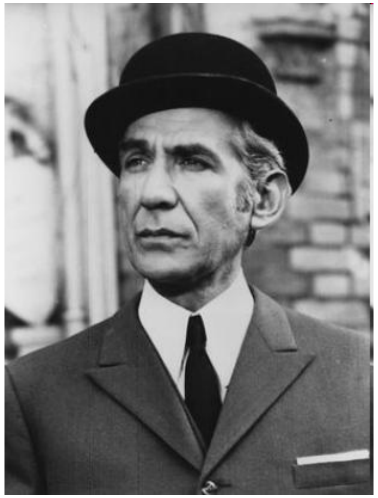
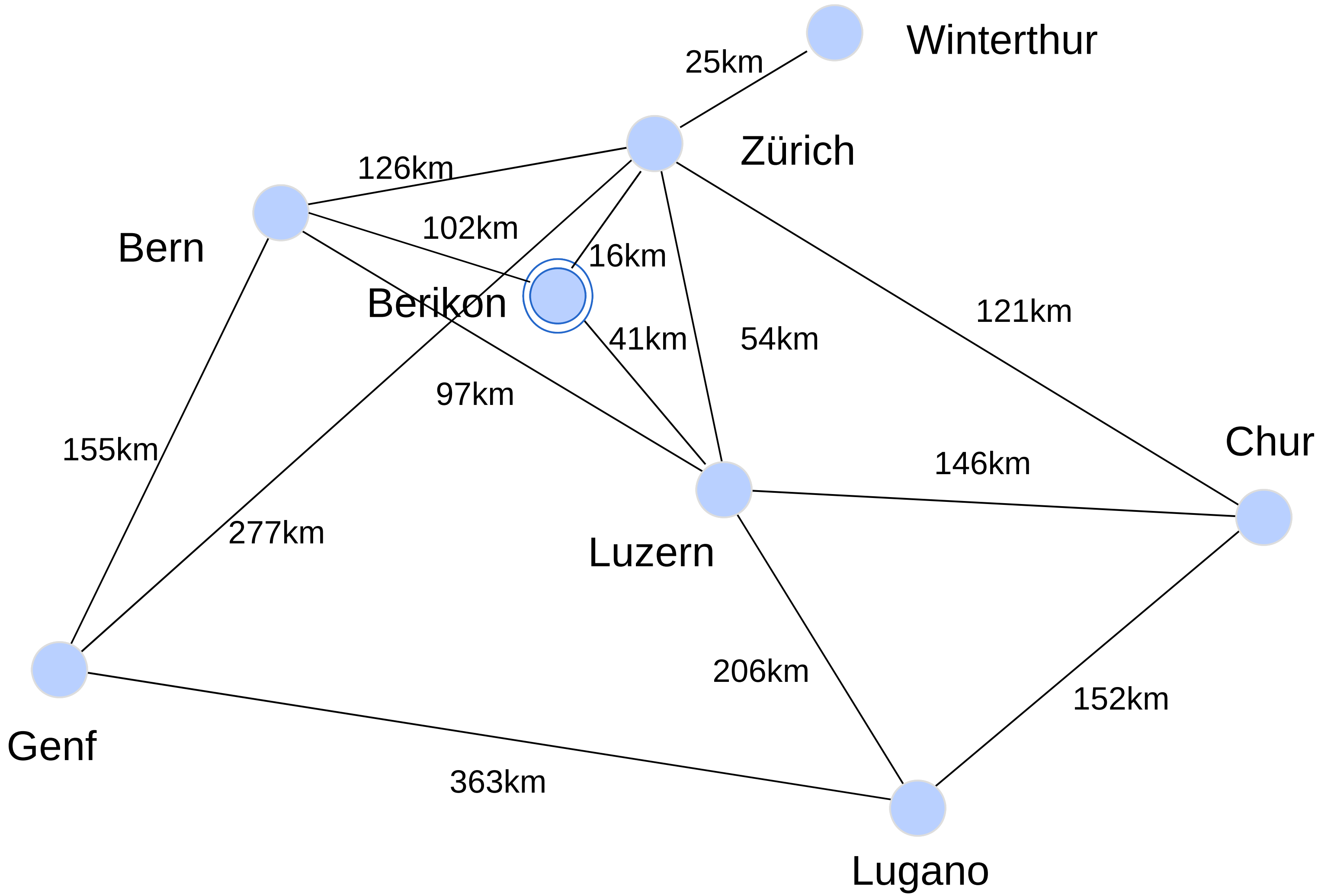
Graphen



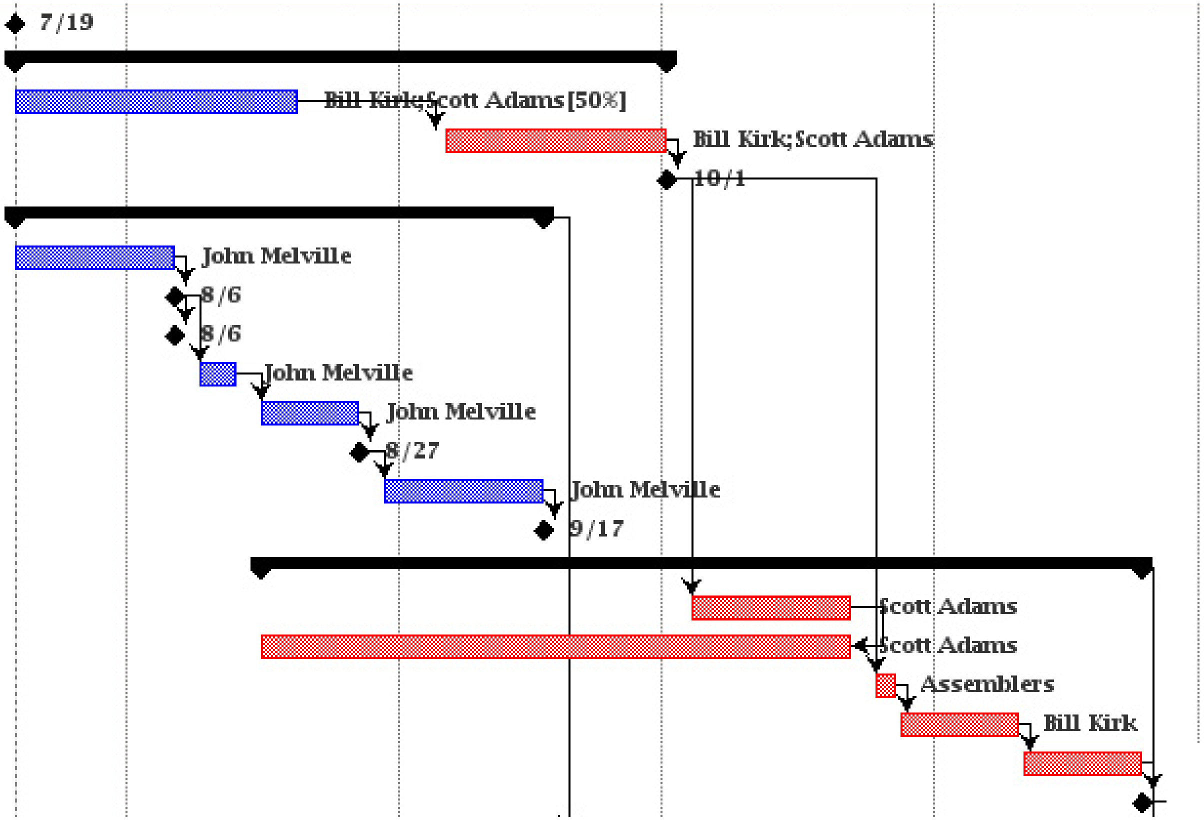
- Sie wissen wie Graphen definiert sind und kennen deren Varianten
- Sie wissen wofür man sie verwendet
- Sie können Graphen in Java implementieren
- Sie kennen die Algorithmen und können sie auch implementieren: Tiefensuche, Breitensuche, kürzester Pfad, topologisches Sortieren

Beispiel Strassennetz



- Typische Aufgabe: finde kürzeste Verbindung zwischen zwei Orten

Netzplan, Aufgabenliste



- Typische Aufgabe: finde mögliche Reihenfolge der Tätigkeiten,

Fragestellungen

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

Graph

- Definition

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

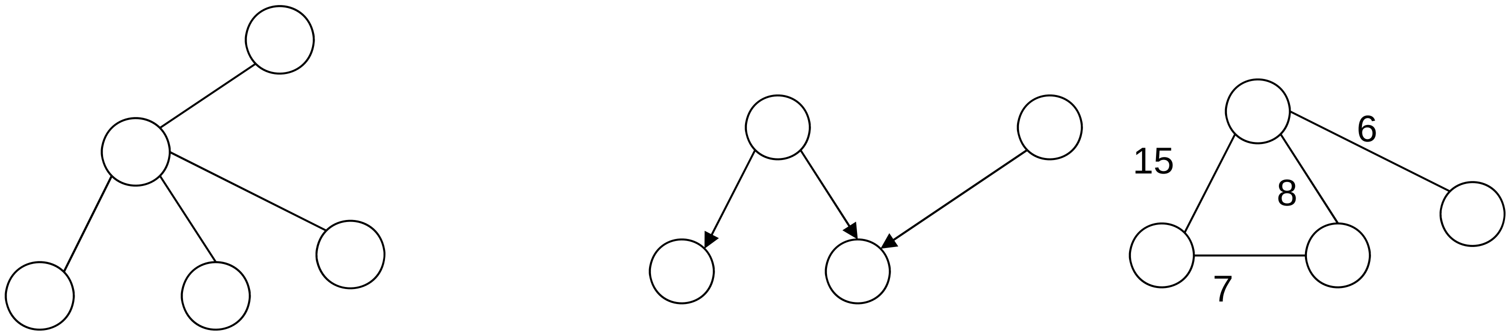
- Implementation
  - Knoten: Objekte mit Namen und anderen Attributen
  - Kanten: Verbindungen zwischen zwei Knoten u.U. mit Attributen

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

Grapheigenschaften

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

Gewichtete Graphen

Ein Graph $G = \langle V, E \rangle$ kann zu einem **gewichteten Graphen** $G = \langle V, E, g_w(E) \rangle$ erweitert werden, wenn man eine Gewichtsfunktion $g_w: E \rightarrow int$ (oder $g_w: E \rightarrow double$) 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 = (v1,v2) gilt: e' = (v2,v1) $\in$ E, so heisst G ungerichteter Graph, ansonsten gerichteter Graph.
- Die Relation E ist in diesem Fall 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.

Grapheigenschaften 2

Anzahl Kanten

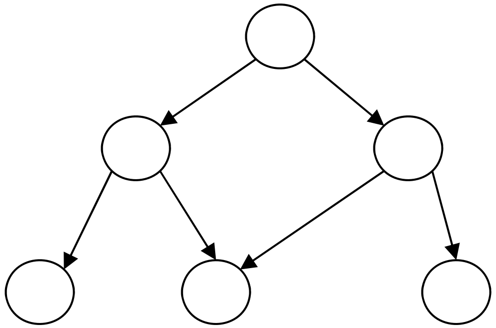
- **Kompletter Graph**: Jeder Knoten ist mit jedem anderen Knoten verbunden.

- **Dichter Graph (dense):** Nur wenige Kanten im Graph (bezogen auf den kompletten Graphen) fehlen,

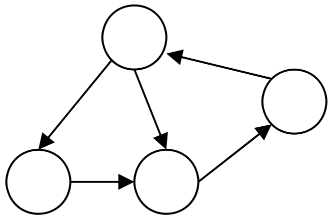
- **Dünner Graph (sparse)** Nur wenige Kanten im Graph (bezogen auf den kompletten Graphen) sind vorhanden

Pfade, Zyklen

- Eine Sequenz von benachbarten Knoten ist ein **einfacher Pfad**, falls kein Knoten zweimal vorkommt z.B. p = {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.

