Universität Bremen

Praktische Informatik 2
Graphenalgorithmen

Thomas Röfer

Cyber-Physical Systems
Deutsches Forschungszentrum für Künstliche Intelligenz

Multisensorische Interaktive Systeme
Fachbereich 3, Universität Bremen

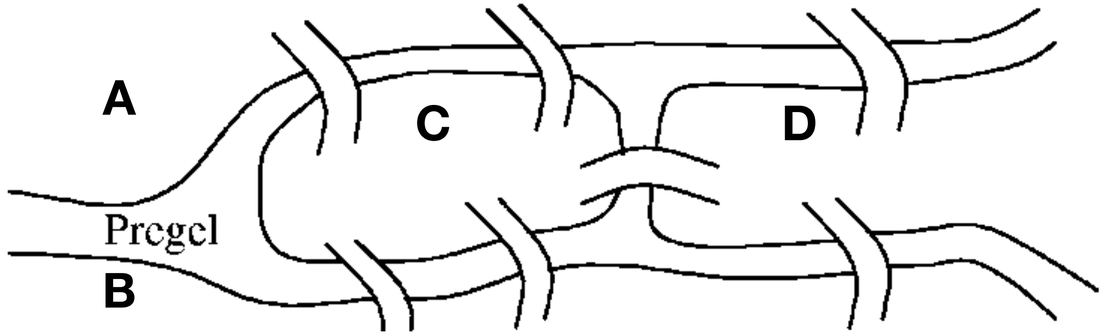
Bild: ©Michael Ihle

Universität Bremen

Motivation

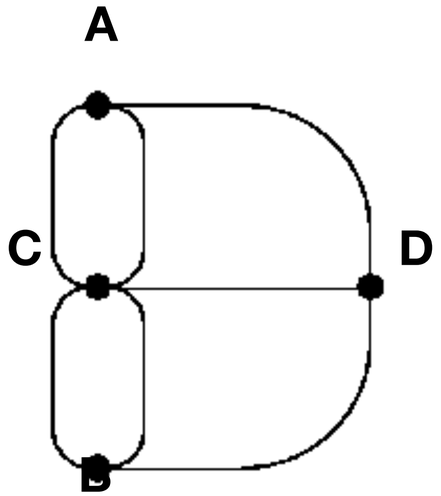
- **Kürzeste Wege**
  - Wie besucht man Orte mit der kürzest möglichen Rundreise?
  - Was ist der kürzeste Weg von Freiburg nach Bremen?
  - Was ist der schnellste Weg von Freiburg nach Bremen?
- **Zuordnungsprobleme**: Wie teilt man 350 Studierende so in Zweiergruppen auf, dass Gruppenmitglieder miteinander auskommen und im gleichen Tutorium sind?
- **Flussprobleme**: Welche Wassermenge kann eine Kanalisation 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 jede Brücke über den Pregel genau einmal überquert wird und der am Ausgangspunkt endet?

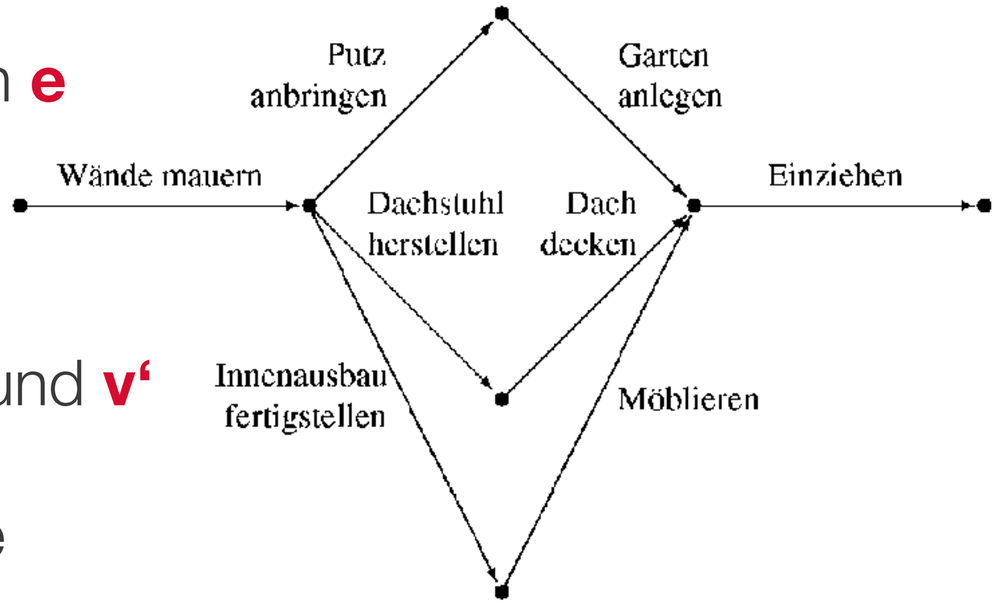
- **Ansatz**: Repräsentiere die Stadtteile durch Knoten
  - Repräsentiere Brücken durch Kanten, die die Knoten verbinden



- **Lösung**: Es gibt keinen solchen Rundweg!
  - Der Grad aller Knoten müsste gerade sein

Gerichteter Graph

- Ein gerichteter Graph **G = (V, E)** (**Digraph**) besteht aus Menge **V = {1,2,3…, |V|}** von **Knoten** (**vertices**) und Menge **E ⊆ V × V** von **Pfeilen/Kanten** (**edges**, **arcs**)
- Ein Paar **(v, v‘) = e ∈ 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‘**
- Da **E** eine Menge ist, gibt es keine parallelen Pfeile



Universität Bremen

Begriffe

- Eingangsgrad eines Knotens (**indegree**)
  - Anzahl der einmündenden Pfeile
  - **indeg(v) = | { v‘ | (v‘, v) ∈ E } |**
- Ausgangsgrad eines Knotens (**outdegree**)
  - Anzahl der ausgehenden Pfeile
  - **outdeg(v) = | { v‘ | (v, v‘) ∈ E } |**
- **Teilgraph**: **G‘ = (V‘, E‘)**, geschrieben **G‘ ⊆ G**, falls **V‘ ⊆ V** und **E‘ ⊆ E**

