

# Algorithmen und Datenstrukturen

## Kapitel 7: Elementare Graphenalgorithmen

Prof. Ingrid Scholl

FH Aachen - FB 5

[scholl@fh-aachen.de](mailto:scholl@fh-aachen.de)

22.05.2017



# Inhalt - Kapitel 7: Elementare Graphenalgorithmen

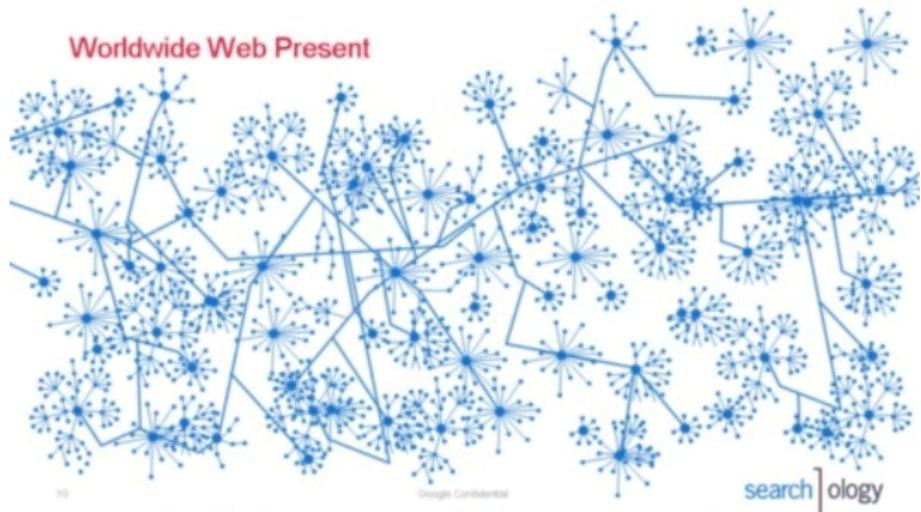
---

## ► Elementare Graphenalgorithmen

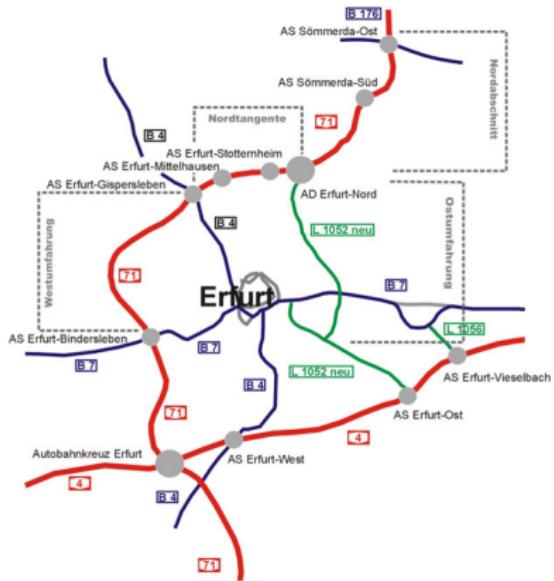
- Beispiele
- Definitionen
- Datenstrukturen
  - Kantenliste
  - Knotenliste
  - Adjazenzmatrix
  - Adjazenzliste
- Algorithmen zu Graphen
  - Tiefensuche
  - Rekursive Tiefensuche
  - Iterative Tiefensuche
  - Pfadsuche mit Tiefensuche
  - Connected Components mit Tiefensuche
  - Breitensuche
- Algorithmen zum Minimalen Spannbaum
  - Prim Algorithmus
  - Kruskal Algorithmus
- Kürzeste Pfade
  - Dijkstra Algorithmus

# Beispiele für Graphen - WWW

Graphen sind eine der wichtigsten Datenstrukturen. Bei vielen Problemstellungen lassen sich die Beziehungen zwischen den Datenobjekten am besten durch Graphen darstellen.

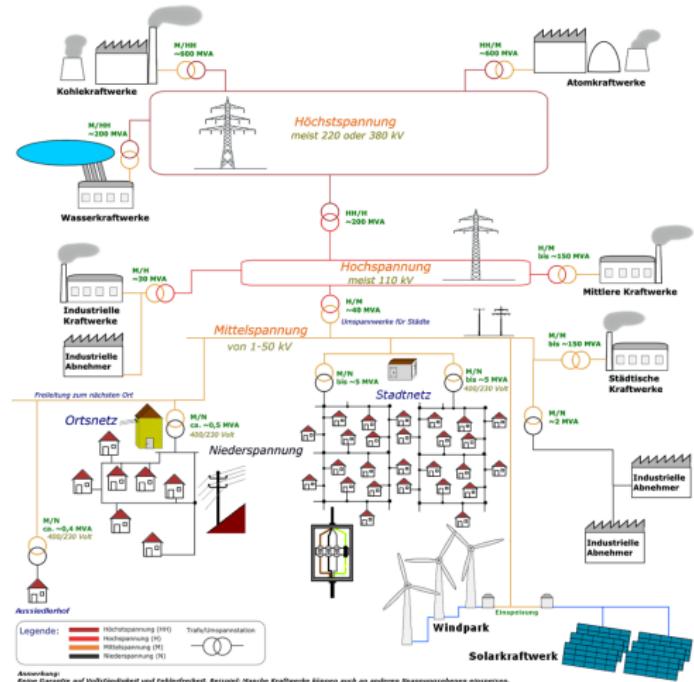


# Beispiele für Graphen - Strassenkarten



[http://www.thueringen.de/de/tlbv/abgeschlossene\\_bauprojekte/a71/](http://www.thueringen.de/de/tlbv/abgeschlossene_bauprojekte/a71/)

# Beispiele für Graphen - Energieversorgungsplan

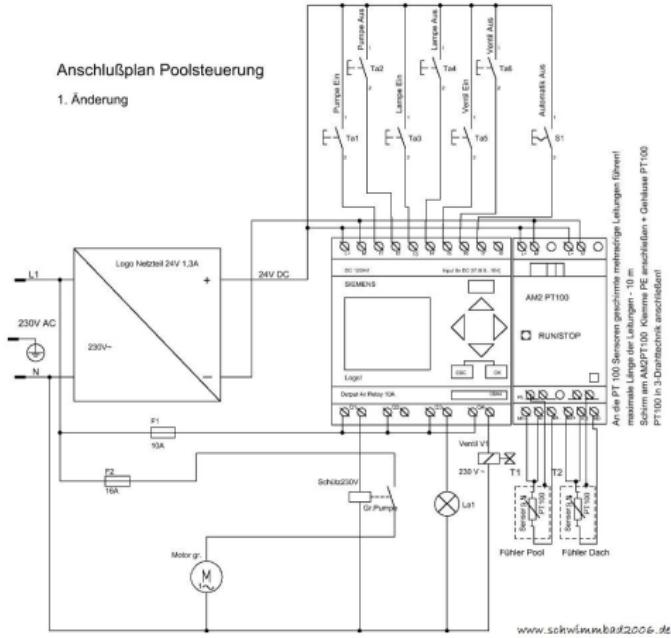


<http://de.wikipedia.org/wiki/Stromnetz>

# Beispiele für Graphen - Schaltplan

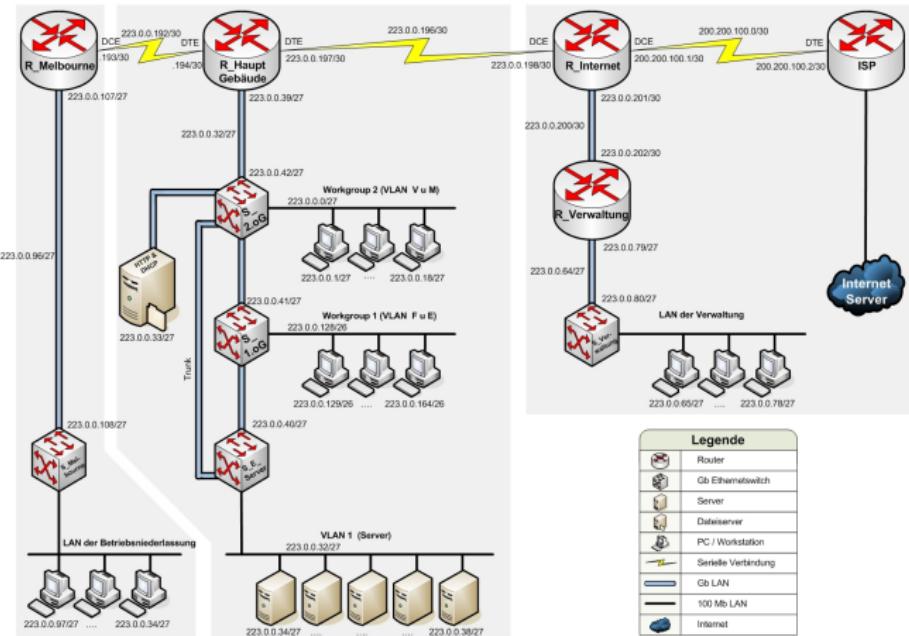
Anschlußplan Poolsteuerung

1. Änderung



[www.schwaigerbaud2006.de](http://www.schwaigerbaud2006.de)

# Beispiele für Graphen - Rechnernetze



# Beispiele für Graphen - Soziale Netzwerke

---



# Typische Anwendungen von Graphen

Anwendung	Element	Verbindung
Straßenkarte	Straßenkreuzung	Straße
Webinhalte	Seite	Link
Schaltung	Bauteil	Draht
Zeitablaufplan	Aufgabe	Beschränkung
Handel	Kunde	Transaktion
Rechnernetz	Rechner	Verbindung
Software	Methode	Aufruf
Soziales Netzwerk	Person	Freundschaft

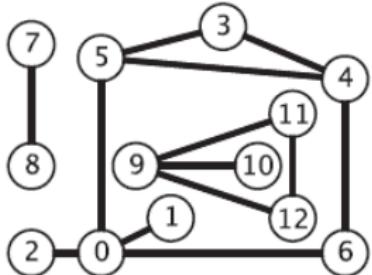
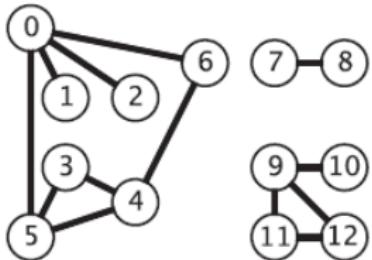
# Definitionen

## Definition (Graph)

Ein Graph  $G = (V, E)$  besteht aus einer Menge von Knoten (vertices)  $V$  und einer Menge von Kanten (edges)  $E$ , die jeweils zwei Knoten miteinander verbinden.

- ▶ Seien  $v \in V$  und  $w \in V$  zwei Knoten aus  $V$ . Dann besteht die Kante  $(v, w) \in E$  aus einem Paar von 2 Knoten. Kanten werden graphisch durch Verbindungslien von zwei Knoten dargestellt.
- ▶ Bei **gewichteten** Graphen sind Kantenverbindungen zusätzlich bewertet.
- ▶ Bei **gerichteten** Graphen bestehen Kanten aus geordneten Paaren  $(v, w)$ . Die Paare  $(v, w)$  und  $(w, v)$  definieren zwei Kanten.
- ▶ Bei **ungerichteten** Graphen bestehen Kanten aus **nicht geordneten** Paaren  $(v, w) = (w, v)$ .
- ▶ Ein Knoten  $w$  ist **adjazent** (benachbart) zu  $v$  genau dann, wenn  $(v, w) \in E$ .
- ▶  $|V|$  sei die Anzahl der Knoten und  $|E|$  die Anzahl der Kanten.

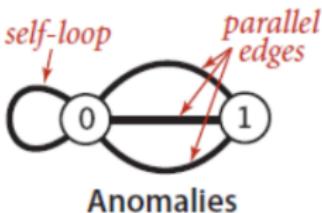
# Beispiel Graph



## Visualisierung eines Graphen:

- ▶ Knoten  $V = 0, 1, \dots, N - 1$  werden im Graphen durch Kreise dargestellt.
- ▶ Kanten  $E$  werden durch Verbindungslienien dargestellt.
- ▶ Beide Abbildungen zeigen denselben Graphen.

# Anomalien

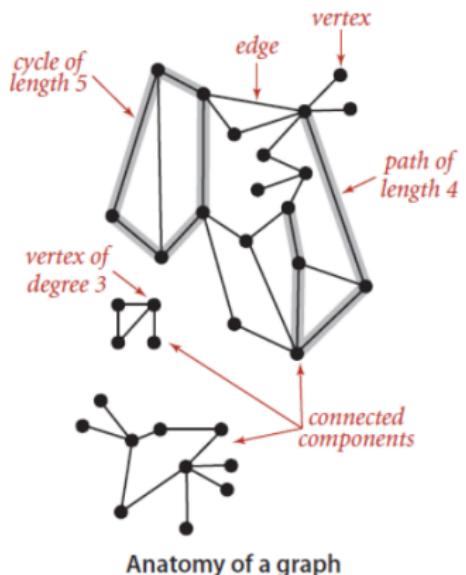


## Anomalien

- ▶ Eine **reflexive Kante (self-loop)**, auch **Schlinge** genannt, ist eine Kante, die einen Knoten mit sich selbst verbindet.
- ▶ Zwei Kanten, die das gleiche Knotenpaar verbinden, sind **parallel**.

Graphen mit parallelen Kanten werden als **Multigraphen** bezeichnet. Graphen ohne parallele oder reflexive Kanten als **einfache Graphen**.

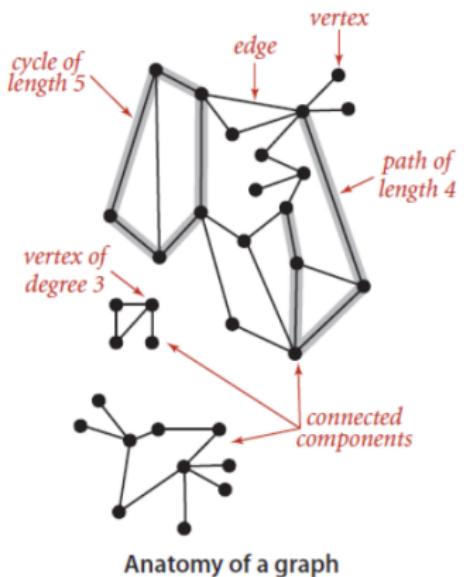
# Definitionen



## Definition (Pfad und Zyklus)

- ▶ Ein **Pfad** in einem Graphen besteht aus einer Folge von Knoten, die durch Kanten verbunden sind. In einem einfachen Pfad kommt jeder Knoten nur einmal vor.
- ▶ Ein **Zyklus** ist ein Pfad, der im selben Knoten beginnt und endet.
- ▶ Als **einfachen Zyklus** wird ein Zyklus bezeichnet, bei dem sich keine Kanten und Knoten wiederholen (bis auf den ersten und den letzten Knoten).
- ▶ Die **Länge eines Pfades oder eines Zyklus** ist die Anzahl der Kanten.

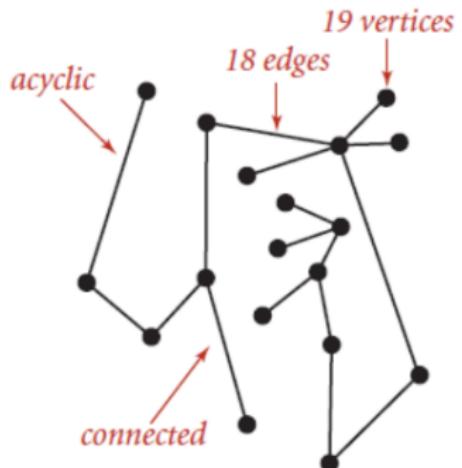
# Definitionen



## Definition (Zusammenhang)

- ▶ Ein Graph ist **zusammenhängend**, wenn es von jedem Knoten aus einen Pfad zu allen anderen Knoten im Graphen gibt.
- ▶ Ein **nicht zusammenhängender** Graph besteht aus einer Menge von **Zusammenhangskomponenten**, d.h. aus mehreren maximal zusammenhängenden Teilgraphen.
- ▶ Ein Graph heißt **stark zusammenhängend**, wenn im Graphen von jedem Knoten zu jedem anderen Knoten eine Kante existiert.

# Definitionen



A tree

## Definition (Azyklischer Graph, Baum)

- ▶ Ein *azyklischer Graph* ist ein Graph ohne Zyklen.
- ▶ Ein *Baum* ist ein azyklischer zusammenhängender Graph.

