Teil IX

Graphen

Überblick

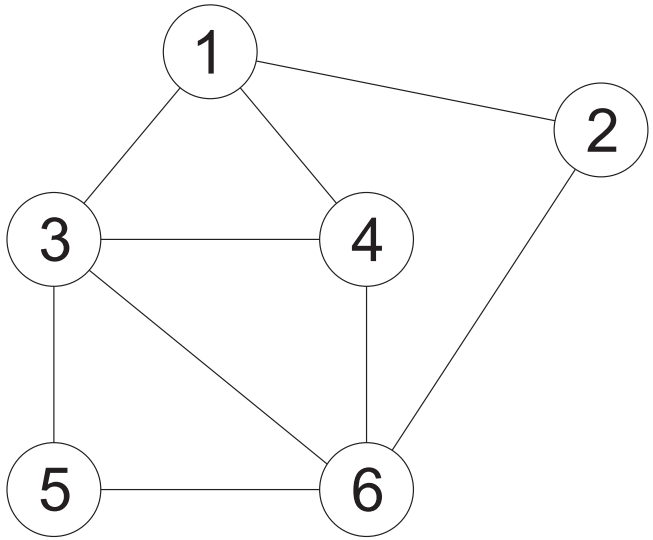
Graphen

- Netzwerke aus Knoten und Kanten
- vielfältiger Einsatz
  - Verbindungsnetzwerke: Bahnnetz, Flugverbindungen, Strassenkarten, ...
  - Verweise: WWW, Literaturverweise, Wikipedia, symbolische Links, ...
  - technische Modelle: Platinen-Layout, finite Elemente, Computergraphik, ...
  - Software-Dokumentation
  - Begriffshierarchien
- Bäume und Listen sind spezielle Graphen !

Arten von Graphen

- Ungerichtete Graphen
  - Strassenverbindungen, Telefonnetz, Nachbarschaft, ..
- Gerichtete Graphen
  - Förderanlagen, Kontrollfluss in Programmen, ...
- Gewichtete Graphen
  - Bahnnetz mit Kosten, Strassennetz mit Kilometern, ...
- *Multi-Graphen, Hyper-Graphen, ...*
  - Folksonomies in Sozialen Bookmarking-Systemen

Ungerichteter Graph G_u

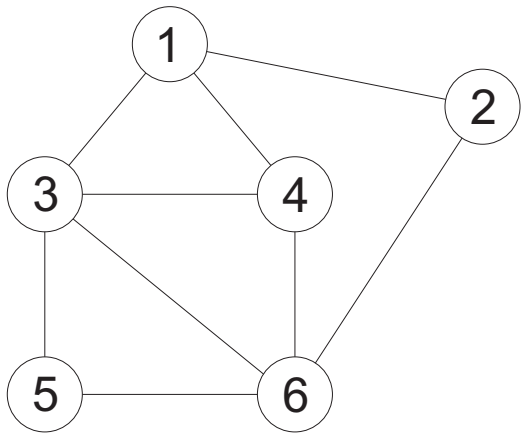


Ungerichtete Graphen formal

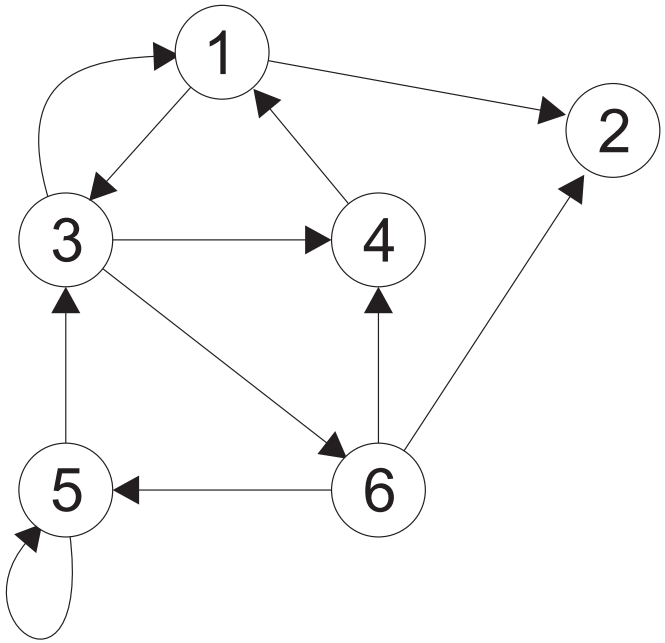
- Graph als Zweitupel $G = (V, E)$
  - endliche Menge V von Knoten (V für englisch *vertices*)
  - Menge E von Kanten (E für englisch *edges*)
    - $e \in E$ ist zweielementige Teilmenge der Knotenmenge V.
- keine *Schleifen*, d.h. Kanten von einem Knoten zu sich selbst
- keine mehrfachen Kanten zwischen zwei Knoten (Parallelkanten)

Beispiel für ungerichteten Graph G_u

- $G_u = (V_u, E_u)$
- $V_u = \{1, 2, 3, 4, 5, 6\}$
- $E_u = \{\{1,2\}, \{1,3\}, \{1,4\}, \{2,6\}, \{4,6\}, \{3,6\}, \{5,6\}, \{3,4\}, \{3,5\}\}$



Gerichteter Graph G_g

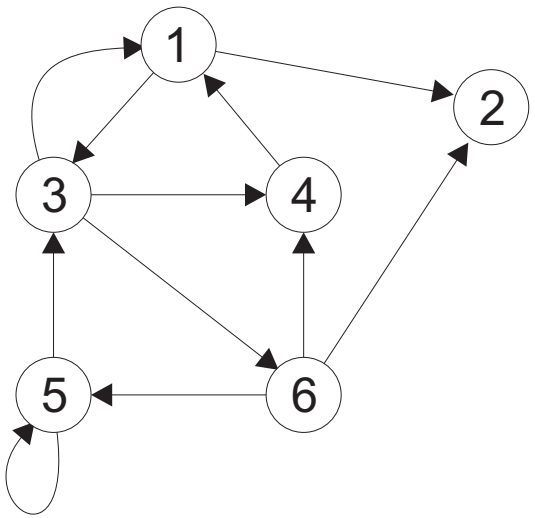


Gerichtete Graphen formal

- Zweitupel $G = (V, E)$ mit
  - V, einer endlichen Menge von Knoten, und
  - E, einer Menge von Kanten.
- jedes $e \in E$ ist nun ein Tupel (a, b) mit $a, b \in V$
- Schleifen (a, a) sind erlaubt

Gerichtete Graphen am Beispiel G_g

- $G_g = (V_g, E_g)$
- $V_g = \{1, 2, 3, 4, 5, 6\}$
- $E_g = \{(1,2), (1,3), (3,1), (4,1), (3,4), (3,6), (5,3), (5,5), (6,5), (6,2), (6,4)\}$



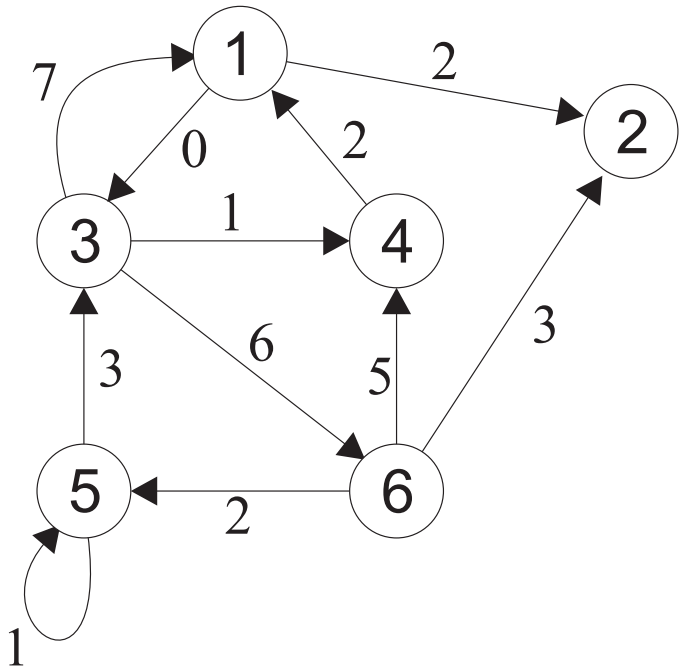
Gewichtete Graphen

- Graph $+$
- Kantengewichte γ, etwa natürliche Zahlen
- $G = (V, E, \gamma)$ mit

$$\gamma \colon E \to \mathbb{N}$$

- gerichtet oder ungerichtet möglich

Gewichteter Graph



Realisierung von Graphen

- dynamische Datenstruktur
- Kanten- und Knotenlisten
- Matrixdarstellung

Direkte Realisierung

- analog zu Bäumen
- Knoten mit Zeiger-Liste
- unüblich, da komplexe Strukturen mit vielen Fehlermöglichkeiten

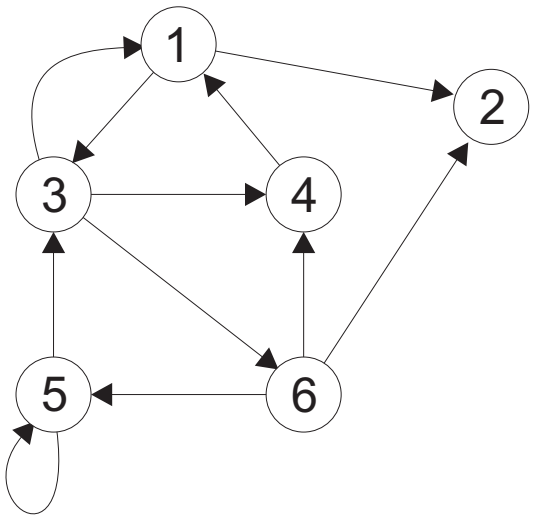
Kanten- und Knotenlisten

- einfache Realisierung bei durchnummerierten Knoten
- historisch erste verwendete Datenstruktur
- als Austauschfomat geeignet
- Auflistung nach Knoten oder nach Kanten sortiert

Kantenlisten

- Kantenliste für G_g:

$$6, 11, 1, 2, 1, 3, 3, 1, 4, 1, 3, 4, 3, 6, 5, 3, 5, 5, 6, 5, 6, 2, 6, 4$$

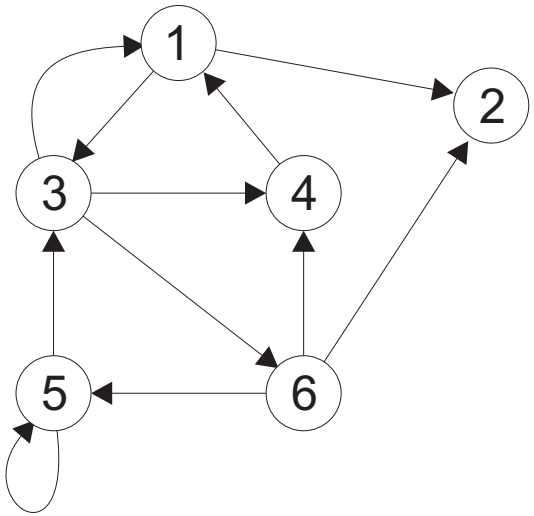


Knotenlisten

- G_g als Knotenliste:

$$6, 11, 2, 2, 3, 0, 3, 1, 4, 6, 1, 1, 2, 3, 5, 3, 2, 4, 5$$

- Teilfolge $2, 2, 3$ bedeutet
  - *„Knoten 1 hat Ausgangsgrad 2 und herausgehende Kanten zu den Knoten 2 und 3“*



Kanten- versus Knotenlisten

- Knotenlisten benötigen weniger Speicherbedarf als Kantenlisten
- Kantenlisten:

$$2 + 2 \cdot |E|$$

- Knotenlisten:

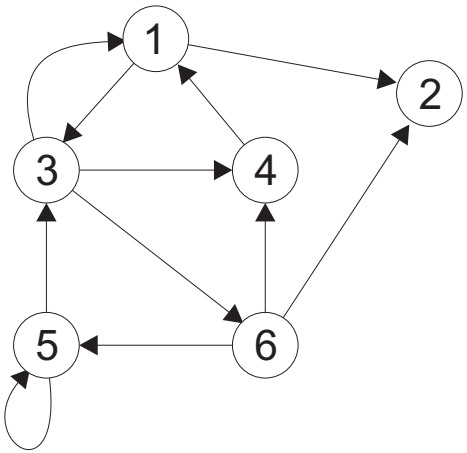
$$2 + |V| + |E|$$

Adjazenzmatrix

- Adjazenzmatrix: Adjazenz bedeutet *Berühren, Aneinandergrenzen*
- Darstellung des Graphen als Boolesche Matrix
- Einträge für Nachbarschaft / direkte Erreichbarkeit durch Kanten
- ungerichtete Graphen: Halb-Matrix (Dreieck) reicht aus
- gewichtete Graphen:
  - Gewichte statt Boolesche Werte
- einige Graphoperationen als Matrixoperationen möglich
  - Ereichbarkeit durch iterierte Matrix-Multiplikation

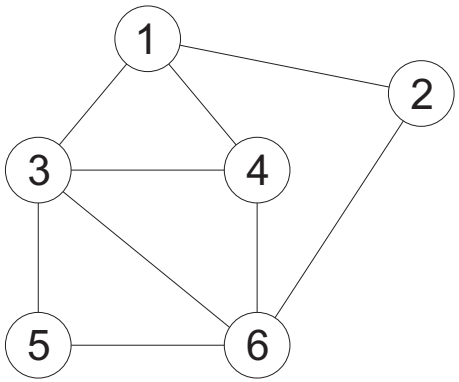
Adjazenzmatrix: gerichtet

$$G_g = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \end{pmatrix}$$



Adjazenzmatrix: ungerichtet

$$G_u = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 0 \end{pmatrix}$$



- Die Matrix ist symmetrisch.

Adjazenzmatrix: ungerichtet als Dreiecksmatrix

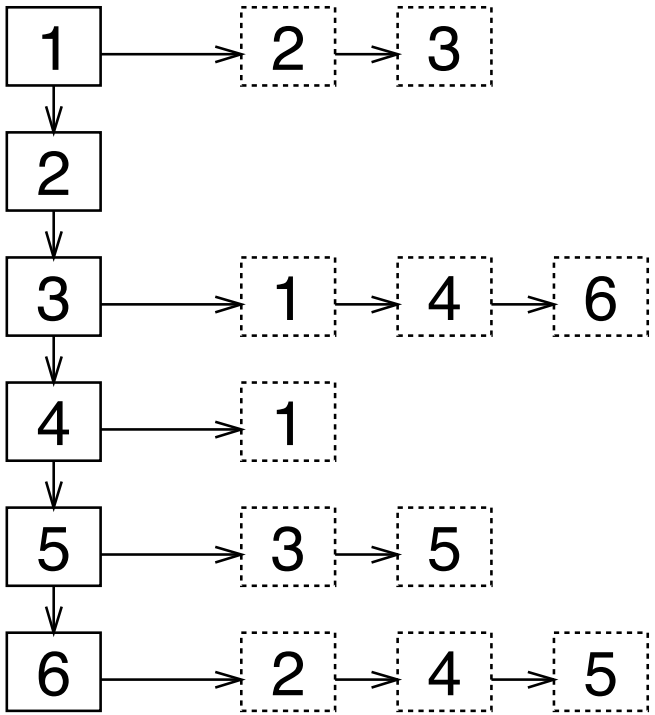
$$G_u = \begin{pmatrix} 0 & & & & & \\ 1 & 0 & & & & \\ 1 & 0 & 0 & & & \\ 1 & 0 & 1 & 0 & & \\ 0 & 0 & 1 & 0 & 0 & \\ 0 & 1 & 1 & 1 & 1 & 0 \end{pmatrix}$$

- Diagonale kann ebenfalls weggelassen werden wenn Schleifen verboten sind

Dynamisch mit Adjazenzlisten

- Liste der Knoten (alternativ: Array)
- pro Knoten die von ihm ausgehenden Kanten
  - als Liste oder Array von Zeigern
  - als Adjazenzliste
- Graph durch $|V| + 1$ verkettete Listen realisiert

Graph mit Adjazenzlisten



Speicherbedarf bei Adjazenzlisten

- seien $n = |V|$ und $m = |E|$
- benötigt werden insgesamt

$$n + \sum_{i=1}^{n} ag(i) = n + m$$

Listenelemente

Komplexitätsbetrachtungen I

- *Kantenlisten*
  - Einfügen von Kanten (Anhängen zweier Zahlen) und von Knoten (Erhöhung der ersten Zahl um 1) besonders günstig
  - Löschen von Kanten: Zusammenschieben in einer Liste
  - Löschen von Knoten: Durchnummerierung der Knoten
- *Knotenlisten*
  - Einfügen von Knoten (Erhöhung der ersten Zahl und Anhängen einer 0) günstig
- *Matrixdarstellung*
  - Manipulieren von Kanten sehr effizient ausführbar
  - Aufwand bei Knoteneinfügung hängt von Realisierung ab
    - ★ eventuell Kopieren der Matrix in eine größere Matrix
- *Adjazenzliste*
  - unterschiedlicher Aufwand, je nachdem ob die Knotenliste als Feld (mit Direktzugriff) oder als verkettete Liste (mit sequenziellem Durchlauf) realisiert

Komplexitätsbetrachtungen II

Operation	Kanten-liste	Knoten-liste	Adjazenz-matrix	Adjazenz-liste
Einfügen Kante	$O(1)$	$O(n+m)$	$O(1)$	$O(1)$ / $O(n)$
Löschen Kante	$O(m)$	$O(n+m)$	$O(1)$	$O(n)$
Einfügen Knoten	$O(1)$	$O(1)$	$O(n^2)$	$O(1)$
Löschen Knoten	$O(m)$	$O(n+m)$	$O(n^2)$	$O(n+m)$

- Löschen eines Knotens erfordert Löschen der zugehörigen Kanten!

Breitensuche/Breitendurchlauf

- Knoten eines Graphen nach Nähe zum Startknoten anordnen / durchlaufen
- breadth-first-search, abgekürzt als BFS
- Realisierung für ungerichteten Graphen
  - Warteschlange als Zwischenspeicher
  - Farbmarkierungen geben Status der Knoten an
    - weiß: unbearbeitet
    - grau: in Bearbeitung
    - schwarz: abgearbeitet
  - pro Knoten wird Entfernung zum Startknoten berechnet
  - ebenfalls möglich: Berechnung eines aufspannenden Baums mit Startknoten als Wurzel (Kanten π für predecessor)

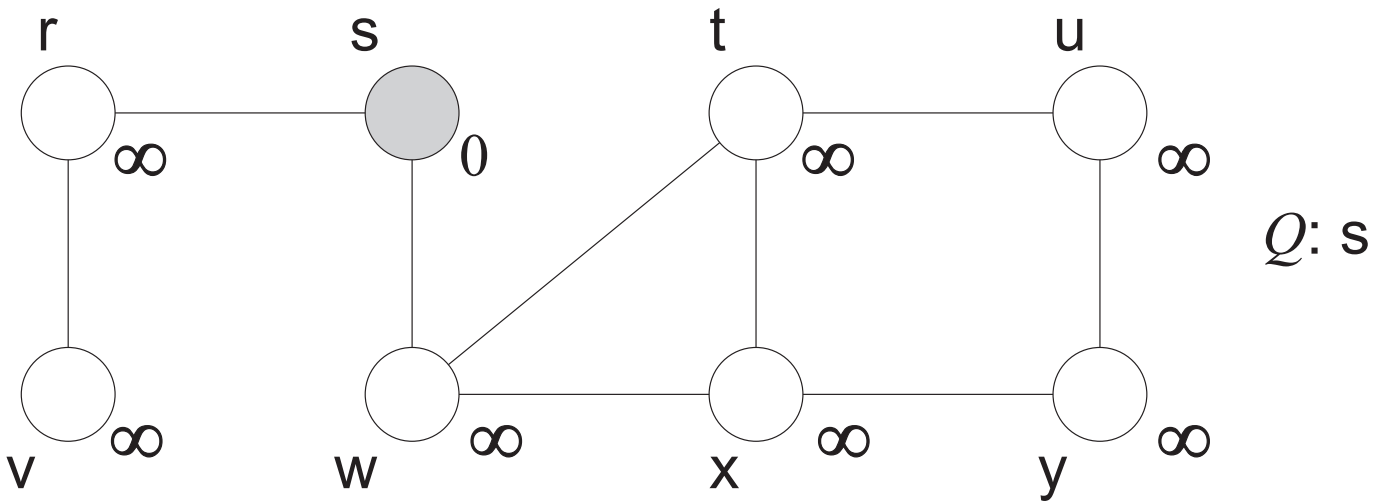
Breitensuche: Algorithmus

```
algorithm BFS (G, s)
   Eingabe: ein Graph G, ein Startknoten s ∈ V[G]
   for each Knoten u ∈ V[G] − s do
      farbe[u]= weiß; d[u] = ∞; π[u] = null
   od
   farbe[s] = grau; d[s] = 0; π[s] = null
   Q = emptyQueue; Q = enqueue(Q,s);
   while ¬ isEmpty(Q) do
      u = front(Q);
      for each v ∈ ZielknotenAusgehenderKanten(u) do
         if farbe(v) = weiß then
            farbe[v] = grau; d[v] = d[u]+1;
            π[v] = u; Q = enqueue(Q,v)
         fi
      od
      dequeue(Q); farbe[u] = schwarz
   od
```

Breitensuche: Initialisierung

- Startknoten
  - in Warteschlange einfügen
  - Farbe grau
  - Entfernung 0
- andere Knoten
  - Entfernung unendlich
  - Farbe weiß

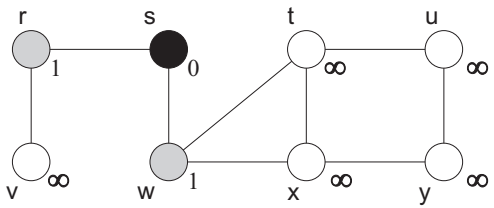
Breitensuche: Erster Schritt



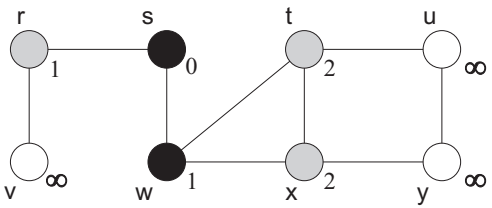
Breitensuche: Einzelschritt

- aktuellen Knoten k aus Warteschlange entnehmen
- k wird schwarz
- alle von k aus erreichbaren weissen Knoten:
  - grau färben
  - Entfernung ist Entfernungswert von k plus 1
  - in die Warteschlange aufnehmen
  - π setzen

