Übersicht

Übersicht

Netzwerk

Objekt bestehend aus

- Elementen und
- Interaktionen bzw. Verbindungen zwischen den Elementen

eher *informales* Konzept

- keine exakte Definition
- Elemente und ihre Verbindungen können ganz unterschiedlichen Charakter haben
- manchmal manifestiert in real existierenden Dingen, manchmal nur gedacht (virtuell)

Beispiele für Netzwerke

- Kommunikationsnetze: Internet, Telefonnetz
- Verkehrsnetze: Straßen-, Schienen-, Flug-, Nahverkehrsnetz
- Versorgungsnetzwerke: Strom, Wasser, Gas, Erdöl
- wirtschaftliche Netzwerke: Geld- und Warenströme, Handel
- biochemische Netzwerke: Metabolische und Interaktionsnetzwerke
- biologische Netzwerke: Gehirn, Ökosysteme
- soziale / berufliche Netzwerke: virtuell oder explizit (Communities)
- Publikationsnetzwerke: Zitationsnetzwerk, Koautor-Netzwerk

Graph

formales / abstraktes Objekt bestehend aus

- Menge von Knoten V (engl. vertices, nodes)
- Menge von Kanten E (engl. edges, lines, links),
  die jeweils ein Paar von Knoten verbinden
- Menge von Eigenschaften der Knoten und / oder Kanten

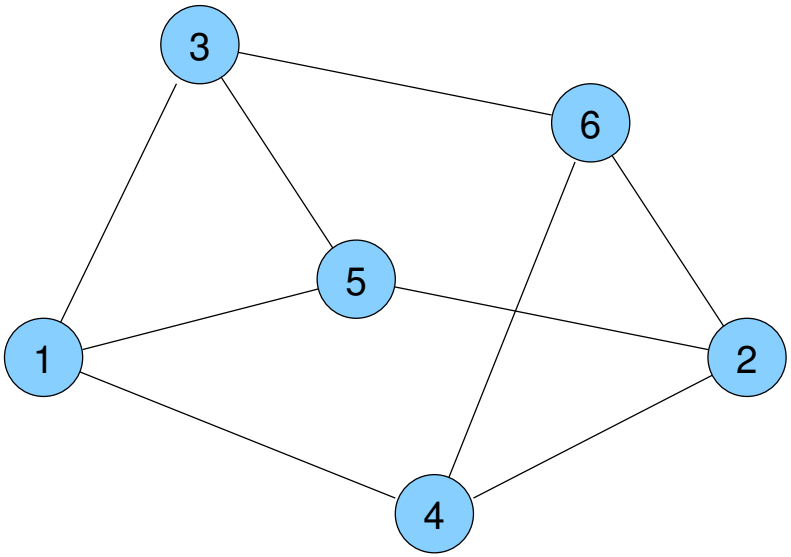
Notation:

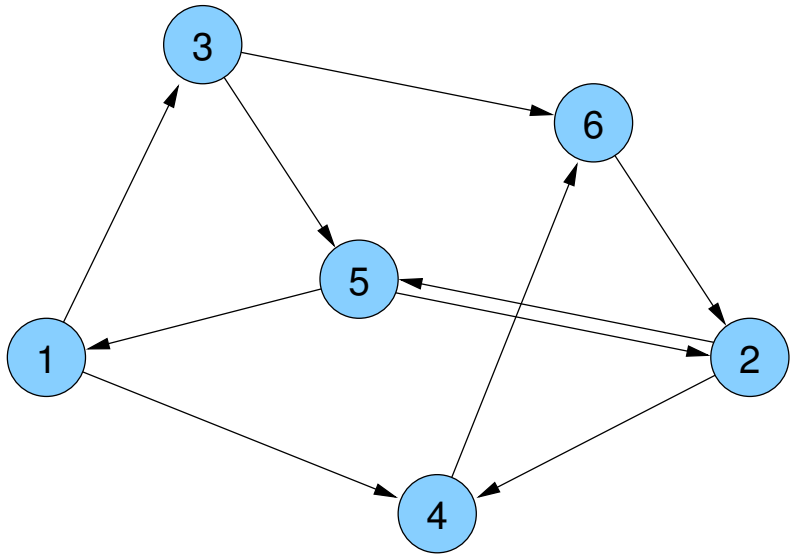
- $G = (V, E)$
  manchmal auch $G = (V, E, w)$ im Fall gewichteter Graphen
- Anzahl der Knoten: $n = |V|$
  Anzahl der Kanten: $m = |E|$

Gerichtete und ungerichtete Graphen

Kanten bzw. Graphen

- ungerichtet: $E \subseteq \{\{v, w\} : v \in V, w \in V\}$
  (ungeordnetes Paar von Knoten bzw. 2-elementige Teilmenge)
- gerichtet: $E \subseteq \{(v, w) : v \in V, w \in V\}$, also $E \subseteq V \times V$
  (geordnetes Paar von Knoten)





Gerichtete und ungerichtete Graphen

Anwendungen:

- Ungerichtete Graphen:

  symmetrische Beziehungen (z.B. $\{v, w\} \in E$ genau dann, wenn Person v und Person w verwandt sind)

- Gerichtete Graphen:

  asymmetrische Beziehungen (z.B. $(v, w) \in E$ genau dann, wenn Person v Person w mag)

  kreisfreie Beziehungen (z.B. $(v, w) \in E$ genau dann, wenn Person v Vorgesetzter von Person w ist

hier:

- Modellierung von ungerichteten durch gerichtete Graphen
- Ersetzung ungerichteter Kanten durch je zwei antiparallele gerichtete Kanten

Nachbarn: Adjazenz, Inzidenz, Grad

- Sind zwei Knoten v und w durch eine Kante e verbunden, dann nennt man
  - v und w adjazent bzw. benachbart
  - v und e inzident (ebenso w und e)

- Anzahl der Nachbarn eines Knotens v: Grad $\deg(v)$

  bei gerichteten Graphen:
  - Eingangsgrad: $\deg^-(v) = |\{(w,v) \in E\}|$
  - Ausgangsgrad: $\deg^+(v) = |\{(v,w) \in E\}|$

Annahmen

- Graph (also Anzahl der Knoten und Kanten) ist endlich
- Graph ist einfach, d.h. E ist eine Menge und keine Multimenge (anderenfalls heißt G Multigraph)
- Graph enthält keine Schleifen (Kanten von v nach v)

Gewichtete Graphen

In Abhängigkeit vom betrachteten Problem wird Kanten und / oder Knoten oft eine Eigenschaft (z.B. eine Farbe oder ein numerischer Wert, das Gewicht) zugeordnet (evt. auch mehrere), z.B.

- Distanzen (in Längen- oder Zeiteinheiten)
- Kosten
- Kapazitäten / Bandbreite
- Ähnlichkeiten
- Verkehrsdichte

Wir nennen den Graphen dann

- knotengewichtet bzw.
- kantengewichtet

Beispiel: $w : E \mapsto \mathbb{R}$
Schreibweise: $w(e)$ für das Gewicht einer Kante $e \in E$

Wege, Pfade und Kreise

- Weg (engl. *walk*) in einem Graphen $G = (V, E)$: alternierende Folge von Knoten und Kanten $x_0, e_1, \ldots, e_k, x_k$, so dass
  - $\forall i \in [0, k] : x_i \in V$ und
  - $\forall i \in [1, k] : e_i = \{x_{i-1}, x_i\}$ bzw. $e_i = (x_{i-1}, x_i) \in E$.
- *Länge* eines Weges: Anzahl der enthaltenen Kanten

- Ein Weg ist ein Pfad, falls er (in sich) kantendisjunkt ist, falls also gilt: $e_i \neq e_j$ für $i \neq j$.
- Ein Pfad ist ein einfacher Pfad, falls er (in sich) knotendisjunkt ist, falls also gilt: $x_i \neq x_j$ für $i \neq j$.

- Ein Weg heißt *Kreis* (engl. *cycle*), falls $x_0 = x_k$.

Operationen

Graph G: Datenstruktur (Typ / Klasse, Variable / Objekt) für Graphen

Node: Datenstruktur für Knoten, Edge: Datenstruktur für Kanten

Operationen:

- G.insert(Edge e): $E := E \cup \{e\}$
- G.remove(Key i, Key j): $E := E \setminus \{e\}$
  für Kante $e = (v, w)$ mit key$(v) = i$ und key$(w) = j$
- G.insert(Node v): $V := V \cup \{v\}$
- G.remove(Key i): sei $v \in V$ der Knoten mit key$(v) = i$
  $V := V \setminus \{v\}$, $E := E \setminus \{(x, y) : x = v \vee y = v\}$
- G.find(Key i): gib Knoten v mit key$(v) = i$ zurück
- G.find(Key i, Key j): gib Kante (v, w) mit key$(v) = i$ und key$(w) = j$ zurück

Operationen

Anzahl der Knoten konstant

$\Rightarrow$ $V = \{0, \ldots, n-1\}$

- (Knotenschlüssel durchnummeriert)

Anzahl der Knoten variabel

- Hashing kann verwendet werden für ein Mapping der n Knoten in den Bereich $\{0, \ldots, \mathcal{O}(n)\}$

$\Rightarrow$ nur konstanter Faktor der Vergrößerung gegenüber statischer Datenstruktur

Übersicht

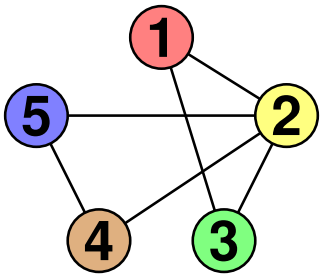
Graphrepräsentation

Darstellung von Graphen im Computer?

Vor- und Nachteile bei z.B. folgenden Fragen:

- Sind zwei gegebene Knoten v und w adjazent?
- Was sind die Nachbarn eines Knotens?
- Welche Knoten sind (direkte oder indirekte) Vorgänger bzw. Nachfolger eines Knotens v in einem gerichteten Graphen?
- Wie aufwendig ist das Einfügen oder Löschen eines Knotens bzw. einer Kante?

Kantenliste



$\{1,2\},\{1,3\},\{2,3\},\{2,4\},\{2,5\},\{4,5\}$

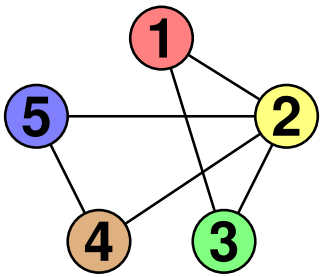
Vorteil:

- Speicherbedarf $\mathcal{O}(m+n)$
- Einfügen von Knoten und Kanten in $\mathcal{O}(1)$
- Löschen von Kanten per Handle in $\mathcal{O}(1)$

Nachteil:

- *G*.find(Key *i*, Key *j*): im worst case $\Theta(m)$
- *G*.remove(Key *i*, Key *j*): im worst case $\Theta(m)$
- Nachbarn nur in $\mathcal{O}(m)$ feststellbar

Adjazenzmatrix



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

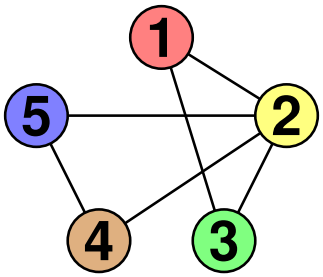
Vorteil:

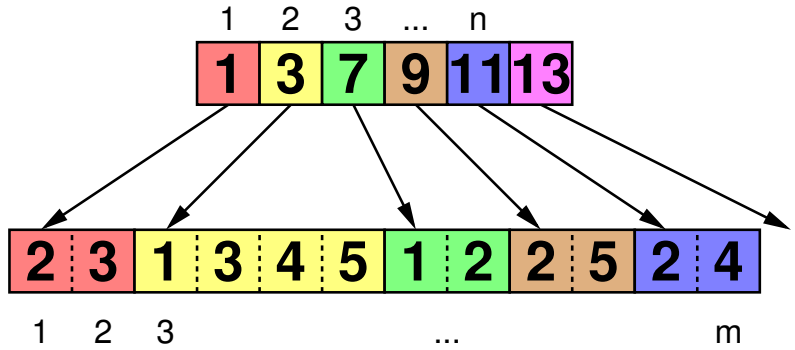
- in $O(1)$ feststellbar, ob zwei Knoten Nachbarn sind
- ebenso Einfügen und Löschen von Kanten

Nachteil:

- kostet $\Theta(n^2)$ Speicher, auch bei Graphen mit $o(n^2)$ Kanten
- Finden aller Nachbarn eines Knotens kostet $O(n)$
- Hinzufügen neuer Knoten ist schwierig

Adjazenzarrays





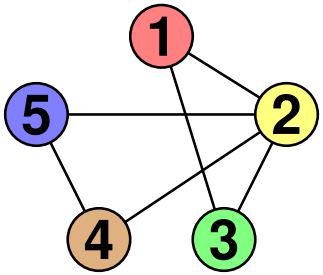
Vorteil:

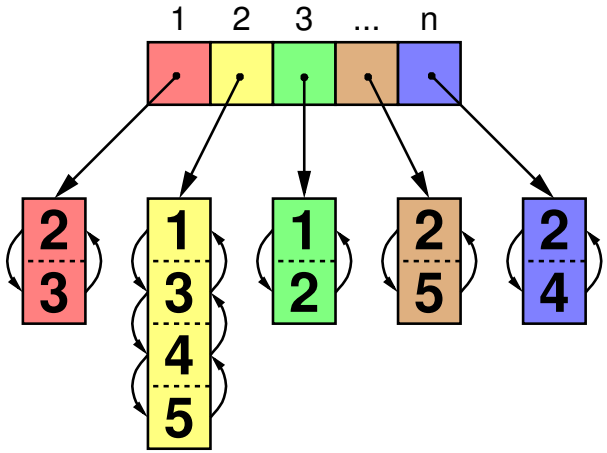
- Speicherbedarf:
  gerichtete Graphen: $n + m + \Theta(1)$
  (hier noch kompakter als Kantenliste mit $2m$)
  ungerichtete Graphen: $n + 2m + \Theta(1)$

Nachteil:

- Einfügen und Löschen von Kanten ist schwierig, deshalb nur für statische Graphen geeignet

Adjazenzlisten





Unterschiedliche Varianten:
einfach/doppelt verkettet, linear/zirkulär

Vorteil:

- Einfügen von Kanten in $\mathcal{O}(d)$ oder $\mathcal{O}(1)$
- Löschen von Kanten in $\mathcal{O}(d)$ (per Handle in $\mathcal{O}(1)$)
- mit unbounded arrays etwas cache-effizienter

Nachteil:

- Zeigerstrukturen verbrauchen relativ viel Platz und Zugriffszeit

Adjazenzliste + Hashtabelle

- speichere Adjazenzliste (Liste von adjazenten Knoten bzw. inzidenten Kanten zu jedem Knoten)
- speichere Hashtabelle, die zwei Knoten auf einen Zeiger abbildet, der dann auf die ggf. vorhandene Kante verweist

Zeitaufwand:

- *G*.find(Key *i*, Key *j*): $\mathcal{O}(1)$ (worst case)
- *G*.insert(Edge *e*): $\mathcal{O}(1)$ (im Mittel)
- *G*.remove(Key *i*, Key *j*): $\mathcal{O}(1)$ (im Mittel)

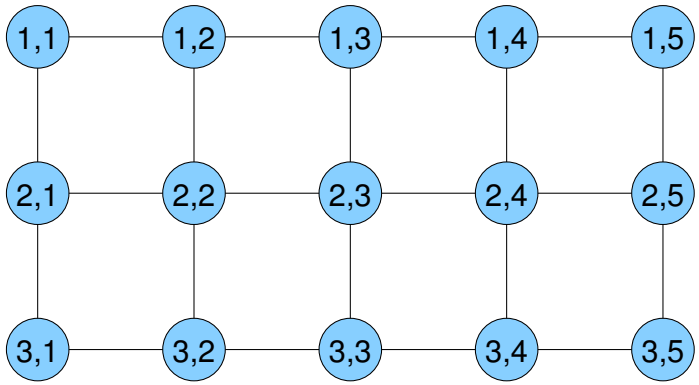
Speicheraufwand: $\mathcal{O}(n+m)$

Implizite Repräsentation

Beispiel: Gitter-Graph (grid graph)

- definiert durch zwei Parameter k und ℓ

$$
\begin{aligned}
V &= [1,\ldots,k] \times [1,\ldots,\ell] \\
E &= \big\{((i,j),(i,j')) \in V^2 : |j-j'| = 1\big\} \cup \\
&\quad \big\{((i,j),(i',j)) \in V^2 : |i-i'| = 1\big\}
\end{aligned}
$$



- Kantengewichte könnten in 2 zweidimensionalen Arrays gespeichert werden:
  eins für waagerechte und eins für senkrechte Kanten

Übersicht

Graphtraversierung

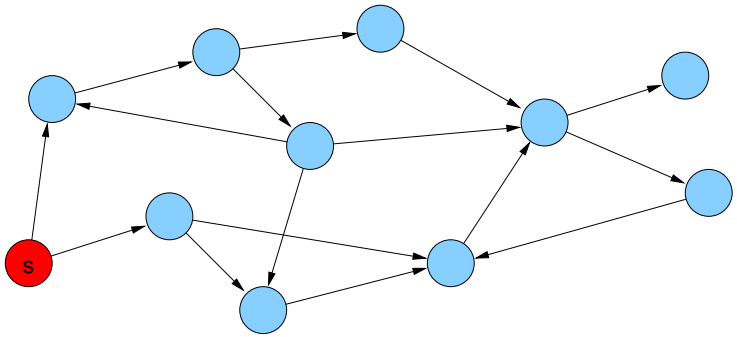
Problem:

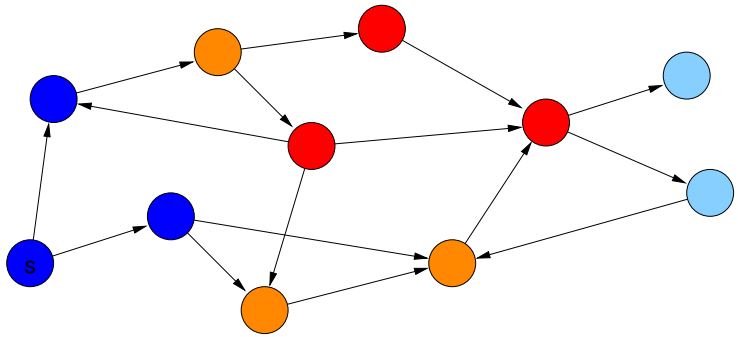
Wie kann man die Knoten eines Graphen systematisch durchlaufen?

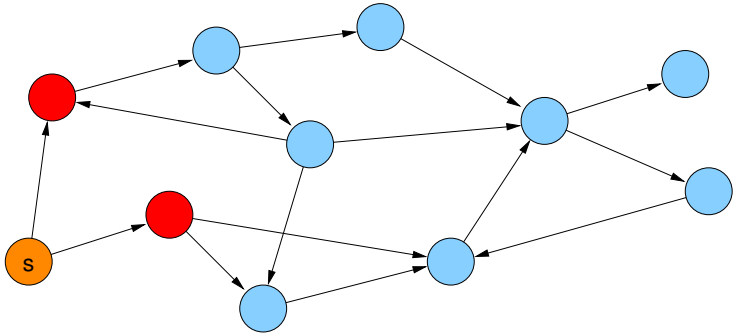
Grundlegende Strategien:

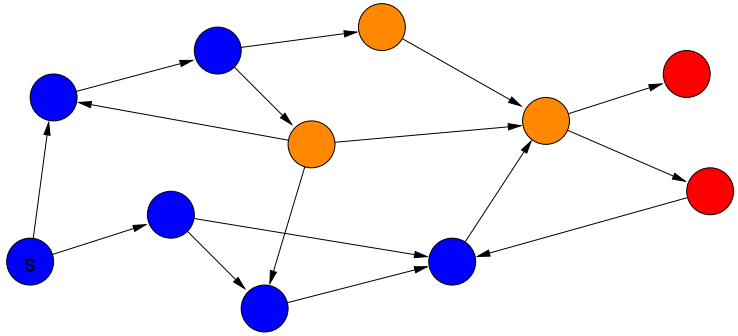
- Breitensuche (breadth-first search, BFS)
- Tiefensuche (depth-first search, DFS)

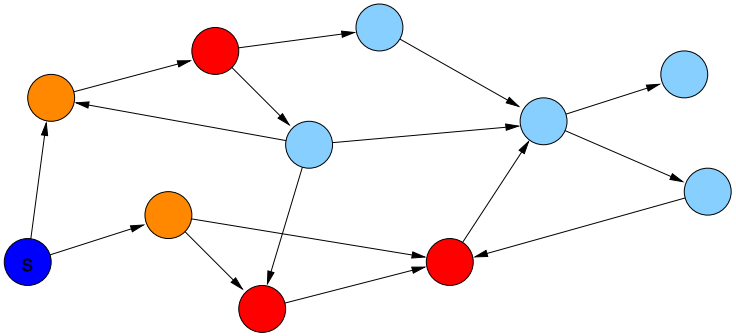
Breitensuche

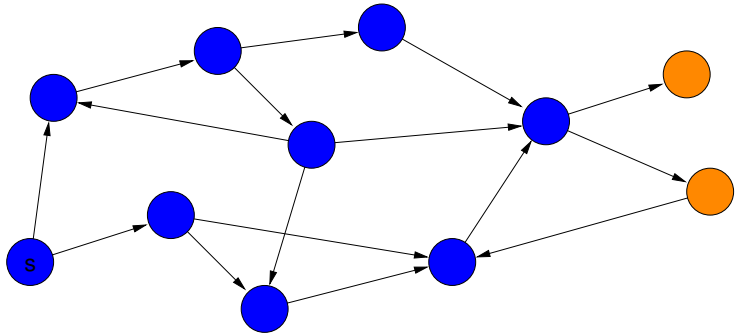






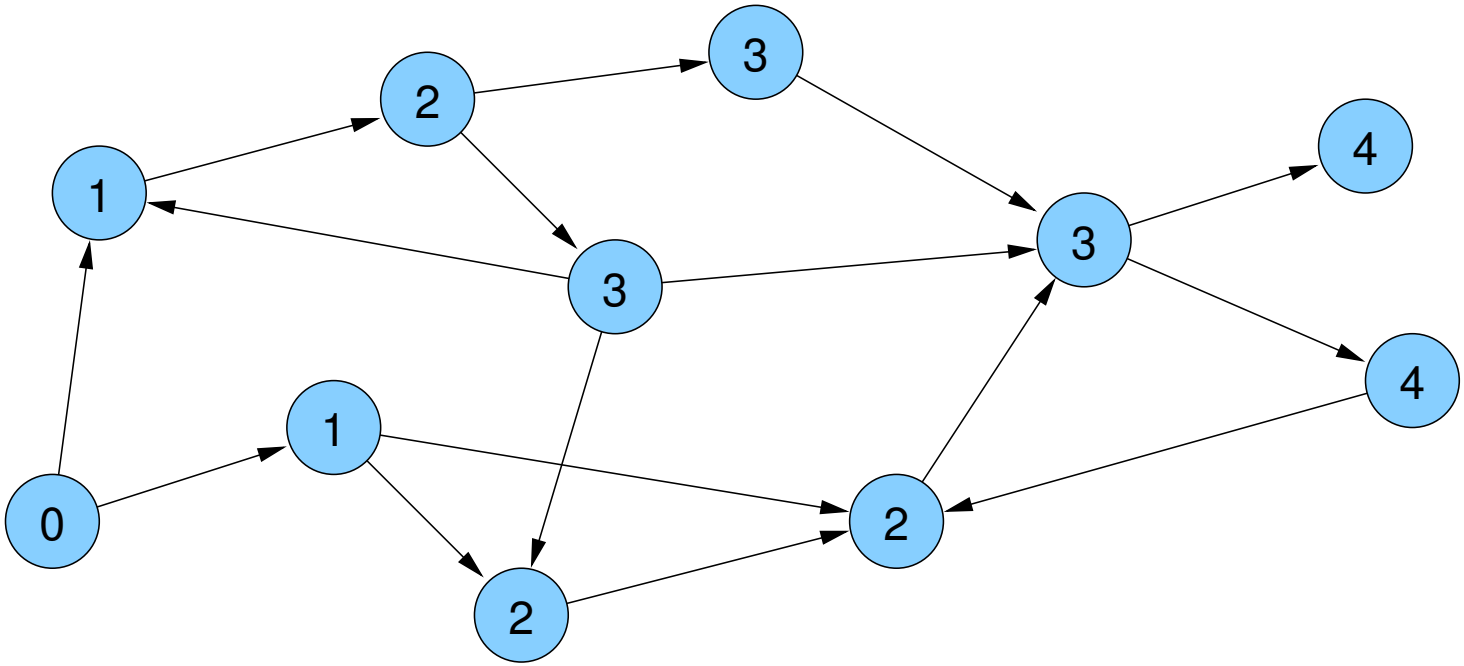






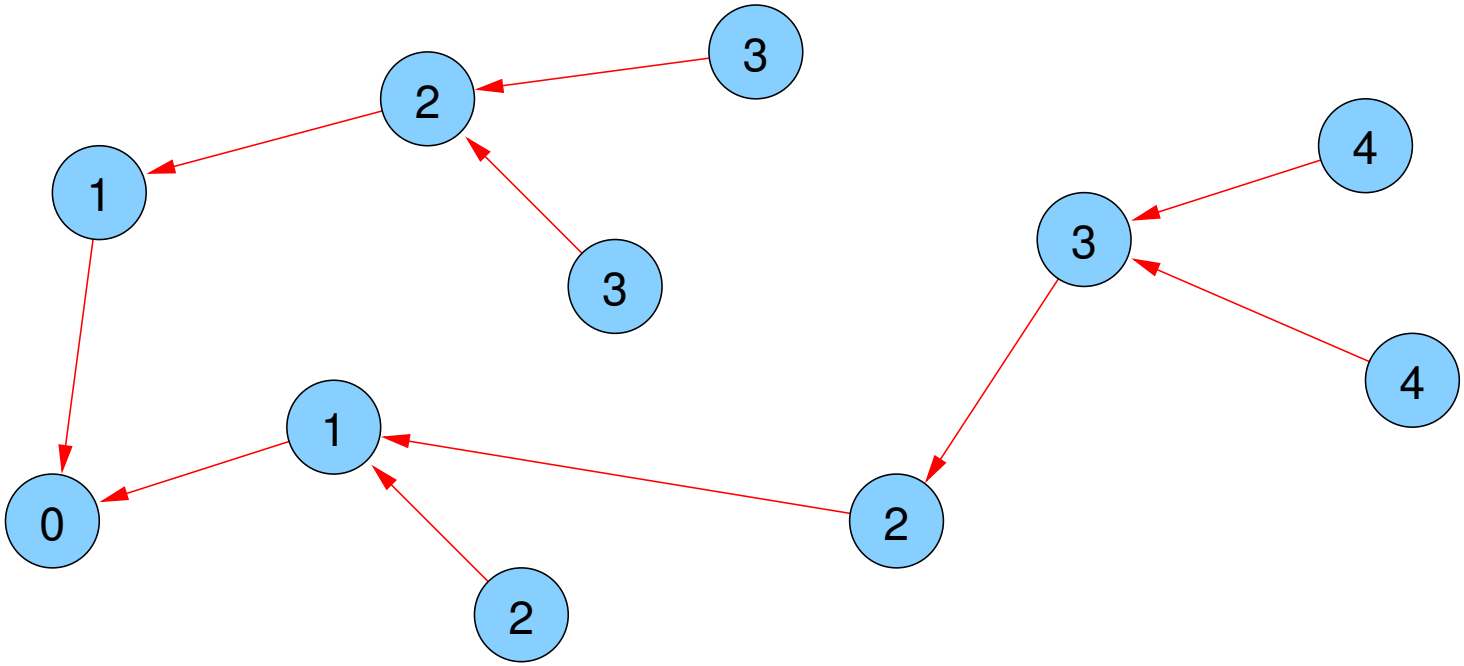
Breitensuche

- $d(v)$: Distanz von Knoten v zu s ($d(s) = 0$)



Breitensuche

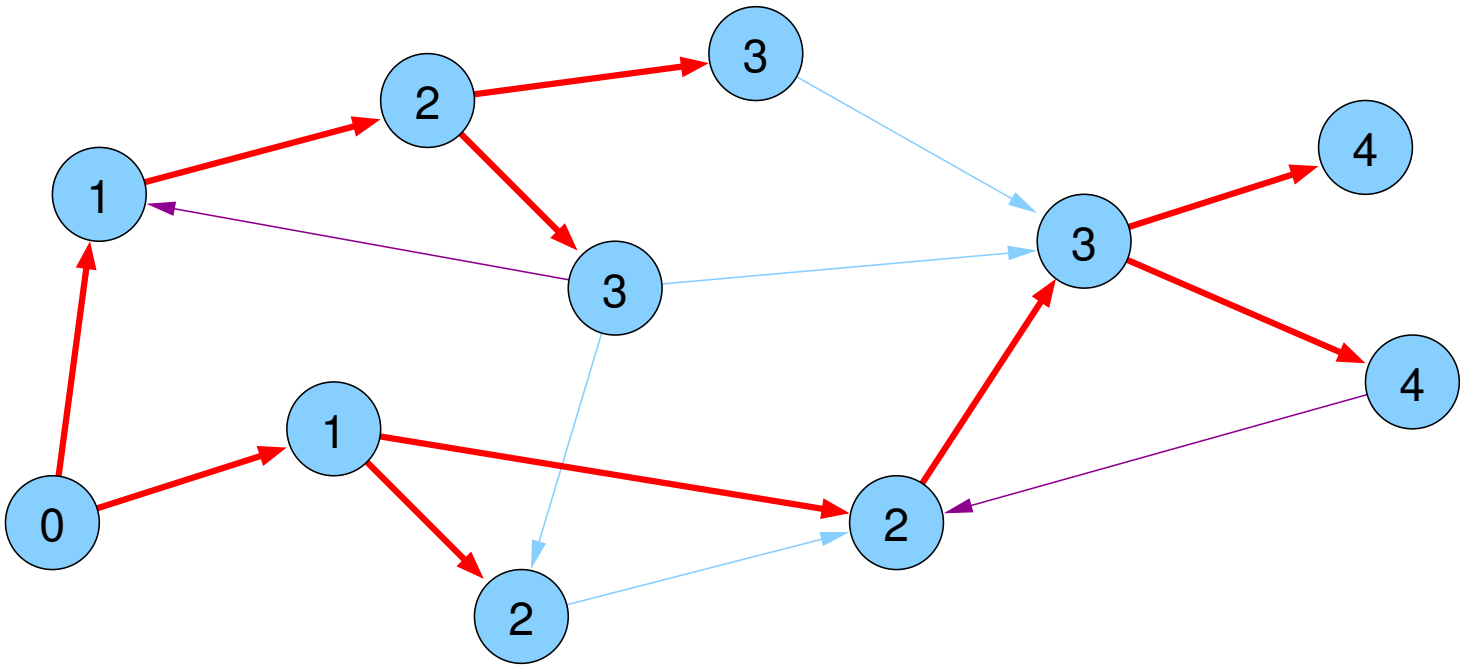
- parent(v): Knoten, von dem v entdeckt wurde
- parent wird beim ersten Besuch von v gesetzt ($\Rightarrow$ eindeutig)



Breitensuche

Kantentypen:

- Baumkanten: zum Kind
- Rückwärtskanten: zu einem Vorfahren
- Kreuzkanten: sonstige

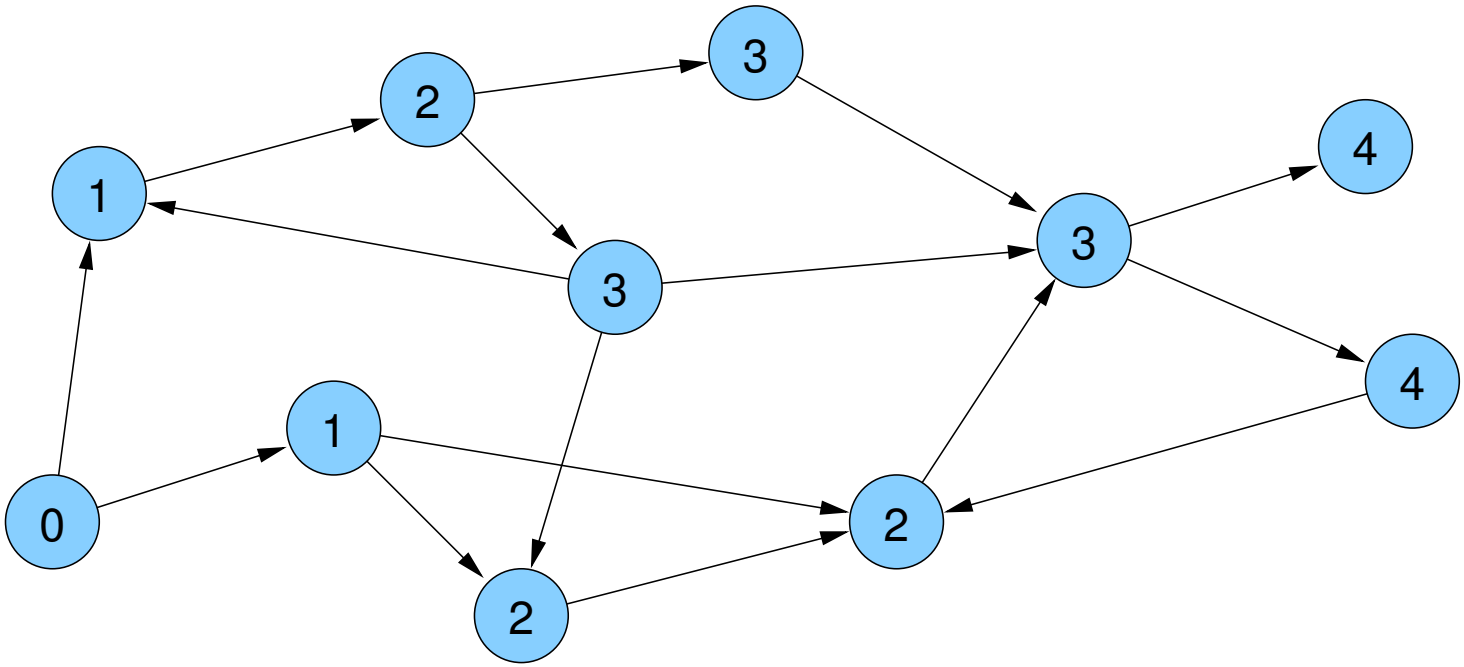


Breitensuche

```
BFS(Node s) {
    d[s] = 0;
    parent[s] = s;
    List<Node> q = ⟨s⟩;
    while ( !q.empty() ) {
        u = q.popFront();
        foreach ((u, v) ∈ E) {
            if (parent[v] == null) {
                q.pushBack(v);
                d[v] = d[u]+1;
                parent[v] = u;
            }
        }
    }
}
```

Breitensuche

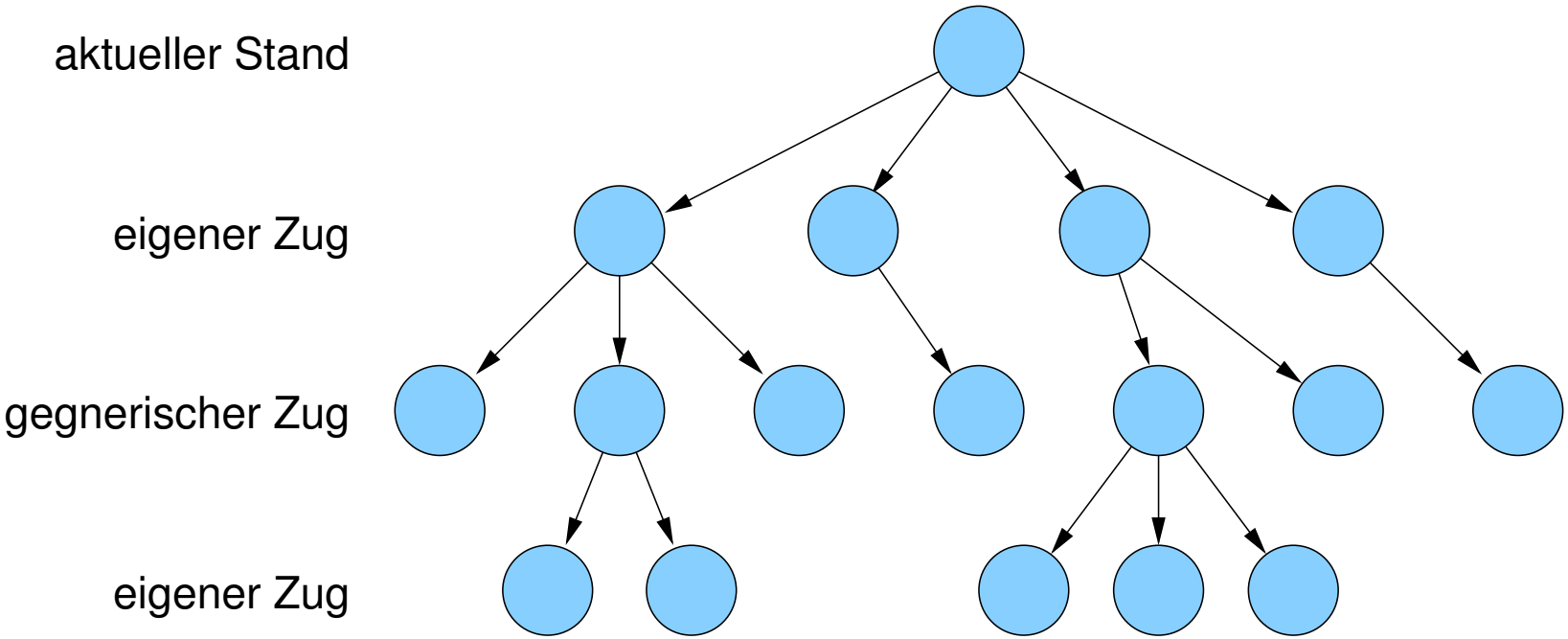
Anwendung: Single Source Shortest Path (SSSP) Problem
in ungewichteten Graphen



Breitensuche

Anwendung: Bestimmung des nächsten Zugs bei Spielen

Exploration des Spielbaums



Problem: halte Aufwand zur Suche eines guten Zuges in Grenzen

Breitensuche

Anwendung: Bestimmung des nächsten Zugs bei Spielen

- Standard-BFS: verwendet FIFO-Queue

  ebenenweise Erkundung

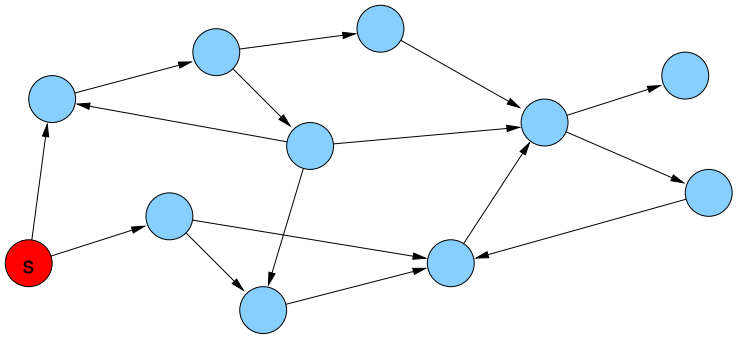
  aber: zu teuer!

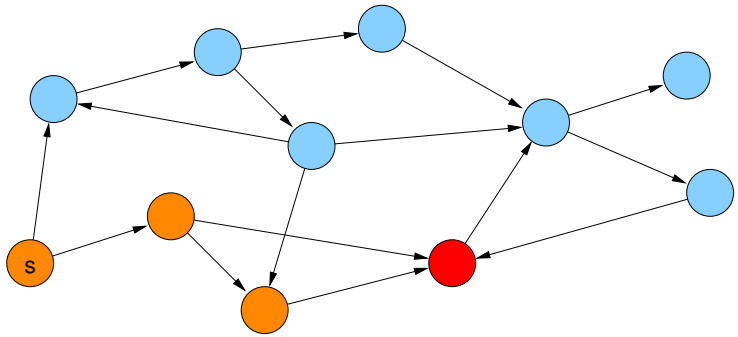
- Best-First Search: verwendet Priority Queue

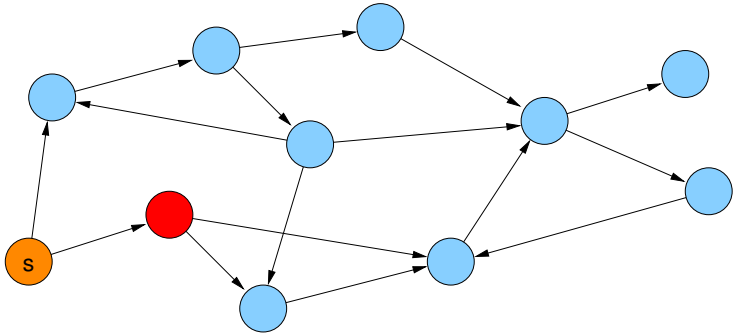
  (z.B. realisiert durch binären Heap)

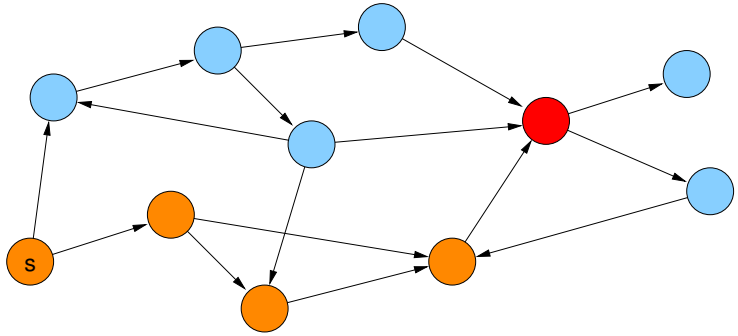
  Priorität eines Knotens wird durch eine Güte-Heuristik des repräsentierten Spielzustands gegeben

