Algorithmen und Datenstrukturen

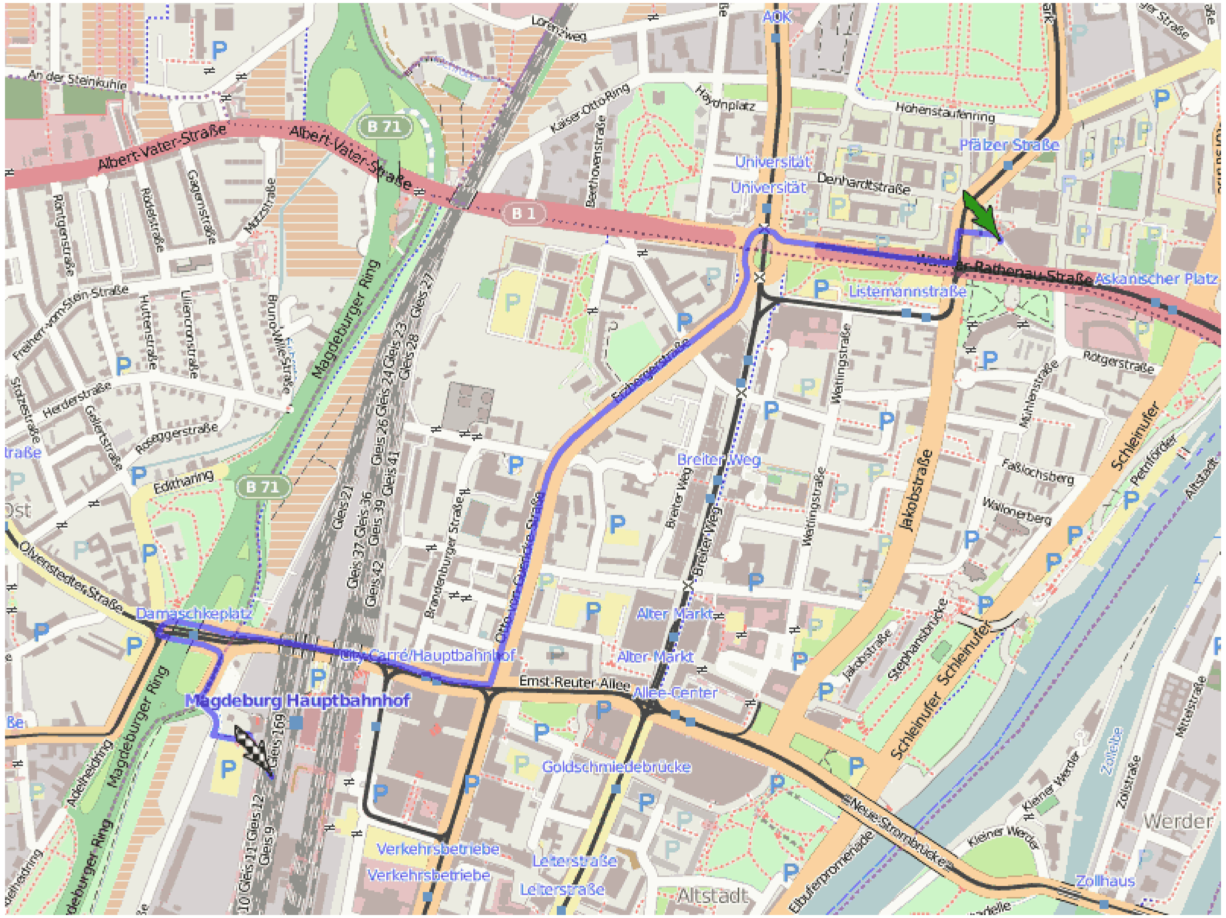
Kürzeste Wege und Minimaler Spannbaum

Dr. Christian Rössl

Überblick

- Grundlegende Algorithmen: Tiefen- & Breitensuche
- Topologisches Sortieren
- Kürzeste Wege
- Fluss in Netzwerken

Motivation



AOK
Lorenzweg
An der Steinkuhle
Haydnplatz
Kaiser-Otto-Ring
Beethovenstraße
Hohenstaufenring
Pfälzer Straße
Universität
Universität
Denhardtstraße
Albert-Vater-Straße
Albert-Vater-Straße
B 71
B 1
Askanischer Platz
Listemannstraße
Rötgerstraße
Magdeburger Ring
Faßlochsberg
Wallonerberg
Schleinufer
Petriförder
Breiter Weg
Editharing
B 71
Olvenstedter Straße
Damaschkeplatz
Jakobstraße
Alter Markt
Alter Markt
City Carré/Hauptbahnhof
Ernst-Reuter-Allee
Allee-Center
Magdeburg Hauptbahnhof
Goldschmiedebrücke
Neue Strombrücke
Verkehrsbetriebe
Verkehrsbetriebe
Leiterstraße
Leiterstraße
Altstadt
Elbuferpromenade
Werder
Zollhaus

