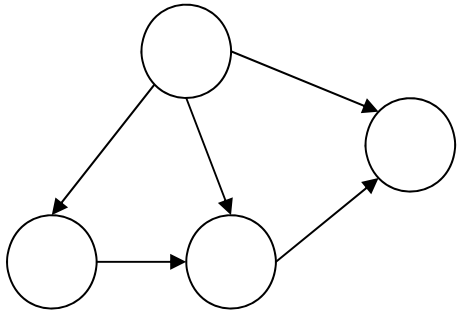
Zurich University of Applied Sciences

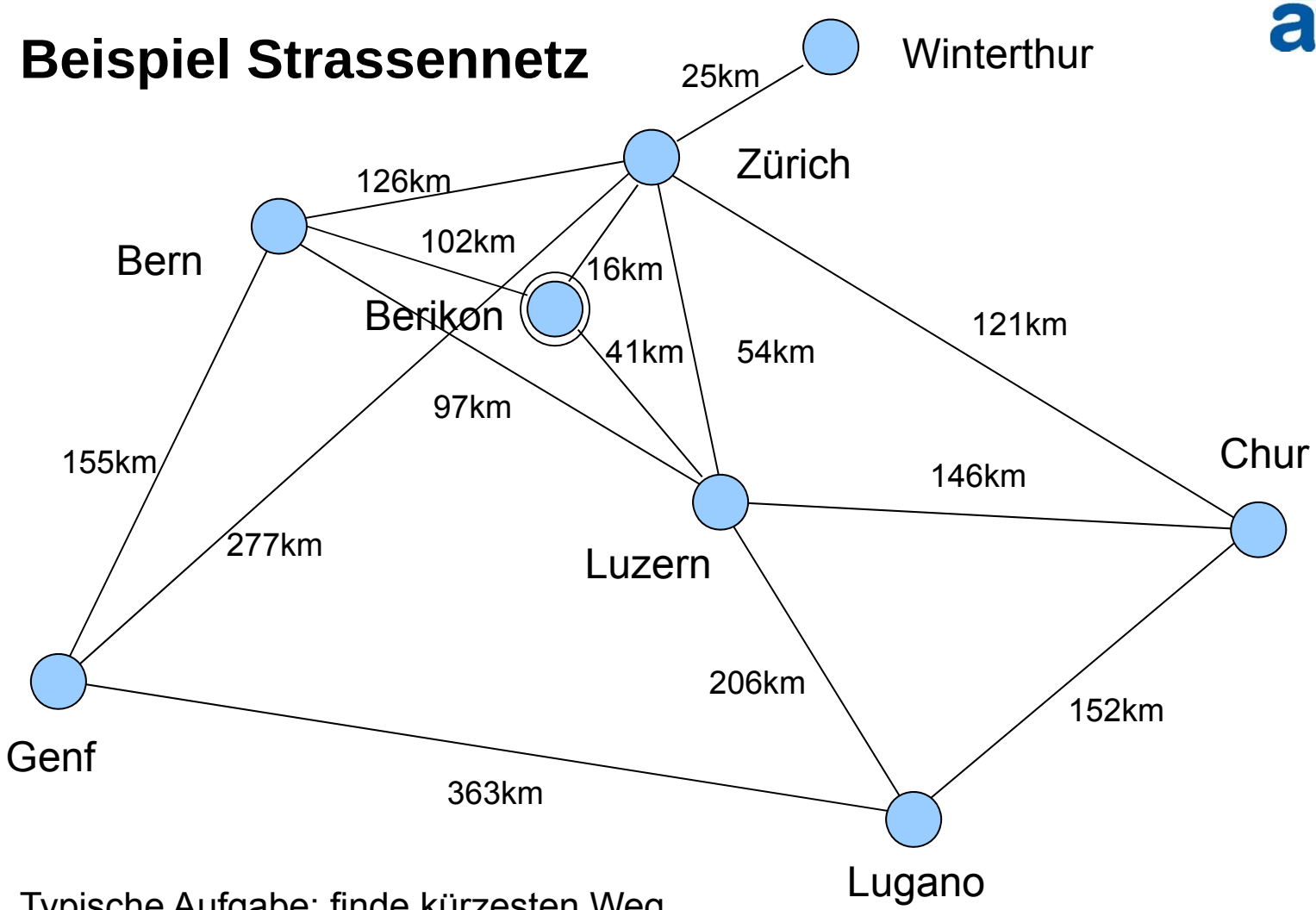
zh aw School of Engineering

Graphen



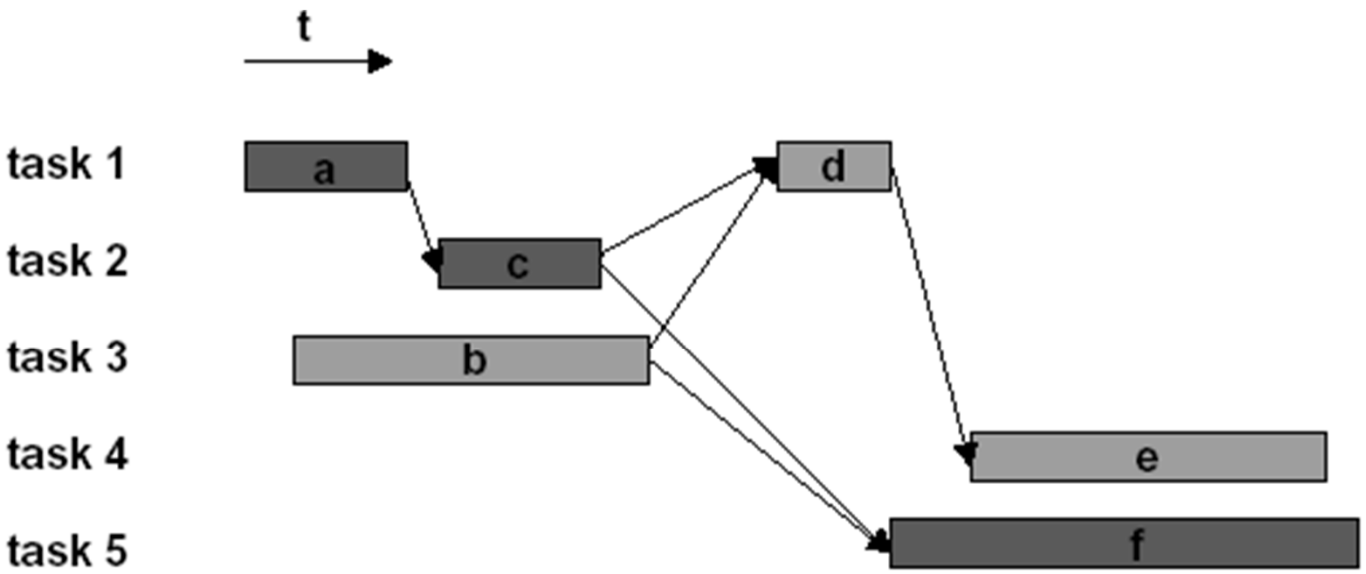
- Sie wissen wie Graphen definiert sind und kennen deren Varianten
- Sie kennen ein paar prototypische Anwendungen von Graphen
- Sie können Graphen in Java implementieren
- Sie kennen die Algorithmen und können sie auch implementieren:
  - Tiefensuche
  - Breitensuche
  - Kürzester Pfad in ungewichteten/gewichteten Graphen
  - Minimum Spanning Tree
  - Topologisches Sortieren
  - Maximaler Fluss

Beispiel Strassennetz



Typische Aufgabe: finde kürzesten Weg

Netzplan



Typische Aufgabe: finde mögliche Reihenfolge der Tätigkeiten,

Student: Wahlfächer

Zurich University of Applied Sciences

zh aw School of Engineering

Fragestellungen

- Kürzeste Verbindung (Auto, Postversand) von A nach B (shortest path)
- Kürzestes Netz um alle Punkte miteinander zu verbinden (minimum spanning tree)
- Kürzester Weg um alle Punkte anzufahren (traveling salesman)
- Maximaler Durchsatz (Auto, Daten) von A nach B (maximal flow)
- Reihenfolge von Tätigkeiten aus einem Netzplan erstellen (topological sort)
- Optimale Reihenfolge mit minimal benötigter Zeit (critical path)

Graph

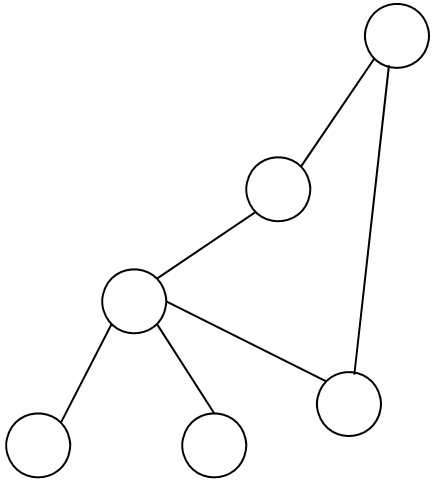
- Definition

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

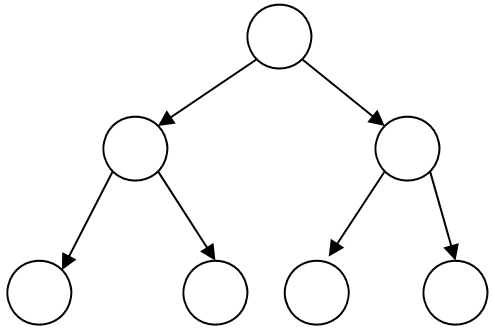
- Implementation
  - Knoten: Objekte mit Namen und anderen Attributen
  - Kanten: Verbindungen zwischen zwei Knoten u.U. mit Attributen