# Definitionen

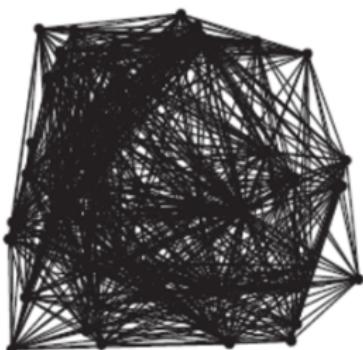


A spanning forest

## Definition (Spannbaum, Spannwald)

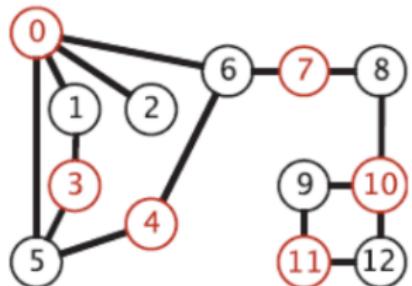
- ▶ Ein *Spannbaum* eines zusammenhängenden Graphen ist ein Teilgraph, der alle Knoten des Graphen enthält und ein Baum ist.
- ▶ Eine disjunkte Menge von Bäumen wird *Wald* genannt.
- ▶ Ein *Spannwald* eines (nicht zusammenhängenden) Graphen ist die Vereinigung der Spannbäume seiner Zusammenhangskomponenten.

# Definitionen

sparse ( $E = 200$ )dense ( $E = 1000$ )Two graphs ( $V = 50$ )

Die **Dichte** eines Graphen gibt an, wie viele der möglichen Knotenpaare tatsächlich durch Kanten verbunden sind. Ein **dünner Graph** hat wenige der möglichen Kanten. Bei einem **dichten Graphen** fehlen nur wenige der möglichen Kanten. Für ungerichtete Graphen ohne Schlingen gilt:  
 $|E| \leq 1/2 \cdot |V| \cdot |V - 1|$ .

# Definitionen



A bipartite graph

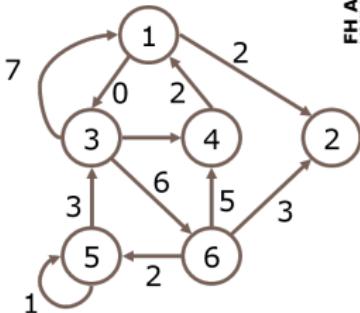
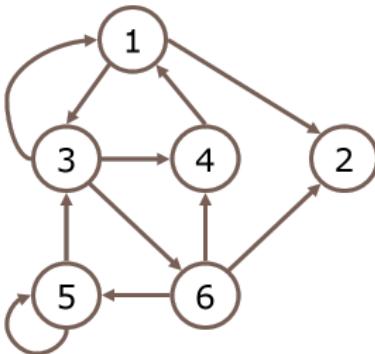
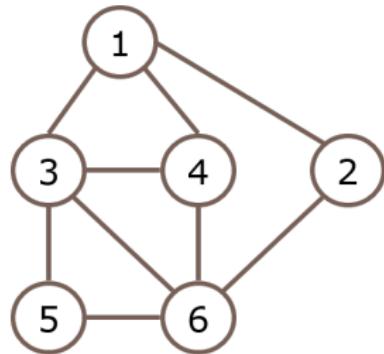
## Definition (Bipartiter Graph)

Ein *bipartiter Graph* ist ein Graph, dessen Knoten in zwei Mengen aufgeteilt werden können, wobei jede Kante jeweils einen Knoten der einen Menge mit einem Knoten der anderen Menge verbindet.

Erste Knotenmenge ist rot markiert, zweite Knotenmenge ist schwarz markiert.

# Definitionen

## Beispiele von Graphen:



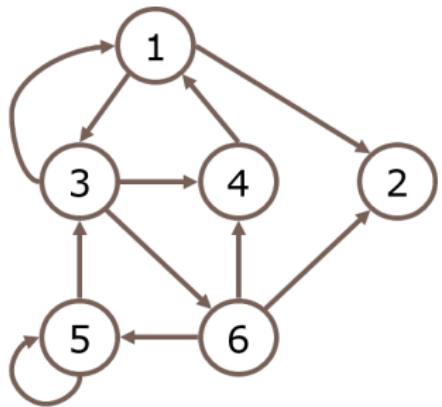
# Datenstrukturen für Graphen

---

Datenstrukturen für gerichtete Graphen:

1. Kantenliste
2. Knotenliste
3. Adjazenzmatrix
4. Adjazenzliste

# Kantenliste

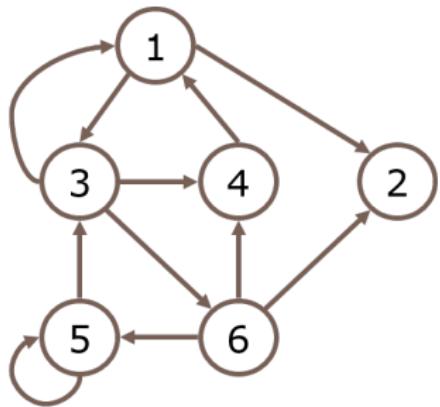


- ▶ Einfache Speicherung von Graphen als Liste von Zahlen.  
Mögliche Datenstrukturen:  
**vector, list, array.**
- ▶  $G[0] = |V| =$  Anzahl der Knoten
- ▶  $G[1] = |E| =$  Anzahl der Kanten
- ▶  $G[2i]$  und  $G[2i + 1]$  sind zwei adjazente Knoten mit  $i \in \{1, 2, \dots, |E|\}$ .

$G =$

{6, 11, 1, 2, 1, 3, 3, 1, 4, 1, 3, 4, 3, 6, 5, 3, 5, 5, 6, 5, 6, 2, 6, 4}

# Kantenliste

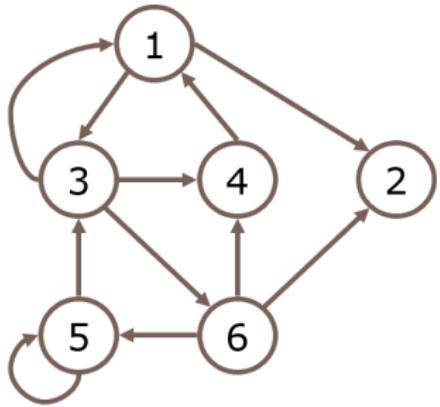


- ▶ Ebenfalls einfache Speicherung von Graphen als Liste von Zahlen.
- ▶ Beachte: bei ungerichteten Graphen wird eine Kante  $(v, w) \in E$  in der Kantenliste als geordnete Paare mit zwei Kanten  $(v, w)$  und  $(w, v)$  gespeichert.

$G =$

$\{6, 11, 1, 2, 1, 3, 3, 1, 4, 1, 3, 4, 3, 6, 5, 3, 5, 5, 6, 5, 6, 2, 6, 4\}$

# Kantenliste



- ▶  $G[0] = |V|$  = Anzahl der Knoten
- ▶  $G[1] = |E|$  = Anzahl der Kanten
- ▶  $G[2i]$  und  $G[2i + 1]$  sind zwei adjazente Knoten mit  $i \in \{1, 2, \dots, |E|\}$ .

$G =$

$\{6, 11, 1, 2, 1, 3, 3, 1, 4, 1, 3, 4, 3, 6, 5, 3, 5, 5, 6, 5, 6, 2, 6, 4\}$

# Aufwand für Kantenliste

## Speicheraufwand für Kantenliste:

$$2 + 2 \cdot |E|$$

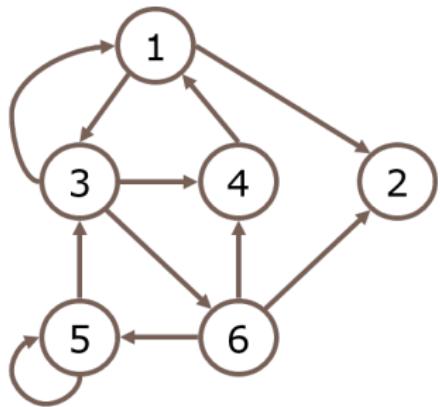
- ▶ 2 Speicherplätze für die Anzahl der Knoten und Anzahl der Kanten,
- ▶ 2 mal Anzahl der Kanten Speicherplätze, da die Kante durch ein Tupel bestehend aus 2 Knoten beschrieben wird.

$G =$

{6, 11, 1, 2, 1, 3, 3, 1, 4, 1, 3, 4, 3, 6, 5, 3, 5, 5, 6, 5, 6, 2, 6, 4}

**Hier:**  $|E| = 11$  und Array  $G$  benötigt 24 Einträge.

# Knotenliste



- ▶ DS der Knotenliste analog zur Kantenliste
- ▶  $G[0] = |V| =$  Anzahl der Knoten
- ▶  $G[1] = |E| =$  Anzahl der Kanten
- ▶  $G[2 \dots n - 1]$  definiert eine Liste von Knoteninformationen mit:

$$ag_i, v_{i1}, v_{i2}, \dots, v_{i, ag_i}$$

$ag_i$  ist die Anzahl der adjazenten Knoten gefolgt von einer Folge der adjazenten Knotennamen.

$$G = \{6, 11, 2, 2, 3, 0, 3, 1, 4, 6, 1, 1, 2, 3, 5, 3, 2, 4, 5\}$$

# Speicheraufwand Knotenliste

## Speicheraufwand für Knotenliste:

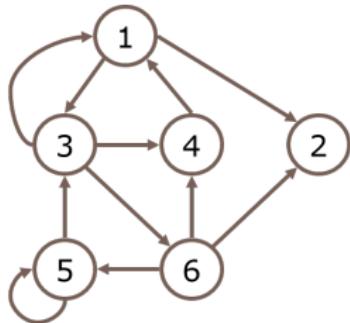
$$2 + |V| + |E|$$

- ▶ 2 Speicherplätze für die Anzahl der Knoten und Anzahl der Kanten,
- ▶ Speicherung der  $ag_i$  mit  $i \in \{1, \dots, |V|\}$  benötigen  $|V|$  Speicherplätze,
- ▶ die Anzahl der adjazenten Knoten ist gleich der Anzahl der Kanten  $|E|$ .

**Frage:** Bei dichten Graphen benötigt welche Datenstruktur weniger Speicher?

# Adjazenzmatrix

---



- ▶ DS ist ein **2D-Array**, eine  $n \times n$ -Matrix  $G$  mit  $n = |V|$ .
- ▶  $G(v, w) = 1$ , falls  $(v, w) \in E$ .
- ▶  $G(v, w) = 0$ , falls  $(v, w) \notin E$ .
- ▶ Bei **ungerichteten Graphen** gilt:  $G(v, w) = G(w, v)$ . Dies ergibt eine **symmetrische Matrix**.
- ▶ Bei **gewichteten Graphen** wird das Gewicht  $\gamma$  in die Matrix eingetragen mit  $G(v, w) = \gamma$ . Kantenmenge besteht aus Tripel:  $(v, w, \gamma) \in E$ .

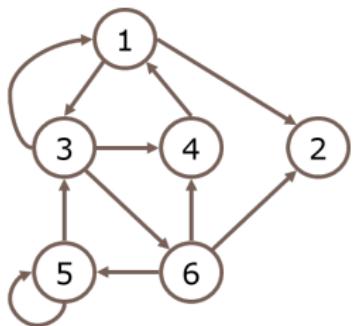
nach  
Knoten **1 2 3 4 5 6**

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

**1 von**  
**2 Knoten**

**3**  
**4**  
**5**  
**6**

# Speicheraufwand Adjazenzmatrix



## Speicheraufwand Adjazenzmatrix

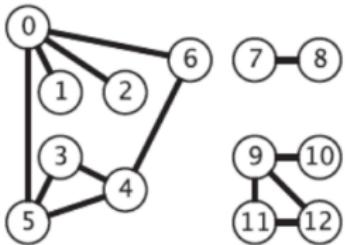
Die zweidimensionale Datenstruktur besteht aus  $|V|$  Zeilen und  $|V|$  Spalten:

$$|V| \cdot |V| = |V|^2$$

nach  
Knoten 1 2 3 4 5 6

$$G = \left( \begin{array}{cccccc} 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \end{array} \right) \quad \begin{array}{l} 1 \text{ von} \\ 2 \text{ Knoten} \\ 3 \\ 4 \\ 5 \\ 6 \end{array}$$

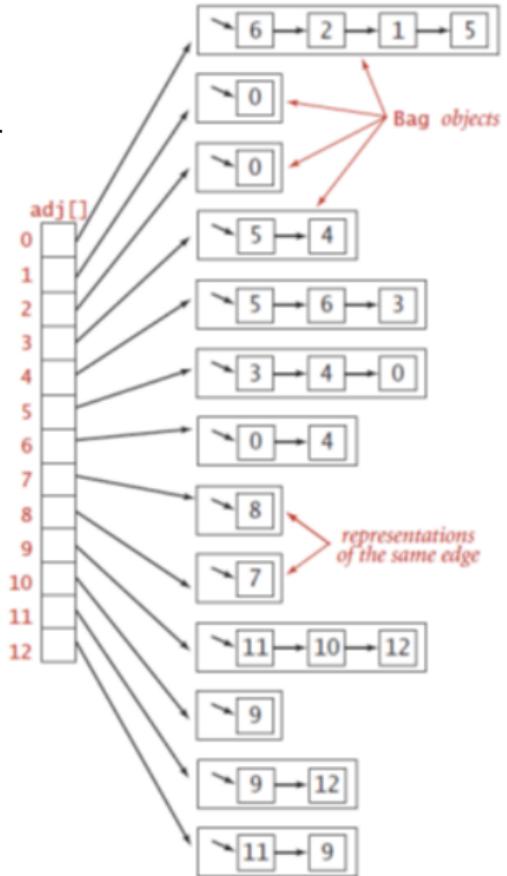
# Adjazenzliste



Graph wird durch 2 DSen realisiert:

**1. Datenstruktur:**  
Vektor über die Knoten

**2. Datenstruktur:**  
zu jedem Knoten eine Liste der adjazenten Knoten

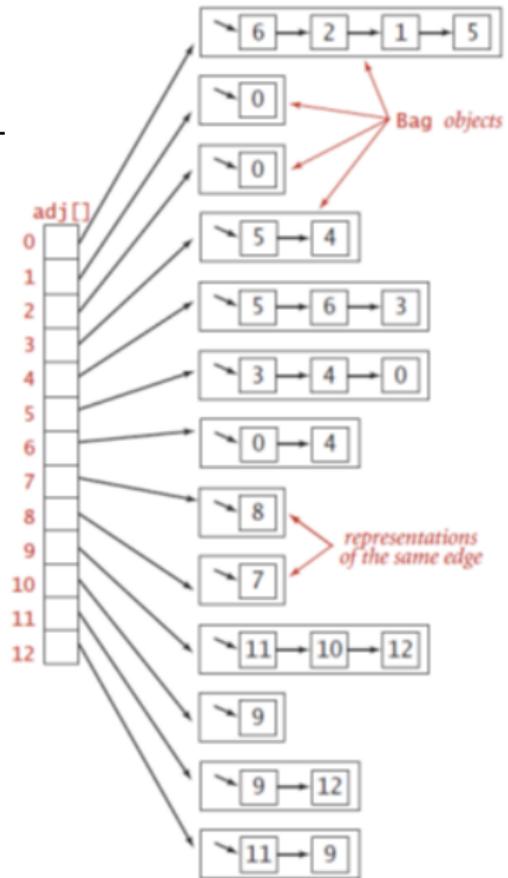


# Adjazenzliste

## Specheraufwand Adj.liste:

Die erste Datenstruktur benötigt  $|V|$  Einträge. Für jeden Knoten  $v \in V$  wird in der zweiten DS eine Liste der adjazenten Knoten gespeichert. Die Gesamtanzahl der adjazenten Knoten ist gleich der Anzahl der Kanten  $|E|$ . Insgesamt:

$$|V| + |E|$$



# Transformation verschiedener Graph-Representationen

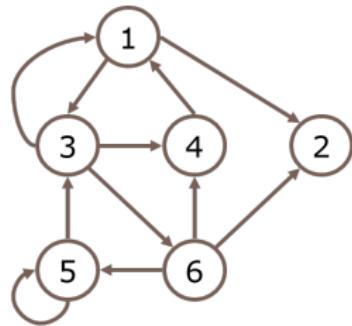
## Beispiel: Knotenliste in Adjazenzliste