Universität Bremen

Begriffe

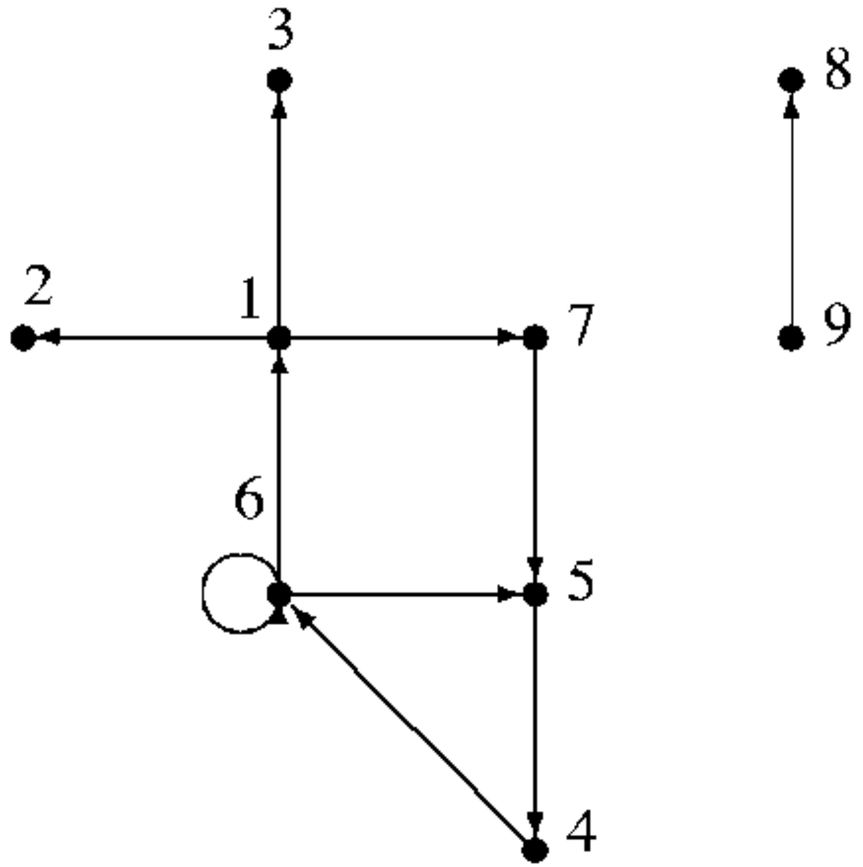
- **Weg** von **v** nach **v‘** (**path**): Eine Folge von Knoten **$(v_0, v_1, \ldots, v_k) \in V$** mit **$v_0 = v$**, **$v_k = v'$** und **$(v_i, v_{i+1}) \in E$**, **$i = 0 \ldots k-1$**
  - **k** ist die **Länge** des Wegs
  - Ein Weg heißt **einfach**, wenn kein Knoten mehrfach besucht wird
  - Ein Weg ist ein Teilgraph
- **Zyklus**: Ein Weg, der am Ausgangsknoten endet
  - Ein Digraph ohne Zyklus heißt **zyklenfrei** (**azyklisch**)
- **Ungerichteter Graph**: Für jeden Pfeil **(v, v‘)** gibt es einen weiteren **(v‘, v)**

Repräsentation: Adjazenzmatrix

- Ein Graph **G = (V, E)** wird in einer Booleschen **|V| × |V|**-Matrix **AG = (a_{ij})** mit **1 ≤ i ≤ |V|**, **1 ≤ j ≤ |V|** gespeichert,

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

	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



Universität Bremen

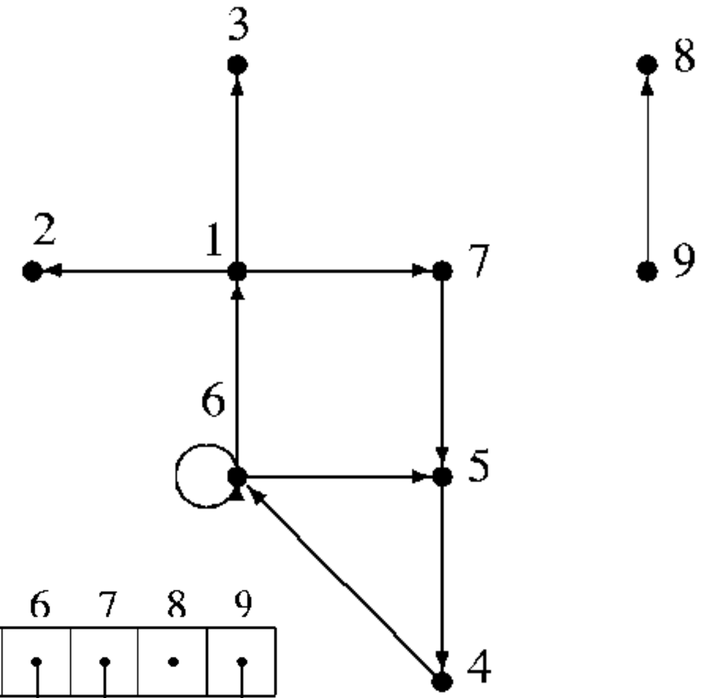
Adjazenzmatrix: Beispiel

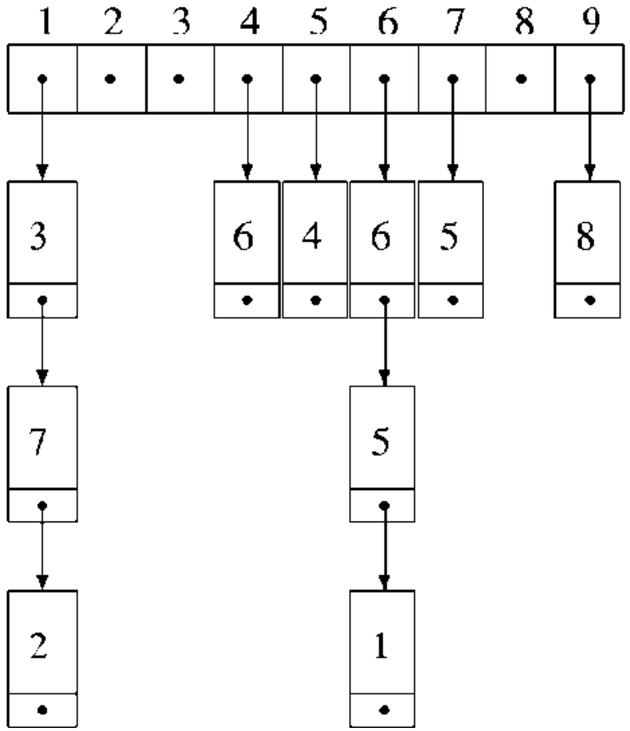
```
class AdjMatrix
{
    private final boolean[ ][ ] a;

    AdjMatrix(final int nodes)
    {
        a = new boolean[nodes][nodes];
    }
```

```
    boolean adjacent(final int i, final int j)
    {
        return a[i][j];
    }

    void addEdge(final int i, final int j)
    {
        a[i][j] = true;
    }
}
```

