

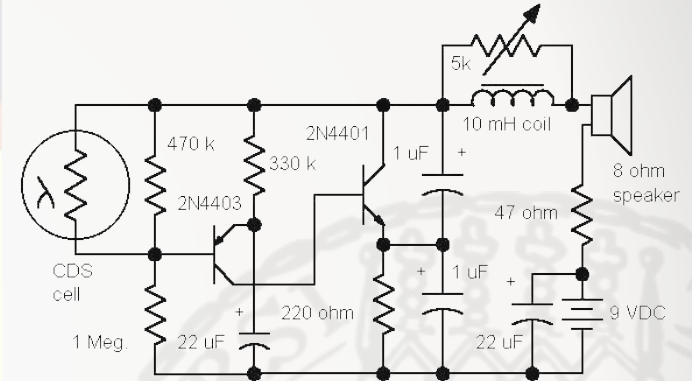
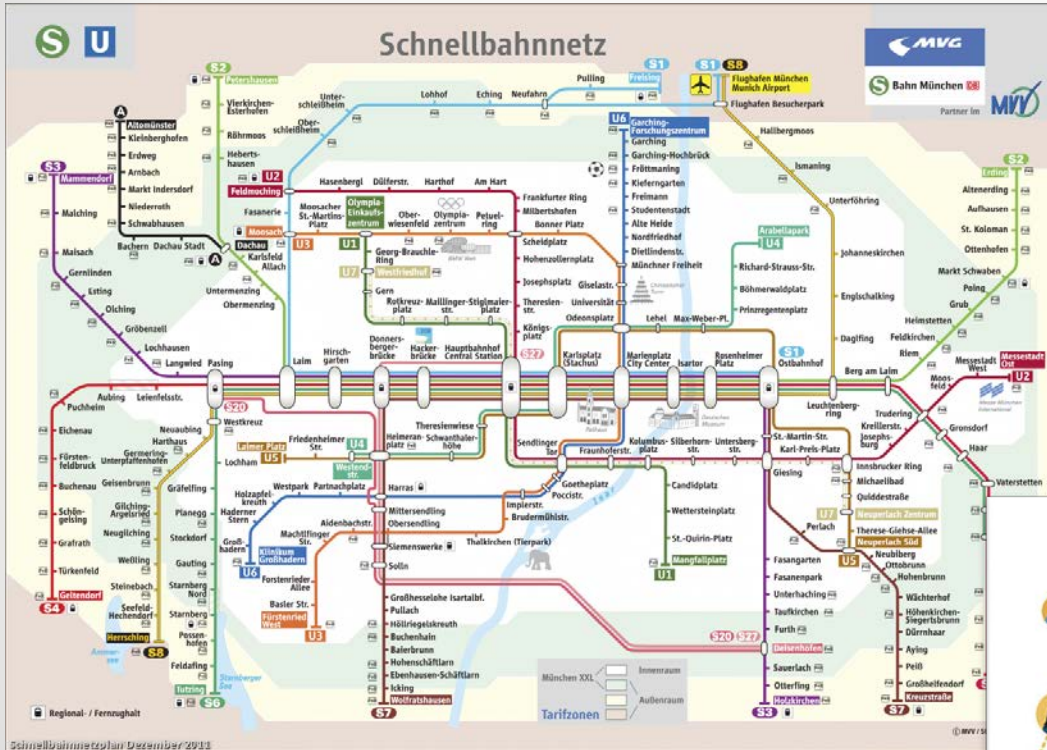
# Kapitel 5: Graphen

Graph-Repräsentationen  
Kürzeste Wege  
Minimale Spannbäume  
Flussnetzwerke



# Motivation zu Graphen

- Viele reale Fragestellungen lassen sich durch Graphen darstellen



# Motivation zu Graphen

- Bezogen auf einen Graphen ergeben sich Fragen:
  - Existiert eine Verbindung zwischen A und B?
  - Existiert eine zweite Verbindung, falls die erste blockiert ist?
  - Wie lautet die kürzeste Verbindung von A nach B?
  - Wie sieht ein minimaler Spannbaum zu einem Graphen aus?
  - Wie plane ich eine optimale Rundreise? (*Traveling Salesman Problem*)

# Gerichteter Graph

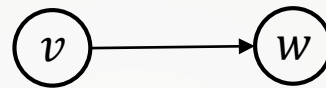
Ein *gerichteter Graph* (engl. digraph = "directed graph") ist ein Paar  $G = (V, E)$  mit einer endlichen, nichtleeren Menge  $V$ , deren Elemente Knoten (nodes, vertices) heißen, und einer Menge  $E \subseteq V \times V$ , deren Elemente Kanten (edges, arcs) heißen.

Bemerkungen:

- $|V| =$  Knotenanzahl
- $|E| \leq |V|^2 =$  Kantenanzahl
- Meist werden die Knoten durchnummeriert:  $i = 0, 1, 2, \dots, |V| - 1$

# Gerichteter Graph

Graphische Darstellung einer Kante von  $v$  nach  $w$ :



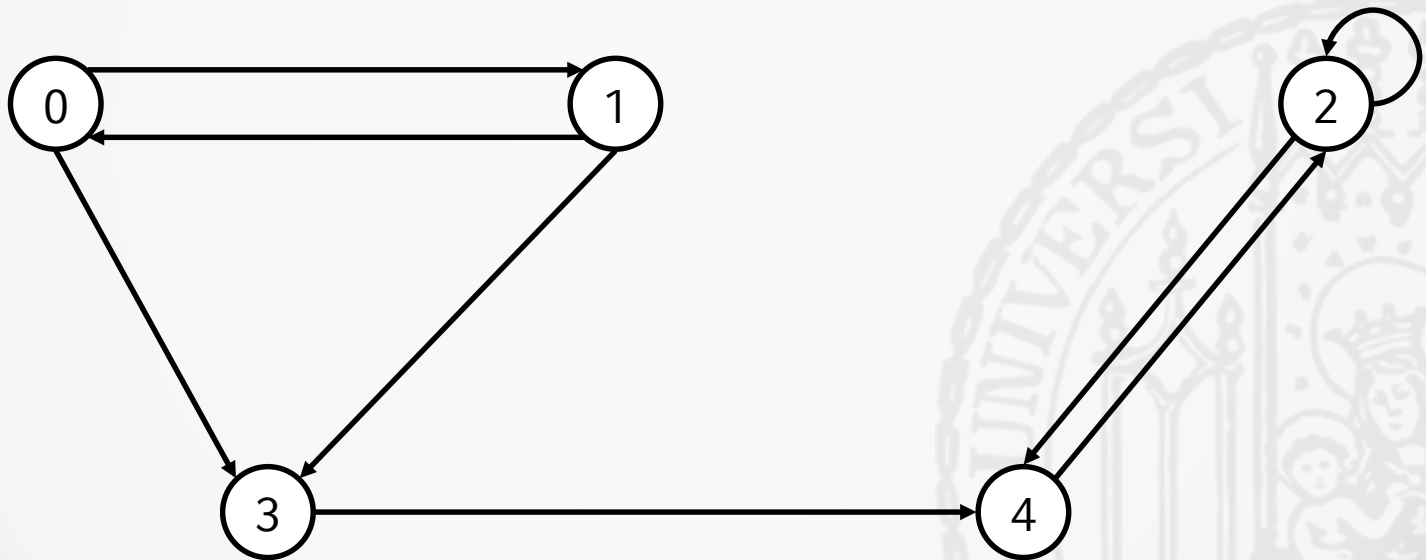
Begriffe:

- $v$  ist *Vorgänger* von  $w$
- $w$  ist *Nachfolger* von  $v$
- $v$  und  $w$  sind *Nachbarn* bzw. *adjazent*



## Gerichteter Graph: Beispiel

- $V = \{0,1,2,3,4\}$
- $E = \{(0,1), (0,3), (1,0), (1,3), (2,2), (2,4), (3,4), (4,2)\}$



## Gerichteter Graph: Definitionen

- *Grad* eines Knotens := Anzahl der ein- und ausgehenden Kanten
- Ein *Pfad* ist eine Folge von Knoten  $v_0, \dots, v_{n-1}$  mit  $(v_i, v_{i+1}) \in E$  für  $0 \leq i \leq n - 1$ , also eine Folge „zusammenhängender“ Kanten.
- *Länge eines Pfades* := Anzahl der Kanten auf dem Pfad
- Ein Pfad heißt *einfach*, wenn alle Knoten auf dem Pfad paarweise verschieden sind.
- Ein *Zyklus* ist ein Pfad mit  $v_0 = v_{n-1}$  und Länge  $n \geq 2$ .
- Ein *Teilgraph*  $G' = (V', E')$  eines Graphen  $G = (V, E)$  ist ein Graph mit  $V' \subseteq V$  und  $E' \subseteq E \cap (V' \times V')$ .

## Gerichteter Graph: Markierungen

- Man kann Markierungen oder Beschriftungen für Kanten und Knoten einführen.
- Häufig verwendet: Kostenfunktionen für Kanten
- Notation:
  - $c[v, w]$  oder  $cost(v, w)$ ,  $c(v, w)$
- Bedeutung:
  - Entfernung zwischen  $v$  und  $w$
  - Reisezeit
  - Reisekosten
  - ...

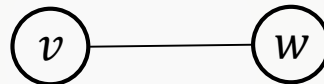


# Ungerichteter Graph

Ein ungerichteter Graph ist ein gerichteter Graph, in dem die Relation  $E$  symmetrisch ist:

$$(v, w) \in E \Rightarrow (w, v) \in E$$

Graphische Darstellung (ohne Pfeil):



Bemerkung:

Die eingeführten Begriffe (Grad eines Knoten, Pfad, ...) verstehen sich analog zu denen für gerichtete Graphen. Bisweilen sind Modifikationen erforderlich, z.B. muss ein Zyklus hier mindestens drei Knoten haben.

# Graph-Darstellungen

- Man kann je nach Zielsetzung den Graphen knoten- oder kantenorientiert abspeichern.
- Die knotenorientierte Darstellungsform ist gebräuchlicher und existiert in vielen verschiedenen Variationen.

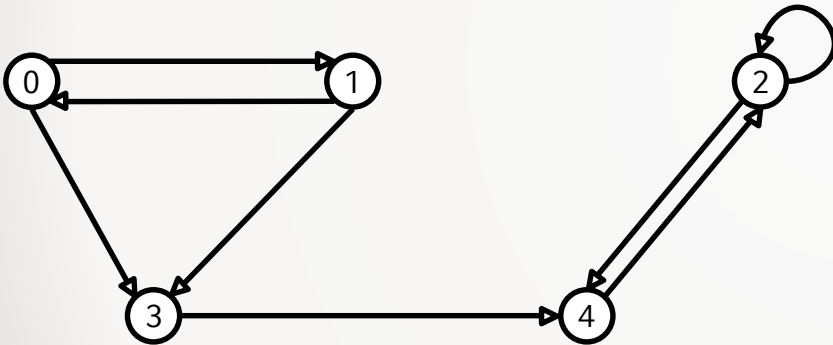
Die Adjazenzmatrix  $A$  ist eine boolesche Matrix mit:

$$A_{ij} = \begin{cases} true & \text{falls } (v_i, v_j) \in E \\ false & \text{sonst} \end{cases}$$

Eine solche Matrix  $[A_{ij}]$  lässt sich als Array  $A[i][j]$  darstellen.

# Boolesche Adjazenzmatrix - Beispiel

Für den Beispiel-Graph  $G_1$  ergibt sich folgende Adjazenzmatrix mit der Konvention *true* = 1, *false* = 0:



nach

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

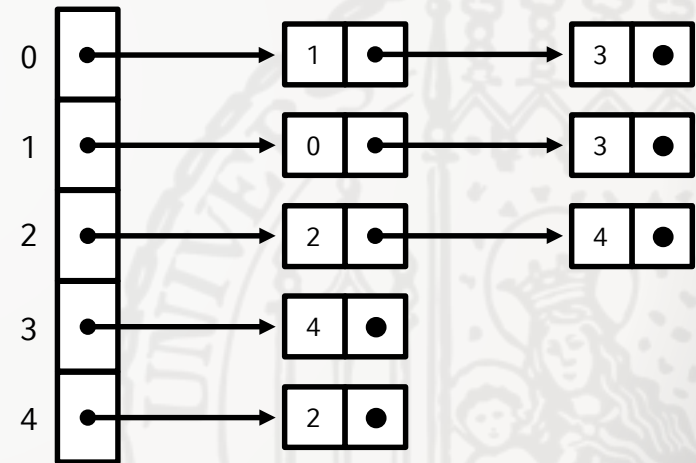
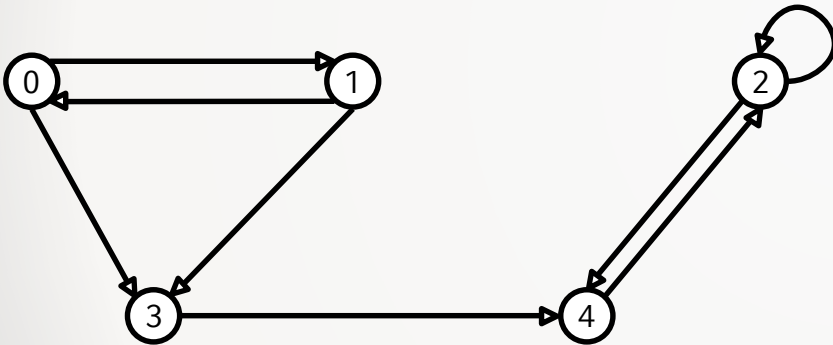
v  
o  
n

# Adjazenzmatrix

- Vorteile
  - Entscheidung, ob  $(i, j) \in E$  in Zeit  $O(1)$
- Nachteile
  - Platzbedarf stets  $O(|V|^2)$ , ineffizient falls  $|E| \ll |V|^2$
  - Initialisierung benötigt Zeit  $O(|V|^2)$
- Kantenbeschriftung
  - statt booleschen Werten Zusatzinformation (bspw. Integer) als Matrixeinträge speichern
  - Bsp: Kosten; Weglängen
  - Definition Kostenadjazenzmatrix: mit  $c(v_i, v_j)$  Kosten für die Kante zwischen  $v_i$  und  $v_j$ 
$$A_{ij} = \begin{cases} c(v_i, v_j) & (v_i, v_j) \in E \\ \infty & \text{sonst} \end{cases}$$
- Achtung: Bei boolescher Adjazenzmatrix bedeutet  $A(i, j) = 0$ , dass keine Kante besteht; bei Kostenadjazenzmatrix, dass die Kosten  $c(v_i, v_j) = 0$  sind.

# Adjazenzliste

- Für jeden Knoten wird eine Liste der Nachbarknoten angelegt.
- Für  $G_1$  ergibt sich folgende Adjazenzliste:



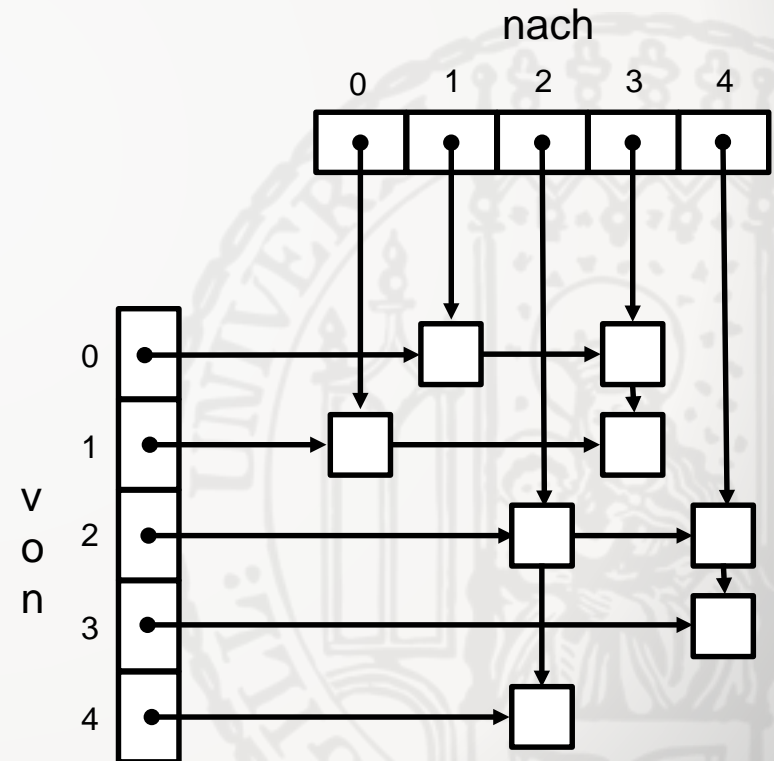
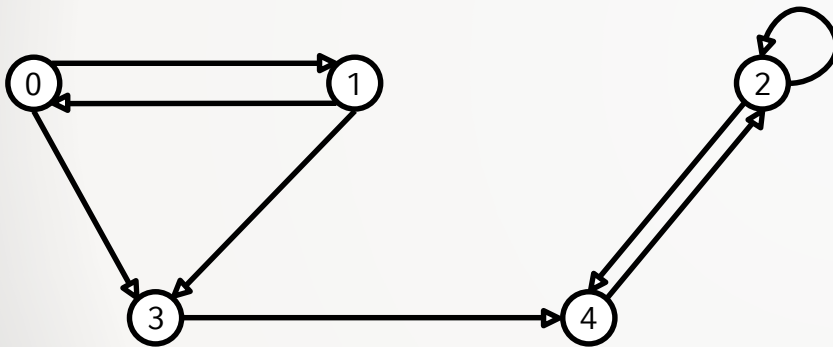


# Adjazenzliste

- Vorteile
  - geringer Platzbedarf von  $O(|V| + |E|)$
  - Initialisierung in Zeit  $O(|V| + |E|)$
- Nachteile
  - Entscheidung, ob  $(i, j) \in E$  in Zeit  $O\left(\frac{|E|}{|V|}\right)$  im Average Case
- Kantenbeschriftung
  - als Zusatzinformation bei Listenelementen

# Mischform

- Verwende zwei eindimensionale Arrays **from** und **to** mit Referenzen auf Kantenobjekte.
- Es gibt einen Referenzenpfad zu einem Objekt von **from[i]** und **to[j]** genau dann wenn der repräsentierte Graph  $G$  eine Kante von Knoten  $v_i$  zu Knoten  $v_j$  enthält.



