Kapitel V

Algorithmen auf Graphen

Inhalt Kapitel V

Zum Inhalt

- Grundlegendes
  - Repräsentation von Graphen **22.1**
  - Breiten- und Tiefensuche **22.2, 22.3**
  - Anwendungen der Tiefensuche **22.4, 22.5**
- Minimale Spannbäume **23**
  - Algorithmus von Prim
  - Algorithmus von Kruskal
- Kürzeste Wege **24,25**
  - Algorithmus von Dijkstra **24.3**
  - Bellman-Ford-Algorithmus **24.1**
  - Floyd-Warshall-Algorithmus **25.2**
- Flüsse in Netzwerken **26.1-26.3**

Repräsentation von Graphen

Graph $G = (V, E)$, wobei $E \subseteq V \times V$ (gerichteter Graph) bzw. $E \subseteq \binom{V}{2}$ (ungerichteter Graph).

Adjazenzlisten

Für jeden Knoten $v \in V$ werden in einer verketteten Liste $Adj[v]$ alle Nachbarn u mit $(v, u) \in E$ (bzw. mit $\{v, u\} \in E$) gespeichert.
Platzbedarf: $O(|V| + |E| \log |V|)$

Adjazenzmatrix

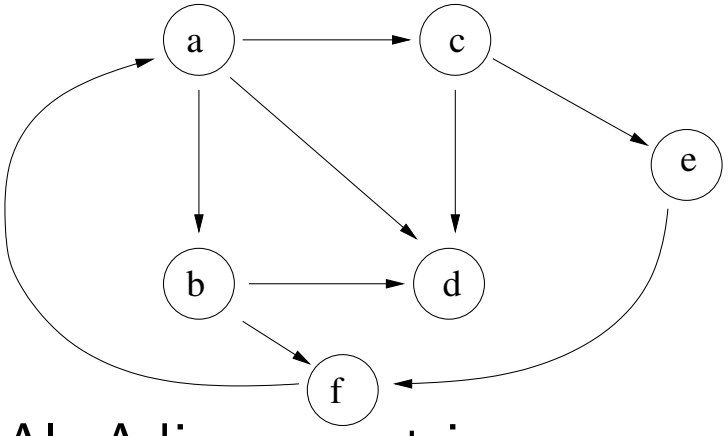
Es wird eine $|V| \times |V|$-Matrix $A = (a_{ij})$ gespeichert, mit

$$a_{ij} = 1 \quad \text{genau dann, wenn} \quad (v_i, v_j) \in E \ .$$

Platzbedarf: $O(|V|^2)$

Beispiel

Ein Graph:



In mathematischer Notation:

$$\begin{aligned} G &= (V, E) \\ V &= \{a, b, c, d, e, f\} \\ E &= \{(a, b), (a, c), (a, d), (c, e), \\ &\quad (b, f), (e, f), (f, a)\} \end{aligned}$$

Als Adjazenzmatrix:

	a	b	c	d	e	f
a	0	1	1	1	0	0
b	0	0	0	1	0	1
c	0	0	0	1	1	0
d	0	0	0	0	0	0
e	0	0	0	0	0	1
f	1	0	0	0	0	0

Mit Adjazenzlisten:

a	$[b; c; d]$
b	$[d; f]$
c	$[d; e]$
d	$[]$
e	$[f]$
f	$[a]$

Breitensuche

Gegeben: Graph G (als Adjazenzlisten), ausgezeichneter Startknoten $s \in V$.

Gesucht für jeden Knoten $v \in V$: kürzester Pfad zu s und Distanz $d[v]$.

Speichere für jedes $v \in V$ den Vorgänger $\pi[v]$ auf kürzestem Pfad zu s.

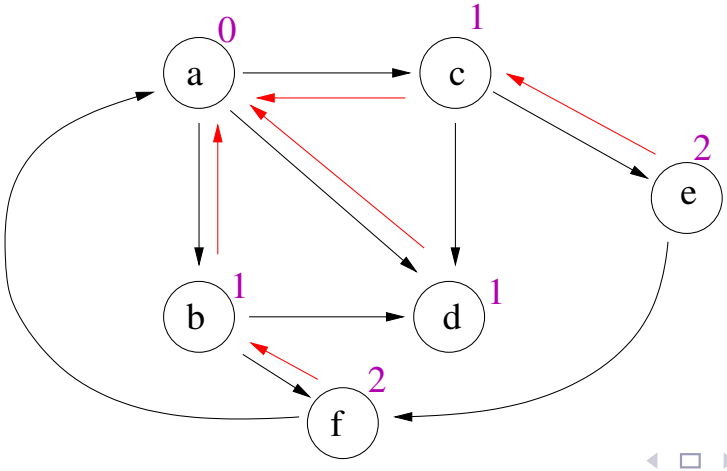
Initialisiere $\pi[v] = \text{NIL}$ und $d[v] = \infty$ für alle $v \in V$.
Setze $d[s] = 0$ und speichere s in einer *FIFO-queue* Q.

Breitensuche in Pseudocode

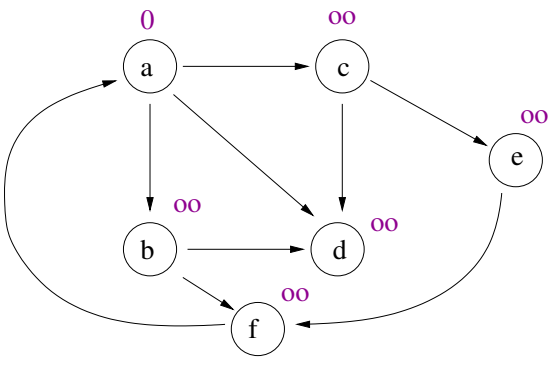
Breitensuche

put(Q,s)
while $Q \neq \emptyset$
  do $v \leftarrow get(Q)$
    for each $u \in Adj[v]$ with $d[u] = \infty$
      do $d[u] \leftarrow d[v] + 1$
        $\pi[u] \leftarrow v$
        $put(Q, u)$

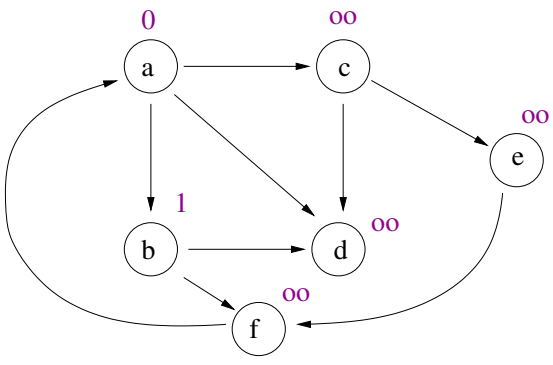
Beispiel mit $s = a$:



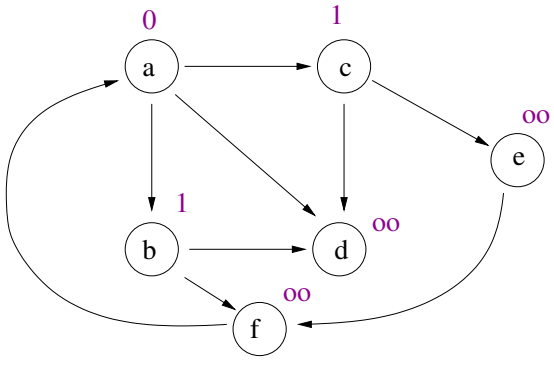
Beispiel



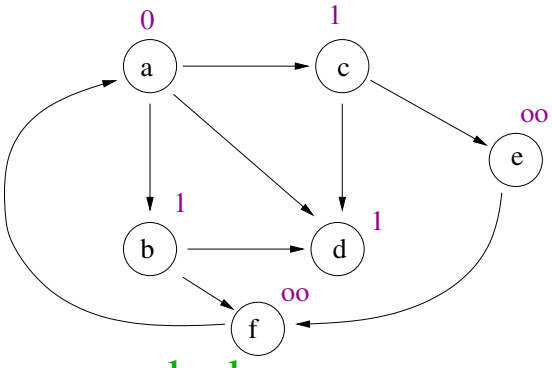
a



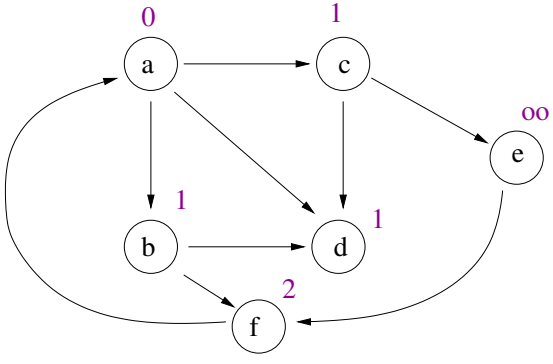
b



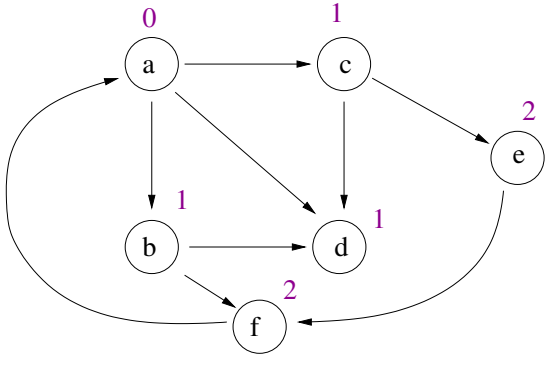
bc



bcd



cdf



dfe

Zeilenweise v.l.n.r. zu lesen. Die Queue ist jeweils grün dargestellt. Die π-Zeiger wurden weggelassen.

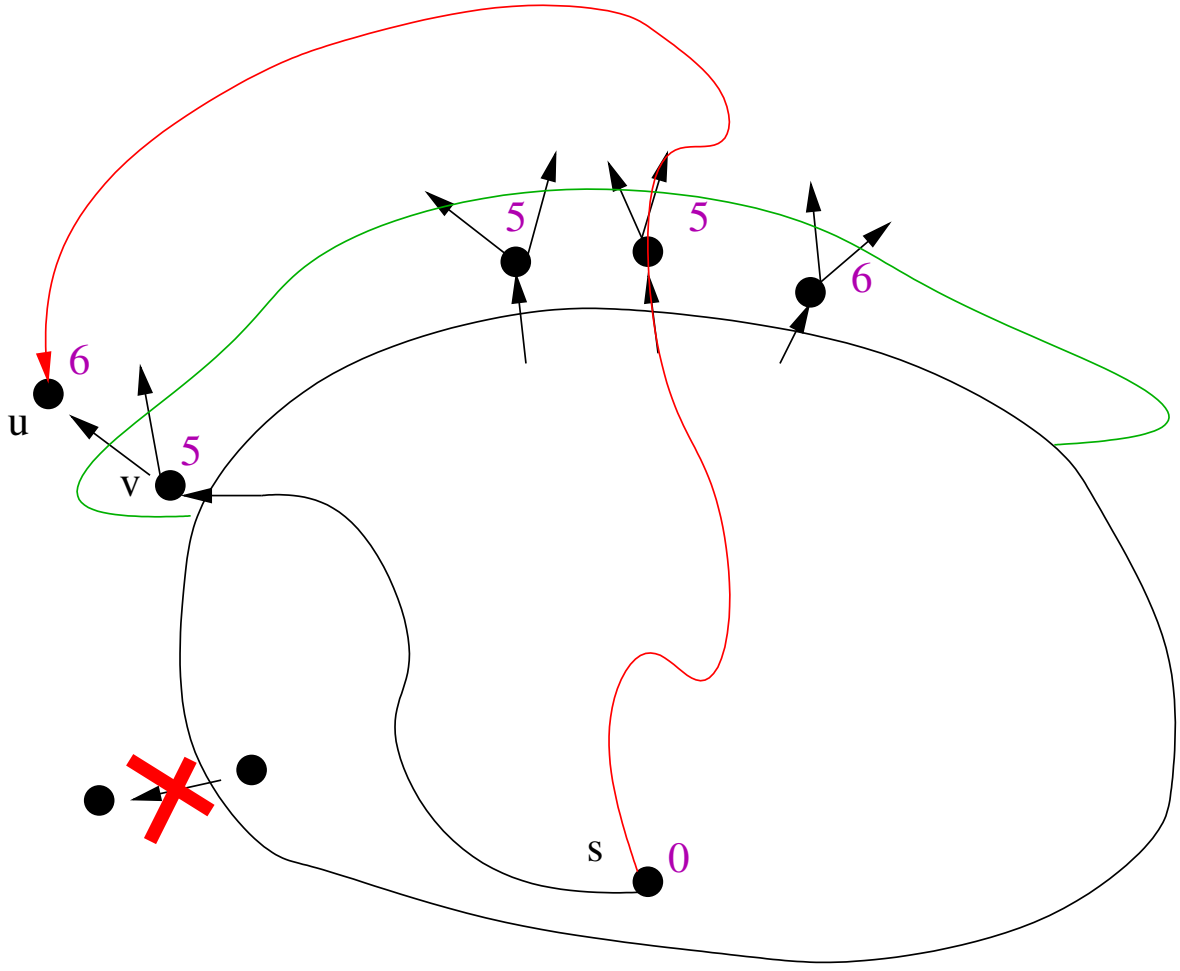
Korrektheit der Breitensuche

Man beweist die Korrektheit (Distanzen stimmen, π-Zeiger definieren kürzeste Pfade) mithilfe der folgenden Invariante:

- Alle d-Einträge $\neq \infty$ und alle π-Einträge $\neq \textsc{Nil}$ sind korrekt.
- Die d-Einträge innerhalb der Schlange unterscheiden sich um maximal Eins und nehmen nicht ab. Also 5,5,5,5,5,5,5,5 oder 5,5,5,5,5,6,6,6,6,6
- Die Nachbarn von abgearbeiteten Knoten sind entweder selbst abgearbeitet, oder in der Queue.

Am Ende sind also alle von s aus erreichbaren Knoten abgearbeitet und korrekt beschriftet.

Korrektheit der Breitensuche



Abgearb. Bereich und kürzester Pfad von s nach v in schwarz. Queue in grün, anderer Pfad vom Startknoten s nach u in rot. Aus dem abgearb. Bereich heraus kommt man nur über die Queue.

Tiefensuche

Tiefensuche (Depth-First-Search, DFS):

Sucht jeden Knoten einmal auf, sondert eine Teilmenge der Kanten aus, die einen Wald (den DFS-Wald) bilden.

Hilfsmittel Färbung:

- Weiß $\hat{=}$ noch nicht besucht.
- Grau $\hat{=}$ schon besucht, aber noch nicht abgefertigt
- Schwarz $\hat{=}$ abgefertigt, d.h. der gesamte von hier erreichbare Teil wurde durchsucht.

Pseudocode für Tiefensuche

Tiefensuche

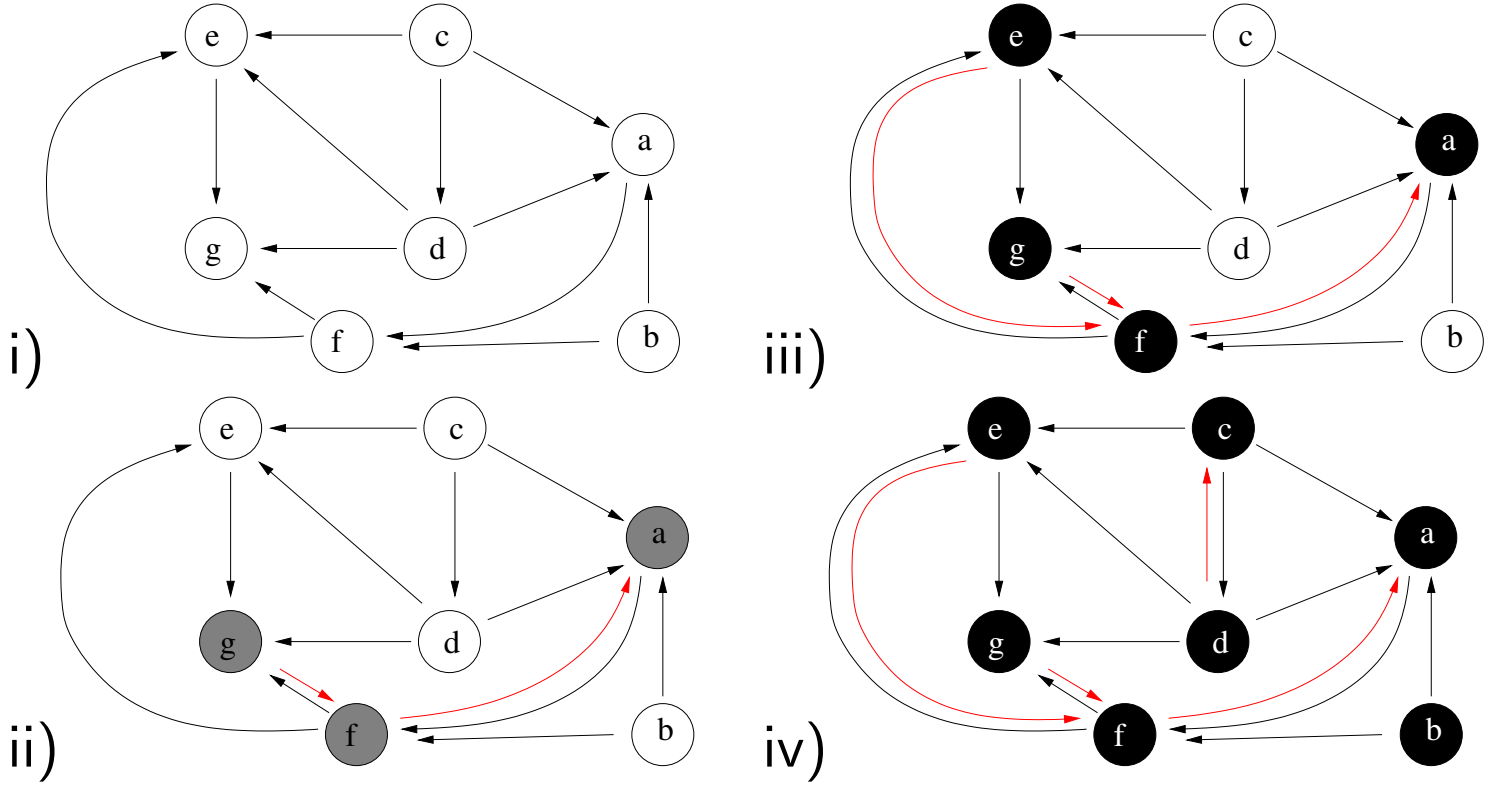
DFS(G)

initialisiere $color[v] \leftarrow white$ und $\pi[v] \leftarrow \text{NIL}$ für alle $v \in V$
for each $v \in V$
do if $color[v] = white$
then DFS-Visit(G, v)

DFS-Visit(G, v)

$color[v] \leftarrow grey$
for each $u \in Adj[v]$ with $color[u] = white$
do $\pi[u] \leftarrow v$
DFS-Visit(G, u)
$color[v] \leftarrow black$

Beispiel



Durchführung der DFS in alphabetischer Reihenfolge.
Gezeigt sind der Eingabegraph, zwei Zwischenstadien, und das Endergebnis. NB: $Adj[f] = [g; e]$, alle anderen in alphabetischer Reihenfolge.

