



Universität Bremen

# Graphenalgorithmen

Thomas Röfer

Begriffe

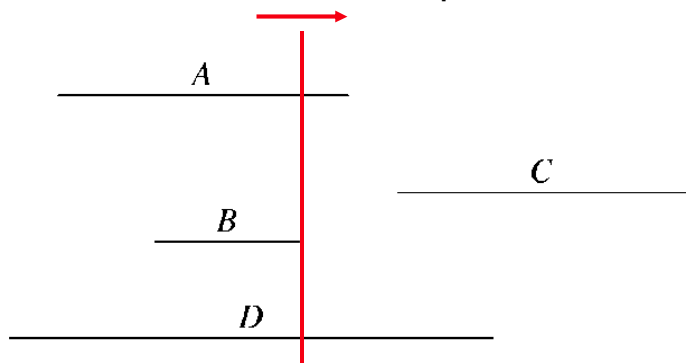
Repräsentationen

Kürzeste Wege

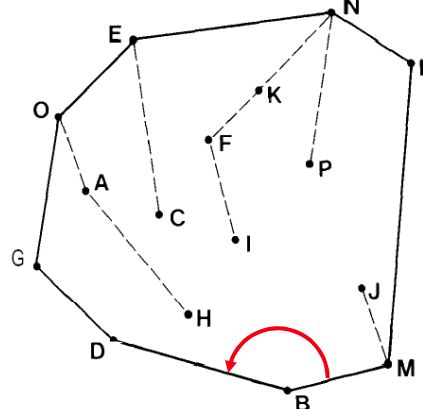
Minimale spannende Bäume

# Rückblick „Geometrische Algorithmen“

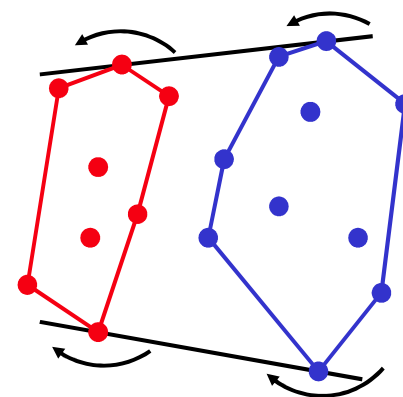
Scan-Line-Prinzip



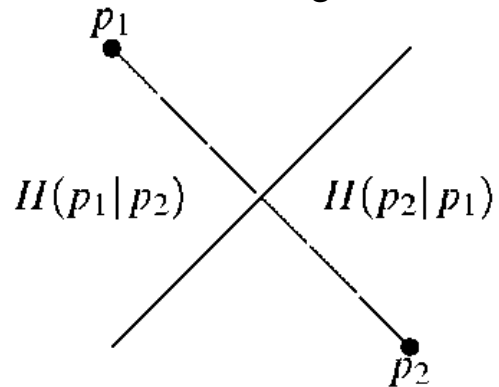
Graham-Scan



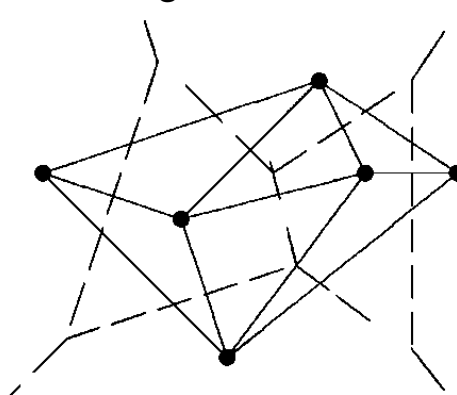
Divide and Conquer



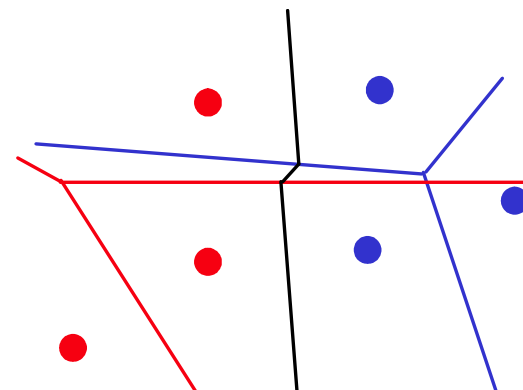
Voronoi-Diagramm



Eigenschaften



Konstruktion





# Motivation

- ▶ **Kürzeste Wege**
  - ▶ Wie besucht man Orte mit der kürzest möglichen Rundreise?
  - ▶ Was ist der schnellste Weg von Freiburg nach Bremen?
  - ▶ Was ist der billigste Weg von Freiburg nach Bremen?
- ▶ **Zuordnungsprobleme**
  - ▶ Wie ordnet man Arbeitskräfte einer Firma am besten denjenigen Tätigkeiten zu, für die sie geeignet sind?
  - ▶ Wie teilt man 210 Studierende so in Dreiergruppen auf, dass die Gruppenmitglieder miteinander auskommen und das gleiche Tutorium besuchen können?
- ▶ **Flussprobleme**
  - ▶ Welche Wassermenge kann die Kanalisation in Bremen höchstens verkraften?
- ▶ **Planungsprobleme**
  - ▶ Wann kann ein Projekt am frühesten beendet werden, wenn alle Tätigkeiten in der richtigen Reihenfolge ausgeführt werden?