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

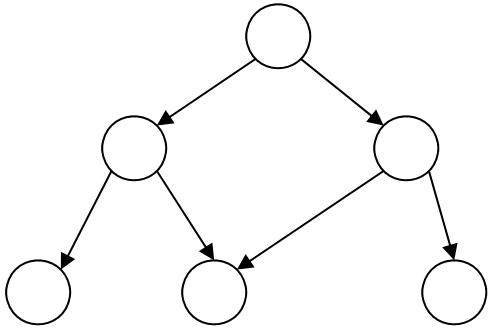
Verschiedene Graphenarten (Einführung)



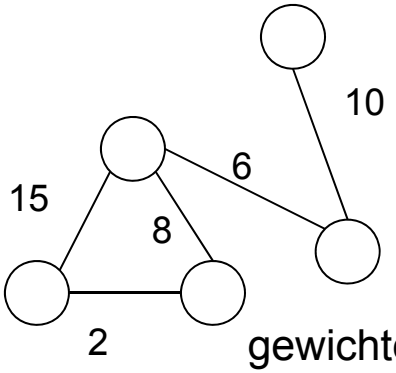
ungerichteter Graph



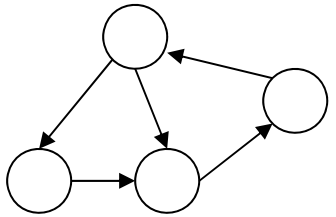
Baum



gerichteter azyklischer Graph



gewichteter Graph



gerichteter zyklischer Graph

Zurich University of Applied Sciences

zh aw School of Engineering

Definitionen 1

- Zwei Knoten x und y sind **benachbart (adjacent),** falls es eine Kante e_{xy}= (x,y) gibt.
- Bei einem **gerichteten (directed) Graph** sind die Kanten gerichtet.
- Bei einem **ungerichteten (undirected) Graph** sind die Kanten nicht gerichtet.
- Bei einem **verbundenen (connected) Graph** ist jeder Knoten von jedem anderen Knoten erreichbar.
- Ein Graph, der nicht verbunden ist, besteht aus **verbundenen Teilgraphen**.
- Zwei Knoten *u* und *v* eines gerichteten Graphen heissen **stark verbunden (strongly connected),** wenn es einen Pfad von *u* nach *v* und einen Pfad von v nach u gibt.

Zurich University of Applied Sciences

zh aw School of Engineering

Definitionen 2

- Eine **stark verbundene Komponente** eines gerichteten Graphen ist ein Teilgraph dessen Knoten alle miteinander stark verbunden sind.
- Besteht ein gerichteter Graph aus einer einzigen stark verbundenen Komponente, so heisst er **stark verbunden**.
- Ein Knoten, von dem alle andern Knoten erreichbar sind, heisst **Wurzel** des Graphen.
- Der gerichtete **azyklische Graph** wird auch DAG genannt **(directed acyclic graph).** Im Gegensatz zu einem Baum kann in einem DAG ein Knoten im allgemeinen von mehr als einem andern Knoten erreicht werden.
- Sind Anfangs- und Endknoten bei einem Pfad gleich, dann ist dies ein zyklischer Pfad oder **geschlossener Pfad** bzw. **Zyklus**.

Zurich University of Applied Sciences

zh aw School of Engineering

Definitionen 3

- 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.
- Graphen mit zyklischen Pfaden werden **zyklische Graphen** genannt.
- Falls nur wenige Kanten im Graph (bezogen auf den kompletten Graphen) fehlen, so ist der Graph **dicht (dense).**
- Graphen mit relativ wenigen Kanten sind **dünn (sparse).**

Beispiel: ein gerichteter Graph

{Def. von G = <V,E> siehe oben}

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

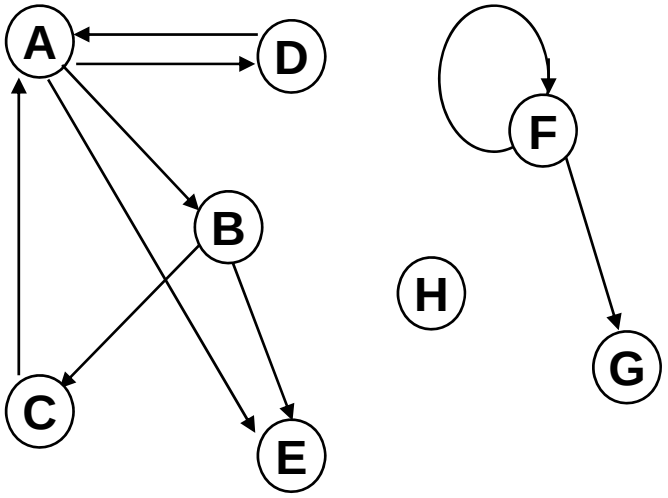
Er enthält einen Zyklus: (A, D, A).

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

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

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

(A, B, E, A) ist kein Pfad und kein Zyklus.

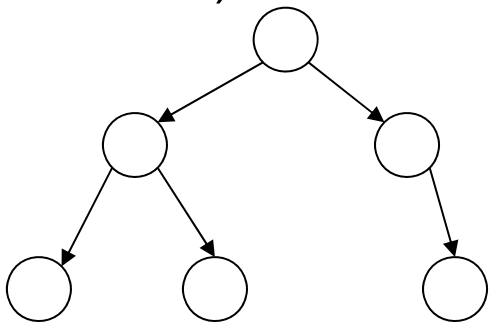


Ungerichtete Graphen

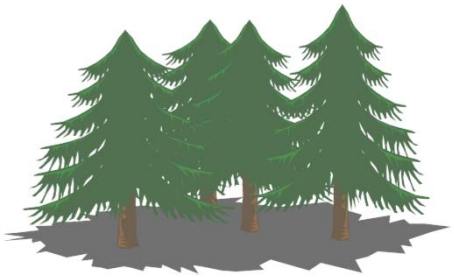
- Sei G = <V,E>.
- Falls für jedes e $\in$ E mit e = (v1,v2) gilt: e' = (v2,v1) $\in$ E,
  so heisst G ungerichteter Graph, ansonsten gerichteter Graph.
- Die Relation E ist in diesem Fall symmetrisch.
- Bei einem ungerichteten Graphen gehört zu jedem Pfeil von x nach y auch ein Pfeil von y nach x.
- Daher lässt man Pfeile ganz weg und zeichnet nur ungerichtete Kanten.

Baum als Spezialfall

- Ein zusammenhängender, gerichteter zyklenfreier Graph ist ein (gerichteter) **Baum**



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



Gewichtete Graphen

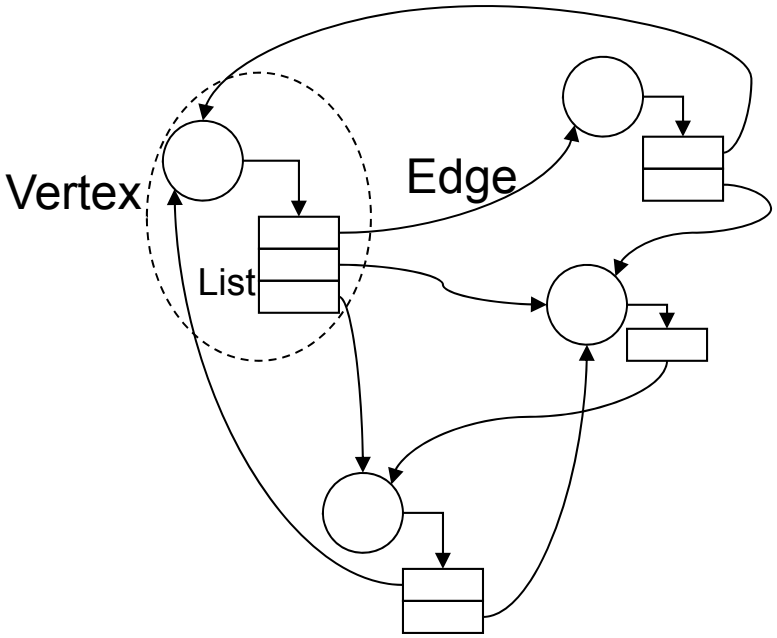
Ein Graph G = <V, E> kann zu einem **gewichteten Graphen** G = <V, E, g_w(E)> erweitert werden, wenn man eine Gewichtsfunktion
g_w: E $\rightarrow$ int (oder g_w: E $\rightarrow$ double) hinzunimmt, die jeder Kante e $\in$ E ein **Gewicht g_w(e)** zuordnet.

Eigenschaften

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

Implementation 1 : Adjazenz-Liste

- jeder Knoten führt eine (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<E> {

    // füge Knoten hinzu, tue nichts, falls Knoten schon
    // existiert
    public Vertex<E> addVertex (E source);

    // füge gerichtete Kante hinzu
    // erstelle Vertices, falls noch nicht vorhanden
    public void addEdge(E source, E dest, double weight);

    // finde den Knoten anhand seines Inhalts
    public Vertex<E> findVertex(E e);

    // Iterator über die Kanten eines Knoten
    public Iterator<Edge<E>> getEdges(Vertex<E> source);

    // Iterator über alle Knoten
    public Iterator<Vertex<E>> getVertices();
}
```

Die Klasse *Vertex* definiert einen Knoten

```
interface VertexI<E> {
        public E getElement();
        public Iterator<Edge<E>> getEdges();
        public void addEdge(Edge<E> edge);
}


class Vertex<E> implements VertexI<E> {
        private E element;
        private List<Edge<E>> adj;            // Adjazenzliste

        public Vertex( E e ){
                element = e;
                adj = new LinkedList<Edge<E>>( );
        }
        public E getElement() {return element;}
        public Iterator<Edge<E>> getEdges(){return adj.iterator();}
        public void addEdge(Edge<E> edge){adj.add(edge);}
}
```

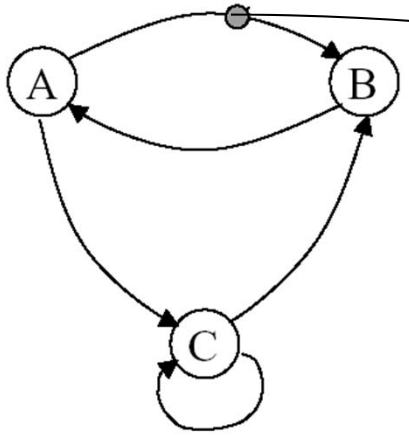
Zurich University of Applied Sciences

zh aw School of Engineering

Die Klasse *Edge* definiert eine Kante

