gibt keinen kürzesten Pfad von s nach v und $\delta(s,v) = \infty$
Angenommen, $\delta(s,u) < \infty$, dann existiert ein Pfad von s über u nach v .

- Fall 3: Es gibt keinen kürzesten Pfad von s nach v und $\delta(s,v) = -\infty$
Trivial.

■ Beweis der **Pfad-Relaxations-Eigenschaft**:

Induktionsannahme: Nach Relaxation der i -ten Kante des Pfades p gilt $d[v_i] = \delta(s,v_i)$.

Induktionsanfang ($i=0$): Nach Initialisierung gilt $d[s] = 0 = \delta(s,s)$

Induktionsschritt: Angenommen, es gilt $d[v_{i-1}] = \delta(s,v_{i-1})$. Nach Relaxation der i -ten Kante (v_{i-1}, v_i) gilt: $d[v_i] \leq \delta(s,v_{i-1}) + w(v_{i-1}, v_i) = \delta(s,v_i)$, da ein kürzester Pfad von s nach v_i über v_{i-1} führt. Da $d[v_i]$ eine obere Schranke ist, folgt $d[v_i] = \delta(s,v_i)$

Bellman-Ford-Algorithmus (allgemeiner Fall)

- Idee:

- Anfang: für alle Knoten gilt: $d[v] = \infty$
- Die Kanten werden *in allen möglichen Reihenfolgen* relaxiert:

- `BELLMAN-FORD(G, w, s)`

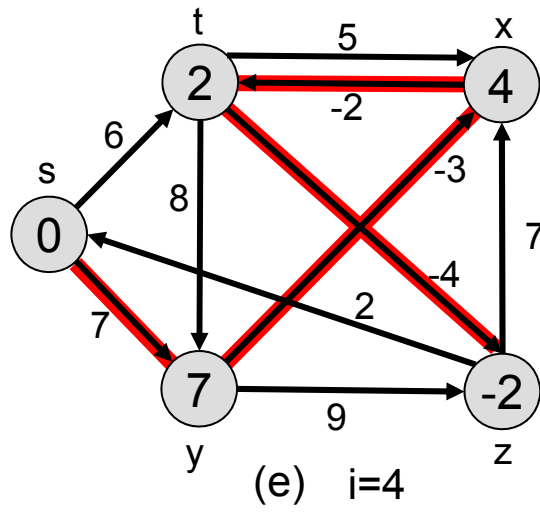
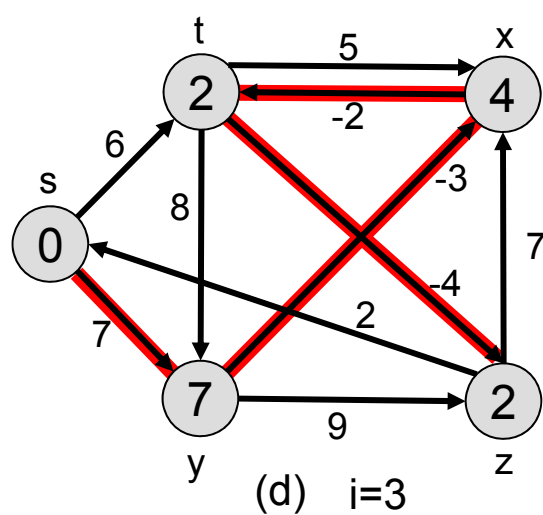
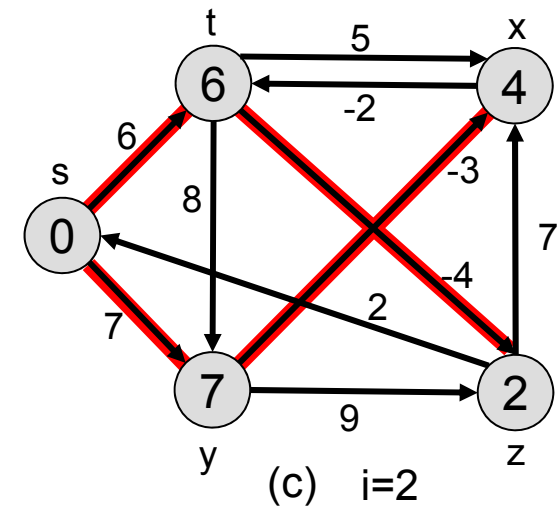
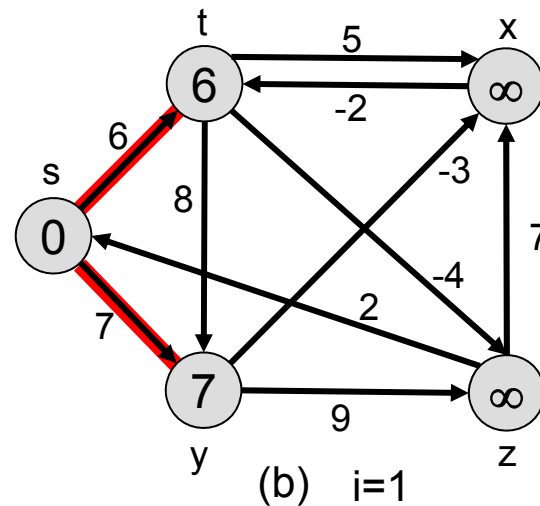
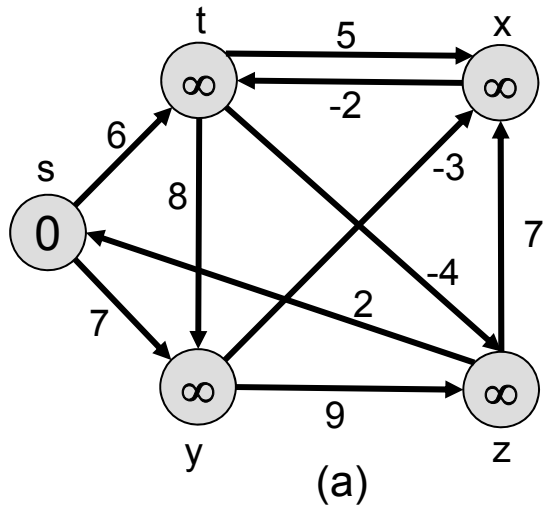
```
1 INITIALIZE-SINGLE-SOURCE(G, s)
2 for( i = 1 to |G.V|-1 )
3   for( each (u,v) ∈ G.E )
4     RELAX(u, v, w)
5 for( each (u,v) ∈ G.E )
6   if( d[v] > d[u] + w(u,v) ) return FALSE
7 return TRUE
```