Abfertigungszeiten

Die DFS liefert noch mehr Information, wenn man eine "Uhr" mitlaufen lässt und zu jedem Knoten v die

- Entdeckungszeit $d[v]$ (*discovery time*) und die
- Abfertigungszeit $f[v]$ (*finishing time*)

abspeichert.
Die Uhr wird vor jeder Zeitzuweisung weitergestellt.

Tiefensuche: Pseudocode

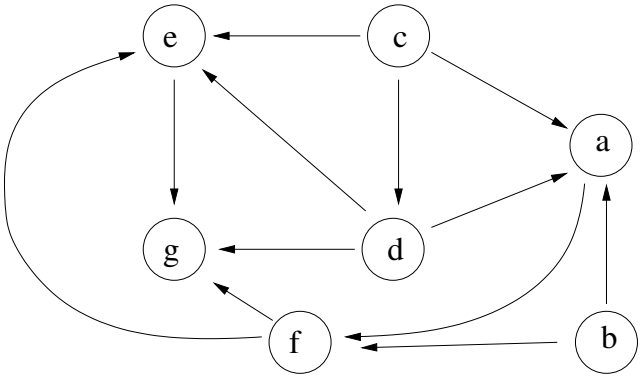
Tiefensuche mit Zeiten

DFS(G)
    initialisiere $color[v] \leftarrow white$ und $\pi[v] \leftarrow \text{NIL}$ für alle $v \in V$
    $time \leftarrow 0$
    for each $v \in V$
        do if $color[v] = white$
            then DFS-Visit(G, v)

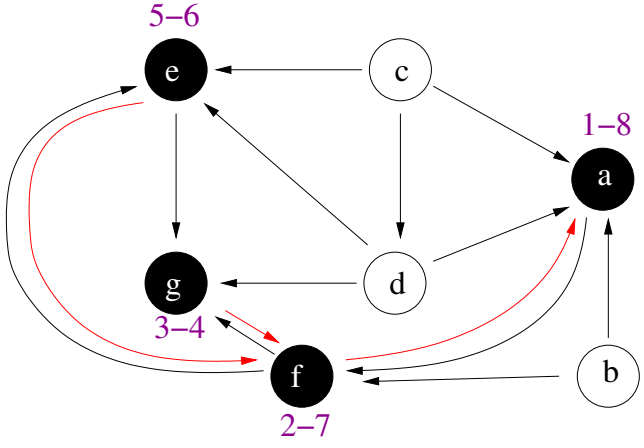
DFS-Visit(G, v)
    $color[v] \leftarrow grey$
    $d[v] \leftarrow ++time$
    for each $u \in Adj[v]$ with $color[u] = white$
        do $\pi[u] \leftarrow v$
            DFS-Visit(G, u)
    $color[v] \leftarrow black$; $f[v] \leftarrow ++time$

Beispiel mit Zeiten

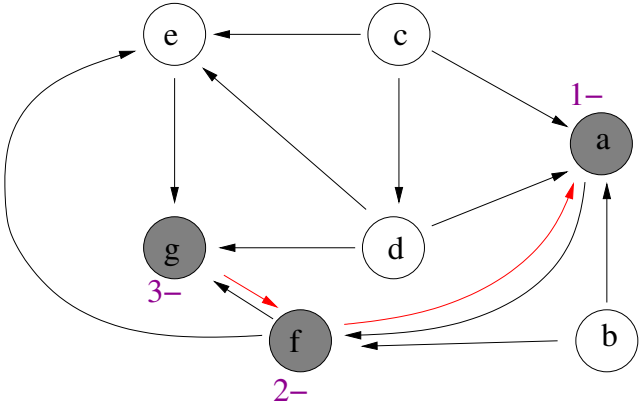
i)



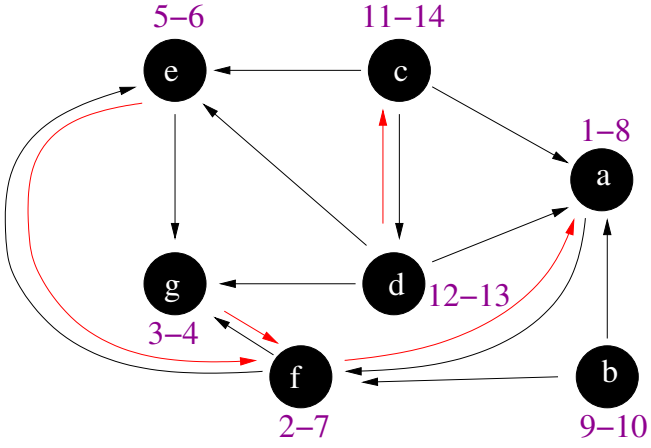
iii)



ii)



iv)



Durchführung der DFS mit Zeiten in alphabetischer Reihenfolge. Gezeigt sind der Eingabegraph, zwei Zwischenstadien, und das Endergebnis.

Klammerungseigenschaft

Seien $u, v \in V$ und $u \neq v$. Die folgenden drei Fälle sind möglich:

- $d[u] < d[v] < f[v] < f[u]$ und v ist Nachfahre von u im DFS-Wald.
- $d[v] < d[u] < f[u] < f[v]$ und u ist Nachfahre von v im DFS-Wald.
- $[d[u], f[u]] \cap [d[v], f[v]] = \emptyset$ und weder ist u Nachfahre von v im DFS-Wald noch umgekehrt.

Insbesondere ist die Konstellation $d[u] < d[v] < f[u] < f[v]$ unmöglich und der DFS-Wald lässt sich aus den Aktivitätsintervallen $[d[v], f[v]]$ eindeutig rekonstruieren.

Klassifikation der Kanten

Durch DFS werden die Kanten eines Graphen in die folgenden vier Typen klassifiziert.

- Baumkanten sind die Kanten des DFS-Waldes, also (u, v) mit $\pi[v] = u$.

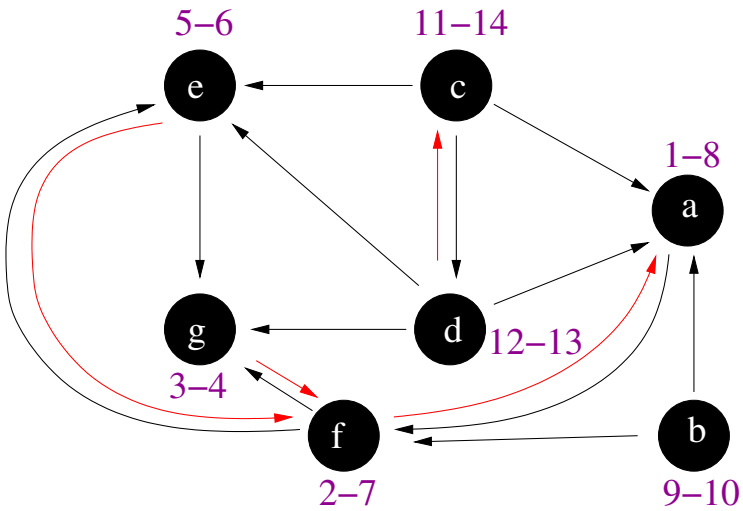
  Kennzeichen: Beim ersten Durchlaufen ist v weiß.
- Rückwärtskanten (u, v), wo v Vorf. von u im DFS-Wald ist.

  Kennzeichen: Beim ersten Durchlaufen ist v grau.
- Vorwärtskanten (u, v), wo v Nachk. von u im DFS-Wald ist.

  Kennzeichen: Beim ersten Durchlaufen ist v schwarz, und $d[u] < d[v]$.
- Querkanten sind alle übrigen Kanten (u, v).
  Kennzeichen: Beim ersten Durchlaufen ist v schwarz, und $d[u] > d[v]$.

Bei ungerichteten Graphen kommen nur Baum- und Rückwärtskanten vor.

Beispiel



- Die Kante (a, f) ist Baumkante.
- Wäre die Adjazenzliste von a gleich $[f; e]$ statt nur $[f]$, so wäre (a, e) eine Vorwärtskante. Bei ihrem ersten Durchlaufen würde e bereits schwarz vorgefunden.
- Wäre die Adjazenzliste von f gleich $[g; e; a]$ statt nur $[g; e]$, so wäre (f, a) eine Rückwärtskante. Bei ihrem ersten Durchlaufen würde a grau vorgefunden.
- Die Kante (e, g) ist Querkante.

Topologische Sortierung

Eine topologische Ordnung eines gerichteten, azyklischen Graphen (dag) ist eine lineare Ordnung der Knoten

$$v_1 \prec v_2 \prec \ldots \prec v_n$$

so dass für jede Kante $(u, v) \in E$ gilt $u \prec v$.

Lemma

Ein gerichteter Graph ist genau dann azyklisch, wenn bei DFS keine Rückwärtskanten entstehen.

Satz

Eine topologische Ordnung auf einem dag G erhält man durch absteigende Sortierung nach den *finishing times* $f[v]$ nach Ausführung von DFS(G).

Topologische Ordnung im Beispiel: $c \prec d \prec b \prec a \prec f \prec e \prec g$.

Beweis des Lemmas

Lemma

Ein gerichteter Graph ist genau dann azyklisch, wenn bei DFS keine Rückwärtskanten entstehen.

Beweis:

- Logisch äquivalent: Zyklus vorhanden $\iff$ Rückwärtskante vorhanden.
- Jede Rückwartskante liegt nach Def. auf einem Zyklus (Beweis von $\Leftarrow$).
- Sei ein Zyklus vorhanden und v sein als erstes entdeckter Knoten und (u, v) die Kante auf dem Zyklus, die zu v zurückführt. Die DFS exploriert alles von v Erreichbare, also auch u. Die genannte Kante wird dann Rückwärtskante. (Beweis von $\Rightarrow$).

Beweis des Satzes

Satz

Eine topologische Ordnung auf einem dag G erhält man durch absteigende Sortierung nach den *finishing times* $f[v]$ nach Ausführung von DFS(G).

Beweis: Sei (u, v) eine Kante. Wir müssen zeigen $f[u] > f[v]$.

- Fall $d[u] < d[v]$. Dann muss die Kante (u, v) beschritten werden, bevor u als abgearbeitet gelten kann. Dann aber war entweder v schon abgearbeitet oder wird es dann. Also $f[v] < f[u]$.
- Fall $d[v] < d[u]$. Nach Lemma kann (u, v) keine Rückwärtskante sein, ist also eine Querkante. Dann ist v beim Beschreiten bereits abgefertigt, also $f[v] < f[u]$.

Zusammenhang

Weg (oder Pfad) von v_0 nach v_k:

$$p = \langle v_0, \ldots, v_k \rangle \text{ mit } (v_i, v_{i+1}) \in E \text{ für alle } i < k \ .$$

Schreibweise: $p : v_0 \rightsquigarrow v_k$.

Für ungerichtete Graphen:

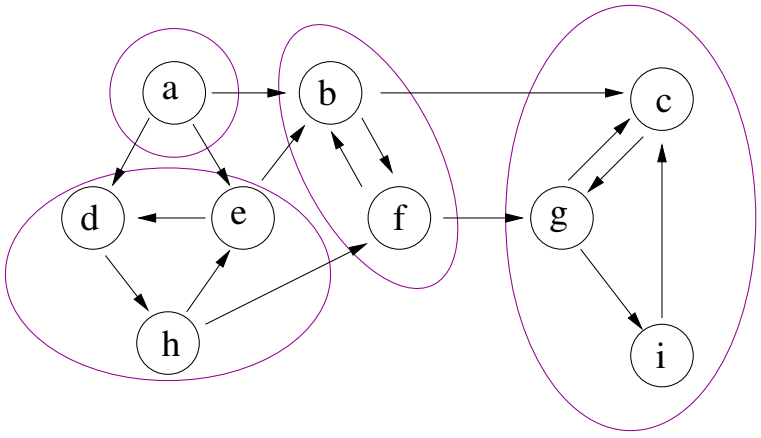
Zwei Knoten u und v heißen zusammenhängend, wenn es einen Weg $p : u \rightsquigarrow v$ gibt.

Für gerichtete Graphen:

Zwei Knoten u und v heißen stark zusammenhängend, wenn es Wege $p : u \rightsquigarrow v$ und $q : v \rightsquigarrow u$ gibt.

Die Äquivalenzklassen bzgl. dieser Äquivalenzrelation heißen (starke) Zusammenhangskomponenten (SCC).

Graph mit SCC



- Es gibt einen Pfad von g nach i, z.B.: $g \to c \to i$ und einen Pfad von i nach g, z.B.: $i \to c \to g$. Also hängen g und i stark zusammen und liegen in derselben SCC.
- Es gibt einen Pfad von a nach h, z.B.: $a \to e \to d \to h$, aber keinen Pfad von h nach a. Also sind a und h nicht stark zusammenhängend und liegen nicht in derselben SCC.
- Jeder Knoten liegt in genau einer SCC.
- Die SCCs bilden selbst die Knoten eines *dag* (ger. azykl. Graph), der eine Kante von SCC U nach SCC V besitzt, wenn es eine Kante von einem Knoten in U zu einem Knoten in V

Zerlegung in starke Zusammenhangskomponenten

Definition

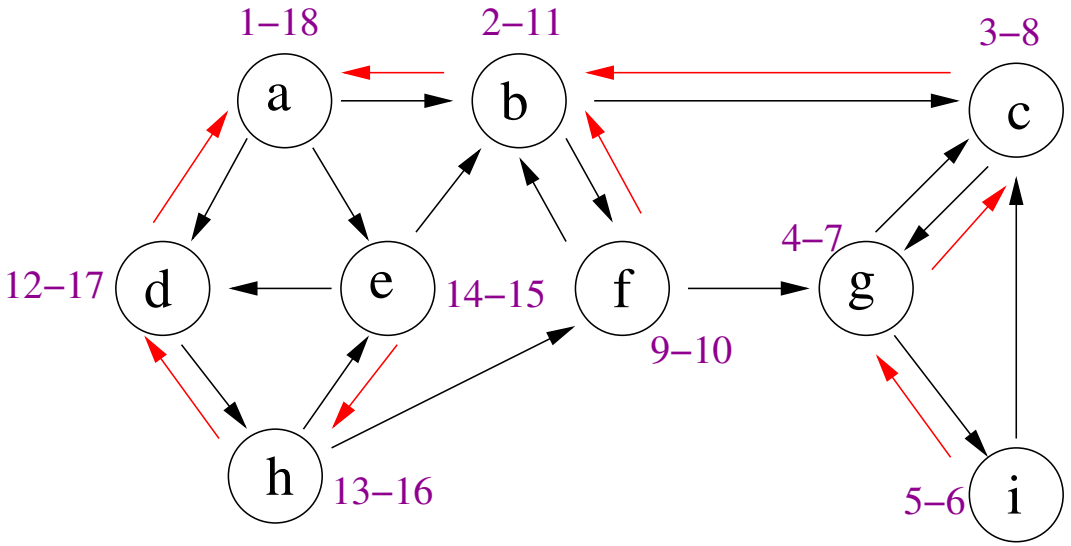
Der zu $G = (V, E)$ transponierte Graph ist $G^T := (V, E^T)$, wobei $(u, v) \in E^T$ gdw. $(v, u) \in E$.

Folgender Algorithmus zerlegt G in seine starken Zusammenhangskomponenten:

Zerlegen in SCC

- Zuerst wird $\mathrm{DFS}(G)$ aufgerufen.
- Sortiere die Knoten nach absteigender *finishing time*.
- Berechne G^T.
- Rufe $\mathrm{DFS}(G^T)$ auf, wobei die Knoten im Hauptprogramm in der Reihenfolge der obigen Sortierung behandelt werden.
- Starke Zusammenhangskomponenten von G sind die Bäume des im zweiten DFS berechneten DFS-Waldes.