Repräsentation: Adjazenzlisten

- Alle Knoten stehen in einem Array
- Jeder Knoten enthält Liste adjazenter Knotenindizes
- Aufwand
  - Platz: **O(|V| + |E|)**
  - Effizient: Pfeile eines Knotens aufzählen
  - Ineffizient: Hinzufügen und Entfernen von Knoten



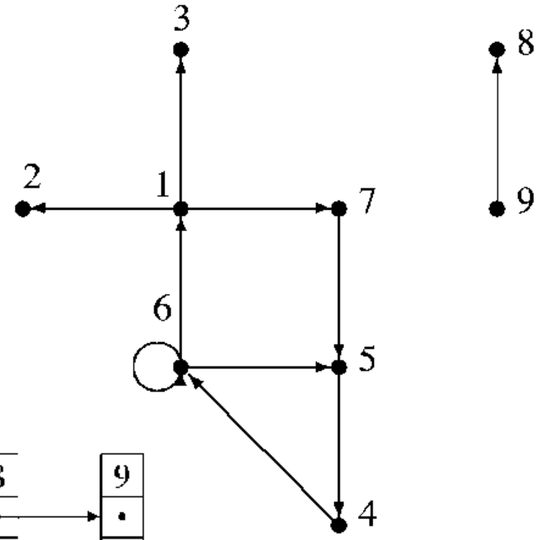


Universität Bremen

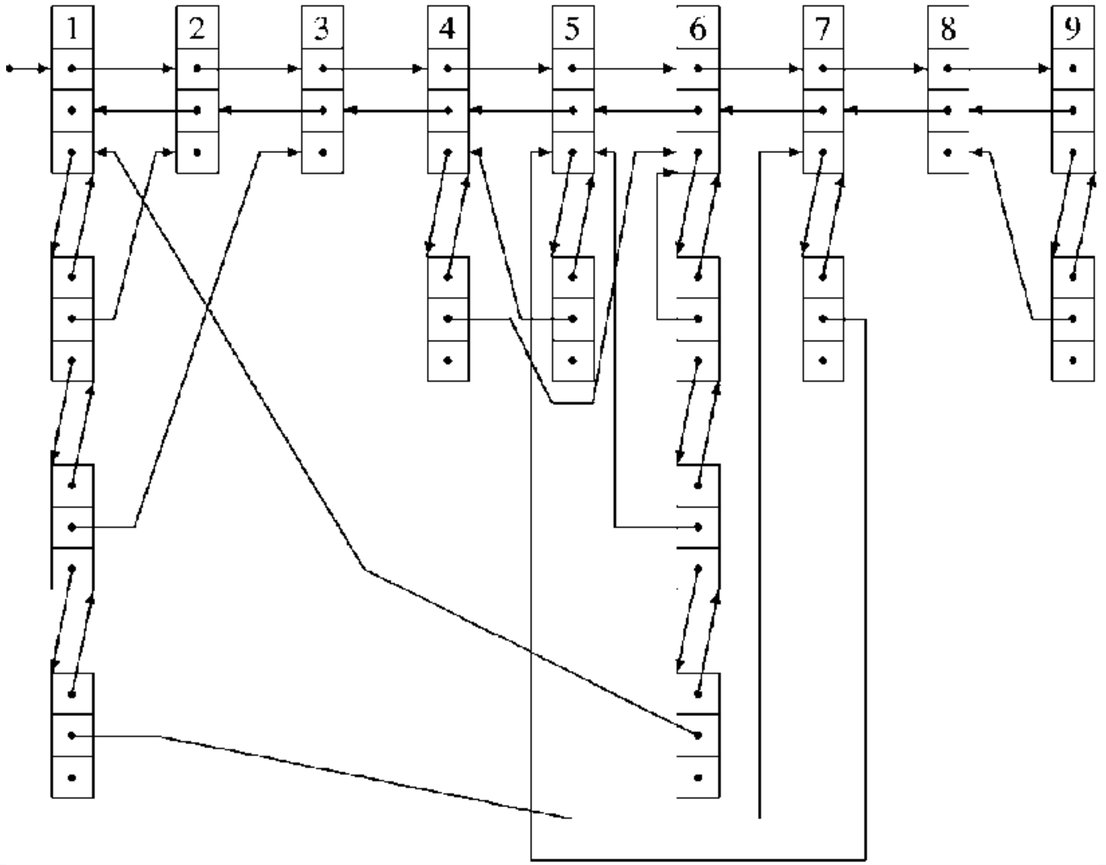
Adjazenzlisten: Beispiel

```
class AdjList
{
    private static class Edge
    {
        final int target;
        final Edge next;

        Edge(final int target, final Edge next)
        {
            this.target = target;
            this.next = next;
        }
    }

    private final Edge[ ] a;

    AdjList(final int nodes)
    {
        a = new Edge[nodes];
    }

    boolean adjacent(final int i, final int j)
    {
        Edge e = a[i];
        for (; e != null && e.target != j; e = e.next);
        return e != null;
    }

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

Repräsentation: Doppelt verkettete Pfeilliste



- 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



Universität Bremen

DCAL: Beispiel

```
class AdjLinkedList extends LinkedList<AdjLinkedList.Node>
{
    static class Node
    {
        private final List<Node> edges = new LinkedList<>();

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

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

Universität Bremen

Kürzeste Wege: Definitionen

- Bewerteter Graph
  - Ein ungerichteter Graph **G = (V, E)** mit einer reellwertigen Bewertungsfunktion **c : E → ℝ**
  - Entsprechend heißt ein Digraph mit Bewertungsfunktion **bewerteter Digraph**
  - Für eine Kante **e ∈ E** heißt **c(e)** Bewertung (Länge, Gewicht, Kosten) der Kante **e**
- Distanzgraph: Länge keiner Kante negativ, also **c : E → $\mathbb{R}_{0+}$**