- Laufzeit: $O(|V| |E|)$

Der Bellman-Ford-Algorithmus erlaubt negative Kantengewichte. Negative Zyklen werden erkannt, das Resultat ist nur korrekt, falls keine negativen Zyklen auftreten.

Bellman-Ford Algorithmus

■ Bsp:



Korrektheit des Bellman-Ford-Algorithmus

- Sei $G = (V, E)$ ein gerichteter Graph mit Gewichtsfunktion $w: E \rightarrow \mathbb{R}$, $s \in V$ ein beliebiger Startknoten.

- **Lemma 24.2:** (Korrekte Berechnung von $\delta(s, v)$ für $0 \leq \delta(s, v) < \infty$)
Angenommen, G enthält keine von s aus erreichbaren negativen Zyklen.
Für einen von s aus erreichbaren Knoten v gilt $d[v] = \delta(s, v)$.

Beweis: Sei $p = (s = v_0, v_1, v_2, \dots, v_k = v)$ ein kürzester Pfad von s nach v .

Nach i Iterationen der Schleife 2-4 wurde die Kantenmenge E i mal relaxiert.

=> Die ersten i Kanten von p wurden in der Reihenfolge, wie sie in p vorkommen, relaxiert.

=> $d[v_i] = \delta(s, v_i)$ (Pfad-Relaxations-Eigenschaft)

p hat höchstens $|V|-1$ Kanten (Zyklenfreiheit kürzester Pfade)

=> $d[v] = \delta(s, v)$

- **Korollar 24.3:** (Korrekte Berechnung von $\delta(s, v)$ für $\delta(s, v) = \infty$)

Für jeden Knoten v gilt: $s \rightsquigarrow v \iff d[v] < \infty$

Beweis: $s \rightsquigarrow v \implies d[v] < \infty$: folgt aus Lemma 24.2

$s \not\rightsquigarrow v \implies d[v] = \infty$: entspricht der Kein-Pfad-Eigenschaft

■ Theorem 24.4: (Korrektheit des Rückgabewerts)

1. G enthält keine von s aus erreichbaren negativen Zyklen:
 1. Für alle Knoten v gilt $d[v] = \delta(s, v)$
 2. G_π beschreibt einen Baum kürzester Pfade
 3. Der Rückgabewert ist TRUE
2. G enthält von s aus erreichbare negative Zyklen:
 1. Der Rückgabewert ist FALSE

Beweis:

1.1: folgt aus Lemma 24.2 / Korollar 24.3

1.2: folgt aus der Vorgängerteilgraph-Eigenschaft

1.3: folgt aus 1.1 und der Dreiecksungleichung:

Schleife 5-7 testet die Dreiecksungleichung für alle Kanten:

Für alle Kanten $e = (u, v)$ gilt:

$$\begin{aligned} d[v] &= \delta(s, v) \\ &\leq \delta(s, u) + w(u, v) = d[u] + w(u, v) \end{aligned}$$

Korrektheit des Bellman-Ford-Algorithmus

■ Beweis (cont.):

2.1: (Beweis durch Widerspruch) Sei $c = (v_0, v_1, v_2, \dots, v_k=v_0)$ ein von s erreichbarer Zyklus mit $w(c) = \sum_{i=1}^k w(v_{i-1}, v_i) < 0$

Annahme: Der Rückgabewert ist TRUE

$\Rightarrow d[v_i] \leq d[v_{i-1}] + w(v_{i-1}, v_i)$ für $i=1, \dots, k$

$$\begin{aligned} \Rightarrow \sum_{i=1}^k d[v_i] &\leq \sum_{i=1}^k d[v_{i-1}] + w(v_{i-1}, v_i) \\ &= \sum_{i=1}^k d[v_{i-1}] + \sum_{i=1}^k w(v_{i-1}, v_i) \\ &= \sum_{i=1}^k d[v_i] + d[v_0] - d[v_k] + w(c) \\ &= \sum_{i=1}^k d[v_i] + w(c) \quad \text{da } v_0 = v_k \\ &\Leftrightarrow 0 \leq w(c) \quad \text{⚡} \end{aligned}$$

Kürzeste Pfade von einem Startknoten in DAGs

■ Idee:

- Knoten können nur in der Reihenfolge der topologischen Sortierung in einem kürzesten Weg vorkommen
- betrachte Knoten in topologisch sortierter Reihenfolge => für alle Wege $p = (s=v_0, v_1, \dots, v_k)$ werden die Kanten in Reihenfolge des Weges relaxiert.

