

# Algorithmen und Datenstrukturen

## Kapitel 5: Graphalgorithmen

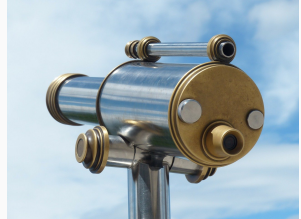
---

Prof. Dr. Peter Kling

Wintersemester 2020/21

# Übersicht

- 1 Elementare Graphalgorithmen
- 2 Minimale Spannbäume
- 3 Kürzeste Pfade
- 4 Paarweise kürzeste Pfade
- 5 Weiterführende Graphprobleme



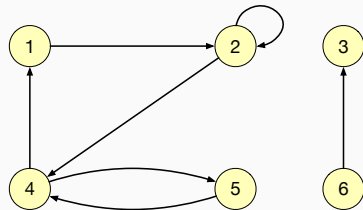
# 1) Elementare Graphalgorithmen

---

# Grundlagen: Gerichtete Graphen

## Gerichteter Graph $G = (V, E)$

- endliche Menge  $V$  von **Knoten**
- Menge  $E \subseteq V \times V$  von **Kanten**
  - Kante  $(u, v) \in E$  führt von  $u$  nach  $v$
  - Kanten der Form  $(u, u) \in E$  heißen **Schleife**



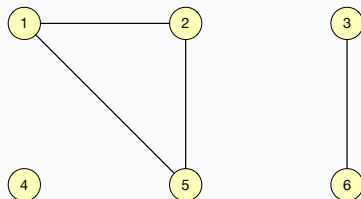
### Im Beispielgraph

- $V = \{1, 2, 3, 4, 5, 6\}$
  - $E = \{(1, 2), (2, 2), (2, 4), (4, 1), (4, 5), (5, 4), (5, 2), (6, 3)\}$
- Knoten  $u, v \in V$  sind **adjacent** falls  $(u, v) \in E$  oder  $(v, u) \in E$
  - der **Eingangsgrad** von  $u \in V$  ist  $\text{indeg}(u) := |\{(v, w) \in E \mid w = u\}|$
  - der **Ausgangsgrad** von  $u \in V$  ist  $\text{outdeg}(u) := |\{(v, w) \in E \mid v = u\}|$
  - der **Grad** von  $u \in V$  ist  $\text{deg}(u) := \text{indeg}(u) + \text{outdeg}(u)$

# Grundlagen: Ungerichtete Graphen

## Ungerichteter Graph $G = (V, E)$

- endliche Menge  $V$  von **Knoten**
- Menge  $E \subseteq \mathcal{P}(V)$  von **Kanten**
  - $e \in E \implies$  2-elementige Teilmenge von  $V$
  - **keine** Kanten der Form  $\{u, u\} \in E$



### Im Beispielgraph

- $V = \{1, 2, 3, 4, 5, 6\}$
  - $E = \{\{1, 2\}, \{1, 5\}, \{2, 5\}, \{3, 6\}\}$
- Knoten  $u, v \in V$  sind **adjacent** falls  $\{u, v\} \in E$
  - der **Grad** von  $u \in V$  ist  $\deg(u) := |\{e \in E \mid u \in e\}|$

# Algorithmen und Datenstrukturen

## Elementare Graphalgorithmen

### Grundlagen: Ungerichtete Graphen

- $\mathcal{P}(V)$  ist die **Potenzmenge** der Menge  $V$ , also die Menge aller Teilmengen von  $V$

#### Grundlagen: Ungerichtete Graphen

Ungerichteter Graph  $G = (V, E)$

- endliche Menge  $V$  von **Knoten**
- Menge  $E \subseteq \mathcal{P}(V)$  von **Kanten**
- $e \in E \implies$  2-elementige Teilmenge von  $V$
- **keine** Kanten der Form  $\{u, u\} \in E$



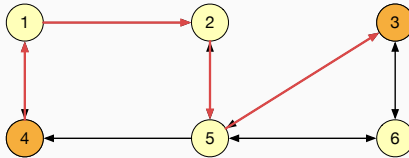
#### Im Beispielgraph

- $V = \{1, 2, 3, 4, 5, 6\}$
- $E = \{\{1, 2\}, \{1, 5\}, \{2, 5\}, \{3, 6\}\}$

- Knoten  $u, v \in V$  sind **adjacent** falls  $\{u, v\} \in E$
- der **Grad** von  $u \in V$  ist  $\deg(u) = |\{e \in E \mid u \in e\}|$

# Grundlagen: Weitere Begriffe

- **Pfad** der Länge  $k$ : Folge  $(v_0, v_1, \dots, v_k)$ , so dass  $\forall i \in \{1, 2, \dots, k\}$ :
  - gerichtet:  $(v_{i-1}, v_i) \in E$
  - ungerichtet:  $\{v_{i-1}, v_i\} \in E$
- **Distanz**  $\text{dist}(v, w)$  von  $v$  zu  $w$  in  $G$ 
  - gerichtet: Anzahl Kanten eines kürzesten **gerichteten** Pfades von  $v$  nach  $w$
  - ungerichtet: Anzahl Kanten eines kürzesten Pfades von  $v$  nach  $w$
- **Durchmesser**  $\text{diam} := \max \{ \text{dist}(v, w) \mid v, w \in V \}$



# Algorithmen und Datenstrukturen

## Elementare Graphalgorithmen

### Grundlagen: Weitere Begriffe

#### Grundlagen: Weitere Begriffe

- **Pfad** der Länge  $k$ : Folge  $(v_0, v_1, \dots, v_k)$ , so dass  $\forall i \in \{1, 2, \dots, k\}$ :
- gerichtet:  $(v_{i-1}, v_i) \in E$
- ungerichtet:  $\{v_{i-1}, v_i\} \in E$
- **Distanz**  $\text{dist}(v, w)$  von  $v$  zu  $w$  in  $G$
- gerichtet: Anzahl Kanten eines kürzesten gerichteten Pfades von  $v$  nach  $w$
- ungerichtet: Anzahl Kanten eines kürzesten Pfades von  $v$  nach  $w$
- **Durchmesser**  $\text{diam} := \max \{ \text{dist}(v, w) \mid v, w \in V \}$

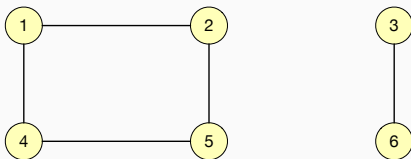


- Distanz von 4 nach 3 ist  $\text{dist}(4, 3) = 4$  (gleichzeitig Durchmesser)



## Grundlagen: Weitere Begriffe

- **Pfad** der Länge  $k$ : Folge  $(v_0, v_1, \dots, v_k)$ , so dass  $\forall i \in \{1, 2, \dots, k\}$ :
  - gerichtet:  $(v_{i-1}, v_i) \in E$
  - ungerichtet:  $\{v_{i-1}, v_i\} \in E$
- **Distanz**  $\text{dist}(v, w)$  von  $v$  zu  $w$  in  $G$ 
  - gerichtet: Anzahl Kanten eines kürzesten **gerichteten** Pfades von  $v$  nach  $w$
  - ungerichtet: Anzahl Kanten eines kürzesten Pfades von  $v$  nach  $w$
- **Durchmesser**  $\text{diam} := \max \{ \text{dist}(v, w) \mid v, w \in V \}$



### Zusammenhang im ungerichteten Graphen

- $G$  heißt **zusammenhängend** wenn  $\text{diam}(G)$  endlich ist
- d. h. es  $\exists$  Pfad von jedem Knoten zu jedem anderen Knoten

# Algorithmen und Datenstrukturen

## Elementare Graphalgorithmen

### Grundlagen: Weitere Begriffe

#### Grundlagen: Weitere Begriffe

- **Pfad** der Länge  $k$ : Folge  $(v_0, v_1, \dots, v_k)$ , so dass  $\forall i \in \{1, 2, \dots, k\}$ :
- **gerichtet**:  $(v_{i-1}, v_i) \in E$
- **ungerichtet**:  $\{v_{i-1}, v_i\} \in E$
- **Distanz**  $\text{dist}(v, w)$  von  $v$  zu  $w$  in  $G$
- **gerichtet**: Anzahl Kanten eines kürzesten **gerichteten** Pfades von  $v$  nach  $w$
- **ungerichtet**: Anzahl Kanten eines kürzesten Pfades von  $v$  nach  $w$
- **Durchmesser**  $\text{diam} := \max \{ \text{dist}(v, w) \mid v, w \in V \}$



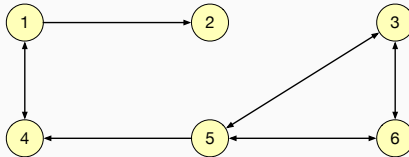
#### Zusammenhang im ungerichteten Graphen

- $G$  heißt **zusammenhängend** wenn  $\text{diam}(G)$  endlich ist
- d. h. es  $\exists$  Pfad von jedem Knoten zu jedem anderen Knoten

- Distanz von 4 nach 2 ist  $\text{dist}(4, 2) = 2$ ; Durchmesser ist  $\text{diam}(G) = \infty$  (Graph ist nicht zusammenhängend)

# Grundlagen: Weitere Begriffe

- **Pfad** der Länge  $k$ : Folge  $(v_0, v_1, \dots, v_k)$ , so dass  $\forall i \in \{1, 2, \dots, k\}$ :
  - gerichtet:  $(v_{i-1}, v_i) \in E$
  - ungerichtet:  $\{v_{i-1}, v_i\} \in E$
- **Distanz**  $\text{dist}(v, w)$  von  $v$  zu  $w$  in  $G$ 
  - gerichtet: Anzahl Kanten eines kürzesten **gerichteten** Pfades von  $v$  nach  $w$
  - ungerichtet: Anzahl Kanten eines kürzesten Pfades von  $v$  nach  $w$
- **Durchmesser**  $\text{diam} := \max \{ \text{dist}(v, w) \mid v, w \in V \}$



## Zusammenhang im gerichteten Graphen

- $G$  heißt **stark zusammenhängend** wenn  $\text{diam}(G)$  endlich ist
- $G$  heißt **schwach zusammenhängend** wenn  $\text{diam}(G)$  endlich ist, falls alle Kanten als **ungerichtet** angesehen werden

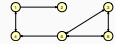
# Algorithmen und Datenstrukturen

## Elementare Graphalgorithmen

### Grundlagen: Weitere Begriffe

#### Grundlagen: Weitere Begriffe

- **Pfad** der Länge  $k$ : Folge  $(v_0, v_1, \dots, v_k)$ , so dass  $\forall i \in \{1, 2, \dots, k\}$ :
- **gerichtet**:  $(v_{i-1}, v_i) \in E$
- **ungerichtet**:  $\{v_{i-1}, v_i\} \in E$
- **Distanz**  $\text{dist}(v, w)$  von  $v$  zu  $w$  in  $G$
- **gerichtet**: Anzahl Kanten eines kürzesten **gerichteten** Pfades von  $v$  nach  $w$
- **ungerichtet**: Anzahl Kanten eines kürzesten Pfades von  $v$  nach  $w$
- **Durchmesser**  $\text{diam} := \max \{ \text{dist}(v, w) \mid v, w \in V \}$



#### Zusammenhang im gerichteten Graphen

- $G$  heißt **stark zusammenhängend** wenn  $\text{diam}(G)$  endlich ist
- $G$  heißt **schwach zusammenhängend** wenn  $\text{diam}(G)$  endlich ist, falls alle Kanten als **ungerichtet** angesehen werden

- Distanz von 4 nach 2 ist  $\text{dist}(4, 2) = 2$ ; Graph ist schwach zusammenhängend (ungerichteter Graph hat Durchmesser 4) aber nicht stark zusammenhängend

Im Folgenden sei  $G = (V, E)$  ein gerichteter Graph.

unger.  
analog

## Operationen

- INSERT( $G, u$ ):  $V \leftarrow V \cup \{u\}$
- INSERT( $G, e$ ):  $E \leftarrow E \cup \{e\}$
- REMOVE( $G, i, j$ ):  $E \leftarrow E \setminus \{e\}$  für  $e = (u, v)$ 
  - für  $u$  und  $v$  mit  $\text{key}(u) = i, \text{key}(v) = j$
- REMOVE( $G, i$ ):  $V \leftarrow V \setminus \{u\}, E \leftarrow E \setminus \{(x, y) \mid x = u \vee y = u\}$ 
  - für  $u$  mit  $\text{key}(u) = i$
- SEARCH( $G, i$ ): gib Knoten  $u$  mit  $\text{key}(u) = i$  aus
- SEARCH( $G, i, j$ ): gib Kante  $(u, v)$  mit  $\text{key}(u) = i$  und  $\text{key}(v) = j$  aus

# Grundlagen: eingeschränkte Knotenmengen

## Statische Knotenmenge

- oft ist die Knotenmenge  $V$  fest

⇒ dann  $V = \{1, 2, \dots, n\}$  mit  $\text{key}(u) = u$  für alle  $u \in V$

## Begrenzte Knotenmenge

- manchmal Knotenmenge  $V$  variabel, aber...
- ...Anzahl der Knoten nach oben begrenzt durch  $n$

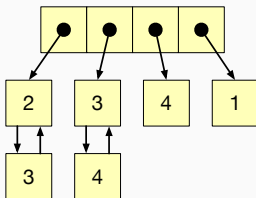
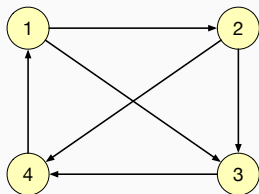
⇒ benutze Hashing!

- hashe Keys der  $n$  Knoten auf  $\{1, 2, \dots, \Theta(n)\}$

### In dieser Vorlesung

- hauptsächlich statische Knotenmengen
- Parameter für Laufzeitanalyse:
  - $n$ : Anzahl Knoten
  - $m$ : Anzahl Kanten
  - $d$ : maximaler (Ausgangs-) Knotengrad

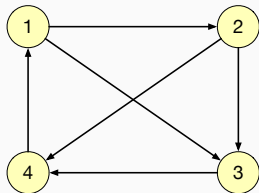
## Adjazenzliste



## Laufzeiten

- INSERT( $G, e$ ):  $O(d)$
- REMOVE( $G, i, j$ ):  $O(d)$
- SEARCH( $G, i, j$ ):  $O(d)$

## Adjazenzmatrix



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

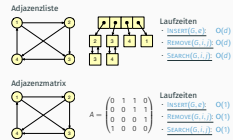
## Laufzeiten

- INSERT( $G, e$ ):  $O(1)$
- REMOVE( $G, i, j$ ):  $O(1)$
- SEARCH( $G, i, j$ ):  $O(1)$

# Algorithmen und Datenstrukturen

## └ Elementare Graphalgorithmen

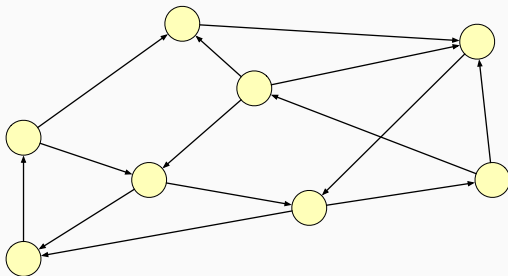
## Grundlagen: Graphrepräsentationen



- Nachteil Adjazenzlisten: Parameter  $d$  kann groß sein
- Nachteil Adjazenzmatrix: Platzbedarf  $\Theta(n^2)$  (schlecht für dünne Graphen)
- es gibt auch andere Repräsentationen
- z. B.: Adjazenzfeld, Adjazenzliste + Hashtabelle, implizite Repräsentationen, ...



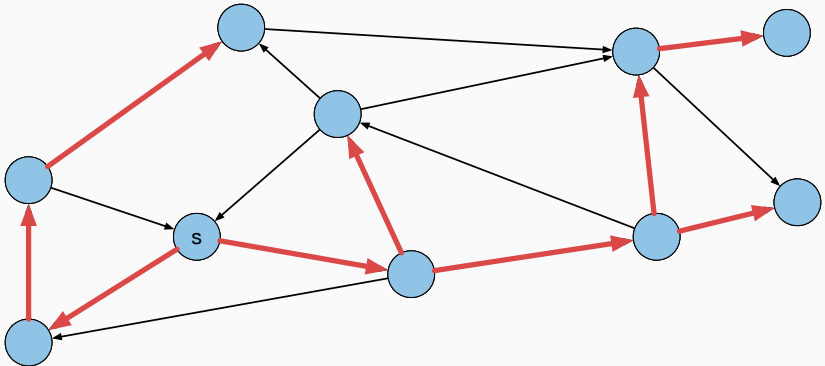
Wie können wir **alle** Knoten eines Graphen systematisch besuchen?



- Grundgerüst für spätere Graphalgorithmen
- zwei grundlegende Varianten:
  - (a) Breitensuche
  - (b) Tiefensuche

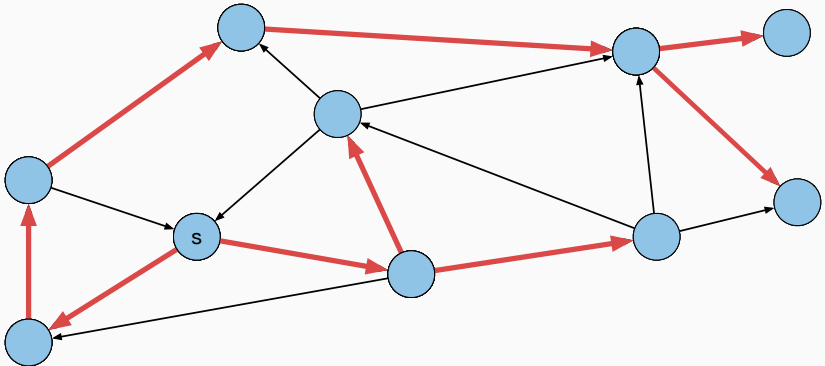
# Illustration Breitensuche

- starte von einem Knoten  $s$
- explore Graph „Distanz für Distanz“



# Illustration Tiefensuche

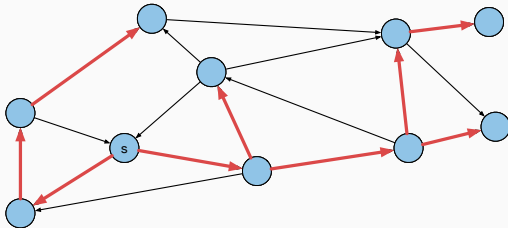
- starte von einem Knoten  $s$
- explore Graph „in die Tiefe“





Gegeben eine Quelle  $s \in V$ , finde alle Knoten  $v \in V$  die von  $s$  aus erreichbar sind.