Kürzeste Wege: Definitionen

- Wege
  - Länge **c(G)** des Graphen **G** ist Summe der Länge aller Kanten
  - Länge eines Wegs **p = (v_0, v_1, …, v_k)** ist 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) | p** ist Weg von **v** nach **v‘ }**
- **Kürzester Weg**: Weg **p** zwischen **v** und **v‘**, wenn **c(p) = d(v, v‘)**

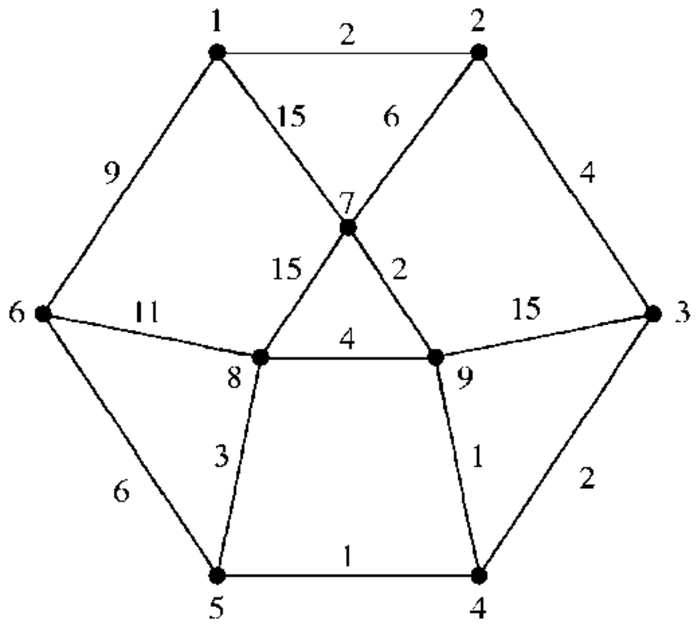
Kürzeste Wege in Distanzgraphen

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

- **Optimalitätsprinzip**: Wenn **$p = (v_0, v_1, \ldots, v_k)$** kürzester Weg von **$v_0$** nach **$v_k$** ist, dann ist jeder Teilweg **$p = (v_i, \ldots, v_j)$, $0 \leq i < j \leq k$** auch ein kürzester Weg

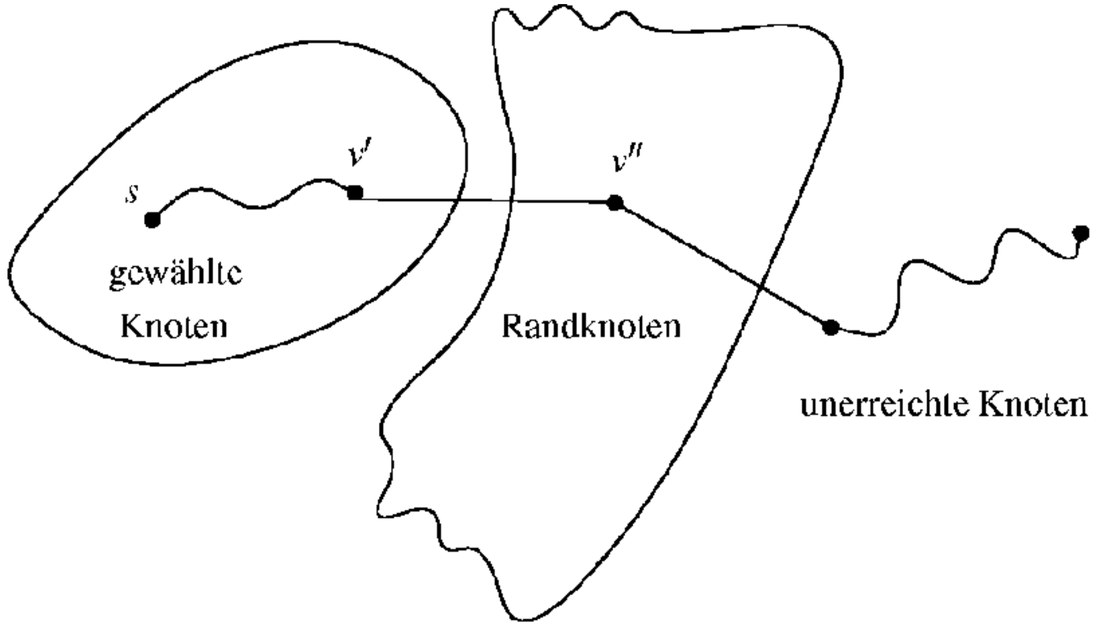
- Für alle kürzesten Wege **sp(s, v)** und Kanten **(v, v‘)** gilt: **$c(sp(s, v)) + c((v, v')) \geq c(sp(s, v'))$**

- Für mindestens einen kürzesten Weg **sp(s, v)** und eine Kante **(v, v‘)** gilt: **$c(sp(s, v)) + c((v, v')) = c(sp(s, v'))$**