```
public class Edge<E> {
    protected Vertex<E> dest;         // Zielknoten der Kante
    protected double weight;          // Kantengewicht
    public Edge(Vertex<E> d, double c) {
        dest = d;
        weight = c;
    }
    Vertex<E> getDest() {
        return dest;
    }
    double getWeight() {
        return weight;
    }
}
```

Implementation 2 Adjazenzmatrix



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

N x N (N = Anzahl Knoten) boolean Matrix;
true dort, wo Verbindung existiert

Zurich University of Applied Sciences

zh aw School of Engineering

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 im Zugriff
  - einfach zu implementieren
  - gut falls der Graph dicht

Adjazenzmatrix

- Distanzentabelle ist eine Adjazenzmatrix
- Ungerichtet → symmetrisch zur Hauptdiagonalen
- Im Prinzip reicht die eine Hälfte → Dreiecksmatrix
  oder die zwei Hälften haben unterschiedliche Bedeutung, z.B. Strassendistanz und Bahndistanz

	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.

Zurich University of Applied Sciences

zh aw School of Engineering

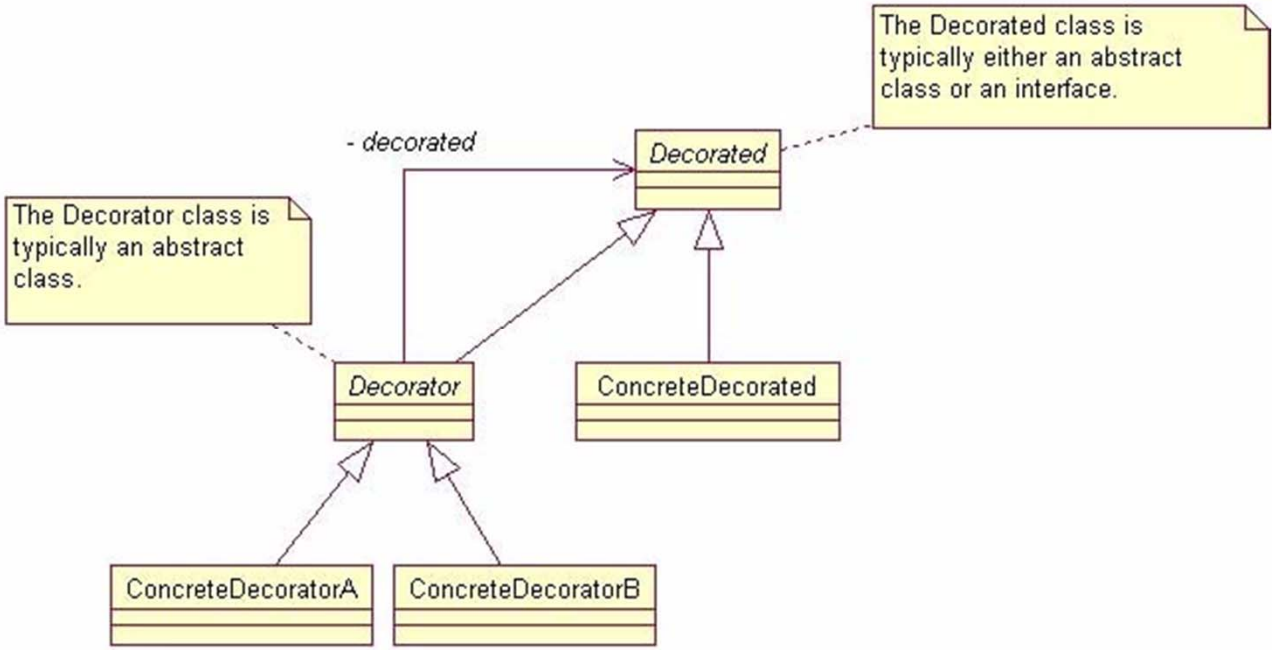
Das Decorator Pattern

- 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 VertexI {
    public String getName();
    public Iterator getEdges();
    public void addEdge(Edge edge);
}
```

```
// Decorator
abstract class DecoratedVertex implements VertexI {
    protected VertexI decorated;
    public DecoratedVertex (VertexI v) {decorated = v;}
    public String getName() {return decorated.getName();}
    public Iterator getEdges(){return decorated.getEdges();}
    public void addEdge(Edge edge){decorated.addEdge(edge);}
    public VertexI getDecorated() {return decorated;}
}
```

delegate

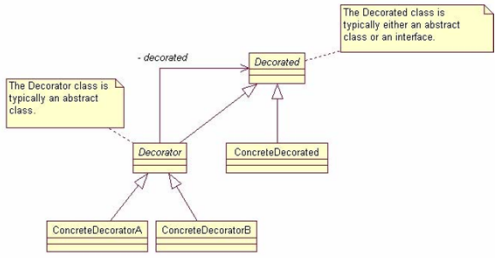
```
// ConcreteDecorated
class Vertex implements VertexI {
    private String name;
    private List<Edge>   adj;
    public Vertex(String name ){
        this.name = name;
        adj  = new LinkedList<Edge>( );
    }
    public String getName() {return name;}
    public Iterator<Edge> getEdges() {return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

```
// ConcreteDecoratorA
class MarkableVertex extends DecoratedVertex {
    private boolean mark=false;
    MarkableVertex (VertexI v) {super(v);}
    public void setMark (boolean m) {mark = m;}
    public boolean getMark() {return mark;}
}
```

```
// ConcreteDecoratorB
class TraversableVertex extends DecoratedVertex {
    private VertexI prev=null;
    public TraversableVertex(VertexI v) {super(v);}
    public void setPrev(VertexI v) {prev = v;}
    public VertexI getPrev() {return prev;}
}
```

new

```
// Client using a markable and traversable vertex
DecoratedVertex v = new TraversableVertex(new MarkableVertex(new Vertex("MyNode")));
String s = v.getName();
((MarkableVertex) v.getDecorated()).setMark(true);
((TraversableVertex) v.getDecorated()).setPrev(v);
```



Graph Algorithmen

Grundformen

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

Häufige Anwendungen

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

Weitere Algorithmen

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

Grundformen: Suchstrategien

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

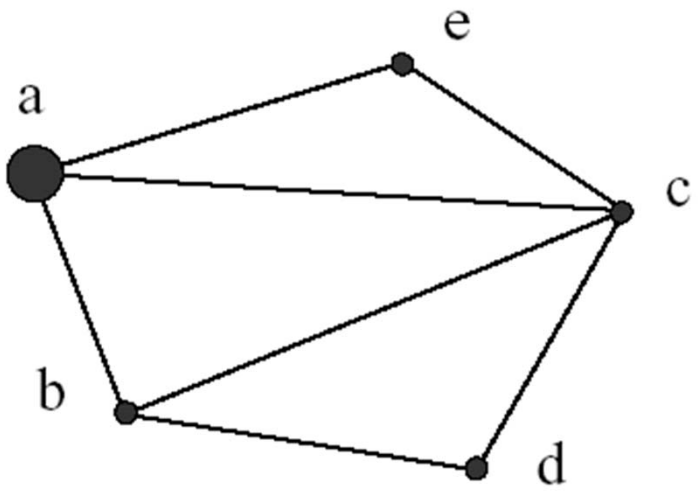
- Tiefensuche (depth-first search):
  - Ausgehend von einem Startknoten geht man **vorwärts (tiefer)** zu einem neuen unbesuchten Knoten, solange einer vorhanden (d.h. erreichbar) ist.

    Hat es keine weiteren (unbesuchten) Knoten mehr**,** geht man rückwärts und betrachtet die noch unbesuchten Knoten. Entspricht der **Preorder** Traversierung bei Bäumen.

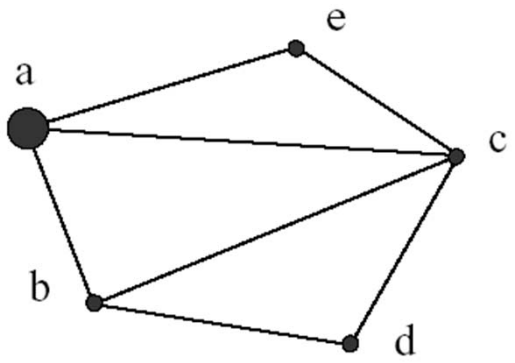
- Breitensuche (breadth-first search):
  - Ausgehend von einem Startknoten betrachtet man zuerst **alle benachbarten Knoten** (d.h. auf dem gleichen Level), bevor man einen Schritt weitergeht. Entspricht der **Levelorder** Traversierung bei Bäumen.

Übung

- Auf welche Arten kann folgender Graph (vom Knoten *a* aus) in Tiefensuche durchsucht werden?



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 worden sind, liegen auf dem Stack.

Tiefensuche

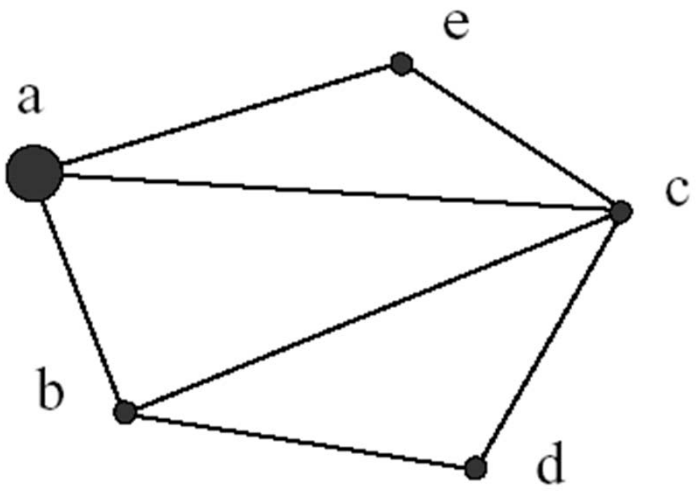
- Bei einem gerichteten Graphen liefert dieser Algorithmus alle Knoten, die von einem Startknoten erreichbar sind.
- Sind noch nicht alle Knoten besucht worden, sucht man in den noch nicht besuchten Knoten nach einem weiteren Startknoten und wiederholt den Algorithmus.

Eigenschaften

- Die bei einer Tiefensuche eines stark verbunden Graphen gefundenen Knoten definieren einen Baum.
- Die bei der Tiefensuche eines nicht stark verbundenen Graphen gefundenen Knoten definieren einen Wald.

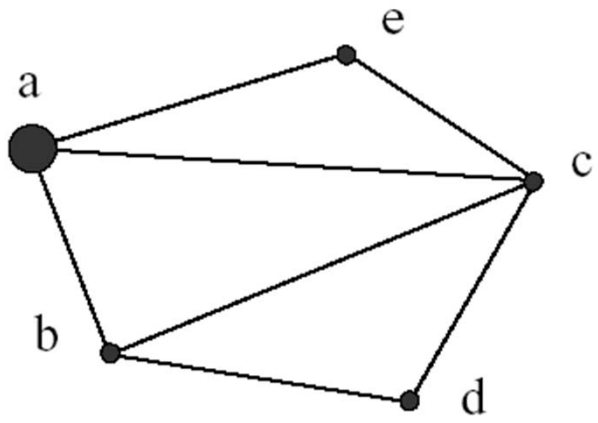
Übung

- Auf welche Arten kann folgender Graph in Breitensuche durchsucht werden



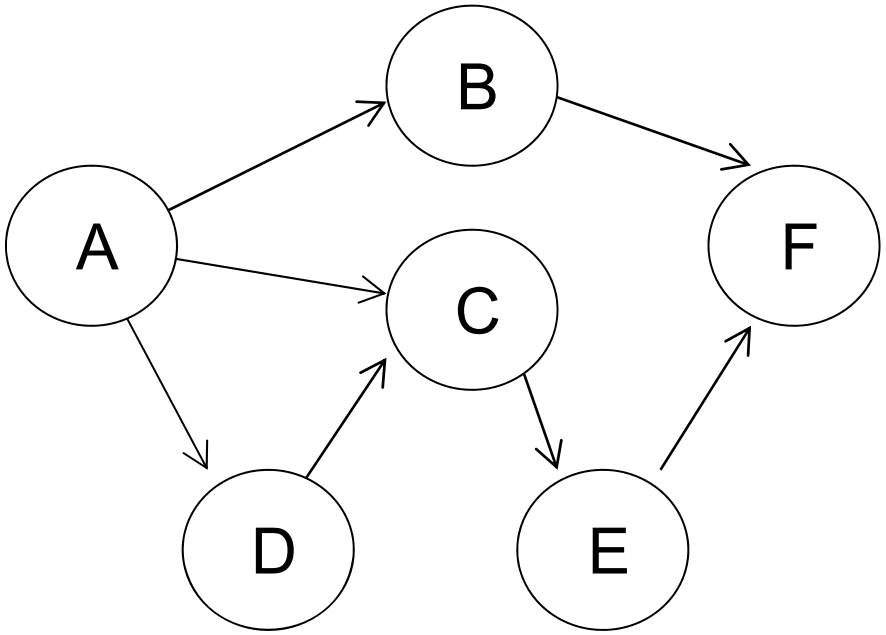
Breitensuche (Pseudo Code)