- $v$  ist von  $s$  erreichbar, wenn es in  $G$  einen Pfad von  $s$  nach  $v$  gibt
- dabei berechnet die Breitensuche für alle  $v \in V$  die Distanz  $\text{dist}(s, v)$



---

## Algorithmus 5.1: BFS( $G, s$ )

---

```
1  for  $u \in V \setminus \{s\}$ 
2       $\text{color}(u) \leftarrow \text{white}$ 
3       $d(u) \leftarrow \infty$ 
4       $\pi(u) \leftarrow \text{NIL}$ 
5   $\text{color}(s) \leftarrow \text{gray}$ 
6   $d(s) \leftarrow 0$ 
7   $\pi(s) \leftarrow \text{NIL}$ 
8   $Q \leftarrow \emptyset$ 
9  ENQUEUE( $Q, s$ )
10 while  $Q \neq \emptyset$ 
11      $u \leftarrow \text{DEQUEUE}(Q)$ 
12     for  $v \in \text{Adj}(u)$ 
13         if  $\text{color}(v) = \text{white}$ 
14              $\text{color}(v) \leftarrow \text{gray}$ 
15              $d(v) \leftarrow d(u) + 1$ 
16              $\pi(v) \leftarrow u$ 
17             ENQUEUE( $Q, v$ )
18      $\text{color}(u) \leftarrow \text{black}$ 
```

---

## Knoten-Eigenschaften

- $\text{Adj}(u)$ : Adjazenzliste von  $u$
- $\text{color}(u)$ :
  - **white**: unentdeckt
  - **gray**: entdeckt, Adjazenzliste nicht ganz durchsucht
  - **black**: entdeckt, Adjazenzliste durchsucht
- $d(u)$ : bislang berechneter Abstand zu  $s$
- $\pi(u)$ : berechneter Vorgänger
- Queue  $Q$ : verwaltet grauen Knoten
- Zeilen 1 bis 9: Initialisierung
- Zeilen 10 bis 18: Hauptteil

Status

# Algorithmen und Datenstrukturen

## Elementare Graphalgorithmen

### Breitensuche: Pseudocode

- BFS steht für Breadth-first Search

Algorithmus 5.1: BFS( $G, s$ )

```

1  for  $u \in V \setminus \{s\}$ 
2    color( $u$ ) ← white
3   $d[s] \leftarrow \infty$ 
4   $\pi[s] \leftarrow \text{nil}$ 
5  color( $s$ ) ← gray
6   $d[s] \leftarrow 0$ 
7   $\pi[s] \leftarrow \text{nil}$ 
8   $Q \leftarrow s$ 
9  Dequeue( $Q, s$ )
10 while  $Q \neq \emptyset$ 
11    $u \leftarrow \text{Dequeue}(Q)$ 
12   for  $v \in \text{Adj}(u)$ 
13     if color( $v$ ) = white
14       color( $v$ ) ← gray
15        $d[v] \leftarrow d[u] + 1$ 
16        $\pi[v] \leftarrow u$ 
17       Dequeue( $Q, v$ )
18   color( $u$ ) ← black
```

## Knoten-Eigenschaften

- $\text{Adj}(u)$ : Adjazenzliste von  $u$
- color( $u$ ):
  - white: unentdeckt
  - gray: entdeckt, Adjazenzliste nicht ganz durchsucht
  - black: entdeckt, Adjazenzliste durchsucht
- $d[u]$ : bislang berechneter Abstand zu  $s$
- $\pi[u]$ : berechneter Vorgänger
- Queue  $Q$ : verwaltet grauen Knoten
- Zeilen 1 bis 5: Initialisierung
- Zeilen 10 bis 18: Hauptteil

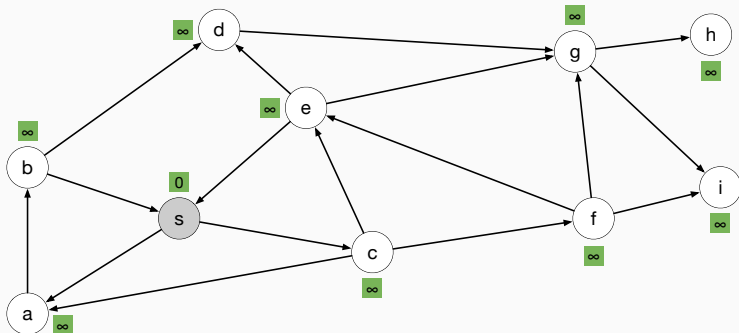
# Breitensuche: Durchlauf am Beispiel

BFS( $G, s$ )

```
1  „initialisiere BFS“  
2  while  $Q \neq \emptyset$   
3       $u \leftarrow \text{DEQUEUE}(Q)$   
4      for  $v \in \text{Adj}(u)$   
5          if  $\text{color}(v) = \text{white}$   
6               $\text{color}(v) \leftarrow \text{gray}; d(v) \leftarrow d(u) + 1$   
7               $\pi(v) \leftarrow u; \text{ENQUEUE}(Q, v)$   
8   $\text{color}(u) \leftarrow \text{black}$ 
```

Queue  $Q$

s

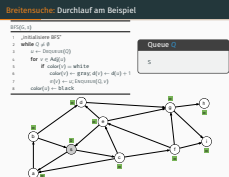


# Algorithmen und Datenstrukturen

## Elementare Graphalgorithmen

### Breitensuche: Durchlauf am Beispiel

- BFS steht für Breadth-first Search



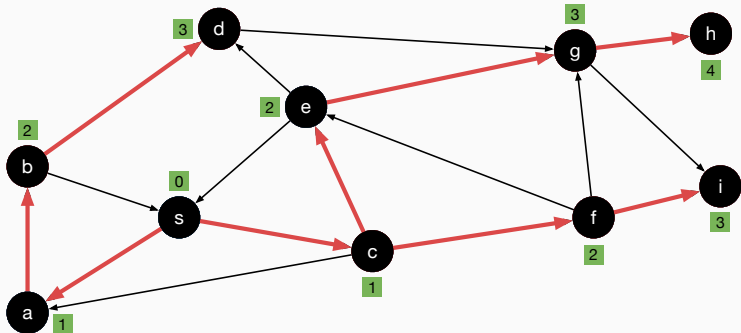


# Breitensuche: Durchlauf am Beispiel

BFS( $G, s$ )

```
1  „initialisiere BFS“  
2  while  $Q \neq \emptyset$   
3     $u \leftarrow \text{DEQUEUE}(Q)$   
4    for  $v \in \text{Adj}(u)$   
5      if  $\text{color}(v) = \text{white}$   
6         $\text{color}(v) \leftarrow \text{gray}$ ;  $d(v) \leftarrow d(u) + 1$   
7         $\pi(v) \leftarrow u$ ;  $\text{ENQUEUE}(Q, v)$   
8   $\text{color}(u) \leftarrow \text{black}$ 
```

Queue  $Q$

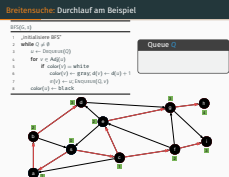


# Algorithmen und Datenstrukturen

## Elementare Graphalgorithmen

### Breitensuche: Durchlauf am Beispiel

- BFS steht für Breadth-first Search



## Theorem 5.1

Bei Eingabe eines Graphen  $G = (V, E)$  und Quelle  $s$  besitzt Algorithmus BFS Laufzeit  $O(|V| + |E|)$ .

### Beweis.

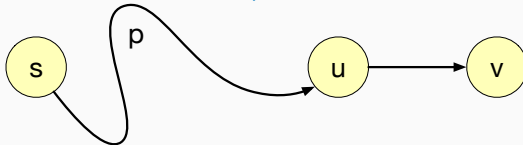
- Zeilen 1 bis 4:
    - Schleifenbody hat Laufzeit  $\Theta(1)$  und...
    - ...wird  $|V| - 1$  mal durchlaufen
  - Zeilen 5 bis 9:  $\Theta(1)$
  - Operationen auf Queue:
    - jeder Knoten wird nur einmal enqueued/dequeued...
    - ...da nur weiße Knoten enqueued werden (und grau werden)...
    - ...und nach Initialisierung kein Knoten weiß wird
  - Operationen auf Adjazenzlisten:
    - Adjazenzliste von jedem  $u$  wird höchstens einmal durchlaufen...
    - ...nämlich höchstens nach dem entfernen von  $u$  aus der Queue
- $O(|V|)$
- $O(|V|)$
- $O(|E|)$

## Lemma 5.1: Pfadkonkatenation

Sei  $G = (V, E)$  ein gerichteter Graph und  $s \in V$ . Für alle  $(u, v) \in E$  gilt  $\text{dist}(s, v) \leq \text{dist}(s, u) + 1$ .

### Beweis.

- $u$  nicht von  $s$  erreichbar  $\implies \text{dist}(s, v) \leq \infty + 1 = \text{dist}(s, u) + 1$
- sei also  $u$  von  $s$  erreichbar und  $p$  ein kürzester Pfad von  $s$  nach  $u$



- dann ist  $p' = p \circ (u, v)$  ein Pfad von  $s$  nach  $v$

$\implies$  (da  $\text{dist}(s, v)$  nach Definition minimale Pfadlänge von  $s$  nach  $v$ )

$$\text{dist}(s, v) \leq |p'| = |p| + 1 = \text{dist}(s, u) + 1$$

# Algorithmen und Datenstrukturen

## Elementare Graphalgorithmen



### Breitensuche: Kürzeste Pfade – Hilfslemma 1

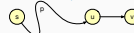
- Lemma 5.1 gilt analog für ungerichtete Graphen

#### Lemma 5.1: Pfadkonkatenation

Sei  $G = (V, E)$  ein gerichteter Graph und  $s \in V$ . Für alle  $(u, v) \in E$  gilt  $\text{dist}(s, v) \leq \text{dist}(s, u) + 1$ .

#### Beweis.

- $u$  nicht von  $s$  erreichbar  $\implies \text{dist}(s, v) \leq \infty + 1 = \text{dist}(s, u) + 1$
- sei also  $u$  von  $s$  erreichbar und  $p$  ein kürzester Pfad von  $s$  nach  $u$



- dann ist  $p' = p \circ (u, v)$  ein Pfad von  $s$  nach  $v$
- $\implies$  (da  $\text{dist}(s, v)$  nach Definition minimale Pfadlänge von  $s$  nach  $v$ )
- $$\text{dist}(s, v) \leq |p'| = |p| + 1 = \text{dist}(s, u) + 1$$

### Lemma 5.2: Obere Distanzschranke

Sei  $G = (V, E)$  ein gerichteter Graph und  $s \in V$ . Betrachte BFS mit Eingabe  $(G, s)$ . Nach Beendigung von BFS gilt für alle  $u \in V$   $d(u) \geq \text{dist}(s, u)$ .

### Beweisskizze.

- sei  $S$  die Menge der Knoten, die bereits in der Queue  $Q$  waren oder sind
  - Schleifeninvariante: Für alle  $u \in S$  gibt es einen Pfad der Länge  $d(u)$  von  $s$  nach  $u$
- ↪ Terminierung der Invariante ergibt die gewünschte Aussage



# Algorithmen und Datenstrukturen

## Elementare Graphalgorithmen

### Breitensuche: Kürzeste Pfade – Hilfslemma 2

- Lemma 5.2 gilt analog für ungerichtete Graphen

**Lemma 5.2: Obere Distanzschranke**

Sei  $G = (V, E)$  ein gerichteter Graph und  $s \in V$ . Betrachte BFS mit Eingabe  $(G, s)$ . Nach Beendigung von BFS gilt für alle  $u \in V$   $d(u) \geq \text{dist}(s, u)$ .

**Beweisskizze.**

- sei  $S$  die Menge der Knoten, die bereits in der Queue  $Q$  waren oder sind
  - Schleifeninvariante: Für alle  $u \in S$  gibt es einen Pfad der Länge  $d(u)$  von  $s$  nach  $u$
- Terminierung der Invariante ergibt die gewünschte Aussage  $\square$

### Lemma 5.3: Queue Distanzen

Zu einem beliebigen Zeitpunkt sei der Inhalt der Queue  $Q$  gegeben durch die Knoten  $(u_1, u_2, \dots, u_k)$  (Kopf  $u_1$ , Tail  $u_k$ ). Dann gilt  $d(u_i) \leq d(u_{i+1})$  für alle  $i \in \{1, 2, \dots, k-1\}$  und  $d(u_k) \leq d(u_1) + 1$ .

### Beweisskizze.

- Beweis per Schleifeninvariante
- Schleifeninvariante: die Aussage des Lemmas



### Korollar 5.1: Zeitliche Monotonie

Betrachte Knoten  $u_i$  und  $u_j$  die von BFS in die Queue eingefügt werden. Dabei werde  $u_i$  vor  $u_j$  eingefügt. Zu dem Zeitpunkt, zu dem  $u_j$  eingefügt wird, gilt  $d(u_i) \leq d(u_j)$ .



# Algorithmen und Datenstrukturen

## Elementare Graphalgorithmen

### Breitensuche: Kürzeste Pfade – Hilfslemma 3

- Korollar 5.1 gilt nach Lemma 5.3 und der Beobachtung, dass jeder Knoten  $u$  höchstens einmal einen endlichen Distanzwert  $d(u)$  zugewiesen bekommt.

**Lemma 5.3: Queue Distanzen**

Zu einem beliebigen Zeitpunkt sei der Inhalt der Queue  $Q$  gegeben durch die Knoten  $(u_1, u_2, \dots, u_b)$  (Kopf  $u_1$ , Tail  $u_b$ ). Dann gilt  $d(u_i) \leq d(u_{i+1})$  für alle  $i \in \{1, 2, \dots, b-1\}$  und  $d(u_b) \leq d(u_1) + 1$ .

**Beweisskizze.**

- Beweis per Schleifenvariante
- Schleifenvariante: die Aussage des Lemmas

□

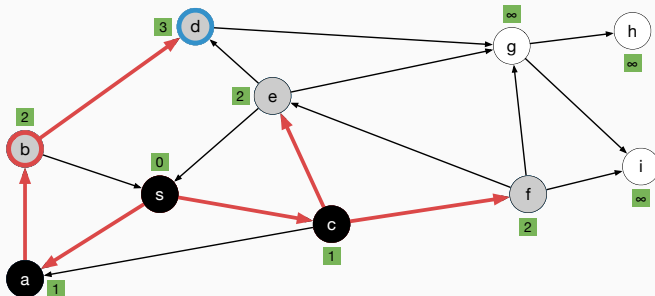
**Korollar 5.1: Zeitliche Monotonie**

Betrachte Knoten  $u$  und  $v$ , die von BFS in die Queue eingefügt werden. Dabei werde  $u$  vor  $v$  eingefügt. Zu dem Zeitpunkt, zu dem  $v$  eingefügt wird, gilt  $d(u) \leq d(v)$ .

### Theorem 5.2: BFS Kürzeste Pfade

Sei  $G = (V, E)$  ein gerichteter oder ungerichteter Graph und  $s \in V$ . Betrachte BFS mit Eingabe  $(G, s)$ . Nach Beendigung von BFS gilt für alle  $u \in V$   $d(u) = \text{dist}(s, u)$ .

Zu jedem von  $s$  erreichbaren Knoten  $u \neq s$  erhält man einen kürzesten Pfad von  $s$  zu  $u$  durch Konkatination eines kürzesten Pfades von  $s$  zum Vorgänger  $\pi(u)$  und der Kante  $(\pi(u), u)$ .



# Algorithmen und Datenstrukturen

## Elementare Graphalgorithmen

### Breitensuche: Kürzeste Pfade

## Theorem 5.2: BFS Kürzeste Pfade

Sei  $G = (V, E)$  ein gerichteter oder ungerichteter Graph und  $s \in V$ . Betrachte BFS mit Eingabe  $(G, s)$ . Nach Beendigung von BFS gilt für alle  $u \in V$   $d(u) = \text{dist}(s, u)$ .

Zu jedem von  $s$  erreichbaren Knoten  $u \neq s$  erhält man einen kürzesten Pfad von  $s$  zu  $u$  durch Konkatenation eines kürzesten Pfades von  $s$  zum Vorgänger  $v(u)$  und der Kante  $(v(u), u)$ .