Breitensuche

$i \xrightarrow{\gamma} j$

- Finde kürzesten Wege in **gewichteten** Graphen
- Länge von Wegen = Summe der Kantengewichte
- *Wir kennen bereits* die Breitensuche in *ungewichteten* Graphen
  - Weglänge gemessen in Anzahl Kanten bzw. $\forall e \in E : \gamma(e) = 1$
  - Berechnet kürzeste Wege vom Startknoten zu *allen* erreichbaren Knoten
- Zur Erinnerung: DFS mit Stack $\rightarrow$ BFS mit Queue

Priority First Search (PFS)

- Idee: Modifiziere die Breitensuche zu einer *Bestensuche*
- Besuche als nächsten immer den *nächstgelegenen* Knoten,
  d.h. wähle immer den Knoten mit dem *geringsten* Abstand zum Startknoten
- Wir können diese Wahl mit Hilfe einer **Priority Queue** effizient gestalten:
  Priorität eines erreichten Knotens = bereits zurückgelegter Weg dorthin
- D.h. aus BFS mit Queue wird *priority first search* PFS mit Priority Queue

Anwendung der PFS

- Algorithmus von **Prim** (1957) zur Berechnung *minimaler Spannbäume*
- Algorithmus von **Dijkstra** (1959) zur Berechnung *kürzester Wege*

- Beide PFS: Ersetze Queue durch Priority Queue
  - Wähle jeweils Knoten mit kleinster Priorität aus
  - D.h. Knoten mit der geringsten Distanz zum Startknoten
- Wichtiger Unterschied zur BFS:

  Ein schon erreichter aber noch nicht abgearbeiteter Knoten (in Front/PQ) könnte auf einem **kürzeren** Weg erreicht werden!
  - D.h. es muss möglich sein, Prioritäten von **beliebigen** Knoten in PQ zu aktualisieren
  - *“Heap alleine reicht nicht!”*

Algorithmus von Dijkstra

- Feld `d[]` misst Distanzen, die PQ als Prioritäten verwendet

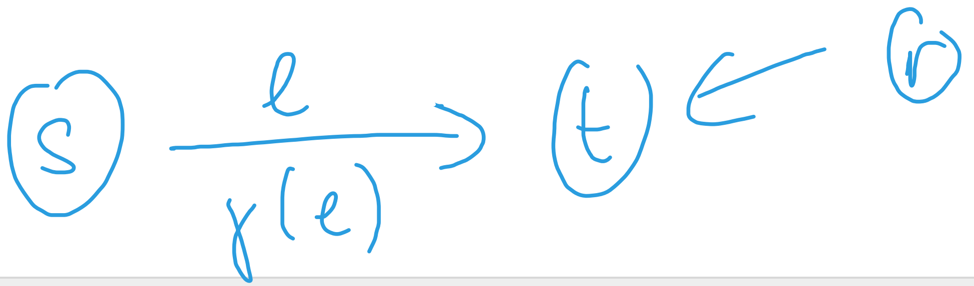
  Initialisierung mit ∞ ⇔ Knoten nicht markiert

- Feld `p[]` speichert *shortest path tree* (Elternknoten)

```
function dijkstra(s₀)
  for each node n do
      d[n] = ∞       # distance to start (priority)
      p[n] = -1      # parent node
  end

  open = empty_priorityqueue[Node,d]
  d[s₀] = 0
  open.push(s₀)
  # p[s₀] = -1   => root

 ...
```




```
  ...

  while not open.is_empty() do
        s = open.pop()
        for each edge e in outgoing(s) do
             t = destination(e)
             pr = d[s] + γ(e)              # t's "priority"

             if not (d[t] < ∞)
                  d[t] = pr                  # "tentative distance"
                  p[t] = s
                  open.push(t)               # not seen yet
             elseif open.contains(t) and pr < d[t]
                  d[t] = pr                  # "tentative distance"
                  p[t] = s
                  open.lower(t)              # lowered priority: update PQ
            end
  end end
end
```