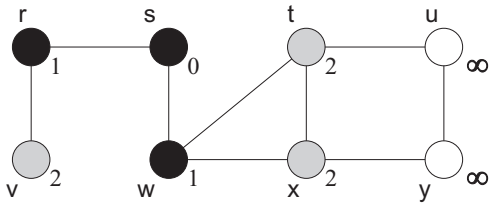
Breitensuche II: Folgeschritte



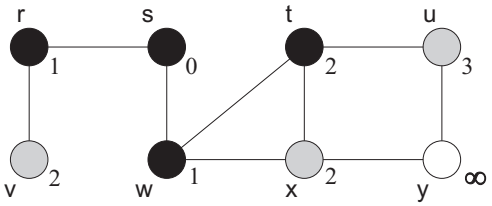
Q: w r



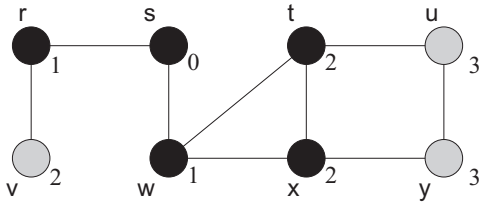
Q: r t x



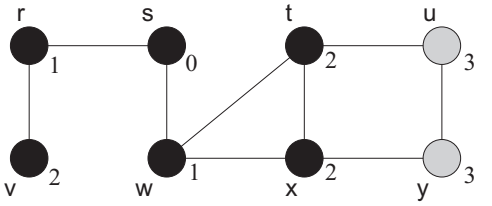
Q: t x v



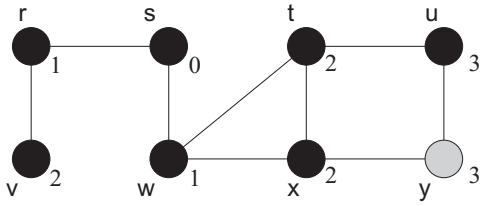
Q: x v u



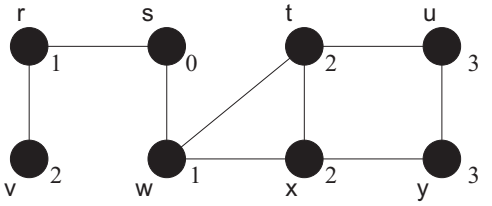
Q: v u y



Q: u y

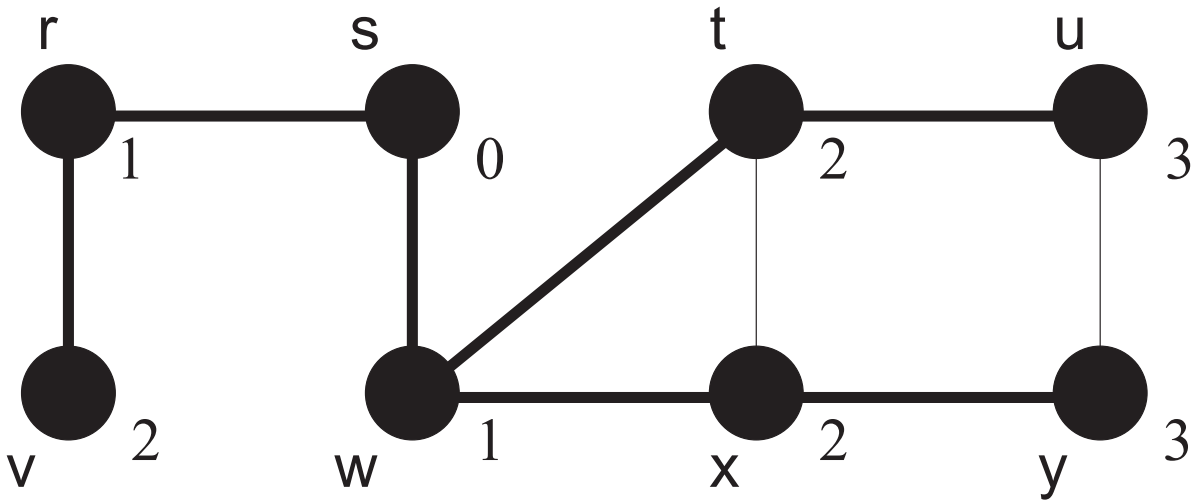


Q: y



Q:

Breitensuche: Ergebnis



Tiefensuche/Tiefendurchlauf

- `DFS` für depth-first search realisiert einen rekursiven Durchlauf durch einen gerichteten Graphen
- Farbmarkierungen für den Bearbeitungsstatus eines Knotens
  - Weiß markiert: noch nicht bearbeitete Knoten
  - graue Knoten: in Bearbeitung
  - schwarze: bereits fertig abgearbeitet

Tiefensuche: Algorithmus

```
algorithm DFS (G)
   Eingabe: ein Graph G

   for each Knoten u ∈ V[G] do
      farbe[u]= weiß; π[u] = null
   od;
   zeit = 0;
   for each Knoten u ∈ V[G] do
      if farbe[u]= weiß then DFS-visit(u) fi
   od
```

Tiefensuche: Algorithmus II

```
algorithm DFS-visit (u)
   Eingabe: ein Knoten u

   farbe[u]= grau; zeit = zeit+1; d[u]=zeit;
   for each v ∈ ZielknotenAusgehenderKanten(u) do
      if farbe(v) = weiß then
         π[v] = u; DFS-visit(v)
      fi
   od;
   farbe[u] = schwarz; zeit = zeit+1; f[u]=zeit
```

Tiefensuche: Vorgehen

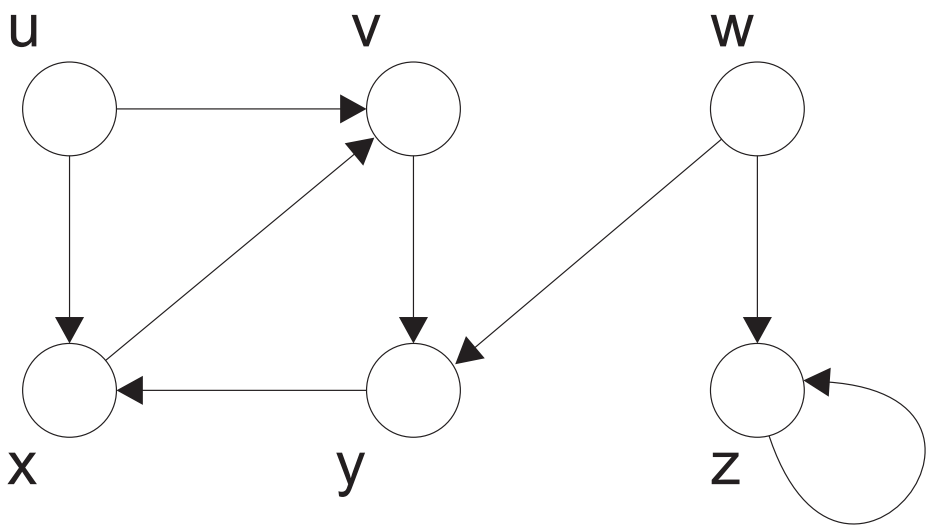
- rekursiver Abstieg
- rekursiver Aufruf nur bei *weissen* Knoten
  - Terminierung der Rekursion garantiert!

- pro Knoten Farbwert plus zwei Zeitwerte
  - Beginn der Bearbeitung d
  - Ende der Bearbeitung f
- Zeit ist für reine Suchfunktion nicht nötig
  - . . . gibt aber Information über Graphstruktur (s. nächste Folie).
  - . . . wird für topologische Sortierung genutzt.

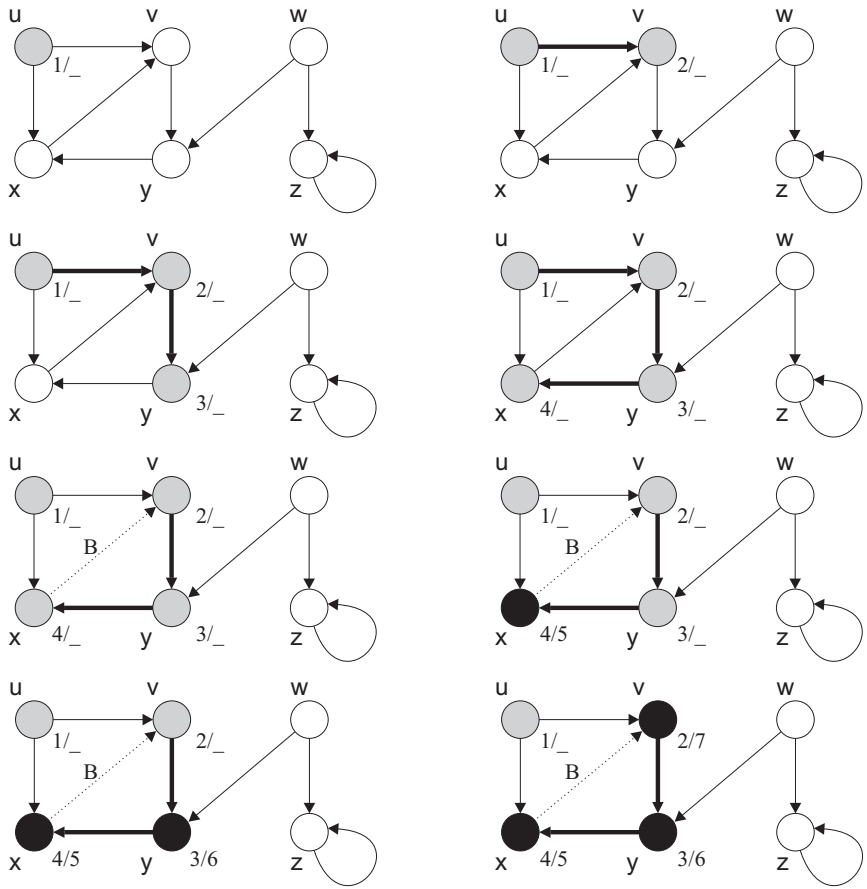
Tiefensuche: gefundene Informationen

- Kanten des aufspannenden Baumes: Zielknoten bei Test ist weiß
- Rückkanten (back edges, B):
  - Zielknoten beim Test grau
  - zeigt einen Zyklus an!
- Vorwärtskanten (forward edges, F) in den aufgespannten Baum:
  - beim Test wird ein schwarzer Knoten gefunden, dessen Bearbeitungsintervall ins Intervall des aktuell bearbeiteten Knotens passt.
- Verbindungskanten (cross edges, C):
  - schwarzer Zielknoten v, dessen Intervall nicht ins Intervall passt ($d[u] > d[v]$):
  - eine Kante, die zwei aufspannende Bäume verbindet

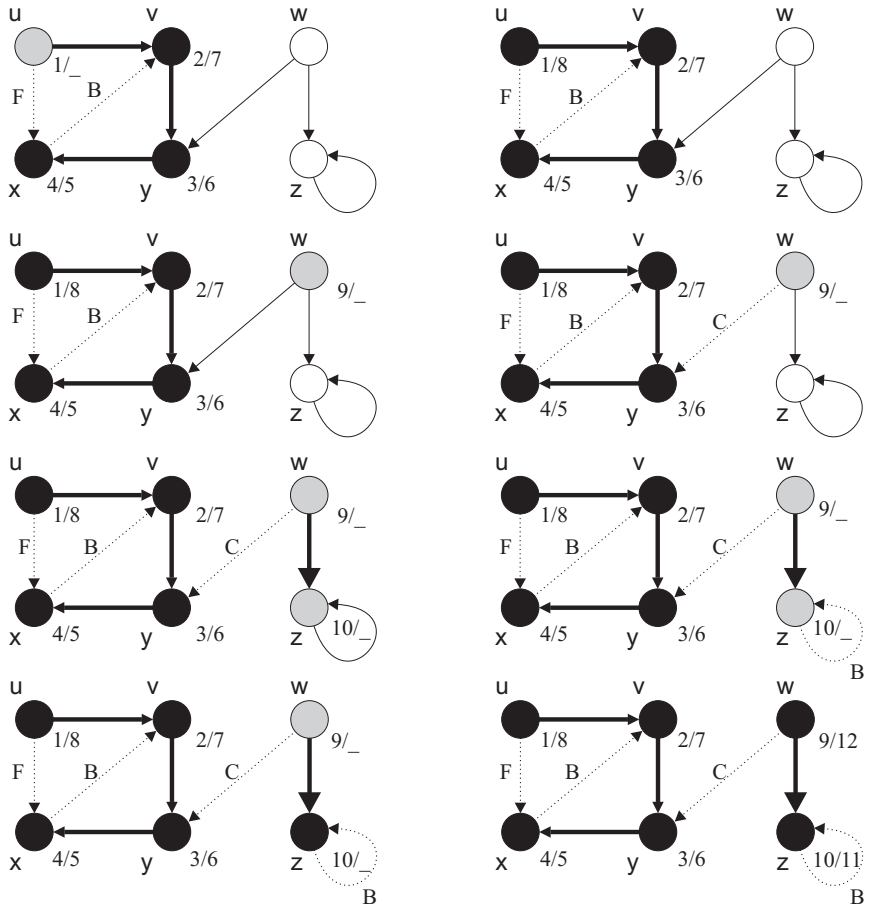
Tiefensuche: Beispielgraph



Tiefensuche II: erste Schritte



Tiefensuche III: restliche Schritte



Einsatz von DFS

- Test auf Zyklenfreiheit
  - basiert auf dem Erkennen von Back-Edges
  - effizienter als beispielsweise Konstruktion der transitiven Hülle
- Topologisches Sortieren
  - „topologisch“ sortieren nach Nachbarschaft, nicht nach totaler Ordnung