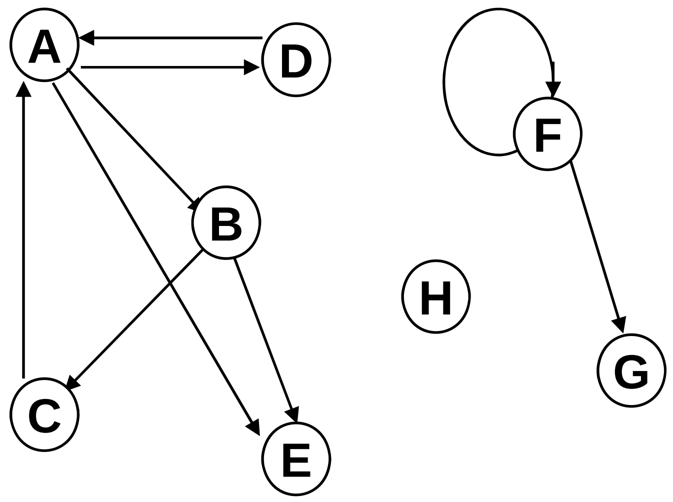
Er enthält einen _____: (A, D, A).

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

(F, F, F, G) ist ein _____.

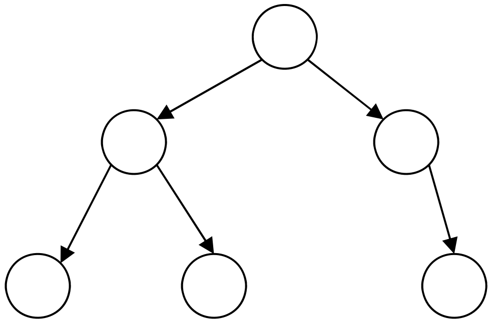
(A, B, C, A) und (A, D, A) und (F, F) sind die einzigen _____.

(A, B, E, A) ist kein ____ und kein _____.

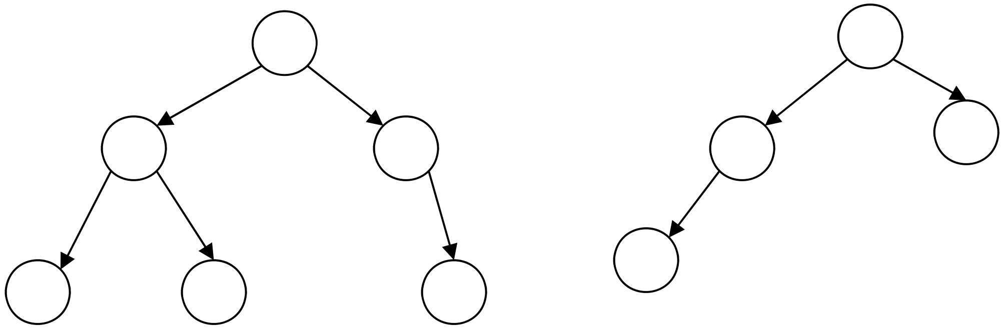


Baum wird zum Spezialfall

- Ein zusammenhängender, gerichteter, zyklenfreier Graph mit genau einer Wurzel ist ein **Baum**

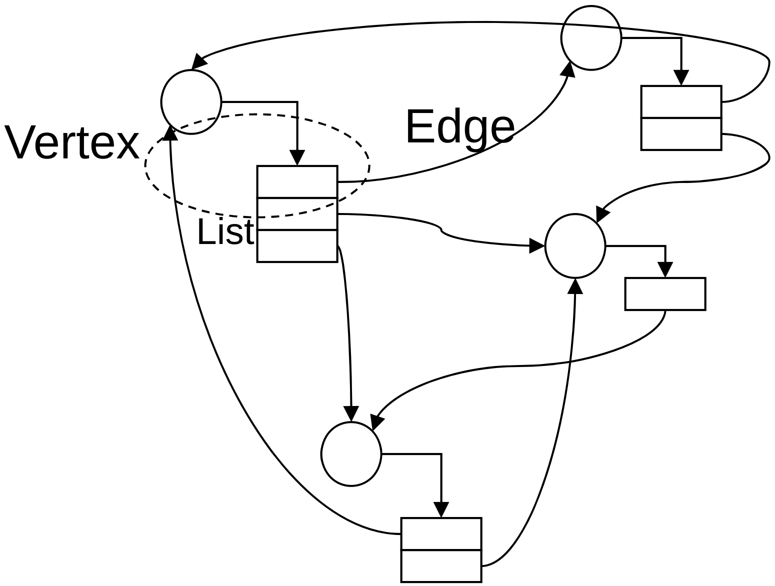


- Eine Gruppe paarweise nicht zusammenhängender Bäume heisst **Wald**.



Implementation 1 : Adjazenz-Liste

- jeder Knoten führt (Adjazenz-) Liste, welch 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<V,E> {

  // füge Knoten hinzu, tue nichts, falls Knoten schon existiert
  public void addNode (String name);

  // finde den Knoten anhand seines Namens
  public V findNode(String name);

  // Iterator über alle Knoten des Graphen
  public Iterable<V> getNodes();

  // füge gerichtete und gewichtete Kante hinzu
  public void addEdge(String source, String dest, double weight);

}
```

Interface, Klasse Vertex definiert einen Knoten

```
public interface Vertex<E>  {
   public String getName();
   public void setName(String name);
    public Iterable<E> getEdges();
    public void addEdge(E edge);
}
```

```
public class VertexImp<E> implementes Vertex<E>  {
    protected String name;   // Name
    protected List<E> edges;  // Kanten
    public VertexImp(){edges  = new LinkedList<E>( ); }
    public VertexImp(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<V> {
    protected V dest;  // Zielknoten der Kante
    protected double weight;  // Kantengewicht

    public Edge() {}

    public Edge(V dest, double weight) {
        this.dest = dest;
        this.weight = weight;
    }

   public void setDest(V node) {this.dest = node;}

   public V getDest() {return dest;}

   public void setWeight(double w) {this.weight = w;}
   double getWeight() {return weight;}
}
```

Das Decorator Pattern

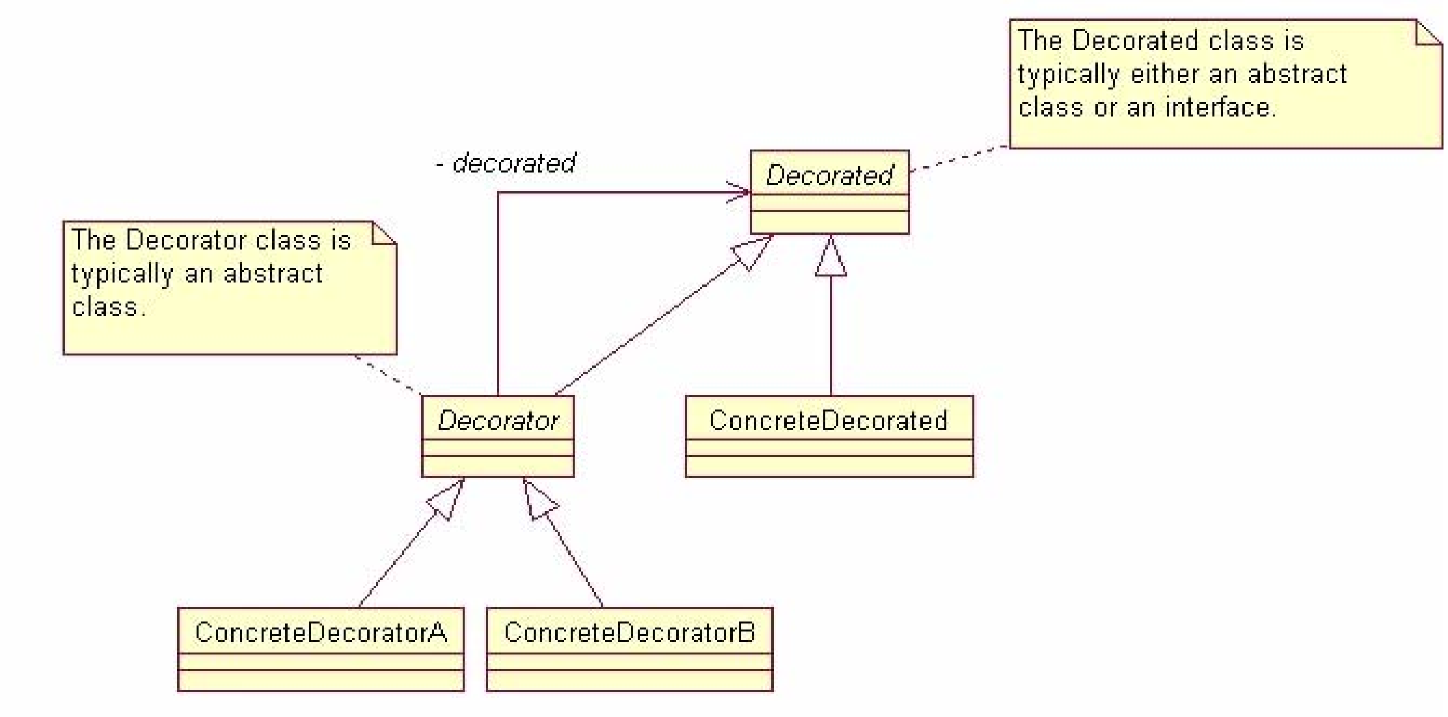
- Für manche Algorithmen auf Graphen genügt die Funktionalität der Klasse *Vertex* nicht.

- Wir möchten, z.B.:
  - Einen Knoten (Vertex) markieren können. Methoden *setMark(), getMark().*
  - Einen Knoten mit einem Wert versehen können. Methoden *setValue(), getValue().*
  - in einem Graphen einen Pfad beschreiben können. Methoden *setPrev(), getPrev().*

- Wenn wir die Klasse Vertex nicht neu schreiben wollen, bietet das *Decorator*-Pattern eine Lösung.

Das Decorator Pattern

- Das Decorator Pattern erlaubt es, eine bestehende Klasse/Interface nachträglich zur Laufzeit mit zusätzlicher Funktionalität zu versehen.
- Der Decorator hat dabei eine Referenz auf das zu dekorierende Objekt.



Das Decorator Pattern

```
// Decorated
interface Vertex<E> {
  public String getName();
  public Iterator getEdges();
  public void addEdge(Edge edge);
}
```

Dekorator

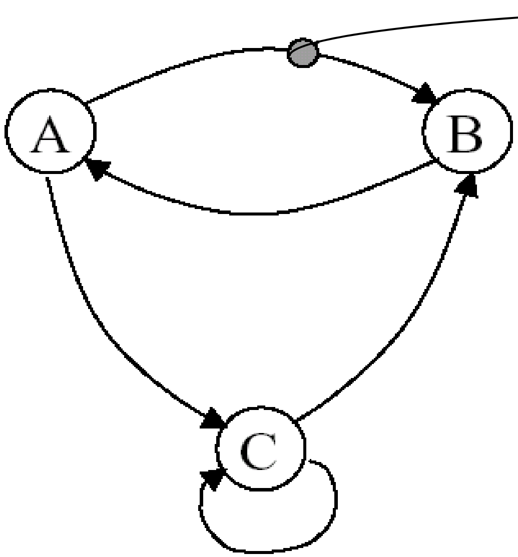
delegate

```
// Decorator
class DijkstraVertex<E> implements Vertex<E> {
   protected Vertex decorated;
   public DecoratedVertex (Vertex v) {decorated = v;}
   public String getName() {return decorated.getName();}
  public Iterator getEdges(){return decorated.getEdges();}
   public void addEdge(Edge edge){decorated.addEdge(edge);}
   public Vertex getDecorated() {return decorated;}
  public void setMark();
  ....
}
```

Concrete Dekorated