# Motivation – Beispiel

- ▶ **Eulerweg (Euler, 1736)**

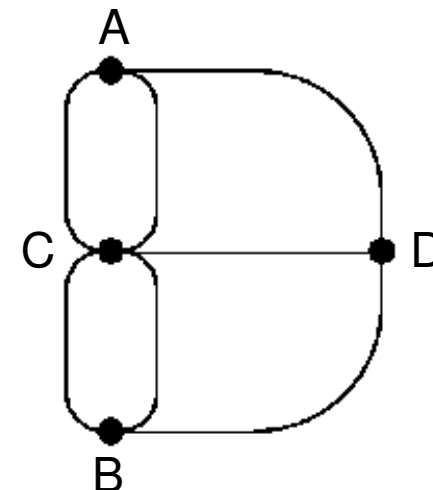
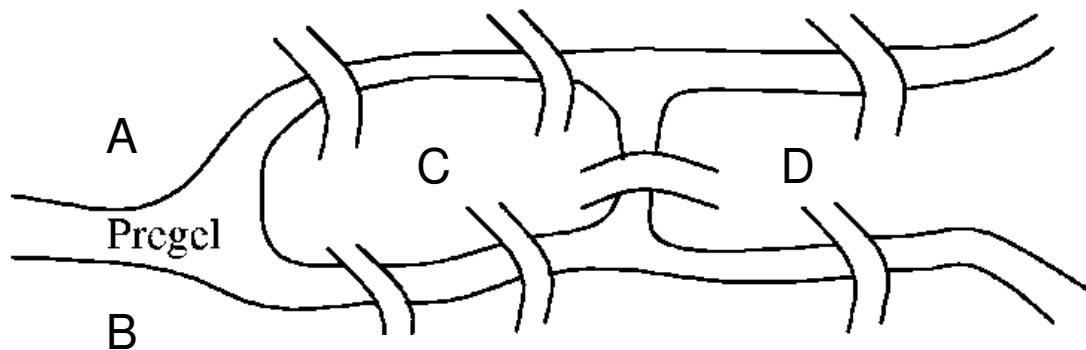
- ▶ Wie muss ein Rundweg durch Königsberg aussehen, auf dem man jede Brücke über den Pregel genau einmal überquert und am Ende zum Ausgangspunkt zurückkehrt?

- ▶ **Ansatz**

- ▶ Repräsentiere die Stadtteile durch Knoten
- ▶ Repräsentiere die Brücken durch Kanten, die die Knoten verbinden

- ▶ **Lösung**

- ▶ Es gibt keinen solchen Rundweg!
- ▶ Der Grad aller Knoten müsste gerade sein



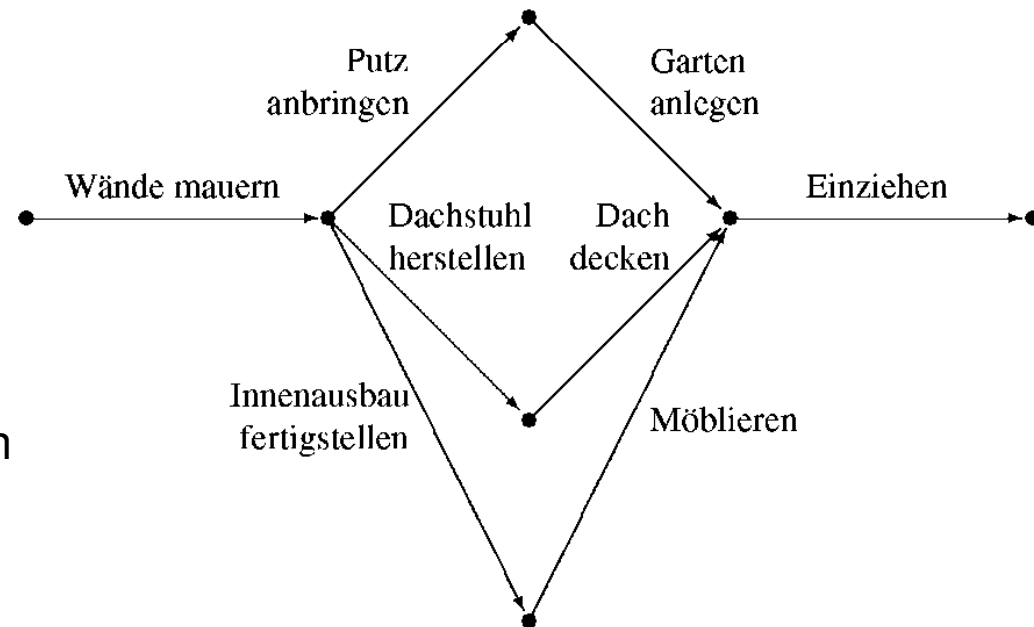
# Gerichteter Graph

## ▶ Definition

- ▶ Ein gerichteter Graph  $G = (V, E)$  (*Digraph*) besteht aus einer Menge  $V = \{1, 2, 3, \dots, |V|\}$  von *Knoten* (*vertices*) und einer Menge  $E \subseteq V \times V$  von Pfeilen (*edges*, *arcs*)
- ▶ Ein Paar  $(v, v') = e \in E$  heißt *Pfeil* von  $v$  nach  $v'$
- ▶  $v$  ist der Anfangsknoten und  $v'$  der Endknoten von  $e$
- ▶  $v$  und  $v'$  heißen *adjazent*
- ▶  $v$  und  $v'$  sind mit  $e$  *inzident*,  $e$  ist *inzident* mit  $v$  und  $v'$

## ▶ Eigenschaften

- ▶ Da  $E$  eine Menge ist, kann jeder Pfeil nur einmal auftreten
  - ▶ *Es gibt keine parallelen Pfeile*