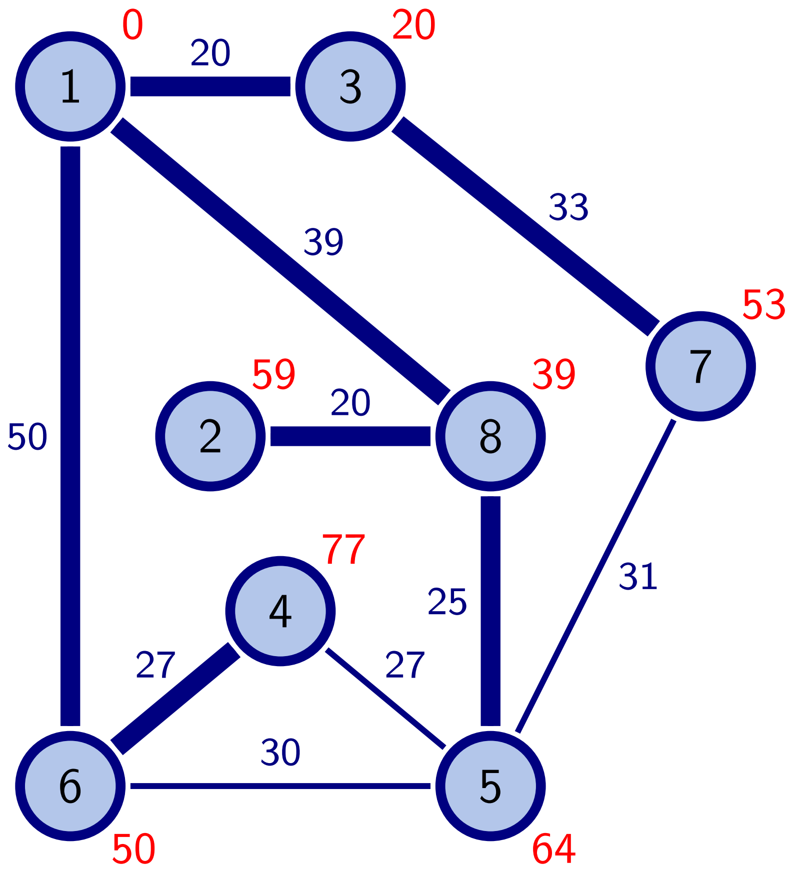
Algorithmus von Dijkstra kürzer

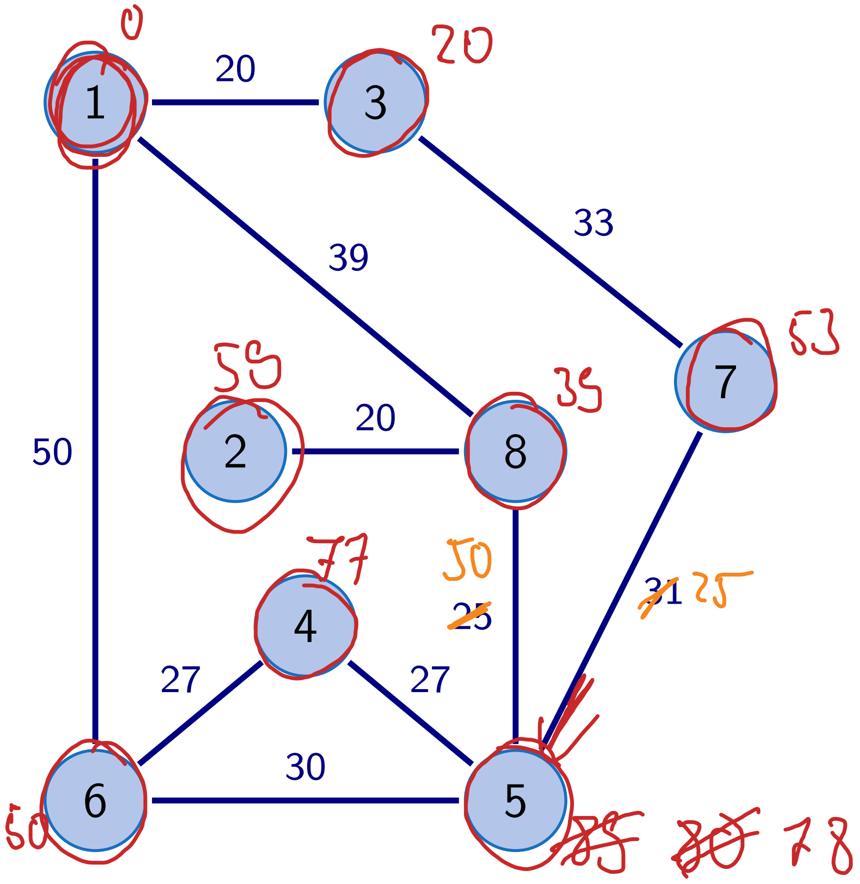
Initialisierung mit *allen* Knoten in PQ mit Prioritäten 0 an s_0 und ∞ sonst

```
function dijkstra(s₀)
  for each node n do
      d[n] = ∞
      p[n] = -1
  end
  d[s₀] = 0
  open = create_queue(nodes,d)        # all nodes (use bottom-up heap construction)

  while not open.is_empty() do
     s = open.pop()
     for each edge e in outgoing(s) do
         t = destination(e)
         pr = d[s] + γ(e)             # assume ∞ + γ = ∞
         if open.contains(t) and pr < d[t] then
             d[t] = pr                # "tentative distance"
             p[t] = s
             open.lower(t)            # lowered priority: update PQ
         end
  end end
end
```

Beispiel: PFS





Nochmal mit variierten Gewichten z.B. 50 statt 25 $(8 \to 5)$, dazu: 25 statt 31 $(7 \to 5)$

Demo

- `aud.example.graph.Traversal`
- Abstrakte Basisklasse `PriorityFirstSearch` implementiert Grundalgorithmus PFS
- Varianten implementieren Methode `priority()`
    - `DijkstraShortestPaths`
    - `PrimMinimumSpanningTree`