```
// Decorated
class VertexImpl<E> implements Vertex<E> {
     private String name;
     private List<Edge>   adj;
     public VertexImpl(String name ){
            this.name = name;
            adj  = new LinkedList<Edge>( );
     }
   public String getName() {return name;}
     public Iterator<Edge> getEdges() {return adj.iterator();}
     public void addEdge(Edge edge) {adj.add(edge);}
}
```

Implementation 2 : Adjazenzmatrix



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

- N x 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 falls der Graph dicht

Adjazenzmatrix

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

	BN	F	FD	GI	KS	K	MA	MR	WÜ
BN	0	181	-	-	-	34	224	-	-
F	181	0	104	66	-	-	88	-	136
FD	-	104	0	106	96	-	-	-	93
GI	66	106	0	0	174	-	30	-	-
KS	-	-	96	-	0	-	-	104	-
K	34	-	-	174	-	0	-	-	-
MA	224	88	-	-	-	-	0	-	-
MR	-	-	-	30	104	-	-	0	-
WÜ	-	136	93	-	-	-	-	-	0

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

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"

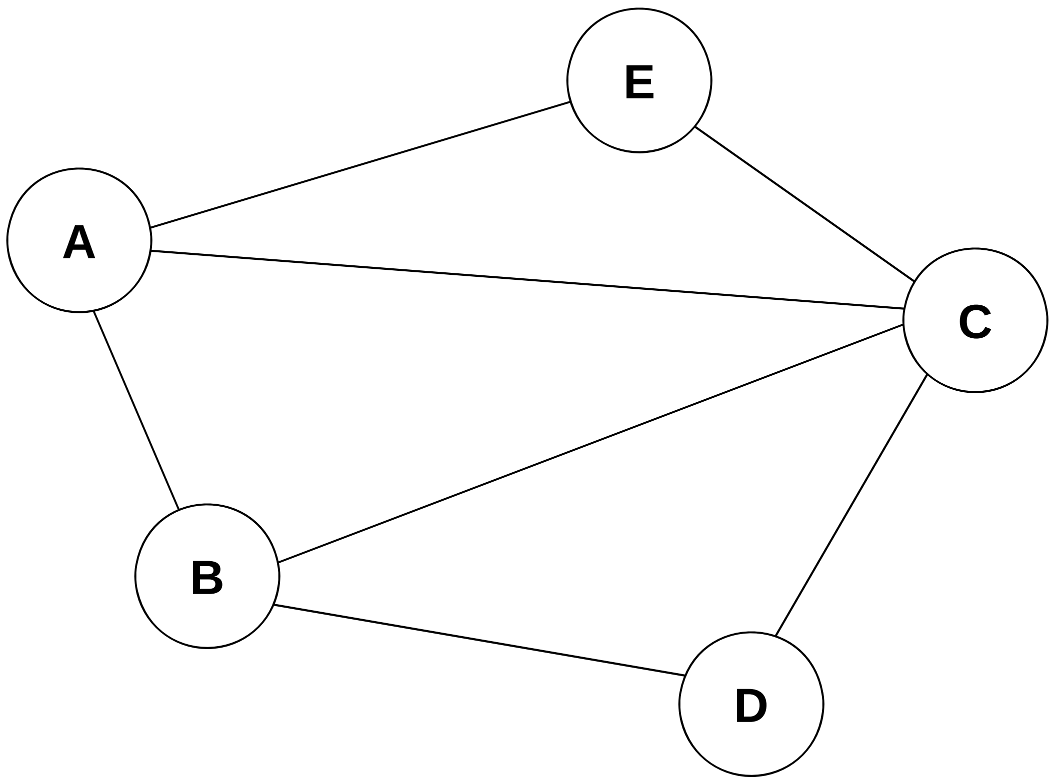
- Genau wie bei den Traversierungen bei Bäumen, sollen bei Graphen die Knoten **systematisch besucht** werden.
- Es werden im Wesentlichen zwei grundlegende "Suchstrategien" unterschieden.

- 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.

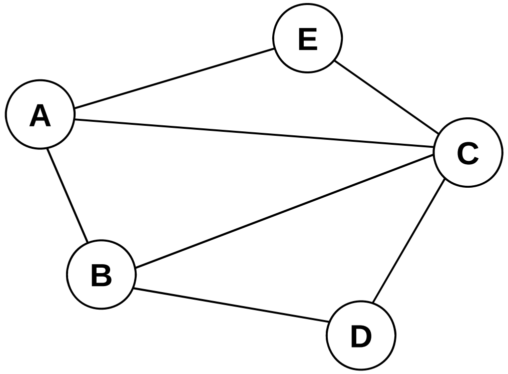
Übung

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



Tiefensuche (Pseudo Code)

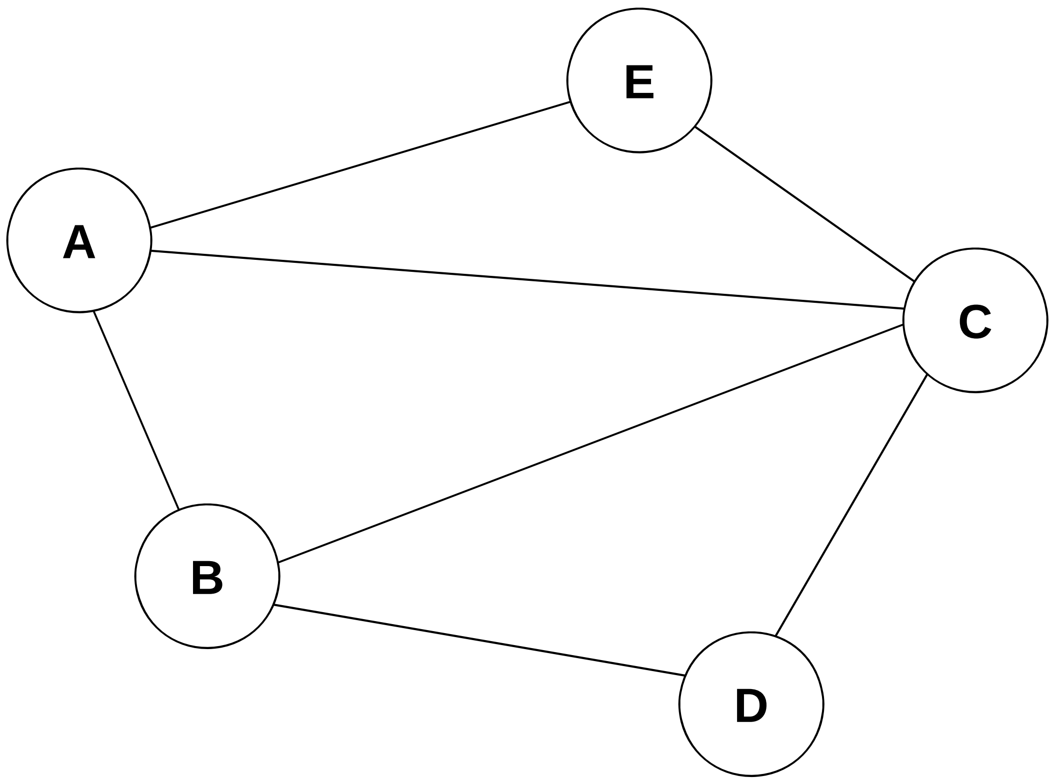
```
void depthFirstSearch()
   s = new Stack();
   mark startNode;
   s.push(startNode)
   while (!s.empty()) {
        currentNode = s.pop()
        print currentNode
        for all nodes n adjacent to currentNode {
           if (!(marked(n)) {
              mark n
              s.push(n)
       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



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

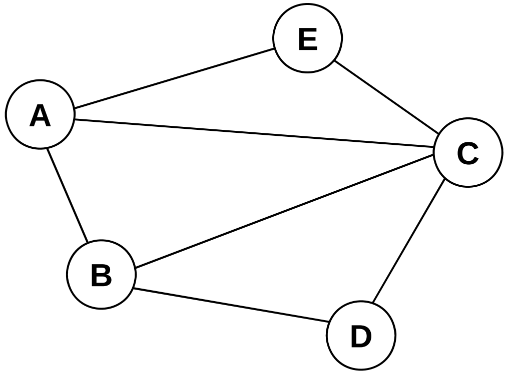
Übung

- Auf welche Arten kann folgender Graph in Breitensuche durchsucht werden



Breitensuche (Pseudo Code)

