

# Graphen

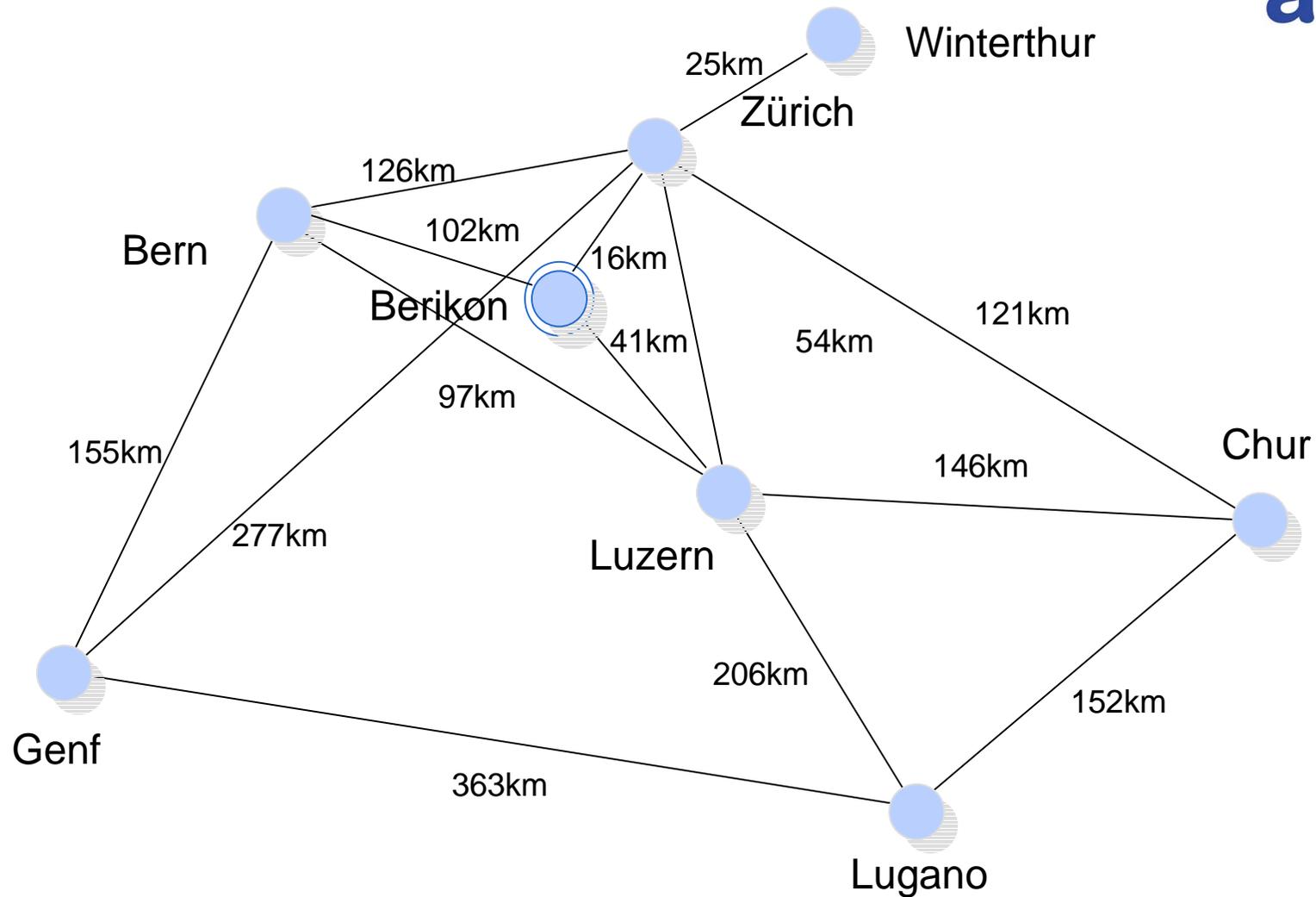


Graf = Lord(E) ... of the Rings

Im Prinzip richtig, aber:  
Bild zeigt keinen Graf sondern  
King of Rohan  
Graf = D-Titelbezeichnung;  
Lord = E-Anrede für niederen Adel  
u.a. Lord, Duke,....

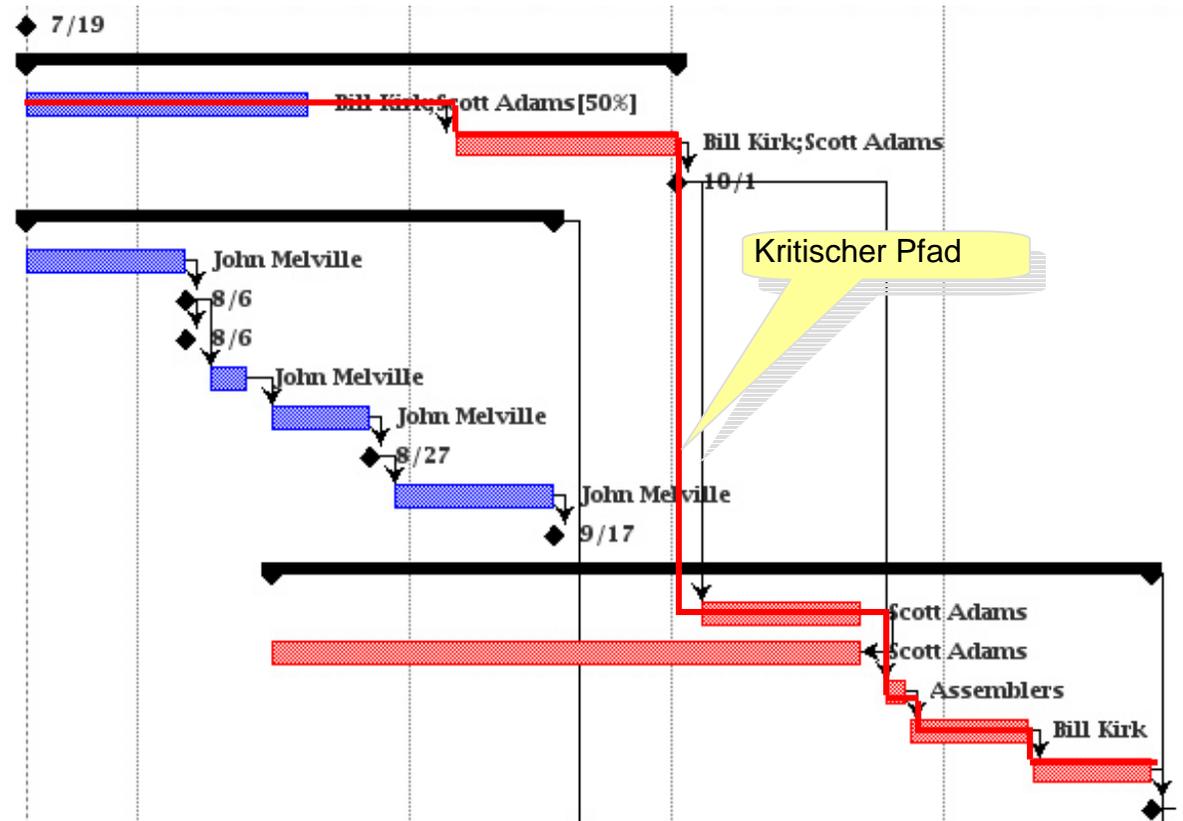
- Sie wissen wie Graphen definiert sind und kennen deren Varianten (nach Duden auch "Graf" Schreibweise erlaubt)
- Sie wissen wofür man sie verwendet
- Sie können Graphen in Java implementieren
- Sie kennen die Algorithmen und können sie auch implementieren: Tiefensuche, Breitensuche, kürzester Pfad, topologisches Sortieren

# Beispiel Strassennetz



■ Typische Aufgabe: finde kürzeste Verbindung zwischen zwei Orten

# Netzplan, Aufgabenliste



## ■ Typische Aufgabe:

- finde mögliche Reihenfolge der Tätigkeiten
- finde den kritischen Pfad

# Fragestellungen

- Kürzeste Verbindung (Verkehr, Postversand) von A nach B (shortest path)
- Reihenfolge von Tätigkeiten aus einem Netzplan erstellen (topological sort)
- Minimal benötigte Zeit bei optimaler Reihenfolge (critical path)
- Maximaler Durchsatz (Verkehr, Daten) von A nach B (maximal flow)
- kürzester Weg um alle Punkte anzufahren (traveling salesman)

## ■ Definition

Ein Graph  $G=(V,E)$  besteht aus einer endlichen Menge von *Knoten*  $V$  und einer Menge von *Kanten*  $E \subseteq V \times V$ .

## ■ Implementation

- Knoten: Objekte mit Namen und anderen Attributen
- Kanten: Verbindungen zwischen zwei Knoten u.U. mit Attributen

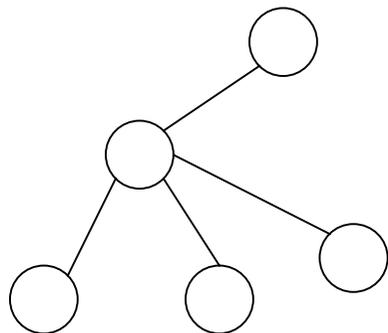
## ■ Hinweise:

- Knoten werden auch vertices bzw. vertex genannt
- Kanten heissen auch edges bzw. edge

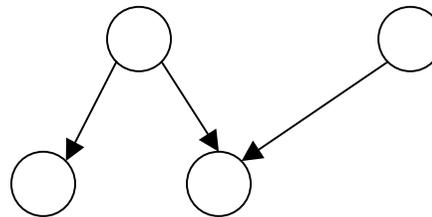
# Grapheigenschaften 1

## Knoten verbunden

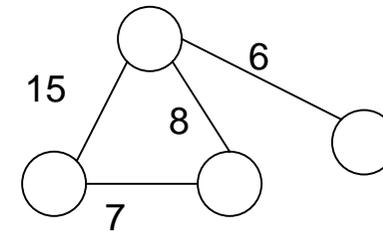
- Zwei Knoten  $x$  und  $y$  sind benachbart (**adjacent**), falls es eine Kante  $e_{xy} = (x,y)$  gibt.
- **Verbundener Graph** (connected graph) ist jeder Knoten mit jedem anderen Knoten verbunden = existiert eine Kante
- **Verbundener Teilgraph** Gesamter Graph besteht aus untereinander nicht verbundenen Teilgraphen
- **Kanten gerichtet und/oder mit Gewicht**



ungerichteter Graph



gerichteter Graph



gewichteter Graph

Ein Graph  $G = (V, E)$  kann zu einem **gewichteten Graphen**  $G = (V, E, g_w(E))$  erweitert werden, wenn man eine Gewichtsfunktion  $g_w: E \rightarrow \text{int}$  (oder  $g_w: E \rightarrow \text{double}$ ) hinzu nimmt, die jeder Kante  $e \in E$  ein **Gewicht**  $g_w(e)$  zuordnet.

## Eigenschaften

- Gewichtete Graphen haben **Gewichte** an den Kanten. z.B. Kosten
- **Gewichtete gerichtete** Graphen werden auch **Netzwerk** genannt.
- Die **gewichtete Pfadlänge** ist die Summe der Kosten der Kanten auf dem Pfad.
- Beispiel: Längenangabe (in km) zwischen Knoten (siehe einführendes Bsp).

# Ungerichtete Graphen

- Sei  $G = (V, E)$ .
- Falls für jedes  $e \in E$  mit  $e = (v_1, v_2)$  gilt:  $e' = (v_2, v_1) \in E$ , so heisst  $G$  ungerichteter Graph, ansonsten gerichteter Graph.
- Die Relation  $E$  ist im Fall ungerichteter Graphen symmetrisch.
- Bei einem ungerichteten Graphen gehört zu jedem Pfeil von  $x$  nach  $y$  auch ein Pfeil von  $y$  nach  $x$ .
- Daher lässt man Pfeile ganz weg und zeichnet nur ungerichtete Kanten.

