Adjazenzlisten

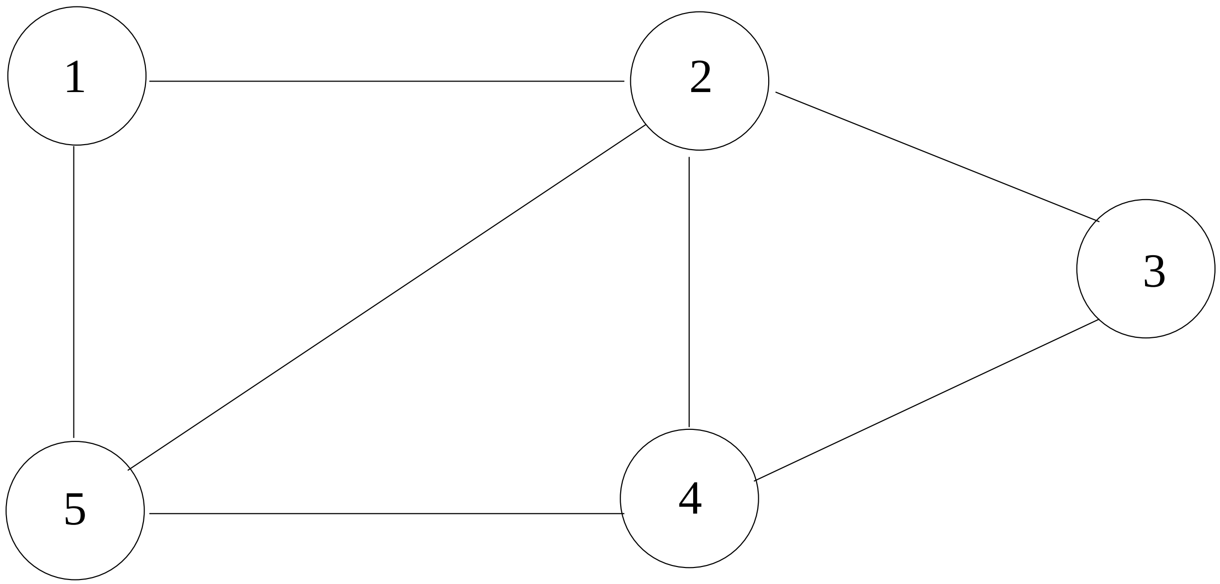
Für einen Graphen $G = (V, E)$ besteht die Adjazenzlistenrepräsentation aus einem Feld **Adj** von $|V|$ Listen.

Jeder dieser Listen ist genau einem Knoten aus V zugeordnet.

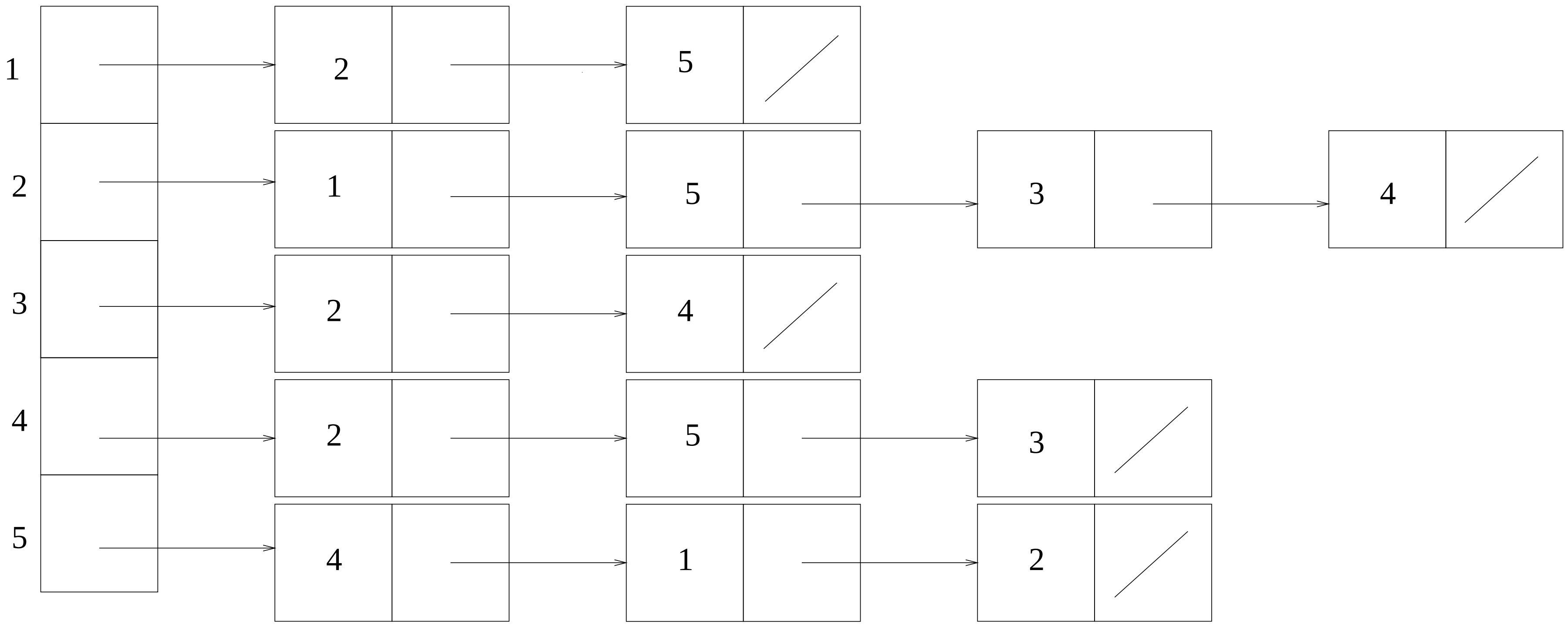
Für jedes $u \in V$ enthält **Adj[u]** eine Liste von Zeigern zu denjenigen $v \in V$ für die gilt $(u, v) \in E$.

Für gerichtete Graphen ist die Summe aller Listenlängen gleich $|E|$, für ungerichtete Graphen gleich $2|E|$.

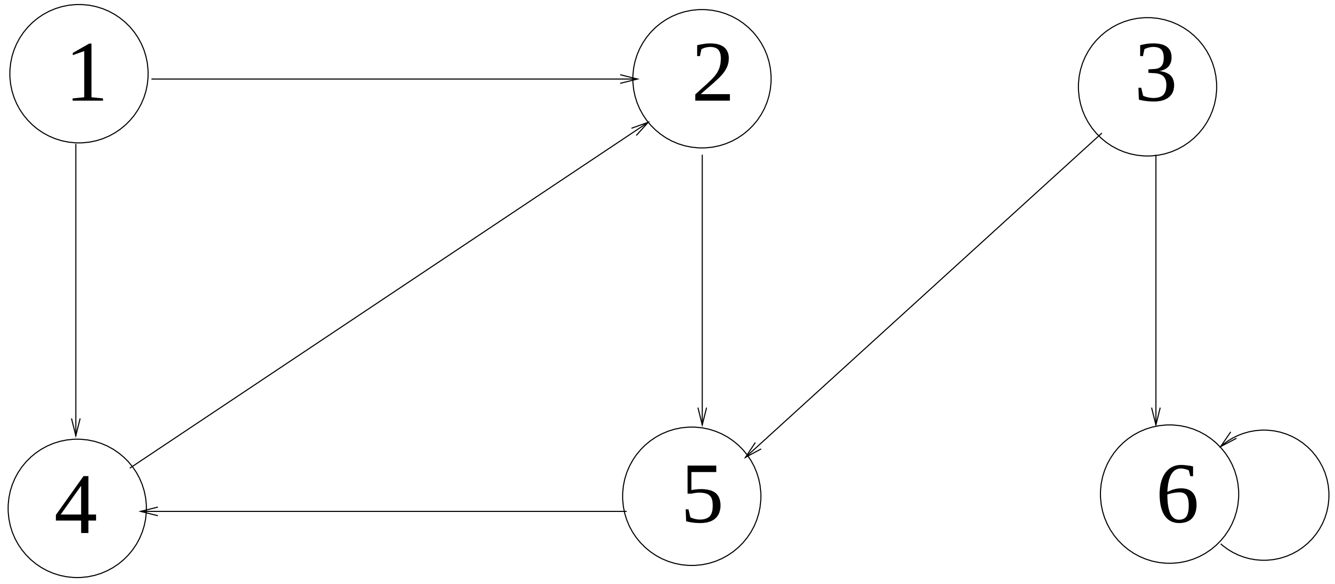
In beiden Fällen genügt $O(max(V, E)) = O(V + E)$ Speicher.



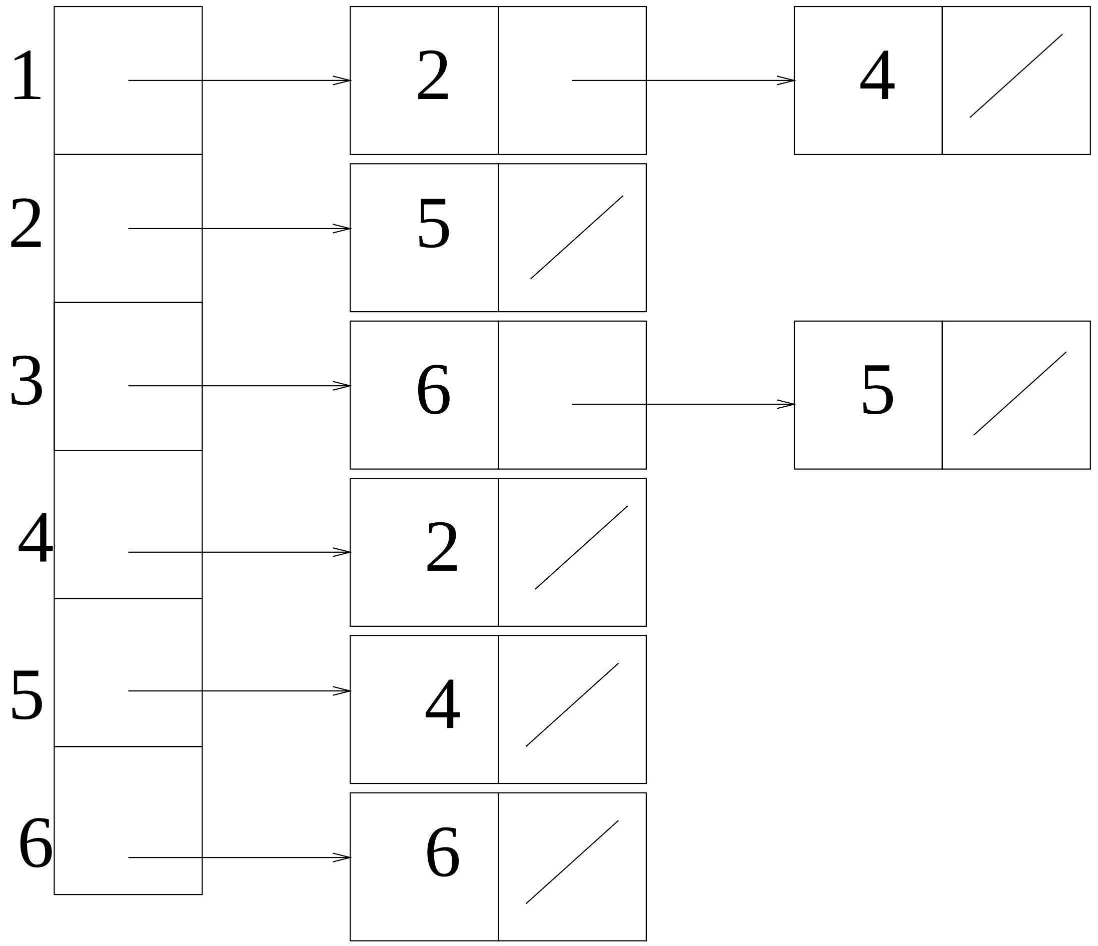
(a)



(b)



(a)



(b)

Gewichtete Graphen

Adjazenzlisten können leicht auf gewichtete Graphen verallgemeinert werden.

Ein Graph $G = (V, E)$ heißt gewichtet, falls es eine Gewichtsfunktion $w : E \to \mathcal{R}$ gibt, die jeder Kante ein Gewicht aus dem reellen Zahlen zuordnet.

Dieses Gewicht wird dann einfach mit in der Adjazenzliste gespeichert.

Suche ist ein bischen aufwendig. Statt Liste könnte man dann aber auch Hashtabelle o.ä. nehmen.

Adjazenzmatrizen

Für einen Graphen $G = (V, E)$ ist die Adjazenzmatrix eine Matrix der Dimension $|V| \times |V|$ mit

$$a_{i,j} = \begin{cases} 1 & \text{für } (i,j) \in E \\ 0 & \text{sonst} \end{cases}$$

Der Speicheraufwand für Adjazenzmatrizen beträgt V^2.

$$
\begin{array}{c|ccccc|}
 & 1 & 2 & 3 & 4 & 5 \\
\hline
1 & 0 & 1 & 0 & 0 & 1 \\
2 & 1 & 0 & 1 & 1 & 1 \\
3 & 0 & 1 & 0 & 1 & 0 \\
4 & 0 & 1 & 1 & 0 & 1 \\
5 & 1 & 1 & 0 & 1 & 0 \\
\hline
\end{array}
$$

(c)

$$
\begin{array}{c|cccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
1 & 0 & 1 & 0 & 1 & 0 & 0 \\
2 & 0 & 0 & 0 & 0 & 1 & 0 \\
3 & 0 & 0 & 0 & 0 & 1 & 1 \\
4 & 0 & 1 & 0 & 0 & 0 & 0 \\
5 & 0 & 0 & 0 & 1 & 0 & 0 \\
6 & 0 & 0 & 0 & 0 & 0 & 1 \\
\hline
\end{array}
$$

(c)

Adjazenzmatrizen

Adjazenzmatrizen A sind für ungerichtete Graphen symmetrisch, das heißt, $A^T = A$.

(Für $A = (a_{i,j})$ ist $A^T = (a_{j,i})$ ist die transponierte Matrix.)

Gewichtete Graphen können ebenso in Adjazenzmatrizen gespeicher werden.

Die Einträge sind dann die Gewichte oder NIL.

Breitensuche

Sei $G = (V, E)$ ein Graph und $v \in V$ ein ausgewählter Startknoten.

Wir wollen systematisch alle von v aus erreichbaren Knoten besuchen.

Eigenschaft:

- gleichzeitig wird Breitensuchebaum aufgebaut
  In einem solchen ist v die Wurzel und jeder von v für aus erreichbare Knoten s besteht ist der Pfad von der Wurzel zu s der minimale Pfad.

Breitensuche

Maxime der Breitensuche:

- Besuche erst alle Söhne, bevor du einen Enkel besuchst.
- genauer: Queue mit offenen Knoten+Färbung der Knoten
  weiss=unbesucht, grau=besucht/entdeckt, aber Söhne noch nicht, schwarz=vollständig abgearbeitet.

Variablen:

- Q ist FIFO-QUEUE
- jeder Knoten u bekommt die Attribute
  1. color ($\in$ { white, grey, black })
  2. d (Distanz von s)
  3. π (Vorgänger von u)

```
BFS(G, s)
for each vertex u ∈ V[G] - {s}
   do color[u] ← WHITE
      d[u] ← ∞
      π[u] ← NIL
color[s] ← GRAY
d[s] ← 0
π[s] ← NIL
Q ← {s}
while Q ≠ ∅
   do u ← head[Q]
      for each v ∈ Adj[u]
         do if color[v] = WHITE
            then color[v] ← GRAY
                 d[v] ← d[u] + 1
                 π[v] ← u
                 ENQUEUE(Q,v)
      DEQUEUE(Q)
      color[u] ← BLACK
```

Z1-4: alle Knoten weisseln

Z5: s wird grau

Z6: distanz von s zu s ist 0

Z7: s hat keinen Vorgänger

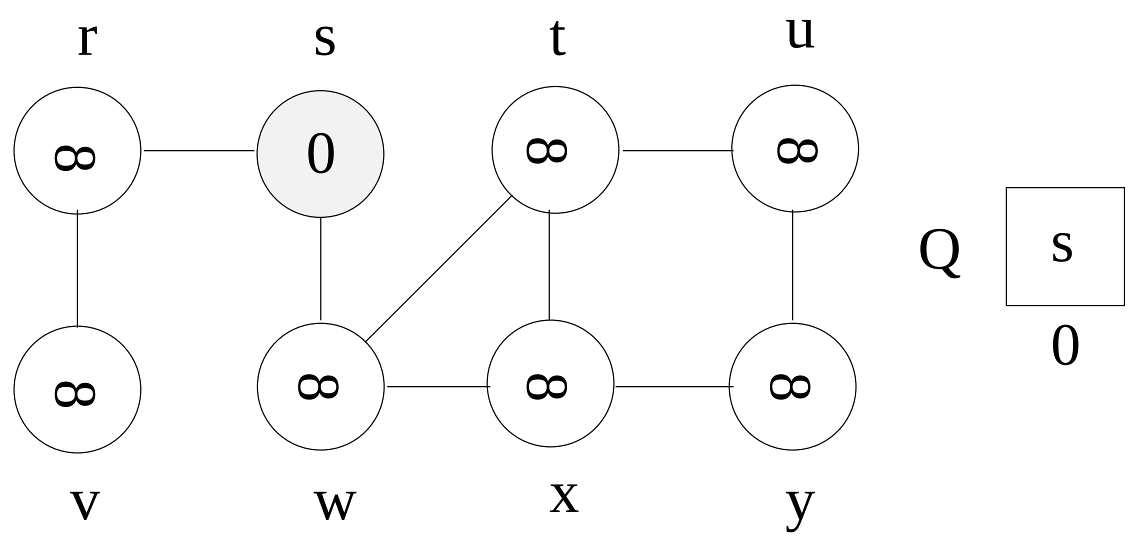
Z10: ersten grauen Knoten u aus Q entnehmen