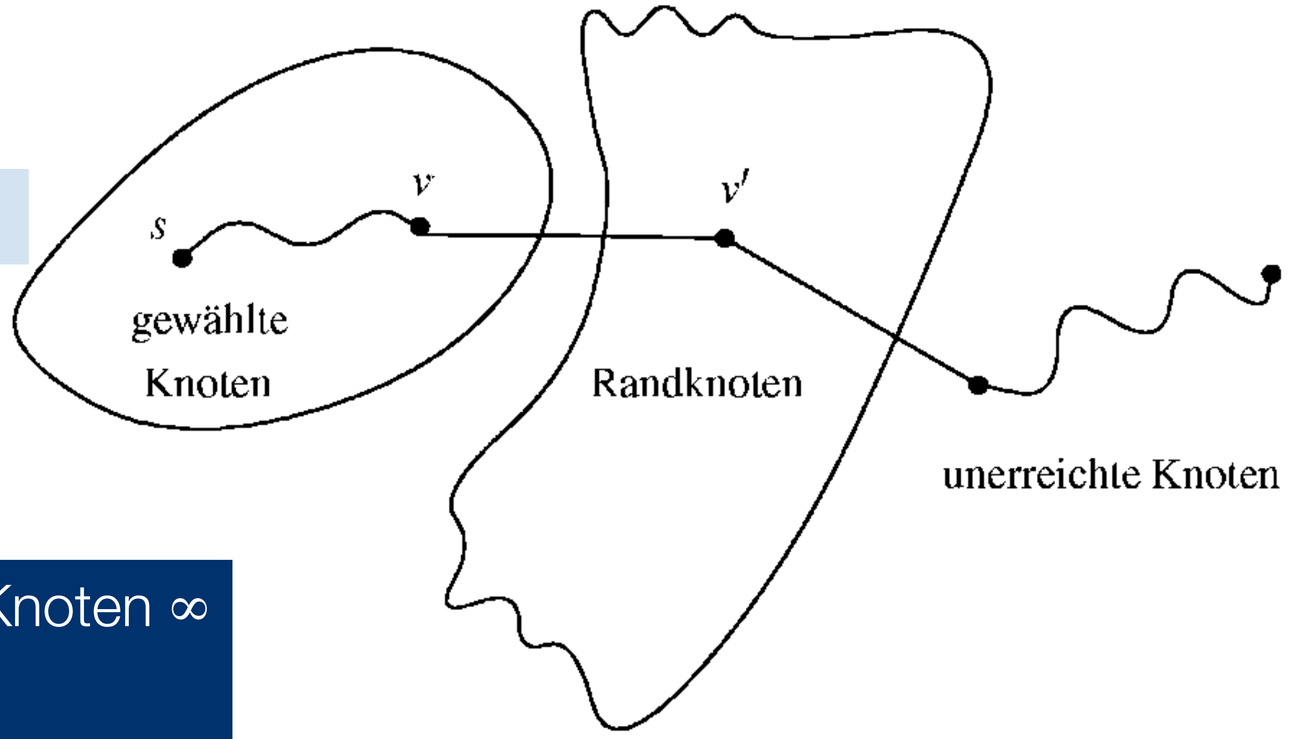


Algorithmus von Dijkstra (1959)

- Jeder Knoten gehört zu einer von drei Klassen
  - **Gewählte Knoten**: Kürzester Weg vom Anfangsknoten **s** bereits bekannt
  - **Randknoten**: Ein Weg von **s** bereits bekannt (von gewählten Knoten über eine Kante erreichbar)
  - **Unerreichte Knoten**: Kein Weg von **s** bekannt
- Pro Knoten **v** merken
  - Bisherige Entfernung zu **s**
  - Vorgänger von **v** auf bisher kürzestem Weg
- Zusätzlich: Eine Menge speichert die Randknoten



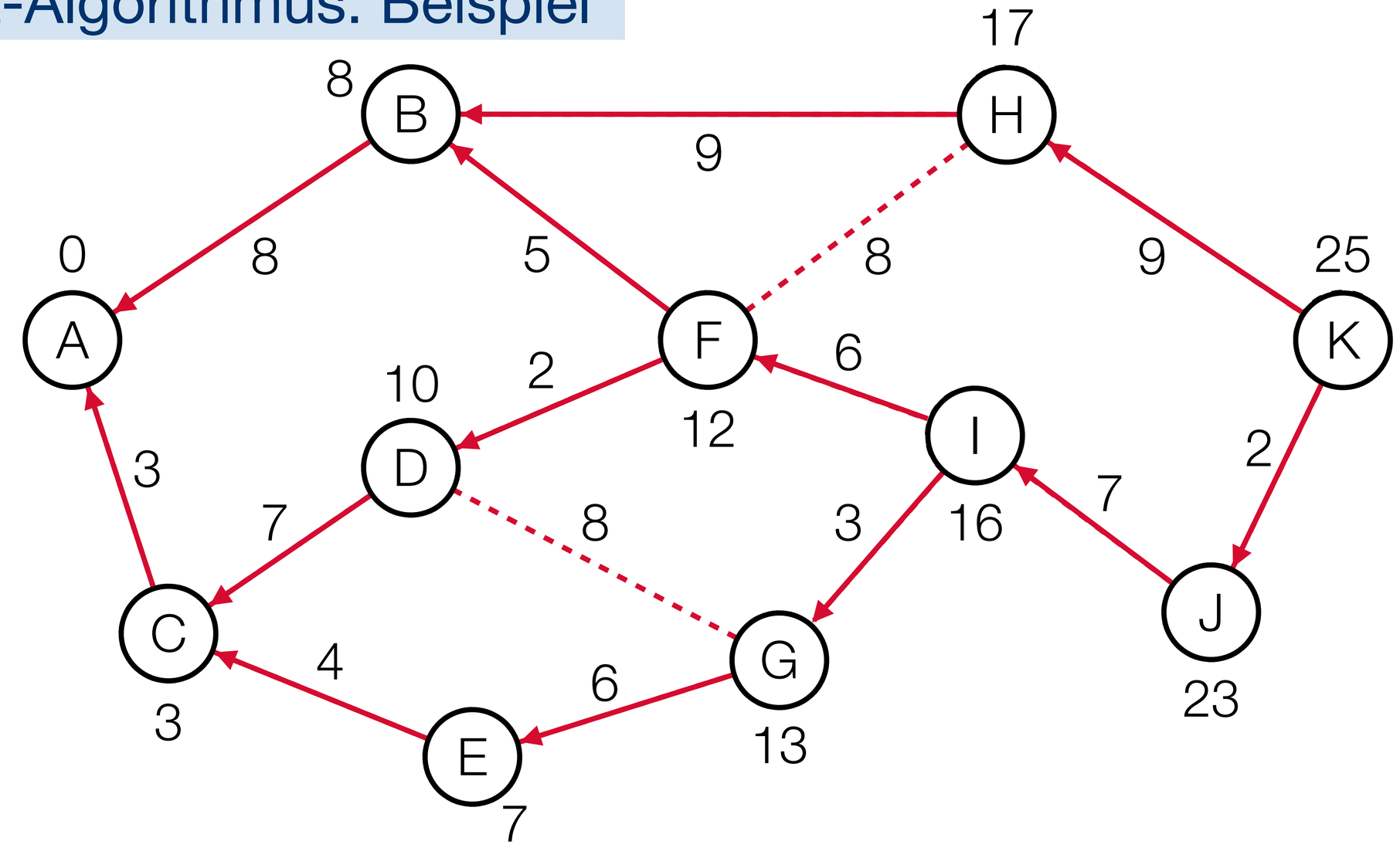
Algorithmus von Dijkstra (1959)



Die Entfernung von **s** ist **0**, von allen anderen Knoten ∞
R ← {s}
Solange **R** ≠ ∅
  Entnimm **v** ∈ **R** mit kleinster **v.entfernung**
  Ergänze Rand **R** bei **v**