- insbesondere gilt nach Theorem 5.2 für jeden von  $s$  aus erreichbaren Knoten  $u$ , dass  $d(u) < \infty$

## Breitensuche: Beweisskizze Theorem 5.2 (1/2)

- zeigen nur erste Aussage; zweite folgt leicht daraus
- $d(v) \geq \text{dist}(s, v)$  gilt nach Lemma 5.2

**Annahme:**  $\exists v \in V$  mit  $d(v) > \text{dist}(s, v)$

- wähle solch ein  $v$  mit minimalem  $\text{dist}(s, v)$  und beachte dass:
  - $v \neq s$ , da  $d(s) = 0 = \text{dist}(s, s)$
  - $v$  ist von  $s$  erreichbar, da sonst  $d(v) = \infty = \text{dist}(s, v)$
- sei  $u$  Vorgänger von  $v$  auf kürzestem Pfad von  $s$  nach  $v$   
 $\implies \text{dist}(s, v) = \text{dist}(s, u) + 1$
- da  $v$  minimal gewählt ist, muss  $d(u) = \text{dist}(s, u)$  gelten  
 $\implies d(v) > \text{dist}(s, v) = \text{dist}(s, u) + 1 = d(u) + 1$

# Algorithmen und Datenstrukturen

## Elementare Graphalgorithmen

### Breitensuche: Beweisskizze Theorem 5.2 (1/2)

Breitensuche: Beweisskizze Theorem 5.2 (1/2)

- zeigen nur erste Aussage; zweite folgt leicht daraus
- $d(v) \geq \text{dist}(s, v)$  gilt nach Lemma 5.2

Annahme:  $\exists v \in V$  mit  $d(v) > \text{dist}(s, v)$ 

- wähle solch ein  $v$  mit minimalem  $\text{dist}(s, v)$  und beachte dass:
  - $v \neq s$ , da  $d(s) = 0 = \text{dist}(s, s)$
  - $v$  ist von  $s$  erreichbar, da sonst  $d(v) = \infty = \text{dist}(s, v)$
- sei  $u$  Vorgänger von  $v$  auf kürzestem Pfad von  $s$  nach  $v$ 
  - $\implies \text{dist}(s, v) = \text{dist}(s, u) + 1$
- da  $v$  minimal gewählt ist, muss  $d(u) = \text{dist}(s, u)$  gelten
  - $\implies d(v) > \text{dist}(s, v) = \text{dist}(s, u) + 1 = d(u) + 1$

- zweite Aussage folgt, da  $d(v) = d(\pi(v)) + 1$  gilt und...
- ...und nach erster Aussage sowohl  $d(v)$  als auch  $d(\pi(v))$  die Längen von kürzesten Pfaden zu den jeweiligen Knoten sind
- erhalten also durch Anhängen von  $(\pi(v), v)$  and kürzestem Pfad von  $s$  nach  $\pi(v)$  einen Pfad der (kürzesten) Länge  $d(\pi(v)) + 1$  von  $s$  nach  $v$

## Breitensuche: Beweisskizze Theorem 5.2 (2/2)

- haben gezeigt:  $d(v) > d(u) + 1$
- betrachte Farbe von  $v$  zum Zeitpunkt der Entnahme von  $u$  aus  $Q$ :

Fall 1:  $\text{color}(v) = \text{white}$

$$\implies d(v) = d(u) + 1 \text{ ⚡}$$

Fall 2:  $\text{color}(v) = \text{black}$

- $v$  wurde bereits aus  $Q$  entfernt, also auch vor  $u$  in  $Q$  eingefügt

$$\implies (\text{Korollar 5.1}) d(v) \leq d(u) \text{ ⚡}$$

Fall 3:  $\text{color}(v) = \text{gray}$

- sei  $w$  Knoten, bei dessen Entnahme  $v$  grau wurde

$$\implies (\text{wieder Korollar 5.1}) d(w) \leq d(u)$$

$$\implies (\text{Zeile 7}) d(v) = d(w) + 1 \leq d(u) + 1 \text{ ⚡}$$

---

```
1 „initialisiere BFS“
2 while  $Q \neq \emptyset$ 
3    $u \leftarrow \text{DEQUEUE}(Q)$ 
4   for  $v \in \text{Adj}(u)$ 
5     if  $\text{color}(v) = \text{white}$ 
6        $\text{color}(v) \leftarrow \text{gray};$ 
7        $d(v) \leftarrow d(u) + 1$ 
8        $\pi(v) \leftarrow u; \text{ENQUEUE}(Q, v)$ 
9    $\text{color}(u) \leftarrow \text{black}$ 
```

---

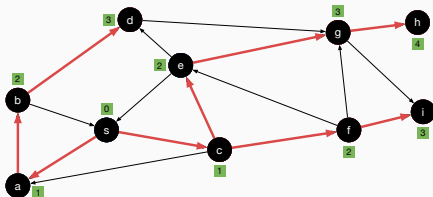
$$\implies \text{erhalten erste Aussage } (d(v) = \text{dist}(s, v))$$

## Breitensuche: Breitensuchbäume

Gegeben einen gerichteten Graphen  $G = (V, E)$  und  $s \in V$  definiere den Graphen  $G_\pi = (V_\pi, E_\pi)$  anhand der von BFS berechneten Werte wie folgt:

$$V_\pi = \{v \in V \mid \pi(v) \neq \text{NIL}\} \cup \{s\}$$

$$E_\pi = \{(\pi(v), v) \mid v \in V_\pi \setminus \{s\}\}$$



### Theorem 5.3: Breitensuchbaum

Der Graph  $G_\pi = (V_\pi, E_\pi)$  ist ein Baum über allen von  $s$  aus in  $G$  erreichbaren Knoten. Für alle  $v \in V_\pi$  ist der eindeutige Pfad von  $s$  zu  $v$  in  $V_\pi$  ein kürzester Pfad von  $s$  zu  $v$  in  $G$ .

# Algorithmen und Datenstrukturen

## Elementare Graphalgorithmen

### Breitensuche: Breitensuchbäume

- für ungerichtete Graphen konstruiert man  $G_\pi$  analog
- Theorem 5.3 folgt einfach durch Beobachtung, dass  $|E_\pi| = |V_\pi| - 1$  gilt (also ist  $G_\pi$  ein Baum) und durch induktive Anwendung der zweiten Aussagen von Theorem 5.2

#### Breitensuche: Breitensuchbäume

Gegeben einen gerichteten Graphen  $G = (V, E)$  und  $s \in V$  definiere den Graphen  $G_s = (V_s, E_s)$  anhand der von BFS berechneten Werte wie folgt:

$$V_s = \{v \in V \mid \pi(v) \neq \text{NIL}\} \cup \{s\}$$

$$E_s = \{(\pi(v), v) \mid v \in V_s \setminus \{s\}\}$$



#### Theorem 5.3: Breitensuchbaum

Der Graph  $G_s = (V_s, E_s)$  ist ein Baum über allen von  $s$  aus in  $G$  erreichbaren Knoten. Für alle  $v \in V_s$  ist der eindeutige Pfad von  $s$  zu  $v$  in  $V_s$  ein kürzester Pfad von  $s$  zu  $v$  in  $G$ .





# Tiefensuche: Pseudocode (iterativ)

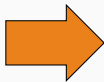
---

BFS( $G, s$ )

---

```
1  for  $u \in V \setminus \{s\}$ 
2      color( $u$ )  $\leftarrow$  white
3      d( $u$ )  $\leftarrow \infty$ 
4       $\pi(u) \leftarrow \text{NIL}$ 
5  color( $s$ )  $\leftarrow$  gray
6  d( $s$ )  $\leftarrow 0$ 
7   $\pi(s) \leftarrow \text{NIL}$ 
8   $Q \leftarrow \emptyset$ 
9  ENQUEUE( $Q, s$ )
10 while  $Q \neq \emptyset$ 
11      $u \leftarrow \text{DEQUEUE}(Q)$ 
12     for  $v \in \text{Adj}(u)$ 
13         if color( $v$ ) = white
14             color( $v$ )  $\leftarrow$  gray
15             d( $v$ )  $\leftarrow$  d( $u$ ) + 1
16              $\pi(v) \leftarrow u$ 
17             ENQUEUE( $Q, v$ )
18     color( $u$ )  $\leftarrow$  black
```

---



---

Algorithmus 5.2: DFS( $G, s$ )

---

```
1  for  $u \in V \setminus \{s\}$ 
2      color( $u$ )  $\leftarrow$  white
3      d( $u$ )  $\leftarrow \infty$ 
4       $\pi(u) \leftarrow \text{NIL}$ 
5  color( $s$ )  $\leftarrow$  gray
6  d( $s$ )  $\leftarrow 0$ 
7   $\pi(s) \leftarrow \text{NIL}$ 
8   $Q \leftarrow \emptyset$ 
9  PUSH( $Q, s$ )
10 while  $Q \neq \emptyset$ 
11      $u \leftarrow \text{POP}(Q)$ 
12     for  $v \in \text{Adj}(u)$ 
13         if color( $v$ ) = white
14             color( $v$ )  $\leftarrow$  gray
15             d( $v$ )  $\leftarrow$  d( $u$ ) + 1
16              $\pi(v) \leftarrow u$ 
17             PUSH( $Q, v$ )
18     color( $u$ )  $\leftarrow$  black
```

---

# Tiefensuche: Pseudocode (rekursiv)

---

## Algorithmus 5.3: DFS( $G$ )

---

```
1  for  $u \in V$ 
2      color( $u$ )  $\leftarrow$  white
3       $\pi(u) \leftarrow \text{NIL}$ 
4  time  $\leftarrow$  0
5  for  $u \in V$ 
6      if color( $u$ ) = white
7          DFS-VISIT( $u$ )
```

---

---

## Algorithmus 5.4: DFS-VISIT( $u$ )

---

```
1  color( $u$ )  $\leftarrow$  gray
2  time  $\leftarrow$  time + 1
3  d( $u$ )  $\leftarrow$  time
4  for  $v \in \text{Adj}(u)$ 
5      if color( $v$ ) = white
6           $\pi(v) \leftarrow u$ 
7          DFS-VISIT( $v$ )
8  color( $u$ )  $\leftarrow$  black
9  time  $\leftarrow$  time + 1
10 f( $u$ )  $\leftarrow$  time
```

---

## Knoten-Eigenschaften

- $\text{Adj}(u)$ : Adjazenzliste von  $u$
- $\text{color}(u)$ :
  - **white**: unentdeckt
  - **gray**: entdeckt, Adjazenzliste nicht ganz durchsucht
  - **black**: entdeckt, Adjazenzliste durchsucht
- Zwei „Zeitstempel“ (zwischen 1 und  $2 \cdot |V|$ )
  - $d(u)$ : Zeit, zu der  $u$  entdeckt wird
  - $f(u)$ : Zeit, zu der  $u$  abgearbeitet ist

Status

# Algorithmen und Datenstrukturen

## Elementare Graphalgorithmen

### Tiefensuche: Pseudocode (rekursiv)

- DFS steht für Depth-first Search

#### Tiefensuche: Pseudocode (rekursiv)

##### Algorithmus 5.3: DFS( $u$ )

```

1 for  $u \in V$ 
2   color( $u$ ) ← white
3   adj( $u$ ) ← NIL
4 time ← 0
5 for  $u \in V$ 
6   if color( $u$ ) = white
7     DFS-Visit( $u$ )

```

##### Algorithmus 5.4: DFS-Visit( $u$ )

```

1 color( $u$ ) ← gray
2 time ← time + 1
3 d( $u$ ) ← time
4 for  $v \in \text{Adj}(\mathbf{u})$ 
5   if color( $v$ ) = white
6     s( $v$ ) ← u
7     DFS-Visit( $v$ )
8 color( $u$ ) ← black
9 time ← time + 1
10 f( $u$ ) ← time

```

#### Knoten-Eigenschaften

- Adj( $u$ ): Adjazenzliste von  $u$
- color( $u$ ):
  - white: unentdeckt
  - gray: entdeckt, Adjazenzliste nicht ganz durchsucht
  - black: entdeckt, Adjazenzliste durchsucht
- Zwei „Zeitstempel“ (zwischen 1 und  $2 \cdot |V|$ )
  - d( $u$ ): Zeit, zu der  $u$  entdeckt wird
  - f( $u$ ): Zeit, zu der  $u$  abgearbeitet ist

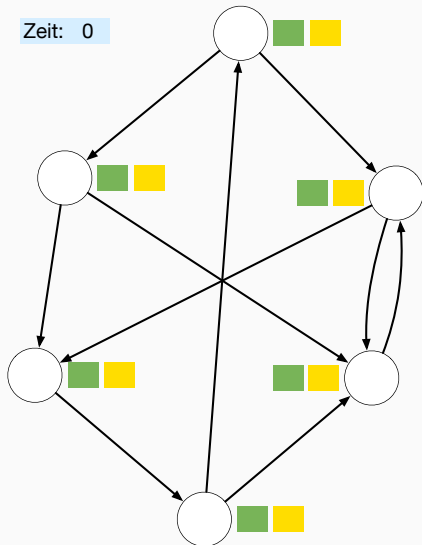
# Tiefensuche: Durchlauf am Beispiel

DFS( $G$ )

```
1  for  $u \in V$ 
2    color( $u$ )  $\leftarrow$  white
3     $\pi(u) \leftarrow$  NIL
4  time  $\leftarrow$  0
5  for  $u \in V$ 
6    if color( $u$ ) = white
7      DFS-VISIT( $u$ )
```

DFS-VISIT( $u$ )

```
1  color( $u$ )  $\leftarrow$  gray
2  time  $\leftarrow$  time + 1
3  d( $u$ )  $\leftarrow$  time
4  for  $v \in \text{Adj}(u)$ 
5    if color( $v$ ) = white
6       $\pi(v) \leftarrow u$ 
7      DFS-VISIT( $v$ )
8  color( $u$ )  $\leftarrow$  black
9  time  $\leftarrow$  time + 1
10 f( $u$ )  $\leftarrow$  time
```



# Algorithmen und Datenstrukturen

## Elementare Graphalgorithmen

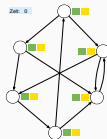
### Tiefensuche: Durchlauf am Beispiel

- DFS steht für Depth-first Search

```

DFS(u)
1 for v in V
2   color[v] ← white
3   d[v] ← nil
4   time ← 0
5 for u in V
6   if color[u] = white
7     DFSVisit(u)

DFSVisit(u)
1 color[u] ← gray
2 time ← time + 1
3 d[u] ← time
4 for v in Adj(u)
5   if color[v] = white
6     DFSVisit(v)
7   color[u] ← black
8 time ← time + 1
9 f[u] ← time
  
```



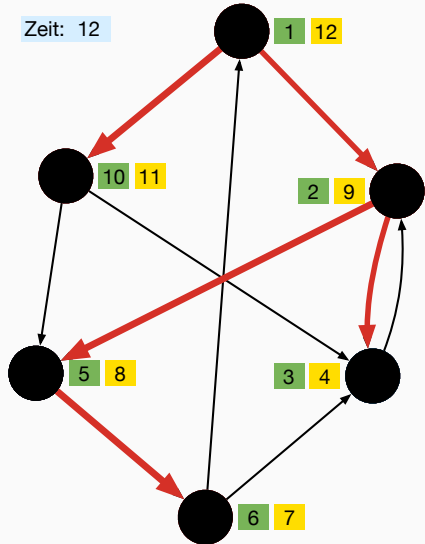
# Tiefensuche: Durchlauf am Beispiel

DFS( $G$ )

```
1  for  $u \in V$ 
2    color( $u$ )  $\leftarrow$  white
3     $\pi(u) \leftarrow \text{NIL}$ 
4  time  $\leftarrow$  0
5  for  $u \in V$ 
6    if color( $u$ ) = white
7      DFS-VISIT( $u$ )
```

DFS-VISIT( $u$ )

```
1  color( $u$ )  $\leftarrow$  gray
2  time  $\leftarrow$  time + 1
3  d( $u$ )  $\leftarrow$  time
4  for  $v \in \text{Adj}(u)$ 
5    if color( $v$ ) = white
6       $\pi(v) \leftarrow u$ 
7      DFS-VISIT( $v$ )
8  color( $u$ )  $\leftarrow$  black
9  time  $\leftarrow$  time + 1
10 f( $u$ )  $\leftarrow$  time
```



Kanten von  $\pi(v)$  zu  $v$  bilden Tiefensuchbaum

# Algorithmen und Datenstrukturen

## Elementare Graphalgorithmen

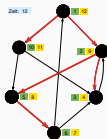
### Tiefensuche: Durchlauf am Beispiel

- DFS steht für Depth-first Search

```

DFS(u)
1 for v in V
2   color[v] ← white
3   d[v] ← nil
4   time ← 0
5   for v in V
6     if color[v] = white
7       DFSVisit(v)

DFSVisit(u)
1 color[u] ← gray
2 time ← time + 1
3 d[u] ← time
4 for v in Adj(u)
5   if color[v] = white
6     DFSVisit(v)
7   color[u] ← black
8 time ← time + 1
9 f[u] ← time
  
```



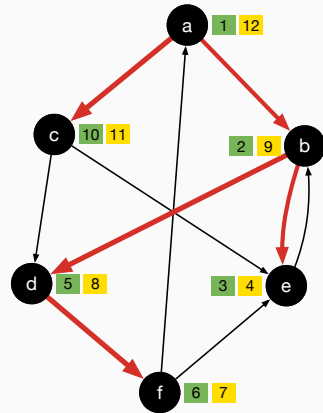
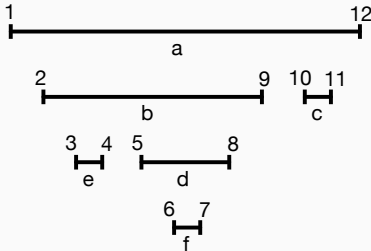


# Tiefensuche: Eigenschaften der Zeitstempel

## Theorem 5.4: Klammerungstheorem

Nach einer Tiefensuche auf einem Graphen  $G = (V, E)$  gilt für alle  $u, v \in V$  mit  $u \neq v$  genau eine der folgenden Bedingungen:

- (a)  $[d(u), f(u)] \cap [d(v), f(v)] = \emptyset$
- (b)  $[d(u), f(u)] \subseteq [d(v), f(v)]$  und  $u$  ist Nachfahre von  $v$  im Tiefensuchbaum.
- (c)  $[d(v), f(v)] \subseteq [d(u), f(u)]$  und  $v$  ist Nachfahre von  $u$  im Tiefensuchbaum.



# Algorithmen und Datenstrukturen

## Elementare Graphalgorithmen

### Tiefensuche: Eigenschaften der Zeitstempel

## Theorem 5.4: Klammerungstheorem

Nach einer Tiefensuche auf einem Graphen  $G = (V, E)$  gilt für alle  $u, v \in V$  mit  $u \neq v$  genau eine der folgenden Bedingungen:

- (a)  $[d(u), f(u)] \cap [d(v), f(v)] = \emptyset$
- (b)  $[d(u), f(u)] \subset [d(v), f(v)]$  und  $u$  ist Nachfahre von  $v$  im Tiefensuchbaum.
- (c)  $[d(v), f(v)] \subset [d(u), f(u)]$  und  $v$  ist Nachfahre von  $u$  im Tiefensuchbaum.



- Das Klammerungstheorem gibt eine einfache Bedingung, um zu überprüfen, ob ein Knoten  $u$  im Tiefensuchbaum ein Nachfahre (nicht unbedingt direkt!) von einem Knoten  $v$  ist.
- Beweisidee des Klammerungstheorems: o. B. d. A.  $d(u) < d(v)$ ; betrachte Zeitpunkt  $d(v)$  und unterscheide die Fälle ob  $u$  zu diesem Zeitpunkt grau oder schwarz ist

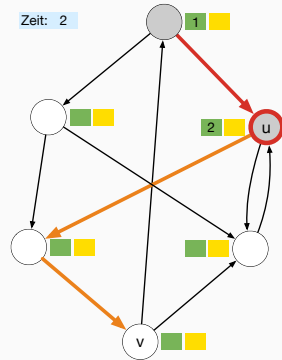
# Tiefensuche: Weiße Pfade

## Theorem 5.5: Theorem vom weißen Pfad

Nach einer Tiefensuche auf einem Graphen  $G = (V, E)$  gilt für alle  $u, v \in V$ :  $v$  ist Nachfahre von  $u$  genau dann wenn  $v$  zum Zeitpunkt  $d(u)$  von  $u$  über einen Pfad weißer Knoten erreicht werden kann.