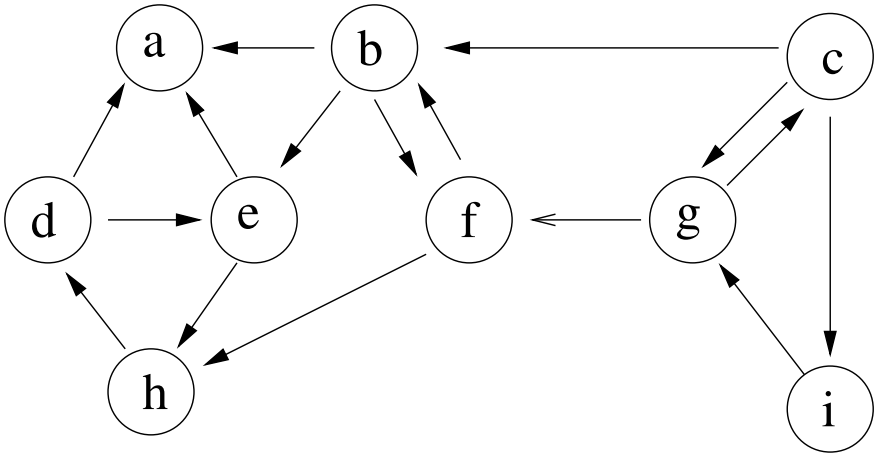
Beispiel: Durchführung der ersten DFS



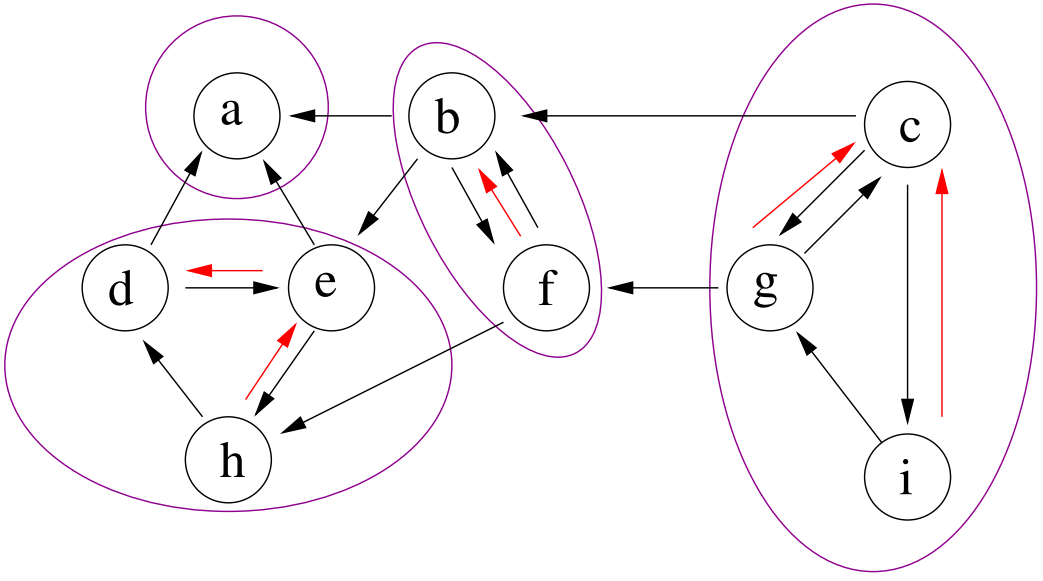
Sortierung nach absteigenden finishing times:

$$a, d, h, e, b, f, c, g, i$$

Bsp: Transponierter Graph und Durchf. der 2. DFS



Reihenfolge für die DFS: $a, d, h, e, b, f, c, g, i$



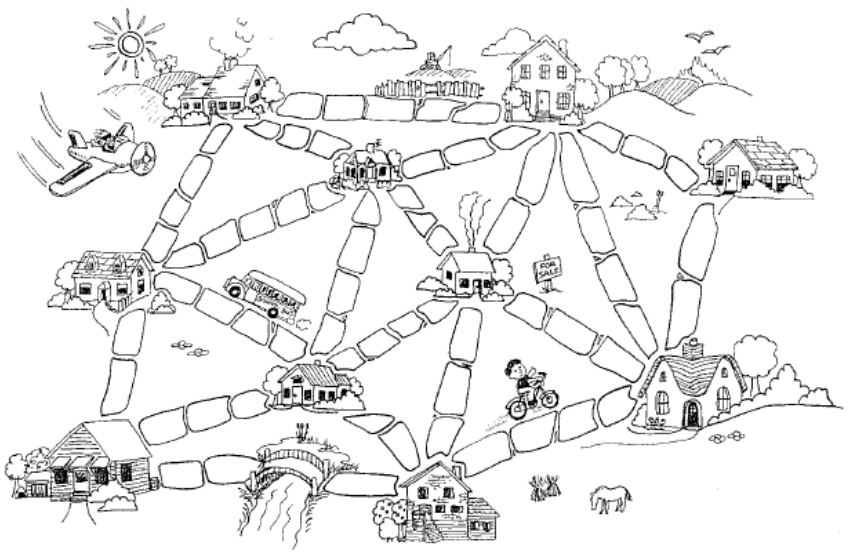
Korrektheitsbeweis

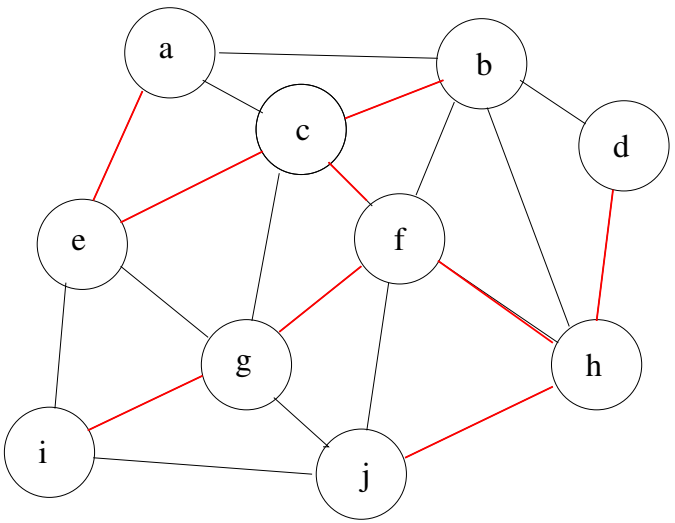
- Sei u Wurzel eines DFS Baums der zweiten Suche und v Nachfahre. Dann gibt einen Pfad in G^T von u nach v, also $v \rightsquigarrow u$ in G selbst.
  In der zweiten Suche kam dann u vor v dran, also wurde in der ersten Runde u später als v abgefertigt. Entweder ist $d[u] > f[v]$ oder $d[u] < d[v]$ (Schachtelungseigenschaft). Der erste Fall ist wg $v \rightsquigarrow u$ nicht möglich, im zweiten Falle ist v ein DFS-Nachfolger von u, also $u \rightsquigarrow v$. Also hängen u und v stark zusammen. Wg. Symmetrie und Transitivität hängt somit jeder Baum der zweiten Suche stark zusammen.
- Umgekehrt sei es jetzt so, dass u und v nicht im selben Baum der 2. DFS liegen. O.B.d.A. nehmen wir an, dass $f[u] > f[v]$. Dann wird also u vor v betrachtet und wenn es einen Pfad gibt von u nach v (in G^T), dann wird dieser auch entdeckt. Also gibt es keinen Pfad von u nach v in G^T und u, v sind nicht in derselben SCC.

Spannbäume

Definition

Gegeben sei ein zusammenhängender, unger. Graph $G = (V, E)$. Ein Spannbaum von G ist eine Teilmenge $T \subseteq E$ der Kanten, sodass (V, T) azyklisch ist und je zwei Knoten $u, v \in V$ durch einen Pfad in T verbunden sind.



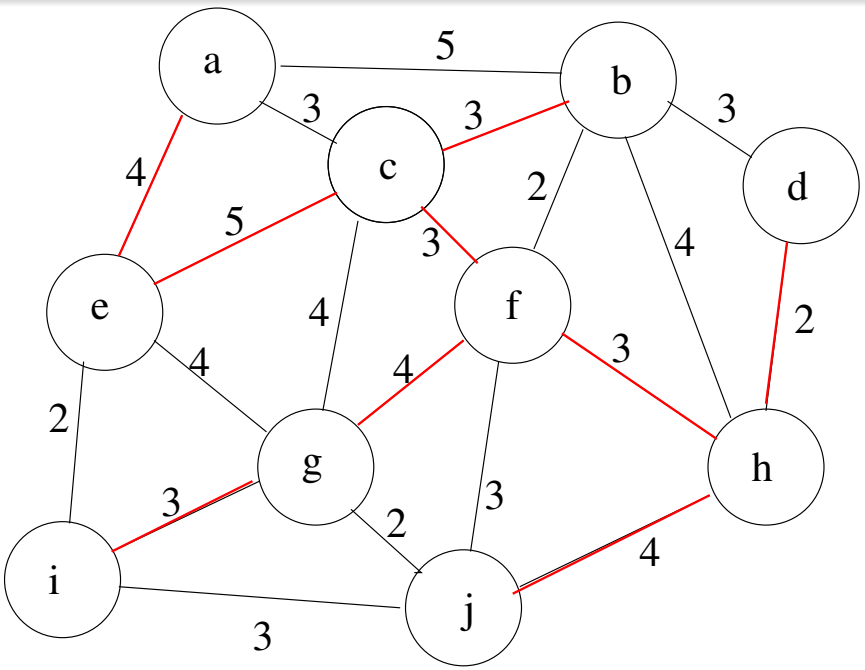


Stadtplan aus [Bell, Witten, Fellows: Comp. Sci. Unplugged, 1998].
Spannbaum im entsprechenden Graphen in Rot.

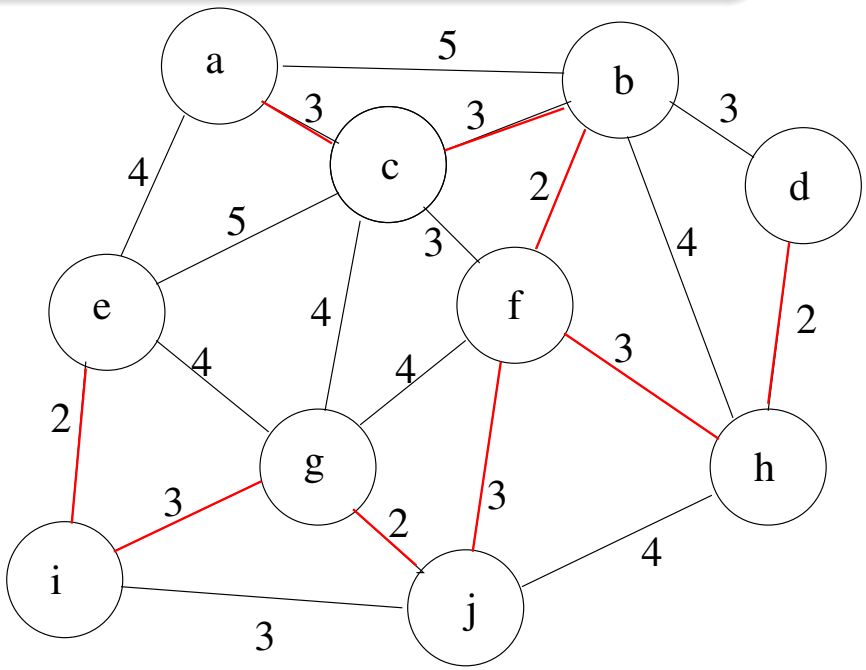
Minimaler Spannbaum

Definition

Sei $G = (V, E)$ ein unger. zush. Graph und $w : E \to \mathbb{R}_0^+$ eine Funktion, die jeder Kante ein "Gewicht" zuordnet. Ein minimaler Spannbaum (MST) ist ein Spannbaum T, sodass $\sum_{e \in T} w(e)$ minimal, also so klein wie möglich, ist.



Ein (nichtminimaler) Spannbaum vom Gewicht 31



Ein minimaler Spannbaum vom Gewicht 23

Anwendungen minimaler Spannbäume (MST)

- Planung von Versorgungsnetzen
- Clustering: Knoten = Datensätze, Kanten = "Nähe". Bilde minimalen Spannbaum, lösche Kanten großen Gewichts. Spannbaum zerfällt in Cluster.
- Elektrische Netzwerke (Kirchhoff'sches Gesetz).
- Routenplanung ("Problem des Handlungsreisenden").
- "Grundrechenart" der Graphalgorithmen. Sehr effizient zu berechnen, wird daher wo immer möglich verwendet. Vgl. Zucker in Lebensmitteln.

Grundalgorithmus

Definition

Sei $A \subseteq E$ Teilmenge eines minimalen Spannbaumes. Kante e heißt sicher für A, falls $A \cup \{e\}$ immer noch Teilmenge eines minimalen Spannbaumes ist.

Grundstruktur eines Greedy Algorithmus zur Berechnung eines minimalen Spannbaums:

Greedy-MST

Beginne mit $A \leftarrow \emptyset$, füge dann sukzessive Kanten hinzu, die sicher für A sind.

Noch benötigt: Strategie zur Findung sicherer Kanten.

Finden sicherer Kanten

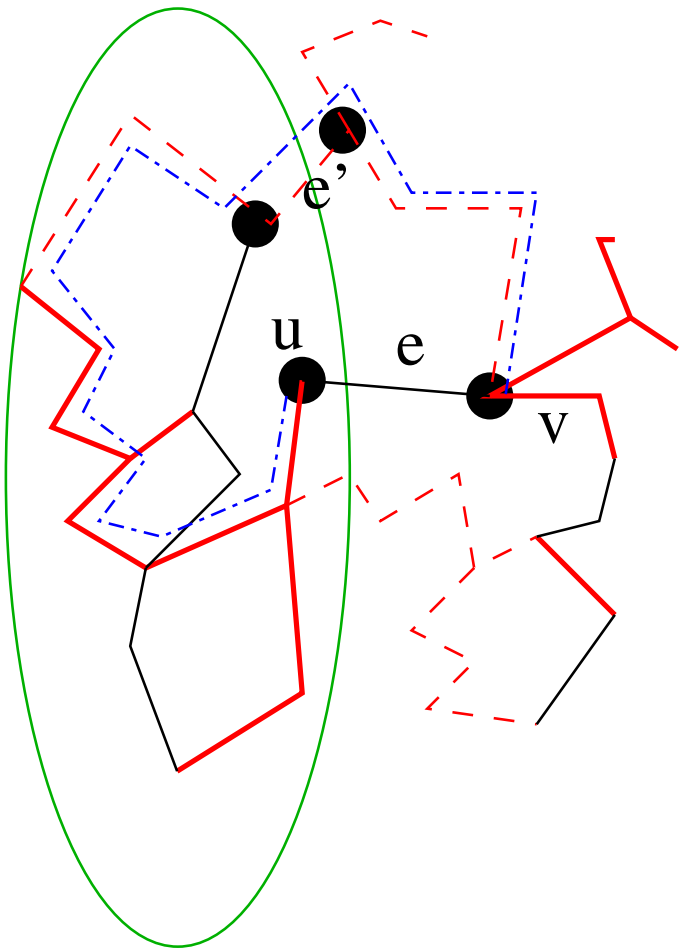
Definition

Für $S \subseteq V$ und $e \in E$ sagen wir "e kreuzt S", falls $e = \{u, v\}$ mit $u \in S$ und $v \in V \setminus S$.

Satz

Sei A Teilmenge eines minimalen Spannbaumes, sei $S \subseteq V$ mit der Eigenschaft: keine Kante in A kreuzt S, und sei e eine Kante minimalen Gewichtes, die S kreuzt.
Dann ist e sicher für A.

Beweis des Satzes



- Rot fett: die Teilmenge A;
- Rot gestrichelt: ein MST, der A erweitert. Dieser heiße T.
- Schwarz: andere Kanten des Graphen (exemplarisch);
- Grün: Die Menge S;
- $e = \{u, v\}$: Kante minimalen Gewichts, die S kreuzt; $u \in S$, $v \notin S$.
- Der MST, der A erweitert, muss auch u mit v durch einen Pfad (blau markiert) verbinden.
- Dieser Pfad muss mindestens eine Kante e' enthalten, die auch S kreuzt. NB $w(e') \geq w(e)$ wg. Minim. von e.
- Ist $e' = e$, so ist natürlich $e = e'$

Der Algorithmus von Prim

Benutzt eine Priority-Queue Q.

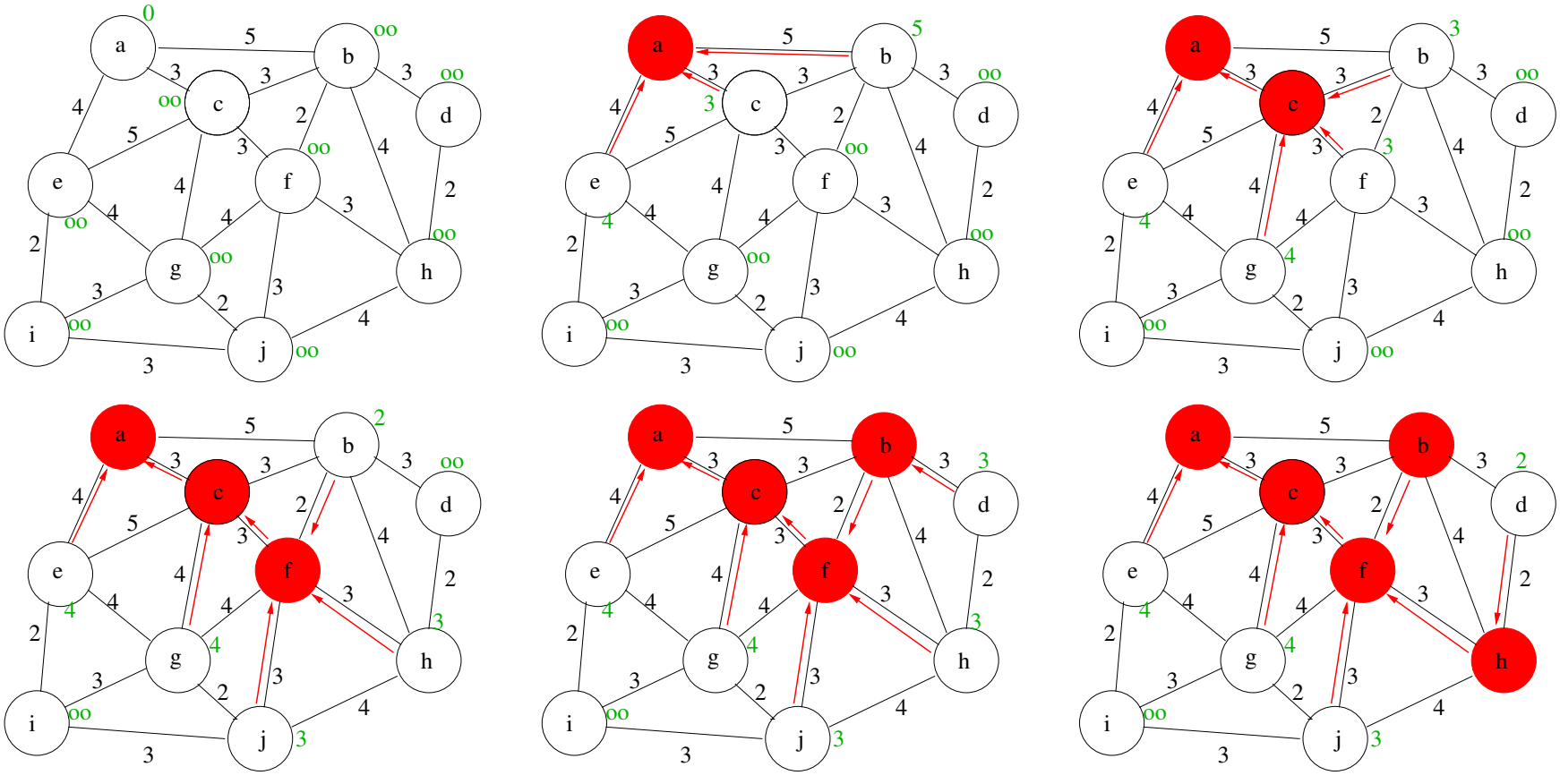
Prim

- Eingabe: G mit Gewichtsfunktion w und Anfangsknoten $r \in V$.
- Initialisiere Q mit allen $v \in V$, mit $key[v] = \infty$.
- Setze $key[r] \leftarrow 0$ und $\pi[r] \leftarrow \text{NIL}$.
- Solange Q nicht leer ist, setze $v \leftarrow \text{Extract-Min}(Q)$:

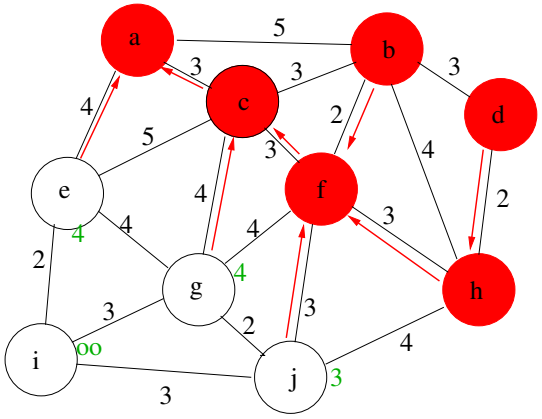
  Für alle Nachbarn $u \in Adj[v]$, die noch in Q sind, und für die $w(\{u,v\}) < key[u]$ ist, setze $\pi[u] \leftarrow v$ und $key[u] \leftarrow w(\{u,v\})$.

