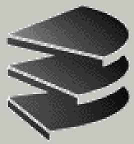



LMU

# Kapitel 4 - Graphalgorithmen

- Darstellungen
  - Adjazenzmatrix
  - Adjazenzliste
- Graphdurchlauf
  - Tiefendurchlauf
  - Breitendurchlauf
- Kürzeste Wege
  - Dijkstra-Algorithmus
  - Floyd-Algorithmus
- Minimal Spanning Tree
  - Prim-Algorithmus
- Anwendungen
  - Euler-Tour

# Motivation

Viele reale Fragestellungen lassen sich durch Graphen darstellen:

- Beziehungen zwischen Personen:
  - Person A kennt Person B
  - Mannschaft A gegen Mannschaft B
- Verbindungen zwischen Punkten:
  - Straßennetz
  - Telefonnetz
- Darstellung elektronischer Schaltungen

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

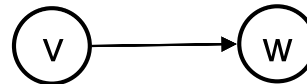
Ein *gerichteter Graph* (engl. digraph = "directed graph") ist ein Paar $G = (V, E)$ mit einer endlichen, nichtleeren Menge $V$, deren Elemente Knoten (nodes, vertices) heißen, und einer Menge $E \subseteq V \times V$, deren Elemente Kanten (edges, arcs) heißen.

Bemerkungen:

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

# Gerichteter Graph

Graphische Darstellung einer Kante von v nach w:

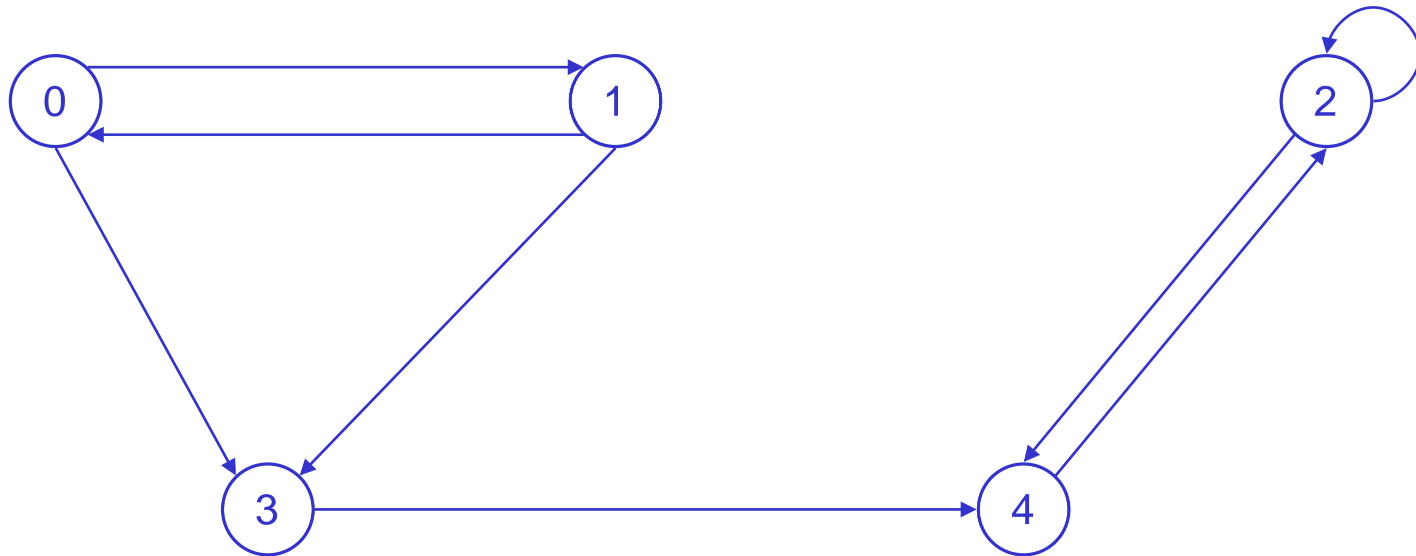



Begriffe:

- 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)}



# Definitionen

- *Grad* eines Knotens := Anzahl der ein- und ausgehenden Kanten
- Ein *Pfad* ist eine Folge von Knoten $v_0, \ldots, v_{n-1}$ mit $(v_i, v_{i+1}) \in E$ für $0 \leq i < n-1$, 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* eines Graphen $G = (V, E)$ ist ein Graph $G'=(V', E')$ mit $V' \subseteq V$ und $E' \subseteq E \cap (V' \times V')$.

# Markierungen

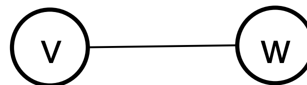
- Man kann Markierungen oder Beschriftungen für Kanten und Knoten einführen.
- Häufig verwendet: Kostenfunktionen für Kanten
- Notation:
  - c[v,w] oder cost(v,w), c(v,w)
- Bedeutung:
  - Entfernung zwischen v und w
  - Reisezeit
  - Reisekosten
  - …

# 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 kanten-orientiert abspeichern.
- Knoten-orientierte Darstellungsform ist gebräuchlicher und existiert in vielen verschiedenen Variationen. (Hier: Adjazenzmatrix)

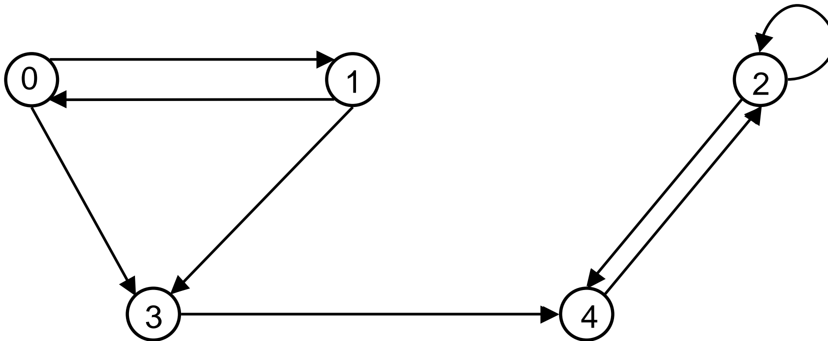
Die Adjazenzmatrix A ist eine boolesche Matrix mit:

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

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:




| von \ 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 |

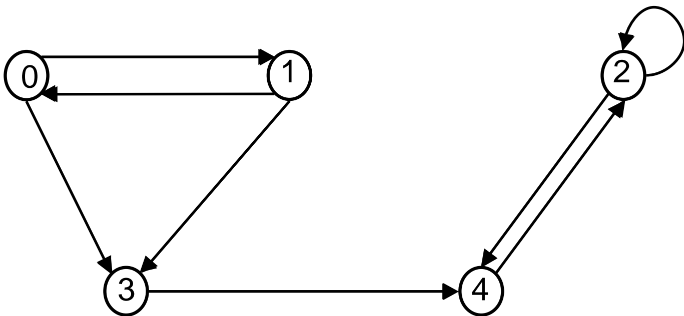
# Adjazenzmatrix

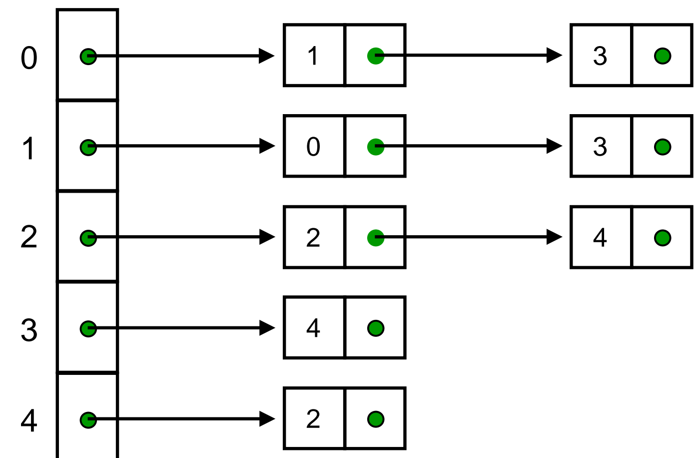
- Vorteile
  - Entscheidung, ob $(i, j) \in E$ in Zeit $O(1)$
- Nachteile
  - Platzbedarf stets $O(|V|^2)$, ineffizient falls |E| << $|V|^2$
  - Initialisierung benötigt Zeit $O(|V|^2)$
- Kantenbeschriftung
  - statt booleschen Werten Zusatzinformation (bspw. Integer) als Matrixeinträge speichern
  - Bsp: Kosten; Weglängen
  - Definition Kostenadjazenzmatrix: $A_{ij} = \begin{cases} c(v_i, v_j) & (v_i, v_j) \in E \\ \infty & sonst \end{cases}$

  mit $c(v_i, vj)$ Kosten für die Kante zwischen $v_i$ und $v_j$
- Achtung: bei boolescher Adjazenzmatrix bedeutet ein Eintrag 0, dass keine Kante besteht; bei Kostenadjazenzmatrix, dass die Kosten 0 sind, z.B. $c(v_i, vi) = 0$.

# Adjazenzliste

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



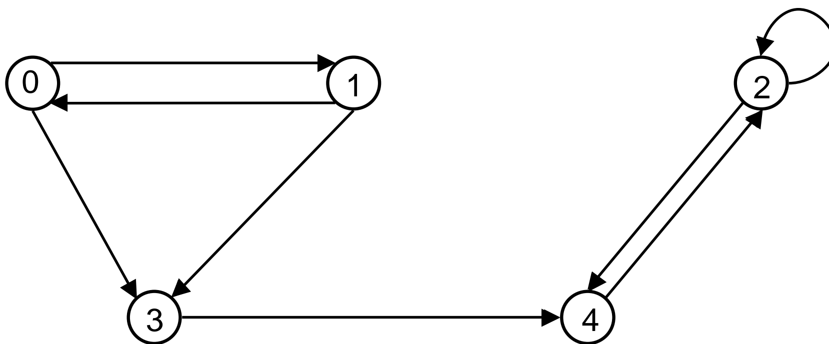



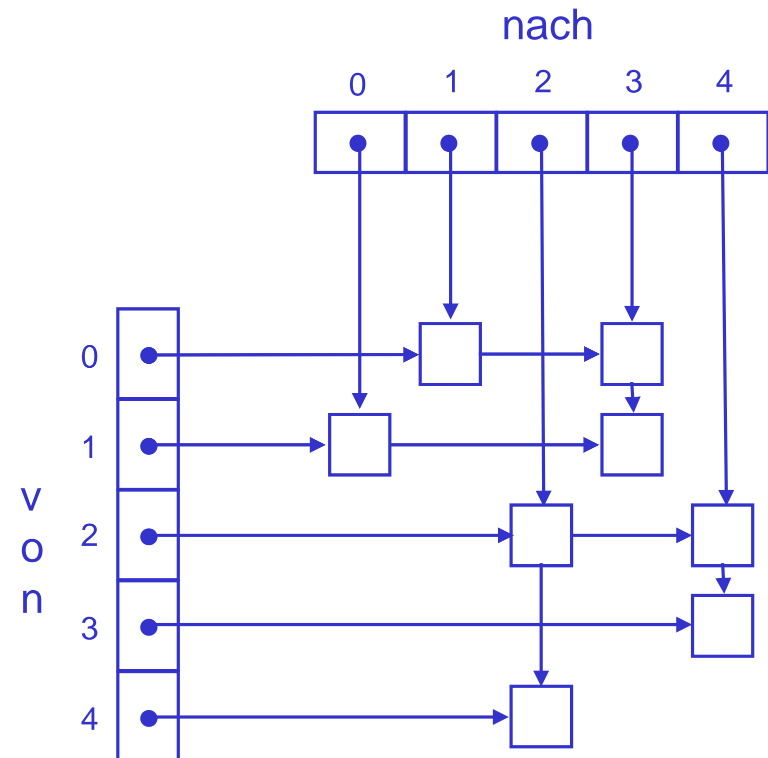
# Adjazenzliste

- 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] gdw. der repräsentierte Graph G eine Kante von Knoten i zu Knoten j enthält



# Mischform

Vorteil:

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

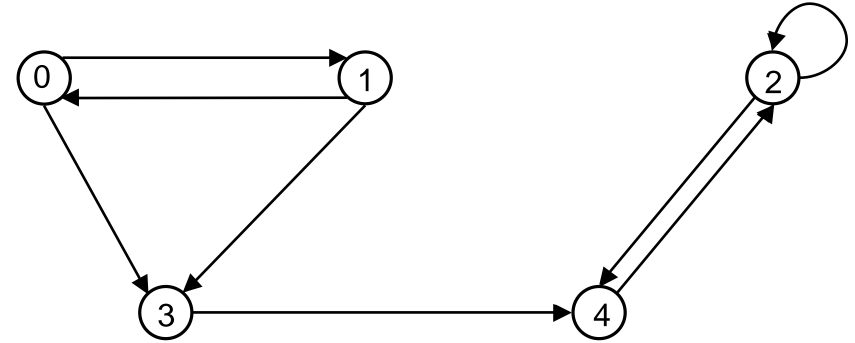
Nachteil:

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

# 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 $v_1,...,v_k$ die Nachfolger von v sind, ist $X_G(v)$ der Baum mit der Wurzel v und den Teilbäumen $X_G(v_1)$, …, $X_G(v_k)$




# Expansion - Beispiel

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

# Expansion - 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

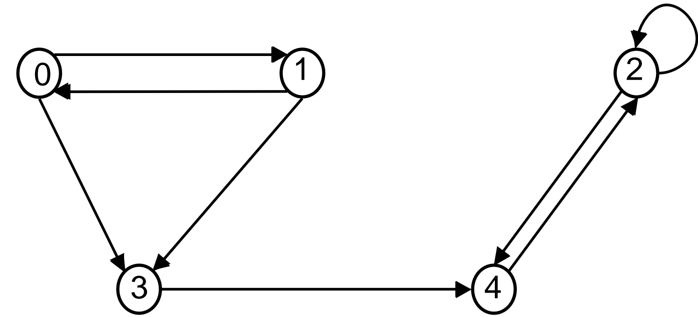
Entspricht Baum-Durchlauf durch Expansion (ggf. mit Abschneiden)

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

Wichtige Modifikation:

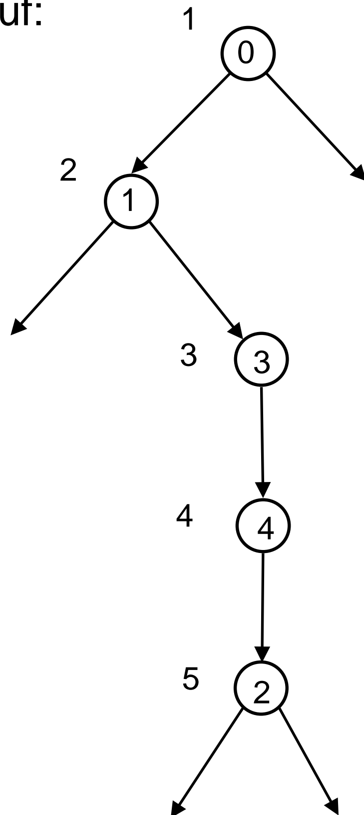
1. Schon besuchte Knoten müssen markiert werden, weil Graph-Knoten im Durchlauf mehrfach vorkommen können. (Zyklen!)
2. Abbruch des Durchlaufs bei schon besuchten Knoten.

# Beispiel

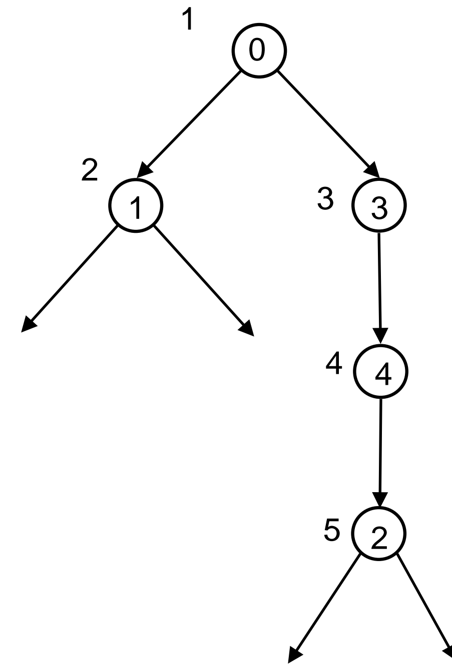



$G_1$ mit Startknoten: v = 0

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 Graphe 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 den ,,Suchbaum'': 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).
  - Markiere x als Baum-Knoten und bearbeite x.
  - Markiere alle ungesehenen Nachbar-Knoten von x als Rand-Knoten.

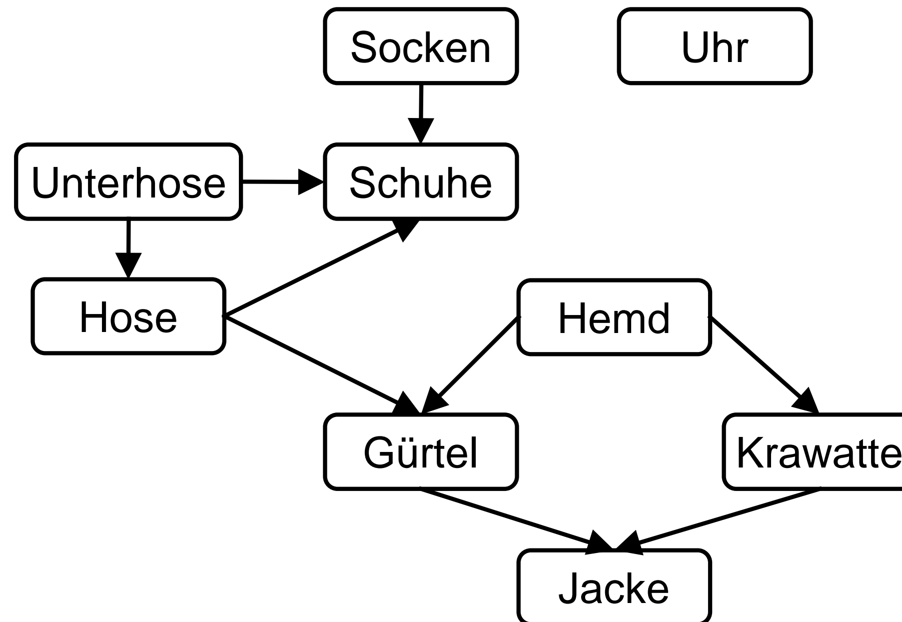
  … **until** (alle Knoten abgearbeitet)

d.h., keine Randknoten mehr und keine ungesehenen Knoten mehr

# Topologisches Sortieren

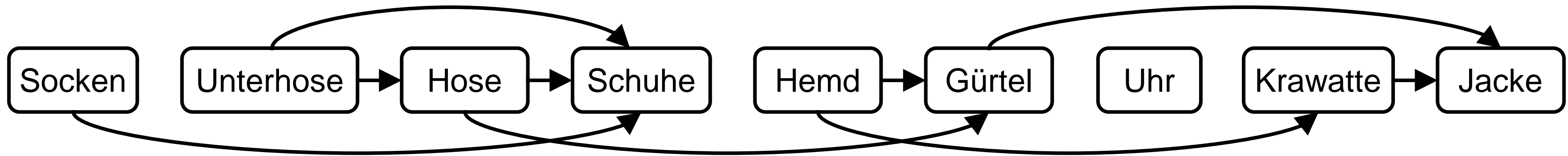
- Abhängigkeiten zwischen Aktionen („erst x, dann y“)
- gesucht: lineare Ordnung der Knoten, sodass für alle $(u,v) \in E$ der Knoten u „vor“ v liegt

→ Abarbeitungsfolge/-plan der Aktionen

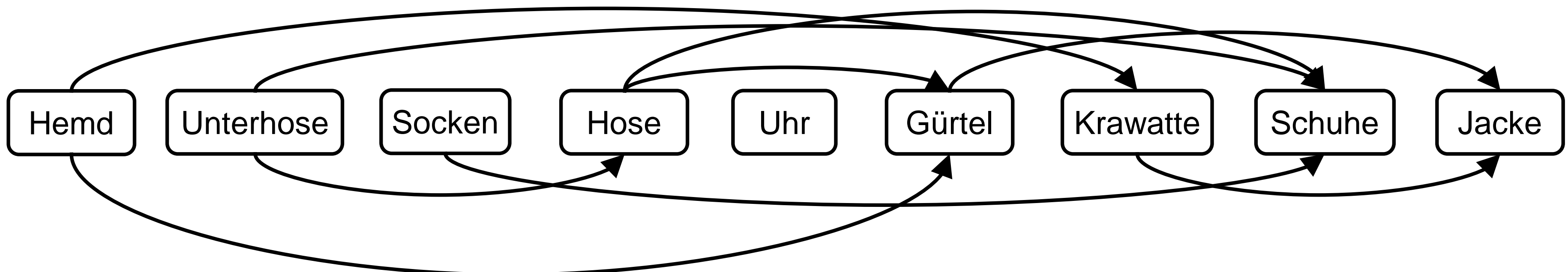



- formal: Einbettung einer Halbordnung/partiellen Ordnung (reflexiv, transitiv, antisymmetrisch) in eine lineare/totale Ordnung

# 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

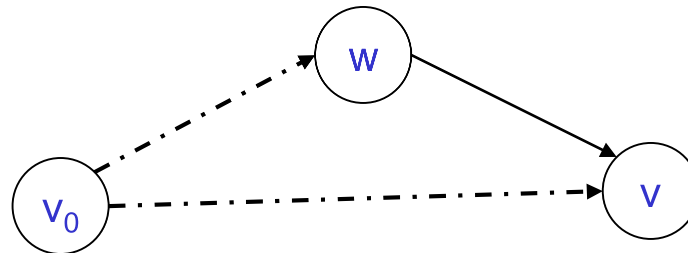
- Problemstellung: Suche kürzesten Weg
  1. Von einem Knoten zu einem anderen
  2. Von einem Knoten zu allen anderen: „Single Source Best Path“
  3. Von allen Knoten zu allen anderen: „All Pairs Best Path“
- Dijkstra-Algorithmus: Single Source Best Path
- Gegeben: Gerichteter Graph G mit Kostenfunktion (=Adjazenzmatrix)
  - c[v,w] $\begin{cases} \geq 0 & \text{falls eine Kante von v nach w existiert} \\ = \infty & \text{falls keine Kante von v nach w existiert} \\ = 0 & \text{für } w = v \end{cases}$
  - Startknoten $v_0$, Endknoten w
- Gesucht: Pfad von $v_0$ zu jedem Knoten w mit minimalen Gesamtkosten

# Ablauf Dijkstra-Algorithmus

- Edsger Wybe Dijkstra (1930-2002): niederländischer Informatiker & Turingpreisträger
- Erweitere sukzessive bekannte beste Pfade:
  - Kosten können durch Erweiterung nur wachsen
  - Falls beste Pfade von $v_0$ zu allen anderen Knoten ungleich w höhere Kosten haben, ist bester gefunden
  - Bester Pfad hat keinen Zyklus
  - Bester 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 des besten Pfads von $v_0$ über Knoten in $S_k$ nach v

# Grundidee Dijkstra-Algorithmus

- Wir speichern im Array D für jeden Knoten v die aktuell gültige Kostenschätzung.
- Als Initialisierung verwenden wir die Kosten der direkten Pfade.
- Obere Schranke D[v] verbesserungsfähig:
  - Betrachte jeden Knoten $w \in V$ als möglichen Zwischenknoten.
  - Existiert eine Kante von w nach v, sodass $D[w] + c[w,v] < D[v]$ ?
  - Setze dann $D[v]= D[w]+c[w,v]$