# Begriffe

- ▶ **Eingangsgrad eines Knotens (*indegree*)**
  - ▶ Anzahl der einmündenden Pfeile
  - ▶  $indeg(v) = |\{v' \mid (v', v) \in E\}|$
- ▶ **Ausgangsgrad eines Knotens (*outdegree*)**
  - ▶ Anzahl der ausgehenden Pfeile
  - ▶  $outdeg(v) = |\{v' \mid (v, v') \in E\}|$
- ▶ **Teilgraph**
  - ▶ Ein Digraph  $G' = (V', E')$ , geschrieben  $G' \subseteq G$ , falls  $V' \subseteq V$  und  $E' \subseteq E$
- ▶ **Weg von  $v$  nach  $v'$  (*path*)**
  - ▶ Ein Folge von Knoten  $(v_0, v_1, \dots, v_k) \in V$  mit  $v_0 = v$ ,  $v_k = v'$  und  $(v_i, v_{i+1}) \in E$ ,  $i = 0 \dots k-1$
  - ▶  $k$  ist die *Länge* des Wegs
  - ▶ Ein Weg heißt *einfach*, wenn kein Knoten mehrfach besucht wird
  - ▶ Ein Weg ist ein Teilgraph
- ▶ **Zyklus**
  - ▶ Ein Weg, der am Ausgangsknoten endet
  - ▶ Ein Digraph heißt zyklensfrei (azyklisch), wenn er keinen Zyklus enthält



# Begriffe

## ▶ **Gerichteter Wald**

- ▶ Ein Digraph  $G = (V, E)$ , wenn  $E$  zyklensfrei ist und  $\text{indeg}(v) \leq 1$  für alle  $v \in V$
- ▶ Jeder Knoten  $v$  mit  $\text{indeg}(v) = 0$  ist eine Wurzel des Waldes

## ▶ **Gerichteter Baum**

- ▶ Ein gerichteter Wald mit genau einer Wurzel

## ▶ **Spannender Wald von $G = (V, E)$**

- ▶ Ein gerichteter Wald  $W = (V, F)$  mit  $F \subseteq E$
- ▶ Falls  $W$  ein Baum, heißt  $W$  *spannender Baum* von  $G$

## ▶ **Ungerichteter Graph**

- ▶ Für jeden Pfeil  $(v, v')$  gibt es einen weiteren  $(v', v)$
- ▶ Ein ungerichteter Graph heißt zyklensfrei, wenn er keinen einfachen Zyklus aus wenigstens drei Pfeilen enthält

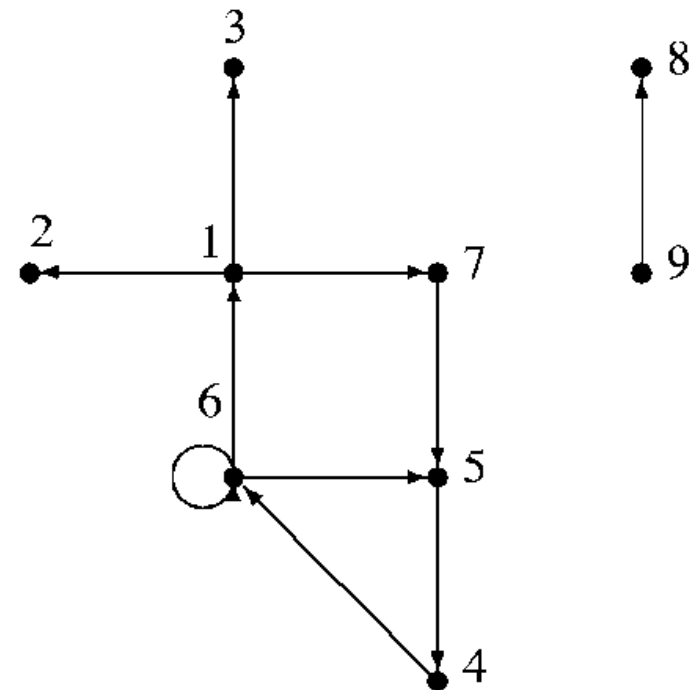
# Repräsentation – Adjazenzmatrix

► **Ansatz**

- Ein Graph  $G = (V, E)$  wird in einer Boole'schen  $|V| \times |V|$ -Matrix  $A_G = (a_{ij})$  mit  $1 \leq i \leq |V|, 1 \leq j \leq |V|$  gespeichert,

$$\text{wobei } a_{ij} = \begin{cases} 0 & \text{falls } (i, j) \notin E \\ 1 & \text{falls } (i, j) \in E \end{cases}$$

| $i \backslash j$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|------------------|---|---|---|---|---|---|---|---|---|
| 1                | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 2                | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3                | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4                | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 5                | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 6                | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 7                | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 8                | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 9                | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |



# Repräsentation – Adjazenzmatrix

## ▶ Aufwand

- ▶ Platz:  $O(|V|^2)$
- ▶ Initialisierung: Alle Matrixeinträge müssen gesetzt werden:  $O(|V|^2)$

## ▶ Effizientere Initialisierung

- ▶ Die Pfeile stehen in einem eindimensionalen Array  $B$
- ▶ In der Matrix  $A$  stehen Indizes auf dieses Array
- ▶  $i$  und  $j$  sind benachbart, wenn ihr Index innerhalb des befüllten Bereichs des Arrays  $[1 \dots b_{max}]$  liegt und dort tatsächlich der richtige Pfeil eingetragen ist
- ▶ Aufwand:  $O(|E|)$
- ▶ Java: lohnt sich nur bei mehrfacher Nutzung

|           | B        |
|-----------|----------|
| 1         |          |
| ⋮         |          |
| $k$       | $i   j$  |
|           |          |
| $k''$     | $i'   j$ |
|           |          |
| $b_{max}$ |          |
|           |          |
| $k'$      | $i   j'$ |
|           |          |

| A    | $j$          | $j'$ |             |  |
|------|--------------|------|-------------|--|
|      |              |      |             |  |
| $i'$ | $1$<br>$k''$ |      |             |  |
|      |              |      |             |  |
| $i$  | $1$<br>$k$   |      | $0$<br>$k'$ |  |
|      |              |      |             |  |

$A[i, j]$  und  $A[i', j]$  sind bedeutsam,  $A[i, j']$  ist es nicht.