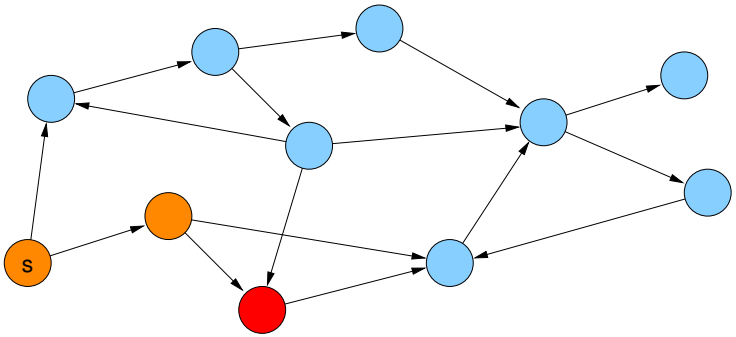
Tiefensuche

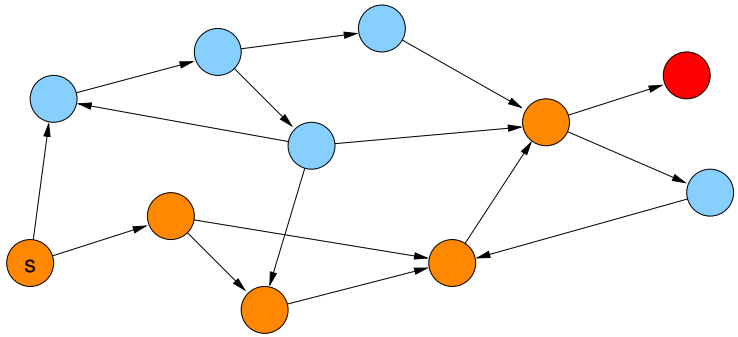




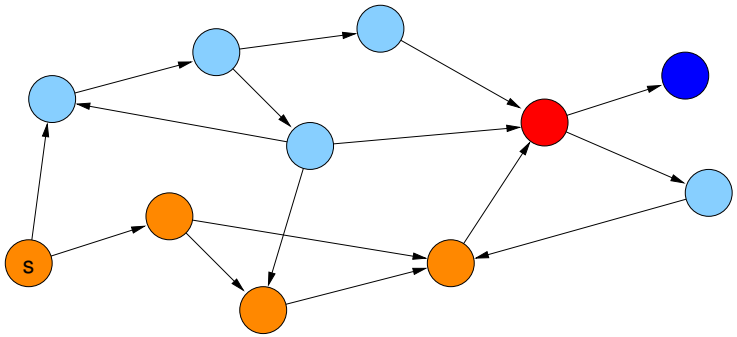


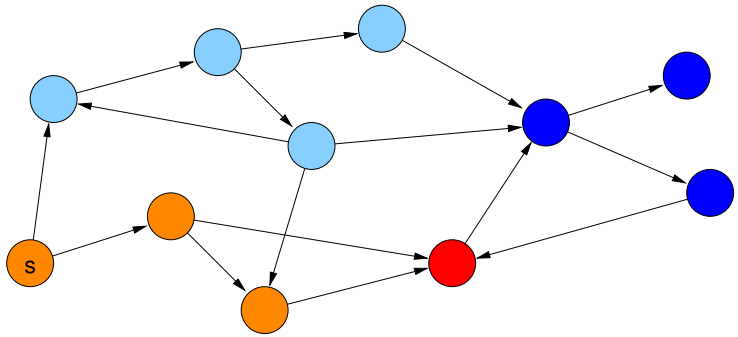


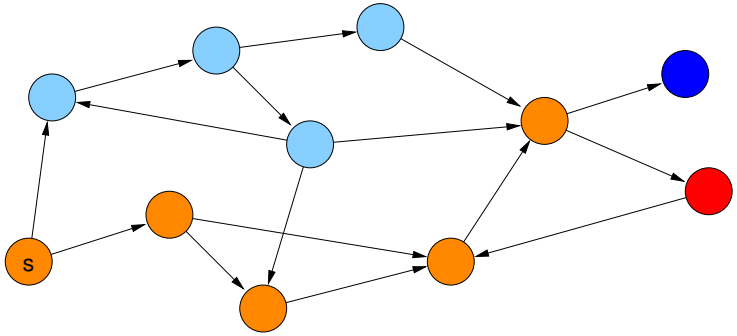


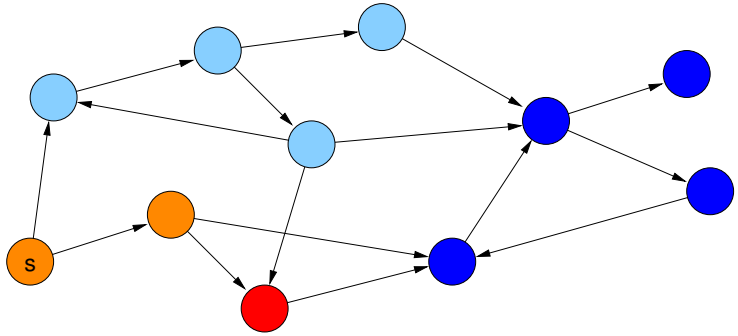


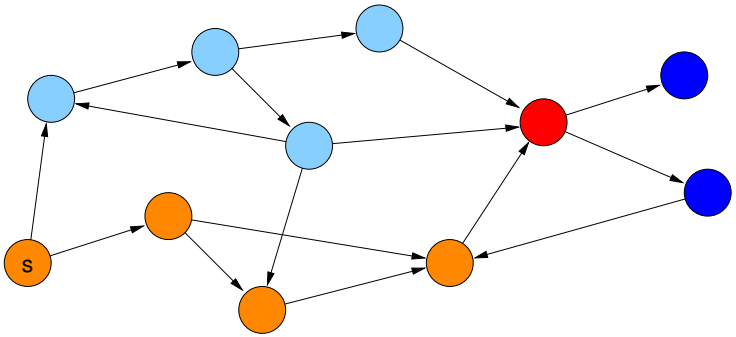
Tiefensuche

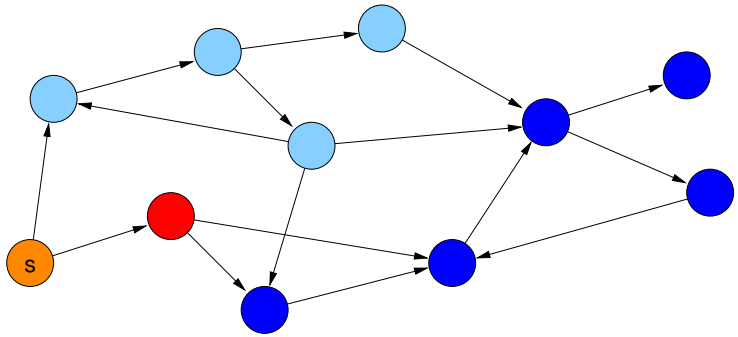




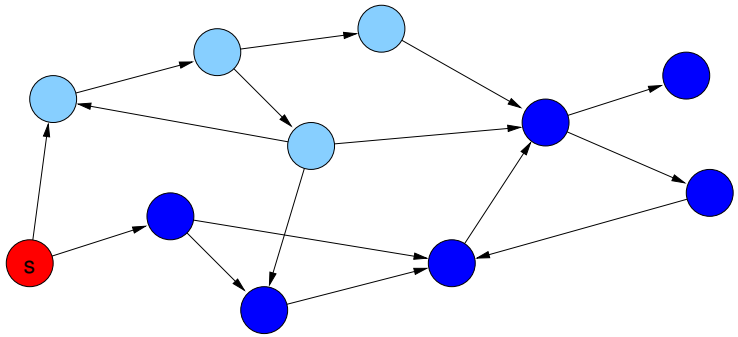


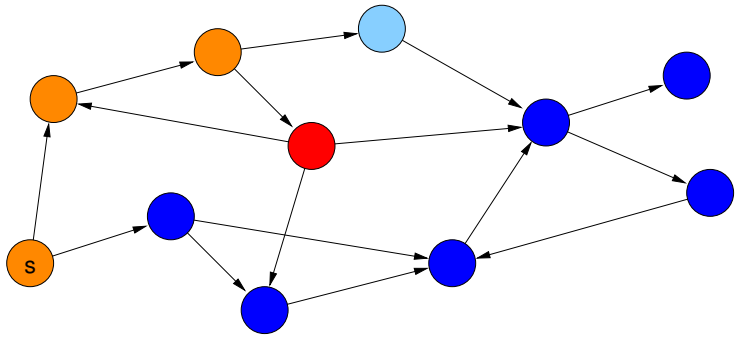


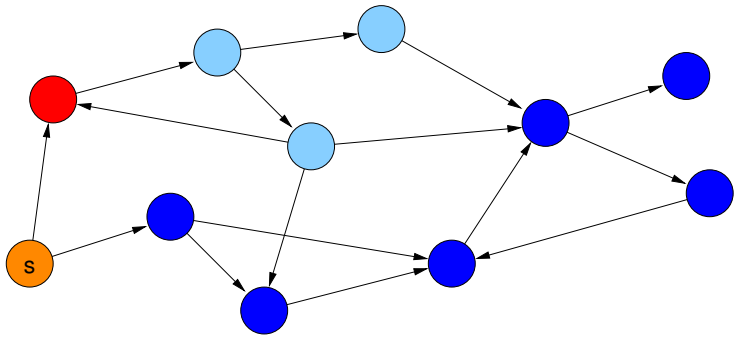


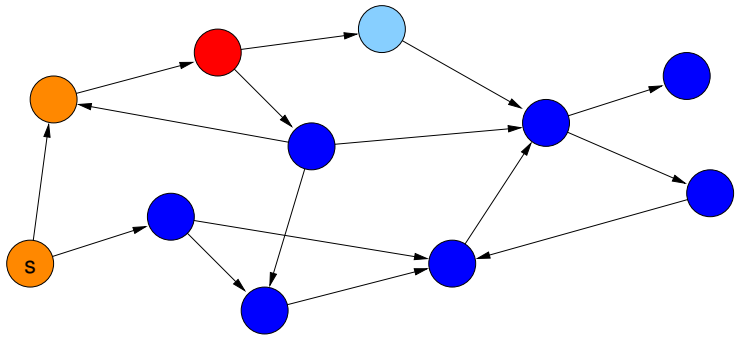


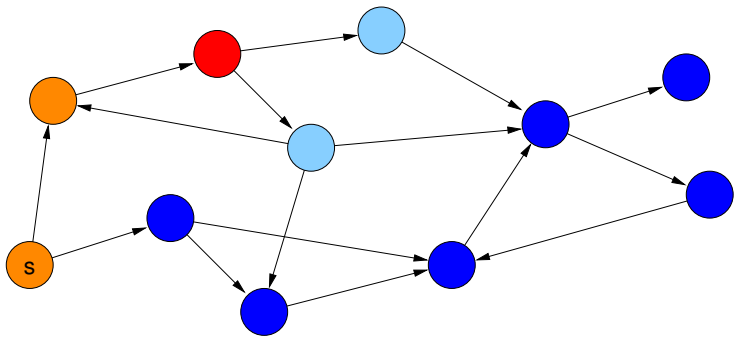
Tiefensuche

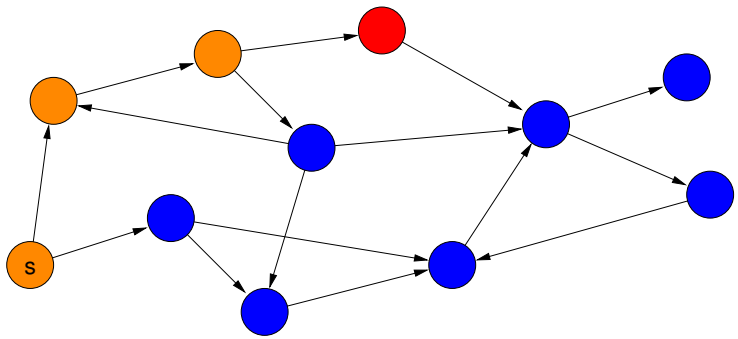




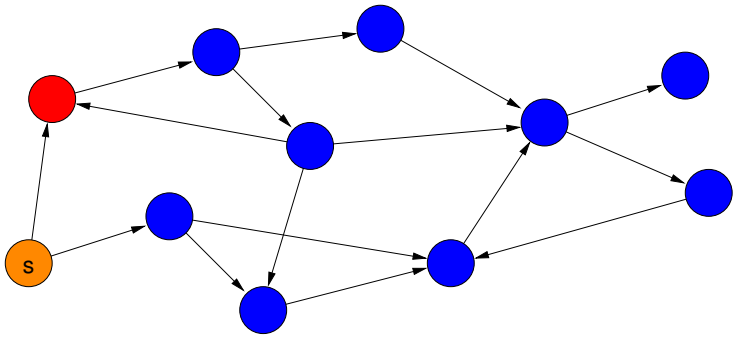


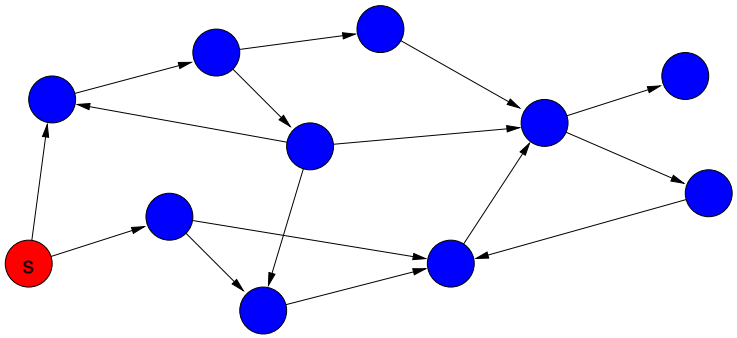


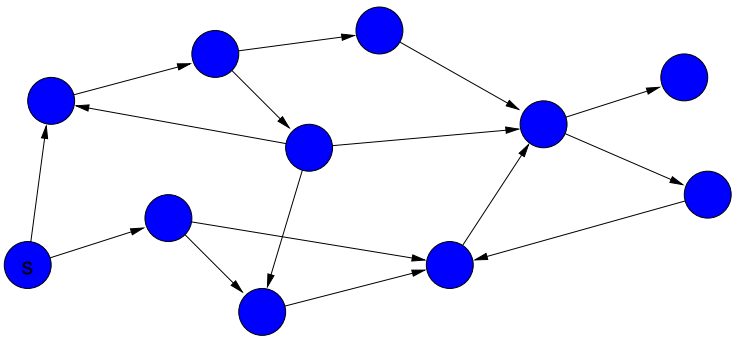




Tiefensuche







Tiefensuche

Übergeordnete Methode:

```
foreach (v ∈ V)
   Setze v auf nicht markiert;

init();
foreach (s ∈ V)
   if (s nicht markiert) {
      markiere s;
      root(s);
      DFS(s,s);
   }
```

```
DFS(Node u, Node v) {
   foreach ((v, w) ∈ E)
      if (w ist markiert)
         traverseNonTreeEdge(v,w);
      else {
         traverseTreeEdge(v,w);
         markiere w;
         DFS(v,w);
      }
   backtrack(u,v);
}
```

Tiefensuche

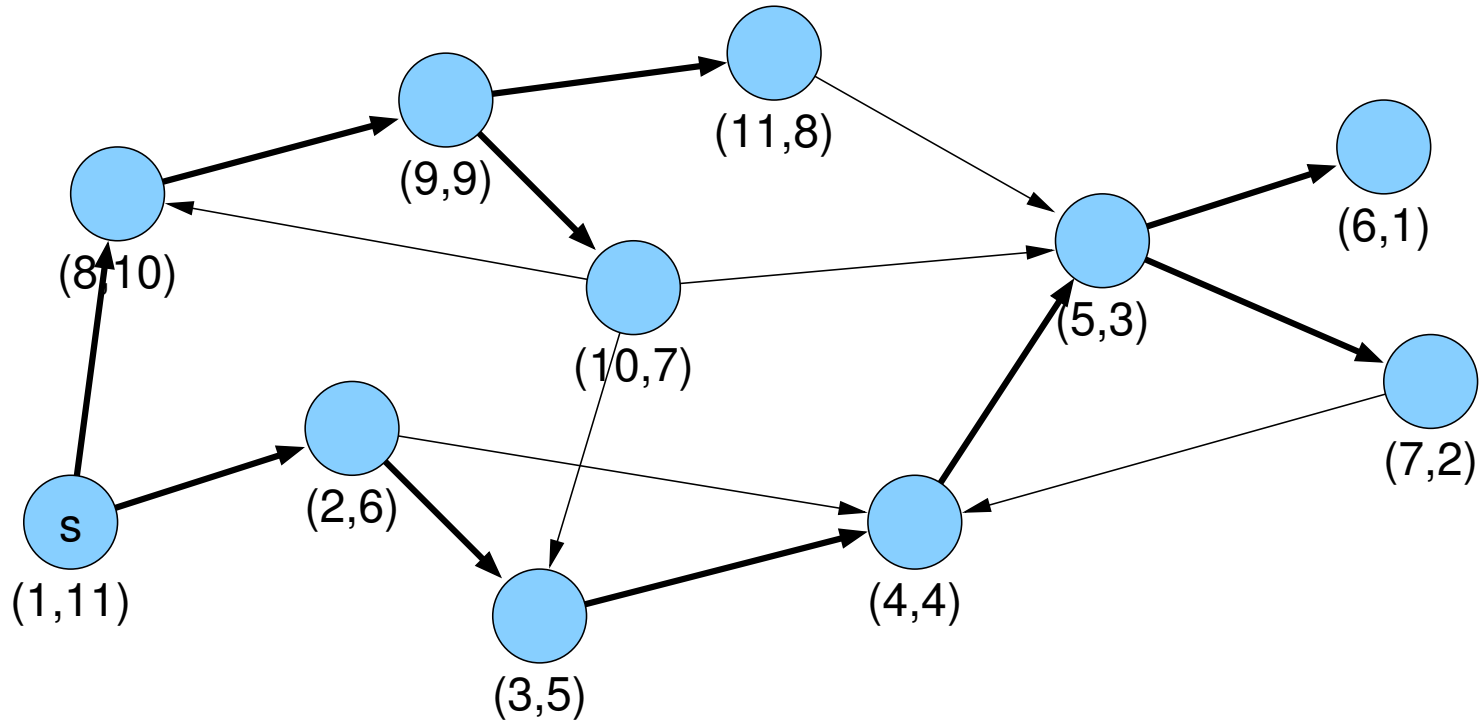
Variablen:

- int[ ] dfsNum; // Explorationsreihenfolge
- int[ ] finishNum; // Fertigstellungsreihenfolge
- int dfsCount, finishCount; // Zähler

Methoden:

- init() { dfsCount = 1; finishCount = 1; }
- root(Node s) { dfsNum[s] = dfsCount; dfsCount++; }
- traverseTreeEdge(Node v, Node w)
  { dfsNum[w] = dfsCount; dfsCount++; }
- traverseNonTreeEdge(Node v, Node w) { }
- backtrack(Node u, Node v)
  { finishNum[v] = finishCount; finishCount++; }

Tiefensuche



DFS-Nummerierung

Beobachtung:

- Knoten im DFS-Rekursionsstack (aktiven Knoten) sind bezüglich dfsNum aufsteigend sortiert

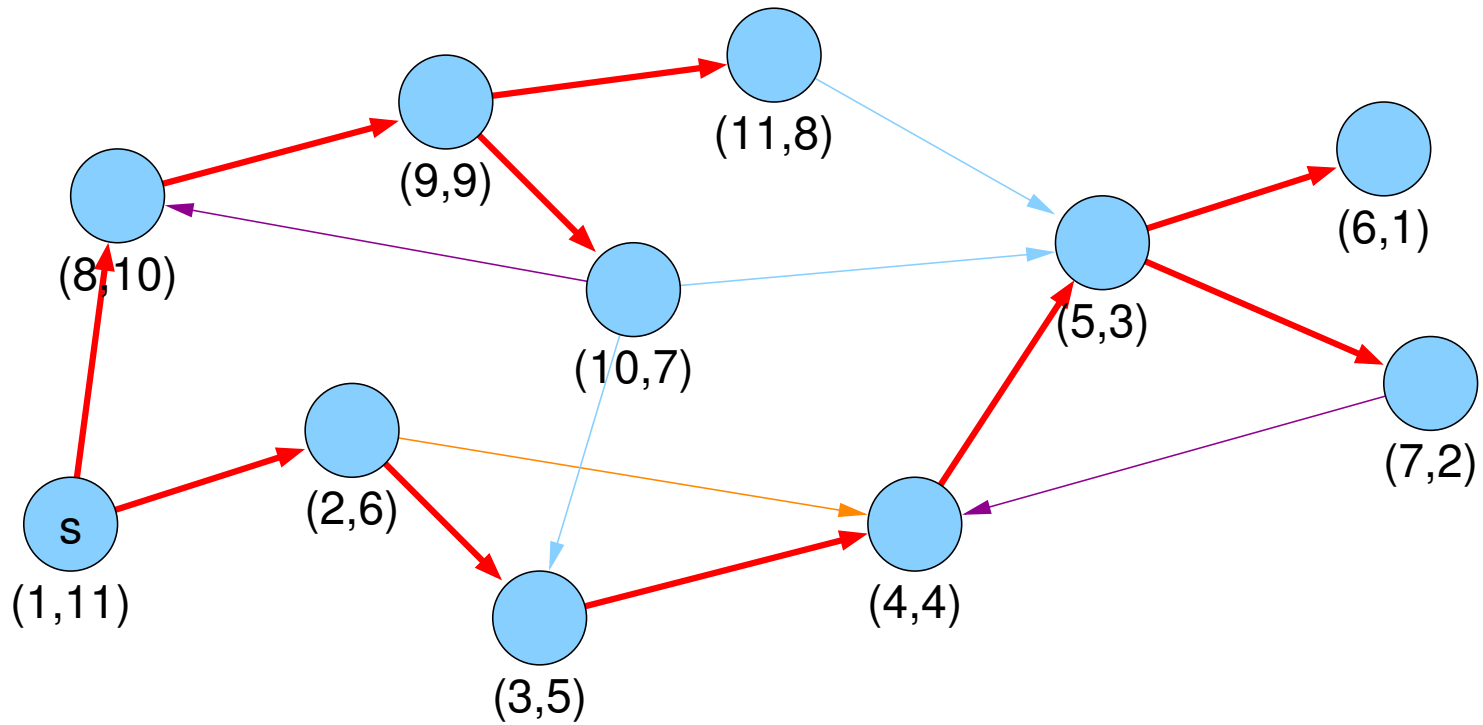
Begründung:

- dfsCount wird nach jeder Zuweisung von dfsNum inkrementiert
- neue aktive Knoten haben also immer die höchste dfsNum

DFS-Nummerierung

Kantentypen:

- Baumkanten: zum Kind
- Vorwärtskanten: zu einem Nachfahren
- Rückwärtskanten: zu einem Vorfahren
- Kreuzkanten: sonstige



DFS-Nummerierung

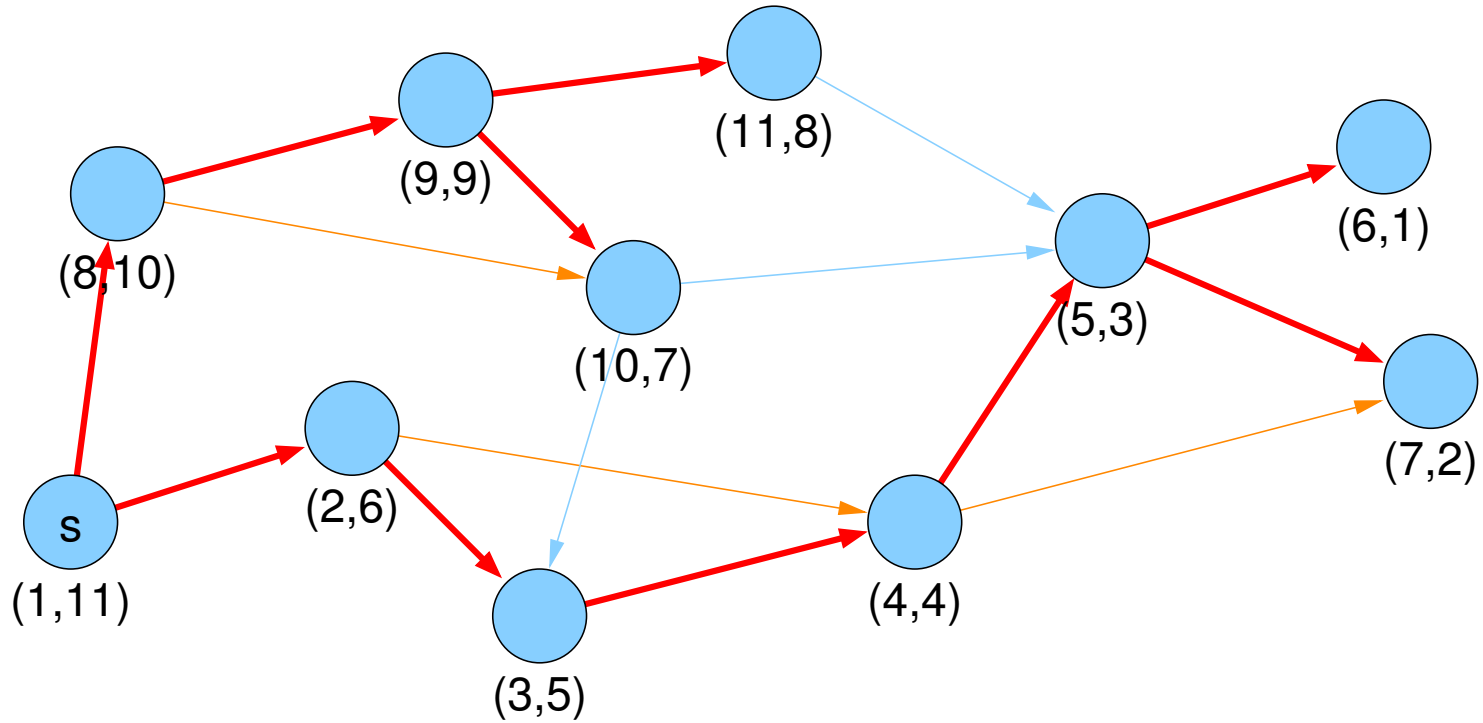
Beobachtung für Kante (v, w):

Kantentyp	dfsNum[v] < dfsNum[w]	finishNum[v] > finishNum[w]
Baum & Vorwärts	ja	ja
Rückwärts	nein	nein (umgekehrt)
Kreuz	nein	ja

DAG-Erkennung per DFS

Anwendung:

- Erkennung von azyklischen gerichteten Graphen (engl. directed acyclic graph / DAG)



- keine gerichteten Kreise

DAG-Erkennung per DFS

Lemma

Folgende Aussagen sind äquivalent:

1. *Graph G ist ein DAG.*
2. *DFS in G enthält keine Rückwärtskante.*
3. $\forall (v, w) \in E : finishNum[v] > finishNum[w]$

Beweis.

- (2)⇒(3): Wenn (2), dann gibt es nur Baum-, Vorwärts- und Kreuz-Kanten. Für alle gilt (3).
- (3)⇒(2): Für Rückwärtskanten gilt sogar die umgekehrte Relation finishNum[v]<finishNum[w].
  Wenn (3), dann kann es also keine Rückwärtskanten geben (2).

...

DAG-Erkennung per DFS

Lemma

Folgende Aussagen sind äquivalent:

1. *Graph G ist ein DAG.*
2. *DFS in G enthält keine Rückwärtskante.*
3. $\forall (v,w) \in E : finishNum[v] > finishNum[w]$

Beweis.

- $\neg(2) \Rightarrow \neg(1)$: Wenn Rückwärtskante (v,w) existiert, gibt es einen gerichteten Kreis ab Knoten w (und G ist kein DAG).
- $\neg(1) \Rightarrow \neg(2)$: Wenn es einen gerichteten Kreis gibt, ist mindestens eine von der DFS besuchte Kante dieses Kreises eine Rückwärtskante (Kante zu einem schon besuchten Knoten, dieser muss Vorfahr sein).

□

Zusammenhang in Graphen

Definition

Ein ungerichteter Graph heißt zusammenhängend, wenn es von jedem Knoten einen Pfad zu jedem anderen Knoten gibt.

Ein maximaler zusammenhängender induzierter Teilgraph wird als Zusammenhangskomponente bezeichnet.

Die Zusammenhangskomponenten eines ungerichteten Graphen können mit DFS oder BFS in $\mathcal{O}(n+m)$ bestimmt werden.

Knoten-Zusammenhang

Definition

Ein ungerichteter Graph $G = (V, E)$ heißt k-fach zusammenhängend (oder genauer gesagt k-*knoten*zusammenhängend), falls

- $|V| > k$ und
- für jede echte Knotenteilmenge $X \subset V$ mit $|X| < k$ der Graph $G - X$ zusammenhängend ist.

Bemerkung:

- "zusammenhängend" ist im wesentlichen gleichbedeutend mit "1-knotenzusammenhängend"

  Ausnahme: Graph mit nur einem Knoten ist zusammenhängend, aber nicht 1-zusammenhängend

Artikulationsknoten und Blöcke

Definition

Ein Knoten v eines Graphen G heißt Artikulationsknoten (engl. *cut-vertex*), wenn sich die Anzahl der Zusammenhangskomponenten von G durch das Entfernen von v erhöht.

Definition

Die Zweifachzusammenhangskomponenten eines Graphen sind die maximalen Teilgraphen, die 2-fach zusammenhängend sind.

Ein Block ist ein maximaler zusammenhängender Teilgraph, der keinen Artikulationsknoten enthält.

Die Menge der Blöcke besteht aus den Zweifachzusammenhangskomponenten, den Brücken (engl. *cut edges*), sowie den isolierten Knoten.

Blöcke und DFS

Modifizierte DFS nach R. E. Tarjan:

- `num[v]`: DFS-Nummer von v
- `low[v]`: minimale Nummer `num[w]` eines Knotens w, der von v aus über beliebig viele (≥ 0) Baumkanten (abwärts), evt. gefolgt von einer einzigen Rückwärtskante (aufwärts) erreicht werden kann

- `low[v]`: Minimum von
  - `num[v]`
  - `low[w]`, wobei w ein Kind von v im DFS-Baum ist (Baumkante)
  - `num[w]`, wobei $\{v, w\}$ eine Rückwärtskante ist

Artikulationsknoten und DFS

Lemma

Sei $G = (V, E)$ ein ungerichteter, zusammenhängender Graph und T ein DFS-Baum in G.

Ein Knoten $a \in V$ ist genau dann ein Artikulationsknoten, wenn

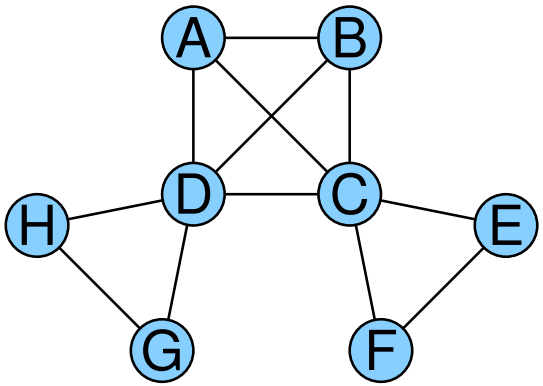
- *a die Wurzel von T ist und mindestens 2 Kinder hat, oder*
- *a nicht die Wurzel von T ist und es ein Kind b von a mit $low[b] \geq num[a]$ gibt.*

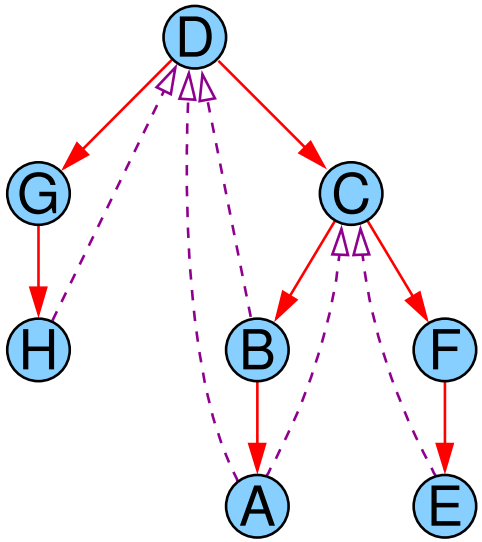
Beweisidee

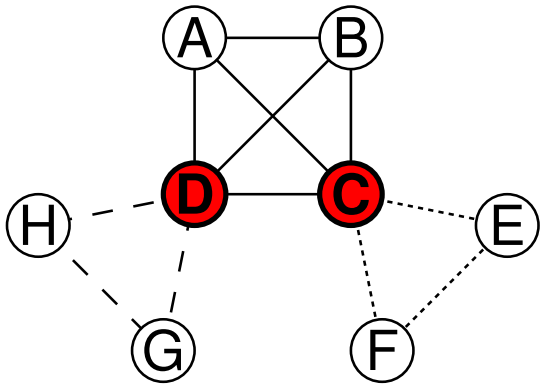
Der Algorithmus beruht auf der Tatsache, dass in Zweifach(knoten)zusammenhangskomponenten zwischen jedem Knotenpaar mindestens zwei (knoten-)disjunkte Wege existieren. Das entspricht einem Kreis.

Artikulationsknoten und Blöcke per DFS

- bei Aufruf der DFS für Knoten v wird `num[`v`]` bestimmt und `low[`v`]` mit `num[`v`]` initialisiert
- nach Besuch eines Nachbarknotens w: Update von `low[`v`]` durch Vergleich mit
  - `low[`w`]` nach Rückkehr vom rekursiven Aufruf, falls (v, w) eine Baumkante war
  - `num[`w`]`, falls (v, w) eine Rückwärtskante war

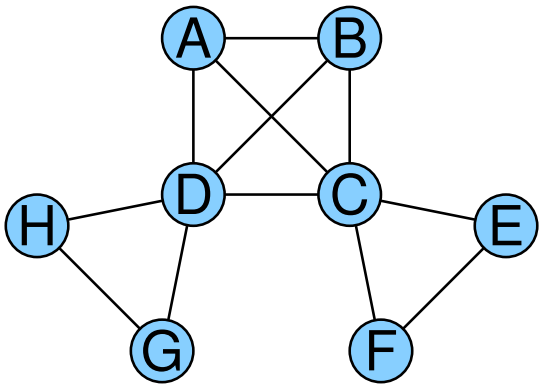


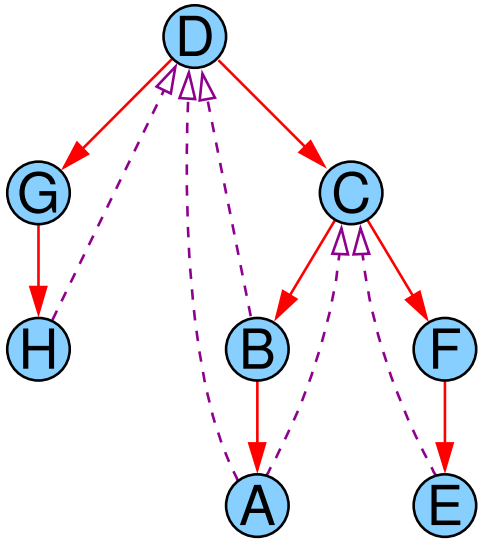


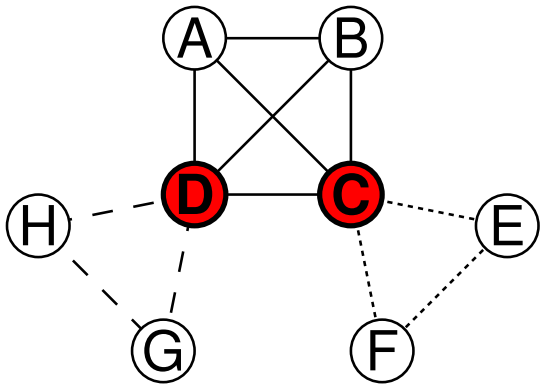


Artikulationsknoten und Blöcke per DFS

- Kanten werden auf einem anfangs leeren Stack gesammelt
- Rückwärtskanten kommen direkt auf den Stack (ohne rek. Aufruf)
- Baumkanten kommen vor dem rekursiven Aufruf auf den Stack
- nach Rückkehr von einem rekursiven Aufruf werden im Fall $\texttt{low}[w] \geq \texttt{num}[v]$ die obersten Kanten vom Stack bis einschließlich der Baumkante $\{v, w\}$ entfernt und bilden den nächsten Block







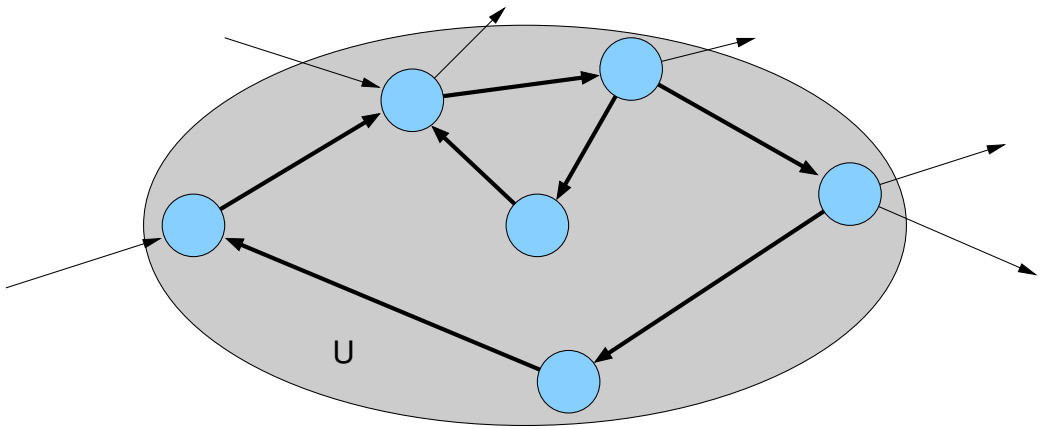
Starke Zusammenhangskomponenten

Definition

Sei $G = (V, E)$ ein gerichteter Graph.

Knotenteilmenge $U \subseteq V$ heißt stark zusammenhängend genau dann, wenn für alle $u, v \in U$ ein gerichteter Pfad von u nach v in G existiert.

Für Knotenteilmenge $U \subseteq V$ heißt der induzierte Teilgraph $G[U]$ starke Zusammenhangskomponente von G, wenn U stark zusammenhängend und (inklusions-)maximal ist.



Starke Zusammenhangskomponenten

Beobachtungen:

- Knoten $x, y \in V$ sind stark zusammenhängend, falls beide Knoten auf einem gemeinsamen gerichteten Kreis liegen (oder $x = y$).

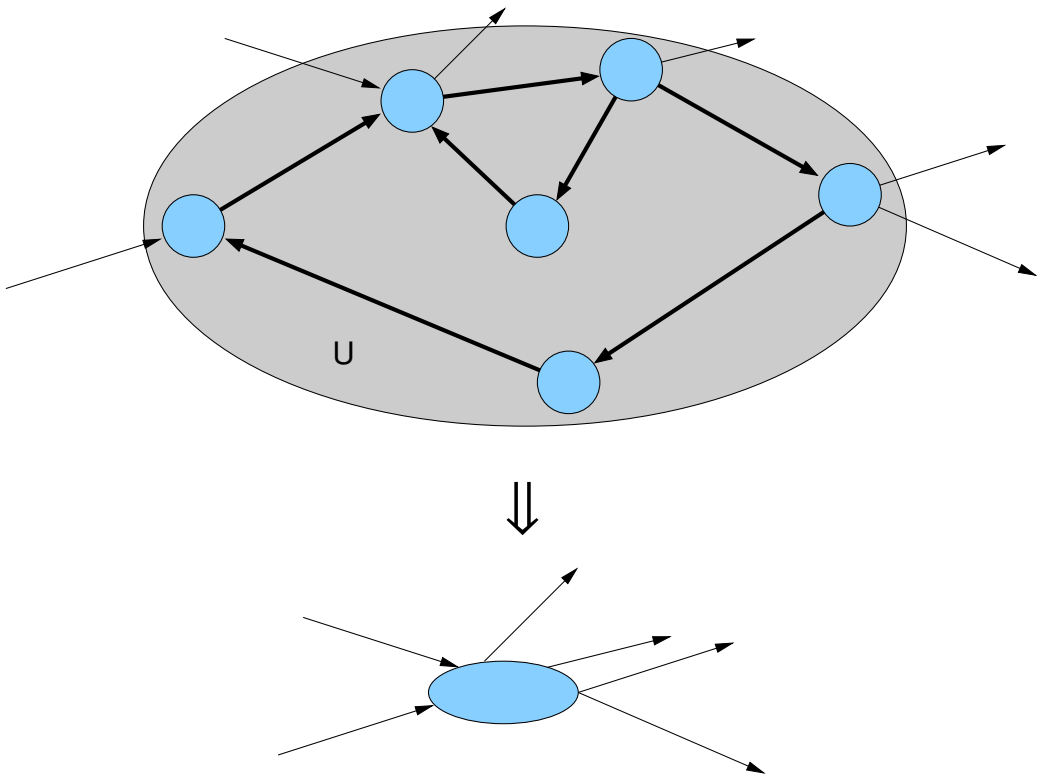
- Die starken Zusammenhangskomponenten bilden eine Partition der Knotenmenge.

  (im Gegensatz zu 2-Zhk. bei ungerichteten Graphen, wo nur die Kantenmenge partitioniert wird, sich aber zwei verschiedene 2-Zhk. in einem Knoten überlappen können)

Starke Zusammenhangskomponenten

Beobachtungen:

- Schrumpft man alle starken Zusammenhangskomponenten zu einzelnen (Super-)Knoten, ergibt sich ein DAG.



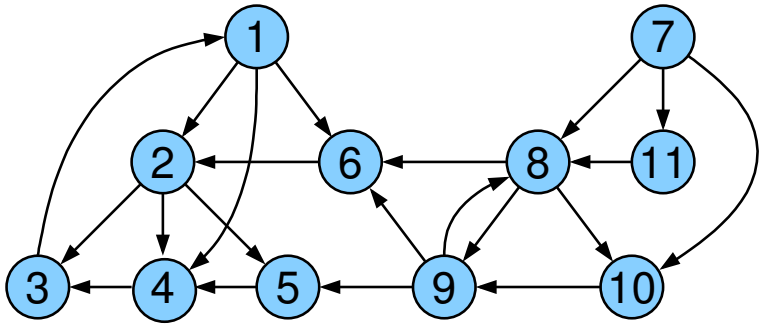
Starke Zhk. und DFS

Idee:

- beginne mit Graph ohne Kanten, jeder Knoten ist eigene SCC
- füge nach und nach einzelne Kanten ein

$\Rightarrow$ aktueller (current) Graph $G_c = (V, E_c)$

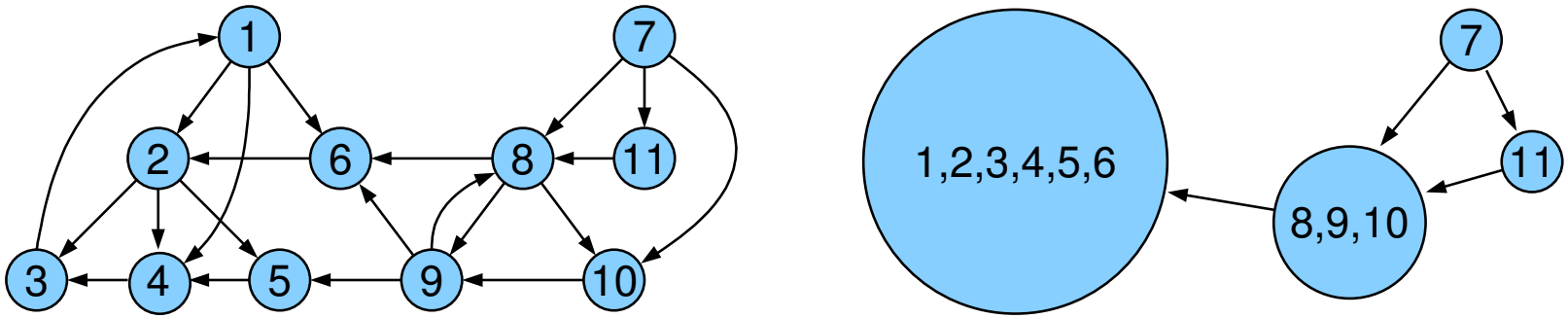
- Update der starken Zusammenhangskomponenten (SCCs)



Starke Zhk. und DFS

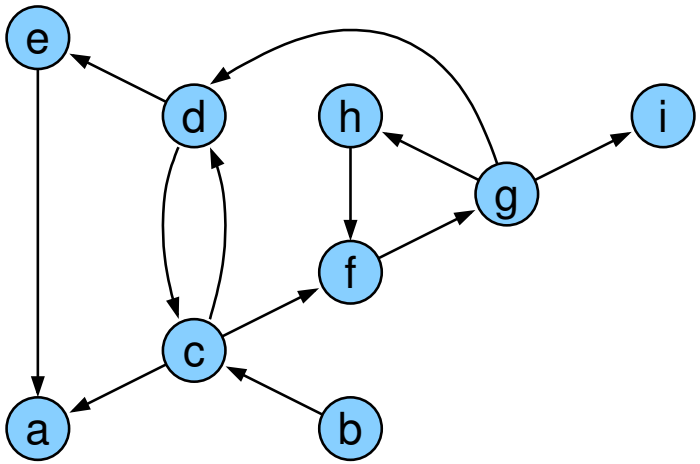
Idee:

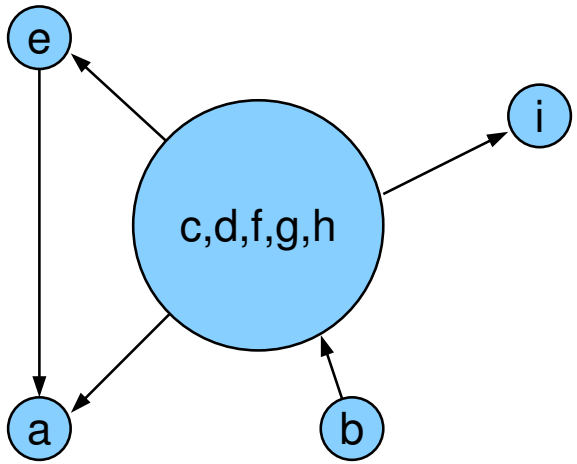
- betrachte geschrumpften (shrunken) Graph G_c^s:
  Knoten entsprechen SCCs von G_c, Kante (C, D) genau dann, wenn es Knoten $u \in C$ und $v \in D$ mit $(u, v) \in E_c$ gibt
- geschrumpfter Graph G_c^s ist ein DAG
- Ziel: Aktualisierung des geschrumpften Graphen beim Einfügen



Starke Zhk. und DFS

Geschrumpfter Graph
(Beispiel aus Mehlhorn / Sanders)





Starke Zhk. und DFS

Update des geschrumpften Graphen nach Einfügen einer Kante:

3 Möglichkeiten:

- beide Endpunkte gehören zu derselben SCC

$\Rightarrow$ geschrumpfter Graph unverändert

- Kante verbindet Knoten aus zwei verschiedenen SCCs, aber schließt keinen Kreis

$\Rightarrow$ SCCs im geschrumpften Graph unverändert, aber eine Kante wird im geschrumpften Graph eingefügt (falls nicht schon vorhanden)

- Kante verbindet Knoten aus zwei verschiedenen SCCs und schließt einen oder mehrere Kreise

$\Rightarrow$ alle SCCs, die auf einem der Kreise liegen, werden zu einer einzigen SCC verschmolzen

Starke Zhk. und DFS

Prinzip:

- Tiefensuche
  - V_c schon markierte (entdeckte) Knoten
  - E_c schon gefundene Kanten
- 3 Arten von SCC: unentdeckt, offen, geschlossen
- unentdeckte Knoten haben Ein- / Ausgangsgrad Null in G_c

$\Rightarrow$ zunächst bildet jeder Knoten eine eigene unentdeckte SCC, andere SCCs enthalten nur markierte Knoten

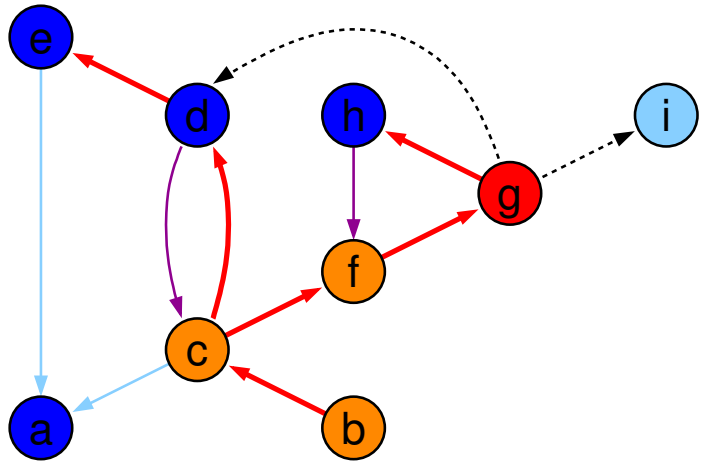
- SCCs mit mindestens einem aktiven Knoten (ohne finishNum) heißen offen
- SCC heißt geschlossen, falls sie nur fertige Knoten (mit finishNum) enthält
- Knoten in offenen / geschlossenen SCCs heißen offen / geschlossen

Starke Zhk. und DFS

- Knoten in geschlossenen SCCs sind immer fertig (mit finishNum)
- Knoten in offenen SCCs können fertig oder noch aktiv (ohne finishNum) sein
- Repräsentant einer SCC: Knoten mit kleinster dfsNum

Starke Zhk. und DFS

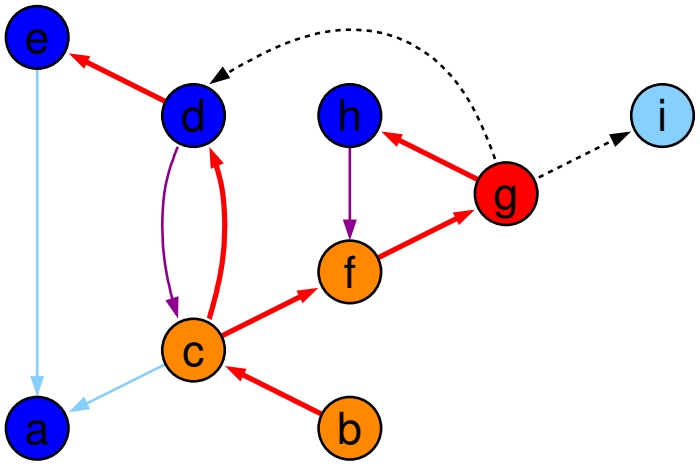
DFS-Snapshot:

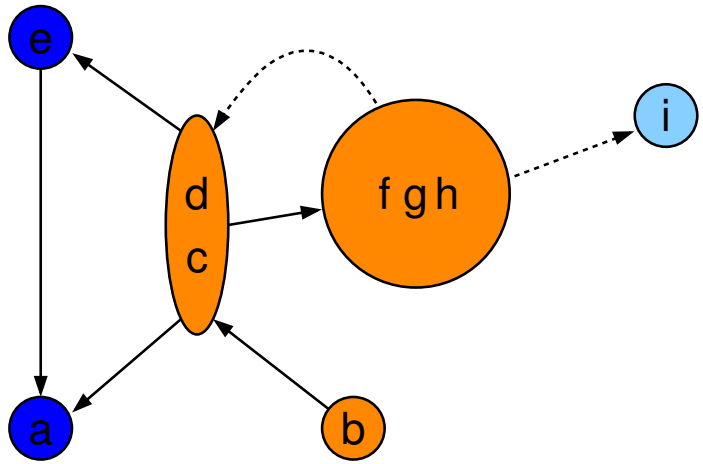


- erste DFS startete bei Knoten a, zweite bei b
- aktueller Knoten ist g, auf dem Rekursionsstack liegen b, c, f, g
- (g, d) und (g, i) wurden noch nicht exploriert
- (d, c) und (h, f) sind Rückwärtskanten
- (c, a) und (e, a) sind Querkanten
- (b, c), (c, d), (d, e), (c, f), (f, g) und (g, h) sind Baumkanten

Starke Zhk. und DFS

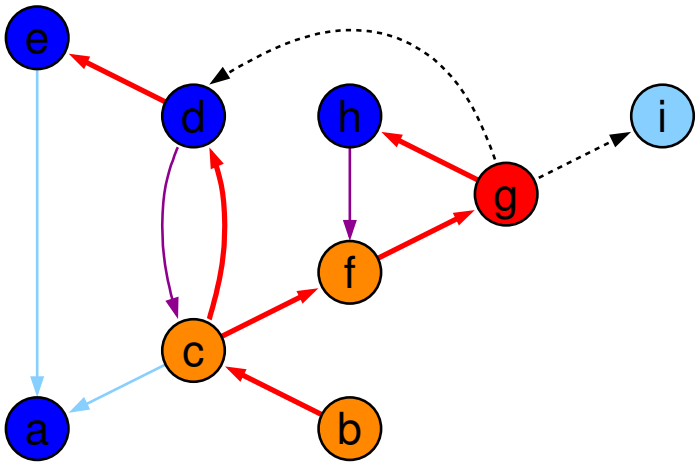
DFS-Snapshot mit geschrumpftem Graph:

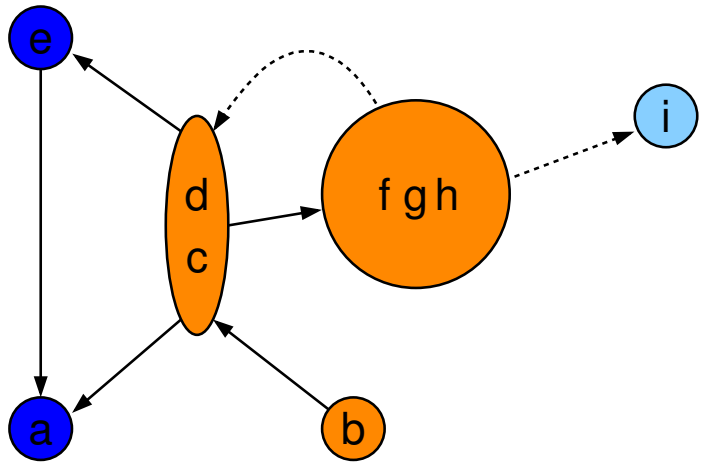




- unentdeckt: $\{i\}$ offen: $\{b\}$, $\{c, d\}$, $\{f, g, h\}$ geschlossen: $\{a\}$, $\{e\}$
- offene SCCs bilden Pfad im geschrumpften Graph
- aktueller Knoten gehört zur letzten SCC
- offene Knoten wurden in Reihenfolge b, c, d, f, g, h erreicht und werden von den Repräsentanten b, c und f genau in die offenen SCCs partitioniert

Starke Zhk. und DFS





Beobachtungen (Invarianten für G_c):

1. Pfade aus geschlossenen SCCs führen immer zu geschlossenen SCCs
2. Pfad zum aktuellen Knoten enthält die Repräsentanten aller offenen SCCs

   offene Komponenten bilden Pfad im geschrumpften Graph
3. Knoten der offenen SCCs in Reihenfolge der DFS-Nummern werden durch Repräsentanten in die offenen SCCs partitioniert

Starke Zhk. und DFS

Geschlossene SCCs von G_c sind auch SCCs in G:

- Sei v geschlossener Knoten und S / S_c seine SCC in G / G_c.
- zu zeigen: $S = S_c$
- G_c ist Subgraph von G, also $S_c \subseteq S$
- somit zu zeigen: $S \subseteq S_c$

- Sei w ein Knoten in S.