- DS Knotenliste: `vector<int>` oder `list<int>`

$$G = \{6, 11, 2, 2, 3, 0, 3, 1, 4, 6, 1, 1, 2, 3, 5, 3, 2, 4, 5\}$$

- DS Adjazenzliste: `map< int, vector<int> >`

1	→	{2, 3}
2	→	{}
3	→	{1, 4, 6}
4	→	{1}
5	→	{3, 5}
6	→	{2, 4, 6}



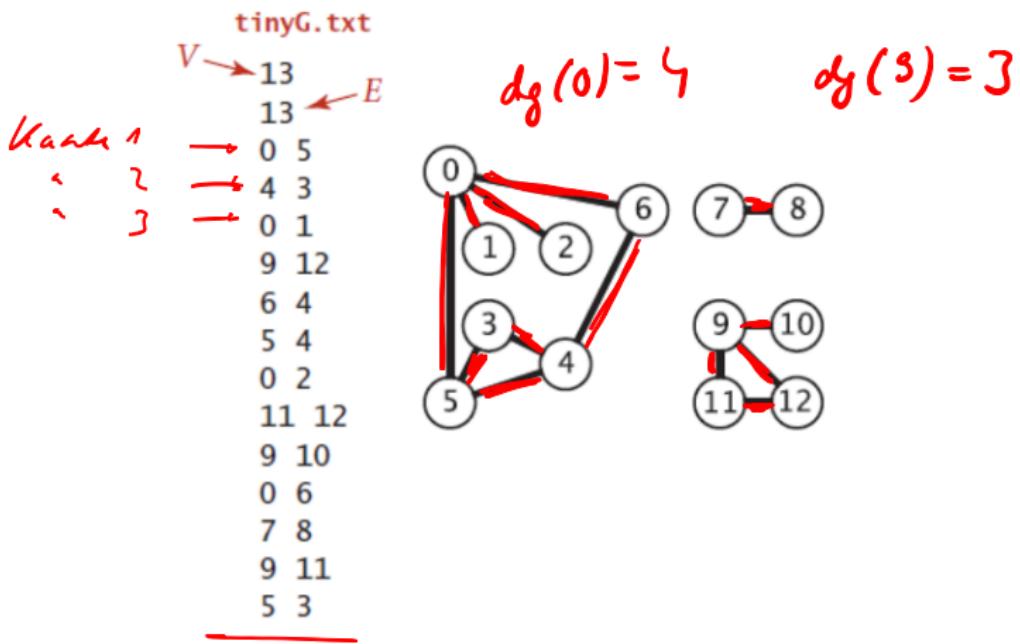
# Beispiel: Knotenliste in Adjazenzliste

```
map<int, vector<int> > nodelist2adjlist(const vector<int> &a) {
    int V = a[0]; // Anzahl Knoten
    int E = a[1]; // Anzahl Kanten
    int nodelistSize = V + E + 2;
    int i = 2; int j = 1;
    map<int, vector<int> > adj;
    while (i < nodelistSize) {
        int ag = a[i];
        adj[j];
        for (int k = i+1; i < i + ag + 1; k++) {
            int node = a[k];
            adj[j].push_back(node);
        }
        i = i + ag + 1;
        j++;
    }
    return adj;
}
```

# Beispiel API für einen Graphen

```
class Graph {  
private:  
    // Graph als Kantenliste  
    vector<int> edgelist; // Größe 2|E| + 2  
    // Alternativ: Graph als Adjazenzliste  
    map< int, vector<int> > adjlist;  
    map< int, vector<int> > edge2adjlist();  
public:  
    Graph(int V); // Erzeugt einen Graphen mit V Knoten,  
    // aber ohne Kanten  
    Graph(string fname); // Einlesen einer Kantenliste  
    // aus einer Datei fname  
    int V(); // Anzahl der Knoten  
    int E(); // Anzahl der Kanten  
    void addEdge(int v, int w); // fügt eine Kante (v,w) hinzu  
    bool removeEdge(int v, int w); // entfernt eine Kante (v,w)  
    vector<int> adj(int v); // Array der Nachbarknoten von v  
};
```

# Beispiel Eingabedatei für Kantenliste



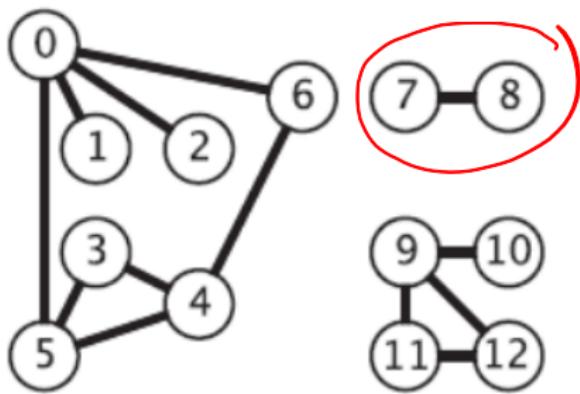
# Nachtrag: Definition - Grad

## Definition (Grad eines Knotens)

Mit  $d_G(v)$  bezeichnet man den **Grad (bzw. die Valenz)** des Knotens  $v$  in einem **ungerichteten Graphen**  $G$ . Dabei ist  $d_G(v)$  in Graphen ohne Mehrfachkanten und Hypergraphen die **Anzahl der Nachbarn von  $v$** , Graphen mit Mehrfachkanten die Summe der Vielfachheiten aller mit  $v$  inzidenten Kanten.

Den kleinsten Grad eines Knotens in  $G$  bezeichnet man dann als **Minimalgrad** von  $G$ , den größten Grad eines Knotens in  $G$  bezeichnet man als **Maximalgrad** von  $G$ . Das arithmetische Mittel aller Eckengrade von  $G$  wird als **Durchschnittsgrad**  $d_G(G)$  bezeichnet.

# Nachtrag: Beispiel - Grad



$$d_G(0) = 4$$

$$d_G(6) = 2$$

$$d_G(5) = 3$$

$$\text{MinDegree}(G) = 1$$

$$\text{MaxDegree}(G) = 4$$

$$\text{AvgDegree}(G) = (4+1+1+2+3+3+2+1+1+3+1+2+2)/13 = 26/13 = 2$$

$$\text{AvgDegree}(G) = \frac{2 \cdot |E|}{|V|} = 2 \cdot 2 \cdot 13/13 = 26/13 = 2$$

# Nachtrag: Beispiel - Grad

```
int degree(Graph G, int v) {
    int degree = 0;
    // hole Array der adjazenten Knoten zu v
    vector<int> adjv = G.adj(v);
    return adjv.size();
}

int maxDegree(Graph G) {
    int max = 0;
    for (int v = 0; v < G.V(); v++) {
        int maxv = degree(G, v);
        if (maxv > max) max = maxv;
    }
    return max;
}

double avgDegree(Graph G) {
    return (double) 2*G.E() / (double) G.V();
}
```

# O-Notationen für typische Graph-Implementierungen

Datenstruktur	Speicher	Kante ( $v,w$ ) hinzufügen	Prüfen, ob $w$ Nachbar von $v$ ist	Über die Nachbarknoten von $v$ iterieren
Kantenliste	$E$	1	$E$	$E$
Adjazenzmatrix	$V^2$	1	1	$V$
Adjazenzliste	$E + V$	1	$degree(v)$	$degree(v)$

# Graphenalgorithmen

## Fragen zum Graphen:

1. Ist der Graph zusammenhängend?
2. Wenn nicht, welche zusammenhängenden Komponenten gibt es?
3. Sind 2 gegebene Knoten miteinander verbunden?
4. Ist der Graph zykelfrei?
5. Topologische Sortierreihenfolge?
6. Operationen auf Graphen: Knoten einfügen und löschen, Kante einfügen und löschen

## Lösung:

Methode, die jeden Knoten des Graphen besucht (traversiert) und dabei jede Kante bzw. Knoten überprüft.

# Algorithmen zur Traversierung

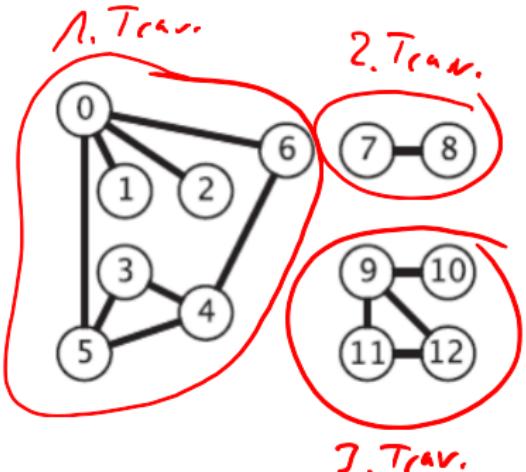
1. Rekursive Tiefensuche
2. Iterative Tiefensuche
3. Iterative Breitensuche

## Annahme

Geg.: Ungerichteter Graph!

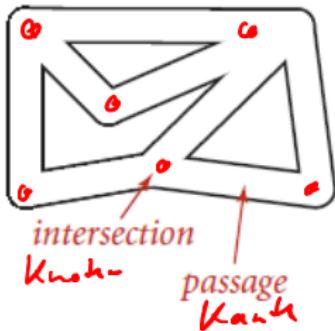
## Ziel der Algorithmen:

- ▶ Besuche jeden Knoten im Graphen
- ▶ Besuche ausgehend von einem Startknoten alle Knoten, die in dem Graphen über einen Pfad erreichbar sind.
- ▶ Bei 1-maliger Suche werden diejenigen Knoten aus dem Graphen gefunden, die zusammenhängend sind. Existiert noch ein **nicht** besuchter Knoten  $v$ , wird die Suche erneut gestartet mit diesem Knoten  $v$ .

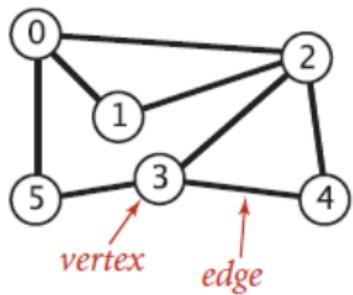


# Analogie: Suche im Labyrinth

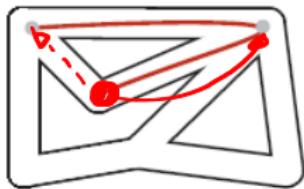
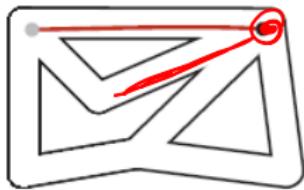
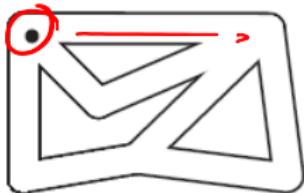
maze



graph



# Analogie: Suche im Labyrinth

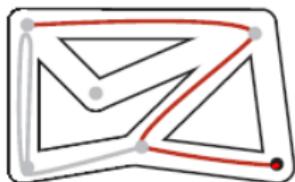
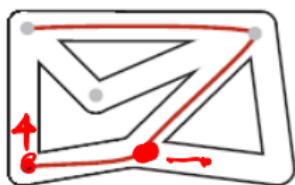
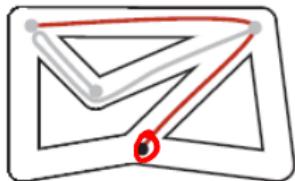


## Methode von Trémaux:

**Ziel:** Erkunde alle Wege eines Labyrinths.

- ▶ Gehe einen beliebigen unmarkierten Weg und rolle einen Faden hinterher.
- ▶ Markiere alle Kreuzungen und Wege, wenn sie das erste Mal betreten werden.
- ▶ Gehe mithilfe des Fadens zurück, wenn man an eine markierte Kreuzung kommt.
- ▶ Gehe noch weiter zurück, wenn man beim Zurückgehen an eine Kreuzung kommt, deren Wege bereits alle besucht wurden.

# Analogie: Suche im Labyrinth



## Methode von Trémaux:

**Ziel:** Erkunde alle Wege eines Labyrinths.

- ▶ Gehe einen beliebigen unmarkierten Weg und rolle einen Faden hinterher.
- ▶ Markiere alle Kreuzungen und Wege, wenn sie das erste Mal betreten werden.
- ▶ Gehe mithilfe des Fadens zurück, wenn man an eine markierte Kreuzung kommt.
- ▶ Gehe noch weiter zurück, wenn man beim Zurückgehen an eine Kreuzung kommt, deren Wege bereits alle besucht wurden.

# Tiefensuche

---

**Annahme:** einfacher, zusammenhängender Graph

## Verfahren der Tiefensuche

1. Markiere alle Knoten  $v \in V$  als nicht besucht.
2. Wähle einen nicht besuchten Knoten  $v \in V$  und markiere diesen als besucht.
3. Besuche (rekursiv) alle benachbarten Knoten  $w$  von  $v$ , die noch nicht markiert sind und führe für diesen Knoten die Tiefensuche durch.

# Rekursive Tiefensuche

## Depth-first search DFS

---

### Algorithm 1: Rekursive Tiefensuche

---

`marked[1..|V|] = false`

**Function** `DFS(G, v)`

*v ist Startknoten*

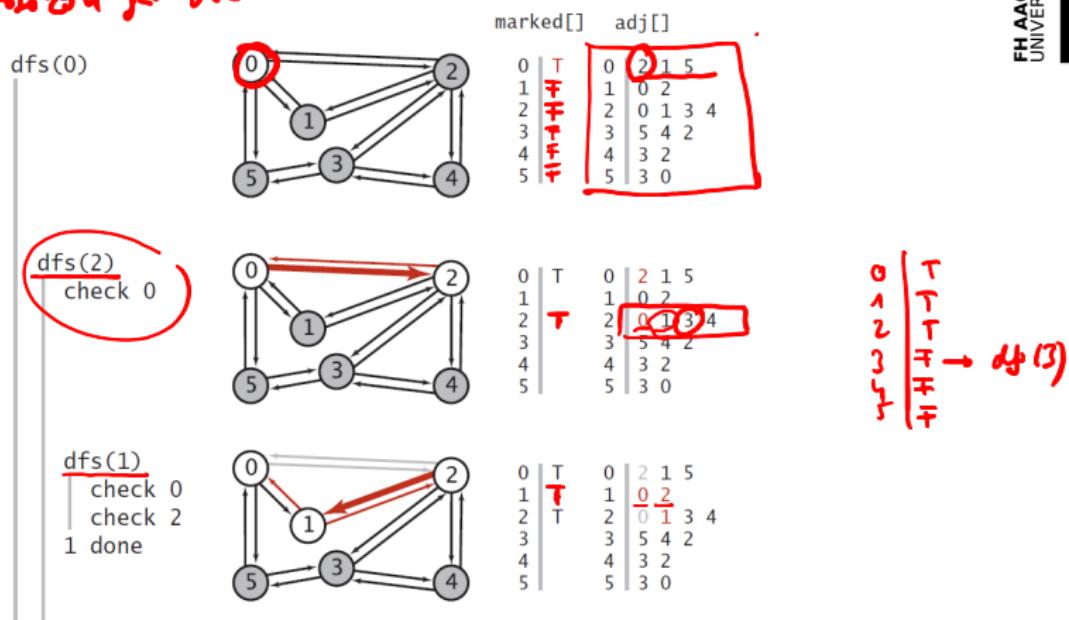
```
  | marked[v] = true
  | for  $\forall w \in G.\text{adj}(v)$  do
  |   | if marked(w) == false then
  |   |   | DFS(G, w)
  |   end
  end
```

---

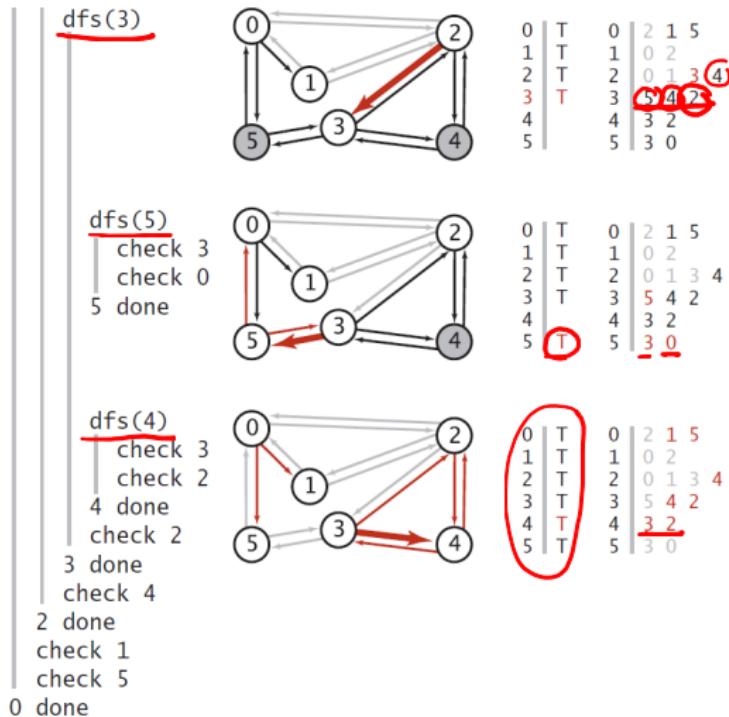
# Ablaufprotokoll Rekursive Tiefensuche (TS)

Startknoten ist  $v = 0$ .

