

12.1 Graphen

12.2 Pfadsuchprobleme

12.3 Tiefen- und Breitensuche

12.4 Kürzeste Pfade

12.5 Minimaler Spannbaum

12.6 Maximaler Durchfluss

12.7 Matching



Innensicht

Algorithmenentwurf und Algorithmen (incl. Aufwände)

- Suche und Sortieren (Reihen, Folgen, Bäume)
- Graphen
- Dyn. Programmieren
- Geometr. Algorithmen

Prozesse

- Prozesse
- Prozesskommunikation
- Parallelität/Synchronisation
- Java: Thread-Programmierung

Außensicht

Skalierbarkeit und Persistenz

- Mengenorientierung
- Assoziativer Zugriff, Schlüsselbegriff
- Aufwände
- Polymorphie
- Schema
- Deskr. Sprachen (SQL)

Autonome Dienste

- Enge/lose Kopplung
- Protokolle / Automaten
- Verteilung



Ungerichtete Kante: Ungeordnetes Tupel $[v, w] \in E$. 

Gerichtete Kante: Geordnetes Tupel $(v, w) \in E$. 

Reflexive Kante (Schlinge, ungerichtet/gerichtet): 

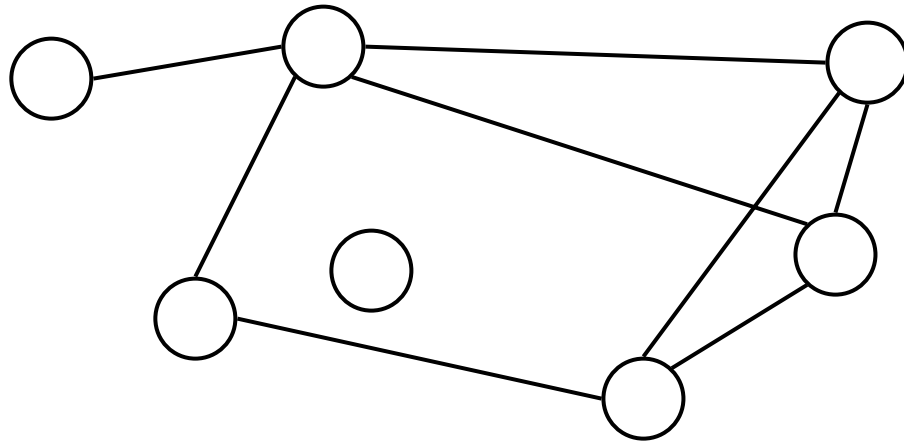
Mehrfache Kanten (multigraph vs. simple): 

Knoten haben Eingangsgrad und Ausgangsgrad:



Ungerichteter Graph

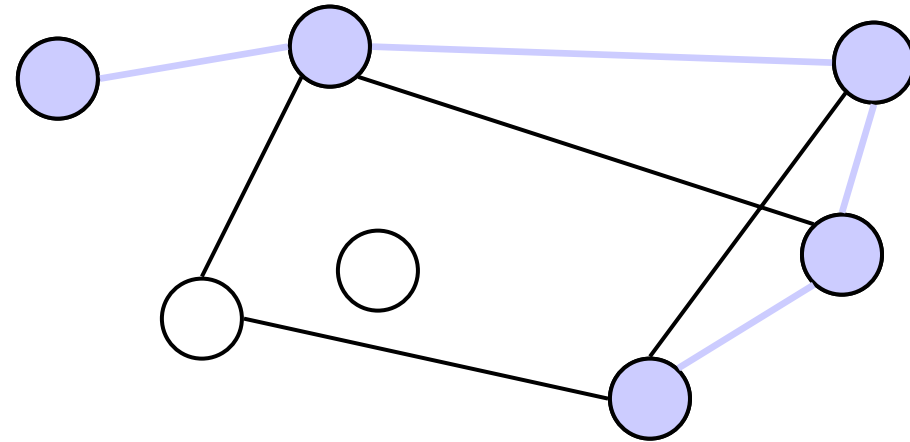
$G = (V, E)$ Kanten in E ungerichtet.



Pfad (Weg, Kantenzug):

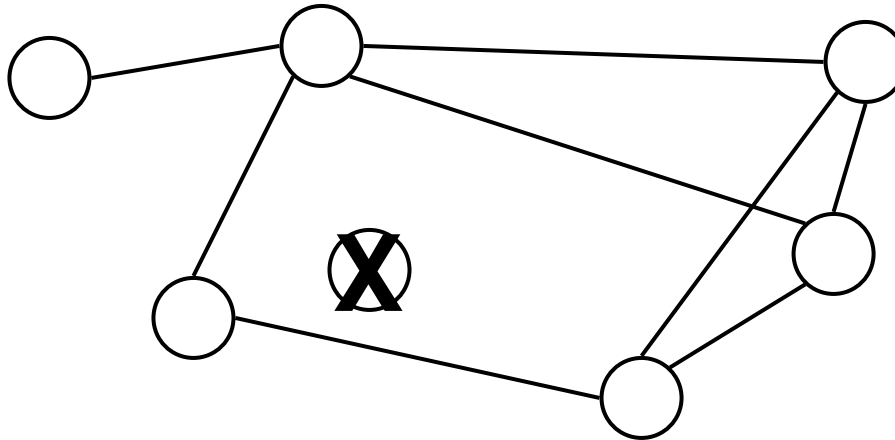
Endliche Folge v_1, \dots, v_n von Knoten, so dass für alle $1 \leq k < n$ $[v_k, v_{k+1}] \in E$.

Der Pfad verbindet v_1 und v_n , v_n ist erreichbar von v_1 .

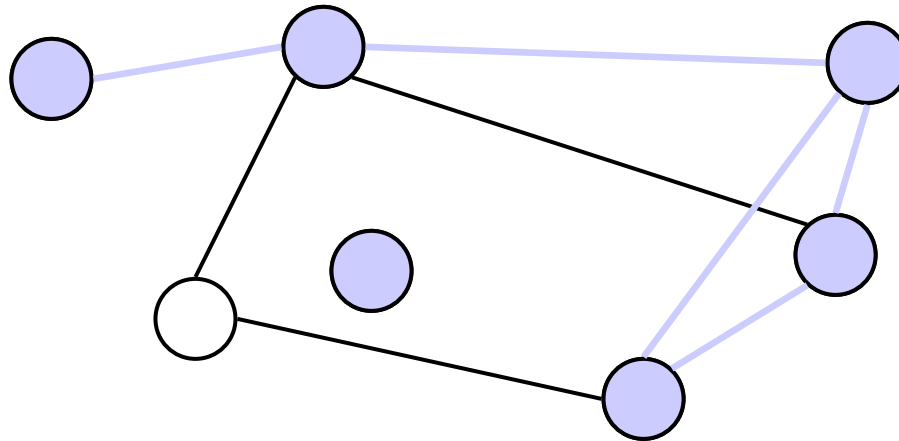


Zusammenhängender (verbundener) Graph:

Es gilt für alle $v, v' \in V$: es existiert ein Pfad, der v und v' verbindet.



Ein Graph $G'=(V',E')$ ist ein Teilgraph eines Graphen $G=(V,E)$ gdw $V' \subseteq V$ und $E' \subseteq E$.

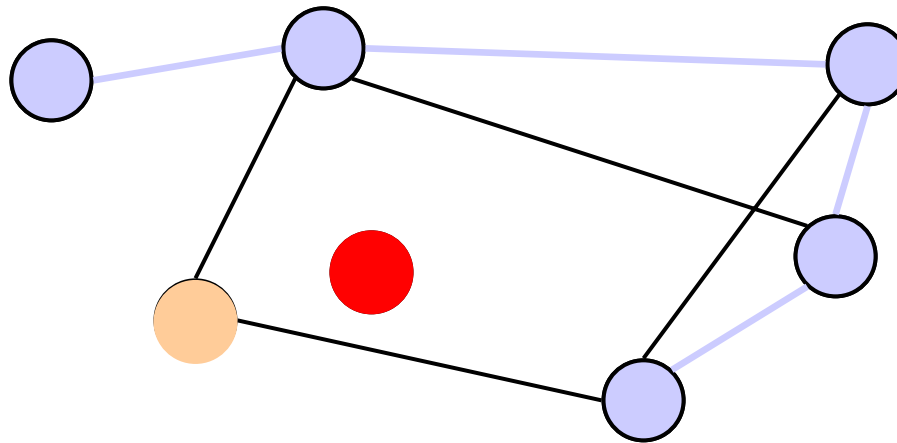


Ein Graph $G'=(V',E')$ ist ein induzierter Teilgraph eines Graphen $G=(V,E)$ gdw. $V' \subseteq V$ und $E' = \{ [u,v] \in E \mid u,v \in V' \}$

Eine Familie $G_i = (V_i, E_i)$ mit $i \in \{1, \dots, n\}$ von Teilgraphen heißt Partitionierung eines Graphen $G = (V, E)$ gdw.

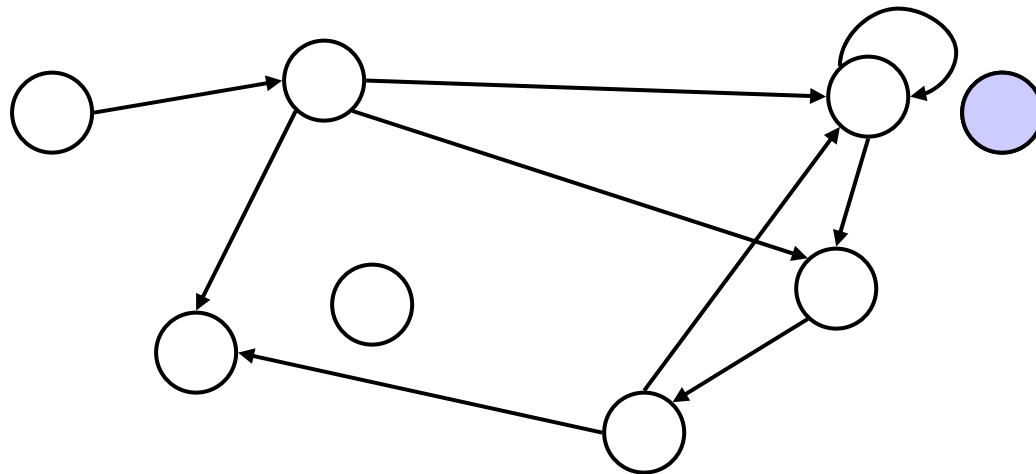
1. jeder Graph G_i ist zusammenhängend und
2. $\forall i, j \in \{1, \dots, n\}, i \neq j \Rightarrow V_i \cap V_j = \emptyset$
3. $\bigcup_{i=1 \dots n} V_i = V$

Jeder Graph G_i wird Komponente von G genannt.



Gerichteter Graph

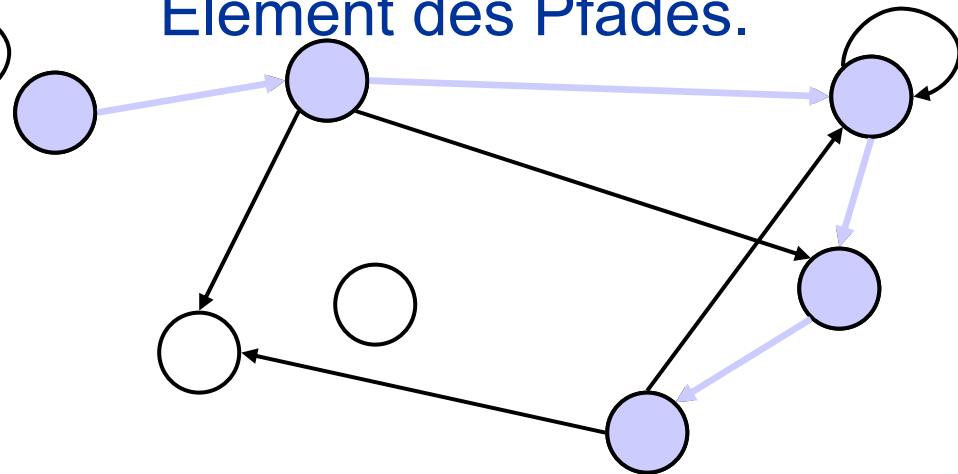
$G = (V, E)$: Kanten in E gerichtet.



Pfad (Weg, Kantenzug):

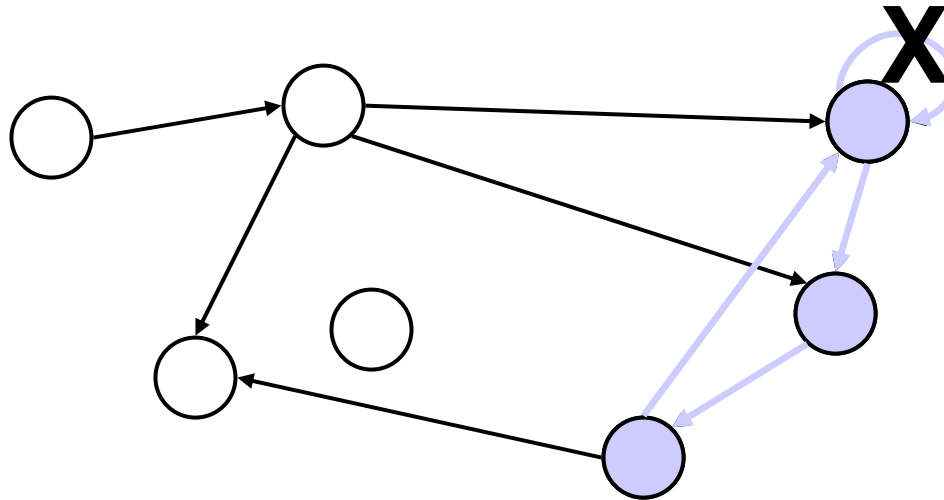
Endliche Folge v_1, \dots, v_n von Knoten, so dass für alle $1 \leq k < n$ $(v_k, v_{k+1}) \in E$.

Der Pfad verbindet v_1 und v_n . Jedes Paar ist ein Element des Pfades.



Zyklus: Pfad mit $v_n = v_1$, $n \geq 1$.

Minimaler Zyklus: Es gibt keinen echten Teilpfad v_i, \dots, v_j mit $1 \leq i, j \leq n$, der Zyklus ist.

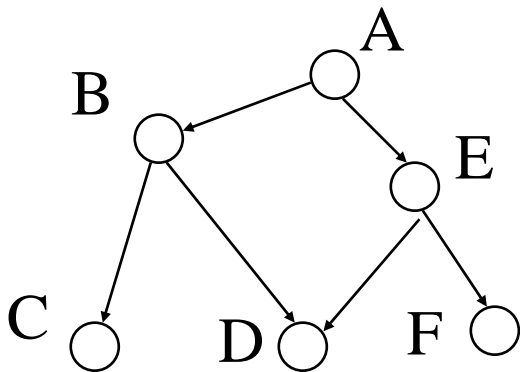


Ist $(v, v') \in E$ so heißt v direkter Vorgänger (Elter) von v' und v' direkter Nachfolger (Kind, Sohn) von v .

■ Auch als $v \rightarrow v'$ notiert.

Falls ein Pfad von v nach v' führt, so heißt v Vorgänger (Vorfahr) von v' und v' Nachfolger (Nachkomme) von v .

■ Auch als $v \rightarrow^* v'$ notiert.



A ist Vorfahr von allen anderen Knoten

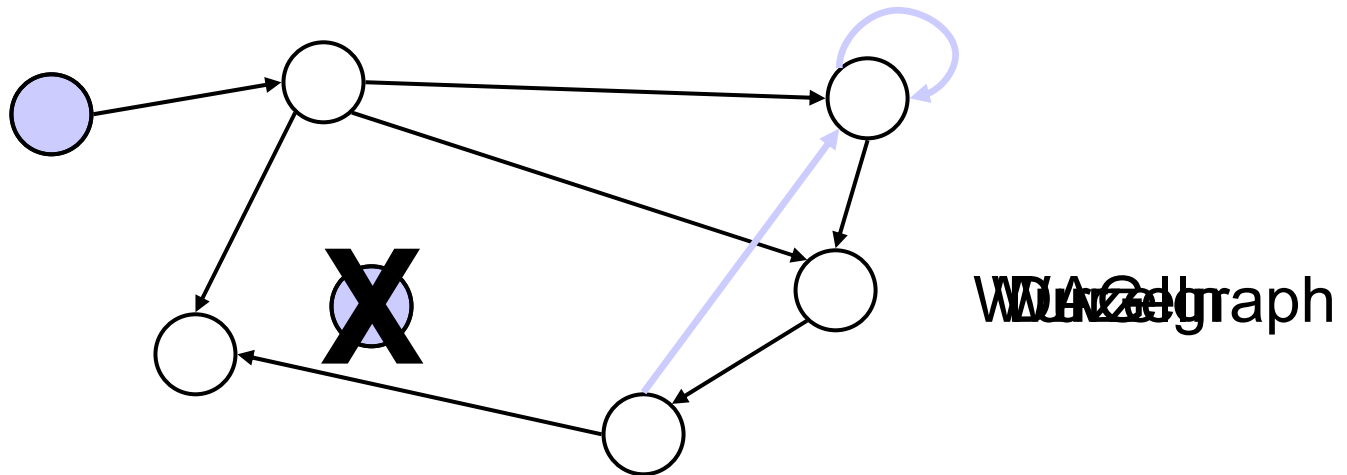
C und D sind Kinder von B

B und E sind Eltern von D

F ist Nachkomme von A und von E



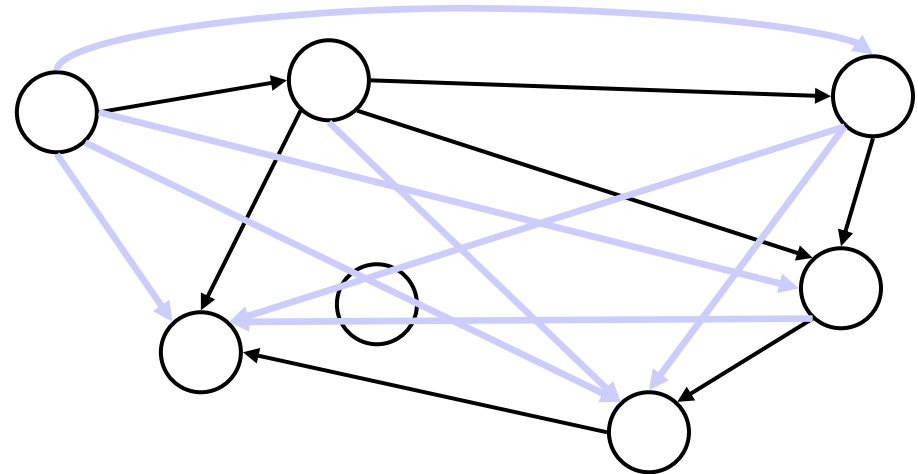
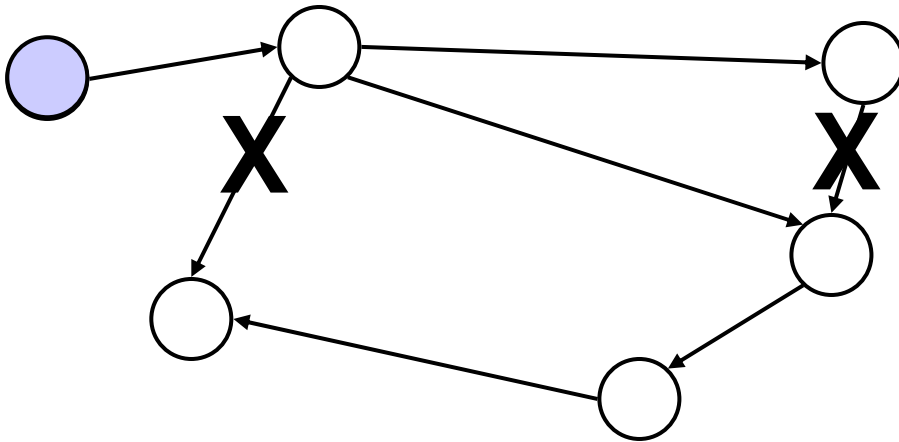
- Gerichteter azyklischer Graph (DAG, *directed acyclic graph*): Gerichteter Graph ohne Zyklen.
- Ein Knoten v eines DAG heißt Wurzel, falls es keine auf ihn gerichteten Kanten gibt. Hat ein DAG nur eine Wurzel, so heißt er Wurzelgraph.