■ `DAG-SHORTEST-PATHS(G, w, s)`

```
1 INITIALIZE-SINGLE-SOURCE( $G, s$ )
2 VL = TOPOLOGICAL-SORT( $G$ )
3 while( VL  $\neq \emptyset$  )
4   u = VL.head; VL = VL.tail
5   for( each v  $\in$  u.Adj )
6     RELAX( u, v, w )
```

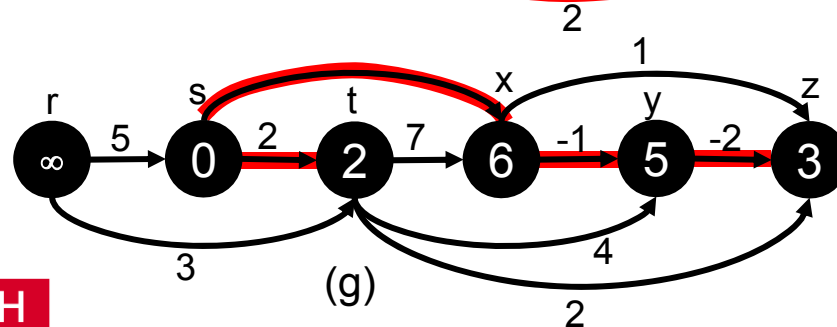
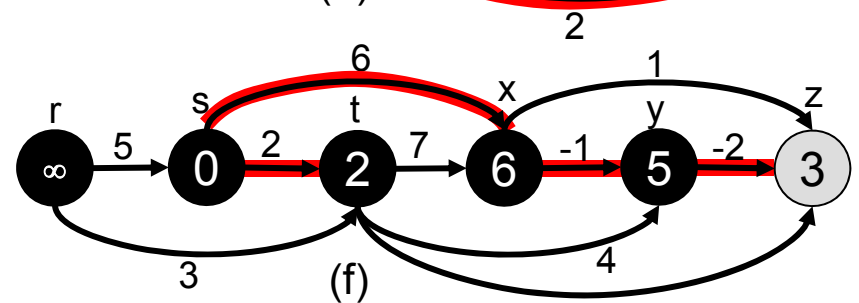
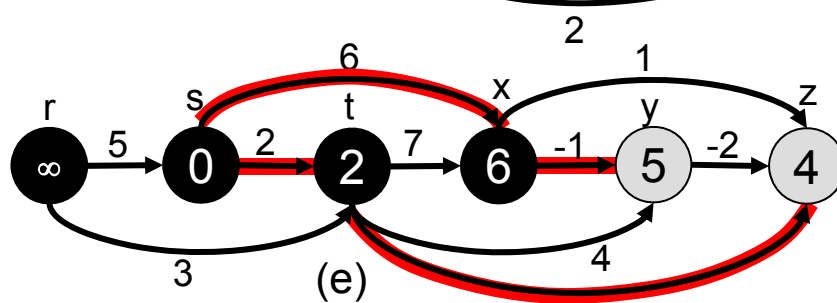
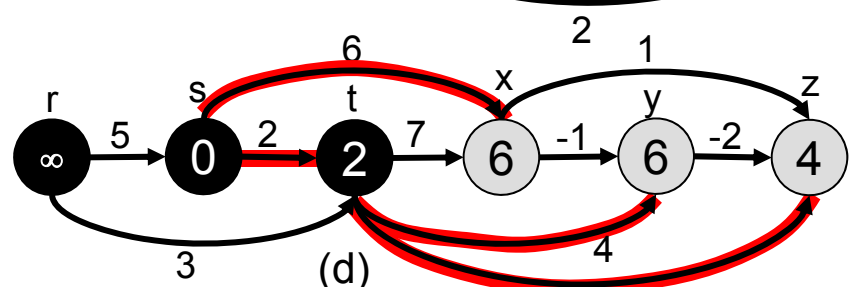
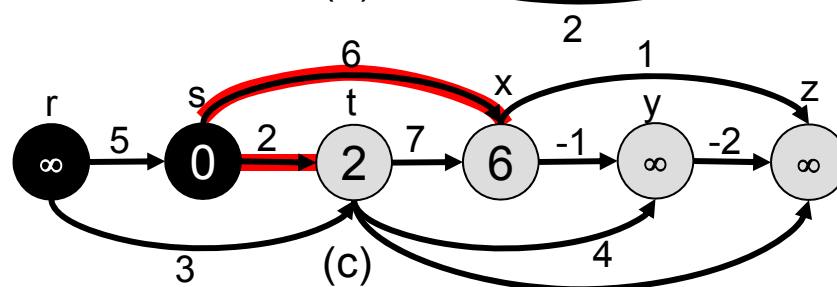
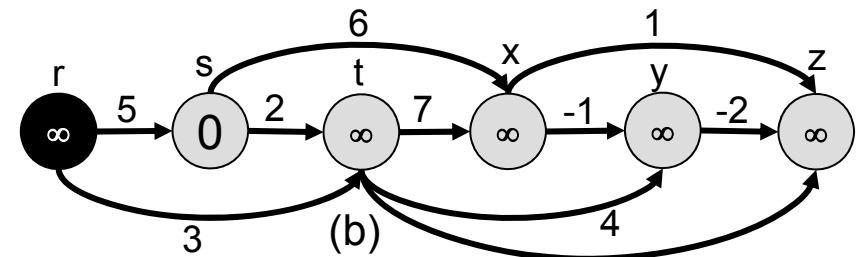
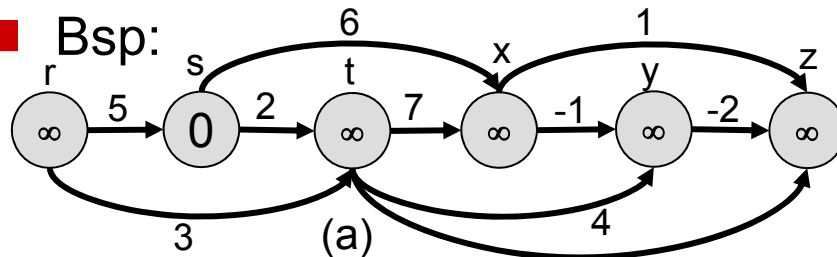
■ Laufzeit:

- Topologische Suche: $O(N + M)$
- Initialisierung: $O(N)$
- Schleife 3-6: $O(M)$

Insgesamt $T_{\text{DAG-SHORTEST-PATHS}}(N, M) = O(N+M)$

Kürzeste Pfade von einem Startknoten in DAGs

■ Bsp:



■ Startknoten: s

Korrektheit von DAG-SHORTEST-PATHS

■ **Theorem 24.5** (Korrektheit von DAG-SHORTEST-PATHS):

Sei $G=(V,E)$ ein DAG mit Startknoten s . Nach Terminierung von DAG-SHORTEST-PATHS gilt $d[v] = \delta(s,v)$ und G_π beschreibt einen Baum kürzester Pfade.