## Mischform

### Vorteil:

- geringer Platzbedarf von  $O(|V| + |E|)$  (wie Adjazenzlisten)
- Initialisierung in Zeit  $O(|V| + |E|)$  (wie Adjazenzlisten)
- auch Vorgängerliste leicht erhältlich (wie Adjazenzmatrix)

### Nachteil:

- Entscheidung, ob Kante  $(i, j) \in E$  in Zeit  $O\left(\frac{|E|}{|V|}\right)$  im Average Case (wie Adjazenzlisten)

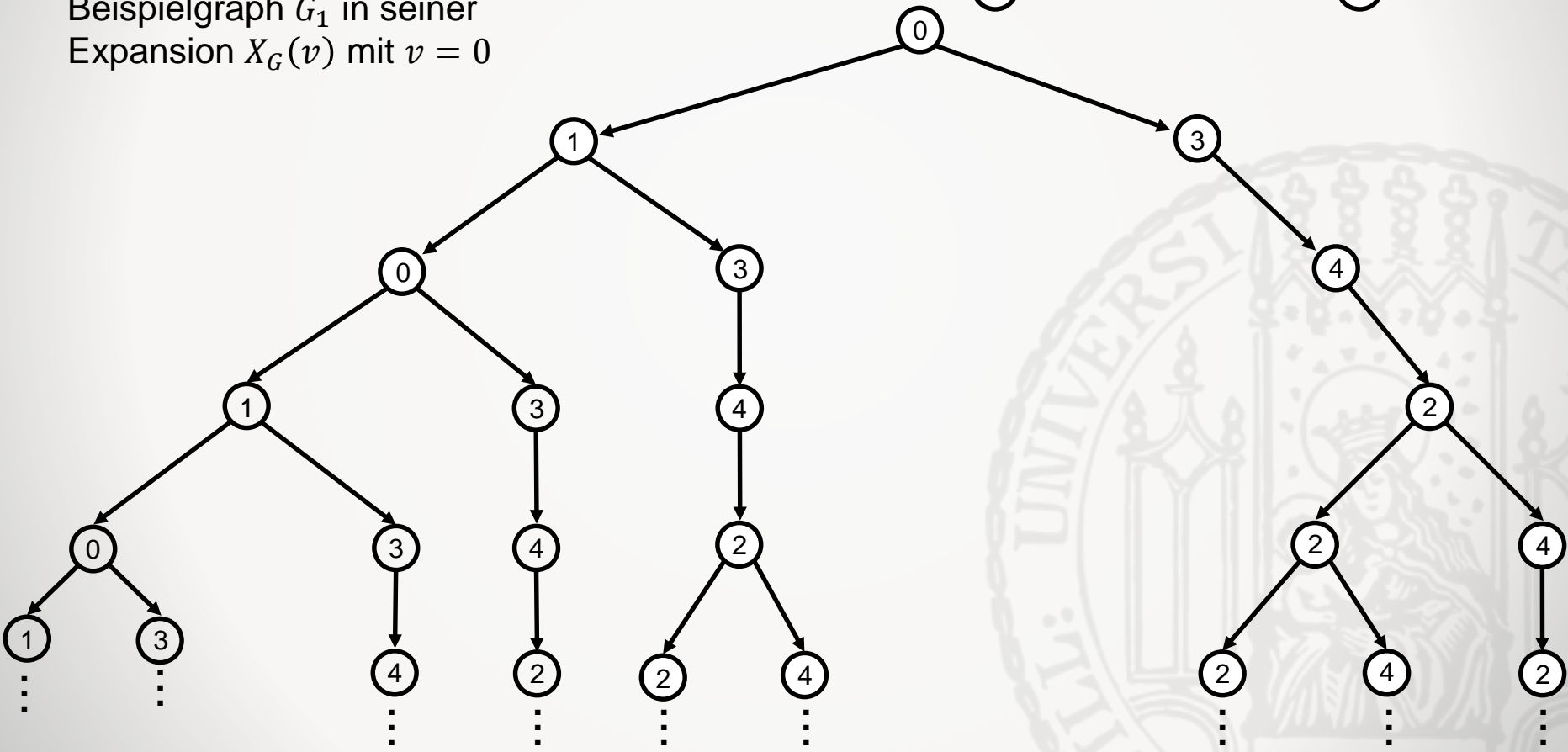
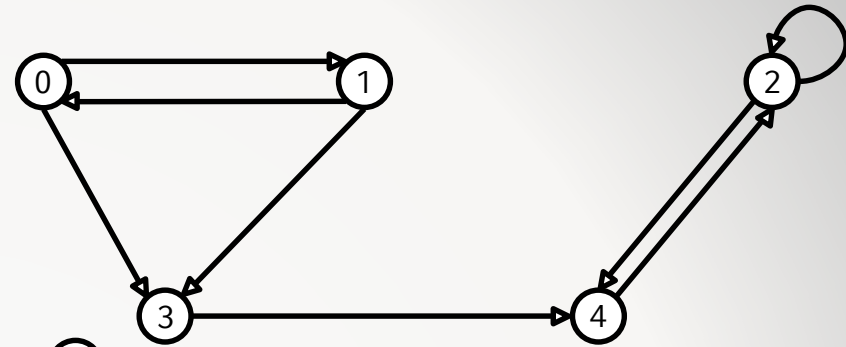
## Expansion eines Graphen

Die *Expansion*  $X_G(v)$  eines Graphen  $G$  in einem Knoten  $v$  ist ein Baum, der wie folgt definiert ist:

- Falls  $v$  keine Nachfolger hat, ist  $X_G(v)$  nur der Knoten  $v$ .
- Falls  $v_1, \dots, v_k$  die Nachfolger von  $v$  sind, ist  $X_G(v)$  der Baum mit der Wurzel  $v$  und den Teilbäumen  $X_G(v_1), \dots, X_G(v_k)$ .

# Expansion eines Graphen: Beispiel

Beispielgraph  $G_1$  in seiner Expansion  $X_G(v)$  mit  $v = 0$





## Expansion eines Graphen: Anmerkungen

- Die Knoten des Graphen können mehrfach im Baum vorkommen.
- Ein Baum ist unendlich, falls der Graph Zyklen hat.
- Der Baum  $X_G(v)$  stellt die Menge aller Pfade dar, die von  $v$  ausgehen.

# Graph-Durchlauf

Entspricht Baum-Durchlauf durch Expansion (ggf. mit Abschneiden)

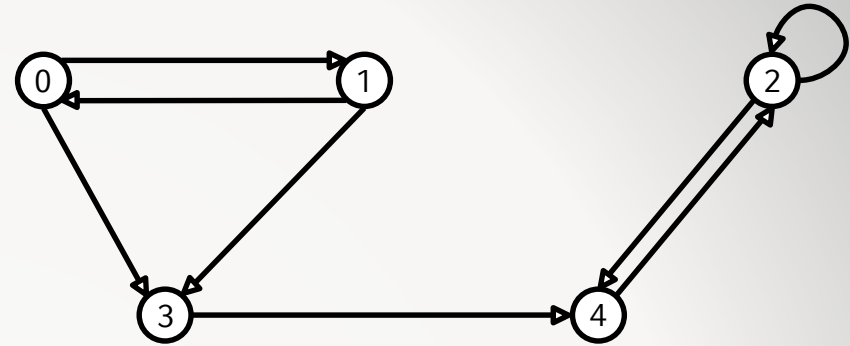
- Tiefendurchlauf: preorder traversal (depth first)
- Breitendurchlauf: level order traversal

Wichtige Modifikation:

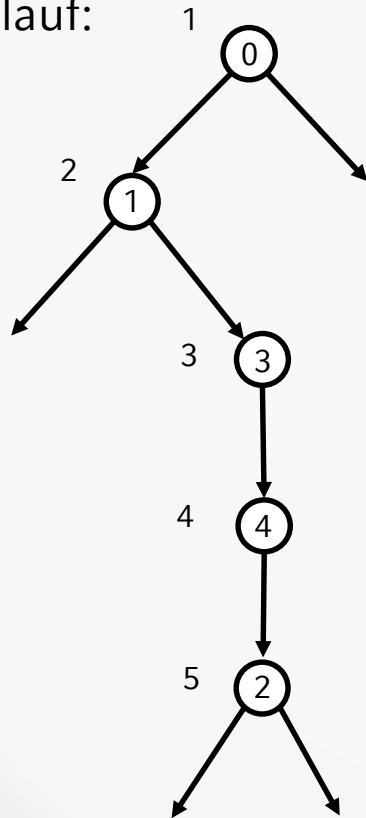
1. Schon besuchte Knoten müssen markiert werden, weil Graphknoten im Durchlauf mehrfach vorkommen können. (Zyklen!)
2. Abbruch des Durchlaufs bei schon besuchten Knoten.

# Graph-Durchlauf: Beispiel

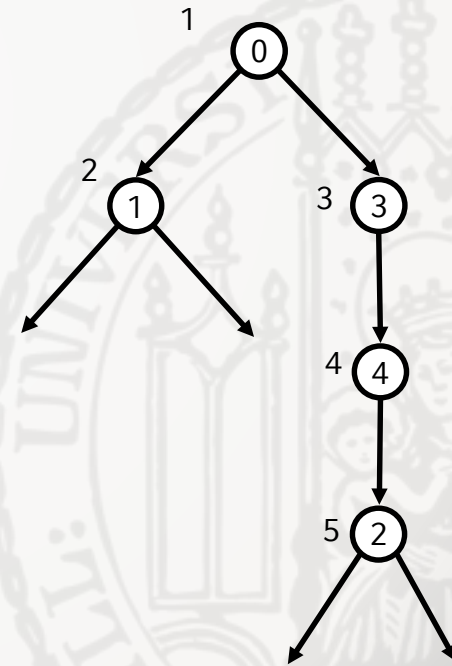
$G_1$  mit Startknoten:  $v = 0$



Tiefendurchlauf:



Breitendurchlauf:



# Ansatz für Graph-Durchlauf

1. Initialisierung: markiere alle Knoten als „not visited“
2. Abarbeiten der Knoten
  - if** (node „not visited“) **then**
    - bearbeite
    - markiere: „visited“
    - weitergehen zu Nachfolger

Für die Markierung „visited“ reicht der Typ boolean. Für andere Berechnungen auf Graphen benötigt man aber auch mehr als die zwei Werte „true“ and „false“.

## Markierungen beim Durchlauf

Während des Graph-Durchlaufs werden folgende Markierungen für die Graph-Knoten verwendet:

- Ungesehene Knoten (unseen vertices):  
Knoten, die noch nicht erreicht worden sind:  $val[v] = 0$
- Baum-Knoten (tree vertices):  
Knoten, die schon besucht und abgearbeitet sind. Diese Knoten ergeben die Expansion:  $val[v] = id > 0$
- Rand-Knoten (fringe vertices), aktive Knoten:  
Knoten, die über eine Kante mit einem Baum-Knoten verbunden sind:  $val[v] = -1$

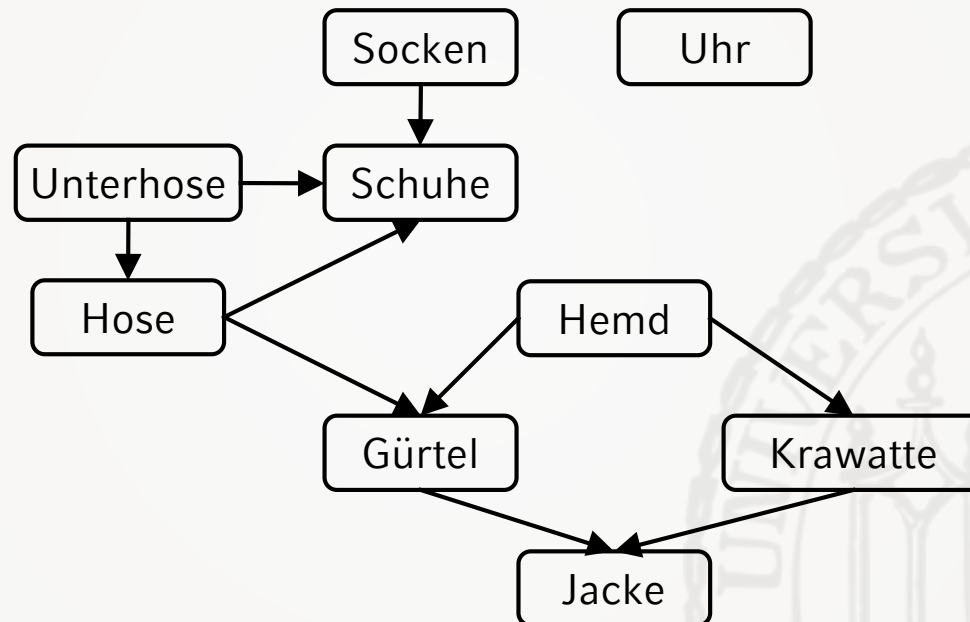


## Beliebiges Auswahlkriterium

- Start: Markiere den Startknoten als Rand-Knoten und alle anderen Knoten als ungesehene Knoten.
  - Schleife: **repeat**
    - Wähle einen Rand-Knoten  $x$  mittels eines Auswahlkriteriums (depth first, breadth first, priority first).
    - Dazu: Priority Queue,
    - Markiere  $x$  als Baum-Knoten und bearbeite  $x$ .
    - Markiere alle ungesehenen Nachbar-Knoten von  $x$  als Rand-Knoten.
- ... **until** (alle Knoten abgearbeitet)

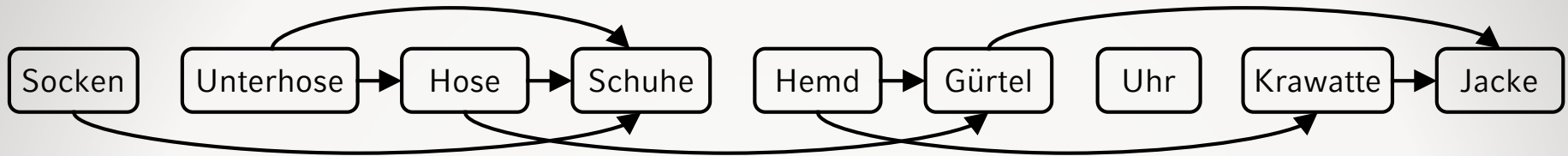
# Topologisches Sortieren

- Abhängigkeiten zwischen Aktionen („erst  $x$ , dann  $y$ “)
- Gesucht: Lineare Ordnung der Knoten, sodass für alle  $(u, v) \in E$  der Knoten  $u$  „vor“  $v$  liegt. → Abarbeitungsfolge/-plan der Aktionen

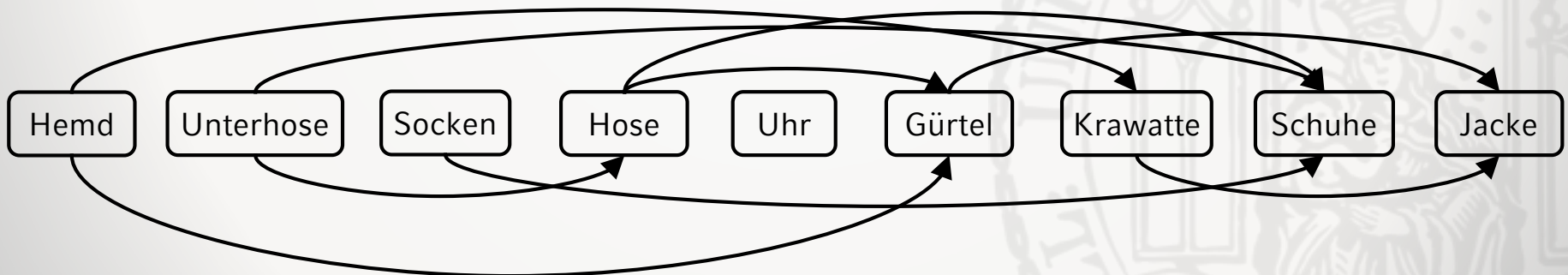


- formal: Einbettung einer Halbordnung/partiellen Ordnung (reflexiv, transitiv, antisymmetrisch) in eine lineare/totale Ordnung

# Topologisches Sortieren und DAGs



- topologische Sortierung genau dann möglich, wenn Graph keine Zyklen enthält
  - anschaulich: alle Kanten „zeigen nach rechts“
- DAG: directed acyclic graph / gerichteter Graph ohne Zyklen
- topologische Sortierung im Allgemeinen nicht eindeutig



## Idee zum Algorithmus

- Nutze Informationen über Anzahl der Vorgänger eines Knotens
  - Knoten ohne Vorgänger können direkt abgearbeitet werden
  - Falls Knoten abgearbeitet wurde, verringert sich für alle seine Nachfolger die Anzahl deren Vorgänger um 1
  - falls für einen Knoten die Vorgängeranzahl 0 erreicht, kann auch dieser ausgegeben werden
    - alle seine Vorgängerknoten wurden bereits abgearbeitet
- Bemerkung: Es existiert ein alternativer Algorithmus, der auf der Tiefensuche basiert.