# Adjazenzmatrix – Implementierung

```
class AdjMatrix
{
    private static class Edge
    {
        int row,
            column;

        Edge(int r, int c)
        {
            row = r;
            column = c;
        }
    }

    private final int numberOfNodes = 10;
    private final int numberOfEdges = 10;
    private int[][] a =
        new int[numberOfNodes][numberOfNodes];
    private Edge[] b = new Edge[numberOfEdges];
    private int bMax;
}
```

```
void init()
{
    bMax = 0;
}

boolean adjacent(int i, int j)
{
    return a[i][j] >= 0 &&
        a[i][j] < bMax &&
        b[a[i][j]].row == i &&
        b[a[i][j]].column == j;
}

void addEdge(int i, int j)
{
    if(!adjacent(i, j))
    {
        a[i][j] = bMax;
        b[bMax++] = new Edge(i, j);
    }
}
}
```

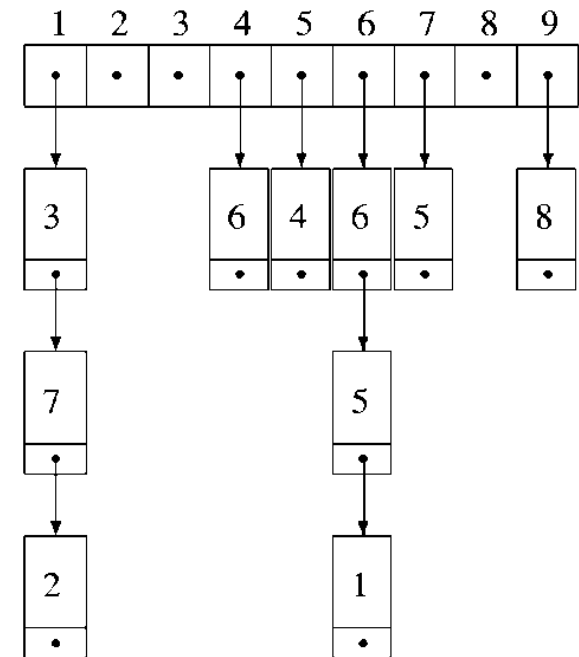
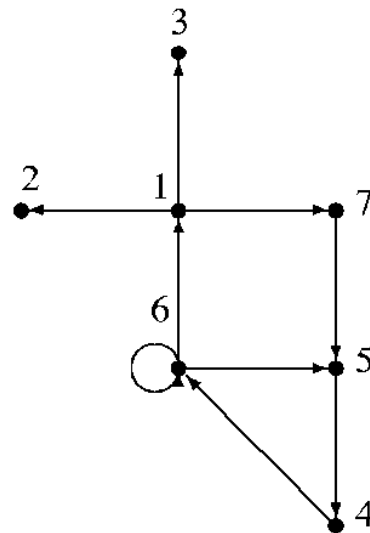
# Repräsentation – Adjazenzlisten

▶ **Ansatz**

- ▶ Alle Knoten stehen in einem Array
- ▶ Jeder Knoten enthält eine Liste der adjazenten Knotenindizes

▶ **Aufwand**

- ▶ Platz:  $O(|V| + |E|)$
- ▶ Effizient: Aufzählen der Pfeile eines Knotens
- ▶ Ineffizient: Hinzufügen und Entfernen von Knoten





# Adjazenzlisten – Implementierung

```
class AdjList
{
    static class Edge
    {
        int endNode;
        Edge next;

        Edge(int e, Edge n)
        {
            endNode = e;
            next = n;
        }
    }

    private final int numberOfNodes = 10;
    private Edge[] a =
        new Edge[numberOfNodes];
}
```

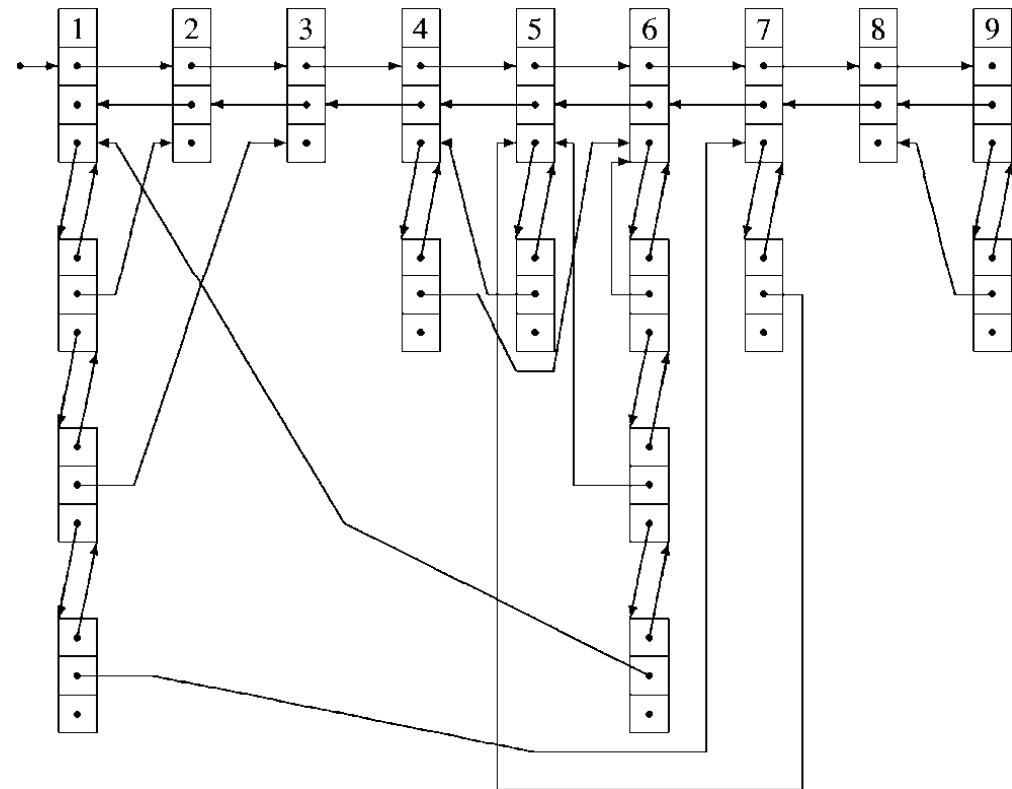
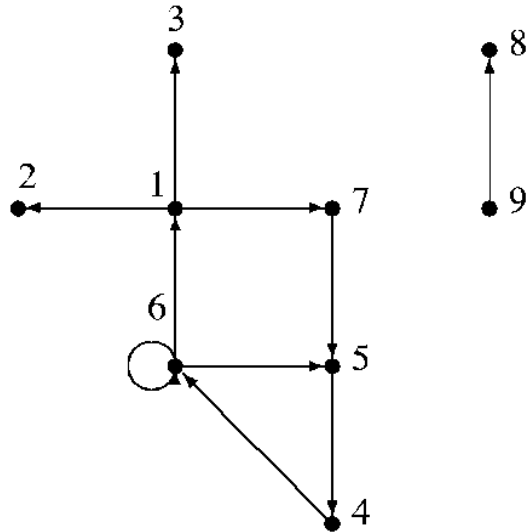
```
boolean adjacent(int i, int j)
{
    Edge e = a[i];
    while(e != null &&
        e.endNode != j)
        e = e.next;
    return e != null;
}

void addEdge(int i, int j)
{
    if(!adjacent(i, j))
        a[i] = new Edge(j, a[i]);
}
}
```