```
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: alle Kanten gleiches Gewicht

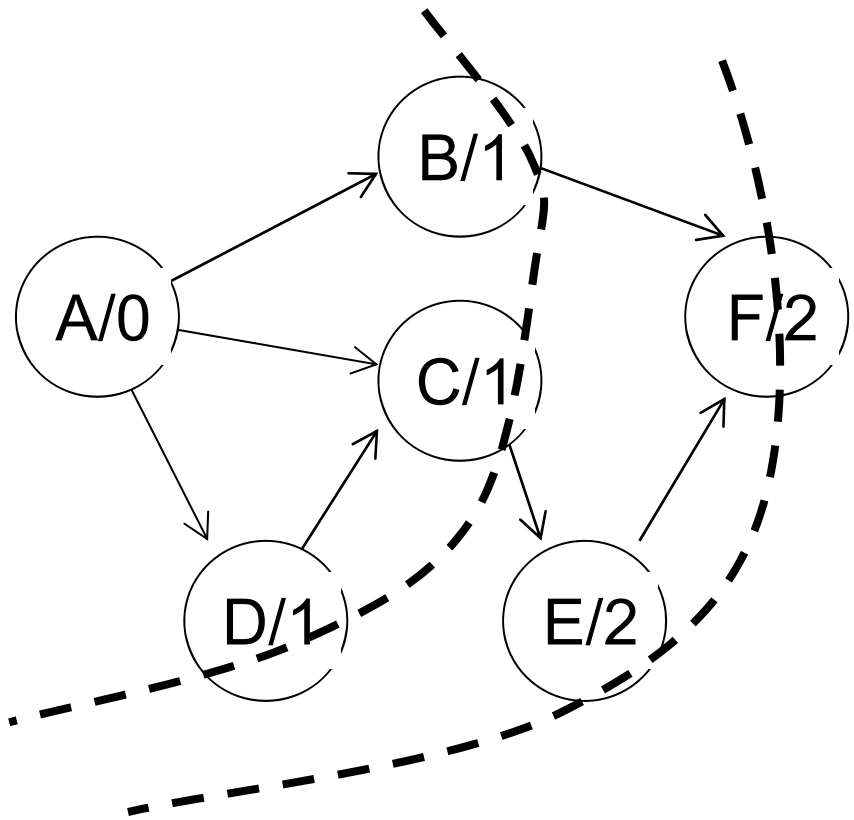
Gesucht ist der kürzeste Weg von einem bestimmten Knoten aus zu jeweils einem anderen.



Lösung : Von A nach F: A->B->F

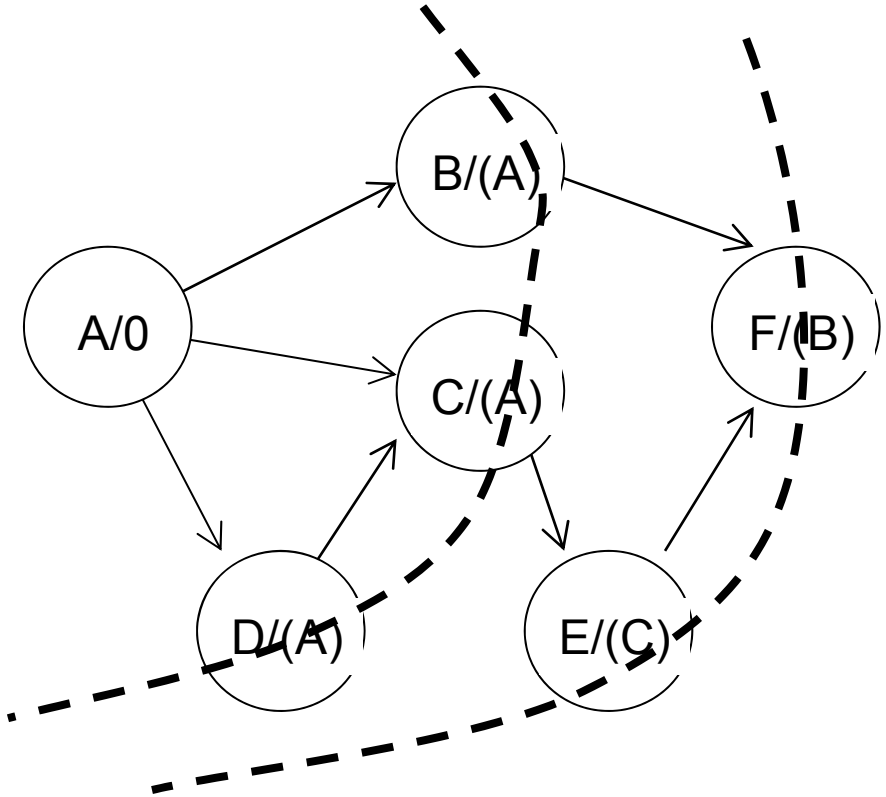
Algorithmus kürzester Pfad

Vom Startpunkt ausgehend werden die Knoten mit ihrer Distanz markiert.