# Algorithmus zum topologischen Sortieren

```
TopoSort(DAG G){
  S = { }      // Menge der abgearbeiteten Knoten
  for (i=0,i<n, i++) {
    P[i] = Anzahl Vorgänger des Knotens i
  }
  while V\S ≠ { } {
    wähle w ∈ V \ S mit P[i]==0;
    gebe w aus;
    S = S ∪ {w};
    für jeden Nachfolger v von w {
      P[v]--;
    }
  }
}
```



# Kürzeste Wege

- Problemstellung: Suche kürzesten Weg
  1. Von einem Knoten zu allen anderen:  
„Single Source Shortest Path“
  2. Von allen Knoten zu einem Ziel:  
„Single Destination Shortest Path“
  3. Von allen Knoten zu allen anderen:  
„All Pairs Shortest Path“
- Gegeben: Gerichteter Graph  $G$  mit Kostenfunktion (=Adjazenzmatrix)

$$c[v, w] \begin{cases} \geq 0 & , \text{ falls eine Kante von } v \text{ nach } w \text{ existiert} \\ = \infty & , \text{ falls keine Kante von } v \text{ nach } w \text{ existiert} \\ = 0 & , \text{ für } w = v \end{cases}$$

Startknoten  $v_0$ , Endknoten  $w$

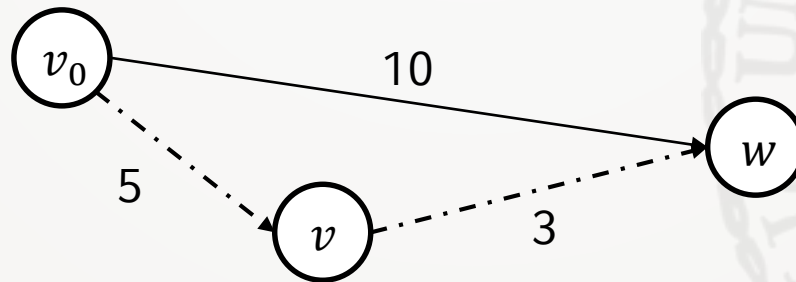
- Gesucht: Pfad von  $v_0$  zu jedem Knoten  $w$  mit minimalen Gesamtkosten

## Eigenschaften von Pfaden

- Pfadkosten können durch Erweiterung eines Pfades nur wachsen, da Kantengewichte stets positiv sind.
- Falls beste Pfade von  $v_0$  zu allen anderen Knoten  $V - \{v_0\}$  höhere Kosten haben, ist der kürzeste Pfad bereits gefunden.
- Ein kürzester Pfad hat keinen Zyklus.
- Ein kürzester Pfad hat max.  $(|V| - 1)$  Kanten.
- Notation:
  - $S_k$ : Menge von  $k$  Knoten  $v$  mit  $k$  besten Pfaden von  $v_0$  nach  $v$
  - $D_k(v)$ : Kosten/Distanz des besten Pfades von  $v_0$  über maximal  $k$  Knoten in  $S_k$  nach  $v$

# Dijkstra-Algorithmus

- Edsger Wybe Dijkstra (1930-2002): niederländischer Informatiker & Turingpreisträger
- Idee:
  - Wir speichern im Array  $D$  für jeden Knoten  $v$  die aktuell gültige Kostenschätzung.
  - Als Initialisierung verwenden wir die Kosten der direkten Pfade aus der Adjazenzmatrix  $c[v, w]$ .
  - In jedem Schritt versuchen wir alle Pfade zu verbessern, indem wir mögliche Zwischenknoten untersuchen, die den Pfad eventuell kürzer machen.



# Dijkstra-Algorithmus: Implementierung

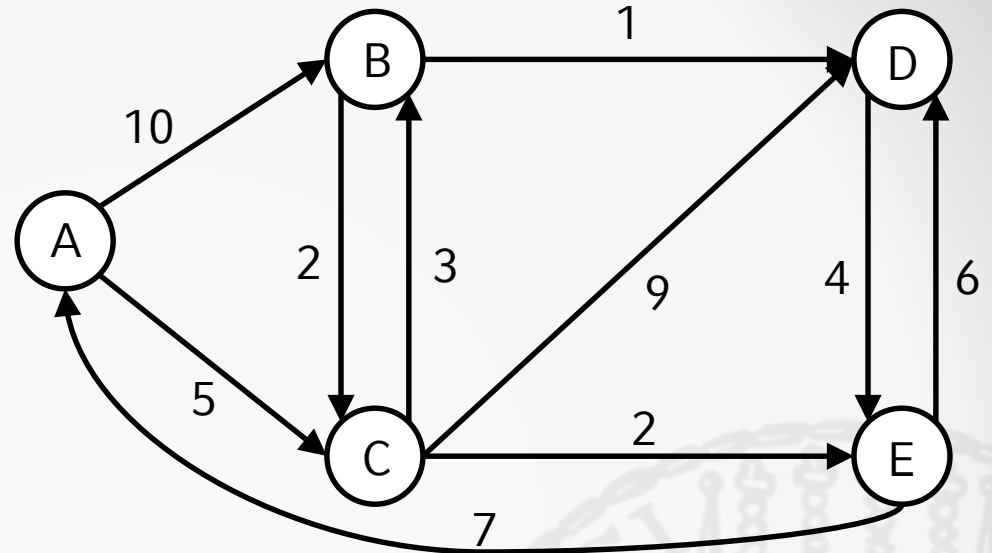
$S$ : Menge der bereits abgearbeiteten Knoten

$D$ : aktuelle Kostenschätzung für den Pfad von  $v_0$  zu allen anderen Knoten

```
Dijkstra( $G, v_0$ ) {  
   $S \leftarrow \{v_0\}$   
  forall  $v \in V$  {  
     $D(v) \leftarrow c[v_0, v]$   
  }  
  while  $V - S \neq \emptyset$  {  
     $w_{min} \leftarrow \operatorname{argmin}_{w \in V - S} D(w)$   
     $S \leftarrow S \cup \{w_{min}\}$   
    for each  $v \in V - S$  {  
       $D(v) = \min(D[v], D[w_{min}] + c[w_{min}, v])$   
    }  
  }  
}
```

# Dijkstra-Algorithmus graphisch

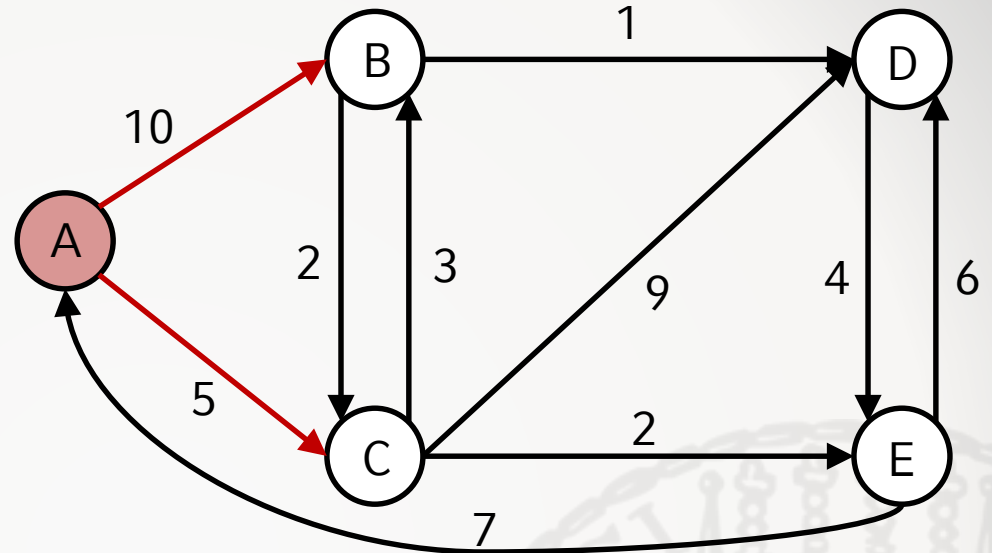
Adjazenzmatrix				
0	10	5	$\infty$	$\infty$
$\infty$	0	2	1	$\infty$
$\infty$	3	0	9	2
$\infty$	$\infty$	$\infty$	0	4
7	$\infty$	$\infty$	6	0



$k$	$w_k$	$S_k$	$D_k(B)$	$D_k(C)$	$D_k(D)$	$D_k(E)$

# Dijkstra-Algorithmus graphisch

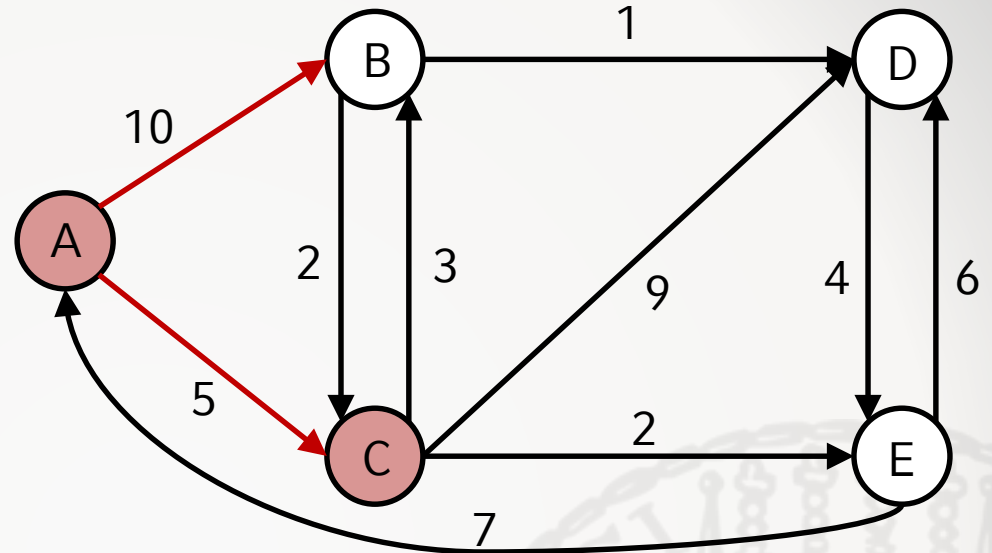
Adjazenzmatrix				
0	10	5	$\infty$	$\infty$
$\infty$	0	2	1	$\infty$
$\infty$	3	0	9	2
$\infty$	$\infty$	$\infty$	0	4
7	$\infty$	$\infty$	6	0



$k$	$w_k$	$S_k$	$D_k(B)$	$D_k(C)$	$D_k(D)$	$D_k(E)$
0	—	{A}	10	5	$\infty$	$\infty$

# Dijkstra-Algorithmus graphisch

Adjazenzmatrix				
0	10	5	$\infty$	$\infty$
$\infty$	0	2	1	$\infty$
$\infty$	3	0	9	2
$\infty$	$\infty$	$\infty$	0	4
7	$\infty$	$\infty$	6	0

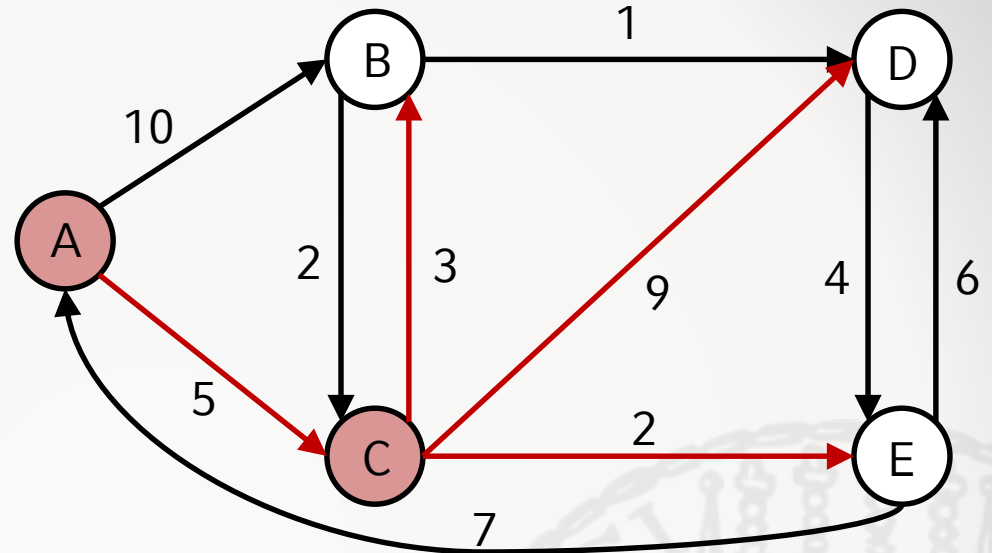


$k$	$w_k$	$S_k$	$D_k(B)$	$D_k(C)$	$D_k(D)$	$D_k(E)$
0	—	{A}	10	5	$\infty$	$\infty$



# Dijkstra-Algorithmus graphisch

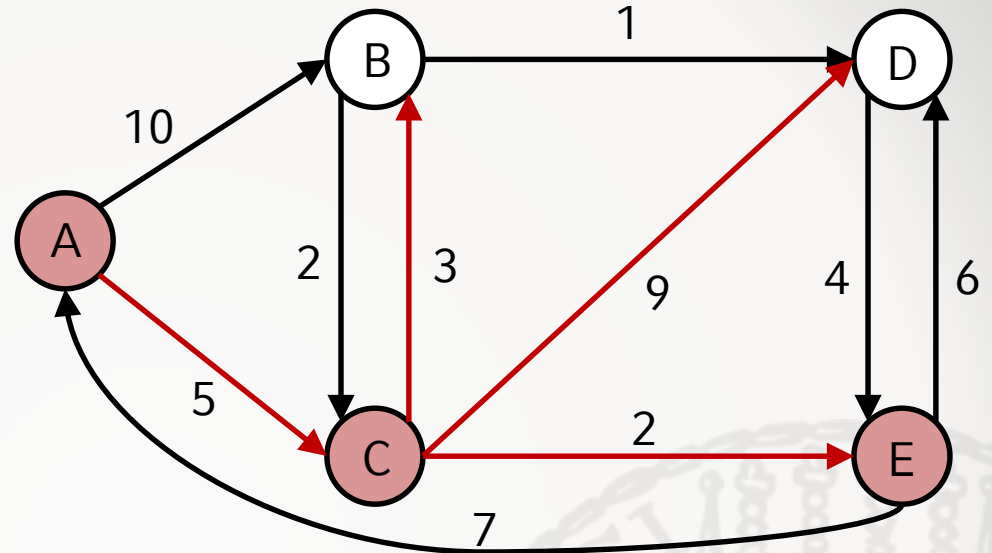
Adjazenzmatrix				
0	10	5	$\infty$	$\infty$
$\infty$	0	2	1	$\infty$
$\infty$	3	0	9	2
$\infty$	$\infty$	$\infty$	0	4
7	$\infty$	$\infty$	6	0



$k$	$w_k$	$S_k$	$D_k(B)$	$D_k(C)$	$D_k(D)$	$D_k(E)$
0	—	{A}	10	5	$\infty$	$\infty$
1	C	{A, C}	8	5	14	7

# Dijkstra-Algorithmus graphisch

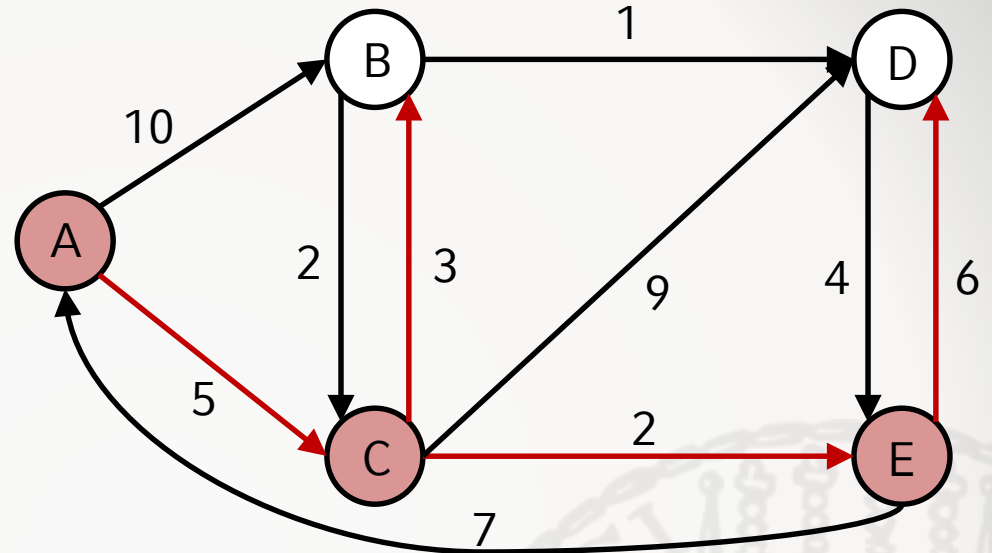
Adjazenzmatrix				
0	10	5	$\infty$	$\infty$
$\infty$	0	2	1	$\infty$
$\infty$	3	0	9	2
$\infty$	$\infty$	$\infty$	0	4
7	$\infty$	$\infty$	6	0



$k$	$w_k$	$S_k$	$D_k(B)$	$D_k(C)$	$D_k(D)$	$D_k(E)$
0	—	{A}	10	5	$\infty$	$\infty$
1	C	{A, C}	8	5	14	7