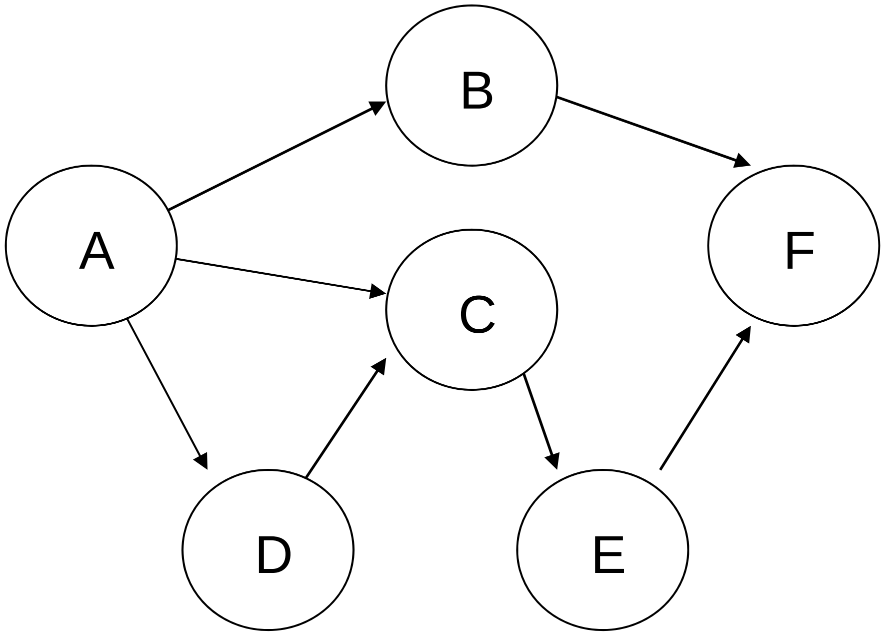
```
void breadthFirstSearch()
   q = new Queue()
   mark startNode
   q.enqueue(startNode)
   while (!q.empty()) {
        currentNode = q.dequeue()
        print currentNode
        for all nodes n adjacent to currentNode {
             if (!(marked(n)) {
                  mark n
                  q.enqueue(n)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



Kürzester Pfad

Kürzester Pfad 1: alle Kanten gleiches Gewicht

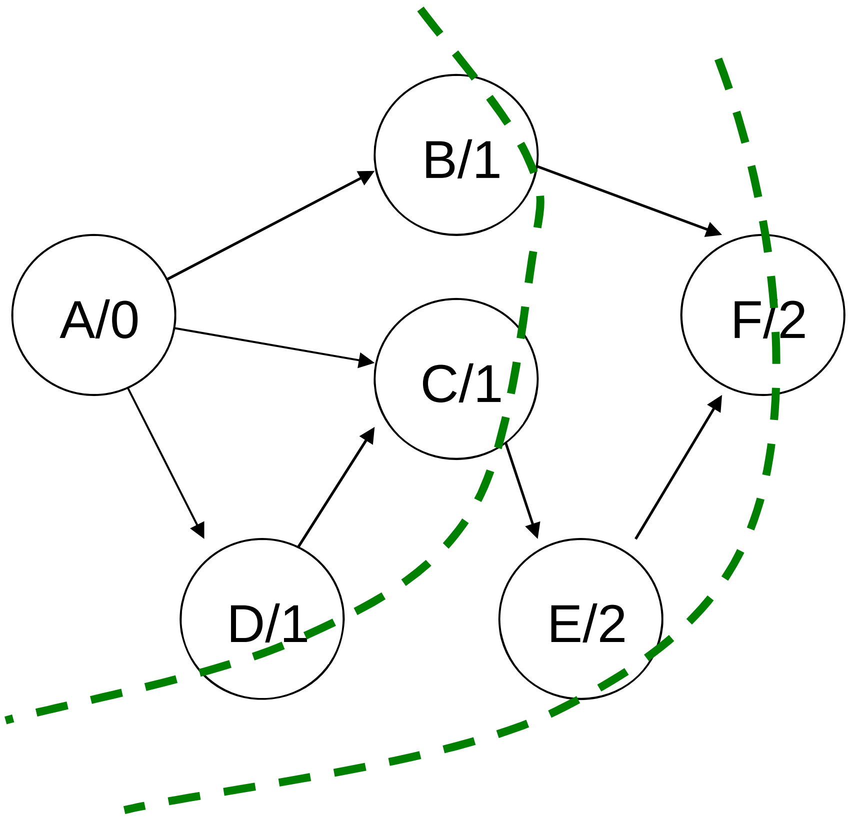
- Gesucht ist der kürzeste Weg von einem bestimmten Knoten aus zu jeweils einem anderen (z.B A nach F)



- Lösung : A nach F über B

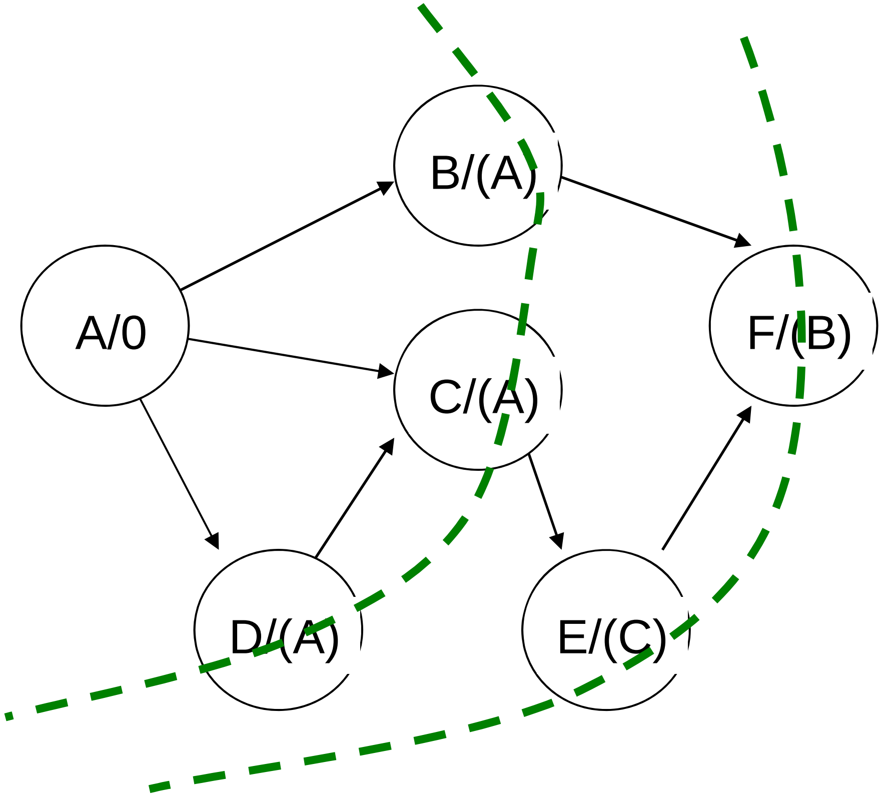
Algorithmus kürzester ungewichteter Pfad

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



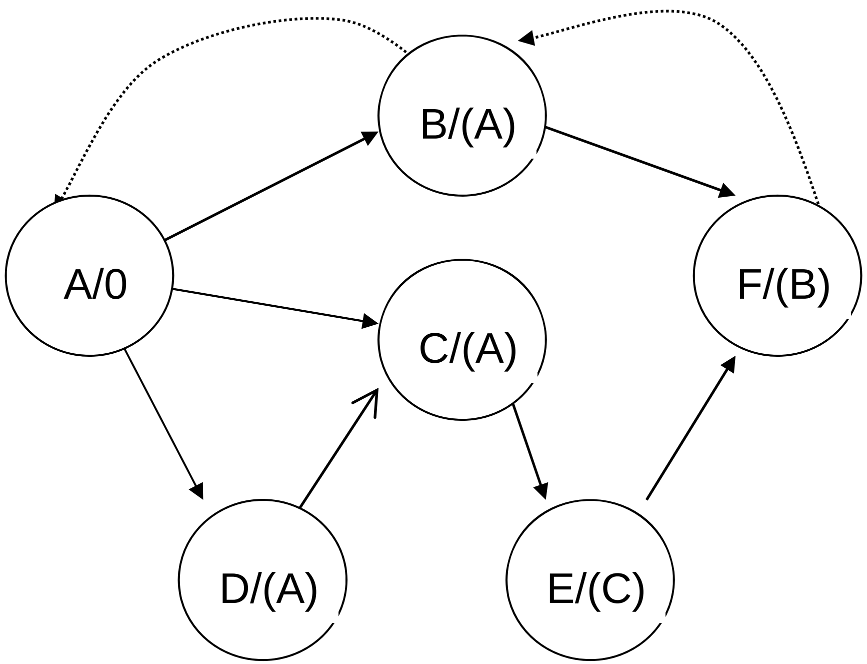
… Algorithmus kürzester ungewichteter Pfad

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



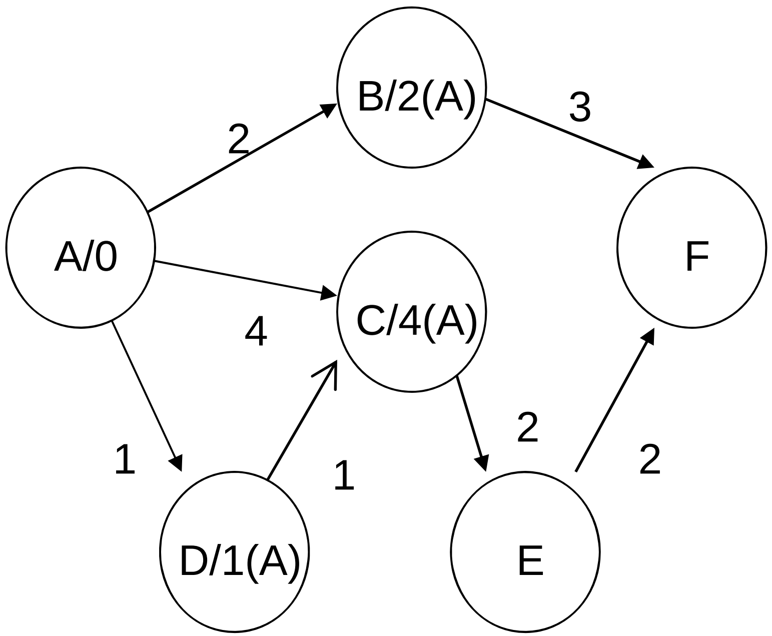
… Algorithmus kürzester ungewichteter Pfad

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



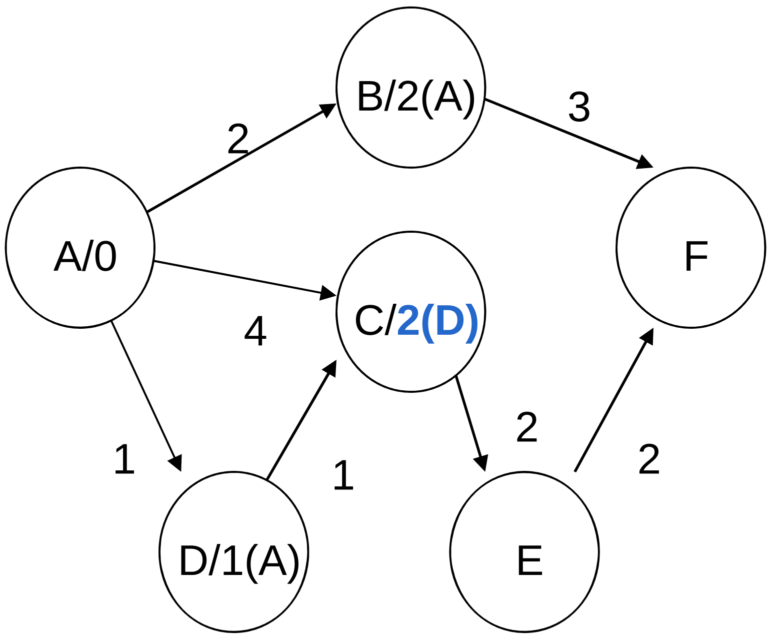
2. Kürzester Pfad bei gewichteten Kanten:

- C ist über D schneller zu erreichen als direkt
  Algorithmus: gleich wie vorher, aber **korrigiere Einträge** für Distanzen



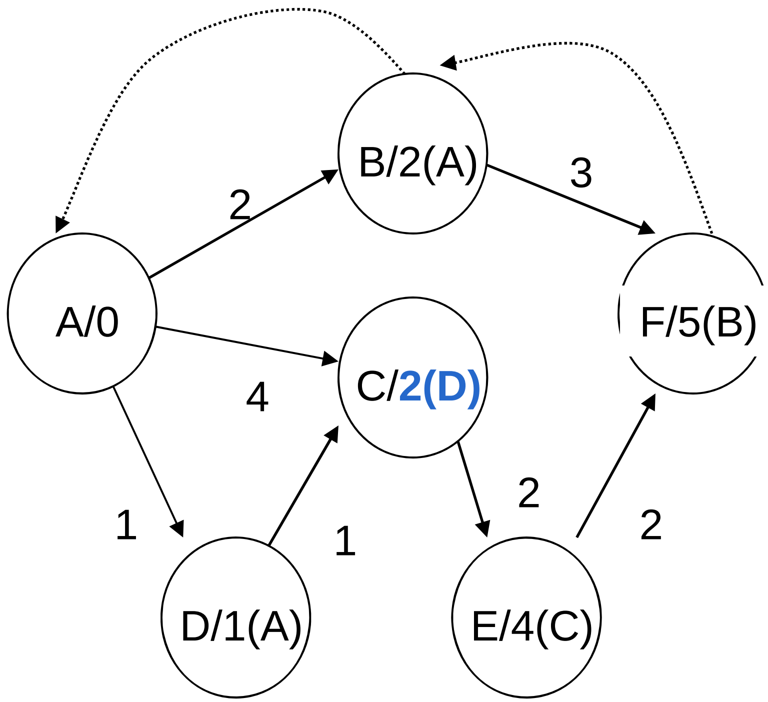
… Kürzester Pfad bei gewichteten Kanten

- 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.



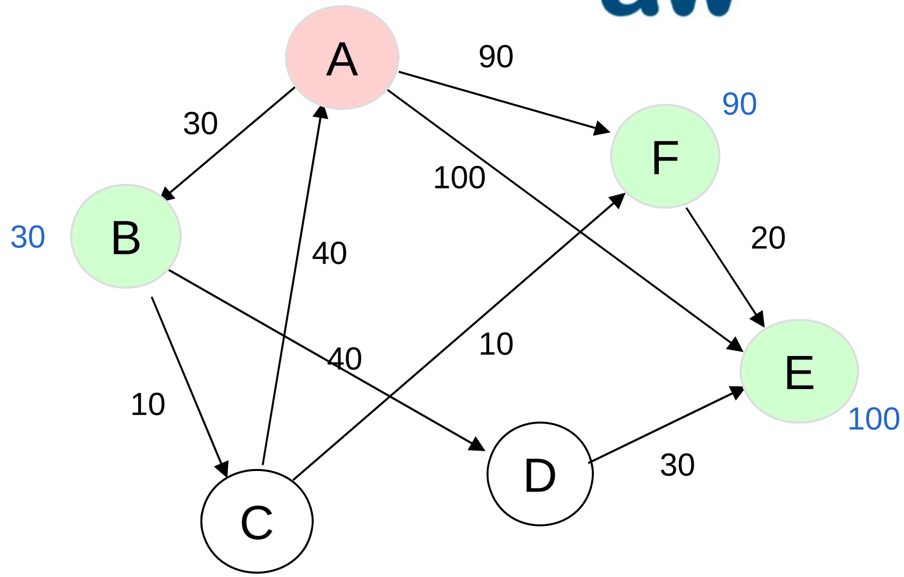
… Kürzester Pfad bei gewichteten Kanten

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

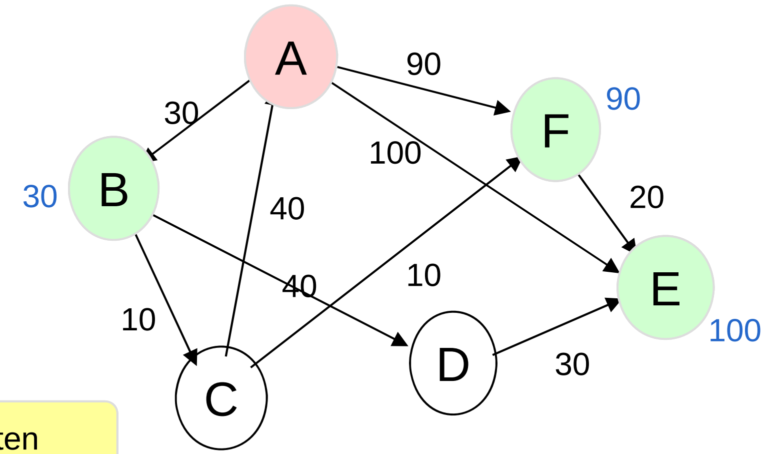
Pseudocode