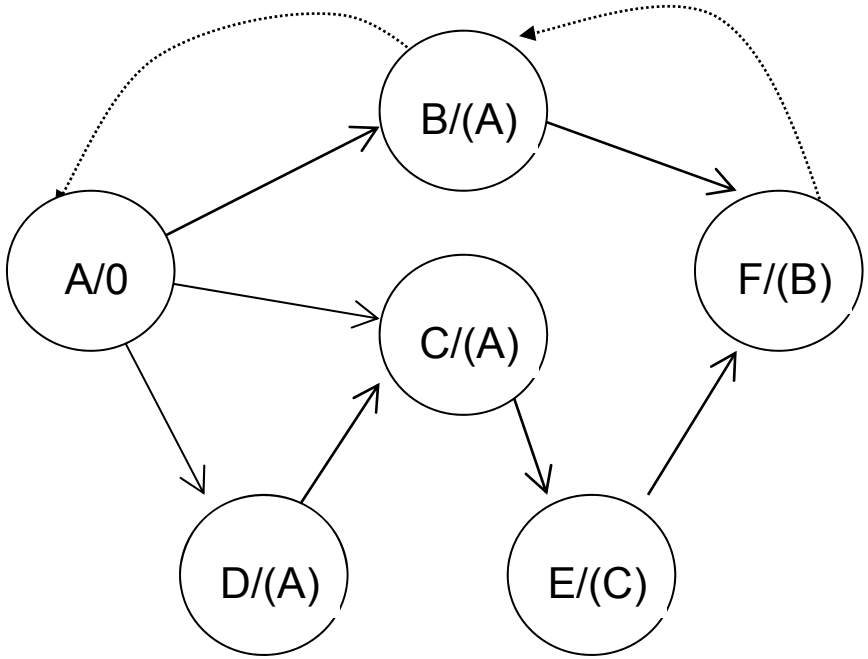
Algorithmus kürzester Pfad 2

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



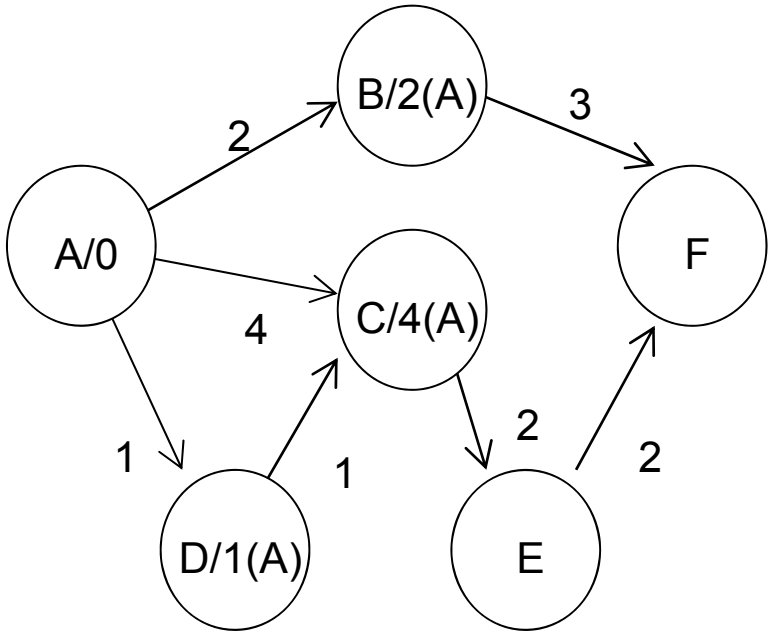
Algorithmus kürzester Pfad 3

Vom Endpunkt aus kann dann rückwärts der kürzeste Pfad gebildet werden



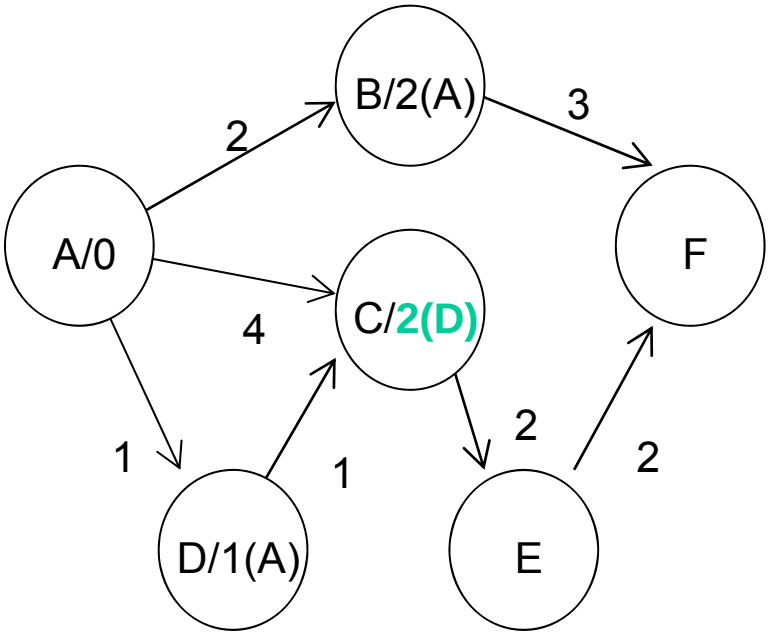
Kürzester Pfad bei gewichteten Kanten

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



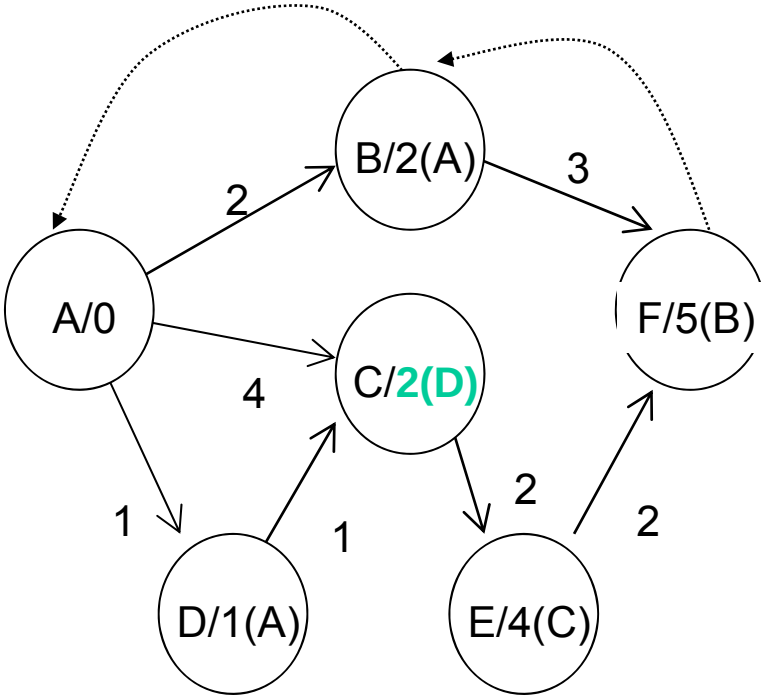
Kürzester Pfad bei gewichteten Kanten

Der Eintrag für C wird auf den neuen Wert gesetzt.



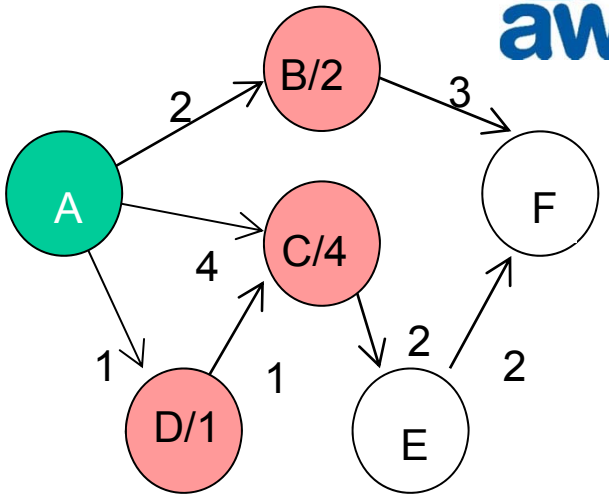
Kürzester Pfad bei gewichteten Kanten

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



Dijkstras Algorithmus

für kürzesten Pfad;
ergibt *Baum* der kürzesten Pfade



- 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** hat.

- Besuche diesen und bestimme dessen benachbarte Knoten

Pseudocode

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

```
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.

Implementation mittels PriorityQueue