Z11: für jeden Nachfolger v von u:

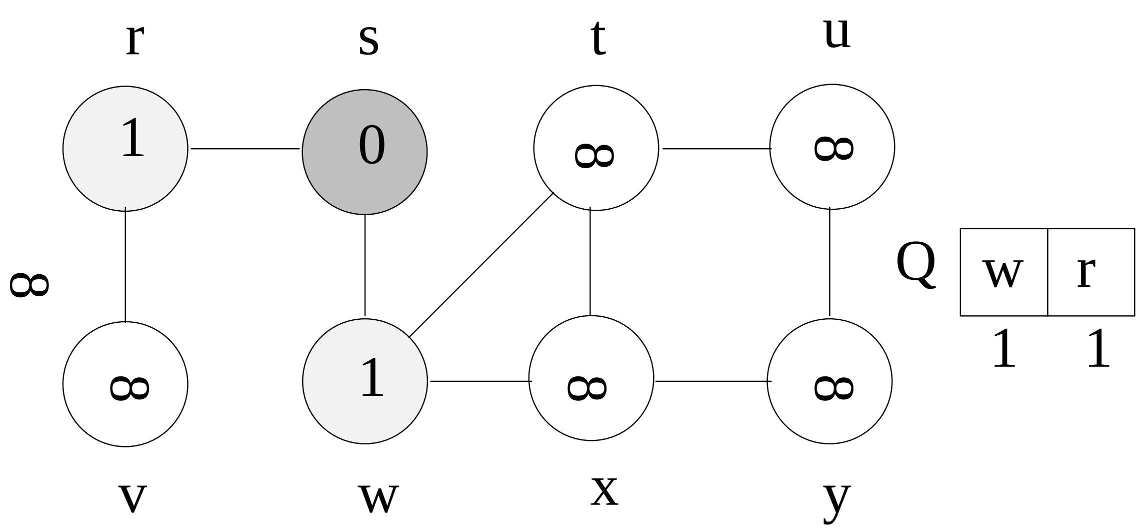
Z12ff: falls v weiss, grau färben, d undπ von v setzen, in Q einfügen

Z17: u aus Q entfernen
könnte besser noch mit POP statt HEAD in Z10 gleichzeitig geschehen

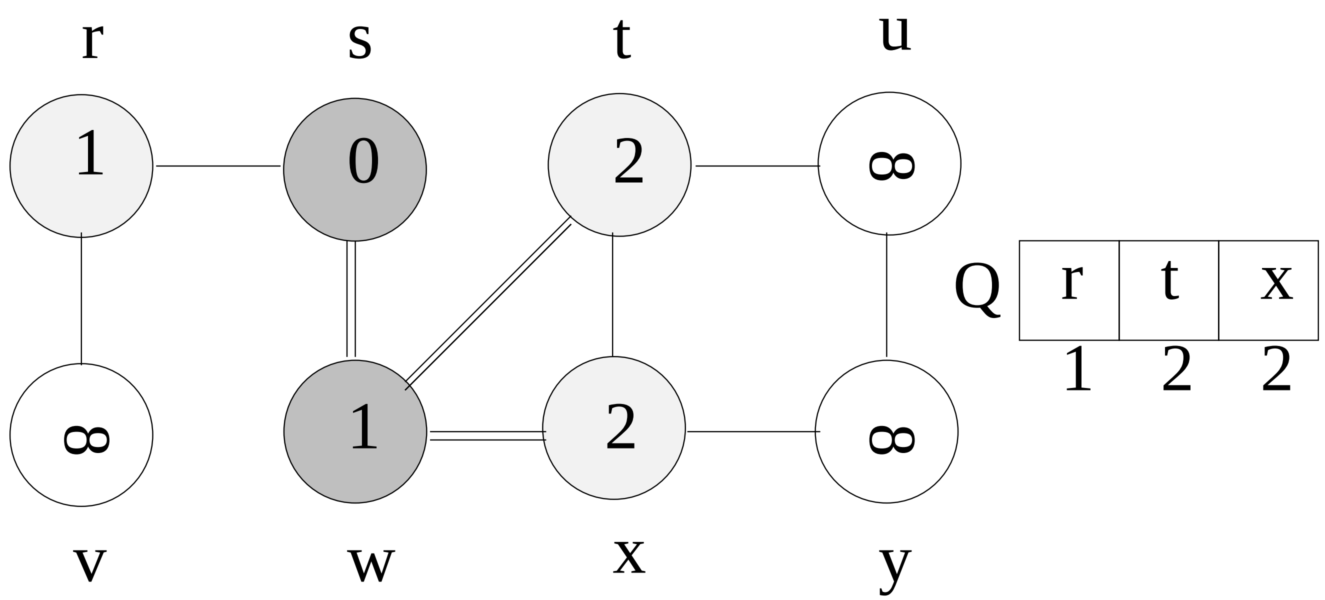
Z19: markiere u als abgearbeitet.



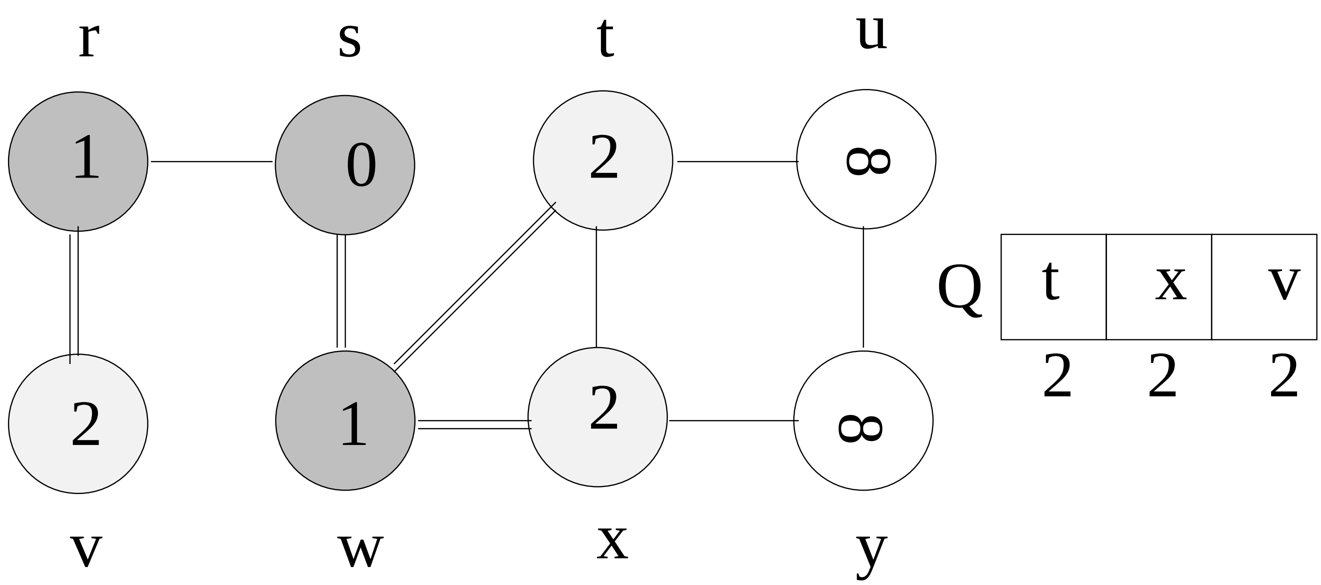
(a)



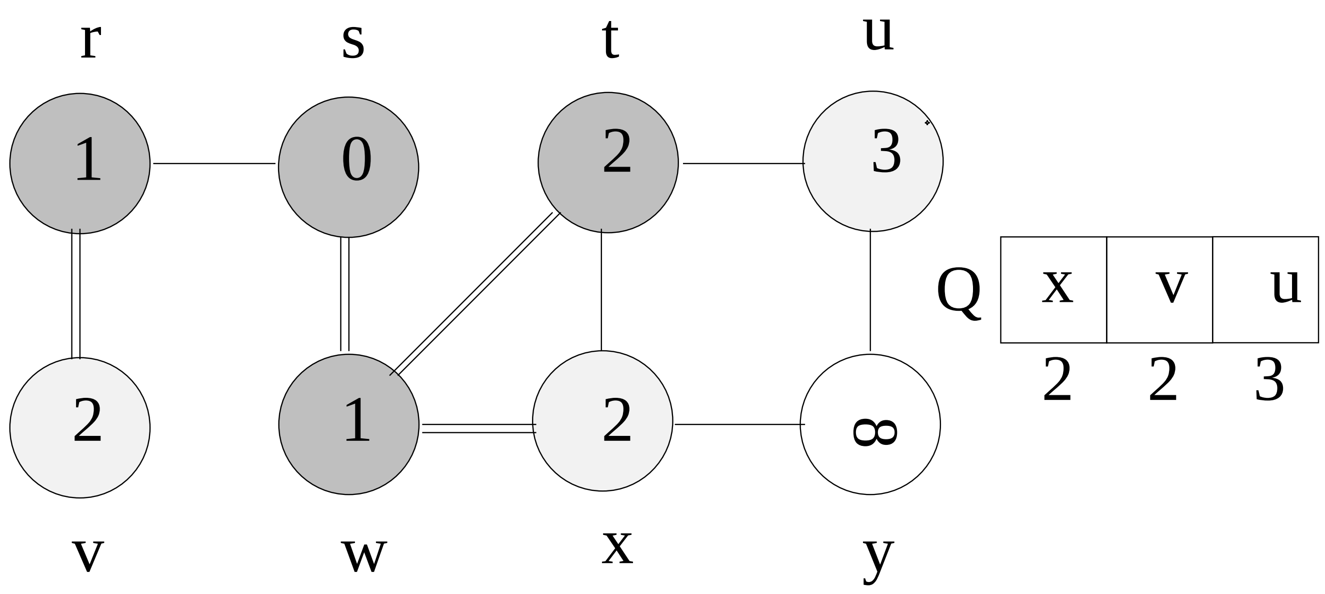
(b)



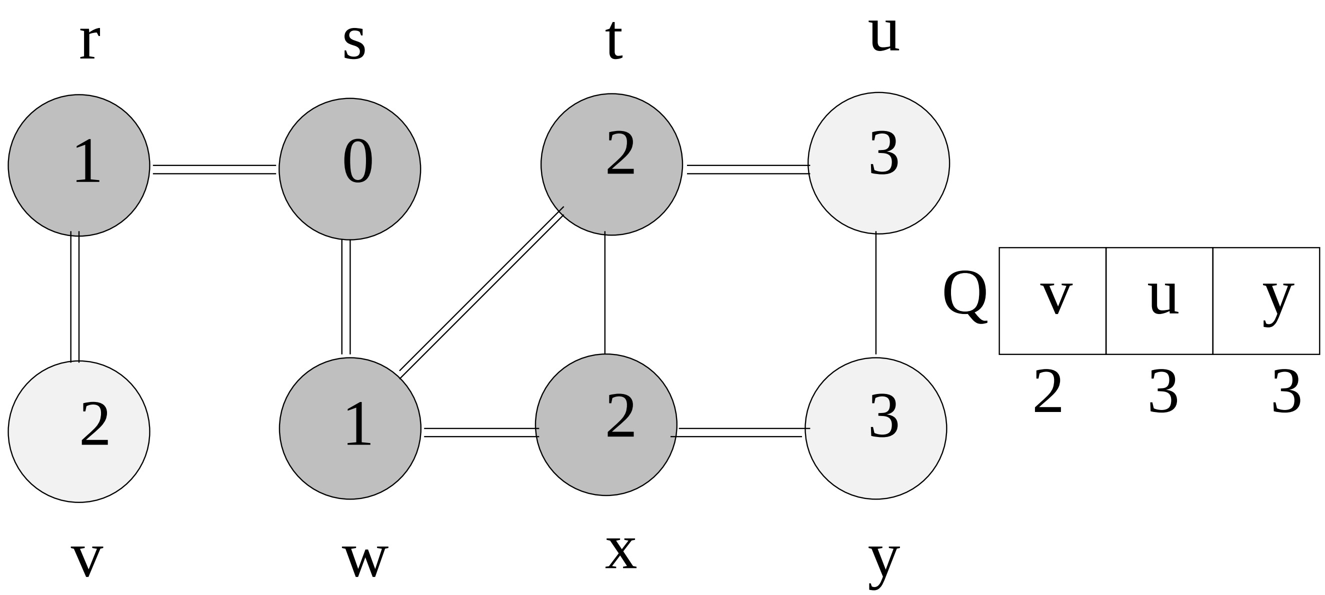
c)



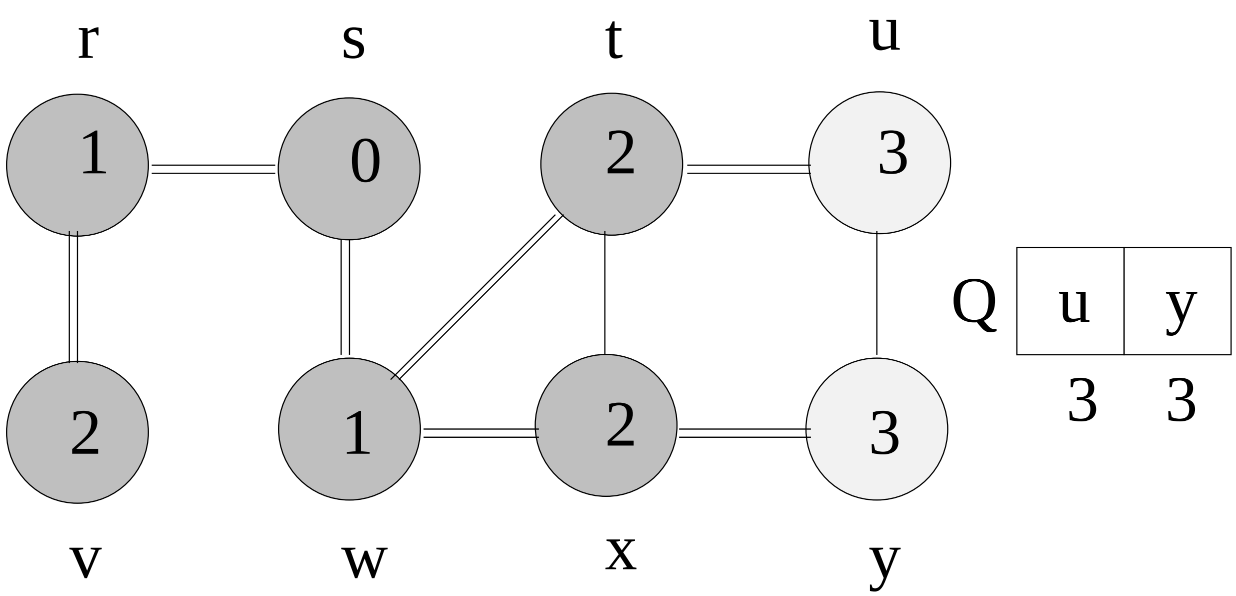
(d)



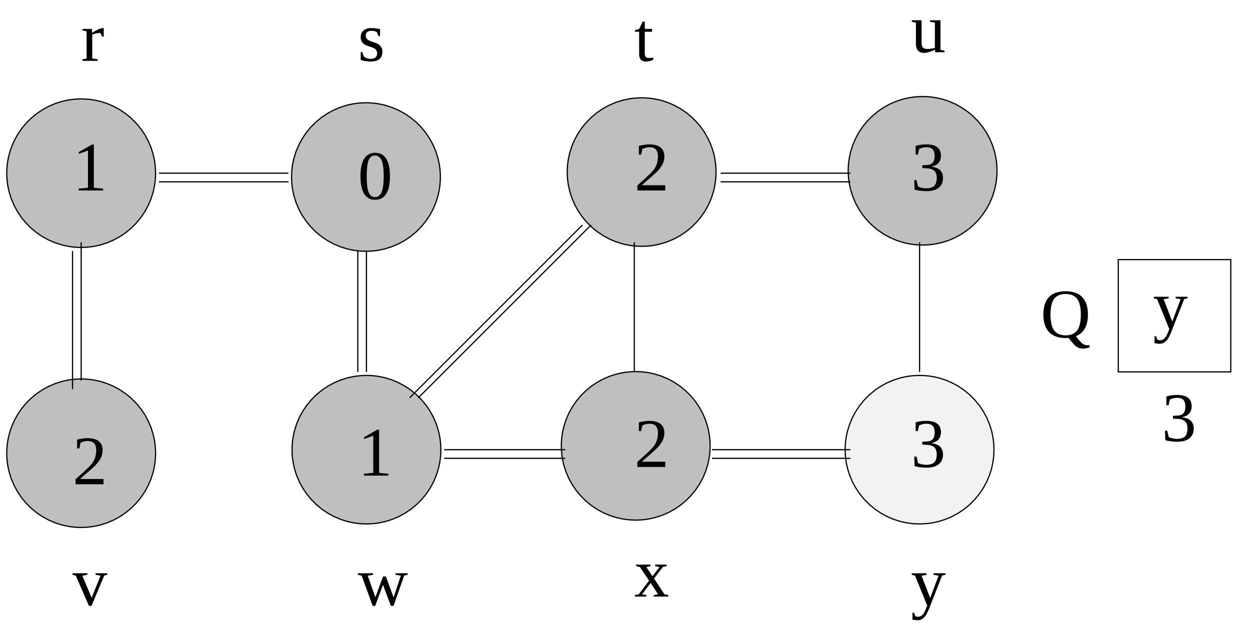
(e)