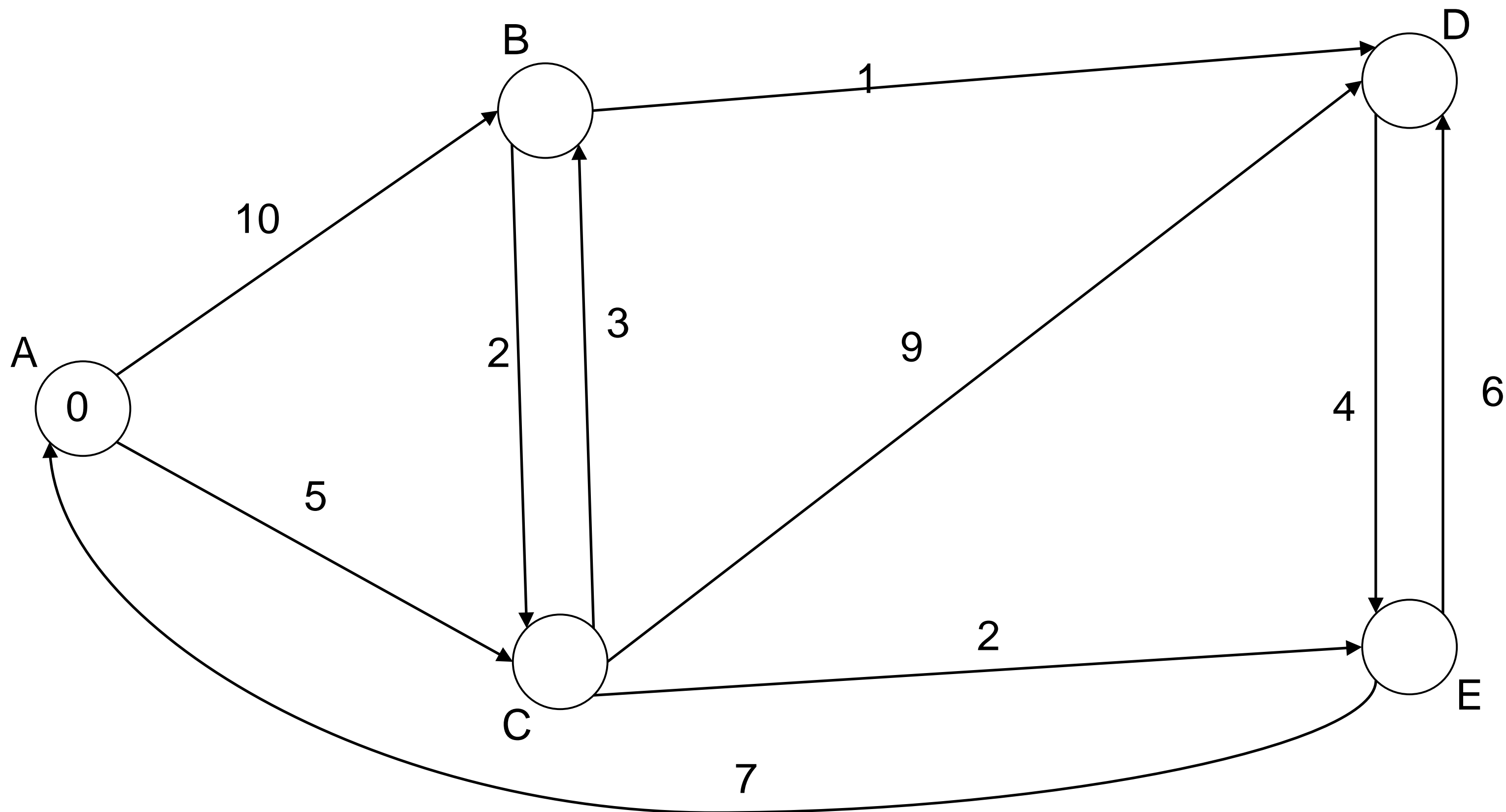


# Implementierung Dijkstra-Algorithmus

- S: bereits abgearbeitete Knoten
- D: aktuelle Kostenschätzung

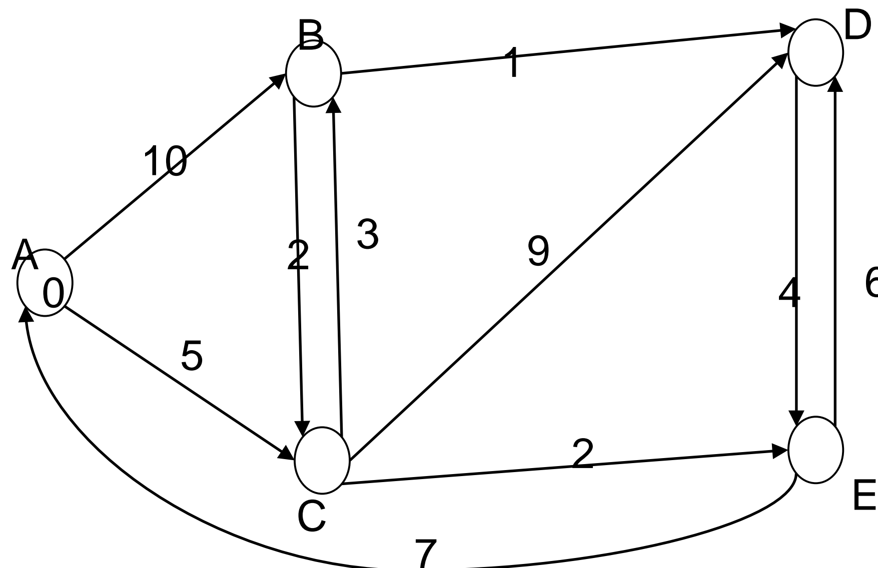
```
Dijkstra(G,v₀){
    S = {v₀};
    forall v ∈ V {
        D[v] = c[v₀,v];
    }
    while V \ S ≠ ∅ {
        choose w ∈ V \ S with D[w] minimal;
        S = S ∪ {w};
        for each v in V \ S {
            D[v] = min(D[v] , D[w]+c[w,v]);
        }
    }
}
```

# Dijkstra-Algorithmus graphisch

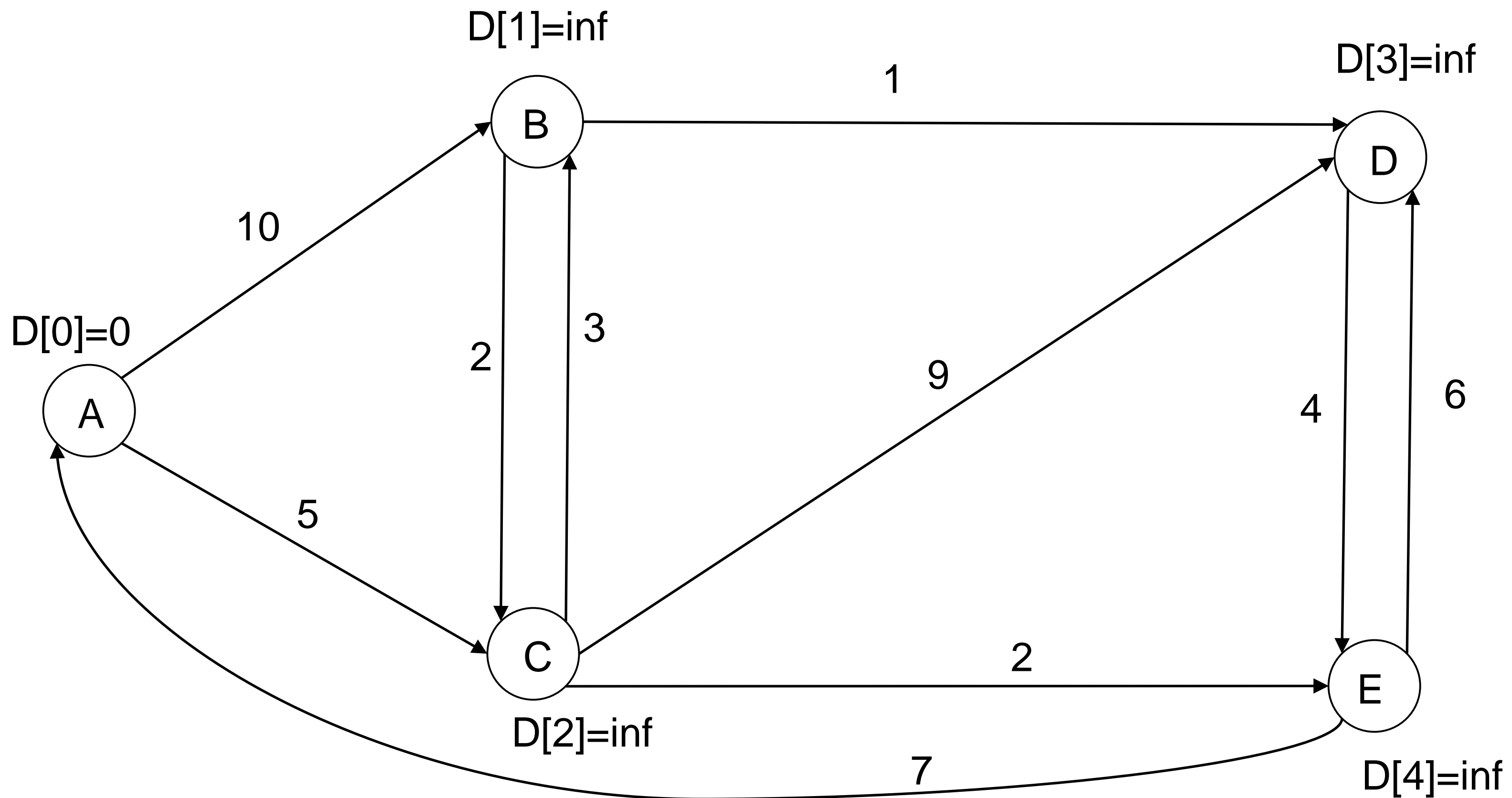


# Dijkstra: Beispiel

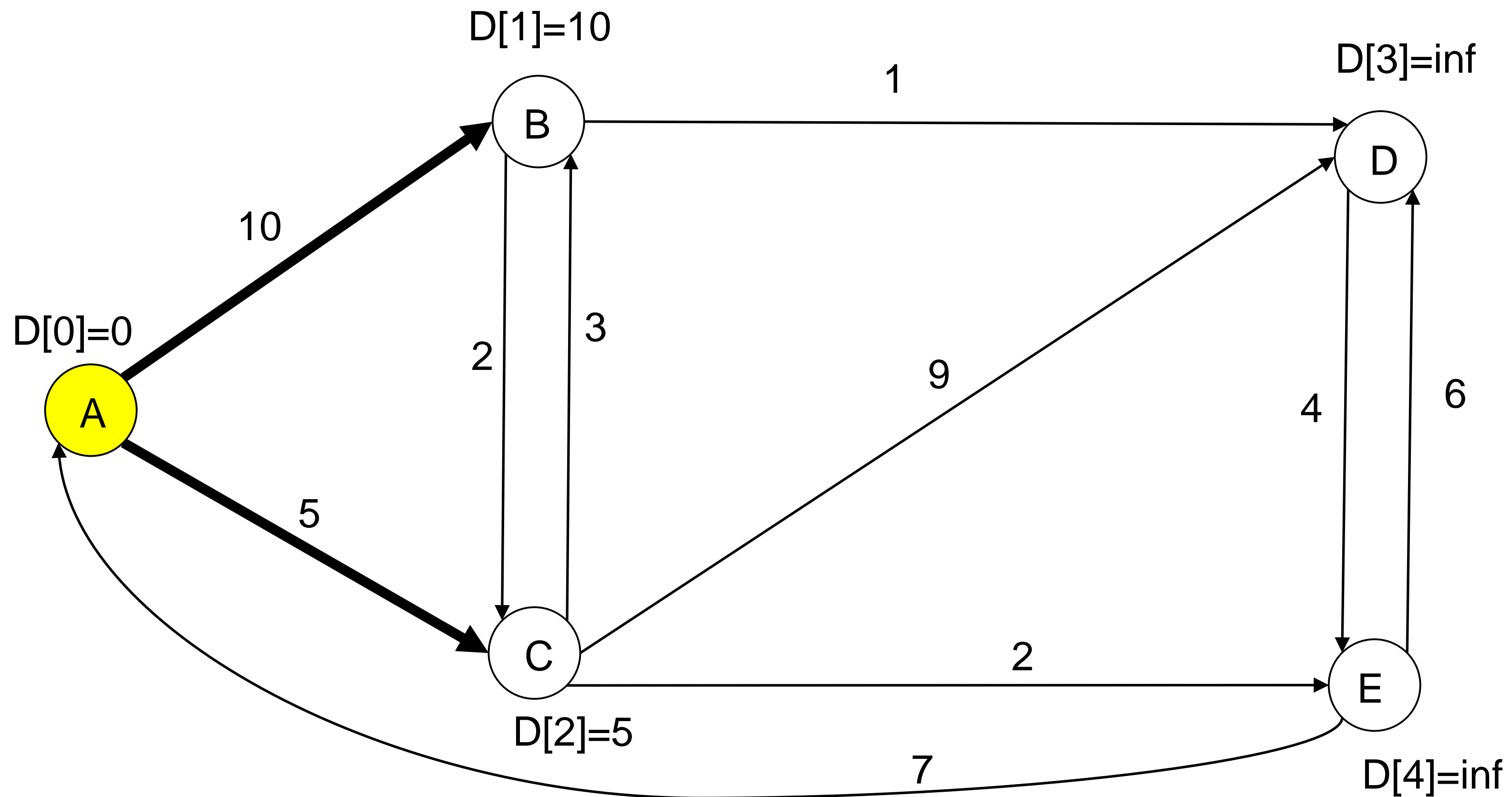
| **k** | **$w_k$** | **$S_k$** | **$D_k$(B)** | **$D_k$(C)** | **$D_k$(D)** | **$D_k$(E)** |
|---|---|---|---|---|---|---|
| 0 | - | { A } | 10 | 5 | ∞ | ∞ |
| 1 | C | { A,C } | **8** | 5 | **14** | **7** |
| 2 | E | { A,C,E } | 8 | 5 | **13** | 7 |
| 3 | B | { A,C,E,B } | 8 | 5 | **9** | 7 |
| 4 | D | { A,C,E,B,D } | 8 | 5 | 9 | 7 |