```
void shortestPath() {
    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 + currentNode.dist
                if ((n.prev == null) || (dist < n.dist)) {
                    n.dist = dist;
                    n.prev = current;
                    q.enqueue(n, n.dist);
                }
            }
        }
    }
}
```

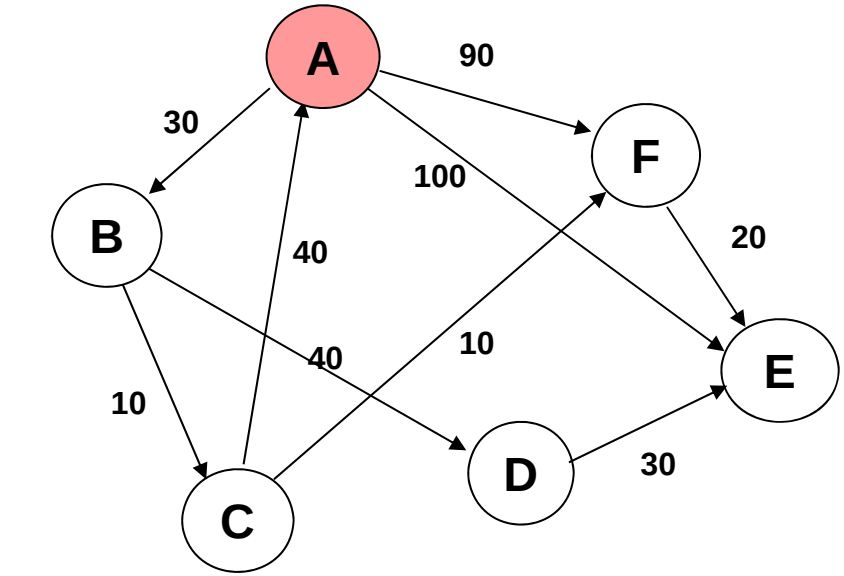
alle benachbarten Knoten

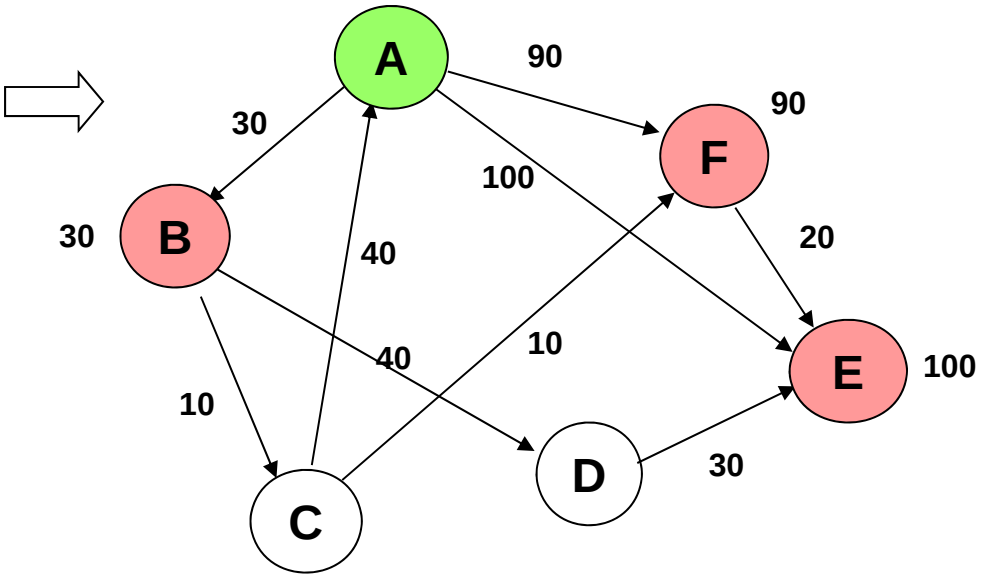
schon mal besucht

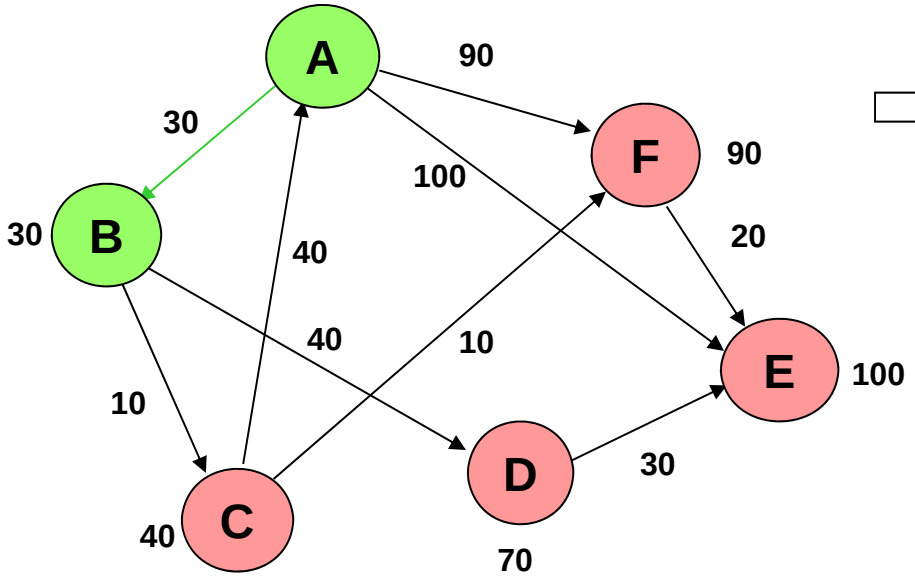
unbesehen

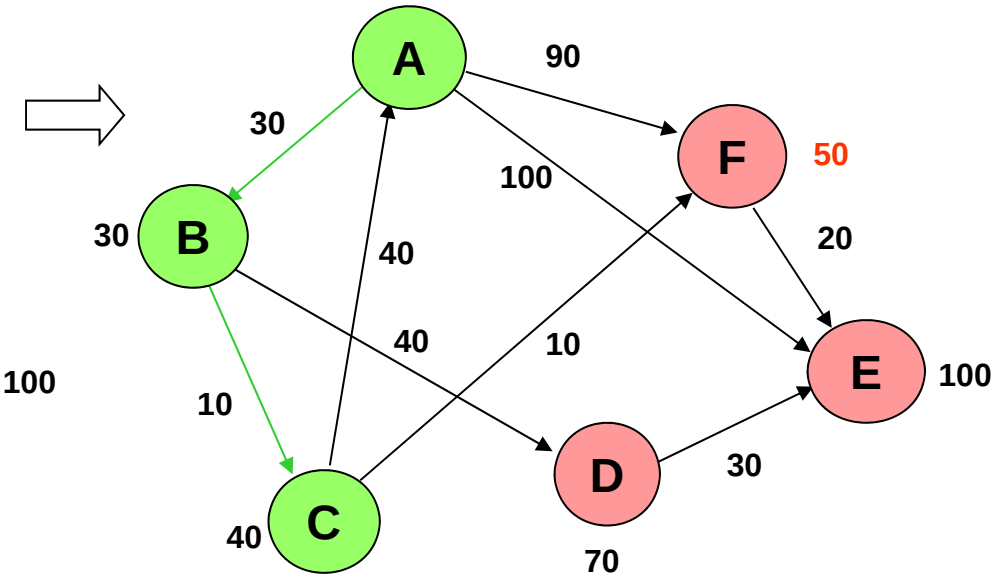