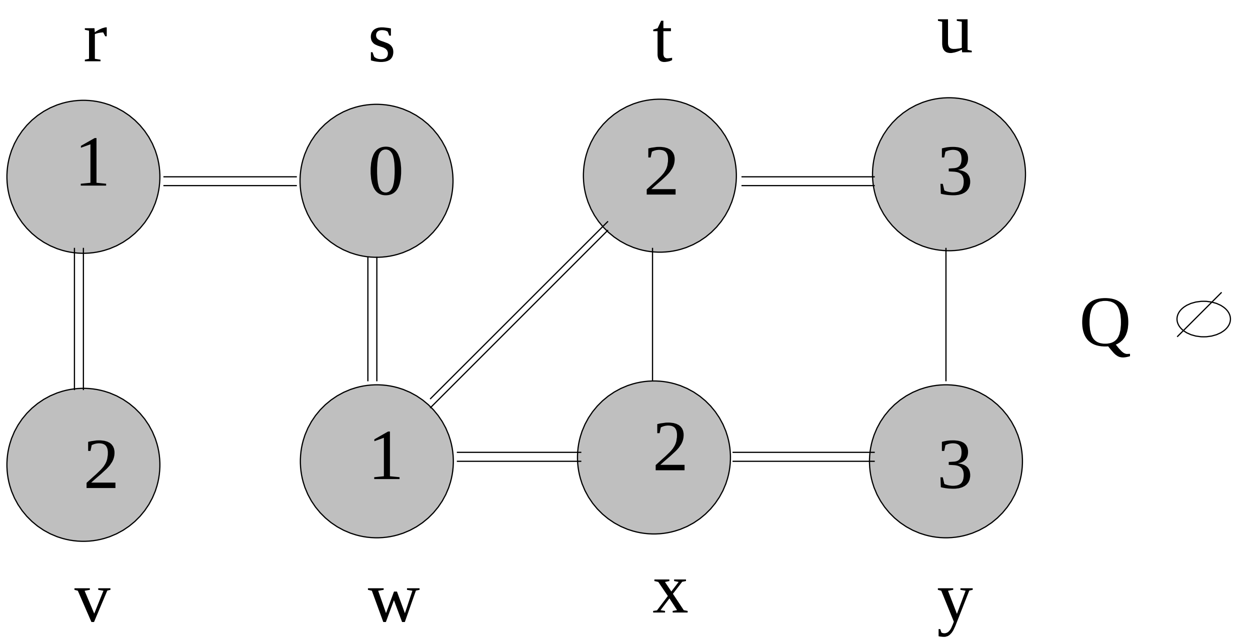
(f)



(g)



(h)



(i)

Analyse

Laufzeit

- weisseln: $O(V)$
- Queue-Operationen: $O(V)$
  nach Z7 wird kein Knoten mehr weiss gefärbt, nur weisse werden in Q eingefügt, also jeder Knoten nur einmal eingefügt und ausgefügt.
- Adjazenzlisten ablaufen: $\Theta(E)$
  Adjazenliste von v wird nur abgelaufen, wenn er DEQUEUEd wird, also genau einmal.
- Gesamt: $O(V + E)$

Distanz

Def Definiere die Distanz $\delta(s, v)$ zwischen zwei Knoten s und v als das minimum der Pfadlängen aller Pfade von s nach v.

Was wir zeigen wollen ist $d[v] = \delta(s, v)$.

Lemma 23.1 Sei $G = (V, E)$ ein gerichteter oder ungerichteter Graph und $s \in V$. Dann gilt für alle $(u, v) \in E$:

$$\delta(s, v) \leq \delta(s, u) + 1$$

Lemma 23.2 Sei $G = (V, E)$ ein gerichteter oder ungerichteter Graph. Man nehme an, daß BFS auf G für ein $s \in V$ angewendet wird. Nach der Terminierung gilt

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

Beweis

Induktion über die Anzahl i die ein Knoten $v \in V$ in Q eingefügt wird.

Die Induktionshypothese ist $d[v] \geq \delta(s, v)$ für alle $v \in V$.

I.A.: Situation nach Z8 von Q, direkt nachdem s in Q eingefügt wurde. Hier gilt:

$$d[s] = 0 = \delta(s, s)$$

und

$$\forall v \in V \setminus \{s\} \;\; d[v] = \infty \geq \delta(s, v)$$

I.S.: Man betrachte einen weissen Knoten v, der von u aus entdeckt wird.

I.H. $\Longrightarrow d[u] \geq \delta(s,u)$. aus Z14 und Lemma 23.1 folgt:

$$\begin{aligned} d[v] &= d[u]+1 \\ &\geq \delta(s,u)+1 \\ &\geq \delta(s,v) \end{aligned}$$

v wird dann in Q eingefügt (Z16), aber nicht noch einmal, da v jetzt grau ist (Z13). Daher ändert sich $d[v]$ nicht mehr udn die I.H. bleibt gültig.

Um zu zeigen, daß $d[v] = \delta(s, v)$ gilt, müssen wir uns genauer anschauen, wie BFS mit Q umgeht. Das nächste Lemma zeigt, daß zu jedem Zeitpuntk höchstens 2 verschiedene Werte für d in der Queue sind.

Lemma 23.3 Q enthalte während der Ausführung von BFS auf $G = (V, E)$ die Knoten $\langle v_1, \ldots, v_r \rangle$. Dann gilt:

$$\begin{aligned} d[v_r] &\leq d[v_1] + 1 \\ d[v_i] &\leq d[v_{i+1}] \end{aligned}$$

für alle $1 \leq i \leq r - 1$

Beweis durch Induktion über die Anzahl der Queueoperationen.

I.A.: Für die initiale Queue mit $Q = \langle s \rangle$: $\surd$

I.S.: Wir müssen zeigen, daß die I.H. nach ENQUEUE und DEQUEUE gilt. Falls Q leer: $\surd$

DEQUEUE(v_1): $\mathbf{head(Q)} = v_2$ und

$$d[v_r] \leq d[v_1] + 1 \leq d[v_2] + 1$$

$\surd$

ENQUEUE(v_{r+1}): dann wird die Adjazenzliste von v_1's durchlaufen. Also (Z16-18):

$$d[v_{r+1}] = d[v_1] + 1$$

I.H. $\Longrightarrow$:

$$d[v_r] \leq d[v_1] + 1 = d[v_{r+1}]$$

$\surd$

Korrektheit von BFS

Satz Sei $G = (V, E)$ ein gerichteter oder ungerichteter Graph. Dann besucht BFS bei Anwendung auf G, s, $s \in V$ jeden von s aus erreichbaren Knoten v und nach der Terminierung gilt $d[v] = \delta(s, v)$ für alle $v \in V$.

Weiter gilt für jeden Knoten $v \neq s$, der von s aus erreichbar ist, daß ein kürzester Pfad von s nach v von s zu $\pi[v]$ gefolgt von der Kante $(\pi[v], v)$.

Beweis

Fall 1: v nicht von s aus erreichbar.

Lemma 23.2: $d[v] \geq \delta(s, v) = \infty$.

Also kann $d[v]$ nicht in Z14 auf einen endlichen Wert gesetzt werden.

Falls also v nicht von s aus erreichbar ist, so wird v nie entdeckt.

Fall 2: v von s aus erreichbar.

Sei V_k die Menge der Knoten mit $\delta(s, v) = k$.

Wir zeigen durch Induktion, daß für jeden Knoten $v \in V_k$ während der Ausführung von BFS folgende Operationen nur einmal ausgeführt werden:

1. v wird grau gefärbt
2. $d[v]$ wird auf k gesetzt
3. $v \neq s \Longrightarrow \pi[v] := u$ für ein $u \in V_{k-1}$
4. v wird in Q eingefügt

Das dies höchstens einmal geschieht ist bereits klar.

I.A.: Sei $k = 0$. Es gilt $V_0 = \{s\}$. Während der initialisierung passieren genau die oben erwähnten Operationen.

I.H.: Beobachtungen:

- Q ist leer nur bei Terminierung
- nachdem u in Q eingefügt wurde, bleiben $d[u]$ und $\pi[u]$ unverändert

Lemma 23.3 impliziert daher, daß, falls die Knoten in der Reihenfolge $\langle v_1, \ldots, v_r \rangle$ in Q eingefügt werden, gilt:

$$d[v_i] \leq d[v_{i+1}]$$

f.a. $1 \leq i \leq r - 1$.

Für einen Knoten $v \in V_k$, $k \geq 1$ impliziert dies mit $d[v] \geq k$ (Lemma 23.2), daß alle Knoten aus V_{k-1} vor v in Q eingefügt werden.

$\delta(s, v) = k \Longrightarrow$ existiert ein Pfad mit k Kanten von s nach v. Also $\exists u \in V_{k-1} \quad (u, v) \in E$ Sei u der erste solche Knoten, der durch BFS grau gefärbt wird. Dann muß u irgendwann einmal das erste Element in Q sein.

Dann wird u's Adjazenzliste abgelaufen und v entdeckt. Dann passiert folgendes:

1. u wird grau gefärbt (Z13)
2. $d[v] := d[u] + 1 = k$ (Z14)
3. $\pi[v] := u$ (Z15)
4. ENQUEUE(Q, u) (Z16)

Da $v \in V_k$ beliebig, folgt die I.H.

Zur Vervollständigung des Beweises beobachte man, daß

$$v \in V_k \Longrightarrow \pi[v] \in V_{k-1}$$

Also erhält man einen kürzesten Pfad von s zu v, indem man einen kürzesten Pfad von s zu $\pi[v]$ nimmt und dann $(\pi[v], v)$ anhängt.

Breitensuchbäume

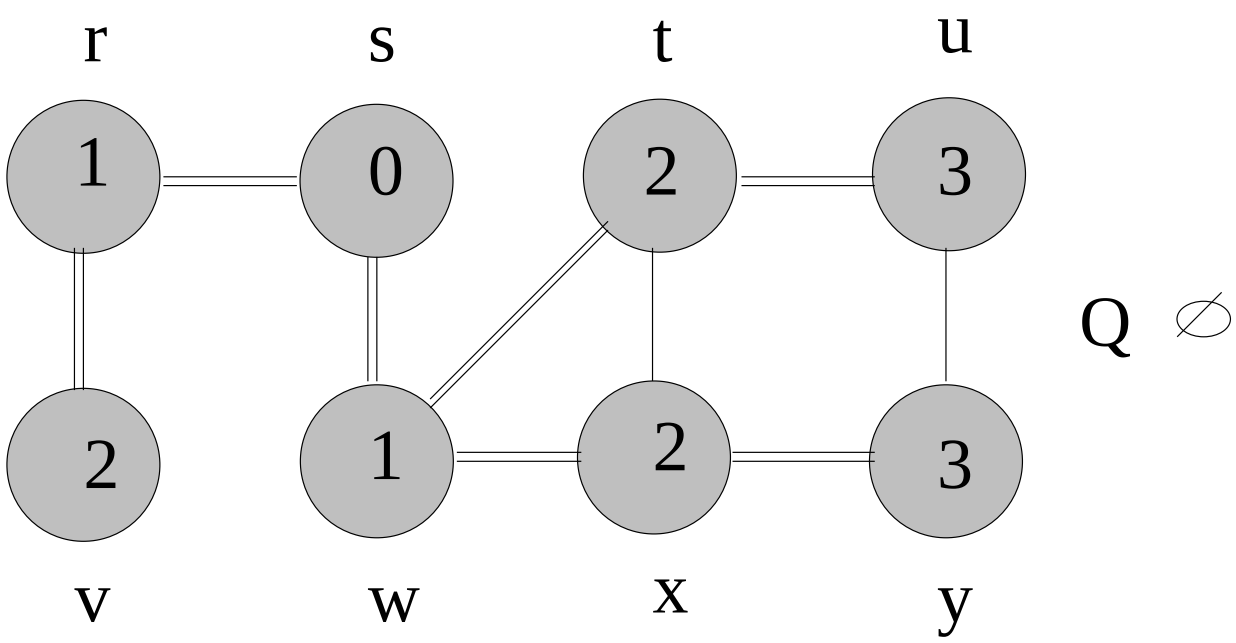
Wir definieren

Für einen Graphen $G = (V, E)$ mit Wurzel s definieren wir den Vorgängerteilgraphen von G als $G_\pi = (V_\pi, E_\pi)$ mit

$$\begin{aligned} V_\pi &= \{v \in V | \pi[v] \neq \textbf{ NIL} \cup \{s\}\} \\ E_\pi &= \{(\pi[v], v) \in E | v \in V_\pi \setminus \{s\}\} \end{aligned}$$

G_π ist ein **Breitensuchbaum**, falls

1. V_π besteht aus allen von s aus erreichbaren Knoten
2. f.a. $v \in V_\pi$ existiert ein eindeutiger Pfad von s nach v in G_π



(i)

Breitensuchbäume

Lemma 23.5 Es werde BFS auf einen gerichteten oder ungerichteten Graphen $G = (V, E)$ angewendet. Nach Terminierung hat BFS die π-Attribute so belegt, daß G_π ein Breitensuchebaum ist.

Beweis Z15 setzt $\pi[v] = u$, falls $(u, v) \in E$ und $\delta(s, v) < \infty$.

Also besteht V_π aus den von s aus erreichbaren Knoten.

Da G_π ein Baum ist, enthält er einen eindeutigen Pfad von s nach v. Obiger Satz ergibt die Minimalität dieses Pfades.

```
PRINT-PATH(G, s, v)
if v = s
   then print s
   else if π[v] = NIL
      then print "no path form" s "to" v "exists"
      else PRINT-PATH(G, s, π[v])
         print v
```

Laufzeit: $O(V)$

Tiefensuche

Sei $G = (V, E)$ ein Graph und $v \in V$ ein ausgewählter Startknoten.

Wir wollen systematisch alle von v aus erreichbaren Knoten besuchen.

Maxime

- zuerst in die Tiefe suchen
  zuerst den Nachfolger besuchen, dann den Nachfolger vom Nachfolger usw.
- dabei sind Rücksprünge notwendig, falls kein noch (nicht besuchter) Nachfolger mehr existiert

Genauer

- Stack statt Queue

Vorgängerteilgraph

Wegen der Rücksprünge muß der Vorgängerteilgraph leicht anders definiert werden:

Def Der Vorgängerteilgraph für die Tiefensuche wird definiert als ein Graph $G_\pi = (V, E_\pi)$ mit

$$E_\pi = \{(\pi[v], v) | v \in V, \pi[v] \neq NIL\}$$

Der Vorgängerteilgraph der Tiefensuche ist ein Tiefensuchewald, der aus mehreren Tiefensuchebäumen zusammengesetzt ist.

Die Kanten in E_π werden Baumkanten genannt.

Tiefensuche

Färbung

- initial alle Knoten weiss
- bei Entdeckung grau
- nach Abarbeitung der Adjazenzliste schwarz

Zeitstempel

- $d[v]$ Zeitstempel, wann v entdeckt wurde
- $f[v]$ Zeitstempel, wann Abarbeitung von vs Adjazenzliste beendet

Zeitstempel nützlich für mehrere Graphalgorithmen und für nachdenken über Tiefensuche.

Zeitstempel zwischen 1 und $2|V|$. (Ein Entdecken und ein Beenden.)

```
DFS(G)
for each vertex u ∈ V[G]
   do color[u] ← WHITE
      π[u] ← NIL
time ← 0
for each vertex u ∈ V[G]
   do if color[u] = WHITE
         then DFS-VISIT(u)
```

```
DFS-VISIT(u)
color[u] ← GRAY
    ▷ White vertex u has just been discovered.
d[u] ← time ← time + 1
for each v ∈ Adj[u]
    ▷ Explore edge (u, v).
    do if color[v] = WHITE
        then π[v] ← u
            DFS-VISIT(v)
color[u] ← BLACK
    ▷ Blacken u; it is finished.
f[u] ← time ← time + 1
```

DFS

Z1-3 alle Knoten weisseln, alle π auf NIL setzen

Z4 globalen Zähler rücksetzen

Z5-7 besuche alle weißen Knoten mit DFS-VISIT
u wird Wurzel eines neuen Baumes im Tiefensuche-wald.