Beweis:

Fall 1: $s \not\rightarrow v \Rightarrow d[v] = \delta(s,v) = \infty$: folgt aus der Kein-Pfad-Eigenschaft.

Fall 2: $s \rightarrow v \Rightarrow d[v] = \delta(s,v)$. Sei $p=(s=v_0, v_1, v_2, \dots, v_k=v)$ ein kürzester Pfad von s nach v . Bzgl. der topologischen Sortierung t gilt $t(v_{i-1}) < t(v_i)$ für $i = \{1, \dots, k\}$.

\Rightarrow DAG-SHORTEST-PATHS bearbeitet die Kanten von p in der Reihenfolge von p

$\Rightarrow d[v] = \delta(s,v)$ (Pfad-Relaxations-Eigenschaft)

■ DAG-SHORTEST-PATHS

- arbeitet auch mit negativen Kantengewichten korrekt.
- eignet sich auch zur Berechnung längster Pfade.

Dijkstras Algorithmus (nicht-negative Kantengewichte)

- Einschränkung: alle Kantengewichte sind nicht-negativ
- Idee:
 - speichere in S alle Knoten, deren kürzeste Pfade bereits berechnet worden sind (Anfang $S = \{s\}$)
 - gehe von S aus zum Knoten $u \notin S$, der am dichtesten an s liegt und berechne $d[u]$
 - speichere alle Knoten $v \notin S$ in einer Priority-Queue, Priorität: obere Schranke $d[v]$

- **DIJKSTRA**(G, w, s)

 INITIALIZE-SINGLE-SOURCE(G, s)

$S = \emptyset$

$Q = G.V$

while($Q \neq \emptyset$)

$u = \text{EXTRACT-MIN}(Q)$

$S = S \cup \{u\}$

for(each $v \in u.\text{Adj}$)

 RELAX-DECREASE(u, v, w, Q)

RELAX-DECREASE(u, v, w, Q)

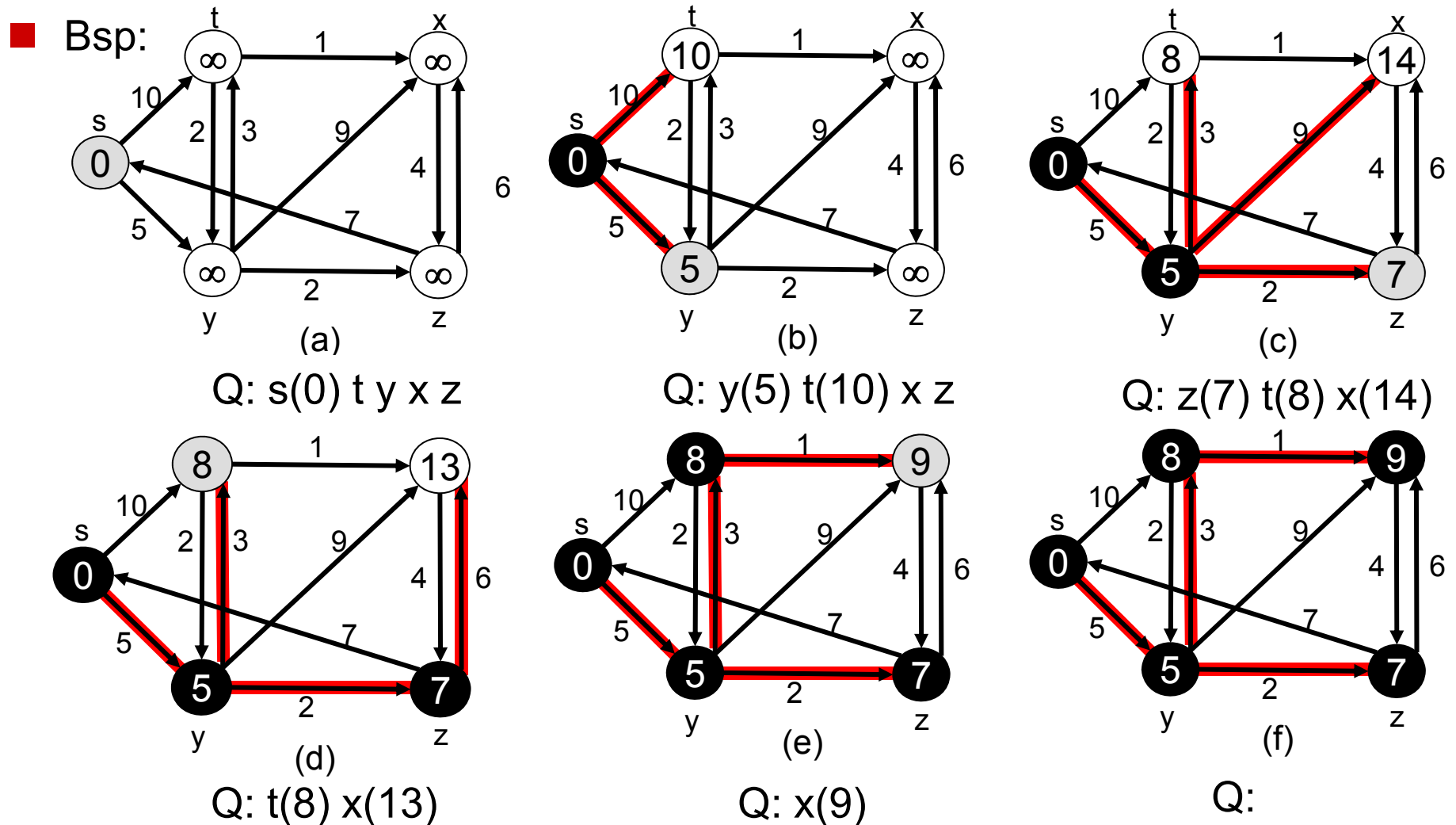
if($d[v] > d[u] + w(u,v)$)