$\Rightarrow$ $\exists$ Kreis C durch v und w.

- Invariante 1: alle Knoten von C sind geschlossen und somit erledigt (alle ausgehenden Kanten exploriert)
- C ist in G_c enthalten, also $w \in S_c$
- damit gilt $S \subseteq S_c$, also $S = S_c$

Starke Zhk. und DFS

Vorgehen:

- Invarianten 2 und 3 helfen bei Verwaltung der offenen SCCs
- Knoten in offenen SCCs auf Stack oNodes
  (in Reihenfolge steigender dfsNum)
- Repräsentanten der offenen SCCs auf Stack oReps
- zu Beginn Invarianten gültig (alles leer)
- vor Markierung einer neuen Wurzel sind alle markierten Knoten erledigt, also keine offenen SCCs, beide Stacks leer
  dann: neue offene SCC für neue Wurzel s,
  s kommt auf beide Stacks

Starke Zhk. und DFS

Prinzip: betrachte Kante $e = (v, w)$

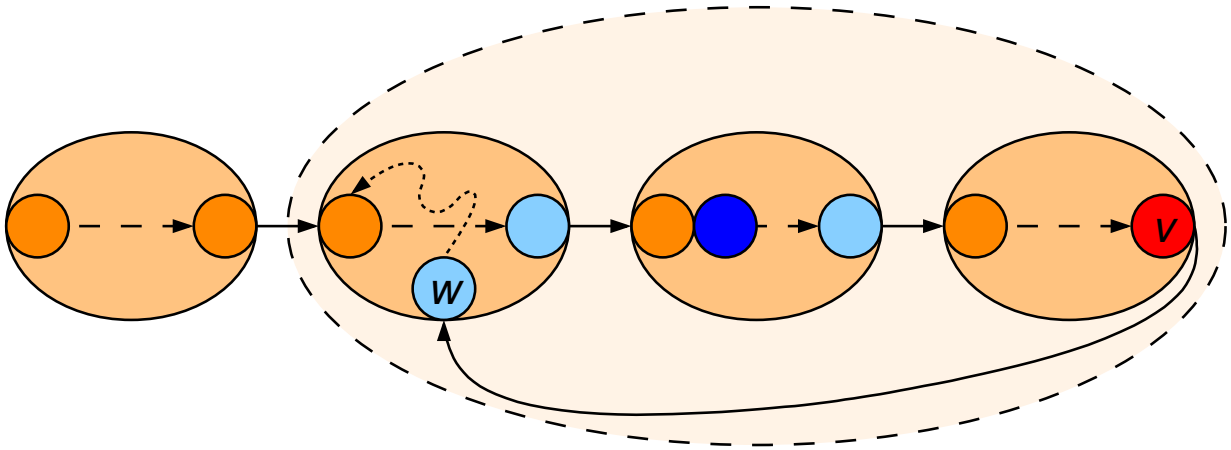
- Kante zu unbekanntem Knoten w (Baumkante):
  neue eigene offene SCC für w (w kommt auf oNodes und oReps)
- Kante zu Knoten w in geschlossener SCC (Nicht-Baumkante):
  von w gibt es keinen Weg zu v, sonst wäre die SCC von w noch nicht geschlossen (geschlossene SCCs sind bereits komplett), also SCCs unverändert
- Kante zu Knoten w in offener SCC (Nicht-Baumkante):
  falls v und w in unterschiedlichen SCCs liegen, müssen diese mit allen SCCs dazwischen zu einer einzigen SCC verschmolzen werden (durch Löschen der Repräsentanten)

Wenn Knoten keine ausgehenden Kanten mehr hat:

- Knoten fertig
- wenn Knoten Repräsentant seiner SCC ist, dann SCC schließen

Starke Zhk. und DFS

Vereinigung offener SCCs im Kreisfall:



- offene SCC entsprechen Ovalen, Knoten sortiert nach dfsNum
- alle Repräsentanten offener SCCs liegen auf Baumpfad zum aktuellen Knoten v in SCC S_k
- Nicht-Baumkante (v, w) endet an Knoten w in offener SCC S_i mit Repräsentant r_i
- Pfad von w nach r_i muss existieren (innerhalb SCC S_i)

$\Rightarrow$ Kante (v, w) vereinigt $S_i, \ldots, S_k$

Starke Zhk. und DFS

- init() {
  component = new int[n];
  oReps = ⟨ ⟩;
  oNodes = ⟨ ⟩;
  dfsCount = 1;
  }

- root(Node w) / traverseTreeEdge(Node v, Node w) {
  oReps.push(w); // Repräsentant einer neuen ZHK
  oNodes.push(w); // neuer offener Knoten
  dfsNum[w] = dfsCount;
  dfsCount++;
  }

Starke Zhk. und DFS

- traverseNonTreeEdge(Node v, Node w) {
  if (w ∈ oNodes) // verschmelze SCCs
  while (dfsNum[w] < dfsNum[oReps.top()])
  oReps.pop();
  }

- backtrack(Node u, Node v) {
  if (v == oReps.top()) { // v Repräsentant?
  oReps.pop(); // ja: entferne v
  do { // und offene Knoten bis v
  w = oNodes.pop();
  component[w] = v;
  } while (w!=v);
  }
  }

Starke Zhk. und DFS

Zeit: $\mathcal{O}(n+m)$

Begründung:

- init, root: $\mathcal{O}(1)$
- traverseTreeEdge: $(n-1) \times \mathcal{O}(1)$
- backtrack, traverseNonTreeEdge:
  da jeder Knoten höchstens einmal in oReps und oNodes landet,
  insgesamt $\mathcal{O}(n+m)$
- DFS-Gerüst: $\mathcal{O}(n+m)$
- gesamt: $\mathcal{O}(n+m)$

Übersicht

Kürzeste Wege

Zentrale Frage: Wie kommt man am schnellsten von A nach B?

Fälle:

- Kantenkosten 1
- DAG, beliebige Kantenkosten
- beliebiger Graph, positive Kantenkosten
- beliebiger Graph, beliebige Kantenkosten

Kürzeste-Wege-Problem

gegeben:

- gerichteter Graph $G = (V, E)$
- Kantenkosten $c : E \mapsto \mathbb{R}$

2 Varianten:

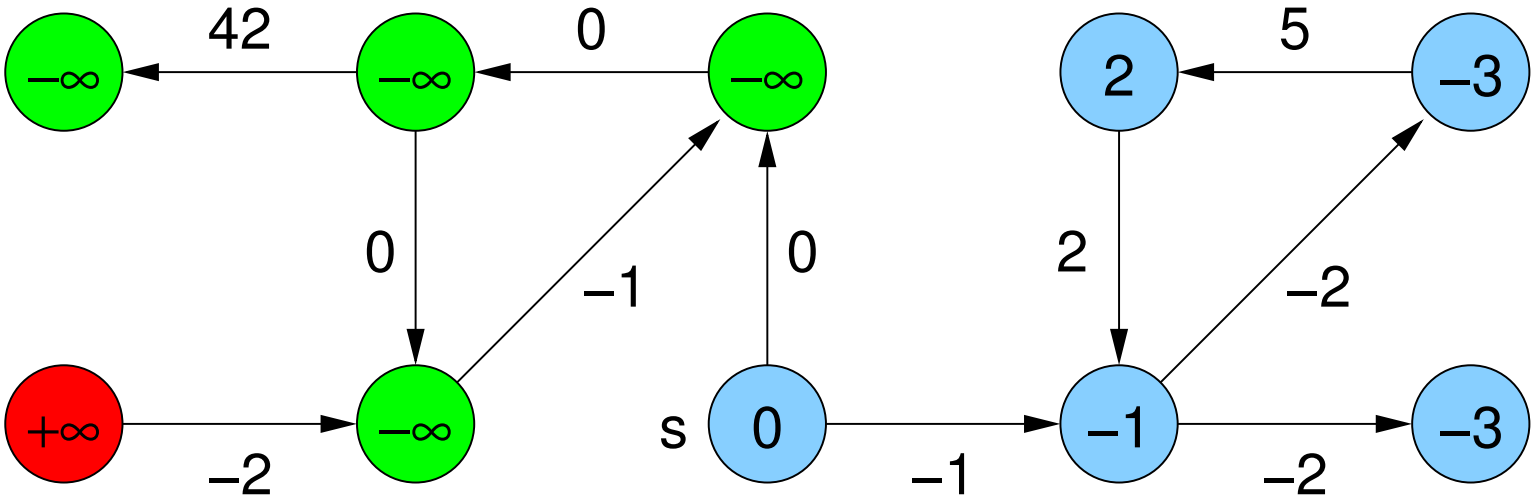
- SSSP (single source shortest paths):

  kürzeste Wege von einer Quelle zu allen anderen Knoten

- APSP (all pairs shortest paths):

  kürzeste Wege zwischen allen Paaren

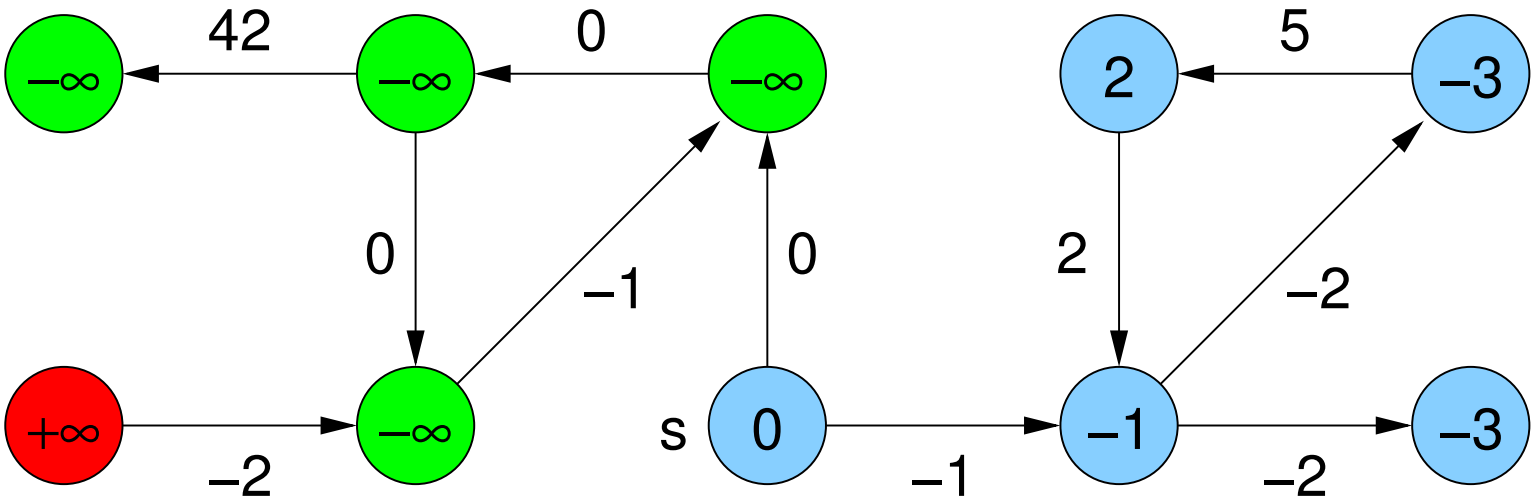
Distanzen



$\mu(s,v)$: Distanz von s nach v

$$\mu(s,v) = \begin{cases} +\infty & \text{kein Weg von } s \text{ nach } v \\ -\infty & \text{Weg beliebig kleiner Kosten von } s \text{ nach } v \\ \min\{c(p) : p \text{ ist Weg von } s \text{ nach } v\} \end{cases}$$

Distanzen



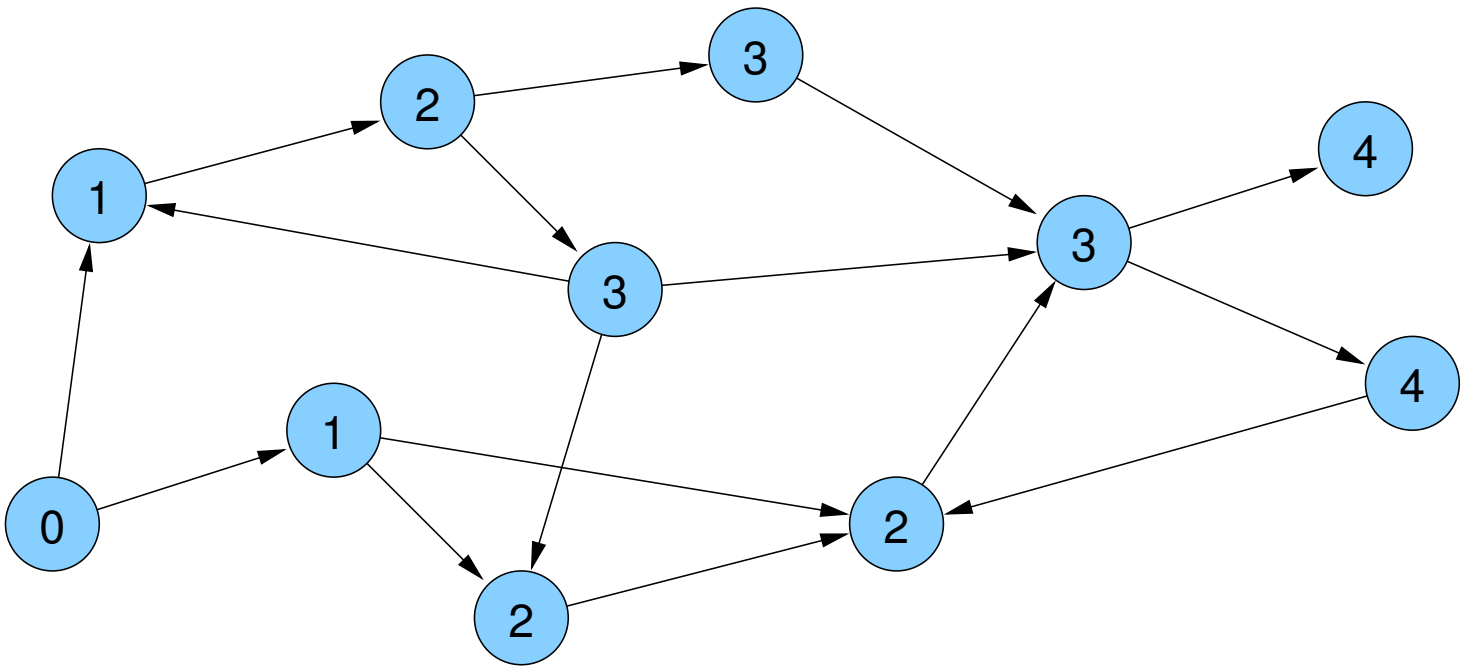
Wann sind die Kosten $-\infty$?

wenn es einen Kreis mit negativer Gewichtssumme gibt
(hinreichende und notwendige Bedingung)

Kürzeste Wege bei uniformen Kantenkosten

Graph mit Kantenkosten 1:

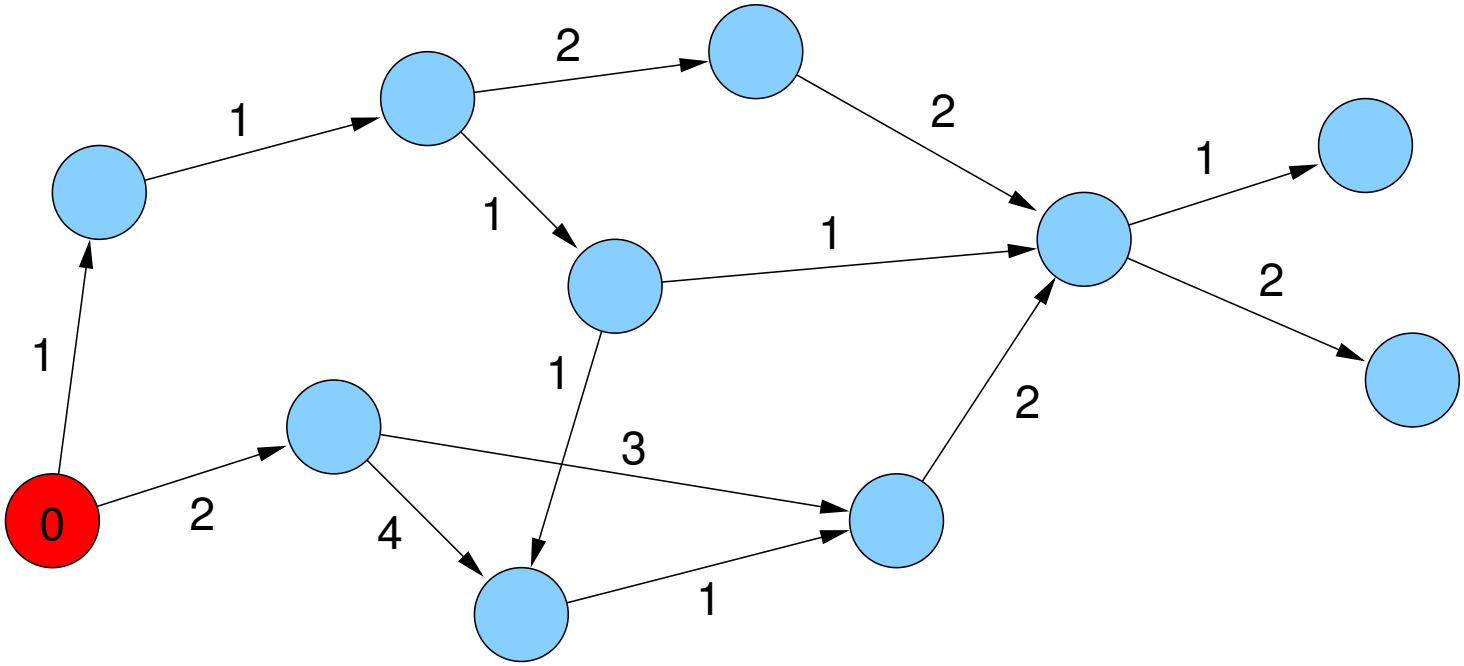
$\Rightarrow$ Breitensuche (BFS)



Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

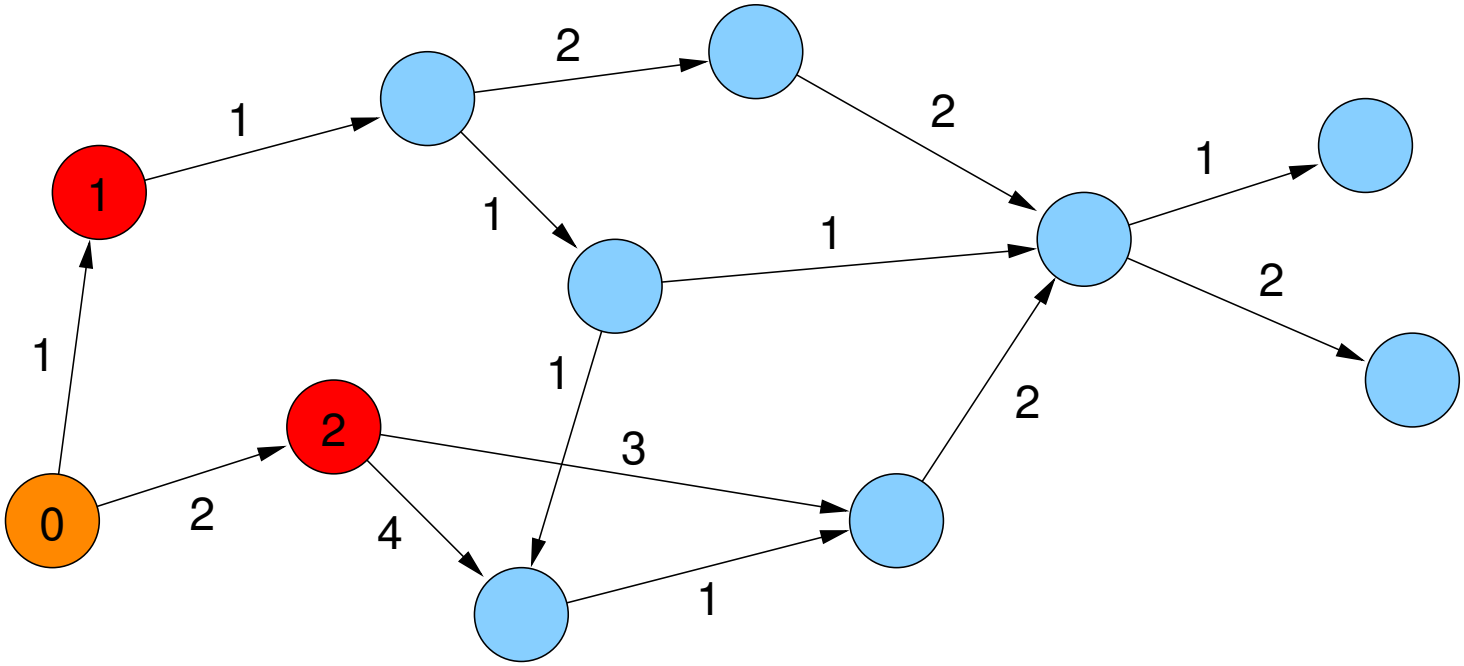
Einfache Breitensuche funktioniert nicht.



Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

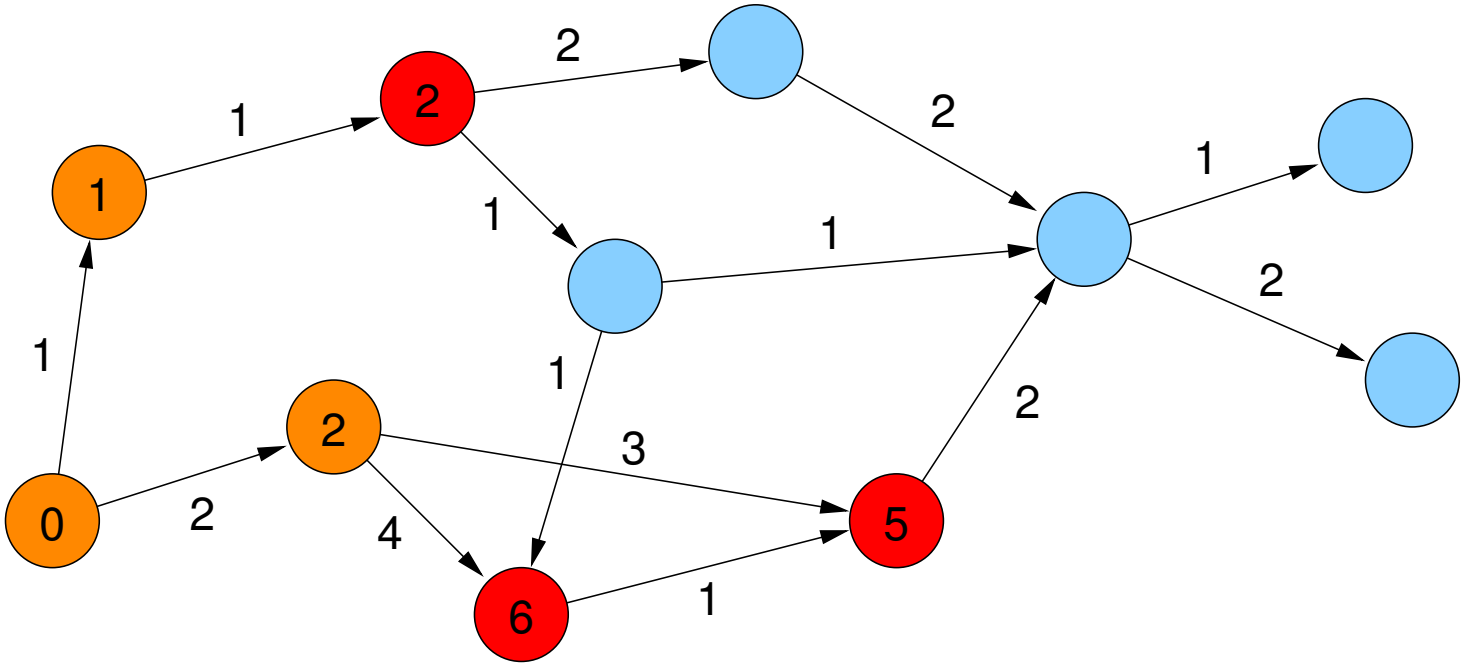
Einfache Breitensuche funktioniert nicht.



Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

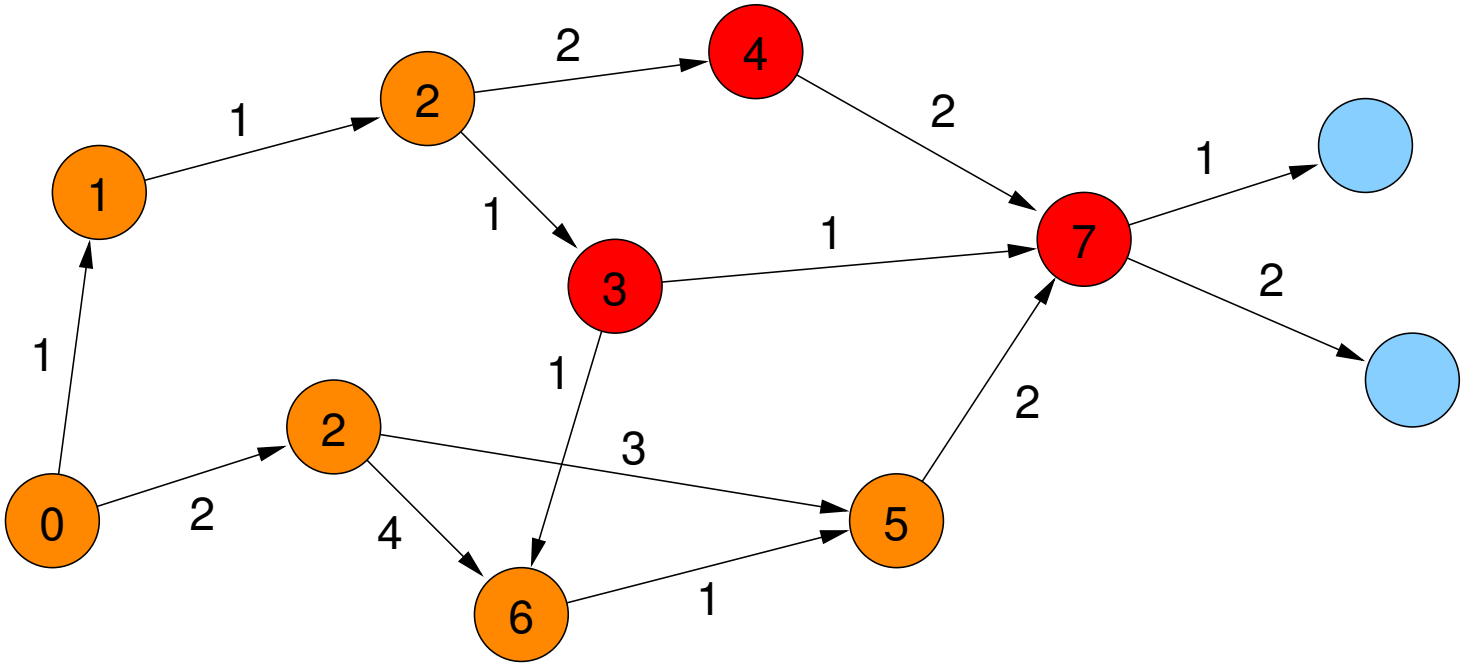
Einfache Breitensuche funktioniert nicht.



Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

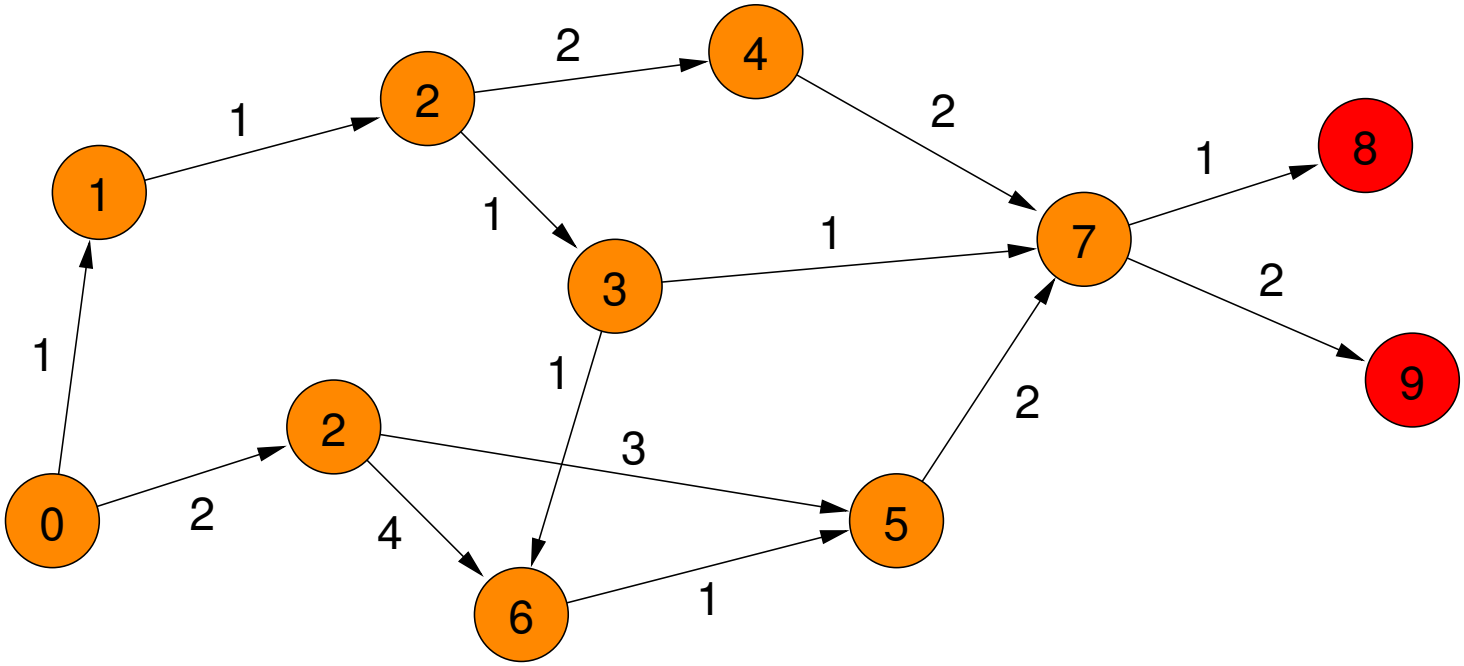
Einfache Breitensuche funktioniert nicht.



Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

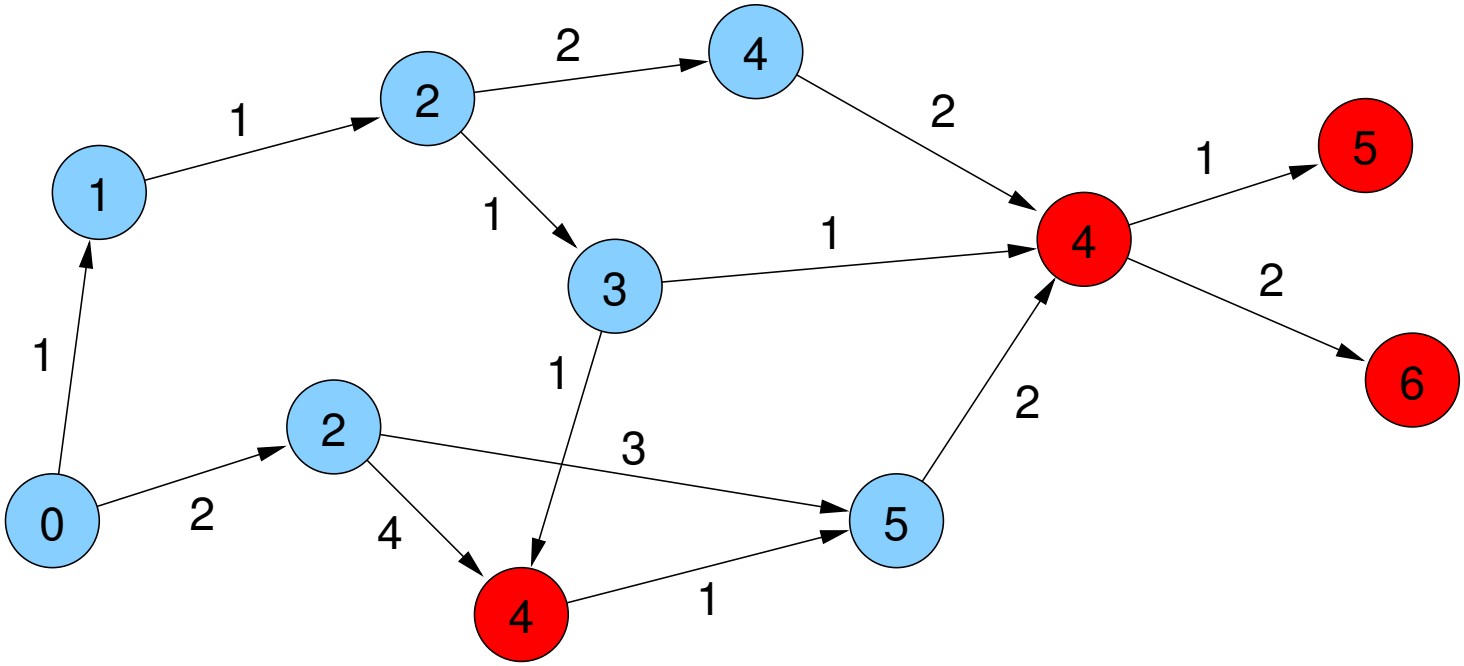
Einfache Breitensuche funktioniert nicht.



Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

Einfache Breitensuche funktioniert nicht.

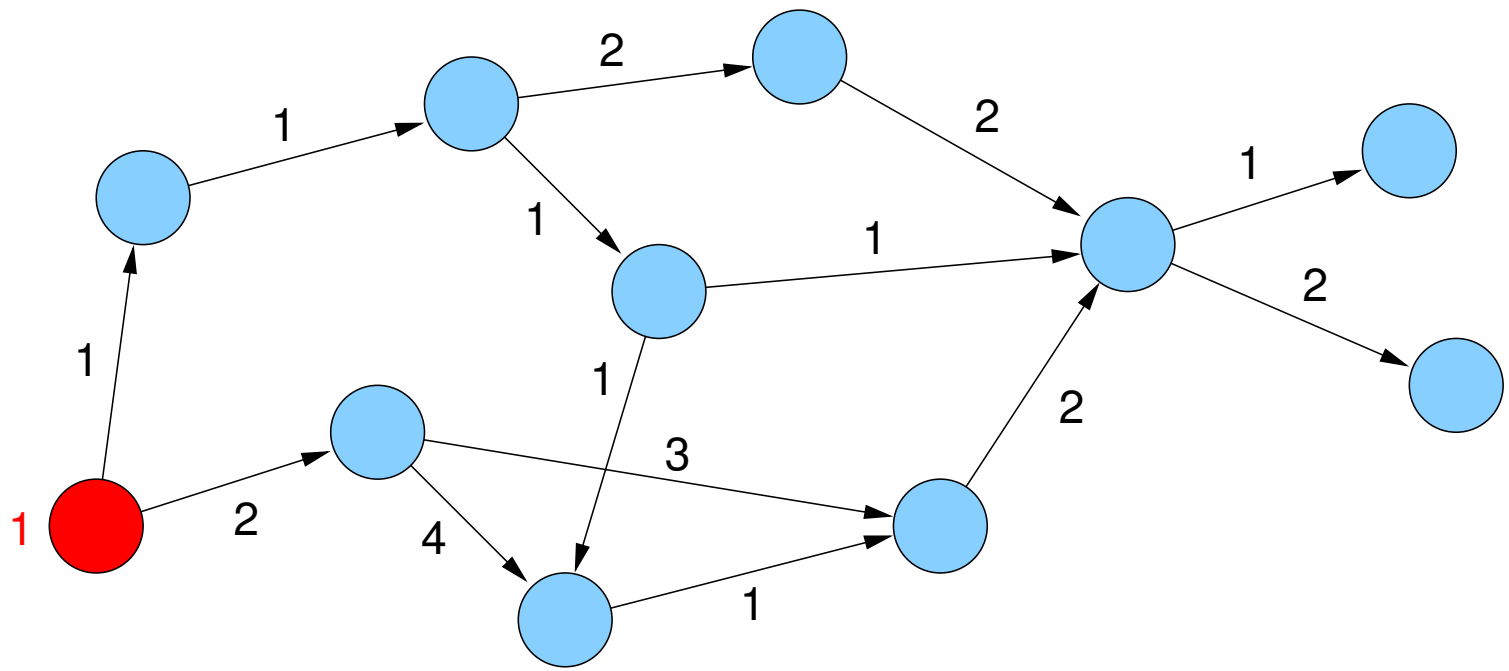


Topologische Sortierung in DAGs

Beliebige Kantengewichte in DAGs

Strategie: in DAGs gibt es topologische Sortierung

(für alle Kanten e=(v,w) gilt topoNum(v) < topoNum(w))

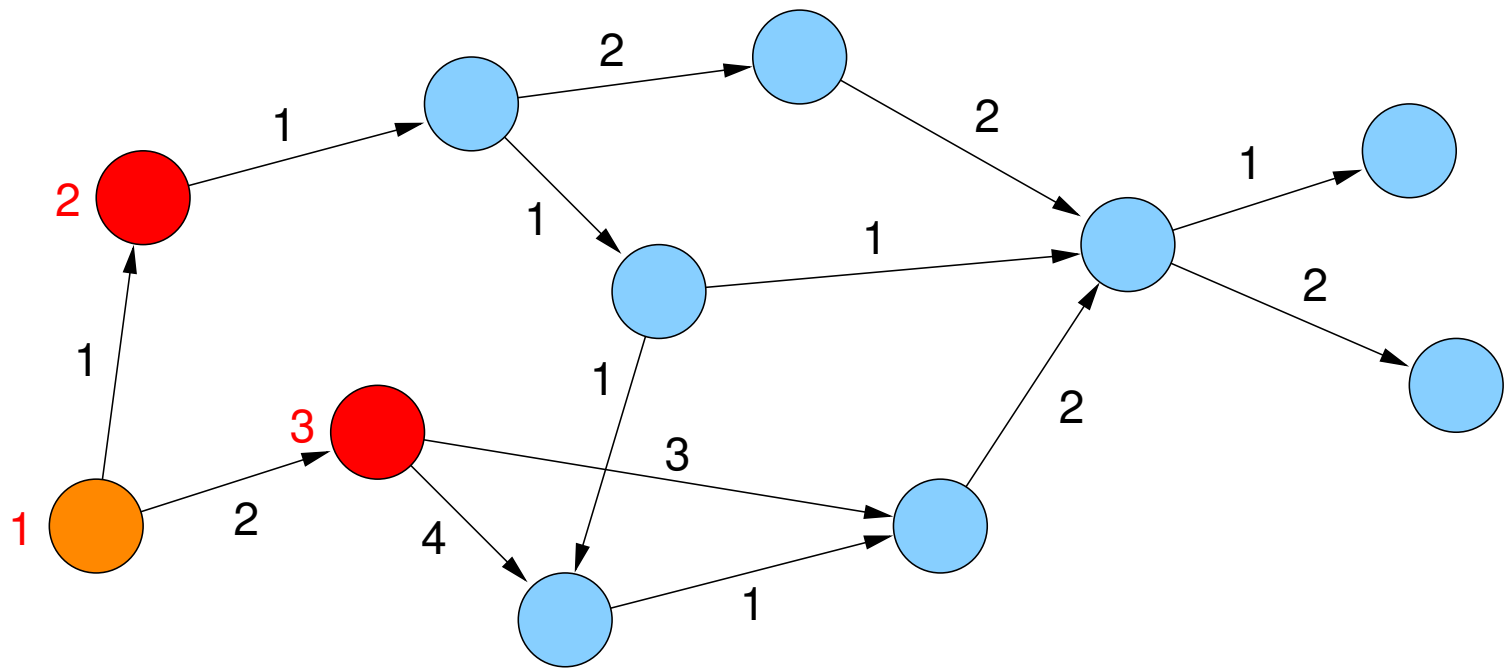


Topologische Sortierung in DAGs

Beliebige Kantengewichte in DAGs

Strategie: in DAGs gibt es topologische Sortierung

(für alle Kanten e=(v,w) gilt topoNum(v) < topoNum(w))

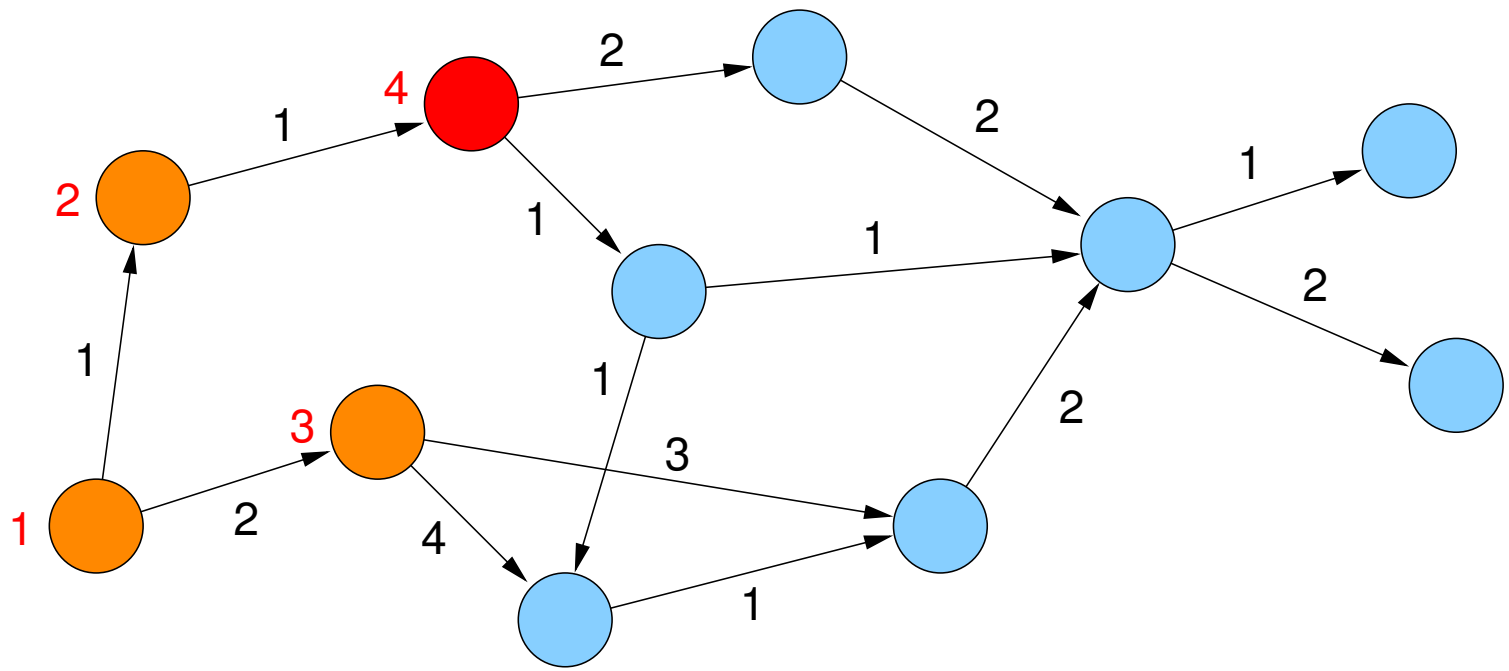


Topologische Sortierung in DAGs

Beliebige Kantengewichte in DAGs

Strategie: in DAGs gibt es topologische Sortierung

(für alle Kanten e=(v,w) gilt topoNum(v) < topoNum(w))

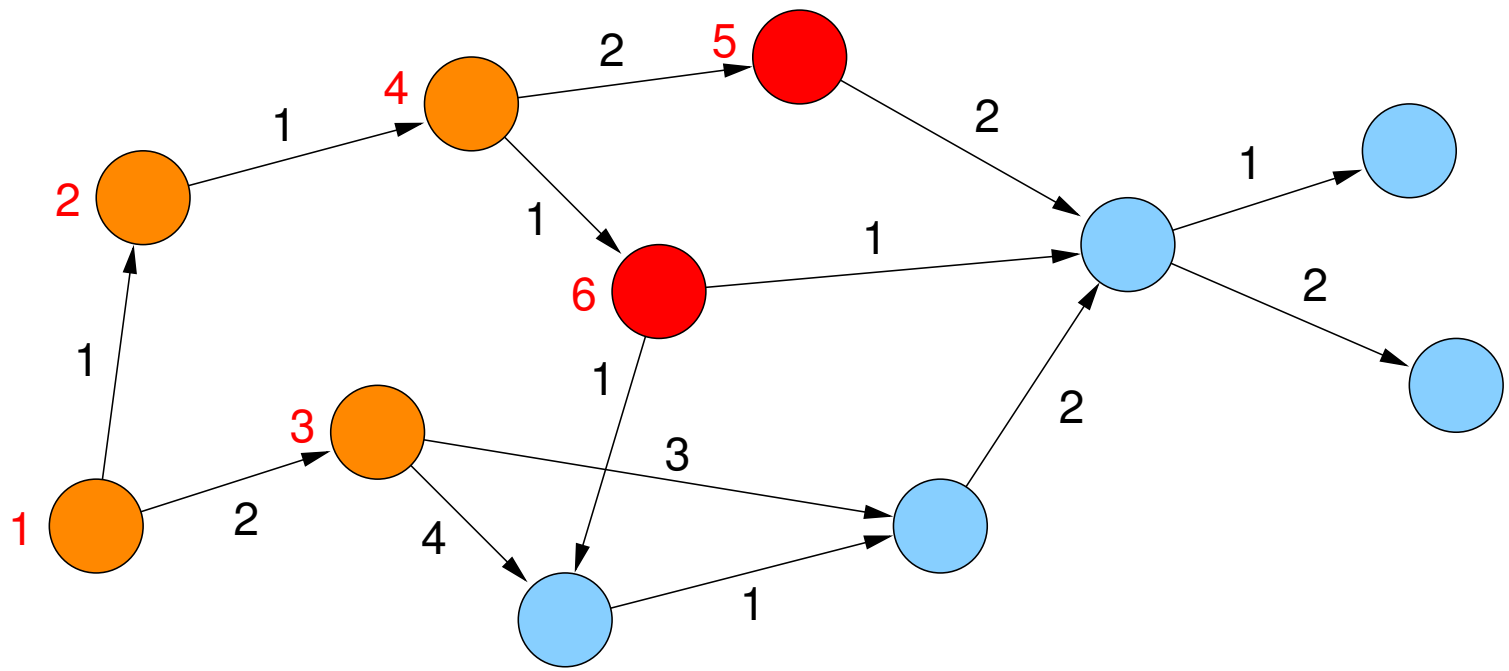


Topologische Sortierung in DAGs

Beliebige Kantengewichte in DAGs

Strategie: in DAGs gibt es topologische Sortierung

(für alle Kanten e=(v,w) gilt topoNum(v) < topoNum(w))