*re. Ablaufprotokoll für die  
re. TS*



# Ablaufprotokoll Rekursive Tiefensuche



# Iterative Tiefensuche

---

## Algorithm 2: Iterative Tiefensuche

---

```
marked[1..|V|] = false
```

```
edgeTo[1..|V|] = 0
```

```
Function DFS_it( $G, s$ )
```

```
    stack  $st$ 
```

```
     $st.push(s)$ 
```

```
    marked[s] = true
```

```
    while ( $!st.isEmpty()$ ) do
```

```
         $v = st.pop()$ 
```

```
        for  $\forall w \in G.adj(v)$  do
```

```
            if ( $marked[w] == false$ ) then
```

```
                 $edgeTo[w] = v$ 
```

```
                marked[w] = true
```

```
                 $st.push(w)$ 
```

```
        end
```

```
    end
```

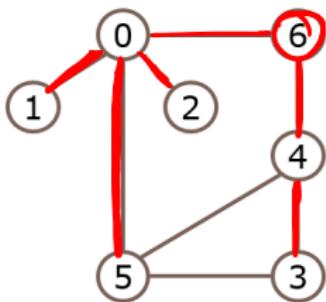
```
end
```

---

~~edge to~~ | 0 1 2 3 4 5 6

# Ablaufprotokoll Iterative Tiefensuche

"Preorder"



Adjazenzliste:

- 0 -> 5 - 2 - 1 - 6
- 1 -> 0
- 2 -> 0
- 3 - 4 - 5
- 4 - [6 - 5 - 3]
- 5 - 0 - 4 - 3
- 6 - 0 - 4

		LIFO							
		Stack	6	4	3	1	2	5	Zustände
0	5	6	1	1	1	1	1	1	
0	5	2	2	2	2	2	2	2	
0	5	5	5	5	5	5	5	5	
0	6	4	4	3	1	2	5		
0	6	4	4	3	1	2	5		
0	6	4	4	3	1	2	5		
0	6	4	4	3	1	2	5		
0	6	4	4	3	1	2	5		

Ablauf protokoll

DFS\_it(0)

0 → push(0) in Stack

0 ← pop() vom Stack

[0] Element verarbeiten

6 →

4 →

5 →

2 →

1 →

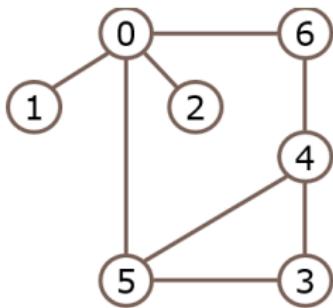
6 →

3 →

3 ←

Stapel leer

# Ablaufprotokoll Iterative Tiefensuche



Adjazenzliste:

**0** - 5 - 2 - 1 - 6

**1** - 0

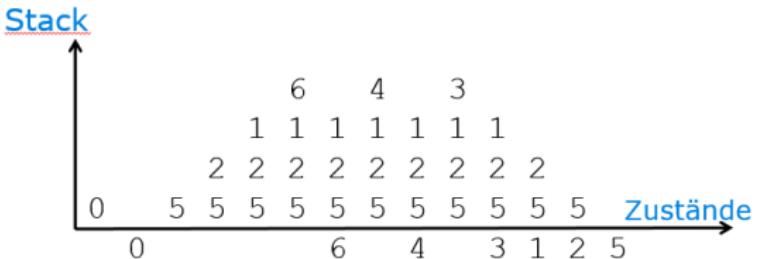
**2** - 0

**3** - 4 - 5

**4** - 6 - 5 - 3

**5** - 0 - 4 - 3

**6** - 0 - 4

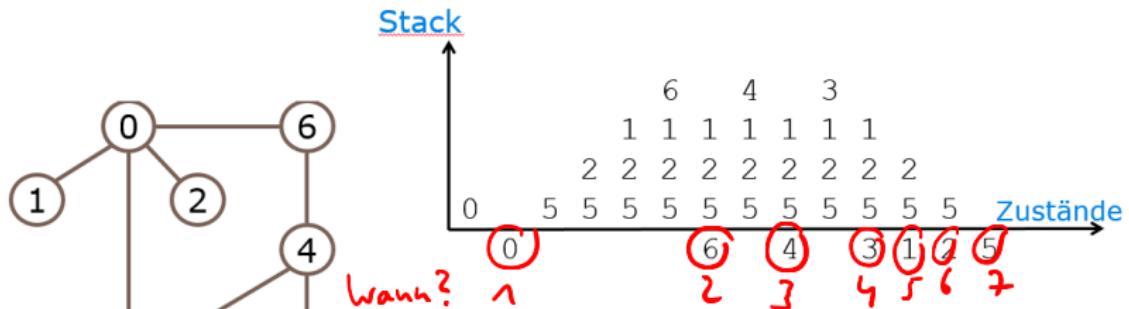


Knoten	0	1	2	3	4	5	6	bearbeitet
marked[i]	0	0	0	0	0	0	0	init
marked[i]	1	-1	-1	0	0	-1	-1	Knoten 0
marked[i]	1	-1	-1	0	-1	-1	2	Knoten 6
marked[i]	1	-1	-1	-1	3	-1	2	Knoten 4
marked[i]	1	-1	-1	4	3	-1	2	Knoten 3
marked[i]	1	5	-1	4	3	-1	2	Knoten 1
marked[i]	1	5	6	4	3	-1	2	Knoten 2
marked[i]	1	5	6	4	3	7	2	Knoten 5

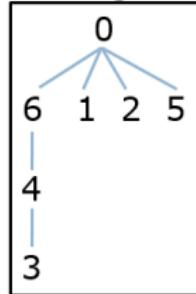
-1 = noch nicht bearbeitet aber im Stack

Knoten 3 wurde als 4. für Knoten besucht

# Ablaufprotokoll Iterative Tiefensuche



Verarbeitungsreihenfolge: 0 6 4 3 1 2 5  
 Suchbaum:



## Adjazenzliste:

**0** - 5 - 2 - 1 - 6

**1** - 0

**2** - 0

**3** - 4 - 5

**4** - 6 - 5 - 3

**5** - 0 - 4 - 3

**6** - 0 - 4

# Fragen

Probleme, die sich mit der Tiefensuche lösen lassen:

1. **Sind zwei gegebene Knoten verbunden?**

## Pfadsuche mit einem Startknoten

Gibt es einen Pfad von einem Startknoten  $s$  zu einem gegebenen Zielknoten  $v$ ? Ein solcher Pfad kann mit einer modifizierten Tiefensuche gefunden werden.

2. **Aus wie vielen Zusammenhangskomponenten (CC - connected components) besteht der Graph?**

## CC durch wiederholte Tiefensuche

Ist nach einem Durchlauf der rekursiven Tiefensuche ein Knoten  $v \in V$  noch nicht besucht, wiederhole die rekursive Tiefensuche mit dem Startknoten  $v$ . Die Anzahl der CC ist gleich der Anzahl wieoft die Tiefensuche gestartet werden musste.

# Modifizierte Tiefensuche zur Pfadsuche

---

## Algorithm 3: Modifizierte Rekursive Tiefensuche

---

```
marked[1..|V|] = false
/* edgeTo speichert letzten Knoten auf Pfad zu diesem Knoten */
edgeTo[1..|V|] = 0
s = v0 /* s ist Startknoten */                      */
DFS(G,s) /* Starten der Pfadsuche */                  */
*/                                                     */
*/                                                     */

Function DFS(G, v)
    marked[v] = true
    for ∀w ∈ G.adj(v) do
        if marked(w) == false then
            edgeTo[w] = v
            DFS(G, w)
    end
end
```

---

# Pfadsuche mit DFS

---

## Algorithm 4: Pfadsuche mit Tiefensuche

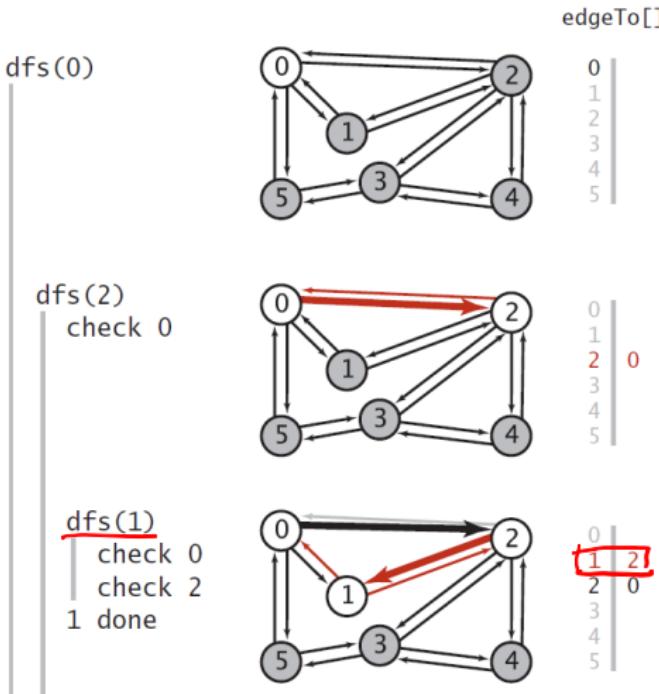
---

```
path[1..k] = Function pathTo(G, s, v)
    if (marked[v] == false) then
        | return null
        stack path /* es existiert kein Pfad von s nach v */
    for (x = v; x ≠ s; x = edgeTo[x]) do
        | path.push(x)
    end
    path.push(s)
    return path
end
```

*pfad = { 5, ③ ②, ①, ⑩, 6, 7 }*

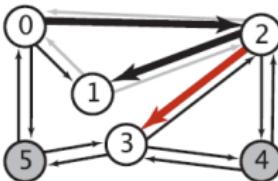
*pathTo(G, 3, 10)*

# Ablaufprotokoll Pfadsuche



# Ablaufprotokoll Pfadsuche

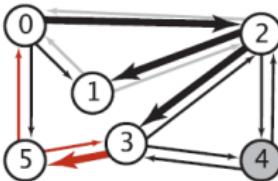
dfs(3)



0	2
1	0
2	0
3	2
4	
5	

dfs(5)

check 3  
check 0  
5 done



0	2
1	0
2	0
3	2
4	
5	3

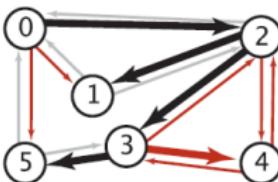
dfs(4)

check 3  
check 2  
4 done  
check 2

3 done  
check 4

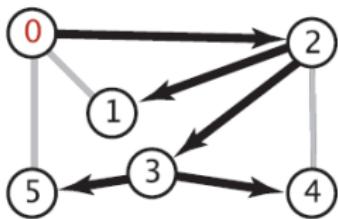
2 done  
check 1

check 5  
0 done



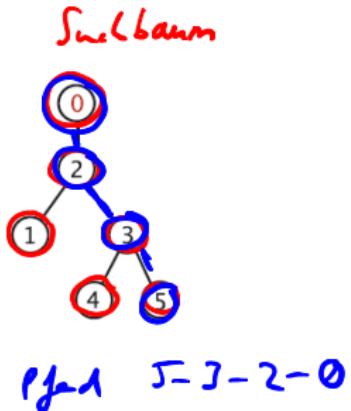
0	2
1	0
2	0
3	2
4	3
5	3

# Pfadsuche mit DFS



x	path
5	5
3	3 5
2	2 3 5
0	0 2 3 5

edgeTo[]
0
1
2
3
4
5



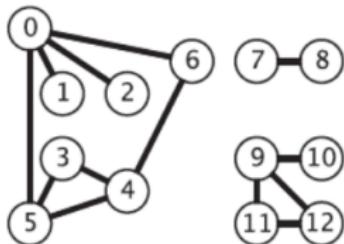
## Ablaufprotokoll der $pathTo(5)$ -Berechnung mit Startknoten 0.

Die Tiefensuche fügt die Kanten  $0-2, 2-1, 2-3, 3-5, 3-4$  in genau dieser Reihenfolge dem Array  $edgeTo[]$  hinzu. Die Kanten bilden einen Baum mit dem Startknoten als Wurzel und liefert die Pfade von 0 zu 1, 2, 3, 4, 5.

# Beispiel Connected Components

## Definition (Zusammenhangskomponente - *Connected Components*)

Ein Graph ist **zusammenhängend**, wenn es von jedem Knoten aus einen Pfad zu allen anderen Knoten im Graphen gibt. Ein **nicht zusammenhängender** Graph besteht aus einer Menge von **Zusammenhangskomponenten**, dh. aus mehreren zusammenhängenden Teilgraphen.



- ▶ Graph besteht aus 3 zusammenhängenden Teilgraphen.
- ▶ Knotenmengen der Zusammenhangskomponenten:  
Komponente 0: {6, 5, 4, 3, 2, 1, 0}  
Komponente 1: {8, 7}  
Komponente 2: {12, 11, 10, 9}

$v$	0	1	2	3	4	5	6	7	8	9	10	11	12
$id[v]$	0	0	0	0	0	0	0	1	1	2	2	2	2

# Connected Components - CC mit DFS

---

## Algorithm 5: CC mit Tiefensuche

---

`marked[1..|V|] = false`

→ `id[1..|V|] = 0`

+ `count = -1`

### Function $CC(G)$

```

    /* Knotennamen  $s \in V$  mit
        $s \in \{0, 1, 2, \dots, |V|-1\}$  */
    for ( $s = 0; s < |V|; s++$ ) do
        if (!marked[s]) then
            dfs( $G, s$ )
            count ++
    end
end

```

↗️ LK TS marken mit  $s$

↑ Wir auf welche TS gestartet

---



---

## Algorithm 6: Modifizierte Rekursive Tiefensuche

---

### Function $DFS(G, v)$

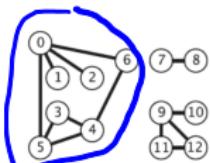
```

    marked[v] = true
    id[v] = count
    for  $\forall w \in G.adj(v)$  do
        if (marked[w] == false)
            then
                |  $DFS(G, w)$ 
        end
    end

```

---

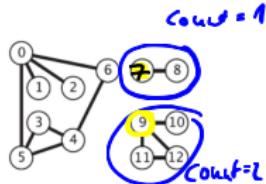
# Ablaufprotokoll CC mit DFS



count	marked[]												id[]													
	0	1	2	3	4	5	6	7	8	9	10	11	12	0	1	2	3	4	5	6	7	8	9	10	11	12
dfs(0)	T												0													
dfs(6)		T											0													
check 0													0													
dfs(4)			T										0													
dfs(5)				T									0													
dfs(3)					T								0													
check 5						T							0													
check 4							T						0													
3 done								T					0													
check 0									T				0													
5 done										T			0													
check 6											T		0													
check 3												T	0													
4 done													T													
6 done														T												
dfs(2)														T												
check 0															T											
2 done																T										
dfs(1)																T										
check 0																	T									
1 done																		T								
check 5																			T							
0 done																				T						

1 zusammenhängender Teilgraph

# Ablaufprotokoll CC mit DFS



*Kennst du mit  
Count = 1*

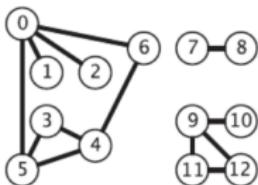
*Kennst du mit  
Count = 2*

```

dfs(7)
| dfs(8)
|   check 7
8 done
7 done
dfs(9)
| dfs(11)
|   check 9
|   dfs(12)
|     check 11
|     check 9
12 done
11 done
dfs(10)
|   check 9
10 done
check 12
9 done
  
```

count	marked[]												id[]													
	0	1	2	3	4	5	6	7	8	9	10	11	12	0	1	2	3	4	5	6	7	8	9	10	11	12
1	T	T	T	T	T	T	T	T	T				0	0	0	0	0	0	0	1						
1	T	T	T	T	T	T	T	T	T				0	0	0	0	0	0	0	1	1					
2	T	T	T	T	T	T	T	T	T	T			0	0	0	0	0	0	0	1	1	2				
2	T	T	T	T	T	T	T	T	T	T			0	0	0	0	0	0	0	1	1	2				
2	T	T	T	T	T	T	T	T	T	T			T		0	0	0	0	0	0	1	1	2			
2	T	T	T	T	T	T	T	T	T	T			T	T		0	0	0	0	0	0	1	1	2	2	2
2	T	T	T	T	T	T	T	T	T	T			T	T		0	0	0	0	0	0	1	1	2	2	2

# Instanzmethoden Connected Components



- ▶ Knotenmengen der Zusammenhangskomponenten:  
 Komponente 0: {6, 5, 4, 3, 2, 1, 0}  
 Komponente 1: {8, 7}  
 Komponente 2: {12, 11, 10, 9}

$v$	0	1	2	3	4	5	6	7	8	9	10	11	12
$id[v]$	0	0	0	0	0	0	0	1	1	2	2	2	2

*Count = 2 ?*

## Instanzmethoden:

- ▶ Ist Knoten  $v$  mit Knoten  $w$  verbunden?

```
bool connected(int v, int w) {
    return id[v] == id[w];
}
```

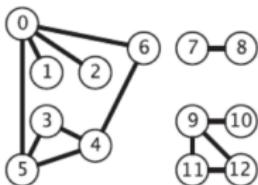
- ▶ Zu welcher Zusammenhangskomponente gehört der Knoten  $v$ ?

```
int id(int v) { return id[v]; }
```

- ▶ Aus wie vielen Zusammenhangskomponenten besteht der Graph?

```
int count() { return count+1; }
```

# Instanzmethoden Connected Components



- ▶ Knotenmengen der Zusammenhangskomponenten:  
Komponente 0: {6, 5, 4, 3, 2, 1, 0}  
Komponente 1: {8, 7}  
Komponente 2: {12, 11, 10, 9}

$v$	0	1	2	3	4	5	6	7	8	9	10	11	12
$id[v]$	0	0	0	0	0	0	0	1	1	2	2	2	2

## Instanzmethoden:

- ▶ Ist Knoten  $v$  mit Knoten  $w$  verbunden?

```
bool connected(int v, int w) {  
    return id[v] == id[w]; }
```

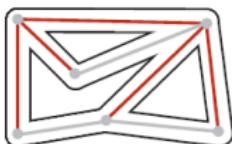
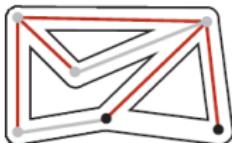
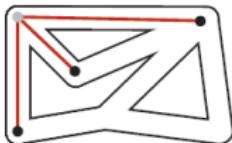
- ▶ Zu welcher Zusammenhangskomponente gehört der Knoten  $v$ ?

```
int id(int v) { return id[v]; }
```

- ▶ Aus wie vielen Zusammenhangskomponenten besteht der Graph?

```
int count() { return count+1; }
```

# Breitensuche



Erkundung des Labyrinths mit der Breitensuche

## Kürzeste-Pfade-Problem

Gibt es einen Pfad vom Startknoten  $s$  zu einem gegebenen Zielknoten  $v$ ? Wenn ja, dann finde den kürzesten Pfad (d.h. denjenigen Pfad mit den wenigsten Kanten).