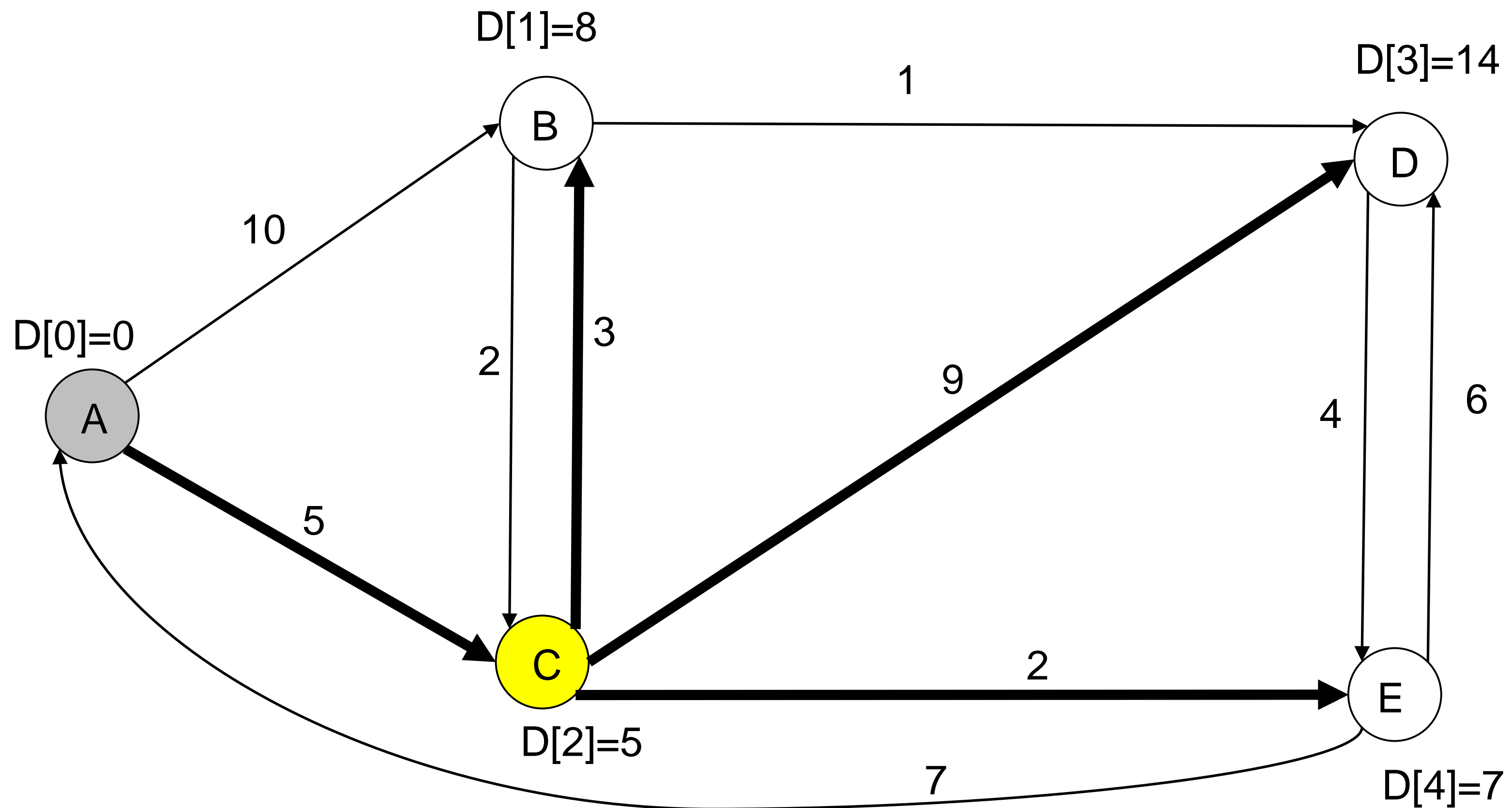
# Dijkstra-Algorithmus graphisch



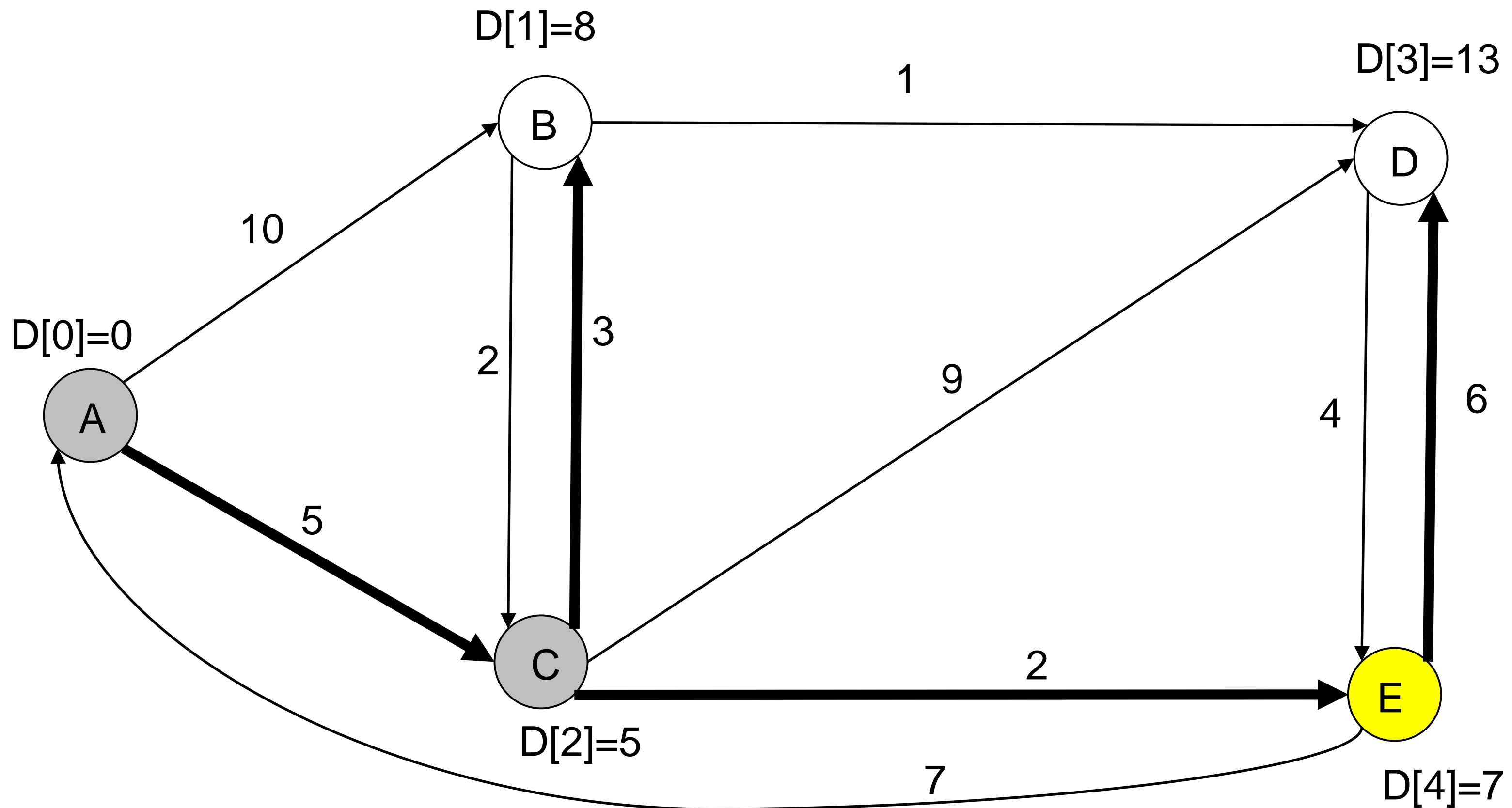
# Dijkstra-Algorithmus graphisch



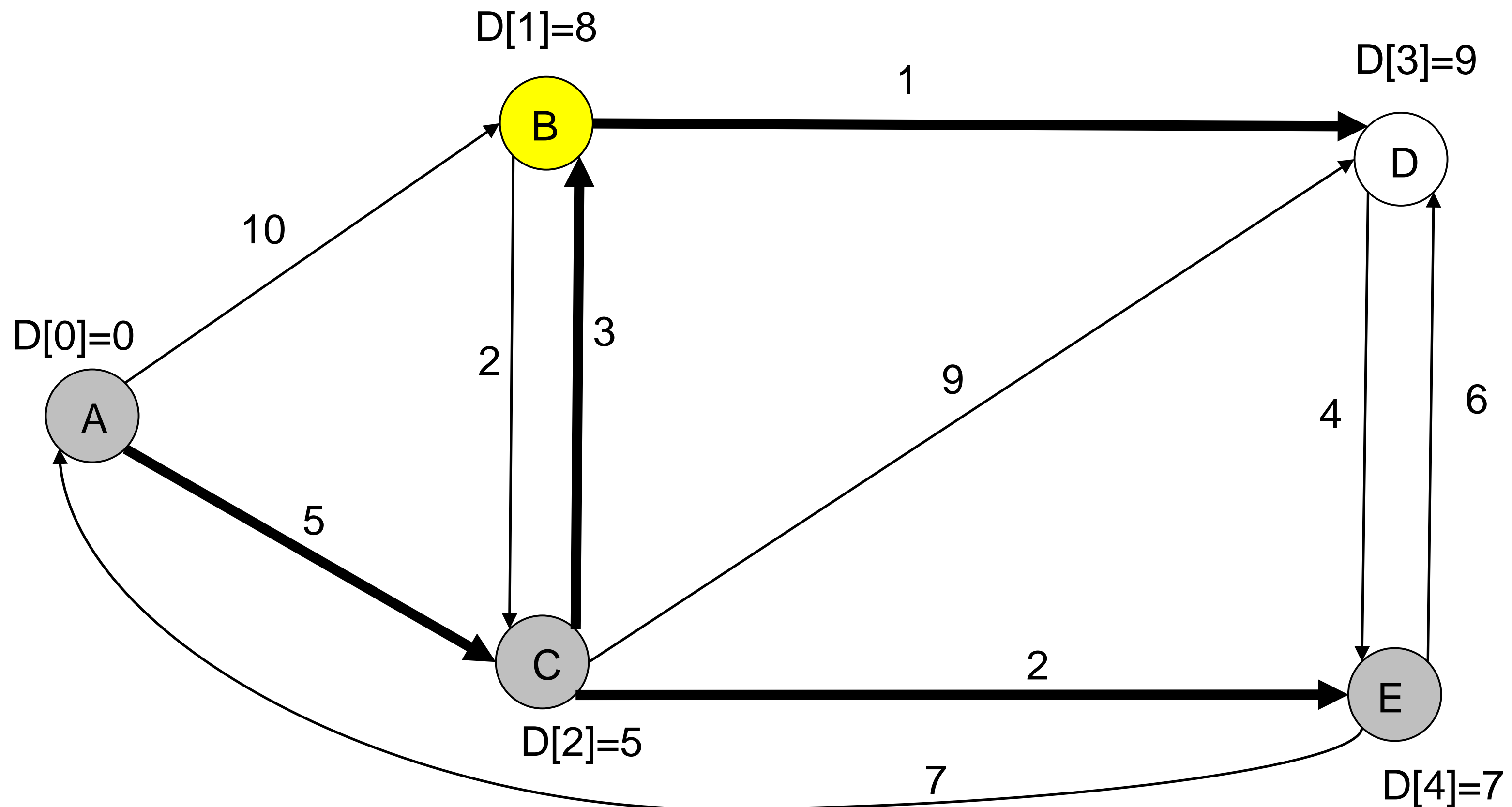
# Dijkstra-Algorithmus graphisch



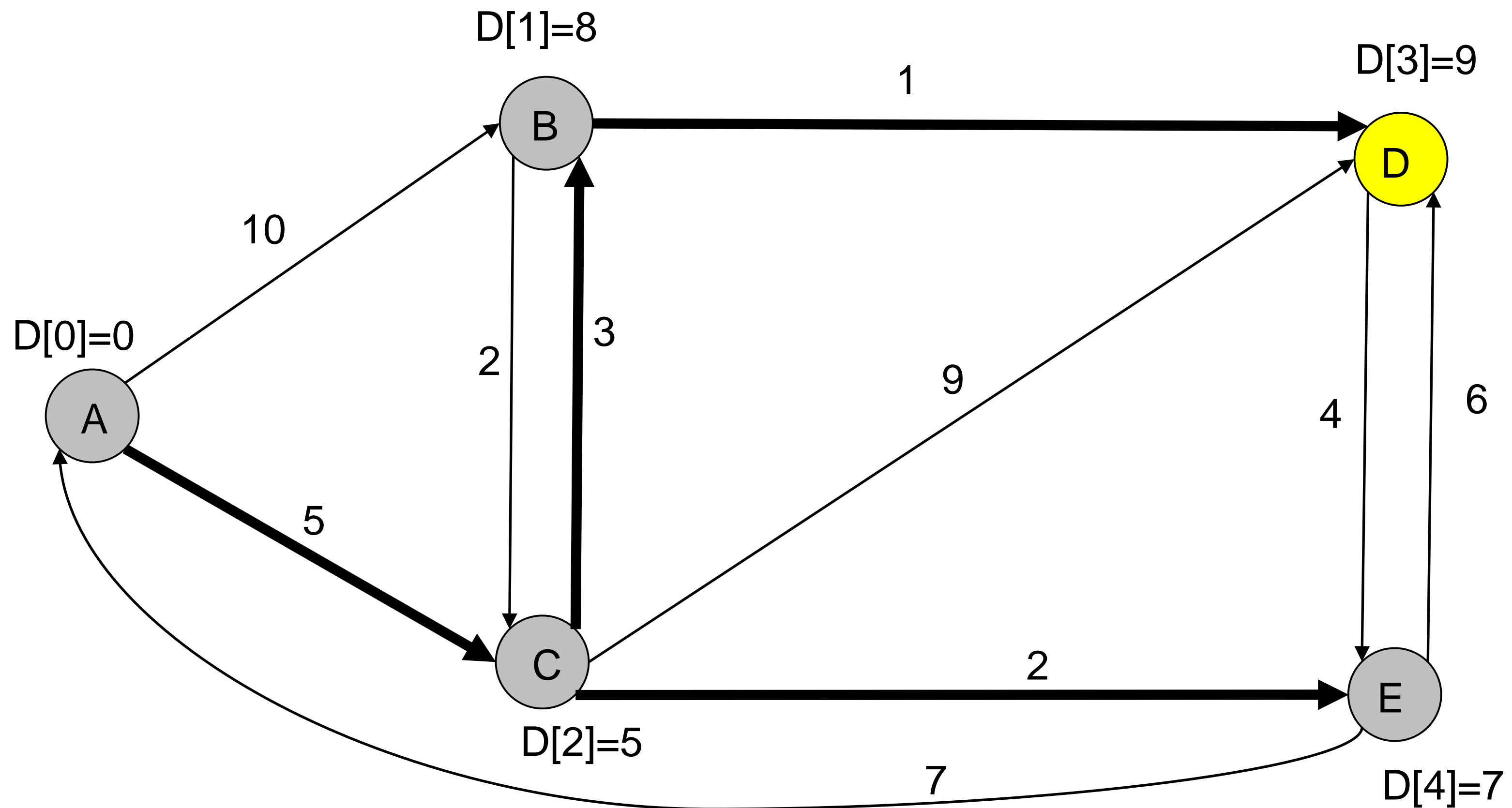
# Dijkstra-Algorithmus graphisch



# Dijkstra-Algorithmus graphisch



# Dijkstra-Algorithmus graphisch



# Analyse Dijkstra-Algorithmus

- 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 Best Path prinzipiell möglich, ergibt Zeitkomplexität $O(|V| \cdot |V|^2) = O(|V|^3)$,
- Andere Möglichkeit: Floyd-Algorithmus
- Robert W Floyd (1936 – 2001): Amerikanischer Informatiker & Turingpreisträger

# Floyd-Algorithmus

- „All Pairs Best Path“
- Gegeben: Gerichteter Graph G mit Kostenfunktion

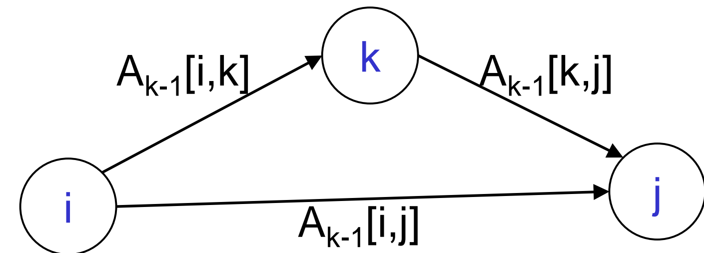
$$c[v,w] \begin{cases} \geq 0 & \text{falls eine Kante von v nach w existiert} \\ = \infty & \text{falls keine Kante von v nach w existiert} \\ = 0 & \text{für } w = v \end{cases}$$

- 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: Dynamische Programmierung

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




- Initialisierung durch direkte Kanten des gegebenen Graphen
  - Für alle i, j $\in$ {1, …, |V|}: setze $A_0[i,j] = c[i,j]$
  - Entspricht Lösung der Elementarprobleme
- Aktualisiere Matrix A[i,j] für Zwischenknoten k:
  - Sind Wege über Knoten k günstiger als bisherige Wege?
  - $A_k[i,j] = \min \{ A_{k-1}[i,j], A_{k-1}[i,k] + A_{k-1}[k,j] \}$
  - Entspricht Zusammensetzen der Teilergebnisse zur Gesamtlösung

# Floyd-Algorithmus: Implementierung

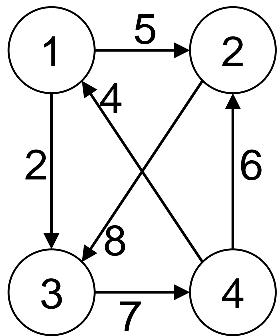
```
Floyd (A,C,P) {  /* A min. Kosten, C Adjazenzmatrix, P Vorgänger bester Pfad */
// Initialisierung
  for (i = 1; i <= n; ++i) {
    for (j = 1; j <= n; ++j) {
           A[i,j] = c[i,j]; P[i,j] = 0;
    }
  }

// Iteration über die möglichen Zwischenknoten k
  for (k = 1; k <= n; ++k) {
    for (i = 1; i <= n; ++i) {
      for (j = 1; j <= n; ++j) {
        if (A[i,j] > A[i,k] + A[k,j]) {
          A[i,j] = A[i,k] + A[k,j];
          P[i,j] = k;
        }
      }
    }
  }
}
```

falls Weg über k besser / kürzer als bisher bester Weg

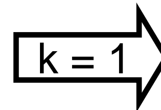
# Floyd-Algorithmus: Beispiel

$$A_{i,k} + A_{k,j} < A_{ij}$$

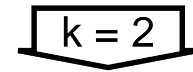



Initialisierung

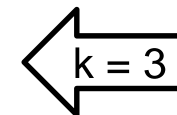
$$\begin{pmatrix} 0 & 5 & 2 & \infty \\ \infty & 0 & 8 & \infty \\ \infty & \infty & 0 & 7 \\ 4 & 6 & \infty & 0 \end{pmatrix}$$