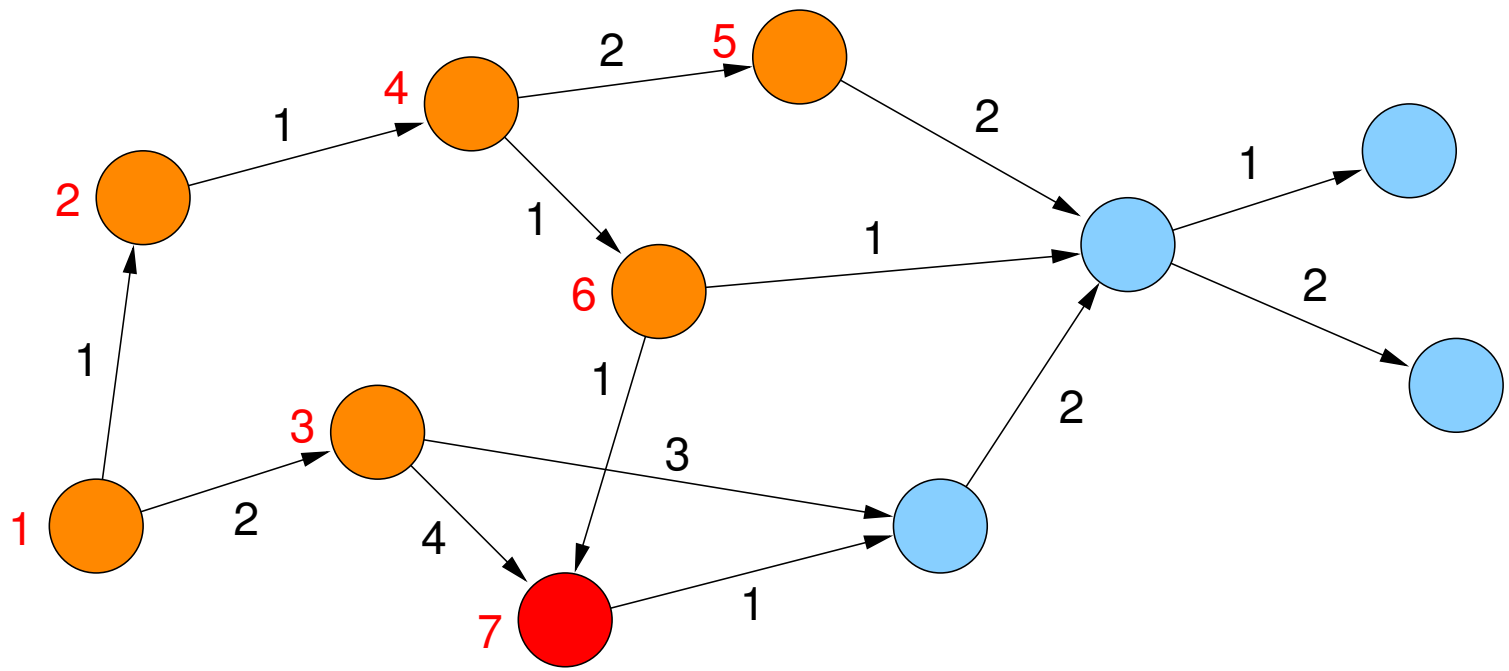


Topologische Sortierung in DAGs

Beliebige Kantengewichte in DAGs

Strategie: in DAGs gibt es topologische Sortierung

(für alle Kanten e=(v,w) gilt topoNum(v) < topoNum(w))

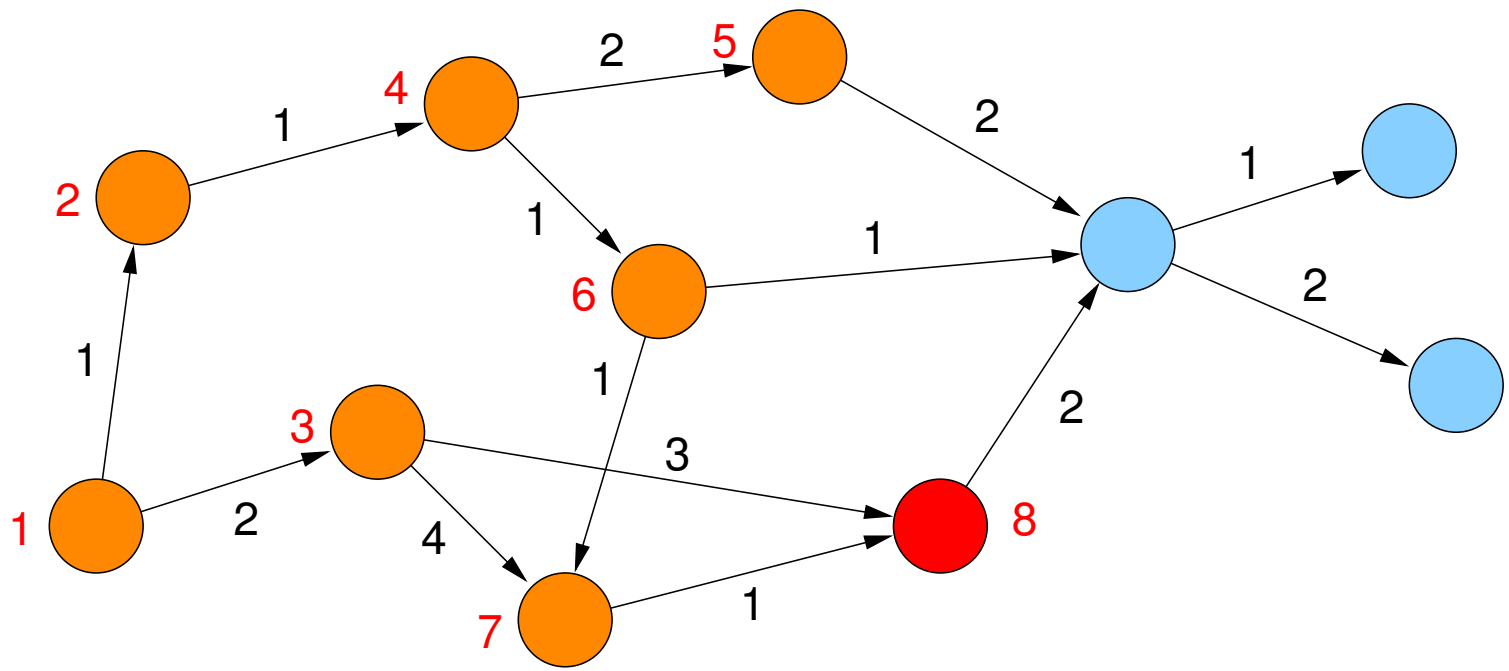


Topologische Sortierung in DAGs

Beliebige Kantengewichte in DAGs

Strategie: in DAGs gibt es topologische Sortierung

(für alle Kanten e=(v,w) gilt topoNum(v) < topoNum(w))

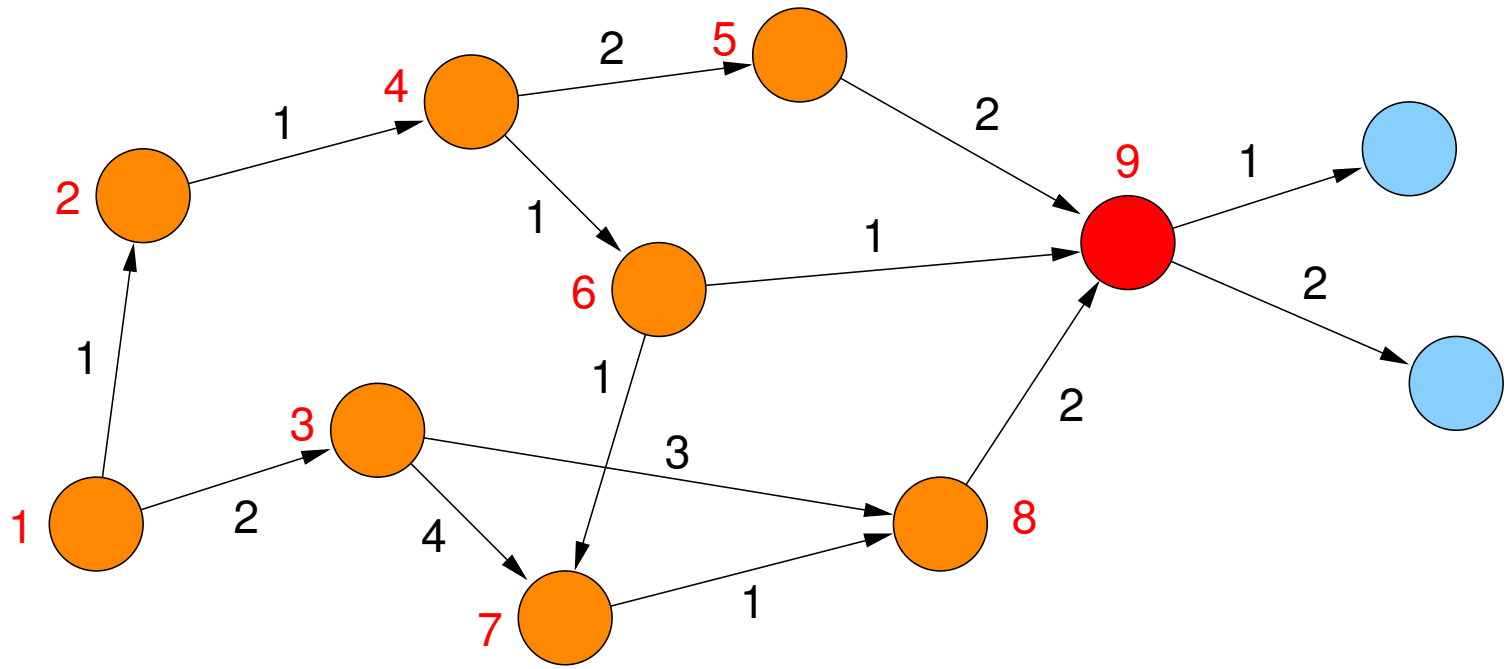


Topologische Sortierung in DAGs

Beliebige Kantengewichte in DAGs

Strategie: in DAGs gibt es topologische Sortierung

(für alle Kanten e=(v,w) gilt topoNum(v) < topoNum(w))

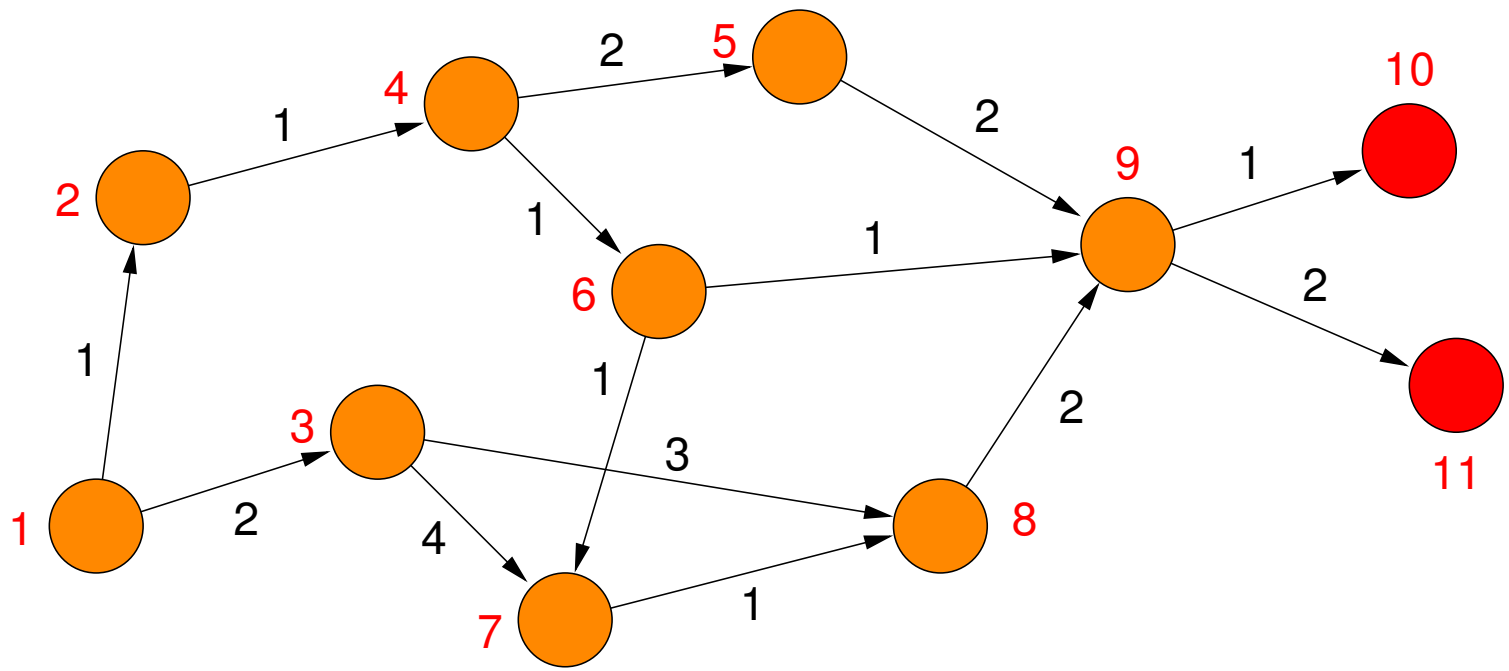


Topologische Sortierung in DAGs

Beliebige Kantengewichte in DAGs

Strategie: in DAGs gibt es topologische Sortierung

(für alle Kanten e=(v,w) gilt topoNum(v) < topoNum(w))

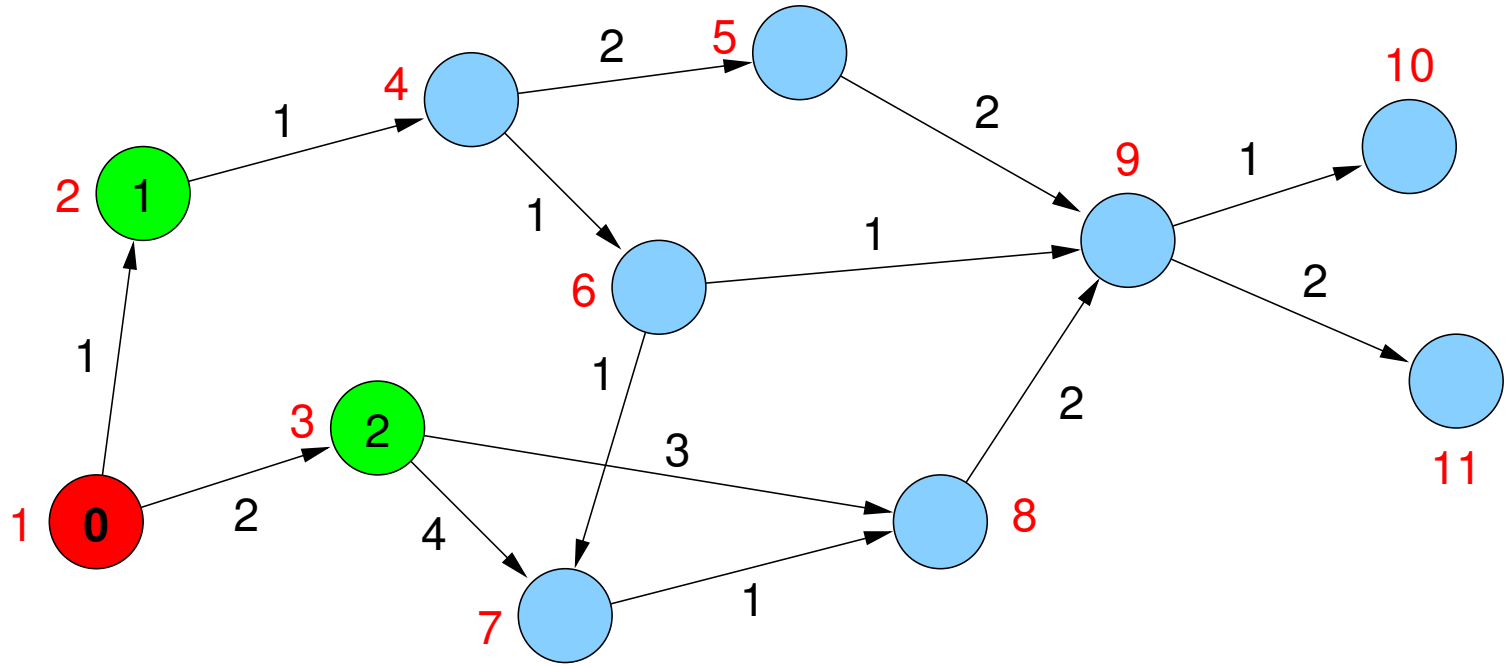


Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

Strategie:

- betrachte Knoten in Reihenfolge der topologischen Sortierung
- aktualisiere Distanzwerte

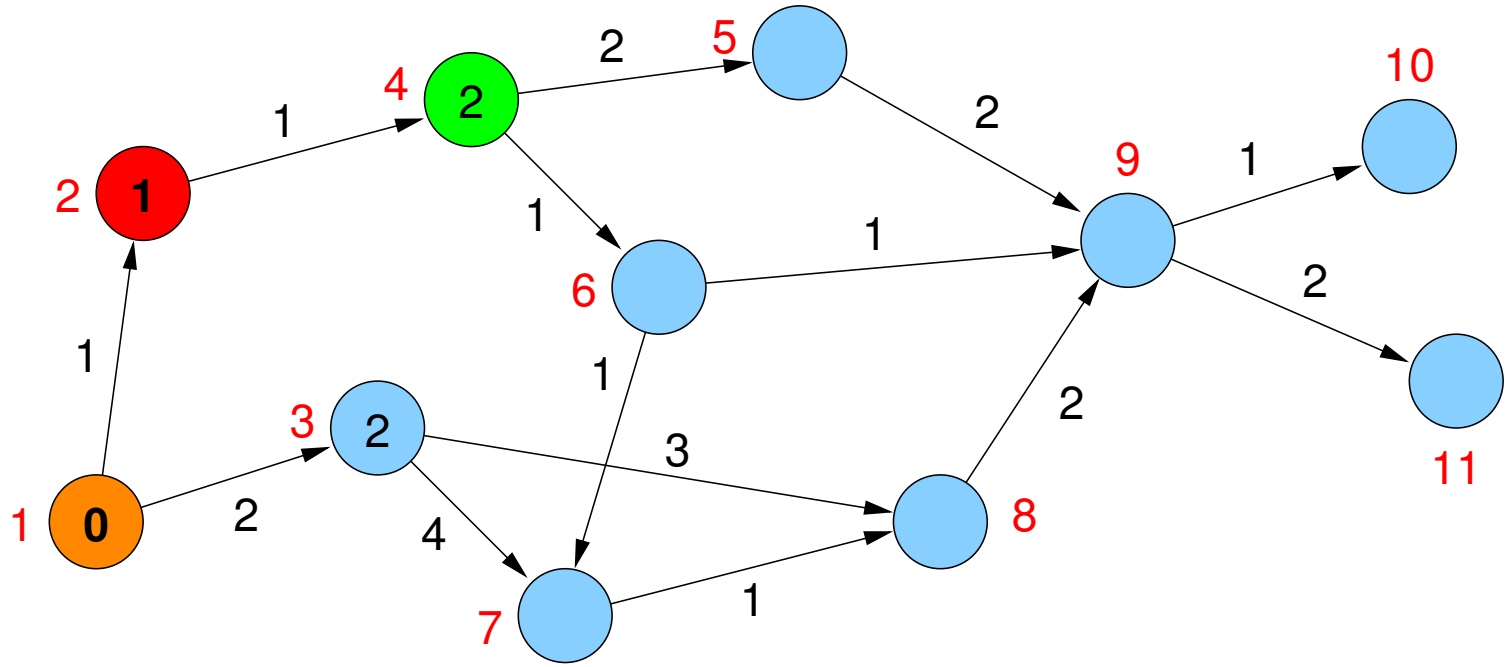


Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

Strategie:

- betrachte Knoten in Reihenfolge der topologischen Sortierung
- aktualisiere Distanzwerte

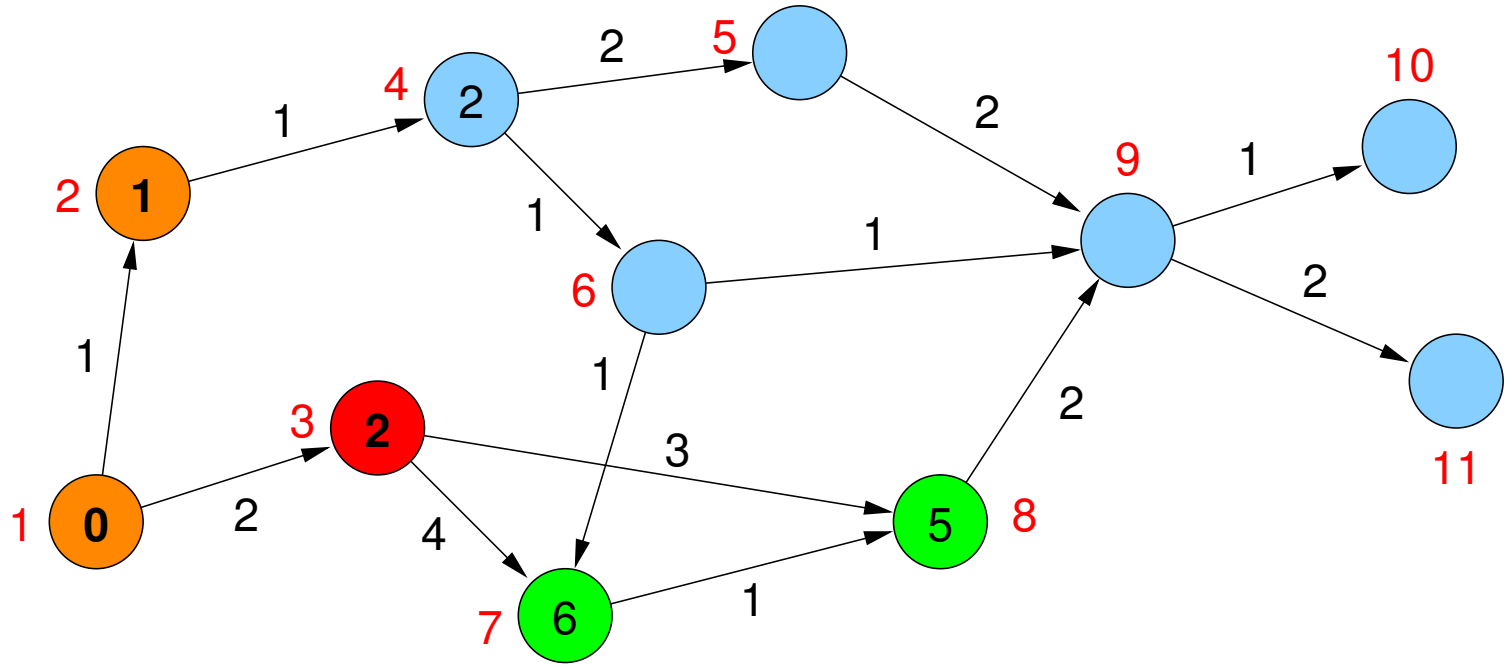


Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

Strategie:

- betrachte Knoten in Reihenfolge der topologischen Sortierung
- aktualisiere Distanzwerte

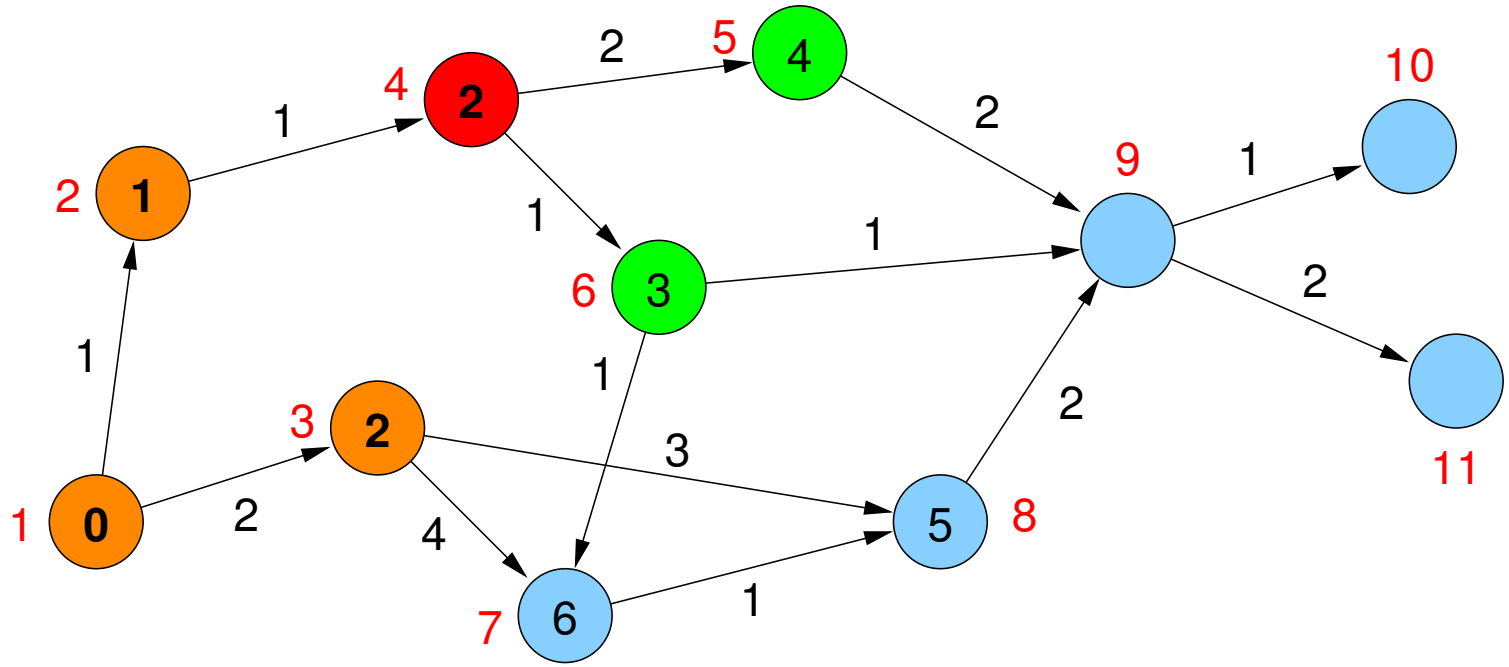


Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

Strategie:

- betrachte Knoten in Reihenfolge der topologischen Sortierung
- aktualisiere Distanzwerte

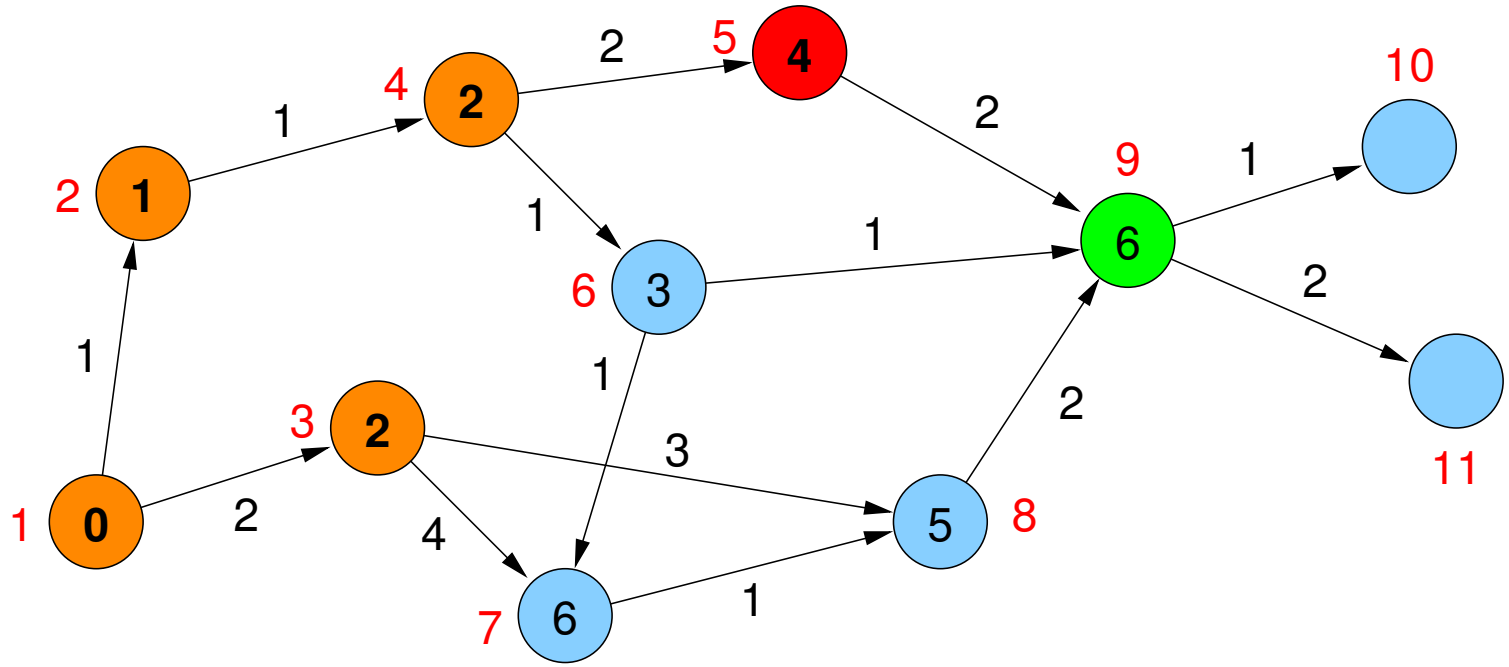


Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

Strategie:

- betrachte Knoten in Reihenfolge der topologischen Sortierung
- aktualisiere Distanzwerte

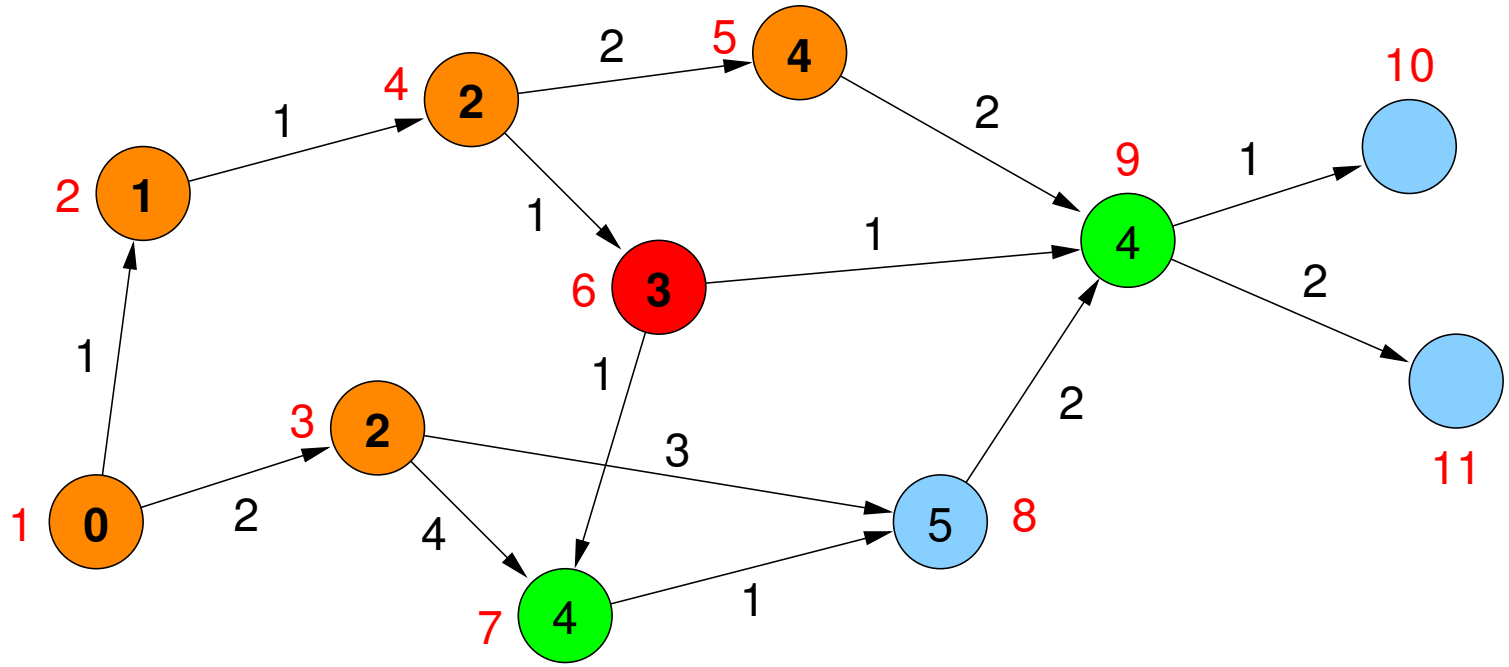


Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

Strategie:

- betrachte Knoten in Reihenfolge der topologischen Sortierung
- aktualisiere Distanzwerte

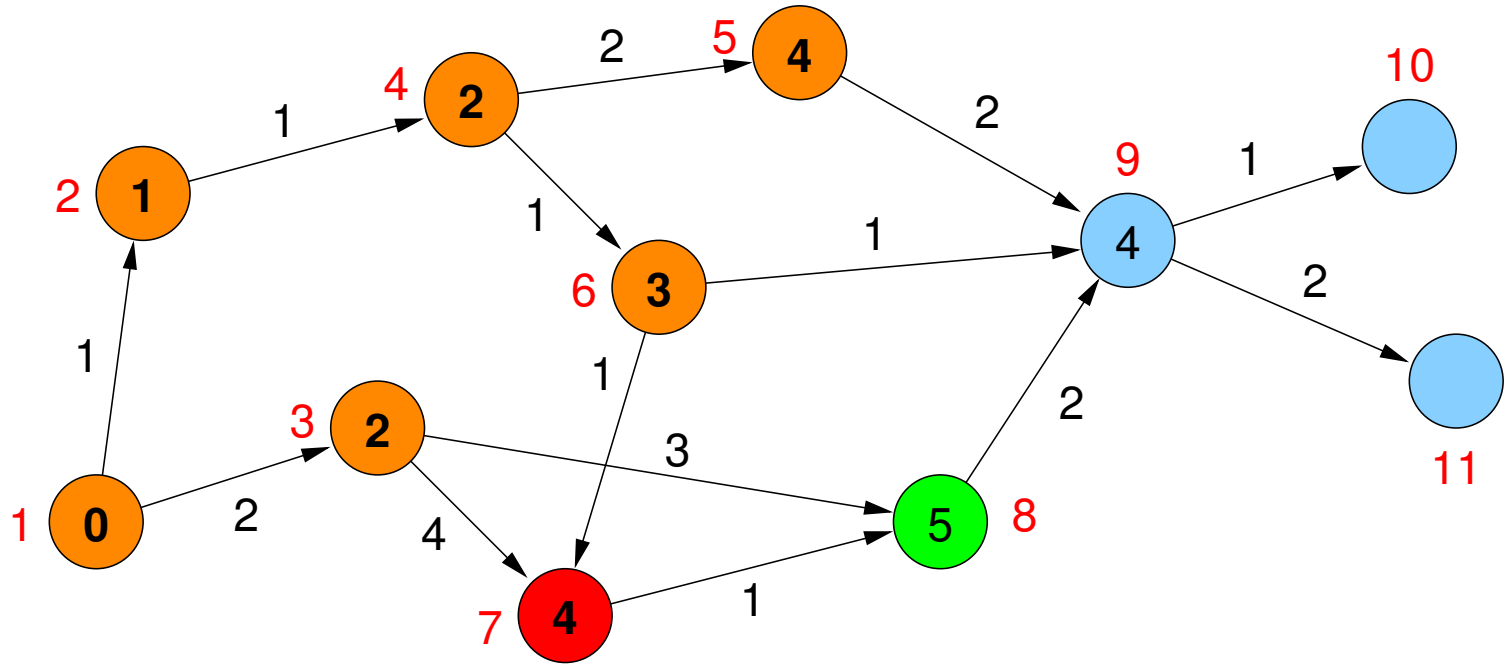


Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

Strategie:

- betrachte Knoten in Reihenfolge der topologischen Sortierung
- aktualisiere Distanzwerte

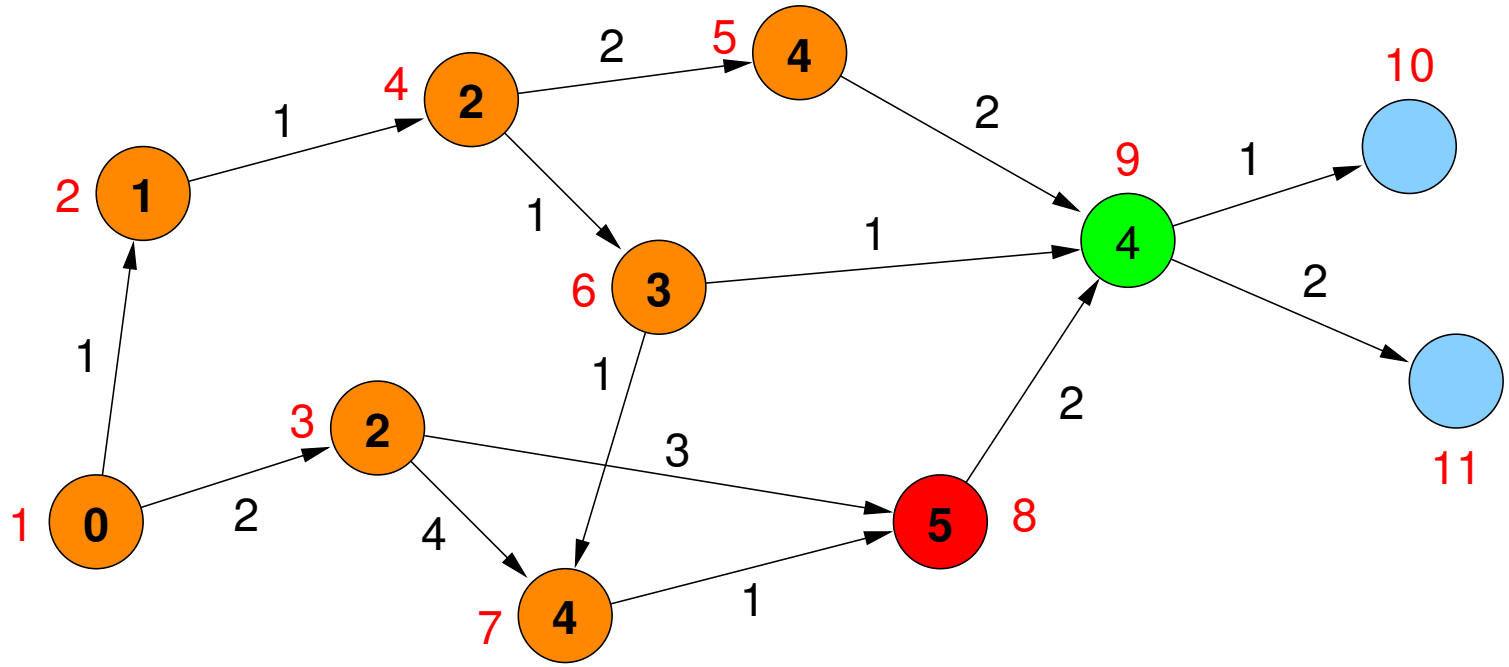


Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

Strategie:

- betrachte Knoten in Reihenfolge der topologischen Sortierung
- aktualisiere Distanzwerte

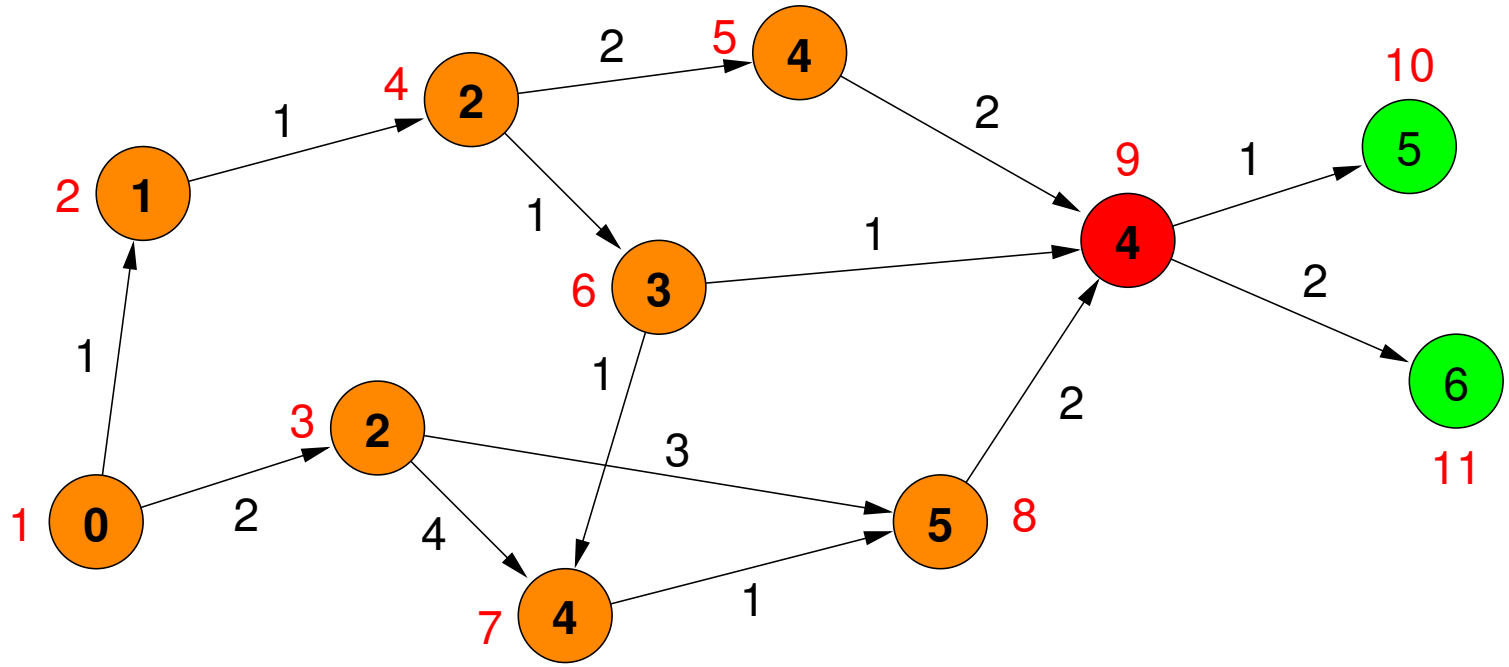


Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

Strategie:

- betrachte Knoten in Reihenfolge der topologischen Sortierung
- aktualisiere Distanzwerte

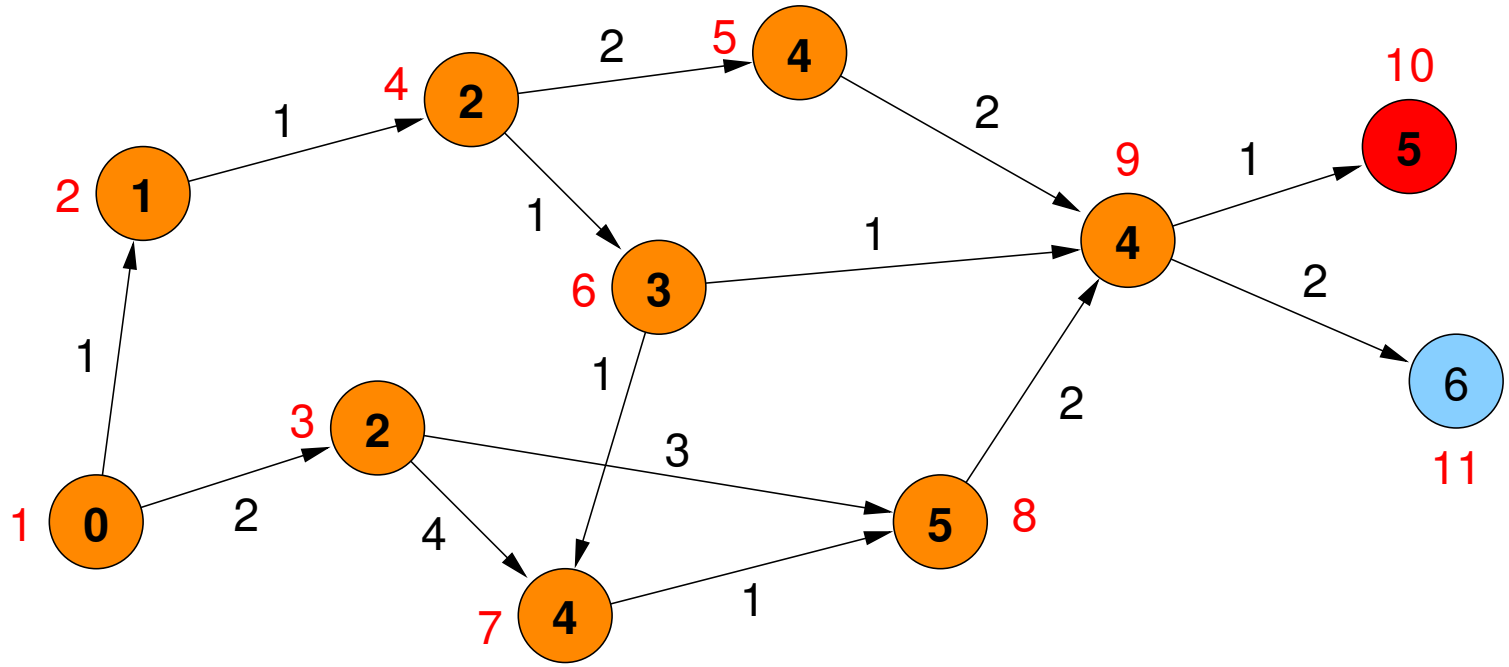


Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

Strategie:

- betrachte Knoten in Reihenfolge der topologischen Sortierung
- aktualisiere Distanzwerte

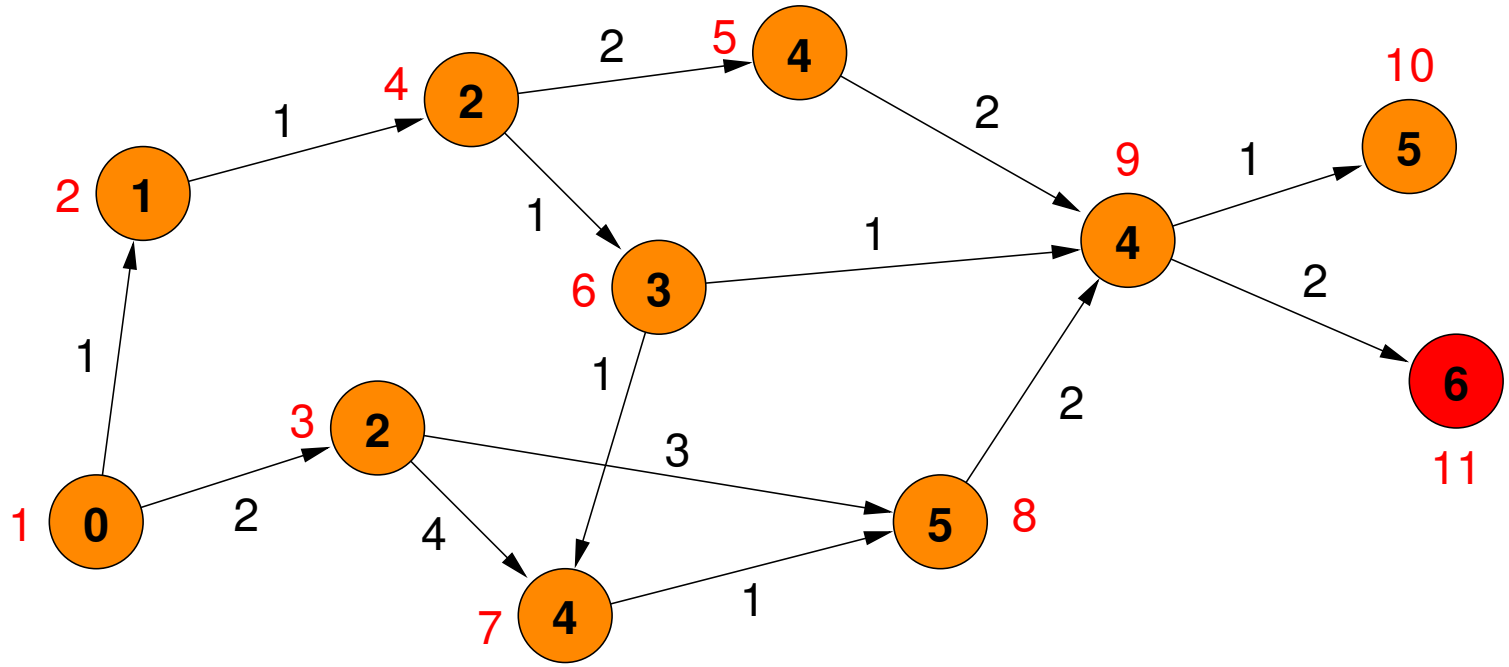


Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

Strategie:

- betrachte Knoten in Reihenfolge der topologischen Sortierung
- aktualisiere Distanzwerte



Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

Topologische Sortierung – warum funktioniert das?