# Repräs. – Doppelt verkettete Pfeilliste

## Ansatz

- ▶ Knoten werden in einer doppelt verketteten Liste gespeichert
- ▶ Jeder Knoten enthält eine doppelt verkettete Pfeilliste (*DCAL: double connected arc list*)
- ▶ Jeder Eintrag der DCAL zeigt auf einen benachbarten Knoten





## DCAL – Implementierung

```
import java.util.LinkedList;

class AdjLinkedList extends LinkedList
{
    static class Node
    {
        private LinkedList edges = new LinkedList();

        boolean adjacent(Node node)
        {
            return edges.contains(node);
        }

        void addEdge(Node node)
        {
            if(!adjacent(node))
                edges.add(node);
        }
    }
}
```



# Kürzeste Wege – Definitionen

## ▶ Bewerteter Graph

- ▶ Ein ungerichteter Graph  $G = (V, E)$  mit einer reellwertigen Bewertungsfunktion  $c : E \rightarrow \mathbb{R}$
- ▶ Entsprechend heißt ein Digraph mit Bewertungsfunktion bewerteter Digraph
- ▶ Für eine Kante  $e \in E$  heißt  $c(e)$  Bewertung (Länge, Gewicht, Kosten) der Kante  $e$

## ▶ Distanzgraph

- ▶ Die Länge jeder Kante ist nicht negativ, also  $c : E \rightarrow \mathbb{R}_0^+$

## ▶ Wege

- ▶ Die Länge  $c(G)$  des Graphen  $G$  ist die Summe der Länge aller Kanten
- ▶ Die Länge eines Wegs  $p = (v_0, v_1, \dots, v_k)$  ist die somit die Summe
$$\sum_{i=0}^{k-1} c((v_i, v_{i+1}))$$
- ▶ Die Distanz  $d$  von einem Knoten  $v$  zu einem anderen Knoten  $v'$  ist
$$d(v, v') = \min\{ c(p) \mid p \text{ ist Weg von } v \text{ nach } v' \}$$

## ▶ Kürzester Weg

- ▶ Ein Weg  $p$  zwischen  $v$  und  $v'$ , wenn  $c(p) = d(v, v')$

# Kürzeste Wege in Distanzgraphen

## ▶ Ansatz

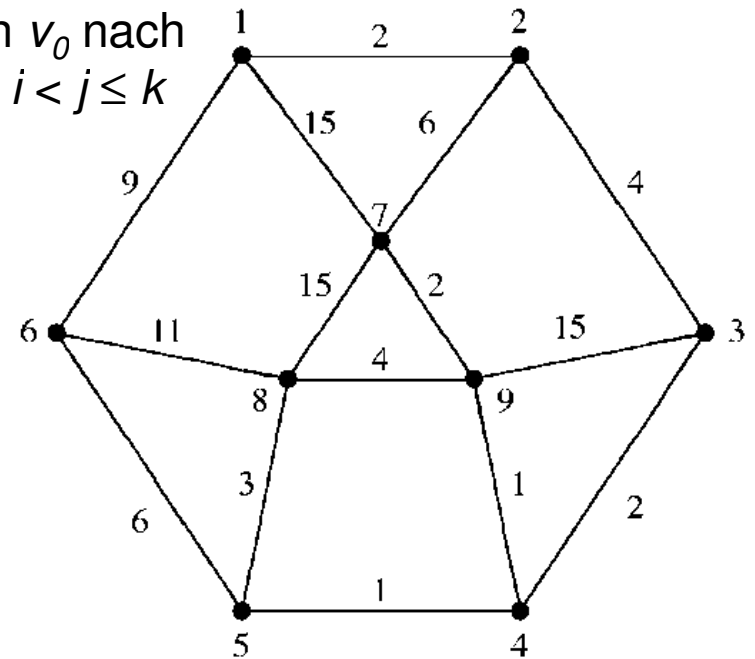
- ▶ Man findet den kürzesten Weg, indem man, beginnend mit dem Startknoten, immer längere Wege konstruiert, und zwar in der Reihenfolge ihrer Länge

## ▶ Optimalitätsprinzip

- ▶ Wenn  $p = (v_0, v_1, \dots, v_k)$  ein kürzester Weg von  $v_0$  nach  $v_k$  ist, dann ist jeder Teilweg  $p = (v_i, \dots, v_j)$ ,  $0 \leq i < j \leq k$  ebenfalls ein kürzester Weg