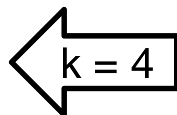

$$\begin{pmatrix} 0 & 5 & 2 & \infty \\ \infty & 0 & 8 & \infty \\ \infty & \infty & 0 & 7 \\ 4 & 6 & 6 & 0 \end{pmatrix}$$




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




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




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

# 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.
  - Innere Schleife läuft auf booleschen Werten:

```
if (not A[i,j])
    A[i,j] = A[i,k] && A[k,j];
```

# 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.

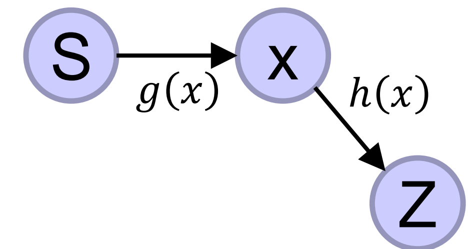
# A*-Algorithmus

Der A*-Algorithmus berechnet den besten Pfad von einem Startknoten zu einem Zielknoten unter Zuhilfenahme einer Heuristik

**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

- Es werden zwei Listen geführt:
  - **OpenList:** enthält Knoten, zu denen (irgend)ein Weg bekannt ist.
  - **ClosedList :** enthält Knoten, zu denen der kürzeste Weg bekannt ist.
- Die **OpenList** wird mit dem Startknoten initialisiert.
- Aus der **OpenList** wird immer der beste Knoten entfernt und:
  - seine (*nicht besuchten)* Nachfolger in die **OpenList** eingefügt sowie
  - er selbst in die **ClosedList** eingefügt.
- Wenn der Zielknoten erreicht wurde, ist auch ein **Pfad gefunden**.
- Wird der Zielknoten nicht erreicht, gibt es auch **keinen Pfad.**

**A* - Algorithmus:**

```
OpenList[startKnoten] = 0
solange OpenList ≠ ∅ :
    aktuellerKnoten = OpenList.entferneMin()
    wenn aktuellerKnoten == zielKnoten:
        Pfad gefunden
    ClosedList = ClosedList ∪ aktuellerKnoten
    knotenErweitern(aktuellerKnoten)
wenn OpenList keinen Knoten mehr enthält gibt es keinen Pfad

Funktion knotenErweitern(knoten)
für alle nachfolger von knoten:
    wenn nachfolger nicht enthalten in ClosedList:
        f = g(knoten) + c(knoten, nachfolger) + h(nachfolger)
        wenn OpenList[nachfolger] == null oder OpenList[nachfolger] > f:
            nachfolger.vorgänger = knoten
            OpenList[nachfolger] = f
```

# Qualitätseigenschaften

- **Vollständig**

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

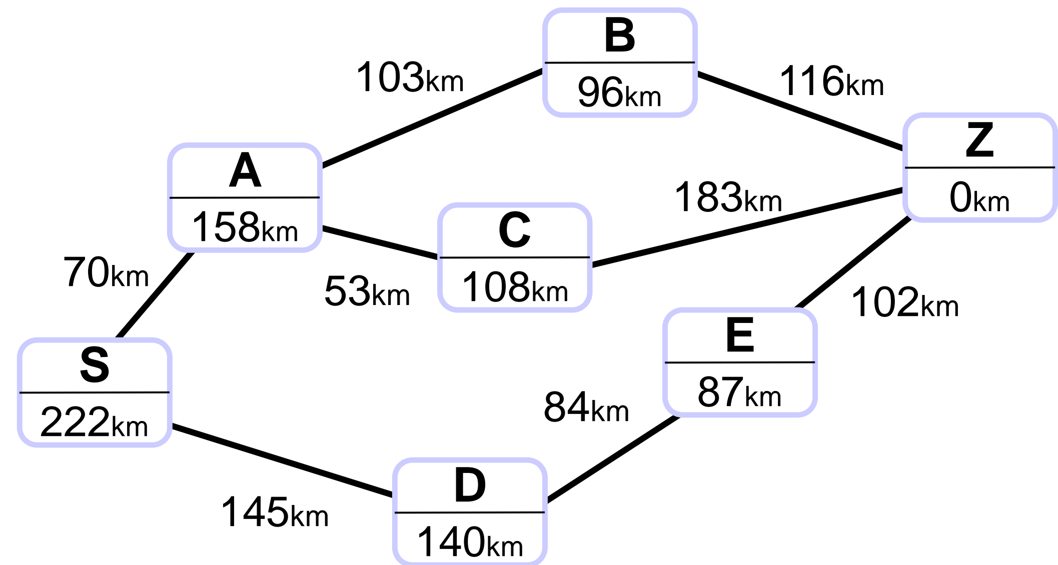
- **Optimal**

  Es wird immer eine optimale Lösung gefunden.

- **Optimal effizient**

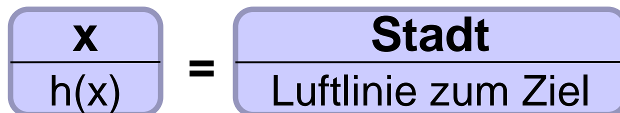
  bezogen auf die Laufzeit gibt es keinen Algorithmus, der die gleiche Heuristik verwendet und weniger Knoten besucht.