DFS-VISIT

Z1 u wird grau

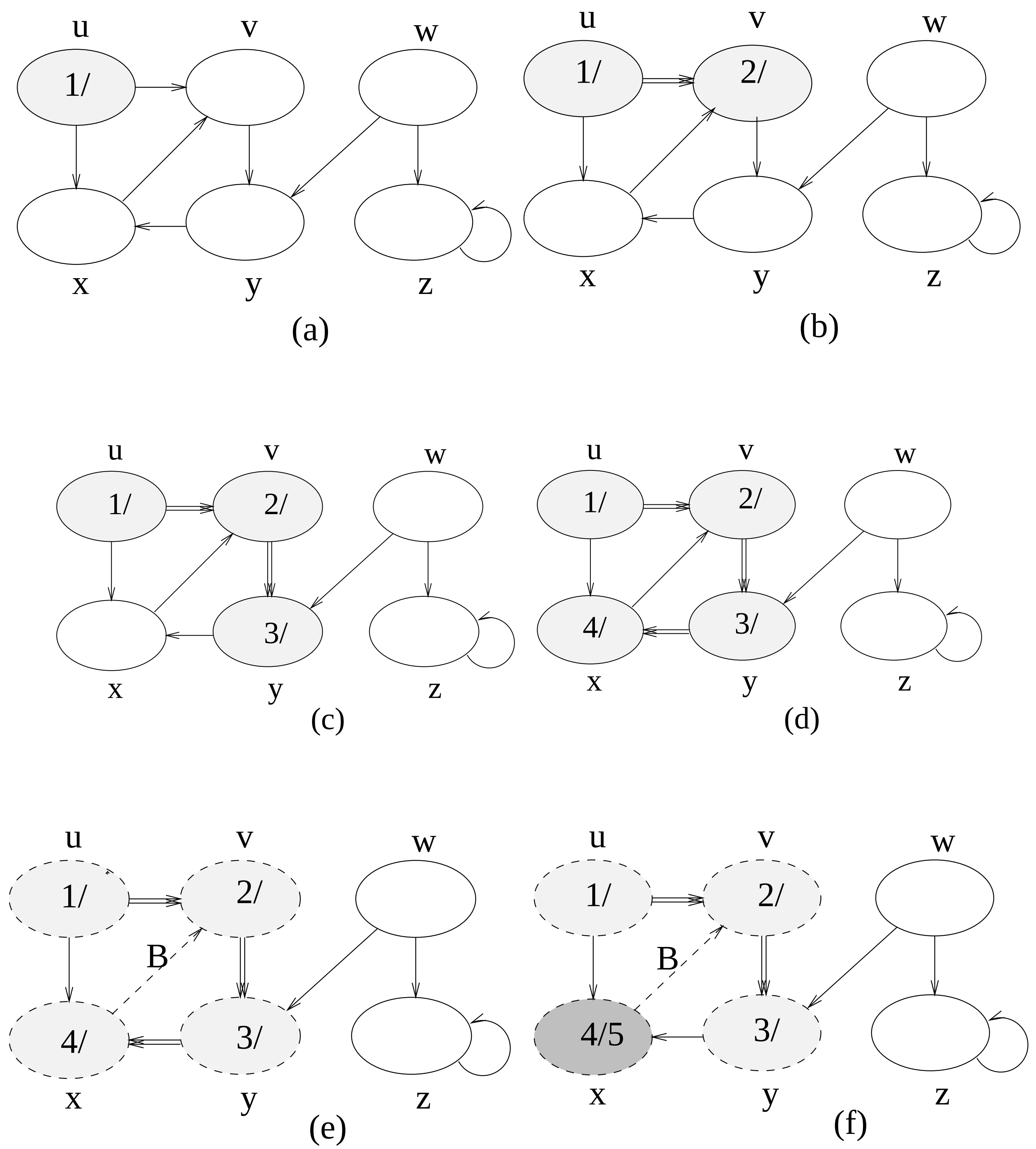
Z2 setzen der Entdeckungszeit $d[u]$, Zeit erhöhen

Z3-6 us Adjazenzliste ablaufen und weißen v rekursiv besuchen
in Z3 sagen wir eine Kante (u, v) wird abgearbeitet

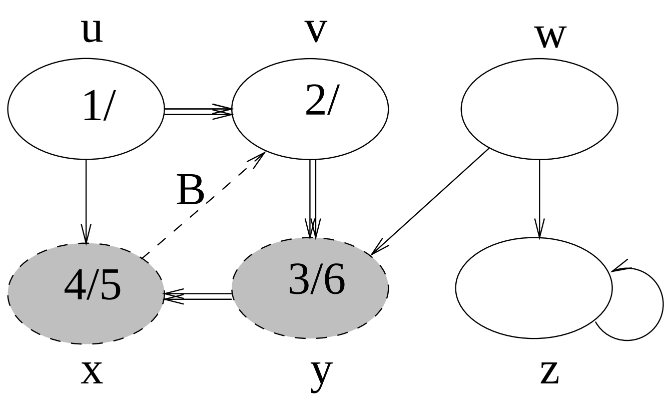
Z7-8 u schwarz färben und Endzeit $f[u]$ setzen

Laufzeit DFS

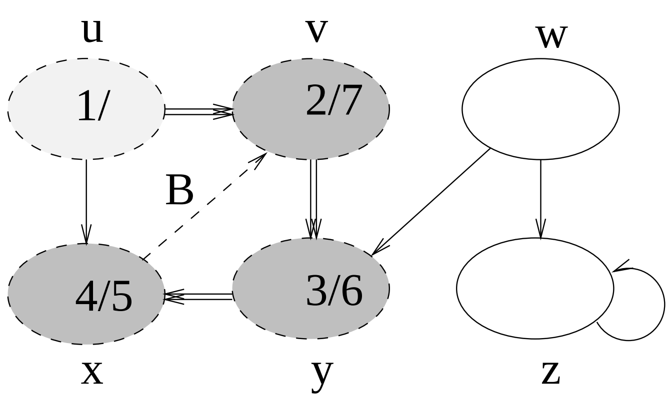
- Schleifen Z1-3 und Z5-7 von DFS: $\Theta(V)$ ohne DFS-VISIT
- DFS-VISIT: genau einmal pro Knoten ausgeführt.
- Da
$$\sum_{v \in V} |Adj[v]| = \Theta(E)$$
Gesamtkosten Z2-5 von DFS-VISIT: $\Theta(E)$.
- Gesamtlaufzeit DFS: $\Theta(V + E)$



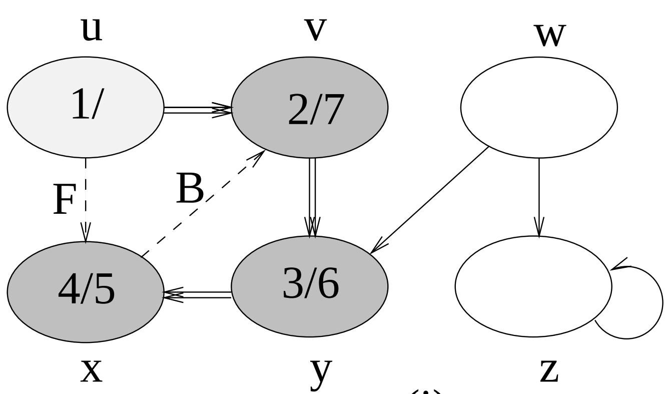
u
v
w
1/
x
y
z
(a)
u
v
w
1/
2/
x
y
z
(b)
u
v
w
1/
2/
3/
x
y
z
(c)
u
v
w
1/
2/
4/
3/
x
y
z
(d)
u
v
w
1/
2/
B
4/
3/
x
y
z
(e)
u
v
w
1/
2/
B
4/5
3/
x
y
z
(f)


u
v
w
1/
2/
B
4/5
3/6
x
y
z

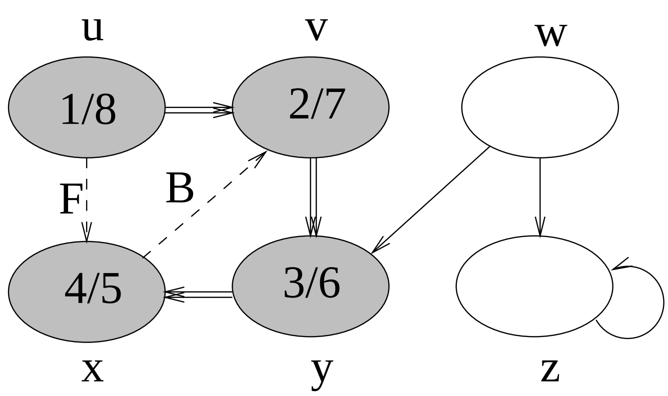
(g)


u
v
w
1/
2/7
B
4/5
3/6
x
y
z

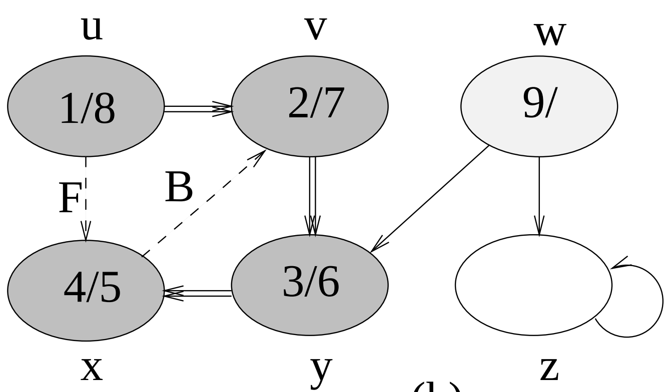
(h)


u
v
w
1/
2/7
F
B
4/5
3/6
x
y
z

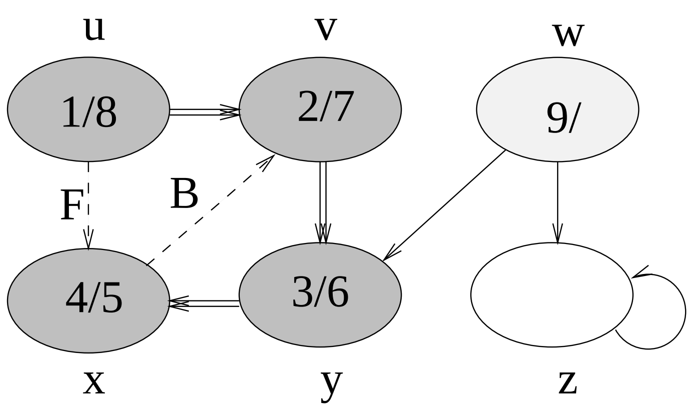
(i)


u
v
w
1/8
2/7
F
B
4/5
3/6
x
y
z

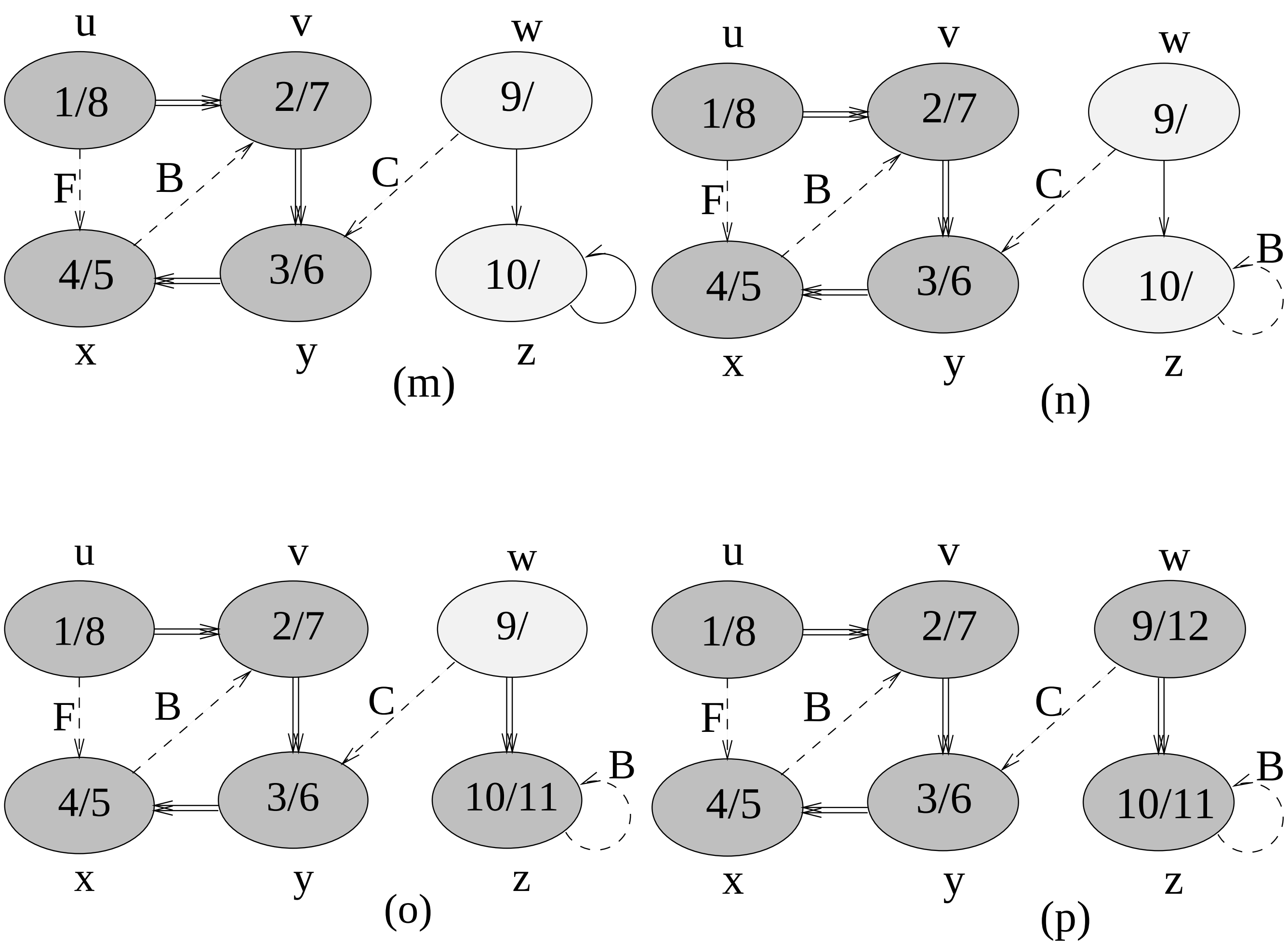
(j)


u
v
w
1/8
2/7
9/
F
B
4/5
3/6
x
y
z

(k)

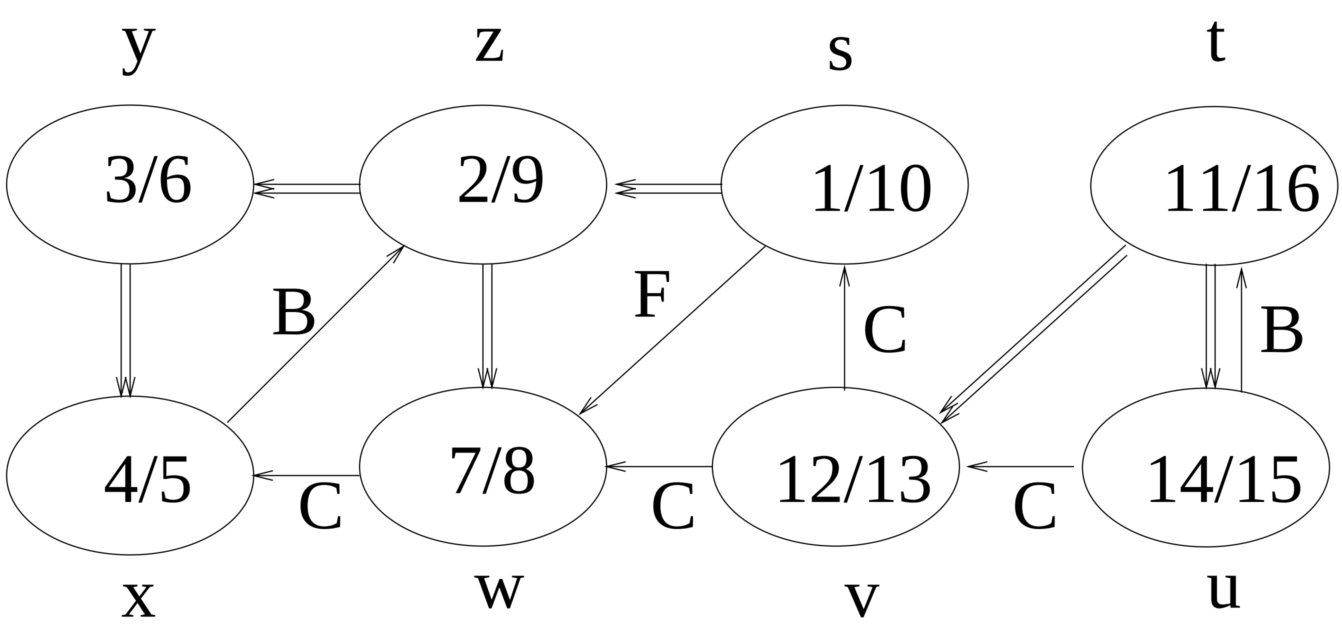

u
v
w
1/8
2/7
9/
F
B
4/5
3/6
x
y
z

(l)