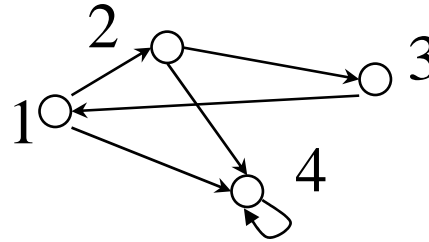
Baum: Wurzelgraph, in dem zu jedem Knoten genau ein Pfad von der Wurzel aus führt.

Wald: DAG mit mehreren Wurzeln, aber eindeutigen Pfaden.

Transitive Hülle $G^+ = (V, E)$
eines Graphen $G = (V, E)$:
 $V \rightarrow_{G^+} V' \iff V \rightarrow_G^* V'$.



Graphische Darstellung:

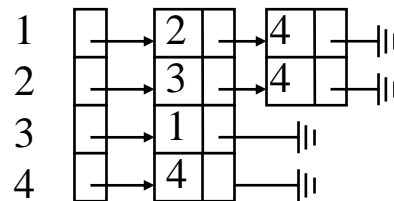


Mengenschreibweise:

$G = (V, E)$ mit $x \circ \longrightarrow \circ y \Rightarrow (x, y) \in E \subseteq V \times V$.

$V = \{1, 2, 3, 4\}$ $E = \{(1, 2), (1, 4), (2, 3), (2, 4), (3, 1), (4, 4)\}$

Adjazenzlisten:

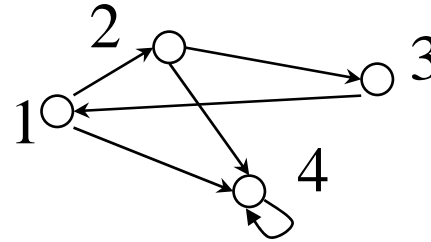


Matrixdarstellung:
(Adjazenzmatrix)

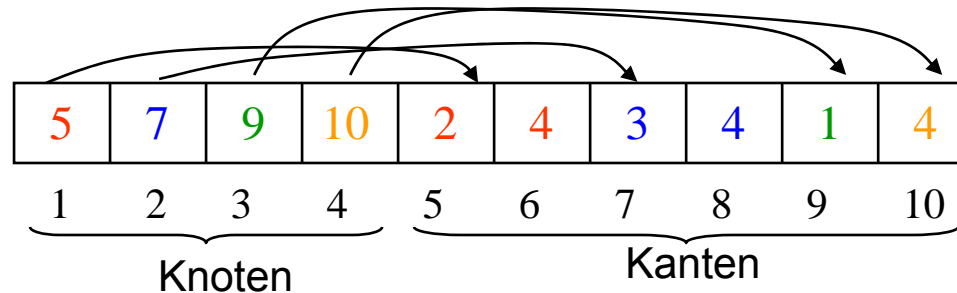
| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 |
| 2 | 0 | 0 | 1 | 1 |
| 3 | 1 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 1 |



Graphische Darstellung:



Als Feld:



- Graphik: + sehr leicht lesbar für Menschen
- extrem kompliziert für maschinelle Bearbeitung
- Mengen: + geeignet für einige mathematische Operationen
- teure Suche nach Knoten und Kanten
- Matrix: + sehr schnell feststellbar, ob eine Kante existiert
+ manche Graphenoperationen lassen sich als Matrizenoperationen darstellen
- Speicherverschwendung bei dünnen Graphen
- Liste: + sehr kompakte Darstellung (wenig Speicher)
- kostspielige Suche nach Kanten
- Feld: + noch kompakter als Liste
- aufwändige Änderungen (Einfügen/Löschen)



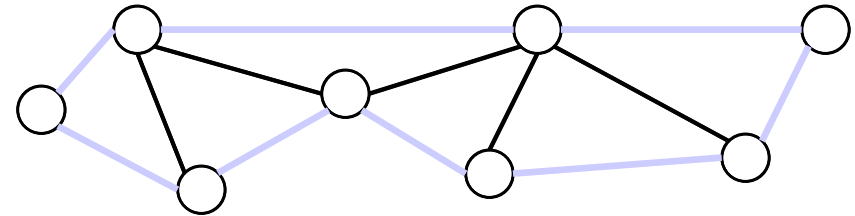
| Operation | Kantenliste | Knotenliste | Adjazenz- matrix | Adjazenz- liste |
|--------------------|-------------|-------------|---------------------|--------------------|
| Einfügen Kante | $O(1)$ | $O(n+m)$ | $O(1)$ | $O(1)/O(n)$ |
| Löschen Kante | $O(m)$ | $O(n+m)$ | $O(1)$ | $O(n)$ |
| Einfügen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



Typische Probleme (1): Knoten interessieren

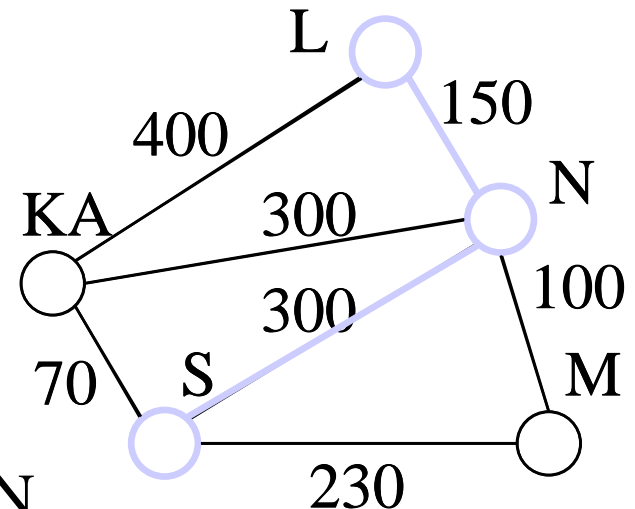
Suche nach einem geschlossenen Pfad durch gegebene Knoten:

- Wie besucht ein Handlungsreisender alle seine Kunden mit einer Rundreise?



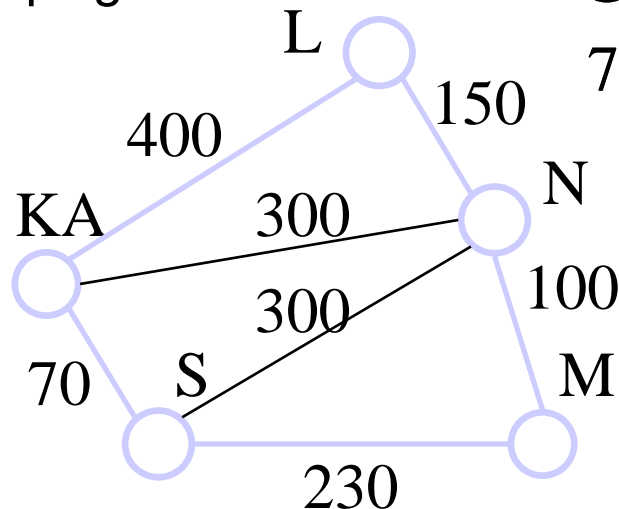
Suche nach einem „billigen“ Pfad:

- Wie komme ich am billigsten von Stuttgart nach Leipzig?



oder beides:

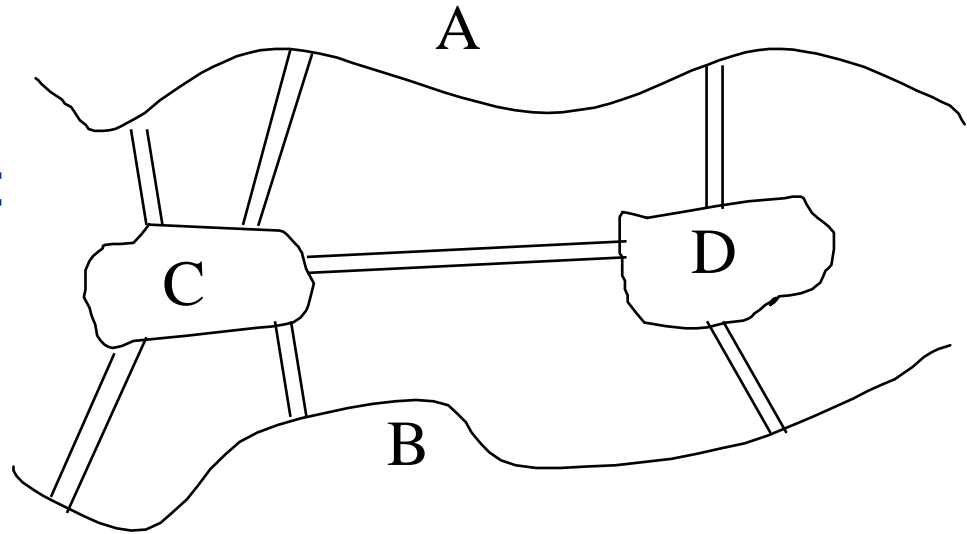
- Billigste Rundreise?



Typische Probleme (2): Kanten interessieren

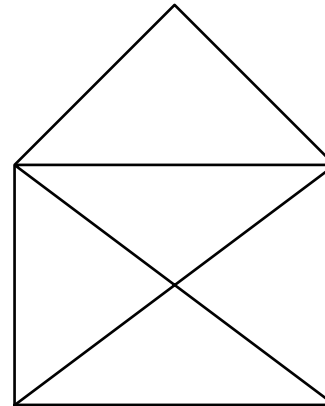
Königsberger Brückenproblem:

- Gibt es einen geschlossenen Pfad, der über alle 7 Brücken führt?



Zeichenproblem:

- Kann das Häuschen mit einem Strich gezeichnet werden?

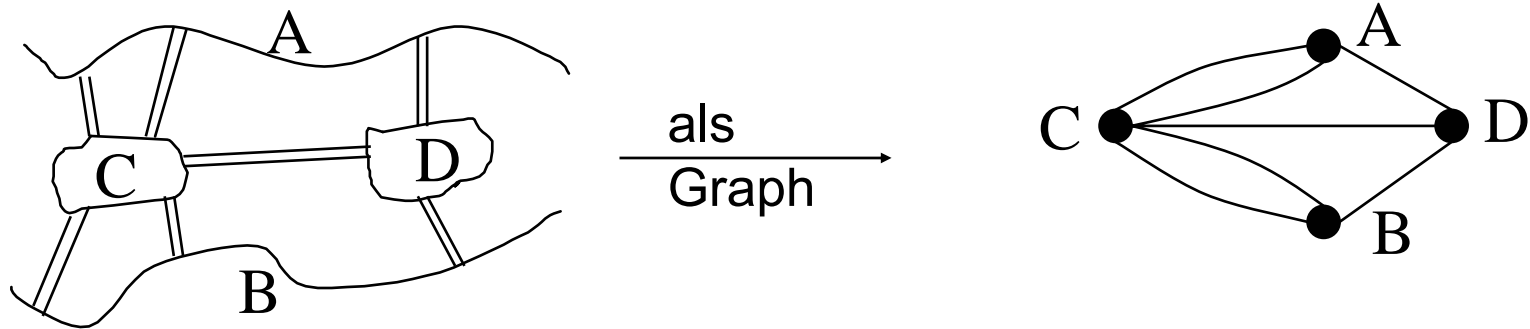


Eulerscher Pfad:

- Pfad, der alle Kanten umfasst und jede genau einmal durchläuft.

Eulerscher Zyklus:

- Eulerscher Pfad, bei dem Start- und Endknoten identisch sind.



Die Frage, ob ein Eulerscher Zyklus existiert, ist leicht zu beantworten.

- Feststellung: Wenn man in einen Knoten kommt, muss man auf anderem Weg wieder herauskommen.
- Also: Es existiert ein Euler-Zyklus gdw. Grad jedes Knotens durch 2 teilbar ist und der Graph zusammenhängend ist.

Gegeben: Ein Graph G
Aufgabe: Besuche alle Knoten.

Tiefensuche (depth-first search, DFS)

Prinzip: Verfolge einen eingeschlagenen Weg so lange weiter, bis es nicht mehr geht oder angebracht ist.

- Erst wenn eine Richtung komplett durchsucht ist, wird die nächste Alternative in Angriff genommen.
- Ariadne-Algorithmus war die erste Realisierung einer Tiefensuche.

Breitensuche (breadth-first search, BFS)

Prinzip: Besuche zuerst alle direkten Nachfolger und besuche danach deren Nachkommen.

- Betrachte zuerst alle Alternativen und verfolge dann jeweils die Richtung weiter.



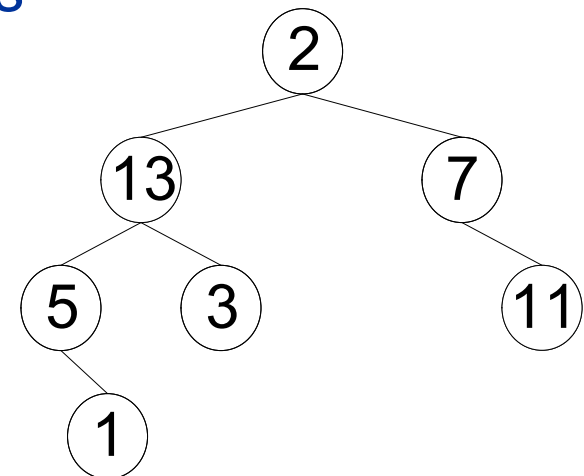
Verfahren zum Durchlaufen aller Knoten eines Baumes. Man geht solange „links-abwärts“ wie möglich und dann wieder zurück.

- Es gilt: Ein rechter Sohn wird erst untersucht, wenn der Unterbaum des linken Sohnes vollständig abgearbeitet wurde.
- Durchlauf vergleichbar Iterator aus Kapitel 11.

Dieses Verhalten kann durch einen Keller implementiert werden.

Beispiel: Tiefensuche liefert das Ergebnis

[2, 13, 5, 1, 3, 7, 11]



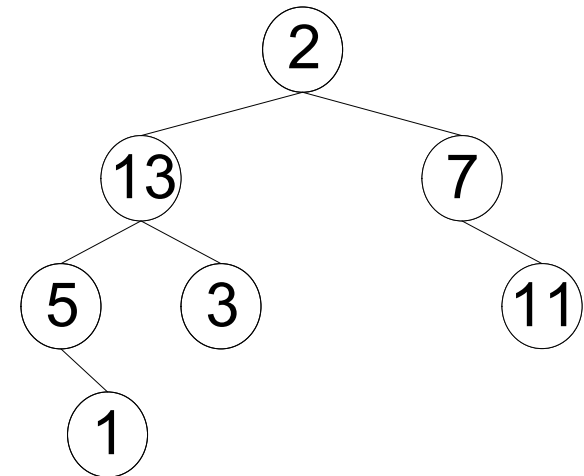
Verfahren zum Durchlaufen aller Knoten eines Baumes. Hier werden die Knoten „zeilenweise von rechts nach links“ durchlaufen.

- Es gilt: Ein Knoten der Tiefe $k+1$ wird erst untersucht, wenn alle Knoten der Tiefe k abgearbeitet sind.

Dieses Verhalten kann durch eine FIFO-Schlange implementiert werden.