Ergänze Rand **R** bei **v**
  Für alle **(v, v‘)** ∈ **E**
    Falls **v.entfernung + c((v, v‘)) < v‘.entfernung**
    v‘.vorgänger ← **v**
    v‘.entfernung ← **v.entfernung** + **c((v, v‘))**
    Füge **v‘** in den Rand **R** ein

Dijkstra-Algorithmus: Beispiel



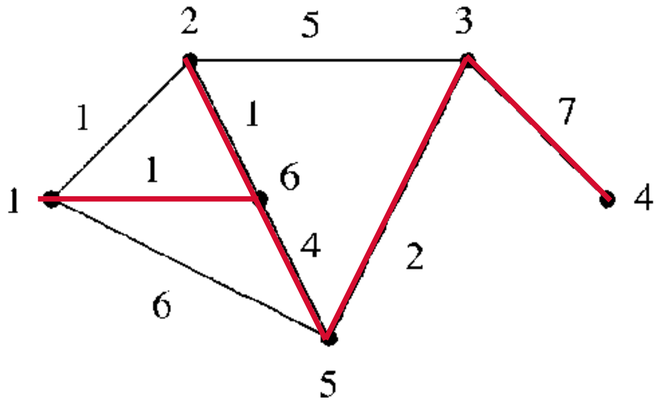
Universität Bremen

Dijkstra: Verwaltung des Rands

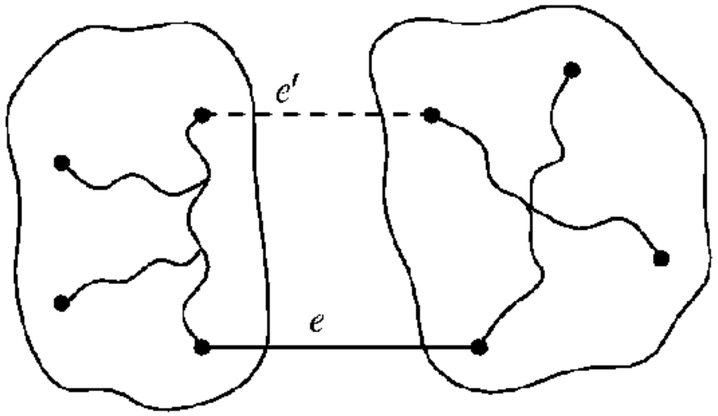
- Aufwand: **|E| · O(**Rand erweitern**) + |V| · O(**Minimum entnehmen**)** (+ Initialisierung der Entfernung **|V|**)
- Implizite Speicherung: Rand in Knotenmenge (Zusätzliche Markierung **gewählt** pro Knoten)
  - Minimum entnehmen (wählen): Minimum aus allen bisher nicht gewählten Knoten
  - Aufwand: $|E| \cdot O(1) + |V| \cdot O(|V|) = O(|E| + |V|^2) = O(|V|^2)$
  - Lohnt sich, wenn Graph dicht mit Kanten besetzt ist
- Explizite Speicherung: Balancierter Suchbaum (Ordnung nach Entfernung vom Start und Knoten)
  - Aufwand: $|E| \cdot O(\log |V|) + |V| \cdot O(\log |V|) = O((|E| + |V|) \log |V|) \Rightarrow O(|E| \log |V|)$
  - Lohnt sich, wenn Graph dünn besetzt mit Kanten ist

Minimale aufspannende Bäume (MST)

- Minimal aufspannender Baum eines Graphen **G** ist aufspannender Baum von **G** von minimaler Gesamtlänge
- Auswahlprozess: Kanten sind entweder gewählt, verworfen oder unentschieden
- Ein **Schnitt** in einem Graphen zerlegt die Knotenmenge **V** in zwei Untermengen **S** und **S‘ = V – S**
- Kante kreuzt Schnitt, wenn sie mit Knoten aus **S** und **S‘** inzident ist

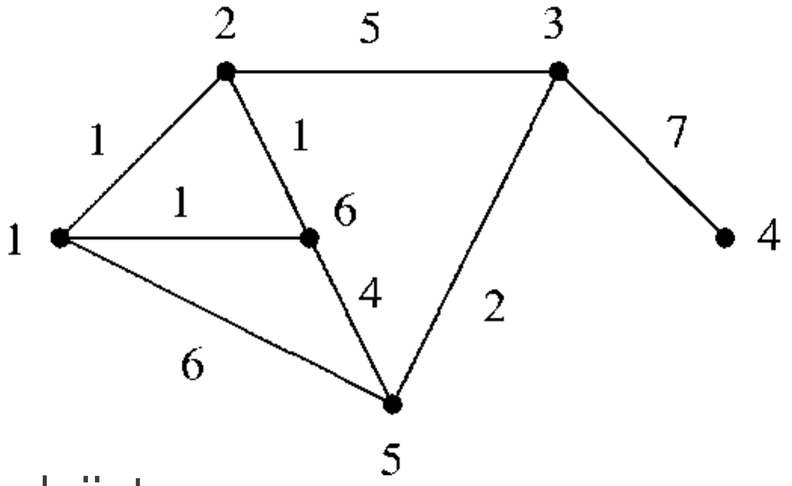


Algorithmus: **MST(V, E) = (V, E‘)**
E‘ ← ∅
solange noch nicht fertig
  Wähle geeignete Kante **e** ∈ **E**
  E‘ ← **E‘** ∪ **{e}**