```
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;
            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**

Implementation mittels PriorityQueue




```
void breadthFirstSearch()
   q = new PriorityQueue()
   startNode.dist = 0;
   q.enqueue(startNode,0)
   while (!q.empty()) {
        current= q.dequeue()
        mark current
        for all edges e of current {
               n = e.node;
               if (!(marked(n)) {
                  dist = e.dist + current.dist
                  if ((n.prev == null) || (dist < n.dist)) {
                         n.dist = dist;
                         n.prev = current
                         q.enqueue(n,-n.dist)
                  }
               }
        }
   }
}
```

alle benachbarten Knoten

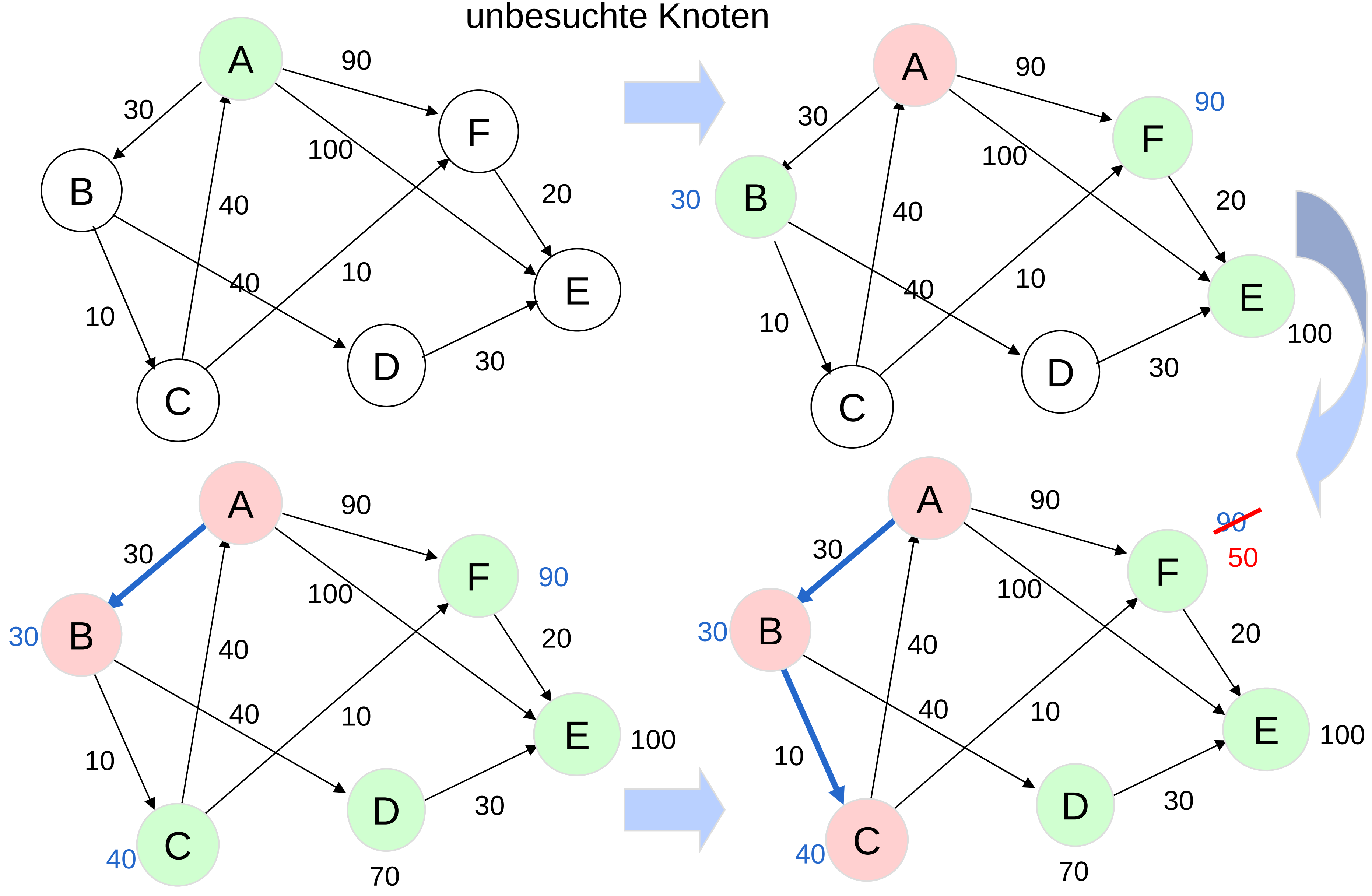
nicht besucht

kürzerer Weg gefunden

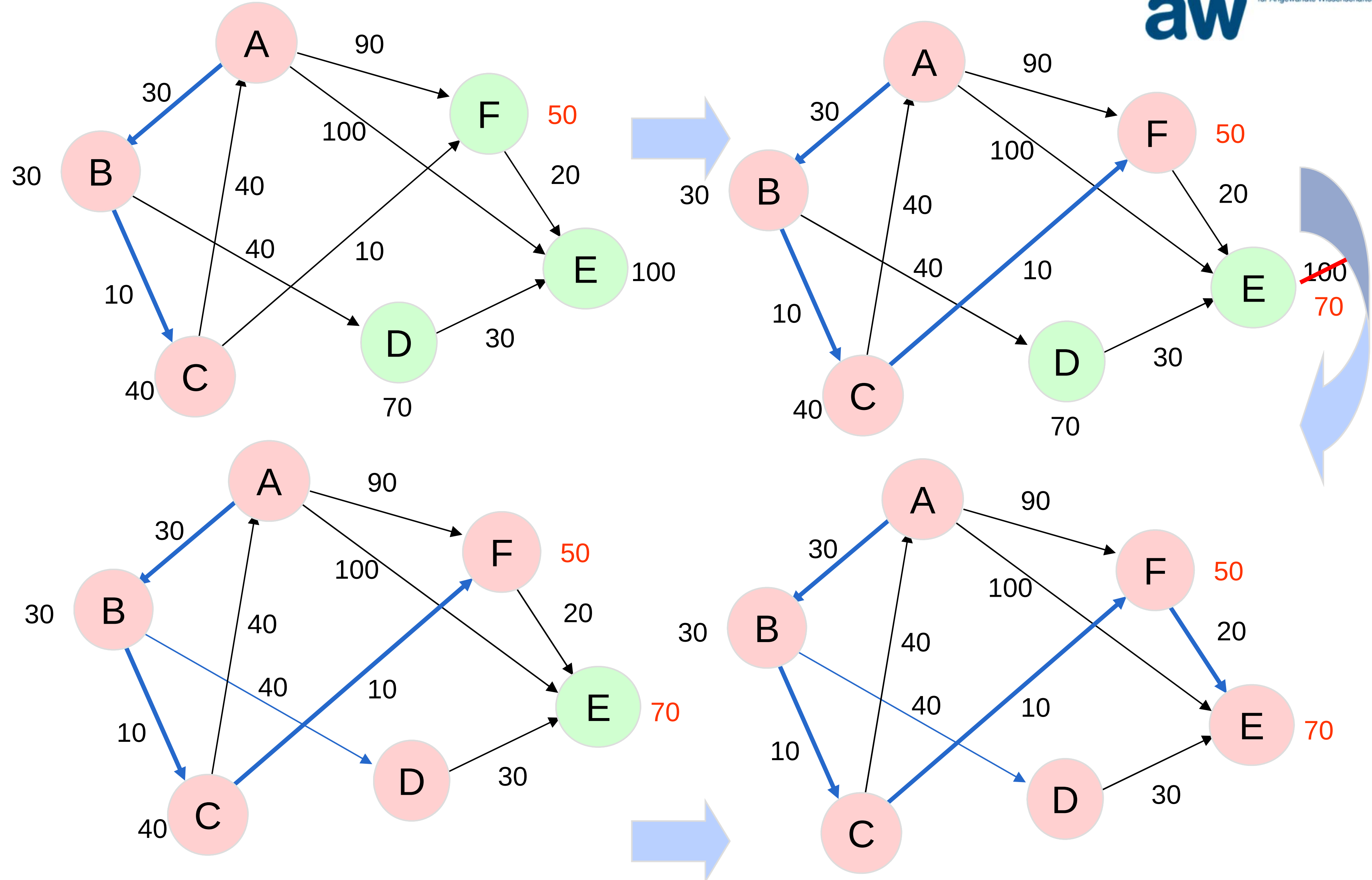
unbesehen

Rückweg

markierte Knoten
Knoten in Queue
unbesuchte Knoten



letzter Stand



Spannbaum

Spannbaum (Spanning Tree)

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.