## Beweisskizze.

- Richtung „ $\Rightarrow$ “:
  - für alle Nachfahren  $w$  von  $u$  gilt  $d(w) > d(u)$  (Theorem 5.4)  $\Rightarrow w$  ist weiß zur Zeit  $d(u)$
  - Anwendung auf **alle** Knoten  $w$  auf dem (eindeutigen) Pfad von  $u$  nach  $v$  im Tiefensuchbaum liefert weißen Pfad
- Richtung „ $\Leftarrow$ “:  $\exists$  weißer Pfad von  $u$  nach  $v$ 
  - Annahme:  $v$  ist kein Nachfahre von  $u$
  - o. B. d. A. sei  $v$  erster nicht-Nachfahre auf weißem Pfad von  $u$  nach  $v$
  - nutze, dass **direkten Vorgänger** von  $v$  auf diesem Pfad Nachfahre ist, ...  
...um zu zeigen, dass  $[d(v), f(v)] \subseteq [d(u), f(u)] \rightsquigarrow$  (Theorem 5.4) ⚡



### Theorem 5.6

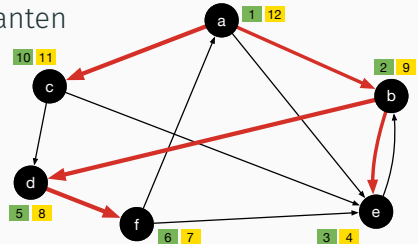
Bei Eingabe eines Graphen  $G = (V, E)$  besitzt Algorithmus DFS Laufzeit  $O(|V| + |E|)$ .

- Beweis erfolgt analog zur Analyse der Breitensuche
- nutze wieder: Gesamtgröße aller Adjazenzlisten ist  $\Theta(|E|)$

## Kantentypen von $G$ anhand des Tiefensuchbaumes

- Baumkanten: Kanten des Tiefensuchbaumes
- Rückwärtskanten: zeigt auf Vorfahre im Tiefensuchbaum
- Vorwärtskanten: zeigt auf Nachfahre im Tiefensuchbaum
- Kreuzungskanten: restliche Kanten

Was sind die Typen  
im Beispiel?



Kantentyp	$d(u) < d(v)$	$f(u) > f(v)$
Baum- & Vorwärtskanten	✓	✓
Rückwärtskanten	✗	✗
Kreuzungskanten	✗	✓

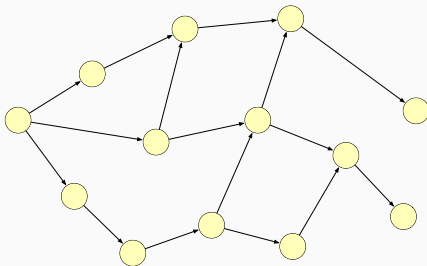
## Lemma 5.4

Für einen gerichteten Graphen  $G = (V, E)$  sind äquivalent:

- (a)  $G$  ist ein DAG (Directed Acyclic Graph)
- (b) DFS erzeugt keine Rückwärtskante
- (c)  $\forall (u, v) \in E: f(u) > f(v)$

## Beweisskizze.

- (b)  $\iff$  (c): aus vorheriger Tabelle
- $\neg(b) \implies \neg(a)$ :
  - sei  $T$  Tiefensuchbaum (Teilgraph von  $G$ )
  - Rückwärtskante erzeugt Kreis in  $T$
- $\neg(a) \implies \neg(b)$ :
  - sei  $(u_1, u_2)$  erste besuchte Kreiskante
  - sei  $(u_1, u_2, \dots, u_k, u_1)$  zugehöriger Kreis
  - DFS besucht alle  $u_1, u_2, \dots, u_k$  und  $(u_k, u_1)$  wird Rückwärtskante  $\square$



# Algorithmen und Datenstrukturen

## Elementare Graphalgorithmen

### Tiefensuche: Anwendung der Klassifikation

- dass  $(u_k, u_1)$  Rückwärtskante wird folgt aus dem Theorem vom weißen Pfad

## Lemma 5.4

Für einen gerichteten Graphen  $G = (V, E)$  sind äquivalent:

- (a)  $G$  ist ein DAG (Directed Acyclic Graph)
- (b) DFS erzeugt keine Rückwärtskante
- (c)  $\forall (u, v) \in E: f(u) > f(v)$

## Beweisskizze.

- $(b) \Rightarrow (c)$  aus vorheriger Tabelle
- $(c) \Rightarrow (a)$
- $(a) \Rightarrow (b)$ 
  - sei  $T$  Tiefensuchbaum (Teilgraph von  $G$ )
  - Rückwärtskante erzeugt Kreis in  $T$
- $(a) \Rightarrow (b)$ 
  - sei  $(u_i, u_j)$  erste besuchte Kreiskante
  - sei  $(u_1, u_2, \dots, u_k, u_1)$  zugehöriger Kreis
  - DFS besucht alle  $u_1, u_2, \dots, u_k$  und  $(u_k, u_1)$  wird Rückwärtskante  $\square$



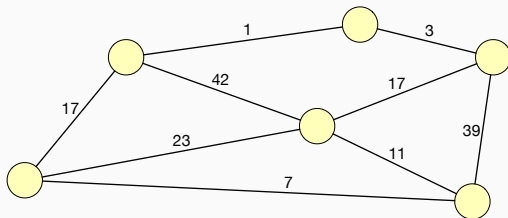
## 2) Minimale Spannbäume

---



## Definition 5.1

Ein gewichteter (ungerichteter) Graph ist ein ungerichteter Graph  $G = (V, E)$  mit einer Gewichtsfunktion  $w: E \rightarrow \mathbb{R}$ .



- sei  $G = (V, E)$  ein gewichteter Graph mit Gewichtsfunktion  $w$
- sei  $H = (V', E')$  Teilgraph von  $G$ 
  - d.h.  $V' \subseteq V$  und  $E' \subseteq E$
- das Gewicht von  $H$  ist definiert als  $w(H) := \sum_{e \in E'} w(e)$

## Algorithmen und Datenstrukturen

## └ Minimale Spannbäume

## └ Kantengewichte

## Definition 5.3

Ein **gewichteter (ungerichteter) Graph** ist ein ungerichteter Graph  $G = (V, E)$  mit einer **Gewichtsfunktion**  $w: E \rightarrow \mathbb{R}$ .



- sei  $G = (V, E)$  ein gewichteter Graph mit Gewichtsfunktion  $w$
- sei  $H = (V', E')$  Teilgraph von  $G$ 
  - d.h.  $V' \subseteq V$  und  $E' \subseteq E$
- das **Gewicht** von  $H$  ist definiert als  $w(H) := \sum_{e \in E'} w(e)$

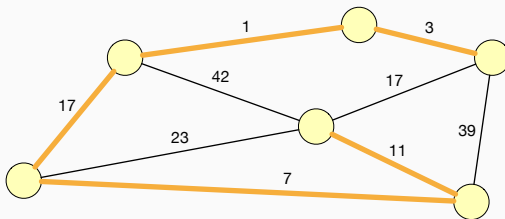
- man kann natürlich auch gewichtete **gerichtete** Graphen anschauen und/oder **Knoten**gewichte betrachten

## Definition 5.2

- Ein Teilgraph  $H$  eines ungerichteten Graphen  $G$  heißt **Spannbaum** von  $G$ , wenn  $H$  ein Baum auf **allen** Knoten von  $G$  ist.
- Ein Spannbaum  $T$  eines gewichteten ungerichteten Graphen  $G$  heißt **minimaler Spannbaum**, wenn  $T$  minimales Gewicht unter allen Spannbäumen von  $G$  hat.

Tree

MST



## Definition 5.2

- Ein Teilgraph  $H$  eines ungerichteten Graphen  $G$  heißt **Spannbaum** von  $G$ , wenn  $H$  ein Baum auf **allen** Knoten von  $G$  ist.
- Ein Spannbaum  $T$  eines gewichteten ungerichteten Graphen  $G$  heißt **minimaler Spannbaum**, wenn  $T$  minimales Gewicht unter allen Spannbäumen von  $G$  hat.



- MST steht für den englischen Begriff Minimal Spanning Tree

# Wie kann man Minimale Spannbäume berechnen?

## Algorithmisches Problem

- Eingabe: ungerichteter, gewichteter Graph  $G = (V, E)$
- Ausgabe: ein minimaler Spannbaum  $T = (V, A)$  von  $G$

## Generisches Verfahren

- beginne mit leerer Kantenmenge  $A = \emptyset \subseteq E$
- erweitere  $A$  sukzessive um eine  $A$ -sichere Kante

### Definition 5.3

Betrachte eine Kantenmenge  $A \subseteq E$  eines gewichteten ungerichteten Graphen  $G = (V, E)$ . Eine Kante  $\{u, v\} \in E$  heißt **A-sicher**, wenn  $A \cup \{\{u, v\}\}$  Teilmenge eines (beliebigen) minimalen Spannbaums von  $G$  ist.

---

## Algorithmus 5.5: GENERIC-MST( $G, w$ )

---

```
1   $A \leftarrow \emptyset$ 
2  while  $A$  is not yet a spanning tree
3      „finde  $A$ -sichere Kante  $\{u, v\}$ “
4       $A \leftarrow A \cup \{\{u, v\}\}$ 
5  return  $A$ 
```

---

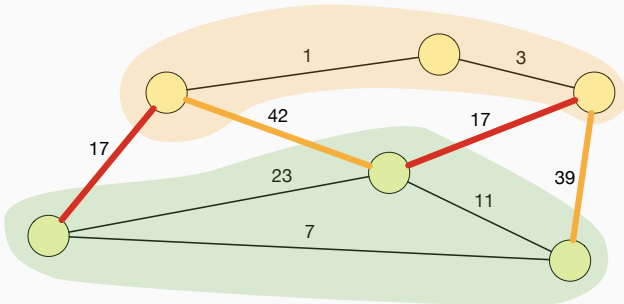
- Korrektheit: ergibt sich aus der Definition von  $A$ -sicheren Kanten
- noch unspezifiziert: Strategie zum Finden von  $A$ -sicheren Kanten!

# Finden von sicheren Kanten über **Schnitte**

## Definition 5.4

Betrachte einen gewichteten ungerichteten Graphen  $G = (V, E)$  und  $A \subseteq E$ .

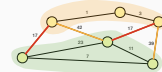
- Ein **Schnitt**  $(C, V \setminus C)$  ist eine Partition der Knotenmenge  $V$  in zwei Teile.
- Eine Kante  $\{u, v\} \in E$  **kreuzt** den Schnitt  $(C, V \setminus C)$ , wenn  $u \in C$  und  $v \in V \setminus C$ .
- Eine Kante  $\{u, v\} \in E$  die den Schnitt  $(C, V \setminus C)$  kreuzt heißt **leicht**, sie eine Kante minimalen Gewichts unter allen  $(C, V \setminus C)$ -kreuzenden Kanten ist.
- Ein Schnitt  $(C, V \setminus C)$  ist **verträglich** mit  $A$ , wenn kein  $e \in A$  ihn kreuzt.



## Definition 5A

Betrachte einen gewichteten ungerichteten Graphen  $G = (V, E)$  und  $A \subseteq E$ .

- Ein Schnitt  $(C, V \setminus C)$  ist eine Partition der Knotenmenge  $V$  in zwei Teile.
- Eine Kante  $\{u, v\} \in E$  **kreuzt** den Schnitt  $(C, V \setminus C)$ , wenn  $u \in C$  und  $v \in V \setminus C$ .
- Eine Kante  $\{u, v\} \in E$  **die** den Schnitt  $(C, V \setminus C)$  **kreuzt** heißt **tauche**, die eine Kante minimalen Gewichts unter allen  $(C, V \setminus C)$ -kreuzenden Kanten ist.
- Ein Schnitt  $(C, V \setminus C)$  ist **verträglich** mit  $A$ , wenn kein  $e \in A$  ihn kreuzt.



- solche Begriffe lassen sich natürlich auch analog für gerichtete Graphen definieren
- Ein Schnitt ist insbesondere verträglich mit der Menge aller Kanten die ihn **nicht** kreuzen (dies ist die maximale Kantenmenge, mit der ein Schnitt verträglich sein kann).



# Charakterisierung sicherer Kanten

## Theorem 5.7

Sei  $G = (V, E)$  ein gewichteter ungerichteter Graph und sei  $A \subseteq E$  in Teil eines MST von  $G$ . Weiter sei  $(C, V \setminus C)$  ein mit  $A$  verträglicher Schnitt und  $e = \{u, v\} \in E$  eine leichte Kante die  $(C, V \setminus C)$  kreuzt. Dann ist  $\{u, v\}$  eine  $A$ -sichere Kante.

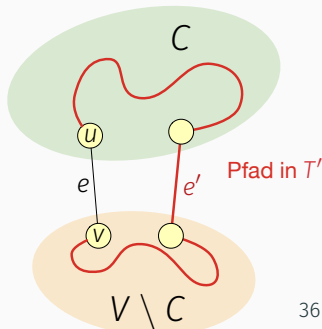
## Beweis.

- sei  $T' = (V, F')$  ein MST mit  $A \subseteq F'$
- $\exists$  eindeutiger Pfad von  $u$  nach  $v$  in  $T'$
- eine Pfadkante  $e'$  muss  $(C, V \setminus C)$  kreuzen
  - es gilt  $w(e) \leq w(e')$  (da  $e$  leicht)
- betrachte  $T = (V, F)$  mit  $F := F' \setminus \{e'\} \cup \{e\}$ 
  - ist Spannbaum mit Gewicht

$$w(T) = w(T') - w(e') + w(e) \leq w(T')$$

$\Rightarrow T = (V, F)$  ist MST mit  $A \cup \{e\} \subseteq F$

□



## Theorem 5.7

Sei  $G = (V, E)$  ein gewichteter ungerichteter Graph und sei  $A \subseteq E$  ein Teil eines MST von  $G$ . Weiter sei  $(C, V \setminus C)$  ein mit  $A$  verträglicher Schnitt und  $e = \{u, v\} \in E$  eine leichte Kante die  $(C, V \setminus C)$  kreuzt. Dann ist  $\{u, v\}$  eine  $A$ -sichere Kante.

## Beweis.

- sei  $T' = (V, F')$  ein MST mit  $A \subseteq F'$
- $\exists$  eindeutiger Pfad von  $u$  nach  $v$  in  $T'$
- eine Pfadkante  $e'$  muss  $(C, V \setminus C)$  kreuzen
- es gilt  $w(e) \leq w(e')$  (da  $e$  leicht)
- betrachte  $T = (V, F)$  mit  $F = F' \setminus \{e'\} \cup \{e\}$
- ist Spannbaum mit Gewicht

$$w(T) = w(T') - w(e') + w(e) \leq w(T')$$

$$\Rightarrow T = (V, F) \text{ ist MST mit } A \cup \{e\} \subseteq F \quad \square$$



- eindeutiger Pfad von  $u$  nach  $v$  in  $T'$  existiert, da  $T'$  ein Baum ist
- $T$  ist Spannbaum, da Konstruktion den Zusammenhang erhält  $T$  genau  $|V| - 1$  Kanten hat

# Was nutzt uns das zur Berechnung eines MST?

## Idee des Algorithmus von Prim

- verfare wie der generische MST-Algorithmus
- seien  $G_A = (V, A)$  und  $s \in V$  (Startknoten)
- finden einer  $A$ -sichere Kante:
  - $C$ : Knoten die in  $G_A$  von  $s$  erreichbar sind
  - wähle leichte Kante, die  $(C, V \setminus C)$  kreuzt
  - diese ist nach Theorem 5.7  $A$ -sicher

## GENERIC-MST( $G, w$ )

```
1   $A \leftarrow \emptyset$ 
2  while  $A$  is not MST
3      „finde  $A$ -sichere Kante  $e$ “
4       $A \leftarrow A \cup \{e\}$ 
5  return  $A$ 
```

## Algorithmus 5.6: PRIM( $G, w, s$ )

```
1  for  $u \in V$ 
2       $\text{key}(u) \leftarrow \infty; \pi(u) \leftarrow \text{NIL}$ 
3   $\text{key}(s) \leftarrow 0$ 
4   $Q \leftarrow \text{BUILDHEAP}(V)$ 
5  while  $Q \neq \emptyset$ 
6       $u \leftarrow \text{DELETEMIN}(Q)$ 
7      for  $v \in \text{Adj}(u)$ 
8          if  $v \in Q$  and  $\text{key}(v) > w(u, v)$ 
9               $\text{DECREASEKEY}(Q, v, w(u, v))$ 
10          $\pi(v) \leftarrow u$ 
```

- speichere noch nicht mit  $s$  verbundene Knoten in min-Heap  $Q$ 
  - Schlüssel  $\text{key}(v)$ : Gewicht günstigster Kante, die  $v$  mit  $s$  in  $G_A$  verbindet
- $\pi(v)$ : Vorgänger von  $v$  im MST
- $A$  implizit durch  $\pi$  gegeben
  - $A = \{ \{ \pi(v), v \} \mid v \in V \setminus \{s\} \}$
- DECREASEKEY: verringert Schlüssel und stellt Heap-Eigenschaft wieder her

## Algorithmen und Datenstrukturen

## Minimale Spannbäume

Was nutzt uns das zur Berechnung eines MST?

- **DECREASEKEY** hatten wir noch nicht, aber einfach zu implementieren  $\rightsquigarrow$  gute Übung!

Was nutzt uns das zur Berechnung eines MST?