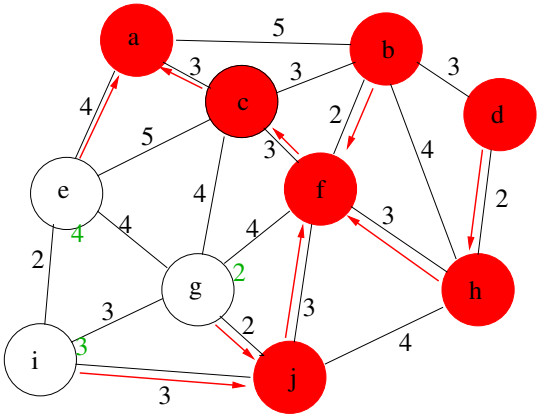
Der Baum A ist implizit gegeben durch $\big\{\, \{v, \pi[v]\} \,;\, v \in (V \setminus Q \setminus \{r\}) \,\big\}$.

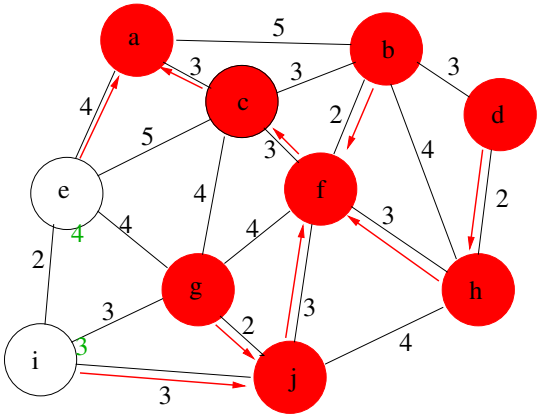
Beispiel für Alg. von Prim

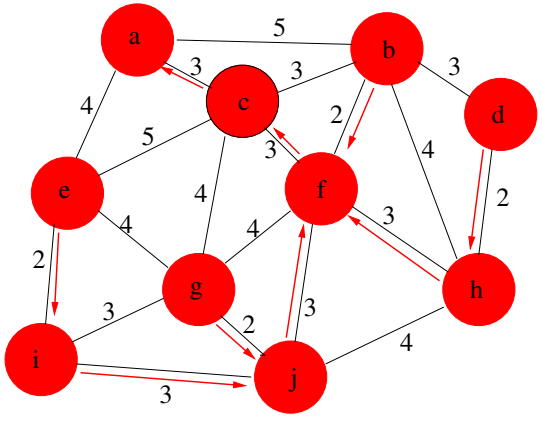


Beispiel









Gewicht des entstehenden Spannbaums: 23 (korrekt)

Korrektheit & Laufzeit des Algorithmus von Prim

- Die Menge A entspricht den Kanten $(\pi[v], v)$, wobei $v \notin Q$, also ein bereits entfernter Knoten ist.
- Die Knoten $v \in Q$ mit $key[v] < \infty$ sind von A aus über eine Kante erreichbar.
- Der Wert $key[v]$ gibt das Gewicht der leichtesten Kante, die v mit A verbindet, an.
- Die π-Zeiger entsprechen jeweils dieser leichtesten Kante.
- Nach dem Lemma ist die π-Kante des jeweils gewählten Knoten sicher. Es folgt, dass zu jedem Zeitpunkt A Teilmenge eines MST ist und zum Schluss ist dann A ein MST.

Laufzeit des Algorithmus mit Q als Heap:

$$O(|V| \log |V|) + O(|E| \log |V|) = O(|E| \log |V|)$$

Jeder Knoten wird einmal abgearbeitet, jede Kante wird zweimal beschritten. Jeweils $O(\log |V|)$ Arbeit für die Heap-Operationen.

Der Algorithmus von Kruskal

Benutzt eine UNION-FIND-Datenstruktur.
Erinnerung: Diese Datenstruktur verwaltet ein System disjunkter Mengen von "Objekten" und bietet folgende Operationen an:

- INIT Initialisieren
- MAKE-SET(x) Fügt eine neue Einermenge mit Inhalt x hinzu.
- FIND(x) Ist x in einer Menge enthalten, so liefere diese in Form eines kanonischen Elementes zurück. Anderenfalls liefere NIL o.ä. zurück. Insbesondere kann man durch den Test FIND(x) = FIND(y) feststellen, ob zwei bereits eingefügte Elemente in derselben Menge liegen.
- UNION(x, y): Sind x und y in zwei verschiedenen Mengen enthalten, so vereinige diese zu einer einzigen. Anschließend gilt also insbesondere FIND(x) = FIND(y).

Beachte: man kann Mengen und Elemente nicht wieder entfernen oder auseinanderreißen.

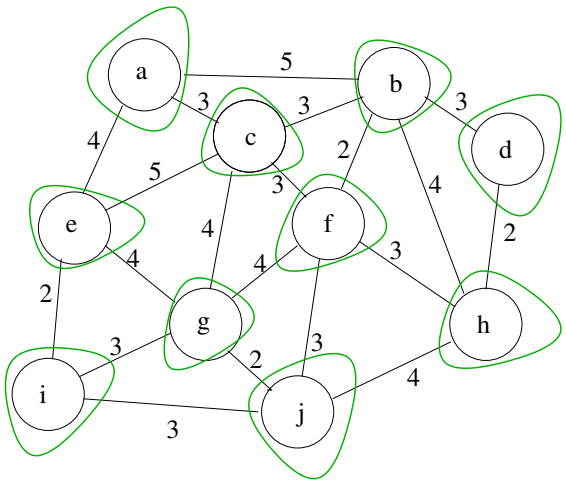
Kruskal's Algorithmus

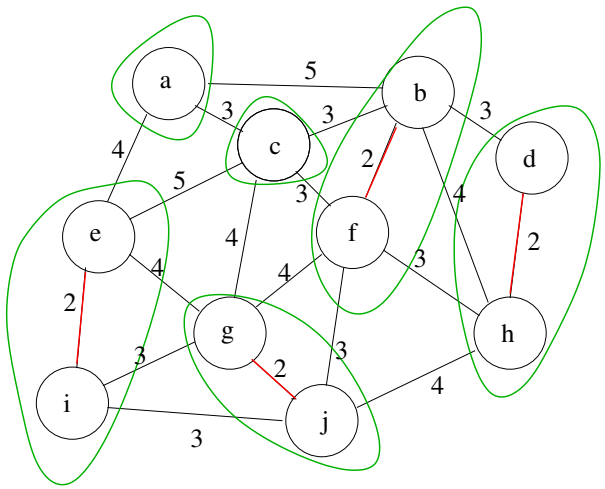
Kruskal

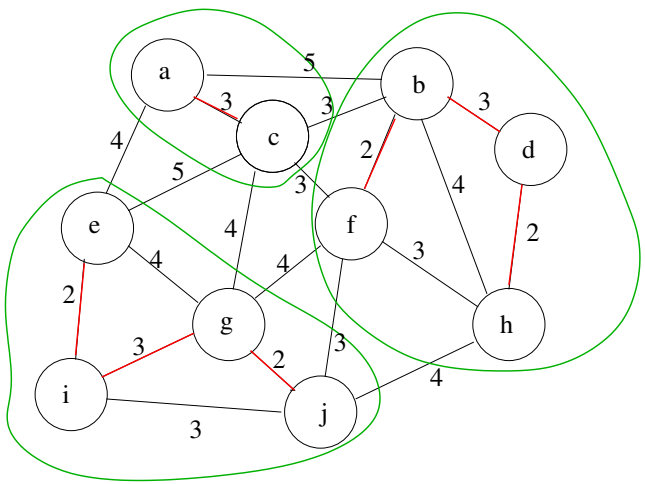
- Setze $A := \emptyset$.
- Rufe $\textsc{Make-Set}(v)$ für jeden Knoten $v \in V$ auf.
- Sortiere die Kanten aufsteigend nach Gewicht.
- Für jede Kante $e = \{u, v\}$, in der sortierten Reihenfolge, prüfe ob $\textsc{Find}(u) \neq \textsc{Find}(v)$.
- Falls ja, füge e zu A hinzu, und rufe $\textsc{Union}(u, v)$ auf, sonst weiter.

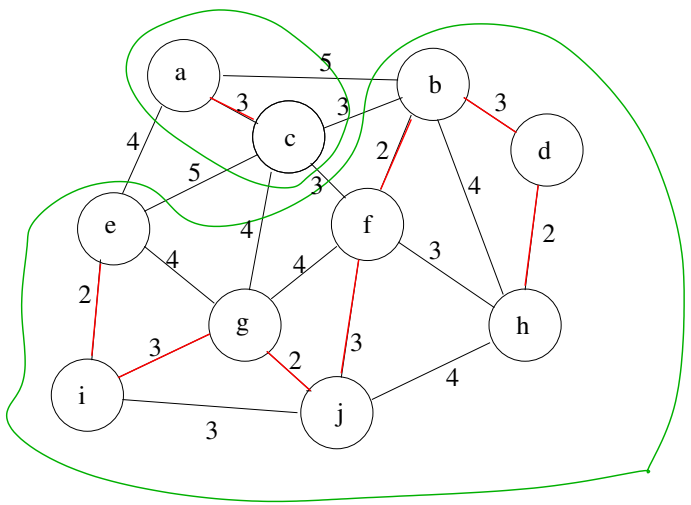
Laufzeit $O(|E| \log |E|)$. Dominiert durch den Sortieraufwand. Reduziert sich auf $O(|E| \log^* |E|)$ falls Kanten schon vorsortiert sind.

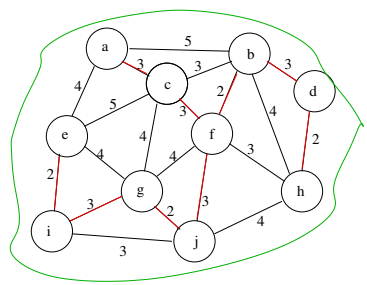
Beispiel


a
b
c
d
e
f
g
h
i
j


a
b
c
d
e
f
g
h
i
j


a
b
c
d
e
f
g
h
i
j


a
b
c
d
e
f
g
h
i
j


a
b
c
d
e
f
g
h
i
j

Korrektheit von Kruskals Algorithmus

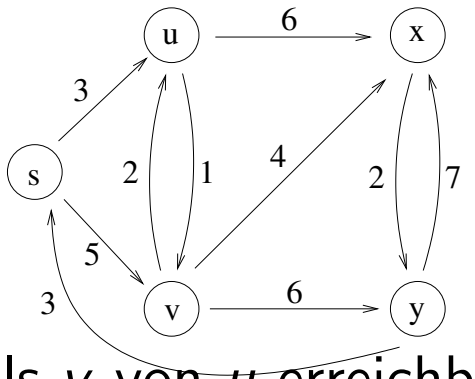
- Die Kantenmenge A ist hier expliziter Bestandteil. Die Invariante lautet also, dass A Teilmenge eines MST ist.
- Es werde zu einem gewissen Zeitpunkt die Kante e gewählt.
- Diese Kante verbindet zwei Zusammenhangskomponenten von A.
- Eine von diesen bezeichnen wir mit S.
- Klar ist, dass diese Menge S von keiner Kante aus A gekreuzt wird.
- Die Kante e hat minimales Gewicht in $E \setminus A$ überhaupt, also insbesondere unter denen, die S kreuzen. Sie ist daher sicher.

Kürzeste Wege

Gegeben: gerichteter Graph $G = (V, E)$ mit Kantengewichten $w : E \to \mathbb{R}$.
Für einen Weg $p : v_0 \rightsquigarrow v_k$

$$p = \langle v_0, \ldots, v_k \rangle \text{ mit } (v_i, v_{i+1}) \in E \text{ für alle } i < k \ .$$

sei das Gewicht des Weges p definiert als:
$w(p) = \sum_{i=1}^{k} w\big((v_{i-1}, v_i)\big)$



Minimaldistanz von u nach v:

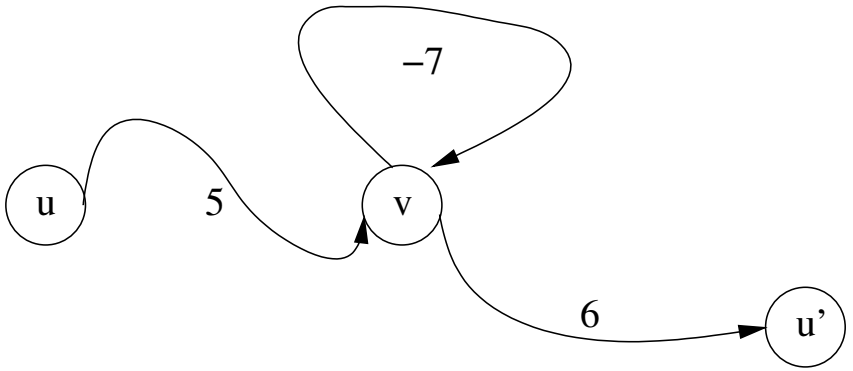
$$\delta(u, v) = \begin{cases} \min\{w(p) \ ; \ p : u \rightsquigarrow v\} & \text{falls } v \text{ von } u \text{ erreichbar ist,} \\ \infty & \text{sonst.} \end{cases}$$

Kürzester Weg von u nach v:

$$\textit{Pfad } p : u \rightsquigarrow v \textit{ mit } w(p) = \delta(u, v).$$

Negative Zyklen und Algorithmen

Zusätzliche Schwierigkeit: Gibt es einen negativen Zyklus $p : v \rightsquigarrow v$ mit $w(p) < 0$, so ist $\delta(u, u')$ nicht wohldefiniert, falls es einen Weg von u nach u' über v gibt.



Algorithmen für kürzeste Wege von einem gegebenen Startpunkt s:

Dijkstra: nimmt an, dass $w(e) \geq 0$ für alle $e \in E$.

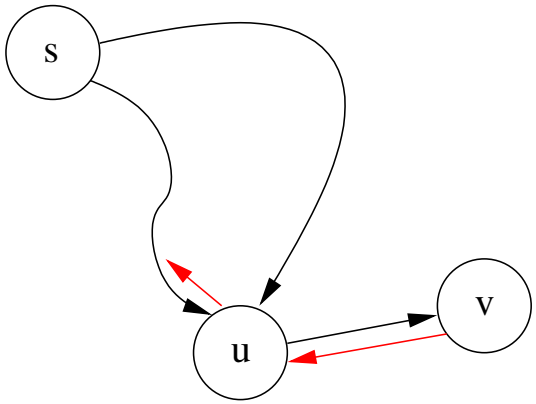
Bellman-Ford: Entdeckt die Präsenz negativer Zyklen und liefert korrekte kürzeste Wege, falls es keinen gibt. Dafür aber aufwendiger als Dijkstra.

Optimale Teillösungen

Ist $p = \langle v_0, \ldots, v_k \rangle$ ein kürzester Weg von v_0 nach v_k, so ist für alle $0 \leq i < j \leq k$ der Pfad

$$p_{ij} = \langle v_i, \ldots, v_j \rangle$$

ein kürzester Weg von v_i nach v_j.



Daher reicht es zur Angabe eines kürzesten Weges von s zu v für alle $v \in V$, für jeden Knoten $v \neq s$ einen Vorgänger $\pi[v]$ anzugeben.

Initialisierung

Initialise

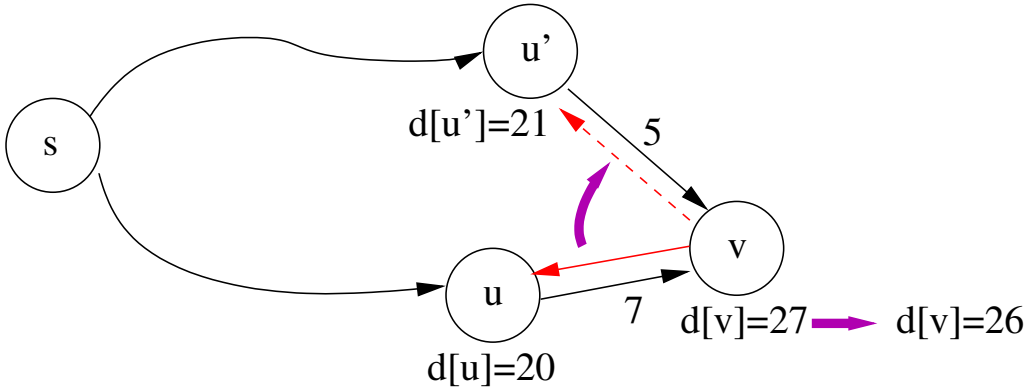
$\textsc{Initialise}(G, s)$:
for $v \in V$ **do**
$d[v] \leftarrow \infty$
$\pi[v] \leftarrow \textsc{nil}$
$d[s] \leftarrow 0$

Relaxierung

Algorithmen halten für jedes $v \in V$ eine Abschätzung $d[v] \geq \delta(s, v)$ und einen vorläufigen Vorgänger $\pi[v]$.