# Dijkstra-Algorithmus graphisch

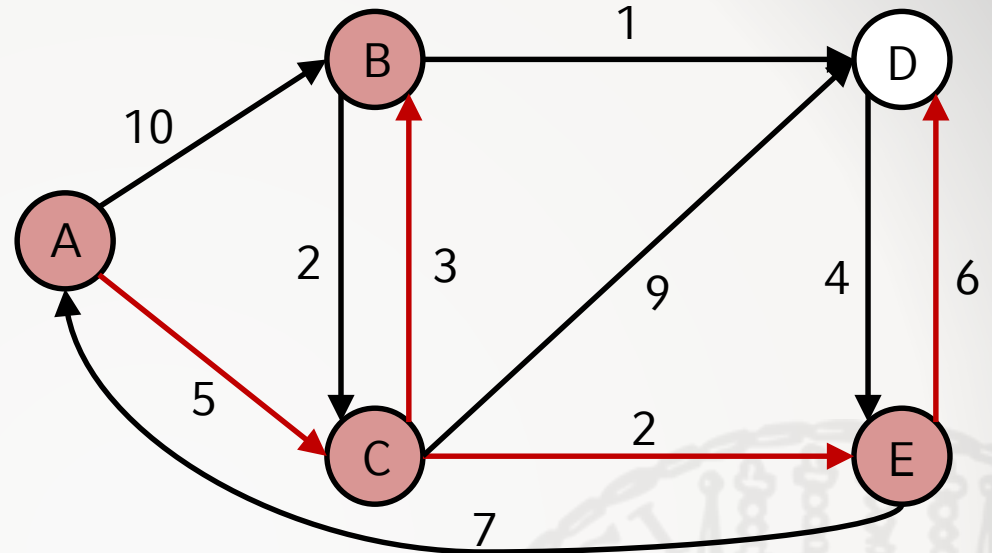
Adjazenzmatrix				
0	10	5	$\infty$	$\infty$
$\infty$	0	2	1	$\infty$
$\infty$	3	0	9	2
$\infty$	$\infty$	$\infty$	0	4
7	$\infty$	$\infty$	6	0



$k$	$w_k$	$S_k$	$D_k(B)$	$D_k(C)$	$D_k(D)$	$D_k(E)$
0	—	{A}	10	5	$\infty$	$\infty$
1	C	{A, C}	8	5	14	7
2	E	{A, C, E}	8	5	13	7

# Dijkstra-Algorithmus graphisch

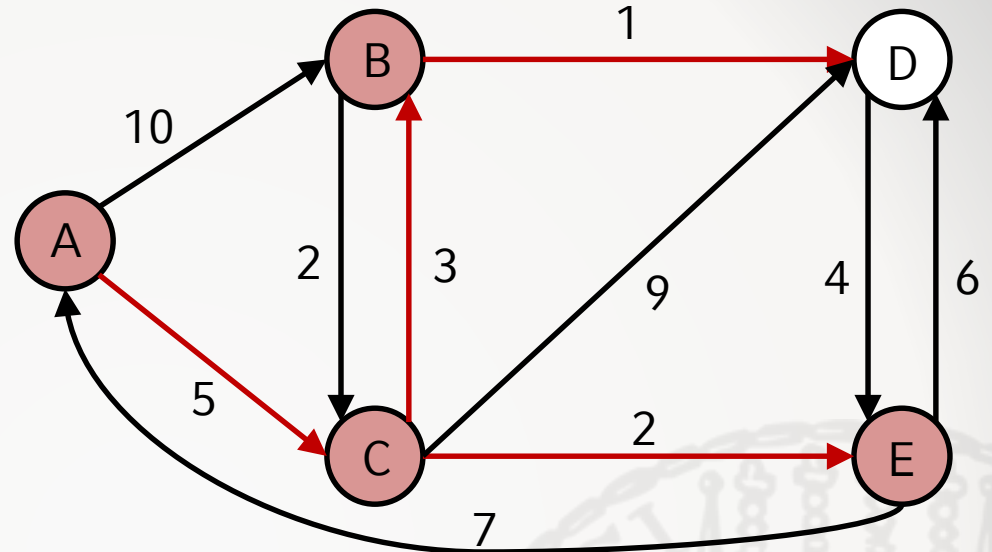
Adjazenzmatrix				
0	10	5	$\infty$	$\infty$
$\infty$	0	2	1	$\infty$
$\infty$	3	0	9	2
$\infty$	$\infty$	$\infty$	0	4
7	$\infty$	$\infty$	6	0



$k$	$w_k$	$S_k$	$D_k(B)$	$D_k(C)$	$D_k(D)$	$D_k(E)$
0	—	{A}	10	5	$\infty$	$\infty$
1	C	{A, C}	8	5	14	7
2	E	{A, C, E}	<b>8</b>	5	13	7

# Dijkstra-Algorithmus graphisch

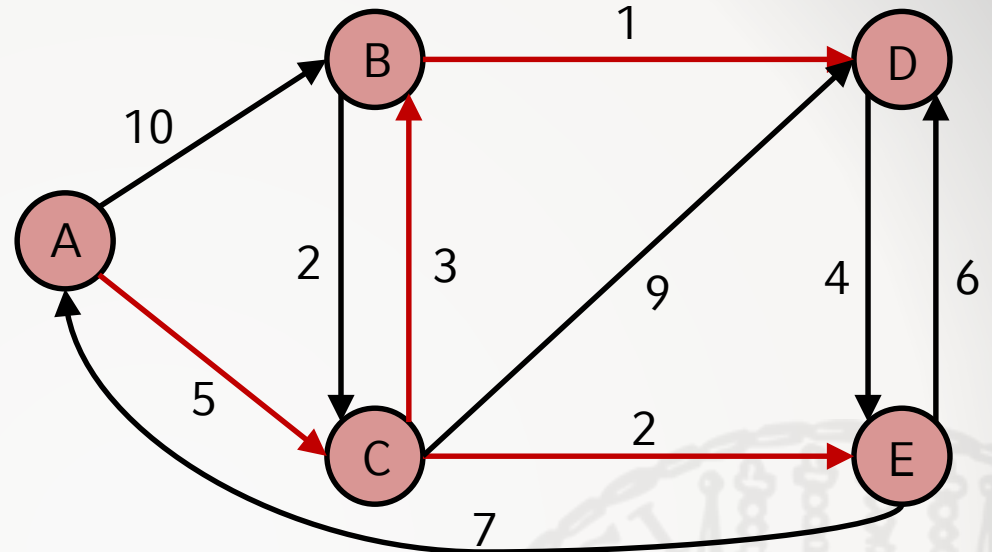
Adjazenzmatrix				
0	10	5	$\infty$	$\infty$
$\infty$	0	2	1	$\infty$
$\infty$	3	0	9	2
$\infty$	$\infty$	$\infty$	0	4
7	$\infty$	$\infty$	6	0



$k$	$w_k$	$S_k$	$D_k(B)$	$D_k(C)$	$D_k(D)$	$D_k(E)$
0	—	{A}	10	5	$\infty$	$\infty$
1	C	{A, C}	8	5	14	7
2	E	{A, C, E}	8	5	13	7
3	B	{A, C, E, B}	8	5	9	7

# Dijkstra-Algorithmus graphisch

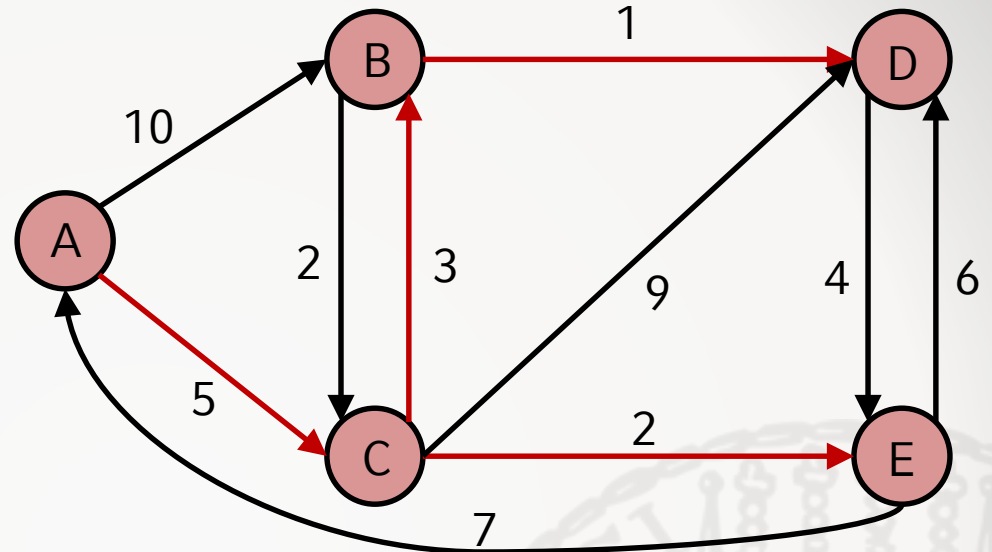
Adjazenzmatrix				
0	10	5	$\infty$	$\infty$
$\infty$	0	2	1	$\infty$
$\infty$	3	0	9	2
$\infty$	$\infty$	$\infty$	0	4
7	$\infty$	$\infty$	6	0



$k$	$w_k$	$S_k$	$D_k(B)$	$D_k(C)$	$D_k(D)$	$D_k(E)$
0	—	{A}	10	5	$\infty$	$\infty$
1	C	{A, C}	8	5	14	7
2	E	{A, C, E}	8	5	13	7
3	B	{A, C, E, B}	8	5	<b>9</b>	7

# Dijkstra-Algorithmus graphisch

Adjazenzmatrix				
0	10	5	$\infty$	$\infty$
$\infty$	0	2	1	$\infty$
$\infty$	3	0	9	2
$\infty$	$\infty$	$\infty$	0	4
7	$\infty$	$\infty$	6	0

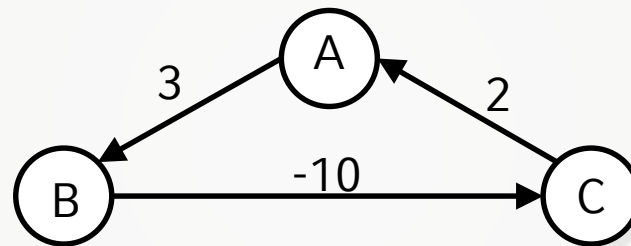


$k$	$w_k$	$S_k$	$D_k(B)$	$D_k(C)$	$D_k(D)$	$D_k(E)$
0	—	{A}	10	5	$\infty$	$\infty$
1	C	{A, C}	8	5	14	7
2	E	{A, C, E}	8	5	13	7
3	B	{A, C, E, B}	8	5	9	7
4	D	{A, C, E, B, D}	8	5	9	7



# Dijkstra-Algorithmus: Analyse

- Liefert optimale Lösung, nicht nur Näherung
- Falls Zyklen mit negativen Kosten zugelassen wären, gäbe es keinen eindeutigen Pfad mit minimalen Kosten mehr



- Komplexität:
  - Falls  $G$  zusammenhängend, mit Adjazenzmatrix  $O(|V|^2)$
  - Einsatz als „All Pairs Shortest Path“ prinzipiell möglich, ergibt Zeitkomplexität  $O(|V| \cdot |V|^2) = O(|V|^3)$ .

# Floyd-Algorithmus

- Robert W Floyd (1936 – 2001):  
Amerikanischer Informatiker & Turingpreisträger
- „All Pairs Shortest Path“
- Gegeben: Gerichteter Graph  $G$  mit Kostenfunktion

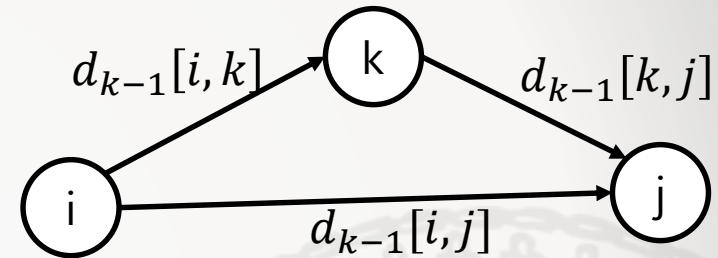
$$c[v, w] \begin{cases} \geq 0 & \text{falls eine Kante } v \text{ nach } w \text{ existiert} \\ = \infty & \text{falls keine Kante } v \text{ nach } w \text{ existiert} \\ = 0 & v = w \end{cases}$$

- Gesucht: Pfad von jedem Knoten  $v$  zu jedem Knoten  $w$  mit minimalen Gesamtkosten
- Idee:
  - Existierende Kanten im Graphen zugrunde legen
  - Versuche sukzessive, zwei Knoten über einen Zwischenknoten günstiger zu verbinden als bisher
  - Lösung: Dynamische Programmierung

# Floyd-Algorithmus: Dynamische Programmierung

- Grundidee

- Betrachte alle Knoten der Reihe nach als mögliche Zwischenknoten  $k$
- Speicherung in einer Matrix  $d[i, j]$ , die in jedem Schritt aktualisiert wird



- Initialisierung durch direkte Kanten des gegebenen Graphen
  - Für alle  $i, j \in \{1, \dots, |V|\}$ : setze  $d_0[i, j] = c[i, j]$
  - Entspricht Lösung der Elementarprobleme
- Aktualisiere Matrix  $d[i, j]$  für Zwischenknoten  $k$ :
  - Sind Wege über Knoten  $k$  günstiger als bisherige Wege?
  - $d_k[i, j] = \min\{d_{k-1}[i, j], d_{k-1}[i, k] + d_{k-1}[k, j]\}$
  - Entspricht Zusammensetzen der Teilergebnisse zur Gesamtlösung

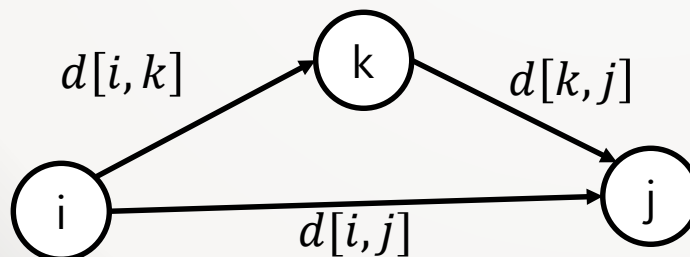
# Floyd-Algorithmus: Pseudo-Code

Gegeben: Kosten  $c[v, w] \in \mathbb{R} \cup \{\infty\}$  von Knoten  $v$  nach  $w$

Für alle Knotenpaare  $i, j \in \{0, \dots, |V| - 1\}$   
 $d[i][j] \leftarrow c[i, j]$

Für alle  $i \in \{0, \dots, |V| - 1\}$   
Für alle  $j \in \{0, \dots, |V| - 1\}$   
Für alle  $k \in \{0, \dots, |V| - 1\}$

$d[i][j] \leftarrow \min(d[i][j], d[i][k] + d[k][j])$



Initialisierung von Matrix  $d$ :

- Jeder Knoten hat Distanz 0 zu sich selbst
- Sonst übernehmen wir erst einmal die direkten (schon bekannten) Verbindungen

Falls Weg über  $k$  besser / kürzer als bisher bester Weg, ist dieser Weg nun der Favorit.

# Floyd-Algorithmus: Weginformationen

Gegeben: Kosten  $c[v, w] \in \mathbb{R} \cup \{\infty\}$  von Knoten  $v$  nach  $w$

Für alle Knotenpaare  $i, j \in \{0, \dots, |V| - 1\}$

$$d[i][j] \leftarrow c[i, j]$$

$$P[i][j] \leftarrow j \text{ (sonst undef.)}$$

Für alle  $i \in \{0, \dots, |V| - 1\}$

Für alle  $j \in \{0, \dots, |V| - 1\}$

Für alle  $k \in \{0, \dots, |V| - 1\}$

$$d[i][j] \leftarrow \min(d[i][j], d[i][k] + d[k][j])$$

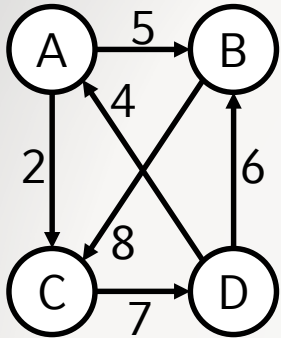
$$P[i][j] \leftarrow k$$

$$\begin{pmatrix} P_{11} & \cdots & P_{1j} & \cdots & P_{1n} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ P_{i1} & \cdots & P_{ij} & \cdots & P_{in} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ P_{n1} & \cdots & P_{nj} & \cdots & P_{nn} \end{pmatrix}$$

Der kürzeste Weg von  $i$  nach  $j$  verläuft über  $P_{ij}$ .

$P$  speichert für zwei Knoten  $i, j$  den Knoten  $k$ , der auf dem optimalen Pfad als nächster Knoten gewählt wird.