# Beispiel: Pfadsuche mit A*



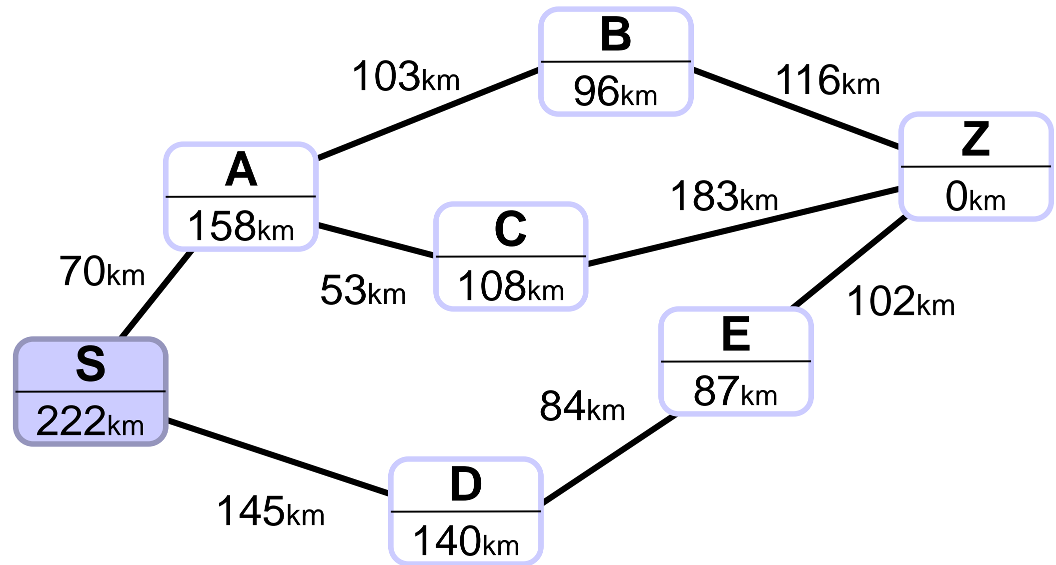

**Idee:**
Darstellung des Straßennetzes als gewichteter Graph.

**Knoten:**



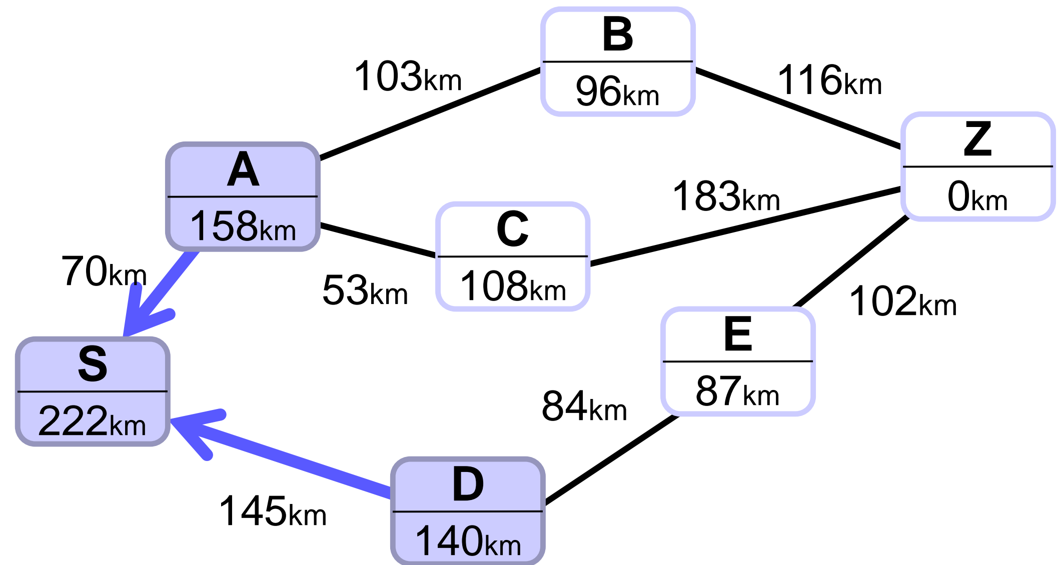

- Jeder Knoten steht für eine Stadt.
- Zwei Knoten sind verbunden wenn es eine *direkte* Straßenverbindung zwischen den entsprechenden Städten gibt .
- Die Kosten der Kanten entsprechen der Länge der Straße zwischen den Städten.
- Gesucht ist **der kürzeste Weg** von Stadt *S* nach Stadt *Z*.
- Als Heuristik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

# Pfadsuche mit A*



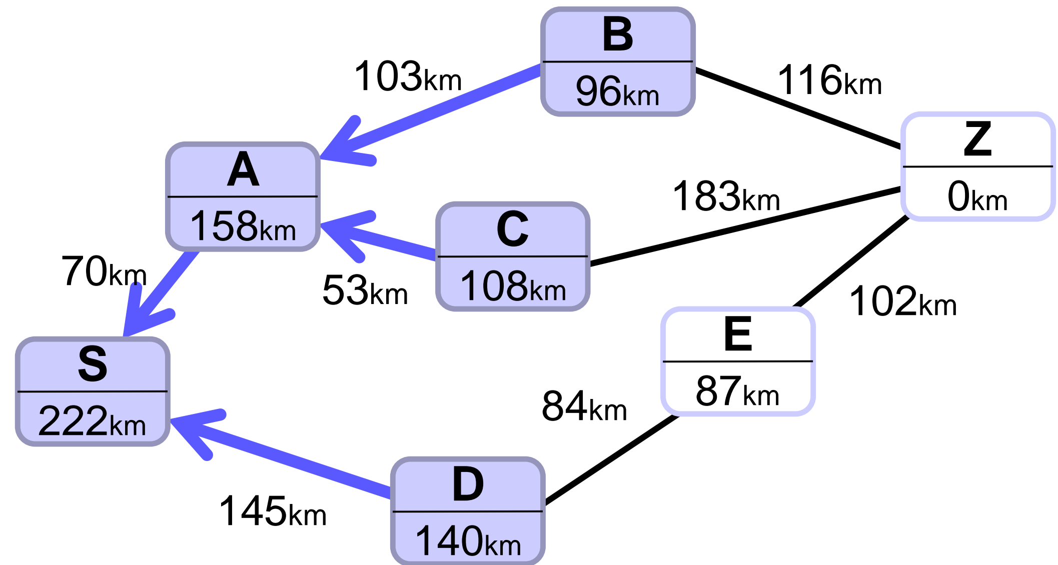

| Schritt | OpenList (Stadt, f ) | ClosedList (Stadt, Entfernung von S) |
| --- | --- | --- |
| 0 | (S, 0) | --- |

# Pfadsuche mit A*



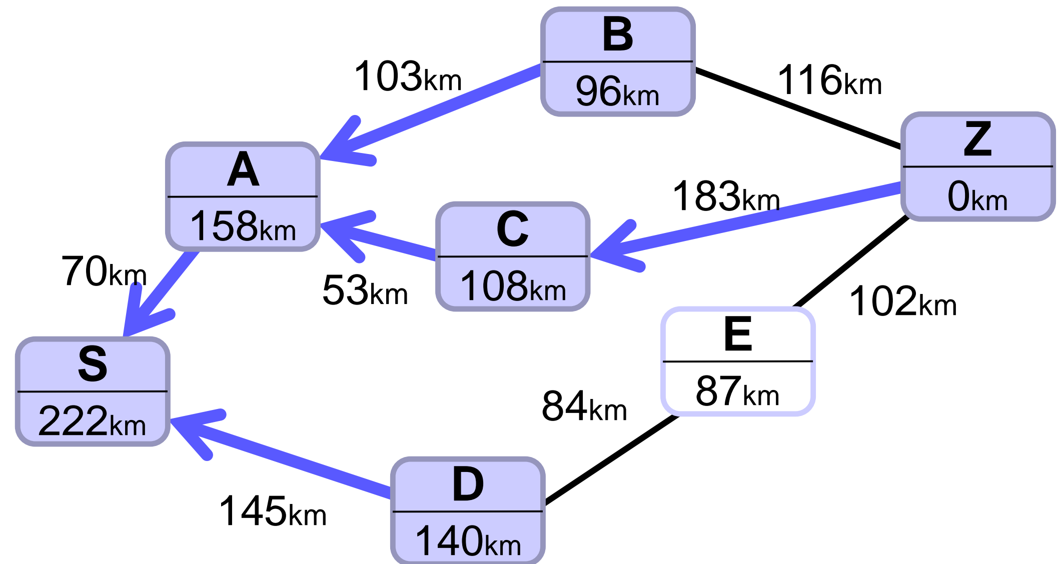

| Schritt | OpenList (Stadt, f ) | ClosedList (Stadt, Entfernung von S) |
| --- | --- | --- |
| 0 | (S, 0) | --- |
| 1 | (A, 228),(D, 285) | (S, 0) |

# Pfadsuche mit A*



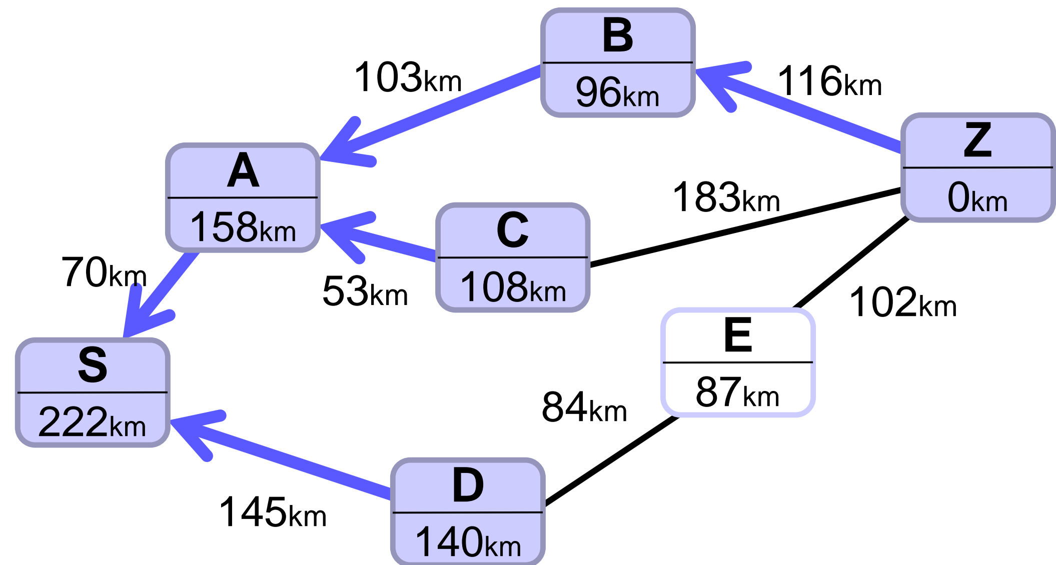

| Schritt | **OpenList** **(Stadt, f )** | **ClosedList** **(Stadt, Entfernung von S)** |
|---|---|---|
| 0 | (S, 0) | --- |
| 1 | (A, 228),(D, 285) | (S, 0) |
| 2 | (D, 285), (B, 269), (C, 231) | (S, 0), (A, 70) |

# Pfadsuche mit A*



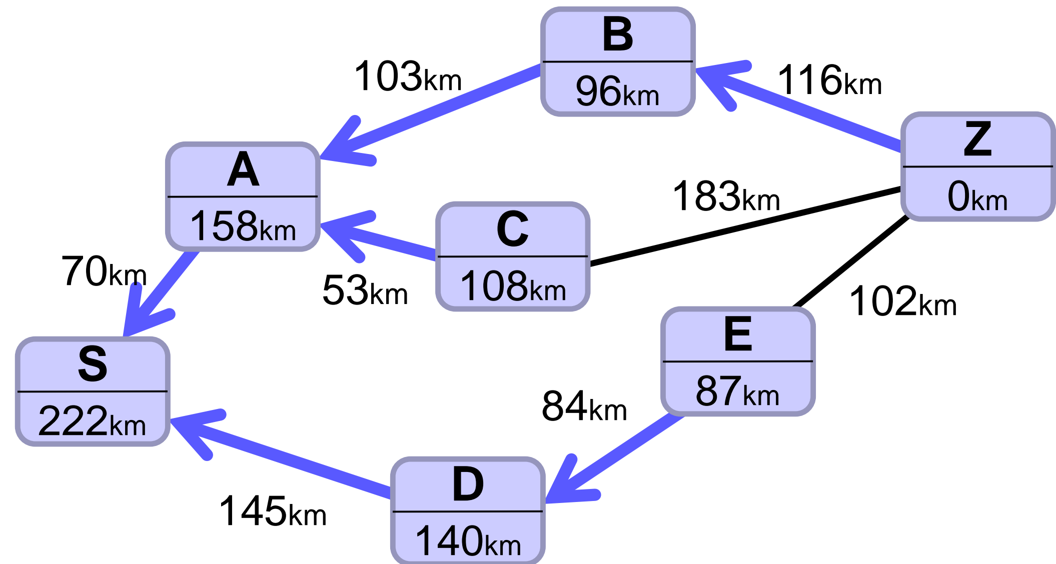

| Schritt | **OpenList** (Stadt, f ) | **ClosedList** (Stadt, Entfernung von S) |
|---|---|---|
| 0 | (S, 0) | --- |
| 1 | (A, 228),(D, 285) | (S, 0) |
| 2 | (D, 285), (B, 269), (C, 231) | (S, 0), (A, 70) |
| 3 | (D, 285), (B, 269), (Z, 306) | (S, 0), (A, 70), (C, 123) |