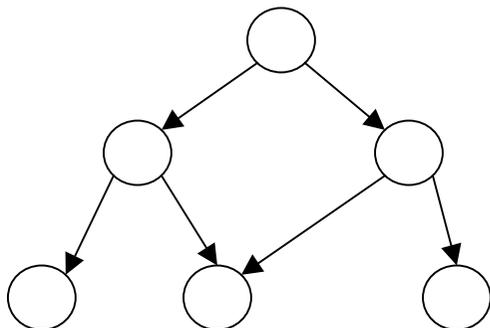
# Grapheneigenschaften 2

## Anzahl Kanten

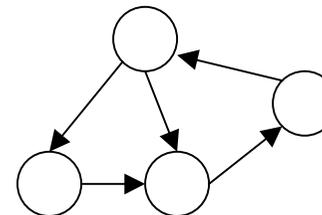
- **Kompletter Graph:** Jeder Knoten ist mit jedem anderen Knoten direkt verbunden.
- **Dichter Graph (dense):** Nur wenige Kanten im Graph (bezogen auf den kompletten Graphen) fehlen,
- **Dünnere Graph (sparse)** Nur wenige Kanten im Graph (bezogen auf den kompletten Graphen) sind vorhanden

# Pfade, Zyklen

- Eine Sequenz von benachbarten Knoten ist ein **einfacher Pfad**, falls kein Knoten zweimal vorkommt z.B.  $p = \{\text{Zürich, Luzern, Lugano}\}$ .
- Die **Pfadlänge** ist die Anzahl der **Kanten** des Pfads.
- Sind Anfangs- und Endknoten bei einem Pfad gleich, dann ist dies ein zyklischer Pfad oder **geschlossener Pfad** bzw. **Zyklus**.
- Graphen mit zyklischen Pfaden werden **zyklische Graphen** genannt.



gerichteter azyklischer Graph



gerichteter zyklischer Graph

# Übung

(B, C, A, D, A) ist ein \_\_\_\_\_ von B nach A.

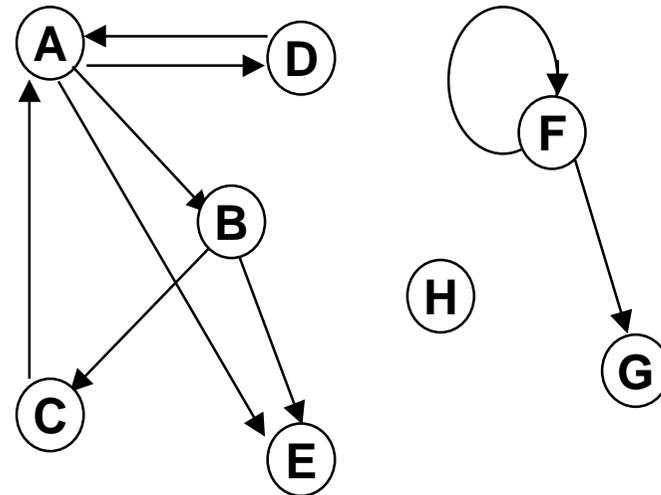
Er enthält einen \_\_\_\_\_: (A, D, A).

(C, A, B, E) ist einfacher \_\_\_\_\_ von C nach E.

(F, F, F, G) ist ein \_\_\_\_\_.

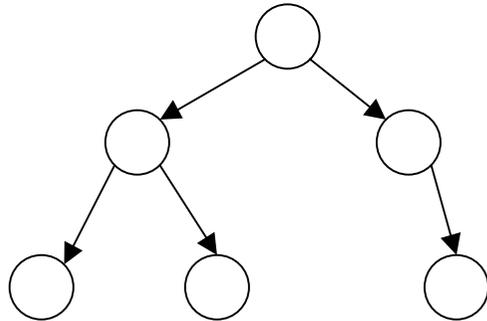
(A, B, C, A) und (A, D, A) und (F, F) sind die einzigen \_\_\_\_\_.

(A, B, E, A) ist kein \_\_\_\_\_ und kein \_\_\_\_\_.

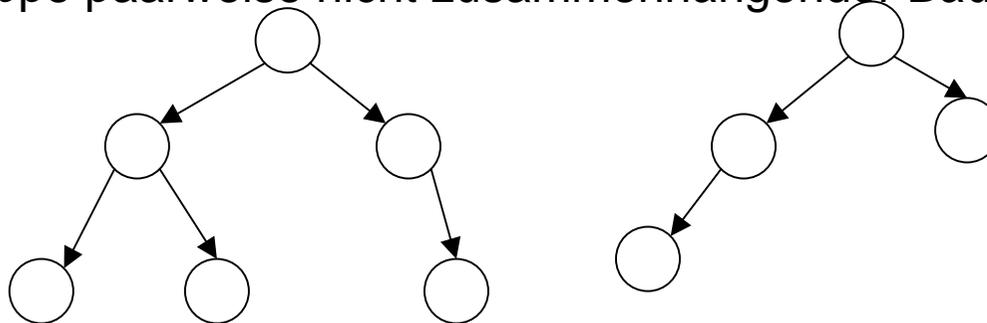


# Baum wird zum Spezialfall

- Ein Baum ein gerichteter, zyklener, verbundener Graph bei dem
  - Jeder Knoten genau eine eingehende Verbindung hat
  - Ein Knoten keine eingehenden Kanten hat (die Wurzel ist)

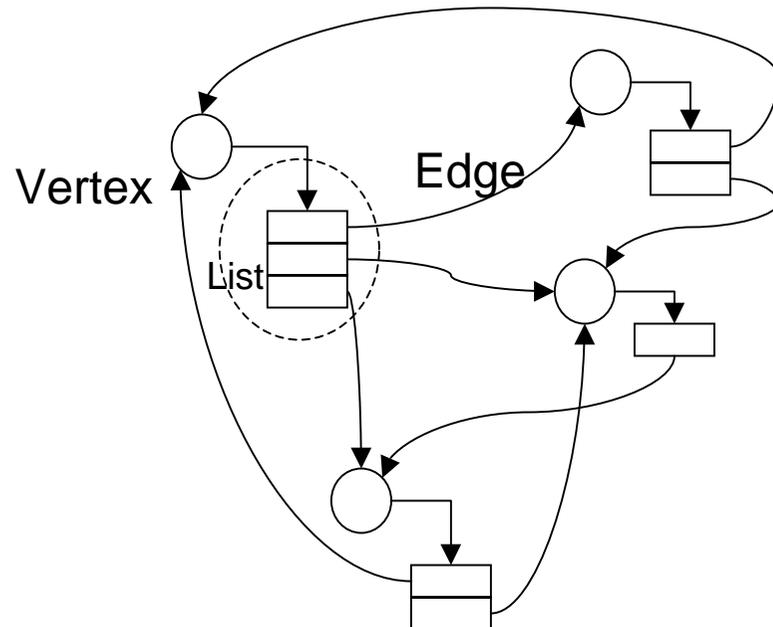


- Eine Gruppe paarweise nicht zusammenhängender Bäume heisst **Wald**.



# Implementation 1 : Adjazenz-Liste

- jeder Knoten führt (Adjazenz-) Liste, welche alle Kanten zu den benachbarten Knoten enthält



Dabei wird für jede Kante ein Eintrag bestehend aus dem Zielknoten und weiteren Attributen (z.B. Gewicht) erstellt. Jeder Knoten führt eine Liste der ausgehenden Kanten.

# Das Graph Interface

```
public interface Graph<N,E> {  
  
    // füge Knoten hinzu, tue nichts, falls Knoten schon existiert  
    public N addNode (String name) throws Throwable;  
  
    // finde den Knoten anhand seines Namens  
    public N findNode(String name);  
  
    // Iterator über alle Knoten des Graphen  
    public Iterable<N> getNodes();  
  
    // füge gerichtete und gewichtete Kante hinzu  
    public void addEdge(String source, String dest, double weight) throws Throwable;  
  
}
```

# Klasse GraphNode definiert einen Knoten

```
public class GraphNode<E> {  
    protected String name;    // Name  
    protected List<E> edges;  // Kanten  
  
    public GraphNode(){  
        edges = new LinkedList<E>( );  
    }  
  
    public GraphNode(String name){  
        this();  
        this.name = name;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    public Iterable<E> getEdges(){  
        return edges;  
    }  
  
    public void addEdge(E edge){  
        edges.add(edge);  
    }  
}
```

# Die Klasse Edge definiert eine Kante

```
public class Edge<N> {  
    protected N dest; // Zielknoten der Kante  
    protected double weight; // Kantengewicht  
  
    public Edge() {}  
  
    public Edge(N dest, double weight) {  
        this.dest = dest;  
        this.weight = weight;  
    }  
    public void setDest(N node) {  
        this.dest = node;  
    }  
    public N getDest() {return dest;}  
  
    public void setWeight(double w) {  
        this.weight = w;  
    }  
    double getWeight() {return weight;}  
}
```

# AdjListGraph Implementation

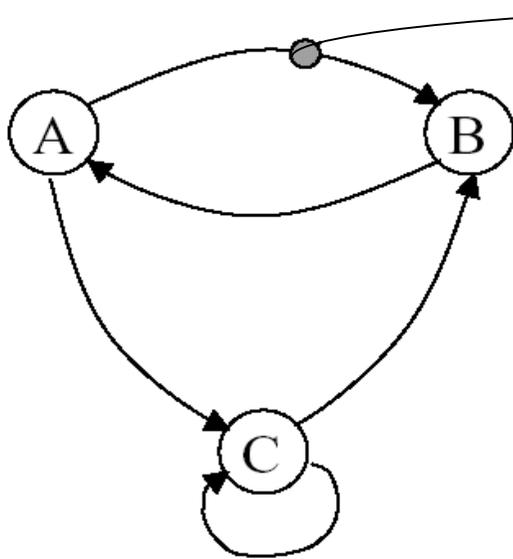
```
public class AdjListGraph<N extends Node,E extends EdgeE>
    implements Graph<N, E> {
    private final List<N> nodes = new LinkedList<N>();
    private final Class nodeClazz;
    private final Class edgeClazz;

    public AdjListGraph(Class nodeClazz, Class edgeClazz) {
        this.nodeClazz = nodeClazz;
        this.edgeClazz = edgeClazz;
    }
    // füge Knoten hinzu, gebe alten zurück falls Knoten schon existiert
    public N addNode(String name) throws Throwable {
        N node = findNode(name);
        if (node == null) {
            node = (N) nodeClazz.getConstructor(new Class[]{}).newInstance();
            node.setName(name);
            nodes.add(node);
        }
        return node;
    }
    ....
}
```

Klassen der Generischen Typen

Erzeuge Instanz

# Implementation 2 : Adjazenzmatrix



	A	B	C
A	false	true	true
B	true	false	false
C	false	true	true

■  $N \times N$  ( $N = \#Knoten$ ) boolean Matrix; true dort wo Verbindung existiert

# Adjazenzmatrix

- falls gewichtete Kanten -> Gewichte (double) statt boolean
- für jede Kante  $e_{xy} = (x,y)$  gibt es einen Eintrag
- sämtliche anderen Einträge sind 0 (oder undefined)
  
- **Nachteil:**
  - ziemlicher (Speicher-)Overhead
  
- **Vorteil:**
  - effizient
  - einfach zu implementieren
  - gut (speichereffizient) falls der Graph dicht

# Adjazenzmatrix

- Distanzentabelle ist eine Adjazenzmatrix
- ungerichtet -> symmetrisch zur Nebendiagonalen
- im Prinzip reicht die eine Hälfte -> Dreiecksmatrix