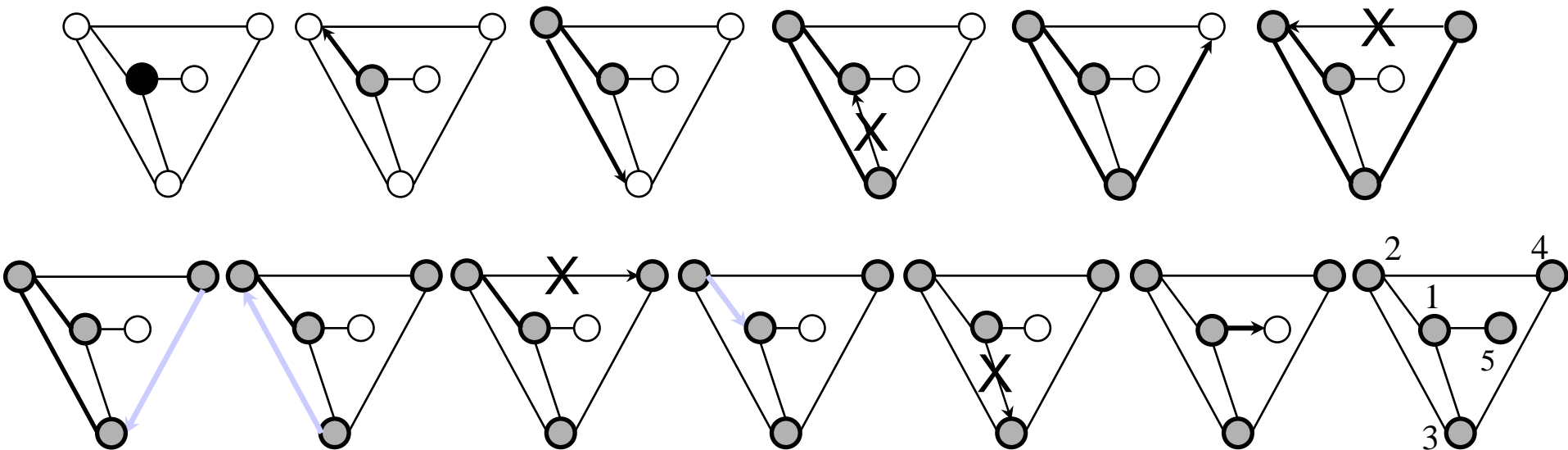
Beispiel: Breitensuche liefert das Ergebnis

[2, 7, 13, 11, 3, 5, 1]



Algorithmus

METHODE Tiefensuche (v)
 markiere aktuellen Knoten v als besucht
 für jede von v abgehende Kante (v, w)
 wenn der Knoten w noch nicht markiert ist
 Tiefensuche (w)



Algorithmus

METHODE Breitensuche(v)

markiere v als besucht

initialisiere Liste mit einem Element v

solange die Liste nicht leer ist

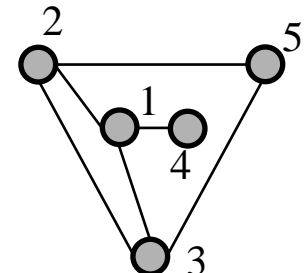
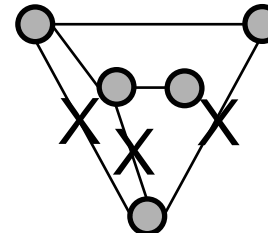
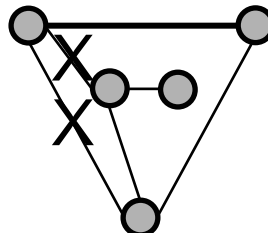
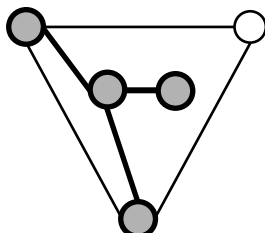
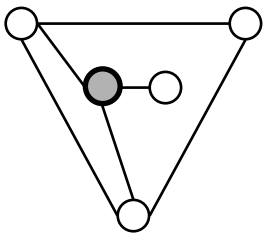
entferne erstes Element w aus der Liste

für alle Kanten (w, x)

falls x noch nicht markiert

hänge x hinten an die Liste an

markiere x als besucht



Eigentliche Arbeit in der Tiefensuche:

METHODE Tiefensuche (v)

markiere aktuellen Knoten v als besucht

erledige Arbeit vor dem Abstieg

für jede von v abgehende Kante (v, w)

wenn der Knoten w noch nicht markiert ist

Tiefensuche (w)

erledige Arbeit während des Abstiegs

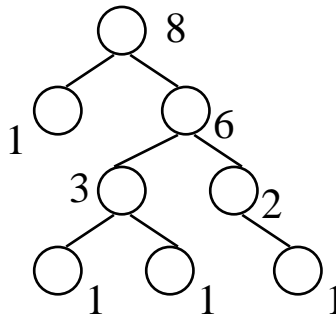
erledige Arbeit nach dem Abstieg

Welche Arbeiten zu erledigen sind, hängt von der Anwendung ab.



Gegeben: Ein Baum G

Gesucht: Bestimme für jeden Knoten V Anzahl der Knoten des Unterbaums, der V als Wurzel hat



METHODE Tiefensuche (v)

markiere aktuellen Knoten v als besucht

Arbeit vor dem Abstieg: **Zähler(v) = 1**

für jede von v abgehende Kante (v, w)

wenn der Knoten w noch nicht markiert ist

Tiefensuche (w)

Arbeit während des Abstiegs: **Zähler(v) += Zähler(w)**

Arbeit nach dem Abstieg: **nichts mehr**



Beispiel für Tiefensuche

METHODE Tiefensuche (v)

markiere aktuellen Knoten v als besucht

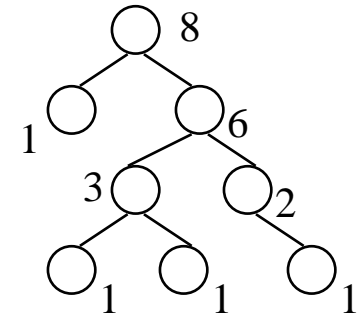
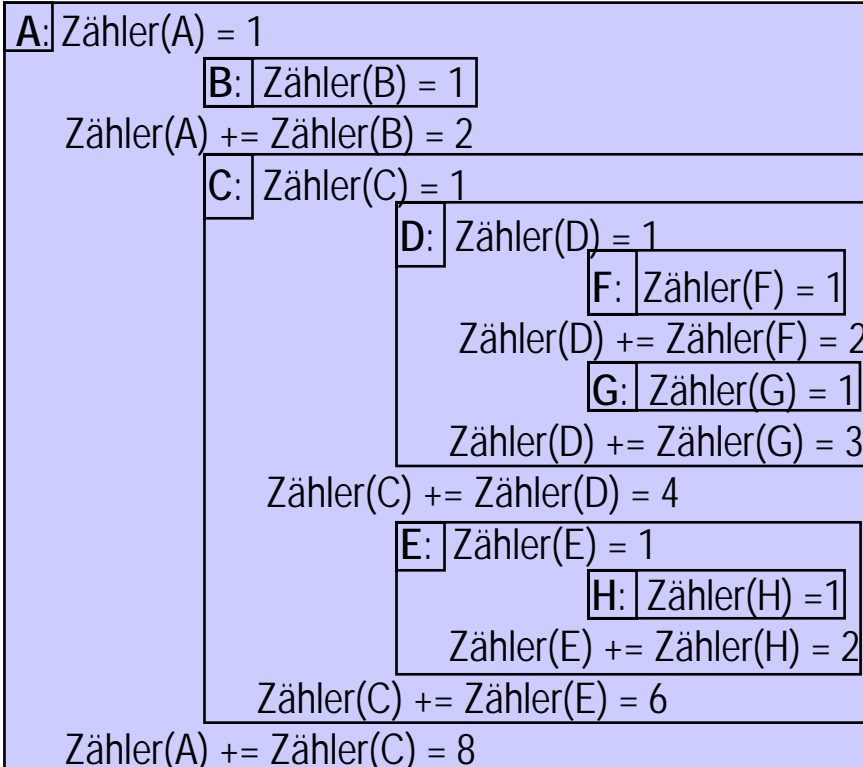
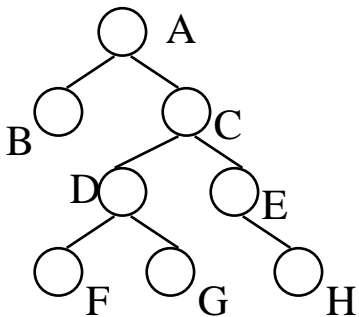
Arbeit vor dem Abstieg: **Zähler(v) = 1**

für jede von v abgehende Kante (v, w)

wenn der Knoten w noch nicht markiert ist

Tiefensuche (w)

Arbeit während des Abstiegs: **Zähler(v) += Zähler(w)**



Bemerkungen

- Wenn ein ungerichteter Graph zusammenhängend ist, wird sowohl mit Tiefensuche als auch mit Breitensuche garantiert jeder Knoten besucht (egal wo man anfängt).
- Wenn ein Graph nicht zusammenhängend ist, muss der Suchalgorithmus erweitert werden zu:

```
METHODE SucheAlles
    solange ein nicht markierter Knoten  $v$  existiert
        Suche( $v$ )
```

Aufwand der Tiefen-/Breitensuche

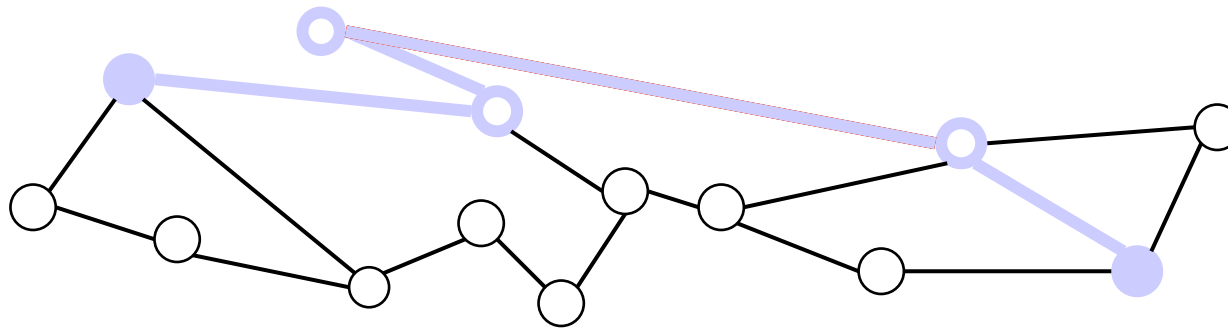
- Jeder Knoten wird einmal besucht, jede Kante höchstens einmal *in jeder Richtung* durchlaufen.
⇒ auch hier $O(|E| + |V|)$.



Suche nach dem kürzesten Pfad

Gegeben: Ein Graph, zwei Knoten v und w .

Gesucht: Der (nach Anzahl der Kanten) kürzeste Pfad von v nach w .



Verallgemeinerung

- Pfadlänge nicht durch Kantenzahl bestimmt, sondern durch Summe der Kantenmarkierungen (die alle positiv sein sollen).

Beispiel

- Gegeben: Eine Straßenkarte mit Entfernungsangaben zwischen Kreuzungen
- Aufgabe: Finde die kürzeste Route von Karlsruhe nach Berlin



Wir betrachten Distanzgraph: $G = (V, E)$ mit $c: E \rightarrow \mathbb{R}_0^+$.

Gerichtete Graphen.

- Falls ungerichtet, ersetze jede Kante durch zwei gerichtete.

Gewichtete Kanten.

- Falls ungewichtet, betrachte jedes Kantengewicht als 1.0.

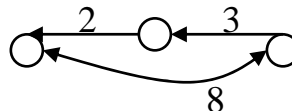
Aufgabe:

Gegeben: Knoten v

Gesucht: Kürzeste Pfade zu allen anderen Knoten

Bemerkung

- In gerichteten Graphen gilt natürlich nicht, dass der kürzeste Pfad von v nach w der gleiche ist wie der von w nach v .



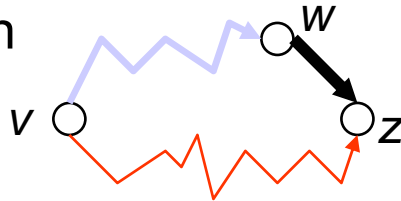
Vorgehensweise

- Iterative Erweiterung einer Menge von "billig" erreichbaren Kanten
- auf dem Greedy-Prinzip (immer das nächstbeste Stück nehmen) basierende Weiterentwicklung der Breitensuche.

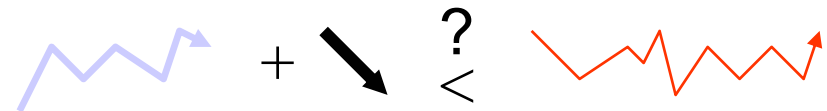
Algorithmus

- markiere alle Knoten als unbesucht
setze $Länge(v) = 0$
setze $Länge(\text{alle anderen Knoten}) = \text{unendlich}$
- solange nicht besuchte Knoten existieren
wähle darunter denjenigen Knoten w , für den $Länge(w)$ minimal
markiere w als besucht
für alle Kanten (w,z) zu unmarkierten Knoten z
falls $Länge(w) + Gewicht(w,z) < Länge(z)$
dann setze $Länge(z) = Länge(w) + Gewicht(w,z)$

Situation



Auswahl nächster Knoten



Beispiel

solange nicht besuchte Knoten existieren

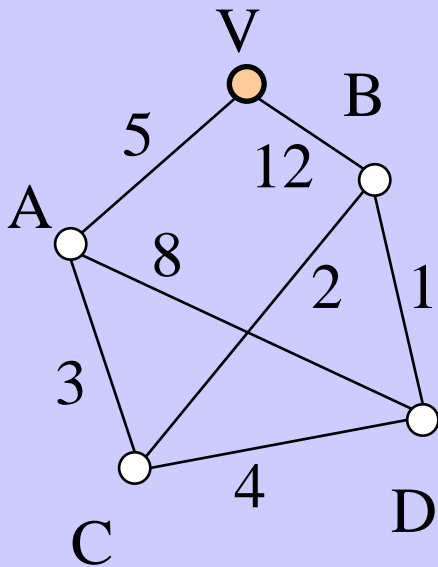
wähle darunter denjenigen Knoten w , für den $Länge(w)$ minimal
markiere w als besucht

für alle Kanten (w,z) zu unmarkierten Knoten z

falls $Länge(w) + Gewicht(w,z) < Länge(z)$

dann setze $Länge(z) = Länge(w) + Gewicht(w,z)$

Beispielgraph:



$Länge(V)=0$, $Länge(A)=\text{unendl.}$, ... $Länge(D)=\text{unendl.}$
 $Länge(A)=5$, $Länge(B)=12$

minimal ist $Länge(A)=5 \Rightarrow V, A$ markiert

$Länge(A) + (A,C) = 5 + 3 < \text{unendl.} \Rightarrow Länge(C)=8$

$Länge(A) + (A,D) = 5 + 8 < \text{unendl.} \Rightarrow Länge(D)=13$

minimal ist $Länge(C)=8 \Rightarrow V, A, C$ markiert

$Länge(C) + (C,B) = 8 + 2 < 12 \Rightarrow Länge(B)=10$

$Länge(C) + (C,D) = 8 + 4 < 13 \Rightarrow Länge(D)=12$

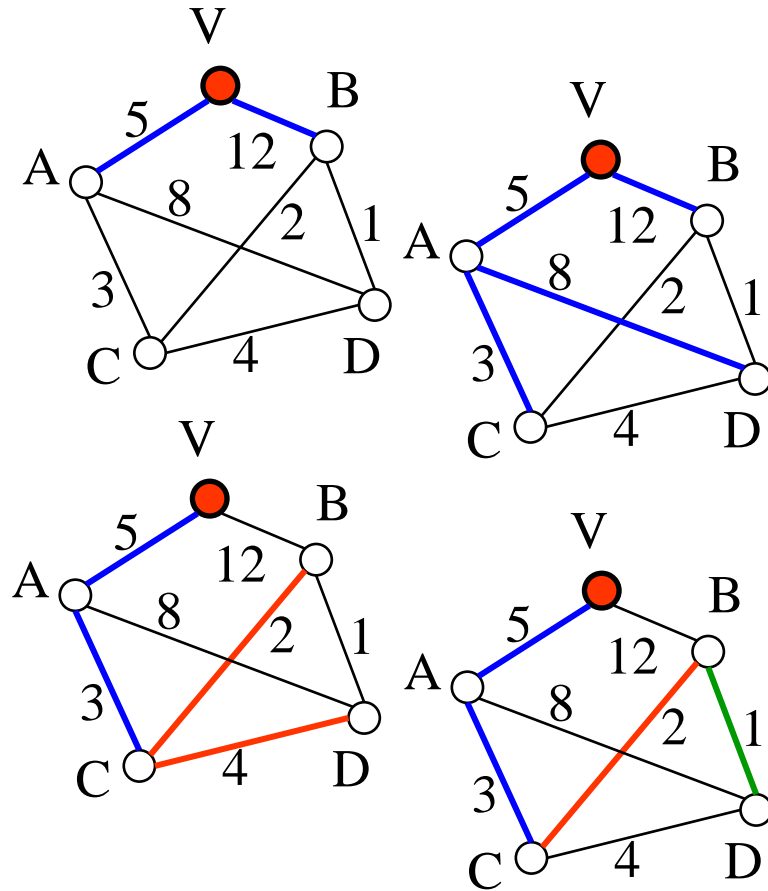
minimal ist $Länge(B)=10 \Rightarrow V, A, C, B$ markiert

$Länge(B) + (B,D) = 10 + 1 < 12 \Rightarrow Länge(D)=11$

minimal ist $Länge(D)=11 \Rightarrow$ alles markiert



Die abgesuchten Wege bilden einen Baum mit eindeutigem Weg von v zu jedem Knoten.



Länge(V)=0, Länge(A)=unendl., ... Länge(D)=unendl.
Länge(A)=5, Länge(B)=12

minimal ist Länge(A)=5 \Rightarrow V, A markiert
Länge(A) + (A, C) = 5 + 3 < unendl. \Rightarrow Länge(C)=8
Länge(A) + (A, D) = 5 + 8 < unendl. \Rightarrow Länge(D)=13

minimal ist Länge(C)=8 \Rightarrow V, A, C markiert
Länge(C) + (C, B) = 8 + 2 < 12 \Rightarrow Länge(B)=10
Länge(C) + (C, D) = 8 + 4 < 13 \Rightarrow Länge(D)=12

minimal ist Länge(B)=10 \Rightarrow V, A, C, B markiert
Länge(B) + (B, D) = 10 + 1 < 12 \Rightarrow Länge(D)=11

minimal ist Länge(D)=11 \Rightarrow alles markiert



Grundgedanke

- Verlängern eines Pfads durch Hinzunahme einer weiteren Kante

Optimalitätsprinzip

- Für jeden kürzester Pfad $p = (v_0, v_1, \dots, v_k)$ von v_0 nach v_k ist jeder Teilpfad $p' = (v_i, \dots, v_j)$, $0 \leq i < j \leq k$, ein kürzester Pfad von v_i nach v_j .

Widerspruchsbeweis

- Angenommen es gäbe kürzeren Pfad p'' von v_i nach v_j . Dann kann in p p' durch p'' ersetzt werden, der entstehende Pfad von v_0 nach v_k wäre kürzer als p .

Somit *Invariante* für länger werdende, bereits bekannte kürzeste Pfade durch Hinzunahme einzelner Kanten (*sp*: shortest path)

1. Für alle kürzesten Pfade $sp(s, v)$ und Kanten (v, v') gilt:
$$c(sp(s, v)) + c((v, v')) \geq c(sp(s, v')).$$
2. Für wenigstens einen kürzesten Pfad $sp(s, v)$ und eine Kante (v, v') gilt:
$$c(sp(s, v)) + c((v, v')) = c(sp(s, v')).$$



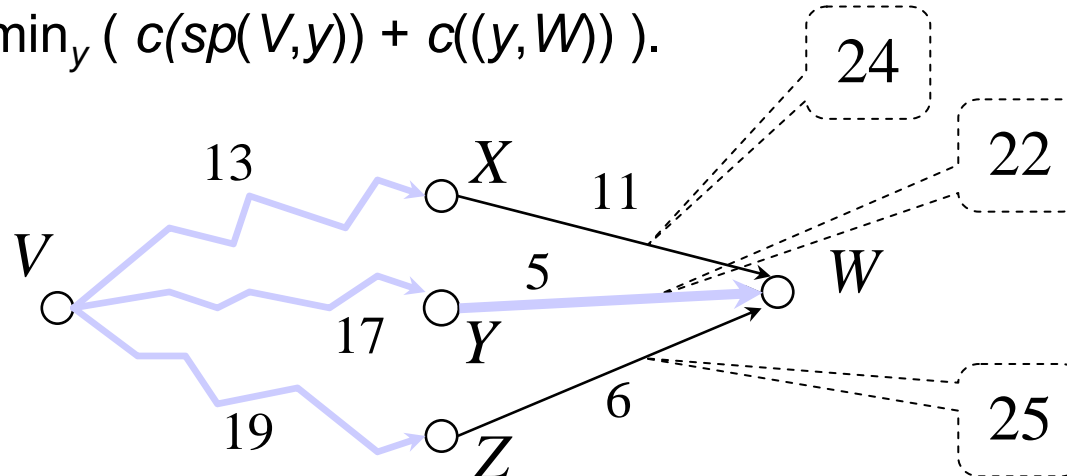
Induktionsbeginn

- Betrachte Knoten a und alle seine abgehenden Kanten.
- Offensichtlich: Wenn (a,b) die kürzeste Kante ist, die von a abgeht, ist (a,b) der kürzeste Pfad von a nach b .