Bemerkungen zum Algorithmus

- PFS in dieser Form = **Algorithmus von Dijkstra**
- Beachte mögliche Aktualisierung von Distanzen
    - `d[t]` wird erniedrigt, Elternknoten `p[t]` wird angepasst
    - Aktualisierung der PQ durch durch `open.lower(t)`

erreichbar

- Berechnen der **kürzesten** Distanzen `d[]` zu *allen* Knoten **und** des ***shortest path tree*** (**SPT**) `p[]`
- Pfade in SPT zur Wurzel = kürzeste Wege zum Start s_0
- Alternativ: kürzester Weg *zwischen zwei* Knoten, d.h.
    - Erster Knoten = Startknoten
    - Abbruch, sobald der zweite Knoten *abgearbeitet* wurde

Backtracking

- PFS berechnet den SPT `p[]` simultan mit Distanz `d[]`
- Mit Hilfe von `p[]` können wir den kürzesten Wege zum Start *zurückverfolgen*
- Allgemeines Prinzip: **backtracking** (“Rückverfolgung”)
  1. Suche z.B. nach einer optimalen Konfiguration

     z.B. Tiefensuche, Breitensuche, Heuristik (oft *trial-and-error*)
  2. Danach Zurückverfolgen des Weges (Entscheidungen, algorithmische Schritte) bis zum Startpunkt

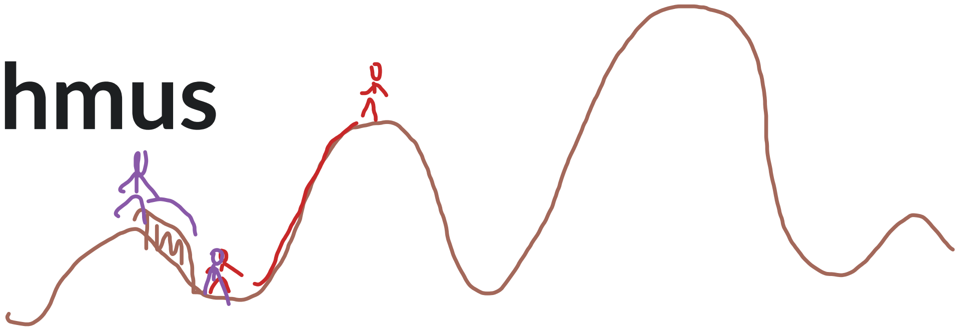
     D.h. “Abspielen in Umgekehrter Reihenfolge” = Weg zur Lösung

Backtracking nach PFS

- Zurückverfolgen des Wegs von Knoten n zum Startknoten s_0

```
function backtrack(n)
    while n ≠ s₀ do
        output(n)
        n = p[n]
    end
    output(s₀)
end
```

Greedy Algorithmus



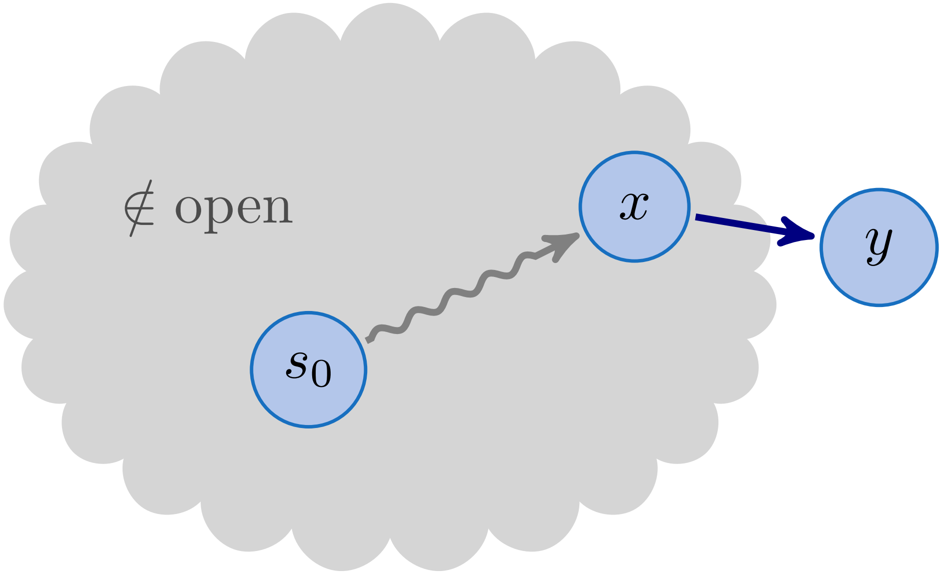
- PFS ist ein typischer **Greedy**-Algorithmus
  - von engl. *greedy* = gierig
  - häufiger Ansatz für Optimierungsalgorithmen – hier: *kürzeste* Wege
- **greedy** = Wähle in jedem Schritt die *aktuell* günstigste Möglichkeit
- D.h. Entscheidung anhand *lokaler* Kriterien
- **Beachte:** Mit diesem Prinzip erreichen wir i.a. nur *lokale Minama* und verharren dort
- Für *Priority First Search*
  - Nimm Knoten mit kürzester Entfernung aus PQ
  - Revidiere ggf. vorhergehende Berechnung
  - PFS findet *global* kürzeste Wege