## Idee des Algorithmus von Prim

- verfähre wie der generische MST-Algorithmus
- seien  $G_s = (V, A)$  und  $s \in V$  (Startknoten)
- finden einer  $A$ -sichere Kante:
  - $C$  Knoten die in  $G_s$  von  $s$  erreichbar sind
  - wähle leichte Kante, die  $(C, V \setminus C)$  kreuzt
  - diese ist nach Theorem 5.7  $A$ -sicher

```

Generiere-MST( $G, w$ )
1   $A \leftarrow \emptyset$ 
2  while  $A$  is not MST
3    find the  $A$ -sichere Kante  $e$ 
4     $A \leftarrow A \cup \{e\}$ 
5  return  $A$ 

```

Algorithmus 5.6: Prim( $G, w, s$ )

```

1  for  $u \in V$ 
2     $key[u] \leftarrow \infty$ ;  $u[s] \leftarrow NIL$ 
3   $key[s] \leftarrow 0$ 
4   $Q \leftarrow Insertion(s)$ 
5  while  $Q \neq \emptyset$ 
6     $v \leftarrow ExtractMin(Q)$ 
7    for  $u \in Adj(v)$ 
8      if  $u \in Q$  and  $key[u] > w(u, v)$ 
9        DecreaseKey( $Q, u, w(u, v)$ )
10      $u[v] \leftarrow u$ 

```

- speichere noch nicht mit  $s$  verbundene Knoten in **min-Heap**  $Q$   
 - Schlüssel  $key(v)$ : Gewicht günstigster Kante, die  $v$  mit  $s$  in  $G_s$  verbindet  
 -  $u[v]$ : Vorgänger von  $v$  im MST  
 -  $A$  implizit durch  $\rightarrow$  gegeben  
 -  $A = \{ \{u[v], v\} \mid v \in V \setminus \{s\} \}$   
 - **DECREASEKEY**: verringert Schlüssel und stellt Heap-Eigenschaft wieder her

# Was nutzt uns das zur Berechnung eines MST?

## Idee des Algorithmus von Prim

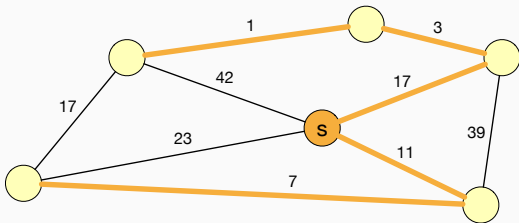
- verfare wie der generische MST-Algorithmus
- seien  $G_A = (V, A)$  und  $s \in V$  (Startknoten)
- finden einer  $A$ -sichere Kante:
  - $C$ : Knoten die in  $G_A$  von  $s$  erreichbar sind
  - wähle leichte Kante, die  $(C, V \setminus C)$  kreuzt
  - diese ist nach Theorem 5.7  $A$ -sicher

## GENERIC-MST( $G, w$ )

```
1   $A \leftarrow \emptyset$ 
2  while  $A$  is not MST
3      „finde  $A$ -sichere Kante  $e$ “
4       $A \leftarrow A \cup \{e\}$ 
5  return  $A$ 
```

## Algorithmus 5.6: PRIM( $G, w, s$ )

```
1  for  $u \in V$ 
2       $\text{key}(u) \leftarrow \infty; \pi(u) \leftarrow \text{NIL}$ 
3   $\text{key}(s) \leftarrow 0$ 
4   $Q \leftarrow \text{BUILDHEAP}(V)$ 
5  while  $Q \neq \emptyset$ 
6       $u \leftarrow \text{DELETEMIN}(Q)$ 
7      for  $v \in \text{Adj}(u)$ 
8          if  $v \in Q$  and  $\text{key}(v) > w(u, v)$ 
9               $\text{DECREASEKEY}(Q, v, w(u, v))$ 
10          $\pi(v) \leftarrow u$ 
```



└ Was nutzt uns das zur Berechnung eines MST?

- **DECREASEKEY** hatten wir noch nicht, aber einfach zu implementieren  $\rightsquigarrow$  gute Übung!

Was nutzt uns das zur Berechnung eines MST?

Idee des Algorithmus von Prim

- verfähre wie der generische MST-Algorithmus
- seien  $G_s = (V, A)$  und  $s \in V$  (Startknoten)
- finden einer  $A$ -sichere Kante:
  - $C$  Knoten die in  $G_s$  von  $s$  erreichbar sind
  - wähle leichte Kante, die  $(C, V \setminus C)$  kreuzt
  - diese ist nach Theorem 5.7  $A$ -sicher

Generiere-MST( $G, w$ )

```

1  A ← ∅
2  while A is not MST
3    find the A-sichere Kante e
4    A ← A ∪ {e}
5  return A

```

Algorithmus 5.6: Prim( $G, w, s$ )

```

1  for  $u \in V$ 
2    key[u] ← ∞;  $u[s] \leftarrow \text{NIL}$ 
3  key[s] ← 0
4  Q ← Successeurs(s)
5  while Q ≠ ∅
6     $v \leftarrow \text{SelectMin}(Q)$ 
7  for  $v \in \text{Adj}(v)$ 
8    if  $v \in Q$  and  $\text{key}[v] > w(u, v)$ 
9      DecreaseKey(Q, v,  $w(u, v)$ )
10   $u[v] \leftarrow u$ 

```



# Wie gut ist der Algorithmus von Prim?

## Laufzeit

- Zeilen 1 bis 4:  $\Theta(|V|)$
- Zeile 6:  $\Theta(|V|)$  Aufrufe, je mit LZ  $O(\log|V|)$ 
  - Anfangs  $|V|$  Knoten in Heap, danach keine weiteren Einfügungen
  - pro Durchlauf wird ein Knoten entfernt  $\rightsquigarrow \Theta(|V|)$  Aufrufe
- Zeilen 8 bis 10:
  - $2 \cdot |E|$  Durchläufe (Gesamtgröße aller Adjazenzlisten)
  - pro Durchlauf LZ  $O(\log|V|)$  (für `DECREASEKEY`)

```
1  for  $u \in V$ 
2       $\text{key}(u) \leftarrow \infty; \pi(u) \leftarrow \text{NIL}$ 
3   $\text{key}(s) \leftarrow 0$ 
4   $Q \leftarrow \text{BUILDHEAP}(V)$ 
5  while  $Q \neq \emptyset$ 
6       $u \leftarrow \text{DELETEMIN}(Q)$ 
7      for  $v \in \text{Adj}(u)$ 
8          if  $v \in Q$  and  $\text{key}(v) > w(u, v)$ 
9               $\text{DECREASEKEY}(Q, v, w(u, v))$ 
10          $\pi(v) \leftarrow u$ 
```

## Korrektheit

- folgt aus Korrektheit von `GENERIC-MST...`
- ...und aus [Theorem 5.7](#)
- (+Korrektheit der Heap-Operationen)

### Theorem 5.8

`PRIM` berechnet einen MST eines gewichteten ungerichteten Graphen  $G = (V, E)$  in Zeit  $O(|V| \cdot \log|V| + |E| \cdot \log|V|)$ .

## Algorithmen und Datenstrukturen

## └ Minimale Spannbäume

└ Wie gut ist der Algorithmus von Prim?

Wie gut ist der Algorithmus von Prim?

Laufzeit

- Zeilen 1 bis 4:  $\Theta(|V|)$
- Zeilen 5 bis 7:  $\Theta(|V|)$  Aufrufe, je mit LZ  $O(\log |V|)$ 
  - Anfangs  $|V|$  Knoten in Heap, danach keine weiteren Einfügungen
  - pro Durchlauf wird ein Knoten entfernt  $\rightarrow \Theta(|V|)$  Aufrufe
- Zeilen 8 bis 10:
  - $2 \cdot |E|$  Durchläufe (Gesamtgröße aller Adjazenzlisten)
  - pro Durchlauf LZ  $O(\log |V|)$  (für DECREASEKEY)

```

1 for u ← v
2   key(u) ← ∞; w(u) ← NIL
3 key(v) ← 0
4 Q ← Insertion(v)
5 while Q ≠ ∅
6   u ← DeleteMin(Q)
7   for v ← Adj(u)
8     if v ∈ Q and key(v) > w(u, v)
9       DecreaseKey(Q, v, w(u, v))
10  w(u) ← u

```

Korrektheit

- folgt aus Korrektheit von GENERIC-MST
- und aus Theorem 5.7
- (=Korrektheit der Heap-Operationen)

Theorem 5.8

**Prim** berechnet einen MST eines gewichteten ungerichteten Graphen  $G = (V, E)$  in Zeit  $O(|V| \cdot \log |V| + |E| \cdot \log |V|)$ .

- Mit **Fibonacci-Heaps** kann die Laufzeit auf  $O(|V| \cdot \log |V| + |E|)$  verbessert werden (wegen Amortisierung).



# Alternative MST-Berechnung: Algorithmus von Kruskal

## Idee des Algorithmus von Kruskal

- verfare wie der generische MST-Algorithmus
- sei wieder  $G_A = (V, A)$
- finden einer  $A$ -sichere Kante:
  - wähle leichteste Kante die keinen Kreis erzeugt
  - diese ist nach Theorem 5.7  $A$ -sicher

---

GENERIC-MST( $G, w$ )

---

```
1   $A \leftarrow \emptyset$ 
2  while  $A$  is not MST
3      „finde  $A$ -sichere Kante  $e$ “
4       $A \leftarrow A \cup \{e\}$ 
5  return  $A$ 
```

---

## Algorithmen und Datenstrukturen

## └ Minimale Spannbäume

└ Alternative MST-Berechnung: Algorithmus von  
Kruskal

- Korrektheit folgt wieder aus Theorem 5.7

## Idee des Algorithmus von Kruskal

- verfähre wie der generische MST-Algorithmus
- sei wieder  $G_n = (V, A)$
- finden einer  $A$ -sichere Kante:
  - wähle leichteste Kante die keinen Kreis erzeugt
  - diese ist nach Theorem 5.7  $A$ -sicher

---

 Generico-MST( $G, w$ )
 

---

```

1   $A \leftarrow \emptyset$ 
2  while  $A$  is not MST
3    finde  $A$ -sichere Kante  $e$ 
4     $A \leftarrow A \cup \{e\}$ 
5  return  $A$ 
  
```

---

# Alternative MST-Berechnung: Algorithmus von Kruskal

## Idee des Algorithmus von Kruskal

- verfare wie der generische MST-Algorithmus
- sei wieder  $G_A = (V, A)$
- finden einer  $A$ -sichere Kante:
  - wähle leichteste Kante die keinen Kreis erzeugt
  - diese ist nach Theorem 5.7  $A$ -sicher

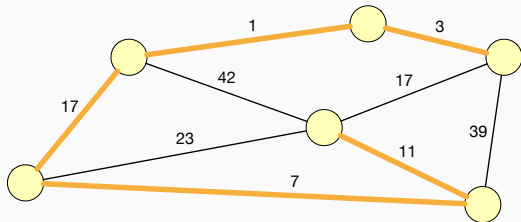
---

GENERIC-MST( $G, w$ )

---

```
1   $A \leftarrow \emptyset$ 
2  while  $A$  is not MST
3    „finde  $A$ -sichere Kante  $e$ “
4     $A \leftarrow A \cup \{e\}$ 
5  return  $A$ 
```

---



## Idee des Algorithmus von Kruskal

- verfähre wie der generische MST-Algorithmus
- sei wieder  $G_n = (V, A)$
- finden einer  $A$ -sichere Kante:
  - wähle leichteste Kante die keinen Kreis erzeugt
  - diese ist nach Theorem 5.7  $A$ -sicher

Generico-MST( $G, w$ )

```


1   $A \leftarrow \emptyset$ 
2  while  $A$  is not MST
3    finde  $A$ -sichere Kante  $e$ 
4     $A \leftarrow A \cup \{e\}$ 
5  return  $A$ 
```

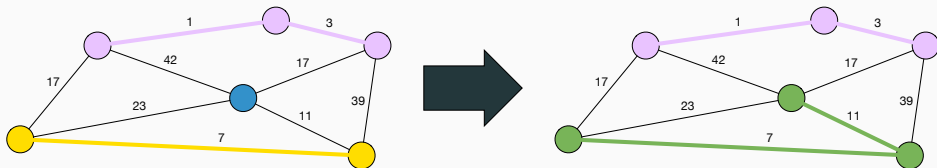


- Korrektheit folgt wieder aus Theorem 5.7

# Verschmelzen von Zusammenhangskomponenten

## Alternative Sichtweise

- $G_A = (V, A)$  ist ein **Wald** 
  - besteht aus mehreren, **Zusammenhangskomponenten** (ZHKen)
  - jede ZHK ist ein Baum
  - initial (wenn  $A = \emptyset$ ) sind es  $|V|$  ZHKen (die einzelnen Knoten)
- wähle günstigste Kante, die zwei Bäume des Waldes verschmilzt



## Alternative Sichtweise

- $G_A = (V, A)$  ist ein Wald 
- besteht aus mehreren, **Zusammenhangskomponenten** (ZHKen)
- jede ZHK ist ein Baum
- initial (wenn  $A = \emptyset$ ) sind es  $|V|$  ZHKen (die einzelnen Knoten)
- wähle günstigste Kante, die zwei Bäume des Waldes verschmilzt



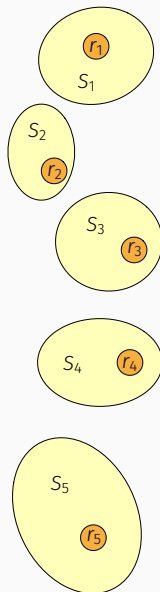
- ZHK eines Graphen  $G$ : ein maximal zusammenhängender Teilgraph von  $G$

## Effiziente Implementierung von Kruskal muss...

- ...für  $\{u, v\} \in E$  effizient entscheiden, ob  $u$  und  $v$  in gleicher ZHK von  $G_A$  liegen
- ...zwei ZHKen effizient verschmelzen können

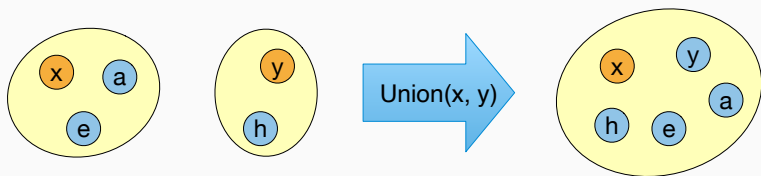
### Disjunkte dynamische Mengen

- gegeben: Grundmenge  $U$  von Objekten (Universum)
- Ziel: verwalte Familie  $\{S_1, S_2, \dots, S_k\}$  disjunkter Teilmengen  $S_i \subseteq U$
- Teilmengen  $S_i$  identifiziert durch ein Element  $r_i$
- $r_i \in S_i$  (der **Repräsentant**)



# Operationen für disjunkte dynamische Mengen

- MAKESET( $x$ ): erzeuge Teilmenge  $\{x\}$  mit Repräsentant  $x$
- UNION( $x, y$ ): vereinigt die beiden Teilmengen, die  $x$  bzw.  $y$  enthalten



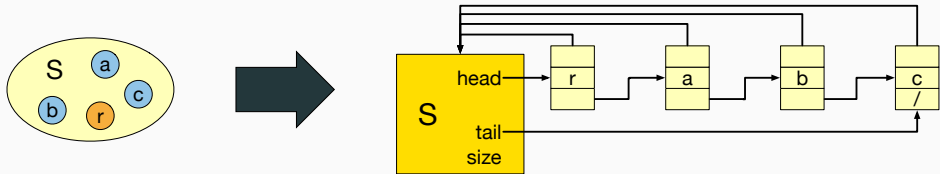
- FINDSET( $x$ ): liefert Repräsentanten der Menge, die  $x$  enthält

Man nennt eine solche Datenstruktur  
auch **Union-Find** Datenstruktur.



# Disjunkte dynamische Mengen mittels verketteter Listen

- je Teilmenge  $S$  eine verkettete Liste der Objekte in  $S$
- Repräsentant ist erstes Objekt der Liste
- jedes Objekt hat Verweis auf Mengenobjekt  $S$
- Listenobjekt speichert Kopf, Ende, und Größe der Liste



- Implementierung von **MAKESET** und **FINDSET** einfach LZ  $\Theta(1)$ 
  - erzeuge entsprechende Struktur bzw. Folge Zeigern zu Repräsentant  $r$
- Implementierung von **UNION**( $x, y$ )
  - bestimme kürzere der beiden Listen
  - hänge kürzere Liste an längere Liste und...
  - ...aktualisiere Mengenobjekt-Zeiger aller Elemente der kürzeren Liste

↪ Laufzeit  $\Theta(\text{Länge der kürzeren Liste})$

## Algorithmen und Datenstrukturen

## └ Minimale Spannbäume



## Disjunkte dynamische Mengen mittels

## Disjunkte dynamische Mengen mittels verketteter Listen

- je Teilmenge  $S$  eine verkettete Liste der Objekte in  $S$
- Repräsentant ist erstes Objekt der Liste
- jedes Objekt hat Verweis auf Mengenobjekt  $S$
- Listenobjekt speichert Kopf, Ende, und Größe der Liste



- Implementierung von `MakeSet` und `FindSet` einfach
    - erzeuge entsprechende Struktur bzw. Folge Zeigern zu Repräsentant
  - Implementierung von `UNION(x, y)`
    - bestimme kürzere der beiden Listen
    - hänge kürzere Liste an längere Liste und...
    - ...aktualisiere Mengenobjekt-Zeiger aller Elemente der kürzeren Liste
- Laufzeit  $\Theta(\text{Länge der kürzeren Liste})$

- andere, effizientere Implementierungen existieren, z. B. mittels Bäumen statt Listen

# Laufzeit **vieler** Union-Find Operationen

## Theorem 5.9

Betrachte  $m$  Operationen auf der Union-Find DS mit verketteten Listen (**MAKESET**, **FINDSET**, **UNION**, und **UNION**). Sei  $n$  die Anzahl der **UNION** Operationen. Die Gesamtlaufzeit aller Operationen ist  $O(m + n \log n)$ .

## Beweisskizze.

- alle **MAKESET** und **FINDSET** Operationen benötigen zusammen LZ  $\Theta(m)$
- betrachte LZ **aller** **UNION** Aufrufe
  - LZ proportional zur Anzahl der Änderungen der Links auf Mengenobjekte
  - Frage: Wie oft wird Link eines Knoten auf sein Mengenobjekt geändert?
    - beim ersten Update: neue Liste hat Länge  $\geq 2$
    - beim zweiten Update: neue Liste hat Länge  $\geq 4$
    - ↪ beim  $k$ -ten Update: neue Liste hat Länge  $\geq 2^k$
    - da Listenlänge  $\leq n$  sein muss, folgt  $k \leq \log n$
  - ↪ jedes der  $n$  Objekte hat  $O(\log n)$  Updates seines Mengenobjektes

---

## Algorithmus 5.7: KRUSKAL( $G, w$ )

---

```
1   $A \leftarrow \emptyset$ 
2  for  $u \in V$ 
3      MAKESET( $u$ )
4  sortiere Kantenmenge  $E$  nach aufsteigendem Gewicht
5  for  $\{u, v\} \in E$  in aufsteigender Reihenfolge
6      if FINDSET( $u$ )  $\neq$  FINDSET( $v$ )
7           $A \leftarrow A \cup \{\{u, v\}\}$ 
8          UNION( $u, v$ )
```

---

### Theorem 5.10

KRUSKAL berechnet einen MST eines gewichteten ungerichteten Graphen  $G = (V, E)$  in Zeit  $O(|V| \cdot \log|V| + |E| \cdot \log|E|)$ .

## Algorithmen und Datenstrukturen

## └ Minimale Spannbäume



## Algorithmus von Kruskal mit Union-Find

Algorithmus 5.2: KRUSKAL( $G, w$ )

```

1  $A \leftarrow \emptyset$ 
2 for  $u \in V$ 
3   MAKESET( $u$ )
4 sortiere Kantenmenge  $E$  nach aufsteigendem Gewicht
5 for  $\{u, v\} \in E$  in aufsteigender Reihenfolge
6   if FINDSET( $u$ )  $\neq$  FINDSET( $v$ )
7      $A \leftarrow A \cup \{\{u, v\}\}$ 
8   UNION( $u, v$ )

```

## Theorem 5.10

KRUSKAL berechnet einen MST eines gewichteten ungerichteten Graphen  $G = (V, E)$  in Zeit  $O(|V| \cdot \log|V| + |E| \cdot \log|E|)$ .