# Floyd-Algorithmus: Beispiel



Initialisierung:

$$\begin{pmatrix} 0 & 5 & 2 & \infty \\ \infty & 0 & 8 & \infty \\ \infty & \infty & 0 & 7 \\ 4 & 6 & \infty & 0 \end{pmatrix}$$

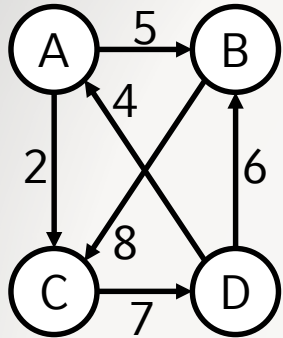
$$\begin{matrix} & k = A \\ \begin{pmatrix} 0 & 5 & 2 & \infty \\ \infty & 0 & 8 & \infty \\ \infty & \infty & 0 & 7 \\ 4 & 6 & \mathbf{6} & 0 \end{pmatrix} \end{matrix}$$

$$\begin{matrix} & k = B \\ \begin{pmatrix} 0 & 5 & 2 & \infty \\ \infty & 0 & 8 & \infty \\ \infty & \infty & 0 & 7 \\ 4 & 6 & 6 & 0 \end{pmatrix} \end{matrix}$$

$$\begin{matrix} & k = C \\ \begin{pmatrix} 0 & 5 & 2 & \mathbf{9} \\ \infty & 0 & 8 & \mathbf{15} \\ \infty & \infty & 0 & 7 \\ 4 & 6 & 6 & 0 \end{pmatrix} \end{matrix}$$

$$\begin{matrix} & k = D \\ \begin{pmatrix} 0 & 5 & 2 & 9 \\ \mathbf{19} & 0 & 8 & 15 \\ \mathbf{11} & \mathbf{13} & 0 & 7 \\ 4 & 6 & 6 & 0 \end{pmatrix} \end{matrix}$$

# Floyd-Algorithmus: Beispiel



Initialisierung:

$$\begin{pmatrix} 0 & 5 & 2 & \infty \\ \infty & 0 & 8 & \infty \\ \infty & \infty & 0 & 7 \\ 4 & 6 & \infty & 0 \end{pmatrix}$$

$P_0$ :

$$\begin{pmatrix} A & B & C & - \\ - & B & C & - \\ - & - & C & D \\ A & B & - & D \end{pmatrix}$$

$k = A$

$$\begin{pmatrix} 0 & 5 & 2 & \infty \\ \infty & 0 & 8 & \infty \\ \infty & \infty & 0 & 7 \\ 4 & 6 & \mathbf{6} & 0 \end{pmatrix}$$

$k = B$

$$\begin{pmatrix} 0 & 5 & 2 & \infty \\ \infty & 0 & 8 & \infty \\ \infty & \infty & 0 & 7 \\ 4 & 6 & 6 & 0 \end{pmatrix}$$

$k = C$

$$\begin{pmatrix} 0 & 5 & 2 & \mathbf{9} \\ \infty & 0 & 8 & \mathbf{15} \\ \infty & \infty & 0 & 7 \\ 4 & 6 & 6 & 0 \end{pmatrix}$$

$k = D$

$$\begin{pmatrix} 0 & 5 & 2 & 9 \\ \mathbf{19} & 0 & 8 & 15 \\ \mathbf{11} & \mathbf{13} & 0 & 7 \\ 4 & 6 & 6 & 0 \end{pmatrix}$$

$P_A$ :

$$\begin{pmatrix} A & B & C & - \\ - & B & C & - \\ - & - & C & D \\ A & B & \mathbf{A} & D \end{pmatrix}$$

$P_B$ :

$$\begin{pmatrix} A & B & C & - \\ - & B & C & - \\ - & - & C & D \\ A & B & A & D \end{pmatrix}$$

$P_C$ :

$$\begin{pmatrix} A & B & C & \mathbf{C} \\ - & B & C & \mathbf{C} \\ - & - & C & D \\ A & B & A & D \end{pmatrix}$$

$P_D$ :

$$\begin{pmatrix} A & B & C & C \\ \mathbf{D} & B & C & C \\ \mathbf{D} & \mathbf{D} & C & D \\ A & B & A & D \end{pmatrix}$$



# Floyd-Algorithmus: Komplexität

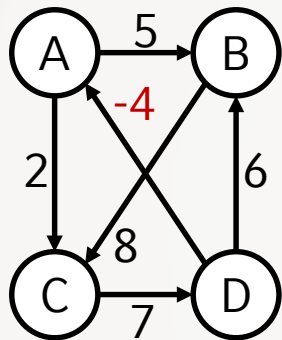
- 3 geschachtelte Schleifen mit  $i, j, k$  über  $V$
- Zeitkomplexität:  $O(|V|^3)$
- Platzkomplexität:  $O(|V|^2)$
- Warshall-Algorithmus
  - Aus demselben Jahr (1962) stammt ein Algorithmus von Warshall, der statt Kosten nur die Existenz von Verbindungen betrachtet (transitive Hülle).
  - Innere Schleife läuft auf booleschen Werten:  
falls  $\neg d[i][j]$   
$$d[i][j] = d[i][k] \wedge d[k][j]$$

# Floyd-Algorithmus: Negative Kanten

- Wir haben negative Kanten nicht ausgeschlossen:

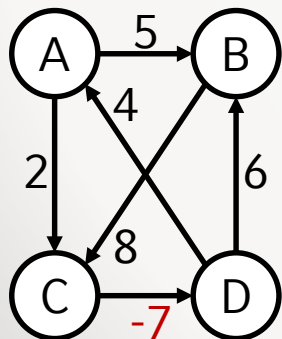
$$c[v, w] \in \mathbb{R} \cup \{\infty\}$$

- Was passiert mit einer negativen Kante?



$$d = \begin{pmatrix} 0 & 5 & 2 & 9 \\ \mathbf{11} & 0 & 8 & 15 \\ \mathbf{3} & \mathbf{8} & 0 & 7 \\ \mathbf{-4} & \mathbf{1} & \mathbf{-2} & 0 \end{pmatrix}$$

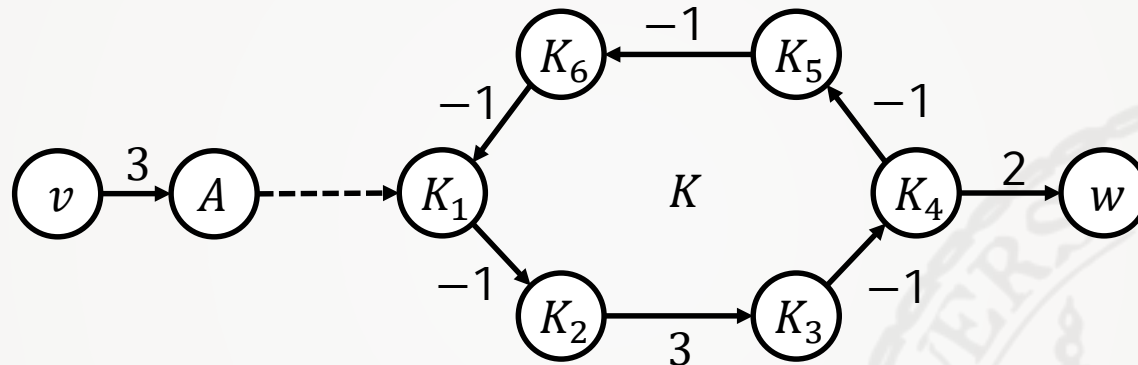
- Scheint ok. Noch ein Beispiel:



$$d = \begin{pmatrix} \mathbf{-1} & \mathbf{1} & \mathbf{1} & \mathbf{-6} \\ \mathbf{5} & 0 & \mathbf{7} & \mathbf{0} \\ \mathbf{-3} & \mathbf{-1} & \mathbf{-1} & \mathbf{-8} \\ \mathbf{3} & \mathbf{5} & \mathbf{5} & \mathbf{-2} \end{pmatrix}$$

# Negative Kreise

- Einzelne negative Kanten  
→ kein Problem, kürzeste Wege bleiben meist wohldefiniert.
- Negative Kreise



- Falls es einen Pfad von  $v$  nach  $K$  und einen Pfad von  $K$  nach  $w$  gibt mit  $K < 0$ , dann ist der kürzeste Pfad von  $v$  nach  $w$  nicht definiert.
- Für  $K \geq 0$  gibt es keine Probleme. Der kürzeste Pfad ist wohldefiniert.

# Informierte Suche

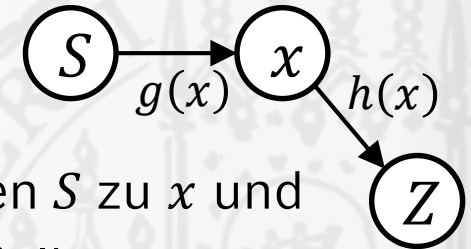
- Dijkstra-Algorithmus:
  - Greedy: Füge Kante sofort hinzu, falls sie geringere Kosten verspricht
  - Kosten zu allen potentiellen Zielen (= Knoten) werden bestimmt
- Verbesserung: Falls das Ziel bekannt ist, können Kanten gezielt ausgewählt werden
- Eine Heuristik ist eine Strategie, um das Auffinden von Lösungen zu beschleunigen, indem zusätzliches Wissen genutzt wird.
- Viele Heuristiken orientieren sich an menschlichen Problemlösungen.
- Heuristik gibt in der Regel eine Schätzung von Kosten an.

## A\*-Algorithmus: Idee

- Besuche zuerst Knoten, die wahrscheinlich schnell zum Ziel führen.
- Jeder besuchte Knoten erhält einen Wert  $f(x)$ , der angibt, wie lange der Pfad vom Start zum Ziel über den Knoten  $x$  im günstigsten Fall ist.
- Der Knoten mit dem niedrigsten  $f$ -Wert wird als nächstes untersucht:

$$f(x) = g(x) + h(x)$$

- Dabei gibt
  - $g(x)$  die tatsächlichen Kosten vom Startknoten  $S$  zu  $x$  und
  - $h(x)$  die geschätzten Kosten von  $x$  bis zum Zielknoten an.



- Die verwendete Heuristik  $h(x)$  darf die Kosten für keinen Knoten  $x$  überschätzen, da sonst die optimale Lösung vielleicht nicht gefunden wird.

## A\*-Algorithmus: Datenstrukturen

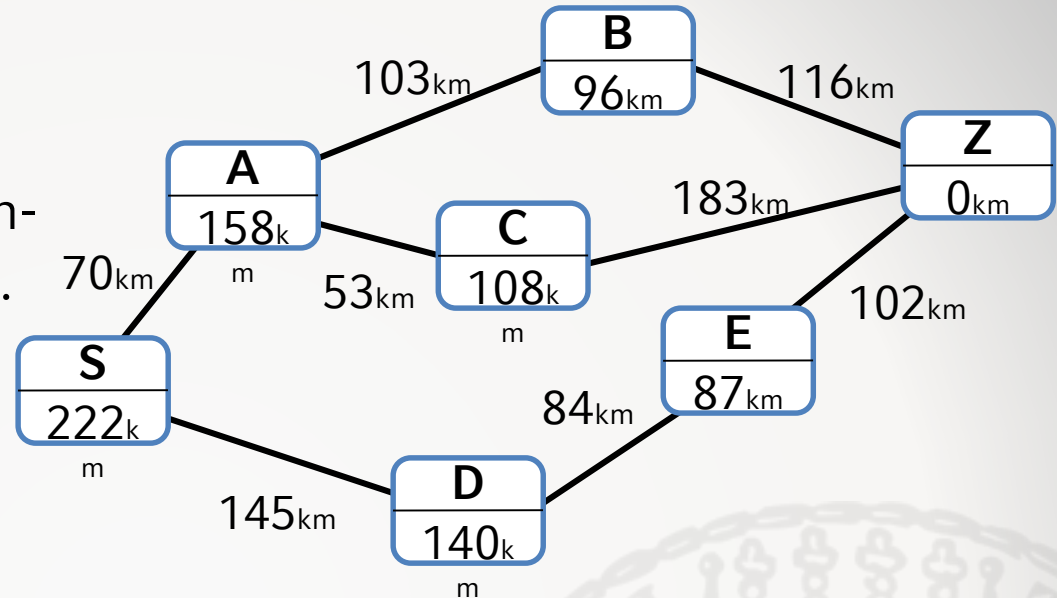
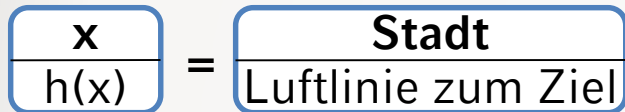
- Jeder Knoten des Graphen kann einer der folgenden Zustände zugeordnet werden. Die Knoten werden dementsprechend in Listen verwaltet:
  - Der Knoten wurde noch nicht verarbeitet und wir kennen noch keinen Weg dorthin.
  - Ein Weg zum Knoten ist bekannt, aber es könnte einen kürzeren Weg geben → **OpenList**
  - Der kürzeste Weg zum Knoten wurde gefunden → **ClosedList**



## A\*-Algorithmus: Beispiel

**Idee:** Darstellung des Straßennetzes als gewichteter Graph.

**Knoten:**

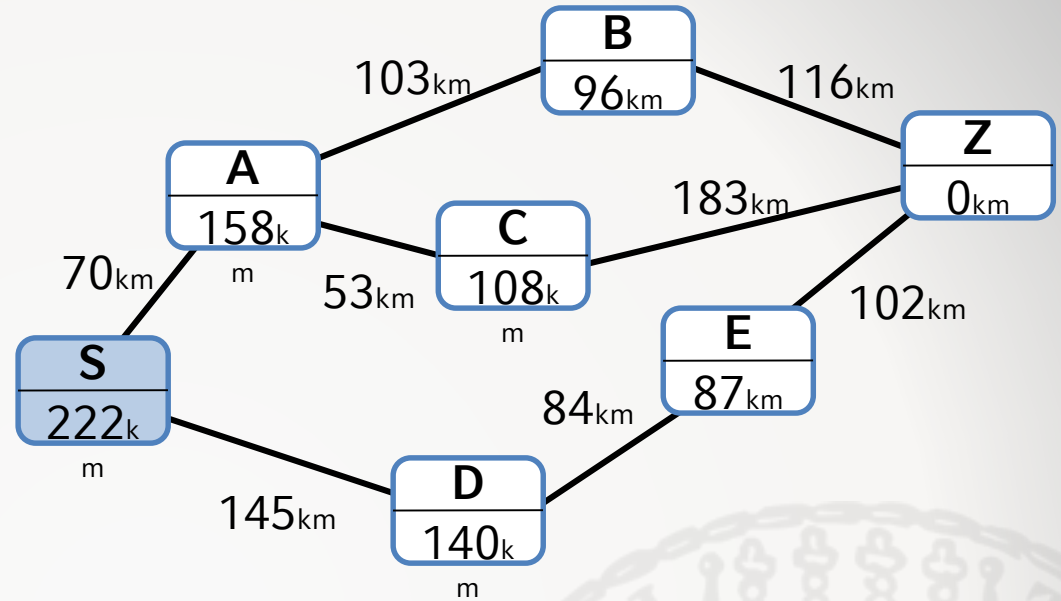


- Jeder Knoten steht für eine Stadt.
- Zwei Knoten sind verbunden wenn es eine *direkte* Straßenverbindung zwischen den entsprechenden Städten gibt.
- Die Kosten der Kanten entsprechen der Länge der Straße zwischen den Städten.
- Gesucht ist **der kürzeste Weg** von Stadt S nach Stadt Z.
- Als Heuristik  $h(x)$  nutzen wir die **Luftlinie** zwischen der Stadt  $x$  und der Zielstadt Z.



# A\*-Algorithmus: Beispiel

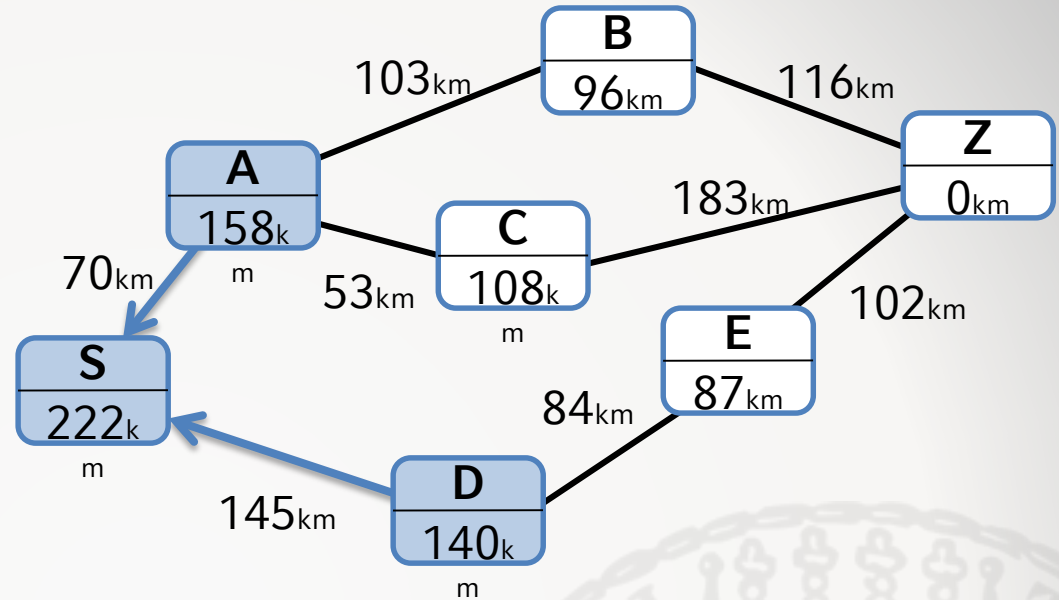
→ = Zeiger auf den Vorgänger



Schritt	OpenList (Stadt, f)	ClosedList (Stadt, Entfernung von S)
0	(S, 0)	---

# A\*-Algorithmus: Beispiel

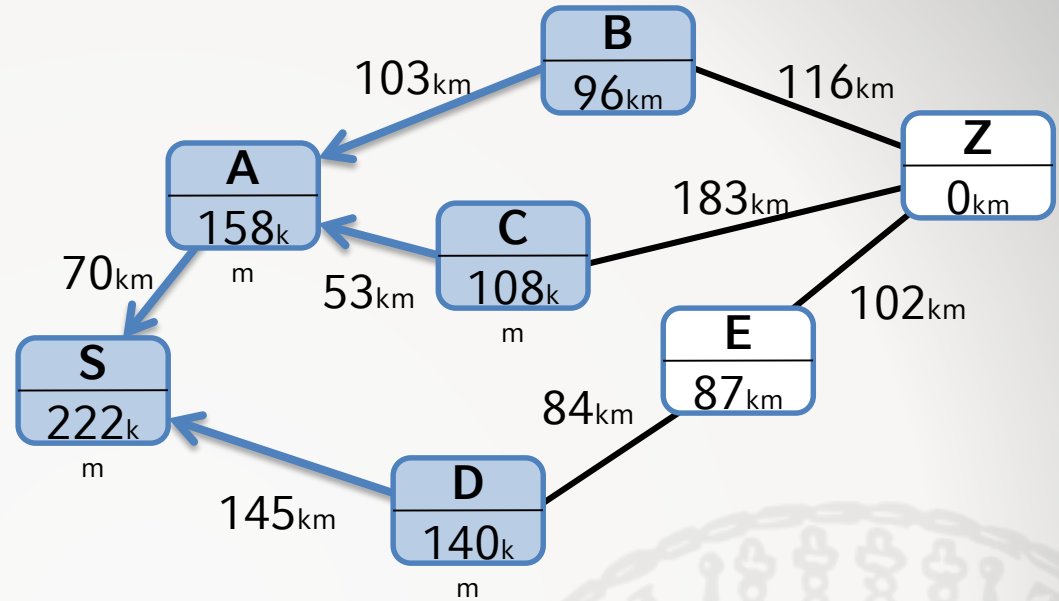
→ = Zeiger auf den Vorgänger



Schritt	OpenList (Stadt, f)	ClosedList (Stadt, Entfernung von S)
0	(S, 0)	---
1	(A, 228), (D, 285)	(S, 0)