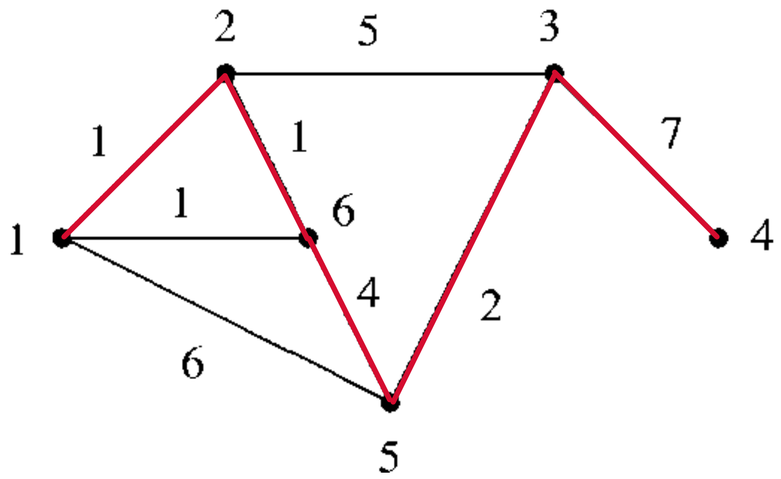


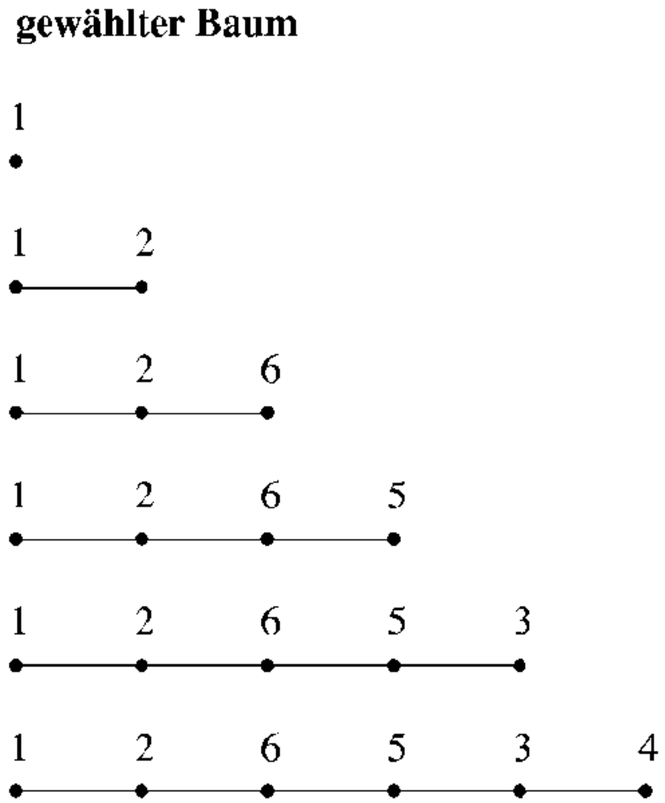
Algorithmus von Jarnik, Prim, Dijkstra

- Zu jedem Zeitpunkt gibt es nur genau einen gewählten Baum
- Zu Beginn besteht dieser nur aus dem Startknoten **s**
- Später bilden ihn alle gewählten Kanten und deren inzidente Knoten
- Algorithmus
  - Wähle einen beliebigen Startknoten **s**
  - Wähle **(|V| - 1)**-mal eine Kante mit minimaler Länge, für die genau ein Endknoten zum gewählten Baum gehört



Algorithmus von Jarnik, Prim, Dijkstra: Beispiel

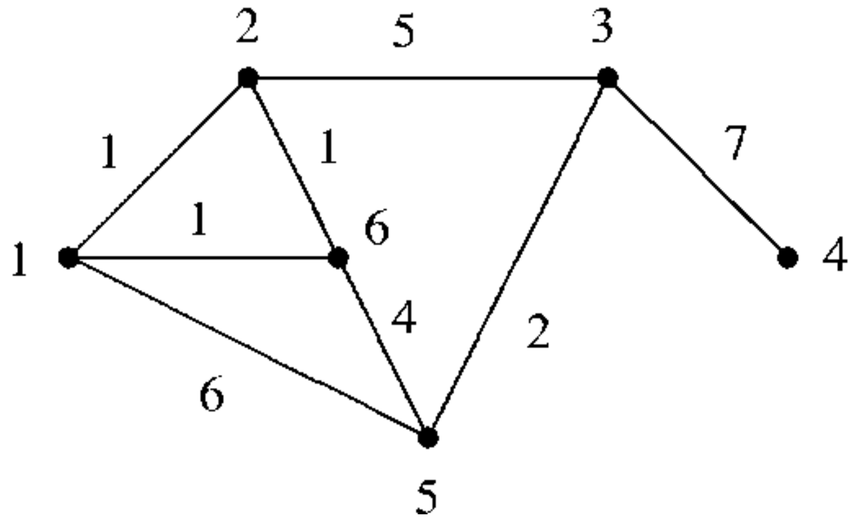




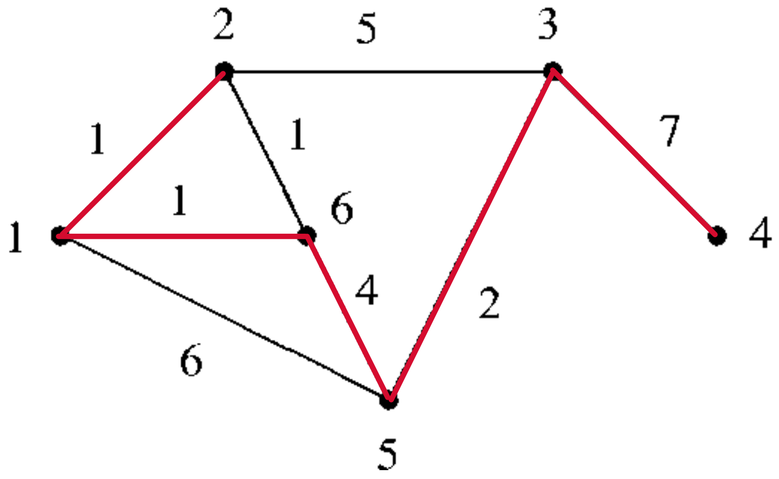
Algorithmus von Kruskal

- Formulierung als Union-Find-Problem
- 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 vereinige Bäume und wähle **e**

Algorithmus: **MST(V, E) = (V, E‘)**
E‘ ← ∅
Sortiere **E** nach aufsteigender Länge
makeSet(v) für alle **v** ∈ **V**
Aufsteigend für alle **(v, w)** ∈ **E, find(v)** ≠ **find(w)**
    union(v, w)
    E‘ ← **E‘** ∪ **{ (v, w) }**



Beispiel: Algorithmus von Kruskal



gewählte Bäume	betrachtete Kante	
2 1 6 5 3 4	1 2	gewählt
2 1	1 6	gewählt
2 1 6	2 6	verworfen
	3 5	gewählt
5 3	5 6	gewählt
2 1 6 5 3	2 3	verworfen
	1 5	verworfen
	3 4	gewählt
2 1 6 5 3 4		