- bei vorsortierten Kanten (oder linearer Sortierung, z.B. über **RADIXSORT**) ist eine Implementierung mit Laufzeit **fast**  $O(|E|)$  möglich
- genauer gesagt, beträgt die Laufzeit dann  $O(\alpha(|V|) \cdot |E|)$ , wobei  $\alpha$  die Inverse der **Ackermann Funktion** ist (für realistische Eingaben gilt  $\alpha(|V|) \leq 5$ )
- siehe auch [Wikipedia](#)

### 3) Kürzeste Pfade

---

# Was ist ein kürzester Pfad?

- wir betrachten nun gewichtete **gerichtete** Graphen  $G = (V, E)$
- d. h. Gewichtsfunktion  $w$  hat die Form  $w: E \rightarrow \mathbb{R}$  mit  $E \subseteq V \times V$
- für einen Pfad  $p = (u_0, u_1, \dots, u_k)$  definieren wir sein **Gewicht**

$$w(p) := \sum_{i=1}^k w(u_{i-1}, u_i)$$

## Algorithmisches Problem

- Eingabe:
  - gerichteter, gewichteter Graph  $G = (V, E)$
  - Startknoten  $u \in V$
  - Zielknoten  $v \in V$
- Ausgabe: Pfad  $p$  mit minimalem Gewicht von  $u$  nach  $v$

## Algorithmen und Datenstrukturen

## └ Kürzeste Pfade

## └ Was ist ein kürzester Pfad?

## Was ist ein kürzester Pfad?

- wir betrachten nun gewichtete **gerichtete** Graphen  $G = (V, E)$
- d. h. Gewichtsfunktion  $w$  hat die Form  $w: E \rightarrow \mathbb{R}$  mit  $E \subseteq V \times V$
- für einen Pfad  $p = (u_0, u_1, \dots, u_k)$  definieren wir sein **Gewicht**

$$w(p) := \sum_{i=1}^k w(u_{i-1}, u_i)$$

## Algorithmisches Problem

- Eingabe:
  - gerichteter, gewichteter Graph  $G = (V, E)$
  - Startknoten  $u \in V$
  - Zielknoten  $v \in V$
- Ausgabe: Pfad  $p$  mit minimalem Gewicht von  $u$  nach  $v$

- dieses Problem wird auch als „Single-pair Shortest Path“ (SPSP) Problem bezeichnet



# Single-Source Shortest Path (SSSP) Problem

Wir starten gleich etwas allgemeiner!

## Algorithmisches Problem

- Eingabe:
  - gerichteter, gewichteter Graph  $G = (V, E)$
  - Startknoten  $s \in V$
- Ausgabe:
  - für alle  $v \in V$  die Distanz  $\text{dist}(s, v)$  sowie...
  - ...ein entsprechender kürzester Pfad

- etwas später: All-pairs Shortest Path Problem (APSP)
  - Distanzen und kürzeste Pfade für **alle** Paare von Knoten

## └ Single-Source Shortest Path (SSSP) Problem

Wir starten gleich etwas allgemeiner!

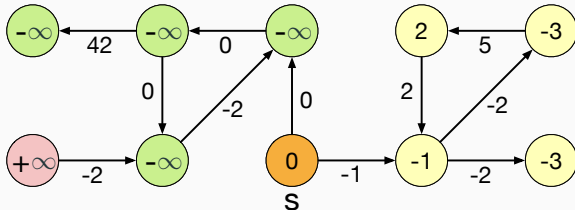
**Algorithmisches Problem**

- Eingabe:
  - gerichteter, gewichteter Graph  $G = (V, E)$
  - Startknoten  $s \in V$
- Ausgabe:
  - für alle  $v \in V$  die Distanz  $d_{\text{sssp}}(s, v)$  sowie...
  - ...ein entsprechender kürzester Pfad

- etwas später: All-pairs Shortest Path Problem (APSP)
  - Distanzen und kürzeste Pfade für **alle** Paare von Knoten

- interessanterweise ist kein Algorithmus bekannt, der für SPSP asymptotisch schneller läuft als der beste Algorithmus für SSSP

## Kürzeste Pfade und negative Kosten



Distanzen  $\text{dist}(s, u)$  Fallen in drei Kategorien:

- $\text{dist}(s, u) = +\infty$ : existiert **kein** Pfad von  $s$  nach  $u$
- $\text{dist}(s, u) = -\infty$ : existiert ein Pfad **beliebig kleiner Kosten** von  $s$  nach  $u$
- $\text{dist}(s, u) = x \in \mathbb{R}$ : kürzester Pfad von  $s$  nach  $u$  ist endlich und hat Kosten  $x$

## Einfache Beobachtung:

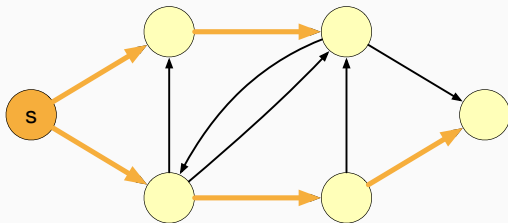
Folgende Eigenschaften sind äquivalent:

- $\text{dist}(s, u) = -\infty$
- es existiert ein Pfad von  $s$  nach  $u$  mit einem negativen Kreis

# Kürzeste Pfade unter **positiven** Kantengewichten

Hatten wir das nicht schon?

- BFS berechnet kürzeste Pfade...
- ...aber nur, wenn alle Kantengewichte **gleich** sind!



## Algorithmen und Datenstrukturen

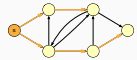
## └ Kürzeste Pfade

└ Kürzeste Pfade unter **positiven**  
Kantengewichten

- Problem von MST (1/2): ignoriert bisherigen (im letzten Bild **orange**) Pfad, vergleicht also nur **2** mit **7**
- Problem von MST (2/2): stattdessen muss der orange Pfad der Länge **6** mit berücksichtigt werden, da  $6 + 2 > 7$

Hatten wir das nicht schon?

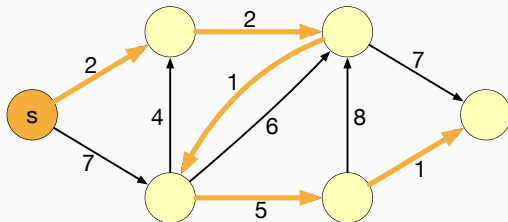
- BFS berechnet kürzeste Pfade...
- ...aber nur, wenn alle Kantengewichte **gleich** sind!



# Kürzeste Pfade unter **positiven** Kantengewichten

Hatten wir das nicht schon?

- BFS berechnet kürzeste Pfade...
- ...aber nur, wenn alle Kantengewichte **gleich** sind!

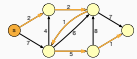


## Idee 1: MST

- wie BFS, aber beachte Kantengewicht
- genauer: wähle kürzeste Kante, die zu unentdecktem Knoten führt

Hatten wir das nicht schon?

- BFS berechnet kürzeste Pfade...
- ...aber nur, wenn alle Kantengewichte **gleich** sind!



Idee 1: MST

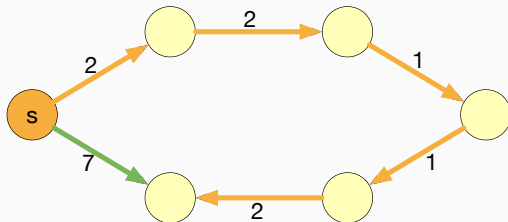
- wie BFS, aber beachte Kantengewicht
- genauer: wähle kürzeste Kante, die zu unentdecktem Knoten führt

- Problem von MST (1/2): ignoriert bisherigen (im letzten Bild **orange**) Pfad, vergleicht also nur **2** mit **7**
- Problem von MST (2/2): stattdessen muss der orange Pfad der Länge **6** mit berücksichtigt werden, da  $6 + 2 > 7$

# Kürzeste Pfade unter **positiven** Kantengewichten

Hatten wir das nicht schon?

- BFS berechnet kürzeste Pfade...
- ...aber nur, wenn alle Kantengewichte **gleich** sind!



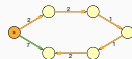
Idee 1: MST **✗**

- wie BFS, aber beachte Kantengewicht
- genauer: wähle kürzeste Kante, die zu unentdecktem Knoten führt



Hatten wir das nicht schon?

- BFS berechnet kürzeste Pfade...
- ...aber nur, wenn alle Kantengewichte **gleich** sind!

Idee 1: MST **✗**

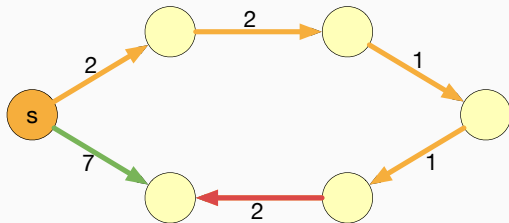
- wie BFS, aber beachte Kantengewicht
- genauer: wähle kürzeste Kante, die zu unentdecktem Knoten führt

- Problem von MST (1/2): ignoriert bisherigen (im letzten Bild **orange**) Pfad, vergleicht also nur **2** mit **7**
- Problem von MST (2/2): stattdessen muss der orange Pfad der Länge **6** mit berücksichtigt werden, da  $6 + 2 > 7$

# Kürzeste Pfade unter **positiven** Kantengewichten

Hatten wir das nicht schon?

- BFS berechnet kürzeste Pfade...
- ...aber nur, wenn alle Kantengewichte **gleich** sind!

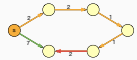


## Idee 2

- wie MST (**PRIM**), aber...
- ...beachte **bisherigen Pfad**!

Hatten wir das nicht schon?

- BFS berechnet kürzeste Pfade...
- ...aber nur, wenn alle Kantengewichte **gleich** sind!



Idee 2

- wie MST (**Pseud**), aber...
- ...beachte **bisherigen Pfad**!

- Problem von MST (1/2): ignoriert bisherigen (im letzten Bild **orange**) Pfad, vergleicht also nur **2** mit **7**
- Problem von MST (2/2): stattdessen muss der orange Pfad der Länge **6** mit berücksichtigt werden, da  $6 + 2 > 7$

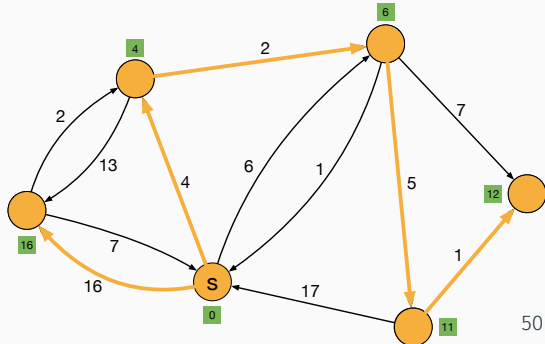
# Umsetzung der Idee

## Vorstufe des Pseudocodes

- 1 sei  $S$  Menge der bereits entdeckten Knoten
- 2 speichere für alle  $u \in S$   $\text{key}(u) = \text{dist}(s, u)$
- 3 initial:  $S = \{s\}$  und  $\text{key}(s) = 0$
- 4 solange  $S \neq V$ :
- 5     finde Knoten  $u \in S$  und  $v \in V \setminus S$  für die...
- 6     ...  $\text{key}(u) + w(u, v)$  minimal ist
- 7     füge  $v$  zu  $S$  hinzu und setze  $\text{key}(v) \leftarrow \text{key}(u) + w(u, v)$

## Kürzeste Pfade

- implizite Speicherung
- durch Zeiger  $\pi(v) \leftarrow u$
- kürzester Pfad von  $s$  zu  $v$ :  
kürzester Pfad zu  $\pi(v)$ ...  
...gefolgt von Kante  $(\pi(v), v)$



## Vorstufe des Pseudocodes

```

1  sei:  $S$  Menge der bereits entdeckten Knoten
2  speichern für alle  $v \in S$   $key(v) \leftarrow dist(s, v)$ 
3  initial:  $S \leftarrow \{s\}$  und  $key(s) \leftarrow 0$ 
4  solange  $S \neq V$ 
5  ...
6  ...  $key(v) \leftarrow w(s, v)$  minimal ist
7  füge  $v$  zu  $S$  hinzu und setze  $key(v) \leftarrow key(s) + w(s, v)$ 

```

## Kürzeste Pfade

- implizite Speicherung
- durch Zeiger  $\pi(v) \leftarrow u$
- kürzester Pfad von  $s$  zu  $u$
- kürzester Pfad zu  $\pi(v)$ , ... gefolgt von Kante  $(\pi(v), v)$



- kürzeste Pfade sind also genauso gespeichert, wie z. B. bei BFS

# Effiziente Implementierung: Dijkstras Algorithmus

---

PRIM( $G, w, s$ )

---

```
1  for  $u \in V$ 
2     $\text{key}(u) \leftarrow \infty; \pi(u) \leftarrow \text{NIL}$ 
3   $\text{key}(s) \leftarrow 0$ 
4   $Q \leftarrow \text{BUILDHEAP}(V)$ 
5  while  $Q \neq \emptyset$ 
6     $u \leftarrow \text{DELETEMIN}(Q)$ 
7    for  $v \in \text{Adj}(u)$ 
8      if  $v \in Q$  and  $\text{key}(v) > w(u, v)$ 
9         $\text{DECREASEKEY}(Q, v, w(u, v))$ 
10      $\pi(v) \leftarrow u$ 
```

---

VS

---

Algorithmus 5.8: DIJKSTRA( $G, w, s$ )

---

```
1  for  $u \in V$ 
2     $\text{key}(u) \leftarrow \infty; \pi(u) \leftarrow \text{NIL}$ 
3   $\text{key}(s) \leftarrow 0$ 
4   $Q \leftarrow \text{BUILDHEAP}(V)$ 
5  while  $Q \neq \emptyset$ 
6     $u \leftarrow \text{DELETEMIN}(Q)$ 
7    for  $v \in \text{Adj}(u)$ 
8      if  $\text{key}(v) > \text{key}(u) + w(u, v)$ 
9         $\text{DECREASEKEY}(Q, v, \text{key}(u) + w(u, v))$ 
10      $\pi(v) \leftarrow u$ 
```

---

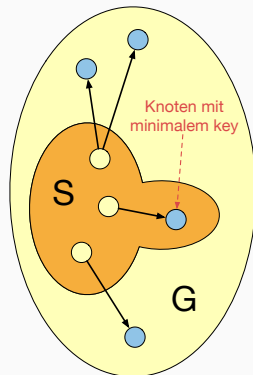
Das kennen wir doch irgendwoher?!?



# Wie gut ist Dijkstras Algorithmus

## Theorem 5.11

**DIJKSTRA** löst SSSP für einen gewichteten gerichteten Graphen  $G = (V, E)$  mit nicht-negativen Kantengewichten in Zeit  $O(|V| \cdot \log|V| + |E| \cdot \log|V|)$ .



## Analyseskizze

- Laufzeit folgt analog zur Analyse von **PRIM**
- für Korrektheit sei  $\Gamma(S) := \{v \in V \setminus S \mid \exists u \in S: (u, v) \in E\}$
- nutze folgende Invariante:
  - $\forall u \in S: \text{key}(u) = \text{dist}(s, u)$  und  $\text{key}(u) \leq \min \{ \text{dist}(s, v) \mid v \in \Gamma(S) \}$
  - $\forall v \in \Gamma(S): \text{key}(v) \geq \text{dist}(s, v)$

## └ Wie gut ist Dijkstras Algorithmus

## Theorem 5.11

Dijkstra löst SSSP für einen gewichteten gerichteten Graphen  $G = (V, E)$  mit nicht negativen Kanten-gewichten in Zeit  $O(|V| \cdot \log|V| + |E| \cdot \log|V|)$ .



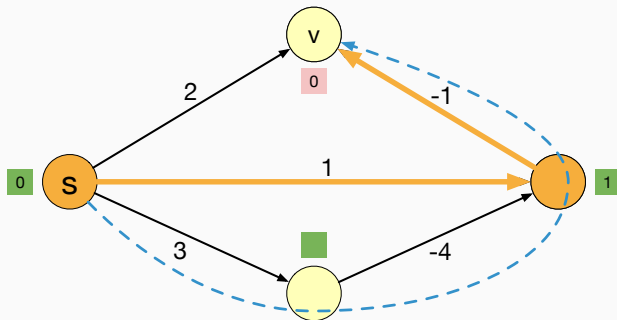
## Analyseskizze

- Laufzeit folgt analog zur Analyse von Prim
- für Korrektheit sei  $\Gamma(S) := \{v \in V \setminus S \mid \exists u \in S: (u, v) \in E\}$
- nutze folgende Invariante:
  - $\forall u \in S: \text{key}(u) = \text{dist}(s, u)$  und  $\text{key}(u) \leq \min \{ \text{dist}(s, v) \mid v \in \Gamma(S) \}$
  - $\forall v \in \Gamma(S): \text{key}(v) \geq \text{dist}(s, v)$

- Mit **Fibonacci-Heaps** kann die Laufzeit auf  $O(|V| \cdot \log|V| + |E|)$  verbessert werden (wegen Amortisierung).



## Wo liegt das Problem bei **negativen Gewichten**?



- Knoten **v** bekommt falschen Distanzwert zugewiesen!
- Varianten und alternative Verfahren existieren...
- ...z. B. unser nächster Algorithmus für APSP!