Relax

$\textsc{Relax}(u, v)$
▷testet, ob der bisher gefundene kürzeste Pfad zu v
▷ durch die Kante (u, v) verbessert werden kann
if $d[v] > d[u] + w((u, v))$
then $d[v] \leftarrow d[u] + w((u, v))$
$\pi[v] \leftarrow u$

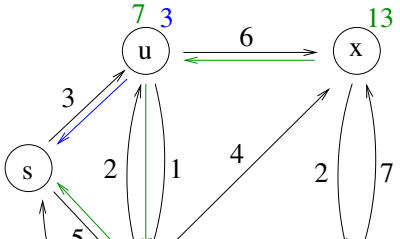


Eigenschaften der Relaxierung

Lemma

Wird für einen Graphen G und $s \in V$ erst $\textsc{Initialize}(G, s)$, und dann eine beliebige Folge von $\textsc{Relax}(u, v)$ für Kanten (u, v) ausgeführt, so gelten die folgenden Invarianten:

1. $d[v] \geq \delta(s, v)$ für alle $v \in V$.
2. Ist irgendwann $d[v] = \delta(s, v)$, so ändert sich $d[v]$ nicht mehr.
3. Gibt es einen kürzesten Pfad von s zu v, der in der Kante (u, v) endet, und ist $d[u] = \delta(s, u)$ vor dem Aufruf $\textsc{Relax}(u, v)$, so ist danach $d[v] = \delta(s, v)$.
4. Ist $d[v] \neq \infty$, so existiert ein Pfad $s \rightsquigarrow \pi[v] \rightarrow v$ der Länge $d[v]$.



Der Algorithmus von Dijkstra

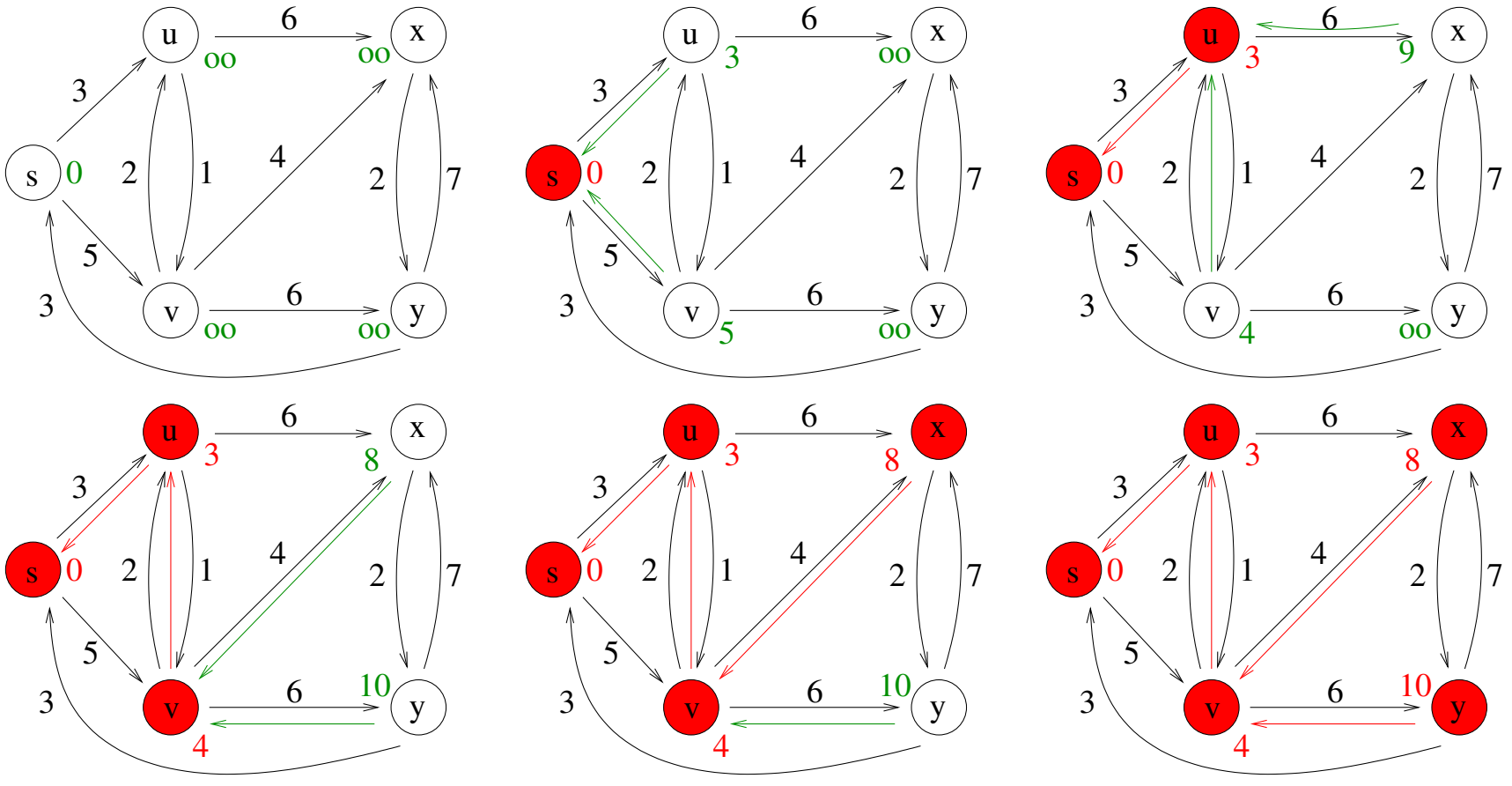
Benutzt eine *priority queue* Q, die Knoten $v \in V$ mit Schlüssel $d[v]$ hält.

Dijkstra Algorithmus

DIJKSTRA(G, w, s)

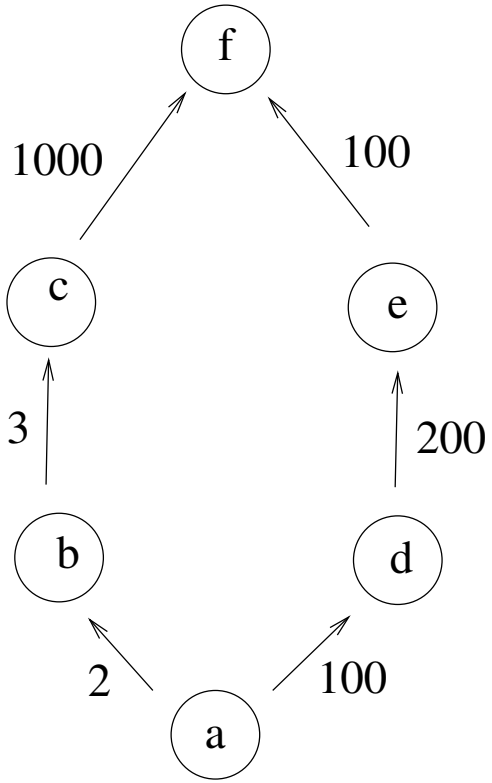
- Rufe INITIALIZE(G, s) auf, setze $Q \leftarrow V$.
- Solange $Q \neq \emptyset$ ist, setze $u \leftarrow$ EXTRACT-MIN(Q).
- Für jedes $v \in Adj[u]$ führe RELAX(u, v) aus.
  Für die Absenkung eines d-Wertes ist DECREASE-KEY zu verwenden.
  Anschließend nächste Iteration.

Beispiel



d- und π-Felder in Grün/Rot. Rot=Permanent.

Weiteres Beispiel



- Knoten werden in der Reihenfolge a,b,c,d,e,f abgefertigt.
- Erst bei der Bearbeitung von e wird $\pi[f]$ auf e gesetzt; bis dahin gilt $\pi[f] = c$.

Korrektheit des Dijkstra Algorithmus

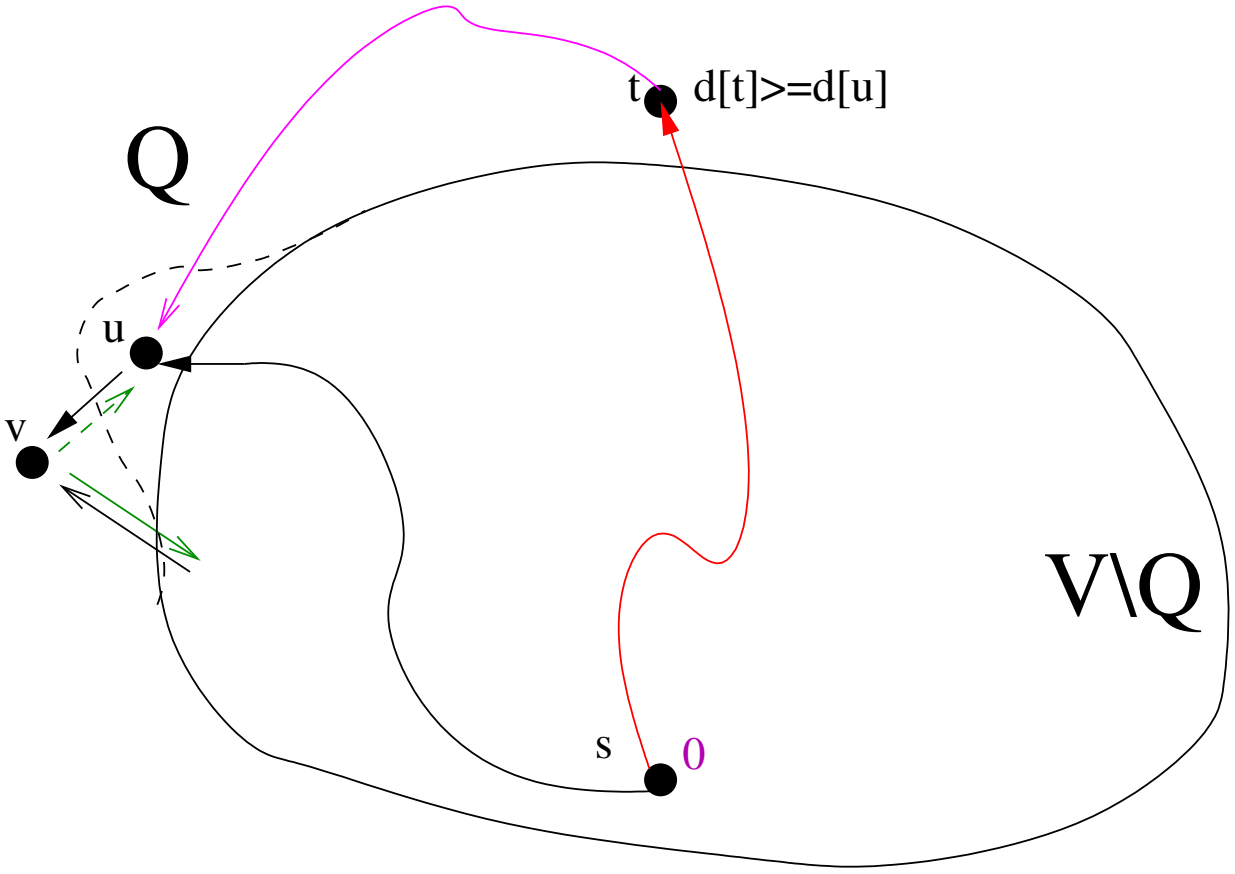
Invariante

$d[u]$ ist für alle Knoten in $V \setminus Q$ korrekt eingetragen; für $v \in Q$ hält $d[v]$ den kürzesten Weg, dessen innere Knoten (also die außer s und v alle in $V \setminus Q$ sind. Die π-Felder realisieren diese kürzesten Wege.

Die Invariante gilt offensichtlich vor der Schleife.

Gilt sie unmittelbar vor einem Schleifen*durchlauf*, so gilt sie auch nachher aus folgendem Grund:
Sei $u = \textsc{Extract-Min}(Q)$ der gewählte Knoten. Der kürzeste Pfad nach u, der nur innere Knoten aus $V \setminus Q$ ist kürzester Pfad überhaupt, denn jeder Pfad nach u muss irgendwann $V \setminus Q$ verlassen und verläuft dann durch einen Knoten mit schlechterem d-Wert (vgl. Korrektheit der Breitensuche). Hier geht die Voraussetzung: "keine negativen Kantengewichte" ein.

Beweis der Invarianten, Forts.



Beliebiger anderer Pfad nach u in rot/lila, verlässt $V \setminus Q$ bei t.
$d[u] \leq d[t] \leq$ Länge des roten Anteils. Länge des lila Anteils ≥ 0.
Durch Relaxierung wird die Invariante für Nachbarn v wiederhergestellt.

Korrektheit und Laufzeit des Dijkstra Algorithmus

Die Invariante gilt also insbesondere, wenn $Q = \emptyset$ und es sind dann alle Wege richtig eingetragen.

Laufzeit

Hängt von der Realisierung der queue Q ab.

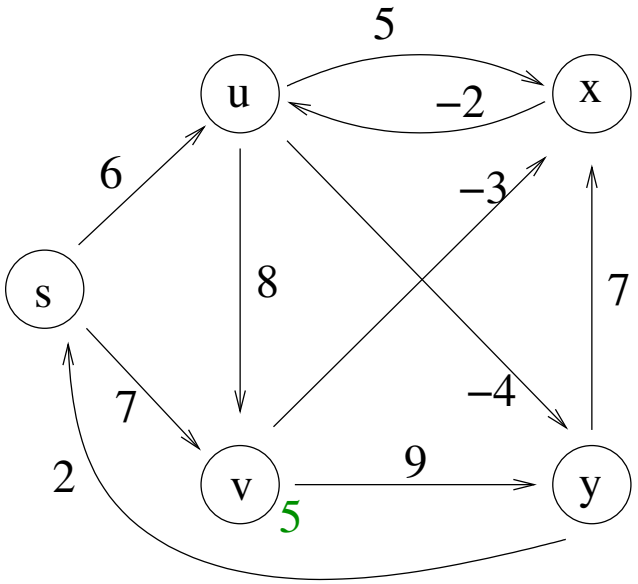
Als Liste: $O(|V|^2)$ *Als Heap:* $O(|E| \log |V|)$
Als Fibonacci-Heap (s. Cormen*):*
$O(|V| \log |V| + |E|)$.

Zusammenfassung Dijkstra:

- Kürzeste Wege in gerichteten Graphen mit nichtnegativen Kantengewichten.
- Sukzessive Relaxierung aller Nachbarn. Jeweils der Knoten mit aktuell kürzestem Abstand wird gewählt.

Negative Kantengewichte

Ein Graph mit negativen Kantengewichten, aber keinen negativen Zyklen.



Negative Zyklen können für Arbitragegeschäfte eingesetzt werden:

- Knoten = Währungen (EUR, GBP, USD, SFR, JPY, . . .)
- Kantengewichte = log(Umtauschkurs)
- Also: Pfadgewichte = log(Kurs bei entsprechender Kette von Umtäuschen)
- Negativer Zyklus: Gelegenheit für Arbitrageschäft.

Der Algorithmus von Bellman-Ford

Bellman-Ford

BELLMAN-FORD(G, w, s)

- Rufe INITIALIZE(G, s) auf.
- Wiederhole $|V| - 1$ mal:

  Für jede Kante $(u, v) \in E$ rufe RELAX(u, v) *auf.*

- Für jede Kante $(u, v) \in E$, teste ob $d[v] > d[u] + w(u, v)$ ist.
- Falls ja für eine Kante, drucke "*negativer Zyklus vorhanden*", sonst brich mit Erfolg ab.

Korrektheit: Nach Ausführung von BELLMAN-FORD(G, w, s) gilt:

Ist kein negativer Zyklus von s erreichbar, dann ist $d[v] = \delta(s, v)$ für alle $v \in V$, und der Algorithmus terminiert erfolgreich. Andernfalls ist wird der negative Zyklus auch entdeckt.

Beweis der Korrektheit

Folgende Invariante ergibt sich leicht aus den Eigenschaften der Relaxierung.

Lemma

Egal, ob negative Zyklen da sind oder nicht, ist $d[v]$ nach der k-ten Iteration kleiner oder gleich der Länge des kürzesten Pfades, der aus höchstens k Kanten besteht.

Gibt es keine negativen Zyklen, so bestehen kürzeste Pfade aus höchstens n Kanten, aus den allgemeinen Eigenschaften der Relaxierung folgt dann die Korrektheit.
Wird erfolgreich abgebrochen, dann würde eine weitere Iteration der Schleife über $|V| - 1$ hinaus nichts neues bringen; es kann daher aufgrund des Lemmas keinen negativen Zyklus geben.

Die Komplexität ist offenbar $O(|V| \cdot |E|)$.

Kürzeste Wege zwischen allen Paaren

Aufgabe: Berechne $\delta(i,j)$ für alle Paare $i,j \in V = \{1,\ldots,n\}$.

Kantengewichte in Matrix $W = (w_{i,j})$, mit $w_{i,i} = 0$.
Wir nehmen jeweils vereinfachend an, dass es keine negativen Zyklen gibt.

Anwendungen:

- Transitive Hülle
- Grafische Repräsentation von Daten
- Routenfindung und Maximale Bandbreite in Netzwerken

Lösung mit Dynamischer Programmierung

Berechne systematisch die Werte

$d_{i,j}^{(m)}$ = *minimales Gewicht eines Weges von* i *zu* j*, der* $\leq m$ *Kanten lang ist.*

$$d_{i,j}^{(0)} = \begin{cases} 0 & \text{falls } i = j \\ \infty & \text{sonst} \end{cases}$$

$$d_{i,j}^{(m)} = \min\left(d_{i,j}^{(m-1)}, \min_{k \neq j}\left(d_{i,k}^{(m-1)} + w_{k,j}\right)\right)$$

$$= \min_{1 \leq k \leq n}\left(d_{i,k}^{(m-1)} + w_{k,j}\right)$$

Keine negativen Zyklen $\rightsquigarrow$ $\delta(i,j) = d_{i,j}^{(n-1)} = d_{i,j}^{(m)}$ für alle $m \geq n-1$.

Kürzeste Wege und Matrizenmultiplikation

Betrachte Matrizen $D^{(m)} = \left(d_{i,j}^{(m)}\right)$. Es gilt

$$D^{(m)} = D^{(m-1)} \odot W$$

wobei $\odot$ eine Art Multiplikation ist mit $\min \mathrel{\hat{=}} \sum$ und $+ \mathrel{\hat{=}} \times$.