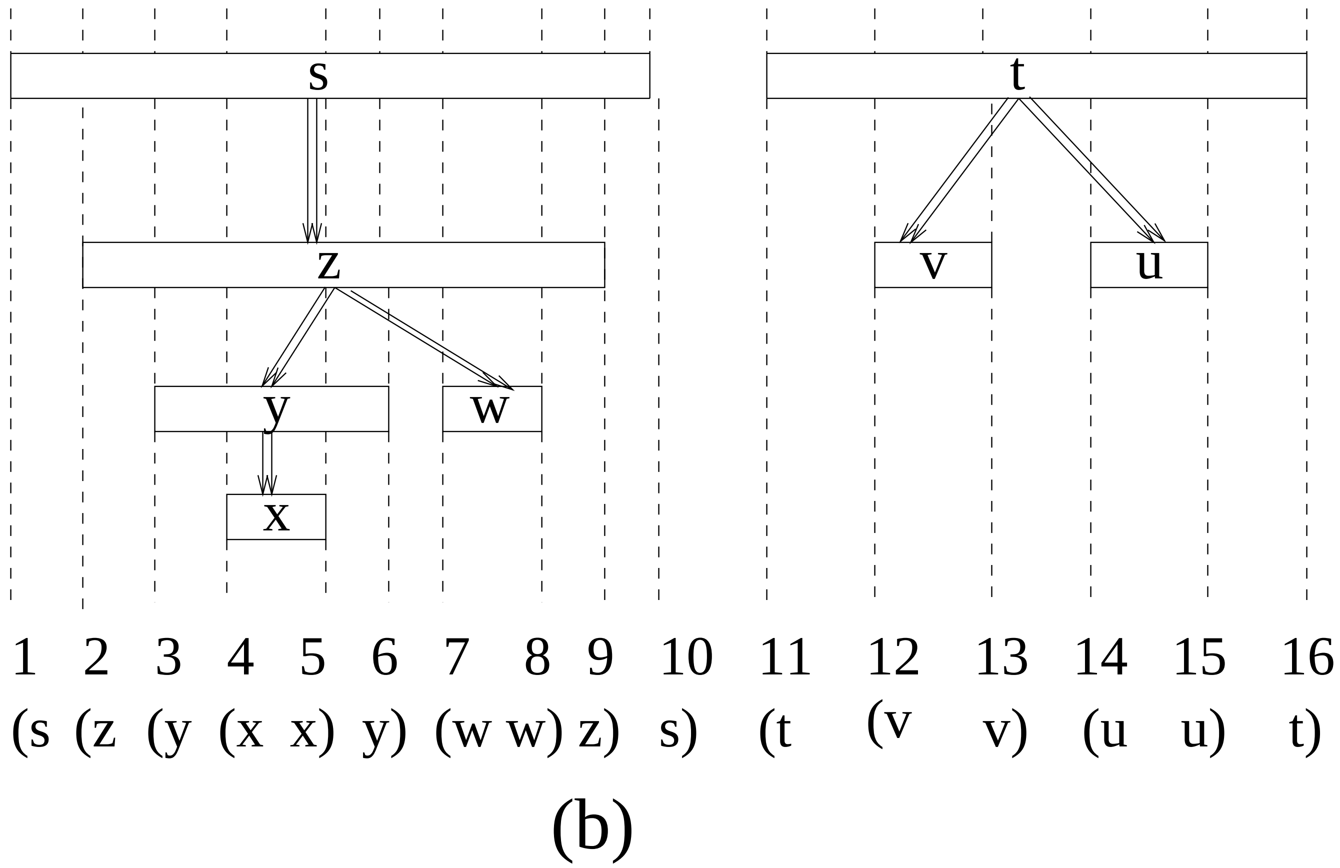

u
v
w
1/8
2/7
9/
F
B
C
4/5
3/6
10/
x
y
z
(m)
u
v
w
1/8
2/7
9/
F
B
C
B
4/5
3/6
10/
x
y
z
(n)
u
v
w
1/8
2/7
9/
F
B
C
B
4/5
3/6
10/11
x
y
z
(o)
u
v
w
1/8
2/7
9/12
F
B
C
B
4/5
3/6
10/11
x
y
z
(p)

Eigenschaften

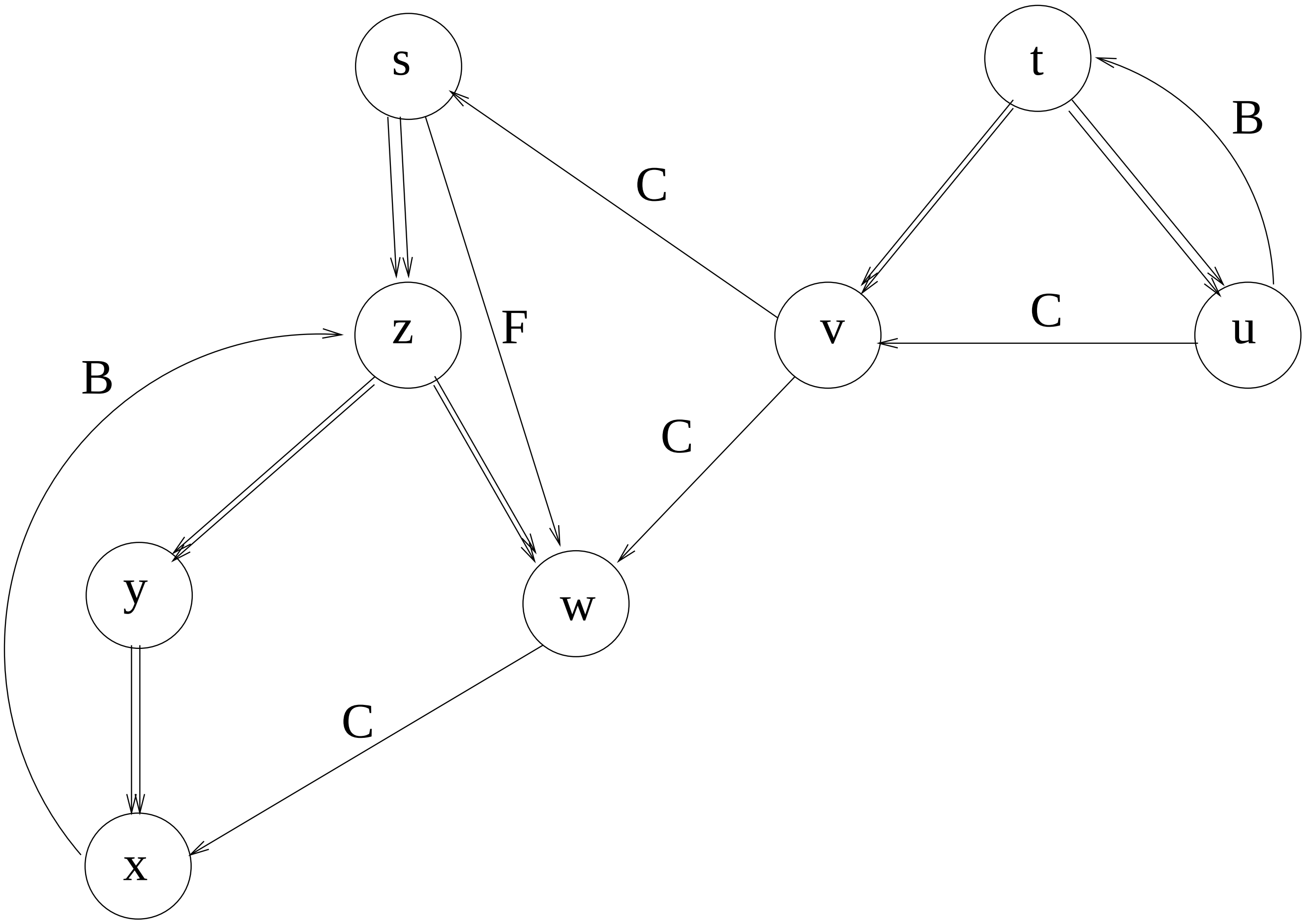
- G_π ist ein Wald
- Entdeckungs und Endzeiten haben Klammerstruktur
  $d[u] \sim$ "(u" und $f[u] \sim$ "u)" dann entsteht ein wohlgeformter Klammerausdruck
- Klassifikation der Kanten im Graphen G


y
z
s
t
3/6
2/9
1/10
11/16
B
F
C
B
4/5
7/8
12/13
14/15
C
C
C
x
w
v
u

(a)


s
t
z
v
u
y
w
x
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
(s (z (y (x x) y) (w w) z) s) (t (v v) (u u) t)

(b)



(c)

Klammertheorem

Die zweite der obigen Bedingungen kann auch anders ausgedrückt werden:

Satz 23.6 Sei $G = (V, E)$ ein gerichteter oder ungerichteter Graph. Für jede Tiefensuche von G und für je zwei Knoten u und v gilt genau eine der drei folgenden Bedingungen:

1. die Intervalle $[d[u], f[u]]$ und $[d[v], f[v]]$ sind disjunkt
2. $[d[v], f[v]]$ enthält $[d[u], f[u]]$ und u ist ein Nachfolger von v im Tiefensuchebaum
3. $[d[u], f[u]]$ enthält $[d[v], f[v]]$ und v ist ein Nachfolger von u im Tiefensuchebaum

Beweis

Fall 1: $d[u] < d[v]$

Fall 1.1: $d[v] < f[u]$

$\Longrightarrow$ v entdeckt als u grau

$\Longrightarrow$ v Nachfolger von u.

Weiter: da u nach v entdeckt, wird v vor u abgearbeitet.

Daher $[d[v], f[v]]$ in $[d[u], f[u]]$ enthalten

Fall 1.2: $f[u] < d[v]$

$\Longrightarrow$ $[d[u], f[u]]$ und $[d[v], f[v]]$ disjunkt

Fall 2: analog

Korollar 23.7(Schachtelung von Nachfolgerintervallen)
Ein Knoten v ist ein Nachfolger im Tiefensuchenwald eines Knotens $u \iff d[u] < d[v] < f[v] < f[u]$

Theorem des weißen Pfades

Eine andere Charakterisierung wann ein Knoten Nachfolger eines anderen im Tiefensuchenwald ist:

Satz 23.8 Im Tiefensuchenwald eines gerichteten oder ungerichteten Graphen $G = (V, E)$ ist ein Knoten v ein Nachfolger eines Knotens u genau dann, wenn zum Zeitpunkt $d[u]$ der Knoten v auf einem Pfad, der nur aus weißen Knoten besteht, von u aus erreicht werden kann.

Beweis $\Longrightarrow$ Sei v ein Nachfolger von u. Sei w ein Knoten auf einem Pfad von u nach v (im Tiefensuchenwald). Nach Korollar gilt $d[u] < d[w]$, also ist w weiß.

$\Longleftarrow$ Sei v von u aus auf einem Pfad mit nur weißen Knoten erreichbar. Annahme: v wird kein Nachfolger von u.

obdA: Alle anderen Knoten vor v werden Nachfolger von u.
(nimm ersten, der kein Nachfolger wird)

Sei w Vorgänger von v und w Nachfolger von u.
(u und w könnten der gleiche Knoten sein)

Korollar: $f[w] \leq f[u]$

v muß nach u entdeckt und vor w abgearbeitet sein.

Also: $d[u] < d[v] < f[w] \leq f[u]$

23.6 impliziert $[d[v], f[v]]$ in $d[[u], f[u]]$ enthalten.

Also ist v Nachfolger von u.

Kantenklassifikation

Klassifikation der Kanten in Baumkanten, Rückkanten, Vorwärtskanten und Kreuzkanten.

Nützlich: nächster Abschnitt: azyklisch $\Longleftrightarrow$ keine Rückkanten

Kantenklassifikation

1. Baumkanten sind alle Kanten in G_π Kante (u, v) ist eine Baumkante, falls v zuerst entdeckt wurde beim Ablaufen der Kante (u, v)
2. Rückkanten sind alle Kanten (u, v), die einen Knoten u mit einem Vorgänger v verbinden
   In einem gerichteten Graphen werden Schleifen als Rückkanten klassifiziert
3. Vorwärtskanten sind alle Kanten (u, v), die einen Knoten v mit einem Nachfolger von v verbinden
4. Kreuzkanten sind alle anderen Kanten
   Innerhalb eines Tiefensuchenbaumes, solange keine Vorgänger Nachfolgerbeziehung besteht oder zwischen zwei verschiedenen Tiefensuchenbäumen

Modifikation von DFS um Kanten (u, v) anhand der Farbe von v zu klassifizieren:

1. v ist weiß: Baumkante
2. v ist grau: Rückkante
3. v ist schwarz: entweder Vorwärts- oder Kreuzkante

Anm zu

(2) Graue Knoten sind immer eine lineare Kette von Kanten, die zum Aufrufstapel von DFS-VISIT korrespondieren.

(3) Man kann zeigen, daß (u, v) eine Vorwärtskante ist, falls $d[u] < d[v]$.

Im ungerichteten Graph kann es Mehrdeutigkeiten bezüglich der Klassifikation geben, da (u, v) und (v, u) die gleiche Kante darstellen. In so einem Fall gilt die **erste** Klassifikation.

Vorwärts und Kreuzkanten kommen in einem ungerichteten Graphen nicht vor:

Satz 23.9 Die Kanten in einem ungerichteten Graphen $G = (V, E)$ werden durch die Tiefensuche entweder als Baumkanten oder als Rückkanten klassifiziert.

Beweis Sei (u, v) eine beliebige Kante von G. obdA: $d[u] < d[v]$. $\Longrightarrow d[u] < d[v] < f[v] < f[u]$

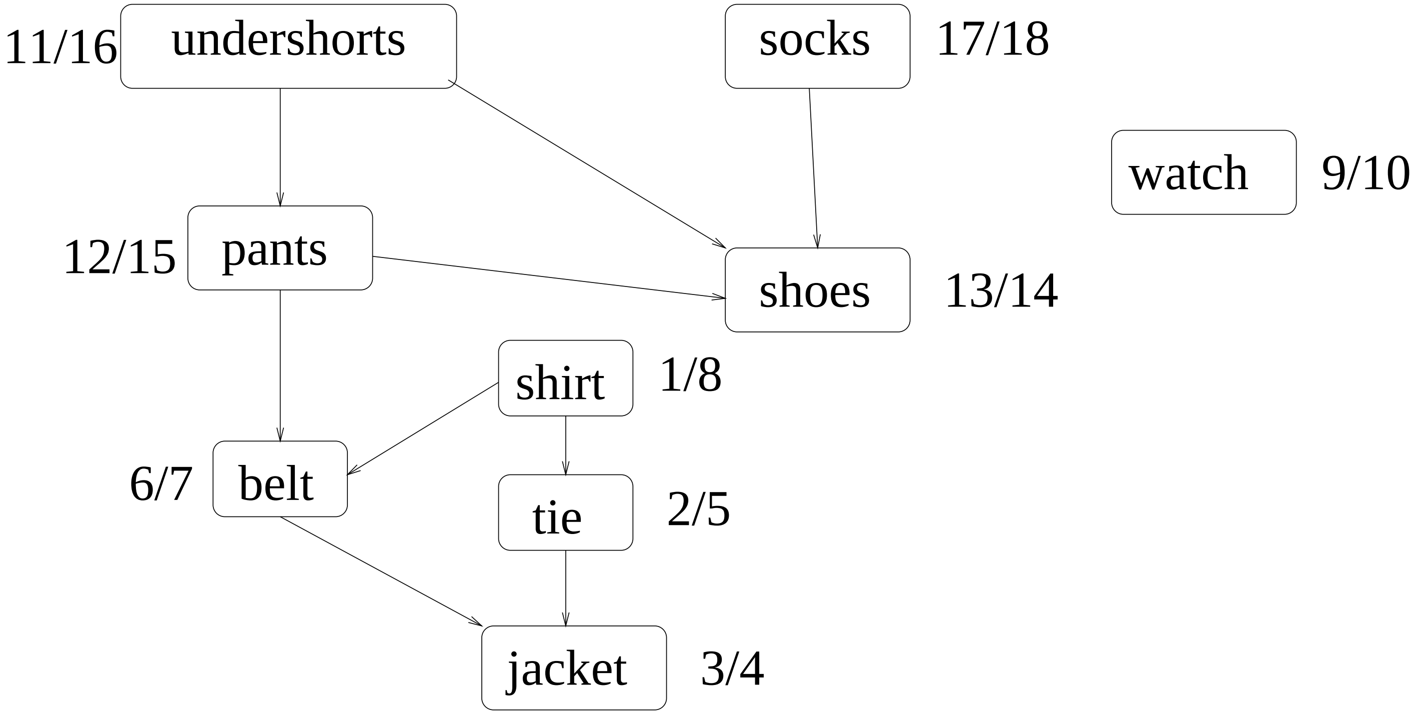
Falls (u, v) zuerst in Richtung $u \to v$ durchlaufen, dann wird (u, v) eine Baumkante.

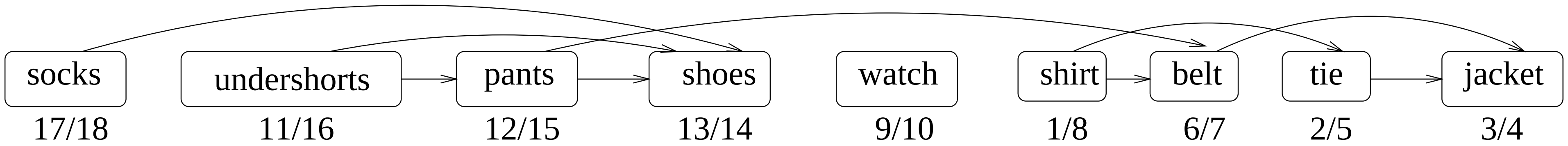
Falls (u, v) zuerst in Richtung $v \to u$ durchlaufen, so wird (u, v) eine Rückkante, da u noch grau ist.

Topologische Ordnung

Def Eine **topologische Ordnung** $<$ eines DAG $G = (V, E)$ ist eine lineare Ordnung aller Knoten in G mit $(u, v) \in E \Longrightarrow u < v$

Bsp Netzpläne


undershorts
11/16
socks
17/18
watch
9/10
pants
12/15
shoes
13/14
shirt
1/8
belt
6/7
tie
2/5
jacket
3/4


socks
undershorts
pants
shoes
watch
shirt
belt
tie
jacket
17/18
11/16
12/15
13/14
9/10
1/8
6/7
2/5
3/4

TOPOLOGICAL-SORT(G)
1 call DFS(G) to compute finishing times $f[v]$
for each vertex v
2 as each vertex is finished,
insert it onto the front of a linked list
3 **return** the linked list of vertices

Anm Die topologisch Sortierten Knoten erscheinen in umgekehrter Ordnung der Endzeit.

Aufwand $\Theta(V + E)$

Korrektheit

Lemma 23.10 Ein gerichteter Graph G ist azyklisch genau dann, wenn DFS keine Rückkanten ergibt.

Beweis $\Longrightarrow$ Annahme: es existiert eine Rückkante (u, v).

Dann ist v ein Vorgänger von u im Tiefensuchewald.

Also gibt es einen Pfad von v nach u in G und mit der Rückkante (u, v) ergibt sich ein Zyklus.

$\Longleftarrow$ Annahme: G enthält einen Zyklus c.

Wir Zeigen das BFS eine Rückkante erzeugt.

