

# Algorithmen und Datenstrukturen (für ET/IT)

Sommersemester 2016

Dr. Tobias Lasser

Computer Aided Medical Procedures  
Technische Universität München



# Programm heute

7 Fortgeschrittene Datenstrukturen

8 Such-Algorithmen

9 Graph-Algorithmen

Tiefensuche

Breitensuche

**Kürzeste Pfade**

Minimaler Spannbaum

## Definition Kürzester Pfad

Sei  $G = (V, E)$  Graph mit **Gewichtsfunktion**

$$w : E \rightarrow \mathbb{R}.$$

- das **Gewicht eines Pfades**  $p = (v_1, \dots, v_k)$ ,  $v_i \in V$ ,  $i = 1, \dots, k$ , ist die Summe der Gewichte aller seiner Kanten

$$w(p) = \sum_{i=1}^{k-1} w(v_i, v_{i+1})$$

- das **Gewicht des kürzesten Pfades** zwischen zwei Knoten  $u$  und  $v$  ( $u, v \in V$ ) ist

$$w_{\min}(u, v) = \begin{cases} \min\{w(p) : p \text{ Pfad von } u \text{ nach } v\} & \text{Pfad existiert} \\ \infty & \text{sonst} \end{cases}$$

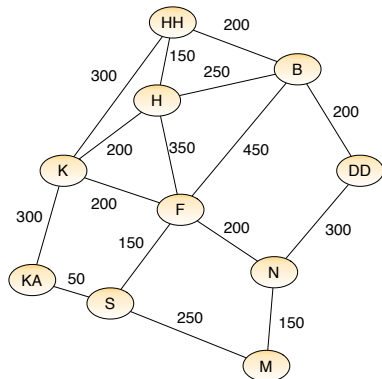
# Beispiel: kürzeste Pfade

## Navigationssystem:

- Knoten sind Großstädte
- Kanten sind Autobahnverbindung
- Gewichte der Kanten sind (grobe) Entfernungen in Kilometern

## Fragestellungen:

- kürzester Weg von M nach HH?
- alle kürzesten Verbindungen von M zu anderen Städten?
- alle kürzesten Verbindungen zwischen allen Städten?

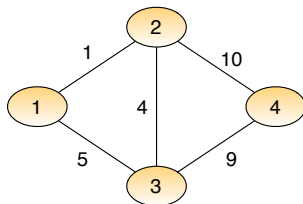


# Kürzeste Pfade: Algorithmen

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

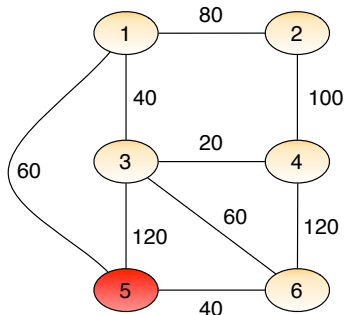
- Kürzeste Pfade in Graphen von fixem **Startknoten  $s$** :
  - **Dijkstra-Algorithmus** (nur positive Kantengewichte)  
Greedy-Algorithmus  
Komplexität:  $O(|E| \log |V|)$
  - **A\*-Algorithmus** (nur positive Kantengewichte)  
Komplexität: mit guter Heuristik, besser als Dijkstra
  - **Bellman-Ford-Algorithmus** (auch negative Gewichte)  
Komplexität:  $O(|V| * |E|)$
- Kürzeste Pfade in Graphen zwischen **allen Knotenpaaren**:
  - **Floyd-Warshall-Algorithmus** (all pairs shortest path)  
basiert auf dynamischer Programmierung  
Komplexität:  $\Theta(|V|^3)$

## Gegenbeispiel Greedy-Algorithmus



- **Greedy-Schema:** füge kürzeste Kante zu Pfad (so dass kein Zyklus entsteht)
  - Starte bei Knoten 1
  - füge Kante (1,2) hinzu (Kosten 1)
  - füge Kante (2,3) hinzu (Kosten 4)
  - füge Kante (3,4) hinzu (Kosten 9)
- Problem: Pfad von 1 nach 4 hat Kosten 14, kürzester Pfad hat aber Kosten 11!