$d[v] = d[u] + w(u,v);$

$\pi[v] = u$

 DECREASE-KEY($Q, v, d[v]$)

Dijkstras Algorithmus



Beschreibung Priority Queue $Q: v(d[v]) \ u(d[u]) \dots$

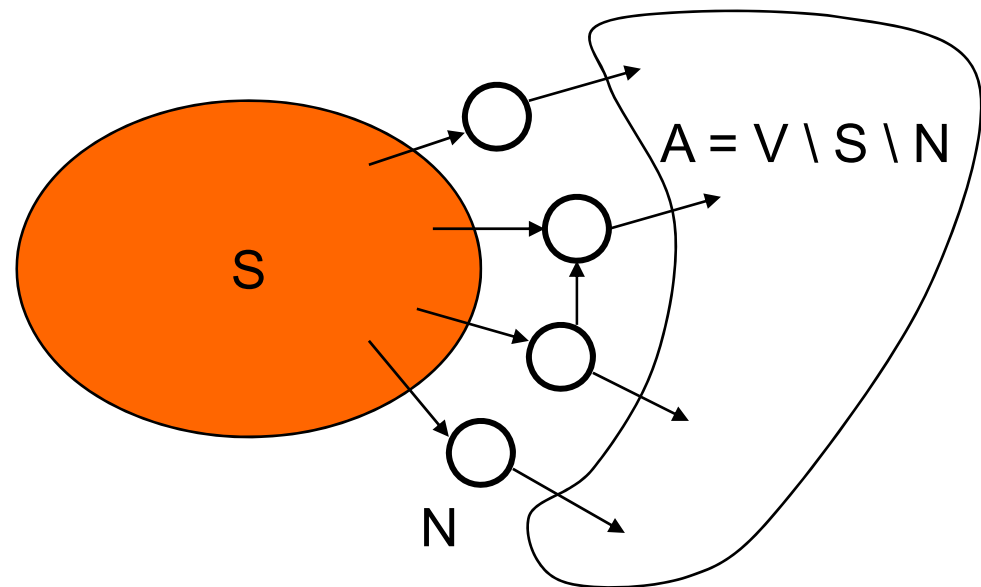
■ Theorem 24.6 (Korrektheit des Dijkstra-Algorithmus):

Sei $G=(V,E)$ ein gerichteter Graph mit Gewichtsfunktion $w: E \rightarrow \mathbb{R}_0^+$, $s \in V$ ein Startknoten. Nach Terminierung des Dijkstra-Algorithmus gilt für alle $u \in V$: $d[u] = \delta(s,u)$.

Beweis: Sei $\delta^S(s,u)$ die Länge eines kürzesten Weges, der nur Knoten aus S enthält. Sei $N = \{ v \in V \mid d[v] < \infty \} \setminus S$

Invariante der while-Schleife:

1. $\forall v \in S: d[v] = \delta(s,v)$
 2. $\forall v \in N: d[v] = \delta^S(s,v)$
- Es ist ausreichend, den Zeitpunkt der Einfügung von v in S zu betrachten (Konvergenzeigenschaft)
 - Initialisierung: $S = \emptyset$, $N = \{s\}$ die Invariante ist somit erfüllt
 - Terminierung: $S = V$, mit Teil 1 der Invariante ist das Theorem bewiesen.



Dijkstras Algorithmus

■ Beweis von Theorem 24.6 (cont.)

- Fortsetzung: Sei $u \in V - S$ mit $d[u]$ minimal wie im Algorithmus gewählt

Teil 1: Zeige $d[u] = \delta(s,u)$

Fall 1: $d[u] = \infty$, d.h. $u \notin N$

=> es gibt keine Kante $e=(v,u)$ mit $v \in S$ und $\delta(s,v) < \infty$, da e relaxiert wäre. Dies gilt auch für alle $v \in V - S$, da $d[u]$ minimal ist.

=> es gibt keinen Pfad $s \rightsquigarrow u \Rightarrow \delta(s,u) = \infty$

Fall 2: $d[u] < \infty$, d.h. $u \in N$. Sei $p = (s, \dots, x, y, \dots, u)$ ein kürzester Pfad von s zu u . (x,y) sei so gewählt, dass y der erste Knoten mit $y \notin S$ gilt.

=> $d[u] \leq d[y]$ (wg. Wahl von u)
= $\delta^S(s,y)$ (wg. Schleifeninvariante)
= $\delta(s,y)$ (da $x \in S$ und die Kante (x,y) relaxiert wurde)
 $\leq \delta(s,u)$ (da y auf p liegt und alle Kantengewichte ≥ 0)

Teil 2: Gültigkeit der Invariante nach Durchlaufen der Schleife

1. $\delta(s,u) \leq d[u] \leq \delta(s,u)$ (wg. Eigenschaft der oberen Schranke und Teil 1)
2. folgt direkt aus Schleifeninvariante und Relaxation aller zu u inzidenten Kanten (u,v) .