# Pfadsuche mit A*



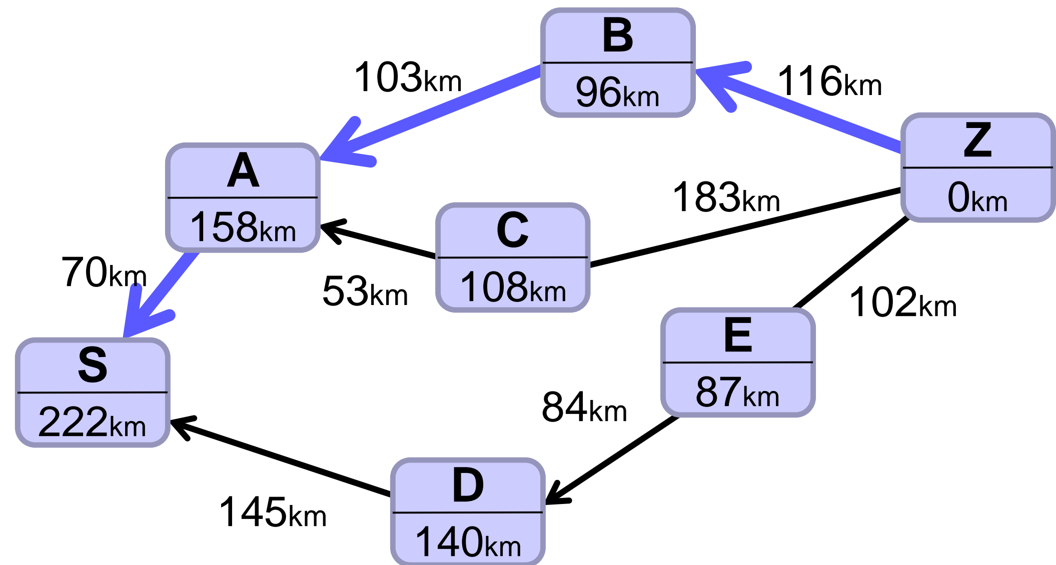

| Schritt | OpenList (Stadt, f ) | ClosedList (Stadt, Entfernung von S) |
|---|---|---|
| 0 | (S, 0) | --- |
| 1 | (A, 228),(D, 285) | (S, 0) |
| 2 | (D, 285), (B, 269), (C, 231) | (S, 0), (A, 70) |
| 3 | (D, 285), (B, 269), (Z, 306) | (S, 0), (A, 70), (C, 123) |
| 4 | (D, 285), (Z, 289) | (S, 0), (A,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

# Pfadsuche mit A*




| 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) |

# Pfadsuche mit A*

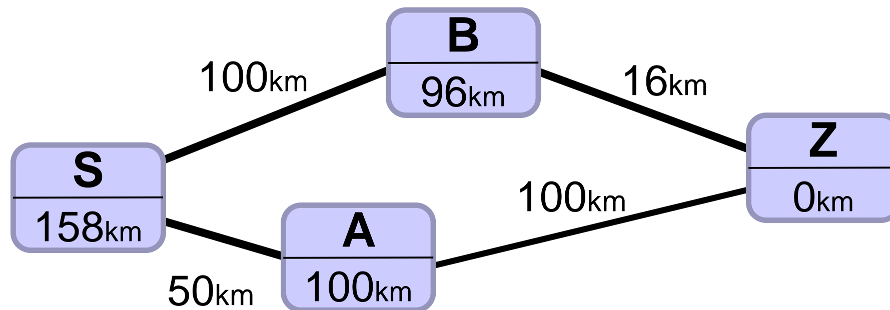



| Schritt | **OpenList** (Stadt, f ) | **ClosedList** (Stadt, Entfernung von S) |
|---|---|---|
| 0 | (S, 0) | --- |
| 1 | (A, 228),(D, 285) | (S, 0) |
| 2 | (D, 285), (B, 269), (C, 231) | (S, 0), (A, 70) |
| 3 | (D, 285), (B, 269), (Z, 306) | (S, 0), (A, 70), (C, 123) |
| 4 | (D, 285), (Z, 289) | (S,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

# Eigenschaft der Heuristik bei A*

- 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

Geg.: Ungerichteter Graph $G = (V, E)$. Es gilt

- Knoten $v, w$ verbunden: es gibt einen Pfad von $v$ nach $w$
- $G$ verbunden: alle Knotenpaare verbunden
- Freier Baum: $G$ verbunden, keine Zyklen
- Spannbaum: freier Baum $G' = (V, E'), E' \subseteq E$;
  falls $G$ bewertet, dann Kosten $G'$ = Summe über Kosten $E'$

- Minimaler Spannbaum (MST) zu $G$ (ungerichteter, bewerteter Graph): Spannbaum mit minimalen Kosten

- MST-Eigenschaft: alle paarweise disjunkten Teilbäume eines MST sind jeweils über eine minimale Kante (bzgl. Kosten) verbunden.

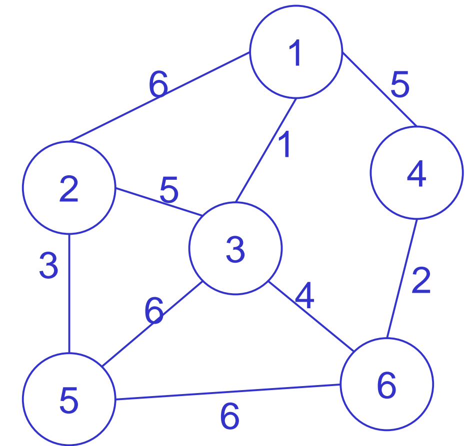
# Prim-Algorithmus

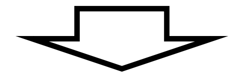
- Sei $V = \{1, \ldots, n\}$ und $T = (U, E')$ der zu konstruierende minimale Spannbaum

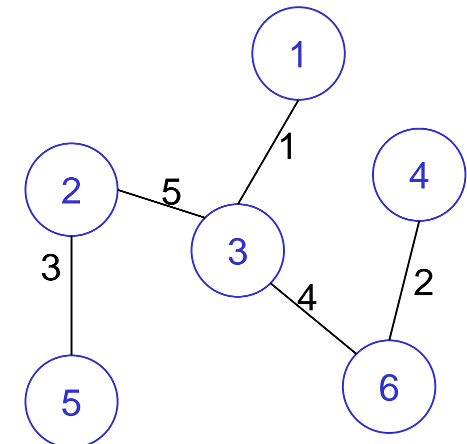
```
Prim {
    E' = ∅;
    U = {v_0};                    // Knoten, die schon besucht wurden
    while (U ≠ V) {
        choose (u, v) minimal (where u ∈ U, v ∈ V − U);
        U = U ∪ {v};
        E' = E' ∪ {(u, v)};
    }
}
```

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

# Beispiel Prim-Algorithmus






- E' = { }; U = { 1 }
  - Aus (1,4), (1,3), (1,2) ist (1,3) minimal
- E' = { (1,3) }; U = { 1,3 }
  - Aus (1,4), (1,2), (3,2), (3,5), (3,6) ist (3,6) minimal
- E' = { (1,3), (3,6) }; U = { 1,3,6 }
  - Aus (1,4), (1,2), (3,2), (3,5), (6,4), (6,5) ist (6,4) minimal
- E' = { (1,3), (3,6), (6,4) }; U = { 1,3,6,4 }
  - Aus (1,2), (3,2), (3,5), (6,5) ist (3,2) minimal
- E' = { (1,3), (3,6), (6,4), (3,2) }; U = { 1,3,6,4,5 }
  - Aus (1,2), (3,2), (5,2) ist (5,2) minimal
- E' = { (1,3), (3,6), (6,4), (3,5), (5,2) }; U = { 1,3,6,4,5,2 }

T = (U,E')

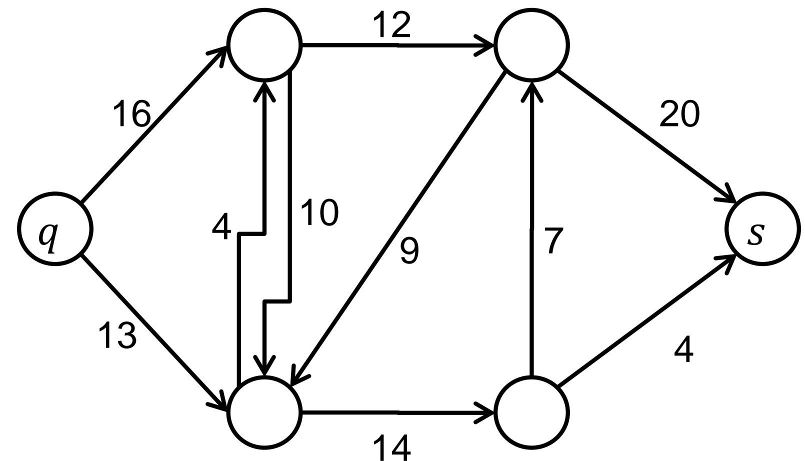
# Paradigma: Greedy-Algorithmen

- Wähle zum aktuellen Zeitpunkt immer jenen Folgezustand, der den bestmöglichen Gewinn verspricht
  - Bsp. minimaler Spannbaum: wähle kostenminimale Kante, welche von den aktuellen Knoten aus erreichbar ist
- Lokale Optimierung mit der Hoffnung, dass globales Optimum erreicht wird
  - eine getroffene Entscheidung wird nicht revidiert
- Greedy-Algorithmen liefern nicht für alle Probleme die optimale Lösung
- Falls Greedy-Algorithmus auf einem Problem optimale Lösungen liefert
  - gilt wie bei der dynamischen Programmierung, dass die optimale Lösung optimale Teillösungen enthält
  - aber im Gegensatz zur dyn. Programmierung gilt die „greedy choice property“: eine lokal optimale Lösung ist <u>stets</u> Teil einer global optimalen Lösung

# Maximaler Fluss

- 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
- anschaulich:
  - Wasserleitungen mit unterschiedlichen Kapazitäten
  - Wie viel Wasser kann von $q$ zu $s$ fließen?
    - Beispiel rechts: Übung

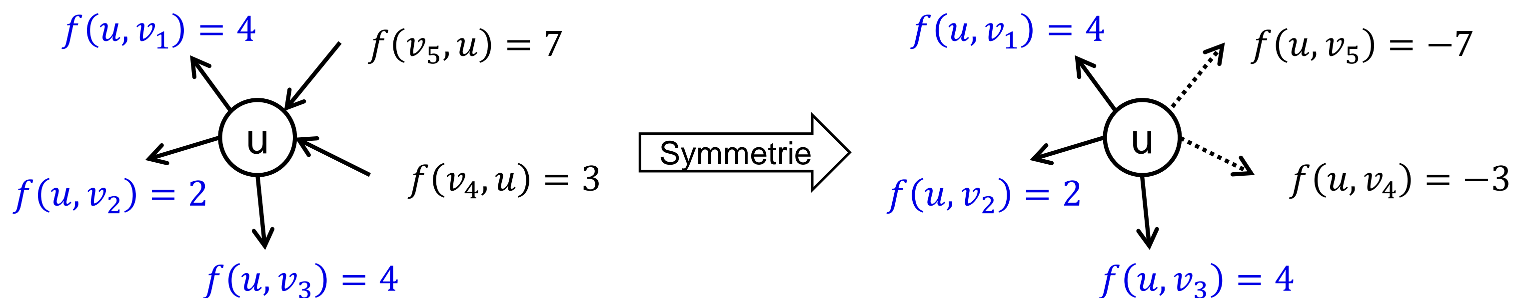
Wir setzen $c(u, v) = 0$, falls $(u, v) \notin E$



# Definition Fluss

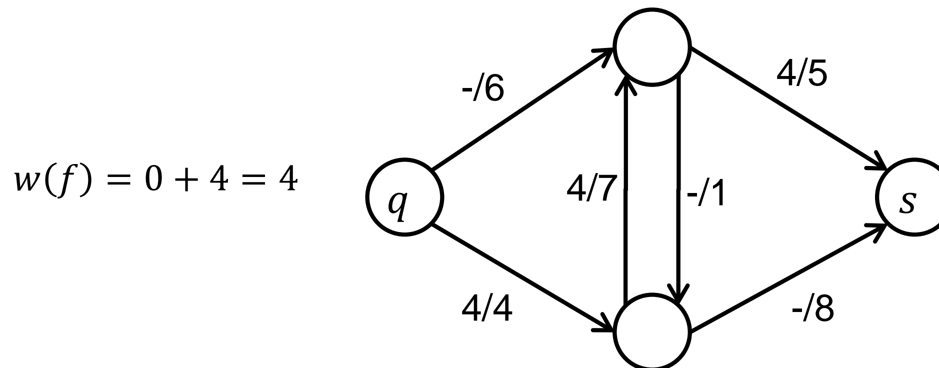
- Ein Fluss ist eine Funktion $f: V \times V \rightarrow R$ mit den Eigenschaften
  - Kapazitätsbeschränkung: Für $u, v \in V$ gilt $f(u, v) \leq c(u, v)$