# Beispiel: Ablauf Dijkstra-Algorithmus 1

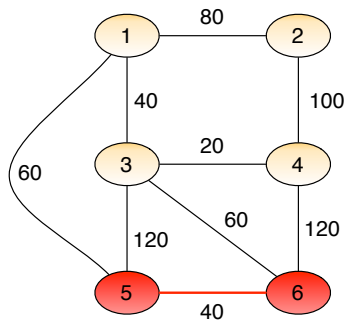


<u>Kürzester Pfad</u>		<u>Länge</u>
5		0

<u>Knoten</u>	<u>Pfad</u>	<u>Pfad Länge</u>
1	5	60
3	5	120
6	5	40

## Beispiel: Ablauf Dijkstra-Algorithmus 2

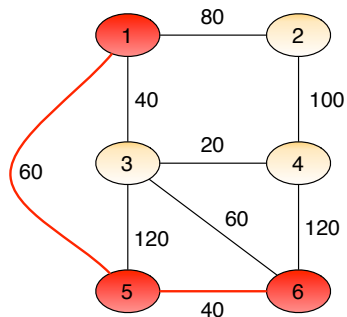


<u>Kürzester Pfad</u>	<u>Länge</u>
5	0
5,6	40

<u>Knoten</u>	<u>Pfad</u>	<u>Pfad</u>	<u>Länge</u>
<u>hinzugefügt</u>			
1	5	60	
3	5,6	100	
4	5,6	160	



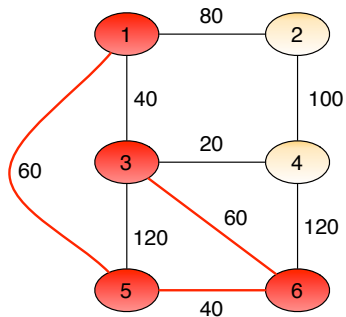
## Beispiel: Ablauf Dijkstra-Algorithmus 3



Kürzester Pfad	Länge
5	0
5,6	40
5,1	60

Knoten hinzugefügt	Pfad	Pfad Länge
2	5,1	140
3	5,6	100
4	5,6	160

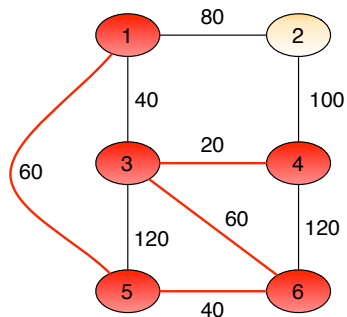
## Beispiel: Ablauf Dijkstra-Algorithmus 4



Kürzester Pfad	Länge
5	0
5,6	40
5,1	60
5,6,3	100

Knoten hinzugefügt	Pfad	Pfad Länge
2	5,1	140
4	5,6,3	120

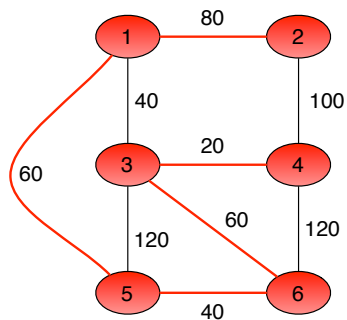
## Beispiel: Ablauf Dijkstra-Algorithmus 5



Kürzester Pfad	Länge
5	0
5,6	40
5,1	60
5,6,3	100
5,6,3,4	120

Knoten hinzugefügt	Pfad	Pfad Länge
2	5,1	140

## Beispiel: Ablauf Dijkstra-Algorithmus 6



<u>Kürzester Pfad</u>	<u>Länge</u>
5	0
5,6	40
5,1	60
5,6,3	100
5,6,3,4	120
5,1,2	140

fertig

# Dijkstra-Algorithmus

Sei  $G = (V, E)$  Graph mit Gewicht  $w : E \rightarrow \mathbb{R}^+$  (positive reelle Zahlen).