- betrachte einen kürzesten Weg von s nach v
- der ganze Pfad beachtet die topologische Sortierung
- d.h., die Distanzen werden in der Reihenfolge der Knoten vom Anfang des Pfades zum Ende hin betrachtet
- damit ergibt sich für v der richtige Distanzwert

- ein Knoten x kann auch nie einen Wert erhalten, der echt kleiner als seine Distanz zu s ist
- die Kantenfolge von s zu x, die jeweils zu den Distanzwerten an den Knoten geführt hat, wäre dann ein kürzerer Pfad (Widerspruch)

Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

Allgemeine Strategie:

- Anfang: setze $d(s) = 0$ und
  für alle anderen Knoten v setze $d(v) = \infty$
- besuche Knoten in einer Reihenfolge, die sicherstellt, dass mindestens ein kürzester Weg von s zu jedem v in der Reihenfolge seiner Knoten besucht wird
- für jeden besuchten Knoten v aktualisiere die Distanzen der Knoten w mit $(v, w) \in E$, d.h. setze

$$d(w) = \min\{\, d(w),\ d(v) + c(v, w) \,\}$$

Kürzeste Wege in DAGs

Topologische Sortierung

- verwende FIFO-Queue q
- verwalte für jeden Knoten einen Zähler für die noch nicht markierten eingehenden Kanten
- initialisiere q mit allen Knoten, die keine eingehende Kante haben (Quellen)
- nimm nächsten Knoten v aus q und
  markiere alle $(v, w) \in E$, d.h. dekrementiere Zähler für w
- falls der Zähler von w dabei Null wird, füge w in q ein
- wiederhole das, bis q leer wird

Kürzeste Wege in DAGs

Topologische Sortierung

Korrektheit

- Knoten wird erst dann nummeriert, wenn alle Vorgänger nummeriert sind

Laufzeit

- für die Anfangswerte der Zähler muss der Graph einmal traversiert werden $O(n+m)$
- danach wird jede Kante genau einmal betrachtet

$\Rightarrow$ gesamt: $O(n+m)$

Test auf DAG-Eigenschaft

- topologische Sortierung erfasst genau dann alle Knoten, wenn der Graph ein DAG ist
- bei gerichteten Kreisen erhalten diese Knoten keine Nummer

Kürzeste Wege in DAGs

DAG-Strategie

1. Topologische Sortierung der Knoten

   Laufzeit $O(n+m)$

2. Aktualisierung der Distanzen gemäß der topologischen Sortierung

   Laufzeit $O(n+m)$

Gesamtlaufzeit: $O(n+m)$

Beliebige Graphen mit nicht-negativen Gewichten

Gegeben:

- beliebiger Graph
  (gerichtet oder ungerichtet, muss diesmal kein DAG sein)
- mit nicht-negativen Kantengewichten

$\Rightarrow$ keine Knoten mit Distanz $-\infty$

Problem:

- besuche Knoten eines kürzesten Weges in der richtigen Reihenfolge
- wie bei Breitensuche, jedoch diesmal auch mit Distanzen $\neq 1$

Lösung:

- besuche Knoten in der Reihenfolge der kürzesten Distanz zum Startknoten s

Kürzeste Pfade: SSSP / Dijkstra

Algorithmus Dijkstra1: löst SSSP-Problem

Eingabe : $G = (V, E), \quad c : E \mapsto \mathbb{R}, \quad s \in V$

Ausgabe : Distanzen $d(s, v)$ zu allen $v \in V$

$P = \emptyset; \quad T = V;$

$d(s, v) = \infty$ for all $v \in V \setminus s;$

$d(s, s) = 0; \quad pred(s) = \perp;$

while $P \neq V$ **do**

  $v = \text{argmin}_{v \in T}\{d(s, v)\};$

  $P = P \cup v; \quad T = T \setminus v;$

  forall the $(v, w) \in E$ **do**

    if $d(s, w) > d(s, v) + c(v, w)$ **then**

      $d(s, w) = d(s, v) + c(v, w);$

      $pred(w) = v;$

Algorithmus Dijkstra2: löst SSSP-Problem

Eingabe : $G = (V, E)$, $c : E \mapsto \mathbb{R}_{\geq 0}$, $s \in V$

Ausgabe : Distanzen $d[v]$ von s zu allen $v \in V$

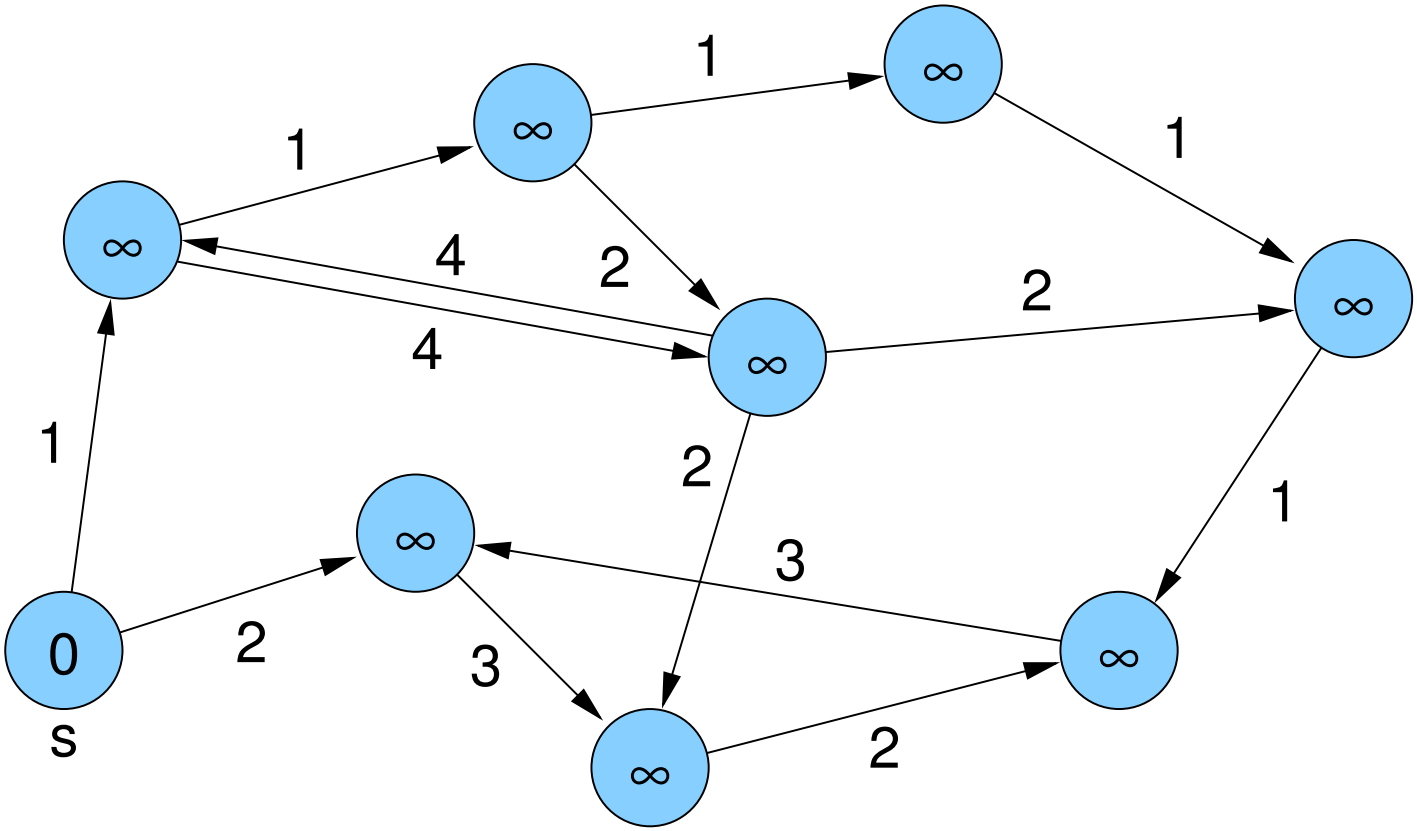
$d[v] = \infty$ for all $v \in V \setminus s$;
$d[s] = 0$; $pred[s] = \perp$;
$pq = \langle\rangle$; pq.insert$(s, 0)$;
while $\neg pq.empty()$ **do**
  $v = pq$.deleteMin();
  forall the $(v, w) \in E$ **do**
    newDist $= d[v] + c(v, w)$;
    if $newDist < d[w]$ **then**
      $pred[w] = v$;
      if $d[w] == \infty$ **then** pq.insert$(w,$ newDist$)$;
      else pq.decreaseKey$(w,$ newDist$)$;
      $d[w] =$ newDist;

Dijkstra-Algorithmus

- setze Startwert $d(s, s) = 0$ und zunächst $d(s, v) = \infty$
- verwende Prioritätswarteschlange, um die Knoten zusammen mit ihren aktuellen Distanzen zuspeichern
- am Anfang nur Startknoten (mit Distanz 0) in Priority Queue
- dann immer nächsten Knoten v (mit kleinster Distanz) entnehmen, endgültige Distanz dieses Knotens v steht nun fest
- betrachte alle Nachbarn von v,
  füge sie ggf. in die PQ ein bzw.
  aktualisiere deren Priorität in der PQ

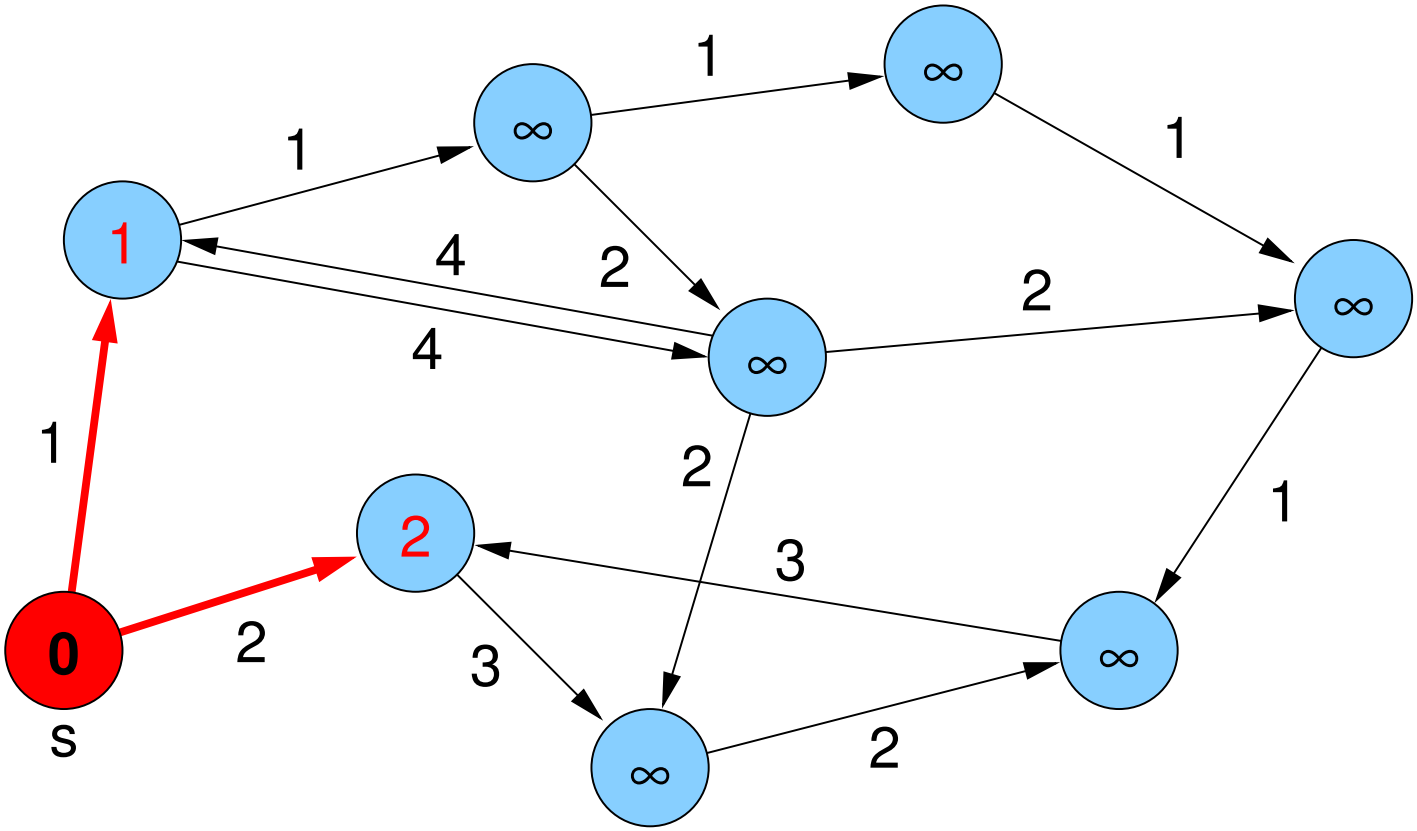
Dijkstra-Algorithmus

Beispiel:



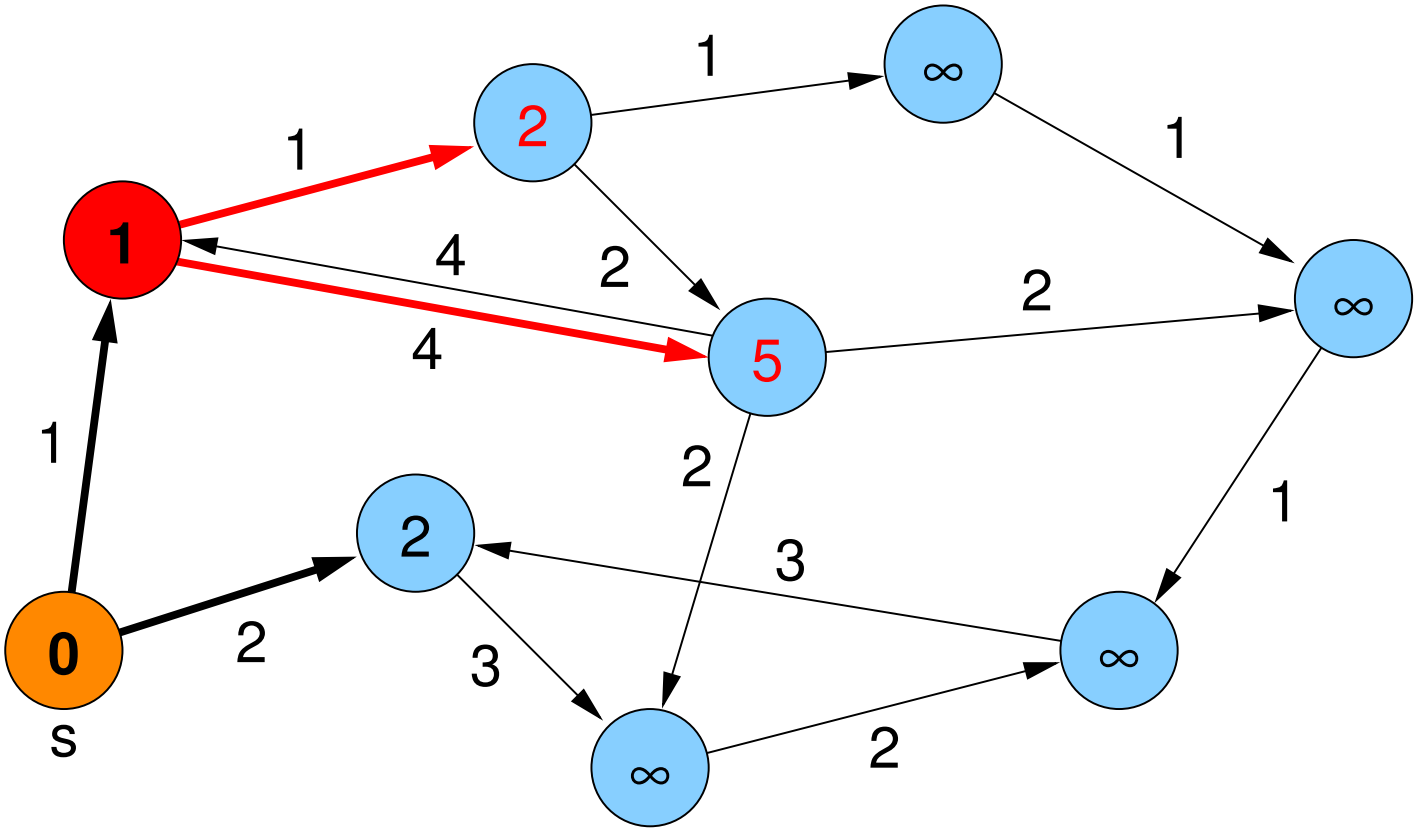
Dijkstra-Algorithmus

Beispiel:



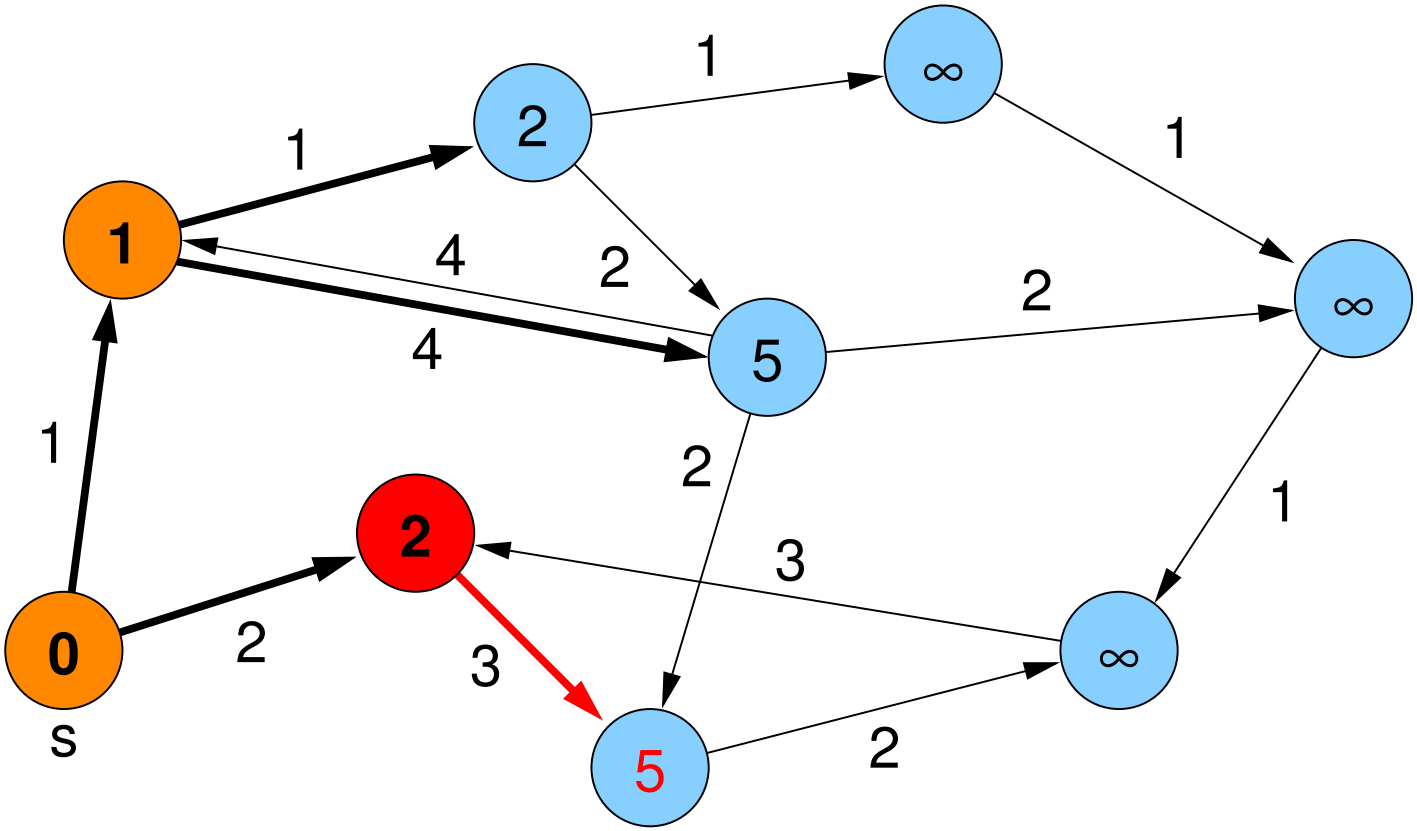
Dijkstra-Algorithmus

Beispiel:



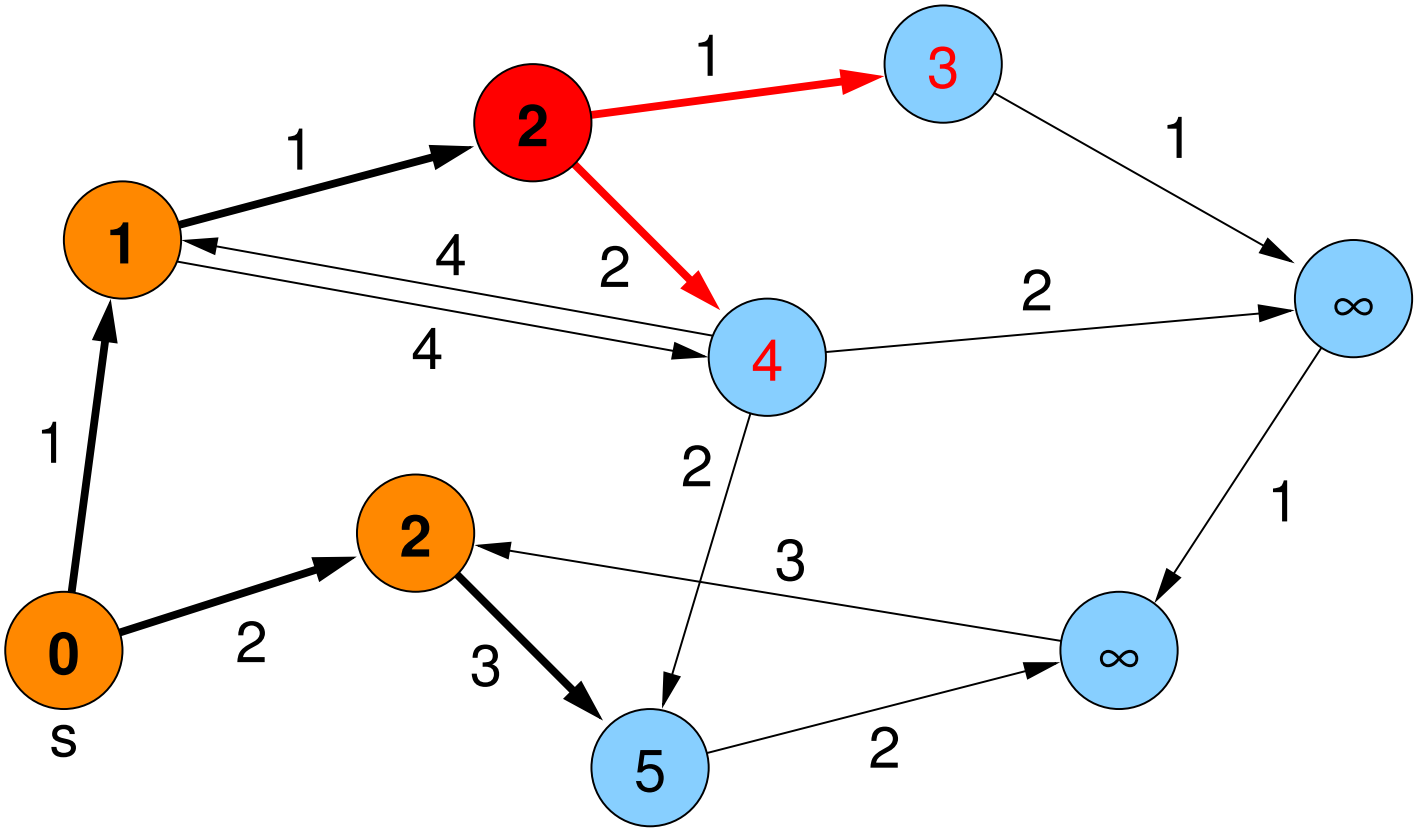
Dijkstra-Algorithmus

Beispiel:



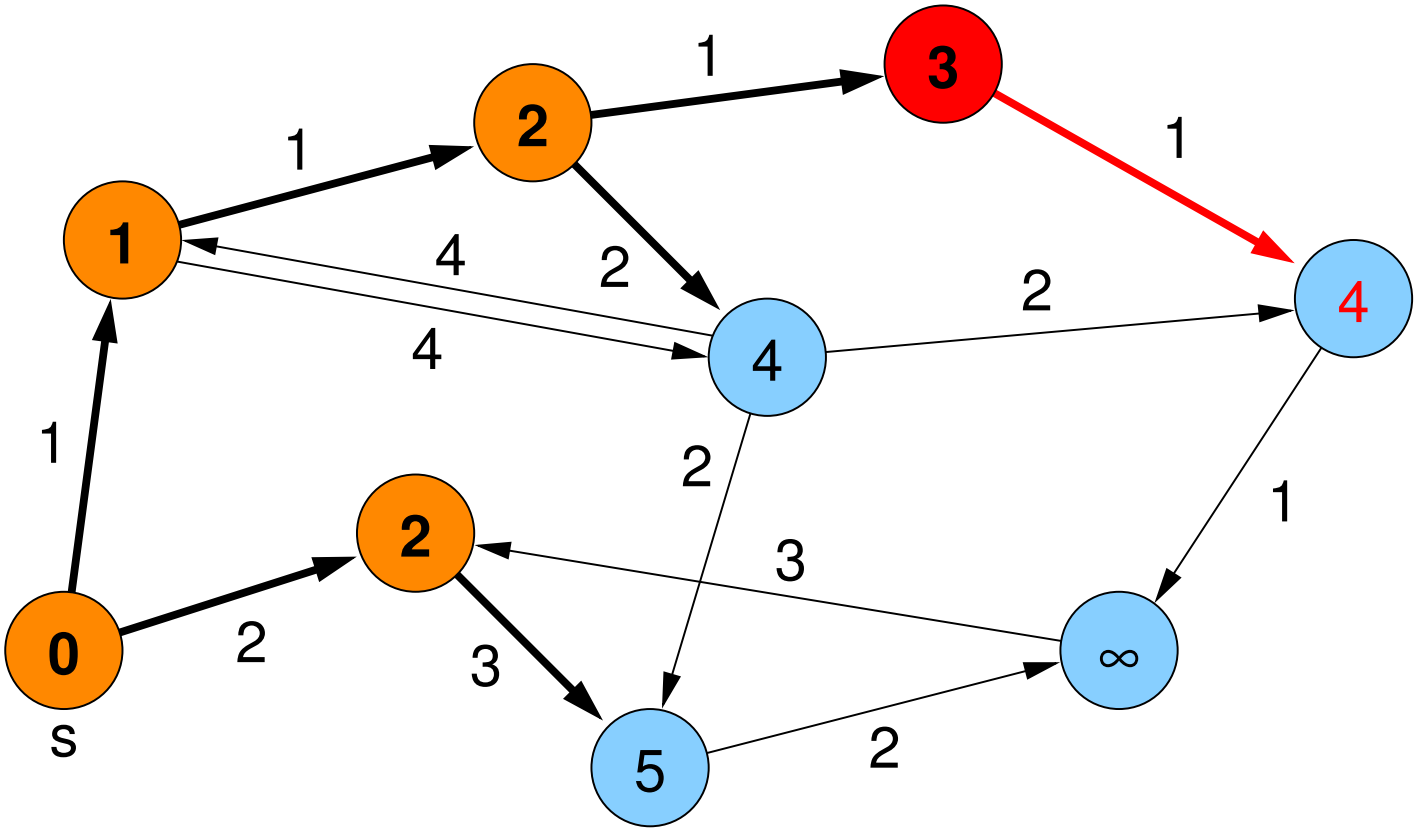
Dijkstra-Algorithmus

Beispiel:



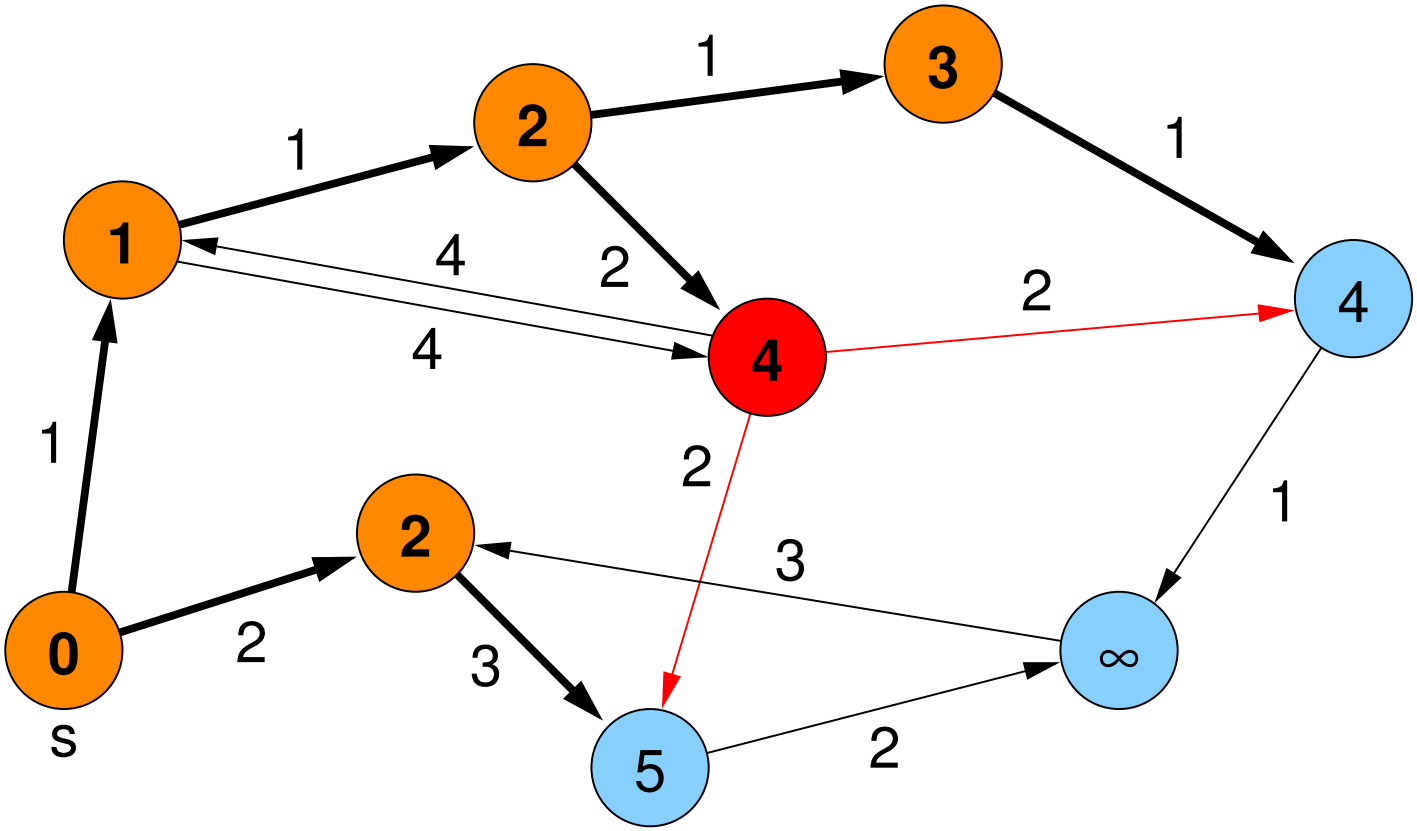
Dijkstra-Algorithmus

Beispiel:



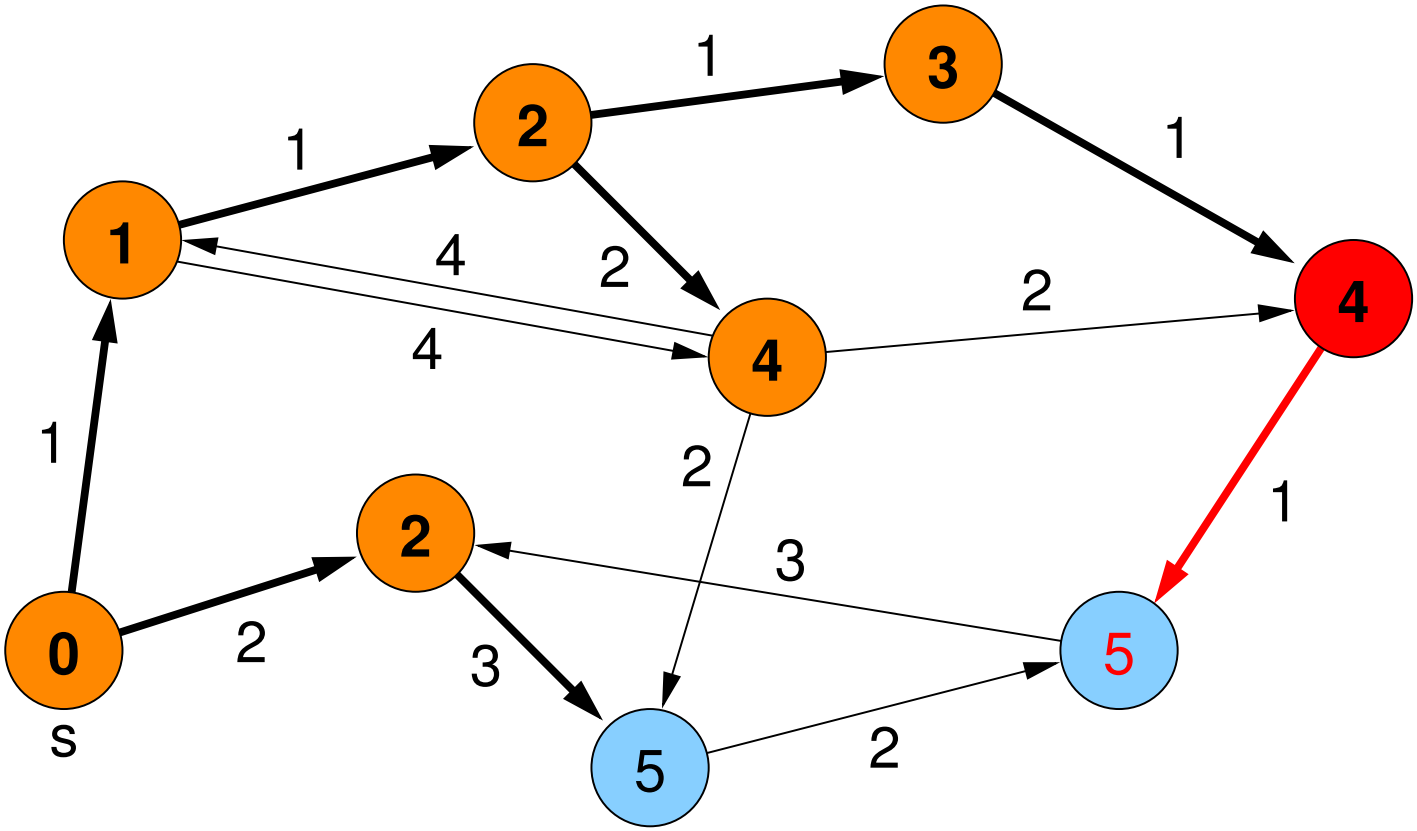
Dijkstra-Algorithmus

Beispiel:



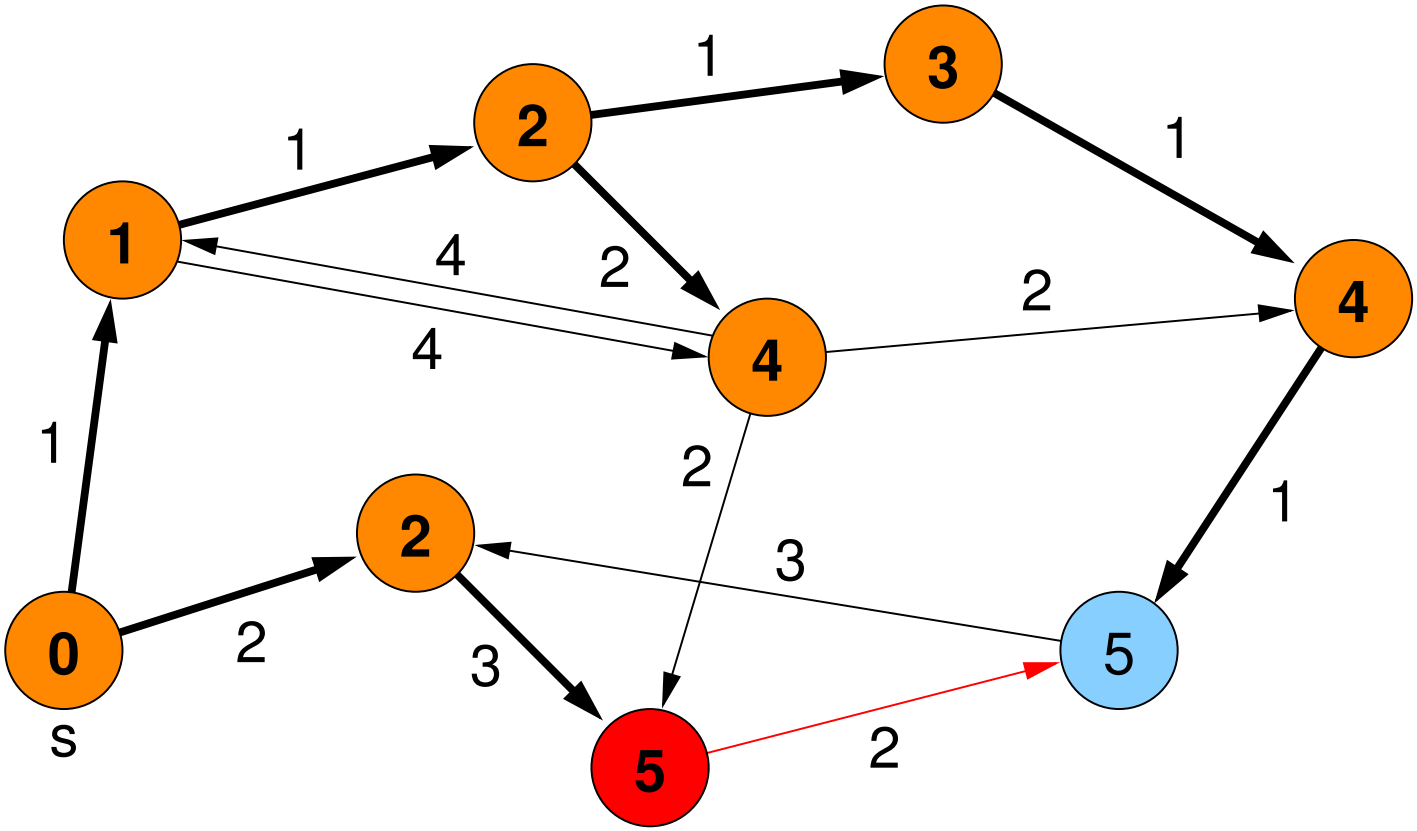
Dijkstra-Algorithmus

Beispiel:



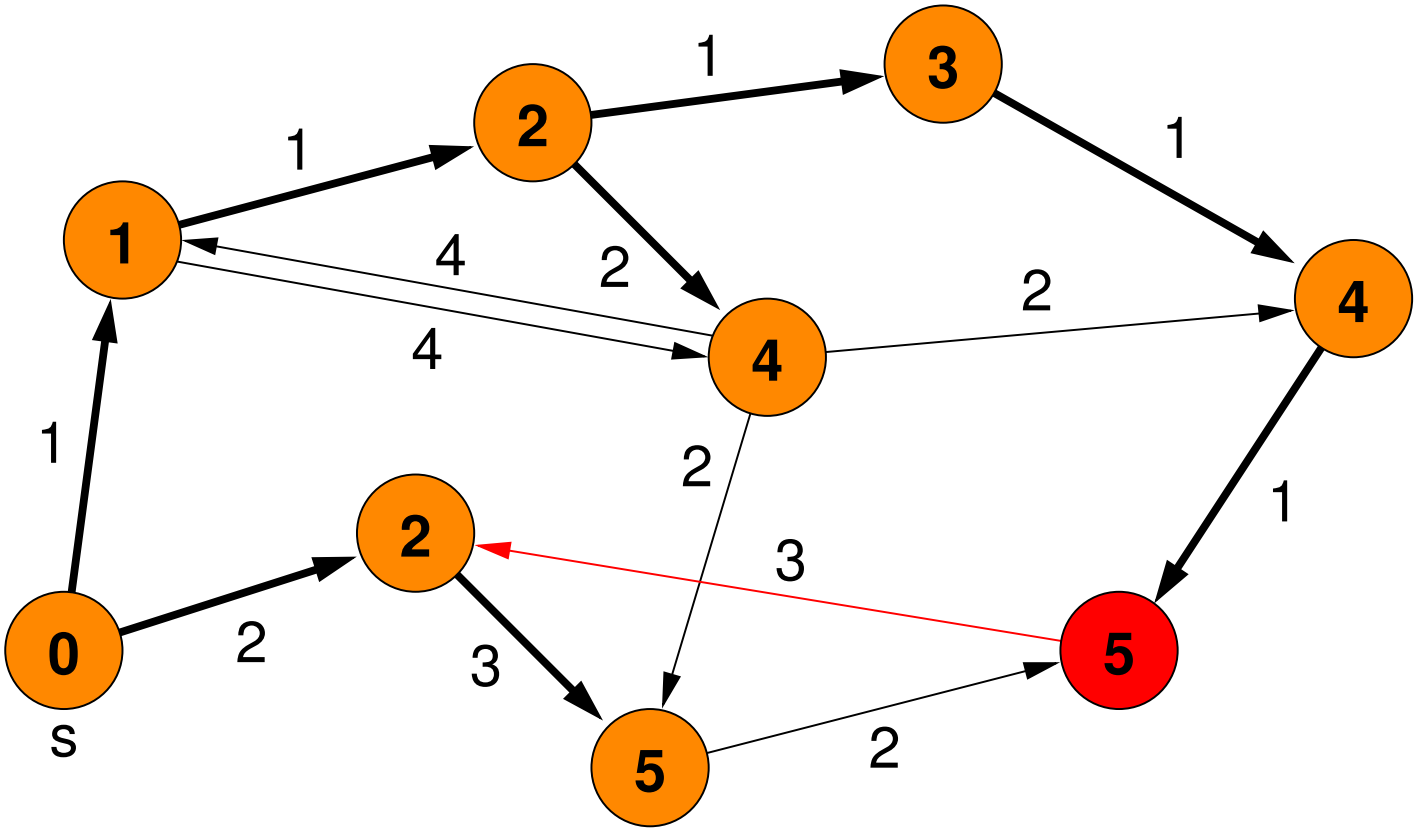
Dijkstra-Algorithmus

Beispiel:



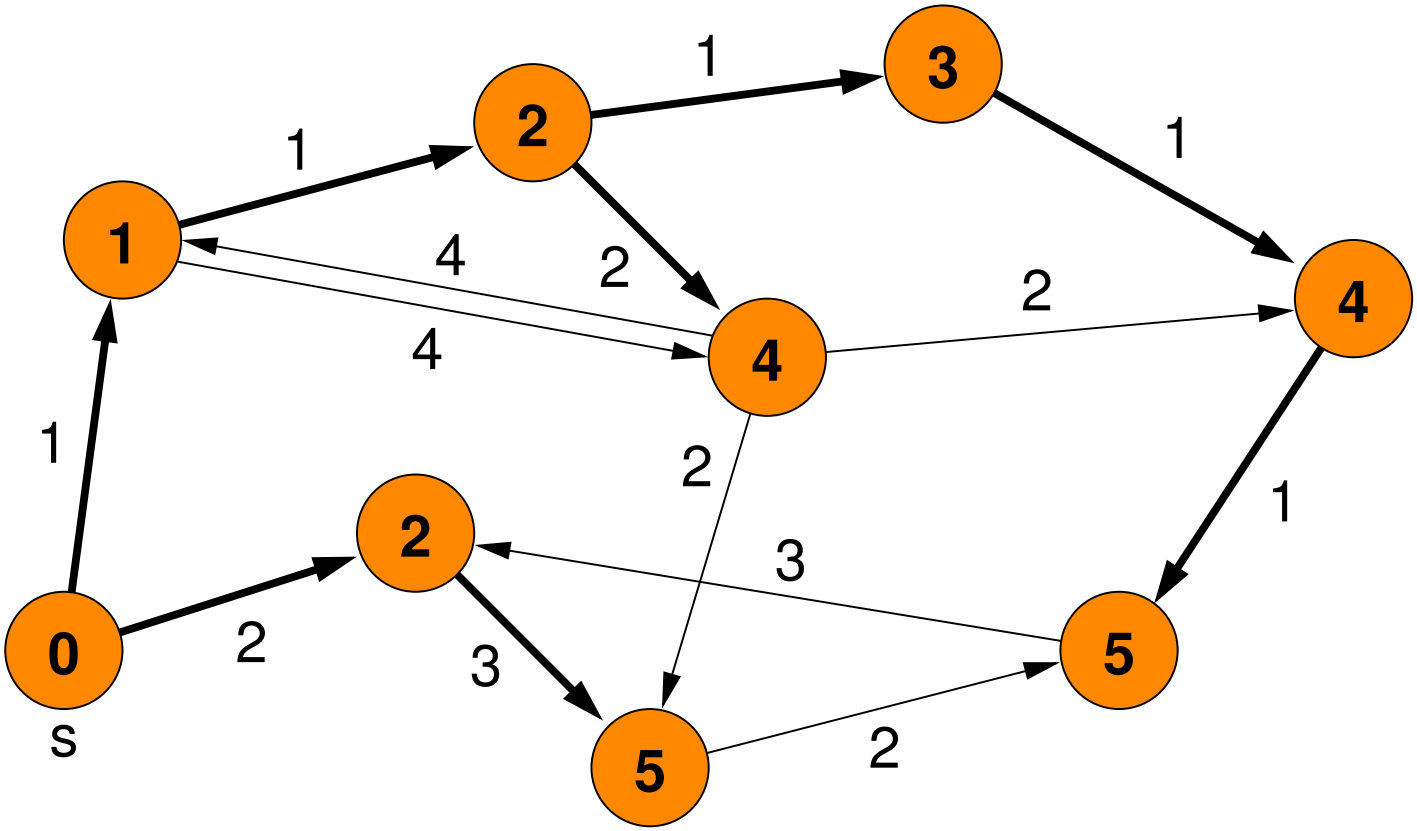
Dijkstra-Algorithmus

Beispiel:



Dijkstra-Algorithmus

Beispiel:



Dijkstra-Algorithmus

Korrektheit:

- Annahme: Algorithmus liefert für w einen zu kleinen Wert $d(s, w)$
- sei w der erste Knoten, für den die Distanz falsch festgelegt wird (kann nicht s sein, denn die Distanz $d(s, s)$ bleibt immer 0)
- kann nicht sein, weil $d(s, w)$ nur dann aktualisiert wird, wenn man über einen von s schon erreichten Knoten v mit Distanz $d(s, v)$ den Knoten w über die Kante (v, w) mit Distanz $d(s, v) + c(v, w)$ erreichen kann
- d.h. $d(s, v)$ müsste schon falsch gewesen sein (Widerspruch zur Annahme, dass w der erste Knoten mit falscher Distanz war)

Dijkstra-Algorithmus

- Annahme: Algorithmus liefert für w einen zu großen Wert $d(s, w)$
- sei w der Knoten mit der kleinsten (wirklichen) Distanz, für den der Wert $d(s, w)$ falsch festgelegt wird (wenn es davon mehrere gibt, der Knoten, für den die Distanz zuletzt festgelegt wird)
- kann nicht sein, weil $d(s, w)$ immer aktualisiert wird, wenn man über einen von s schon erreichten Knoten v mit Distanz $d(s, v)$ den Knoten w über die Kante (v, w) mit Distanz $d(s, v) + c(v, w)$ erreichen kann (dabei steht $d(s, v)$ immer schon fest, so dass auch die Länge eines kürzesten Wegs über v zu w richtig berechnet wird)
- d.h., entweder wurde auch der Wert von v falsch berechnet (Widerspruch zur Def. von w) oder die Distanz von v wurde noch nicht festgesetzt
- weil die berechneten Distanzwerte monoton wachsen, kann letzteres nur passieren, wenn v die gleiche Distanz hat wie w (auch Widerspruch zur Def. von w)

Dijkstra-Algorithmus

- Datenstruktur: Prioritätswarteschlange
  (z.B. Fibonacci Heap: amortisierte Komplexität $\mathcal{O}(1)$ für `insert` und `decreaseKey`, $\mathcal{O}(\log n)$ `deleteMin`)

- Komplexität:
  - $n \times \mathcal{O}(1)$ `insert`
  - $n \times \mathcal{O}(\log n)$ `deleteMin`
  - $m \times \mathcal{O}(1)$ `decreaseKey`

  $\Rightarrow \mathcal{O}(m + n \log n)$

- aber: nur für nichtnegative Kantengewichte(!)