Sei v der erste entdeckte Knoten von c und (u, v) die Vorgängerkante in c.

Zum Zeitpunkt $d[v]$ gibt es einen Pfad mit weißen Knoten von v zu u. Also wird u ein Nachfolger von v.

Also ist (u, v) eine Rückkante.

Korrektheit

Satz 23.11 TOPOLOGICAL-SORT(G) erzeugt eine topologische Ordnung eines DAG $G = (V, E)$.

Beweis Es werde DFS benutzt um die Endzeiten zu bestimmen.

Es genügt zu zeigen:

Für je zwei verschiedene Knoten $u, v \in V$: $\exists (u, v) \in E \implies f[v] < f[u]$

Wenn DFS (u, v) abarbeitet kann v nicht grau sein, sonst wäre (u, v) eine Rückkante.

Falls v weiß ist, so wird v Vorgänger von u und damit $f[v] < f[u]$.

Falls v schwarz ist, gilt unmittelbar $f[v] < f[u]$.

Starke Zusammenhangskomponenten

Zerlegung von gerichteten Graphen in Zusammenhangskomponenten wird oft als erster Schritt in vielen Algorithmen benötigt, um ein Problem in Teilproblem zu zerlegen.

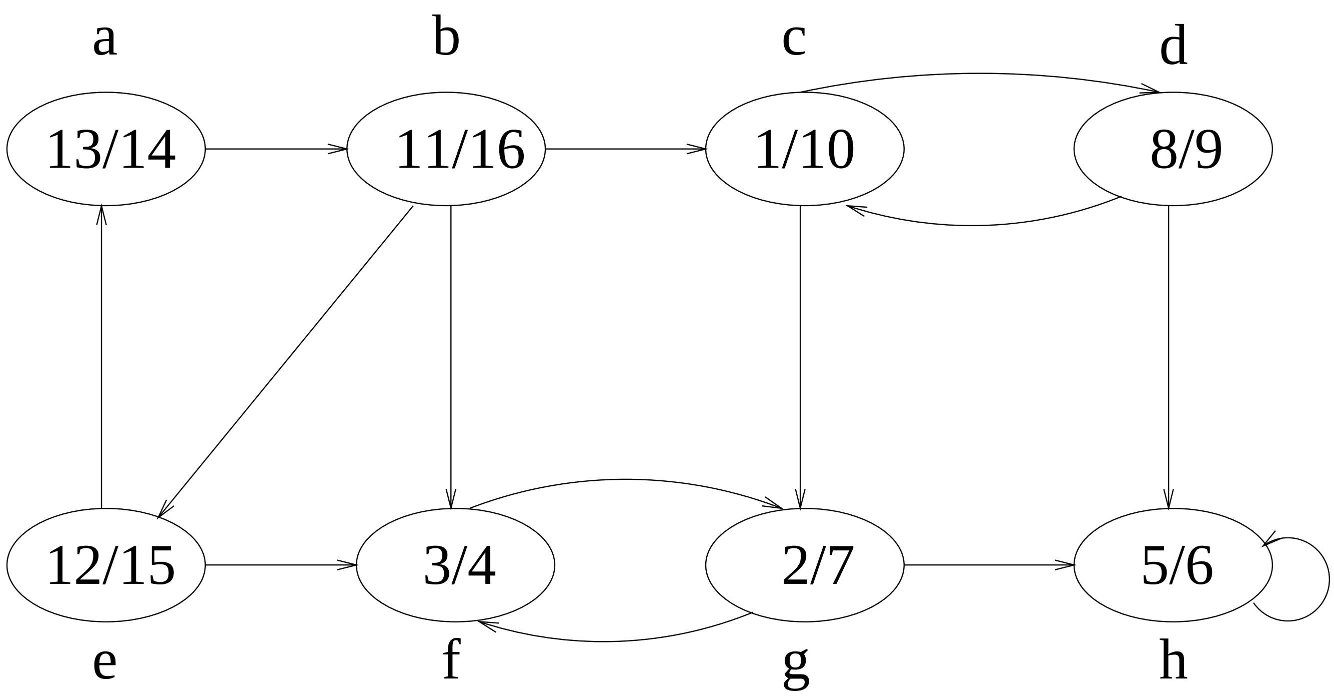
Wir werden DFS benutzen, um diese Zerlegung durchzuführen.

Wir benutzen dazu den transponierten Graphen $G^T = (V, E^T)$ mit

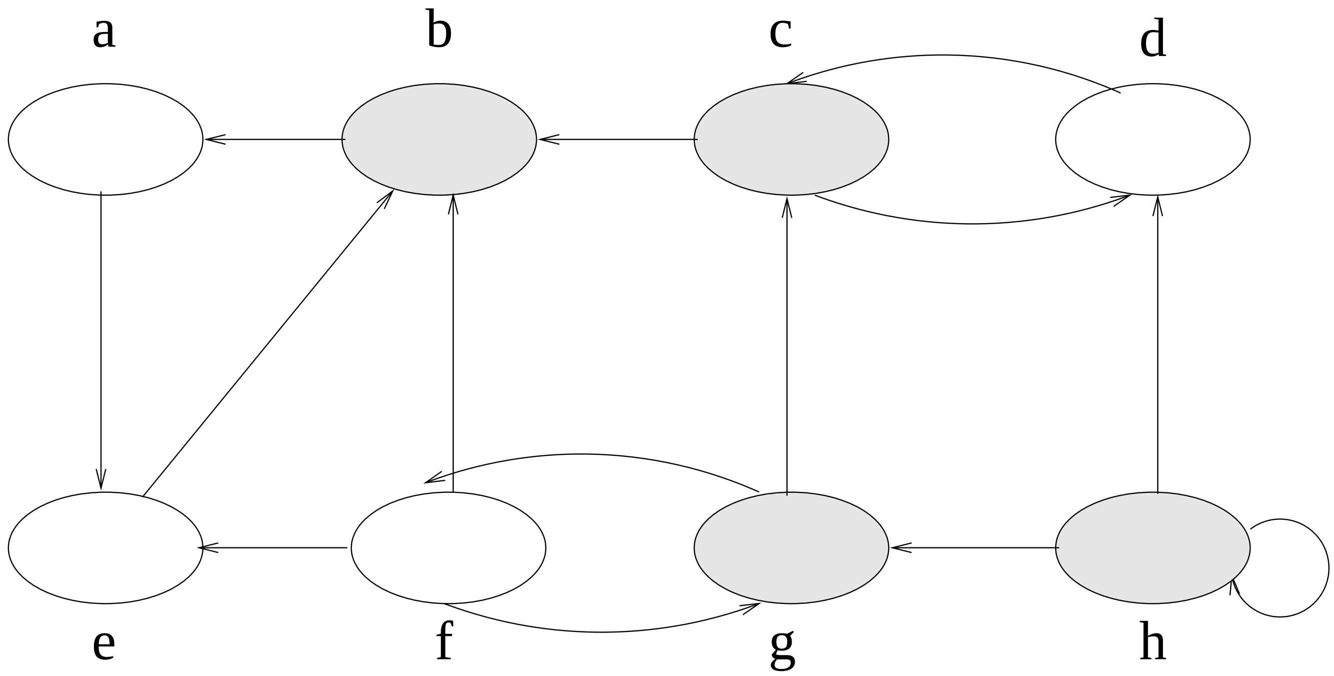
$$E^T = \{(u, v) | (v, u) \in E\}$$

bei Adjazenzlistendarstellung kann G^T in $O(V + E)$ erzeugt werden.

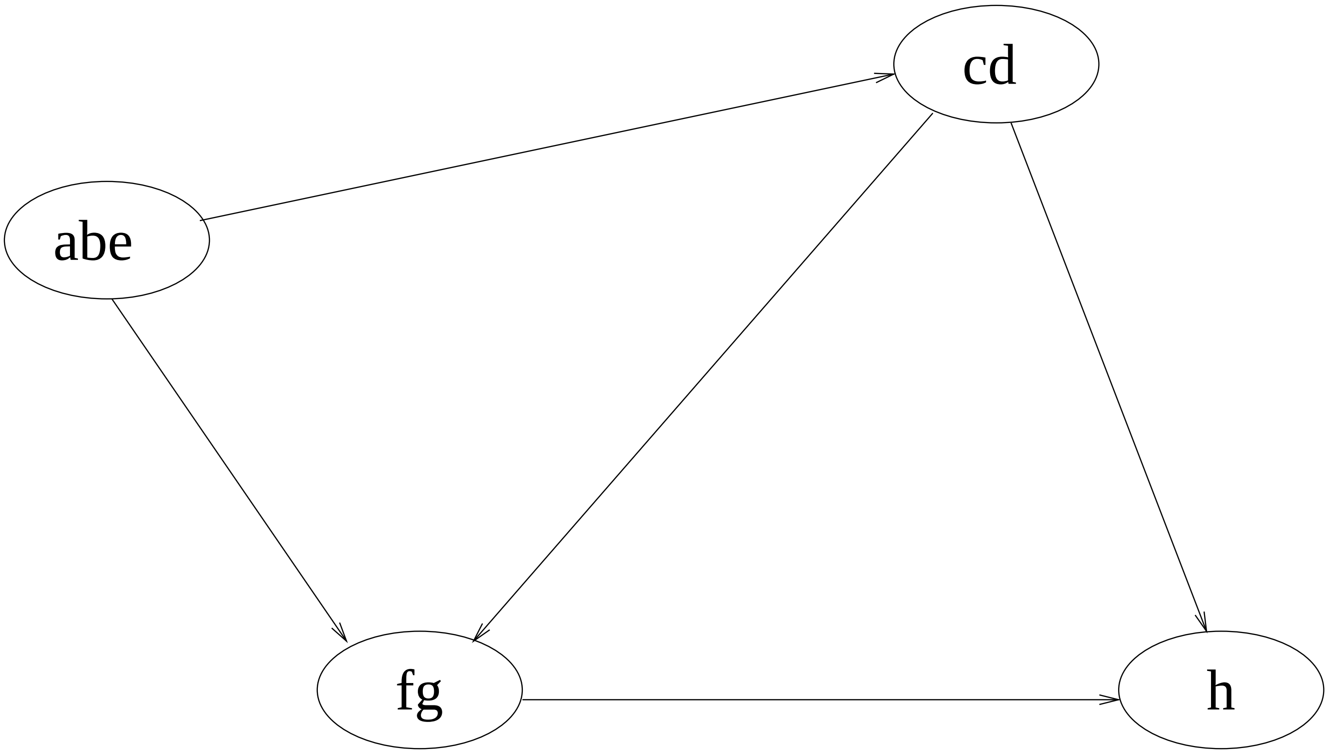
G und G^T haben die gleichen starken Zusammenhangskomponenten.



(a)



(b)



(c)

STRONGLY-CONNECTED-COMPONENTS(G)

1 call DFS(G) to compute finishing times $f[v]$ for each vertex v

2 compute G^T

3 call DFS(G^T), but in the main loop of DFS, consider in order of decreasing $f[u]$

4 output the vertices of each tree resulting from 3 as a separate strongly connected component

Korrektheit

Zuerst zwei nützliche Beobachtungen

Lemma 23.12 Falls zwei Knoten in der gleichen Zusammenhangskomponente sind, so verläßt kein Pfad zwischen diesen die Zusammenhangskomponente

Beweis Übung

Satz 23.13 Bei jeder Tiefensuche werden alle Knoten einer strengen Zusammenhangskomponente in den gleichen Tiefensuchebaum gehängt.

Beweis Sei v der zuerst besuchte Knoten einer Zusammenhangskomponente. Dann sind alle mit v verbundenen Knoten auf weissen Pfaden erreichbar und werden somit zu Nachfolgern zu v.

Vereinbarungen:

- $d[u]$ und $f[u]$ beziehen sich auf den ersten Aufruf von DFS auf G.
- $u \rightsquigarrow v$ bezieht sich auf einen Pfad in G
- $\Phi(u)$ sei der Urahn von u: derjenige Knoten w, erreichbar von u, der zuletzt vollständig abgearbeitet wurde.
  $\Phi(u) = W$ mit $u \rightsquigarrow w$ und $f[w]$ maximal
  $\Phi(u) = u$ möglich, da u von u aus erreichbar und damit $f[u] \leq f[\Phi(u)]$.

Es gilt

$$\Phi(\Phi(u)) = \Phi(u)$$

da für je zwei Knoten $u, v \in V$:

$$u \rightsquigarrow v \implies f[\Phi(v)] \leq f[\Phi(u)]$$

da $\{w|v \rightsquigarrow w\} \subseteq \{w|u \rightsquigarrow w\}$ und der Urahn die maximale Endzeit hat.

Da $u \rightsquigarrow \Phi(u)$ gilt $f[\Phi(\Phi(u))] \leq f[\Phi(u)]$.

Vorher: $f[u] \leq f[\Phi(u)]$, also $f[\Phi(u)] \leq f[\Phi(\Phi(u))]$

Also gilt $f[\Phi(\Phi(u))] = f[\Phi(u)]$. $\surd$.

Anm In jeder Zshgskomp. gibt es einen Urahn aller Knoten. (Repräsentant)

Satz 23.14 In einem gerichteten Graphen $G = (V, E)$ ist der Urahn $\Phi(u)$ jedes Knotens $v \in V$ bei jeder Tiefensuche von G ein Vorgänger von u.

Beweis Falls $\Phi(u) = u$ $\surd$

$\Phi(u) \neq u$. Betrachte die Farben der Knoten zum Zeitpunkt $d[u]$.

Falls $\Phi(u)$ schwarz, so gilt $f[\Phi(u)] < f[u]$ (Widerspruch.).

Falls $\Phi(u)$ grau, dann ist er Vorgänger von u.

Falls $\Phi(u)$ weiss: Zwei Unterfälle, gemäß der Farbe der Knoten auf dem Pfad von u zu $\Phi(u)$ (falls existent).

1. Falls jeder Knoten weiss, dann wird $\Phi(u)$ Vorgänger. Dann gilt aber $f[\Phi(u)] < f[u]$. (Widerspruch.)
2. Sei t der letzte nicht-weiße Knoten. Dann muß t grau sein, da niemals eine Kante von einem schwarzen zu einem weißen Knoten existiert.

   Weiter ist ts Nachfolger weiss.

   Dann gibt es aber einen Pfad von weissen Knoten von t zu $\Phi(u)$ und somit ist Φ Nachfolger von t.

   Also $f[t] > f[\Phi(u)]$. (Widerspruch Def Φ.).

Korollar 23.15 In jeder Tiefensuche eines gerichteten Graphen $G = (V, E)$ liegen für alle Knoten u u und $\Phi(u)$ in der gleichen Zusammenhangskomponente.

Beweis Es gilt $u \rightsquigarrow \Phi(u)$ nach Definition Φ.

Außerdem gilt $\Phi(u) \rightsquigarrow u$, da $\Phi(u)$ ein Vorgänger von u ist. $\surd$

Satz 23.16 In einem gerichteten Graphen $G = (V, E)$ gilt für je zwei Knoten $u, v \in V$, daß sie genau dann in der gleichen Zusammenhangskomponente liegen, wenn sie den gleichen Urahn in einer Tiefensuche von G haben.

Beweis

$\Longrightarrow$ Wenn u und v in der gleichen Zusammenhangskomponente liegen, gilt $u \rightsquigarrow v$ und $v \rightsquigarrow u$. Nach Def. gilt $\Phi(u) = \Phi(v)$.

$\Longleftarrow$ Es gelte $\Phi(u) = \Phi(v)$. Korollar 23.15 ergibt, daß u und $\Phi(u)$ in der gleichen Zusammenhangskomponente liegen. Analog für v.

Also liegen auch u und v in der gleichen Zusammenhangskomponente

Wir können jetzt dem Algorithmus besser verstehen:

- strenge Zusammenhangskomponenten sind Mengen von Knoten mit gleichem Urahn
- der Urahn ist der erste in einer Zshgskomponente entdeckte und zuletzt abgearbeitete Knoten
- insbesondere werden alle Knoten der Zshgskomponente vor dem Urahn vollständig abgearbeitet.
- der Knoten r mit $f[r]$ maximal ist Urahn.
- in Zshgskomponente von r sind diejenigen Knoten, von denen aus man r erreichen kann, aber keinen Knoten, der eine Endzeit größer $f[r]$ hat.
- da $f[r]$ maximal, sind in Zshgskomp. von r alle Knoten, von denen aus man r erreichen kann. D.h. alle Knoten, die man von r aus in G^T erreichen kann.

Satz 23.17 STRONGLY-CONNECTED-COMPONENTS(G) erzeugt in korrekter Weise die strengen Zusammenhangskomponenten von G.

Beweis Beweis durch Induktion über die Anzahl der Tiefensuchebäume von G^T.

I.H.: der erzeugte Tiefensuchenbaum ist eine starke Zusammenhangskomponente

I.A.: trivial

I.S.: Sei T ein Tiefensuchebaum mit Wurzel r, erzeugt durch DFS(G^T).

Sei $C(r)$ die Menge aller Knoten mit Urahn r:

$$C(r) = \{v \in V | \Phi(v) = r\}$$

Wir zeigen: $u \in T \iff u \in C(r)$ ($\Longrightarrow$ Behauptung.)

$\Longleftarrow$ Satz 23.13 impliziert:$\forall v \in C(r) \;\; v \in T$.

Mit $r \in C(r)$ und r Wurzel von T folgt diese Richtung

$\Longrightarrow$

wir zeigen:

$$\forall w \;\; f[\Phi(w)] > f[r] \vee f[\Phi(w)] < f[r]$$

impliziert $w \notin T$

Durch Induktion über die Anzahl der Bäume, kann ein Knoten w mit $f[\Phi(w)] > f[r]$ nicht in T sein, da zu dem Zeitpunkt, zu dem r besucht wird, w bereits in einem Baum mit Wurzel $\Phi(w)$ plaziert ist.