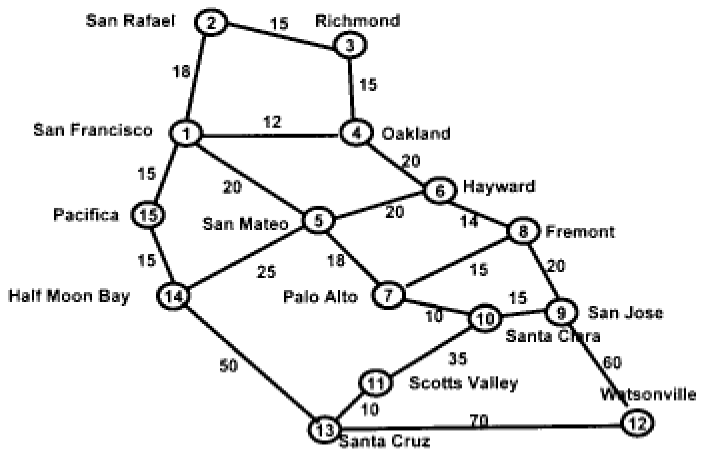
Algorithmus

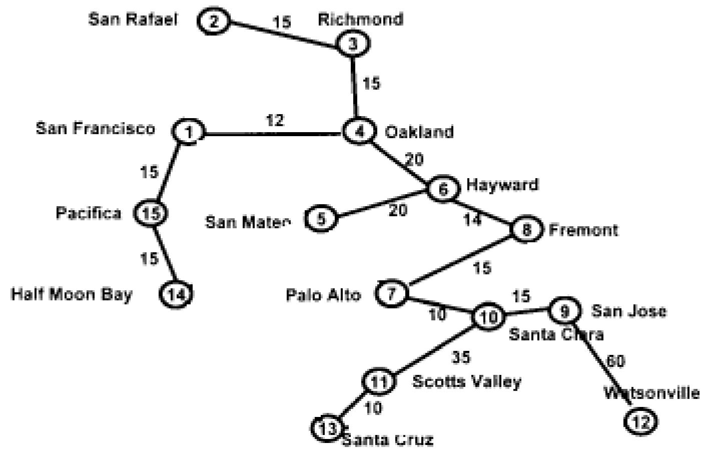
- z.B. Prim-Jarnik

- Prim-Jarnik ist ähnlich wie Dijkstras Algorithmus

Anwendung

- Gesucht Netz mit minimaler Streckenlänge, das alle Städte verbindet



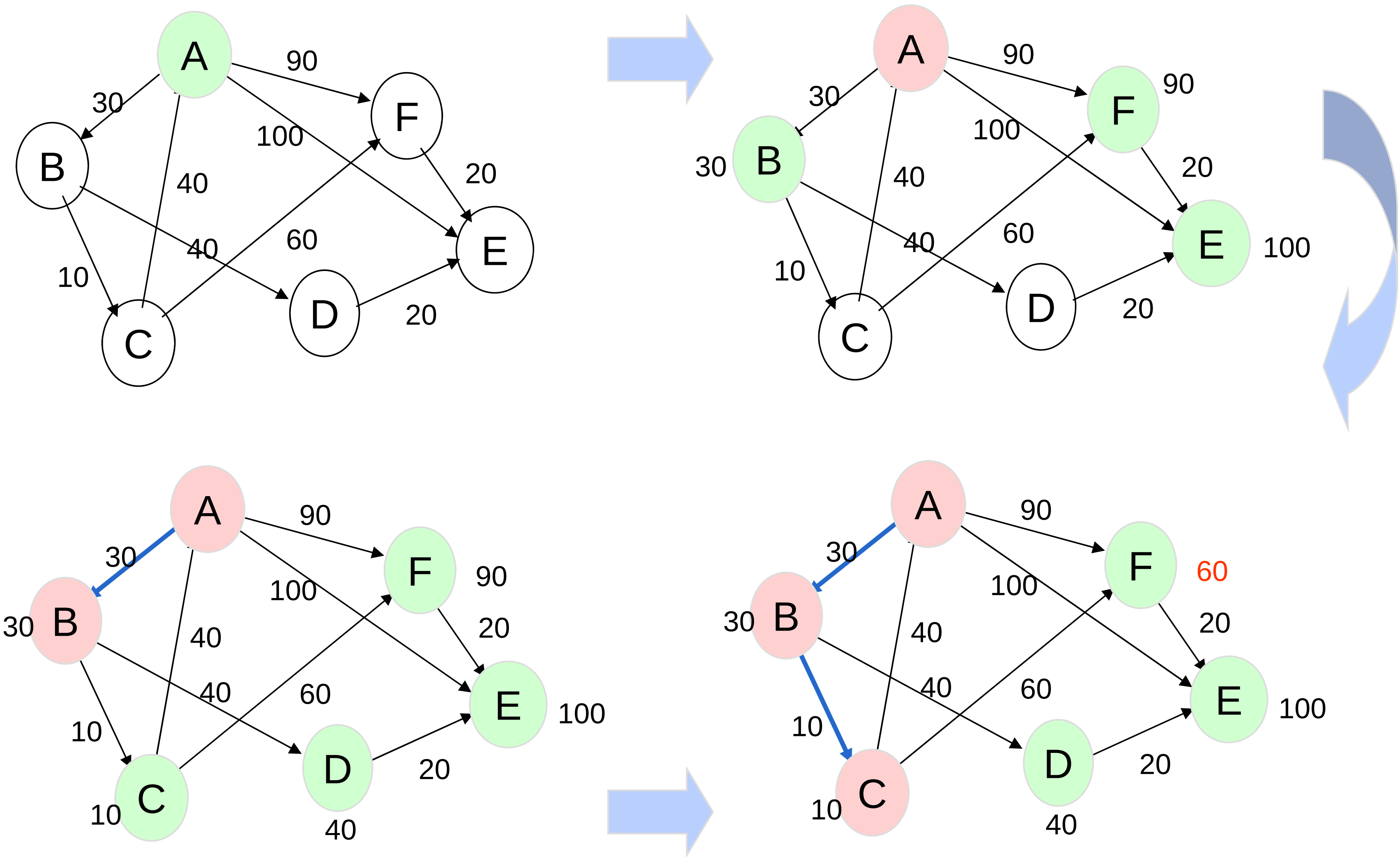


Prim-Jarnik mittels PriorityQueue

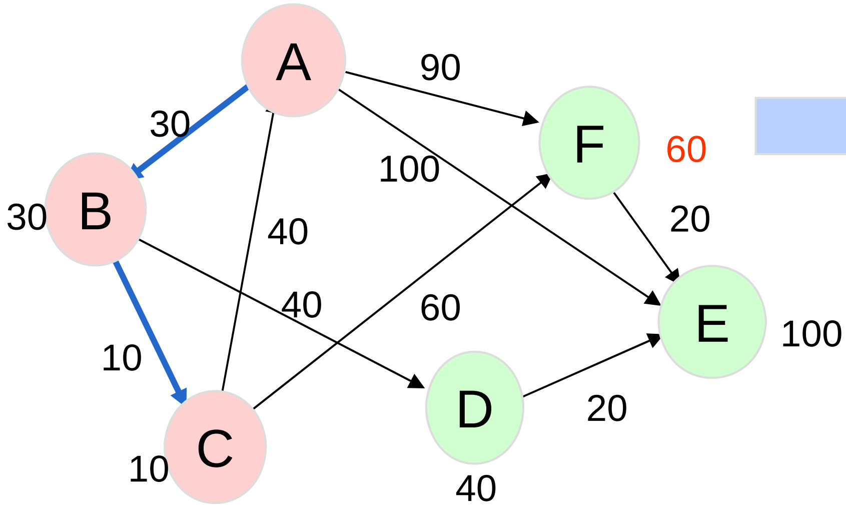
```
void MST() {
   q = new PriorityQueue()
    startNode.dist = 0;
   q.enqueue(startNode,0)
   while (!q.empty()) {
       current = q.dequeue();
       mark current
       for all edges e of current {
             n = e.node;
             if (!(marked(n)) {
               dist = e.dist;
               if ((n.prev == null) || (dist < n.dist)) {
                 n.dist = dist;
                 n.prev = current;
                 q.enqueue(n, -n.dist);
              }
          }
       }
   }
}
```

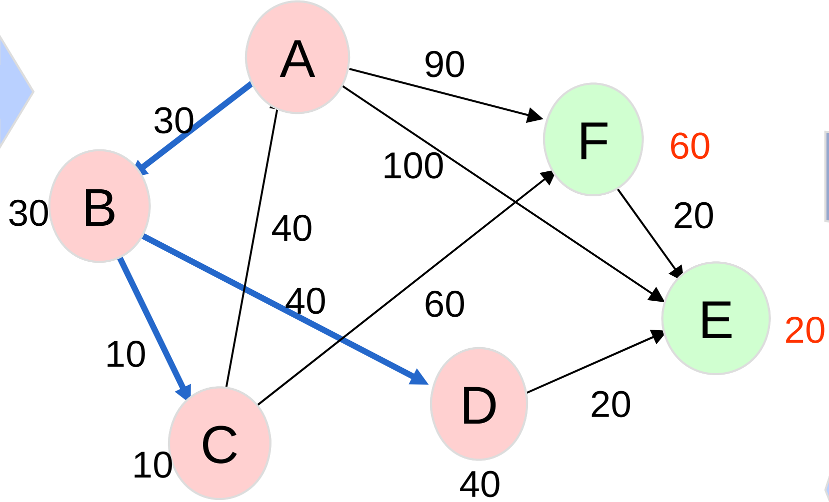
Prinzip: Finde immer einen erreichbaren Knoten, mit der "leichtesten" Kante.

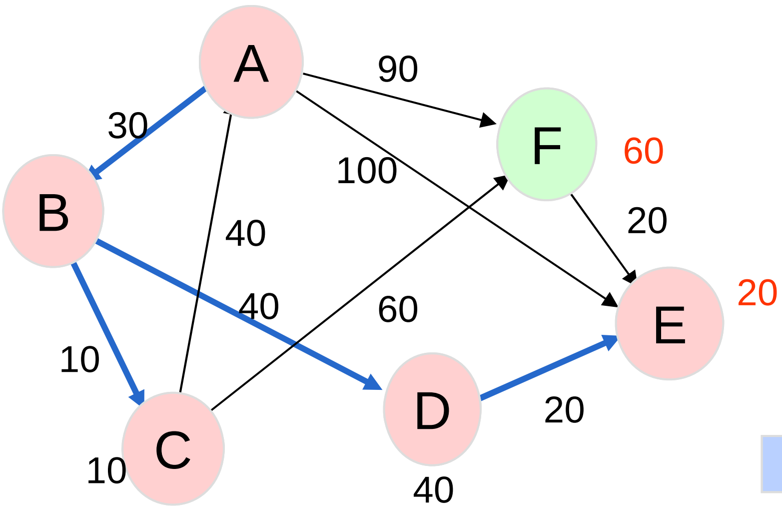
markierte Knoten
Knoten in Queue
unbesuchte Knoten