Dijkstra-Algorithmus

- Der *Algorithmus von Dijkstra* berechnet **kürzeste Wege**,
  - d.h. er erreicht tatsächlich ein **globales Optimum**
  - … obwohl er ein Greedy-Algorithmus ist

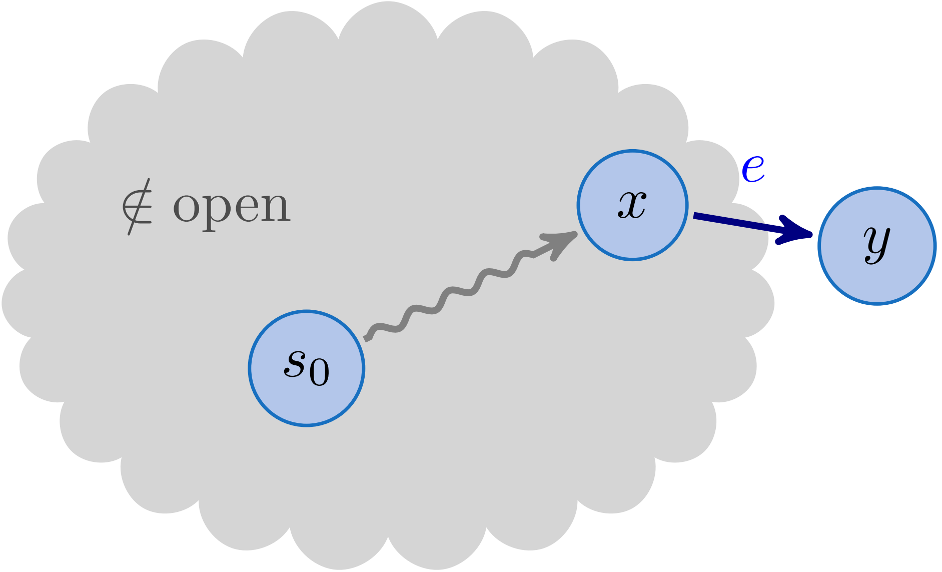
  Dabei gibt es eine wichtige Einschränkung – wir werden gleich sehen, welche & warum
- Wir wollen diese Aussage beweisen
- Dabei bezeichnen $d(s_0, v)$ und $d[v]$ die *tatsächliche* und die berechnete Distanz
- Zu Beginn sind alle Knoten in der PQ open mit $d[s_0] = 0$ und $d[v] = \infty$ für $v \neq s_0$

Beweis zum Dijkstra-Algorithmus



- Knoten im grauen Bereich sind abgearbeitet
- **z.z.**: Sobald ein Knoten v aus der PQ open entnommen wird, gilt $d[v] = d(s_0, v)$

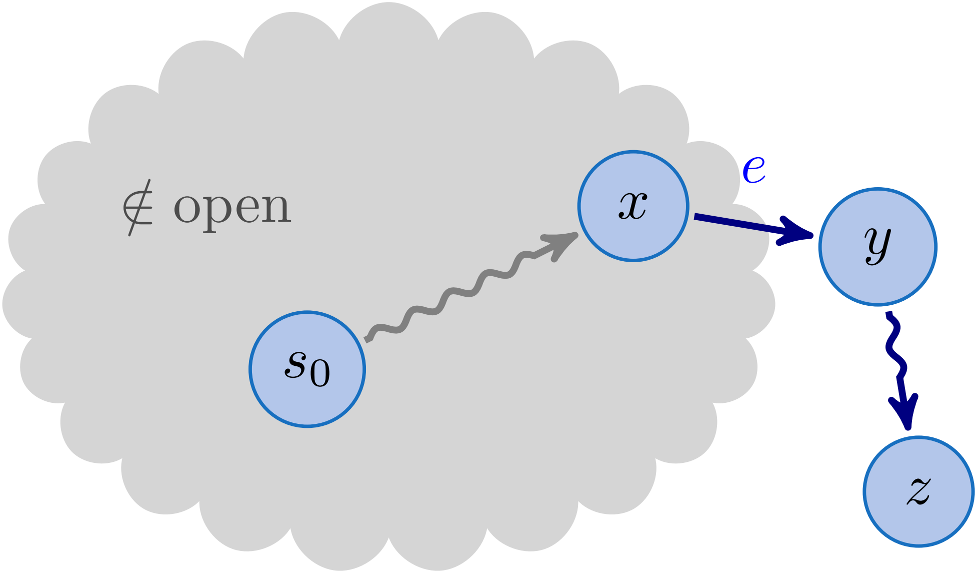
Betrachte *Relaxation*



Relaxation = Entfernen einer Kante e – und damit einer “Bedingung” aus dem System

```
if d[x] + γ(e) < d[y]
    d[y] = d[x] + γ(e)
end
```