Entfernung in Kilometer

	Atlanta	Boston	Chicago	Denver	Houston	Las Vegas	Los Angeles	Miami	New Orleans	New York	Phoenix	Portland	Salt Lake City	San Francisco	Seattle	Washington
Atlanta		1'778	1'166	2'269	1'263	3'279	3'578	1'175	749	1'442	2'993	1'080	3'114	4'117	4'356	1'397
Boston	1'778		1'583	3'155	2'963	4'381	4'800	2'474	2'449	338	4'282	1'205	3'803	4'982	4'909	435
Chicago	1'166	1'583		1'602	1'900	2'827	3'246	2'327	1'496	1'287	2'885	400	2'250	3'429	3'332	1'333
Denver	2'269	3'155	1'602		1'694	1'232	1'650	3'707	2'251	2'858	1'470	1'971	851	2'130	2'144	2'886
Houston	1'263	2'963	1'900	1'694		2'356	2'505	2'017	560	2'627	1'920	2'139	2'653	3'114	3'945	2'609
Las Vegas	3'279	4'381	2'827	1'232	2'356		440	4'401	2'944	4'084	444	3'197	693	920	2'051	4'103
Los Angeles	3'578	4'800	3'246	1'650	2'505	440		4'514	3'057	4'479	607	3'615	1'111	618	1'841	4'471
Miami	1'175	2'474	2'327	3'707	2'017	4'401	4'514		1'486	2'136	3'911	2'202	4'262	5'106	5'503	2'102
New Orleans	749	2'449	1'496	2'251	560	2'944	3'057	1'486		2'112	2'471	1'625	3'102	3'665	4'395	2'095
New York	1'442	338	1'287	2'858	2'627	4'084	4'479	2'136	2'112		3'944	908	3'507	4'686	4'612	99
Phoenix	2'993	4'282	2'885	1'470	1'920	444	607	3'911	2'471	3'944		3'201	1'115	1'215	2'491	3'939
Portland	1'080	1'205	400	1'971	2'139	3'197	3'615	2'202	1'625	908	3'201		2'621	3'800	3'727	954
Salt Lake City	3'114	3'803	2'250	851	2'653	693	1'111	4'262	3'102	3'507	1'115	2'621		1'181	1'366	3'554
San Francisco	4'117	4'982	3'429	2'130	3'114	920	618	5'106	3'665	4'686	1'215	3'800	1'181		1'308	4'731
Seattle	4'356	4'909	3'332	2'144	3'945	2'051	1'841	5'503	4'395	4'612	2'491	3'727	1'366	1'308		4'657
Washington	1'397	435	1'333	2'886	2'609	4'103	4'471	2'102	2'095	99	3'939	954	3'554	4'731	4'657	

# Vergleich der Implementierungen

■ Alle hier betrachteten Möglichkeiten zur Implementierung von Graphen haben ihre spezifischen Vor- und Nachteile:

	Vorteile	Nachteile
Adjazenzmatrix	Berechnung der Adjazenz sehr effizient	hoher Platzbedarf und teure Initialisierung: wachsen quadratisch mit $O(n^2)$
Adjazenzliste	Platzbedarf ist proportional zu $n+m$	Effizienz der Kantensuche abhängig von der mittleren Anzahl ausgehender Kanten pro Knoten

■ **n** ist dabei die Knotenzahl und **m** die Kantenzahl eines Graphen **G**.

# Graph Algorithmen

## Traversierungen

- Tiefensuche (depth-first search)
- Breitensuche (breadth-first search)

## häufige Anwendungen

- Ungewichteter kürzester Pfad (unweighted shortest path)
- Gewichteter kürzester Pfad (positive weighted shortest path)
- Topologische Sortierung (topological sorting)

## weitere Algorithmen

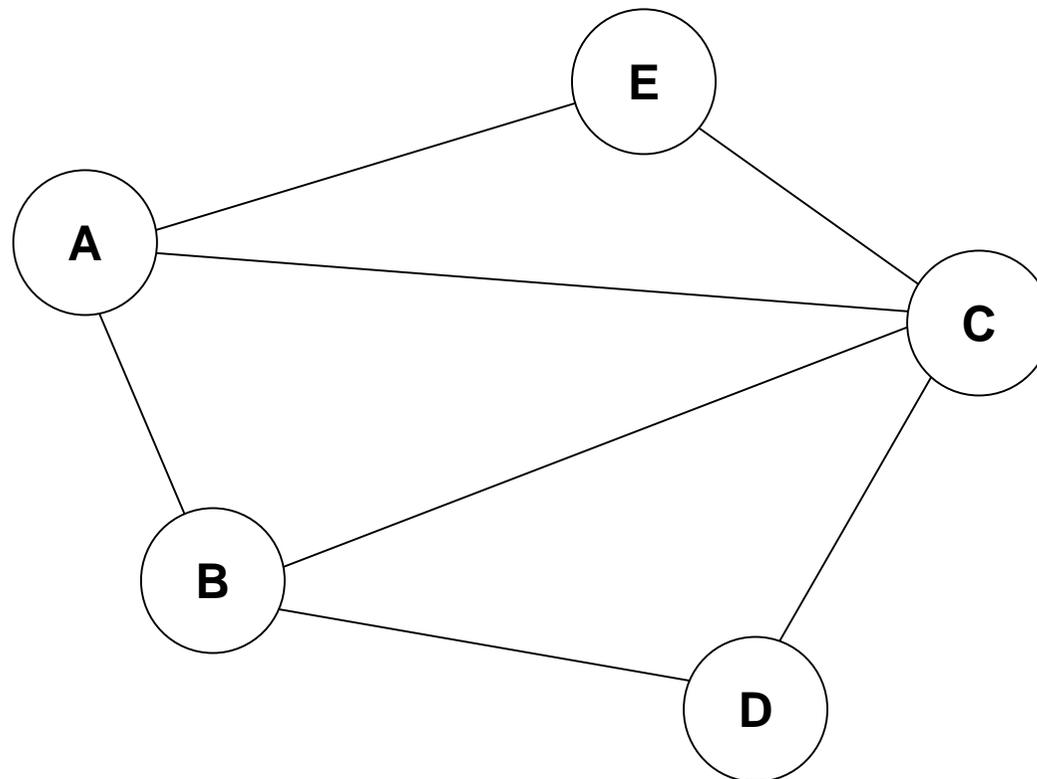
- Maximaler Fluss
- Handlungsreisender (traveling salesman)
- .....

# Graphentraversierungen

# Grundformen: "Suchstrategien"

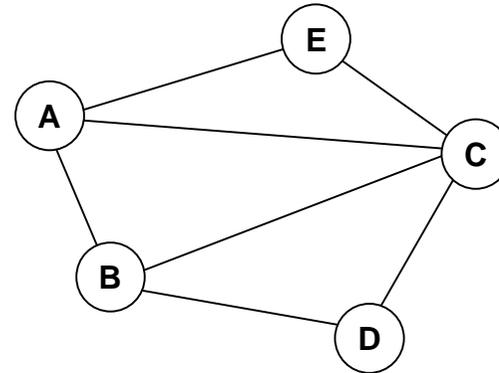
- Genau wie bei den Traversierungen bei Bäumen, sollen bei Graphen die Knoten **systematisch besucht** werden.
- Es werden im Wesentlichen zwei grundlegende "Suchstrategien" unterschieden.
- **Tiefensuche: (depth-first)**
  - Ausgehend von einem Startknoten geht man **vorwärts (tiefer)** zu einem neuen unbesuchten Knoten, solange einer vorhanden (d.h. erreichbar) ist. Hat es keine weiteren (unbesuchten) Knoten mehr, geht man rückwärts und betrachtet die noch unbesuchten Knoten. Entspricht der **Preorder** Traversierung bei Bäumen.
- **Breitensuche: (breadth-first)**
  - Ausgehend von einem Startknoten betrachtet man zuerst **alle benachbarten Knoten** (d.h. auf dem gleichen Level), bevor man einen Schritt weitergeht. Entspricht der **Levelorder** Traversierung bei Bäumen.

- Auf welche Arten kann folgender Graph in Tiefensuche durchsucht werden (Start bei A)



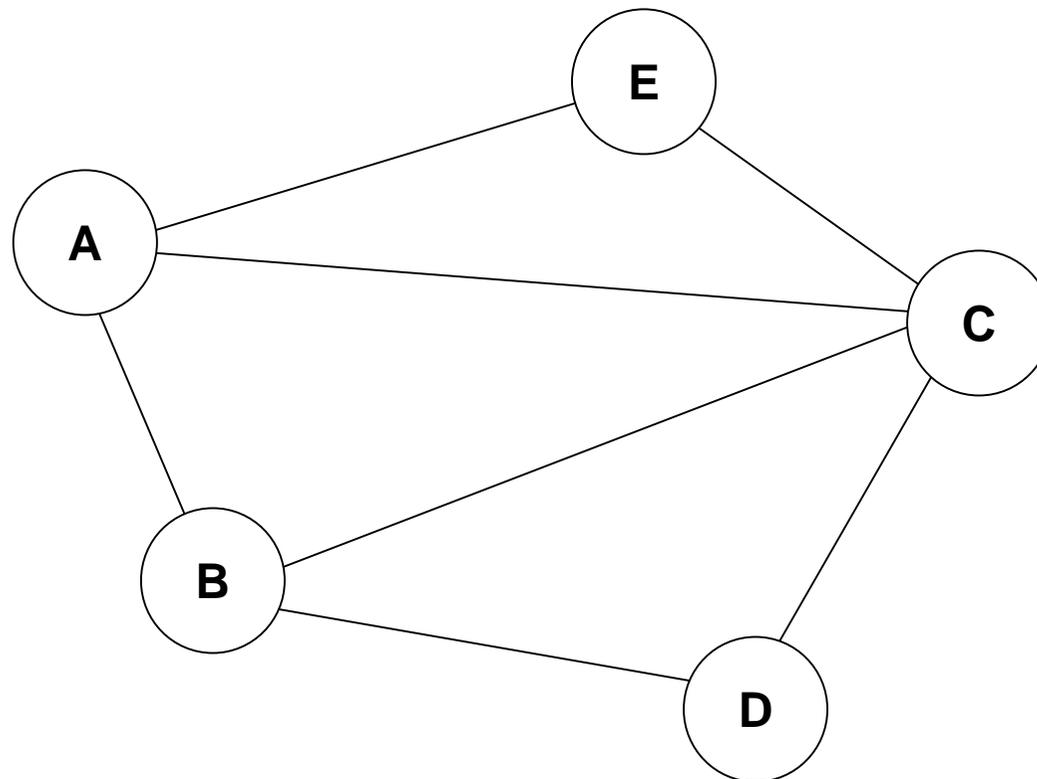
# Tiefensuche (Pseudo Code)

```
void depthFirstSearch()
  s = new Stack();
  mark startNode;
  s.push(startNode)
  while (!s.empty()) {
    currentNode = s.pop()
    print currentNode
    for all nodes n adjacent to currentNode {
      if (!(marked(n))) {
        mark n
        s.push(n)
      }
    }
  }
}
```



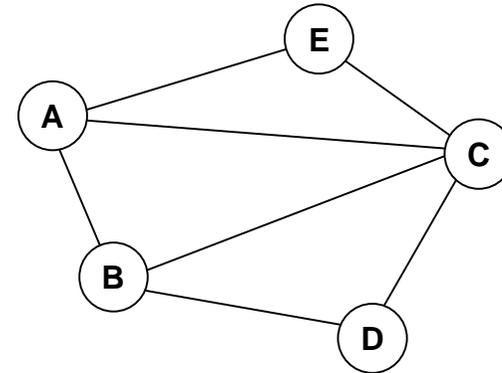
- Am Anfang sind alle Knoten nicht markiert. Knoten, die besucht wurden, liegen auf dem Stack