Jeder Knoten w mit $f[\Phi(w)] < f[r]$ kann nicht in T sein, da dies $w \rightsquigarrow r$ implizieren würde, also

$$f[\Phi(w)] \geq f[\Phi(r)] = f[r]$$

was ein Widerspruch zu $f[\Phi(w)] < f[r]$ ist.

Minimale Spannbäume

Motivation

- Verkabelung

$G = (V, E)$, V sind Orte, E sind mögliche Strecken mit assoziierten Kosten $w(u, v)$ für $(u, v) \in E$

Gesucht: azyklische Teilmenge $T \subseteq E$ mit

$$w(T) = \sum_{(u,v) \in T} w(u, v)$$

minimal und alle Knoten aus V durch T verbunden.

T heißt minimaler Spannbaum.

Sichere Kanten

Invariante:

> A ist immer eine Teilmenge eines minimalen Spannbaums.

Eine Kante (u, v) ist sicher, falls auch $A \cup (u, v)$ eine Teilmenge eines minimalen Spannbaums ist.

Generische Version

```
GENERIC-MST(G, w)
A ← ∅
while A does not form a spanning tree
   do find an edge (u, v) that is safe for A
      A ← A ∪ {(u, v)}
return A
```

Schnitte (Cuts)

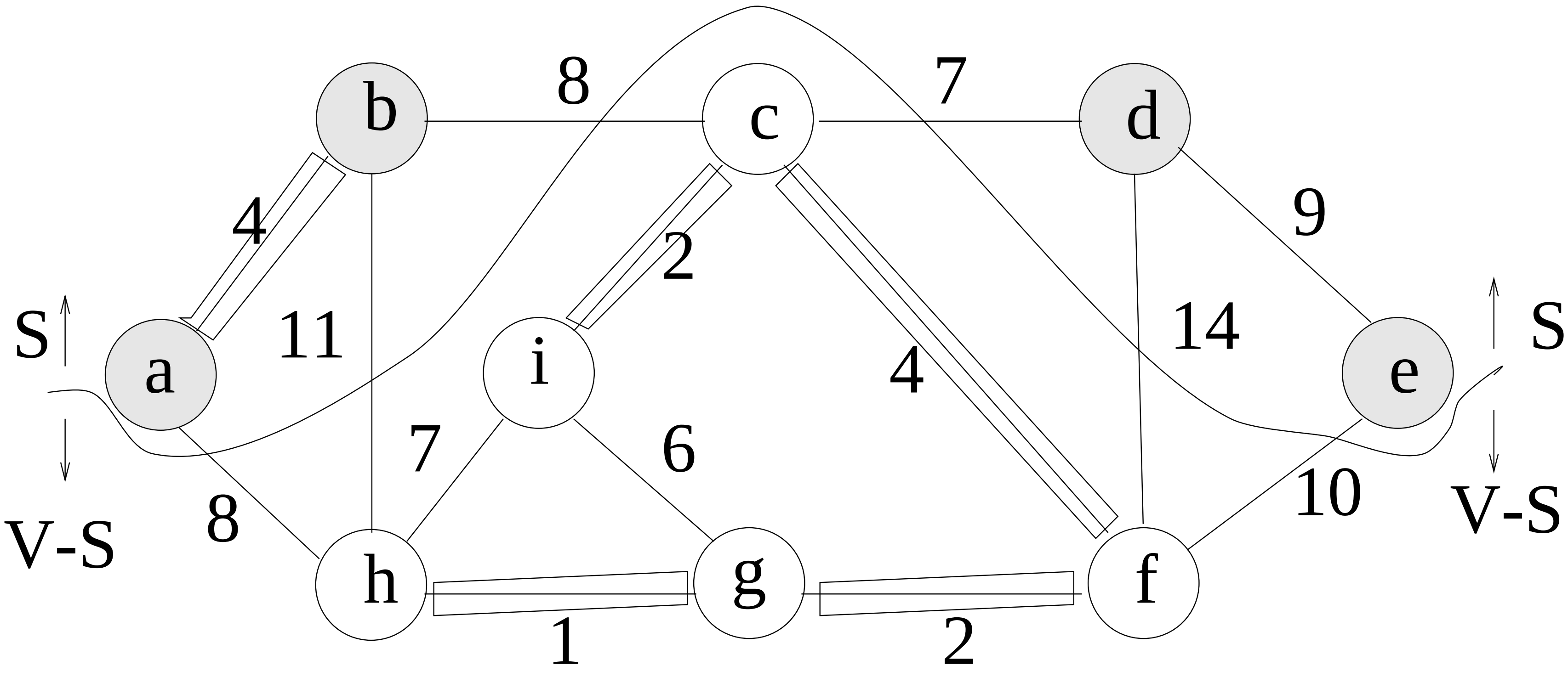
Def. Ein Schnitt $(S, V \setminus S)$ eines ungerichteten Graphen $G = (V, E)$ ist eine Partitionierung von V.

$(u, v) \in E$ kreuzt (crosses) einen Schnitt $(S, V \setminus S)$, falls $u \in S$ und $v \in V \setminus S$.

$A \subseteq E$ respektiert einen Schnitt, falls keine Kante aus A den Schnitt kreuzt.

Eine einen Schnitt kreuzende Kante heißt leicht, falls ihr Gewicht minimal unter allen kreuzenden Kanten ist.

Letzteres kann generalisiert werden für andere Eigenschaften als kreuzend.



(a)

Regel zur Bestimmung von sicheren Kanten

Satz 24.1 Sei $G = (V, E)$ ein zusammenhängender, ungerichteter Graph mit reeller Gewichtsfunktion w für E. Sei $A \subseteq E$ in einem minimalen Spannbaum für G enthalten. Sei $(S, V \setminus S)$ ein Schnitt von G, der A respektiert und (u, v) eine leichte kreuzende Kante. Dann ist (u, v) sicher für A.

Beweis

Sei T ein minimaler Spannbaum mit $A \subseteq T$, der (u, v) nicht enthält.

(u, v) erzeugt für einen Pfad von u nach v einen Zyklus.

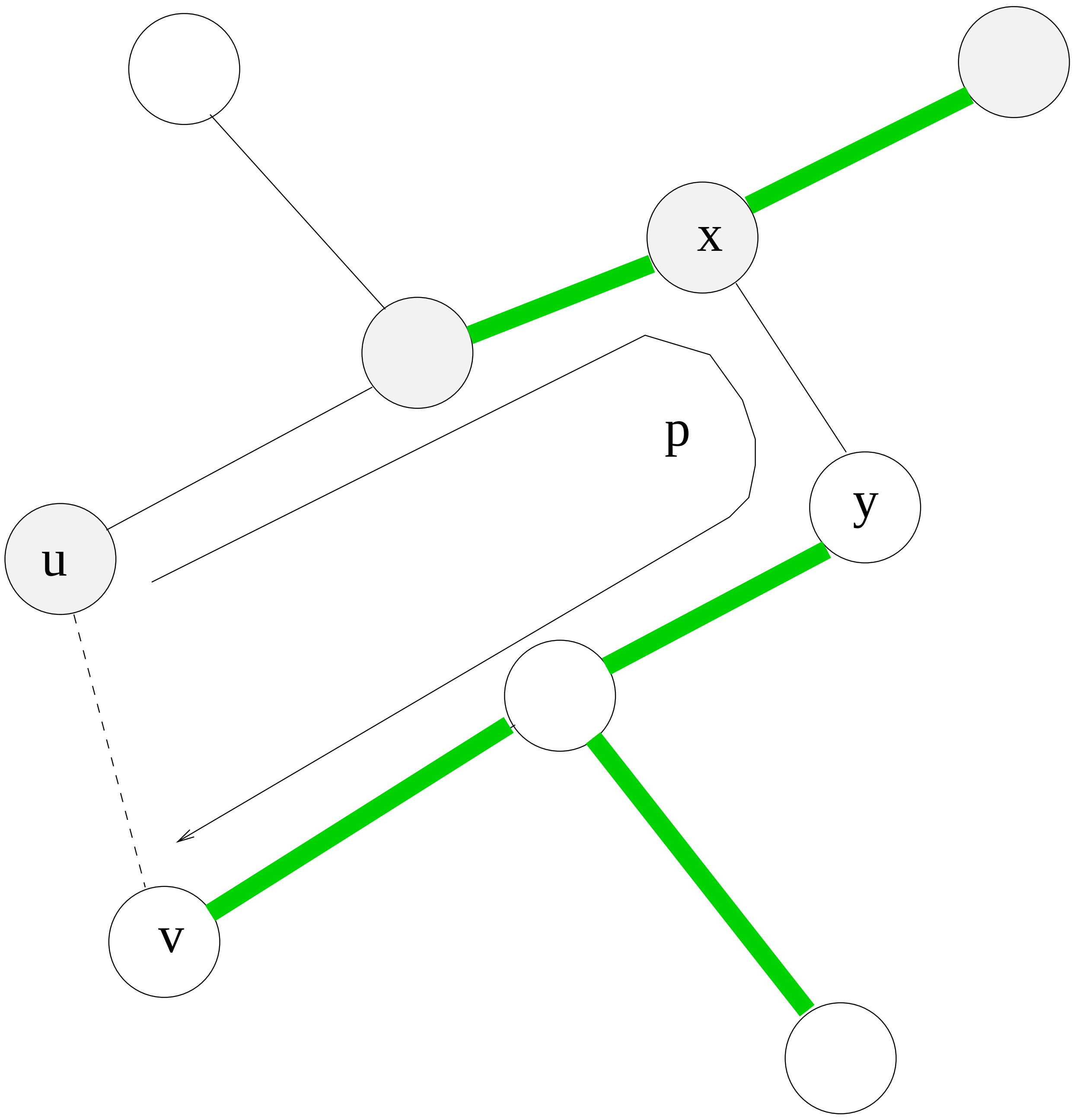
Da u und v auf entgegengesetzten Seiten des Schnittes liegen, gibt es mindestens eine Kante (x, y) in T, die ebenfalls den Schnitt kreuzt.

(x, y) ist nicht in A, da A den Schnitt respektiert.

Da (x, y) auf dem eindeutigen Pfad von u nach v in T liegt, zerlegt das Entfernen von (x, y) T in zwei Komponenten.

Hinzufügen von (u, v) erzeugt einen neuen Spannbaum $T' = T \setminus \{(x, y)\} \cup \{u, v\}$.

Wir zeigen, daß T' ein minimaler Spannbaum ist.


x
p
y
u
v

Da (u, v) eine leichte Kante ist, gilt $w(u, v) \leq w(x, y)$, Also

$$\begin{aligned} w(T') &= w(T) - w(x, y) + w(u, v) \\ &\leq w(T) \end{aligned}$$

Also ist T' ein minimaler Spannbaum.

Es gilt nun zu zeigen: (u, v) ist sicher für A. Es gilt:

- $A \subseteq T'$, da $A \subseteq T$ und $(u, v) \notin A$. Also
- $A \cup \{(u, v)\} \subseteq T'$.

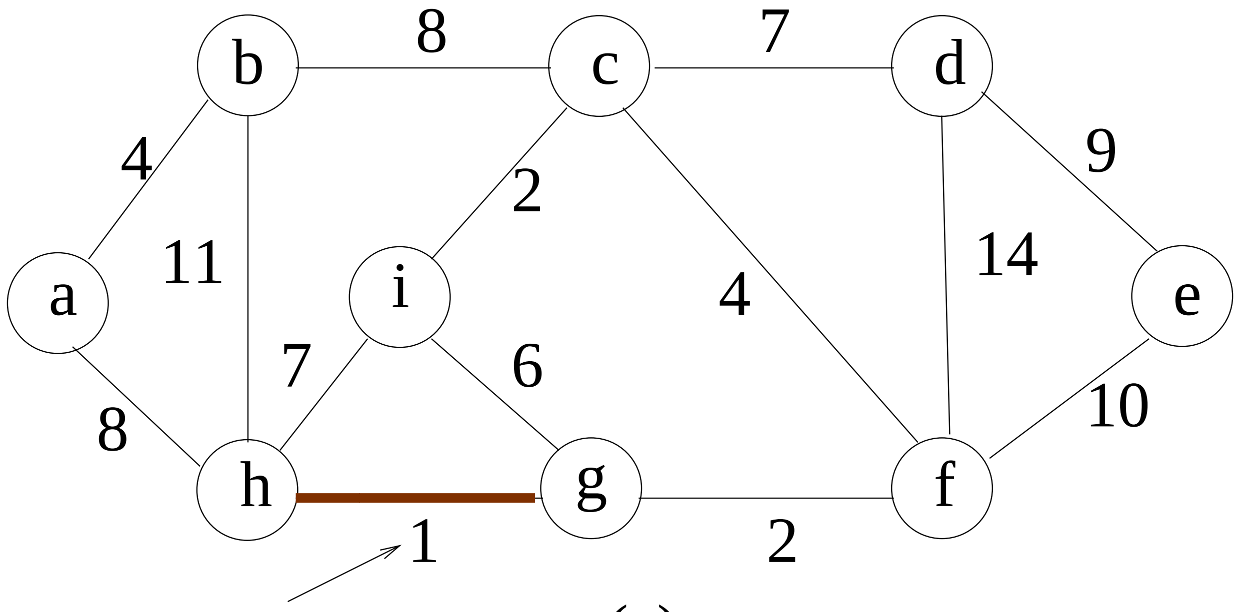
Da T' ein minimaler Spannbaum ist, folgt die Behauptung.

Korollar 24.2 Sei $G = (V, E)$ ein zusammenhängender, ungerichteter Graph mit reeller Gewichtsfunktion w für E. Sei $A \subseteq E$ eine Teilmenge eines minimalen Spannbaums. Sei C eine Zusammenhangskomponente im Wald $G_A = (V, A)$.

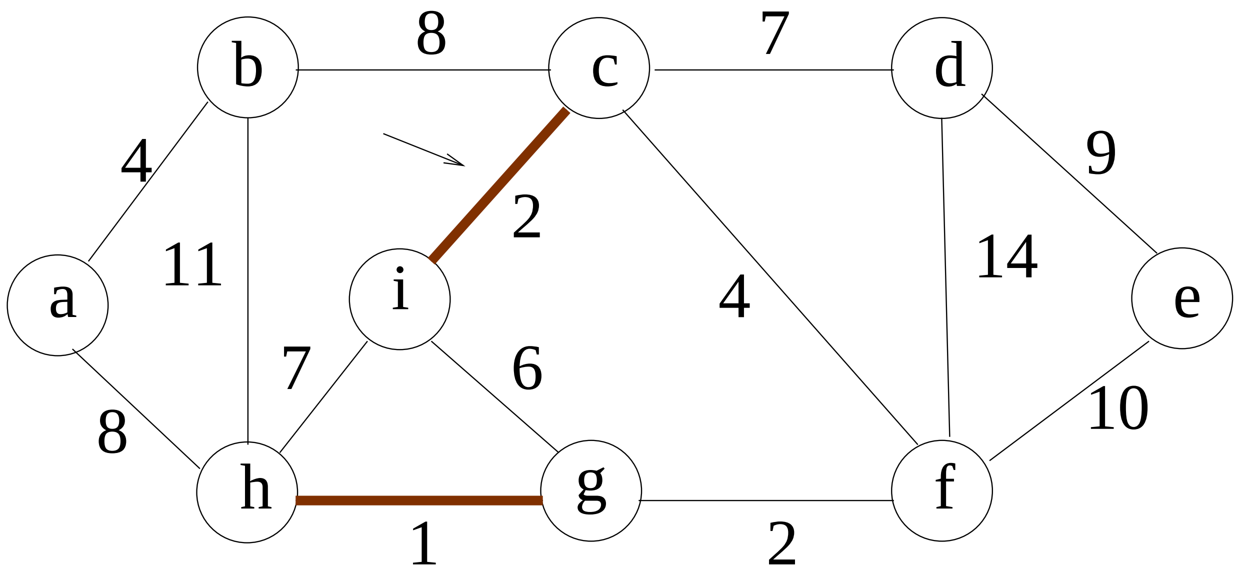
Falls (u, v) eine leichte Kante ist, die C mit einer anderen Komponente verbindet, so ist (u, v) sicher für A.

Beweis Der Schnitt $(C, V \setminus C)$ respektiert A. Daher ist (u, v) eine leichte Kante.