Ansatz für *Widerspruchsbeweis*

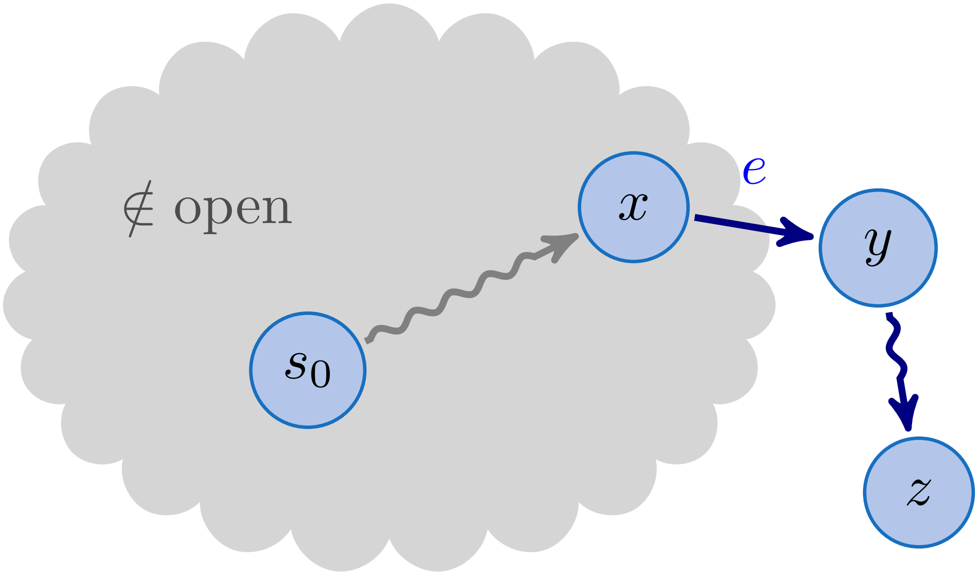


- **Annahme:** z ist der erste Knoten, der aus open entnommen wird mit $d[z] > d(s_0, z)$
- Damit gibt es einen kürzesten Pfad von s_0 nach z – sonst wäre $d(s_0, z) = \infty = d[z]$
- Sei y der erste Knoten auf diesem Pfad, der nicht in open ist,

  sei x sein Vorgänger im Pfad, so dass $x \xrightarrow{e} y$

Konstruiere Widerspruch

Annahme: $d[z] > d(s_0, z)$ “als *erster* abgearbeiteter Knoten”

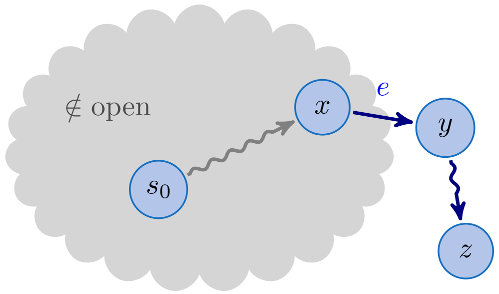


Es gilt

$$d[x] = d(s_0, x)$$

$$\begin{aligned} d[y] &= d[x] + \gamma(e) \\ &= d(s_0, x) + \gamma(e) \\ &= d(s_0, y) \end{aligned}$$

Annahme: $d[z] > d(s_0, z)$ "als erster abgearbeiteter Knoten"



Es gilt

$$d[x] = d(s_0, x)$$

$$\begin{aligned} d[y] &= d[x] + \gamma(e) \\ &= d(s_0, x) + \gamma(e) \\ &= d(s_0, y) \end{aligned}$$

- z kommt aus PQ open

  $\Rightarrow \; d[z] \leq d[y]$

- $d(s_0, y) + d(y, z) = d(s_0, z)$

  Denn Teile von kürzesten Wege sind selbst kürzeste Wege.

Wir fassen zusammen:

- Es gilt

$$\begin{aligned} d[z] \overset{PQ}{\leq} d[y] &= d(s_0, y) \\ &\leq \underbrace{d(s_0, y) + d(y, z)} \\ &= \qquad d(s_0, z) \end{aligned}$$

- Das ist ein **Widerspruch** zur Annahme $d[z] > d(s_0, z)$
- Damit kann es keinen solchen Knoten z geben, für den die Annahme gilt
- Daraus folgt die Behauptung:

  Sobald ein Knoten v aus der PQ open entnommen wird, gilt $d[v] = d(s_0, v)$

Stimmt das immer?

$$d[z] \leq d[y] = d(s_0, y) \ \leq \ d(s_0, y) \ + \ d(y, z)$$
$$= d(s_0, z)$$

Die Ungleichung

$$d(s_0, y) \leq d(s_0, y) \ + \ d(y, z)$$

gilt nur dann, wenn wir $d(y, z) \geq 0$ zusichern können

Der Dijkstra-Algorithmus berechnet kürzeste Pfade, wenn gilt

$$\forall e \in E \ : \ \gamma(e) \geq 0$$

D.h. alle **Kantengewichte sind nicht-negativ**.