Induktionsschritt

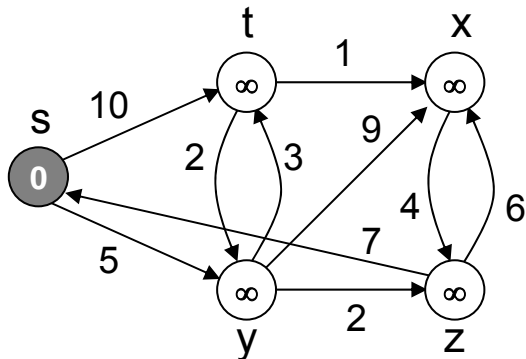
- Wir kennen alle kürzesten Pfade von V zu allen Knoten y , von denen man direkt zu W kommen kann.
 - Beispiel: $c(\text{sp}(V,X)) = 13$, $c(\text{sp}(V,Z)) = 19$
- Dann ist der kürzeste Weg von V nach W gegeben durch:

$$c(\text{sp}(V,W)) = \min_y (c(\text{sp}(V,y)) + c((y,W))).$$

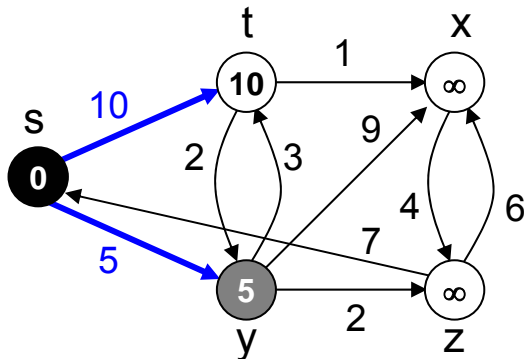




Coreman, Leiserson, Rivest, Stein,
Introduction to Algorithms,
MIT Press, 2001

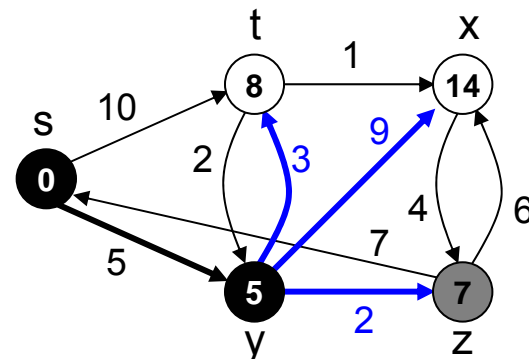


Graph nach Initialisierung
Startknoten: s

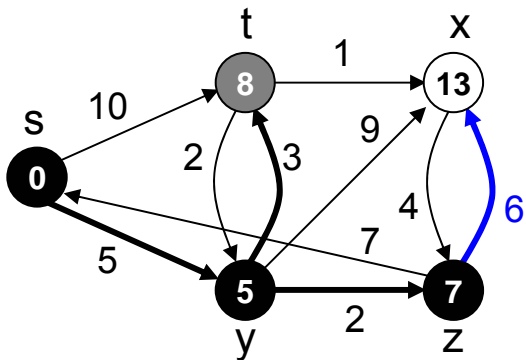


Iteration 1
 $q = [(s \rightarrow 0), (t \rightarrow \infty), (x \rightarrow \infty), (y \rightarrow \infty), (z \rightarrow \infty)]^*$

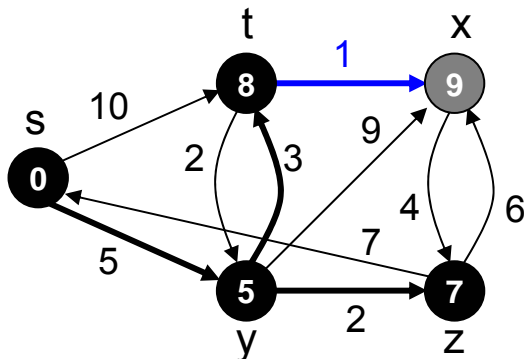
*Zustand vor Iteration



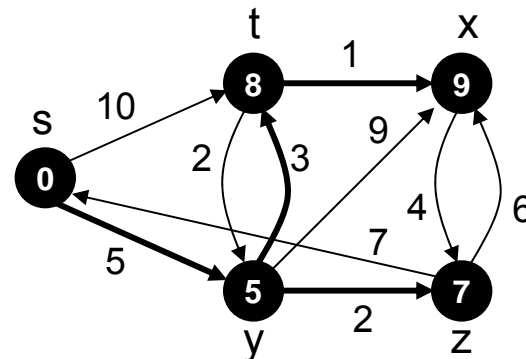
$q = [(y \rightarrow 5), (t \rightarrow 10), (x \rightarrow \infty), (z \rightarrow \infty)]$
Iteration 2



$q = [(z \rightarrow 7), (t \rightarrow 8), (x \rightarrow 14)]$
Iteration 3



$q = [(t \rightarrow 8), (x \rightarrow 13)]$
Iteration 4



$q = [(x \rightarrow 9)]$
Iteration 5

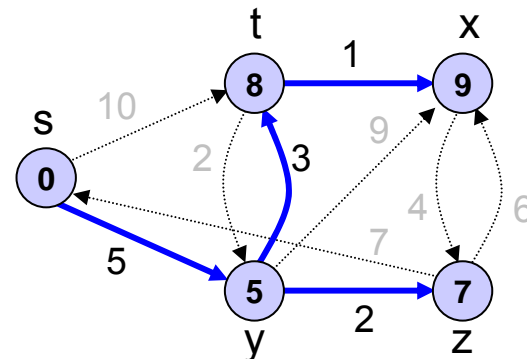
Verwendung einer Prioritätswarteschlange q



Anmerkung

- Dijkstras Algorithmus ermittelt zunächst nur die Distanz. Um tatsächlich die Kantenfolge zu ermitteln, die einen kürzesten Pfad von Knoten a zu Knoten b bildet, ist
 - analog zu BFS & DFS ein Verzeichnis v für die Vorgänger zu verwenden
 - der Vorgänger eines Knotens immer dann zu aktualisieren, wenn das Gewicht des Knotens (Verzeichnis) d verkleinert wird.
 - `If (d.value(v) + weight < d.value(z) // Weg über a`
`// kürzer`
`{ d.associate(z, d.value(v) + weight);`
`v.associate(z,v);}` `// Weg anpassen`
- Ein kürzester Pfad ist dann im durch v aufgespannten Baum durch Verfolgen der Kanten von b nach a zu ermitteln.

Der durch Dijkstra (implizit über das Verzeichnis v) erzeugte **aufspannende Baum** hat folgende Gestalt:



Aufwand hängt von der Implementierung der
Prioritätswarteschlange ab.

Variante 1: Feld der Länge $|V|$ für die Kanten $1 \dots |V|$

- Einfügen, Löschen: $O(1)$
- Suchen des Minimums: $O(V)$
- Komplexität: $O(V^2)$

Variante 2: Binärer Heap bei dünn besetztem Graphen

- Suchen des Minimums: $O(\log V)$
- Löschen: $O(\log V)$
- Komplexität: $O((E + V) \cdot \log V)$

...



Dijkstras Algorithmus funktioniert nicht bei negativen Kantengewichten.

- Grund (s.a. Korrektheitsbeweis): eine Kante mit negativem Gewicht kann die Distanz eines bereits bearbeiteten Knoten nachträglich verbessern.

Allerdings können negative Kantengewichte sinnvoll sein.

- Beispiel: Verbindungen werden mit negativen Kosten (also Gewinnen) versehen, um Anreize für Ihre Auswahl zu schaffen.



Wir betrachten Distanzgraph: $G = (V, E)$ mit $c: E \rightarrow \mathbb{R}$.

Aufgabe:

- Gegeben: Knoten s
- Gesucht: Kürzeste Pfade zu allen anderen Knoten

Bellman-Ford Algorithmus

- Ausgangspunkt: Startknoten s
- ausgehend von s werden alle Pfade der Länge $1, 2, 3, \dots, |V|-1$ betrachtet (der längste Pfad ohne Zyklus hat eine Länge von maximal $|V|-1$)
- bei der Betrachtung aller Pfade einer Länge werden pro möglichem Endstück eines Pfades eventuelle Verbesserungen der Weglänge zu einem Knoten zur Berechnung der Distanz herangezogen.



Auswahlschritt

- Wähle eine Kante $(v, v') \in E$ mit
 $v.\text{Entfernung} + c((v, v')) < v'.\text{Entfernung}$
 $v'.\text{Vorgänger} = v$
 $v'.\text{Entfernung} = v.\text{Entfernung} + c((v, v'))$

Zur Auswahl der Kante (v, v') eignet sich eine Breitensuche

- Lediglich die Randknoten sind von Interesse

Initialisierung

- $v.\text{Entfernung} = 0$ für $v=s$ und ∞ sonst

Invariante

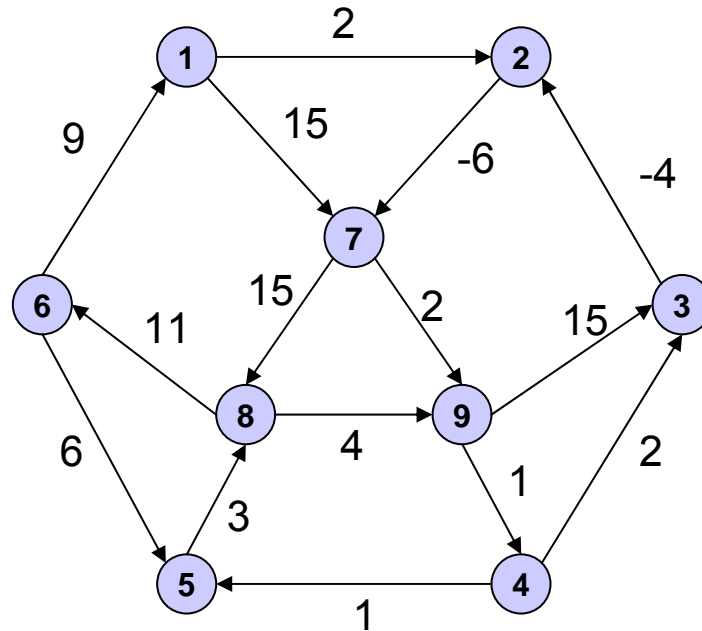
- $v.\text{Entfernung}$ hat einen endlichen Wert, dann gibt es einen Weg von s nach v mit Länge $v.\text{Entfernung}$

Achtung

- Verfahren terminiert nicht, falls es einen von s aus erreichbaren negativen Zyklus im Graphen gibt.



Beispielgraph

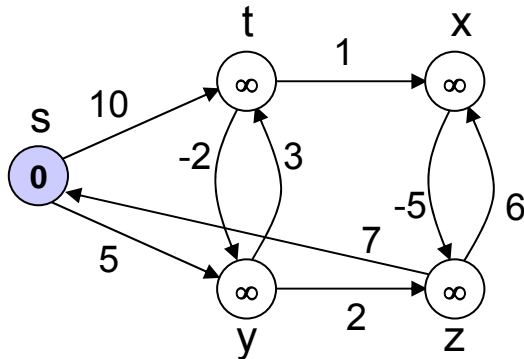


Ottman; Algorithmen und Datenstrukturen,
Spektrum Verlag, 1996

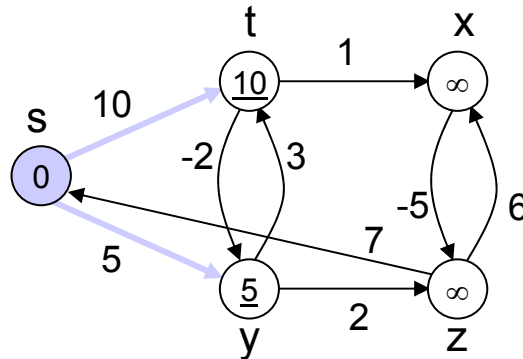
Beobachtung

- Es gibt keinen kürzesten Weg
- Negativer Zyklus (2, 7, 9, 4, 3, 2) mit der Länge -5
 - Es gibt immer einen kürzeren Weg mit einem Abstecher zum negativen Zyklus

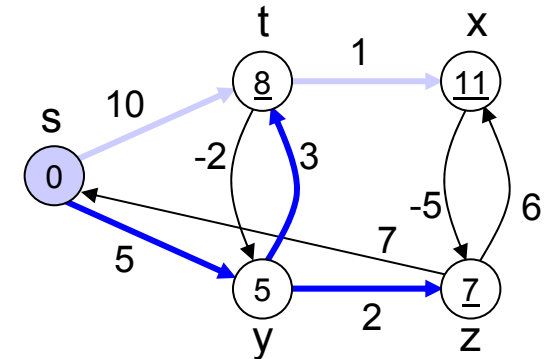




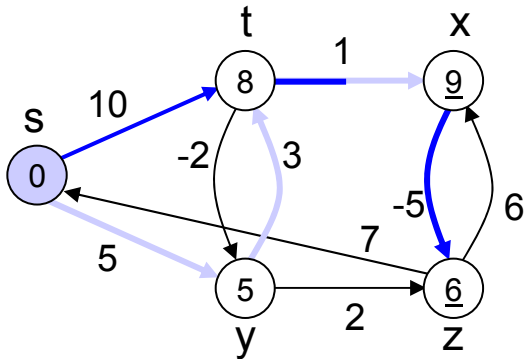
Graph nach Initialisierung
Startknoten: s



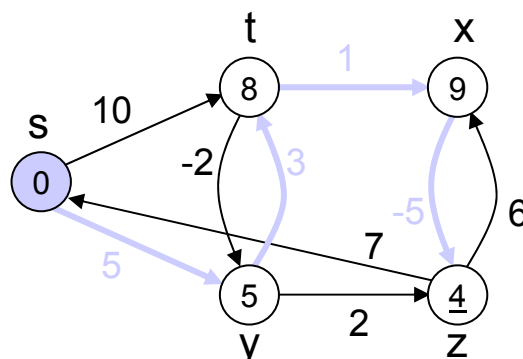
Iteration 1 (i=1)



Iteration 2 (i=2)



Iteration 3 (i=3)



Iteration 4 (i=4)



Saake; Algorithmen und Datenstrukturen,
dpunkt.verlag, 2004

*Beruht auf einer konzeptionell
quasiparallelen Bearbeitung aller
Kanten zu einem Zeitpunkt ohne
Berücksichtigung zwischen-
zeitlicher Änderungen



In jeder Phase wird jeder Knoten höchstens einmal betrachtet

- $O(|E|)$ für jede Phase

Es genügt Wege mit einer Länge von höchstens $|V|$ zu betrachten

- Nach $|V|$ Phasen kann abgebrochen werden

Aufwand: $O(|E| * |V|)$



Aufgabe

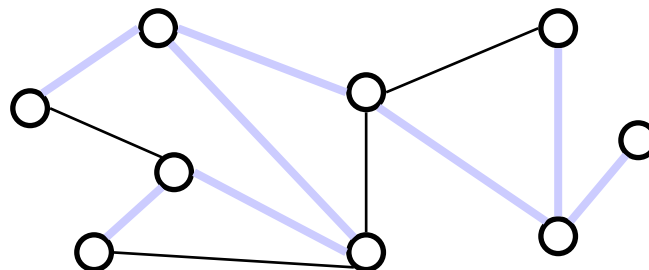
- Gegeben: Ein ungerichteter Graph mit Kantengewichten.
- Gesucht: Zusammenhängender Teilgraph der alle Knoten enthält, wobei die Summe der Kantengewichte minimal ist.

Eigenschaft: Der gesuchte Teilgraph ist azyklisch.

- Denn gäbe es einen Zyklus, dann könnte man zumindest eine Kante entfernen (also Kosten verringern) und trotzdem alles erreichen.
- Also ist der gesuchte Teilgraph ein Baum.

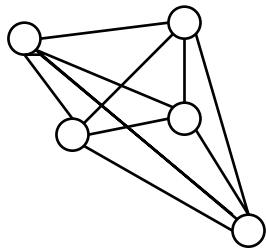
Spannbaum (spannender Baum):

- Teilgraph in Form eines Baums, der alle Knoten des Graphen enthält.



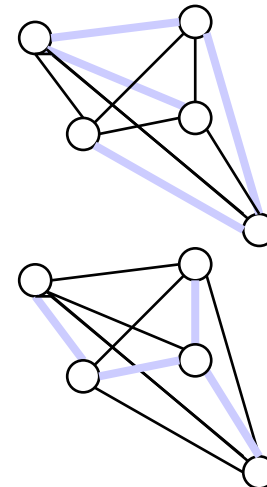
Beispiel (kennen wir aus Info I):

- Gegeben: Eine Landkarte als Graph G .
- Gesucht: Es soll ein Wasserleitungsnetz mit minimalen Kosten aufgebaut werden, das alle Ortschaften versorgt. Die Gesamtkosten des Netzes sind proportional zur Summe der Längen aller vorkommenden Leitungen.
- Ähnliche Probleme gibt es beim Entwurf von Rechnernetzen.



Was ist besser?

oder



Anderes Beispielproblem

- Gegeben ist eine Landkarte als Graph G .
- Die Post muss alle Ortschaften versorgen.
- Die Briefe werden hierarchisch zusammengefasst.
- In jedem Ort kommt ein großes Paket an und mehrere kleine werden in die Nachbarorte verschickt.
- Die Gesamtkosten sind proportional zur Summe der Längen aller vorkommenden Verbindungswege.
- Welche Transportwege (Teilgraph von G) sollten gewählt werden, um die Kosten zu minimieren?