## 4) Paarweise kürzeste Pfade

---

# Kürzeste Pfade für **alle** Knotenpaare

Haben gesehen:

- SSSP: (Single-Source Shortest Path)
  - kürzeste Pfade von **einem** Knoten zu **allen** anderen Knoten
- für nicht-negative Kantengewichte (Dijkstra)

## Nun: All-pairs Shortest Path (APSP)

- Eingabe: gerichteter, gewichteter Graph  $G = (V, E)$
- Ausgabe:
  - für alle  $u, v \in V$  die Distanz  $\text{dist}(u, v)$
  - (...sowie ein entsprechender kürzester Pfad)
- negative Kantengewichte erlaubt (aber **keine negativen Kreise!**)
- SSSP-Algorithmen für neg. Kantengewichte existieren natürlich
  - z. B. **Bellman-Ford Algorithmus**

## Algorithmen und Datenstrukturen

## └ Paarweise kürzeste Pfade

## └ Kürzeste Pfade für alle Knotenpaare

Haben gesehen:

- SSSP (Single-Source Shortest Path)
  - ↳ kürzeste Pfade von einem Knoten zu **allen** anderen Knoten
  - ↳ für nicht-negative Kantengewichte (Dijkstra)

**Num: All-pairs Shortest Path (APSP)**

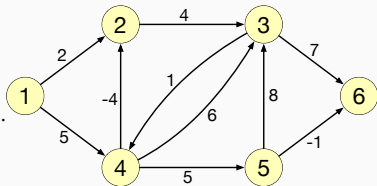
- Eingabe: gerichteter, gewichteter Graph  $G = (V, E)$
- Ausgabe:
  - ↳ für alle  $u, v \in V$  die Distanz dist[ $u, v$ ]
  - ↳ (...sowie ein entsprechender kürzester Pfad)

- negative Kantengewichte erlaubt (aber keine negativen Kreise)
- SSSP-Algorithmen für neg. Kantengewichte existieren natürlich
  - ↳ z.B. Bellman-Ford Algorithmus

- wir befassen uns zunächst nur mit der Berechnung der Distanzen für APSP
- später sehen wir, wie man auch die entsprechenden Pfade berechnen kann
- Bellman-Ford ist allerdings langsamer als Dijkstra

# APSP: Eingabe & Ausgabe

- müssen  $|V|^2$  Werte ausgeben
- z. B. als Matrix  $D$  mit den Distanzwerten
- verwende Adjazenzmatrix statt Adjazenzliste
  - Adjazenzliste wäre speichereffizienter, aber...
  - ...Ausgabe braucht bereits  $\Theta(|V|^2)$  Speicher
- nutzen gewichtete Adjazenzmatrix  $W = (W_{ij})$



$W$	1	2	3	4	5	6
1	0	2	$\infty$	5	$\infty$	$\infty$
2	$\infty$	0	4	$\infty$	$\infty$	$\infty$
3	$\infty$	$\infty$	0	1	$\infty$	7
4	$\infty$	-4	6	0	5	$\infty$
5	$\infty$	$\infty$	8	$\infty$	0	-1
6	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	0

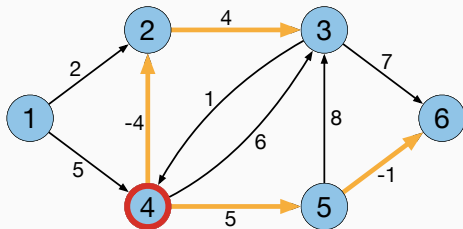
$D$	1	2	3	4	5	6
1	0	1	5	5	10	9
2	$\infty$	0	4	5	10	9
3	$\infty$	-3	0	1	6	5
4	$\infty$	-4	0	0	5	4
5	$\infty$	5	8	9	0	-1
6	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	0

$$W_{ij} = \begin{cases} 0 & \text{wenn } i = j \\ w(i,j) & \text{wenn } i \neq j \wedge (i,j) \in E \\ \infty & \text{wenn } i \neq j \wedge (i,j) \notin E \end{cases}$$

## APSP: Algorithmische Idee

- berechne Matrix  $D$  sukzessive
  - im  $k$ -ten Schritt, berechne Distanzmatrix  $D^{(k)}$  der...
  - ...kürzesten Wege die nur Knoten 1 bis  $k$  benutzen
    - Anfangs- und Endknoten dürfen  $> k$  sein
- $\rightsquigarrow D^{(0)} = W$  und  $D^{(|V|)} = D$

$D^{(6)}$	1	2	3	4	5	6
1	0	1	5	5	10	9
2	$\infty$	0	4	5	10	9
3	$\infty$	-3	0	1	6	5
4	$\infty$	-4	0	0	5	4
5	$\infty$	5	8	9	0	-1
6	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	0



Können wir  $D^{(k)}$  aus den  $D^{(l)}$  mit  $l < k$  berechnen?

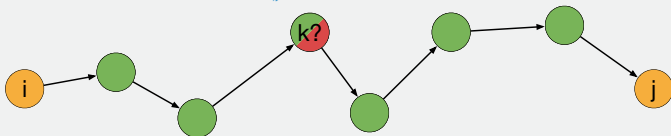
## APSP: Kürzeste Pfade über $\{1, 2, \dots, k\}$

- betrachte kürzesten Pfad  $P_{i,j}$  von  $i$  nach  $j$  in  $G$
- $\Rightarrow P_{i,j}$  enthält keinen Knoten doppelt
- sonst enthält  $P_{i,j}$  einen (nicht-negativen!) Kreis...
  - ...dessen Entfernung kürzeren Pfad liefert

keine  
neg.  
Kreise!

### Was nutzt uns diese Beobachtung?

- betrachte kürzesten Pfad  $P_{i,j}$  von  $i$  nach  $j$  über  $\{1, 2, \dots, k-1, k\}$



$\rightsquigarrow$  Knoten  $k$  kommt auf  $P_{i,j}$  maximal einmal vor

## Algorithmen und Datenstrukturen

## └ Paarweise kürzeste Pfade

└ APSP: Kürzeste Pfade über  $\{1, 2, \dots, k\}$ 

- gibt es mehrere kürzeste Pfade, so wählen wir einen mit den wenigsten Kanten (welcher ist egal)

APSP: Kürzeste Pfade über  $\{1, 2, \dots, k\}$ 

- betrachte kürzesten Pfad  $P_{ij}$  von  $i$  nach  $j$  in  $G$
- $P_{ij}$  enthält keinen Knoten doppelt
  - sonst enthält  $P_{ij}$  einen (nicht-negativen) Kreis...
  - ...dessen Entfernung kürzeren Pfad liefert

Knoten  
und  
Kanten

Was nutzt uns diese Beobachtung?

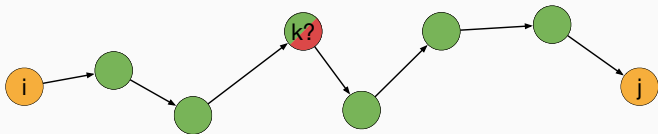
- betrachte kürzesten Pfad  $P_{ij}$  von  $i$  nach  $j$  über  $\{1, 2, \dots, k-1, k\}$

→ Knoten  $k$  kommt auf  $P_{ij}$  maximal einmal vor



## APSP: Rekursion für kürzeste Pfade

- betrachte kürzesten Pfad  $P_{i,j}$  von  $i$  nach  $j$  über  $\{1, 2, \dots, k-1, k\}$
- Knoten  $k$  kommt auf  $P_{i,j}$  maximal einmal vor



Fall 1:  $k$  nicht auf  $P_{i,j}$

$\Rightarrow P_{i,j}$  ist kürzester Pfad von  $i$  nach  $j$  über  $\{1, 2, \dots, k-1\}$

Fall 2:  $k$  auf  $P_{i,j}$

$\Rightarrow P_{i,j}$  ist kürzester Pfad von  $i$  nach  $k$  über  $\{1, 2, \dots, k-1\}$ ...

...gefolgt von kürzestem Pfad von  $k$  nach  $j$  über  $\{1, 2, \dots, k-1\}$

## APSP: Die Rekursion als Formel

- betrachte gewichteten Graph  $G = (V, E)$  mit  $V = \{1, 2, \dots, n\}$ ...
- ...gegeben durch seine gewichtete Adjazenzmatrix  $W = (W_{ij})$
- sei  $D^{(k)} = (D_{ij}^{(k)})$  Distanzmatrix über  $\{1, 2, \dots, k\}$

Für  $k \in \{0, 1, \dots, n\}$  gilt:

$$D_{ij}^{(k)} = \begin{cases} W_{ij} & \text{wenn } k = 0 \\ \min \{ D_{ij}^{(k-1)}, D_{ik}^{(k-1)} + D_{kj}^{(k-1)} \} & \text{wenn } k > 0 \end{cases}$$

---

### Algorithmus 5.9: FLOYDWARSHALL( $W, n$ )

---

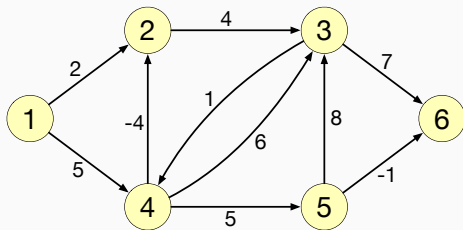
```
1   $D^{(0)} \leftarrow W$ 
2  for  $k \leftarrow 1$  to  $n$ 
3      for  $i \leftarrow 1$  to  $n$ 
4          for  $j \leftarrow 1$  to  $n$ 
5               $D_{ij}^{(k)} = \min \{ D_{ij}^{(k-1)}, D_{ik}^{(k-1)} + D_{kj}^{(k-1)} \}$ 
6  return  $D^{(n)}$ 
```

---

## APSP: Beispiel für Floyd-Warshall Algorithmus (1/6)

Für  $k \in \{1, 2, \dots, 6\}$  gilt:

$$D_{ij}^{(k)} = \min \{ D_{ij}^{(k-1)}, D_{ik}^{(k-1)} + D_{kj}^{(k-1)} \}$$



$D^{(0)}$	1	2	3	4	5	6
1	0	2	$\infty$	5	$\infty$	$\infty$
2	$\infty$	0	4	$\infty$	$\infty$	$\infty$
3	$\infty$	$\infty$	0	1	$\infty$	7
4	$\infty$	-4	6	0	5	$\infty$
5	$\infty$	$\infty$	8	$\infty$	0	-1
6	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	0

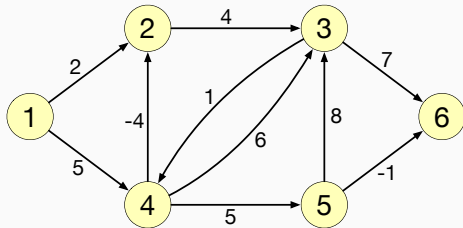


$D^{(1)}$	1	2	3	4	5	6
1	0	2	$\infty$	5	$\infty$	$\infty$
2	$\infty$	0	4	$\infty$	$\infty$	$\infty$
3	$\infty$	$\infty$	0	1	$\infty$	7
4	$\infty$	-4	6	0	5	$\infty$
5	$\infty$	$\infty$	8	$\infty$	0	-1
6	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	0

## APSP: Beispiel für Floyd-Warshall Algorithmus (2/6)

Für  $k \in \{1, 2, \dots, 6\}$  gilt:

$$D_{ij}^{(k)} = \min \{ D_{ij}^{(k-1)}, D_{ik}^{(k-1)} + D_{kj}^{(k-1)} \}$$



$D^{(1)}$	1	2	3	4	5	6
1	0	2	$\infty$	5	$\infty$	$\infty$
2	$\infty$	0	4	$\infty$	$\infty$	$\infty$
3	$\infty$	$\infty$	0	1	$\infty$	7
4	$\infty$	-4	6	0	5	$\infty$
5	$\infty$	$\infty$	8	$\infty$	0	-1
6	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	0

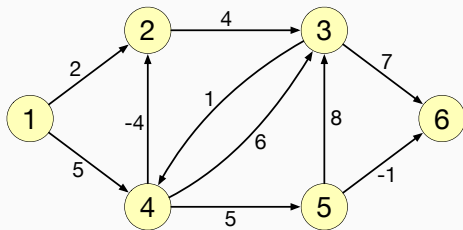


$D^{(2)}$	1	2	3	4	5	6
1	0	2	6	5	$\infty$	$\infty$
2	$\infty$	0	4	$\infty$	$\infty$	$\infty$
3	$\infty$	$\infty$	0	1	$\infty$	7
4	$\infty$	-4	0	0	5	$\infty$
5	$\infty$	$\infty$	8	$\infty$	0	-1
6	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	0

## APSP: Beispiel für Floyd-Warshall Algorithmus (3/6)

Für  $k \in \{1, 2, \dots, 6\}$  gilt:

$$D_{ij}^{(k)} = \min \{ D_{ij}^{(k-1)}, D_{ik}^{(k-1)} + D_{kj}^{(k-1)} \}$$



$D^{(2)}$	1	2	3	4	5	6
1	0	2	6	5	$\infty$	$\infty$
2	$\infty$	0	4	$\infty$	$\infty$	$\infty$
3	$\infty$	$\infty$	0	1	$\infty$	7
4	$\infty$	-4	0	0	5	$\infty$
5	$\infty$	$\infty$	8	$\infty$	0	-1
6	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	0

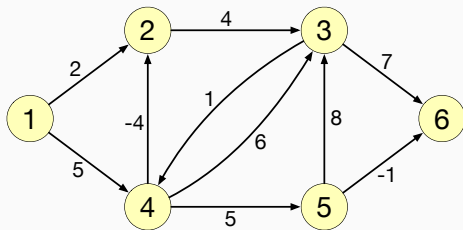


$D^{(3)}$	1	2	3	4	5	6
1	0	2	6	5	$\infty$	13
2	$\infty$	0	4	5	$\infty$	11
3	$\infty$	$\infty$	0	1	$\infty$	7
4	$\infty$	-4	0	0	5	7
5	$\infty$	$\infty$	8	9	0	-1
6	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	0

## APSP: Beispiel für Floyd-Warshall Algorithmus (4/6)

Für  $k \in \{1, 2, \dots, 6\}$  gilt:

$$D_{ij}^{(k)} = \min \{ D_{ij}^{(k-1)}, D_{ik}^{(k-1)} + D_{kj}^{(k-1)} \}$$



$D^{(3)}$	1	2	3	4	5	6
1	0	2	6	5	$\infty$	13
2	$\infty$	0	4	5	$\infty$	11
3	$\infty$	$\infty$	0	1	$\infty$	7
4	$\infty$	-4	0	0	5	7
5	$\infty$	$\infty$	8	9	0	-1
6	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	0

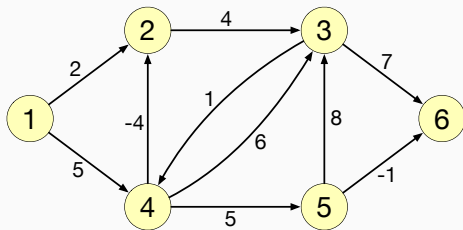


$D^{(4)}$	1	2	3	4	5	6
1	0	1	5	5	10	12
2	$\infty$	0	4	5	10	11
3	$\infty$	-3	0	1	6	7
4	$\infty$	-4	0	0	5	7
5	$\infty$	5	8	9	0	-1
6	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	0

## APSP: Beispiel für Floyd-Warshall Algorithmus (5/6)

Für  $k \in \{1, 2, \dots, 6\}$  gilt:

$$D_{ij}^{(k)} = \min \{ D_{ij}^{(k-1)}, D_{ik}^{(k-1)} + D_{kj}^{(k-1)} \}$$



$D^{(4)}$	1	2	3	4	5	6
1	0	1	5	5	10	12
2	$\infty$	0	4	5	10	11
3	$\infty$	-3	0	1	6	7
4	$\infty$	-4	0	0	5	7
5	$\infty$	5	8	9	0	-1
6	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	0

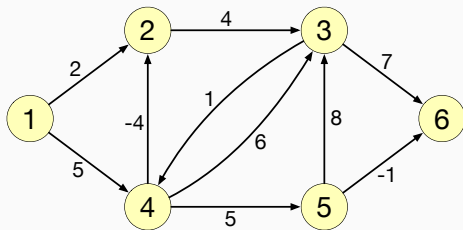


$D^{(5)}$	1	2	3	4	5	6
1	0	1	5	5	10	9
2	$\infty$	0	4	5	10	9
3	$\infty$	-3	0	1	6	5
4	$\infty$	-4	0	0	5	4
5	$\infty$	5	8	9	0	-1
6	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	0

## APSP: Beispiel für Floyd-Warshall Algorithmus (6/6)

Für  $k \in \{1, 2, \dots, 6\}$  gilt:

$$D_{ij}^{(k)} = \min \{ D_{ij}^{(k-1)}, D_{ik}^{(k-1)} + D_{kj}^{(k-1)} \}$$



$D^{(5)}$	1	2	3	4	5	6
1	0	1	5	5	10	9
2	$\infty$	0	4	5	10	9
3	$\infty$	-3	0	1	6	5
4	$\infty$	-4	0	0	5	4
5	$\infty$	5	8	9	0	-1
6	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	0



$D^{(6)}$	1	2	3	4	5	6
1	0	1	5	5	10	9
2	$\infty$	0	4	5	10	9
3	$\infty$	-3	0	1	6	5
4	$\infty$	-4	0	0	5	4
5	$\infty$	5	8	9	0	-1
6	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	0



### Theorem 5.12

`FLOYDWARSHALL` löst APSP für einen gewichteten gerichteten Graphen  $G = (V, E)$  ohne negative Zyklen in Zeit  $O(|V|^3)$ .

- Laufzeitschranke folgt trivial aus Pseudocode
  - drei verschachtelte for-Schleifen, jeweils über alle  $|V|$  Knoten
- Korrektheit folgt aus der Herleitung der Rekursionsformel

## APSP: Ausgabe des Pfades mit Floyd-Warshall

- Konstruiere Vorgängermatrix  $\Pi = (\Pi_{ij})$ 
  - d.h.  $\Pi_{ij}$  enthält Vorgänger von  $j$  auf kürzestem Pfad von  $i$  nach  $j$
- dazu konstruieren wir Matrizen  $\Pi^{(k)}$  für  $k \in \{1, 2, \dots, n\}$ 
  - $\Pi^{(k)}$  entspricht Vorgängermatrix zu Distanzen in  $D^{(k)}$
  - $\Pi_{ij}^{(k)}$  enthält also Vorgänger von  $j$  auf kürzestem Pfad von  $i$  nach  $j$  über Knoten aus  $\{1, 2, \dots, k\}$

Für  $k \in \{1, 2, \dots, n\}$  gilt:

$$\Pi_{ij}^{(k)} = \begin{cases} \Pi_{ij}^{(k-1)} & \text{wenn } D_{ij}^{(k-1)} \leq D_{ik}^{(k-1)} + D_{kj}^{(k-1)} \\ \Pi_{kj}^{(k-1)} & \text{wenn } D_{ij}^{(k-1)} > D_{ik}^{(k-1)} + D_{kj}^{(k-1)} \end{cases}$$

## 5) Weiterführende Graphprobleme

---

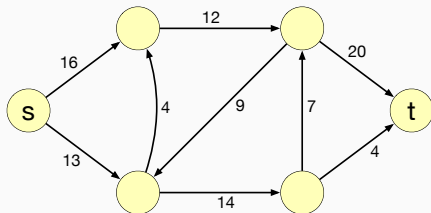
## Definition 5.5

Ein gewichteter gerichteter Graph  $G = (V, E)$  mit Kantengewichten (Kapazitäten)  $c: E \rightarrow \mathbb{R}_{\geq 0}$  heißt **Flussnetzwerk**, wenn es eine **Quelle**  $s \in V$  und eine **Senke**  $t \in V$  gibt, so dass für jedes  $v \in V$  ein Pfad  $s \rightsquigarrow v \rightsquigarrow t$  von  $s$  über  $v$  nach  $t$  existiert.

capacities

## Weitere Annahmen

- $G$  enthält keine Schleifen
- falls  $(u, v) \in E$  gilt, dann folgt  $(v, u) \notin E$
- für  $(u, v) \notin E$  definieren wir  $c(u, v) := 0$



# Algorithmen und Datenstrukturen

## └ Weiterführende Graphprobleme

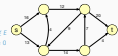
### └ Flussnetzwerke

## Definition 5.5

Ein gewichteter gerichteter Graph  $G = (V, E)$  mit Kantengewichten (Kapazitäten)  $c: E \rightarrow \mathbb{R}_{\geq 0}$  heißt **Flussnetzwerk**, wenn es eine **Quelle**  $s \in V$  und eine **Senke**  $t \in V$  gibt, so dass für jedes  $v \in V$  ein Pfad  $s \rightsquigarrow v \rightsquigarrow t$  von  $s$  über  $v$  nach  $t$  existiert.