Minimum Spanning Tree (MST)

> Als *Spannbaum* oder *spanning tree* eines ungerichteten (und zusammenhängenden) Graphen bezeichnet man einen Teilgraphen, der (1.) ein Baum ist und (2.) alle Knoten des Graphen enthält.

> Ein Spannbaum eines gewichteten Graphen heißt *minimaler Spannbaum* oder *minimum spanning tree* (MST), wenn die Summe der Kantengewichte minimal ist. D.h. es gibt *keinen* weiteren Spannbaum mit *geringerem* Gesamtgewicht.

- Der MST ist i.a. *nicht eindeutig*!

Kürzeste Wege und MST berechnen mit PFS

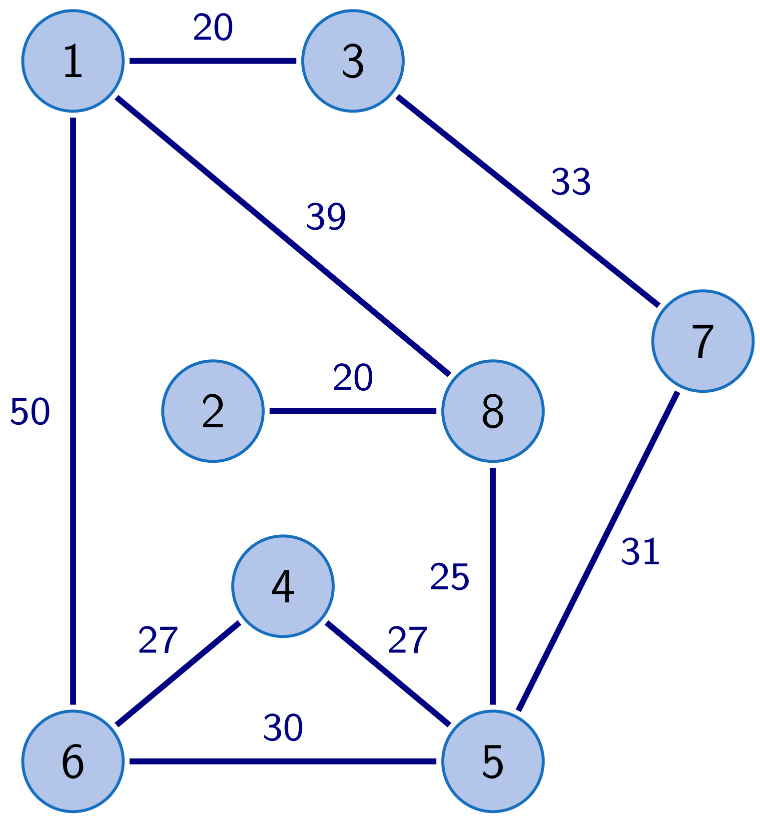
- **Algorithmus von Dijkstra** zur Berechnung kürzester Wege
  - Wähle Knoten mit kürzester Distanz
  - Priorität = aggregierte Wegstrecke `pr = d[s] + γ(e)`
  - Minimiere Länge von Pfaden im Spannbaum $\rightarrow$ SPT
- **Algorithmus von Prim** zur Berechnung des MST
  - Wähle Knoten, der über kürzeste Kante erreichbar ist
  - Priorität = Kantenlänge `pr = γ(e)`
  - Minimiere Summe der Kantengewichte im Spannbaum $\rightarrow$ MST
- **Aufwand** für PFS (mit binärem Heap) ist $O(\,|E|\,\log\,|V|)$
- *"Länge"* jeweils gemessen in *Kantengewichten* γ