Berechnung eines minimalen Spannbaums

- Erster Ansatz: Der Algorithmus für die Suche nach dem kürzesten Pfad lieferte für einen gegebenen Knoten einen Spannbaum.
- In jedem Schritt wurde eine Kante so hinzugefügt, dass der entstehende Pfad minimale Länge hatte.
- Aufwand: $O(|E| + |V| \cdot \log|V|)$.

Jetzt für jeden Knoten durchzuführen

$$O(|V| \cdot (|E| + |V| \cdot \log|V|)).$$

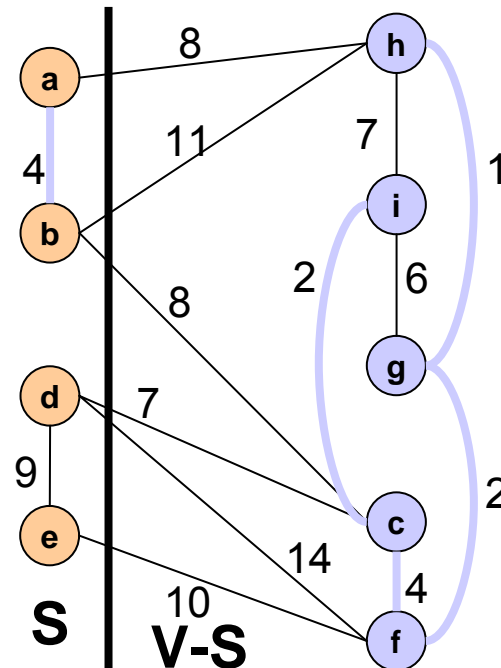
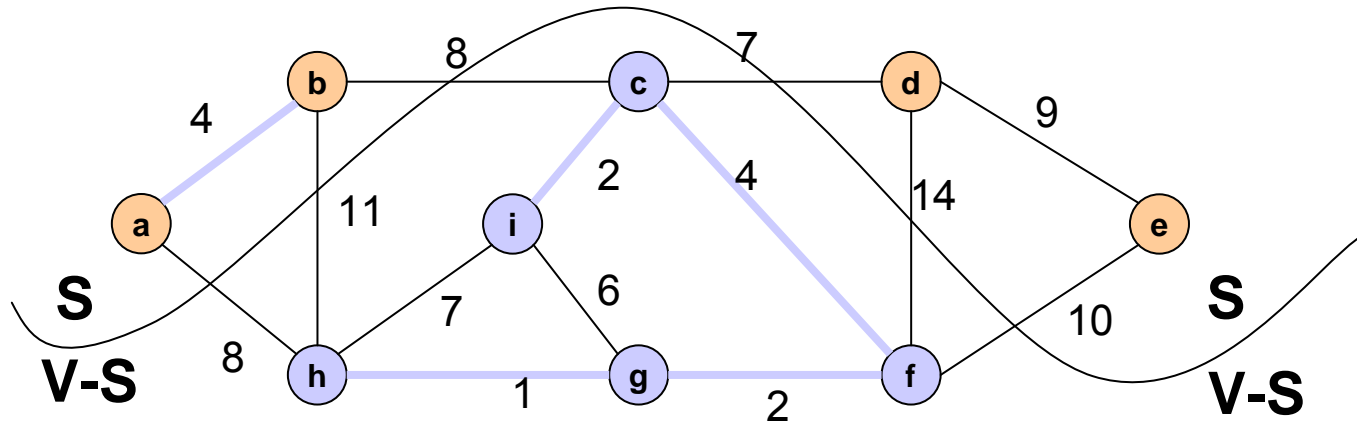
Herausforderung

- Auswahl der hinzuzufügenden Kante

Vorgehensweise

- Schnitt $(S, V-S)$ eines unidirektionalen Graphen $g = (V, E)$
- Kante, die den Schnitt kreuzt: ein Endpunkt in S der andere in $V-S$
- Kreuzende Kante mit minimalem Gewicht gesucht





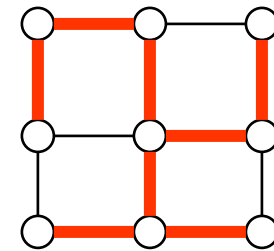
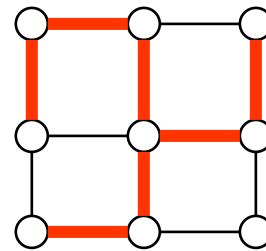
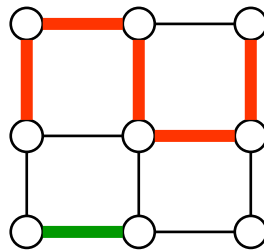
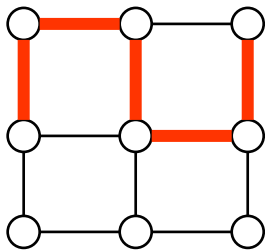
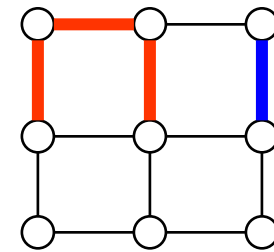
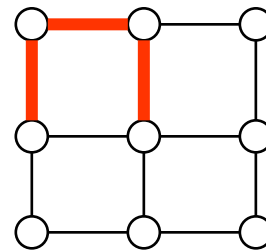
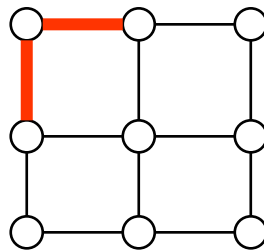
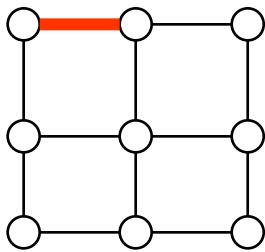
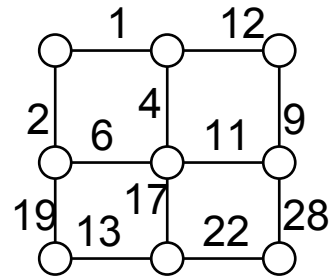
Cormen; Introduction to Algorithms,
MIT Press, 2001



Aus Informatik 1 bekannt

- Beginne mit sortierter Kantenliste in aufsteigender Reihenfolge
- Kante gehört zur Lösung, wenn sie
 - einen vorhandenen Baum um einen noch nicht betrachteten Knoten erweitert
 - zwei noch nicht betrachtete Knoten verbindet (zu einem neuen Baum)
 - zwei verschiedene Bäume verbindet
- Jetzt kann man die Kanten nacheinander betrachten und sofort entscheiden, ob die Kante zur Lösung gehört oder nicht.
- Es ist garantiert, dass kein Zyklus entsteht.





Induktion

- Zu jedem Zeitpunkt haben wir einen Wald minimal spannender Bäume.

Induktionsanfang

- Jeder Knoten ist für sich ohne Kanten ein minimal spannender Baum.

Induktionsschritt

- Vereinige durch Hinzufügen zwei Bäume zu einem, so dass dieser seine Knoten auch minimal aufspannt.

der Graph $G = (V, E)$ habe n Knoten
sortiere E nach aufsteigender Länge
Erzeuge zu jedem Knoten seinen minimalen Spannbaum (Induktionsanfang)
für i von 1 bis $n-1$ (Induktionsschritt)
 füge die kürzeste noch nicht hinzugefügte Kante zum
 Wald hinzu, sofern dies keinen Zyklus entstehen lässt

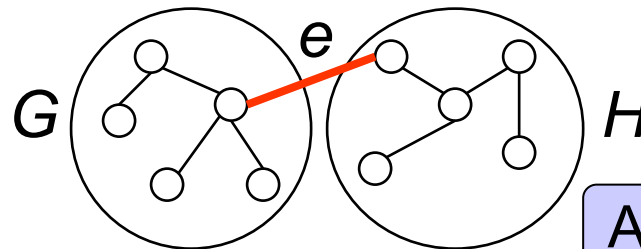


Induktionsanfang

- trivial

Induktionsschritt

- Nach Induktion gilt, dass der Teilgraph G ein minimaler Spannbaum ist und ebenso Teilgraph H .
- Nach Verfahren ist e die kürzeste Kante, die G und H verbindet.
- $G \cup H \cup e$ ist auch minimal, weil $G \cup H \cup f$ nicht besser sein kann, und eine „Umordnung“ G oder H „länger“ macht.
- Würde man G und H mit mehr als einer Kante verbinden, würde ein Zyklus entstehen, außer man entfernt dafür andere (kürzere) Kanten \Rightarrow Summe wieder länger.



Aufwand: $O(|E| * \log |V|)$.

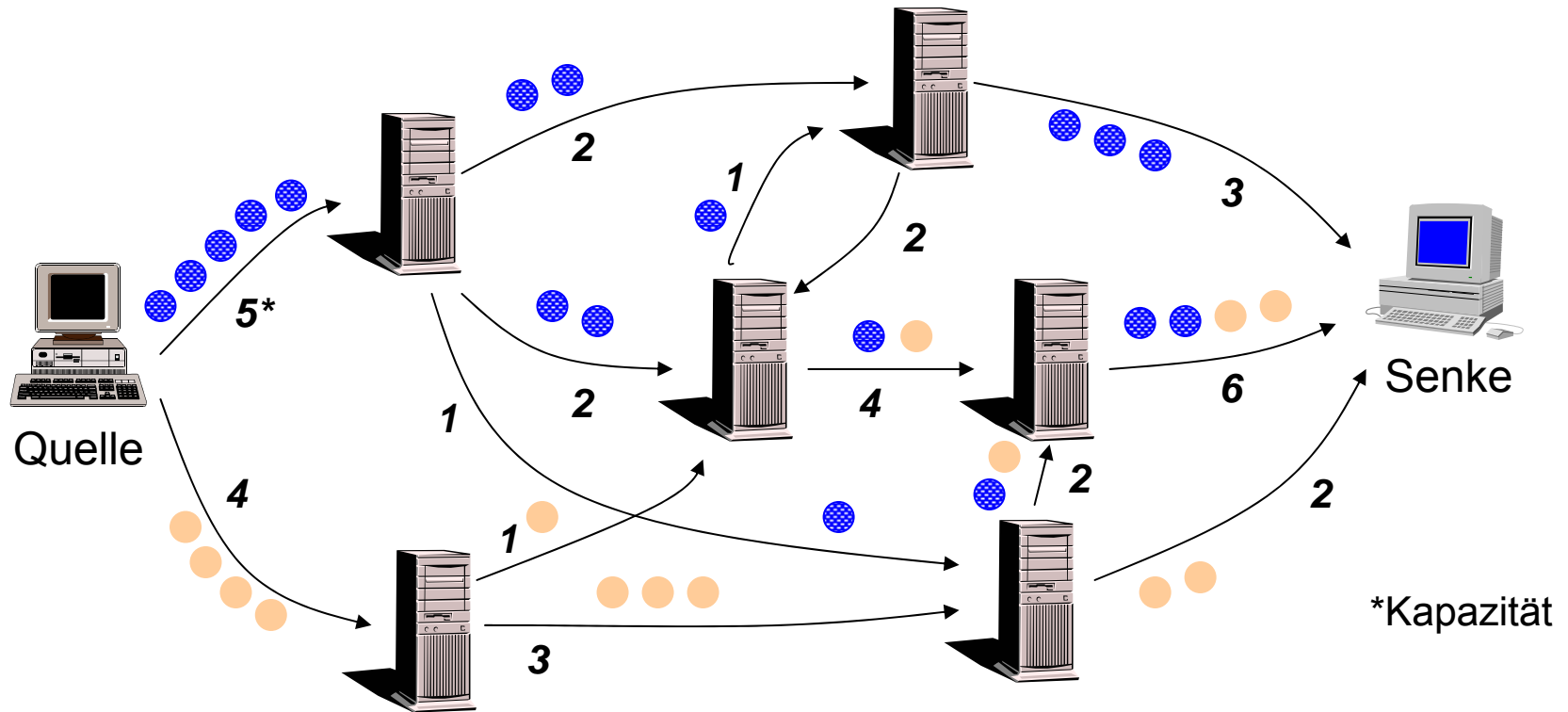
Motivation: Logistische Probleme

Beispiel: Paketvermittlung im Computernetzwerk

- Menge von Computern (=Knoten eines Graphen), einer bildet die Quelle einer zu übermittelnden Datenmenge, einer die Senke
- Verschiedene Computer sind über Einzelverbindungen (=Kanten) verknüpft und können Daten über diese Verbindung übertragen
 - Daten werden als Pakete fester Länge übertragen
 - Jede Verbindung hat eine Kapazität (Maximale Menge pro Zeiteinheit zu übertragender Datenpakete)
- Die Vernetzung ist derart, dass verschiedene Pfade bestehen, über die Daten von der Quelle zur Senke transportiert werden können.
- Frage: wie viele Datenpakete können pro Zeiteinheit maximal von der Quelle bis zur Senke übertragen werden



Paketvermittlung im Computernetzwerk



Frage: wie viele Datenpakete können pro Zeiteinheit maximal von der *Quelle* bis zur *Senke* übertragen werden

Antwort: 9

Ausgangspunkt

- Gerichteter Graph $G=(V,E)$, Quelle $q \in V$ und Senke $s \in V$
- Kapazitätsfunktion $c: E \rightarrow \mathbb{R}^+$ gibt jeder Kante $e \in E$ eine Kapazität
- Flussfunktionen $f: E \rightarrow \mathbb{R}$ gibt zu jeder Kante $e \in E$ den aktuellen (Durch-) Fluss an (negative Flüsse sind zulässig)
- Der Wert eines Flusses für eine Quelle q ist bestimmt durch den dort herausfließenden aktuellen Fluss:

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

Nebenbedingungen (definieren korrekten Fluss)

- Kapazitätsbedingung: $\forall e \in E: |f(e)| \leq c(e)$
- Flusssymmetrie: $\forall u, v \in V: f(u, v) = -f(v, u)$
- Flusserhaltung: $\forall u \in V \setminus \{q, s\}: \sum_{v \in V} f(u, v) = 0$

Gesucht: Maximaler Fluss von Quelle q zur Senke s =

$$\max \{ flow(G, f, q) \mid f \text{ ist korrekter Fluss in } G \text{ bzgl. } q \text{ und } s \}$$



Ziel

- Bestimmt für einen mit Kapazitäten versehenen Graphen den maximalen Durchfluss von einer Quelle zu einem Ziel.

Vorgehensweise

- Kombination aus Greedy u. Zufallsauswahl.

Grundlegender Ablauf

- Beginne mit Fluss $f := 0$
- Suche einen (zyklenfreien) Pfad P von q nach s , der noch eine Restkapazität k (=nutzbarer Fluss) größer als 0 aufweist
 - Falls ein Pfad gefunden wird: Nutze die Restkapazität vollständig aus (modifiziere f entsprechend) und wiederhole diesen Schritt
 - Falls kein Pfad gefunden wird: fertig



Lester Randolph Ford (Junior!)

- * 23. September 1927
- 1956 zusammen mit Fulkerson Maxflow-Mincut Theorem
- Verantwortlich für viele fundamentelle Untersuchungen zu Flüssen in Netzen

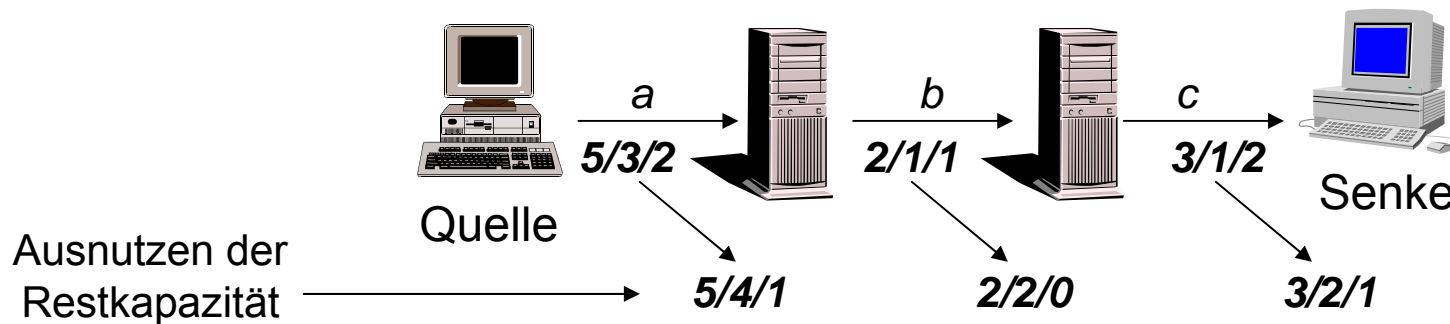
Delbert Ray Fulkerson

- * 14. August 1924
- + 10. Januar 1976
- Nach ihm ist der „Fulkerson-Preis“ für aussergewöhnlich gute Arbeiten im Bereich der diskreten Mathematik benannt worden.

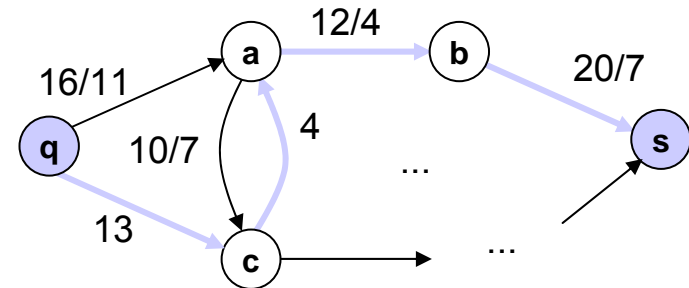
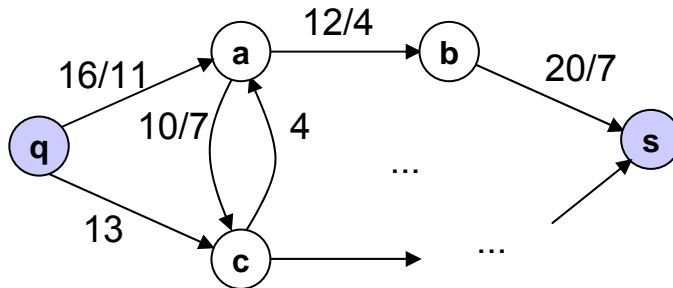


Ausschnitt

- folgender Pfad von der Quelle bis zur Senke sei gefunden (Notation: $c/f/c-|f|$). Die Restkapazität beträgt 1 (wegen Kante b) und kann genutzt werden



Beispiel: Folgender Graph befinde sich in der angegebenen Flusssituation (Notation: c oder c/f)



- Wird der rechts angegebene Pfad ausgewählt, so kann über diesen Pfad ein Fluss im Umfang von 8 Einheiten erfolgen.

Warum?