## Weitere Annahmen

- $G$  enthält keine Schleifen
- falls  $(u, v) \in E$  gilt, dann folgt  $(v, u) \notin E$
- für  $(u, v) \notin E$  definieren wir  $c(u, v) := 0$



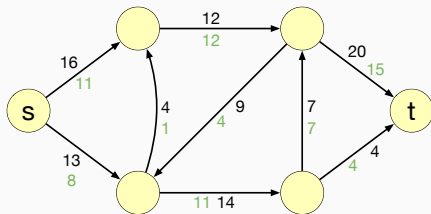
- Quelle **s**: source
- Senke **t**: target
- Erinnerung: Schleifen sind Kanten der Form  $(u, u)$
- Kanten  $(u, v) \in E$  und  $(v, u) \in E$  heißen **anti-parallel**
- der Verzicht auf anti-parallele Kanten vereinfacht einige technische Details, ist aber keine ernsthafte Einschränkung (siehe Beispiel auf der Folie (nicht in der Handout-Version))

# Flussnetzwerke: (Maximale) Flüsse

## Definition 5.6

Betrachte ein Flussnetzwerk  $G = (V, E)$  mit Kapazitäten  $c: E \rightarrow \mathbb{R}_{\geq 0}$ . Ein **Fluss** ist eine Funktion  $f: E \rightarrow \mathbb{R}_{\geq 0}$  welche die folgenden Bedingungen erfüllt:

- Kapazitätsbedingung:  $\forall (u, v) \in E$  gilt  $f(u, v) \leq c(u, v)$
- Flusserhaltung:  $\forall u \in V \setminus \{s, t\}$  gilt  $\sum_{(v, u) \in E} f(v, u) = \sum_{(u, v) \in E} f(u, v)$
- $|f| := \sum_{(s, u) \in E} f(s, u) - \sum_{(u, s) \in E} f(u, s)$  heißt **Wert** von  $f$



# Algorithmen und Datenstrukturen

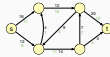
## └ Weiterführende Graphprobleme

### └ Flussnetzwerke: (Maximale) Flüsse

## Definition 5.6

Betrachte ein Flussnetzwerk  $G = (V, E)$  mit Kapazitäten  $c: E \rightarrow \mathbb{R}_{\geq 0}$ . Ein Fluss ist eine Funktion  $f: E \rightarrow \mathbb{R}_{\geq 0}$  welche die folgenden Bedingungen erfüllt:

- Kapazitätsbedingung:  $\forall (u, v) \in E$  gilt  $f(u, v) \leq c(u, v)$
- Flusserhaltung:  $\forall u \in V \setminus \{s, t\}$  gilt  $\sum_{(u, v) \in E} f(u, v) = \sum_{(v, u) \in E} f(v, u)$
- $|f| := \sum_{(s, v) \in E} f(s, v) - \sum_{(u, s) \in E} f(u, s)$  heißt Wert von  $f$



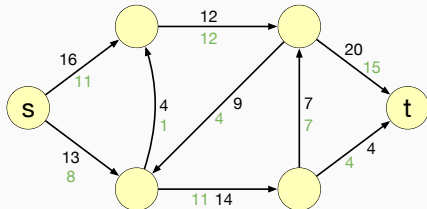
- $s$  hat in Flussnetzwerken typischerweise Eingangsgrad  $0$ , so dass die zweite Summe in der Definition von  $|f|$  meist weggelassen werden kann

- betrachte einen gegebenen Fluss  $f$  für  $G = (V, E)$
- definiere das **Residualnetzwerk**  $G_f = (V, E_f)$  mit **Restkapazitäten**  $c_f: E_f \rightarrow \mathbb{R}_{>0}$
- Kanten  $(u, v) \in E_f$  entsprechen einem der folgenden Typen
  - $(u, v) \in E$  und  $f(u, v) < c(u, v)$  ; hier gilt  $c_f(u, v) = c(u, v) - f(u, v)$
  - $(v, u) \in E$  und  $f(v, u) > 0$  ; hier gilt  $c_f(u, v) = f(v, u)$

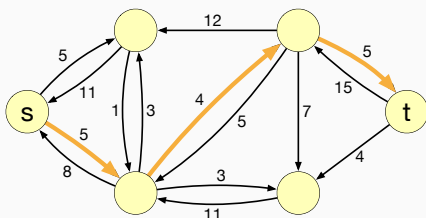
**Augmentierender Pfad**  $p = (s = u_0, u_1, \dots, u_k = t)$

- Pfad von  $s$  nach  $t$  in  $G_f$
- hat **Wert**  $|p| := \min \{ c_f(u_{i-1}, u_i) \mid i \in \{1, 2, \dots, k\} \}$

Flussnetzwerk  $G$  und Fluss  $f$



Residualnetzwerk  $G_f$





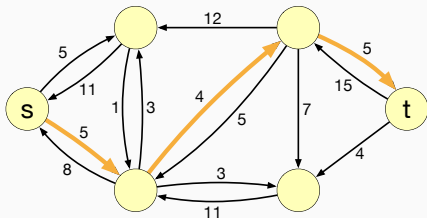
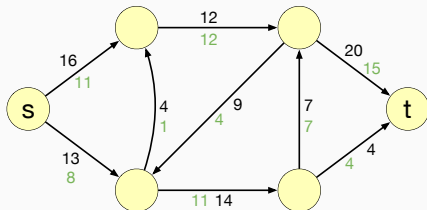
Augmentierender Pfad  $p = (s = u_0, u_1, \dots, u_k = t)$

- Pfad von  $s$  nach  $t$  in  $G_f$
- hat Wert  $|p| := \min \{ c_f(u_{i-1}, u_i) \mid i \in \{1, 2, \dots, k\} \}$

## Methode von Ford und Fulkerson

- initialisiere den Fluss  $f$  mit  $f(u, v) = 0$  für alle  $(u, v) \in E$
- solange ein augmentierender Pfad  $p$  existiert: finde solch ein  $p$  und...
- ...erhöhe  $f$  „entlang“  $p$  um  $|p|$

## Beispiel



# Algorithmen und Datenstrukturen

## — Weiterführende Graphprobleme

### — Maximale Flüsse: Algorithmische Idee

(2/2)

Maximale Flüsse: Algorithmische Idee (2/2)

**Augmentierender Pfad**

- Pfad von  $s$  nach  $t$  in  $G$
- hat Wert  $|p| = \min \{ c_f(u_i, u_{i+1}) \mid i \in \{1, 2, \dots, k\} \}$

**Methode von Ford und Fulkerson**

- initialisiere den Fluss  $f$  mit  $f(u, v) = 0$  für alle  $(u, v) \in E$
- solange ein augmentierender Pfad  $p$  existiert: finde solch ein  $p$  und...
- ...erhöhe  $f$  „entlang“  $p$  um  $|p|$

Beispiel

- erhöhen von  $f$  entlang  $p$ : für jede Kante  $(u_{i-1}, u_i)$  von  $p$  erhöhe  $f(u_{i-1}, u_i)$  um  $|p|$  falls  $(u_{i-1}, u_i) \in E$  und erniedrige  $f(u_{i-1}, u_i)$  um  $|p|$  falls  $(u_{i-1}, u_i) \notin E$

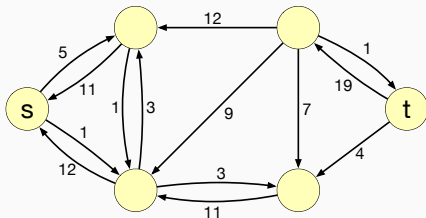
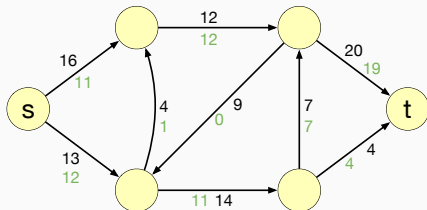
Augmentierender Pfad  $p = (s = u_0, u_1, \dots, u_k = t)$

- Pfad von  $s$  nach  $t$  in  $G_f$
- hat Wert  $|p| := \min \{ c_f(u_{i-1}, u_i) \mid i \in \{1, 2, \dots, k\} \}$

## Methode von Ford und Fulkerson

- initialisiere den Fluss  $f$  mit  $f(u, v) = 0$  für alle  $(u, v) \in E$
- solange ein augmentierender Pfad  $p$  existiert: finde solch ein  $p$  und...
- ...erhöhe  $f$  „entlang“  $p$  um  $|p|$

## Beispiel



# Algorithmen und Datenstrukturen

## — Weiterführende Graphprobleme

### — Maximale Flüsse: Algorithmische Idee

(2/2)

Maximale Flüsse: Algorithmische Idee (2/2)

**Augmentierender Pfad**

- Pfad von  $s$  nach  $t$  in  $G$
- hat Wert  $|p| = \min \{ c_f(u_i, u_{i+1}) \mid i \in \{1, 2, \dots, k\} \}$

**Methode von Ford und Fulkerson**

- initialisiere den Fluss  $f$  mit  $f(u, v) = 0$  für alle  $(u, v) \in E$
- solange ein augmentierender Pfad  $p$  existiert: finde solch ein  $p$  und...
- ...erhöhe  $f$  „entlang“  $p$  um  $|p|$

Beispiel

- erhöhen von  $f$  entlang  $p$ : für jede Kante  $(u_{i-1}, u_i)$  von  $p$  erhöhe  $f(u_{i-1}, u_i)$  um  $|p|$  falls  $(u_{i-1}, u_i) \in E$  und erniedrige  $f(u_{i-1}, u_i)$  um  $|p|$  falls  $(u_{i-1}, u_i) \notin E$

# Eigenschaften der Ford-Fulkerson Methode

- sei  $|f^*|$  der Wert eines maximalen Flusses
- ↪ Laufzeit  $O(|E| \cdot |f^*|)$ 
  - maximal  $|f^*|$  Erhöhungen des Flows
  - je LZ  $O(|E|)$  zum Finden eines augmentierenden Pfades

## Verbesserung (Edmonds-Karp Algorithmus)

- beweisbar: Distanz von  $s$  nach  $v$  in  $G_f$  (ungewichtet) steigt pro Schritt
- ↪ Anzahl Augmentierungen via kürzester Pfade ist höchstens  $O(|V| \cdot |E|)$
- ↪ Laufzeit  $O(|V| \cdot |E|^2)$ 
  - maximal  $O(|V| \cdot |E|)$  Erhöhungen des Flows
  - je LZ  $O(|E|)$  zum Finden eines kürzesten augmentierenden Pfades

# Algorithmen und Datenstrukturen

## — Weiterführende Graphprobleme

### — Eigenschaften der Ford-Fulkerson Methode

- finde irgendeinen augmentierenden Pfad z. B. via BFS
- nehmen hier o. B. d. A. **integrale** Werte für Kapazitäten und Fluss an
- finde eines **kürzesten** augmentierenden Pfad auch wieder via BFS
- wir führen hier aus Zeitgründen keine Analyse durch, aber alle Details sind im Cormen und mit dem bisher gelernten vollständig nachvollziehbar
- für Infos zu spannenden Anwendungen von maximalen Netzwerkflüssen siehe z. B. [Wikipedia](#)

- sei  $|f^*|$  der Wert eines maximalen Flusses
- Laufzeit  $O(|E| \cdot |f^*|)$ 
  - maximal  $|f^*|$  Erhöhungen des Flusses
  - je LZ  $O(|E|)$  zum Finden eines augmentierenden Pfades

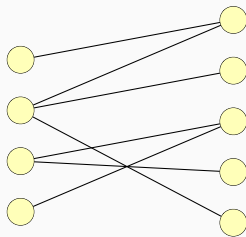
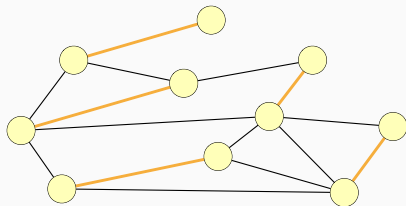
#### Verbesserung (Edmonds- Karp Algorithmus)

- **Laufzeit** Distanz von  $s$  nach  $t$  in  $G_f$  (ungewichtet) steigt pro Schritt
- Anzahl Augmentierungen via **kürzester** Pfade ist höchstens  $O(|V| \cdot |E|)$
- Laufzeit  $O(|V| \cdot |E|^2)$  Erhöhungen des Flusses
- maximal  $O(|V| \cdot |E|)$  Erhöhungen des Flusses
- je LZ  $O(|E|)$  zum Finden eines **kürzesten** augmentierenden Pfades

# Matchings und bipartite Graphen

## Definition 5.7

Betrachte einen ungerichteten Graphen  $G = (V, E)$ . Ein **Matching** ist eine Teilmenge  $M \subseteq E$ , so dass für alle Knoten  $v \in V$  gilt  $|\{e \in M \mid v \in e\}| \leq 1$ .



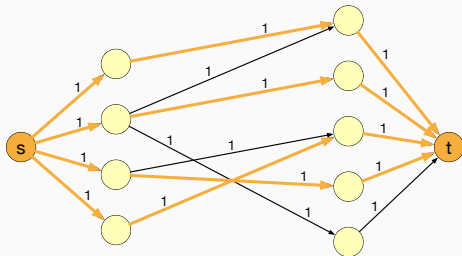
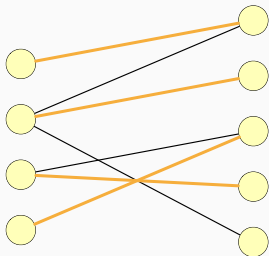
## Definition 5.8

Ein ungerichteter Graph  $G = (V, E)$  heißt **bipartit**, wenn  $V = V_1 \cup V_2$  mit  $V_1 \cap V_2 = \emptyset$  und  $|V_i \cap e| = 1$  für alle  $e \in E$  und  $i \in \{1, 2\}$ .

# Maximum-Matchings

## Algorithmisches Problem

Gegeben ein ungerichteter, bipartiter Graph  $G = (V, E)$ , bestimme ein Matching  $M \subseteq E$  mit der größtmöglichen Kardinalität  $|M|$ .



- kann auf Finden eines maximalen Flusses reduziert werden...
- ...und somit in Laufzeit  $O(|V| \cdot |E|)$  gelöst werden.
  - maximaler Fluss  $f^*$  hat Wert  $|f^*| = |M|$  des Maximum-Matchings



# Algorithmen und Datenstrukturen

## — Weiterführende Graphprobleme

### — Maximum-Matchings

#### Maximum-Matchings

##### Algorithmisches Problem

Gegeben ein ungerichteter, bipartiter Graph  $G = (V, E)$ , bestimme ein Matching  $M \subseteq E$  mit der größtmöglichen Kardinalität  $|M|$ .



- kann auf Finden eines maximalen Flusses reduziert werden...
- ...und somit in Laufzeit  $O(|V| \cdot |E|)$  gelöst werden.
- maximaler Fluss  $f^*$  hat Wert  $f^* = |M|$  des Maximum-Matchings

- effizientere Algorithmen für bipartite Graphen und Spezialfälle existieren, siehe z. B. [Wikipedia](#)
- natürlich ist das Problem auch auf nicht-bipartiten und/oder gewichteten Graphen interessant, siehe auch hier z. B. [Wikipedia](#)
- Beachte: „Maximum-Matching“  $\neq$  „maximales Matching“
- „Maximum-Matching“: größtmögliche Anzahl an Kanten unter allen Matchings
- „maximales Matching“: Matching, dass nicht durch hinzunehmen einer weiteren Kante vergrößert werden kann
- im Englischen unterscheidet man entsprechend auch „maximum matching“ und „maximal matching“

# Ausblick auf schwere Graphprobleme

- alle gezeigten Graphalgorithmen hatten **polynomielle** Laufzeit
  - bzgl. Anzahl der Knoten und der Kanten
- es gibt eine Vielzahl an **schweren** Probleme
  - für diese ist **kein** Algorithmus polynomieller Laufzeit **bekannt**

## Erkenntnis der Komplexitätstheorie

Effiziente Lösbarkeit bestimmter Graphprobleme erlaubt effiziente Lösung vieler anderer Graphprobleme.

mehr dazu in Kapitel 7 und weiterführenden VL

## Beispiele schwerer Graphprobleme

- **Hamiltonkreisproblem**: Finde Kreis der Länge  $|V|$ .
- **Isomorphie von Graphen**: Sind zwei Graphen  $G$  und  $G'$  isomorph?
- **Cliquenproblem**: Finde vollständig verbundenen Teilgraphen mit  $k$  Knoten.
- **Knotenfärbung**: Finde Färbung, die adjazente Knoten verschieden färbt.

# Algorithmen und Datenstrukturen

## └ Weiterführende Graphprobleme

### └ Ausblick auf schwere Graphprobleme

#### Ausblick auf schwere Graphprobleme

- alle gezeigten Graphalgorithmen hatten **polynomielle** Laufzeit
  - tagt Anzahl der Knoten und der Kanten
- es gibt eine Vielzahl an **schweren** Problemen
  - für diese ist **kein** Algorithmus polynomieller Laufzeit **bekannt**

#### Erkenntnis der Komplexitätstheorie

Effiziente Lösbarkeit bestimmter Graphprobleme erlaubt effiziente Lösung vieler anderer Graphprobleme.

mehr dazu in Kapitel 7 und weiterführenden VL

#### Beispiele schwerer Graphprobleme

- **Hamiltonkreisproblem:** Finde Kreis der Länge  $|V|$ .
- **Isomorphie von Graphen:** Sind zwei Graphen  $G$  und  $G'$  isomorph?
- **Cliqueproblem:** Finde vollständig verbundenen Teilgraphen mit  $k$  Knoten.
- **Knotenfärbung:** Finde Färbung, die adjazente Knoten verschieden färbt.

- Stichwort **NP-härte**
- Stichwort **Reduktion**; wir haben hier eine Reduktion von Maximum-Matching auf maximale Flüsse gesehen