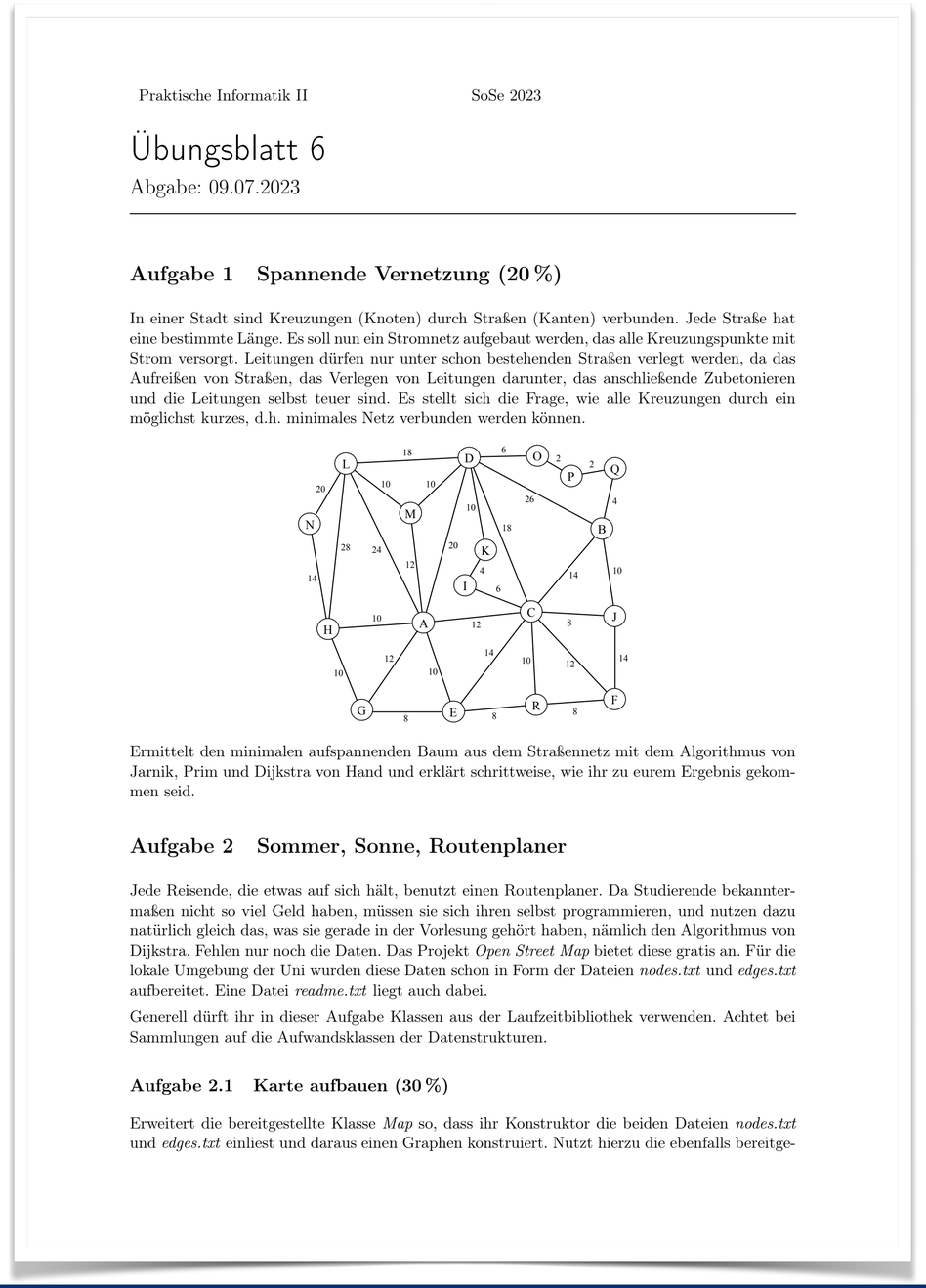
Universität Bremen

Zusammenfassung der Konzepte

- **Gerichteter** und **ungerichteter Graph**
- **Eingangsgrad** und **Ausgangsgrad**
- **Weg** und **Zyklus**
- **Adjazenzmatrix**, **Adjazenzliste**, **DCAL**
- **Kürzester Weg**
- **Minimal aufspannender Baum**

Übungsblatt 6

- Aufgabe 1: Minimal aufspannenden Baum bestimmen
- Aufgabe 2
  - Open-Street-Map-Karte einlesen und zeichnen
  - Dichtesten Knoten suchen
  - Kürzesten Weg von Start zu Ziel
- Bonusaufgabe
  - Kostenfaktoren pro Kante
  - A* statt Dijkstra



Praktische Informatik II SoSe 2023

Übungsblatt 6

Abgabe: 09.07.2023

Aufgabe 1 Spannende Vernetzung (20 %)

In einer Stadt sind Kreuzungen (Knoten) durch Straßen (Kanten) verbunden. Jede Straße hat eine bestimmte Länge. Es soll nun ein Stromnetz aufgebaut werden, das alle Kreuzungspunkte mit Strom versorgt. Leitungen dürfen nur unter schon bestehenden Straßen verlegt werden, da das Aufreißen von Straßen, das Verlegen von Leitungen darunter, das anschließende Zubetonieren und die Leitungen selbst teuer sind. Es stellt sich die Frage, wie alle Kreuzungen durch ein möglichst kurzes, d.h. minimales Netz verbunden werden können.

Ermittelt den minimalen aufspannenden Baum aus dem Straßennetz mit dem Algorithmus von Jarnik, Prim und Dijkstra von Hand und erklärt schrittweise, wie ihr zu eurem Ergebnis gekommen seid.

Aufgabe 2 Sommer, Sonne, Routenplaner

Jede Reisende, die etwas auf sich hält, benutzt einen Routenplaner. Da Studierende bekanntermaßen nicht so viel Geld haben, müssen sie sich ihren selbst programmieren, und nutzen dazu natürlich gleich das, was sie gerade in der Vorlesung gehört haben, nämlich den Algorithmus von Dijkstra. Fehlen nur noch die Daten. Das Projekt *Open Street Map* bietet diese gratis an. Für die lokale Umgebung der Uni wurden diese Daten schon in Form der Dateien *nodes.txt* und *edges.txt* aufbereitet. Eine Datei *readme.txt* liegt auch dabei.

Generell dürft ihr in dieser Aufgabe Klassen aus der Laufzeitbibliothek verwenden. Achtet bei Sammlungen auf die Aufwandsklassen der Datenstrukturen.

Aufgabe 2.1 Karte aufbauen (30 %)

Erweitert die bereitgestellte Klasse *Map* so, dass ihr Konstruktor die beiden Dateien *nodes.txt* und *edges.txt* einliest und daraus einen Graphen konstruiert. Nutzt hierzu die ebenfalls bereitge-