- Startknoten  $s \in V$  für kürzesten Pfad
- Graph repräsentiert mit Adjazenzliste  $adj$
- jeder Knoten (ausser  $s$ ) hat Vorgänger im kürzesten Pfad  $pred$
- jeder Knoten hat Markierung  $d$ 
  - Distanz (bzgl. Gewicht) von Startknoten  $s$
- Hilfsmittel: Priority Queue  $Q$

## Algorithmus: Dijkstra

**Input:** Graph  $G = (V, E)$ ,  $w : E \rightarrow \mathbb{R}^+$ , Startknoten  $s \in V$

**Output:** Vorgänger-Liste  $\text{pred}$ , Distanz-Markierung  $d$

**Dijkstra**( $G, w, s$ ):

```
for each (Knoten  $v \in V$ ) { // Initialisierung
```

```
     $\text{pred}[v] = \text{null}; d[v] = \infty;$ 
```

```
}
```

```
 $d[s] = 0;$ 
```

```
Q = Priority Queue mit Elementen  $V$ , Schlüsseln  $d$ ;
```

```
while ( !Q.isEmpty() ) {
```

```
     $u = Q.extractMin();$ 
```

```
    for each ( $v \in \text{adj}[u]$  mit  $v \in Q$ ) {
```

```
        if ( $d[u] + w(u, v) < d[v]$ ) {
```

```
             $\text{pred}[v] = u; d[v] = d[u] + w(u, v);$ 
```

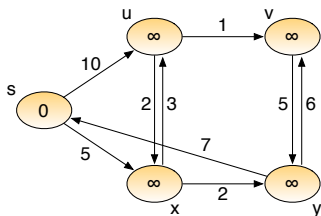
```
            Q.decreaseKey( $v, d[v]$ );
```

```
        }
```

```
    }
```

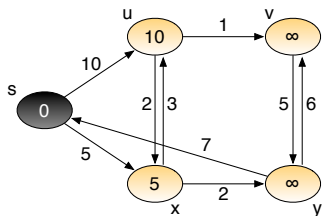
```
}
```

# Beispiel: Ablauf Dijkstra-Algorithmus 1



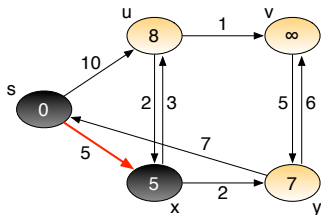
Q: 

(s,0)	(u, $\infty$ )	(v, $\infty$ )	(x, $\infty$ )	(y, $\infty$ )
-------	----------------	----------------	----------------	----------------



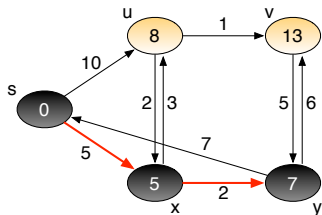
Q: 

(x, 5)	(u, 10)	(v, $\infty$ )	(y, $\infty$ )
--------	---------	----------------	----------------



Q: 

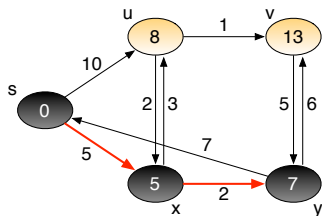
(y, 7)	(u, 8)	(v, $\infty$ )
--------	--------	----------------



Q: 

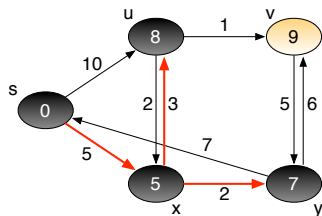
(u, 8)	(v, 13)
--------	---------

## Beispiel: Ablauf Dijkstra-Algorithmus 2



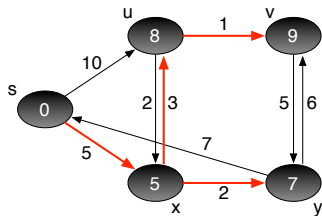
Q: 

(u, 8)	(v, 13)
--------	---------



Q: 

(v, 9)
--------



Q:



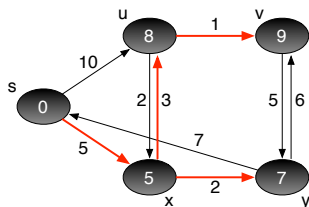
# Dijkstra-Algorithmus

Nach Ausführung von **Dijkstra**( $G, w, s$ ) gilt für  $v \in V$ :

- $d[v]$  = Gewicht  $w_{min}(s, v)$  des kürzesten Pfades von  $s$  zu  $v$
- $pred[v]$  = Vorgängerknoten
- Kürzester Pfad von  $s$  zu  $v$ :

$pred[v], pred[pred[v]], \dots, s$

Beispiel:



# Dijkstra: Laufzeit

Laufzeit:

- Annahme: Q implementiert als binärer Min-Heap
- Zeile 1-3:  $O(|V|)$
- Zeile 5: entspricht buildMinHeap, also  $O(|V|)$
- Zeile 6-15: Ausführung  $|V|$  mal
  - Zeile 7:  $O(\log |V|)$
  - Zeile 8-14: Ausführung inkl. äusserer while-Schleife: insgesamt  $|E|$  mal (siehe DFS/BFS)
  - Zeile 12:  $O(\log |V|)$

Gesamt:  $O((|V| + |E|) \log |V|)$

einfacher:  $O(|E| \log |V|)$

**Dijkstra**( $G, w, s$ ):

```
1  for each (Knoten  $v \in V$ ) {
2       $\text{pred}[v] = \text{null}; d[v] = \infty;$ 
3  }
4   $d[s] = 0;$ 
5   $Q = \text{Priority Queue}(V, d);$ 
6  while ( ! $Q.\text{isEmpty}()$  ) {
7       $u = Q.\text{extractMin}();$ 
8      for each ( $v \in \text{adj}[u]$  mit  $v \in Q$ ) {
9          if ( $d[u] + w(u, v) < d[v]$ ) {
10              $\text{pred}[v] = u;$ 
11              $d[v] = d[u] + w(u, v);$ 
12              $Q.\text{decreaseKey}(v, d[v]);$ 
13         }
14     }
15 }
```

# Dijkstra: Komplexität

- Komplexität des Dijkstra Algorithmus hängt entscheidend von der Implementierung der Priority Queue ab!
- Varianten:
  - als verkettete Liste:  $O(|V|^2)$
  - als binärer Heap:  $O(|E| \log |V|)$
  - als Fibonacci Heap:  $O(|E| + |V| \log |V|)$

# Dijkstra: Korrektheit

*(nur Beweisidee, dies ist kein formaler Beweis!)*

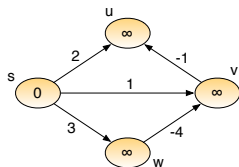
**Annahme:** bisherige Iterationen waren korrekt, bisher bearbeitete

Knoten:  $X \subset V$

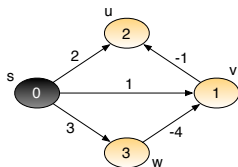
- nächster Iterationsschritt nimmt kürzeste direkte Verbindung von Knoten  $x \in X$  zu noch nicht bearbeitetem Knoten  $y \in V \setminus X$  hinzu
- $d[y]$  ist nun  $d[x] + w(x, y)$
- jeder andere Pfad zu  $y$  hat entweder
  - eine Kante, die aus  $X$  heraus geht und ist damit nicht kürzer als  $(x, y)$
  - mehrere Kanten, und ist damit nicht kürzer als  $d[y]$ , da die Kanten positives Gewicht haben

**Entscheidende Annahme:** Kanten haben positives Gewicht!

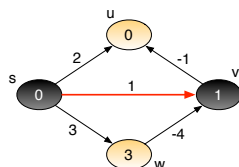
# Dijkstra: Negativ-Beispiel



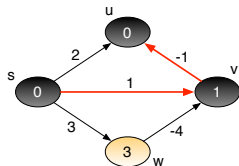
Q: (s, 0) (u, ∞) (v, ∞) (w, ∞)



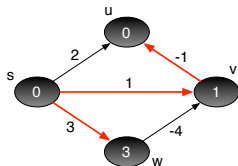
Q: (v, 1) (u, 2) (w, 3)