■ Laufzeit:

- while-Schleife: $N = |V|$ Iterationen
 - ◆ N EXTRACT-MIN Operationen auf einer N -elementigen Warteschlange
 - ◆ for-Schleife: insgesamt $M = |E|$ Iterationen (jede Kante ein mal)
 - ◆ M RELAX- und damit DECREASE-KEY Operationen auf einer N -elementigen Warteschlange
- Implementierung der Warteschlange:
 - ◆ durch binären Heap mit zusätzlichem N -elementigen Array zum Auffinden der Knoten im Heap (für DECREASE-KEY)
 - ◆ EXTRACT-MIN: $O(\log N)$ DECREASE-KEY: $O(\log N)$
- $T_{\text{Dijkstra}}(N, M) = O(N \log N) + O(M \log N) = O(M \log N)$
- Fibonacci-Heaps:
 - ◆ EXTRACT-MIN: $O(\log N)$ DECREASE-KEY: $O(1)$ (amortisiert)
 - ◆ $T_{\text{Dijkstra}}(N, M) = O(N \log N + M)$

5.7 Weitere Graphprobleme im Überblick

- Das All-Pairs-Shortest-Paths Problem

- berechne kürzeste Wege zwischen allen Knotenpaaren

Ausgabe: $N \times N$ Matrix $D = (d_{ij})$ mit allen paarweisen Distanzen

$N \times N$ Matrix $\Pi = (\pi_{ij})$ mit allen Vorgängern

- Annahme: nicht-negative Kantengewichte

- Algorithmus: N * Dijkstras Algorithmus

- Laufzeit: $O(N(N+M) \log N)$

- sonst:

- Algorithmus: N * Bellman-Ford Algorithmus

- Laufzeit $O(N^2 M)$

- Repräsentation von Graphen für all-pairs:

- gewichtete Adjazenzmatrix:

$$w_{ij} = \begin{cases} 0 & : i = j \\ w(i, j) & : i \neq j \wedge (i, j) \in E \\ \infty & : i \neq j \wedge (i, j) \notin E \end{cases}$$

Das All-Pairs-Shortest-Paths Problem

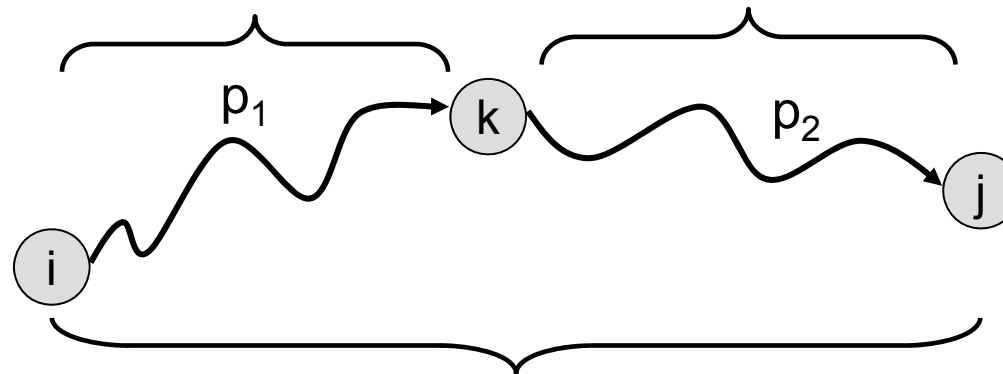
■ Floyd-Warshall-Algorithmus:

- $d_{ij}^{(k)}$: Länge des kürzesten Weges von i nach j , auf dessen Pfad neben i und j nur Knoten aus $\{1, \dots, k\}$ besucht werden

■ Rekursive Definition:
$$d_{ij}^{(k)} = \begin{cases} w_{ij} & : k = 0 \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & : k > 0 \end{cases}$$

all intermediate vertices in $\{1, 2, \dots, k-1\}$

all intermediate vertices in $\{1, 2, \dots, k-1\}$



p: all intermediate vertices in $\{1, 2, \dots, k-1\}$

Floyd-Warshall Algorithmus

■ FLOYD-WARSHALL(W)

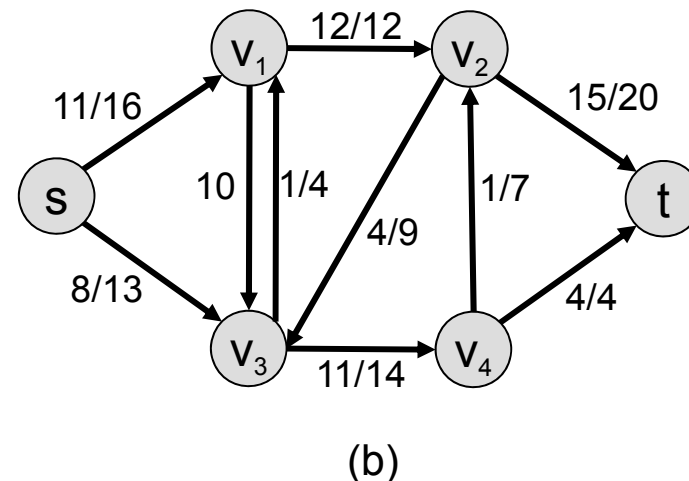
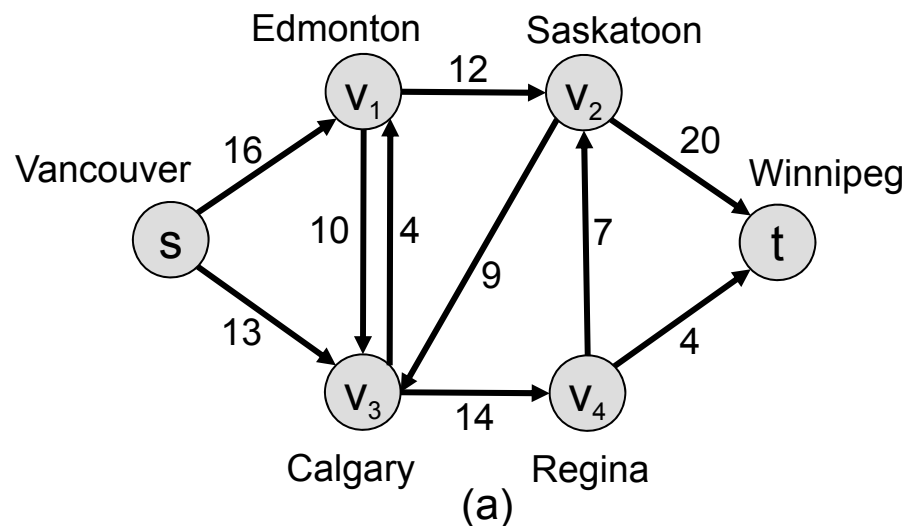
```
1 n = W.number_of_rows
2  $D^{(0)} = W$ 
3 for( k = 1 to n )
4   for( i = 1 to n )
5     for( j = 1 to n )
6        $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$ 
7 return  $D^{(n)}$ 
```