# A\*-Algorithmus: Beispiel

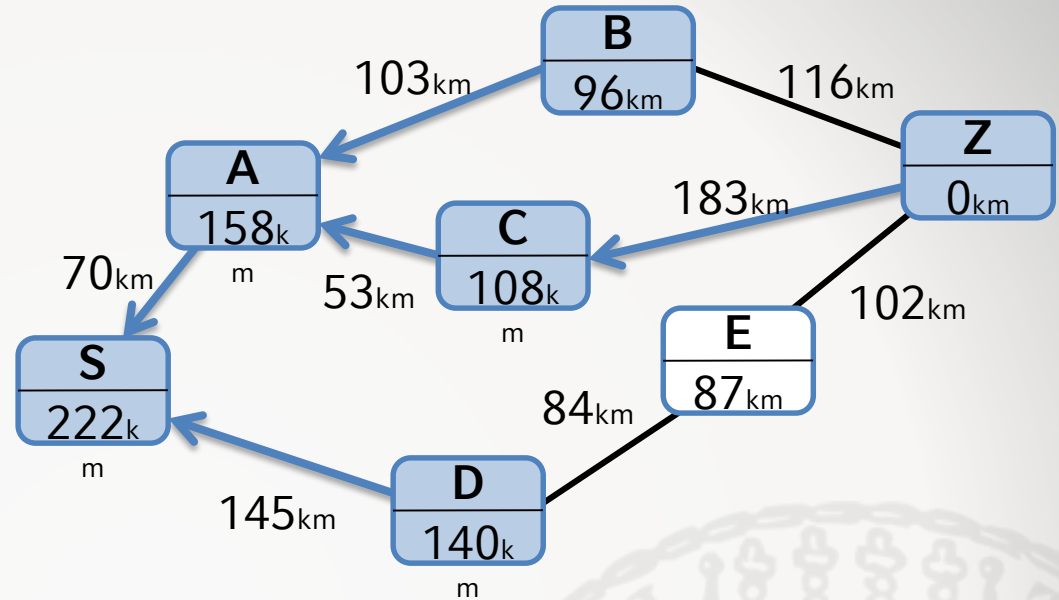
→ = Zeiger auf den Vorgänger



Schritt	OpenList (Stadt, f)	ClosedList (Stadt, Entfernung von S)
0	(S, 0)	---
1	(A, 228), (D, 285)	(S, 0)
2	(D, 285), (B, 269), (C, 231)	(S, 0), (A, 70)

# A\*-Algorithmus: Beispiel

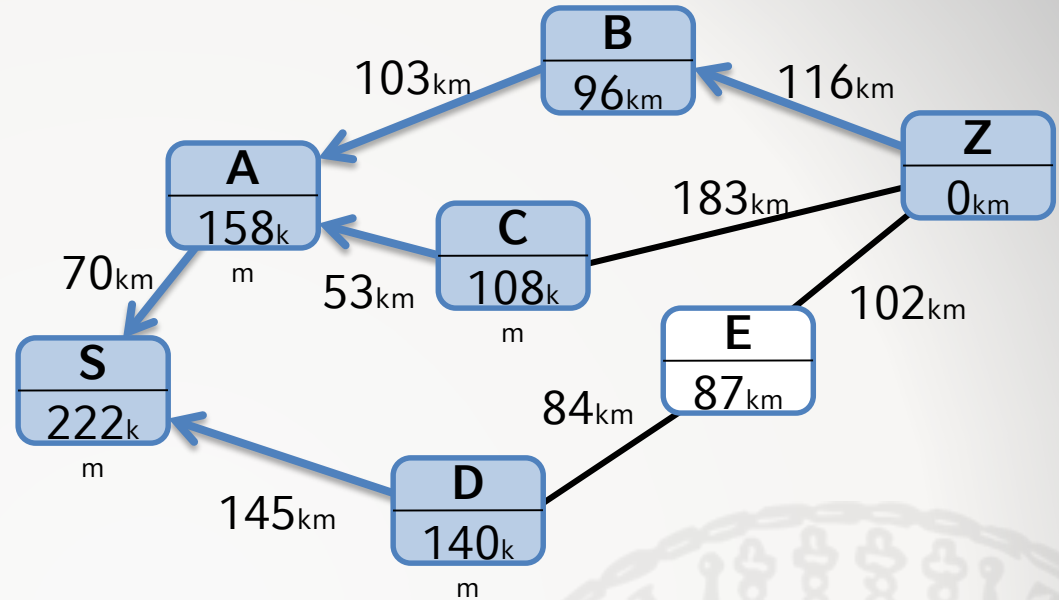
→ = Zeiger auf den Vorgänger



Schritt	OpenList (Stadt, f)	ClosedList (Stadt, Entfernung von S)
0	(S, 0)	---
1	(A, 228), (D, 285)	(S, 0)
2	(D, 285), (B, 269), (C, 231)	(S, 0), (A, 70)
3	(D, 285), (B, 269), (Z, 306)	(S, 0), (A, 70), (C, 123)

# A\*-Algorithmus: Beispiel

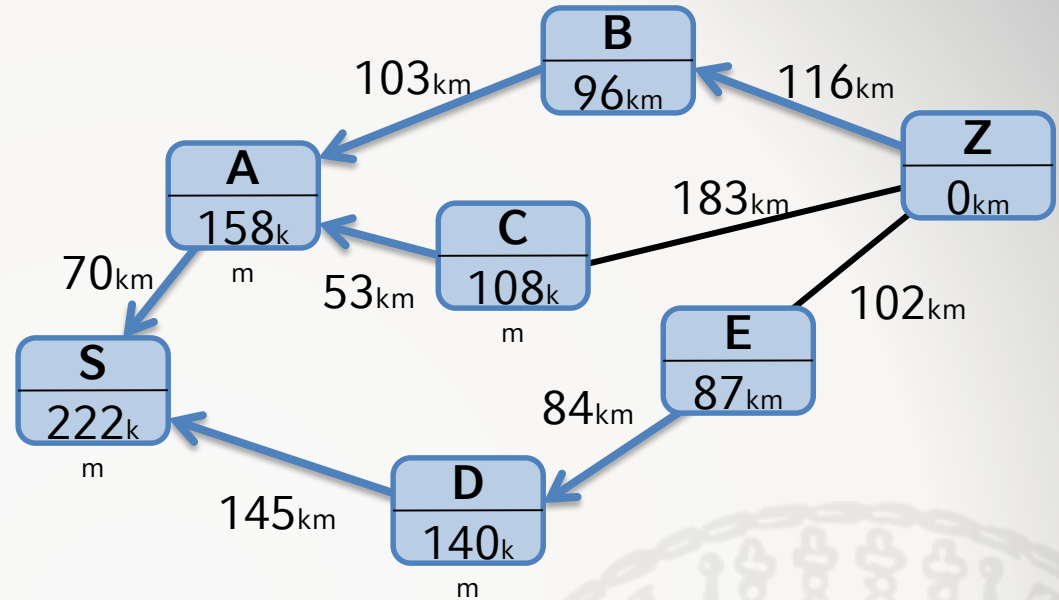
→ = Zeiger auf den Vorgänger



Schritt	OpenList (Stadt, f)	ClosedList (Stadt, Entfernung von S)
0	(S, 0)	---
1	(A, 228), (D, 285)	(S, 0)
2	(D, 285), (B, 269), (C, 231)	(S, 0), (A, 70)
3	(D, 285), (B, 269), (Z, 306)	(S, 0), (A, 70), (C, 123)
4	(D, 285), (Z, 289)	(S, 0), (A, 70), (C, 123), (B, 173)

# A\*-Algorithmus: Beispiel

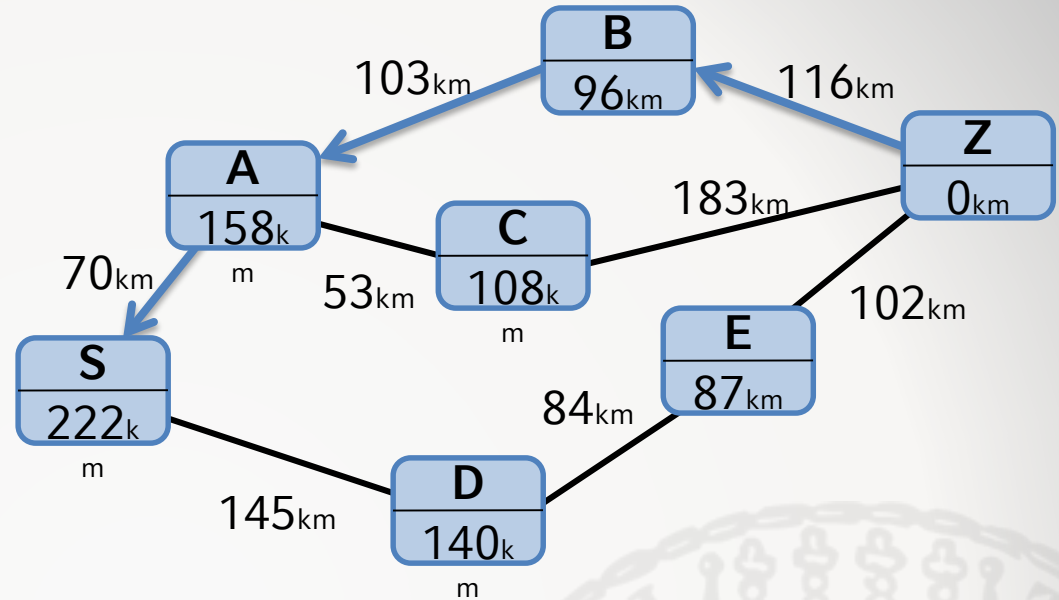
→ = Zeiger auf den Vorgänger



Schritt	OpenList (Stadt, f)	ClosedList (Stadt, Entfernung von S)
0	(S, 0)	---
1	(A, 228), (D, 285)	(S, 0)
2	(D, 285), (B, 269), (C, 231)	(S, 0), (A, 70)
3	(D, 285), (B, 269), (Z, 306)	(S, 0), (A, 70), (C, 123)
4	(D, 285), (Z, 289)	(S, 0), (A, 70), (C, 123), (B, 173)
5	(Z, 289), (E, 316)	(S, 0), (A, 70), (C, 123), (B, 173), (D, 145)

# A\*-Algorithmus: Beispiel

→ = Zeiger auf den Vorgänger



Schritt	OpenList (Stadt, f)	ClosedList (Stadt, Entfernung von S)
0	(S, 0)	---
1	(A, 228), (D, 285)	(S, 0)
2	(D, 285), (B, 269), (C, 231)	(S, 0), (A, 70)
3	(D, 285), (B, 269), (Z, 306)	(S, 0), (A, 70), (C, 123)
4	(D, 285), (Z, 289)	(S, 0), (A, 70), (C, 123), (B, 173)
5	(Z, 289), (E, 316)	(S, 0), (A, 70), (C, 123), (B, 173), (D, 145)
6	(E, 316)	<b>Pfad gefunden: S → A → B → Z</b>



## A\*-Algorithmus: Qualitätseigenschaften

- **Vollständig**

Wenn es eine Lösung gibt, so wird diese auch gefunden.

- **Optimal**

Es wird immer eine optimale Lösung gefunden.

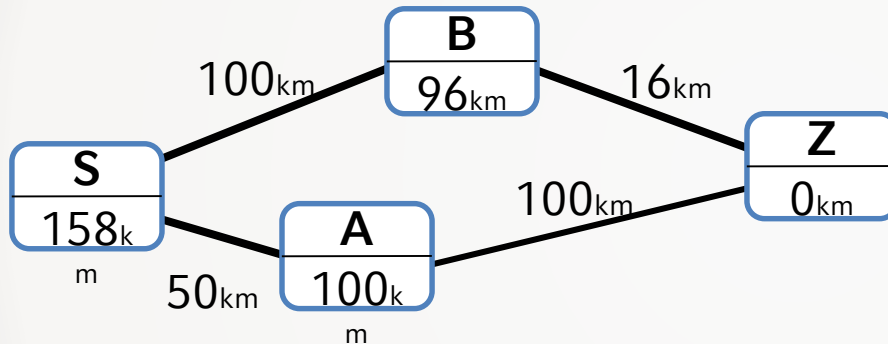
- **Optimal effizient**

Bezogen auf die Laufzeit gibt es keinen Algorithmus, der die gleiche Heuristik verwendet und weniger Knoten besucht.

## A\*-Algorithmus: Eigenschaft der Heuristik

- Die verwendete Heuristik für  $h(x)$  darf die Kosten für keinen Knoten  $x$  überschätzen.

Werden die Kosten überschätzt, so ist die **Optimalität** des Algorithmus nicht mehr gewährleistet:



OpenList	ClosedList
(S, 0)	---
(A, 150), (B, 196)	(S, 0)
(B, 196), (Z, 150)	(S, 0), (A, 150)

- Für die *schlechteste* Heuristik  $h(x) = 0$  gilt:
  - die geschätzten Kosten für jeden Knoten entsprechen genau den Kosten, um diesen Knoten zu erreichen.
  - Der A\*-Algorithmus bildet den Dijkstra-Algorithmus nach.

# Minimaler Spannbaum

Gegeben:

Ungerichteter zusammenhängender Graph  $G = (V, E)$   
mit Kantengewichten  $c: E \rightarrow \mathbb{R}$ .

Gesucht:

Ungerichteter Subgraph  $G' = (V, E')$  mit

- $E' \subseteq E$ .
- $G'$  ist zyklfrei und zusammenhängend.  
(Äquivalent:  $G'$  ist ein Baum).
- Für alle Subgraphen  $G'' = (V, E'')$  gilt

$$\sum_{e \in E'} c(e) \leq \sum_{e \in E''} c(e)$$



# Prim-Algorithmus

- Idee:
  - Beginne mit einem beliebigen Knoten des Graphen.
  - Finde sukzessive die minimale Kante, die den Subgraphen mit einem noch nicht gewählten Knoten verbindet.
  - In jedem Schritt wird der aktuelle Subgraph um jeweils diese Kante und den inzidenten Knoten erweitert.

$V' \leftarrow v_0$  beliebig

$E' \leftarrow \emptyset$

Solange  $V \neq V'$ :

$(u, v) = \underset{e \in V' \times (V - V')}{\operatorname{argmin}} c(e)$

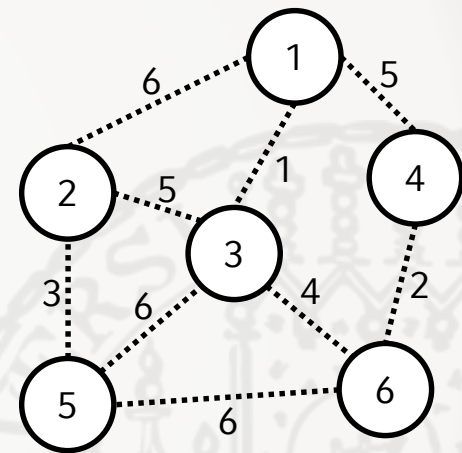
$E' = E' \cup \{(u, v)\}$

$V' = V' \cup v$

- Komplexität ist  $O(|V|^2)$ , denn für jeden neu einzufügenden Knoten werden die Kanten zu anderen Knoten überprüft.