Q: (u, 0) (w, 3)



Q: (w, 3)



Q:

# Dijkstra: Anwendungen

Dijkstra ist einer der am häufigsten verwendete Graph-Algorithmen

## Beispiele:

- Routenplanung in GIS (Geographic Information System)
  - Navigationssystem im Auto
  - Maps Applikation (Google, Bing, Apple etc.)
  - Routen mit Flugzeugen, Bahn usw.
- Routing Protokolle für IP Netzwerke
  - z.B. Open Shortest Path First
- Pfadplanung von Robotern, UAV/Dronen, etc.
- Segmentierung von medizinischen Bilddaten

# Programm heute

7 Fortgeschrittene Datenstrukturen

8 Such-Algorithmen

9 Graph-Algorithmen

Tiefensuche

Breitensuche

Kürzeste Pfade

Minimaler Spannbaum

# Minimaler Spannbaum

Sei  $G = (V, E)$  zusammenhängender ungerichteter Graph mit Gewichtsfunktion  $w : E \rightarrow \mathbb{R}$ .

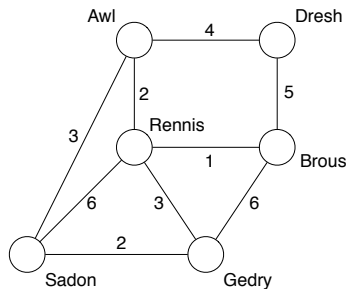
- **Spannbaum:** Teilgraph  $G' = (V, E')$ , der ein Baum ist und alle Knoten von  $G$  enthält.
- **minimaler Spannbaum:** Spannbaum  $G'$  mit minimalem Gewicht

$$w_{G'} = \sum_{(x,y) \in E'} w(x,y)$$



# Minimaler Spannbaum: Beispiel

**Beispiel:** 6 (virtuelle) Städte und Kosten für Strassenbau dazwischen (in Million Euro):

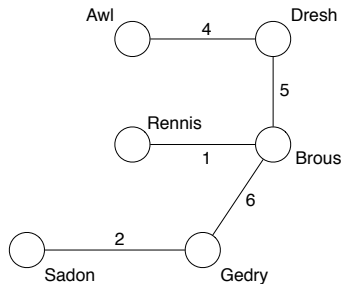


**Problem:** Strassenbau mit minimalen Kosten, so daß alle Städte verbunden sind (direkt oder über andere Städte)

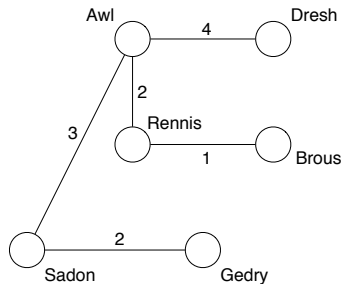
**Lösung:** minimaler Spannbaum

# Minimaler Spannbaum: Beispiel 2

Mögliche Lösungen:



Gewicht: 18



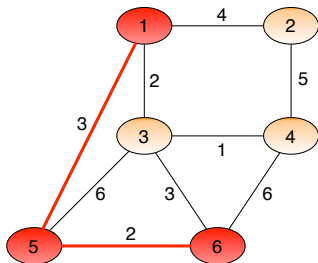
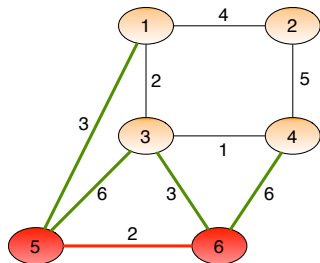
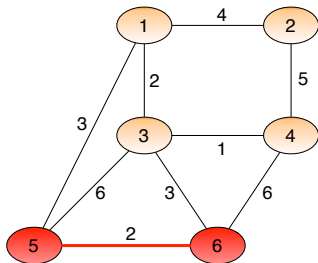
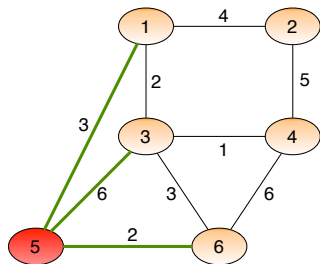
Gewicht: 12