### Kürzeste-Pfad-Suche von $s$ nach $v$ :

- ▶ Beginne bei  $s$  und prüfe für alle adjazenten Knoten, ob sie gleich  $v$  sind.
- ▶ Prüfe dann alle adjazenten Knoten, die über eine Kante erreichbar sind.
- ▶ usw.

Breitensuche entspricht einer Gruppe von Personen im Labyrinth, die in alle Richtungen ausschwärmen.

# Algorithmus Iterative Breitensuche

---

## Algorithm 7: Iterative Tie-fensuche

---

```

marked[1..|V|] = false
edgeTo[1..|V|] = 0
Function DFS_it(G, s)
    stack st
    st.push(s)
    marked[s] = true
    while (!st.isEmpty()) do
        v = st.pop()
        for  $\forall w \in G.\text{adj}(v)$  do
            if (marked[w] == false)
                then
                    edgeTo[w] = v
                    marked[w] = true
                    st.push(w)
    end
end

```

---

## Algorithm 8: Iterative Breitensuche

---

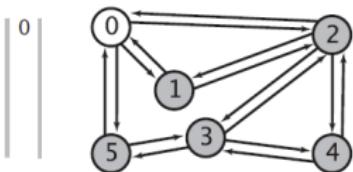
```

marked[1..|V|] = false
edgeTo[1..|V|] = 0
Function BFS(G, s)
    queue q
    q.enqueue(s)
    marked[s] = true
    while (!q.isEmpty()) do
        v = q.dequeue()
        for  $\forall w \in G.\text{adj}(v)$  do
            if (marked[w] == false)
                then
                    edgeTo[w] = v
                    marked[w] = true
                    q.enqueue(w)
    end
end

```

# Ablaufprotokoll BFS

queue



marked []

0	T
1	
2	
3	
4	
5	

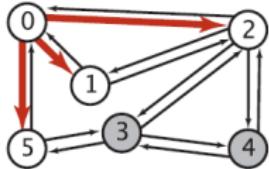
edgeTo []

0	
1	
2	
3	
4	
5	

adj []

0	2	1	5
1	0	2	
2	0	1	3
3	5	4	2
4	3	2	
5	3	0	

2
1
5

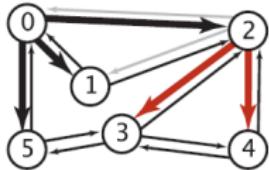


0	T
1	T
2	T
3	
4	
5	T

0	
1	
2	
3	
4	
5	

0	2	1	5
1	0	2	
2	0	1	3
3	5	4	2
4	3	2	
5	3	0	

1
5
3
4

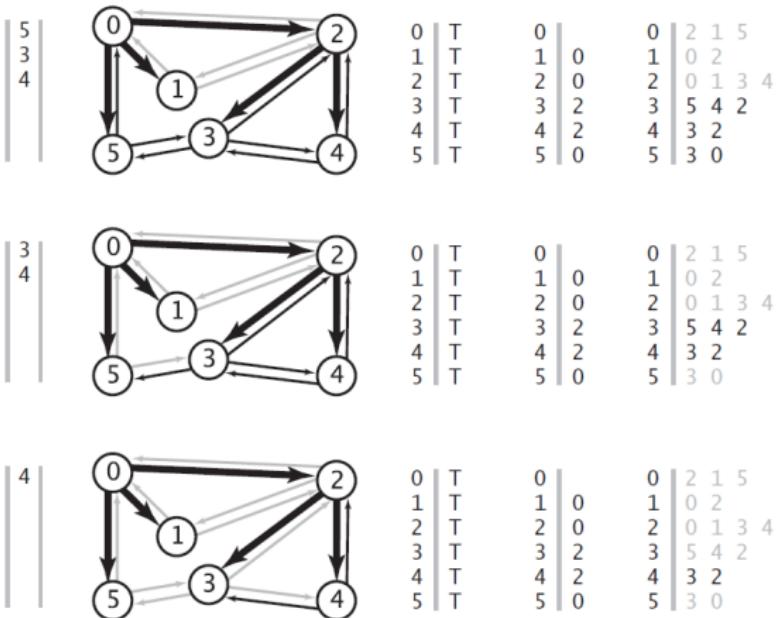


0	T
1	T
2	T
3	T
4	T
5	T

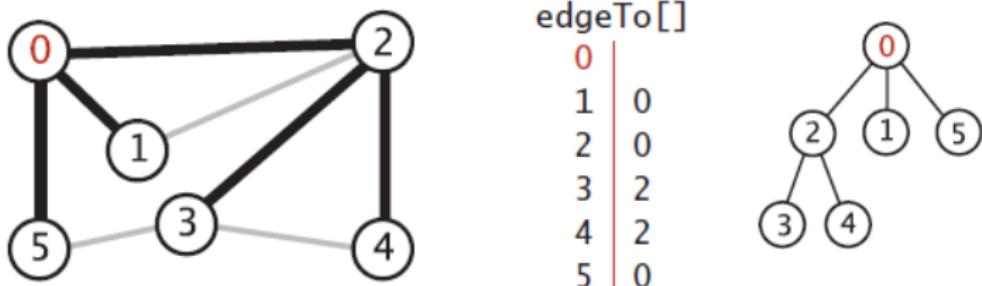
0	
1	
2	
3	
4	
5	

0	2	1	5
1	0	2	
2	0	1	3
3	5	4	2
4	3	2	
5	3	0	

# Ablaufprotokoll BFS



# Ergebnis Ablaufprotokoll BFS



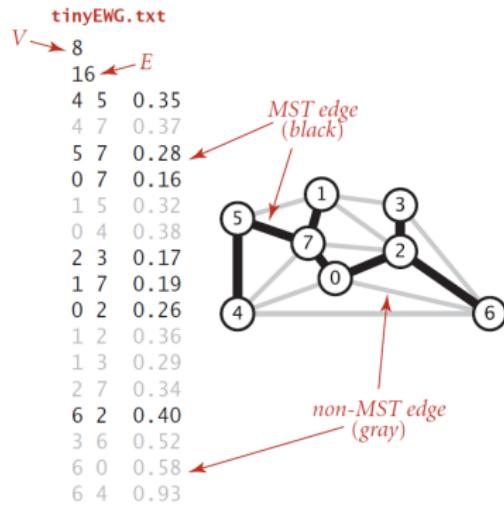
## Fazit: Breitensuche

Die Breitensuche liefert für alle verbundenen Knoten die kürzesten Pfade. Der Suchbaum hat somit die geringste Höhe.

# Minimaler Spannbaum - MST

Gegeben sei ein **ungerichteter und gewichteter Graph**.

**Ziel:** Finde einen minimalen Spannbaum (*minimum spanning tree - MST*)



## Definition (Minimaler Spannbaum)

Ein **Spannbaum** eines Graphen ist ein zusammenhängender Teilgraph, der keine Zyklen aufweist und alle Knoten umfasst. Der **minimale Spannbaum** (*minimum spanning tree - MST*) eines kantengewichteten Graphen ist ein Spannbaum, dessen Summe seiner Kantengewichte minimal ist gegenüber irgendeines anderen Spannbaums.

# MST Anwendungen

Anwendung	Knoten	Kante
Schaltkreis	Komponente	Draht
Fluglinie	Flughafen	Flugroute
Stromverteilung	Kraftwerk	Übertragungsleitungen
Bilderkennung	Merkmale	Abstandsbeziehung

# MST Annahmen

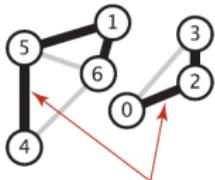
---

Folgenden Bedingungen gelten für den Graphen:

- ▶ Der Graph  $G$  sei zusammenhängend.
- ▶ Kantengewichte können Abstände, Zeit oder auch andere Kosten repräsentieren.
- ▶ Kantengewichte können Null oder auch negativ sein.
- ▶ Die Kantengewichte seien alle verschieden. Bei gleichen Kantengewichten kann es zu nicht eindeutigen Minimalen Spannbäumen kommen.

# MST Annahmen

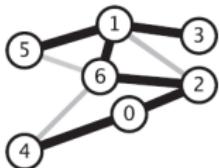
no MST if graph is not connected



*can independently compute  
MSTs of components*

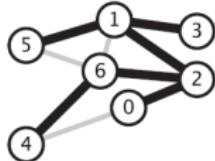
4 5	0.61
4 6	0.62
5 6	0.88
1 5	0.11
2 3	0.35
0 3	0.6
1 6	0.10
0 2	0.22

weights can be 0 or negative



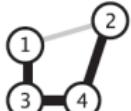
4 6	0.62
5 6	0.88
1 5	0.02
0 4	-0.99
1 6	0
0 2	0.22
1 2	0.50
1 3	0.97
2 6	-0.17

weights need not be  
proportional to distance



4 6	0.62
5 6	0.88
1 5	0.02
0 4	0.64
1 6	0.90
0 2	0.22
1 2	0.50
1 3	0.97
2 6	-0.17

MST may not be unique  
when weights have equal values



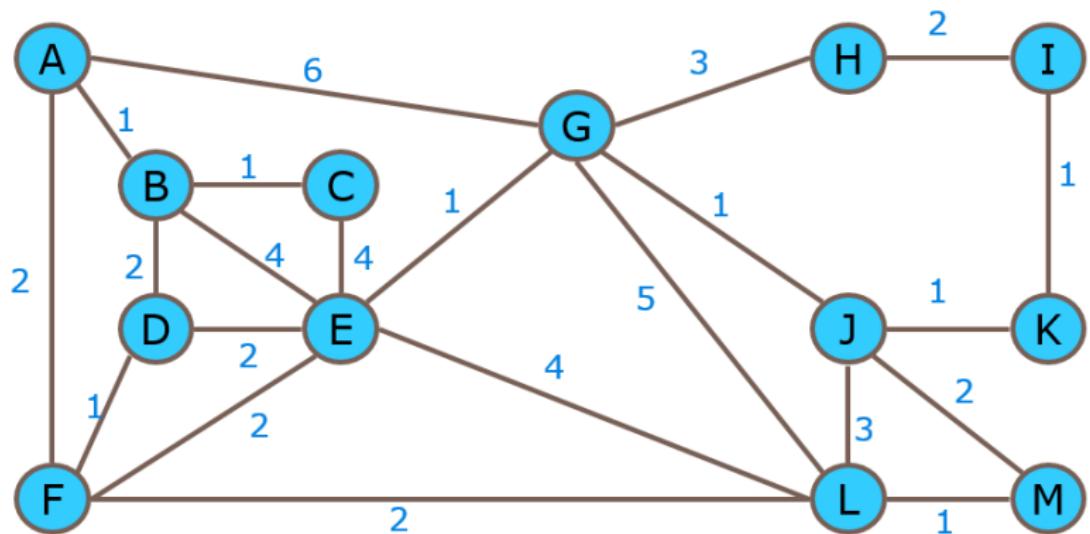
1 2	1.00
1 3	0.50
2 4	1.00
3 4	0.50



1 2	1.00
1 3	0.50
2 4	1.00
3 4	0.50

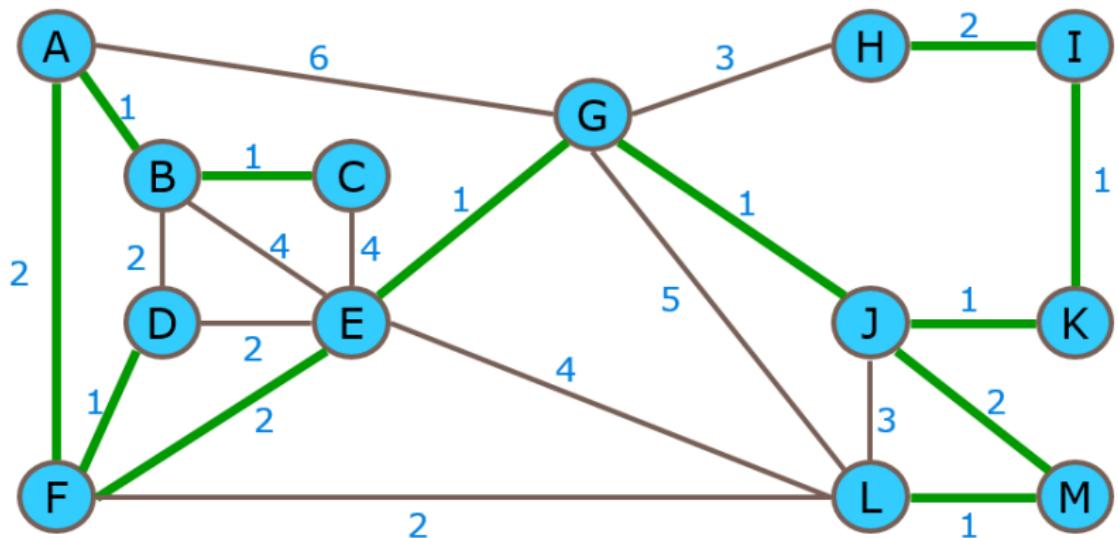
# Beispiel MST

Gegeben: Gewichteter Graph G mit Gewichten  $w > 0$ , wobei mehrere Kanten auch das gleiche Gewicht haben können.