## ▶ Invarianten

- ▶ Für alle kürzesten Wege  $sp(s, v)$  und Kanten  $(v, v')$  gilt:  
 $c(sp(s, v)) + c((v, v')) \geq c(sp(s, v'))$
- ▶ Für mindestens einen kürzesten Weg  $sp(s, v)$  und eine Kante  $(v, v')$  gilt:  
 $c(sp(s, v)) + c((v, v')) = c(sp(s, v'))$



# Algorithmus von Dijkstra (1959)

## ▶ Jeder Knoten gehört zu einer von drei Klassen

### ▶ Gewählte Knoten

- ▶ *Zu ihnen ist der kürzeste Weg vom Anfangsknoten  $s$  bereits bekannt*

### ▶ Randknoten

- ▶ *Zu ihnen kennt man einen Weg von  $s$*

### ▶ unerreichte Knoten

- ▶ *Zu ihnen kennt man keinen Weg*

## ▶ Pro Knoten $v$ merken

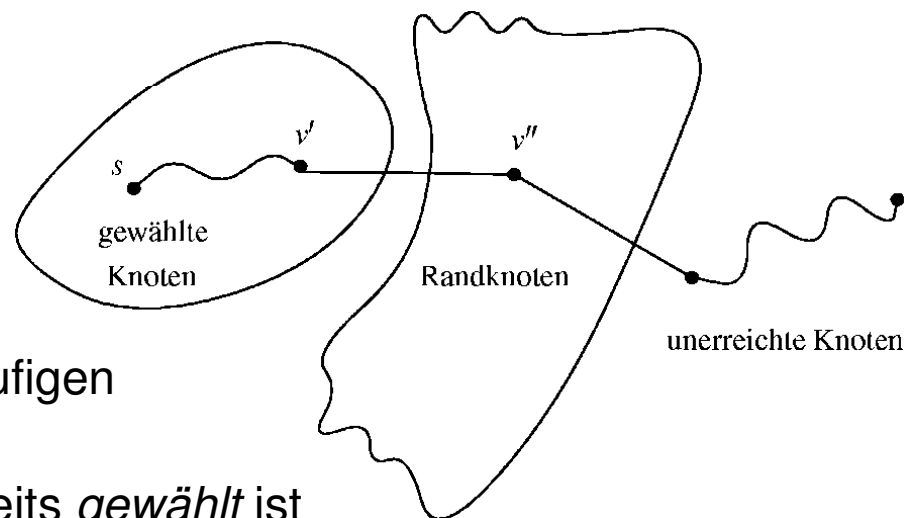
- ▶ Die bisherige Entfernung zu  $s$

- ▶ Den Vorgänger von  $v$  auf dem vorläufigen kürzesten Weg von  $s$  nach  $v$

- ▶ Eine Markierung, ob der Knoten bereits *gewählt* ist

## ▶ Zusätzlich

- ▶ Eine Menge speichert die Randknoten



# Algorithmus von Dijkstra (1959)

## ▶ Initialisierung

- ▶ Alle Knoten außer dem Startknoten  $s$  sind nicht gewählt
- ▶ Die Entfernung zu  $s$  ist 0, zu allen anderen Knoten  $\infty$

## ▶ Alle zu $s$ adjazenten Knoten gehören zum Rand $R$

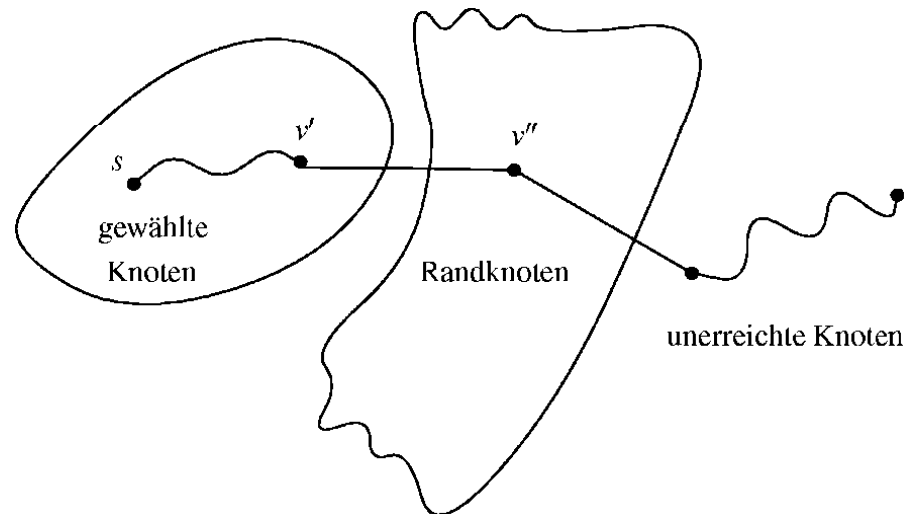
- ▶  $R := \emptyset$
- ▶ ergänze Rand  $R$  bei  $s$

## ▶ Berechne Wege ab $s$

- ▶ Solange  $R \neq \emptyset$ 
  - ▶ wähle  $v \in R$  mit  $v$ .Entfernung minimal
  - ▶  $v$  aus  $R$
  - ▶  $v$ .gewählt := true
  - ▶ ergänze Rand  $R$  bei  $v$

## ▶ Ergänze Rand $R$ bei $v$

- ▶ Für alle  $(v, v') \in E$
- ▶ Falls nicht  $v'$ .gewählt und  $v$ .Entfernung +  $c((v, v')) < v'$ .Entfernung)
  - ▶  $v'$ .Vorgänger :=  $v$
  - ▶  $v'$ .Entfernung :=  $v$ .Entfernung +  $c((v, v'))$
  - ▶  $R := R \cup \{v'\}$