Monotone Priority Queues

Beobachtung:

- aktuelles Distanz-Minimum der verbleibenden Knoten ist beim Dijkstra-Algorithmus monoton wachsend

Monotone Priority Queue

- Folge der entnommenen Elemente hat monoton steigende Werte
- effizientere Implementierung möglich, falls Kantengewichte ganzzahlig

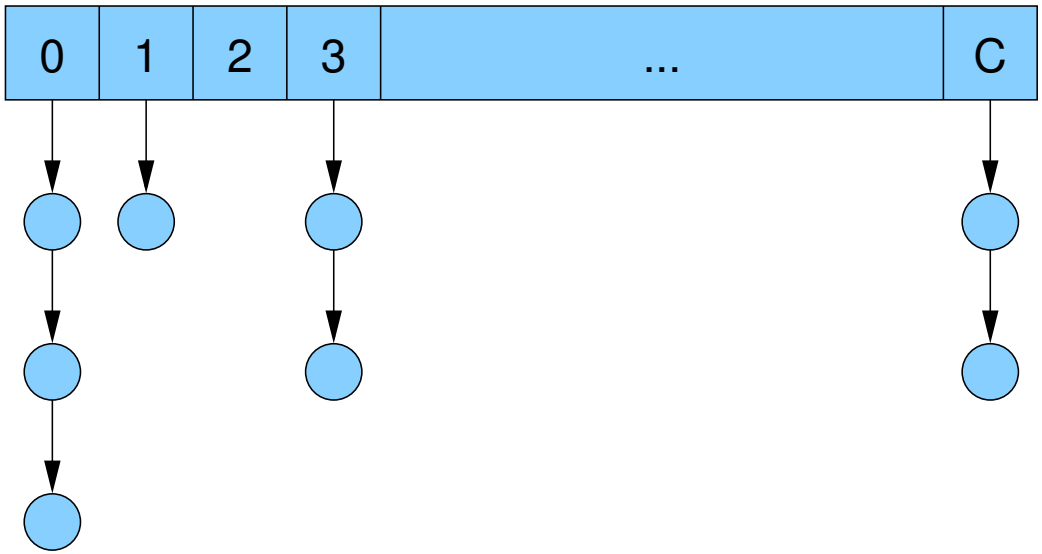
Annahme: alle Kantengewichte im Bereich $[0, C]$

Konsequenz für Dijkstra-Algorithmus:

$\Rightarrow$ enthaltene Distanzwerte immer im Bereich $[d, d + C]$

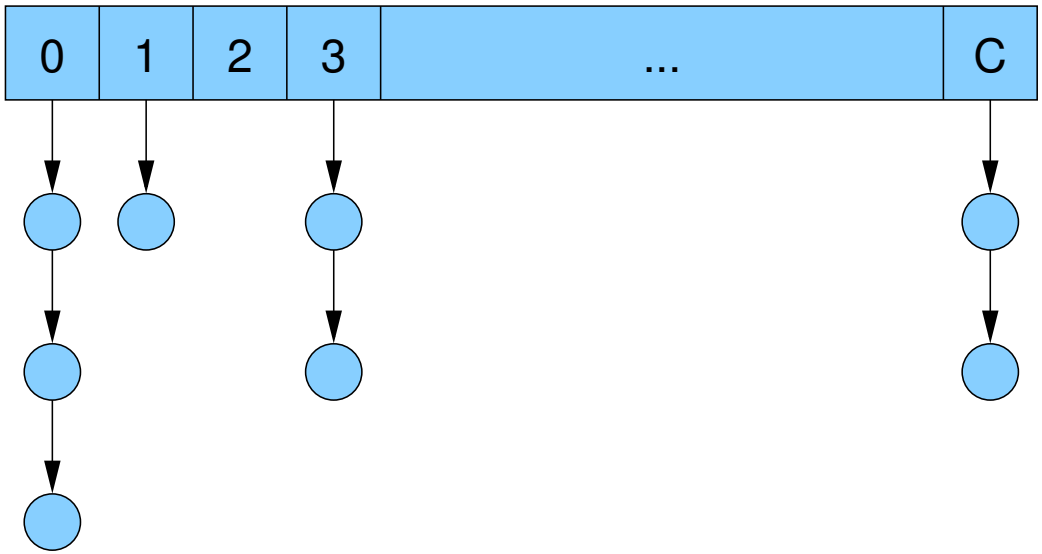
Bucket Queue

- Array B aus $C + 1$ Listen
- Variable d_{min} für aktuelles Distanzminimum mod$(C + 1)$



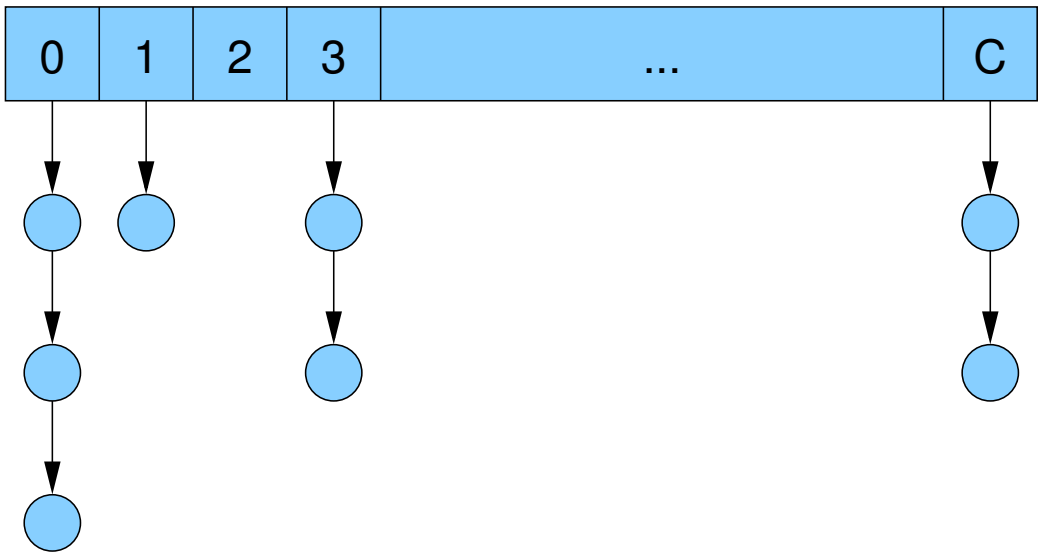
Bucket Queue

- jeder Knoten v mit aktueller Distanz $d[v]$ in Liste $B[d[v] \bmod (C+1)]$
- alle Knoten in Liste $B[d]$ haben dieselbe Distanz, weil alle aktuellen Distanzen im Bereich $[d, d+C]$ liegen



Bucket Queue / Operationen

- insert(v): fügt v in Liste $B[d[v] \bmod (C+1)]$ ein ($\mathcal{O}(1)$)
- decreaseKey(v): entfernt v aus momentaner Liste ($\mathcal{O}(1)$ falls Handle auf Listenelement in v gespeichert) und fügt v in Liste $B[d[v] \bmod (C+1)]$ ein ($\mathcal{O}(1)$)
- deleteMin(): solange $B[d_{\min}] = \emptyset$, setze $d_{\min} = (d_{\min} + 1) \bmod (C+1)$.
  Nimm dann einen Knoten u aus $B[d_{\min}]$ heraus ($\mathcal{O}(C)$)



Dijkstra mit Bucket Queue

- insert, decreaseKey: $\mathcal{O}(1)$
- deleteMin: $\mathcal{O}(C)$
- Dijkstra: $\mathcal{O}(m + C \cdot n)$
- lässt sich mit Radix Heaps noch verbessern
- verwendet exponentiell wachsende Bucket-Größen
- Details in der Vorlesung Effiziente Algorithmen und Datenstrukturen
- Laufzeit ist dann $\mathcal{O}(m + n \log C)$

Beliebige Graphen mit beliebigen Gewichten

Gegeben:

- beliebiger Graph mit beliebigen Kantengewichten

$\Rightarrow$ Anhängen einer Kante an einen Weg kann zur Verkürzung des Weges (Kantengewichtssumme) führen (wenn Kante negatives Gewicht hat)

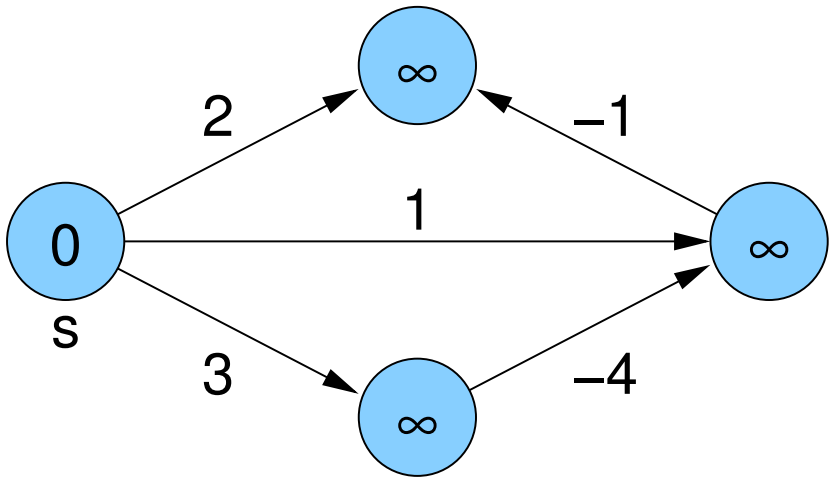
$\Rightarrow$ es kann negative Kreise und Knoten mit Distanz $-\infty$ geben

Problem:

- besuche Knoten eines kürzesten Weges in der richtigen Reihenfolge
- Dijkstra kann nicht mehr verwendet werden, weil Knoten nicht unbedingt in der Reihenfolge der kürzesten Distanz zum Startknoten s besucht werden

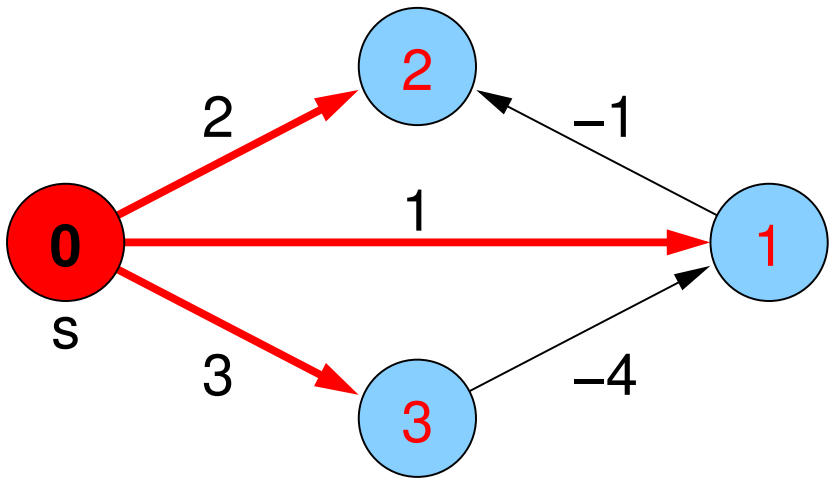
Beliebige Graphen mit beliebigen Gewichten

Gegenbeispiel für Dijkstra-Algorithmus:



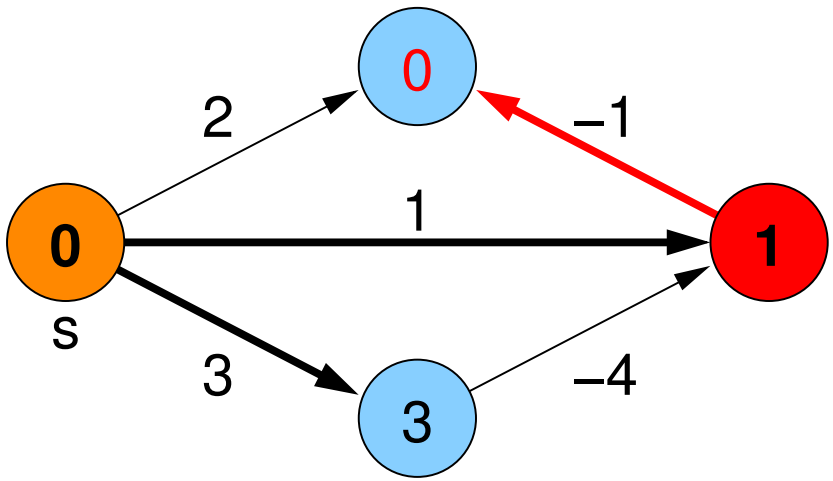
Beliebige Graphen mit beliebigen Gewichten

Gegenbeispiel für Dijkstra-Algorithmus:



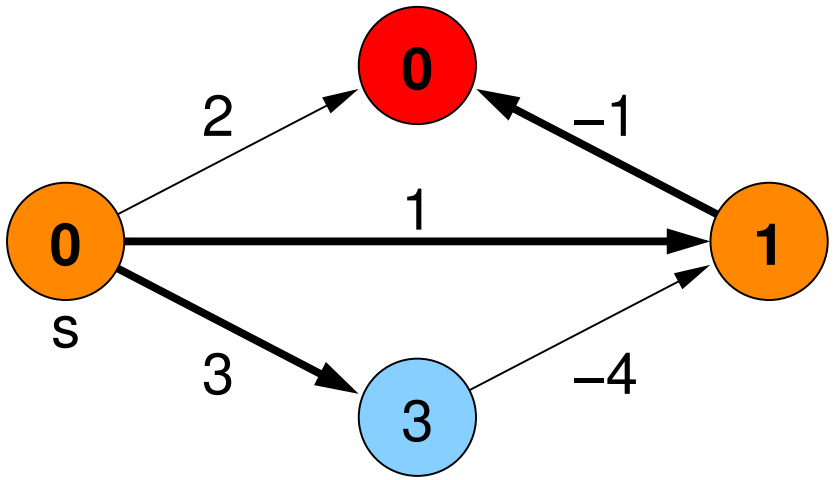
Beliebige Graphen mit beliebigen Gewichten

Gegenbeispiel für Dijkstra-Algorithmus:



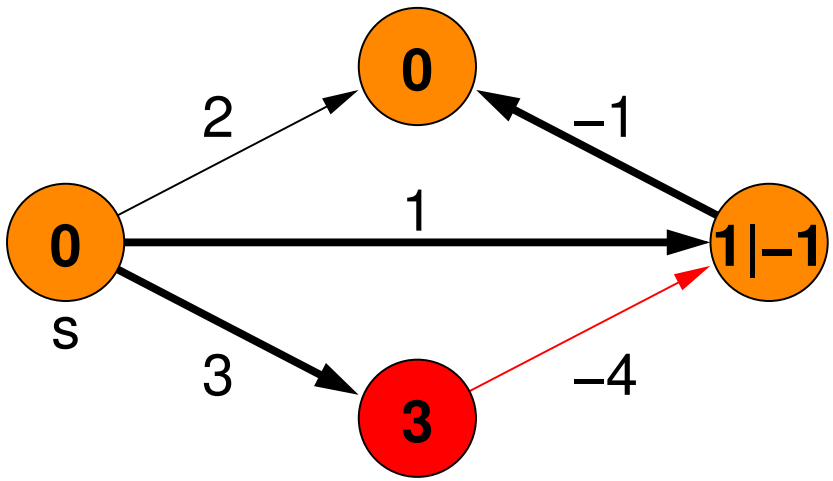
Beliebige Graphen mit beliebigen Gewichten

Gegenbeispiel für Dijkstra-Algorithmus:



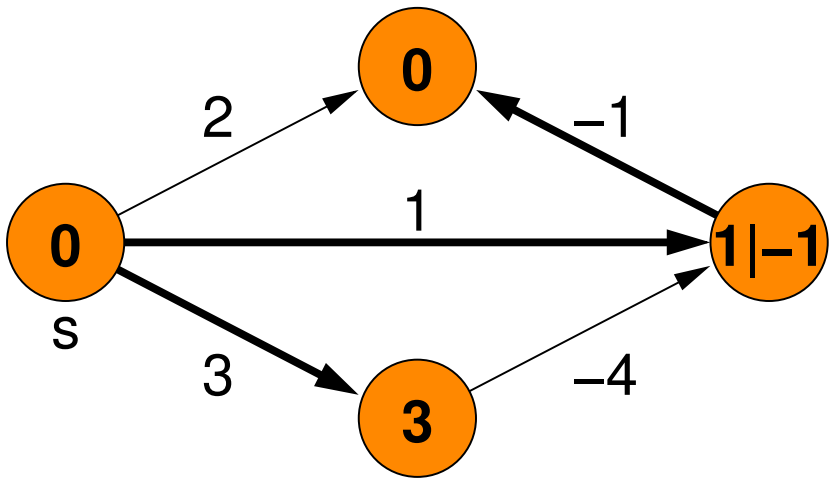
Beliebige Graphen mit beliebigen Gewichten

Gegenbeispiel für Dijkstra-Algorithmus:



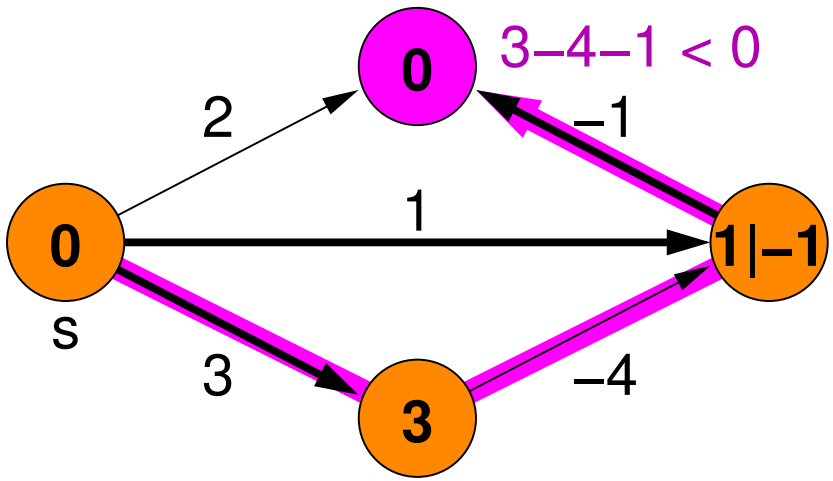
Beliebige Graphen mit beliebigen Gewichten

Gegenbeispiel für Dijkstra-Algorithmus:



Beliebige Graphen mit beliebigen Gewichten

Gegenbeispiel für Dijkstra-Algorithmus:



Beliebige Graphen mit beliebigen Gewichten

Lemma

Für jeden von s erreichbaren Knoten v mit $d(s,v) > -\infty$ gibt es einen einfachen Pfad (ohne Kreis) von s nach v der Länge $d(s,v)$.

Beweis.

Betrachte kürzesten Weg mit Kreis(en):

- Kreis mit Kantengewichtssumme > 0 nicht enthalten:
  Entfernen des Kreises würde Kosten verringern
- Kreis mit Kantengewichtssumme $= 0$:
  Entfernen des Kreises lässt Kosten unverändert
- Kreis mit Kantengewichtssumme < 0:
  Distanz von s ist $-\infty$

□

Bellman-Ford-Algorithmus

Folgerung

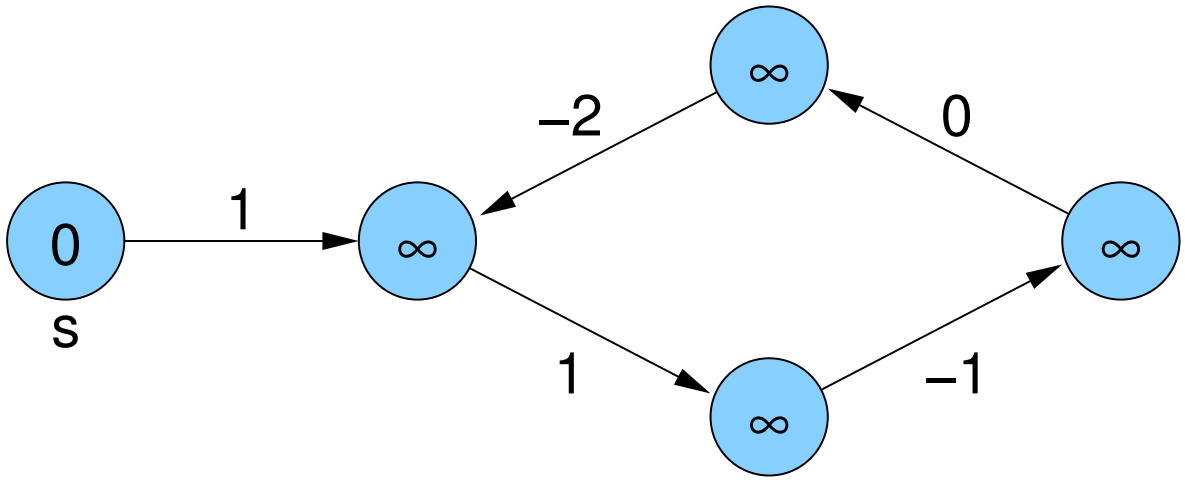
In einem Graph mit n Knoten gibt es für jeden erreichbaren Knoten v mit $d(s,v) > -\infty$ einen kürzesten Weg bestehend aus $< n$ Kanten zwischen s und v.

Strategie:

- anstatt kürzeste Pfade in Reihenfolge wachsender Gewichtssumme zu berechnen, betrachte sie in Reihenfolge steigender Kantenanzahl
- durchlaufe (n-1)-mal alle Kanten im Graph und aktualisiere die Distanz
- dann alle kürzesten Wege berücksichtigt

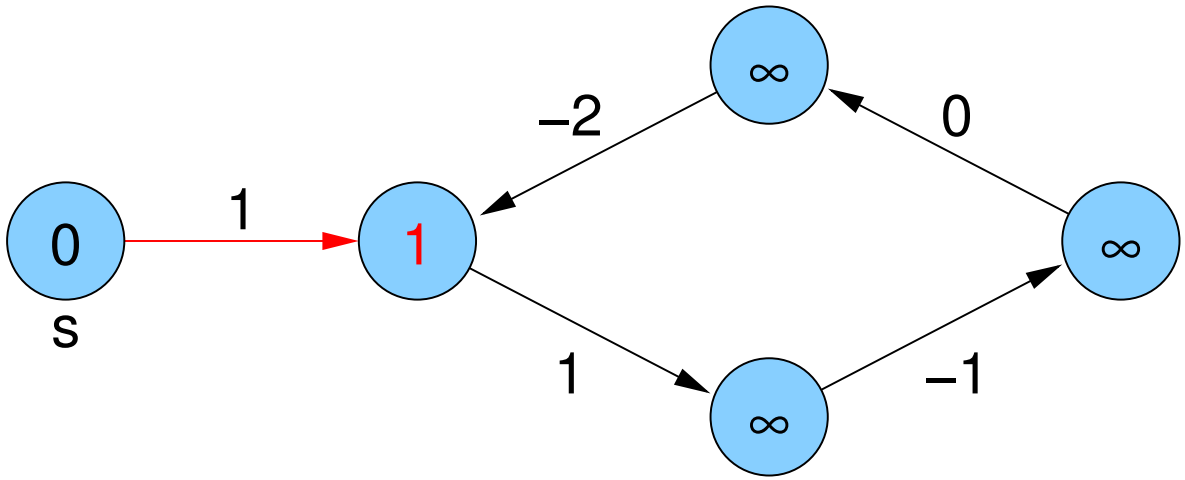
Bellman-Ford-Algorithmus

Problem: Erkennung negativer Kreise



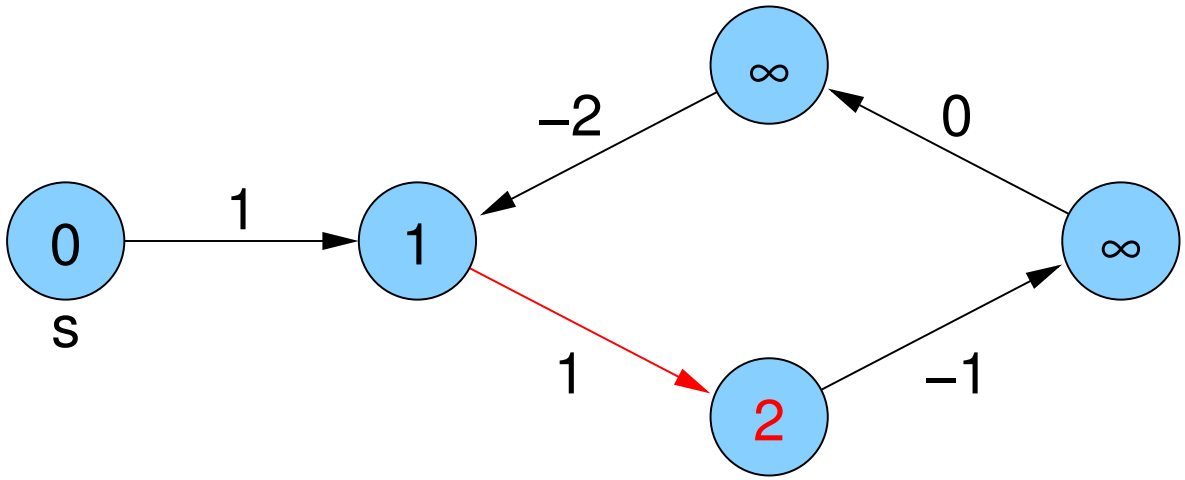
Bellman-Ford-Algorithmus

Problem: Erkennung negativer Kreise



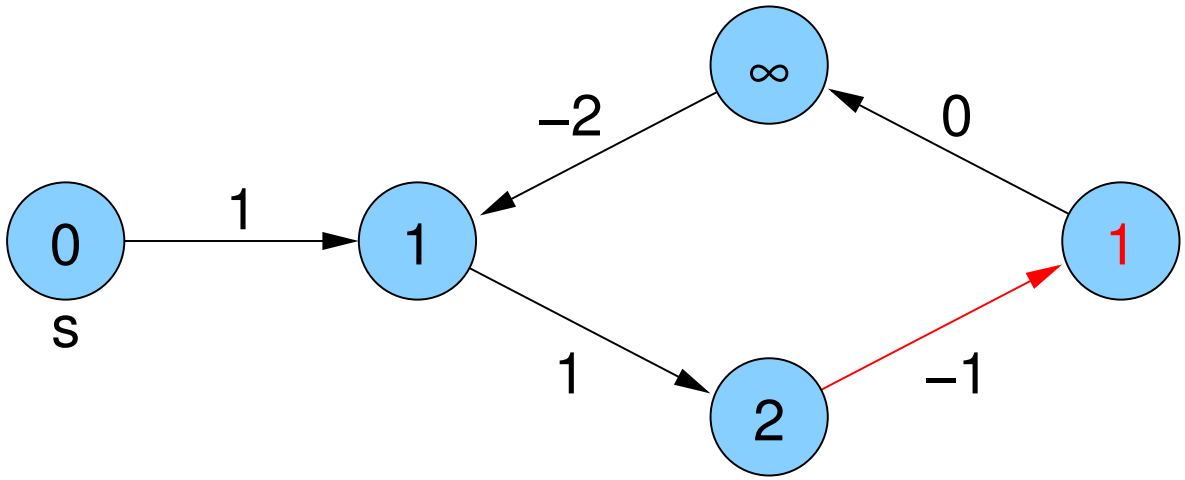
Bellman-Ford-Algorithmus

Problem: Erkennung negativer Kreise



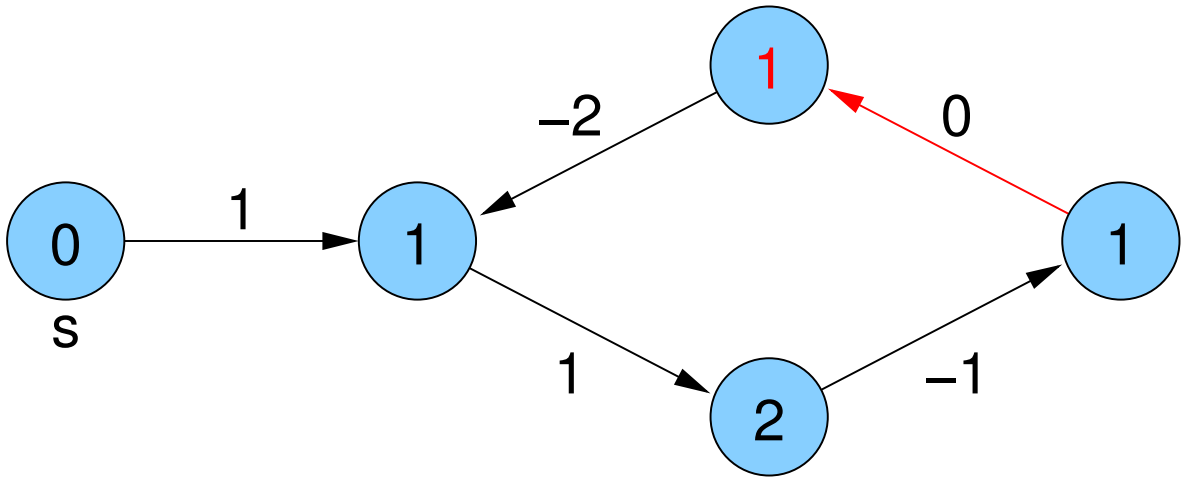
Bellman-Ford-Algorithmus

Problem: Erkennung negativer Kreise



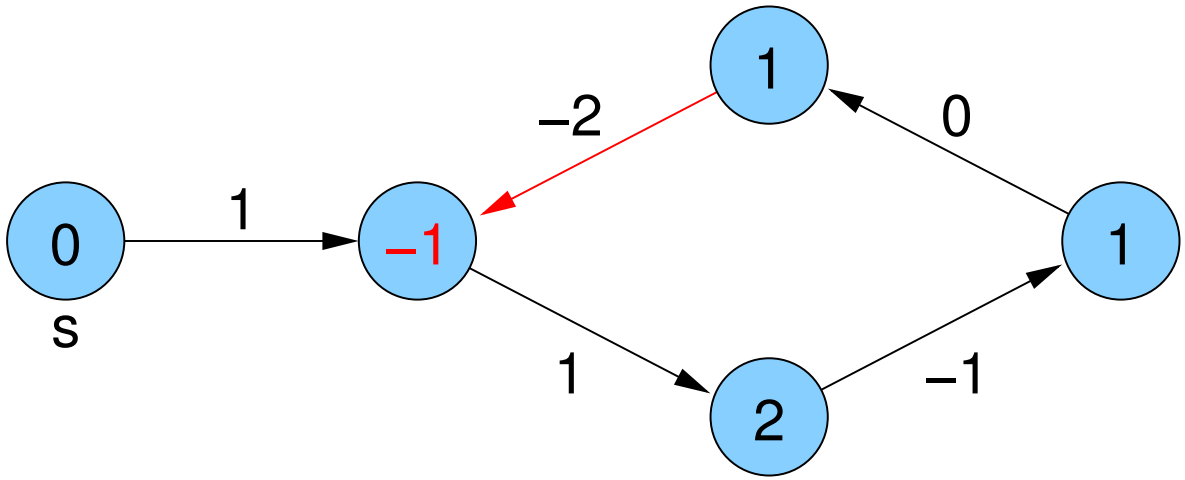
Bellman-Ford-Algorithmus

Problem: Erkennung negativer Kreise



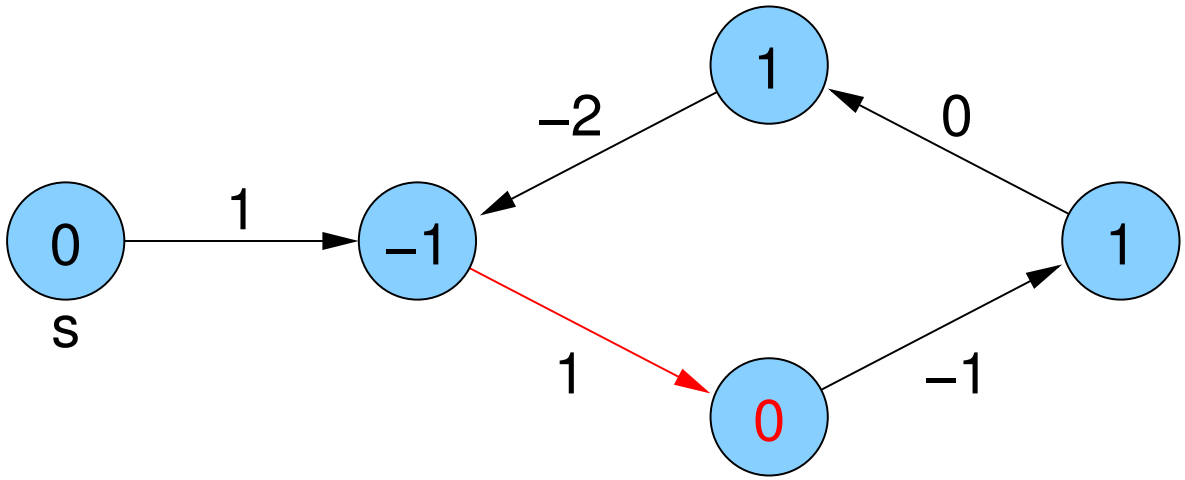
Bellman-Ford-Algorithmus

Problem: Erkennung negativer Kreise



Bellman-Ford-Algorithmus

Problem: Erkennung negativer Kreise



Bellman-Ford-Algorithmus

Keine Distanzverringerung mehr möglich:

- Annahme: zu einem Zeitpunkt gilt für alle Kanten (v, w) $d[v] + c(v, w) \geq d[w]$

$\Rightarrow$ (per Induktion) für alle Knoten w und jeden Weg p von s nach w gilt: $d[s] + c(p) \geq d[w]$

- falls sichergestellt, dass zu jedem Zeitpunkt für kürzesten Weg p von s nach w gilt $d[w] \geq c(p)$, dann ist $d[w]$ zum Schluss genau die Länge eines kürzesten Pfades von s nach w (also korrekte Distanz)

Bellman-Ford-Algorithmus

Zusammenfassung:

- keine Distanzerniedrigung mehr möglich
  ($d[v] + c(v,w) \geq d[w]$ für alle w):
  fertig, alle $d[w]$ korrekt für alle w

- Distanzerniedrigung möglich selbst noch in n-ter Runde
  ($d[v] + c(v,w) < d[w]$ für ein w):
  Es gibt einen negativen Kreis, also Knoten w mit Distanz $-\infty$.

Bellman-Ford-Algorithmus

```
BellmanFord(Node s) {
  foreach (v ∈ V) d[v] = ∞;
  d[s] = 0;   parent[s] = s;
  for (int i = 0; i < n – 1; i++) {   // n – 1 Runden
    foreach (e = (v, w) ∈ E)
      if (d[v] + c(e) < d[w]) {   // kürzerer Weg?
        d[w] = d[v] + c(e);
        parent[w] = v;
      }
  }
  foreach (e = (v, w) ∈ E)
    if (d[v] + c(e) < d[w]) {   // kürzerer Weg in n-ter Runde?
      infect(w);
}
```

Bellman-Ford-Algorithmus

```
infect(Node v) {   // −∞-Knoten
  if (d[v] > −∞) {
    d[v] = −∞;
    foreach (e = (v, w) ∈ E)
      infect(w);
  }
}
```

Gesamtlaufzeit: $\mathcal{O}(m \cdot n)$

Bellman-Ford-Algorithmus

Bestimmung der Knoten mit Distanz $-\infty$:

- betrachte alle Knoten, die in der n-ten Phase noch Distanzverbesserung erfahren
- aus jedem Kreis mit negativem Gesamtgewicht muss mindestens ein Knoten dabei sein
- jeder von diesen Knoten aus erreichbare Knoten muss Distanz $-\infty$ bekommen
- das erledigt hier die infect-Funktion
- wenn ein Knoten zweimal auftritt (d.h. der Wert ist schon $-\infty$), wird die Rekursion abgebrochen

Bellman-Ford-Algorithmus

Bestimmung eines negativen Zyklus:

- bei den oben genannten Knoten sind vielleicht auch Knoten, die nur an negativen Kreisen über ausgehende Kanten angeschlossen sind, die selbst aber nicht Teil eines negativen Kreises sind
- Rückwärtsverfolgung der parent-Werte, bis sich ein Knoten wiederholt
- Kanten vom ersten bis zum zweiten Auftreten bilden einen negativen Zyklus

Bellman-Ford-Algorithmus

Ursprüngliche Idee der Updates vorläufiger Distanzwerte stammt von Lester R. Ford Jr.

Verbesserung (Richard E. Bellman / Edward F. Moore):

- verwalte FIFO-Queue von Knoten, zu denen ein kürzerer Pfad gefunden wurde und deren Nachbarn am anderen Ende ausgehender Kanten noch auf kürzere Wege geprüft werden müssen
- wiederhole: nimm ersten Knoten aus der Queue und prüfe für jede ausgehende Kante die Distanz des Nachbarn

  falls kürzerer Weg gefunden, aktualisiere Distanzwert des Nachbarn und hänge ihn an Queue an (falls nicht schon enthalten)
- Phase besteht immer aus Bearbeitung der Knoten, die am Anfang des Algorithmus (bzw. der Phase) in der Queue sind
  (dabei kommen während der Phase schon neue Knoten ans Ende der Queue)

Kürzeste einfache Pfade bei beliebigen Kantengewichten

Achtung!

Fakt

Die Suche nach kürzesten einfachen Pfaden
(also ohne Knotenwiederholungen / Kreise)
in Graphen mit beliebigen Kantengewichten
(also möglichen negativen Kreisen)
ist ein $\mathcal{NP}$-vollständiges Problem.

(Man könnte Hamilton-Pfad-Suche damit lösen.)

All Pairs Shortest Paths (APSP)

gegeben:

- Graph mit beliebigen Kantengewichten,
  der aber keine negativen Kreise enthält

gesucht:

- Distanzen / kürzeste Pfade zwischen allen Knotenpaaren

Naive Strategie:

- n-mal Bellman-Ford-Algorithmus
  (jeder Knoten einmal als Startknoten)

$\Rightarrow$ $O(n^2 \cdot m)$

APSP / Kantengewichte

Bessere Strategie:

- reduziere n Aufrufe des Bellman-Ford-Algorithmus auf n Aufrufe des Dijkstra-Algorithmus

Problem:

- Dijkstra-Algorithmus funktioniert nur für nichtnegative Kantengewichte

Lösung:

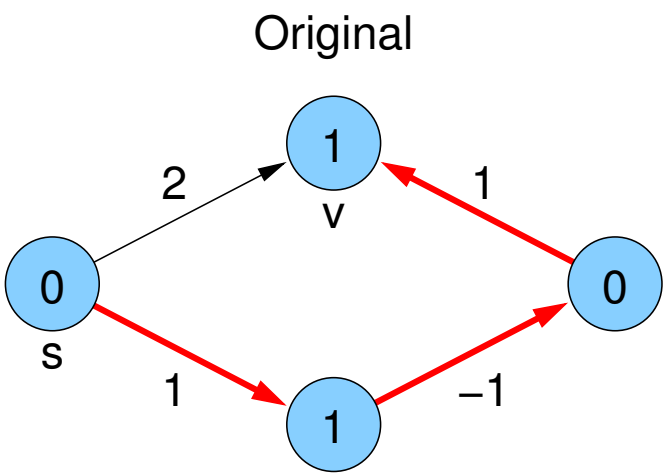
- Umwandlung in nichtnegative Kantenkosten ohne Verfälschung der kürzesten Wege

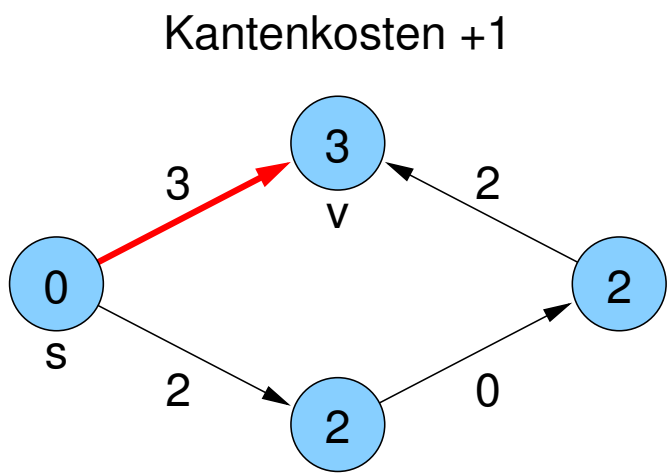
Naive Modifikation der Kantengewichte

Naive Idee:

- negative Kantengewichte eliminieren, indem auf jedes Kantengewicht der gleiche Wert c addiert wird

$\Rightarrow$ verfälscht kürzeste Pfade





Knotenpotential

Sei $\Phi : V \mapsto \mathbb{R}$ eine Funktion, die jedem Knoten ein Potential zuordnet.

Modifizierte Kantenkosten von $e = (v, w)$:

$$\bar{c}(e) = \Phi(v) + c(e) - \Phi(w)$$

Lemma

Seien p und q Wege von v nach w in G.

$c(p)$ und $c(q)$ bzw. $\bar{c}(p)$ und $\bar{c}(q)$ seien die aufsummierten Kosten bzw. modifizierten Kosten der Kanten des jeweiligen Pfads.

Dann gilt für jedes Potential Φ:

$$\bar{c}(p) < \bar{c}(q) \quad \Leftrightarrow \quad c(p) < c(q)$$

Knotenpotential

Beweis.

Sei $p = (v_1, \ldots, v_k)$ beliebiger Weg und $\forall i: \; e_i = (v_i, v_{i+1}) \in E$

Es gilt:

$$\begin{aligned} \bar{c}(p) &= \sum_{i=1}^{k-1} \bar{c}(e_i) \\ &= \sum_{i=1}^{k-1} (\Phi(v_i) + c(e_i) - \Phi(v_{i+1})) \\ &= \Phi(v_1) + c(p) - \Phi(v_k) \end{aligned}$$

d.h. modifizierte Kosten eines Pfads hängen nur von ursprünglichen Pfadkosten und vom Potential des Anfangs- und Endknotens ab.
(Im Lemma ist $v_1 = v$ und $v_k = w$) □

Potential für nichtnegative Kantengewichte

Lemma

Annahme:

- *Graph hat keine negativen Kreise*
- *alle Knoten von s aus erreichbar*

Sei für alle Knoten v das Potential $\Phi(v) = d(s,v)$.

Dann gilt für alle Kanten e: $\bar{c}(e) \geq 0$

Beweis.