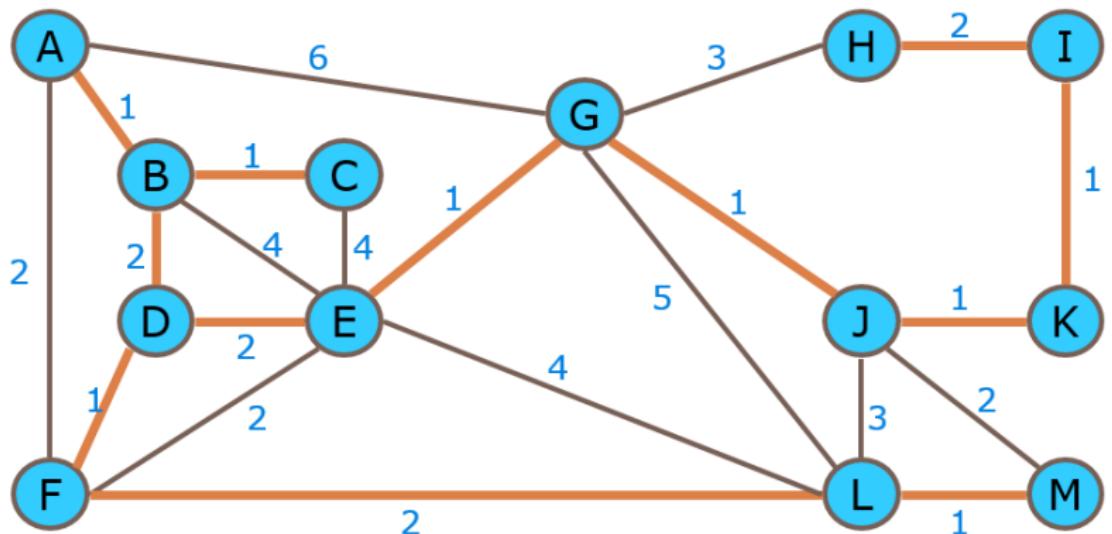
# Beispiel MST

**Beispiel 1:** Minimaler Spannbaum zum Graphen G mit Gesamtkosten = 16:



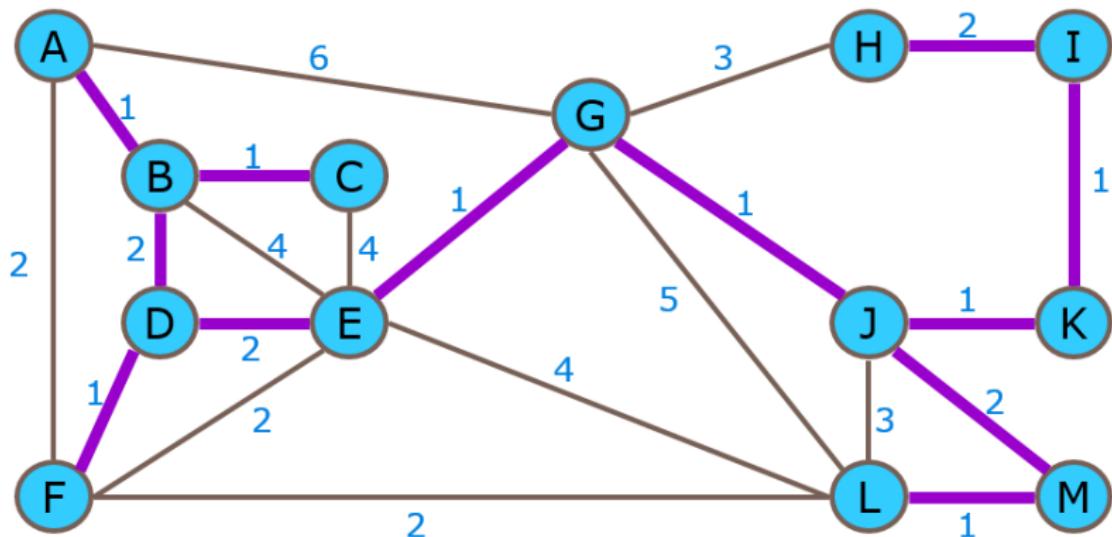
# Beispiel MST

**Beispiel 2:** Minimaler Spannbaum zum Graphen G mit Gesamtkosten = 16:



# Beispiel MST

**Beispiel 3:** Minimaler Spannbaum zum Graphen G mit Gesamtkosten = 16:



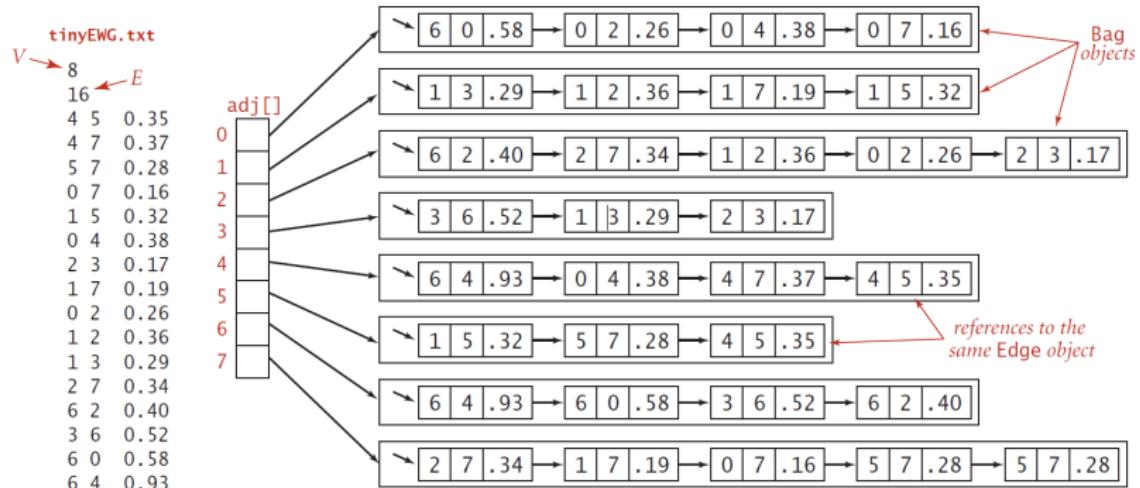
# API für gewichtete Kante

```
class Edge {  
private:  
    int either;      // ein Knoten der Kante  
    int other;       // der andere Knoten der Kante  
    double weight;   // Kantengewicht  
public:  
    Edge(int v, int w, double weight); // Konstruktor  
    double weight();                 // Gewicht dieser Kante  
    int either();                   // einer der beiden Knoten  
    int other();                    // der andere Knoten  
    int compareTo(Edge e);          // vergleicht Kante mit e  
    string toString();              // String-Repäsentation  
};
```

# API für kantengewichteten Graphen

```
class EdgeWeightedGraph {  
private:  
    int V;    // Anzahl Knoten von G  
    int E;    // Anzahl Kanten von G  
    ...  
public:  
    EdgeWeightedGraph(int V); // Erzeugt leeren Graphen mit V Knoten  
    EdgeWeightedGraph(iostream in); // Graph einlesen aus Eingabe  
    int V();           // liefert Anzahl Knoten  
    int E();           // liefert Anzahl der Kanten  
    void add(Edge e); // fügt Kante e dem Graphen hinzu  
    vector<Edge> adj(int v); // liefert Array der adjazenten Kanten  
    vector<Edge> edges();   // alle Kanten dieses Graphen  
    string toString();  
};
```

# Kantengewichtete Graphenrepräsentation

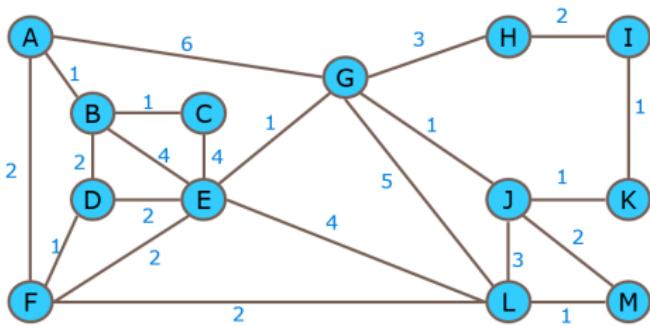


# MST-Algorithmen

---

1. Algorithmus von Prim
2. Algorithmus von Kruskal

# Prim Algorithmus



## Algorithmus von Prim

- ▶ Wähle einen beliebigen Startknoten für den MST.
- ▶ Füge aus der Menge der adjazenten Kanten zum Baum diejenige Kante hinzu, die den Baum mit den minimalen Kosten erweitert.
- ▶ Nach  $|V| - 1$  Kanten ist der MST gefunden.

# DS für Prim Algorithmus

---

Datenstrukturen für den Prim Algorithmus:

- ▶ **Knoten im Baum:** knotenindiziertes boolesches Array `marked[]`, wobei `marked[v]` den Wert `true` hat, wenn  $v$  im Baum liegt.
- ▶ **Kanten im Baum:** Verwende dazu zwei DS:
  - (a) Warteschlange `mst`, um die Kanten des minimalen Spannbaums zu speichern
  - (b) knotenindiziertes Array `edgeTo` für `Edge`-Objekte, wobei `edgeTo[v]` das `Edge`-Objekt ist, das  $v$  mit dem Baum verbindet.
- ▶ **Randkanten:** Randkanten sind die Menge der adjazenten Kanten, die den Baum um einen nicht besuchten Knoten erweitern. Dafür wird eine Prioritätswarteschlange verwendet `MinPQ<Edge>`.

# Prioritätssuche

Zustände der Priority Queue

	BC1											
	BD2	D2				M1						
	AB1	AF2	F2	F1	L2	E2	E2					
	AF2	BE4	E4	E2	E2	J3	J2	G1	J1	K1	I1	
A*	AG6	AG6	G6	G6	G6	G5	G5	J2	H3	H3	H3	H2

AB	BC	BD	DF	FL	LM	DE	EG	GJ	JK	KI	IH
1	1	2	1	2	1	2	1	1	1	1	2

Kanten des minimalen Spannbaumes



Kosten

# Prim API

```
#include <queue>
class PrimMST {
private:
    vector<bool> marked;          // MST-Knoten
    queue<Edge> mst;             // MST-Kanten
    priority_queue<Edge> pq;      // Menge der Randkanten in PQ
public:
    PrimMST(EdgeWeightedGraph G);
    void visit(EdgeWeightedGraph G, int v);
    queue<Edge> edges() {
        return mst;
    }
    double weight() {
    }
};
```

# Prim Implementierung

```
PrimMST(EdgeWeightedGraph G) {  
    visit(G, 0); // setzt voraus, dass G zusammenhängend ist  
    while (!pq.isEmpty()) {  
        Edge e = pq.top(); // Hole Kante mit geringstem Gewicht  
        pq.pop();          // aus PQ  
        int v = e.either();  
        int w = e.other();  
  
        // Überspringen, falls beide Knoten im Baum markiert sind  
        if (marked[v] && marked[w])) continue;  
  
        mst.push(e);           // Füge Kante e zum Baum hinz  
        if (!marked[v]) visit(G, v); // Knoten v oder w zum Baum  
        if (!marked[w]) visit(G, w); // hinzufügen  
    }  
}
```

# Prim Implementierung

---

```
void visit(EdgeWeightedGraph G, int v) {
    // Markiert v und legt alle Kanten von v zu unmarkierten
    marked[v] = true;
    vector<Edge> edges = G.adj(v);
    for(int i = 0; i < edges.size(); i++){
        if (!marked[edges[i].other()]) pq.push(edges[i]);
    }
}
```

# Aufwandsanalyse Prim Algorithmus

## Aufwand Prim Algorithmus

Die Prioritätssuche benötigt Speicher proportional zu  $E$  und eine Worst Case Gesamtzeit:

$$\sim O(E \cdot \log E)$$

um den minimalen Spannbaum eines zusammenhängenden kantengewichteten Graphen mit  $E$  Kanten und  $V$  Knoten zu berechnen.

**Beweis:** Engpass in diesem Algorithmus ist die Anzahl der Vergleiche der Kantengewichte innerhalb der Prioritätswarteschlange (PQ). Hier können maximal  $E$  Kanten als MinHeap gespeichert sein. Die Kosten für das Einfügen und die Entnahme sind dann  $\sim \log E$ . Es werden höchsten  $E$  Kanten eingefügt, ergibt dies einen Gesamtaufwand von  $\sim E \cdot \log E$ .

# Algorithmus von Kruskal

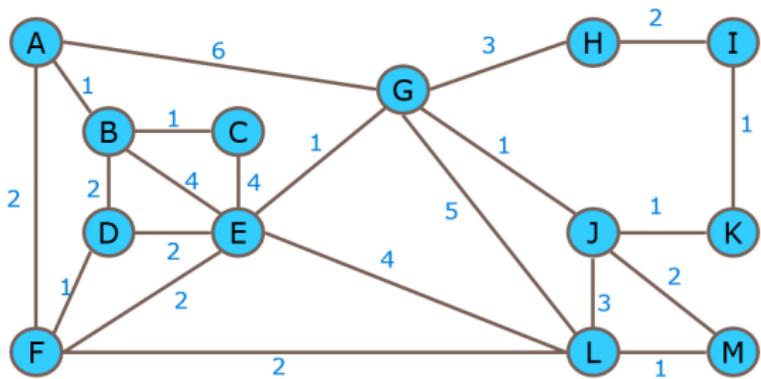
## Algorithmus von Kruskal

**Idee:** Jeder Knoten ist zu Beginn ein Teilgraph. Suche diejenige Kante, die mit minimalen Kosten 2 Teilgraphen verbindet und dabei kein Zykel erzeugt.

- ▶ Sortiere die Kanten nach ihren Gewichten.
- ▶ Verarbeite die Kanten in der Reihenfolge ihrer Gewichte (vom kleinsten zum größten).
- ▶ Nehme immer eine Kanten zum MST hinzu, wenn dadurch kein Zykel im Baum entsteht.
- ▶ Nach  $V - 1$  Kanten ist der MST gefunden.

Der Graph mit  $V$  Knoten besteht zuerst aus  $V$  Bäumen mit der Höhe 0. In  $V-1$  Schritten werden jeweils 2 Bäume unter Verwendung der kürzesten möglichen Kante miteinander kombiniert bis nur noch 1 Baum übrig bleibt. Dies ist dann der minimale Spannbaum des Graphen.

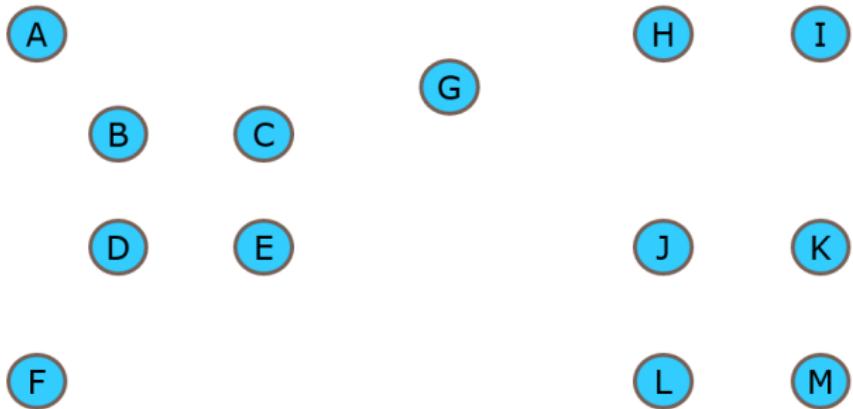
# Beispiel Kruskal Algorithmus



Erzeuge die Kantenlisten nach ihren Gewichten:

Gewicht	Kantenlisten
1	AB, BC, DF, EG, IK, LM, GJ, IK
2	AF, BD, EF, FL, HI, JM
3	GH, JL
4	...

# Beispiel Kruskal Algorithmus



Gewicht	Kantenlisten
1	AB, BC, DF, EG, IK, LM, GJ, IK
2	AF, BD, EF, FL, HI, JM
3	GH, JL
4	...

# Kruskal API

```
class KruskalMST {  
private:  
    queue<Edge> mst;          // MST-Kanten  
    vector<int> marked;        // BaumId zu jedem Knoten  
public:  
    KruskalMST(EdgeWeightedGraph G);  
    queue<Edge> edges(); // liefert MST  
    double weight();       // berechnet Gesamtkosten des MST  
}
```

# Kruskal Implementierung

```
KruskalMST(EdgeWeightedGraph G) {  
    priority_queue<Edge> pq;  
    vector<Edge> e = G.edges(); // liefert alle Kanten von G  
    for (int i = 0; i < e.size(); i++) pq.push(e);  
    for (int i = 0; i < G.V(); i++) marked[i] = i;  
  
    while (!pq.isEmpty()) {  
        Edge e = pq.top();  
        pq.pop();  
        int v = e.either(); int w = e.other();  
        if (marked[v] != marked[w]) {  
            mst.push(e);  
            for (int i = 0; i < G.V(); i++) {  
                if (marked[i] == marked[w])  
                    marked[i] = marked[v];  
            }  
        }  
    }  
}
```

# Aufwandsanalyse Kruskal

## Aufwand Kruskal

Der Algorithmus von Kruskal benötigt Speicher  $\sim E$  und Worst Case Laufzeit

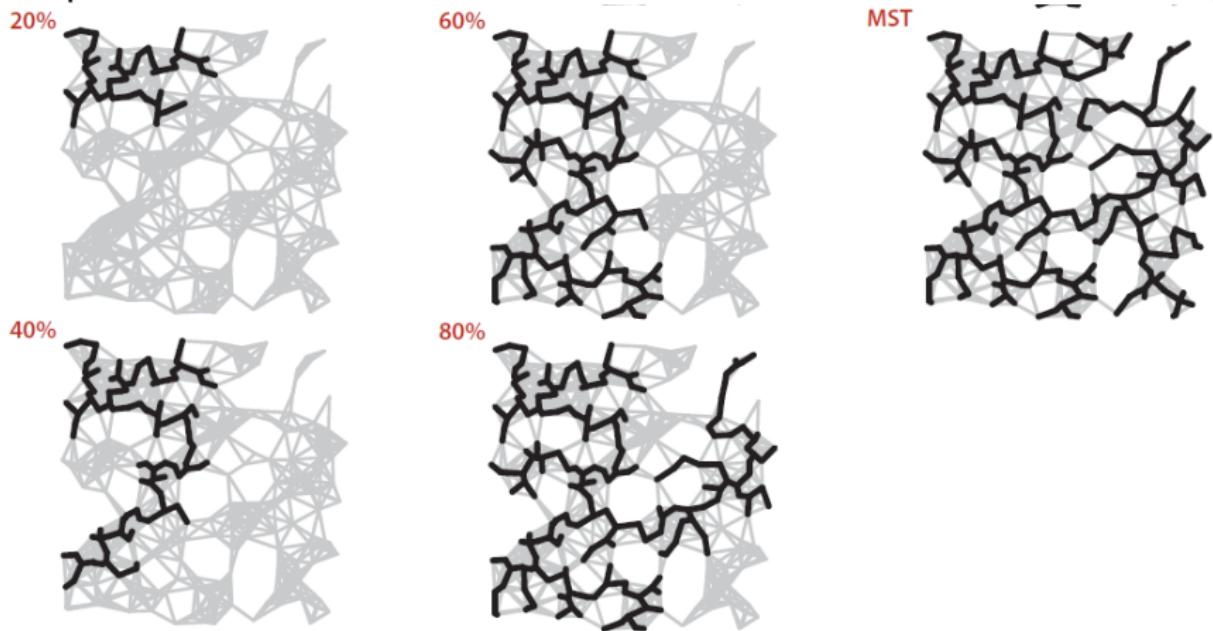
$$\sim O(E \cdot \log E)$$

um den MST eines kantengewichteten zusammenhängenden Graphen mit  $E$  Kanten und  $V$  Knoten zu berechnen.