Topologisches Sortieren

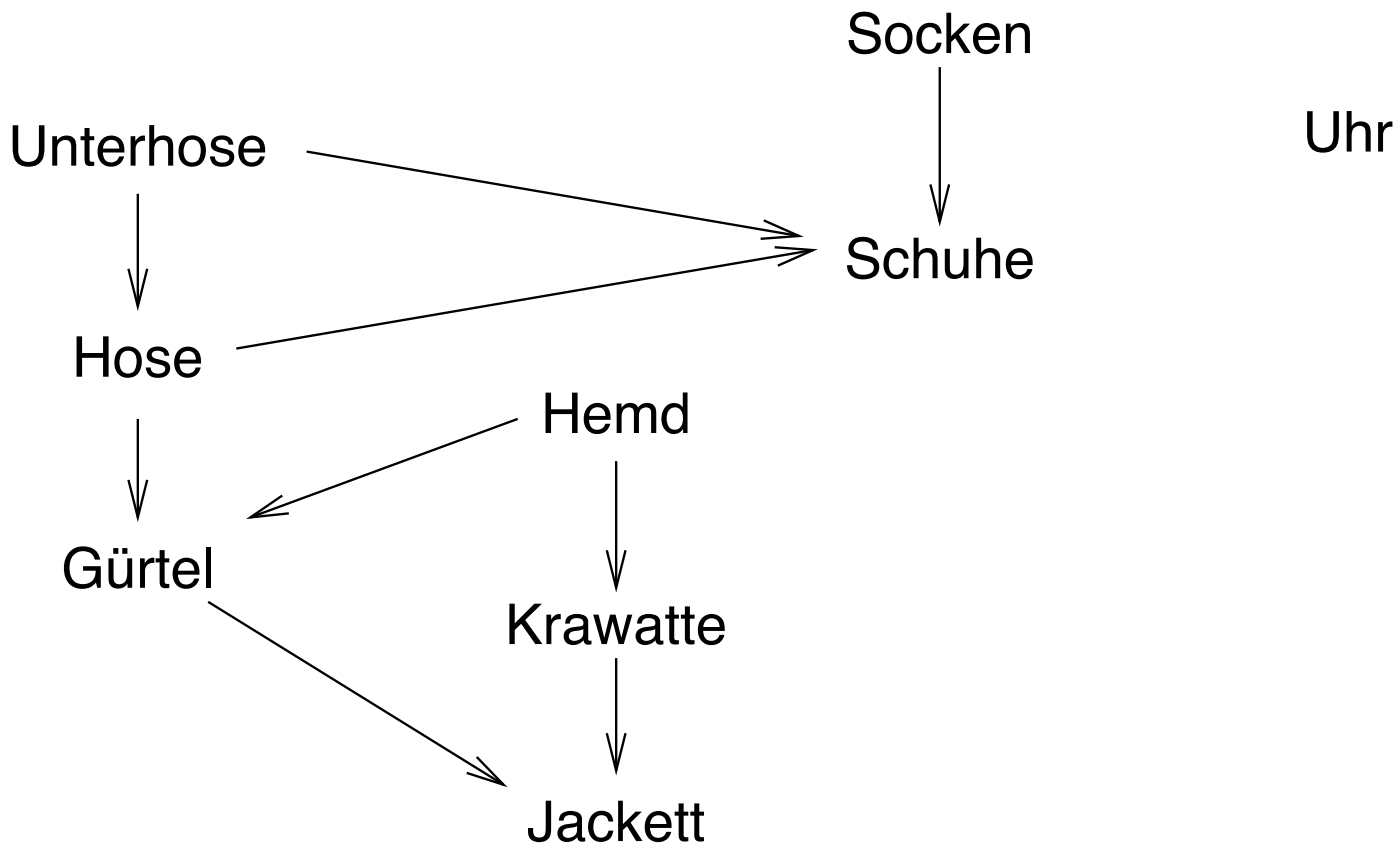
- gegeben: azyklischer gerichteter Graph
- gesucht: Reihenfolge der Knoten, so dass jeder Knoten nach all seinen Vorgängern kommt
  - keine „Rückkanten“
- Scheduling bei kausalen / zeitlichen Abhängigkeiten
- mathematisch: Konstruktion einer totalen Ordnung aus einer Halbordnung

Der zerstreute Professor

Ein zerstreuter Professor hat Probleme, morgens seine Kleidungsstücke in der richtigen Reihenfolge anzuziehen. Daher legt er die Reihenfolgebedingungen beim Ankleiden fest:

```
Unterhose vor Hose
Hose vor Gürtel
Hemd vor Gürtel
Gürtel vor Jackett
Hemd vor Krawatte
Krawatte vor Jackett
Socken vor Schuhen
Unterhose vor Schuhen
Hose vor Schuhen
Uhr: egal
```

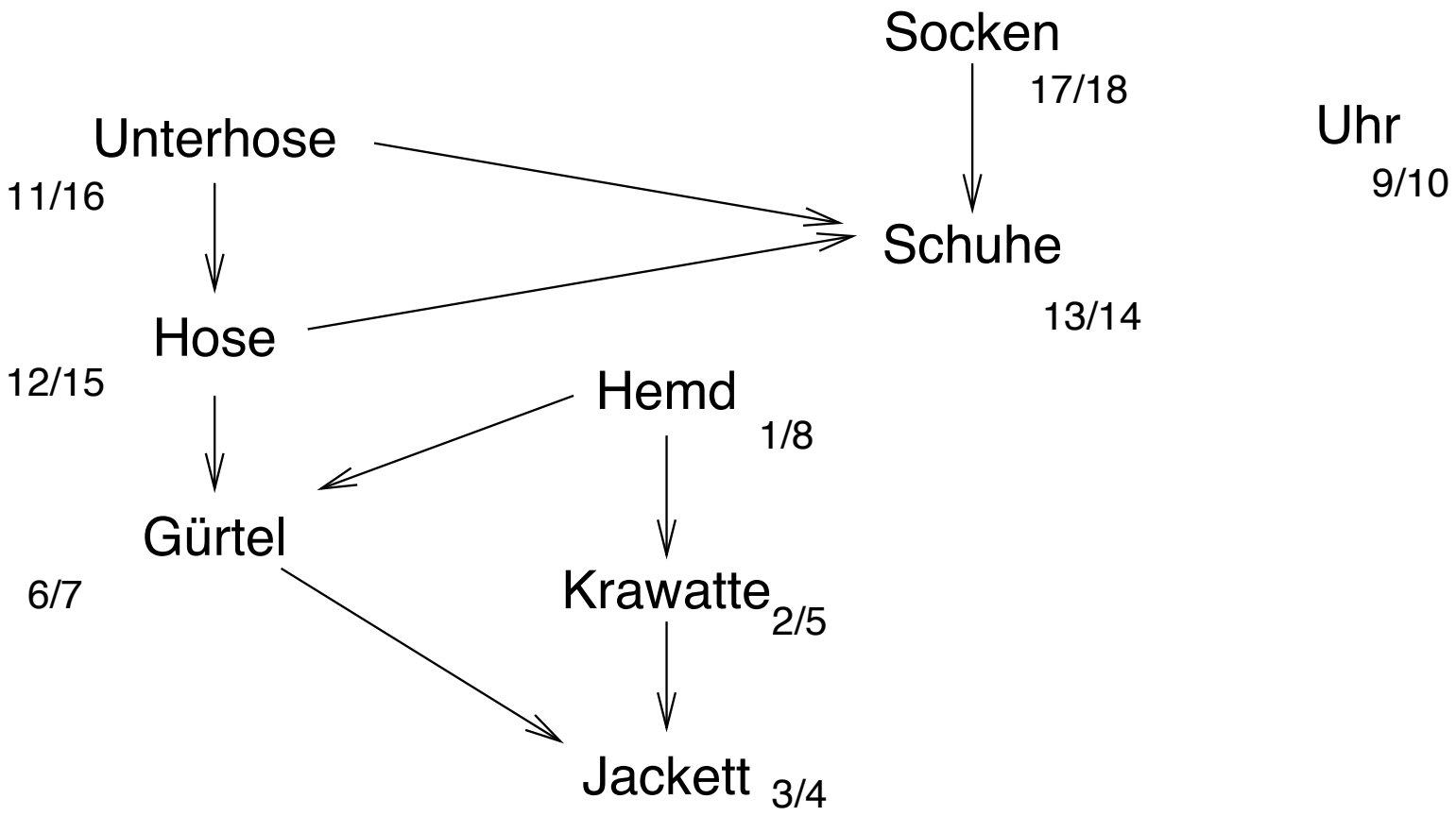
Topologisches Sortieren: Vorgaben



Topologisches Sortieren und DFS

- DFS erstellt topologische Ordnung „on-the-fly“
- Sortierung nach f-Wert (invers) ergibt korrekte Reihenfolge
- statt expliziter Sortierung nach f:
  1. Knoten beim Setzen des f-Wertes vorne in eine verkettete Liste einhängen

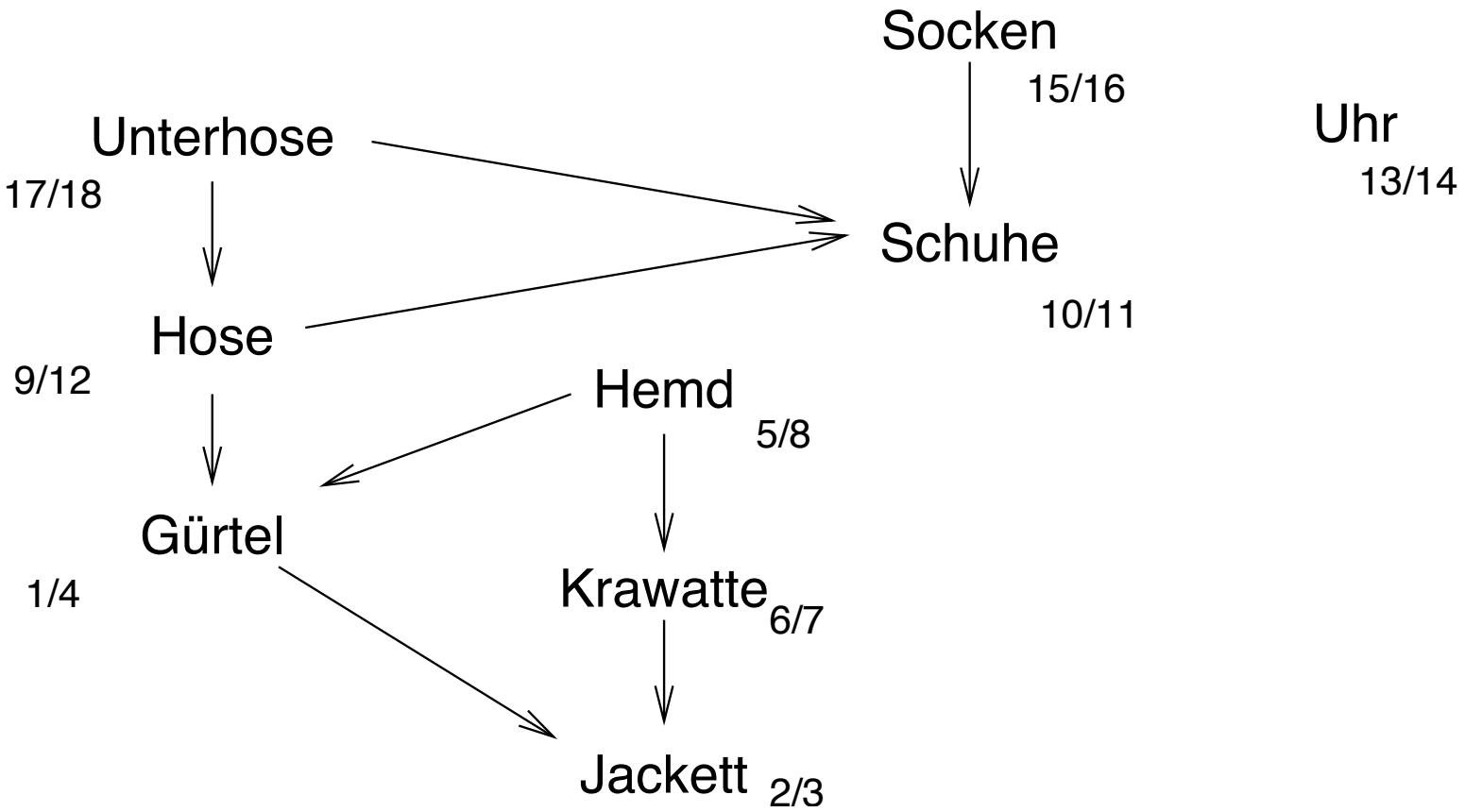
Topologisches Sortieren mit DFS: Ergebnisgraph