- Auf welche Arten kann folgender Graph in Breitensuche durchsucht werden

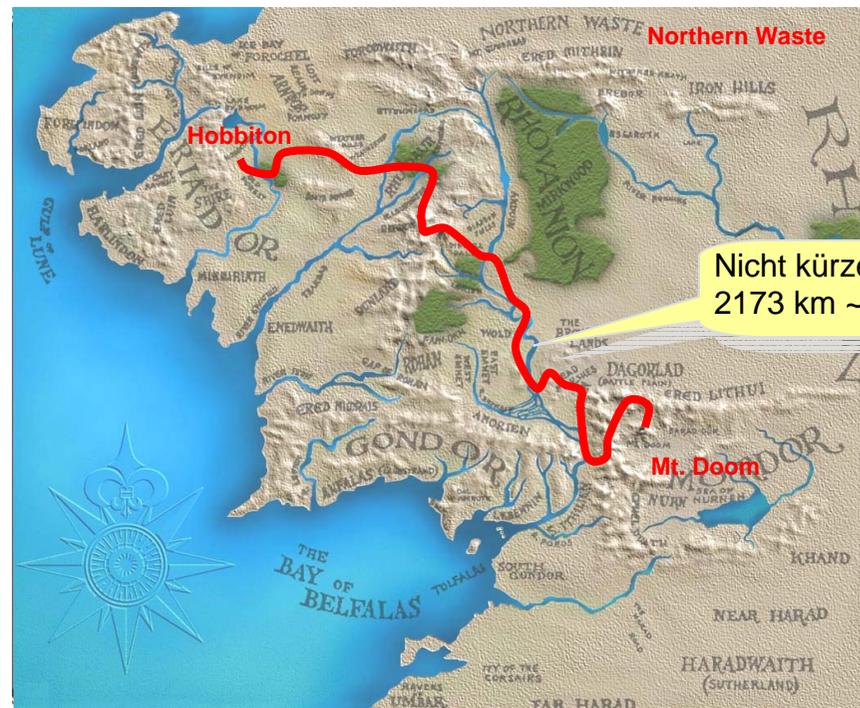


# Breitensuche (Pseudo Code)

```
void breadthFirstSearch()  
    q = new Queue()  
    mark startNode  
    q.enqueue(startNode)  
    while (!q.empty()) {  
        currentNode = q.dequeue()  
        print currentNode  
        for all nodes n adjacent to currentNode {  
            if (!(marked(n))) {  
                mark n  
                q.enqueue(n)  
            }  
        }  
    }  
}
```



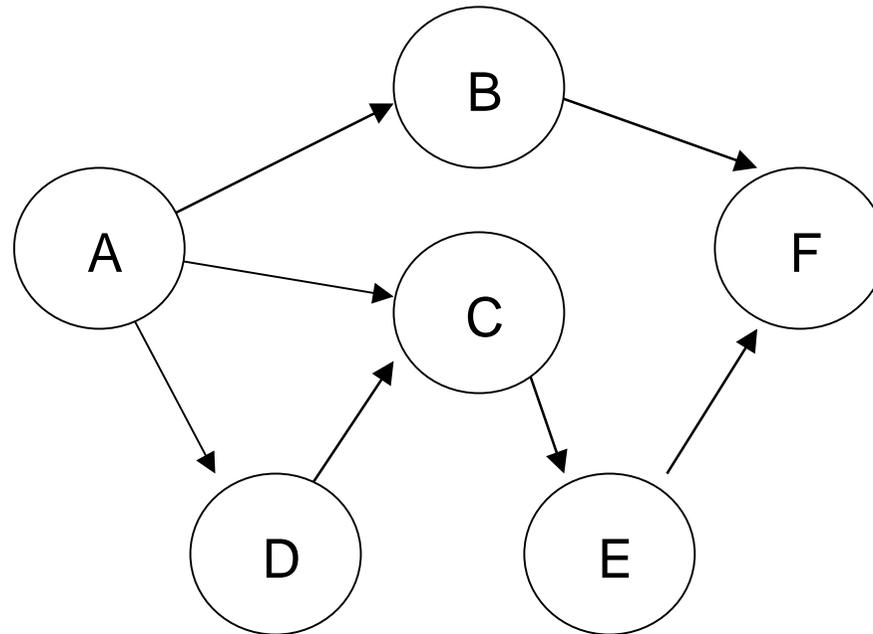
## Kürzester Pfad



Mittelerde

# Kürzester Pfad 1: alle Kanten gleiches Gewicht

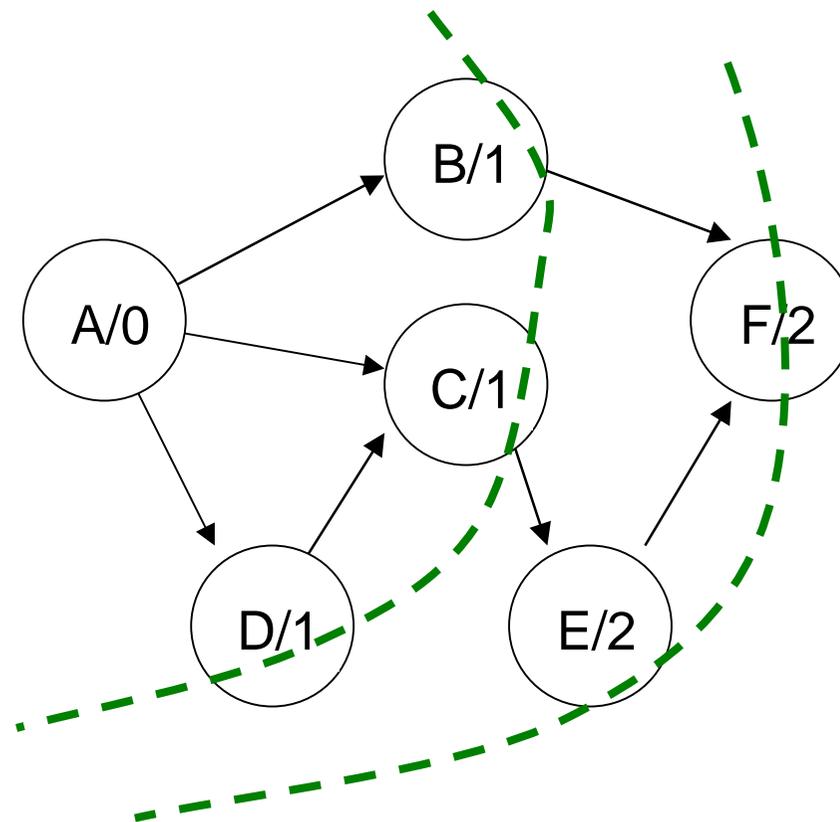
- Gesucht ist der kürzeste Weg von einem bestimmten Knoten aus zu jeweils einem anderen (z.B A nach F)



- Lösung : A nach F über B

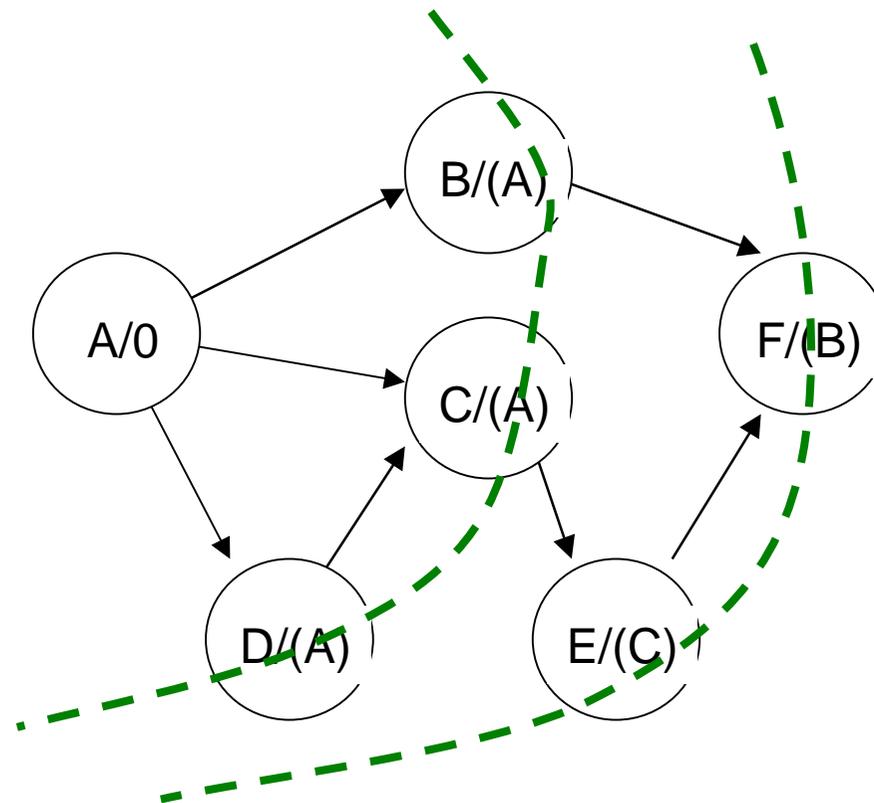
# Algorithmus kürzester ungewichteter Pfad

- Vom Startpunkt ausgehend werden die Knoten mit ihrer Distanz markiert.



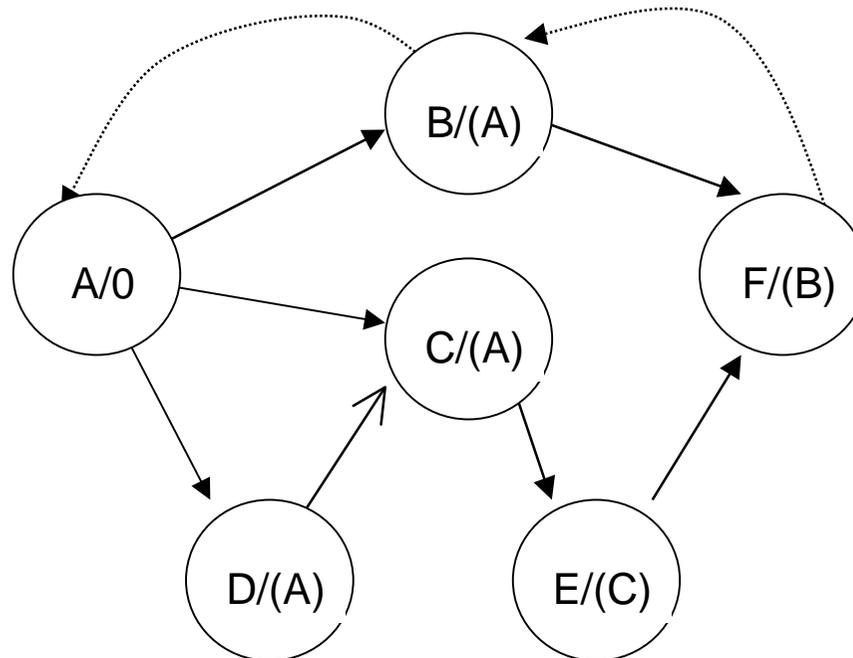
## ... Algorithmus kürzester ungewichteter Pfad

- Gleichzeitig wird noch eingetragen, von welchem Knoten aus der Knoten erreicht wurde