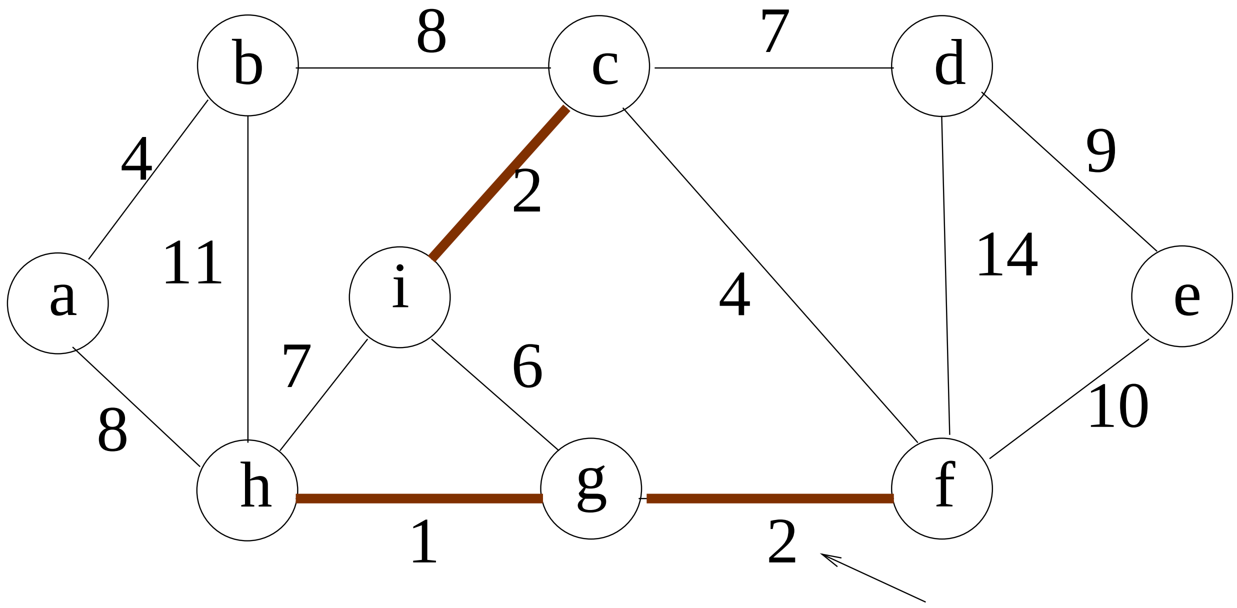
```
MST-KRUSKAL(G, w)
A ← ∅
for each vertex v ∈ V[G]
   do MAKE-SET(v)
sort the edges of E by nondecreasing weight w
for each edge (u, v) ∈ E
   do if FIND-SET(u) ≠ FIND-SET(v)
         then A ← A ∪ {(u, v)}
              UNION(u, v)
return A
```


b
8
c
7
d
4
9
2
11
14
a
i
4
e
7
6
8
10
h
g
f
1
2

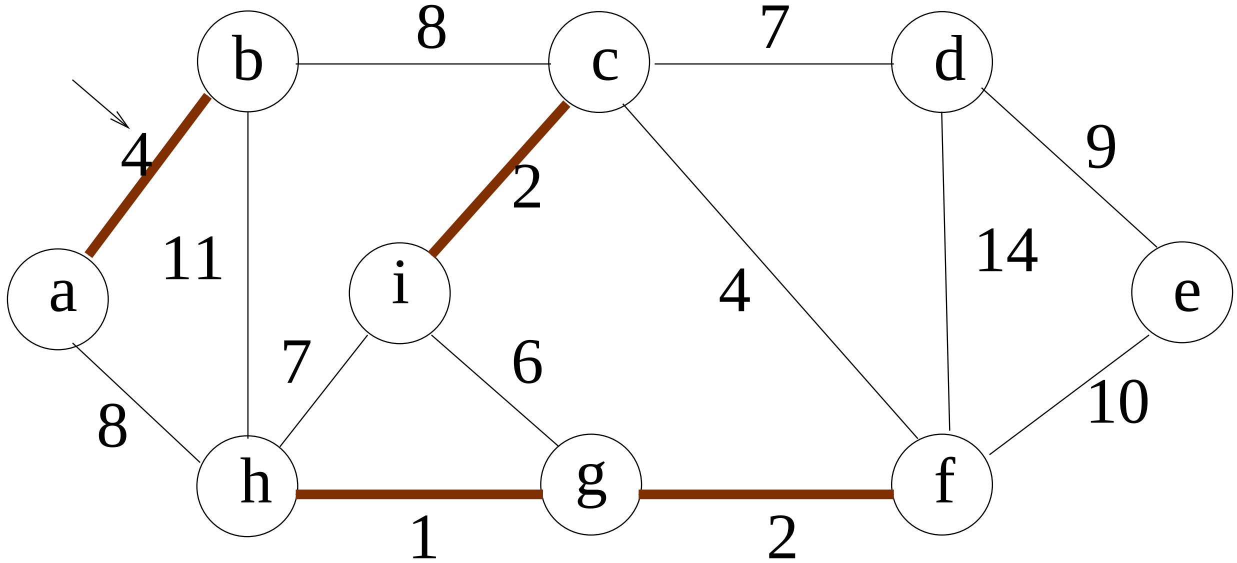
(a)


b
8
c
7
d
4
9
2
11
14
a
i
4
e
7
6
8
10
h
g
f
1
2

(b)


b
c
d
a
i
e
h
g
f
8
7
4
2
9
11
14
4
7
6
10
8
1
2

(c)


b
c
d
a
i
e
h
g
f
8
7
4
2
9
11
14
4
7
6
10
8
1
2

(d)

Laufzeit des Algorithmus von Kruskal:

- Initialisierung: $O(V)$
- Sortieren: $O(E \lg E)$
- $O(E)$ Vereinigungen: $O(E\alpha(E, V))$
  α ist inverse Ackermannfunktion. Hier wurde schnellster Unionalgorithmus angenommen, den es gibt. (ist euch leider unbekannt).

  Es gilt aber: $\alpha(E, V) = O(\lg E)$
- Zusammen: $O(E \lg E)$

Algorithmus von Prim

Startet bei einer beliebigen Wurzel r und fügt Kanten zu noch nicht erreichten Knoten hinzu.

Alle noch nicht erreichten Knoten werden in einer Prioritätsschlange Q verwaltet, deren Schlüssel (key) das Minimum aller Kanten vom entsprechenden Knoten bis zum bisherigen Baum darstellen.

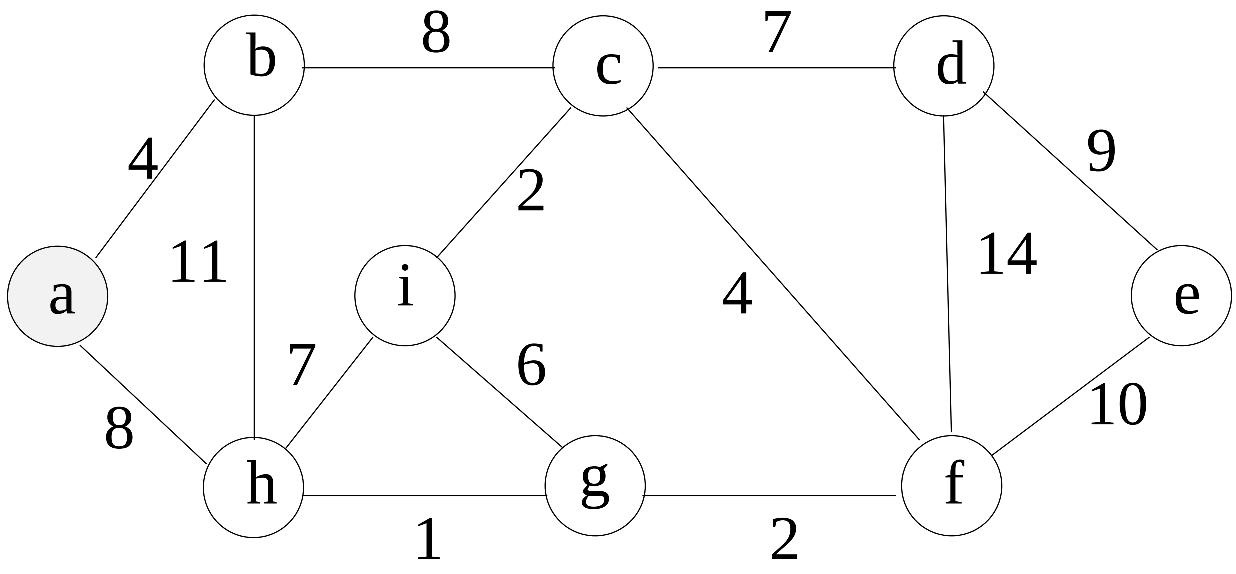
key[v]$= \infty$ falls so eine Kante nicht existiert.

$\pi[v]$ ist der Vater von v im bisherigen Baum.

A wird implizit gehalten:

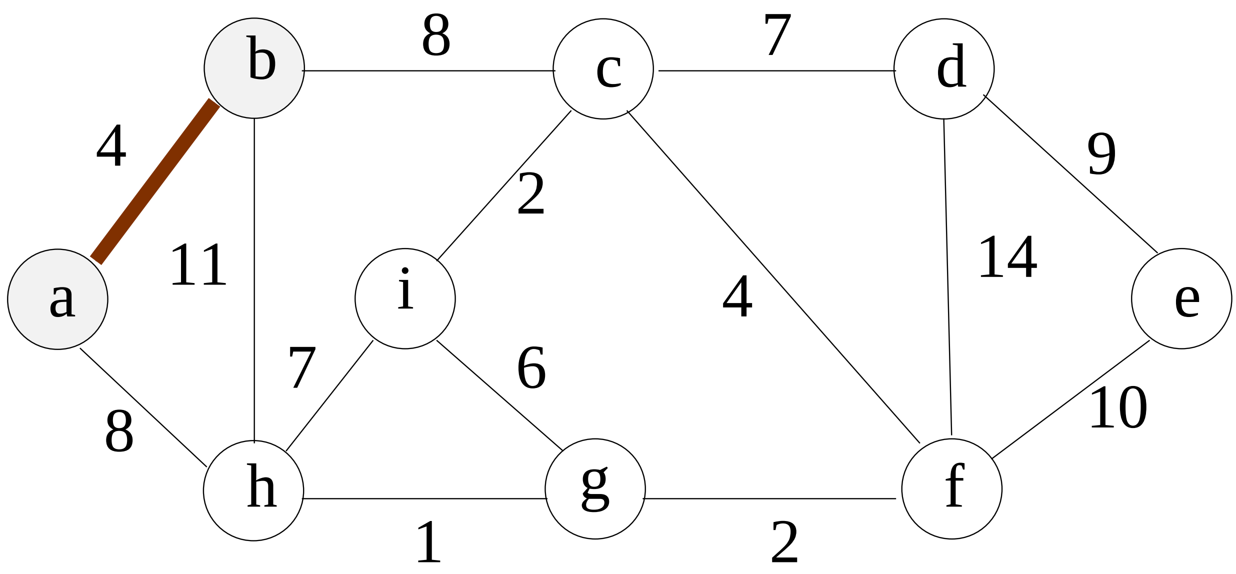
$$A = \{(v, \pi[v]) | v \in V \setminus \{r\} \setminus Q\}$$

MST-PRIM(G, w, r)
$Q \leftarrow V[G]$
for each $u \in Q$
    do $key[u] \leftarrow \infty$
$key[r] \leftarrow 0$
$\pi[r] \leftarrow$ NIL
while $Q \neq \emptyset$
    do $u \leftarrow$ EXTRACT-MIN(Q)
        for each $v \in Adj[u]$
            do if $v \in Q$ and $w(u, v) < key[v]$
                then $\pi[v] \leftarrow u$
                    $key[v] \leftarrow w(u, v)$



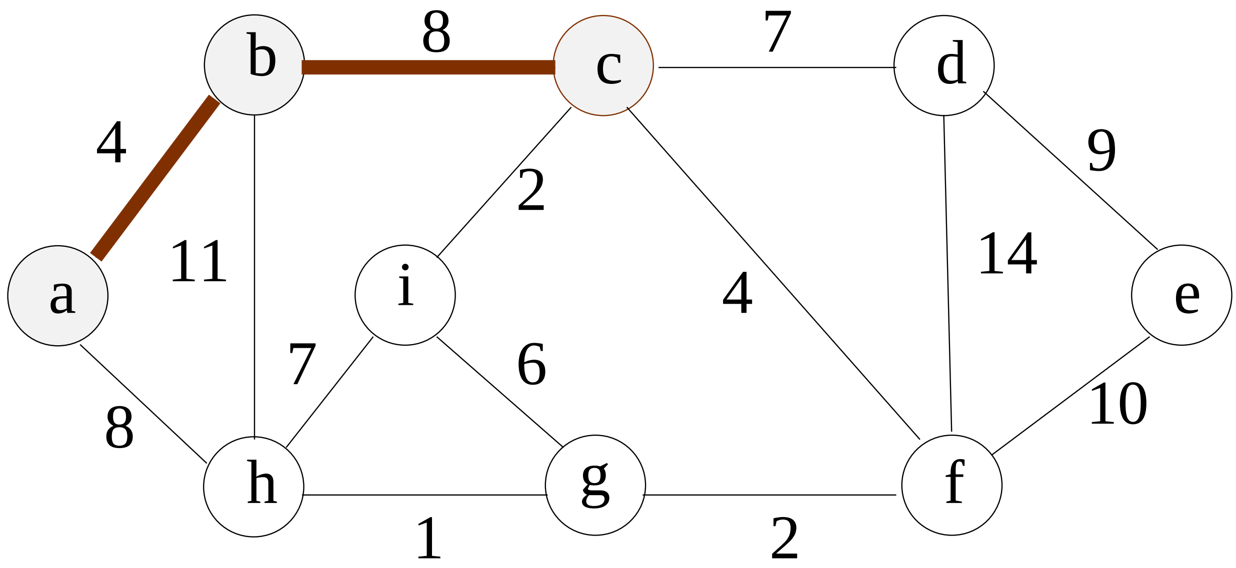
b
8
c
7
d
4
2
9
a
11
i
14
e
4
7
6
10
8
h
g
f
1
2

(a)



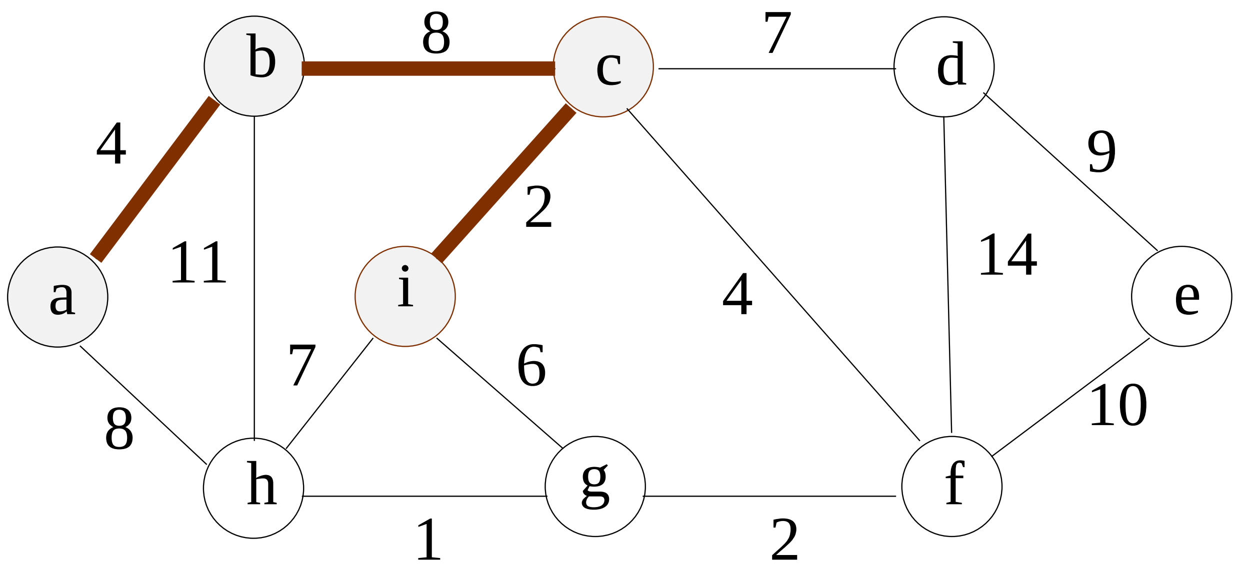
b
8
c
7
d
4
2
9
a
11
i
14
e
4
7
6
10
8
h
g
f
1
2

(b)



a
b
c
d
e
f
g
h
i
4
8
7
9
11
2
14
4
7
6
10
8
1
2

(c)



a
b
c
d
e
f
g
h
i
4
8
7
9
11
2
14
4
7
6
10
8
1
2

(d)

Laufzeit hängt von der Implementierung von Q ab.

Binary Heaps:

- Initialisierung: $O(V)$
- Schleife Z6-7: $O(V \lg V)$
- Schleife Z8-11: $O(E)$ Durchläufe
- Z9: $O(1)$, falls Bit in Knoten anzeigt, ob dieser in Q ist oder nicht
- Z11: entspricht DECREASE-KEY: $O(\lg V)$
- Summe: $O(V \lg V + E \lg V)$

Fibonacci Heaps:

- $O(E + V \lg V)$

Kürzeste Wege

Oft gebraucht

Kürzester Weg von A nach B

Gegeben: $G = (V, E)$ und $w : E \to \mathcal{R}$

Pfadlänge von $p = \langle v_0, \ldots, v_k \rangle$

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

Länge des kürzesten Pfades von u nach v

$$\delta(u, v) = \begin{cases} \min\{w(p) | u \overset{p}{\leadsto} v\} & \text{falls } u, v \text{ verbunden} \\ \infty & \text{sonst} \end{cases}$$

Kürzester Pfad ist einer mit eben diesem Gewicht.

Wir betrachten nur positive Gewichte/Längen.

Varianten

Hier (Single-Source shortest-paths problem)

Gegeben einen Startknoten s, finde die kürzesten Pfade zu allen $v \in V$.

Varianten

- single-destination shortest-paths problem
- single-pair shortest path problem
- all-pairs shortest path problem
  hier gibt es schnellere als die offensichtliche Lösung

Vereinbarungen

$\pi[v]$ ist Vorgänger im kürzesten Pfad

PRINT-PATH aus Breitensuche kann wiederverwendet werden.

Wiederholung (Vorgängerteilgraph):

$$V_\pi = \{v \in V | \pi[v] \neq NIL\} \cup \{s\}$$

$$E_\pi = \{(\pi[v], v) \in E | v \in V_\pi \setminus \{s\}\}$$