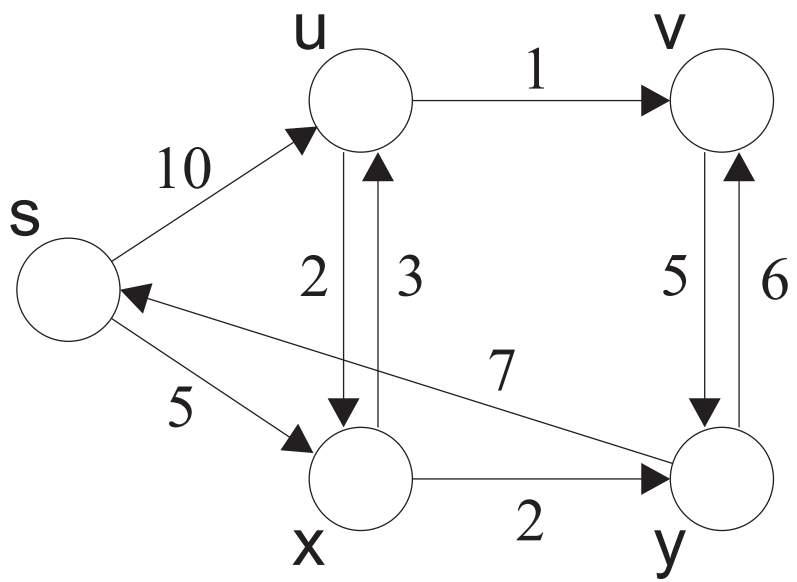
Topologisches Sortieren: Ergebnisliste

```
18 Socken
16 Unterhose
15 Hose
14 Schuhe
10 Uhr
8  Hemd
7  Gürtel
5  Krawatte
4  Jackett
```

Topologisches Sortieren: Alternativer Durchlauf



Algorithmen auf gewichteten Graphen



Gewichtete Graphen

- ungerichtete gewichtete Graphen
  - Flugverbindungen mit Meilen oder Kosten
  - Strassennetze mit km
  - Rohrsysteme mit Durchfluss
- gerichtete gewichtete Graphen
  - Strassennetze mit Einbahnstrassen
  - Rohre mit Ventilen
  - Förderbänder

Optimierungsprobleme

- Gegeben:
  - Menge zulässiger Lösungen
  - Zielfunktion (Kostenfunktion): ordnet jeder Lösung einen Wert (reelle Zahl) zu.
- Aufgabe: bestimme zulässige Lösung mit minimalem Wert der Zielfunktion
  - statt minimalem kann auch maximaler Wert gesucht sein
  - evtl. wird die Menge der zulässigen Lösungen durch Nebenbedingungen (Constraints) eingeschränkt
- Beispiele:
  - kürzester Weg von s nach t
  - Stundenplan mit möglichst wenig Lehrstunden
  - Schedule mit möglichst geringer Ausführungszeit
  - Kommunikationsnetz mit möglichst geringem Kabelverbrauch (= minimaler Spannbaum)

Kürzeste Wege

- Graph $G = (V, E, \gamma)$ mit einer Gewichtsfunktion $\gamma\colon E \to \mathbb{N}$
  - als Kantengewichte natürliche Zahlen
- Weg oder auch *Pfad* durch G ist Liste von aneinanderstoßenden Kanten:

$$P = \langle (v_1, v_2), (v_2, v_3), (v_3, v_4), \ldots, (v_{n-1}, v_n) \rangle$$

  P steht hier für das englische *path*.
- *Gewicht* oder auch *Länge* eines Pfades: Aufsummierung der einzelnen Kantengewichte

$$w(P) = \sum_{i=1}^{n-1} \gamma((v_i, v_{i+1}))$$

- Distanz zweier Punkte $d(u, v)$ ist Gewicht des kürzesten Pfades von u nach v

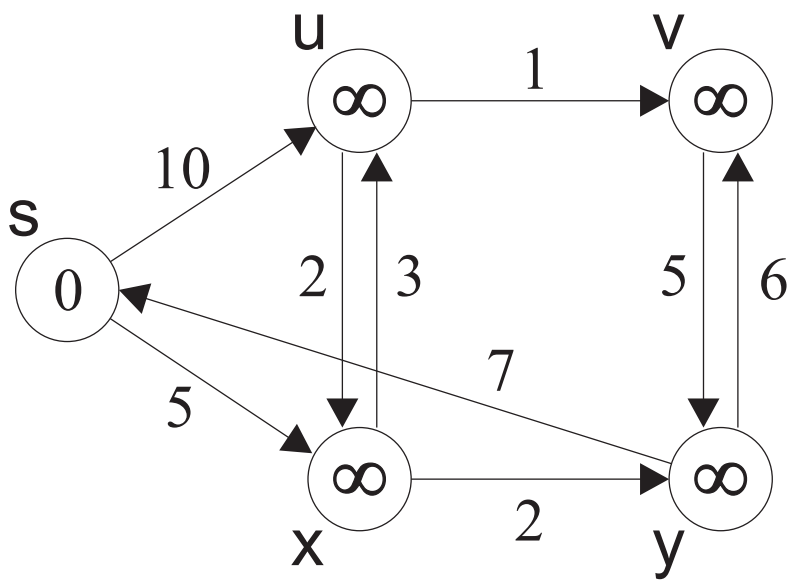
Dijkstras Algorithmus

- Dijkstra 1959
- iterative Erweiterung einer Menge von „billig“ erreichbaren Knoten
- Greedy-Algorithmus
  - ähnlich Breitensuche
  - nur für nichtnegative Gewichte
  - berechnet (iterativ verfeinernd) Distanzwerte D
  - Prioritätswarteschlange zum Herauslesen des jeweils minimalen Elements

Dijkstras Algorithmus

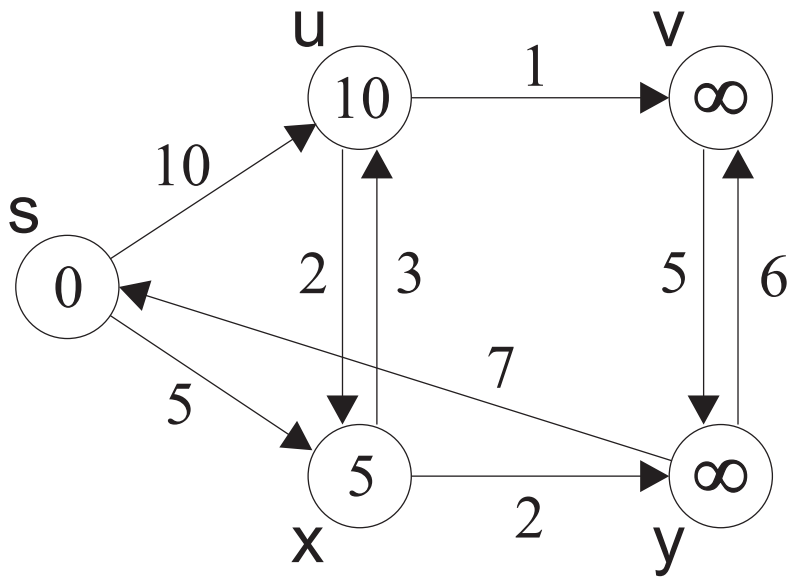
```
algorithm Dijkstra (G, s)
   Eingabe: Graph G mit Startknoten s

      for each Knoten u ∈ V[G] − s do
         D[u] := ∞
      od;
      D[s] := 0; PriorityQueue Q := V;
      while not isEmpty(Q) do
         u := extractMinimal(Q);
         for each v ∈ ZielknotenAusgehenderKanten(u) ∩ Q do
            if D[u] + γ((u, v)) < D[v] then
               D[v] := D[u] + γ((u, v));
               adjustiere Q an neuen Wert D[v]
            fi
         od
      od
```

Dijkstras Algorithmus: Initialisierung



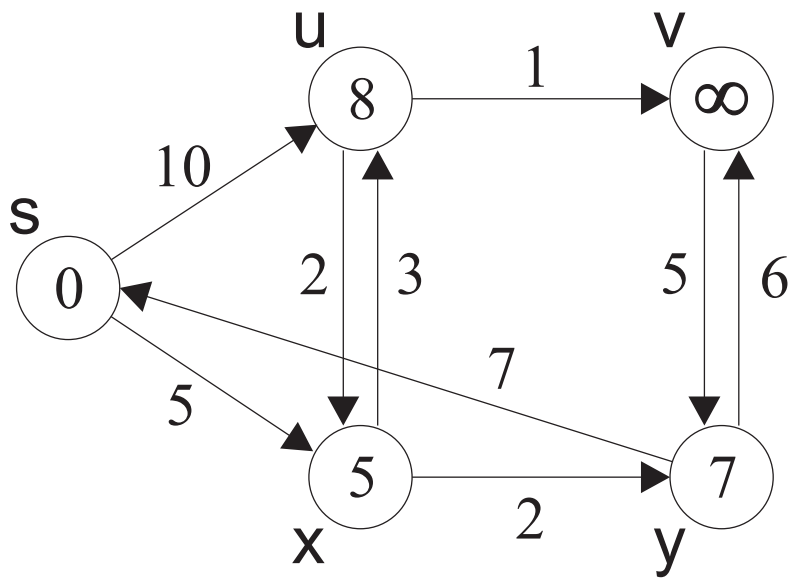
$$Q = \langle (s:0), (u:\infty), (v:\infty), (x:\infty), (y:\infty) \rangle$$

Dijkstras Algorithmus: Schritt 1



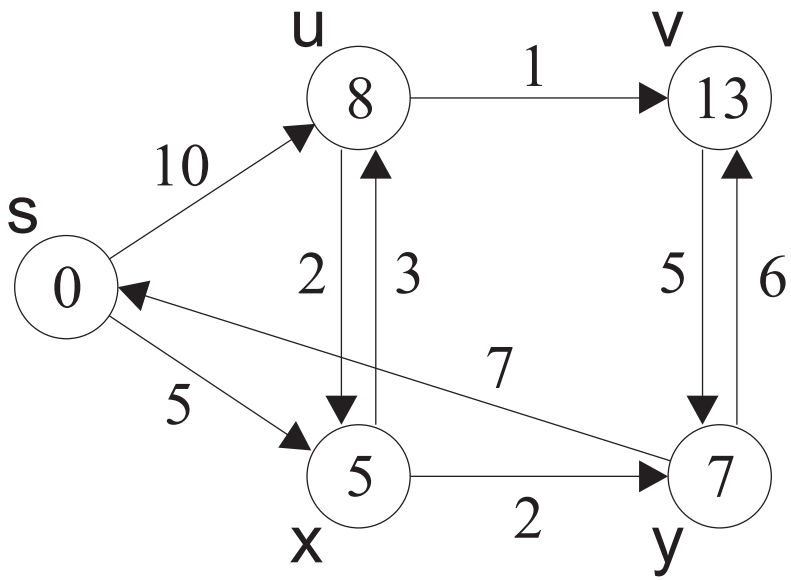
$$Q = \langle (x : 5), (u : 10), (v : \infty), (y : \infty) \rangle$$