Rückweg

markierte Knoten
Knoten in Queue
unbesuchte Knoten

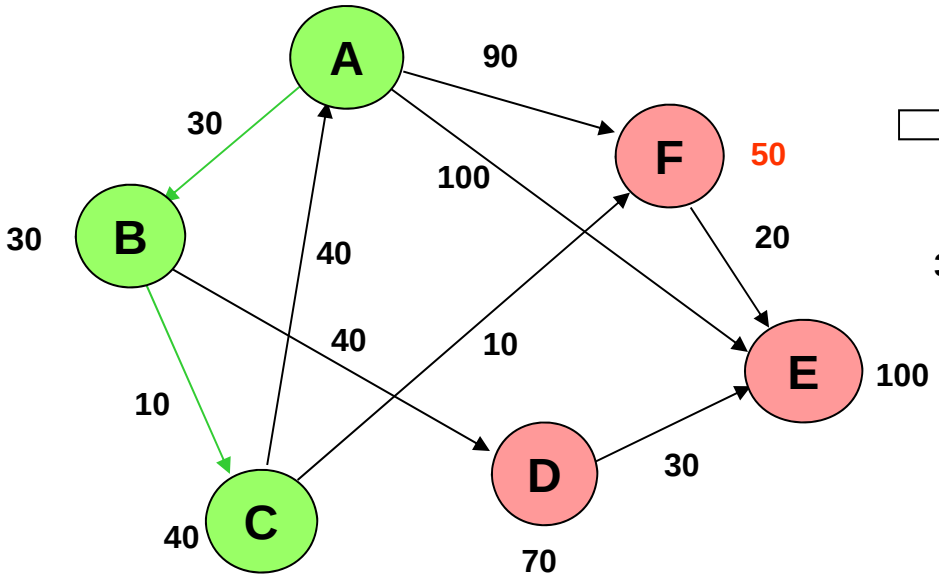


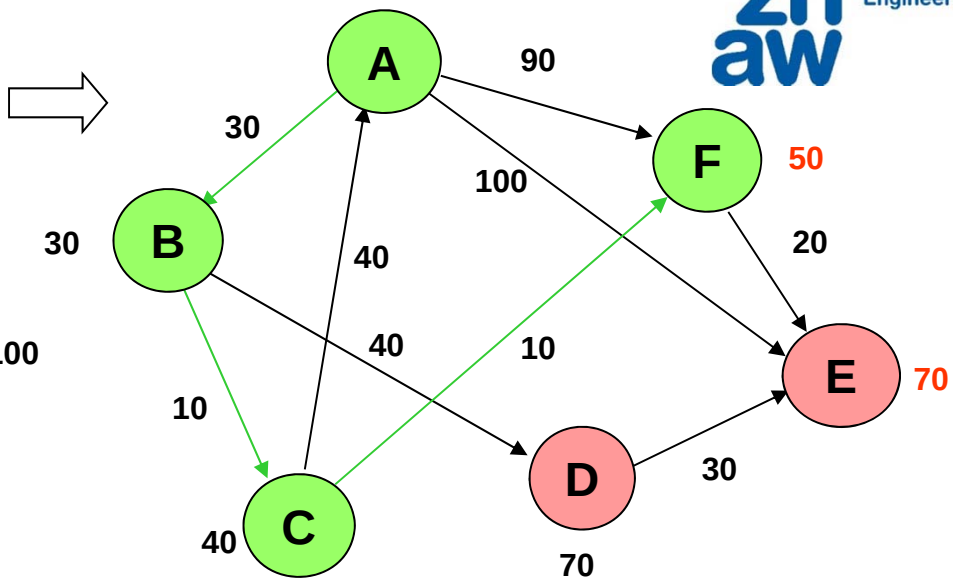


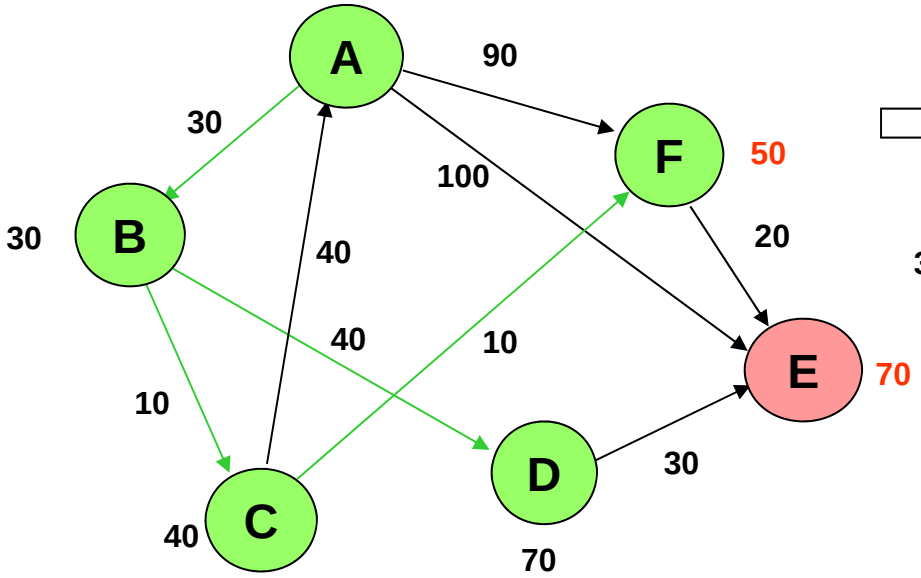


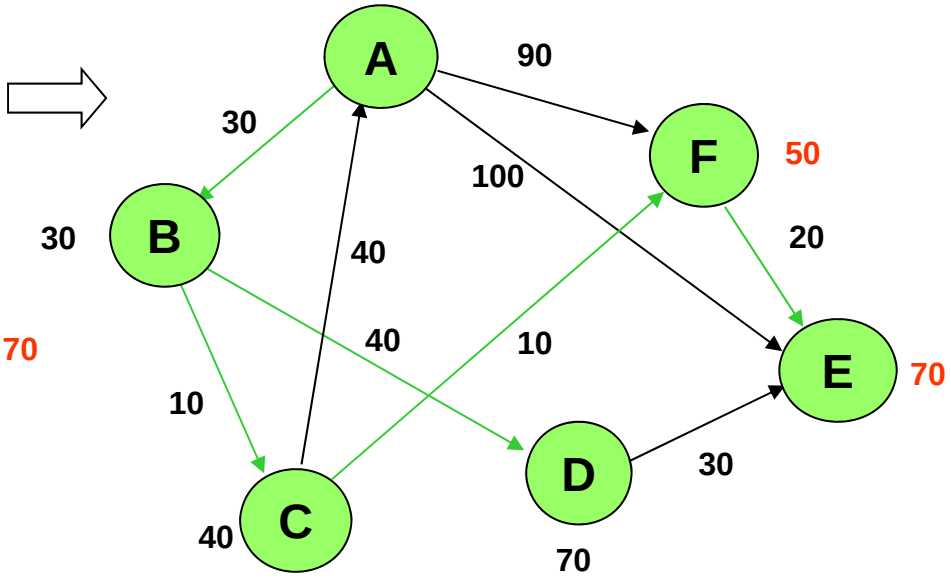


letzter Stand



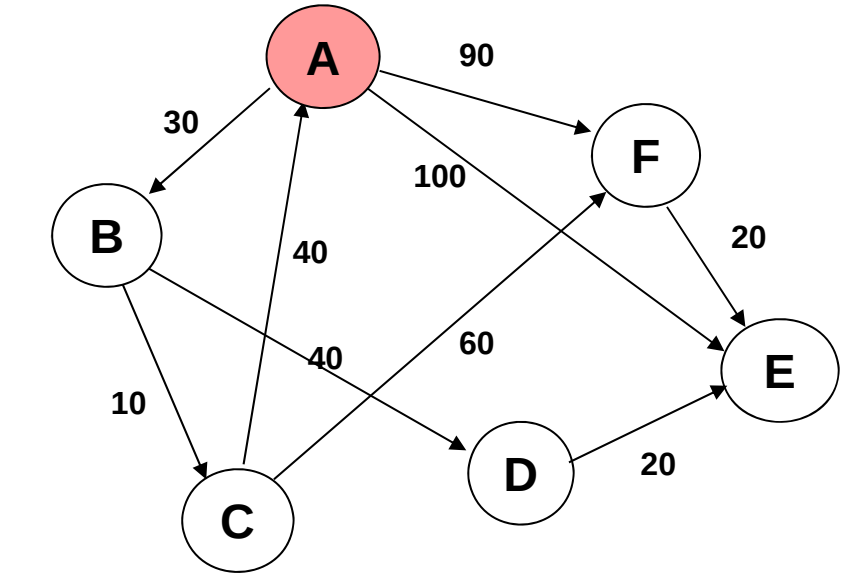


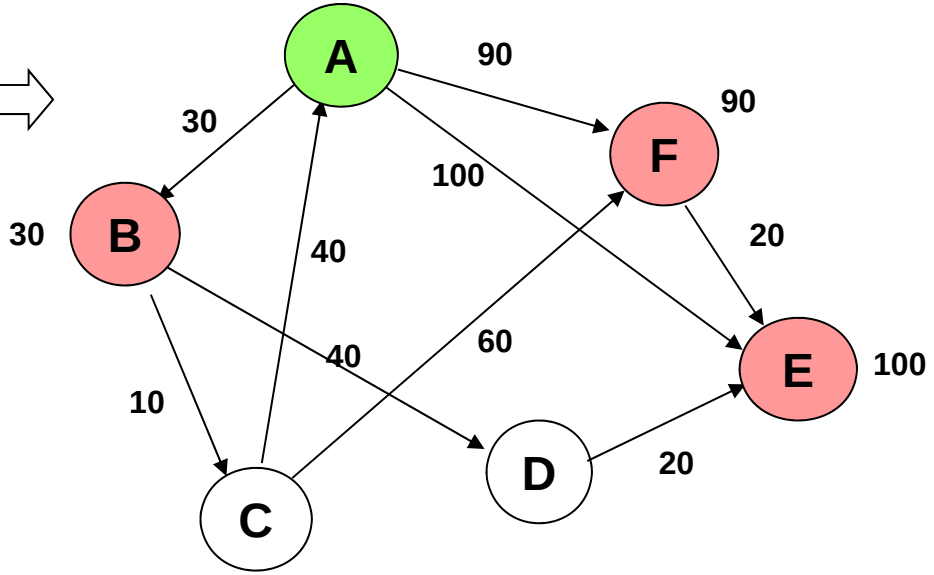


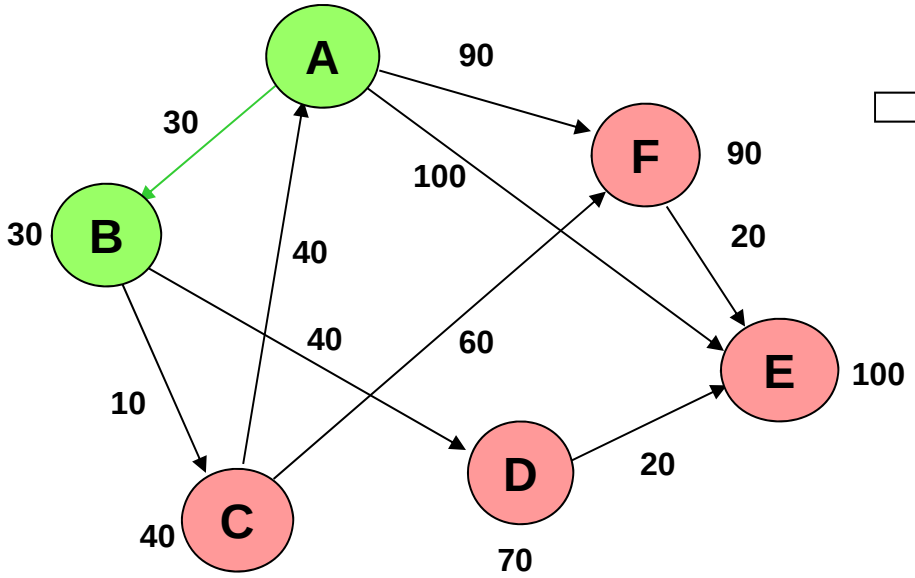


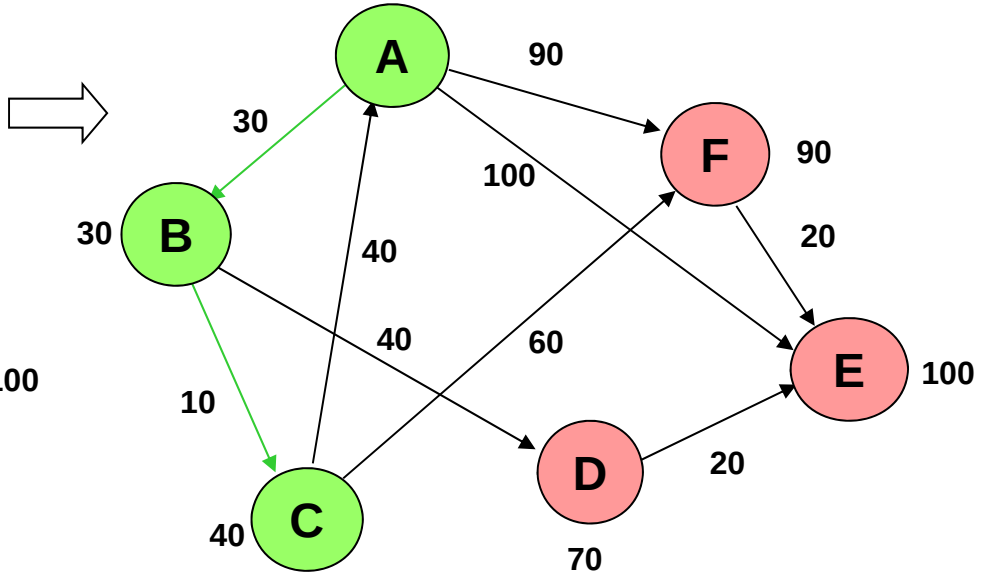
Variante

markierte Knoten
Knoten in Queue
unbesuchte Knoten

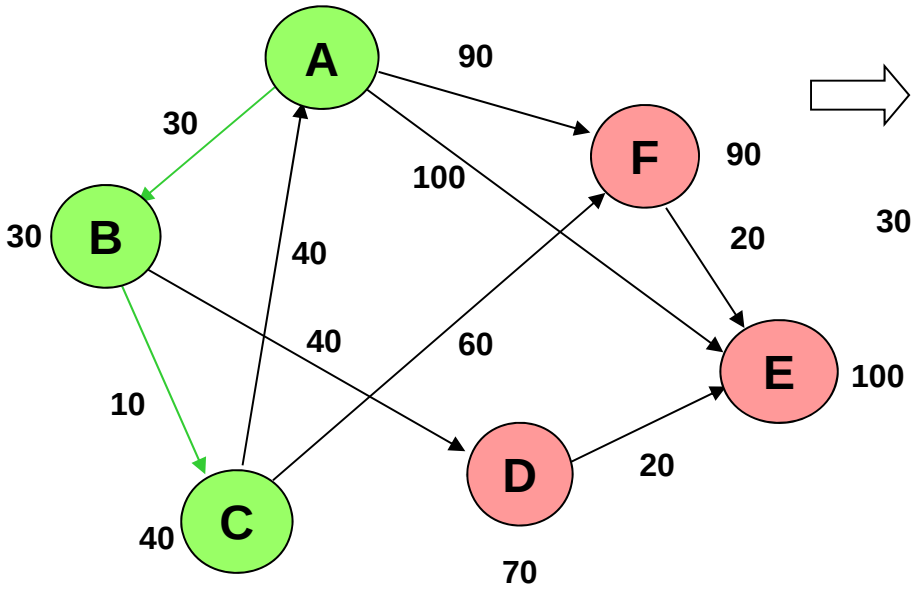


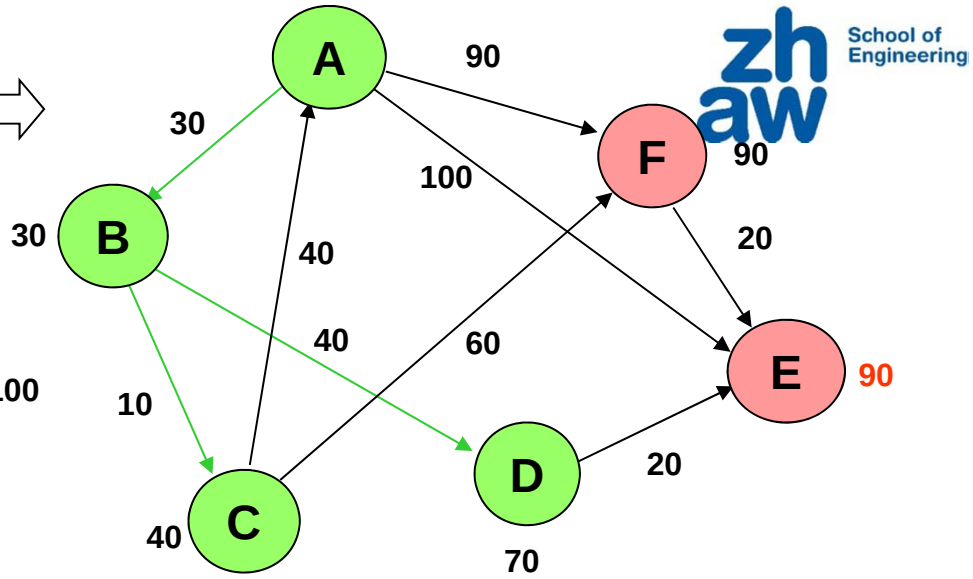