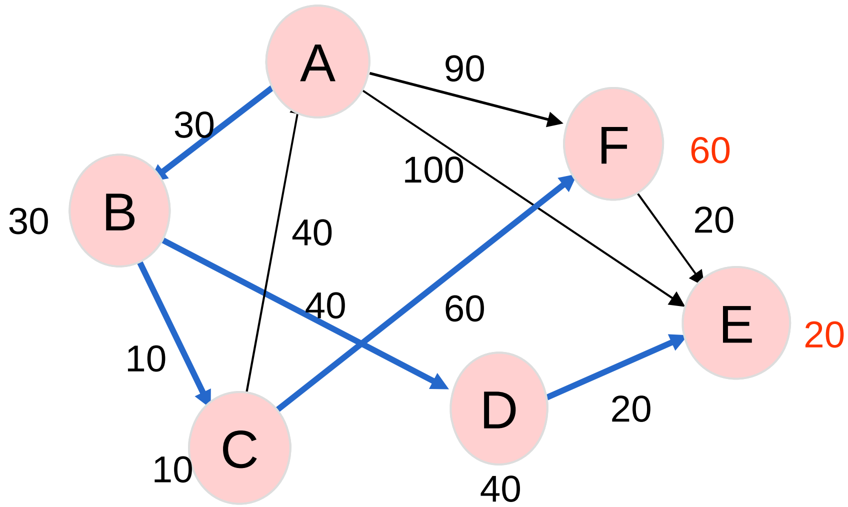


letzter Stand



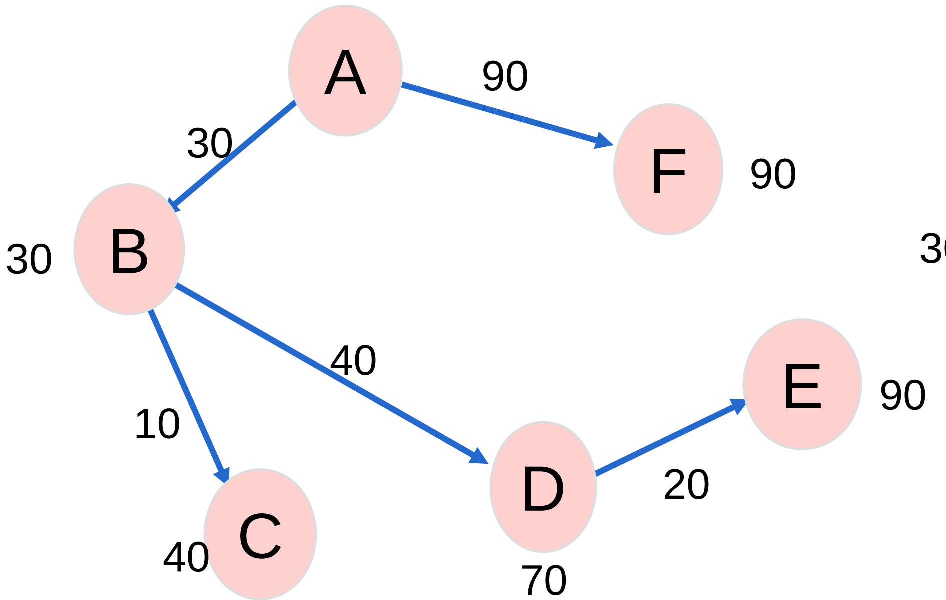






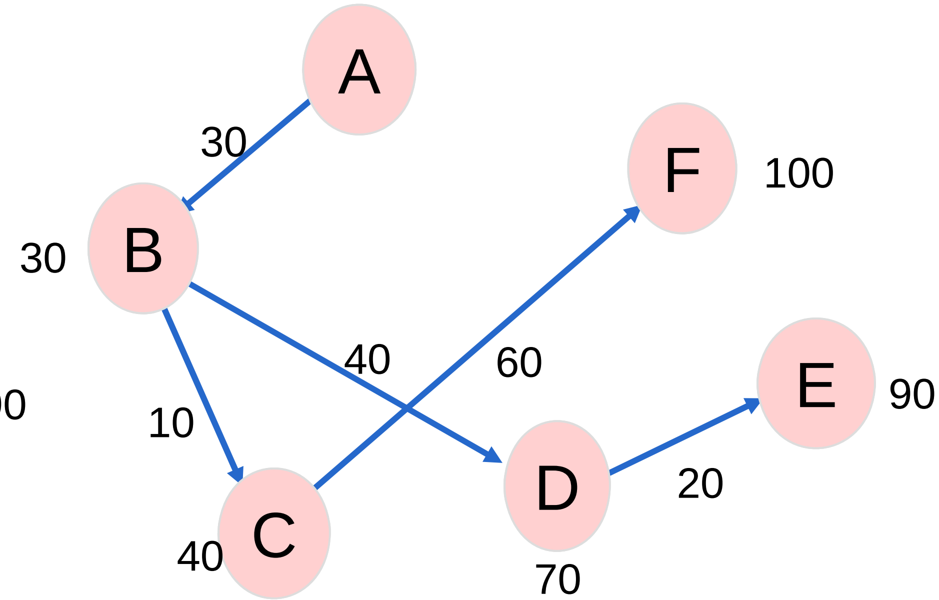
Shortest Path vs. Minimum Spanning Tree

Dijkstra: Shortest path



Total = 190
Path A-F = 90

MST: Minimum Sum

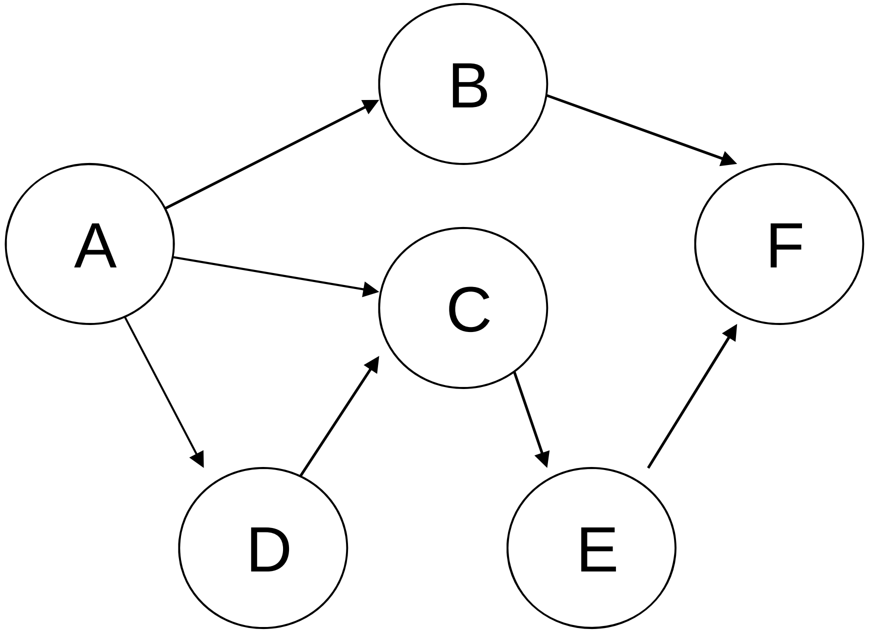


Total = 160
Path A-F = 100

Topologisches Sortieren

Sortierung eines gerichteten Graphen: topologisches Sortieren

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



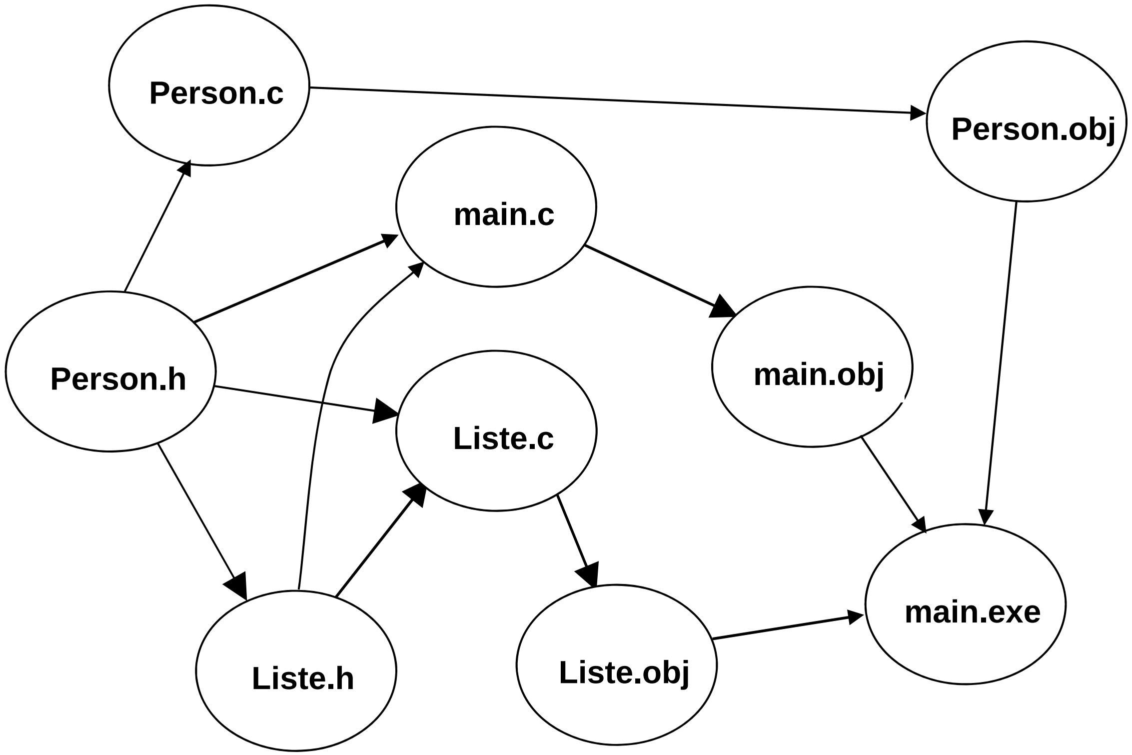
Anwendung:
Die Kanten geben Abhängigkeiten zwischen Modulen in einem Programm an.

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



Übung

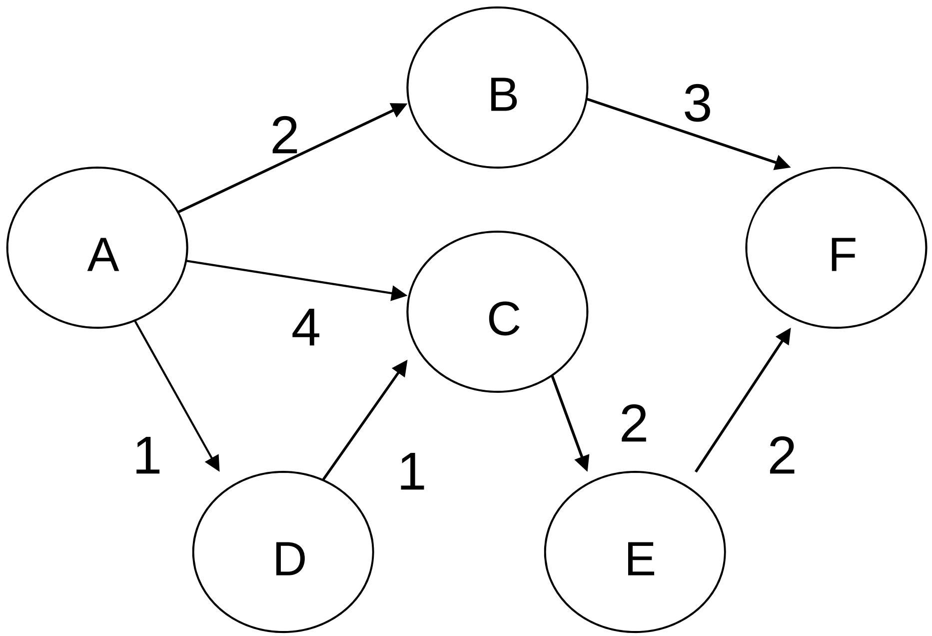
- Beschreiben Sie einen Algorithmus (in Pseudocode), der eine korrekte Auflistung eines azyklischen gerichteten Graphen liefert.

- Hinweis: Zähle die eingehenden Kanten; welche können ausgegeben werden?

Maximaler Fluss

Maximaler Fluss

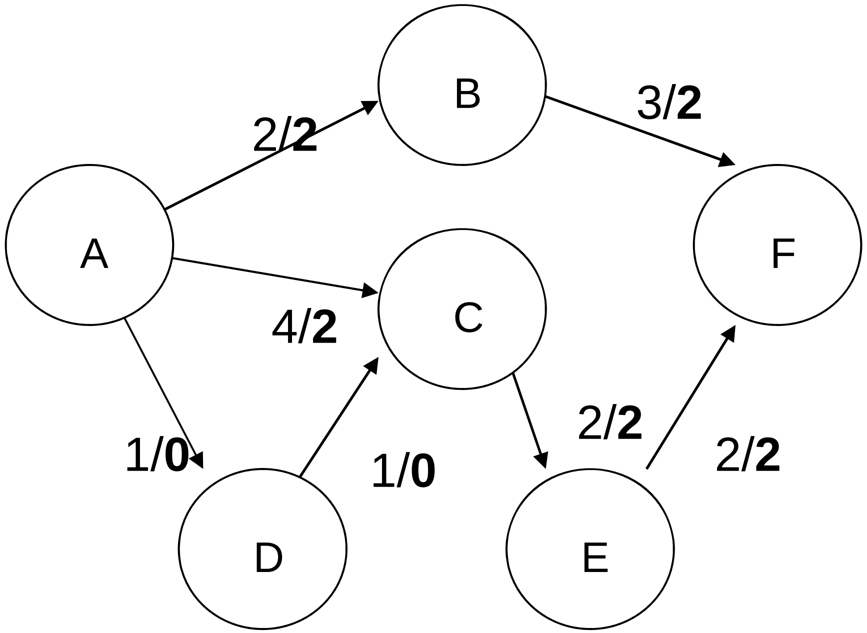
- Die Kanten geben den maximalen Fluss zwischen den Knoten an. Wieviel fliesst von A nach F ?