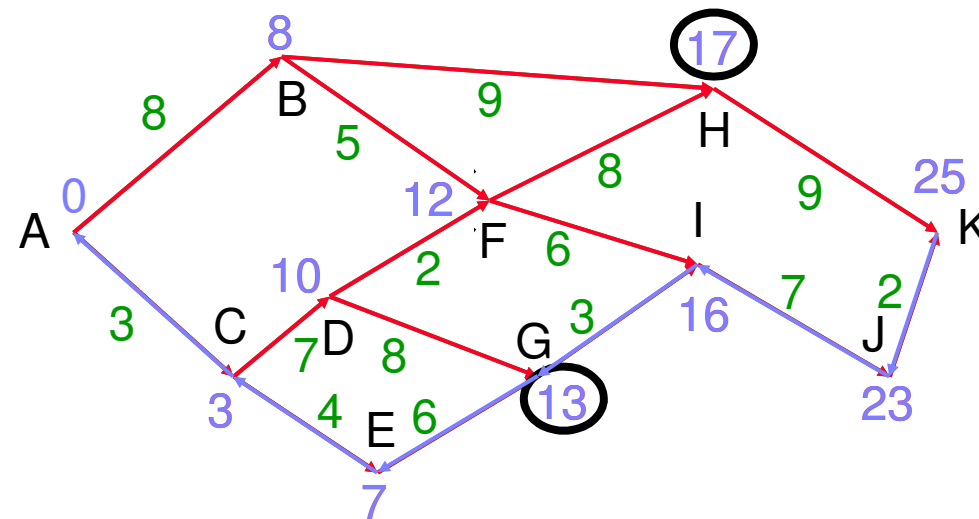




# Dijkstra-Algorithmus – Beispiel

Unbesuchte Pfeile

Besuchte Pfeile



Kosten pro Pfeil

Endgültige Wegkosten



## Dijkstra – Verwaltung des Rands

### ▶ Aufwand

- ▶  $|E| \cdot O(\text{Rand erweitern}) + |V| \cdot O(\text{Minimum entnehmen})$

### ▶ Implizite Speicherung des Rands

- ▶ Rand wird nicht separat gespeichert, sondern implizit in der Knotenmenge
- ▶ *Minimum entnehmen* heißt dann, alle nicht gewählten Knoten  $v$  nach der minimalen  $v$ .Entfernung zu durchsuchen
- ▶ Aufwand:  $|E| \cdot O(1) + |V| \cdot O(|V|) = O(|E| + |V|^2)$
- ▶ Lohnt sich, wenn Graph dicht mit Kanten besetzt ist

### ▶ Explizite Speicherung des Rands

- ▶ Nutzung einer Vorrangwarteschlange zur Speicherung des Rands
- ▶ Aufwand, z.B. bei Verwendung eines Linksbaums:  
 $|E| \cdot O(\log |V|) + |V| \cdot O(\log |V|) = O(|E| \log |V| + |V| \log |V|)$
- ▶ Lohnt sich, wenn Graph dünn besetzt mit Kanten ist

# Minimale spannende Bäume

## Definition

- Ein minimaler spannender Baum (MST: minimum spanning tree) eines Graphen  $G$  ist ein spannender Baum von  $G$  von minimaler Gesamtlänge

## Greedy-Algorithmus $MST(V, E) = (V, E')$

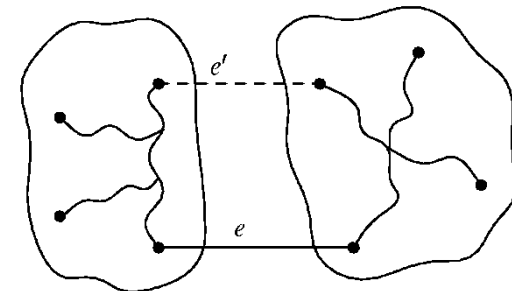
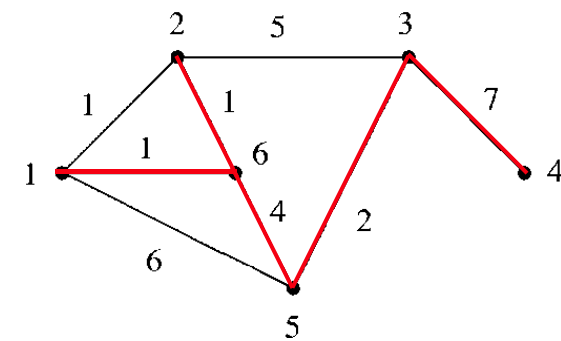
- $E' := \emptyset$
- solange noch nicht fertig
  - wähle geeignete Kante  $e \in E$
  - $E' := E' \cup \{e\}$

## Auswahlprozess

- Kanten sind entweder gewählt, verworfen oder unentschieden

## Schnitt

- Ein Schnitt in einem Graphen zerlegt die Knotenmenge  $V$  in zwei Untermengen  $S$  und  $\bar{S} = V - S$
- Eine Kante kreuzt den Schnitt, wenn sie mit einem Knoten aus  $S$  und einem aus  $\bar{S}$  inzident ist