■ Laufzeit: $T(N) = \Theta(N^3)$ Platz: $S(N) = \Theta(N^2)$

Maximaler Fluss

■ Maximaler Fluss (Maximum Flow):

- Gerichteter Graph $G=(V,E)$ mit Kantenkapazitäten $c:E \rightarrow \mathbb{R}^+$
- zwei ausgezeichnete Knoten: Quelle s und die Senke t
- Was ist der maximale Fluss $f(s,t)$ unter den Randbedingungen:
 - ◆ Capacity constraint: $f(u,v) \leq c(u,v)$
 - ◆ Skew symmetry: $f(u,v) = -f(v,u)$
 - ◆ Flow conservation: $\sum_v f(u,v) = 0 \quad \forall u \in V \setminus \{s,t\}$



Maximaler Fluss

- Augmentierender Pfad:

- Pfad $(s=v_0, v_1, v_2, \dots, v_k = t)$ mit $f(v_i, v_{i-1}) \leq c(v_i, v_{i-1})$ für alle Kanten $i = 1, \dots, k$

- Ford-Fulkerson-Methode:

- FORD-FULKERSON-METHOD(G, s, t)

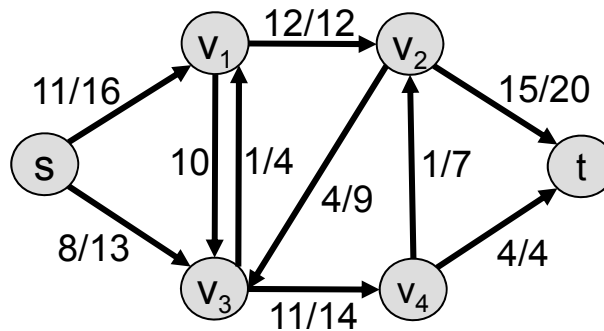
```
1 initialize flow  $f$  to 0
2 while( there exists an augmenting path  $p$  )
3   augment flow  $f$  along  $p$ 
4 return  $f$ 
```

- Residuale Netzwerke (residual networks):

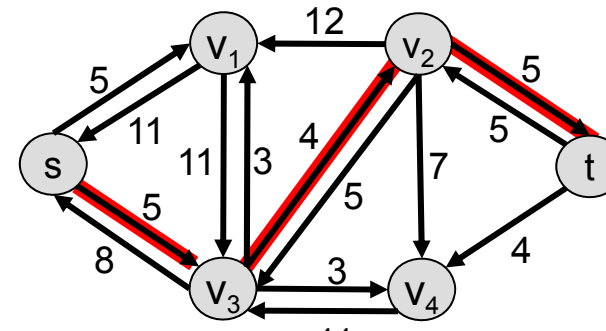
- setze Kapazität auf residuale Kapazität: $c_f(u, v) = c(u, v) - f(u, v)$
 - füge Kanten in Gegenrichtung mit Kapazität $c_f(v, u) = f(u, v)$ ein

Maximaler Fluss

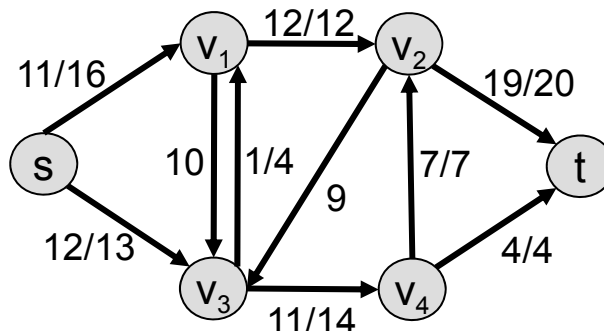
■ Bsp:



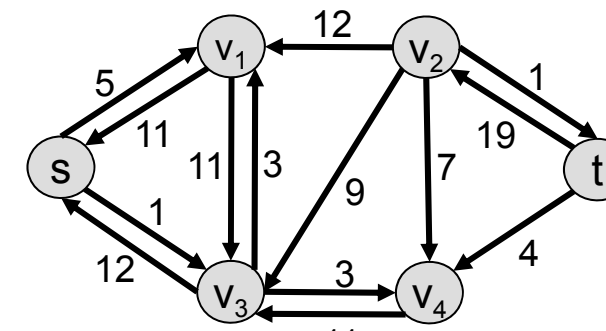
(a)



(b)



(c)

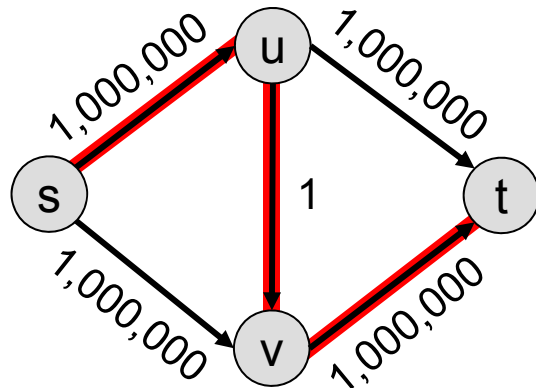


(d)