# Minimaler Spannbaum: Algorithmen

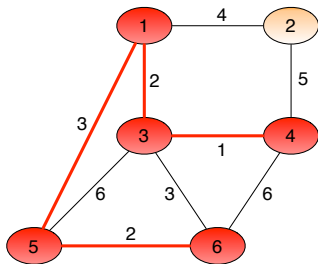
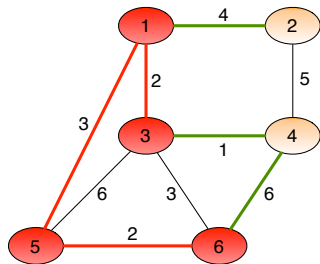
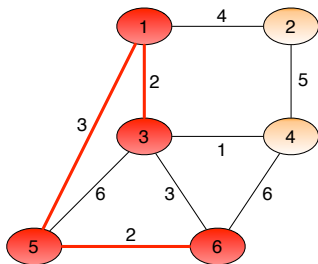
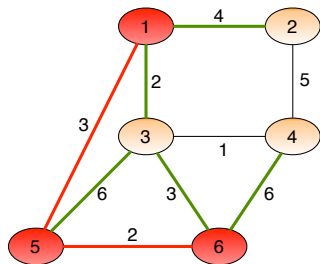
Sei  $G = (V, E)$  zusammenhängender ungerichteter Graph mit Gewichtsfunktion  $w : E \rightarrow \mathbb{R}$ .

- Minimaler Spannbaum von  $G$ :
  - Algorithmus von Kruskal: Greedy-Algorithmus  
Komplexität:  $O(|E| \log |V|)$
  - Algorithmus von Prim: Greedy-Algorithmus  
Komplexität:  $O(|E| \log |V|)$
  - viele Varianten davon als parallele Algorithmen

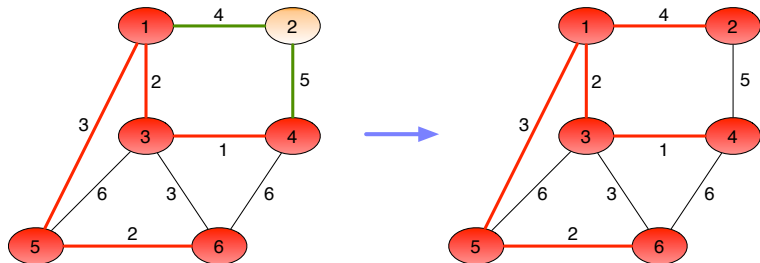
# Beispiel-Ablauf: Prim Algorithmus 1



## Beispiel-Ablauf: Prim Algorithmus 2



## Beispiel-Ablauf: Prim Algorithmus 3



### Beobachtungen:

- Zwischenlösungen von Prim Algorithmus sind Bäume
- es werden öfters mehrere Kanten zum selben Knoten betrachtet (s. oben)
  - Vereinfachung: betrachte nur Kante mit minimalem Gewicht

# Prim Algorithmus

Sei  $G = (V, E)$  zusammenhängender ungerichteter Graph mit Gewichtsfunktion  $w : E \rightarrow \mathbb{R}$ .

- Startknoten  $s \in V$  für minimalen Spannbaum
- Graph repräsentiert als Adjazenzliste  $adj$
- jeder Knoten (ausser  $s$ ) hat Vorgänger im Spannbaum  $pred$
- jeder Knoten hat Markierung  $g$ 
  - kleinstes Gewicht um Knoten mit aktuellem Spannbaum zu verbinden
- Hilfsmittel: Priority Queue  $Q$