Dijkstras Algorithmus: Schritt 2



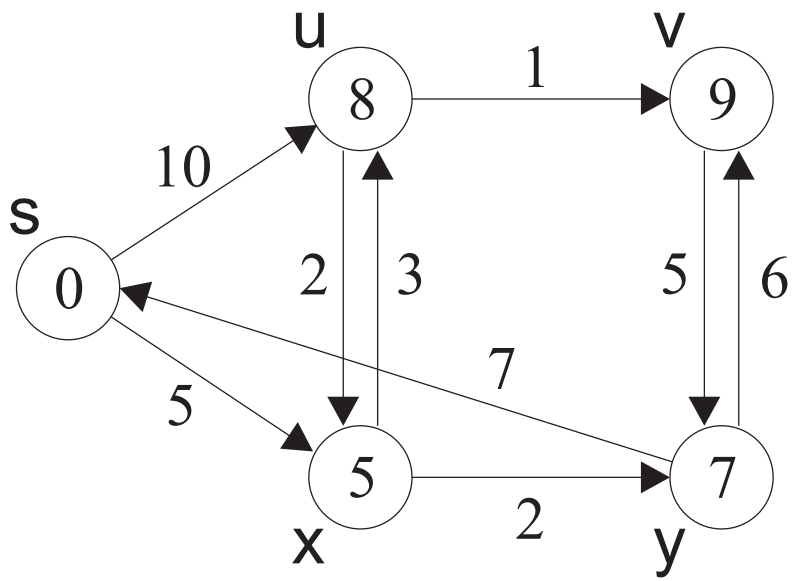
$$Q = \langle (y, 7), (u : 8), (v : \infty) \rangle$$

Dijkstras Algorithmus: Schritt 3



$$Q = \langle (u : 8), (v : 13) \rangle$$

Dijkstras Algorithmus: Schritt 4



$$Q = \langle (v : 9) \rangle$$

Wird das Optimum berechnet?

- Annahme: vorherige Iterationsschritte korrekt (Induktionsbeweis)
- Iterationsschritt: jeweils die „billigste“ Verbindung zu einem noch nicht bearbeiteten Knoten hinzunehmen
  - Da die bisher bearbeiteten Knoten den korrekten Distanzwert haben, ist der neue Distanzwert durch den „billigsten“ aus dem bisher bearbeiteten Teilgraphen um genau eine Kante hinausgehenden Pfad bestimmt.
  - Jeder Pfad zum Zielknoten dieses Pfades, der um mehr als eine Kante aus dem bearbeiteten Bereich hinausgeht, ist teurer als die gewählte, da Kosten mit zusätzlich hinzugenommenen Kanten nicht sinken können.

Greedy-Verfahren I

- greedy = gierig, gefräßig
- Greedy-Strategie konstruiert die Lösung schrittweise (hier: Kante für Kante).
- Dazu werden in jedem Schritt alle zulässigen Erweiterungsmöglichkeiten (hier: noch nicht betrachtete Knoten) nach ihrem lokalen Nutzen (= unmittelbarer Nutzen in diesem Schritt) bewertet (hier: aktueller Abstand zu Start) und die lokal günstigste Erweiterungsmöglichkeit gewählt (hier: Knoten, der bisher als am dichtesten zum Start erkannt wurde).

Greedy-Verfahren II

- Typisch für Greedy-Verfahren: Festhalten an getroffenen Entscheidungen (kein Backtracking)
- Je nach Problem und konkretem Algorithmus liefern Greedy-Verfahren das Optimum (wie bei Dijkstra und Kruskal) oder nur ein lokales Optimum (wie bei Bestimmung des kürzesten Pfads mit Tiefensuche).

Minimaler Spannbaum (Minimal Spanning Tree (MST)

- Sei (V, E) ungerichteter, kantengewichteter zusammenhängender Graph.
- zusammenhängend = von jedem Knoten v gibt es einen Weg zu jedem anderen Knoten w
- Spannbaum: Baum, der alle Knoten des Graphen enthält, und dessen Kanten auch Kanten des Graphen sind.
- Beispiel: Tiefen-/Breitensuchbaum
  - ist hier Baum und nicht Wald, da Graph zusammenhängend
- Minimaler Spannbaum (MST) = Spannbaum, dessen Summe der Kantengewichte minimal ist.
  - billigste Möglichkeit, alle Knoten zu verbinden
- Beispielanwendung: Energieversorgung im Auto

Kruskal-Algorithmus für MST

- gegeben: Graph $G = (V, E)$
- gesucht: Baum $T = (V, B)$ mit $B \subseteq E$, so dass $\sum_{e \in B} \gamma(e)$ minimal.
- Algorithmus von Kruskal:

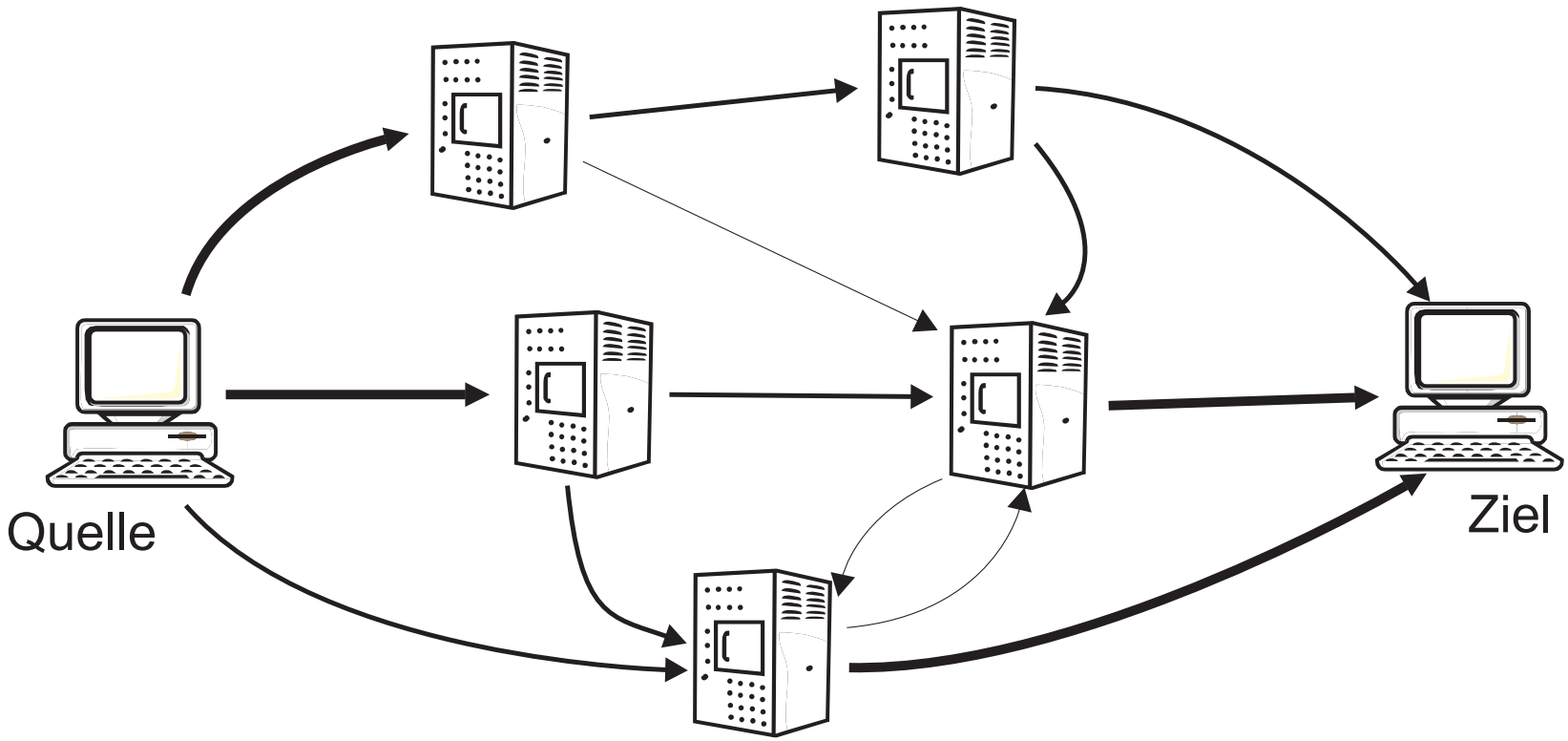
```
B = leere Menge;
betrachte jeden Knoten als Baum;
sortiere die Kanten aufsteigend nach Gewicht;
for (alle Kanten (u,v) aufsteigend sortiert nach γ):
   if (u und v gehören zu verschiedenen Bäumen) {
      // verbinde die beiden Bäume zu einem Baum
      B = B ∪ {(u,v)};
   }
```

- ist greedy
- Laufzeit wird dominiert durch Sortieren: $O(m \log m)$

Maximaler Durchfluss

- logistische Aufgabe:
  - Verteilungsnetz mit Kapazitäten
    - Wasserrohre
    - Förderbänder
    - Paketvermittlung im Rechnernetz
- Quelle liefert (beliebig viele) Objekte pro Zeiteinheit
- Senke verbraucht diese
- jede Verbindung hat eine maximale Kapazität c und einen aktuellen Fluss f
- Wie hoch ist die Übertragungskapazität?

Beispiel Paketvermittlung



Korrekter Fluss

1. Einschränkungen der Kapazität der Kanten werden eingehalten (auch bei negativem Fluss!):

$$|f(u,v)| \leq c((u,v))$$

2. *Konsistenz des Flusses*: bei in beiden Richtungen nutzbaren Verbindungen wird als Nettoeffekt nur in eine Richtung gesendet, und der entstehende negative Fluss nimmt den korrekten Wert an:

$$f(u,v) = -f(v,u)$$

3. Bewahrung des Flusses für jeden Knoten $v \in V - \{q,z\}$ mit Ausnahme der Quelle q und des Ziels z:

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

Maximaler Fluss

- Wert eines Flusses

$$val(G, F, q) = \sum_{u \in V} f(q, u)$$

- Maximaler Fluss

$$\textbf{max}\{val(G, F, q) \mid F \text{ ist korrekter Fluss in } G \text{ bezgl. } q \text{ und } z\}$$