## ... Algorithmus kürzester ungewichteter Pfad

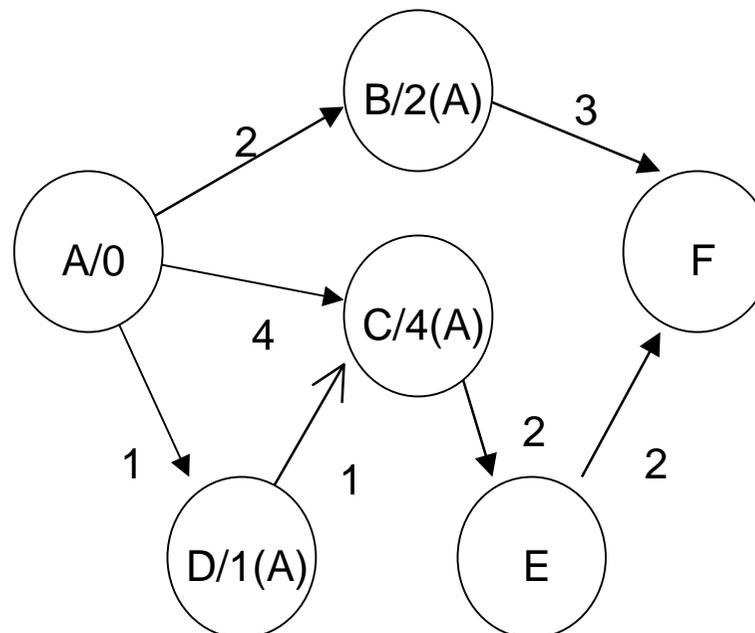
- Vom Endpunkt aus kann dann rückwärts der kürzeste Pfad gebildet werden



## 2. Kürzester Pfad bei gewichteten Kanten:

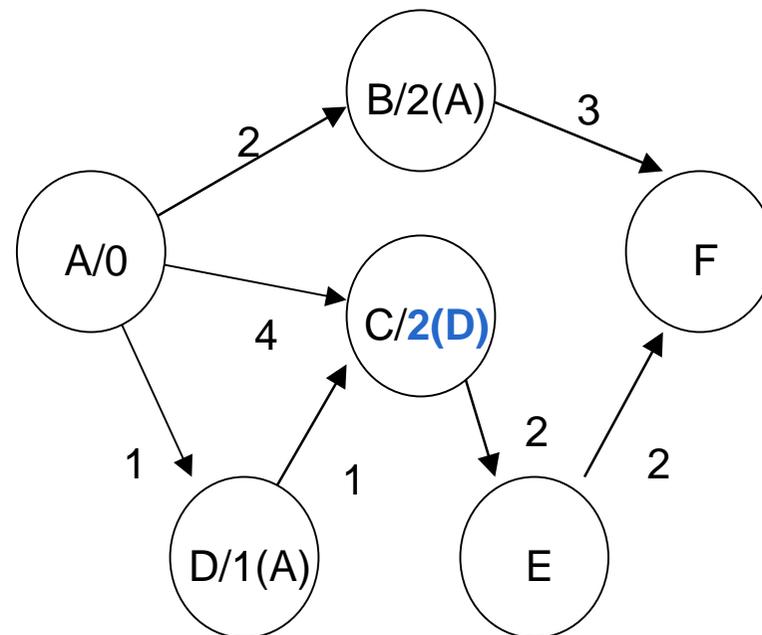
■ C ist über D schneller zu erreichen als direkt

Algorithmus: gleich wie vorher, aber **korrigiere Einträge** für Distanzen



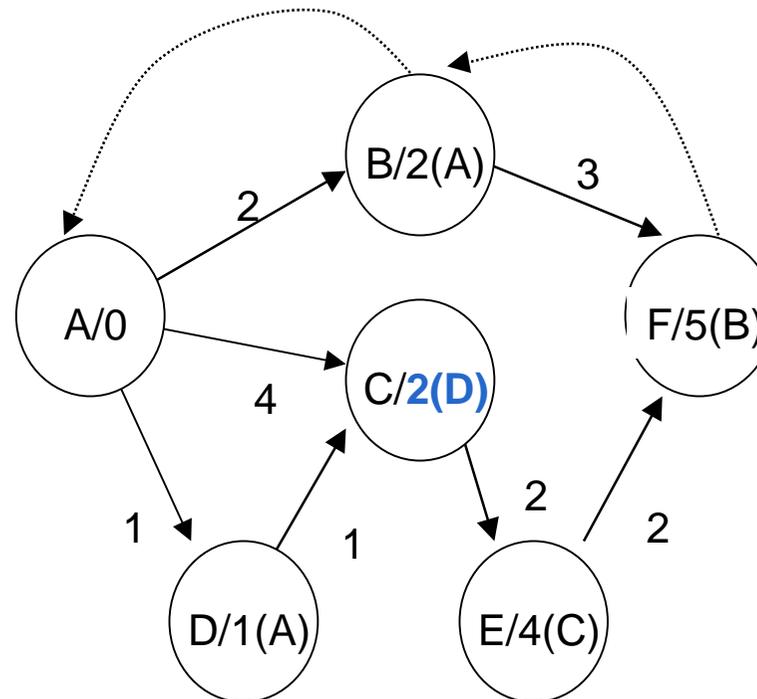
## ... Kürzester Pfad bei gewichteten Kanten

- Der Eintrag für C wird auf den neuen Wert gesetzt; statt "markiert" gehe so lange weiter, bis der neue Weg länger als der angetroffene ist.



## ... Kürzester Pfad bei gewichteten Kanten

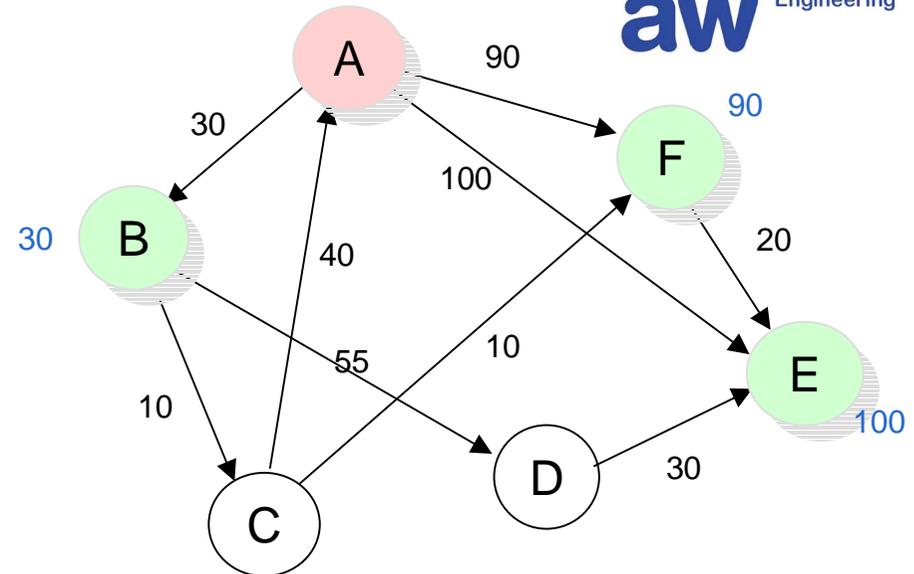
E und F werden normal behandelt. Der Pfad kann wie vorher rückwärts gebildet werden



# Dijkstras Algorithmus

- Teilt die Knoten in 3 Gruppen auf

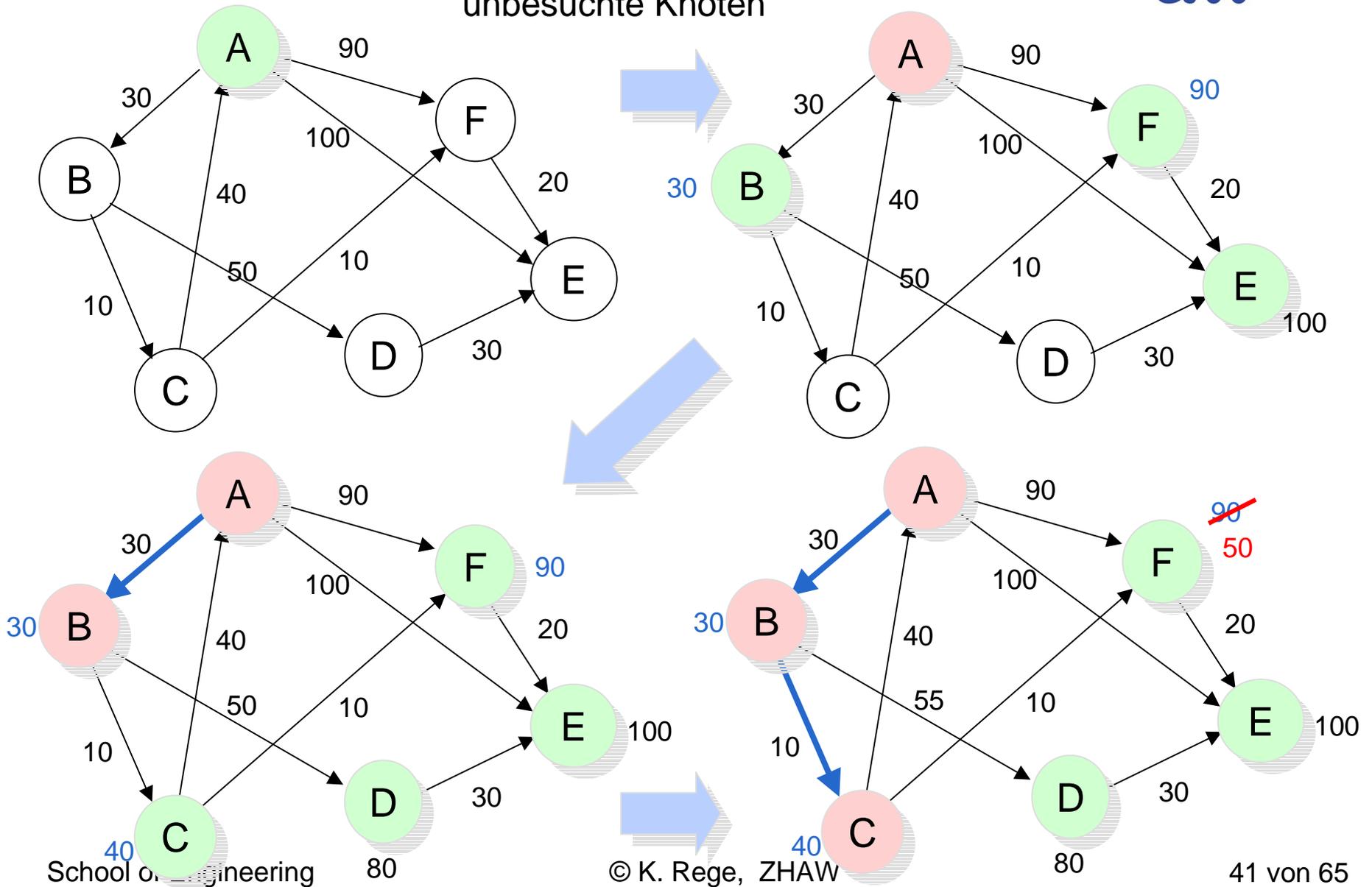
- **besuchte Knoten**
- **benachbart zu besuchtem Knoten**
- *unbesehene Knoten (der Rest)*