- für alle Knoten v gilt nach Annahme: $d(s,v) \in \mathbb{R}$ (also $\neq \pm\infty$)
- für jede Kante $e = (v,w)$ ist

$$\begin{aligned} d(s,v) + c(e) &\geq d(s,w) \\ d(s,v) + c(e) - d(s,w) &\geq 0 \end{aligned}$$

□

Johnson-Algorithmus für APSP

- füge neuen Knoten s und Kanten (s, v) für alle v hinzu mit $c(s, v) = 0$

$\Rightarrow$ alle Knoten erreichbar

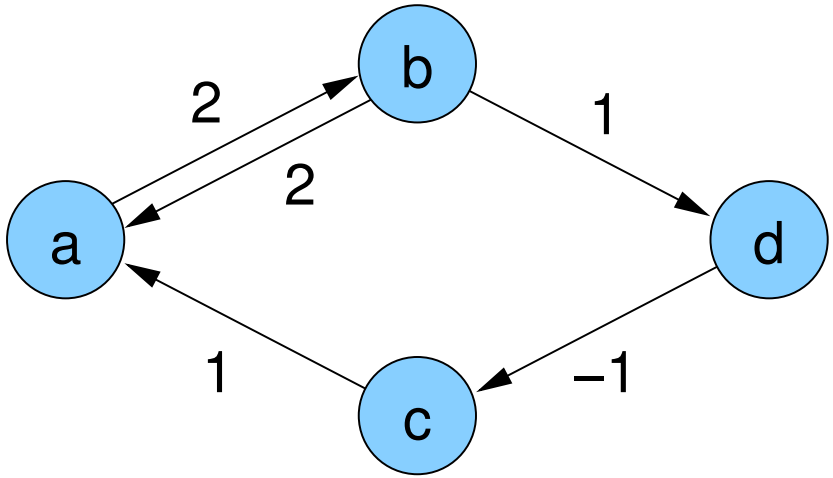
- berechne $d(s, v)$ mit Bellman-Ford-Algorithmus
- setze $\Phi(v) = d(s, v)$ für alle v
- berechne modifizierte Kosten $\bar{c}(e)$

$\Rightarrow$ $\bar{c}(e) \geq 0$, kürzeste Wege sind noch die gleichen

- berechne für alle Knoten v die Distanzen $\bar{d}(v, w)$ mittels Dijkstra-Algorithmus mit modifizierten Kantenkosten auf dem Graph ohne Knoten s
- berechne korrekte Distanzen $d(v, w) = \bar{d}(v, w) + \Phi(w) - \Phi(v)$

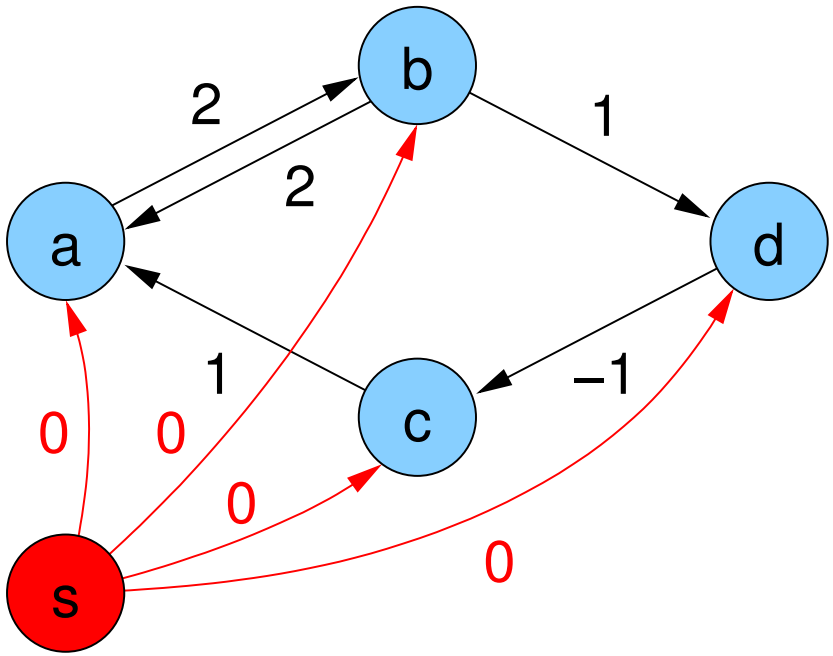
Johnson-Algorithmus für APSP

Beispiel:



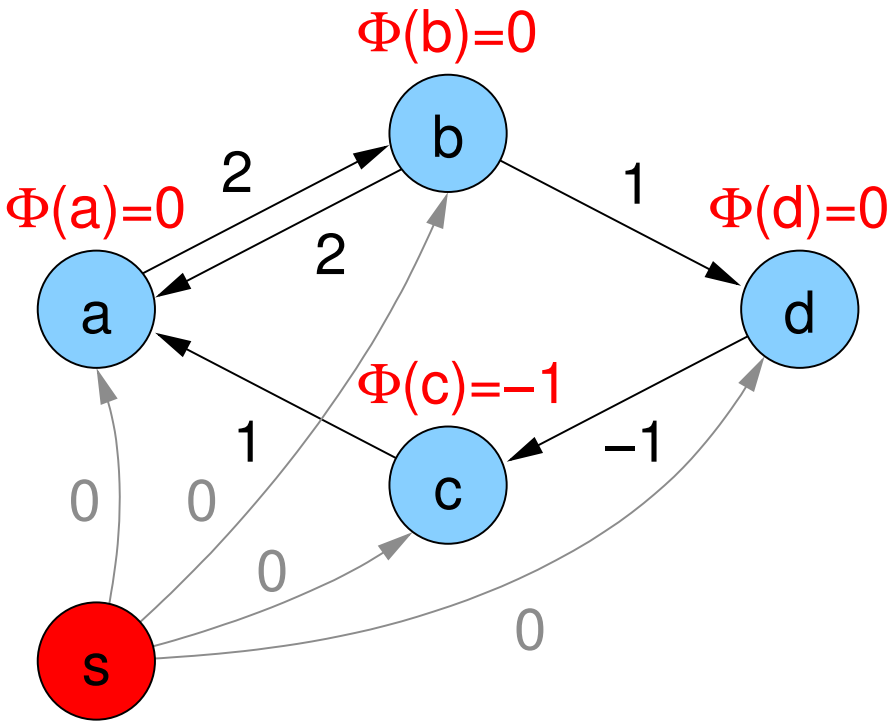
Johnson-Algorithmus für APSP

1. künstlicher Startknoten s:



Johnson-Algorithmus für APSP

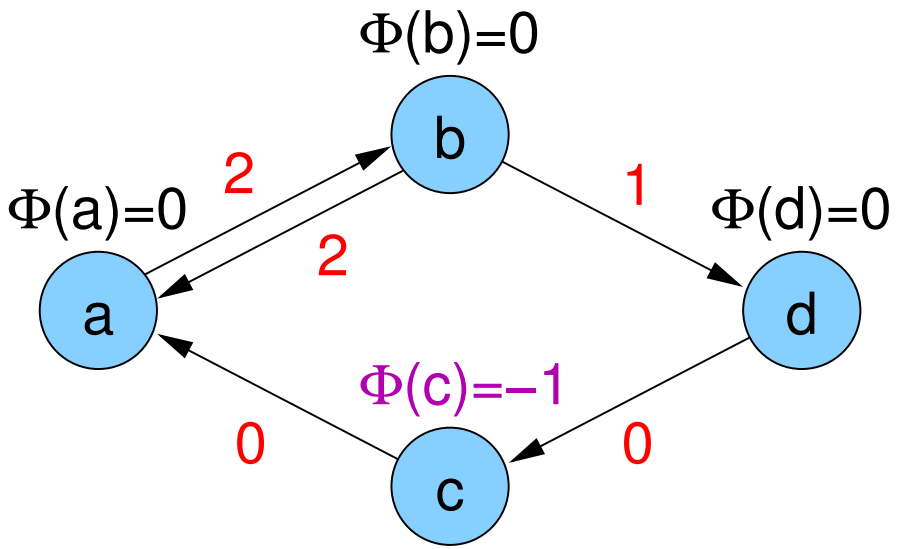
2. Bellman-Ford-Algorithmus auf s:



Johnson-Algorithmus für APSP

3. $\bar{c}(e)$-Werte für alle $e = (v, w)$ berechnen:

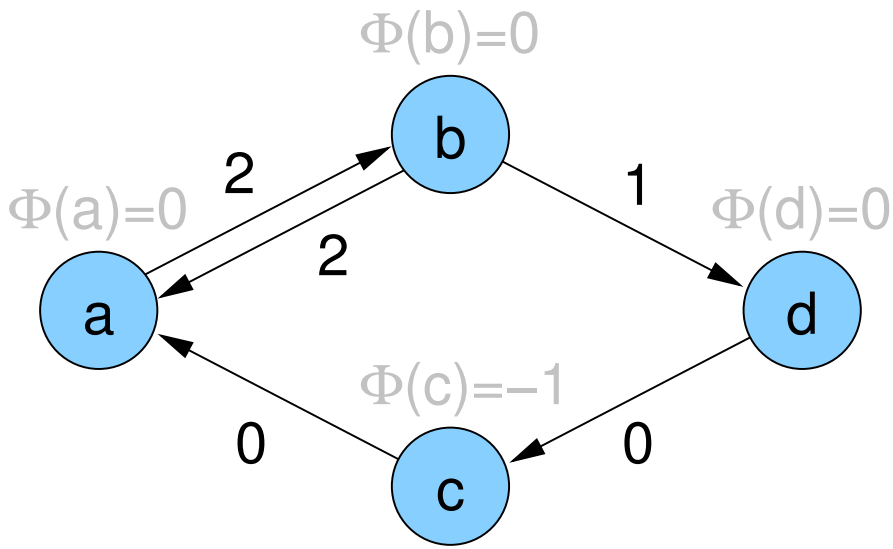
$$\bar{c}(e) = \Phi(v) + c(e) - \Phi(w)$$



Johnson-Algorithmus für APSP

4. Distanzen $\bar{d}$ mit modifizierten Kantengewichten via Dijkstra:

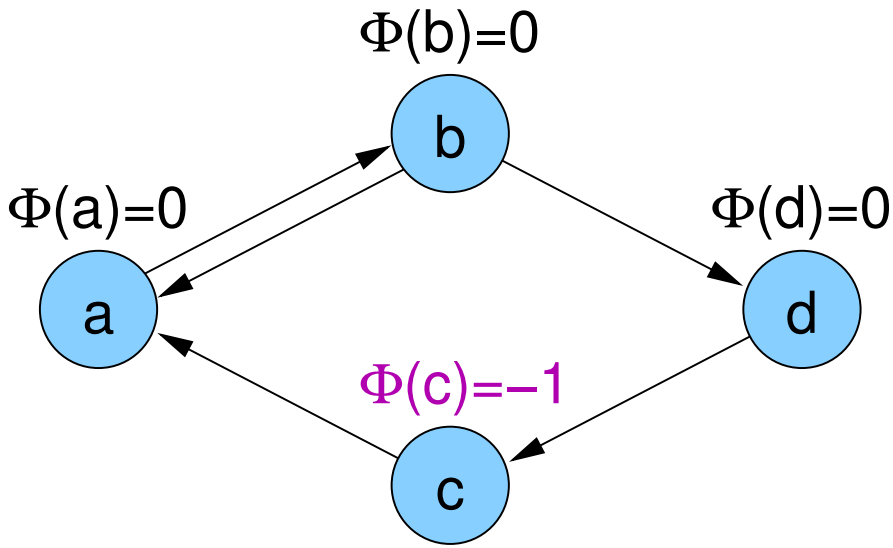
$\bar{\mathbf{d}}$	a	b	c	d
a	0	2	3	3
b	1	0	1	1
c	0	2	0	3
d	0	2	0	0



Johnson-Algorithmus für APSP

5. korrekte Distanzen berechnen: $d(v,w) = \bar{d}(v,w) + \Phi(w) - \Phi(v)$

d	*a*	*b*	*c*	*d*
a	0	2	2	3
b	1	0	0	1
c	1	3	0	4
d	0	2	−1	0



Johnson-Algorithmus für APSP

Laufzeit:

$$\begin{aligned} T_{\text{Johnson}}(n,m) &= \mathcal{O}(T_{\text{Bellman-Ford}}(n+1, m+n) + n \cdot T_{\text{Dijkstra}}(n,m)) \\ &= \mathcal{O}((m+n)\cdot(n+1) + n\cdot(n\log n + m)) \\ &= \mathcal{O}(m\cdot n + n^2 \log n) \end{aligned}$$

(bei Verwendung von Fibonacci Heaps)

Floyd-Warshall-Algorithmus für APSP

Grundlage:

- geht der kürzeste Weg von u nach w über v, dann sind auch die beiden Teile von u nach v und von v nach w kürzeste Pfade zwischen diesen Knoten
- Annahme: alle kürzesten Wege bekannt, die nur über Zwischenknoten mit Index kleiner als k gehen

$\Rightarrow$ kürzeste Wege über Zwischenknoten mit Indizes bis einschließlich k können leicht berechnet werden:

  - entweder der schon bekannte Weg über Knoten mit Indizes kleiner als k
  - oder über den Knoten mit Index k (hier im Algorithmus der Knoten v)

Floyd-Warshall-Algorithmus für APSP

Algorithmus Floyd-Warshall: löst APSP-Problem

Eingabe : Graph $G = (V, E)$, $c : E \mapsto \mathbb{R}$
Ausgabe : Distanzen $d(u, v)$ zwischen allen $u, v \in V$

for $u, v \in V$ **do**
  $d(u, v) = \infty$; pred$(u, v) = \perp$;
for $v \in V$ **do** $d(v, v) = 0$;
for $(u, v) \in E$ **do**
  $d(u, v) = c(u, v)$; pred$(u, v) = u$;
for $v \in V$ **do**
  for $\{u, w\} \in V \times V$ **do**
    if $d(u, w) > d(u, v) + d(v, w)$ **then**
      $d(u, w) = d(u, v) + d(v, w)$;
      pred$(u, w) =$ pred(v, w);

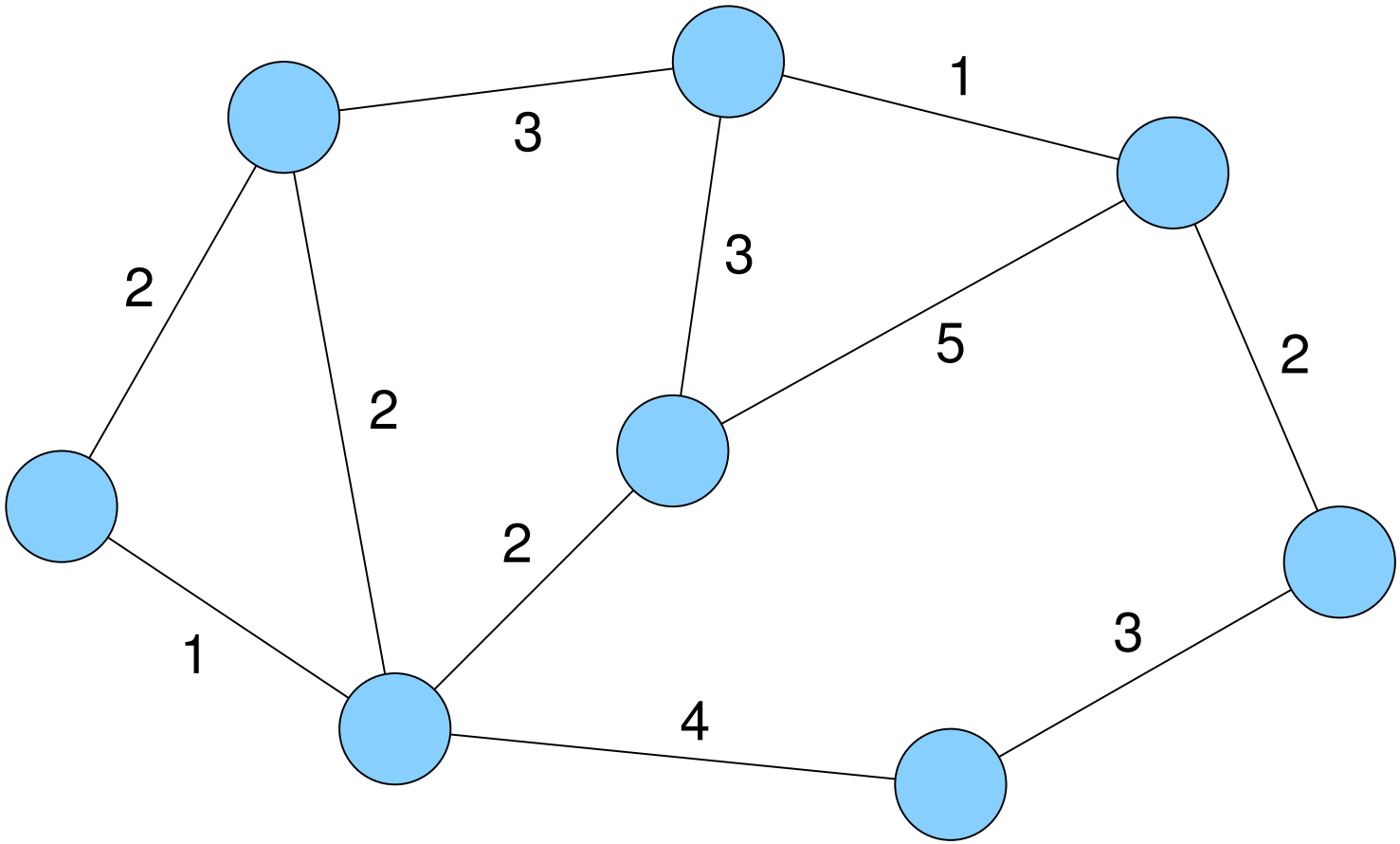
Floyd-Warshall-Algorithmus für APSP

- Komplexität: $\mathcal{O}(n^3)$
- funktioniert auch, wenn Kanten mit negativem Gewicht existieren
- Kreise negativer Länge werden nicht direkt erkannt und verfälschen das Ergebnis, sind aber indirekt am Ende an negativen Diagonaleinträgen der Distanzmatrix erkennbar

Übersicht

Minimaler Spannbaum

Frage: Welche Kanten nehmen, um mit minimalen Kosten alle Knoten zu verbinden?



Minimaler Spannbaum

Eingabe:

- ungerichteter Graph $G = (V, E)$
- Kantenkosten $c : E \mapsto \mathbb{R}_+$

Ausgabe:

- Kantenteilmenge $T \subseteq E$, so dass Graph (V, T) verbunden und $c(T) = \sum_{e \in T} c(e)$ minimal

Beobachtung:

- T formt immer einen Baum
  (wenn Kantengewichte echt positiv)

$\Rightarrow$ Minimaler Spannbaum (MSB) / Minimum Spanning Tree (MST)

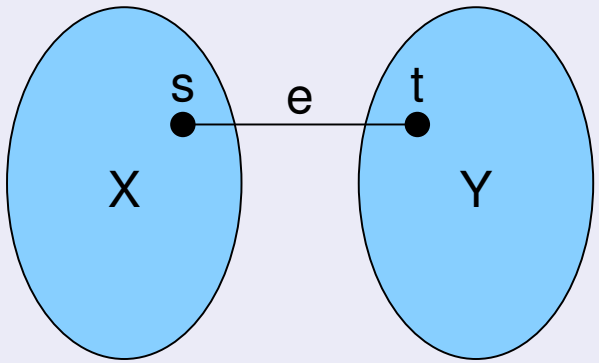
Minimaler Spannbaum

Lemma

Sei

- (X, Y) *eine Partition von* V *(d.h.* $X \cup Y = V$ *und* $X \cap Y = \emptyset$*) und*
- $e = \{s, t\}$ *eine Kante mit minimalen Kosten mit* $s \in X$ *und* $t \in Y$.

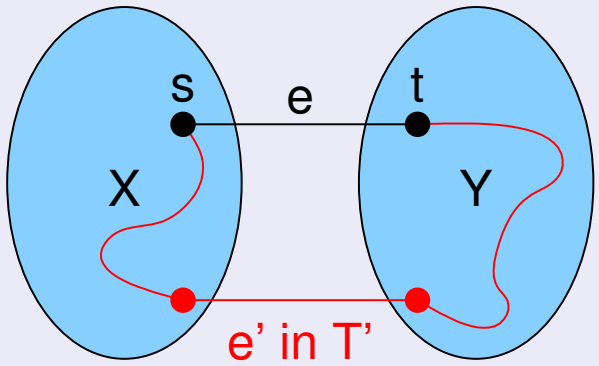
Dann gibt es einen minimalen Spannbaum T, der e enthält.



Minimaler Spannbaum

Beweis.

- gegeben X, Y und $e = \{s, t\}$: (X, Y)-Kante minimaler Kosten
- betrachte beliebigen MSB T', der e nicht enthält
- betrachte Verbindung zwischen s und t in T', darin muss es mindestens eine Kante e' zwischen X und Y geben



- Ersetzung von e' durch e führt zu Baum T'', der höchstens Kosten von MSB T' hat (also auch ein MSB ist)

□

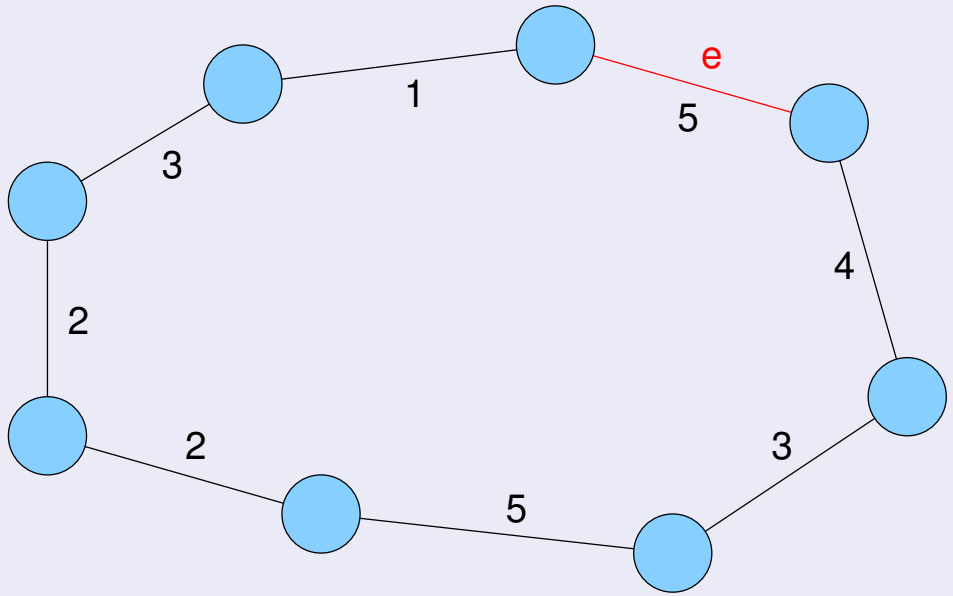
Minimaler Spannbaum

Lemma

Betrachte

- *beliebigen Kreis C in G*
- *eine Kante e in C mit maximalen Kosten*

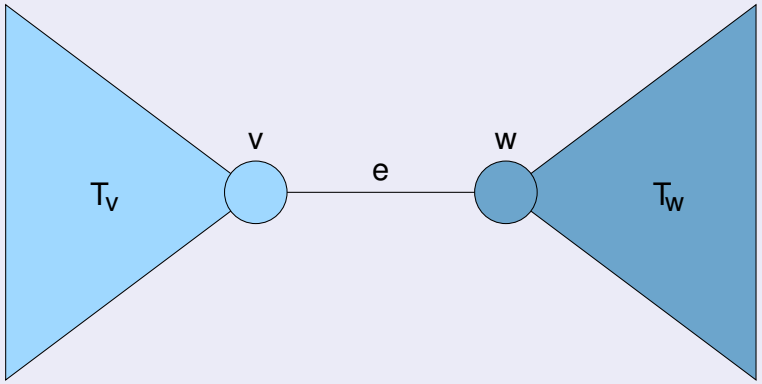
Dann ist jeder MSB in G ohne e auch ein MSB in G

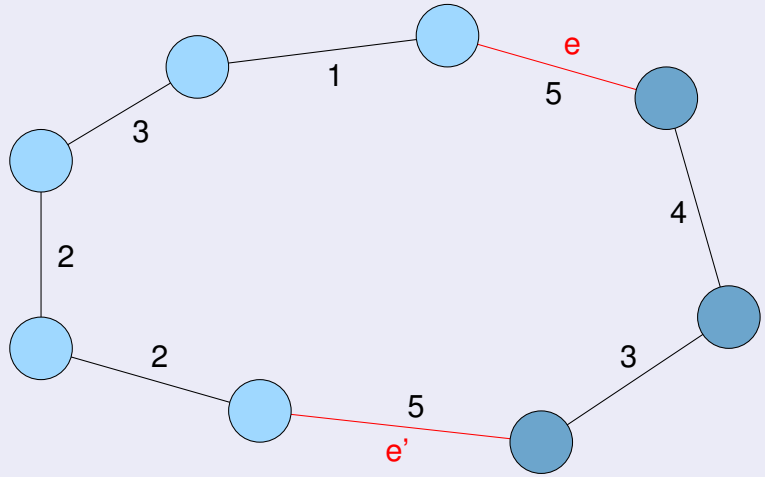


Minimaler Spannbaum

Beweis.

- betrachte beliebigen MSB T in G
- Annahme: T enthält e



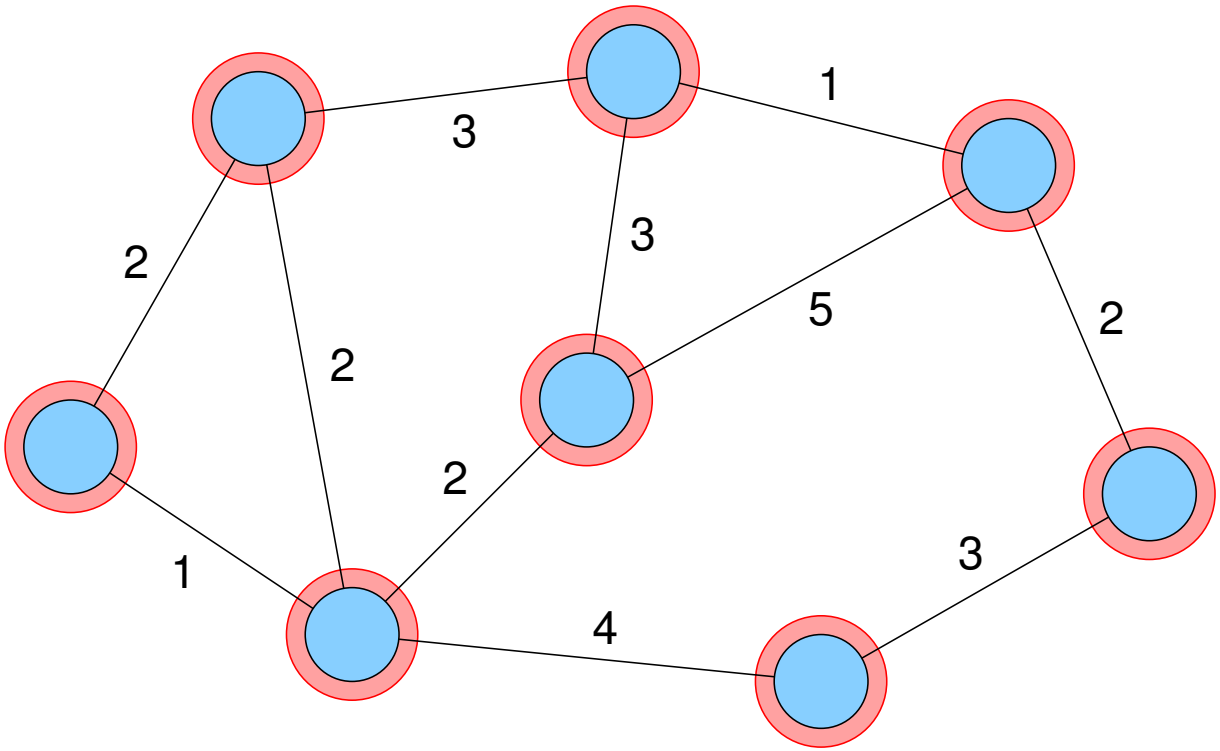


- es muss (mindestens) eine weitere Kante e' in C geben, die einen Knoten aus T_v mit einem Knoten aus T_w verbindet
- Ersetzen von e durch e' ergibt einen Baum T' dessen Gewicht nicht größer sein kann als das von T, also ist T' auch MSB

Minimaler Spannbaum

Regel:

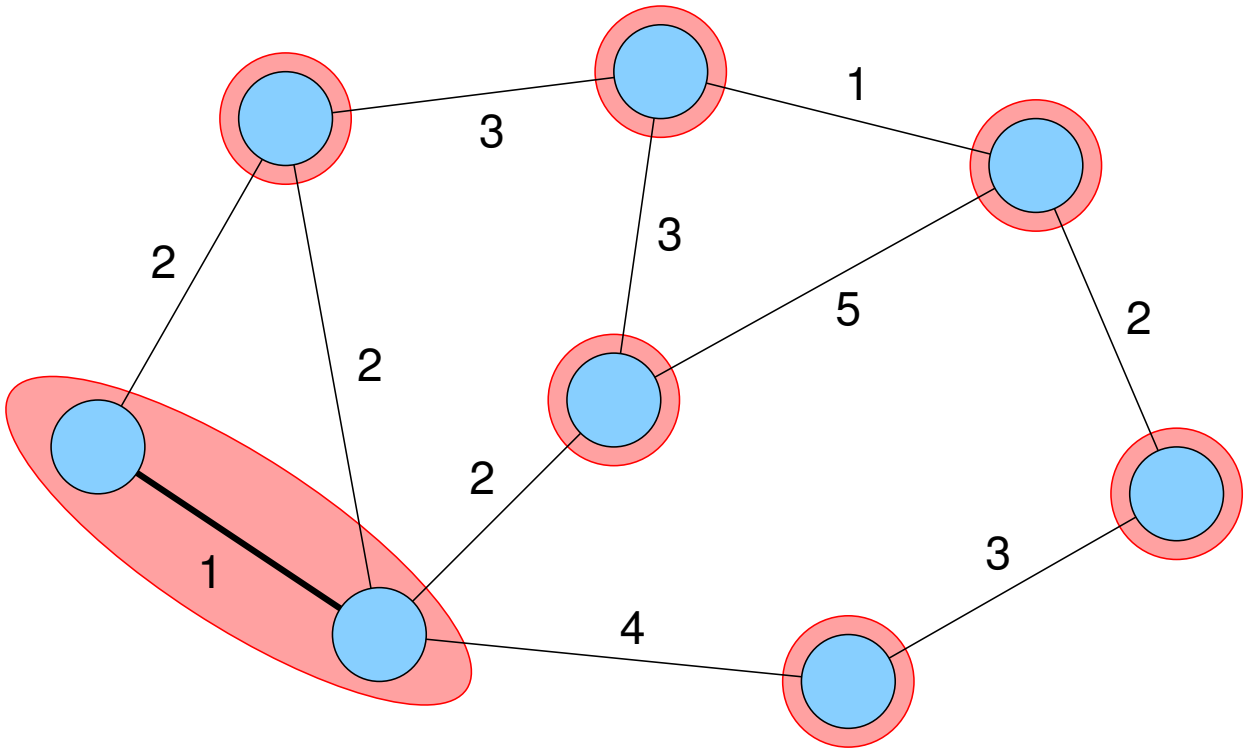
- wähle wiederholt Kante mit minimalen Kosten, die zwei Zusammenhangskomponenten verbindet
- bis nur noch eine Zusammenhangskomponente übrig ist



Minimaler Spannbaum

Regel:

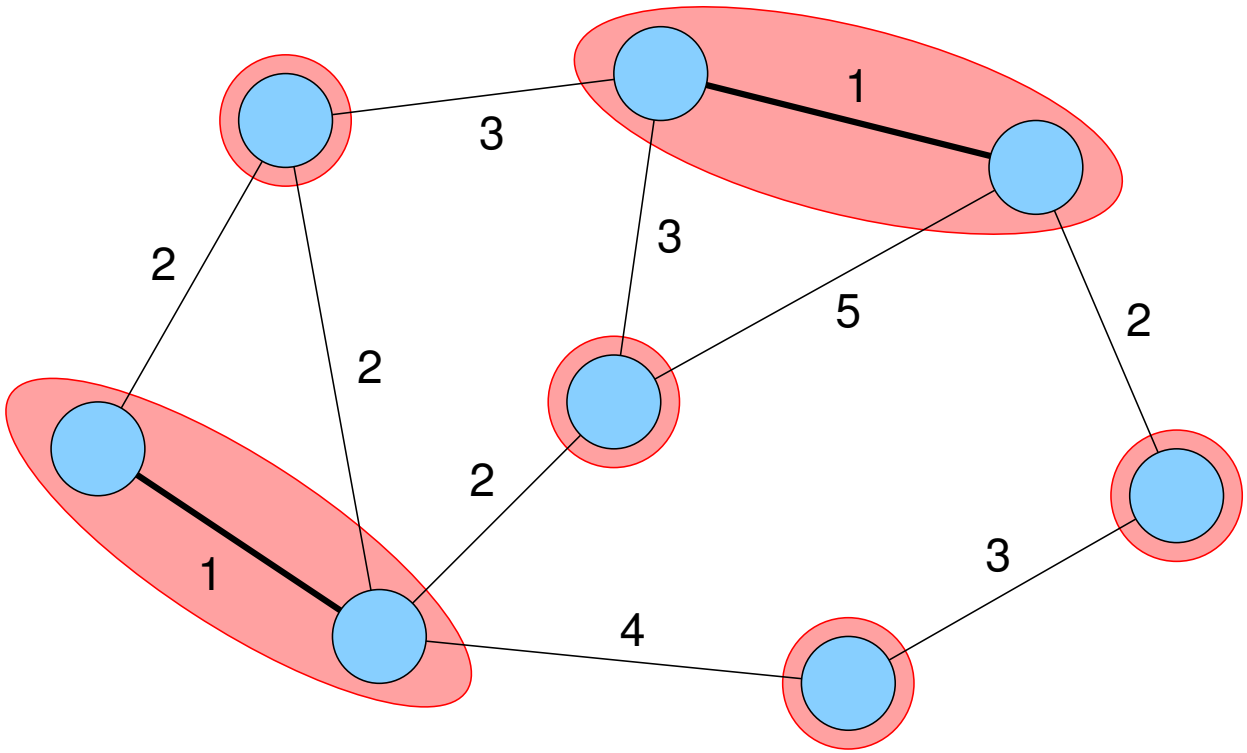
- wähle wiederholt Kante mit minimalen Kosten, die zwei Zusammenhangskomponenten verbindet
- bis nur noch eine Zusammenhangskomponente übrig ist



Minimaler Spannbaum

Regel:

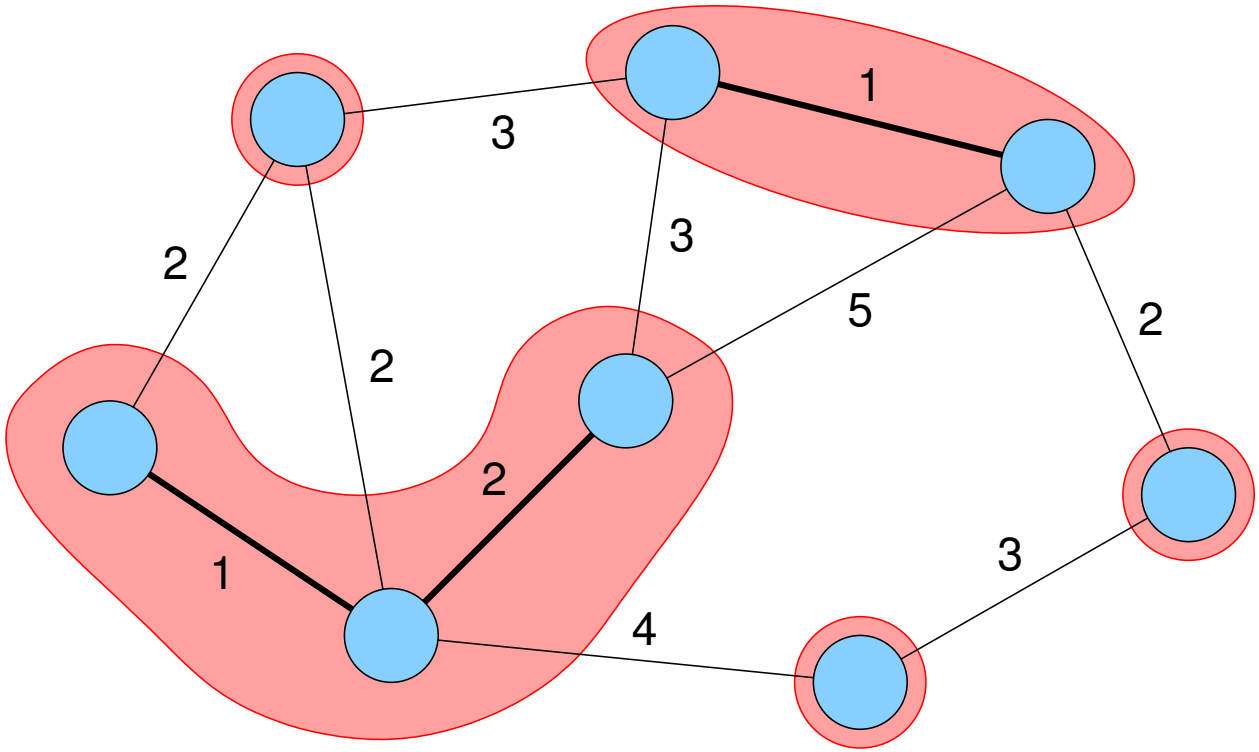
- wähle wiederholt Kante mit minimalen Kosten, die zwei Zusammenhangskomponenten verbindet
- bis nur noch eine Zusammenhangskomponente übrig ist



Minimaler Spannbaum

Regel:

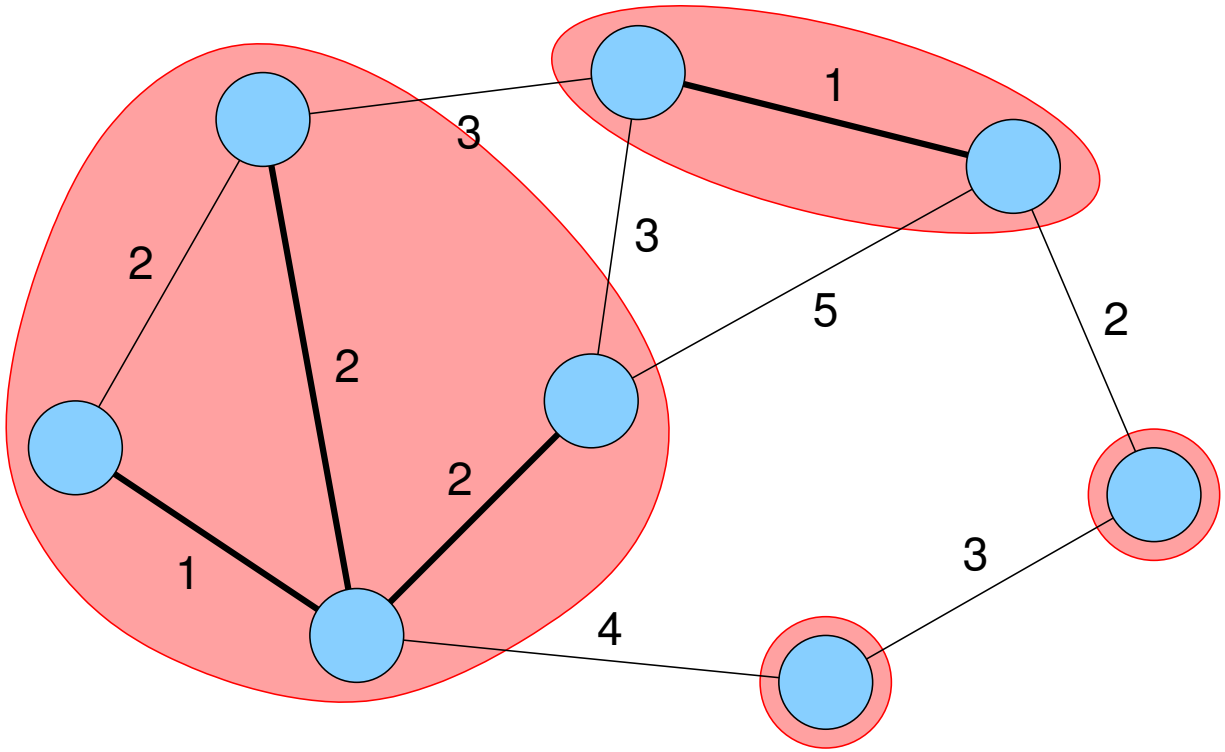
- wähle wiederholt Kante mit minimalen Kosten, die zwei Zusammenhangskomponenten verbindet
- bis nur noch eine Zusammenhangskomponente übrig ist



Minimaler Spannbaum

Regel:

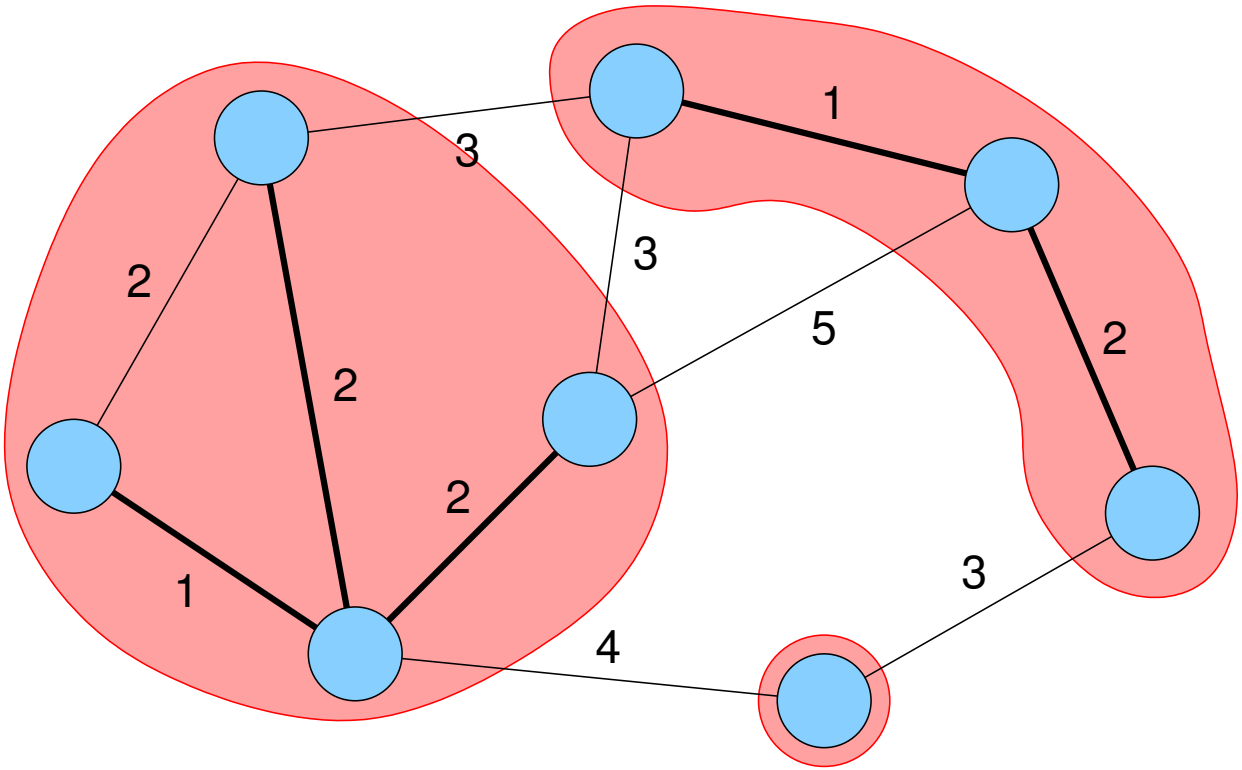
- wähle wiederholt Kante mit minimalen Kosten, die zwei Zusammenhangskomponenten verbindet
- bis nur noch eine Zusammenhangskomponente übrig ist



Minimaler Spannbaum

Regel:

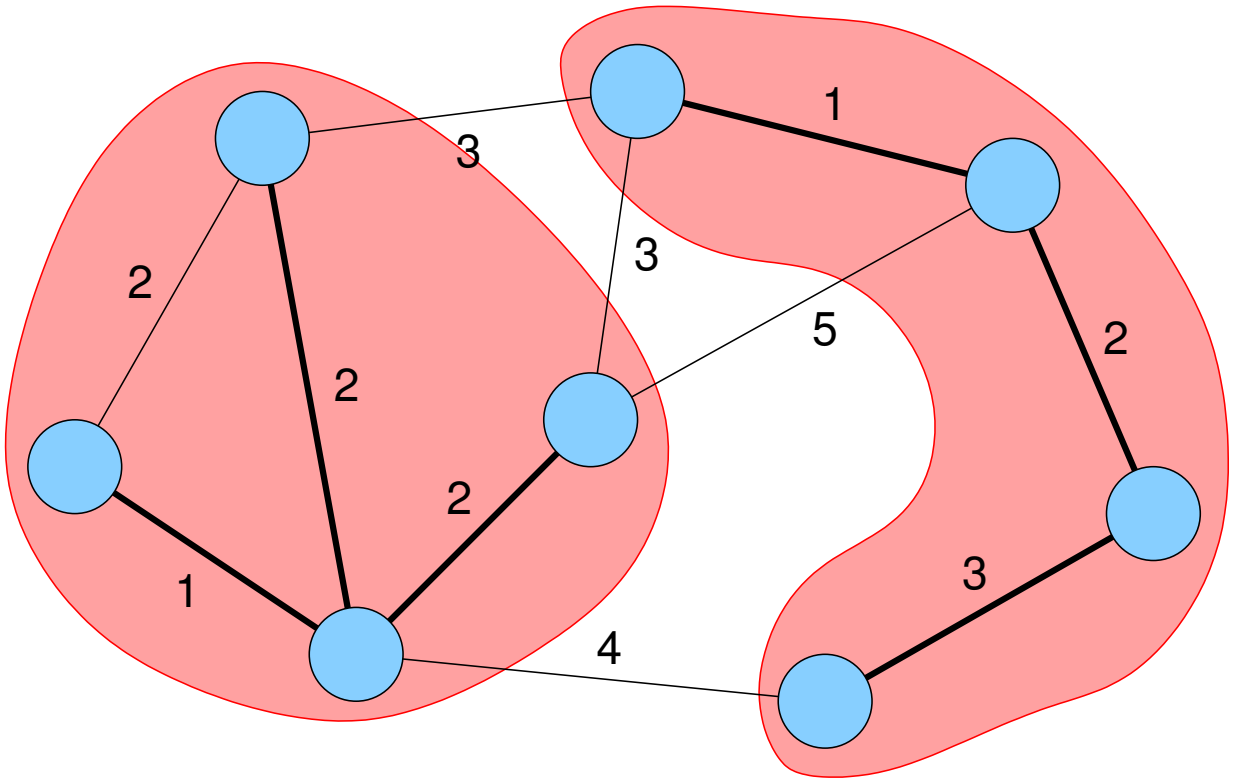
- wähle wiederholt Kante mit minimalen Kosten, die zwei Zusammenhangskomponenten verbindet
- bis nur noch eine Zusammenhangskomponente übrig ist



Minimaler Spannbaum

Regel:

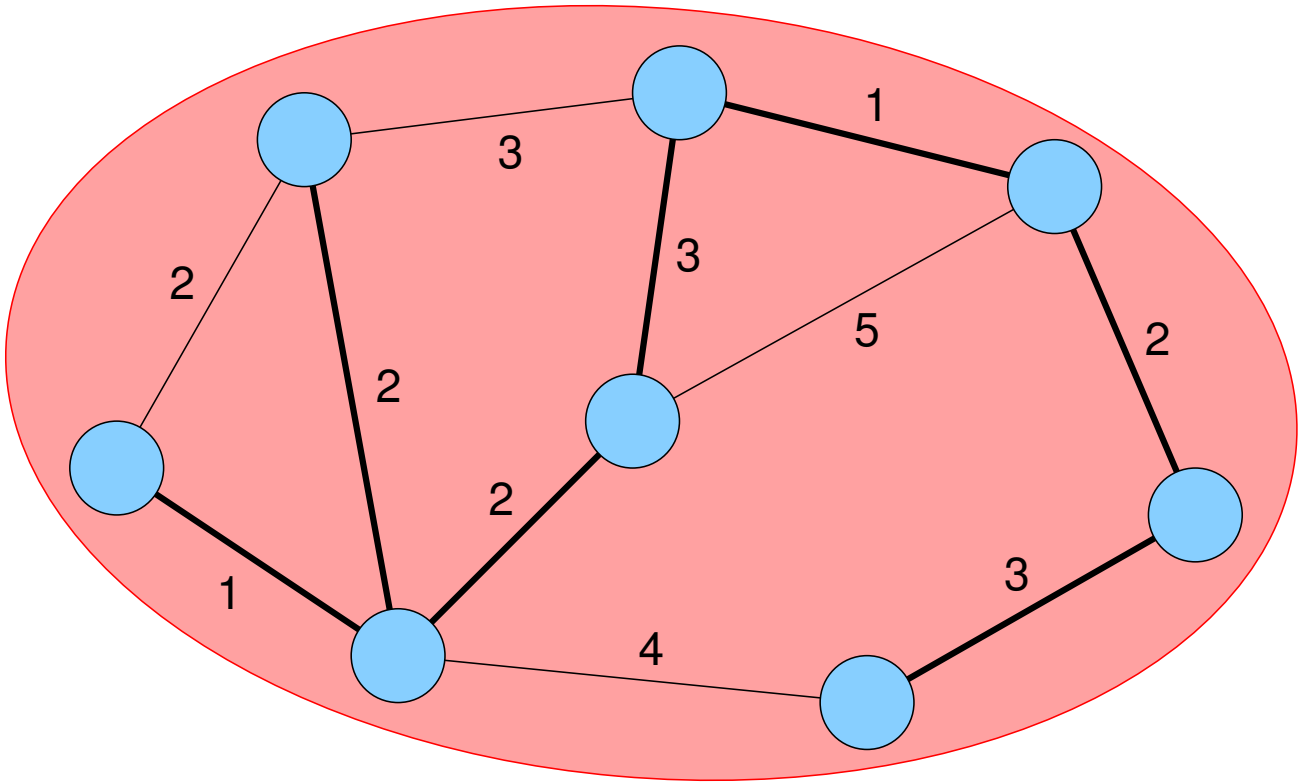
- wähle wiederholt Kante mit minimalen Kosten, die zwei Zusammenhangskomponenten verbindet
- bis nur noch eine Zusammenhangskomponente übrig ist



Minimaler Spannbaum

Regel:

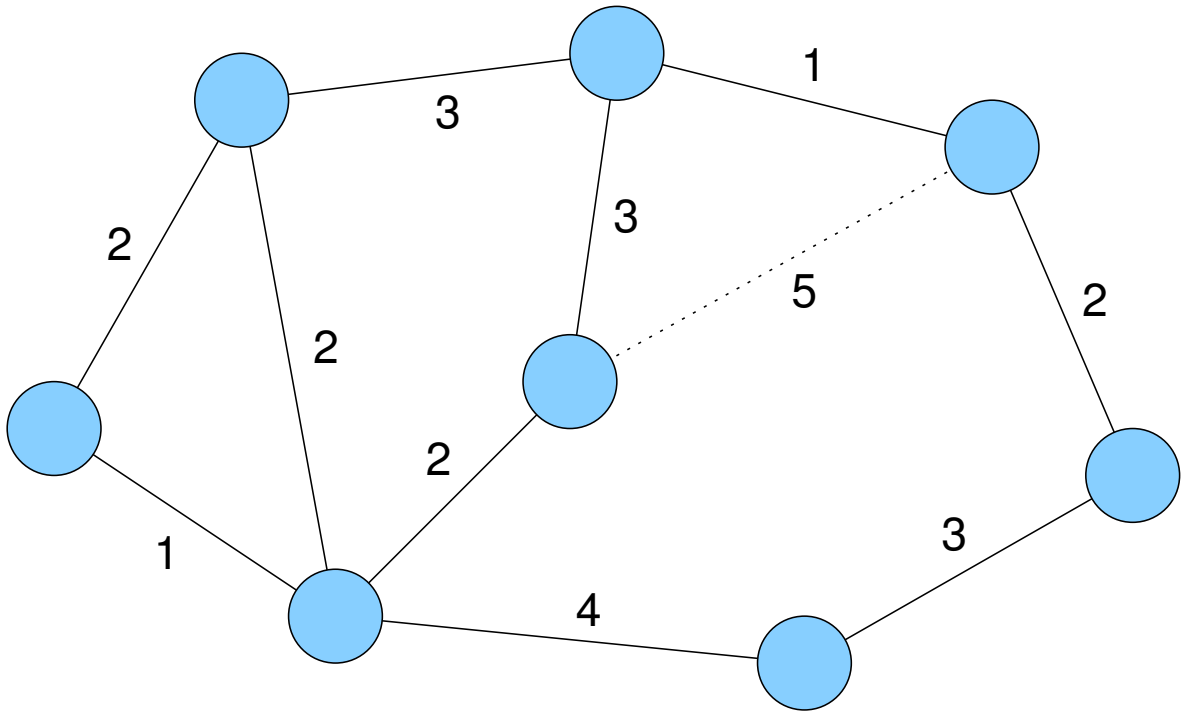
- wähle wiederholt Kante mit minimalen Kosten, die zwei Zusammenhangskomponenten verbindet
- bis nur noch eine Zusammenhangskomponente übrig ist



Minimaler Spannbaum

Regel:

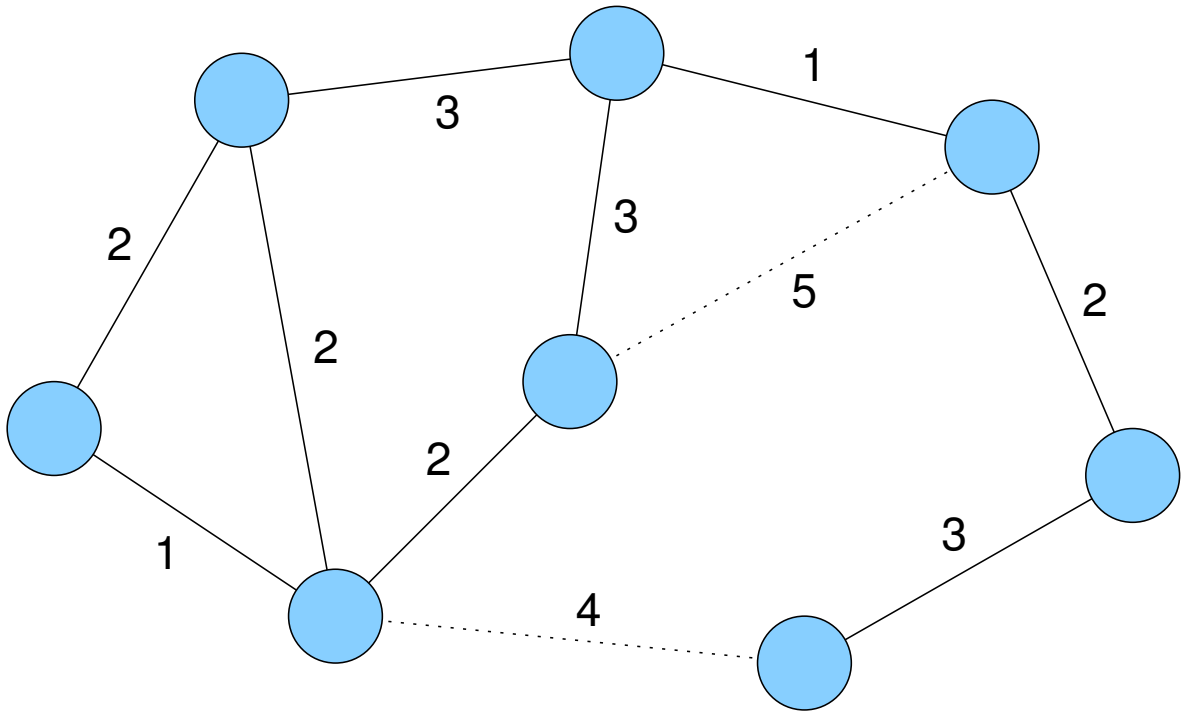
- lösche wiederholt Kante mit maximalen Kosten, so dass Zusammenhang nicht zerstört
- bis ein Baum übrig ist



Minimaler Spannbaum

Regel:

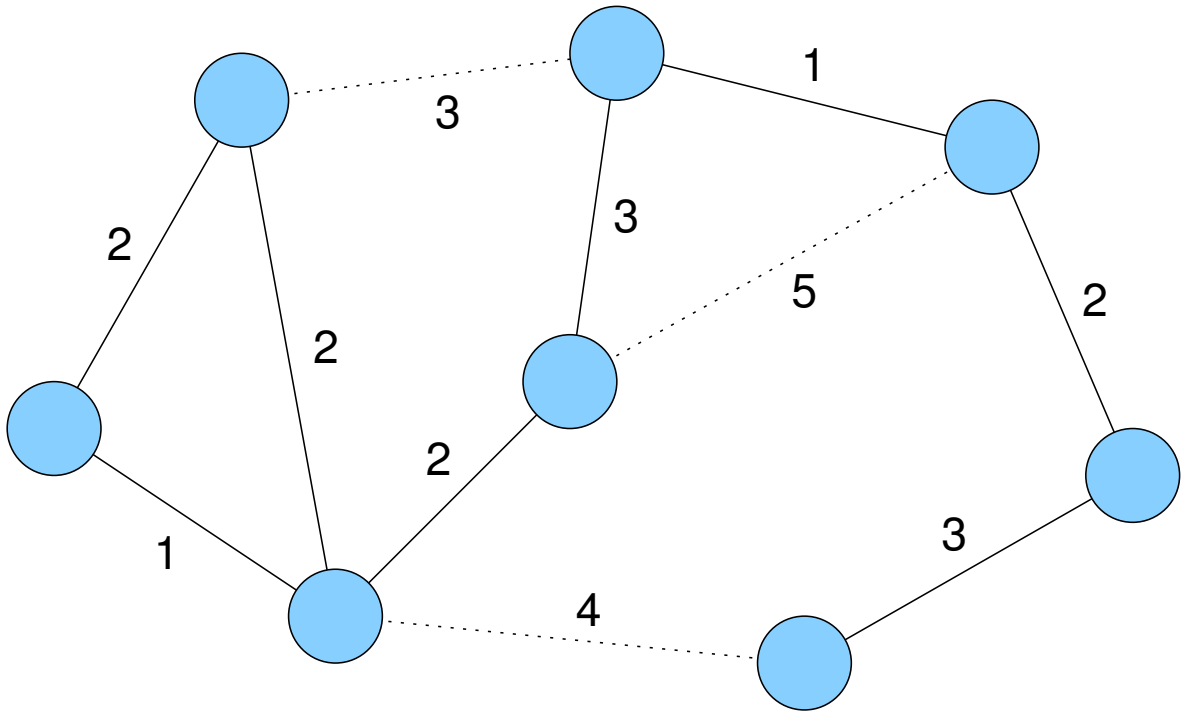
- lösche wiederholt Kante mit maximalen Kosten, so dass Zusammenhang nicht zerstört
- bis ein Baum übrig ist



Minimaler Spannbaum

Regel:

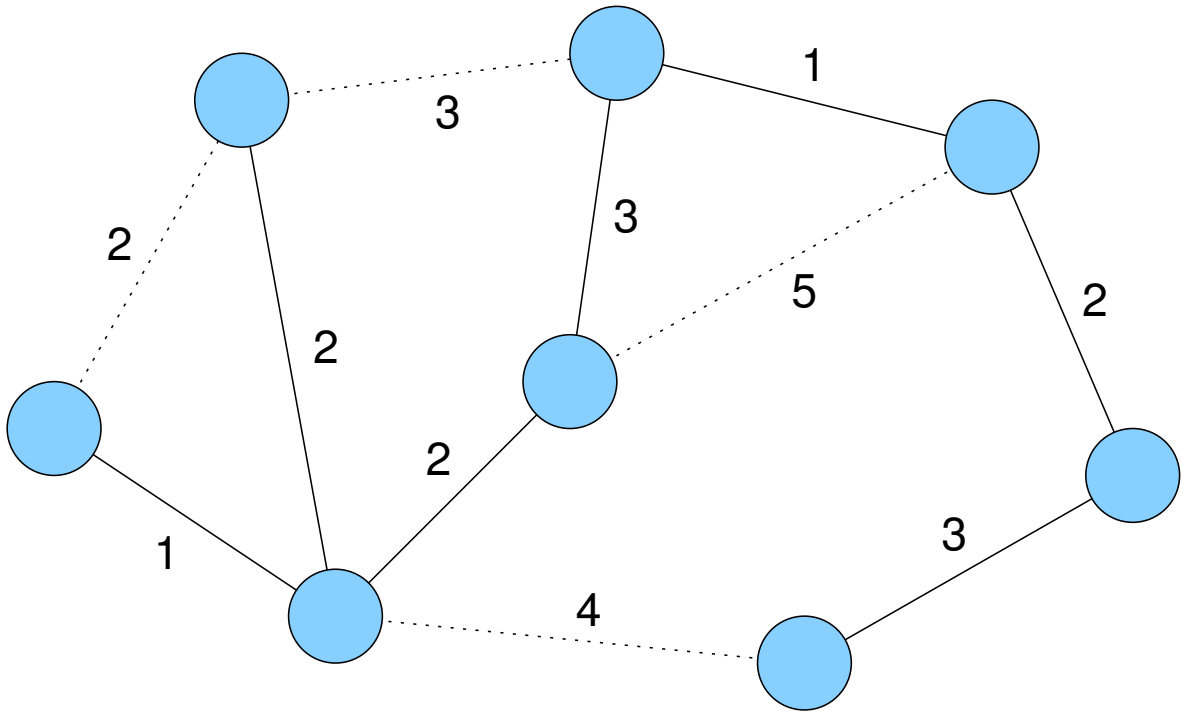
- lösche wiederholt Kante mit maximalen Kosten, so dass Zusammenhang nicht zerstört
- bis ein Baum übrig ist



Minimaler Spannbaum

Regel:

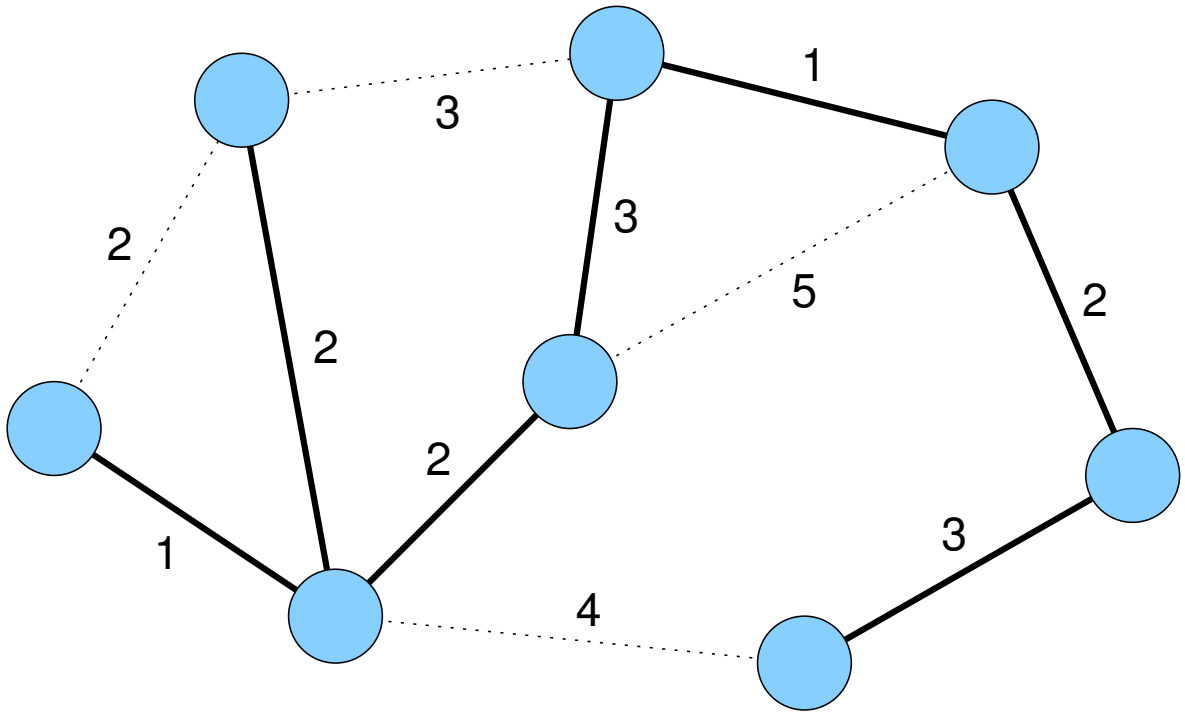
- lösche wiederholt Kante mit maximalen Kosten, so dass Zusammenhang nicht zerstört
- bis ein Baum übrig ist



Minimaler Spannbaum

Regel:

- lösche wiederholt Kante mit maximalen Kosten, so dass Zusammenhang nicht zerstört
- bis ein Baum übrig ist



Minimaler Spannbaum

Problem: Wie implementiert man die Regeln effizient?

Strategie aus dem ersten Lemma:

- sortiere Kanten aufsteigend nach ihren Kosten
- setze $T = \emptyset$ (leerer Baum)
- teste für jede Kante $\{u, v\}$ (in aufsteigender Reihenfolge), ob u und v schon in einer Zusammenhangskomponente (also im gleichen Baum) sind
- falls nicht, füge $\{u, v\}$ zu T hinzu (nun sind u und v im gleichen Baum)

Algorithmus von Kruskal

```
Set<Edge> MST_Kruskal (V, E, c) {
    T = ∅;
    S = sort(E);   // aufsteigend sortieren
    foreach (e = {u, v} ∈ S)
        if (u und v in verschiedenen Bäumen in T)
            T = T ∪ e;
    return T;
}
```

Problem:

- Umsetzung des Tests auf gleiche / unterschiedliche Zusammenhangskomponente

Union-Find-Datenstruktur

Union-Find-Problem:

- gegeben sind (disjunkte) Mengen von Elementen
- jede Menge hat genau einen Repräsentanten
- union soll zwei Mengen vereinigen, die durch ihren jeweiligen Repräsentanten gegeben sind
- find soll zu einem gegebenen Element die zugehörige Menge in Form des Repräsentanten finden

Anwendung:

- Knoten seien nummeriert von 0 bis $n - 1$
- Array int parent[n], Einträge verweisen Richtung Repräsentant
- anfangs parent[i]=i für alle i

Union-Find-Datenstruktur

```
int find(int i) {
    if (parent[i] == i)    return i;    // ist i Wurzel des Baums?
    else {    // nein
        k = find( parent[i] );    // suche Wurzel
        parent[i] = k;    // zeige direkt auf Wurzel
        return k;    // gibt Wurzel zurück
    }
}

union(int i, int j) {
    int ri = find(i);
    int rj = find(j);    // suche Wurzeln
    if (ri ≠ rj)
        parent[ri] = rj;    // vereinigen
}
```

Algorithmus von Kruskal

```
Set<Edge> MST_Kruskal (V, E, c) {
    T = ∅;
    S = sort(E);    // aufsteigend sortieren
    for (int i = 0; i < |V|; i++)
        parent[i] = i;
    foreach (e = {u, v} ∈ S)
        if (find(u) ≠ find(v)) {
            T = T ∪ e;
            union(u, v);    // Bäume von u und v vereinigen
        }
    return T;
}
```

Gewichtete union-Operation mit Pfadkompression

- Laufzeit von find hängen von der Höhe des Baums ab
- deshalb wird am Ende von find jeder Knoten auf dem Suchpfad direkt unter die Wurzel gehängt, damit die Suche beim nächsten Mal direkt zu diesem Knoten kommt (Pfadkompression)
- weiterhin sollte bei union der niedrigere Baum unter die Wurzel des höheren gehängt werden (gewichtete Vereinigung)

$\Rightarrow$ Höhe des Baums ist dann $O(\log n)$

Gewichtete union-Operation

```
union(int i, int j) {
    int ri = find(i);
    int rj = find(j);    // suche Wurzeln
    if (ri ≠ rj)
        if (height[ri] < height[rk])
            parent[ri] = rj;
        else {
            parent[rj] = ri;
            if (height[ri] == height[rk])
                height[ri]++;
        }
}
```

union / find - Kosten

Situation:

- Folge von union / find -Operationen auf einer Partition von n Elementen, darunter $n-1$ union-Operationen

Komplexität:

- amortisiert $\log^* n$ pro Operation, wobei

$$\log^* n = \min\{i \geq 1 : \underbrace{\log \log \ldots \log}_{i\text{–mal}} n \leq 1\}$$

- bessere obere Schranke: mit inverser Ackermannfunktion (Vorlesung Effiziente Algorithmen und Datenstrukturen I)
- Gesamtkosten für Kruskal-Algorithmus: $\mathcal{O}(m \log n)$ (Sortieren)

Algorithmus von Prim

Problem: Wie implementiert man die Regeln effizient?

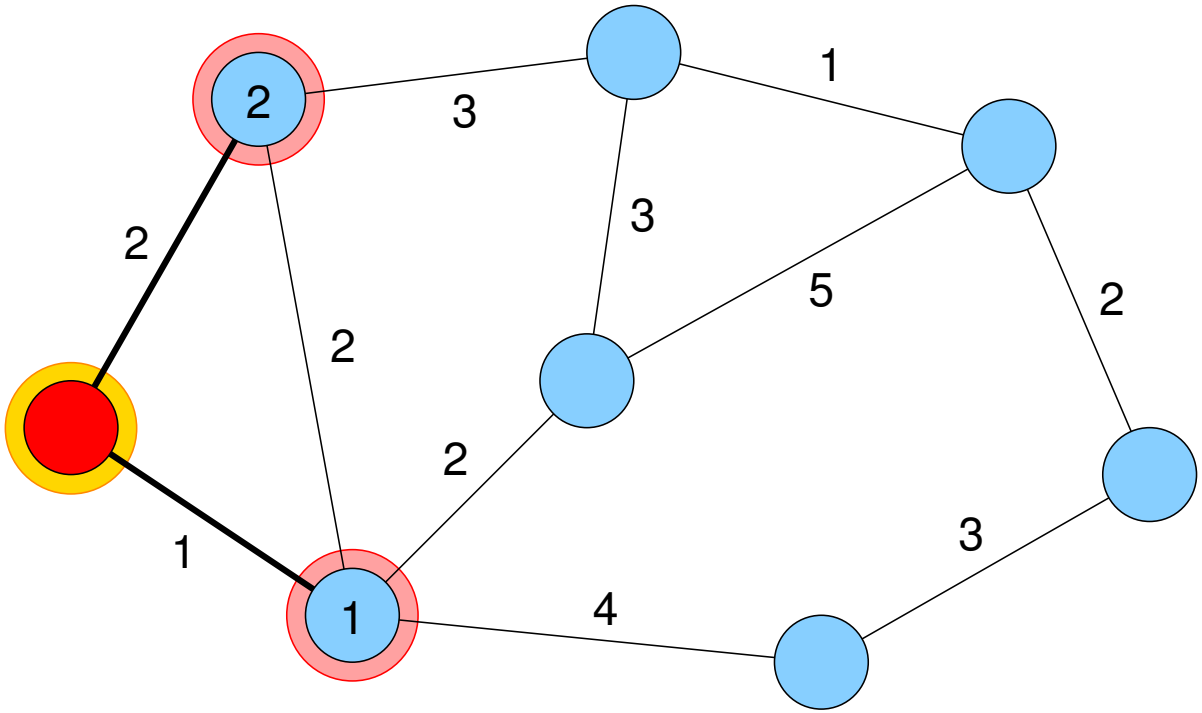
Alternative Strategie aus dem ersten Lemma:

- betrachte wachsenden Baum T, anfangs bestehend aus beliebigem einzelnen Knoten s
- füge zu T eine Kante mit minimalem Gewicht von einem Baumknoten zu einem Knoten außerhalb des Baums ein (bei mehreren Möglichkeiten egal welche)

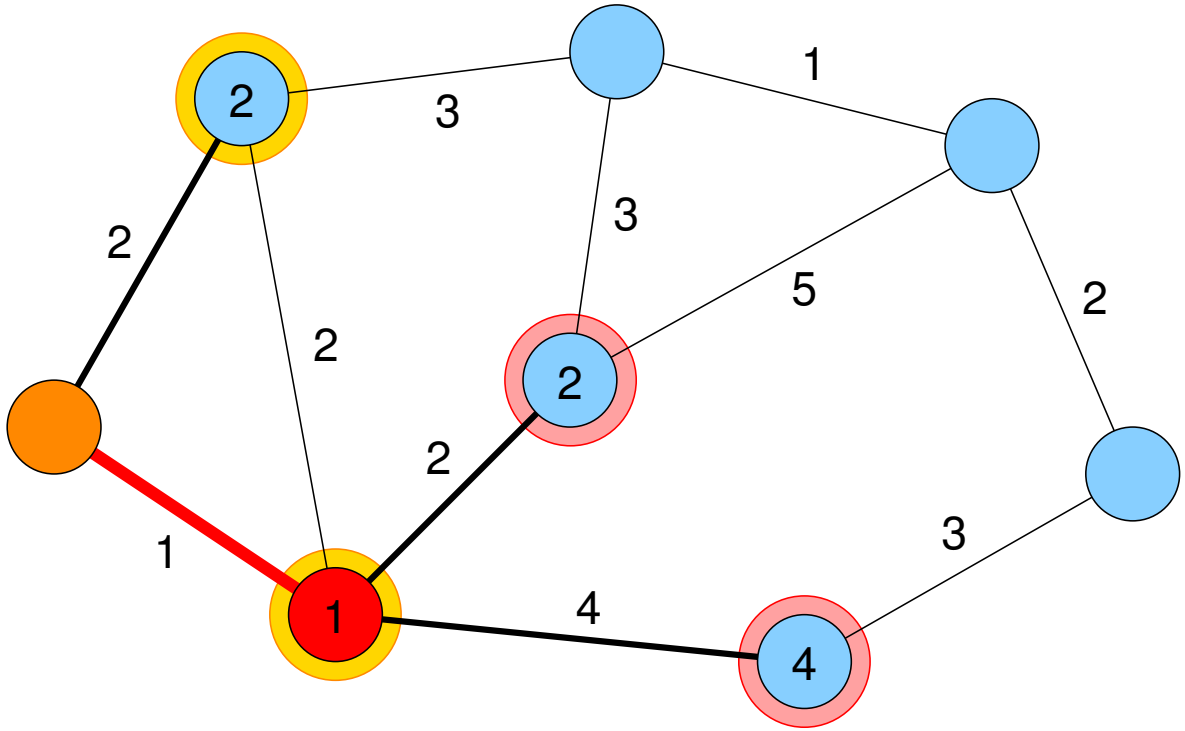
$\Rightarrow$ Baum umfasst jetzt 1 Knoten / Kante mehr

- wiederhole Auswahl bis alle n Knoten im Baum

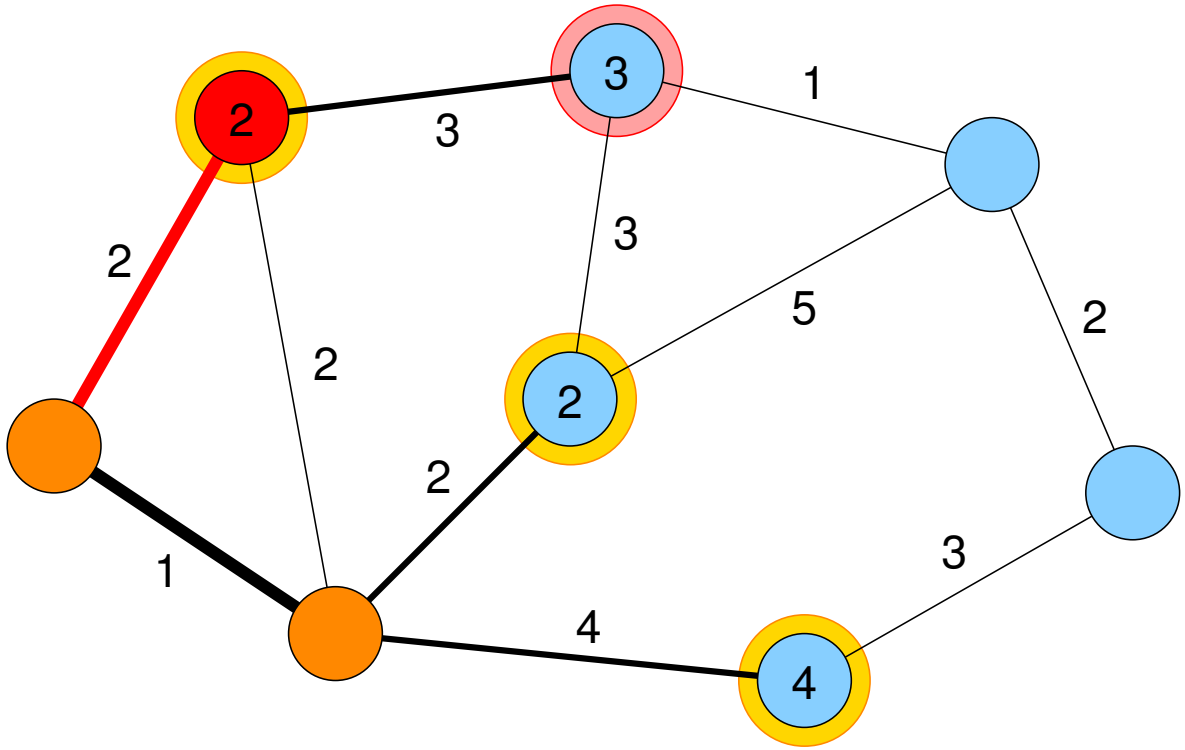
Algorithmus von Prim



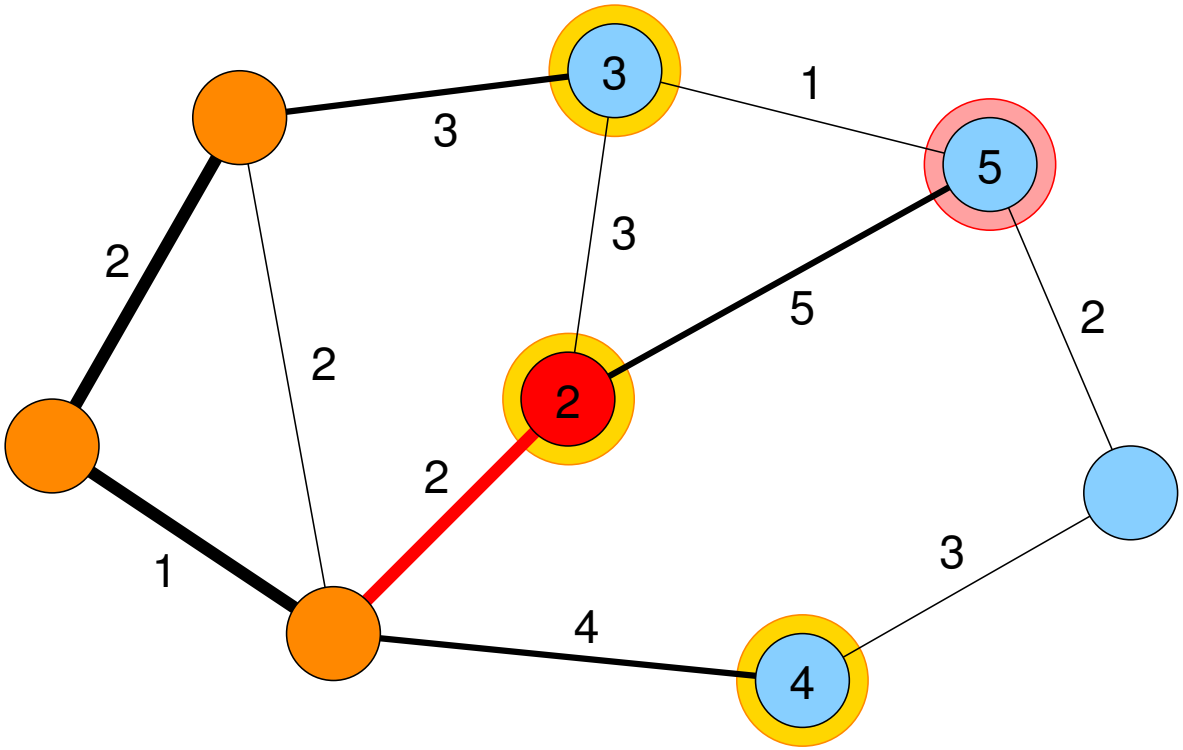
Algorithmus von Prim



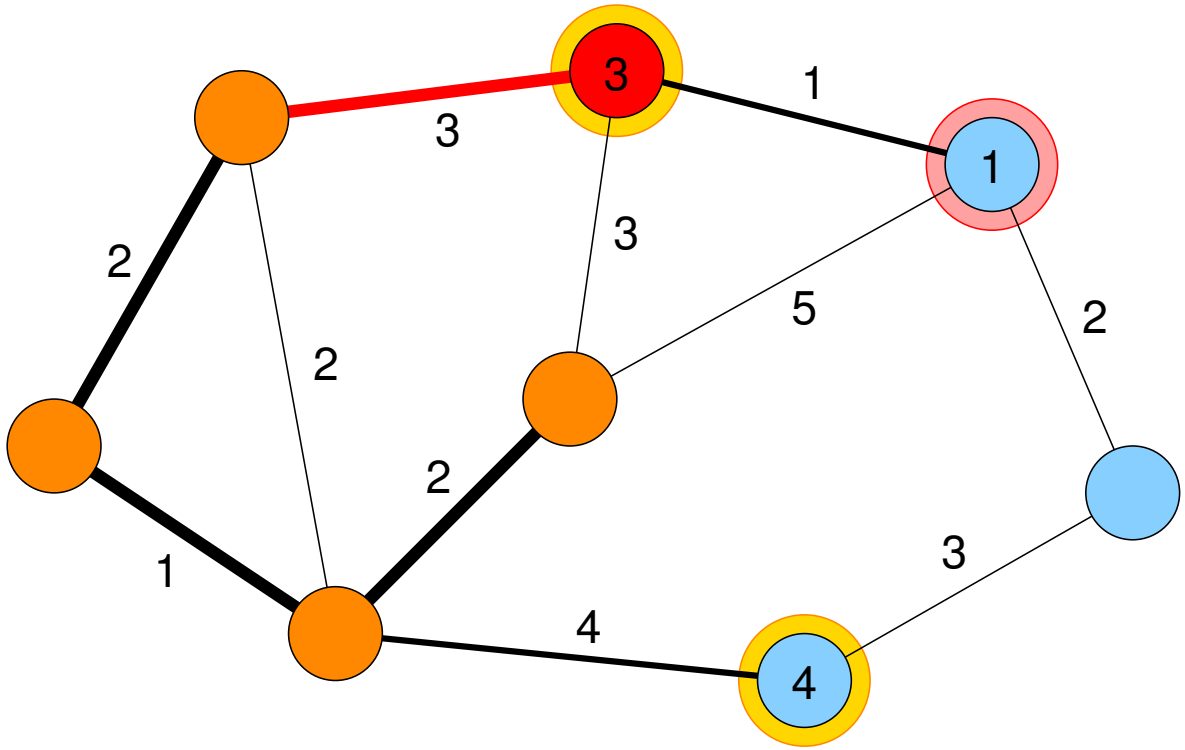
Algorithmus von Prim



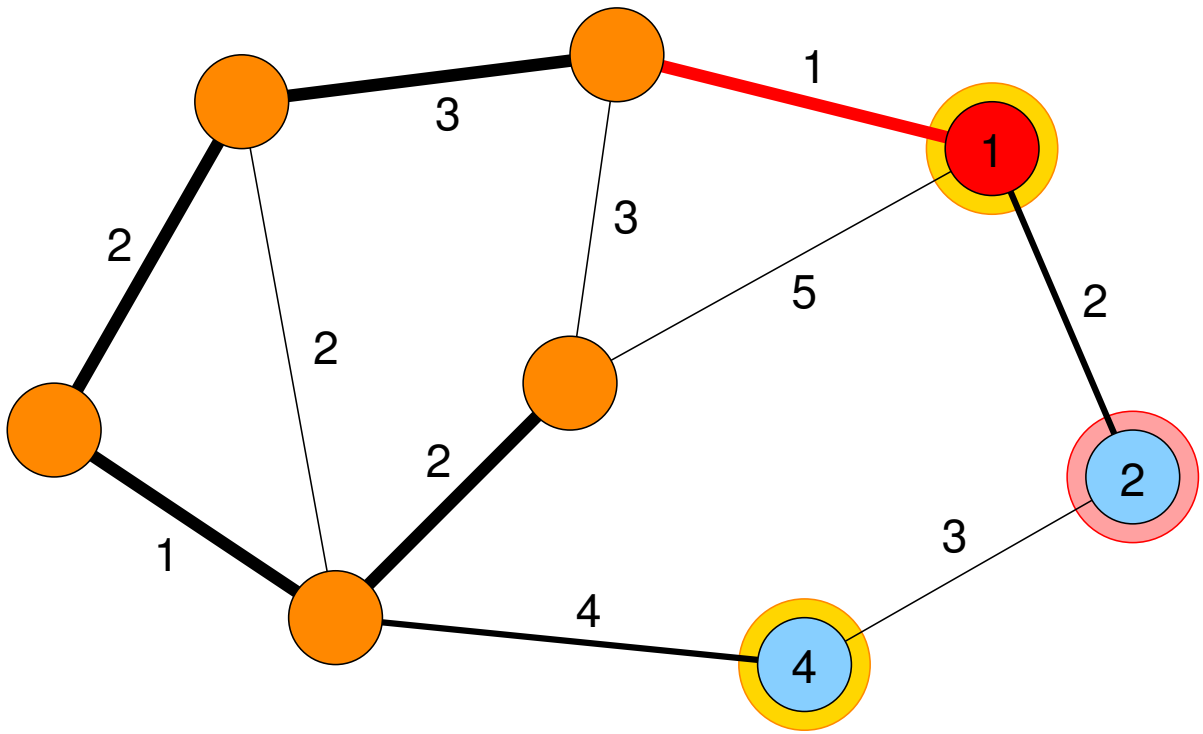
Algorithmus von Prim



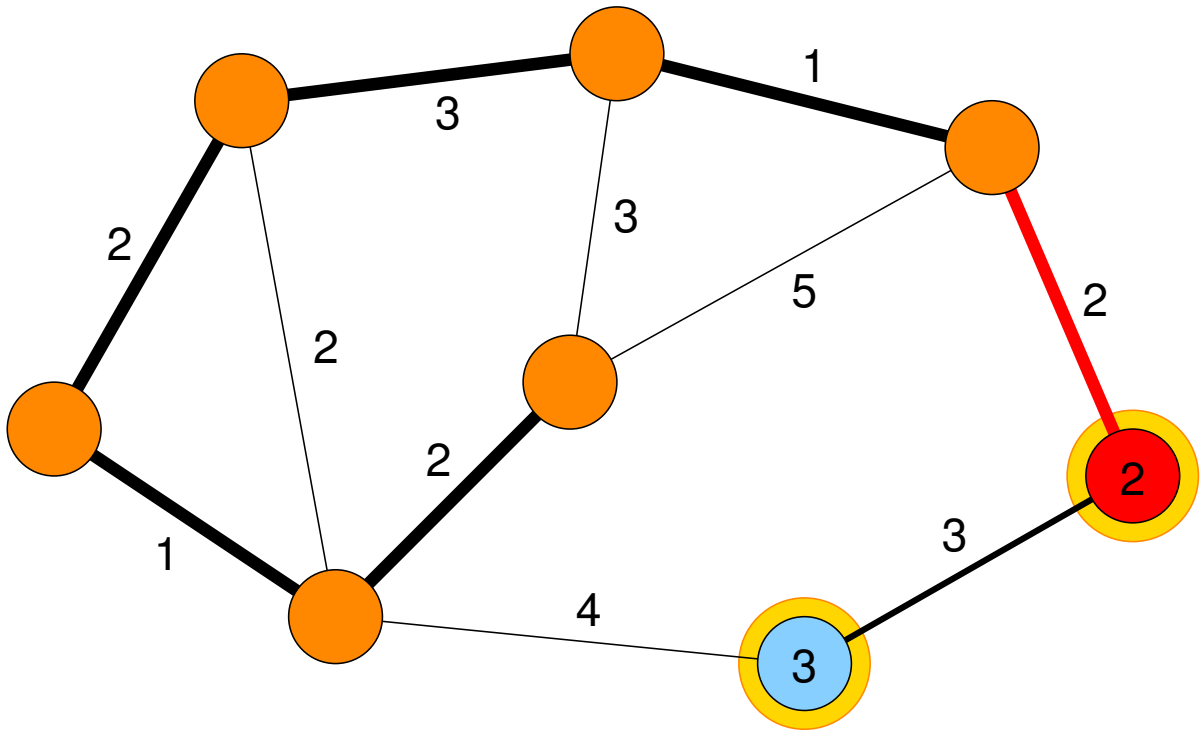
Algorithmus von Prim



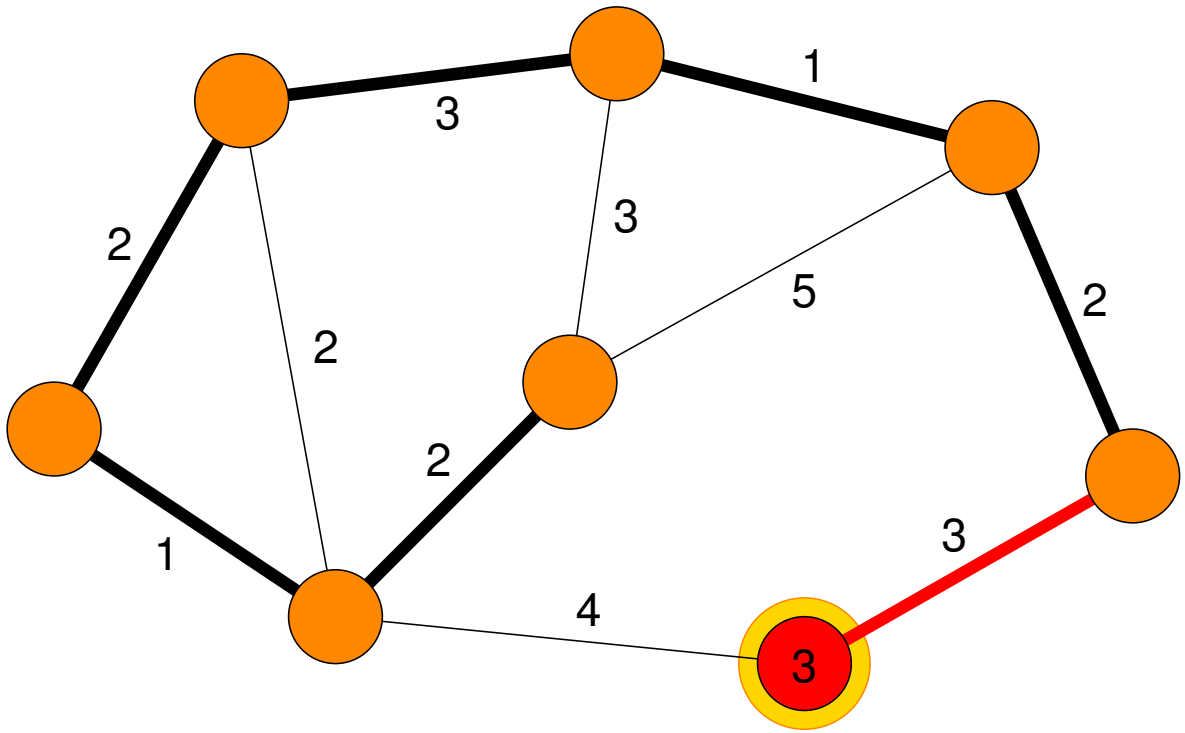
Algorithmus von Prim



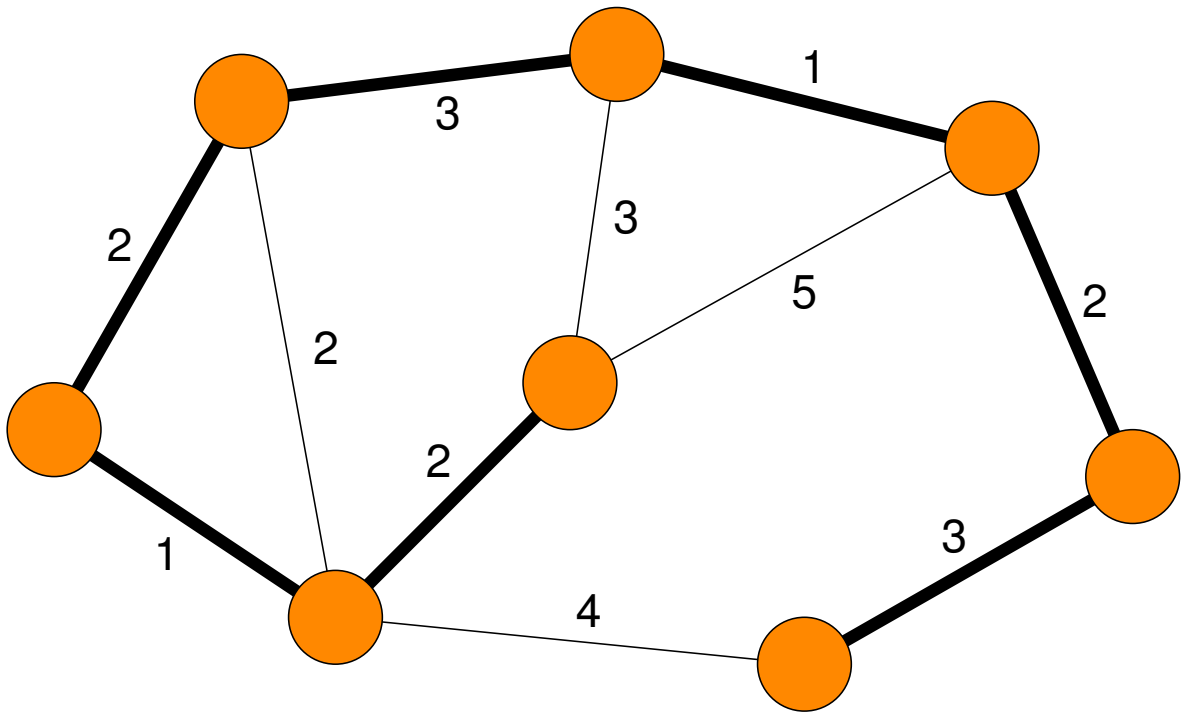
Algorithmus von Prim



Algorithmus von Prim



Algorithmus von Prim



Algorithmus Jarník-Prim: findet minimalen Spannbaum

Eingabe : $G = (V, E)$, $c : E \mapsto \mathbb{R}_+$, $s \in V$

Ausgabe : Minimaler Spannbaum in Array *pred*

$d[v] = \infty$ for all $v \in V \setminus s$;
$d[s] = 0$; $pred[s] = \perp$;
$pq = \langle\rangle$; pq.insert$(s, 0)$;
while $\neg pq.empty()$ **do**
  $v = pq$.deleteMin();
  forall the $\{v, w\} \in E$ **do**
    newWeight $= c(v, w)$;
    if $newWeight < d[w]$ **then**
      $pred[w] = v$;
      if $d[w] == \infty$ **then** pq.insert$(w,$ newWeight$)$;
      else
        if $w \in pq$ **then** pq.decreaseKey$(w,$ newWeight$)$;
      $d[w] =$ newWeight;

Jarník-Prim-Algorithmus

Laufzeit:

$$\mathcal{O}\Big(n \cdot (T_{\text{insert}}(n) + T_{\text{deletMin}}(n)) + m \cdot T_{\text{decreaseKey}}(n)\Big)$$

Binärer Heap:

- alle Operationen $\mathcal{O}(\log n)$, also
- gesamt: $\mathcal{O}((m+n)\log n)$

Fibonacci-Heap: amortisierte Kosten

- $\mathcal{O}(1)$ für insert und decreaseKey,
- $\mathcal{O}(\log n)$ deleteMin
- gesamt: $\mathcal{O}(m + n\log n)$