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



# Problemstellung 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



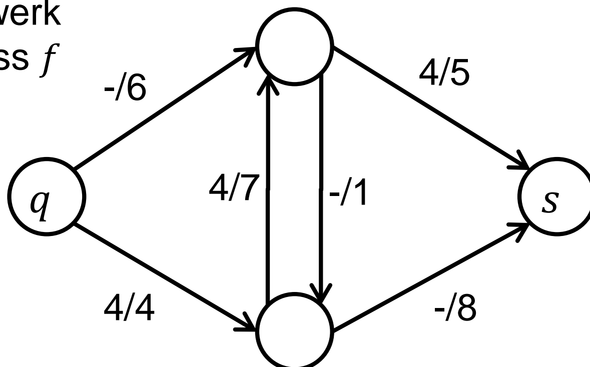

- Problem des maximalen Flusses
  - Gegeben ein Flussnetzwerk $(G, c)$
  - Gesucht ein Fluss $f$ auf $(G, c)$ mit maximalen Wert $w(f)$

# Residualnetzwerk

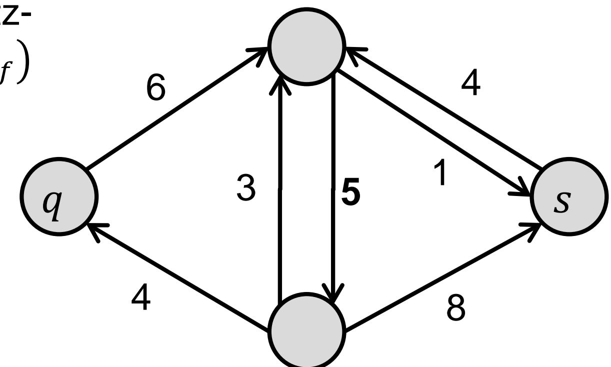
- Restkapazität $c_f(u,v)$ zwischen $u,v \in V$ ist $c_f(u,v) = c(u,v) - f(u,v)$
  - beachte: formal ist $f(u,v) < 0$ möglich (vgl. Symmetrie)
- Der Restgraph $G_f = (V, E_f)$ bzgl. Flussnetzwerk $(G,c)$ und Fluss $f$ ist definiert durch die Kantenmenge $E_f = \{(u,v) \in V \times V \mid c_f(u,v) > 0\}$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

  → *„Flussnetzwerk minus Fluss = Residualnetzwerk"*

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

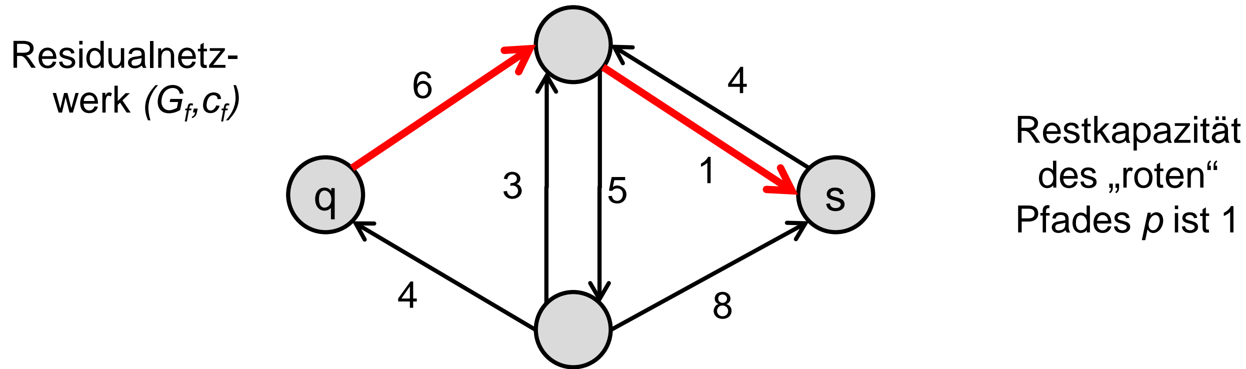



Residualnetzwerk $(G_f, c_f)$

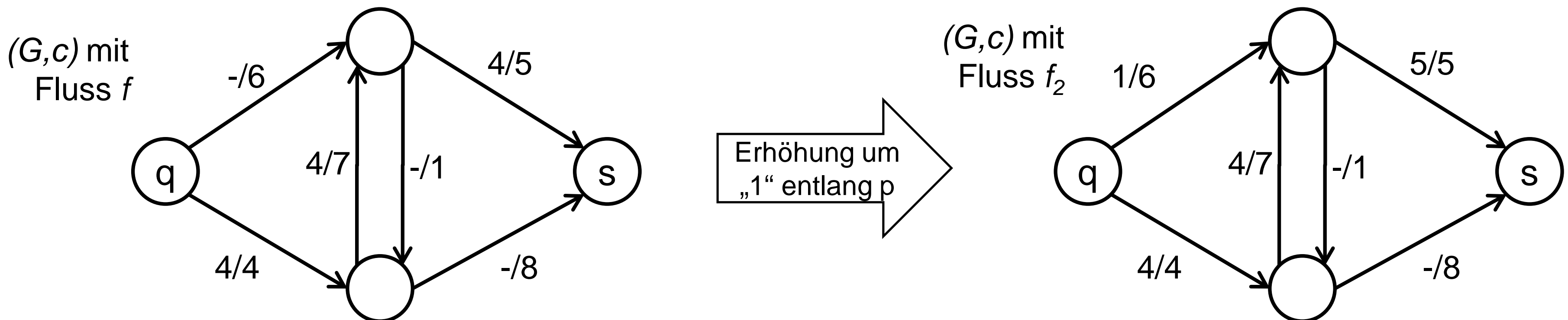


# Flussvergrößernder Pfad

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

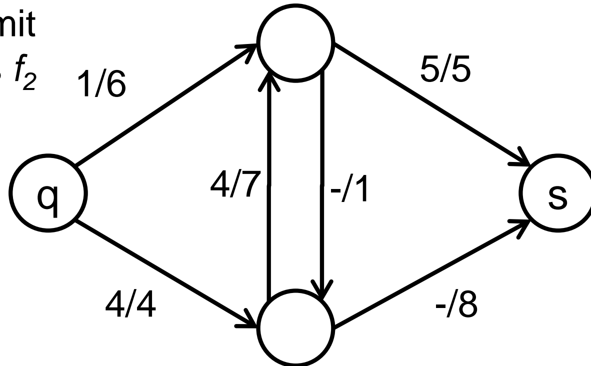



- Fluss *f* in *(G,c)* kann entlang des Pfades *p* um $c_f(p)$ erhöht werden

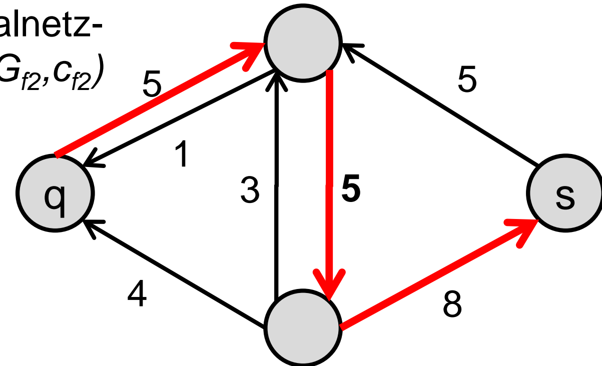


# Beispiel (fortgesetzt)

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

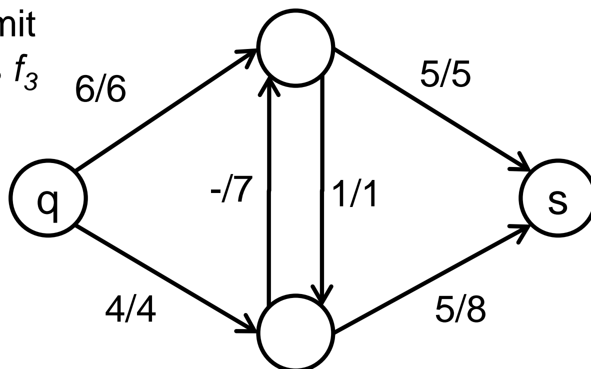



Residualnetzwerk $(G_{f2}, c_{f2})$

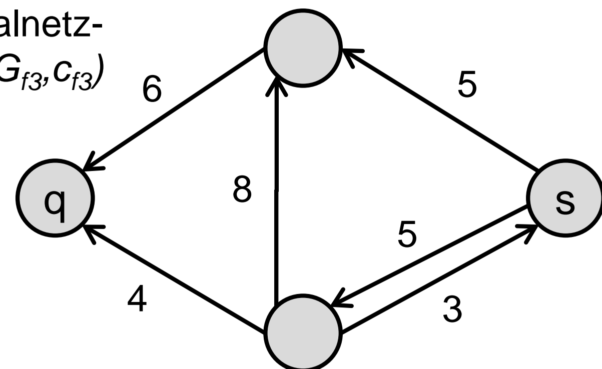



Achtung: „Rückwärtskante" entlang *p*

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




Residualnetzwerk $(G_{f3}, c_{f3})$




kein Pfad von *q* nach *s* möglich
→ maximalen Fluss gefunden

# Ford-Fulkerson-Methode

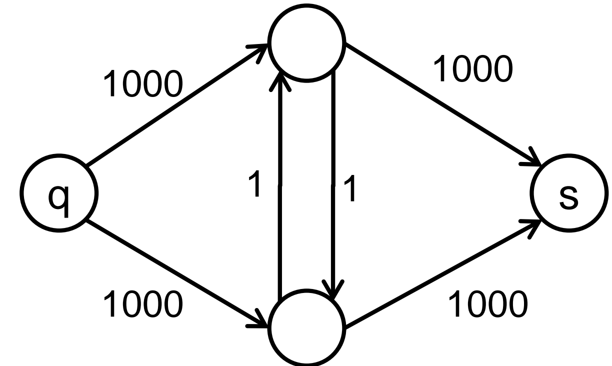
- Algorithmus FordFulkersonMethod(G,c) {

```
    Initialisiere Fluss f zu 0
    while es gibt einen flussvergrößernden Pfad p {
      erhöhe f entlang p
    }
    return f
  }
```

- Formal ist Erhöhung von *f* entlang *p* definiert durch

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

# 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 <u>kürzesten</u> flussvergrößernden Pfad
    - im Beispiel würden mittlere Kanten vermieden → schnellere Terminierung
  - → Algorithmus von Edmonds und Karp

# Edmonds-Karp-Algorithmus

- Algorithmus EdmondsKarp(G,c) {

    Initialisiere Fluss *f* zu 0

    while es gibt einen flussvergrößernden Pfad {

      finde einen kürzesten flussvergrößernden Pfad *p*

      erhöhe *f* entlang *p*

    }

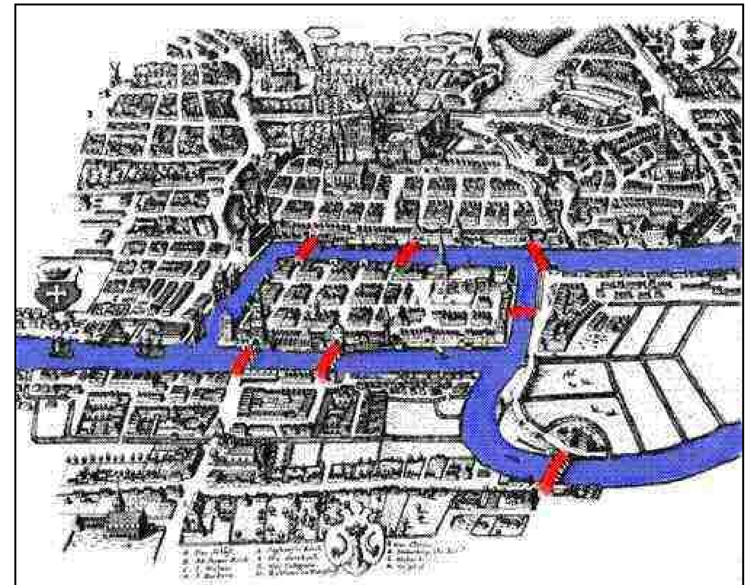
    return *f*

  }

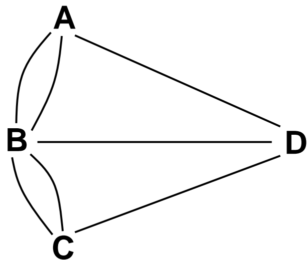
- Laufzeit der Methode ist polynomiell in der Größe des Netzwerks
  - $O(V \cdot |E|^2) = O(|V|^5)$ bei spezieller Implementierung
- Weitere Verbesserungen durch Dinic (1970) führen zu
  - $O(|V|^2 \cdot E) = O(|V|^4)$

# 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?




$$\begin{bmatrix} 0 & 2 & 0 & 1 \\ 2 & 0 & 2 & 1 \\ 0 & 2 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

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