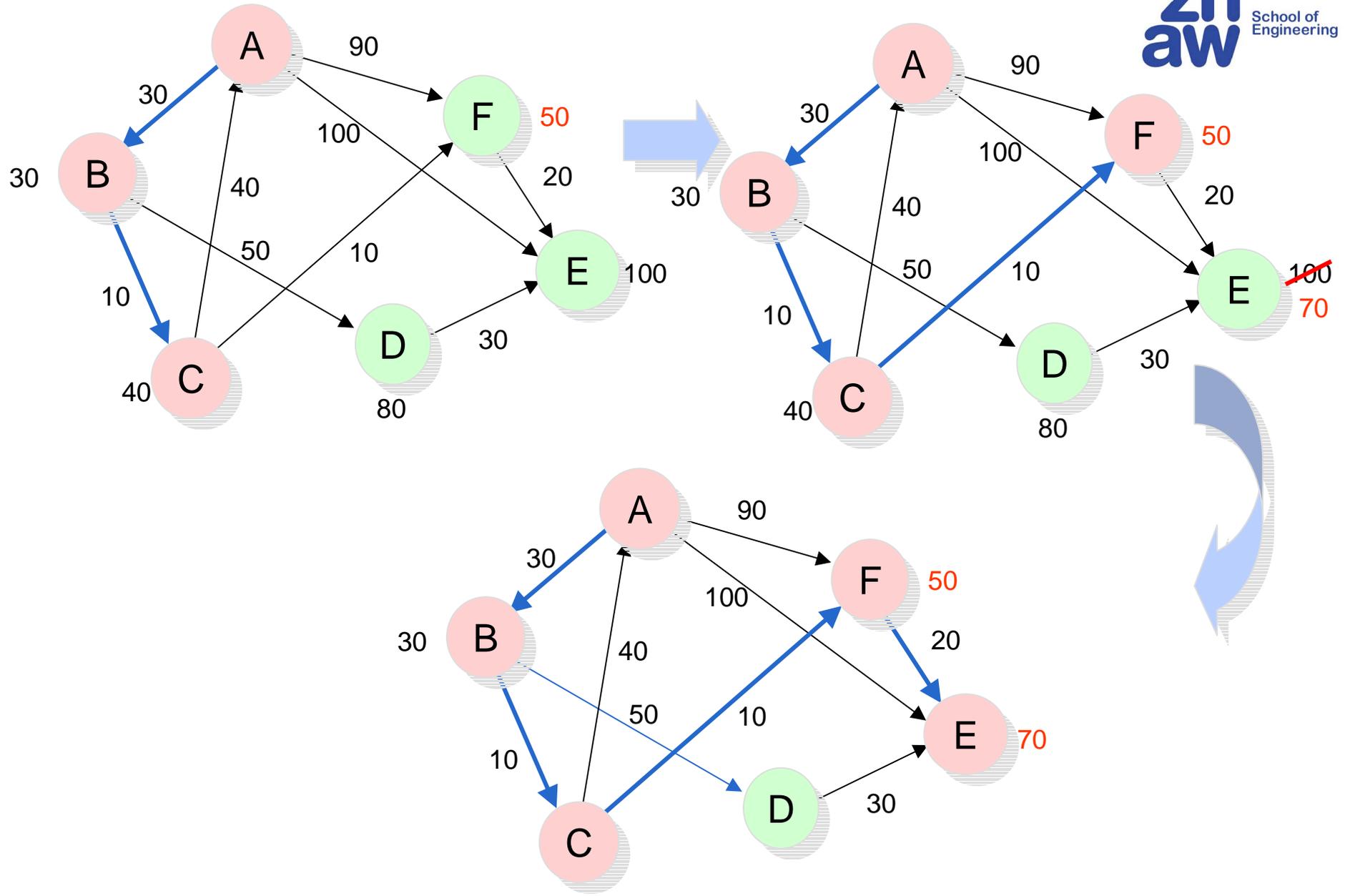
- Suche unter den **benachbarten Knoten** denjenigen, dessen Pfad zum Startknoten das **kleinste Gewicht** (=kürzeste Distanz) hat.

- **Besuche diesen** und bestimme dessen **benachbarte Knoten**

markierte Knoten  
 Knoten in Queue  
 unbesuchte Knoten



letzter Stand



# Pseudocode nach Dijkstras

```
for all nodes n in G {
    n.mark = black;           // Knoten noch unbesehen
    n.dist = inf;             // Distanz zum Startknoten
    n.prev = null;           // Vorgängerknoten in Richtung Start
}
dist(start) = 0;
current = start;
start.mark = red;
for all nodes in RED {
    current = findNodeWithSmallestDist();
    current.mark = green;
    for all n in succ(current) {
        if (n.mark != green) {
            n.mark = red;
            if (n == goal) return;
            dist = current.dist+edge(current, n);
            if (dist < n.dist) {
                n.dist = dist;
                n.prev = current;
            }
        }
    }
}
}
```

Schritt 1: Nur der Startknoten ist rot. Hat kleinste Distanz. Grün markiert, d.h. kleinste Distanz gefunden. Alle von ihm ausgehenden Knoten werden rot markiert und die Distanz zum Startknoten eingetragen.

Schritt 2: Der Knoten mit kleinster Distanz kann grün markiert werden. Um auf einem andern Weg zu ihm zu gelangen, müsste man über einen andern roten Knoten mit grösserer Distanz gehen.

Markieren alle von diesem direkt erreichbaren Knoten rot.

Schritt n: In jedem Schritt wird ein weiterer Knoten grün. Dabei kann sich die Distanz der roten Knoten ändern.

Demo-Applet: <http://www-b2.is.tokushima-u.ac.jp/~ikeda/suuri/dijkstra/DijkstraApp.shtml?demo6>





outlook

# Spannbaum

# Spannbaum (Spanning Tree)



outlook

## Definitionen

- Ein *Spannbaum* eines Graphen ist ein Baum, der alle Knoten des Graphen enthält.
- Ein minimaler Spannbaum (minimum spanning tree) ist ein Spannbaum eines gewichteten Graphen, sodass die Summe aller Kanten minimal ist.

## Algorithmus

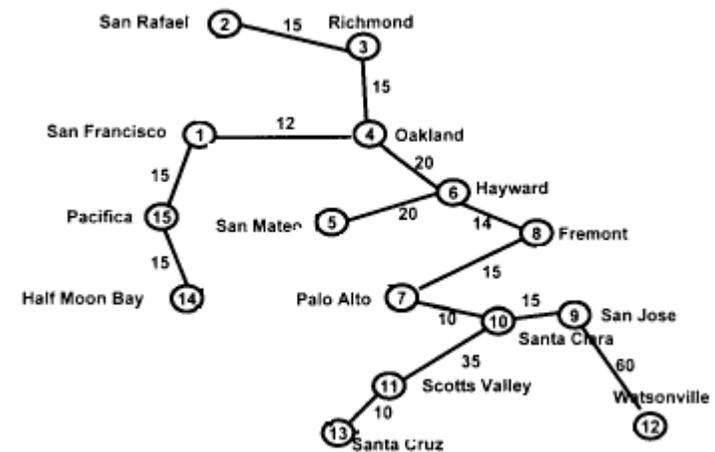
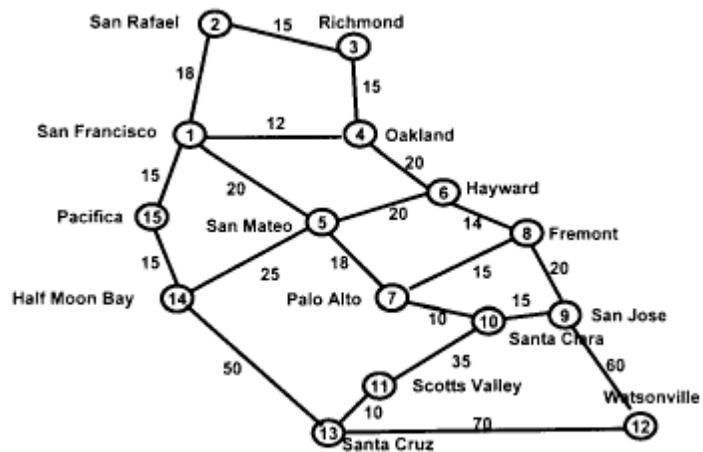
- z.B. Prim-Jarnik
- Prim-Jarnik ist ähnlich wie Dijkstras Algorithmus

# Anwendung



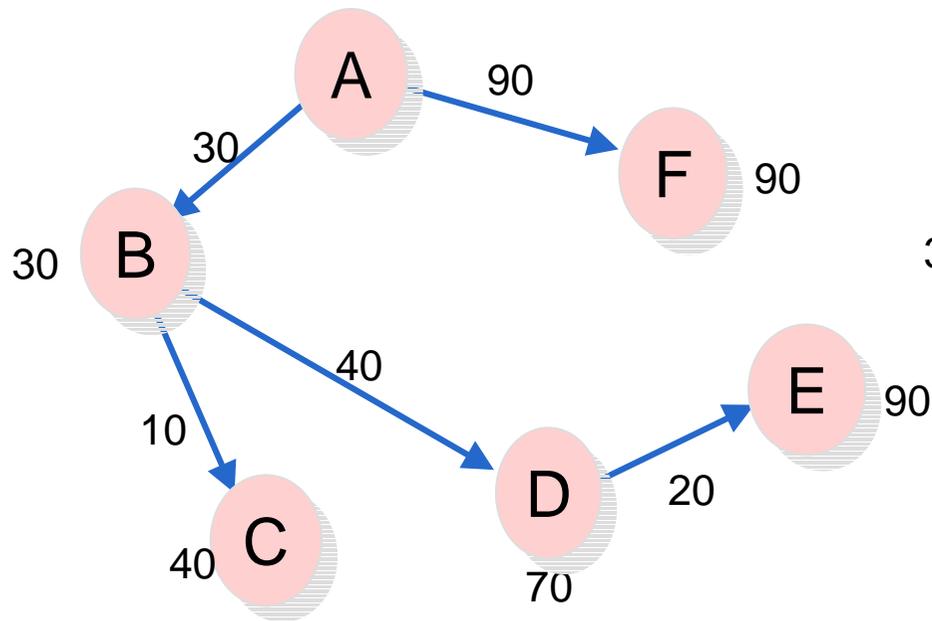
outlook

- Gesucht Netz mit minimaler Streckenlänge, das alle Städte verbindet



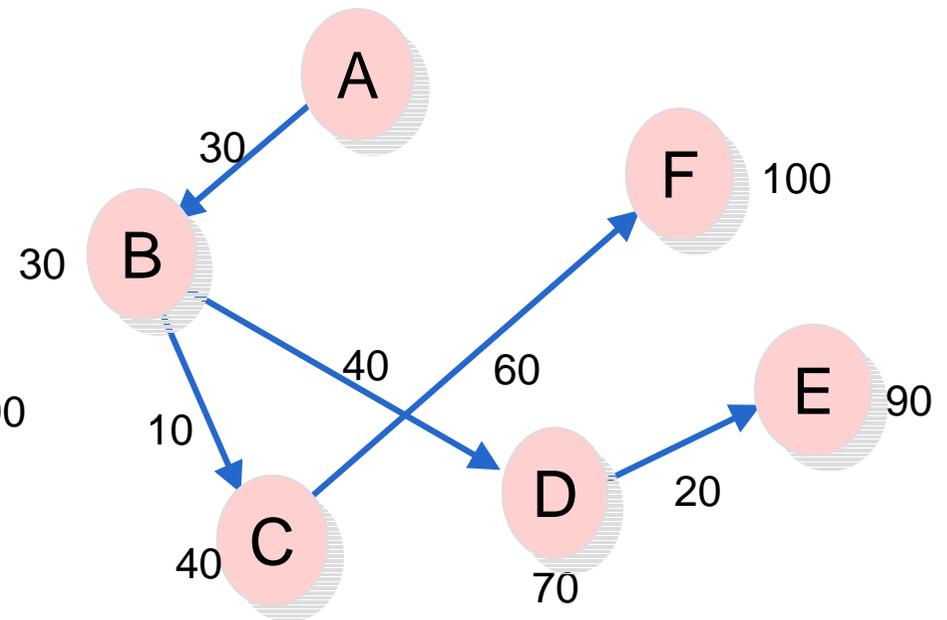
# Shortest Path vs. Minimum Spanning Tree