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

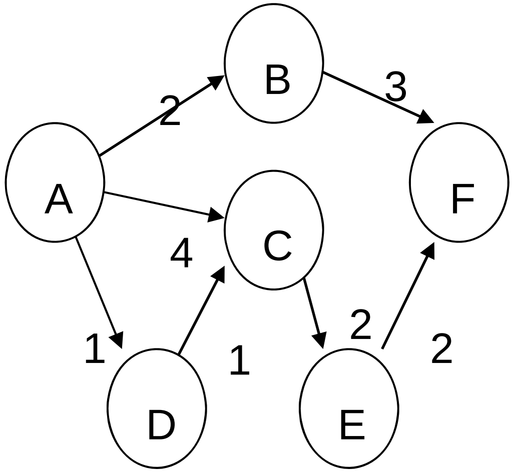
Maximaler Fluss

- Resultat : 4

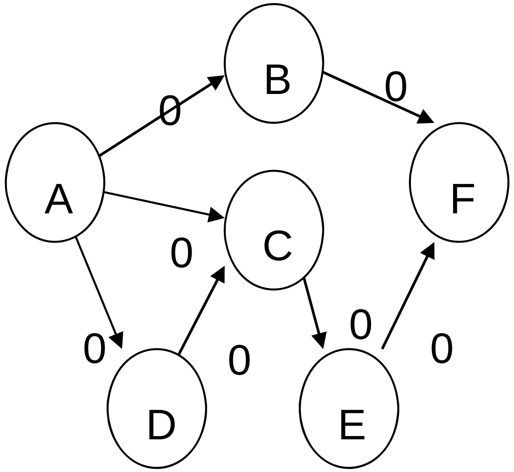


Lösungsidee maximaler Fluss

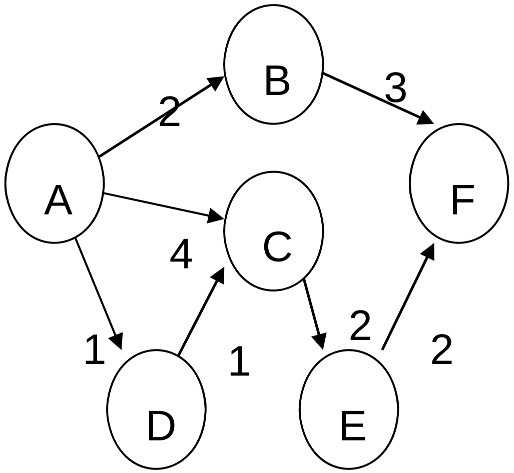
- Noch 2 zusätzliche Versionen des Graphen



Original



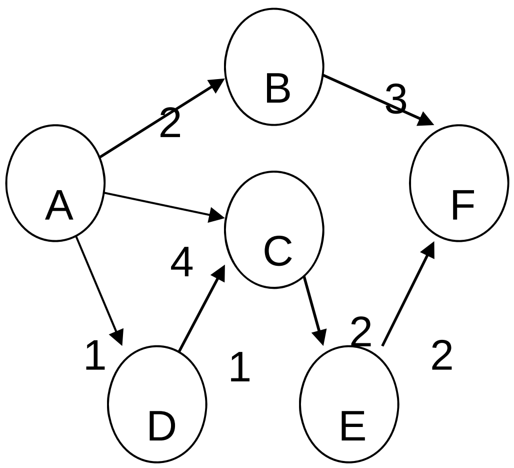
Vorläufiger Fluss



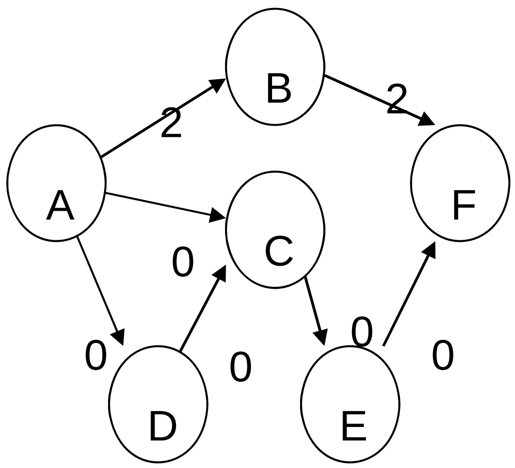
Rest Fluss

Lösungsidee maximaler Fluss

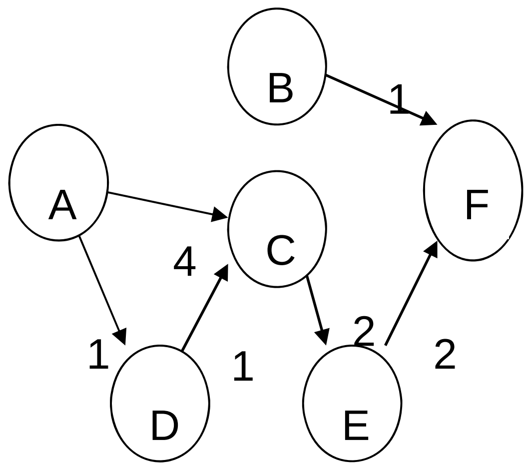
- Pfad A B F einfügen : Fluss 2 Kante A B löschen



Original



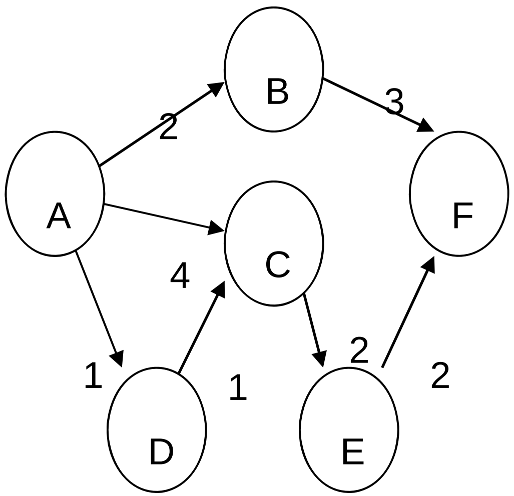
Vorläufiger Fluss



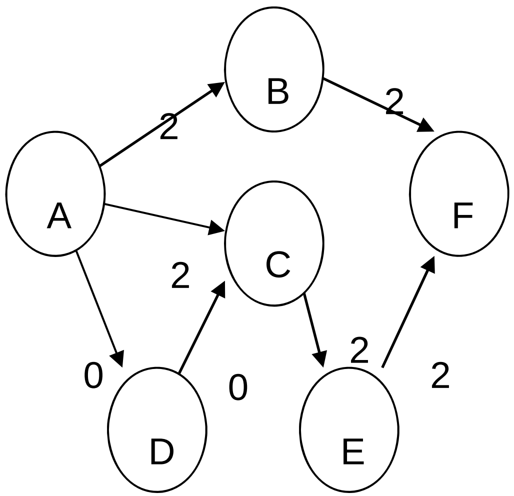
Rest Fluss

Lösungsidee maximaler Fluss

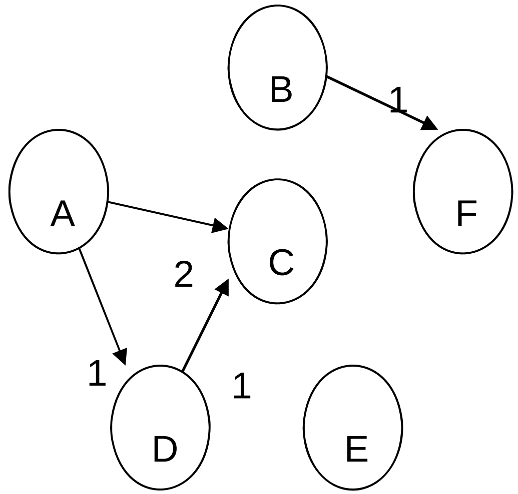
- Pfad A C E F : Fluss 2 Kanten C E F löschen



Original



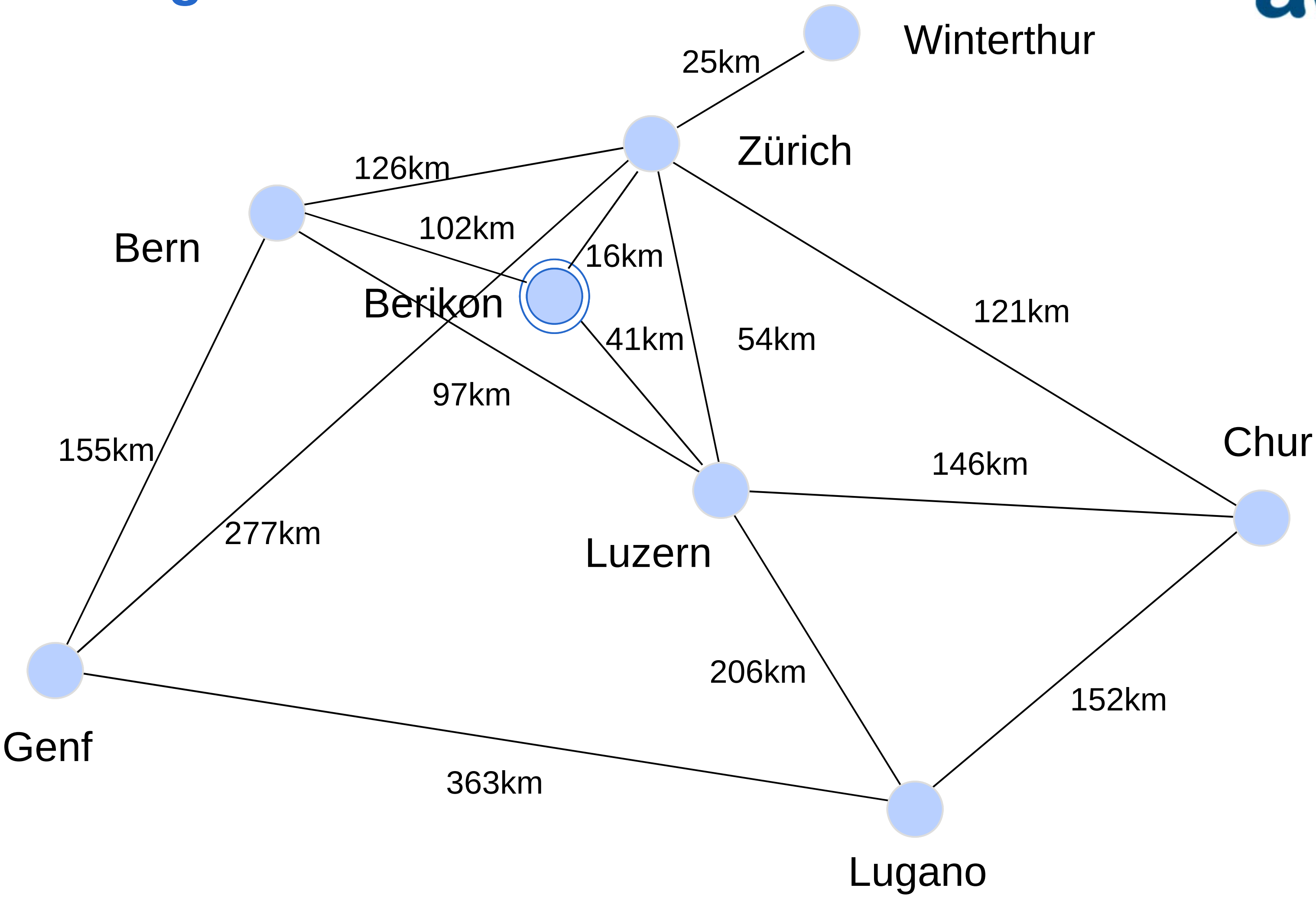
Vorläufiger Fluss



Rest Fluss

Traveling Salesman

Traveling Salesman Problem: TSP



- Finden Sie die kürzeste **geschlossenen Reiseroute** durch die Schweiz, in der jede Stadt genau einmal besucht wird.

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 -> $O(a^N)$

Lösungen des TSP

- 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

Zusammenfassung

- 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