Matrix $D^{(n-1)} = \left(\delta(i,j)\right)$ kann ausgerechnet werden in Zeit $\Theta(n^4)$.

Bessere Methode durch iteriertes Quadrieren:

Da für $m \geq n-1$ gilt $D^{(m)} = D^{(n-1)}$, und $\odot$ assoz. ist, berechne $D^m = D^{(n-1)}$ für $m = 2^{\lceil \log(n-1) \rceil}$ mittels

$$D^{(1)} = W$$
$$D^{(2k)} = D^{(k)} \odot D^{(k)}$$

Zeitkomplexität: nur $\Theta(n^3 \log n)$.

Der Algorithmus von Floyd-Warshall

Betrachte Weg von i nach j:

$$\langle i = v_0, v_1, \ldots, v_{\ell-1}, v_\ell = j \rangle$$

Knoten $v_1, \ldots, v_{\ell-1}$ sind die Zwischenknoten.

Dynamische Programmierung: Berechne systematisch die Werte

$d_{i,j}^{(k)}$ = *minimales Gewicht eines Weges von i zu j, der nur Zwischenknoten $\{1, \ldots, k\}$ verwendet.*

$d_{i,j}^{(0)} = w_{i,j}$

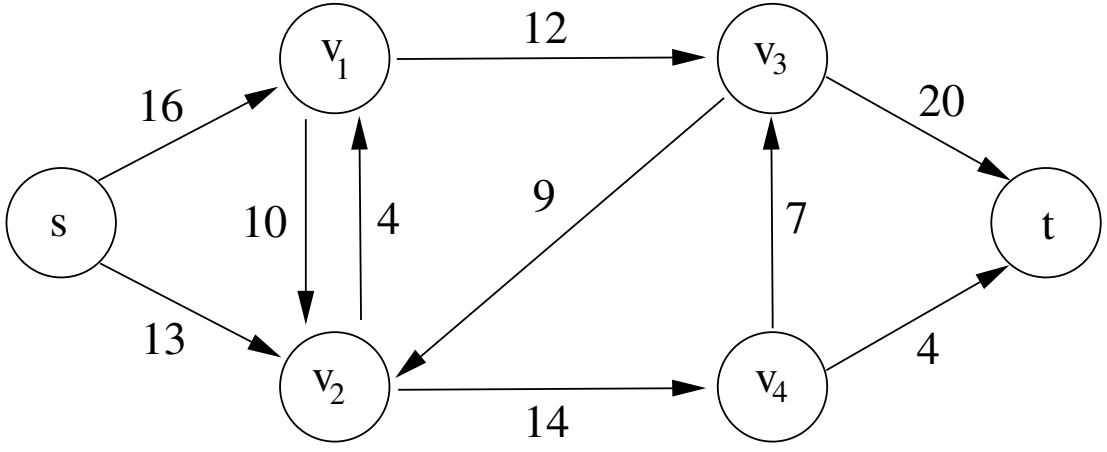
$d_{i,j}^{(k)} = \min\big(d_{i,j}^{(k-1)}, gcd(X^7 + 2 * X^6 - X^4 - 3 * X^3 - 3 * X^2 - 2 * X, X$

Klar: $\delta(i,j) = d_{i,j}^{(n)}$.

Matrix $D^{(n)} = \big(d_{i,j}^{(n)}\big) = \big(\delta(i,j)\big)$ kann in Zeit $\Theta(n^3)$ berechnet werden.

Flüsse in Netzwerken

Gegeben: gerichteter Graph $G = (V, E)$ mit Quelle $s \in V$ und Senke $t \in V$, für $(u, v) \in E$ Kapazität $c(u, v) \geq 0$. Für $(u, v) \notin E$ sei $c(u, v) = 0$.



Gesucht: Ein möglichst großer Fluss durch das Netzwerk von s nach t, der aber nirgends die Kapazität überschreitet.

Beispiel: 10 durch die Kanten $(s, v_1), (v_1, v_3), (v_3, t)$ und weitere 10 durch die Kanten $(s, v_2), (v_2, v_4)$. Dort Aufteilung in 4 durch (v_4, t) und 6 durch (v_4, v_2) und (v_2, t). Macht 20 insgesamt.

Flüsse formal

Gesucht: Ein Fluss durch G: Funktion $f : V \times V \to \mathbb{R}$ mit

1. $f(u, v) \leq c(u, v)$
2. $f(u, v) = -f(v, u)$
3. Für alle $u \in V \setminus \{s, t\}$:

$$\sum_{v \in V} f(u, v) = 0$$

Wert des Flusses f

$$|f| := \sum_{v \in V} f(s, v)$$

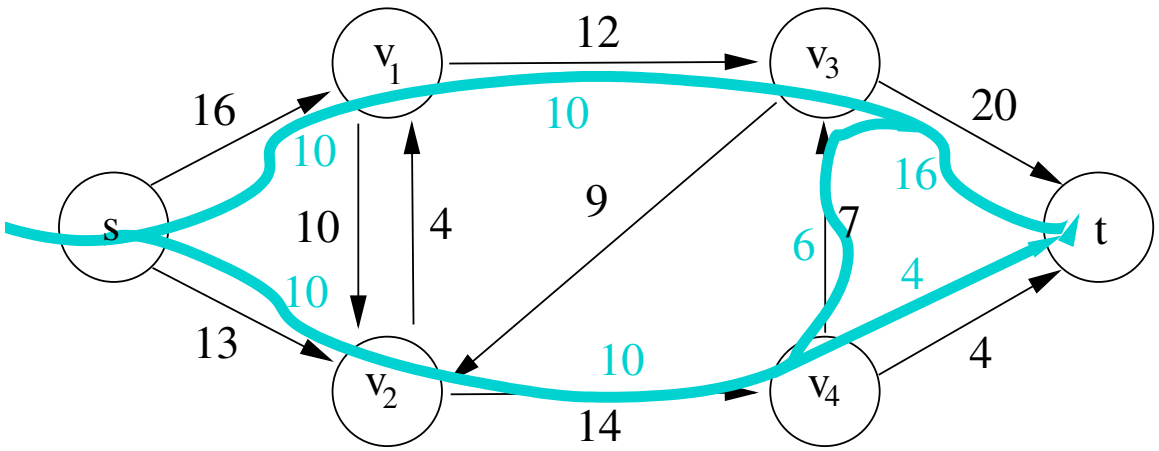
soll maximiert weden.

Grundanwendung: Kapazität von Pipeline-Netzwerken / Stromnetzen, ökologische Modelle, etc.
Abgeleitete Anwendungen: Disjunkte Pfade in Graphen, Bildanalyse.

Notation für Flüsse

Für $X, Y \subseteq V$ sei $f(X, Y) := \sum_{x \in X} \sum_{y \in Y} f(x, y)$.

Abkürzung: $f(v, X) = f(\{v\}, X)$. Eigenschaft 3 lautet damit: $f(u, V) = 0$.



Hier ist z.B.: $f(s, v_1) = 10 = f(s, v_2)$ und $f(s, v_3) = f(s, v_4) = f(s, s) = f(s, t) = 0$. Also $f(s, V) = 20$ (Wert des Flusses).
Außerdem: $f(v_3, v_1) = -10, f(v_3, v_4) = -6, f(v_3, t) = 16$. Also $f(v_3, V) = 0$.

Flusseigenschaften

Lemma

Für alle $X, Y, Z \subseteq V$ mit $Y \cap Z = \emptyset$ gilt:

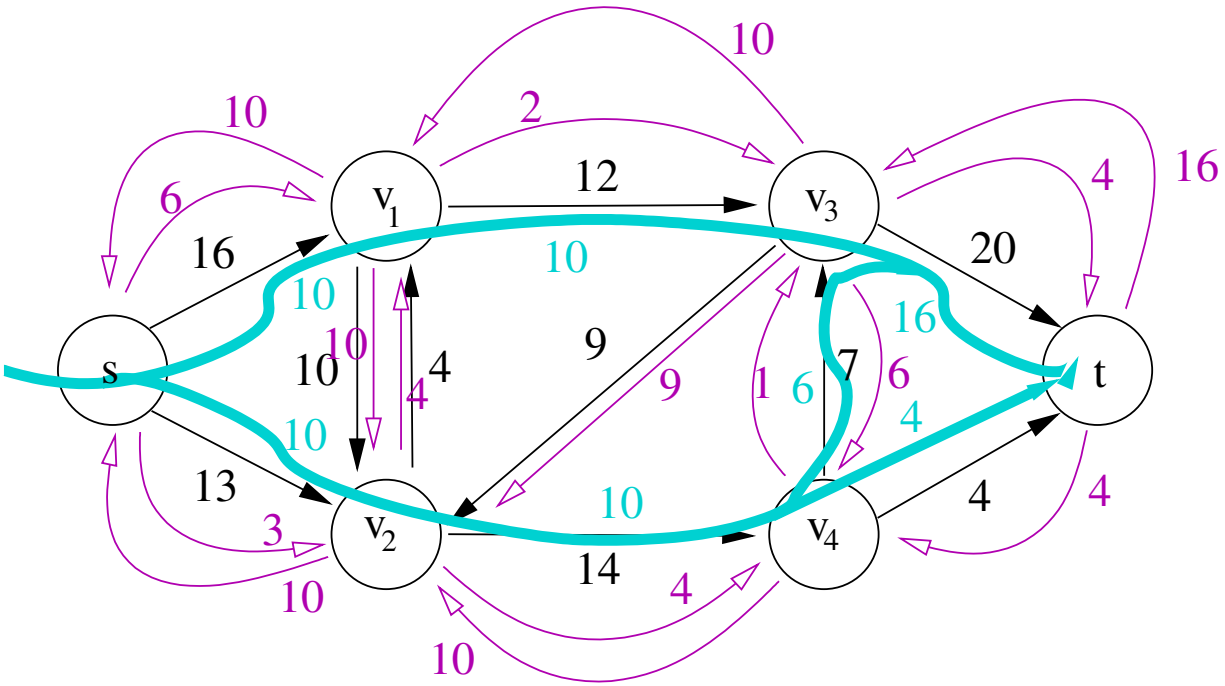
- $f(X, X) = 0$
- $f(X, Y) = -f(Y, X)$
- $f(X, Y \cup Z) = f(X, Y) + f(X, Z)$
- $f(Y \cup Z, X) = f(Y, X) + f(Z, X)$

Folgt direkt aus der Definition, z.B.:
$f(X, X) = \sum_{x \in X} \sum_{y \in X} f(x, y) =$
$\sum_{x \in X} f(x, x) + \sum_{x,y \in X \; x<y} f(x, y) + f(y, x) = 0$

Restnetzwerke und Erweiterungspfade

Sei f ein Fluss in einem Netzwerk $G = (V, E)$ mit Kapazität c.
Für $u, v \in V$ ist die Restkapazität $c_f(u, v) = c(u, v) - f(u, v)$.
Das Restnetzwerk $G_f = (V, E_f)$ ist gegeben durch

$$E_f := \{(u, v)\,;\; c_f(u, v) > 0\}\ .$$



Z.B.: $c_f(v_3, v_1) = c(v_3, v_1) - f(v_3, v_1) = 0 - (-10) = 10$

Addition von Flüssen

Lemma

Ist f' ein Fluss in G_f, so ist $f + f'$ ein Fluss in G mit Wert $|f| + |f'|$.

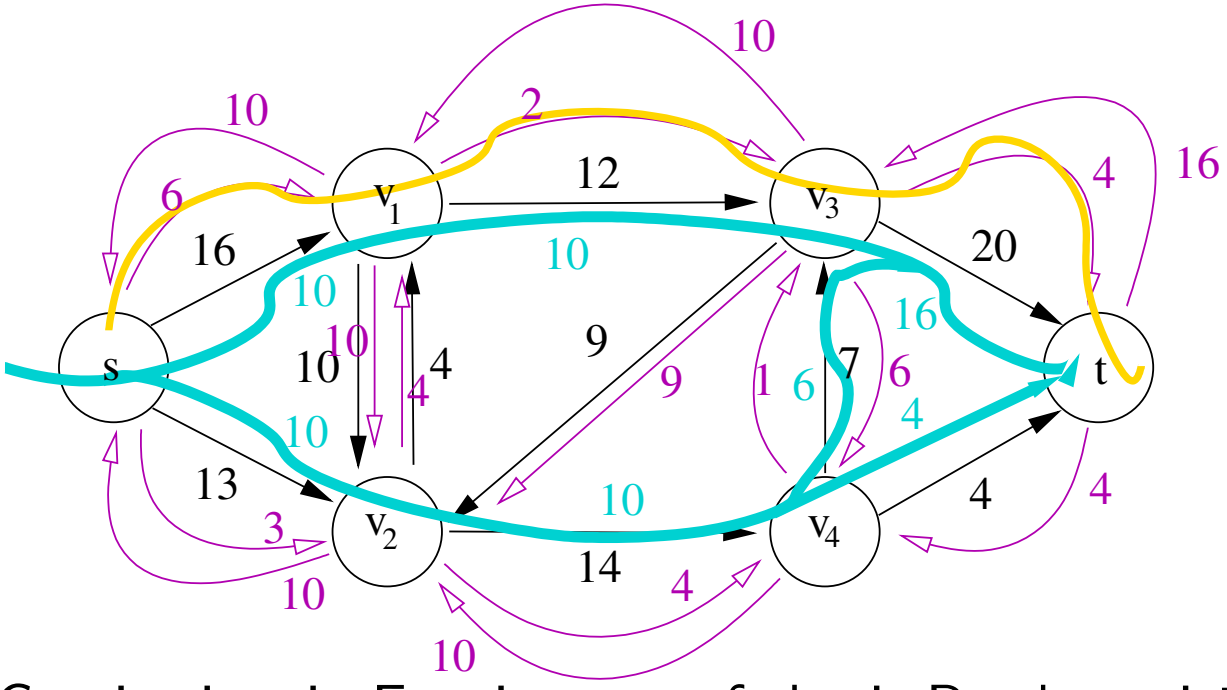
Bedingung für f: $f(u, v) \leq c(u, v)$
Bedingung für f': $f(u, v) \leq c_f(u, v) = c(u, v) - f(u, v)$
Also: $f(u, v) + f'(u, v) \leq f(u, v) + c(u, v) - f(u, v) = c(u, v)$.
Außerdem: $(f + f')(s, V) = f(s, V) + f'(s, V)$

Erweiterungspfad

Definition

Ein Weg $p : s \rightsquigarrow t$ im Restnetzwerk G_f eines Flusses f ist ein Erweiterungspfad (des Flusses), seine Restkapazität ist

$$c_f(p) = \min\{c_f(u, v) \; ; \; (u, v) \text{ Kante in } p\}$$



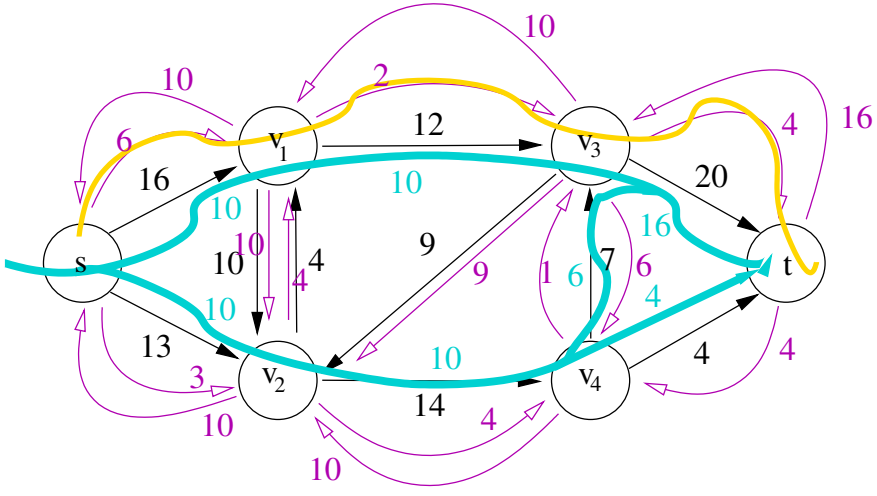
Gezeigt ist ein Erweiterungspfad mit Restkapazität 2.

Zu Erweiterungspfad gehöriger Fluss

Definition

Für einen Erweiterungspfad p definiere einen Fluss f_p in G_f:

$$f_p(u,v) = \begin{cases} c_f(p) & (u,v) \text{ in } p \\ -c_f(p) & (v,u) \text{ in } p \\ 0 & \text{sonst} \end{cases}$$



Hier ist $f_p(s, v_1) = f_p(v_1, v_3) = f_p(v_3, t) = 2$ und $f_p(v_1, s) = f_p(v_3, v_1) = f_p(t, v_3) = -2$ und $f_p(-,-) = 0$, sonst.

Das Max-Flow-Min-Cut Theorem

Definition

Ein Schnitt in G ist eine Zerlegung (S, T) mit $s \in S \subseteq V$ und $t \in T = V \setminus S$.