## Algorithmus: Prim

**Input:** Graph  $G = (V, E)$ ,  $w : E \rightarrow \mathbb{R}$ , Startknoten  $s \in V$

**Output:** Vorgänger-Liste `pred`

**Prim**( $G, w, s$ ):

```
for each (Knoten  $v \in V$ ) { // Initialisierung
```

```
     $pred[v] = \text{null}; g[v] = \infty;$ 
```

```
}
```

```
 $g[s] = 0;$ 
```

```
Q = Priority Queue mit Elementen  $V$ , Schlüsseln  $g$ ;
```

```
while ( !Q.isEmpty() ) { // Hauptschleife
```

```
     $u = Q.extractMin();$ 
```

```
    for each ( $v \in adj[u]$  mit  $v \in Q$ ) {
```

```
        if ( $w(u, v) < g[v]$ ) {
```

```
             $pred[v] = u; g[v] = w(u, v);$ 
```

```
            Q.decreaseKey( $v, g[v]$ );
```

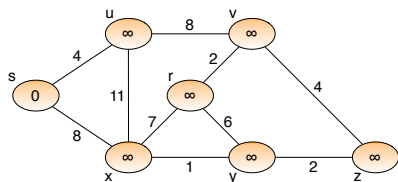
```
        }
```

```
    }
```

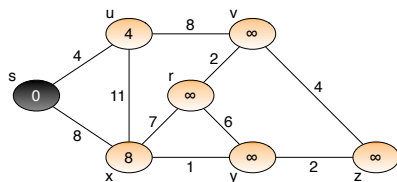
```
}
```



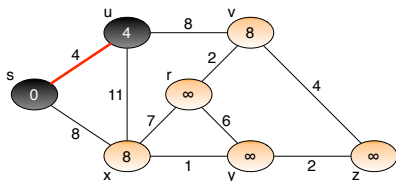
# Beispiel: Ablauf Prim Algorithmus 1



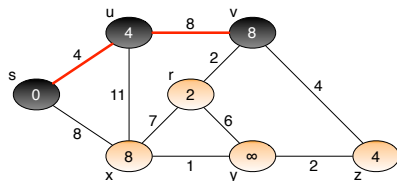
Q: (s, 0) (u,  $\infty$ ) (v,  $\infty$ ) (r,  $\infty$ ) (x,  $\infty$ ) (y,  $\infty$ ) (z,  $\infty$ )



Q: (u, 4) (x, 8) (r,  $\infty$ ) (v,  $\infty$ ) (y,  $\infty$ ) (z,  $\infty$ )

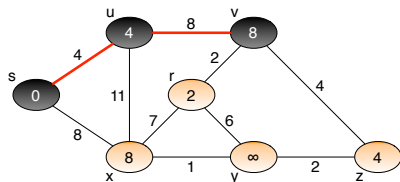


Q: (v, 8) (x, 8) (r,  $\infty$ ) (y,  $\infty$ ) (z,  $\infty$ )

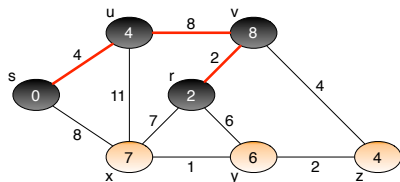


Q: (r, 2) (z, 4) (x, 8) (y,  $\infty$ )

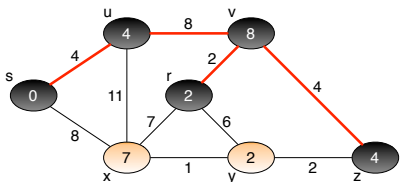
# Beispiel: Ablauf Prim Algorithmus 2



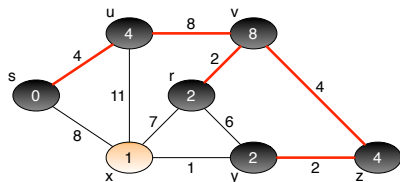
Q: (r, 2) (z, 4) (x, 8) (y, ∞)



Q: (z, 4) (y, 6) (x, 7)

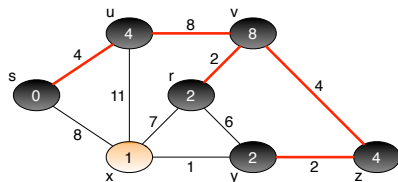


Q: (y, 2) (x, 7)