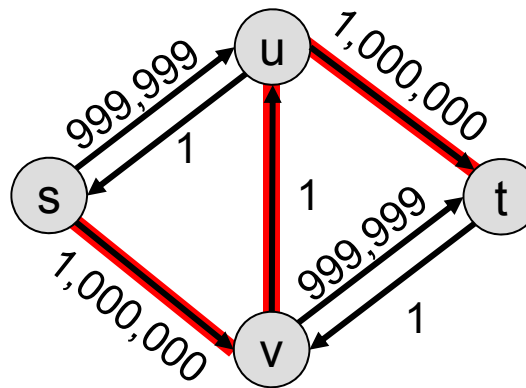
- a) Flussnetzwerk: Kante: Fluss / Kapazität
- b) Residuales Netzwerk, Augmentierender Pfad
- c) Flussnetzwerk nach Augmentierung
- d) Residuales Netzwerk

Maximaler Fluss

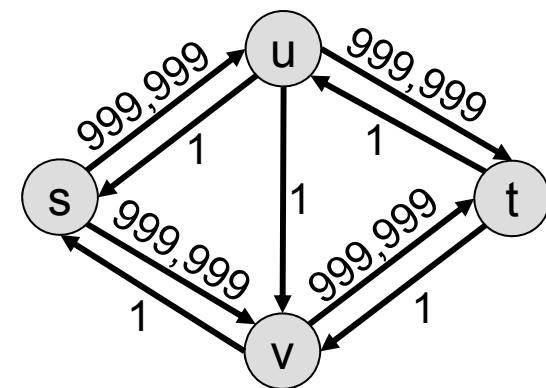
- Laufzeit Ford-Fulkerson: $O(E |f^*|)$



(a)



(b)



(c)

- Edmonds-Karp Algorithmus:

- Wähle als augmentierenden Pfad den kürzesten Weg (uniformes Kantengewicht von 1) von s nach t
- Man kann zeigen, dass die Distanz von s zu t im Residualen Netzwerk monoton steigt => # Augmentierungen = $O(|V||E|)$
- Finden eines kürzesten Weges bei uniformen Kantengewichten: BFS-Algorithmus, $O(|E|)$ Zeit
- Gesamtzeit: $O(|V| |E|^2)$

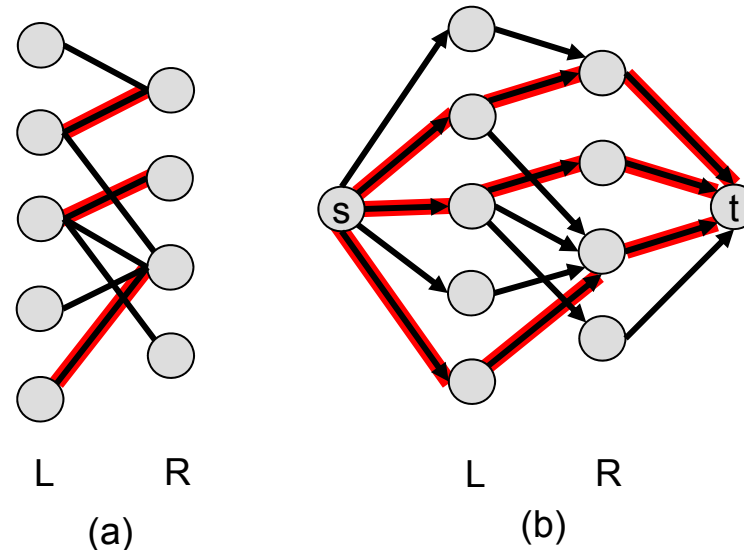
Maximales bipartites Matching

■ Matching

- Teilmenge $M \subseteq E$, so dass $\forall v \in V: |\{e \in M \mid v \in e\}| \leq 1$

■ Maximales bipartites Matching:

- Bipartiter Graph $G = (V \cup U, E)$
- Finde maximales Matching M , d.h. $|M|$ maximal



- Beschreibung als Flussproblem, Laufzeit: $O(|V||E|)$

■ maximales, gewichtetes bipartites Matching:

- Laufzeit: $O(|V||E|\log |V|)$

Schwere Graphprobleme

- Für viele Graphprobleme sind keine Algorithmen mit Laufzeit $O(|V|^k)$ für konstantes k bekannt (polynomielle Laufzeit).
- Theoretische Betrachtungen zeigen, dass wenn eines dieser Probleme in polynomieller Zeit lösbar ist, dann gilt dies auch für sehr viele andere (siehe Kap. 7).
- Beispiele:
 - Hamiltonian Path: gibt es einen einfachen Kreis der Länge $|V|$?
 - Subgraph: Ist Graph G ein Teilgraph von G' ?
 - Graphisomorphie: Sind Graphen G und G' isomorph?
 - Clique: Gibt es einen vollständig verbundenen Teilgraphen mit k Knoten?
 - Dreifärbbarkeit: Gibt es eine Funktion $f:V \rightarrow \{0,1,2\}$ mit $f(v) \neq f(u) \quad \forall \{u,v\} \in E$?

HINWEIS ZUR KLAUSUR

■ Erlaubte Hilfsmittel:

- Kugelschreiber, etc.
- keine Elektronik, keine Lupen
- ein Din-A4 Blatt, **handschriftlich** beschrieben (nicht bedruckt, nicht kopiert)

Vorderseite

zur freien Verfügung	Max Mustermann <Matr.Nr>
-------------------------	-----------------------------

Rückseite

zur freien Verfügung	Max Mustermann <Matr.Nr>
-------------------------	-----------------------------

Hinweis: Das Blatt wird zu Beginn der Klausur abgestempelt und mit abgegeben (geht nicht in die Benotung ein).