**Beweis:** Die PQ wird mit allen Kanten initialisiert  $\sim O(E)$ . Maximal werden  $E$  Kanten in der PQ sein. Pro Entnahme werden  $\sim O(\log E)$  Vergleiche benötigt. Insgesamt werden maximal  $E$  Kanten in die PQ eingefügt und  $V - 1$  Bäume miteinander verschmolzen zu einem MST, was der Wachstumsordnung  $E \cdot \log E$  entspricht.

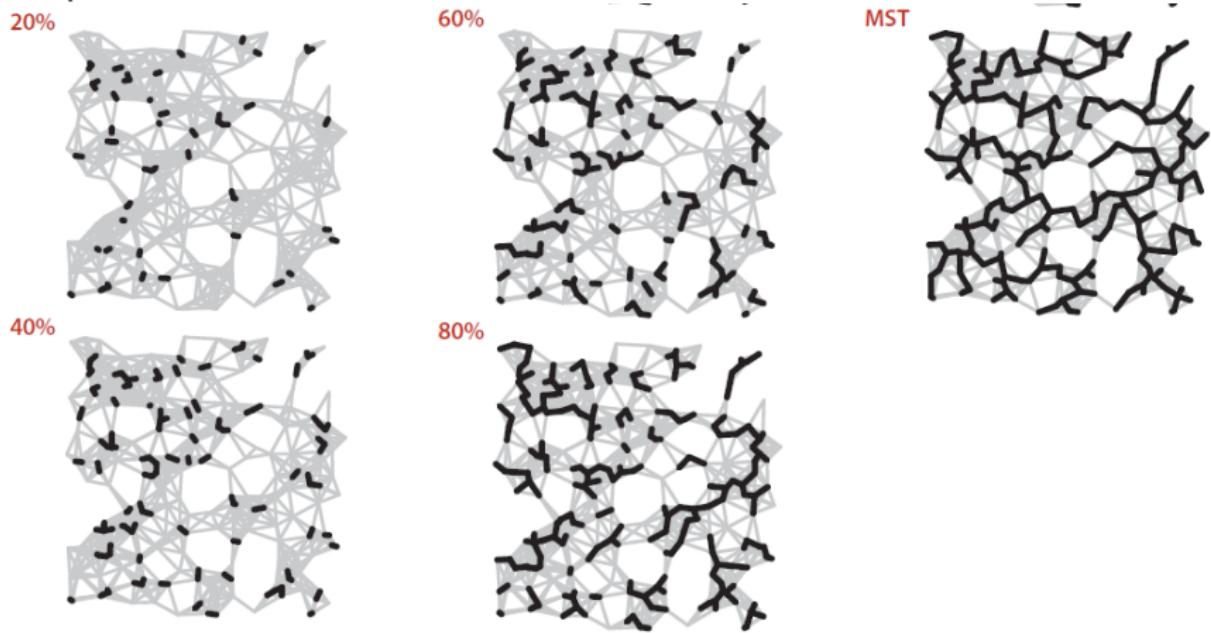
# Beispiel Prim

Graph G mit  $V=250$  Knoten und  $E=1273$  Kanten:



# Beispiel Kruskal

Graph G mit  $V=250$  Knoten und  $E=1273$  Kanten:



# Kürzeste Pfade Algorithmus

## Anwendung Navigationssystem:

Suche kürzeste Route von einem Ort zum Anderen.

Repräsentation durch einen kantengewichteten, gerichteten Graphen (kantengewichteter Digraph):

- ▶ Knoten repräsentieren die Orte,
- ▶ Kanten die Straßen,
- ▶ Gewichte der Kanten modellieren die Kosten (Strecke oder Fahrzeiten),
- ▶ Einbahnstraßen sind gerichtete Kanten.

## Kürzeste-Pfade-Modell

Finde den Weg mit den geringsten Kosten, um von einem Knoten zu einem anderen zu gelangen.

# Definition Gerichteter Graph

## Definition (Gerichteter Graph (Digraph))

Ein *gerichteter Graph (auch Digraph genannt)* besteht aus einer Menge von Knoten und einer Menge von gerichteten Kanten. Jede gerichtete Kante verbindet ein geordnetes Paar von Knoten.

# Beispiel Kürzester Pfad

**Gegeben:** Kantengewichteter Digraph

**Gesucht:** Kürzester Weg vom Startknoten 0 zum Zielknoten 6.

edge-weighted digraph

4->5 0.35

5->4 0.35

4->7 0.37

5->7 0.28

7->5 0.28

5->1 0.32

0->4 0.38

0->2 0.26

7->3 0.39

1->3 0.29

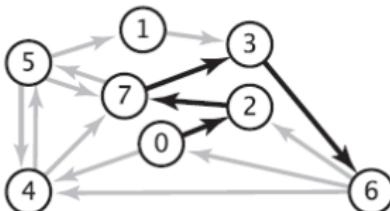
2->7 0.34

6->2 0.40

3->6 0.52

6->0 0.58

6->4 0.93



shortest path from 0 to 6

0->2 0.26

2->7 0.34

7->3 0.39

3->6 0.52

# Definition Kürzester Pfad

## Definition (Kürzester Pfade mit einem Startknoten)

Ein *Kürzester Pfad* von einem Knoten  $s$  zu einem Knoten  $t$  in einem kantengewichteten Digraphen ist ein gerichteter Pfad von  $s$  zu  $t$  mit der Eigenschaft, dass kein anderer Pfad ein niedrigeres Gewicht hat.

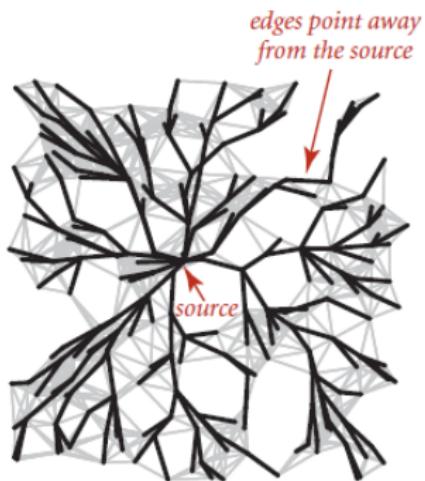
- ▶ API und Implementierung für kantengewichtete Digraphen,
- ▶ Dijkstra-Algorithmus für Graphen mit positiven Gewichten.

# Eigenschaften der kürzesten Pfade

---

- ▶ Pfade sind gerichtet, dh. ein kürzester Pfad muss die Richtung seiner Kanten berücksichtigen.
- ▶ Gewichte sind nicht proportional zu euklidischen Abständen.
- ▶ Nicht alle Knoten sind u.U. erreichbar.
- ▶ Kürzeste Pfade sind nicht notwendigerweise eindeutig.
- ▶ Hier: nur Graphen mit positiven Gewichten  $> 0$  und ohne parallele Kanten.

# Definition Kürzeste Pfade Baum



Gegeben sei ein kantengewichteter Digraph und ein Knoten  $s \in V$ .

## Definition (Kürzeste Pfade Baum)

Ein **Kürzester-Pfade-Baum** für einen Startknoten  $s$  ist ein Teilgraph, der  $s$  und alle von  $s$  aus erreichbaren Knoten enthält, die einen gerichteten Baum mit der Wurzel  $s$  bilden, so dass jeder Baumpfad ein kürzester Pfad im Digraphen ist.

Kürzester Pfad Baum  
mit 250 Knoten

# API für gewichtete, gerichtete Kanten

```
class DirectedEdge{  
private:  
    int v; // erster Knoten der Kanten  
    int w; // zweiter Knoten der Kante  
    double weight; // Gewicht  
public:  
    double weight() {  
        return weight;}  
    int from(){  
        return v;}  
    int to(){  
        return w;}  
};
```

# API für kantengewichtete Digraphen

```
class EdgeWeightedDigraph{
private:
    ...
public:
    EdgeWeightedDigraph(int V); // Leerer Digraph mit V Knoten
    int V(); // Anzahl Knoten
    int E(); // Anzahl Kanten
    void addEdge(DirectedEdge e); // Füge e zum Digraphen hinzu
    vector<DirectedEdge> adj(int v); // Adj. Kanten zu v
    vector<DirectedEdge> edges(); // alle Kanten im Digraphen
};
```

# Implementierung für kantengewichtete Digraphen

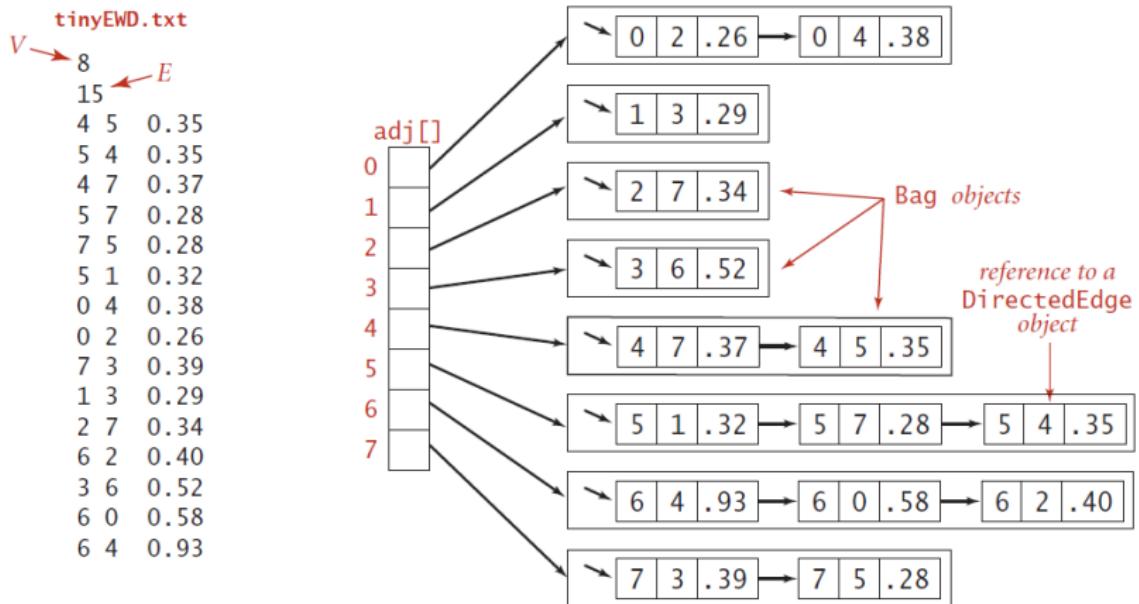
```
class EdgeWeightedDigraph{
private:
    int V; // Anzahl Knoten
    int E; // Anzahl Kanten
    map<int, vector<DirectedEdge> > adjlist; // Adjazenzliste

public:
    EdgeWeightedDigraph(int V) {
        this.V = V;
        this.E = E;
        for (int v = 0; v < V; v++) {
            adjlist[v]; // erzeugt leeren Vektor der adj. Kanten zu
        }
    }
    int V() { return V; }
    int E() { return E; }
    ...
}
```

# Implementierung für kantengewichtete Digraphen

```
void addEdge(DirectedEdge e) {  
    adjlist[e.from()].add(e);  
    E++;  
}  
vector<DirectedEdge> adj(int v) {  
    return adjlist[v];  
}  
vector<DirectedEdge> edges() {  
    vector<DirectedEdge> alledges;  
    vector<DirectedEdge> tmp;  
    for (int v = 0; v < V; v++) {  
        tmp = adjlist[v];  
        for (int i = 0; i < adjlist[v].size(); i++) {  
            alledges.push_back(tmp[i]);  
        }  
    }  
}  
};
```

# Beispiel: Kantengewichteter Digraph

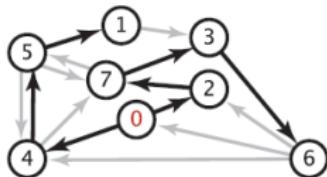


# API Kürzeste Pfade

---

```
// class for shortest paths
class SP {
...
public:
    SP(EdgeWeightedDigraph G, int s); // Konstruktor
    double distTo(int v);           // Abstände von s zu v,
                                    //   = wenn kein Pfad existiert
    bool hasPathTo(int v);          // Gibt es einen Pfad von s zu v
    vector<DirectedEdge> pathTo(int v); // Pfad von s zu v,
                                         //   null, wenn keiner vorhanden
}
```

# Datenstrukturen für kürzeste Pfade



	edgeTo[]	distTo[]
0	null	0
1	5->1	0.32
2	0->2	0.26
3	7->3	0.37
4	0->4	0.38
5	4->5	0.35
6	3->6	0.52
7	2->7	0.34

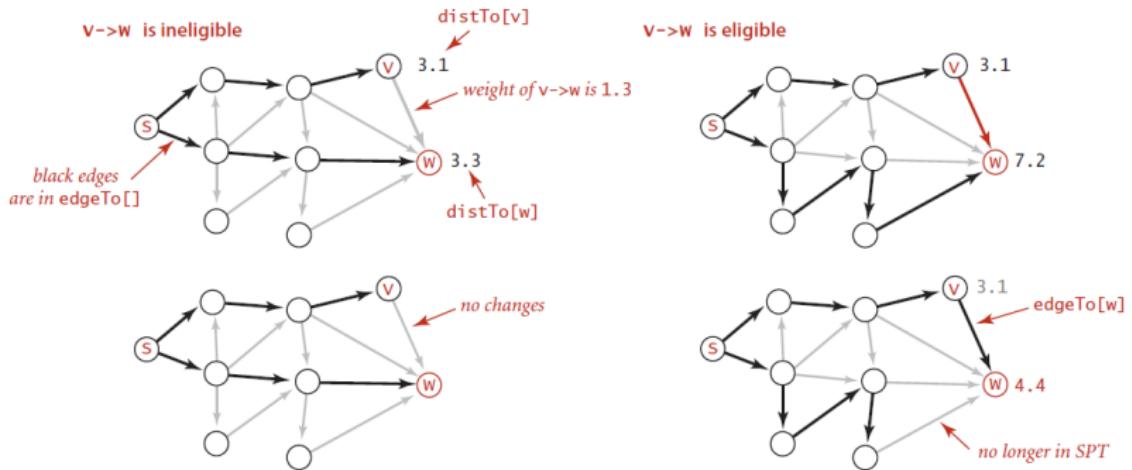
- ▶ **Kanten im Kürzeste-Pfade-Baum:**  
Knotenindiziertes Array  
`edgeTo[v]` zum Speichern der Elternknoten-Kanten-Repräsentation
- ▶ **Abstand zum Startknoten:**  
Knotenindiziertes Array  
`distTo[v]`, das die Länge des kürzesten bekannten Pfad von s nach v ist.  
Initialisierung: `distTo[s]=0`, alle anderen unendlich.