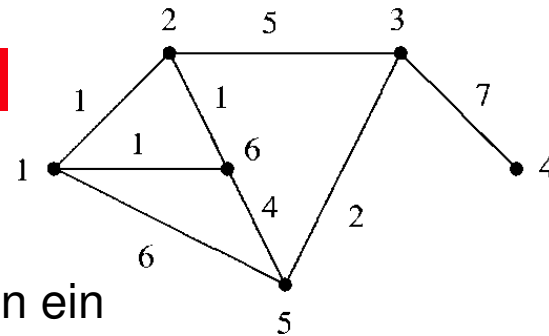




# Minimale spannende Bäume

- ▶ **Regel 1: Wähle eine Kante**
  - ▶ Wähle einen Schnitt, den keine der gewählten Kanten kreuzt
  - ▶ Wähle eine kürzeste unter den unentschiedenen Kanten, die den Schnitt kreuzen
- ▶ **Regel 2: Verwirf eine Kante**
  - ▶ Wähle einen einfachen Zyklus, der keine verworfene Kante enthält
  - ▶ Verwirf die längste unter den unentschiedenen Kanten im Zyklus
- ▶ **Präzisierung des Algorithmus**
  - ▶ Wähle geeignete Kante  $e \in E$ 
    - ▶ *Wiederhole*
      - ▶ *wende eine anwendbare Auswahlregel an*
      - ▶ *bis Kante  $e \in E$  mit Regel 1 gewählt oder es keine unentschiedene Kante mehr gibt*
  - ▶ Noch nicht fertig
    - ▶ *Es gibt noch unentschiedene Kanten*

# Algorithmus von Kruskal



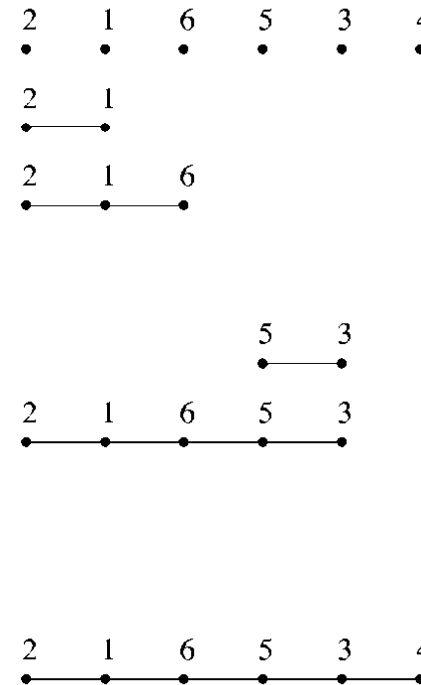
## ▶ Ansatz

- ▶ Anfangs ist jeder Knoten des Graphen ein gewählter Baum
- ▶ Auswahlschritt für jede Kante  $e$  in aufsteigender Reihenfolge der Kantenlänge
  - ▶ Falls  $e$  beide Endknoten im selben Baum hat, verwirf  $e$ , sonst wähle  $e$

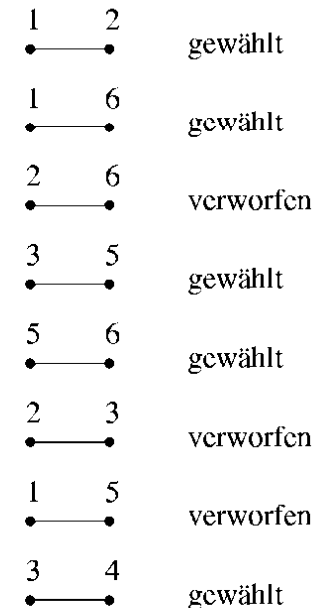
## ▶ Algorithmus $MST(V, E) = (V, E')$

- ▶  $E' := \emptyset$
- ▶ Sortiere  $E$  nach aufsteigender Länge
- ▶  $makeSet(v)$  für alle  $v \in V$
- ▶ Für alle  $(v, w) \in E$ , aufsteigend
  - ▶ Falls  $find(v) \neq find(w)$ 
    - ▶  $union(find(v), find(w))$
    - ▶  $E' := E' \cup \{ (v, w) \}$

gewählte Bäume



betrachtete Kante



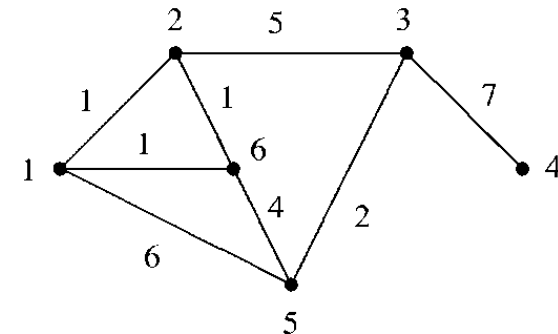
# Algorithmus von Jarnik, Prim, Dijkstra

## ▶ Ansatz

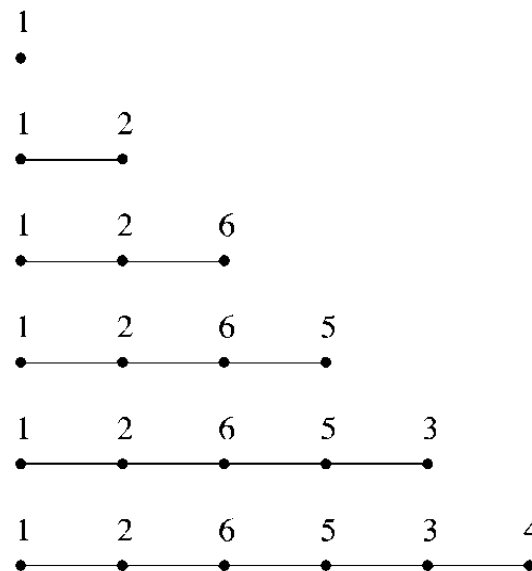
- ▶ Zu jedem Zeitpunkt gibt es nur genau einen gewählten Baum
- ▶ Zu Beginn besteht dieser nur aus dem Startknoten  $s$
- ▶ Später bilden alle gewählten Kanten und deren inzidente Knoten den gewählten Baum

## ▶ Algorithmus

- ▶ Wähle einen beliebigen Startknoten  $s$
- ▶ Führe Auswahlsschritt  $(|V| - 1)$ -mal durch
  - ▶ *Wähle eine Kante mit minimaler Länge, für die genau ein Endknoten zum gewählten Baum gehört*



gewählter Baum



gewählte Kante