- Der über Kante (a,c) laufende Fluss von 7 Einheiten kann alternativ über den gefundenen Pfad von Knoten a ausgehend mit "abgeleitet" werden. Danach ist die Kapazität der Kante (a,c) entsprechend zu korrigieren.
- D.h. der über Kante (a,c) laufende Fluss im Umfang von 7 Einheiten kann in die Kapazität der Kante (c,a) mit eingerechnet werden.



"Ausgleichsflüsse" betreffen nicht nur bereits vorhandene Zyklen $(u,v), (v,u) \in E$.

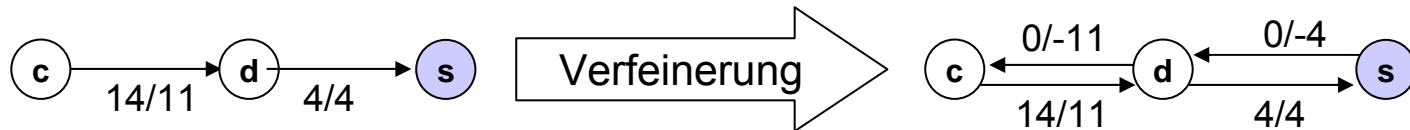
- \Rightarrow für jede Kante (u,v) in G mit $(v,u) \notin E$ wird eine Kante (v,u) eingefügt und mit Kapazität 0 versehen.

Weiterhin wird bei Erhöhung des Flusses einer Kante (u,v) gleichzeitig der Fluss (v,u) reduziert ($>$ Flusssymmetrie). D.h.

- nach $f((u,v)) := f((u,v)) + k$ für $1 \leq k$ folgt
- $f((v,u)) := -f((u,v))$ // also um k verkleinert

Beispiel (Ausschnitt)

- Notation c/f



Definition Restkapazität

- In einem gerichteter Graph $G=(V,E)$ mit Kapazitätsfunktion $c: E \rightarrow \mathbb{R}^+$ und Flussfunktionen $f: E \rightarrow \mathbb{R}$ sei die Restkapazität $c_f: E \rightarrow \mathbb{R}$ definiert durch $c_f((u,v)) = c((u,v)) - f((u,v))$

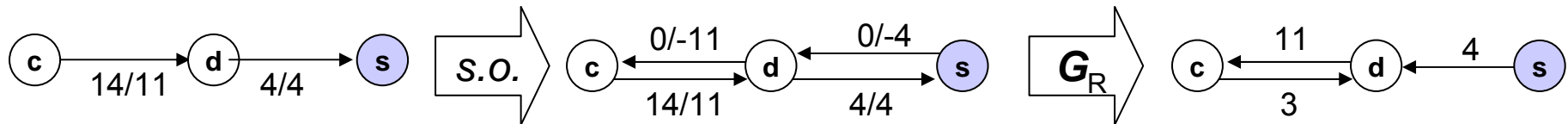
Definition Restkapazitätsgraph

- Zu einem gerichteter Graph $G=(V,E)$ mit Kapazitätsfunktion $c: E \rightarrow \mathbb{R}^+$ und Flussfunktionen $f: E \rightarrow \mathbb{R}$ sei der Restkapazitätsgraph $G_f=(V,E_f)$ definiert durch

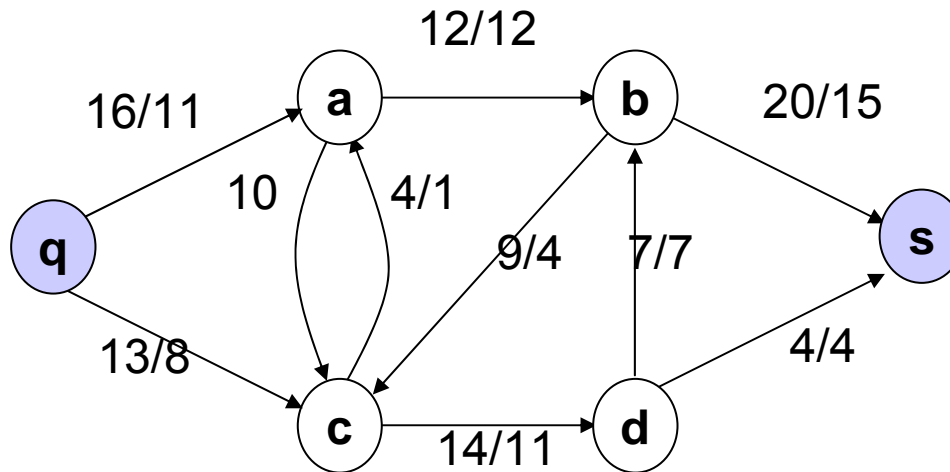
$$E_f = \{(u,v) \in V \times V \mid c_f((u,v)) > 0\}$$

Beispiel (Ausschnitt)

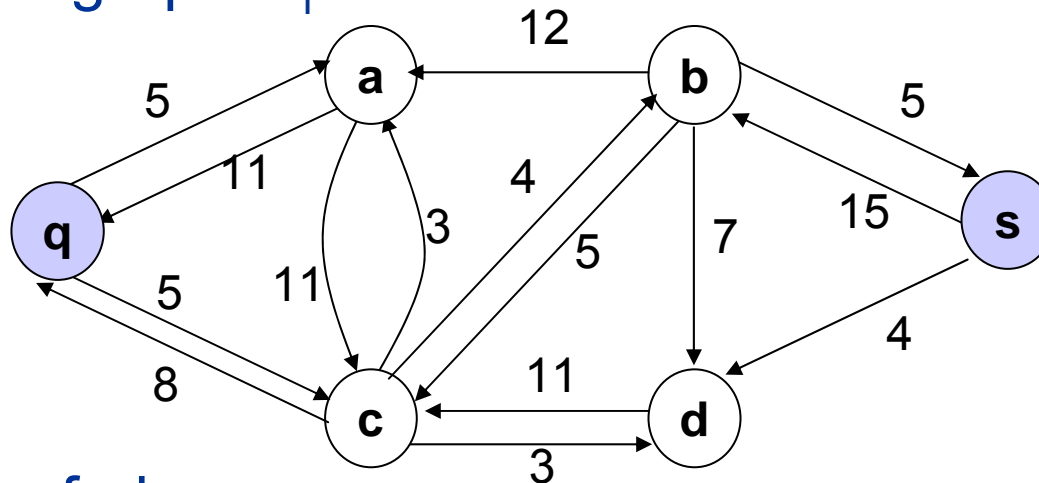
- Notation c/f oder c_f



Flussgraph G



Restkapazitätsgraph G_f



Erweiterungspfad p

■ $p = [q, c, b, s]$ mit $c_f(p) = c(c, b) = 4$



G_f ist Flussgraph mit den durch c_f gegebenen Kapazitäten

- Sei f ein Fluss in G und f' ein Fluss in G_f . Summe $f + f'$ ist Fluss in G mit $|f + f'| = |f| + |f'|$
- Es gilt: $(f_1 + f_2)(u, v) = f_1(u, v) + f_2(u, v)$

Beweis

- Flusssymmetrie
 - $(f + f')(u, v) = f(u, v) + f'(u, v) = -f(v, u) - f'(v, u) = -(f(v, u) + f'(v, u)) = -(f + f')(v, u)$
- Kapazitätsbedingung
 - Es gilt: $f'(u, v) \leq c_f(u, v)$
 - $(f + f')(u, v) = f(u, v) + f'(u, v) \leq f(u, v) + (c(u, v) - f(u, v)) = c(u, v)$
- Flusserhaltung ($u \in V - \{q, s\}$)
 - $$\sum_{v \in V} (f + f')(u, v) = \sum_{v \in V} (f(u, v) + f'(u, v)) = \sum_{v \in V} f(u, v) + \sum_{v \in V} f'(u, v) = 0$$
- Damit ergibt sich
 - $$|f + f'| = \sum_{v \in V} (f + f')(q, s) = \sum_{v \in V} (f(q, s) + f'(q, s)) = \sum_{v \in V} f(q, s) + \sum_{v \in V} f'(q, s) = |f| + |f'|$$



Erweiterungspfad p

- Zyklenfreier Pfad von q nach s in G_f
- Restkapazität von p
 - $c_f(p) = \min \{c_f(u,v): (u,v) \text{ ist in } p\}$

Satz

- Sei $G = (V,E)$ ein Flussgraph, f ein Fluss in G und p ein Erweiterungspfad in G_f . Die Funktion $f_p: V \times V \rightarrow \mathbb{R}$ ist wie folgt definiert:

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

- Dann ist f_p Fluss in G_f mit dem Wert $|f_p| = c_f(p) > 0$

Weiteres

- Funktion $f': V \times V \rightarrow \mathbb{R}$, $f' = f + f_p$
- $|f'| = |f| + |f_p| > |f|$



nach Verfeinerung; insb. Erweiterung von E

- Beginne mit $f((u,v)) := 0 =: f((v,u)) \quad \forall (u,v) \in E$
- Suche einen (zyklenfreien) Pfad $P=[p_0, p_1, \dots, p_m]$ im Restkapazitätsgraphen G_R
 - von $q=p_0$ nach $s=p_m$,
 - der noch eine Restkapazität größer als 0 aufweist, d.h. für den nutzbaren Fluss k dieses Pfades gilt
$$k = \min\{ c_f((p_n, p_{n+1})) \mid 0 \leq n < m \} > 0$$
- Pfad gefunden?
 - Falls ja: Nutze die Restkapazität vollständig aus (modifiziere f entsprechend) und wiederhole den Schritt; d.h.
 - $f((p_n, p_{n+1})) := f((p_n, p_{n+1})) + k$ für $0 \leq n < m$ und
 - $f((p_{n+1}, p_n)) := -f((p_n, p_{n+1}))$
 - Wiederhole vorhergehenden Schritt
 - Falls nein: fertig



Problem

- Ist Fluss maximal, wenn kein Erweiterungspfad gefunden wird?

Idee

- Betrachte Kapazitäten an Schnitten des Flussgraphen G

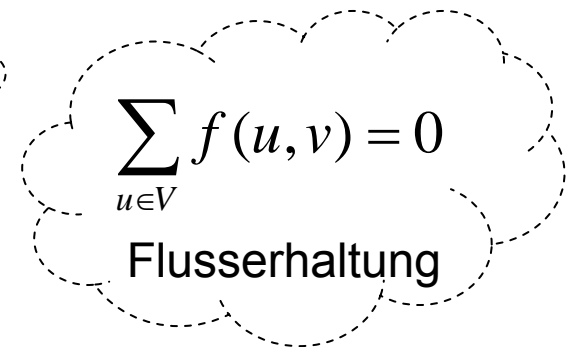
Schnitt

- $\text{cut}(S, T)$ von Flussgraphen $G = (V, E)$ ist Partitionierung in S und $T = V - S$ mit $s \in S$ und $t \in T$
- Netto Fluss

- Positiver Fluss aus einem Knoten minus positiver Fluss in einen Knoten

$$\sum_{\substack{v \in V \\ f(u, v) > 0}} f(u, v) - \sum_{\substack{r \in V \\ f(r, u) > 0}} f(r, u)$$

- Netto Fluss über $\text{cut}(S, T) = f(S, T)$
 - Kapazität von $\text{cut}(S, T) = c(S, T)$



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

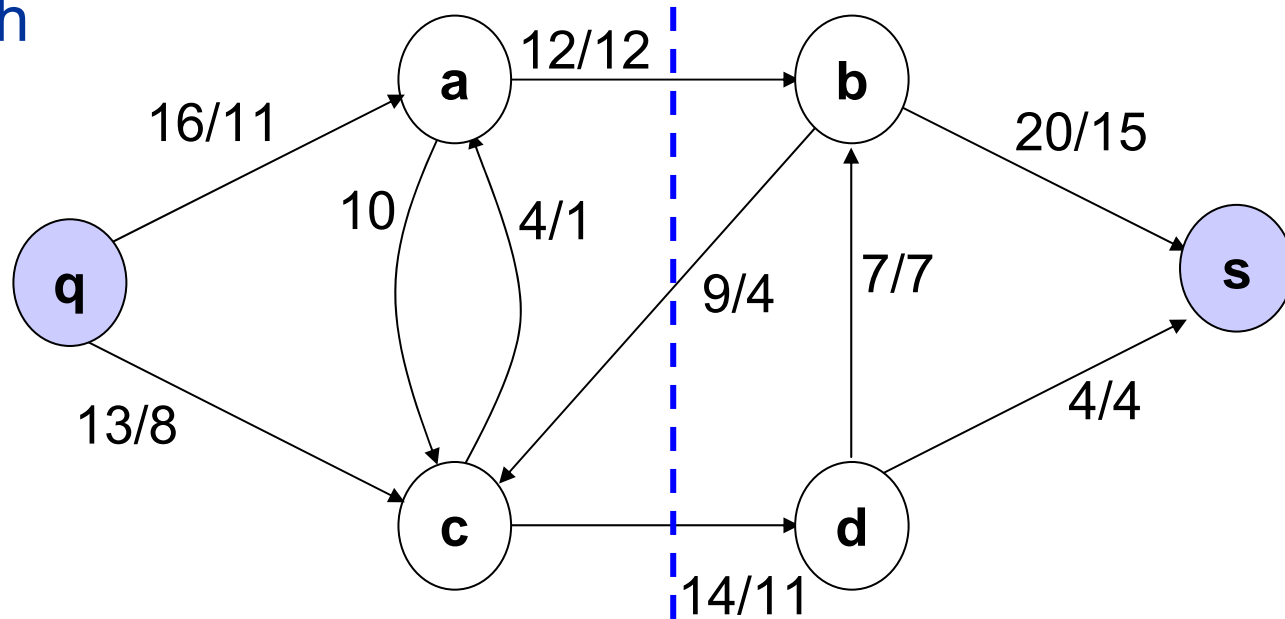
Flusserhaltung

Minimaler Schnitt

- Kapazität ist minimal über allen Schnitten des Flussgraphen



Flussgraph



Schnitt

■ $\text{cut}(\{q, a, c\}, \{b, d, s\})$

■ Netto Fluss

■ $f(a, b) + f(c, d) + f(c, b) =$

→ Netto Fluss ist an allen Schnitten identisch.

■ Kapazität

■ $c(a, b) + c(c, d) =$



Aussage

- Kann kein Erweiterungspfad im Restkapazitätsgraph gefunden werden, dann ist der Fluss maximal.

Theorem

- f ist Fluss im Flussgraphen $G=(V,E)$ mit Quelle q und Senke s . Die folgenden Bedingungen sind äquivalent
 - (1) f ist maximaler Fluss in G
 - (2) Der Restkapazitätsgraph G_f enthält keine Erweiterungspfade
 - (3) $|f| = c(S,T)$ für einen Schnitt von G

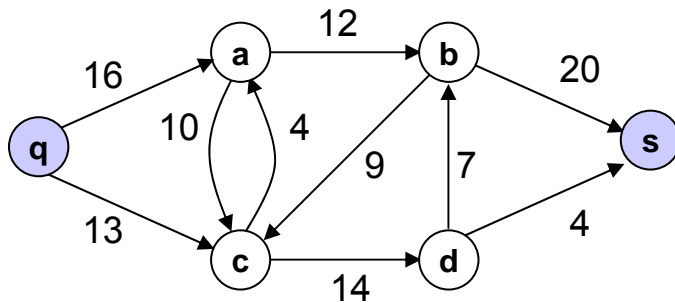
Beweis

- (1) \rightarrow (2)
 - f sei maximaler Fluss in G . G_f habe Erweiterungspfad p . Dann ist Fluss $f+f_p$ Fluss in G mit einem Flusswert größer $|f| \rightarrow$ Widerspruch.
- (2) \rightarrow (3)
 - G_f habe keinen Erweiterungspfad. Definiere $S = \{v \in V: \text{es existiert ein Pfad von } q \text{ nach } s \text{ in } G_f\}$ und $T=V-S$. Für alle $u \in S, v \in T$ gilt $f(u,v) = c(u,v)$. Deshalb $|f|=f(S,T)=c(S,T)$
- (3) \rightarrow (1)
 - Es gilt $|f| \leq c(S,T)$ für alle Schnitte. Also impliziert $|f|=c(S,T)$ dass f ein maximaler Fluss ist.