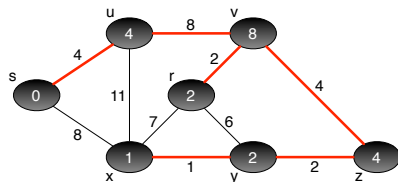


Q: (x, 1)

## Beispiel: Ablauf Prim Algorithmus 3



Q: (x, 1)



Q:

Nach Ausführung von  $\text{Prim}(G, w, s)$  gilt für  $v \in V$ :

- $\text{pred}[v]$  = Vorgängerknoten im Spannbaum
- Pfad in Spannbaum von  $v$  zu Wurzel  $s$ :

$\text{pred}[v], \text{pred}[\text{pred}[v]], \dots, s$

# Prim: Laufzeit

Laufzeit:

- Annahme: Q implementiert als binärer Min-Heap
- Zeile 1-3:  $O(|V|)$
- Zeile 5: entspricht buildMinHeap, also  $O(|V|)$
- Zeile 6-14: Ausführung  $|V|$  mal
  - Zeile 7:  $O(\log |V|)$
  - Zeile 8-13: Ausführung inkl. äusserer while-Schleife: insgesamt  $|E|$  mal (siehe DFS/BFS)
  - Zeile 11:  $O(\log |V|)$

Gesamt:  $O((|V| + |E|) \log |V|)$

einfacher:  $O(|E| \log |V|)$

**Prim**( $G, w, s$ ):

```
1  for each (Knoten  $v \in V$ ) {
2       $\text{pred}[v] = \text{null}; g[v] = \infty;$ 
3  }
4   $g[s] = 0;$ 
5   $Q = \text{Priority Queue}(V, g);$ 
6  while ( ! $Q.\text{isEmpty}()$  ) {
7       $u = Q.\text{extractMin}();$ 
8      for each ( $v \in \text{adj}[u]$  mit  $v \in Q$ ) {
9          if ( $w(u, v) < g[v]$ ) {
10              $\text{pred}[v] = u; g[v] = w(u, v);$ 
11              $Q.\text{decreaseKey}(v, g[v]);$ 
12         }
13     }
14 }
```

# Prim: Komplexität

- Komplexitätsanalyse von Prim fast identisch mit Dijkstra!
- Komplexität des Algorithmus von Prim hängt entscheidend von der Implementierung der Priority Queue ab!
- Varianten:
  - als verkettete Liste:  $O(|V|^2)$
  - als binärer Heap:  $O(|E| \log |V|)$
  - als Fibonacci Heap:  $O(|E| + |V| \log |V|)$

# Prim: Anwendungen

- Planung von Netzwerken
  - Strassennetz
  - Kommunikations-Netzwerk
  - elektronische Schaltungen
- Clustering von Daten
  - Daten als Knoten, "Nähe" als Kanten, entferne "lange" Kanten aus minimalem Spannbaum → Clustering
- Extrahieren/Tracking von Objekten aus Bildern in Computer Vision

# Ausblick: Graphen-Algorithmen

- **Fluss in Graphen:** statt Kantengewichten gibt es Kapazitäten, betrachtet wird Fluss von Quelle zu Senke
  - Problem: finde maximalen Fluss
  - Anwendung: z.B. Fluss in Kommunikations-Netzwerken
- **Einfärben von Graphen:** Färbe Knoten von Graph so ein, dass keine benachbarten Knoten dieselbe Farbe haben
  - Anwendungen: z.B. Scheduling, Sudoku
- **Planare Graphen:** lässt sich Graph ohne Kanten-Überschneidung zeichnen?
  - Anwendung: z.B. Chip- bzw. Platinen-Design
- **Klassifikation von medizinischen Daten:** über analytische Operationen auf Adjazenzmatrix, z.B. Laplace-Operator
  - Anwendungen: z.B. Identifikation von Melanomen, Tracking von Endoskopen

# Zusammenfassung

7 Fortgeschrittene Datenstrukturen

8 Such-Algorithmen

9 Graph-Algorithmen

- Tiefensuche

- Breitensuche

- Kürzeste Pfade

- Minimaler Spannbaum