Ford-Fulkerson-Algorithmus

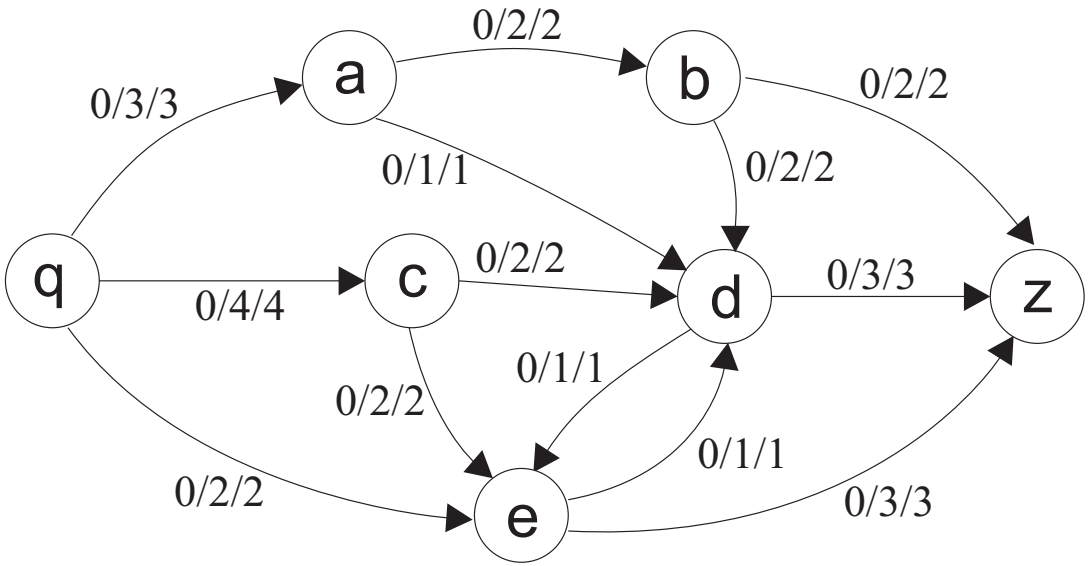
- Berechnet maximalen Fluss
- Vorgehensweise: Mischung aus Greedy und Zufallsauswahl
- Prinzip: Füge so lange verfügbare Pfade zum Gesamtfluss hinzu wie möglich.
- Finden eines nutzbaren Pfades etwa durch Tiefensuche
- Für Kanten werden drei Werte notiert:
  - *aktuelle Fluss f* entlang der Kante
    - ★ im initialisierten Graphen ist dieser Wert überall 0
  - vorgegebene *Kapazität c*
  - abgeleitete noch verfügbare Kapazität $c - f$

Ford-Fulkerson-Algorithmus

```
initialisiere Graph mit leerem Fluss;
do
   wähle nutzbaren Pfad aus;
   füge Fluss des Pfades zum
      Gesamtfluss hinzu;
while noch nutzbarer Pfad verfügbar
```

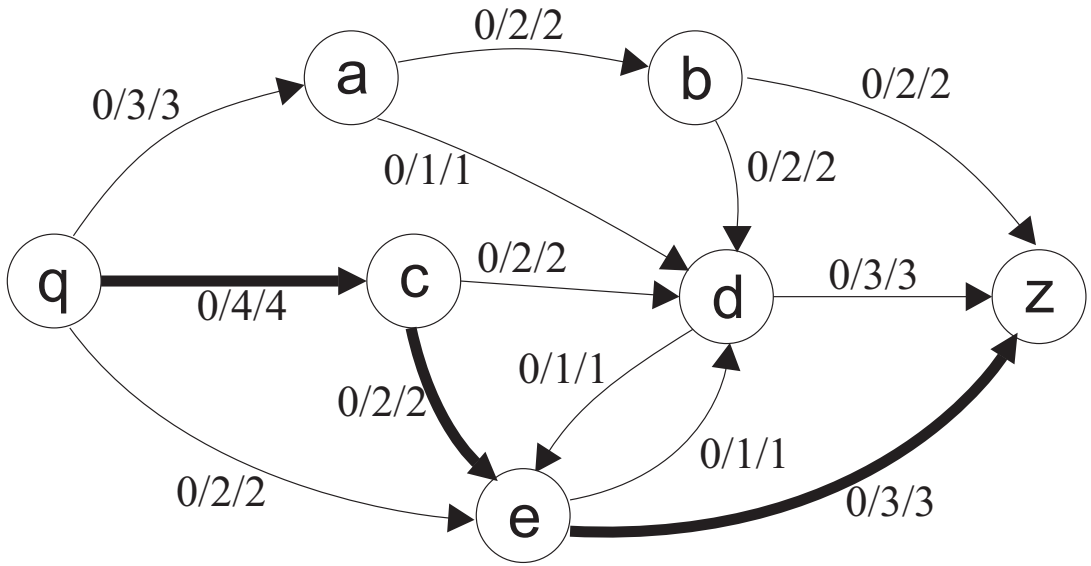
Ford-Fulkerson-Algorithmus I

Graph mit Kapazitäten mit leerem Fluss



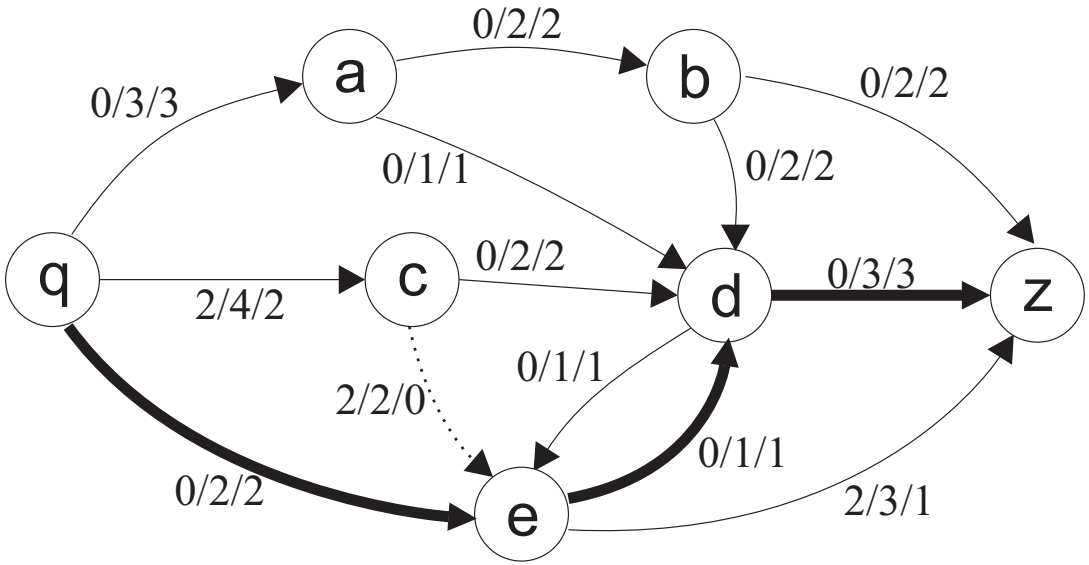
Ford-Fulkerson-Algorithmus II

- erster Pfad: $q \rightarrow c \rightarrow e \rightarrow z$
- nutzbare Kapazität 2



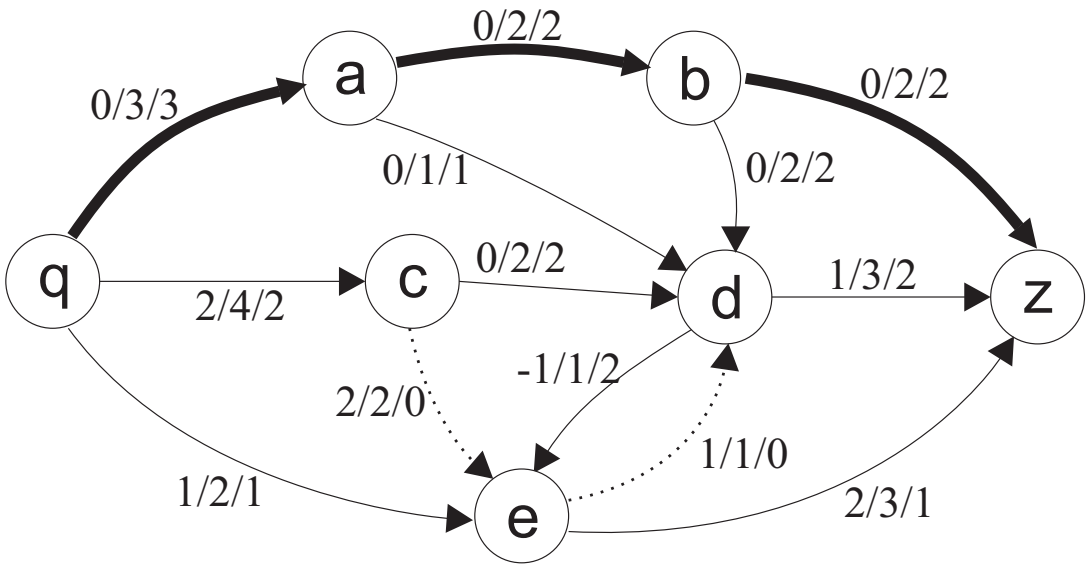
Ford-Fulkerson-Algorithmus III

- Adjustierung der Kantenbeschriftung
  - erschöpfte Kanten werden gestrichelt dargestellt
- Auswahl des zweiten Pfades

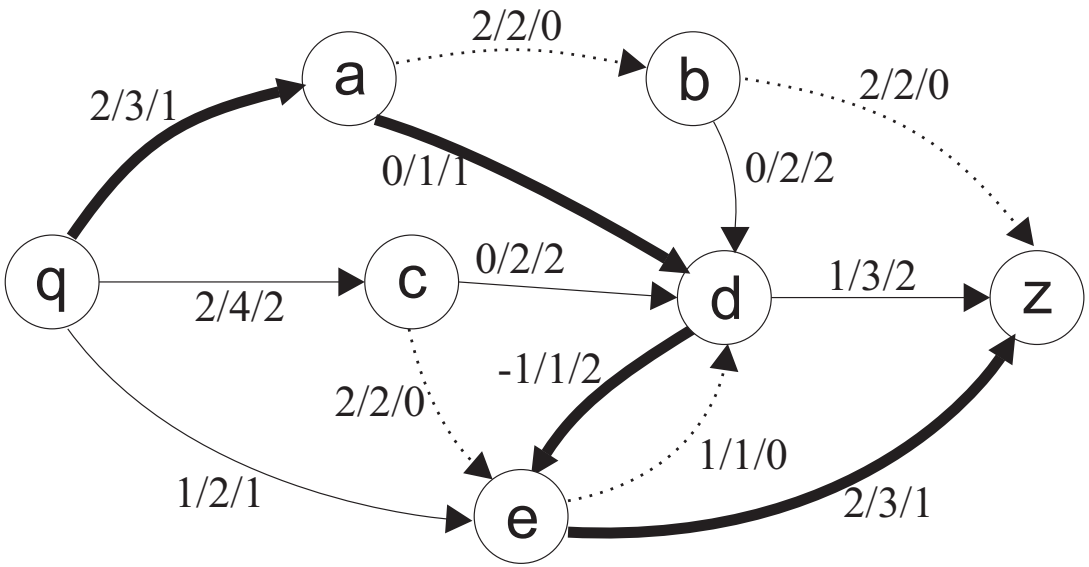


Ford-Fulkerson-Algorithmus IV

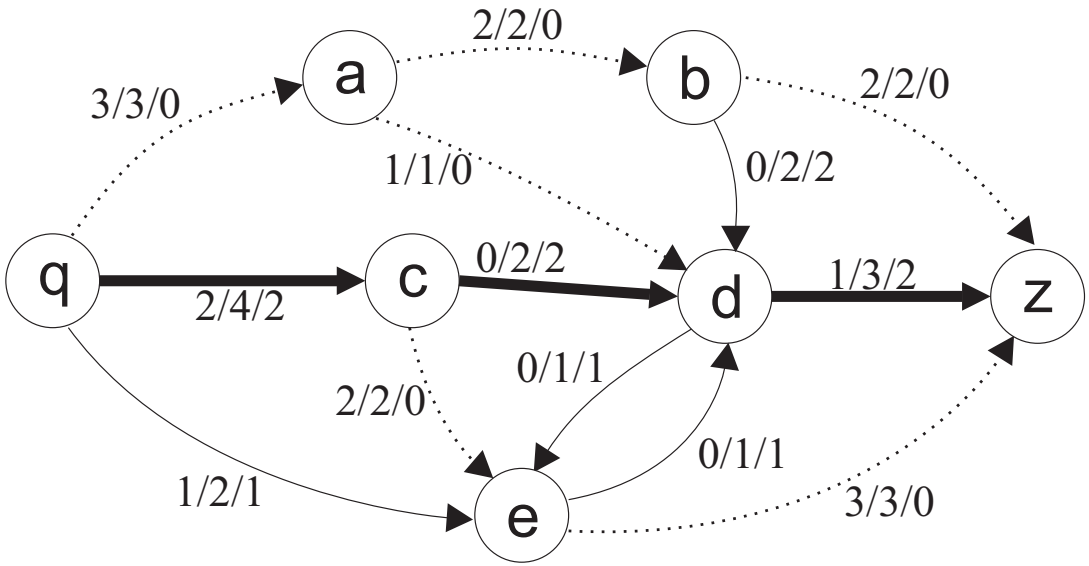
- u.s.w.
- Besonderheit: Adjustierung einer Rückkante mit negativem Wert!