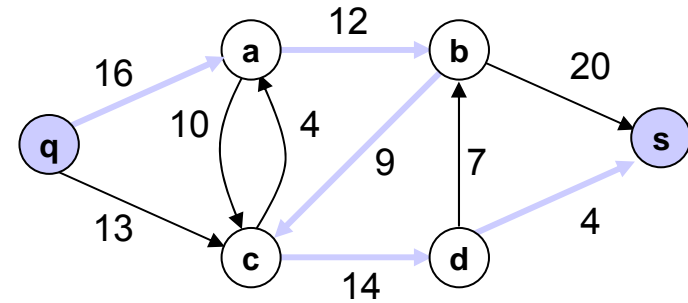




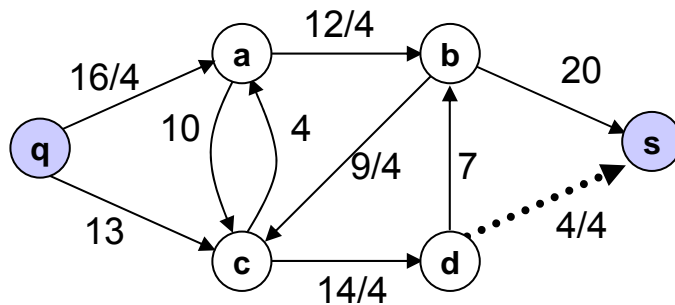
Cormen; Introduction to Algorithms,
The MIT-Press, 2001



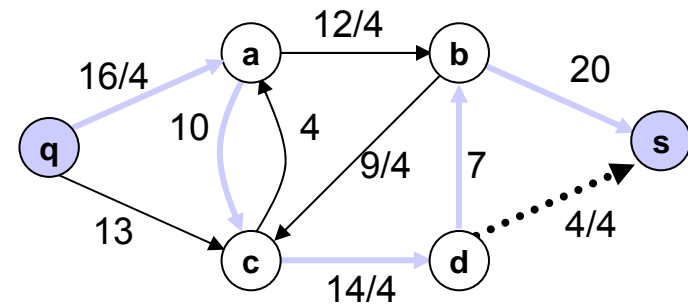
Graph nach Initialisierung



Iteration 1: Bestimmung des ersten Pfades
Restkapazität des Pfades: 4

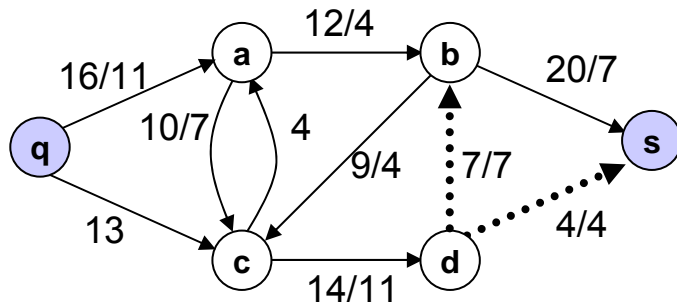


Graph nach Iteration 1
Kanten ohne Restkapazität gepunktet

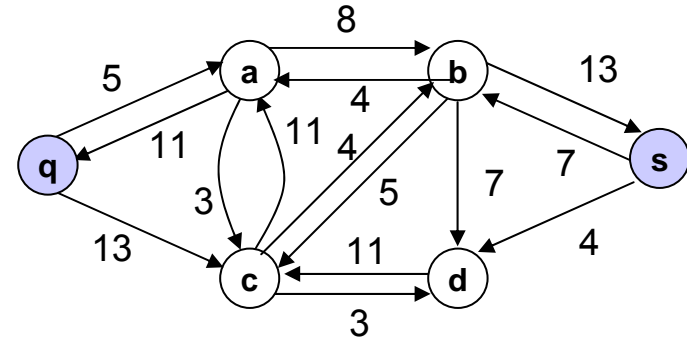


Iteration 2: Bestimmung des zweiten Pfades
Restkapazität des Pfades: 7

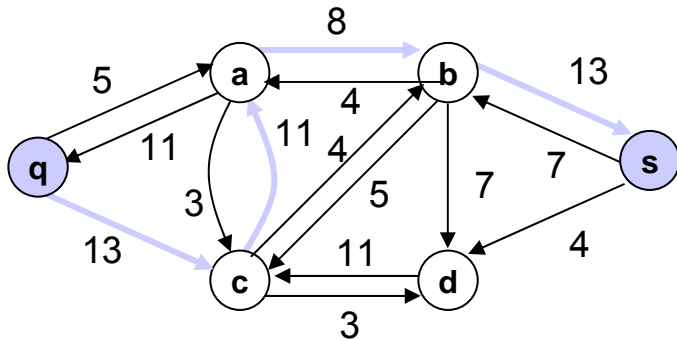




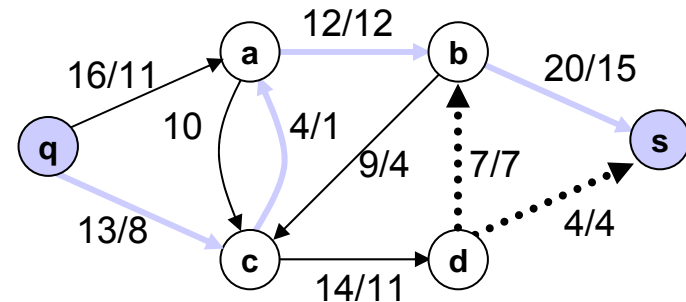
Graph nach Iteration 2



Im Folgenden: "Ausgleichsflüsse"
Daher Darstellung Restkapazitätsgraph

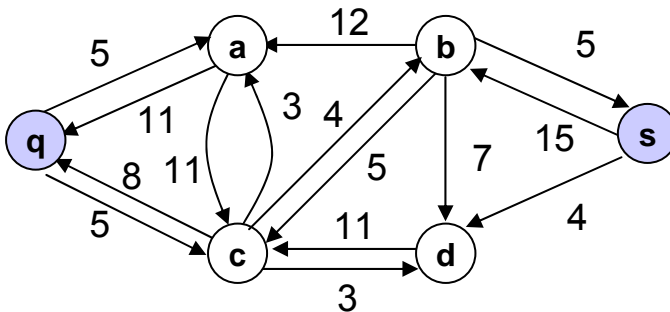


Iteration 3: Bestimmung des dritten Pfades
Restkapazität des Pfades: 8

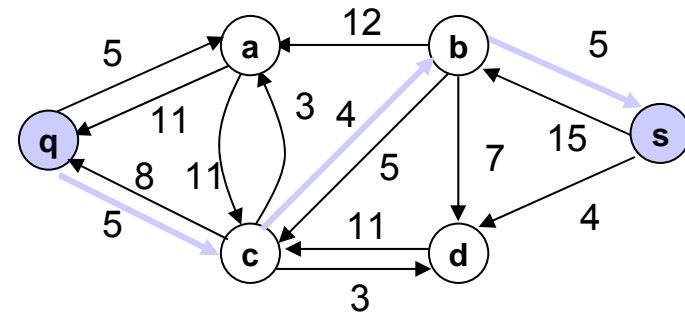


Graph nach Iteration 3

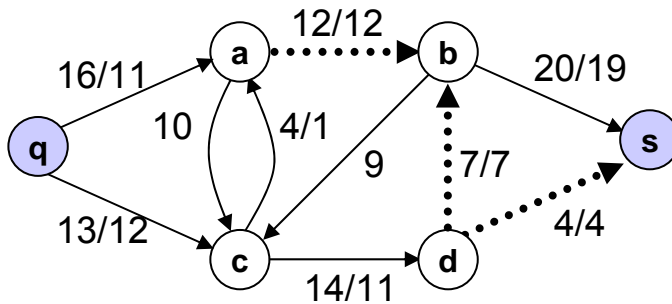




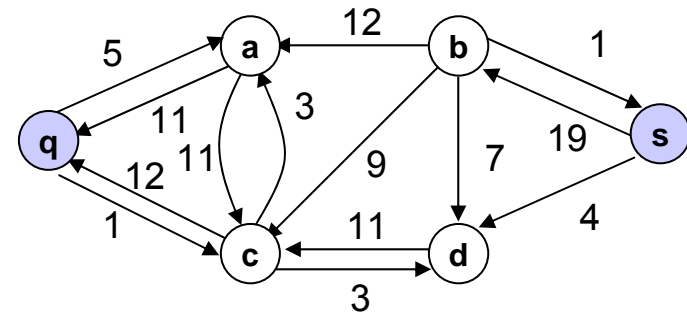
Darstellung Restkapazitätsgraph



Iteration 4: Bestimmung des dritten Pfades
Restkapazität des Pfades: 4



Graph nach Iteration 4
(Fertig > Kein Pfad von q nach s zu finden)



Darstellung Restkapazitätsgraph
(Fertig > Kein Pfad von q nach s zu finden)



Beobachtung

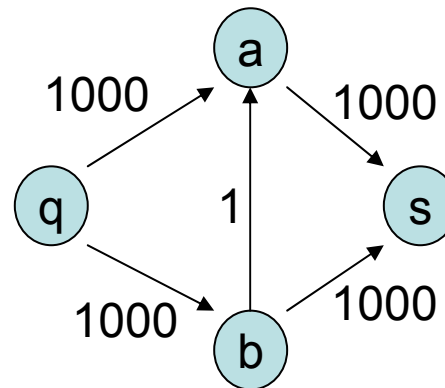
- Aufwand abhängig von Bestimmung des Erweiterungspfads.
- Anzahl der Iterationen hängt von der Auswahl des Pfads in jeder Iteration ab.



Aufwand

- $O(E |f^*|)$
 - f^* entspricht dem maximalen Fluss
- Beobachtung
 - Wert des Flusses erhöht sich in jedem Durchlauf mindestens um Eins.
- Aufwand zum Finden eines Pfads
 - $O(E)$ mit Tiefensuche oder Breitensuche

Beispiel für hohen Aufwand



- Ausgewählter Pfad: $p=[q, b, a, s]$ mit $c_f(p)=1$
 - Nächster Pfad: $p_2=[q, a, b, s] \dots$
- 2000 Durchläufe



Bemerkungen

- Die Suche nach einem Pfad mit ausreichender Restkapazität kann auf Grundlage der Tiefensuche erfolgen.
 - Der Graph kann vor jeder Pfadsuche derart modifiziert werden ($G \rightarrow G_f$), dass voll ausgenutzte Kanten (mit Restkapazität ≤ 0) entfernt werden.
 - Damit reduziert sich das Finden eines Pfads mit Restkapazität > 0 auf das Finden eines Pfads von q nach s in G_f .

Edmonds-Karps Algorithmus

- Breitensuche
- Erweiterungspfad ist kürzester Pfad.
- Aufwand: $O(VE^2)$

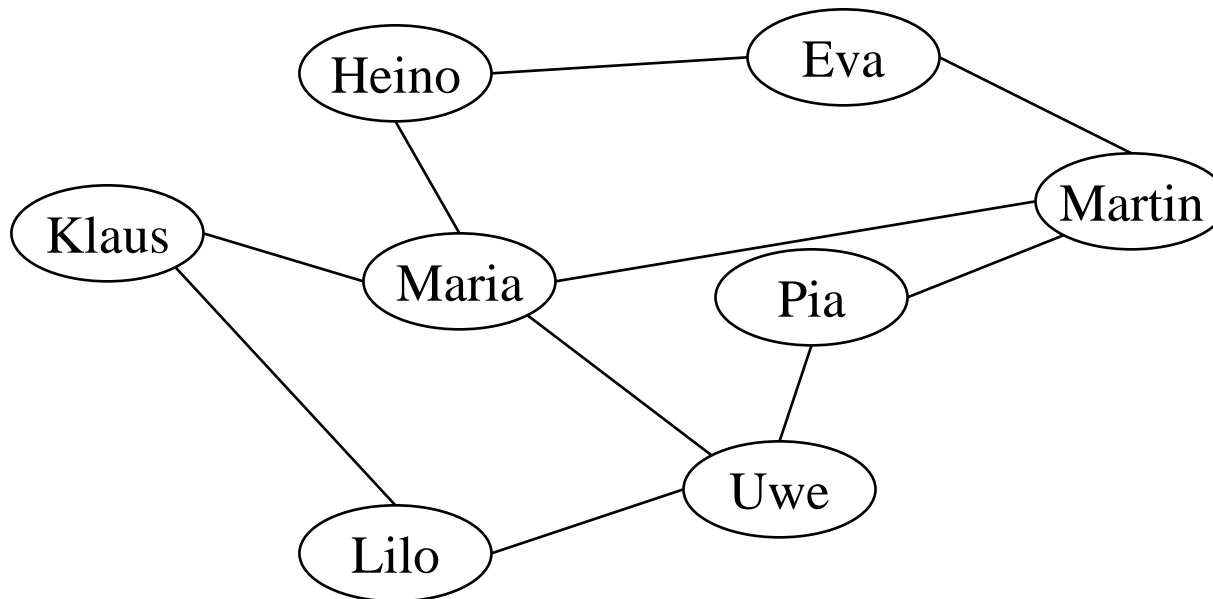


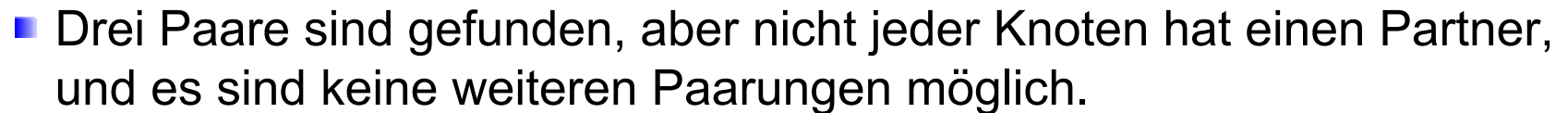
Ausgangspunkt

- Zuordnungsprobleme (verschiedene Dinge einander zuordnen)
 - Männer / Frauen im Tanzkurs,
 - Arbeiten / Arbeitskräfte,
 - Koffer / Schließfächer ...

Ein motivierendes Beispiel

- Gegeben: Wir befinden uns im Tanzkurs. Jeder Teilnehmer (Knoten) weiß, mit wem er gerne tanzt (Kante).
- Aufgabe: Bestimme mögliche Paarungen.





- Es ist ja noch ein Herr und eine Dame übrig geblieben!

Ungerichteter Graph $G = (V, E)$

- Matching M ist Teilmenge der Kanten von G so dass folgendes gilt:
 - Keine zwei Kanten in M haben gleiche Endknoten
- Größe der Zuordnung
 - $|M|$, Anzahl der Kanten in M

Unabhängige Kanten

- Zwei Kanten (u, v) und (x, y) heißen **unabhängig**, wenn u, v, x, y vier verschiedene Knoten sind.
 - Kanten haben also keine gleichen Endknoten

Zuordnung

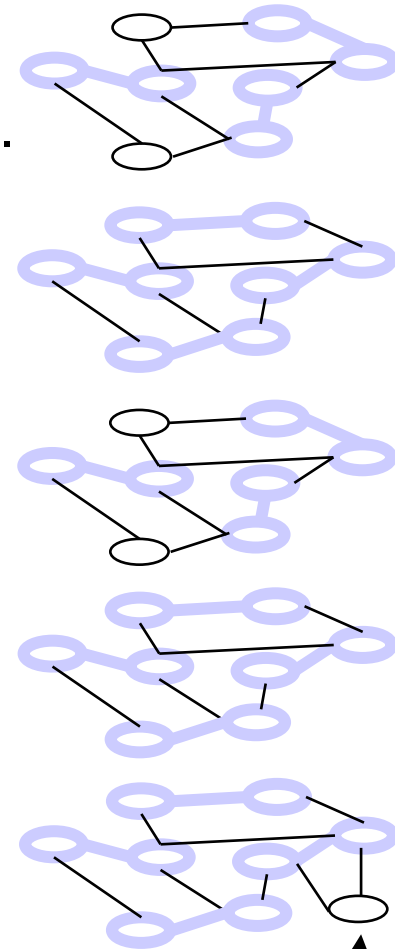
- Eine Kantenmenge M heißt unabhängig, wenn alle ihre Elemente paarweise unabhängig sind. Solche Kantenmengen heißen auch Zuordnung (**Matching**).

Benachbarte Kanten

- Zwei Kanten (u, v) und (x, y) . Wenn $u=x$ oder $u=y$ oder $v=x$ oder $v=y$, dann heißen die Kanten **benachbart** (oder verbunden oder adjazent).

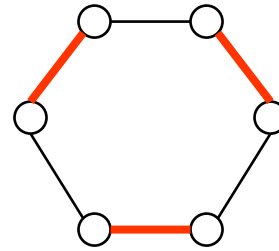
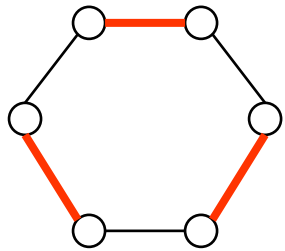


- Ein Knoten heißt **frei** bzgl. eines Matchings, wenn er keine Kante des Matchings hat, sonst heißt er **gematcht**.
- Ein Matching M heißt **perfekt**, wenn es alle Knoten des Graphen überdeckt.
- Ein Matching M heißt **maximal** (nicht erweiterbar) wenn es um keine Kante erweitert werden kann.
- Ein Matching M heißt **Maximum** wenn es kein Matching mit mehr Kanten gibt, d.h. $|M|$ ist maximale Größe.
- Ein Matching M bei dem nur ein Knoten frei bleibt, heißt **fast perfekt**.

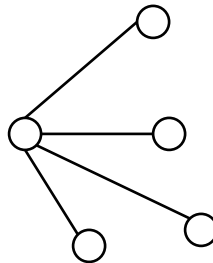


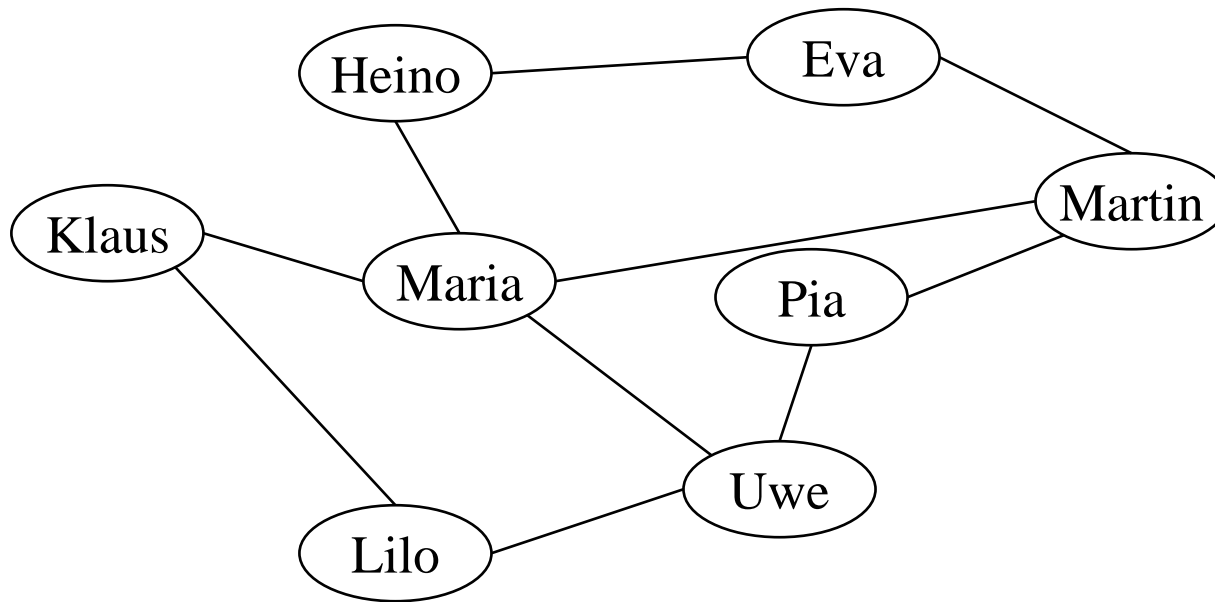
ist nicht perfekt

Ein „gerader Kreis“ hat genau 2 perfekte Matchings.

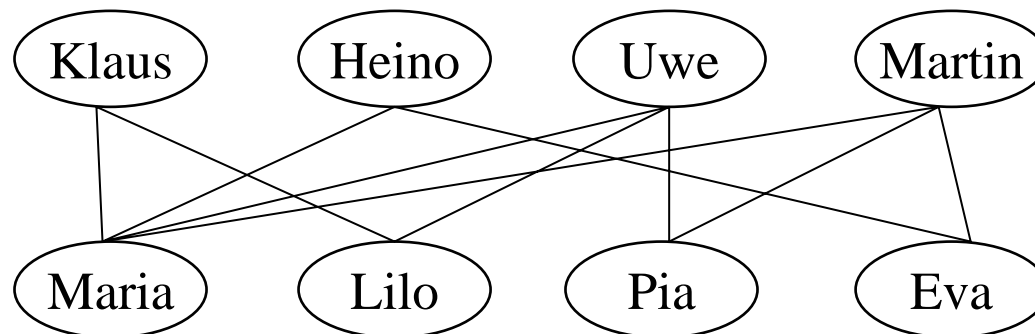


Nicht jeder Graph hat ein (fast) perfektes Matching.