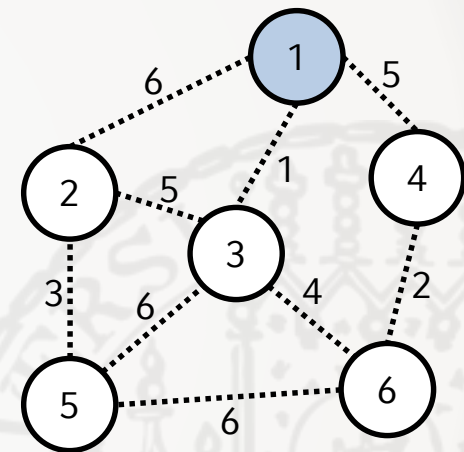
# Prim-Algorithmus: Beispiel

$V'$	$E'$



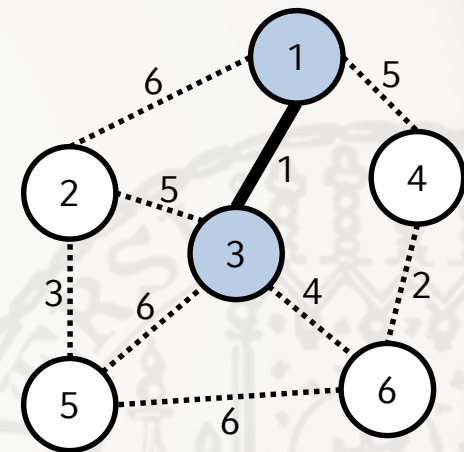
# Prim-Algorithmus: Beispiel

$V'$	$E'$
{1}	$\emptyset$



# Prim-Algorithmus: Beispiel

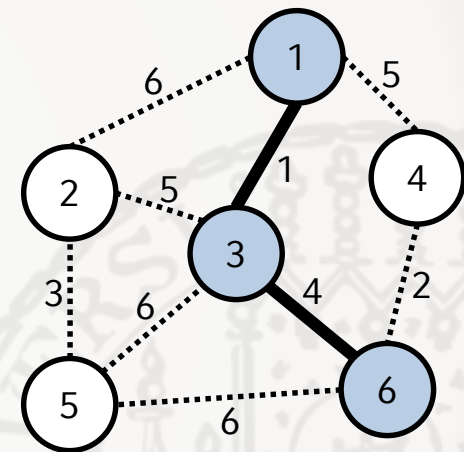
$V'$	$E'$
{1}	$\emptyset$
{1,3}	{{(1,3)}





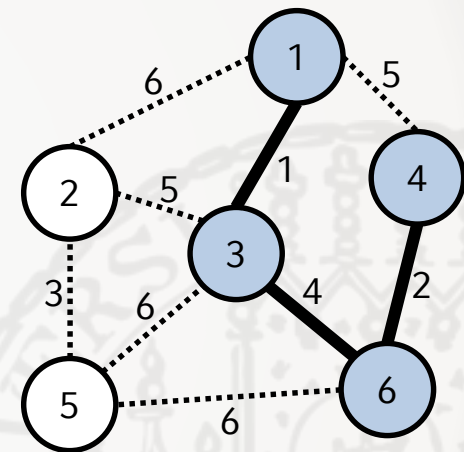
# Prim-Algorithmus: Beispiel

$V'$	$E'$
{1}	$\emptyset$
{1,3}	{(1,3)}
{1,3,6}	{(1,3), (3,6)}



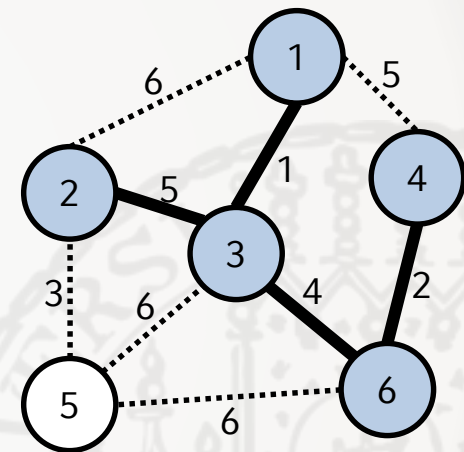
# Prim-Algorithmus: Beispiel

$V'$	$E'$
{1}	$\emptyset$
{1,3}	{(1,3)}
{1,3,6}	{(1,3), (3,6)}
{1,3,6,4}	{(1,3), (3,6), (6,4)}



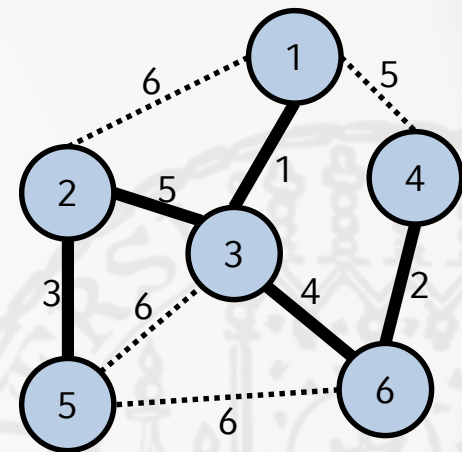
# Prim-Algorithmus: Beispiel

$V'$	$E'$
{1}	$\emptyset$
{1,3}	{(1,3)}
{1,3,6}	{(1,3), (3,6)}
{1,3,6,4}	{(1,3), (3,6), (6,4)}
{1,3,6,4,2}	{(1,3), (3,6), (6,4), (3,2)}



# Prim-Algorithmus: Beispiel

$V'$	$E'$
{1}	$\emptyset$
{1,3}	{(1,3)}
{1,3,6}	{(1,3), (3,6)}
{1,3,6,4}	{(1,3), (3,6), (6,4)}
{1,3,6,4,2}	{(1,3), (3,6), (6,4), (3,2)}
{1,3,6,4,2,5}	{(1,3), (3,6), (6,4), (3,2), (2,5)}



# Algorithmus von Kruskal

- Idee (ähnlich zu Prim):
  - Starte mit leerer Kantenmenge.
  - Füge sukzessive minimale Kanten bezüglich ihrer Kosten hinzu, sodass kein Kreis entsteht.
  - Stoppe, falls keine solche Kante mehr gefunden werden kann (die nächste Kante bildet einen Kreis, alle Knoten erreichbar).

$E' \leftarrow \emptyset$

Sortiere  $E$  aufsteigend nach  $c(E)$ .

Solange  $E \neq \emptyset$

$e \leftarrow \min(E)$

$E = E - \{e\}$

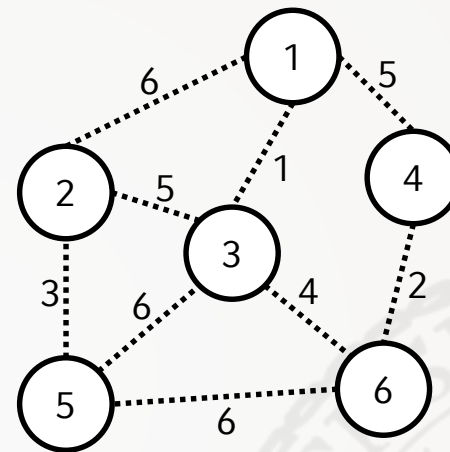
Falls  $G(V, E' \cup \{e\})$  kreisfrei

$E' = E' \cup \{e\}$

- Das Sortieren dominiert hier die Laufzeit:  $O(|E| \log|E|)$ .

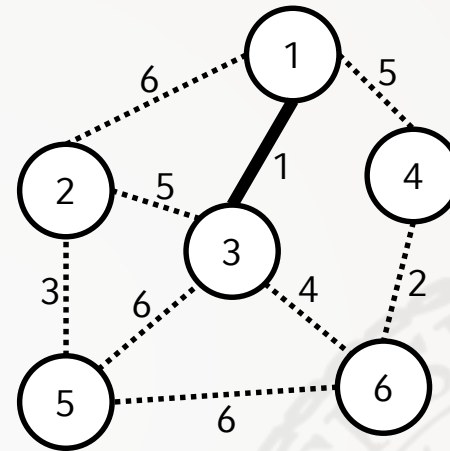
# Algorithmus von Kruskal: Beispiel

<i>E</i>
(1,3) → 1
(4,6) → 2
(2,5) → 3
(3,6) → 4
(1,4) → 5
(2,3) → 5
(1,2) → 6
(3,5) → 6
(5,6) → 6



# Algorithmus von Kruskal: Beispiel

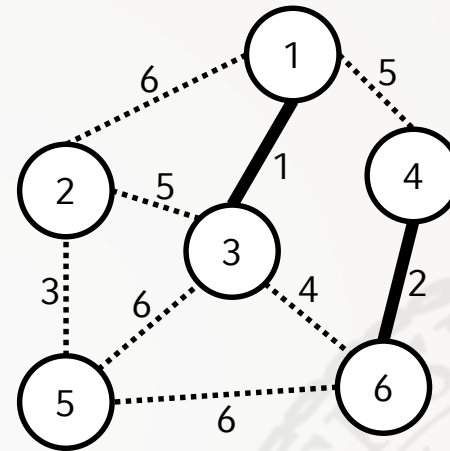
<i>E</i>
(1,3) → 1
(4,6) → 2
(2,5) → 3
(3,6) → 4
(1,4) → 5
(2,3) → 5
(1,2) → 6
(3,5) → 6
(5,6) → 6





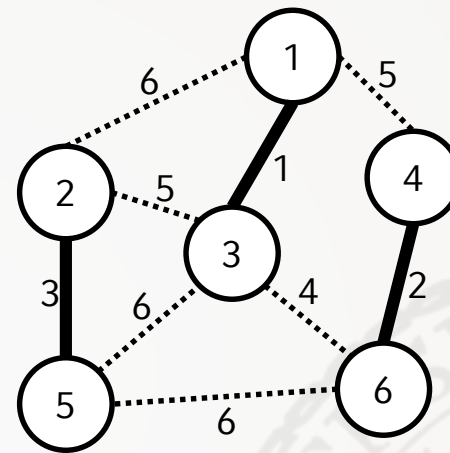
# Algorithmus von Kruskal: Beispiel

<i>E</i>
(1,3) → 1
(4,6) → 2
(2,5) → 3
(3,6) → 4
(1,4) → 5
(2,3) → 5
(1,2) → 6
(3,5) → 6
(5,6) → 6



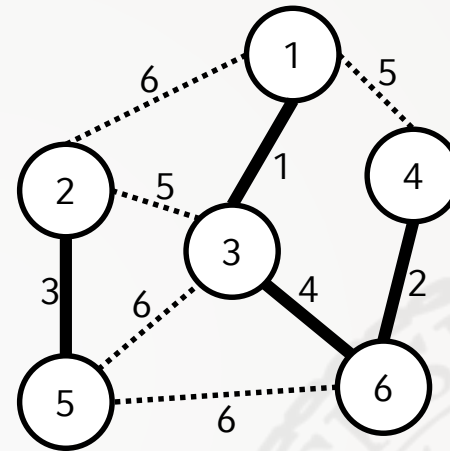
# Algorithmus von Kruskal: Beispiel

<i>E</i>
(1,3) → 1
(4,6) → 2
<b>(2,5) → 3</b>
(3,6) → 4
(1,4) → 5
(2,3) → 5
(1,2) → 6
(3,5) → 6
(5,6) → 6



# Algorithmus von Kruskal: Beispiel

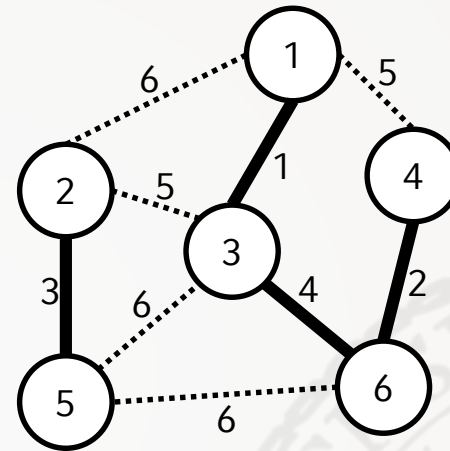
<i>E</i>
(1,3) → 1
(4,6) → 2
(2,5) → 3
<b>(3,6) → 4</b>
(1,4) → 5
(2,3) → 5
(1,2) → 6
(3,5) → 6
(5,6) → 6



# Algorithmus von Kruskal: Beispiel

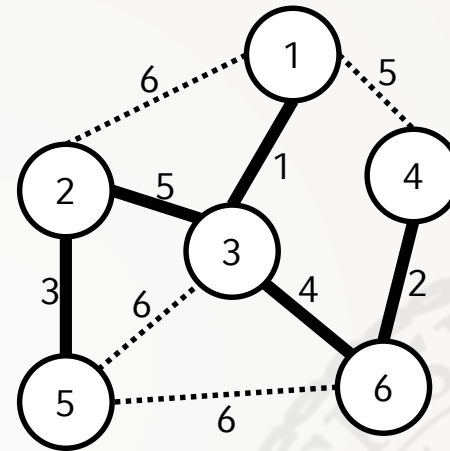
<i>E</i>
(1,3) → 1
(4,6) → 2
(2,5) → 3
(3,6) → 4
<b>(1,4) → 5</b>
(2,3) → 5
(1,2) → 6
(3,5) → 6
(5,6) → 6

→ Kreis



# Algorithmus von Kruskal: Beispiel

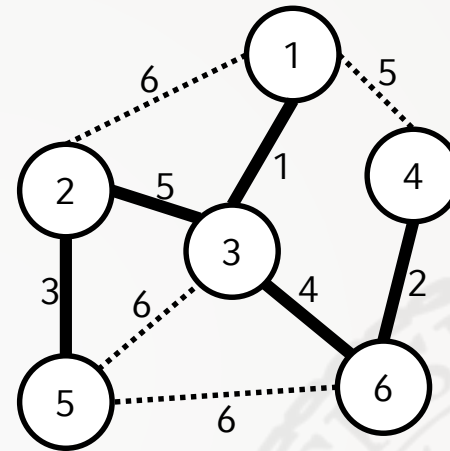
<i>E</i>
(1,3) → 1
(4,6) → 2
(2,5) → 3
(3,6) → 4
(1,4) → 5
<b>(2,3) → 5</b>
(1,2) → 6
(3,5) → 6
(5,6) → 6



# Algorithmus von Kruskal: Beispiel

<i>E</i>
(1,3) → 1
(4,6) → 2
(2,5) → 3
(3,6) → 4
(1,4) → 5
(2,3) → 5
<b>(1,2) → 6</b>
(3,5) → 6
(5,6) → 6

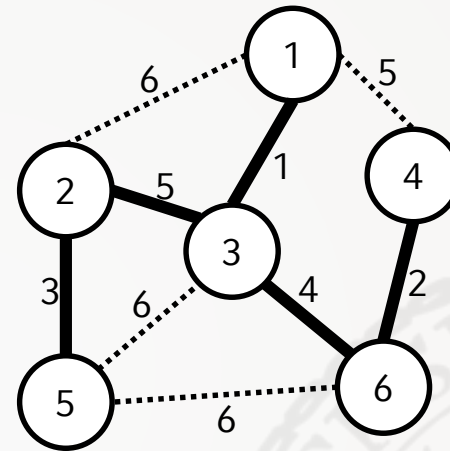
→ Kreis



# Algorithmus von Kruskal: Beispiel

<i>E</i>
(1,3) → 1
(4,6) → 2
(2,5) → 3
(3,6) → 4
(1,4) → 5
(2,3) → 5
(1,2) → 6
<b>(3,5) → 6</b>
(5,6) → 6

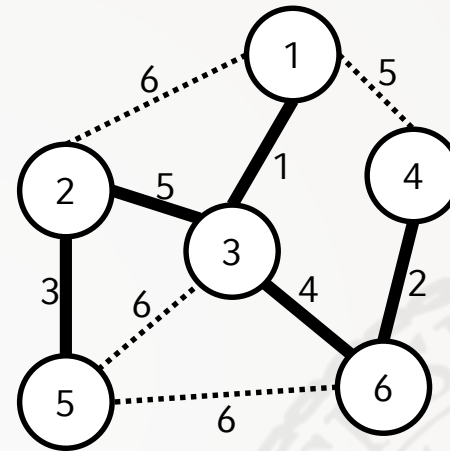
→ Kreis





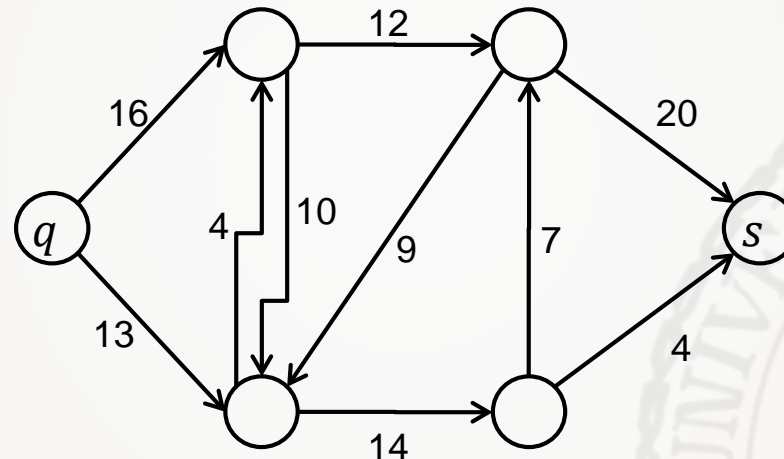
# Algorithmus von Kruskal: Beispiel

<i>E</i>
(1,3) → 1
(4,6) → 2
(2,5) → 3
(3,6) → 4
(1,4) → 5
(2,3) → 5
(1,2) → 6
(3,5) → 6
<b>(5,6) → 6</b> → Kreis



# Flussnetzwerke

- Ein Flussnetzwerk  $(G, c)$  ist ein gerichteter Graph  $G = (V, E)$ , wobei
  - jede Kante  $(u, v) \in E$  die Kapazität  $c(u, v) \geq 0$  hat
  - und es eine Quelle  $q \in V$  und eine Senke  $s \in V$  gibt.
- Wir setzen  $c(u, v) = 0$ , falls  $(u, v) \notin E$

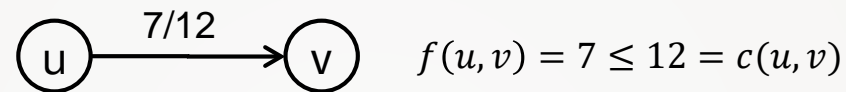


- anschaulich:
  - Wasserleitungen mit unterschiedlichen Kapazitäten
  - Verkehrsströme, Straßenkapazitäten
  - Kommunikationswege mit Bandbreiten

# Fluss in einem Flussnetzwerk

- Ein Fluss ist eine Funktion  $f: V \times V \rightarrow \mathbb{R}$  mit den Eigenschaften

- Kapazitätsbeschränkung: Für  $u, v \in V$  gilt  $f(u, v) \leq c(u, v)$



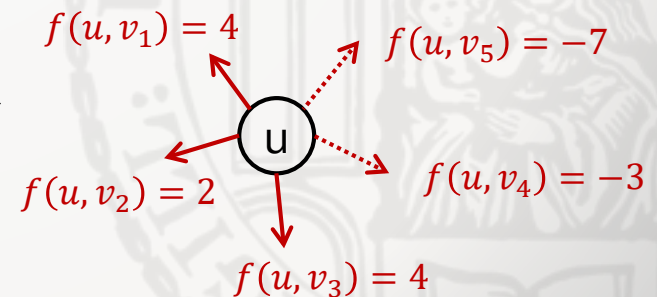
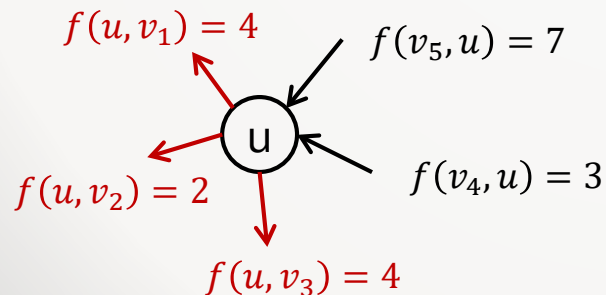
- Symmetrie: Für  $u, v \in V$  gilt  $f(u, v) = -f(v, u)$