# Kantenrelaxation

Eine Kante  $v \rightarrow w$  wird relaxiert, wenn der kürzeste Weg von  $s$  nach  $w$  über die Kante von  $v$  nach  $w$  verläuft.

```
void relax(DirectedEdge e) {
    int v = e.from();
    int w = e.to();
    if (distTo[w] > distTo[v] + e.weight())
    {
        distTo[w] = distTo[v] + e.weight();
        edgeTo[w] = e;
    }
}
```

# Beispiel Kantenrelaxation (2 Fälle)

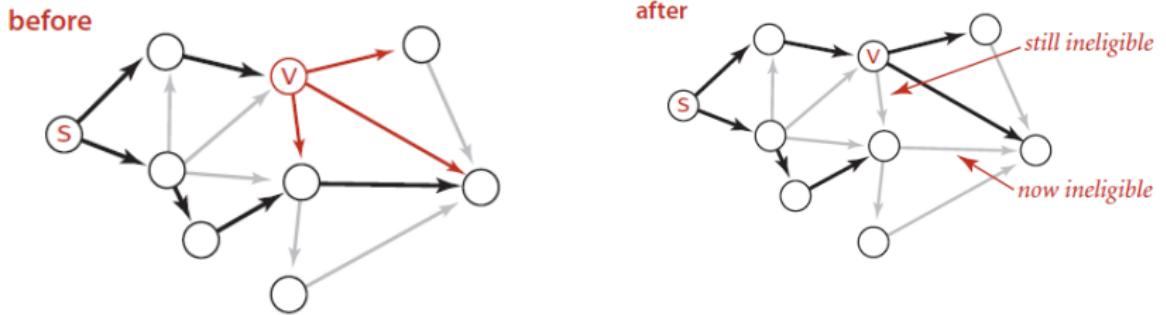


# Knotenrelaxation

Alternative: Ein Knoten  $w$  wird relaxiert, wenn der kürzeste Weg von  $s$  über die Kante  $v \rightarrow w$  zum Knoten  $w$  verläuft.

```
void relax(EdgeWeightedDigraph G, int v) {
    vector<DirectedEdge> e = G.adj(v);
    for (int i=0; i < e.size(); i++){
        int w = e[i].to();
        if (distTo[w] > distTo[v] + e.weight())
        {
            distTo[w] = distTo[v] + e.weight();
            edgeTo[w] = e
        }
    }
}
```

# Beispiel Knotenrelaxation

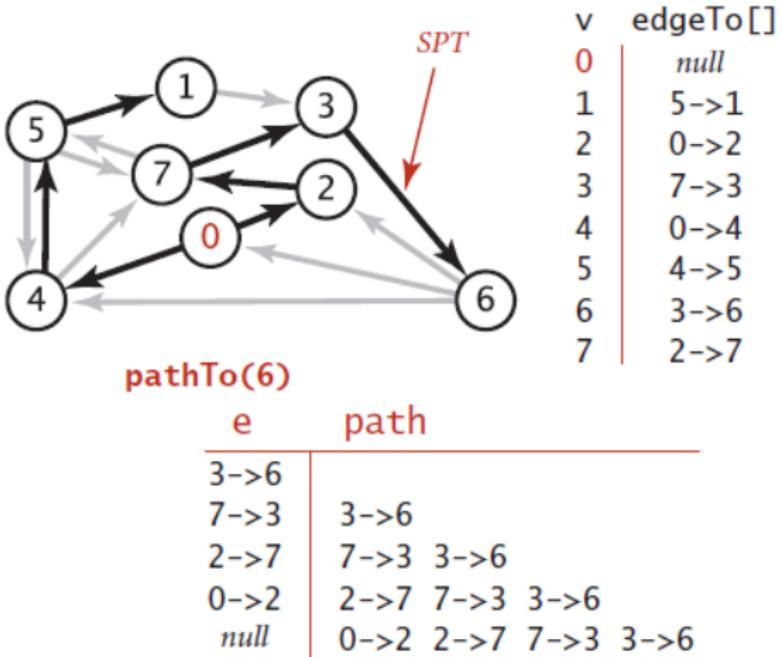


# Client-Abfragemethode

**Analog zur Pfadsuche:** mit den Arrays können die Methoden `pathTo()`, `hasPathTo()`, `distTo()` den kürzesten Pfad liefern.

```
double distTo(int v) { return distTo[v]; }
bool hasPathTo(int v) { return distTo[v] < infinity; }
vector<DirectedEdge> pathTo(int v) {
    if (!hasPathTo(v)) return null;
    stack<DirectedEdge> path;
    for (DirectedEdge e=edgeTo[v]; e!=null; e=edgeTo[e.from()])
        path.push(e);
    return path;
}
```

# Ablaufprotokoll pathTo()-Berechnung



# Dijkstra Algorithmus

---

- ▶ Analog zum Algorithmus von Prim zur Berechnung des minimalen Spannbaums.
- ▶ Der Dijkstra-Algorithmus berechnet einen Kürzeste-Pfade-Baum (SPT).
- ▶ Prioritätswarteschlange  $pq$  speichert die Kosten von Kanten, die den aktuellen SPT um eine Kante erweitert zu noch nicht besuchten Knoten.
- ▶ Priorisiert werden die Gesamtkosten des Pfades vom Startknoten  $s$ .

# API Dijkstra-Algorithmus

```
class DijkstraSP {  
private:  
    vector<DirectedEdge> edgeTo;  
    vector<double> distTo;  
    priority_queue<double> pq;  
    void relax(EdgeWeightedDigraph G, int v);  
public:  
    DijkstraSP(EdgeWeightedDigraph G, int s); // Konstruktor  
    double distTo(int v);  
    bool hasPathTo(int v);  
    vector<DirectedEdge> pathTo(int v);  
};
```

# Implementierung Dijkstra-Algorithmus

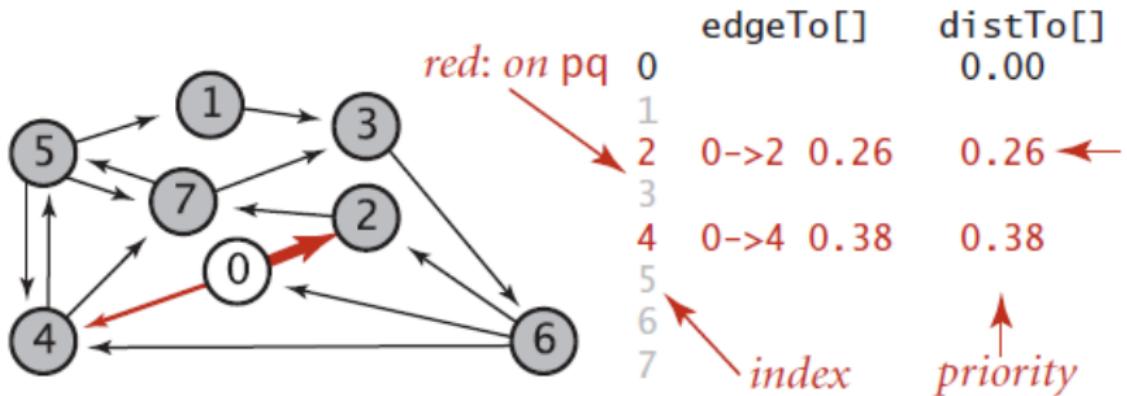
```
DijkstraSP(EdgeWeightedDigraph G, int s) {
    for (int v = 0; v < G.V(); v++) {
        distTo[v] = infinity;
    }
    distTo[s] = 0.0;

    pq.insert(s, 0.0);
    while (!pq.isEmpty())
        relax(G, pq.delMin());
}
```

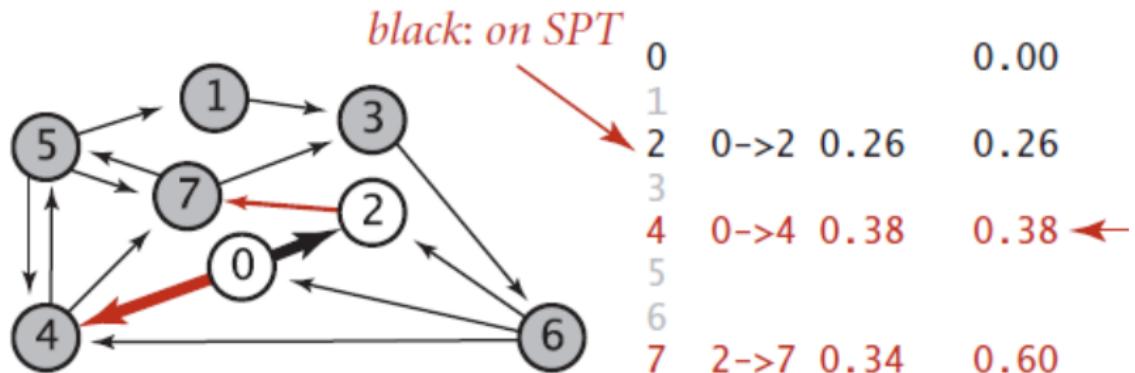
# Implementierung Dijkstra-Algorithmus

```
void relax(EdgeWeightedDigraph G, int v) {  
    vector<DirectedEdge> e = G.adj(v);  
    for (int i = 0; i < e.size(); i++) {  
        int w = e.to();  
        if (distTo[w] > distTo[v] + e.weight())  
        {  
            distTo[w] = distTo[v] + e.weight();  
            edgeTo[w] = e;  
            if (pq.contains(w))  
                pq.change(w, distTo[w]);  
            else  
                pq.insert(w, distTo[w]);  
        }  
    }  
}
```

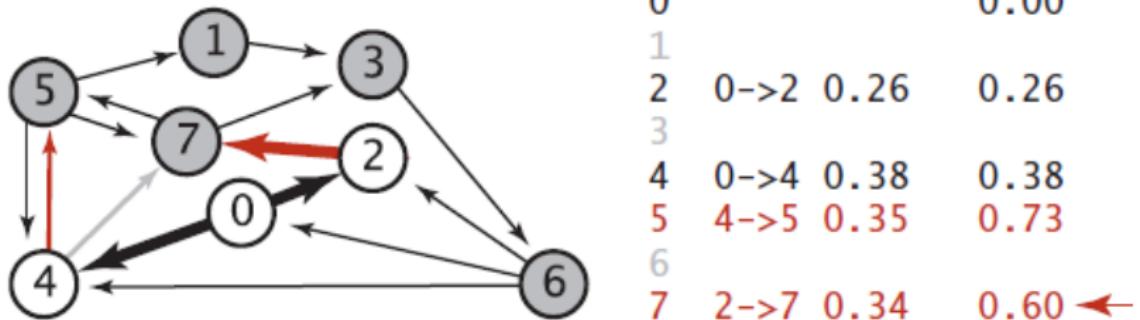
# Ablaufprotokoll Dijkstra-Algorithmus



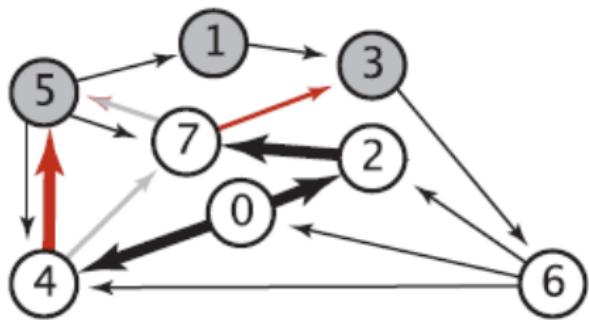
# Ablaufprotokoll Dijkstra-Algorithmus



# Ablaufprotokoll Dijkstra-Algorithmus

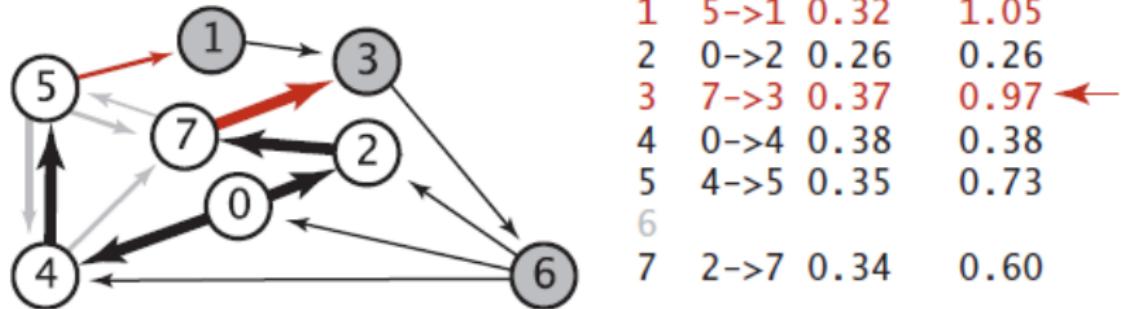


# Ablaufprotokoll Dijkstra-Algorithmus

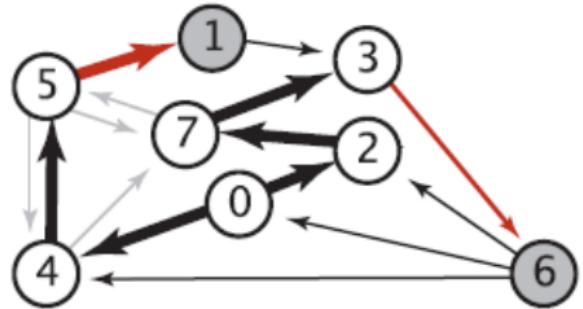


0		0.00	
1			
2	0->2	0.26	0.26
3	7->3	0.37	0.97
4	0->4	0.38	0.38
5	4->5	0.35	0.73 ←
6			
7	2->7	0.34	0.60

# Ablaufprotokoll Dijkstra-Algorithmus

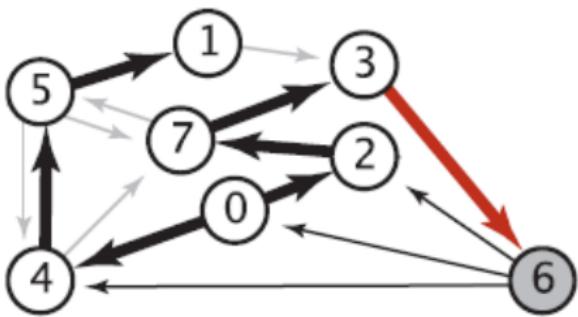


# Ablaufprotokoll Dijkstra-Algorithmus



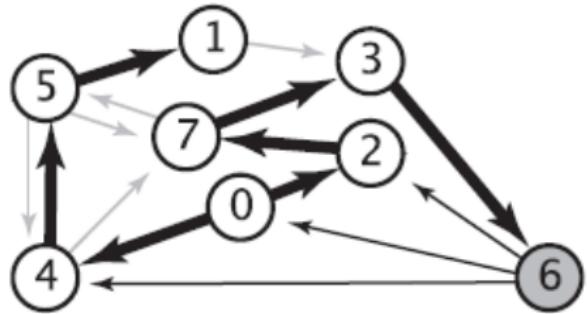
0		0.00	
1	5->1	0.32	1.05 ←
2	0->2	0.26	0.26
3	7->3	0.37	0.97
4	0->4	0.38	0.38
5	4->5	0.35	0.73
6	3->6	0.52	1.49
7	2->7	0.34	0.60

# Ablaufprotokoll Dijkstra-Algorithmus



0		0.00	
1	5->1	0.32	1.05
2	0->2	0.26	0.26
3	7->3	0.37	0.97
4	0->4	0.38	0.38
5	4->5	0.35	0.73
6	3->6	0.52	1.49 ←
7	2->7	0.34	0.60

# Ablaufprotokoll Dijkstra-Algorithmus



0			0.00
1	5->1	0.32	1.05
2	0->2	0.26	0.26
3	7->3	0.37	0.97
4	0->4	0.38	0.38
5	4->5	0.35	0.73
6	3->6	0.52	1.49
7	2->7	0.34	0.60

# Laufzeitanalyse Dijkstra Algorithmus

## Laufzeit Dijkstra

Gegeben sei ein kantengewichteter Digraph mit  $E$  Kanten und  $V$  Knoten. Der Algorithmus von Dijkstra berechnet den Kürzeste-Pfade-Baum ausgehend von einem Startknoten und benötigt zusätzlichen **Speicher**  $\sim O(V)$  und **Laufzeit**

$$\sim O(E \log V)$$

**Beweis:** analog zum Prim-Algorithmus zur Berechnung des MST

# Inhalt - Kapitel 7:

## Elementare Graphenalgorithmen

### ► Elementare Graphenalgorithmen

- Beispiele
- Definitionen
- Datenstrukturen
  - Kantenliste
  - Knotenliste
  - Adjazenzmatrix
  - Adjazenzliste
- Algorithmen zu Graphen
  - Tiefensuche
  - Rekursive Tiefensuche
  - Iterative Tiefensuche
  - Pfadsuche mit Tiefensuche
  - Connected Components mit Tiefensuche
  - Breitensuche
- Algorithmen zum Minimalen Spannbaum
  - Prim Algorithmus
  - Kruskal Algorithmus
- Kürzeste Pfade
  - Dijkstra Algorithmus

Prof. Ingrid Scholl  
FH Aachen  
Fachbereich für Elektrotechnik und Informationstechnik  
Graphische Datenverarbeitung und Grundlagen der Informatik  
MASKOR Institut  
Eupener Straße 70  
52066 Aachen  
T +49 (0)241 6009-52177  
F +49 (0)241 6009-52190  
[scholl@fh-aachen.de](mailto:scholl@fh-aachen.de)  
[www.fh-aachen.de](http://www.fh-aachen.de)