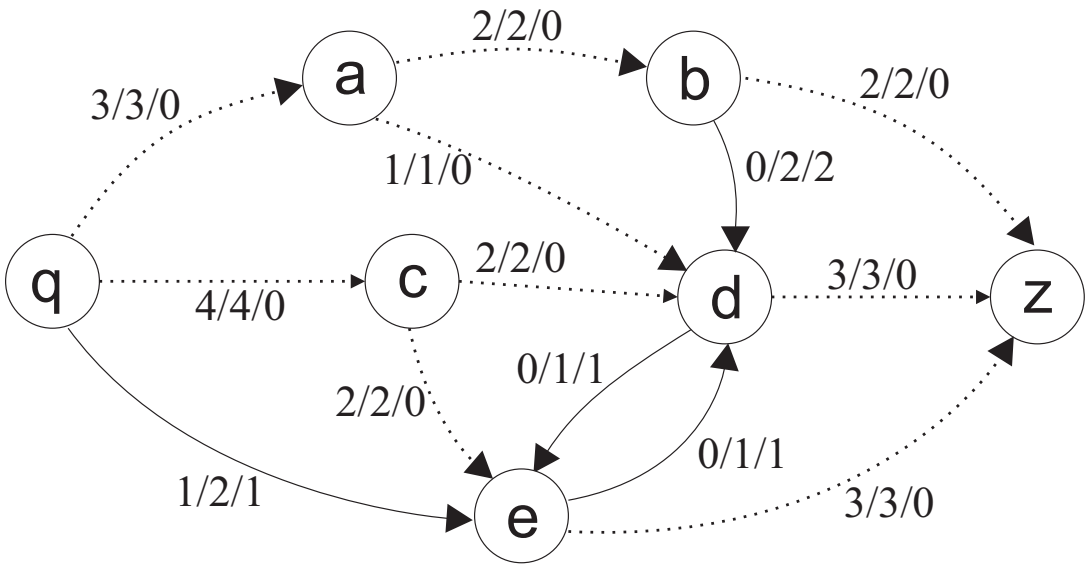
Ford-Fulkerson-Algorithmus VI



Ford-Fulkerson-Algorithmus VII



Ford-Fulkerson-Algorithmus: Abschluss



Weitere Fragestellungen $\rightsquigarrow$ Diskrete Strukturen II

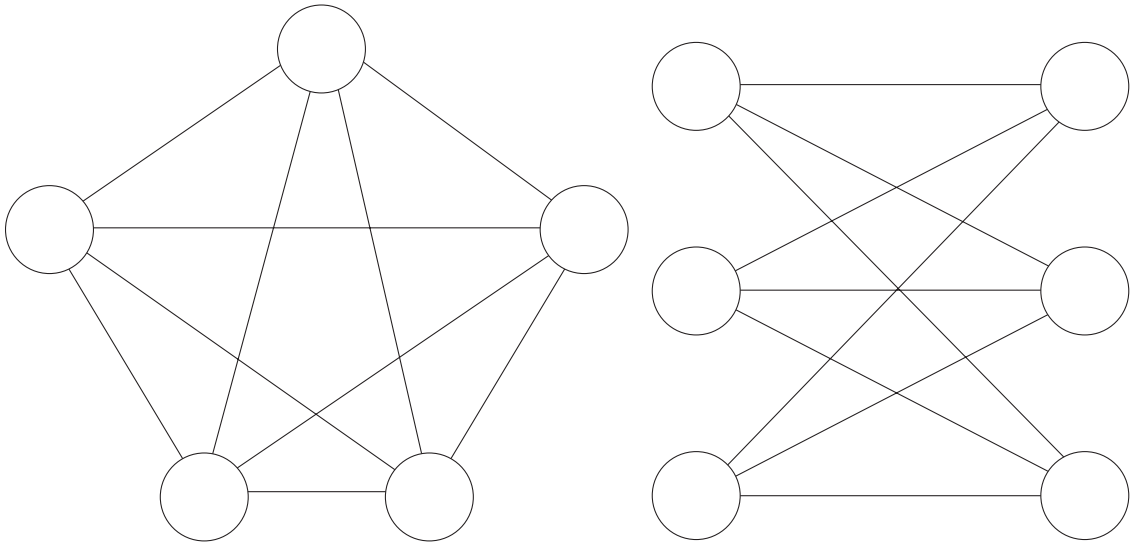
- Handlungsreisender (Traveling Salesman, TSP)
- Planare Graphen
  - sind gegebene Graphen zeichenbar ohne überschneidende Kanten?
- Einfärben von Graphen
  - benachbarte Knoten haben verschiedene Farben
  - Einfärbealgorithmus?
  - Bestimmung der minimal benötigten Anzahl von Farben
- Eulerkreise und Eulerzüge

Planare Graphen

- Grundproblem: Ist ein Graph planar?
- weitere Fragestellungen:
  - Finden „schöner“ planarer Darstellungen
    - Winkel zwischen Kanten an Knoten nicht zu klein
    - Längen der Kanten nicht zu stark differieren
    - Knoten gemäß bestimmter geometrischer Strukturen angeordnet
  - Darstellungen mit einer minimalen Anzahl von Kantenüberschneidungen
    - Minimierung der Anzahl von Brücken in Förderanlagen oder auch in elektronischen Schaltungen
  - Zerlegen eines nichtplanaren Graphen in minimale Anzahl planarer Teilgraphen
    - im Chip-Design Leiterbahnen in mehreren Ebenen anordnen
- typisches Anwendungsgebiet: Chip-Design, Leiterplatten, ...

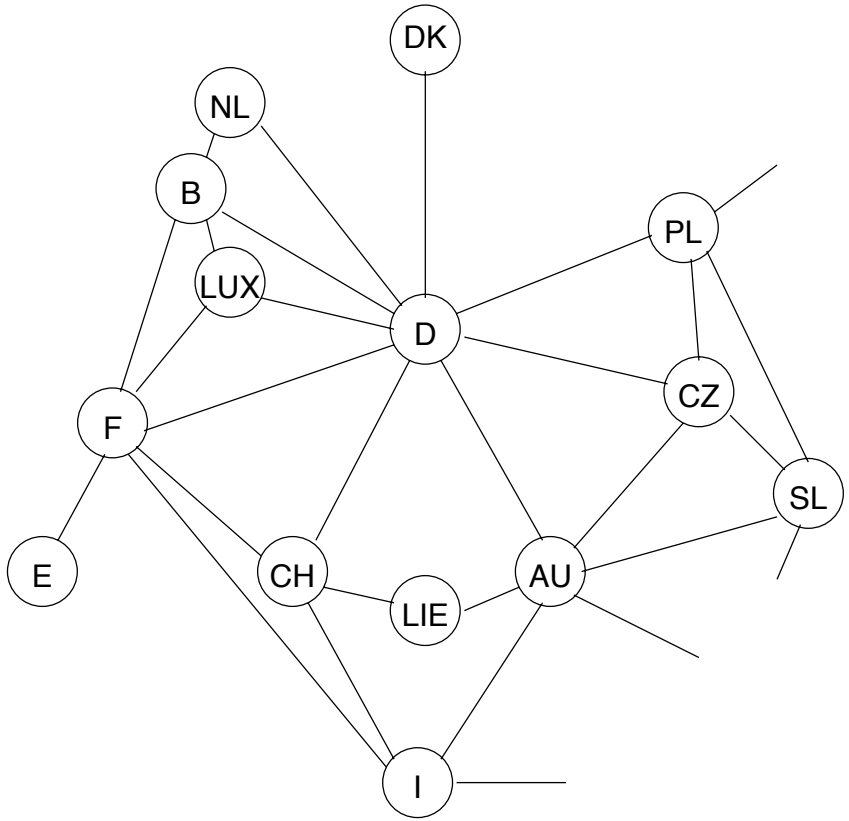
K_5 und $K_{3,3}$

- Graph ist planar wenn er keine (isomorphen) Teilgraphen $\mathbf{K}_5$ und $\mathbf{K}_{3,3}$ enthält



Ländernachbarschaft

- Einfärben von Landkarten als Graph-Einfärbe-Problem



Eulerkreise und Eulerzüge

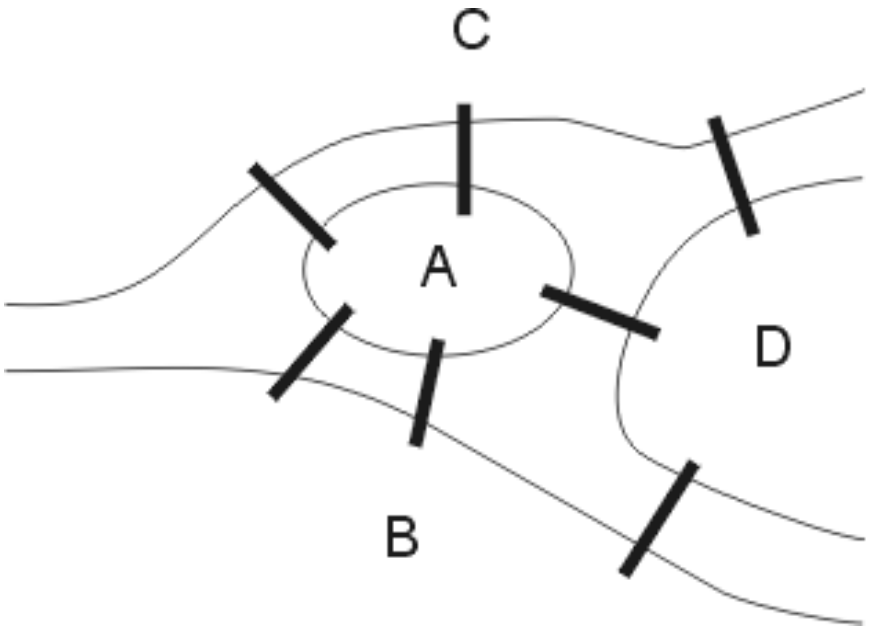
Eulerkreis: Zyklus, der alle Kanten eines Graphen genau einmal durchläuft.

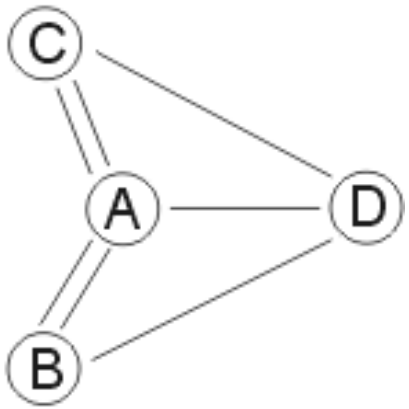
- Fragestellung: kann man alle Kanten eines Graphen abgehen, so dass man am Ende am Ausgangspunkt ankommt, ohne einen Weg zweimal gegangen zu sein
- Beispiel: Königsberger Brückenproblem

Eulerzug oder Eulerweg: Pfad, der alle Kanten eines Graphen genau einmal durchläuft

- Fragestellung: kann man einen Graphen zeichnen, ohne abzusetzen
- Beispiel: Haus vom Nikolaus

Königsberger Brückenproblem





Königsberger Brückenproblem

```
function Eulertour (Graph F)
    Tour := (s), für beliebigen Anfangsknoten s aus F
    while es gibt einen Knoten u in der Tour,
    von dem noch eine Kante ausgeht do
        v := u
        repeat
            Nimm eine Kante v-w, die in v beginnt;
            Füge w hinter v in Tour ein;
            v := w
            Entferne die Kante aus F
        until v = u
    endwhile
    return Tour
end
```