„ $u$  gibt  $v$  7 Einheiten“

→ „ $v$  nimmt  $u$  7 Einheiten“

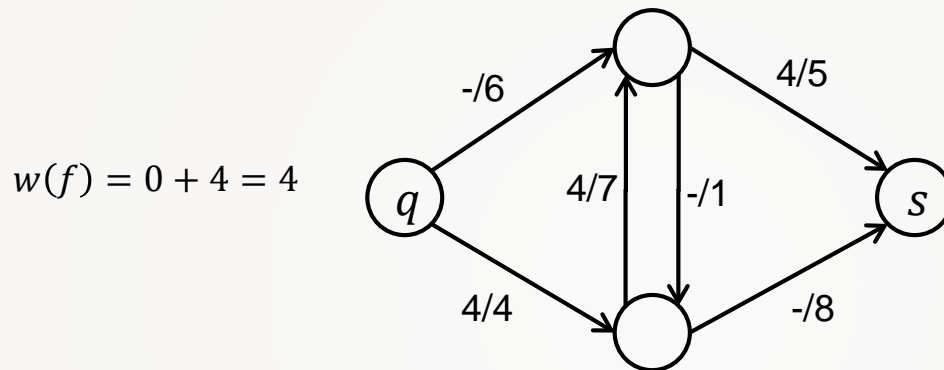
→ „ $v$  gibt  $u$   $-7$  Einheiten“

- Flusserhaltung: Für  $u \in V - \{q, s\}$  gilt  $\sum_{v \in V} f(u, v) = 0$



## Maximaler Fluss

- Der Wert  $w(f)$  eines Flusses  $f$  ist definiert als  $w(f) = \sum_{v \in V} f(q, v)$ 
  - entspricht Gesamtfluss aus der Quelle  $q$  heraus

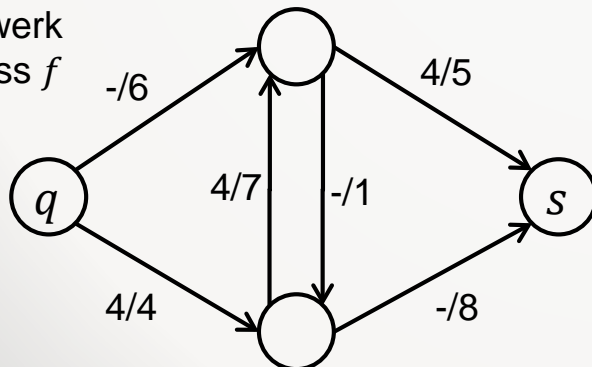


- Problem des maximalen Flusses
  - Gegeben ein Flussnetzwerk  $(G, c)$
  - Gesucht ein Fluss  $f$  auf  $(G, c)$  mit maximalem Wert  $w(f)$

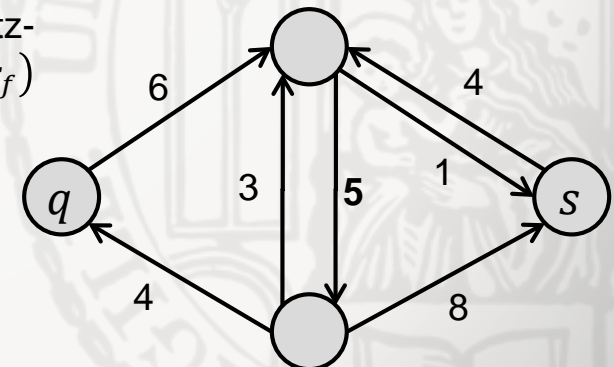
# Residualnetzwerk

- Restkapazität  $c_f(u, v)$  zwischen  $u, v \in V$  ist  $c_f(u, v) = c(u, v) - f(u, v)$ 
  - beachte: formal ist  $f(u, v) < 0$  möglich (vgl. Symmetrie)
- Der Restgraph  $G_f = (V, E_f)$  bzgl. Flussnetzwerk  $(G, c)$  und Fluss  $f$  ist definiert durch die Kantenmenge  $E_f = \{(u, v) \in V \times V \mid c_f(u, v) > 0\}$
- $(G_f, c_f)$  ist das sogenannte Residualnetzwerk
  - „Flussnetzwerk minus Fluss = Residualnetzwerk“

Flussnetzwerk  
( $G, c$ ) mit Fluss  $f$

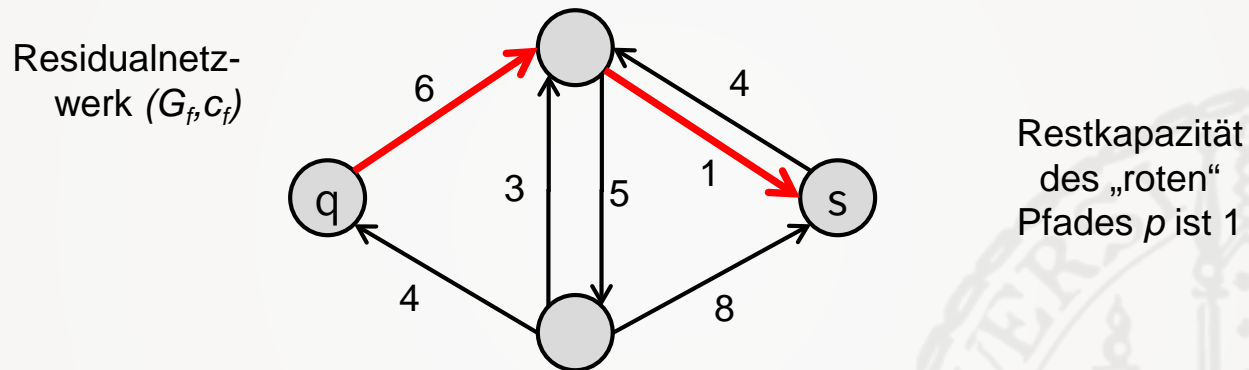


Residualnetzwerk  
( $G_f, c_f$ )

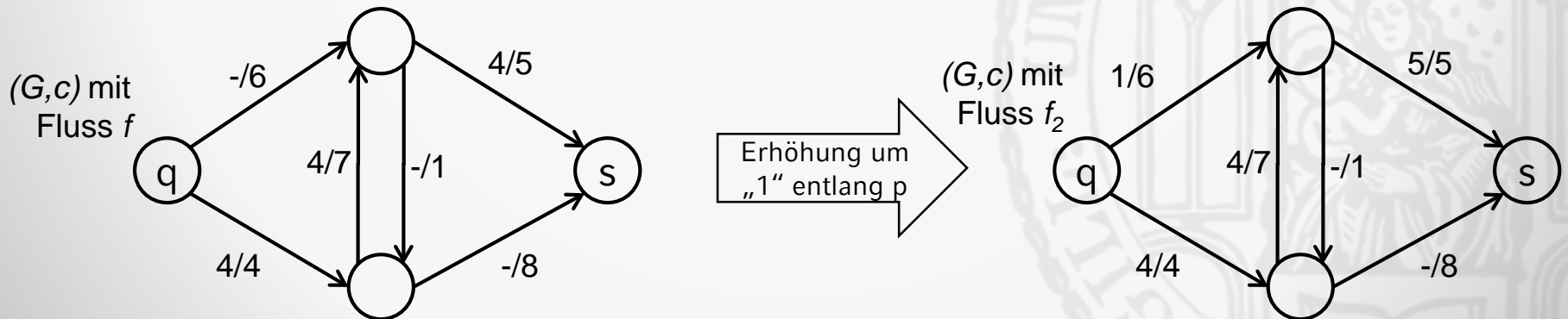


# Flussvergrößernder Pfad

- Ein Pfad  $p$  von  $q$  nach  $s$  im Residualnetzwerk heißt flussvergrößernder oder augmentierender Pfad.
- Die Restkapazität von  $p$  ist  $c_f(p) = \min\{c_f(u, v) \mid (u, v) \in p\}$



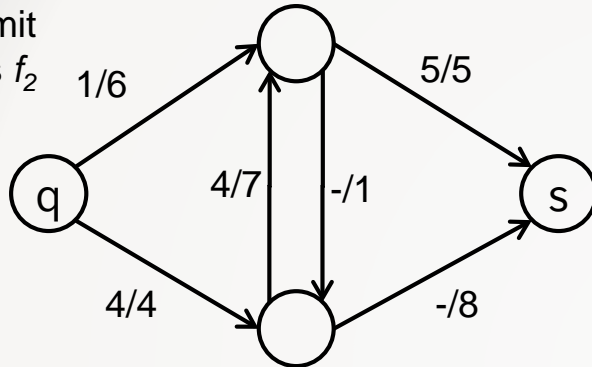
- Fluss  $f$  in  $(G, c)$  kann entlang des Pfades  $p$  um  $c_f(p)$  erhöht werden



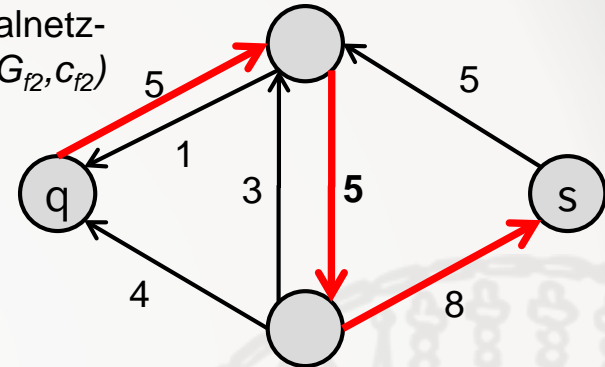


# Beispiel (fortgesetzt)

$(G, c)$  mit Fluss  $f_2$

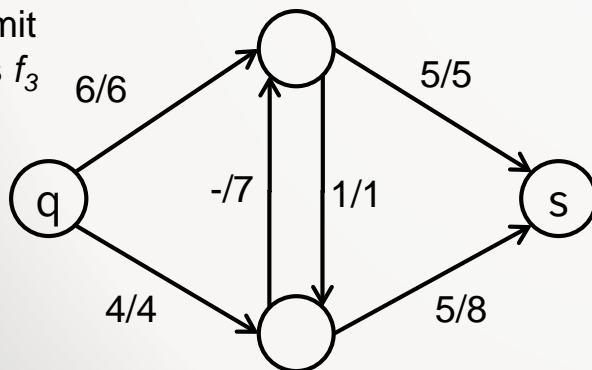


Residualnetzwerk  $(G_{f_2}, c_{f_2})$

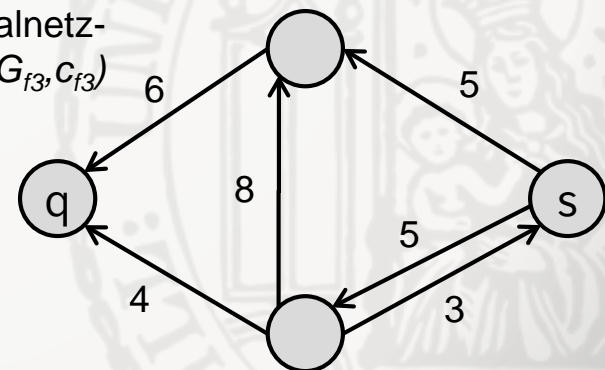


Achtung: „Rückwärtskante“ entlang  $p$

$(G, c)$  mit Fluss  $f_3$



Residualnetzwerk  $(G_{f_3}, c_{f_3})$



kein Pfad von  $q$  nach  $s$  möglich  
 $\rightarrow$  maximalen Fluss gefunden



## Min-Cut-Max-Flow-Theorem

- Ein s-t-Schnitt  $(S, T)$  in einem Flussnetzwerk ist eine Partition der Knotenmenge in zwei disjunkte Mengen  $s \in S$  und  $t \in T$ .
- Die Kapazität eines Schnitts ist das Gesamtgewicht der Kanten von  $S$  nach  $T$ :

$$c(S, T) = \sum_{u \in S, v \in T | (u, v) \in E} c(u, v)$$

- Die folgenden Aussagen sind äquivalent:
  - $f$  ist der maximale Fluss in  $G$ .
  - Das Residualnetzwerk  $G_f$  enthält keinen augmentierenden Pfad.
  - Für mindestens einen Schnitt  $(S, T)$  ist der Wert des Flusses gleich der Kapazität des Schnittes:  $|f| = c(S, T)$
- Damit gilt: Der maximale Fluss entspricht der Kapazität des minimalen Schnittes.

# Ford-Fulkerson-Methode

- Idee:

Initialisiere Fluss  $f$  mit 0

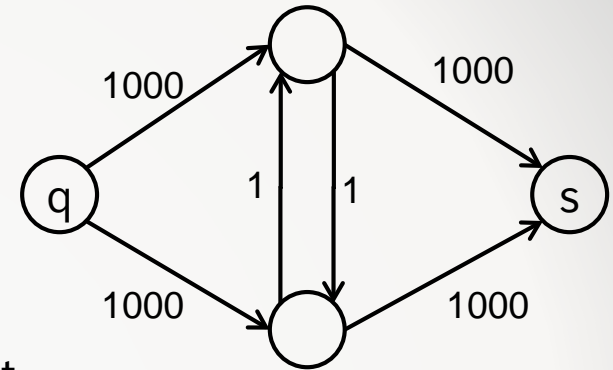
Solange es einen flussvergrößernden Pfad  $p$  gibt  
erhöhe  $f$  entlang  $p$

- Formal ist Erhöhung von  $f$  entlang  $p$  für alle  $u, v \in V$  definiert durch

$$f_{neu}(u, v) = f_{alt}(u, v) + \begin{cases} c_f(p) & , \text{ falls } (u, v) \text{ auf } p \\ -c_f(p) & , \text{ falls } (v, u) \text{ auf } p \\ 0 & , \text{ sonst} \end{cases}$$

# Laufzeit der Ford-Fulkerson-Methode

- Die Laufzeit kann beliebig schlecht sein
  - im Beispiel: wähle flussvergrößernden Pfad stets über mittlere Kanten
  - sehr langsame Erhöhung des Gesamtflusses
- Falls alle Kapazitäten ganzzahlig sind, benötigt die Methode  $O(f^*)$  Iterationen, um das Problem zu lösen (dabei ist  $f^*$  der Wert des maximalen Flusses)
  - in jeder Iteration wird der Wert des Flusses um  $c_f(p) \geq 1$  erhöht
  - zu Beginn 0 und am Ende  $f^*$
- Verbesserung:
  - wähle einen kürzesten flussvergrößernden Pfad
    - im Beispiel würden mittlere Kanten vermieden → schnellere Terminierung
  - Algorithmus von Edmonds und Karp



# Edmonds-Karp-Algorithmus

- Idee:

Initialisiere Fluss  $f$  mit 0

Solange es einen flussvergrößernden Pfad  $p$  gibt

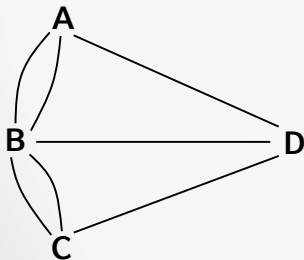
finde einen kürzesten flussvergrößernden Pfad  $p$

erhöhe  $f$  entlang  $p$

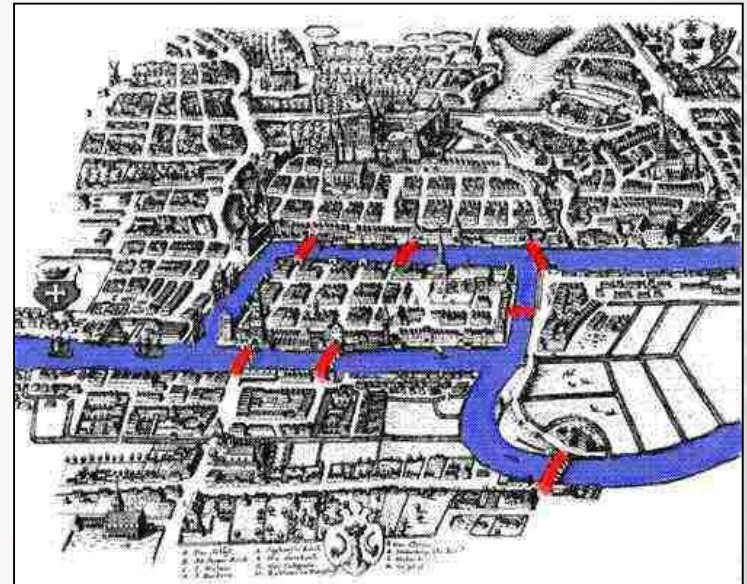
- Laufzeit der Methode ist polynomiell in der Größe des Netzwerks
  - $O(|V| \cdot |E|^2) = O(|V|^5)$  bei spezieller Implementierung
- Weitere Verbesserungen durch Dinic (1970) führten zu
  - $O(|V|^2 \cdot |E|) = O(|V|^4)$

# Graph-Anwendungen: Euler-Tour

- Historisches Problem („Königsberger Brückenproblem“):
  - 1736 lebte der deutsche Mathematiker Leonhard Euler in Königsberg
    - Fluß Pregel bildete dort eine Insel mit mehreren Brücken.
  - Häufige Frage: Ist ein Spaziergang möglich, so dass man
    - schließlich wieder am Ausgangspunkt ankommt und
    - alle Brücken genau einmal überquert?
- Graphentheoretisch:
  - „Geschlossene Euler-Tour“
  - Existiert geschlossener, einfacher Pfad über alle Kanten?



0	2	0	1
2	0	2	1
0	2	0	1
1	1	1	0



- Eulers Antwort: Genau dann, wenn alle Knoten von geradem Grad sind bzw. die Spalten- / Zeilensummen der Adjazenzmatrix alle gerade sind.