Dijkstra: Shortest path



Total = 190  
Path A-F = 90

Prim-Jarnik: Min Spanning Tree

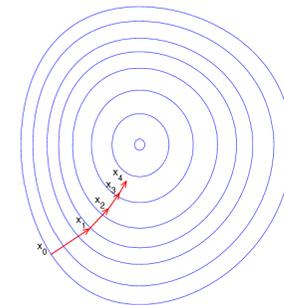


Total = 160  
Path A-F = 100

# Greedy Algorithms

# Gierige Algorithmen (Greedy)

- Spezielle Klasse von Algorithmen
- Sie zeichnen sich dadurch aus, dass sie den Folgezustand auswählen, der zum Zeitpunkt der Wahl den grössten Gewinn bzw. das beste Ergebnis berechnet durch eine (lokale) Bewertungsfunktion verspricht.
- Greedy-Algorithmen
  - sind oft schnell: z.B. Dijkstra
  - können aber in lokalen Maxima stecken bleiben

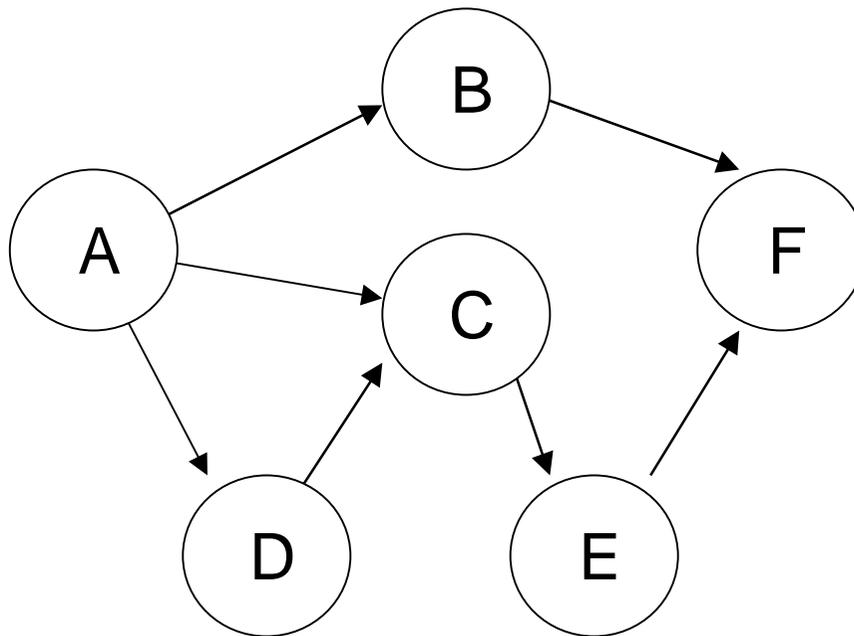


Gradientenverfahren

# Topologisches Sortieren

# Sortierung eines gerichteten Graphen: topologisches Sortieren

- Die Knoten eines gerichteten, unzyklischen Graphs in einer korrekten Reihenfolge auflisten



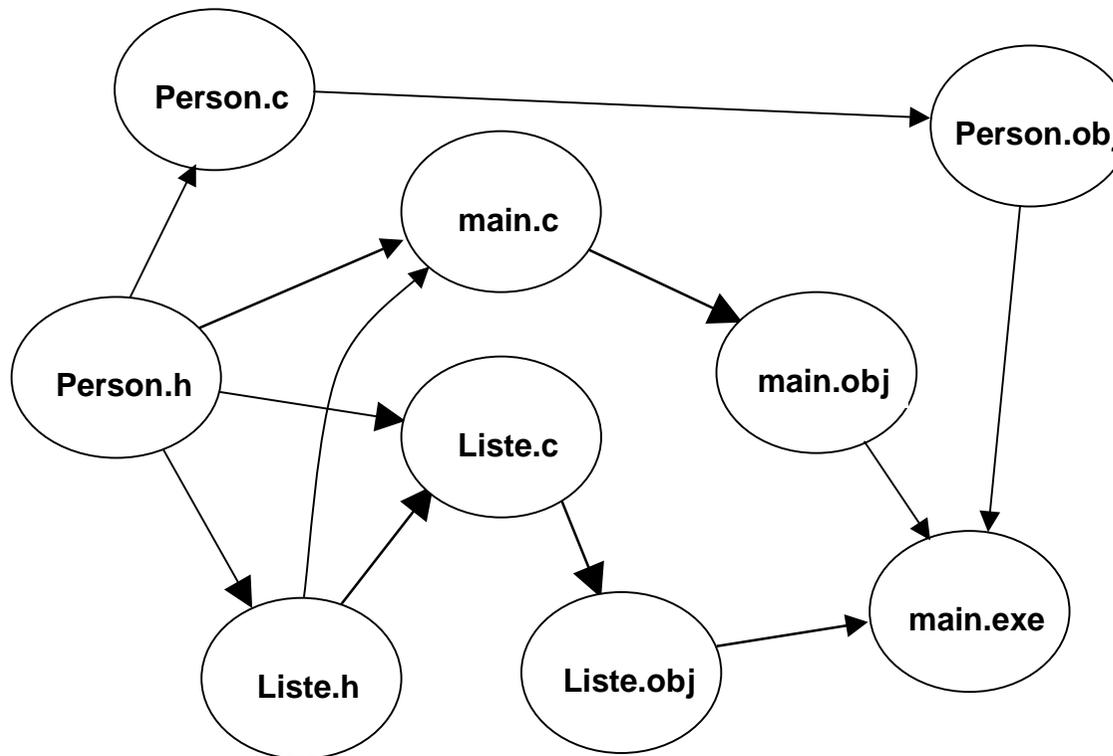
Anwendung:  
Die Kanten geben  
Abhängigkeiten  
zwischen Modulen in einem  
Programm an.

Topologisches Sortieren  
zeigt eine mögliche  
Compilationsreihenfolge.

- Lösung : A B D C E F oder A D C E B F

# Topologisches Sortieren, Beispiel

- Compilations-Reihenfolge cc: c-> obj; cl: obj->exe

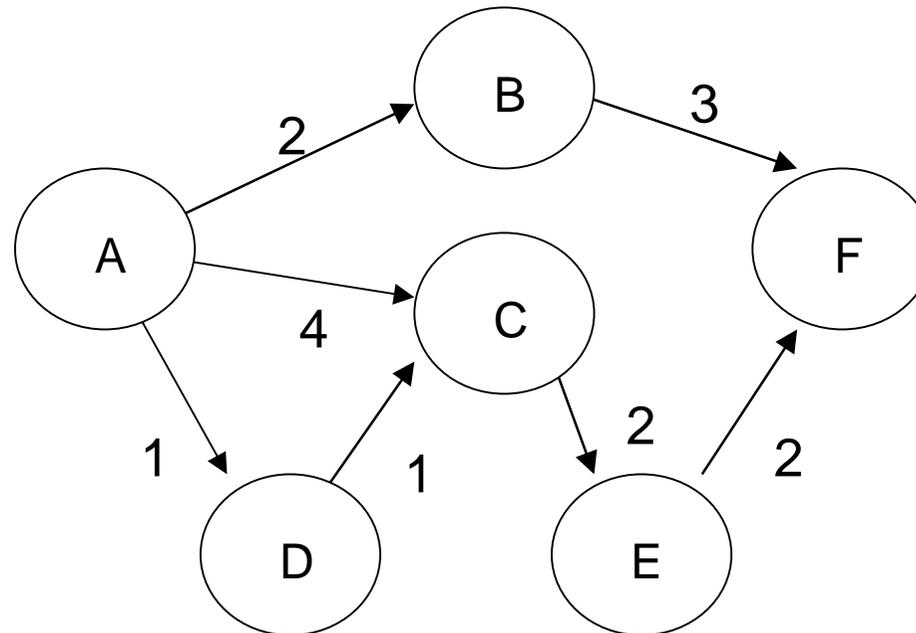


- Beschreiben Sie einen Algorithmus (in Pseudocode), der eine korrekte geordnete Auflistung eines azyklischen gerichteten Graphen liefert.
  
- Hinweis: Zähle die eingehenden Kanten; welche können ausgegeben werden?

# Maximaler Fluss

# Maximaler Fluss

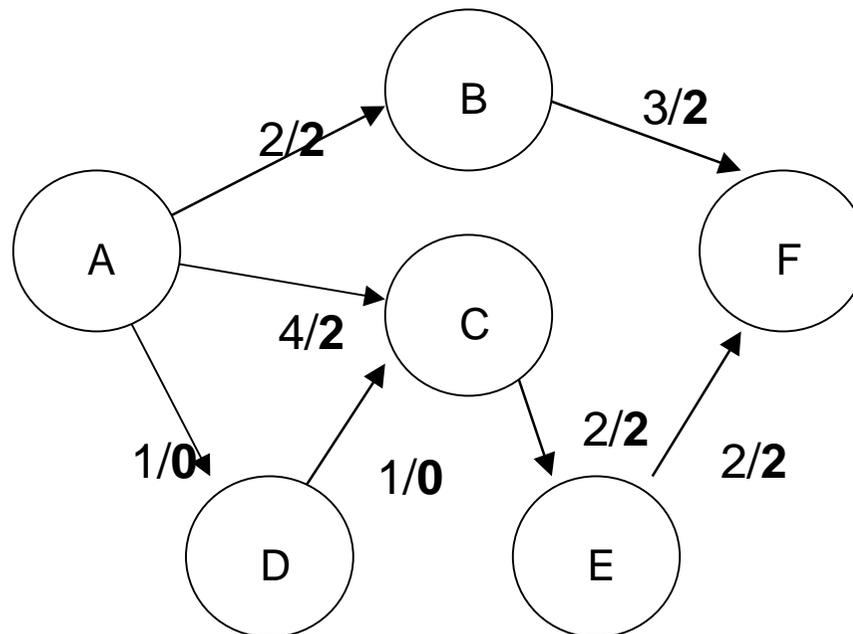
- Die Kanten geben den maximalen Fluss zwischen den Knoten an. Wieviel fließt von A nach F ?