→ Knoten in zwei Gruppen aufteilbar; Kanten jeweils zwischen diesen Gruppen



Bipartiter Graph

- Ungerichteter Graph $G=(X \cup Y, E)$ mit $X \cap Y = \emptyset$ und nur Kanten $(x_i, y_j) \in E$ mit $x_i \in X, y_j \in Y$ oder umgekehrt.

Graph und Bipartiter Graph

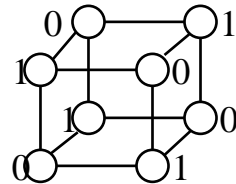
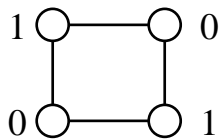
- Zusammenhängende Graphen mit bestimmten Eigenschaften lassen sich in bipartite Graphen überführen.
- Diese Eigenschaften sind durch den Satz von König gegeben:
Graph bipartit \Leftrightarrow keine ungeraden Zyklen

Vorgehen

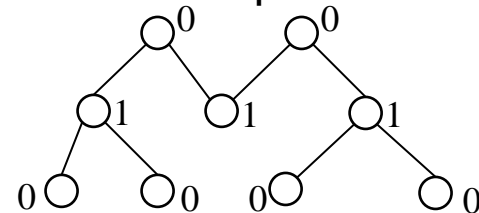
Markiere v mit 0
solange es eine Kante (x, y) gibt mit x markiert und y nicht
markiere y mit $\neg \text{Markierung}(x)$

Beispiele

- Alle Hyperwürfel sind bipartit.

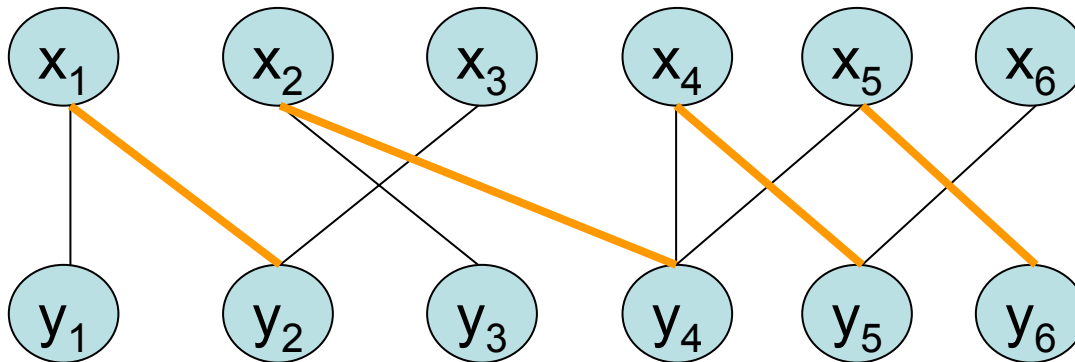


- Alle Bäume sind bipartit.



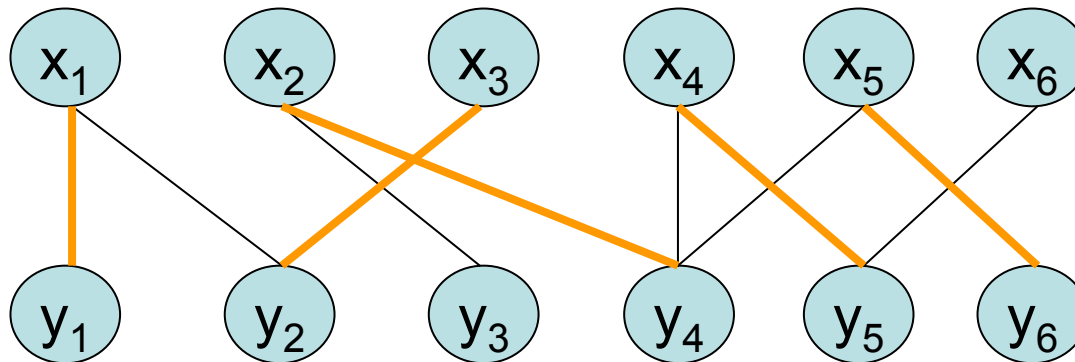
Bipartiter Graph

- $G = (X \cup Y, E)$ mit $X = \{x_1, x_2, \dots, x_6\}$ und $Y = \{y_1, y_2, \dots, y_6\}$



→ Zuordnung nicht maximal

- Mehr Kanten beispielsweise wenn (x_1, y_1) und (x_3, y_2) verwendet statt (x_1, y_2)

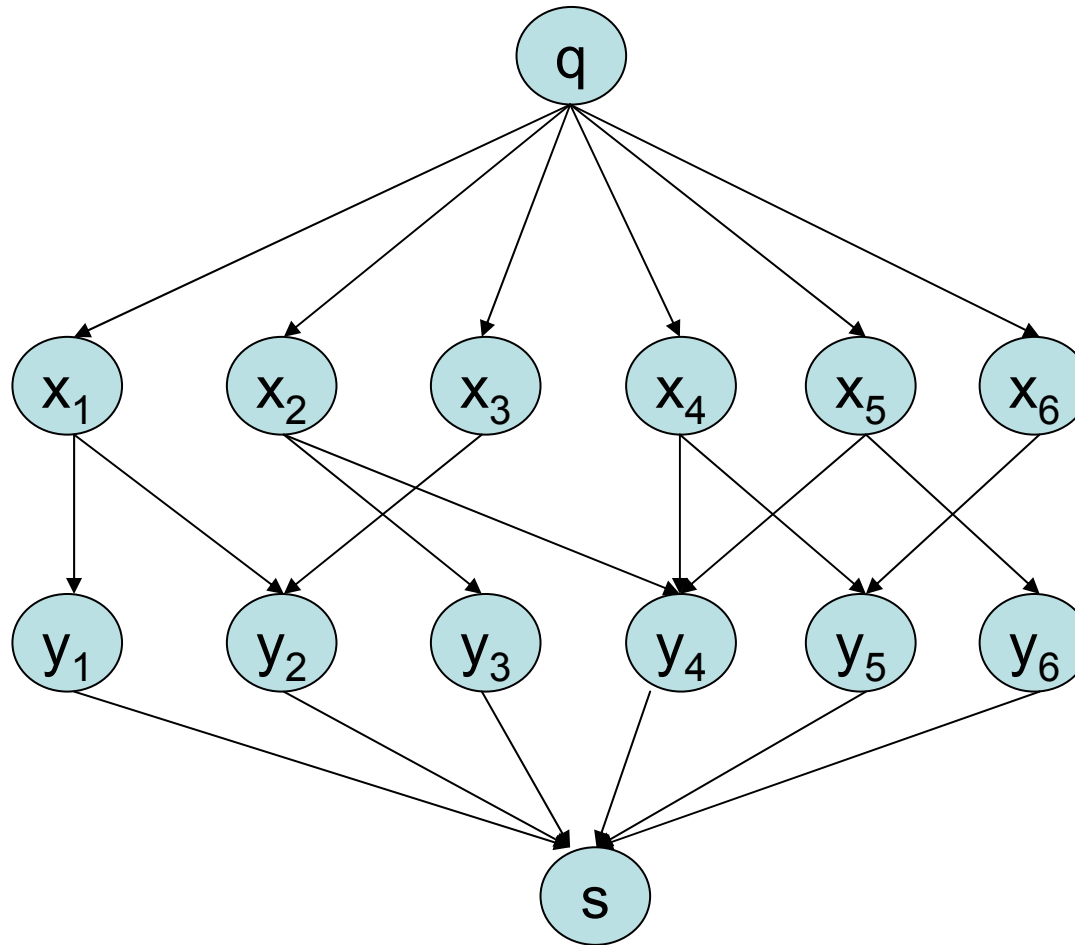


Vorgehen

- Gegeben Graph $(X \cup Y, E)$
- Hinzufügen von zwei weiteren Knoten Quelle q und Senke s
- Jede Kante (x_i, y_j) von $G = (X \cup Y, E)$ wird im Graphen $G' = (X \cup Y \cup \{q, s\}, E')$ zu einem Pfeil von x_i nach y_j .
- In E' existiert ein Pfeil von q zu jedem Knoten $x_i \in X$ und von jedem $y_j \in Y$ existiert ein Pfeil zu s
- Es ist also $E' = E \cup \{(q, x): x \in X\} \cup \{(y, s): y \in Y\}$
- Jede Kante erhält Kapazität 1

→ Maximale Zuordnung in G entspricht einem maximalen Fluss in G' .





Beispiel

- Ablösung von (x_1, y_2) durch (x_1, y_1) und (x_3, y_2) in G entspricht Erweiterungspfad $e=[q, x_3, y_2, x_1, y_1, s]$ in G'



Erweiterungspfad in G'

- Vorwärtspfeil e mit Fluss $f(e) = 0$
- Rückwärtspfeil e' mit Fluss $f(e') = 1$
- Vorwärts- und Rückwärtspfeile wechseln sich ab
- Pfad beginnt und endet mit einem Vorwärtspfeil

Entsprechung in G

- Pfad, dessen Kanten abwechselnd zur Zuordnung gehören bzw. nicht zur Zuordnung gehören.
- Wird als **alternierender** Pfad bezeichnet.
 - Beispiel: x_2, y_4, x_5, y_6

Vergrößerung der Zuordnung

- **Vergrößernd alternierender Pfad**: Alternierender Pfad mit freien Endknoten.
 - Beispiel: $y_3, x_2, y_4, x_4, y_5, x_6$
- Freie Kante wird zu gebundener und umgekehrt.

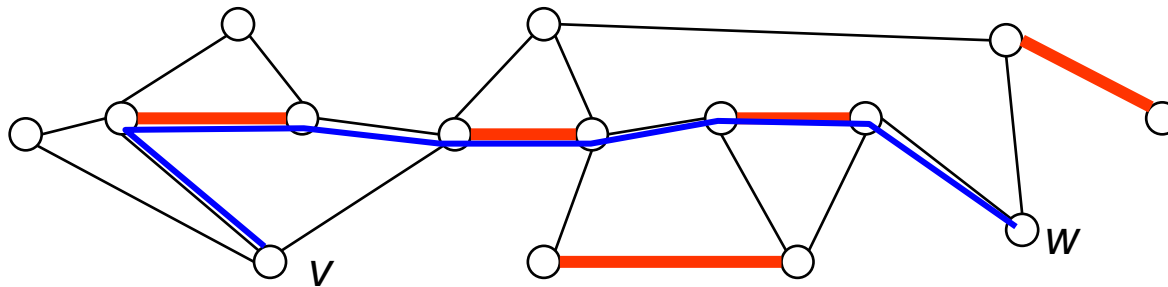


Vergrößernde Wege in allgemeinen Graphen

- Nicht nur wie bisher in bipartiten Graphen

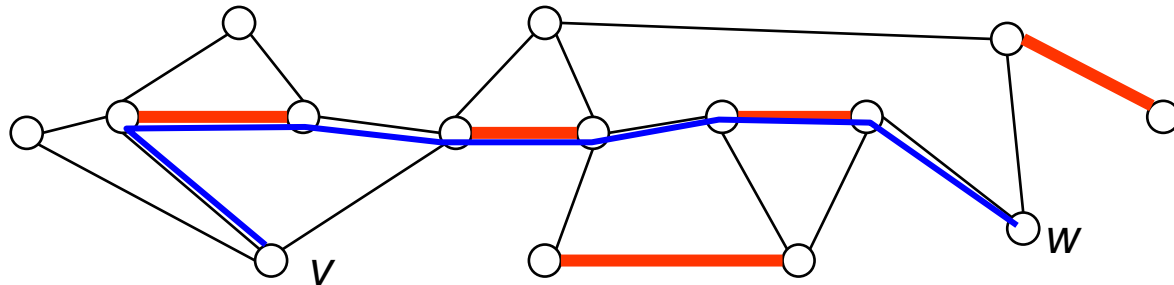
Satz von Berge

- Ein Matching M in einem Graphen G ist Maximum gdw. G keinen bzgl. M vergrößernd alternierenden Pfad enthält

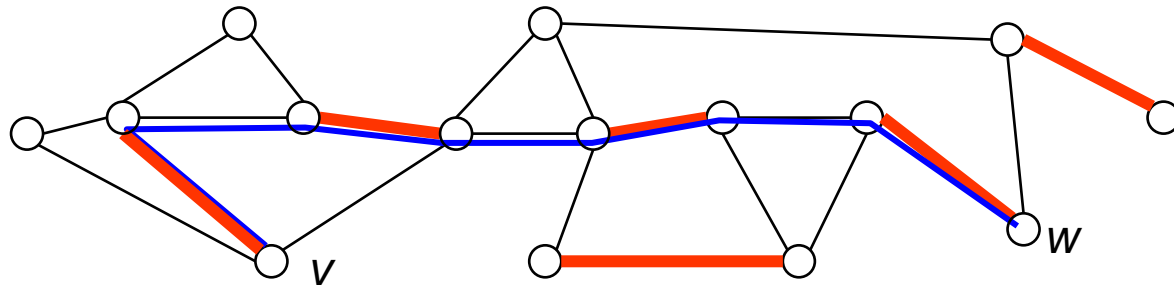


■

Angenommen, M ist maximum Matching. Gebe es einen vergrößernd alternierenden Pfad von v nach w , z.B.:



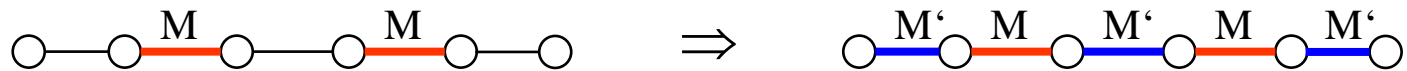
Vergrößerung durch Vertauschen der Matchingzugehörigkeit aller Kanten auf dem Pfad.



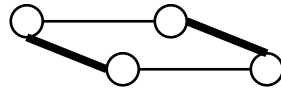
Das ist immer noch ein Matching, jetzt ein größeres als M .
Widerspruch!

Betrachte Matching M mit vergrößernd alternierendem Pfad. Sei M' aus der Vergrößerung entstehendes Matching.

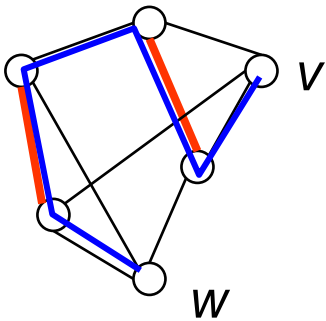
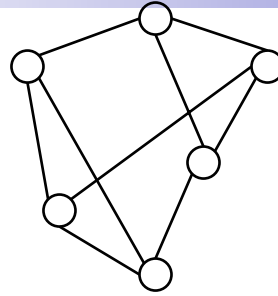
Dann ist der Pfad M - M' -alternierend:



- Beachte: Ein Zyklus kann kein vergrößernd alternierender Pfad sein!

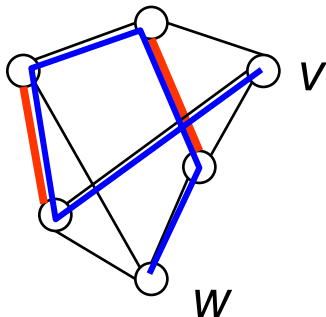
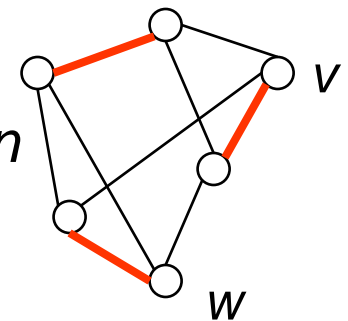


Also $|M| < |M'|$, M ist nicht Maximum.



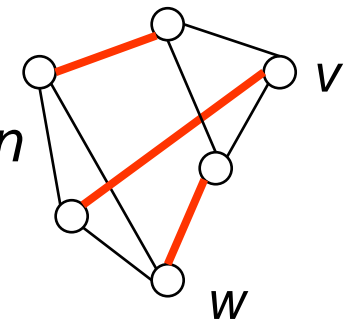
Kein maximum Matching da vergrößernd alternierender Pfad zwischen nicht gematchten Knoten v, w .

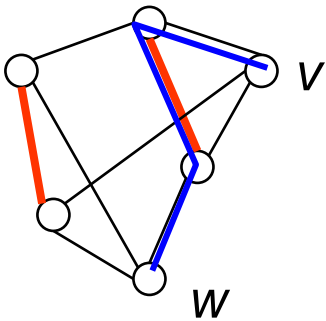
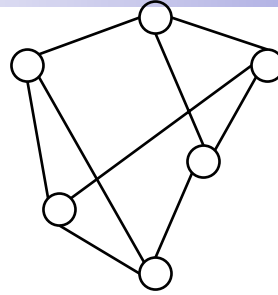
Vergrößern



Kein maximum Matching da vergrößernd alternierender Pfad zwischen nicht gematchten Knoten v, w .

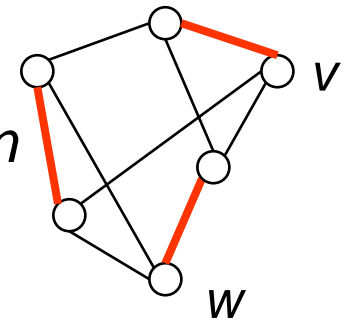
Vergrößern





Kein maximum Matching da vergrößernd alternierender Pfad zwischen nicht gematchten Knoten v, w .

Vergrößern

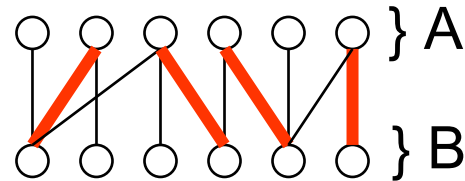


Benutze irgend einen einfachen Algorithmus um ein maximales Matching M zu finden
solange ein vergrößernder M -alternierender Pfad vorhanden ist
vertausche die M -Zugehörigkeit der Kanten auf diesem Pfad

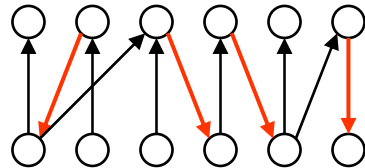
- Der Algorithmus fügt in jedem Schritt eine Kante zu M hinzu.
- Da es nur endlich viele Kanten gibt, terminiert er.
- Wenn er terminiert, hat er ein maximum Matching gefunden.
- Noch offen: Wie findet man M -alternierende Pfade?



- Finden eines alternierenden Pfades geht einfach in bipartiten Graphen G mit zwei Klassen A und B .
- Sei ein Matching gegeben:



- Definiere G' so dass alle Matching-Kanten von A nach B und alle anderen von B nach A gerichtet sind.



- Jeder Pfad wechselt zwischen M' und $G \setminus M'$.
- Fertig, sobald ein ungematchter Knoten w erreicht.