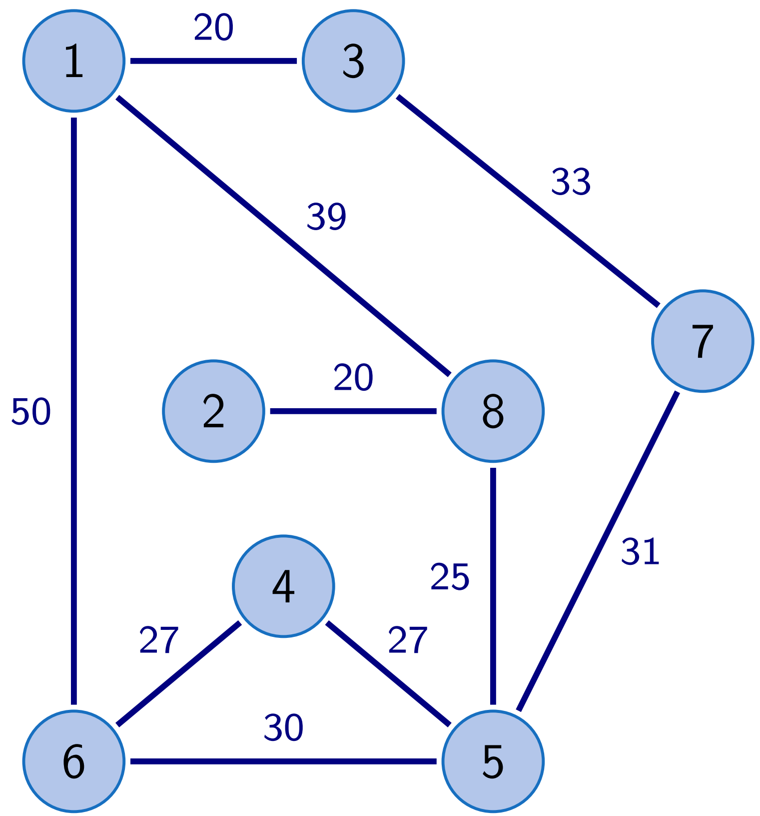
Algorithmus von Prim

```
function prim(s₀)
  for each node n do
      d[n] = ∞
      p[n] = -1
  end
  d[s₀] = 0
  open = create_queue(nodes,d)          # all nodes (use bottom-up heap construction)

  while not open.is_empty() do
     s = open.pop()
     for each edge e in outgoing(s) do
         t = destination(e)
         pr = γ(e)                      # assume ∞ + γ = ∞
         if open.contains(t) and pr < d[t] then
             d[t] = pr
             p[t] = s
             open.lower(t)              # lowered priority: update PQ
         end
  end end
end
```

Beispiel

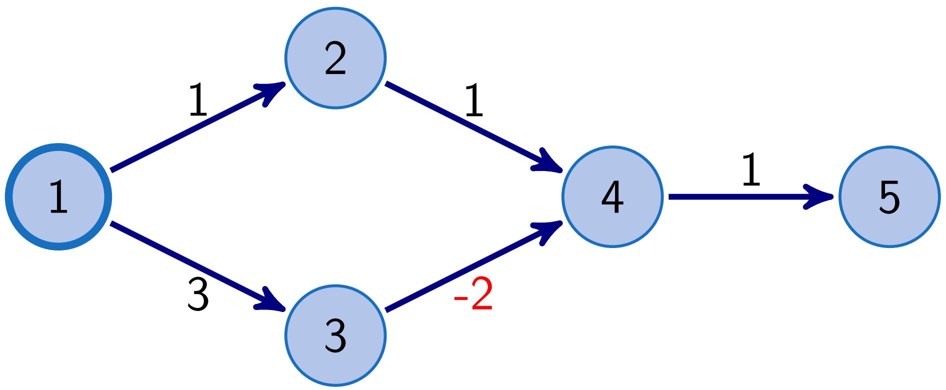




Verschiedene Startknoten: *Die Summe der Kantengewichte im MST ist jeweils gleich*

Negative Kantengewichte in gerichteten Graphen

- Dijkstras Algorithmus funktioniert nur für **nicht-negative** Kantengewichte!
- Beispiel: Knoten 4 wird zuerst über 2 erreicht



- Negative Gewichte können sinnvoll sein: z.B. stelle Gewinne & Kosten dar
- Durch “Umwege” können Kosten gesenkt werden
    - Voraussetzung: Entscheidung anhand globaler Betrachtung
    - Grundsätzlich nicht möglich mit Greedy-Algorithmus

Bellman-Ford Algorithmus

- Berechnet kürzeste Wege
  - Negative Kantengewichte sind dabei erlaubt
  - Aber keine Zyklen mit negativen Gewicht! – In diesem ist der kürzeste Weg *undefiniert*!

    Deshalb betrachten wir hier nur *gerichtete* Graphen!
- Idee
  - Betrachte in jedem Schritt *alle* Kanten $e = (s, t) \in E$ "simultan"
  - Aktualisiere dabei Kosten zum Erreichen von t über s
  - Wiederhole $|V|$ Schritte, danach muss jeder (erreichbare) Knoten erreicht worden sein
- Aufwand $O(|V|\,|E|)$

Bellman-Ford Algorithmus

```
function bf(s₀)
  for each node n do
      d[n] = ∞; p[n] = -1
  end
  d[s₀]=0;

  for |V| iteration do
    # for all edges (s,t)
    for each s in V do
        for each e in outgoing(s) do
            t = destination(e)
            if d[t] > d[s] + γ(e)
                d[t] = d[s] + γ(e)
                p[t] = s
            end
    end end
  end
end
```

- Variante: Früher Abbruch außen, sobald keine Änderung in Iteration innen
- Variante: Betrachte in jeder Iteration nur Knoten, wo sich etwas ändern könnte

Referenz

- Saake & Sattler Kapitel 16.4.1, 16.4.2 & 16.4.5
- Goodrich, Tamassia & Goldwasser Kapitel 10.1 & 10.2
- Cormen et al. Kapitel 23 & 24
- Sedgewick & Wayne Kapitel 4.3 & 4.4