- Hinweis: Was in einen Knoten hinein fließt, muss auch wieder heraus

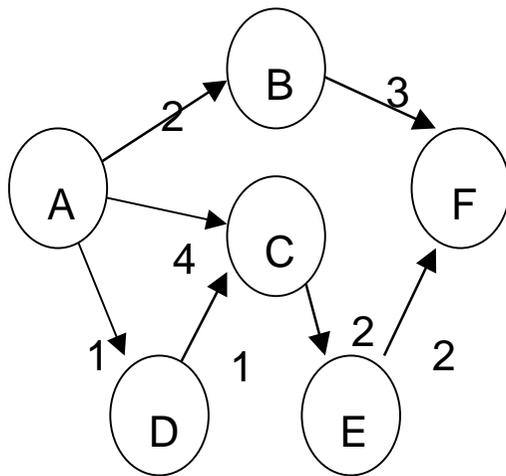
# Maximaler Fluss

■ Resultat : 4

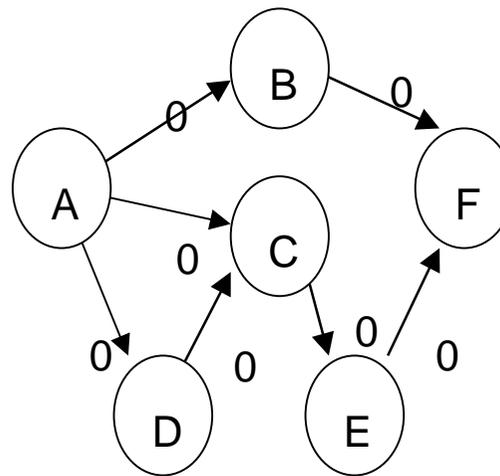


# Lösungsidee maximaler Fluss

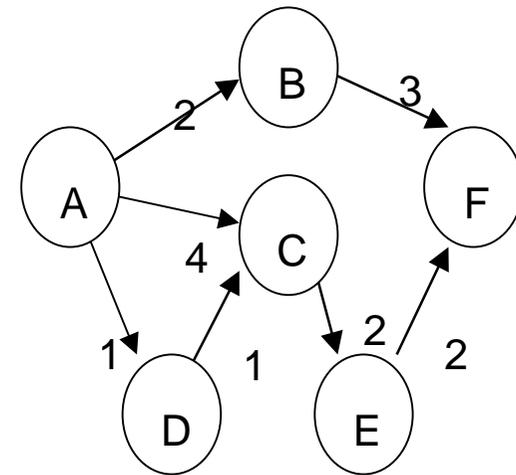
■ Noch 2 zusätzliche Versionen des Graphen



Original



Vorläufiger Fluss

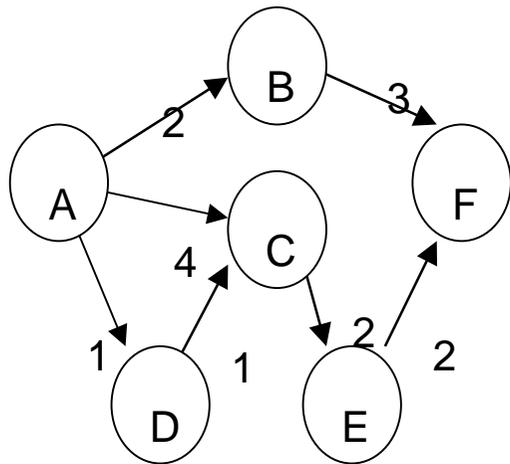


Rest Fluss

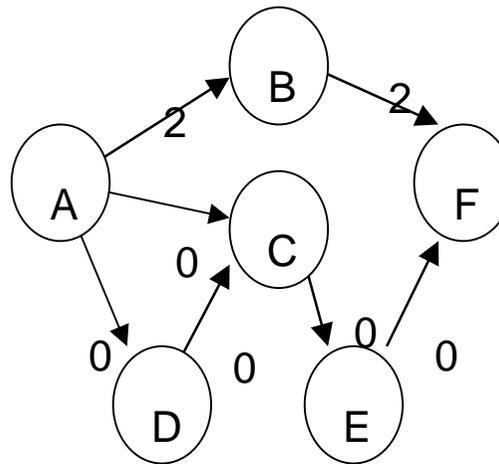
# Lösungsidee maximaler Fluss

■ Pfad A B F einfügen : Fluss 2

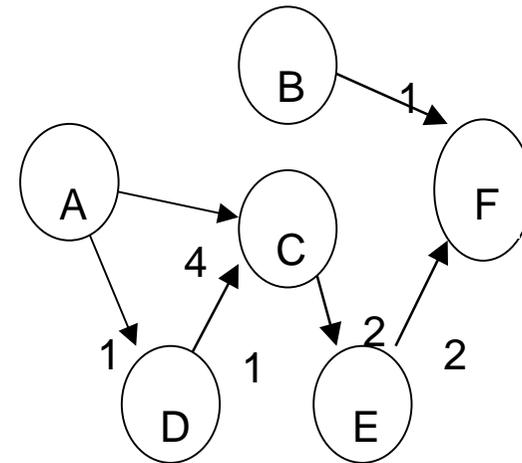
Kante A B löschen



Original



Vorläufiger Fluss

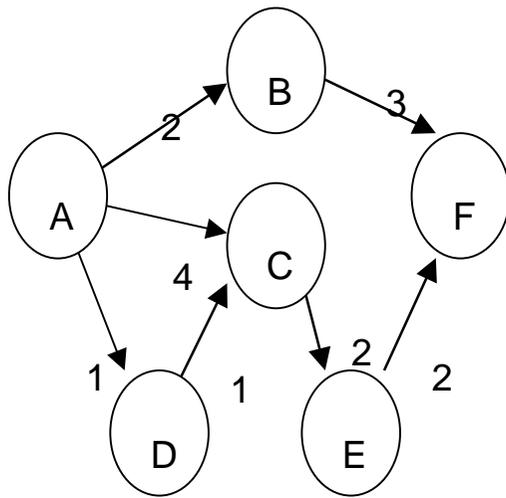


Rest Fluss

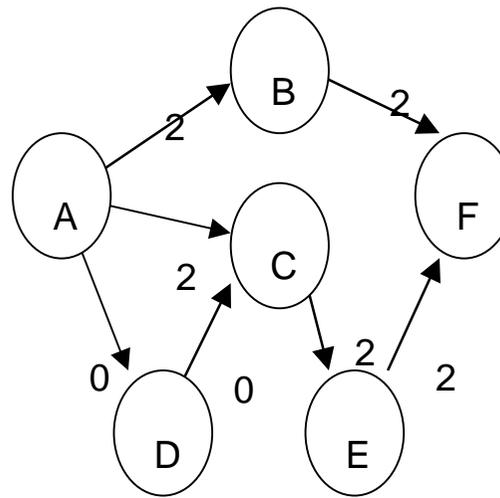
# Lösungsidee maximaler Fluss

■ Pfad A C E F : Fluss 2

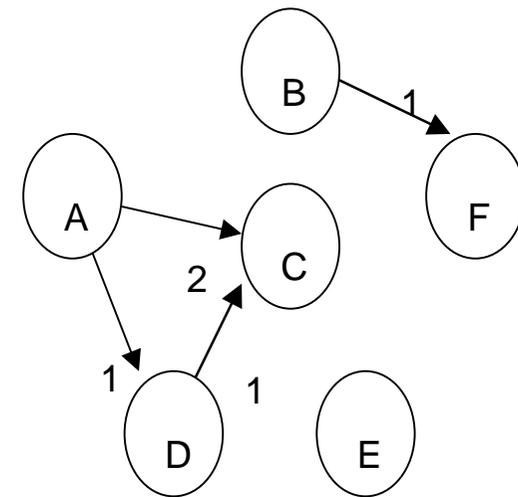
Kanten C E F löschen



Original



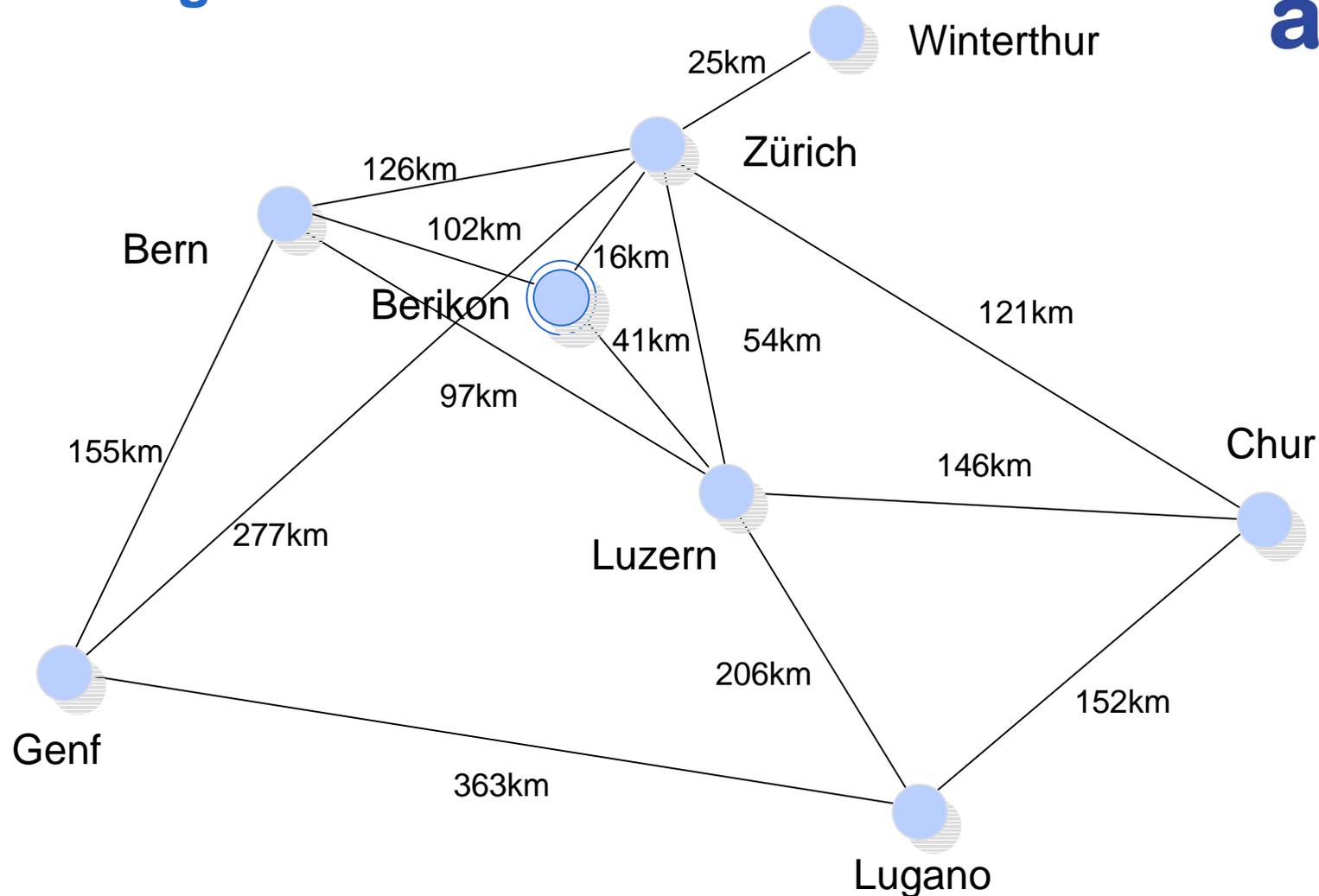
Vorläufiger Fluss



Rest Fluss

# Traveling Salesman

# Traveling Salesman Problem: TSP



■ Finden Sie die kürzeste **Reiseroute** durch die Schweiz, in der jede Stadt genau einmal besucht wird. Option: am Schluss wieder am Ursprungsort

# Eigenschaften des TSP

- Es ist relativ einfach eine Lösung im Beispiel zu finden:
- Aber: manchmal gibt es **überhaupt keine** Lösung. Beispiel: Wenn mehr als eine Stadt nur über einen Weg erreichbar ist.
- Ob es der kürzeste Weg ist, lässt sich nur durch Bestimmen sämtlicher möglicher Wege zeigen ->  $O(N!)$

- Bis heute keine effiziente Lösung des TSP bekannt. Alle bekannten Lösungen sind von der Art :

## Allgemeiner TSP-Algorithmus:

Erzeuge alle möglichen Routen;

Berechne die Kosten (Weglänge) für jede Route;

Wähle die kostengünstigste Route aus.

- Die Komplexität aller bekannten TSP-Algorithmen ist  $O(a^N)$ , wobei N die Anzahl der Kanten ist. -> Exponentielle Aufwand
- Das heisst: Wenn wir eine einzige Kante hinzunehmen, verdoppelt sich der Aufwand zur Lösung des TSP!
- Oft begnügt man sich mit einer "guten" Lösung, anstelle des Optimums

## ■ Graphen

- gerichtete, ungerichtete
- zyklische, azyklische
- gewichtete, ungewichtete

## ■ Implementationen von Graphen

- Adjazenz-Liste, Adjazenz-Matrix

## ■ Algorithmen

- Grundformen: Tiefensuche/ Breitensuche
- kürzester Pfad (ungewichtet/gewichtet)
- Topologisches Sortieren
- Maximaler Fluss
- Traveling Salesman