Satz

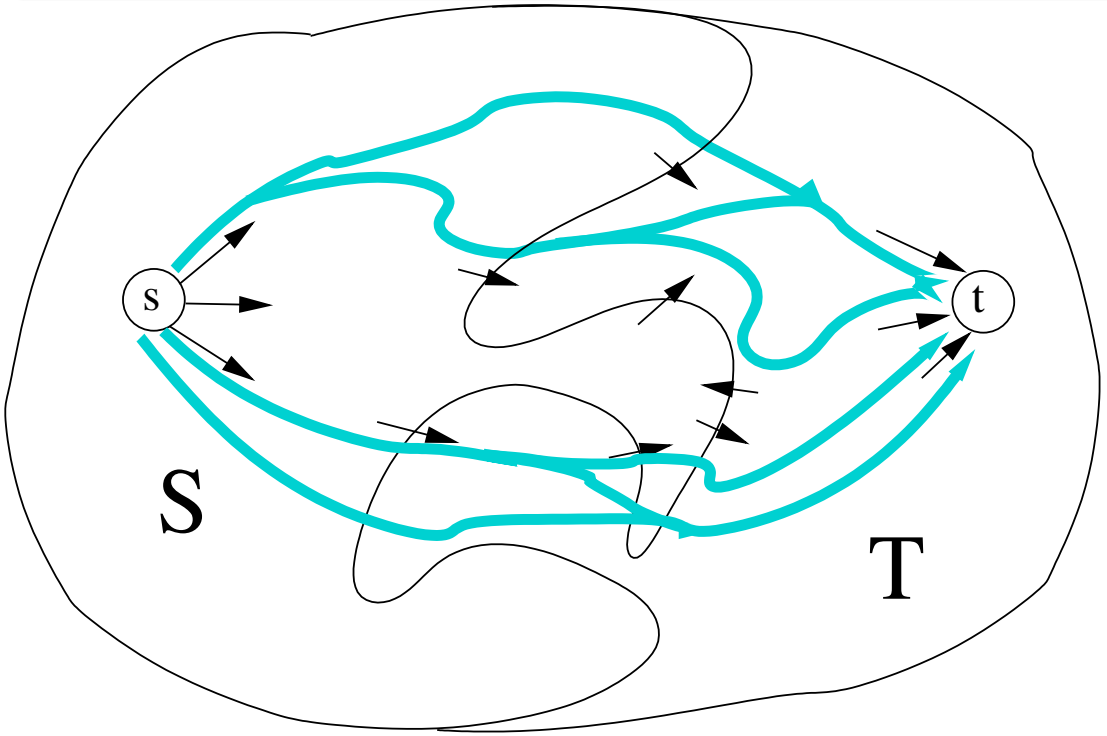
Die folgenden Aussagen sind äquivalent:

1. f ist ein maximaler Fluss in G.
2. Im Restnetzwerk G_f gibt es keinen Erweiterungspfad.
3. Es gibt einen Schnitt (S, T) mit $|f| = c(S, T)$.

Fluss muss durch den Schnitt hindurch

Lemma

Ist (S, T) ein Schnitt, so ist $f(S, T) = |f|$.



$f(S, T) = f(S, T) + f(S, S) = f(S, V) = f(s, V) + f(S - s, V) = f(s, V) = |f|$.

NB: $x \neq s \Rightarrow f(x, V) = 0$.

Max-Flow-Min-Cut, $1 \Rightarrow 2$

$1 \Rightarrow 2$

Ist f maximaler Fluss in G, so gibt es im Restnetzwerk keinen Erweiterungspfad.

Wäre p ein Erweiterungspfad, so wäre $f + f_p$ ein Fluss mit größerem Wert als f.

Max-Flow-Min-Cut, $3 \Rightarrow 1$

$3 \Rightarrow 1$

Ist f ein Fluss und (S, T) ein Schnitt mit $|f| = c(S, T)$, so ist f maximal.

Für jeden Fluss f' gilt $|f'| \leq c(S, T)$, denn $|f'| = f'(S, T) \leq c(S, T)$. Ersteres wg. Lemma, letzteres ist eine Bedingung für Flüsse.

Max-Flow-Min-Cut, $2 \Rightarrow 3$

$2 \Rightarrow 3$

Gibt es im Restnetzwerk G_f keinen Erweiterungspfad, so existiert ein Schnitt (S, T) mit $|f| = c(S, T)$.

Wir setzen $S = \{v \mid s \rightsquigarrow v \text{ in } G_f\}$ und $T = S \setminus V$.
Nach Voraussetzung ist $t \notin S$ also liegt ein Schnitt vor.
Nach dem Lemma ist $|f| = f(S, T)$. Für $u \in S$ und $v \in T$ muss gelten $f(u, v) = c(u, v)$, denn sonst gäbe es eine Kante $(u, v) \in G_f$ und also wäre dann mit u auch v von s erreichbar in G_f.
Also ist $|f| = f(S, T) = c(S, T)$.

Die Ford-Fulkerson-Methode

Ford-Fulkerson Methode

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

- Initialisiere $f(u, v) = 0$ für alle $u, v \in V$.
- Solange es einen Erweiterungspfad p in G_f gibt:

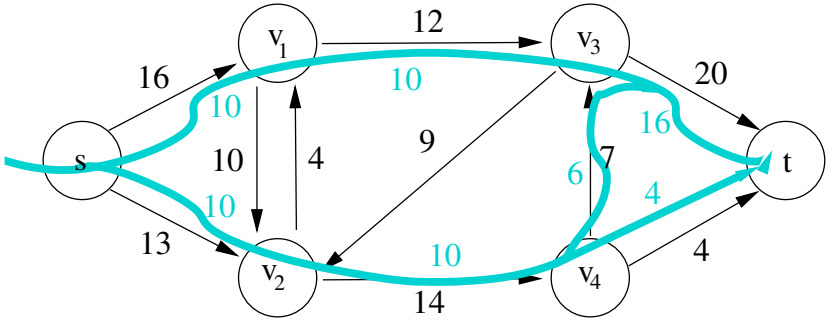
Setze für jede Kante (u, v) in p

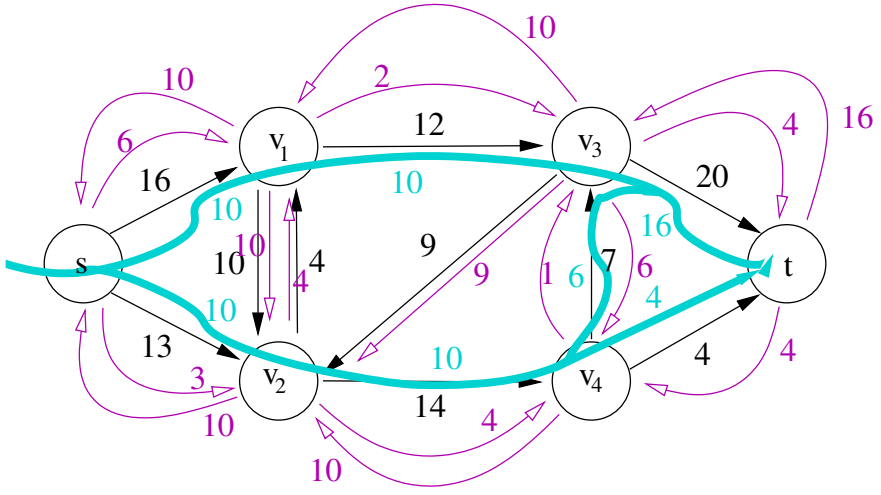
$$f(u, v) \leftarrow f(u, v) + c_f(p) \quad ; \quad f(v, u) \leftarrow -f(u, v)$$

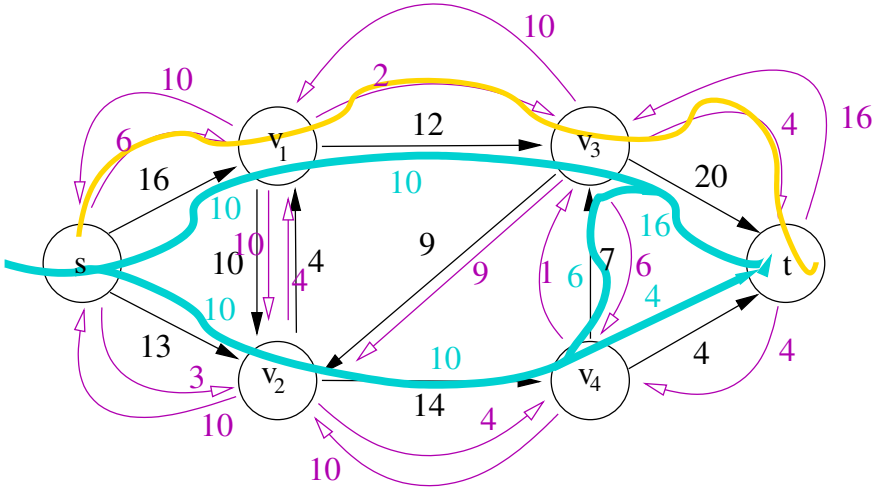
Korrektheit folgt aus dem Max-Flow-Min-Cut-Theorem.
Komplexität hängt davon ab, wie man nach Erweiterungspfaden sucht.

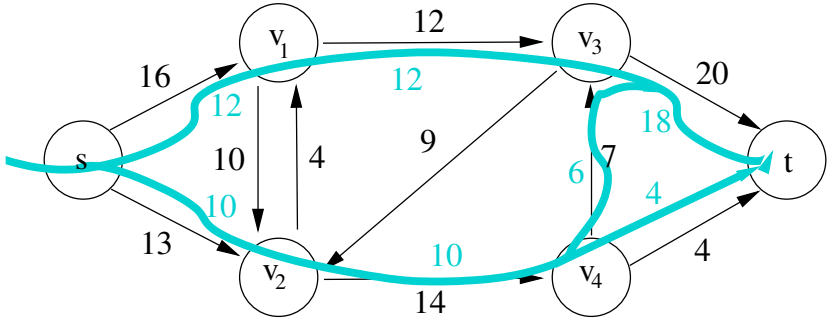
Ist $c(x, y) \in \mathbb{N}$ für alle $(x, y) \in E$, so ist die Laufzeit $O(|E| \cdot |f^*|)$, für einen maximalen Fluss f^*.

Beispiel

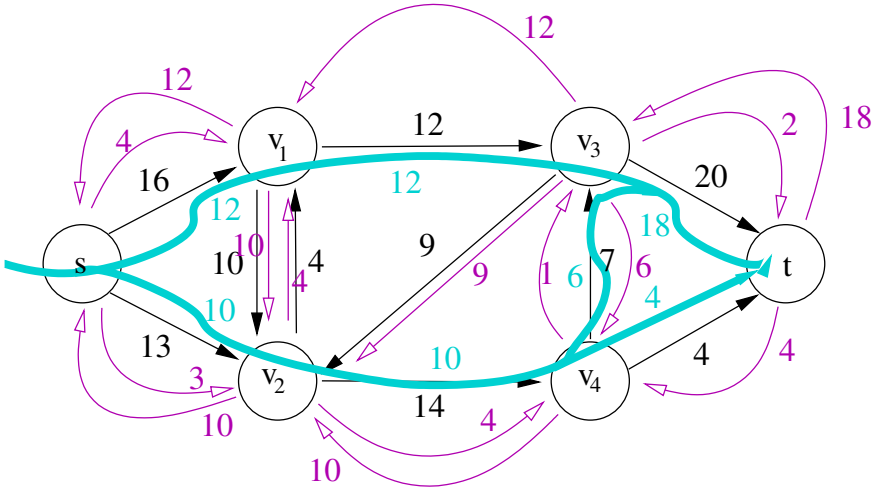

s
v1
v2
v3
v4
t
16
13
12
20
14
10
9
4
7
6
16
4

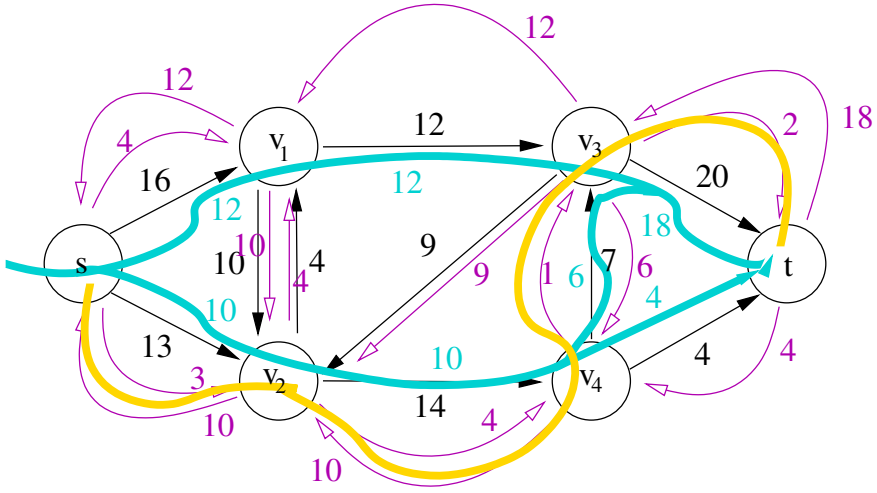

s
v1
v2
v3
v4
t
16
13
12
20
14
10
9
4
7
6
16
4
2
3

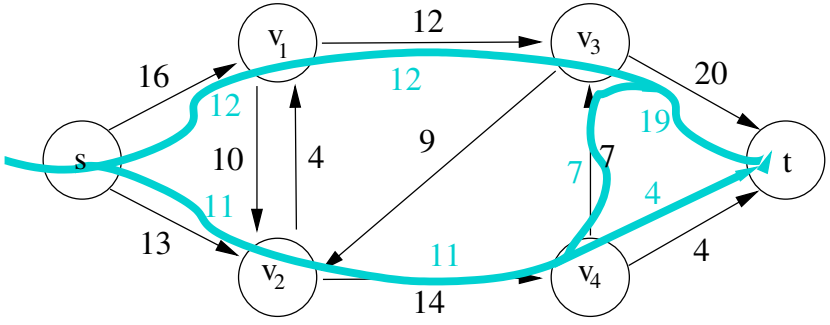

s
v1
v2
v3
v4
t
16
13
12
20
14
10
9
4
7
6
16
4
2
3

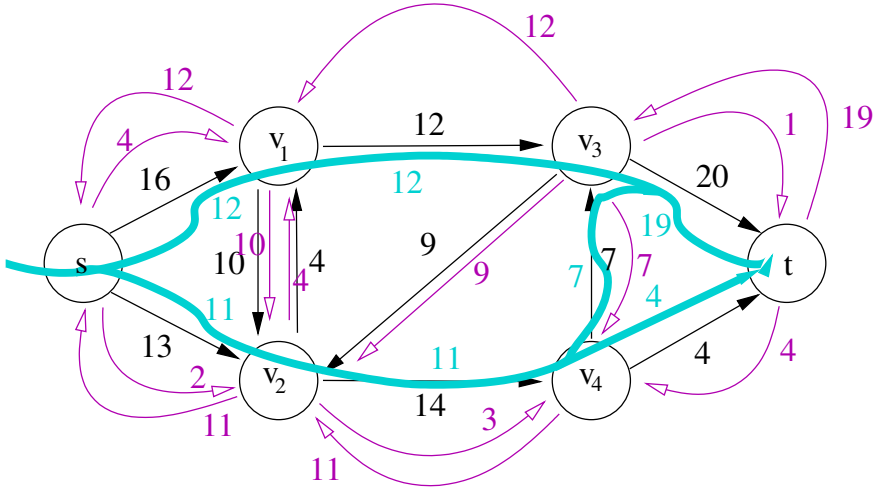

s
v1
v2
v3
v4
t
16
13
12
20
14
10
9
4
7
6
18
12

Beispiel, Forts.

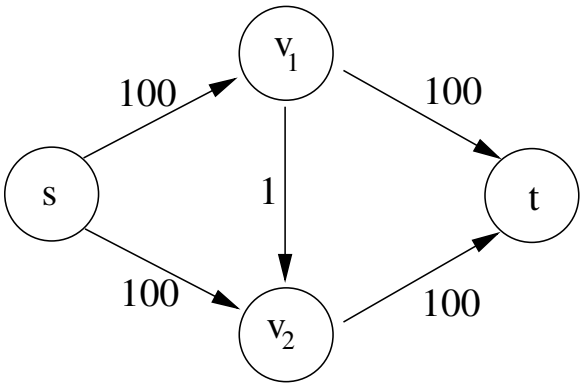


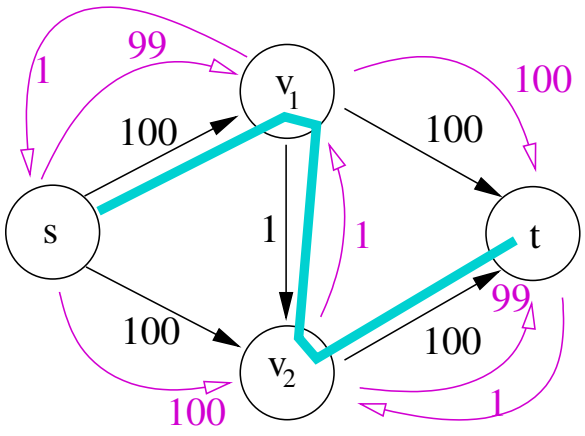


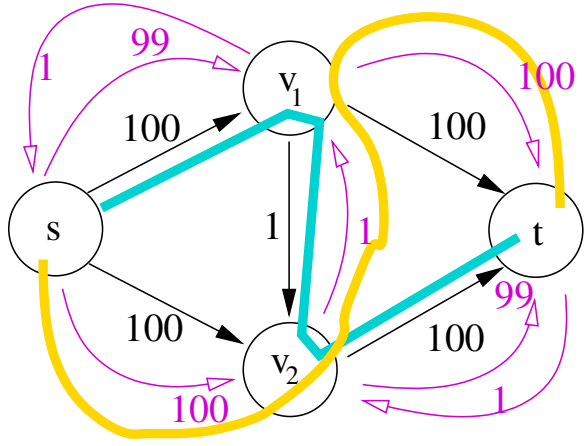


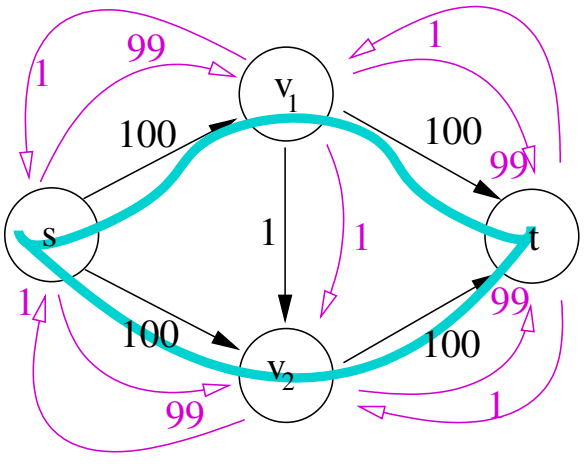


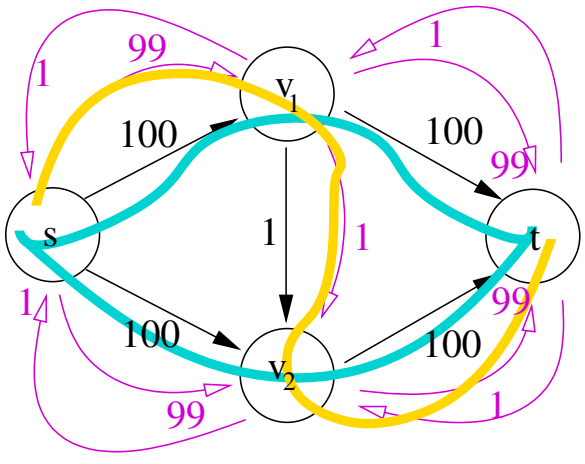
Beispiel für schlechte Laufzeit

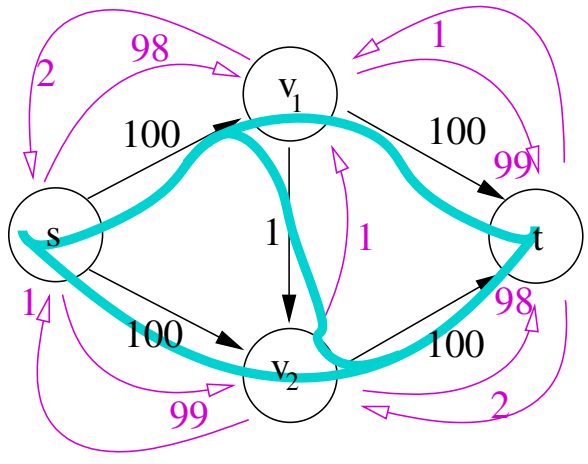




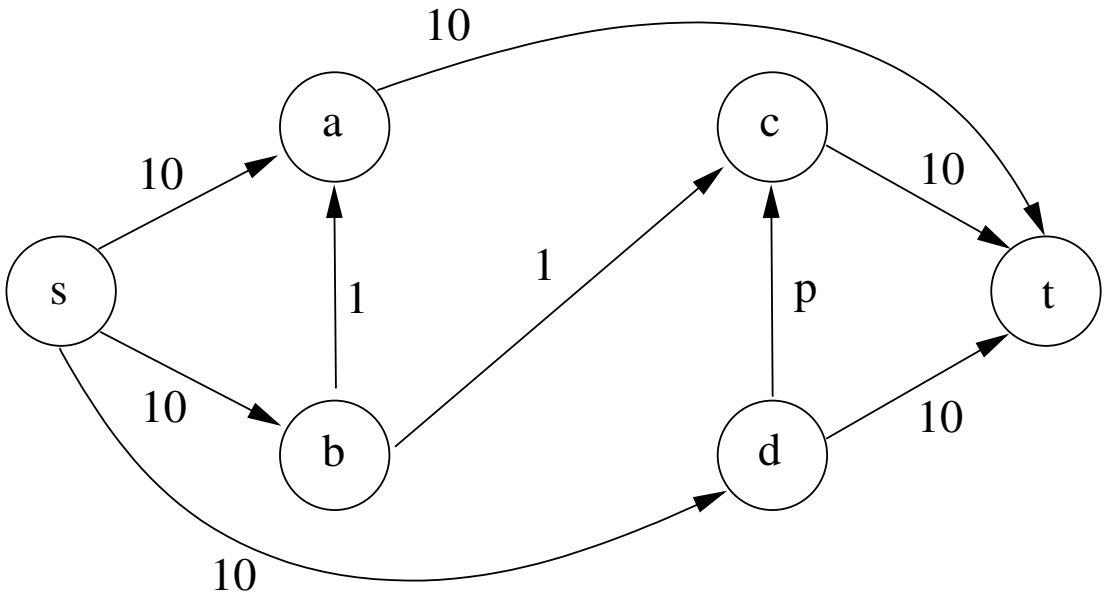








Irrationale Kantengewichte nach U. Zwick



Hier ist p die kleinere Lösung von $p^2 = 1 - p$, also $p = \frac{\sqrt{5}-1}{2} = 0.6\ldots$.

Der Algorithmus von Edmonds-Karp

Edmonds-Karp

Algorithmus von Edmonds-Karp:

> *Wähle bei Ford-Fulkerson immer den kürzestmöglichen Erweiterungspfad.*
> *Auffindbar z.B. mit Breitensuche in G_f.*

Für $v \in V$, sei $\delta_f(s, v)$ die Distanz von s zu v in G_f.

Idee hinter Edmonds-Karp:

Distanz wird größer

Lemma

Sei f' eine Erweiterung des Flusses f durch Hinzunahme des Flusses entlang einem kürzesten Erweiterungspfad im Restnetzwerk G_f. Es gilt $\delta_f(s,v) \leq \delta_{f'}(s,v)$ für alle v.

Beweis durch Induktion über $\delta_{f'}(s,v)$. Ist $\delta_{f'}(s,v) = \infty$, so folgt die Behauptung. Ist $\delta_{f'}(s,v) = 0$, so ist $v = s$ und $\delta_f(s,v) = 0$. Ist schließlich $\delta_{f'}(s,v) > 0$, so fixiere einen kürzesten Pfad von s nach v in $G_{f'}$. Sei u der Knoten unmittelbar vor v.
Falls $(u,v) : E_f$, so gilt
$\delta_{f'}(s,v) = \delta_{f'}(s,u) + 1 \geq \delta_f(s,u) + 1$ wg. Ind.Hyp. und dann $\cdots \geq \delta_f(s,v)$ mit Dreiecksungleichung.
Ist (u,v) nicht in E_f, aber in $E_{f'}$, dann muss der Erweiterungspfad von s nach u in G_f über die Kante (v,u) gelaufen sein und dieser ist gleichzeitig ein kürzester Pfad nach Voraussetzung. Es ist dann $\delta_f(s,v) < \delta_f(s,u) \leq \delta_{f'}(s,u) < \delta_{f'}(s,v)$.

Laufzeit von Edmonds-Karp

Satz

Die Zahl der Iterationen der äußeren Schleife beim Algorithmus von Edmonds-Karp ist $O(|V| \cdot |E|)$.

Sei $f_0, f_1, f_2, \ldots$ die Folge der Flüsse bei der Abarbeitung des Edmonds-Karp Algorithmus. $f_0 = 0$.
Kante (u, v) ist zur Zeit i kritisch, wenn (u, v) in G_{f_i} vorkommt und $f_{i+1}(u, v) - f_i(u, v) = c_{f_i}(u, v)$, also die vorhandene Kapazität voll ausschöpft.

Es folgt: $f_{i+1}(u, v) - f_i(u, v) = c(u, v) - f_i(u, v)$, also $f_{i+1}(u, v) = c(u, v)$.

Da (u, v) auf einem kürzesten Pfad liegt, gilt $\delta_{f_i}(s, u) + 1 = \delta_{f_i}(s, v)$.

Laufzeit von Edmonds-Karp, Forts.

Sollte (u, v) später noch einmal kritisch werden, so muss zunächst zu einem Zeitpunkt die umgekehrte Kante (v, u) wieder vorhanden sein und auf dem entsprechenden Erweiterungspfad liegen. Sei $j > i$ dieser Zeitpunkt.
Es ist dann $\delta_{f_j}(s, v) + 1 = \delta_{f_j}(s, u)$, da ja auch dann (v, u) auf einem kürzesten Pfad liegt.
Es folgt
$\delta_{f_i}(s, v) = \delta_{f_i}(s, u) + 1 \leq \delta_{f_j}(s, u) + 1 = \delta_{f_j}(s, v) + 2$.
D.h. mit jedem Kritischwerden der Kante (u, v) erhöht sich der Abstand von v zu s im Restnetzwerk um 2. Daher kann die Kante nur insgesamt $|V|/2$ mal kritisch werden. Es gibt $2|E|$ Kanten in Restnetzwerken, bei jedem Schritt ist eine Kante kritisch. Folglich gibt es hoechstens $|E| \cdot |V|$ Schritte.

Satz

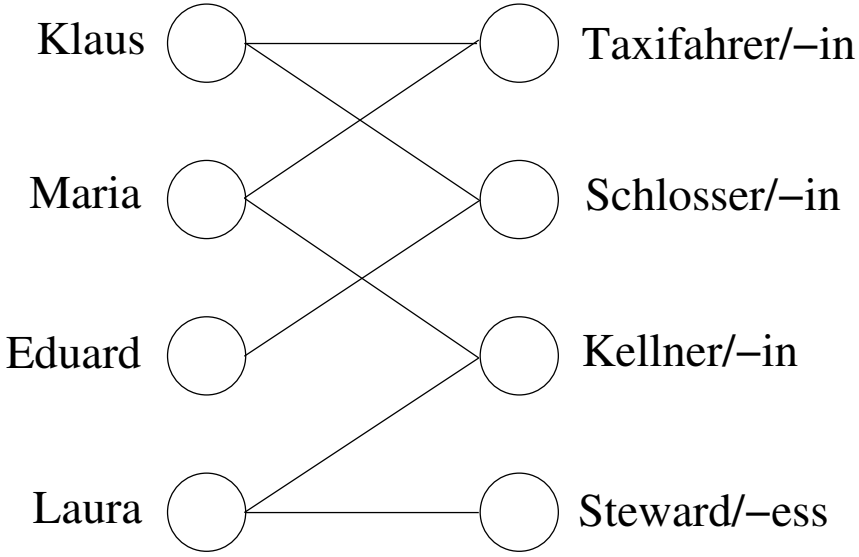
Laufzeit des Edmonds-Karp Algorithmus ist $O(|V| \cdot |E|^2)$.

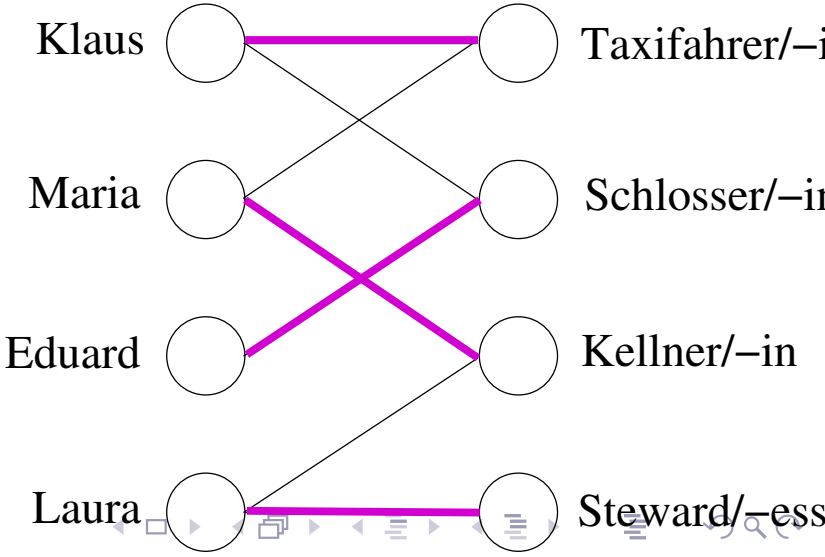
Anwendung: Maximale bipartite Matchings

Gegeben: ein ungerichteter Graph $G = (V, E)$ mit $V = L \cup R$ und $L \cap R = \emptyset$ und Kanten verbinden nur Knoten in L mit Knoten in R (bipartiter Graph).

Gesucht: $M \subseteq E$ sodass wenn $(u, v) \in M$ und $(u', v') \in M$, dann $u = u' \Leftrightarrow v = v'$. Also nicht $u = u'$ aber $v \neq v'$ und auch nicht $v = v'$ aber $u \neq u'$. Außerdem soll M so groß wie möglich sein.

Solch eine Kantenmenge M heißt maximales Matching. Ohne die Maximalitätsbedingung spricht man von Matching.



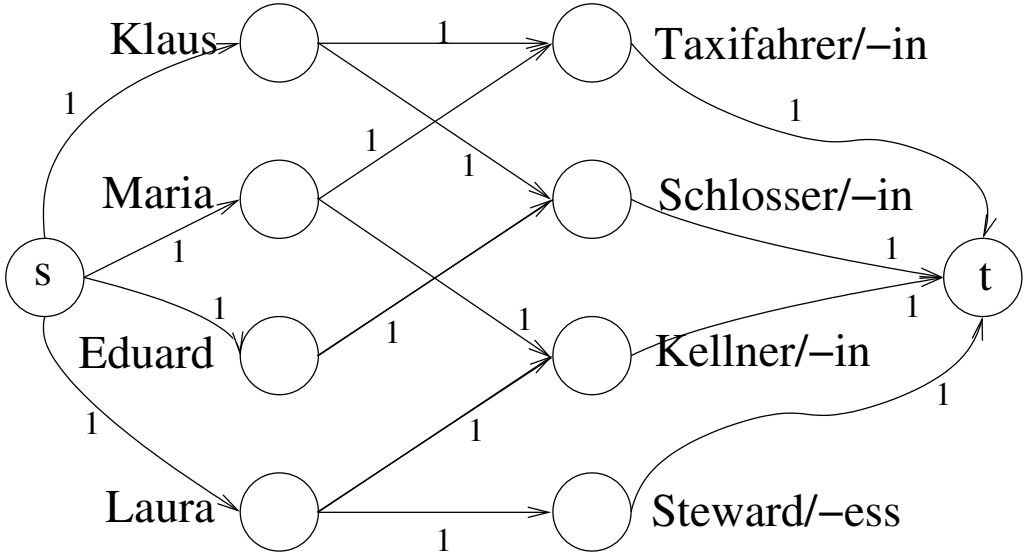


Lösung mit Ford-Fulkerson

Idee: Betrachte $G' = (V', E')$, wobei $V' = V \cup \{s, t\}$, und

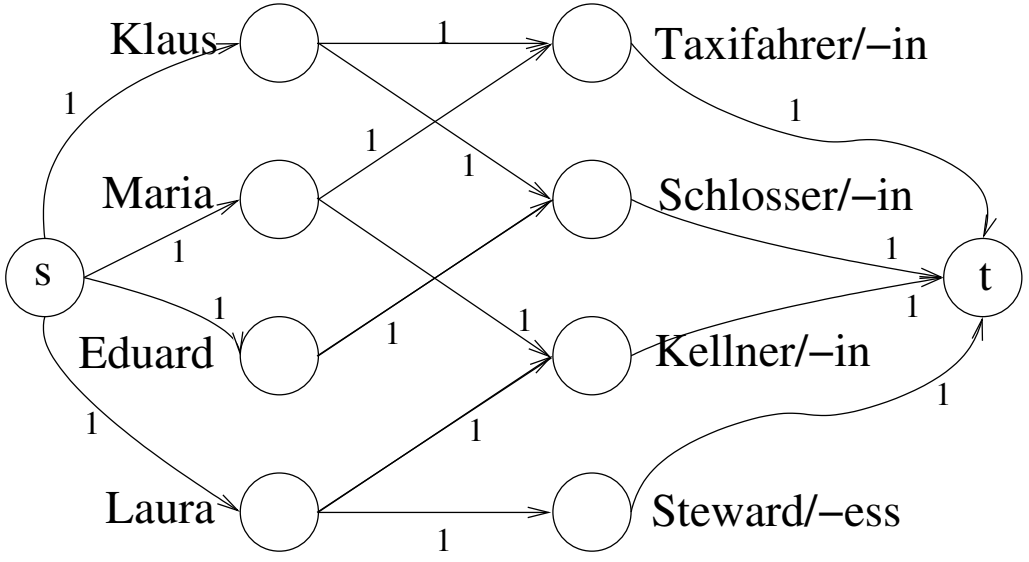
$$E' = E \cup \{(s, \ell)\,;\, \ell \in L\} \cup \{(r, t)\,;\, r \in R\}$$

mit Kapazität $c(e) = 1$ für alle $e \in E'$.



Zusammenhang zwischen Matching und Fluss

Beobachtung: Jedes Matching M in G entspricht einem ganzzahligen Fluss in G' mit $|f| = |M|$, und umgekehrt.



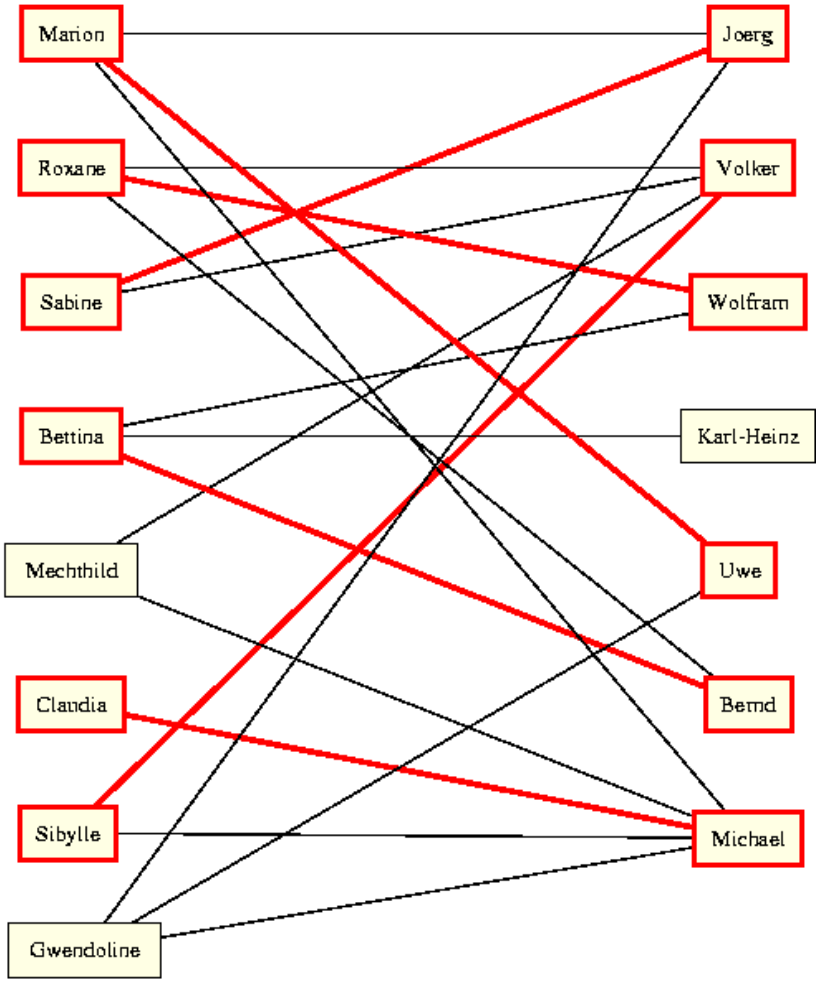
Satz

Ist die Kapazitätsfunktion c ganzzahlig, so ist auch der mit der Ford-Fulkerson-Methode gefundene maximale Fluss ganzzahlig.

Beweis: Alle Flüsse während eines Laufs von Ford-Fulkerson sind ganzzahlig.

Heiratsproblem

Offensichtliche Anwendung:



ENDE

Aus `www.leda-tutorial.org`.