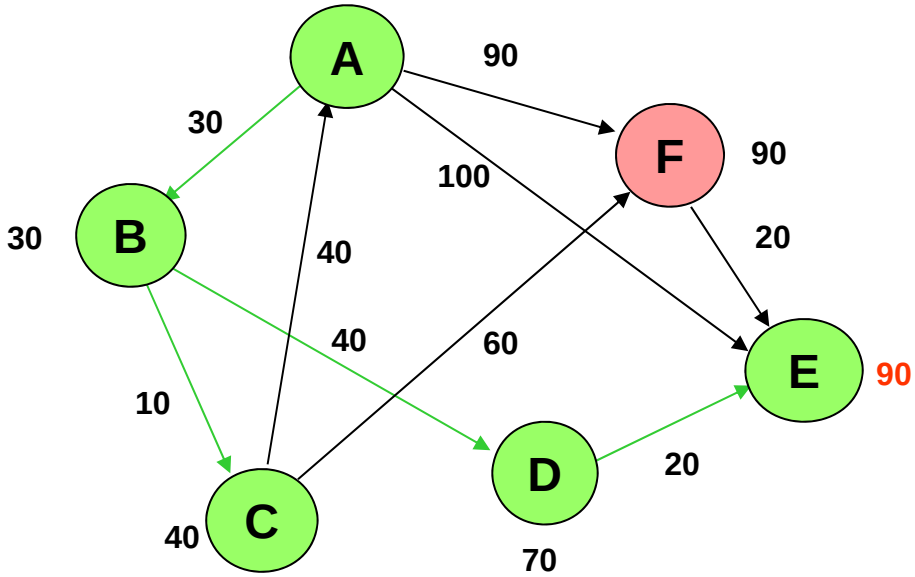


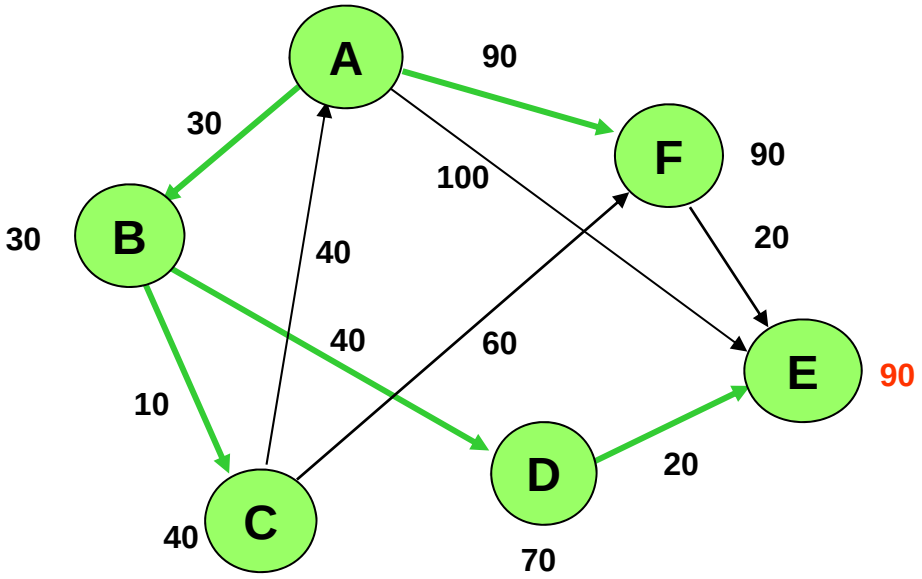


letzter Stand









Zurich University of Applied Sciences

zh aw School of Engineering

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.

Verschiedene Algorithmen

- Prim-Jarnik
- Kruskal

- Prim-Jarnik ist ähnlich wie Dijkstras Algorithmus
- Dijkstras Algorithmus liefert einen Spannbaum (die grünen Kanten im vorhergehenden Beispiel) aber nicht unbedingt einen minimalen.

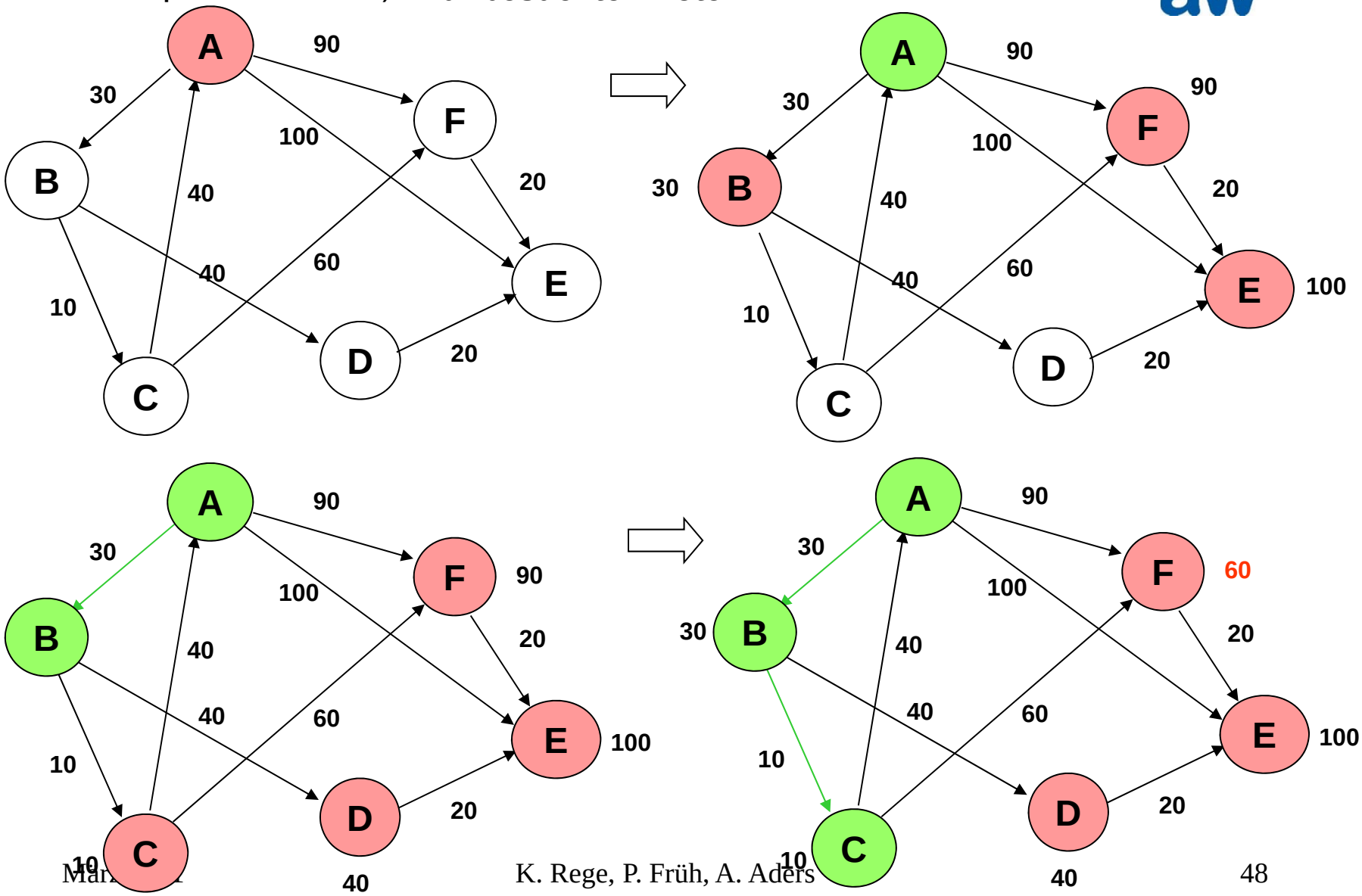
Prim-Jarnik mittels PriorityQueue

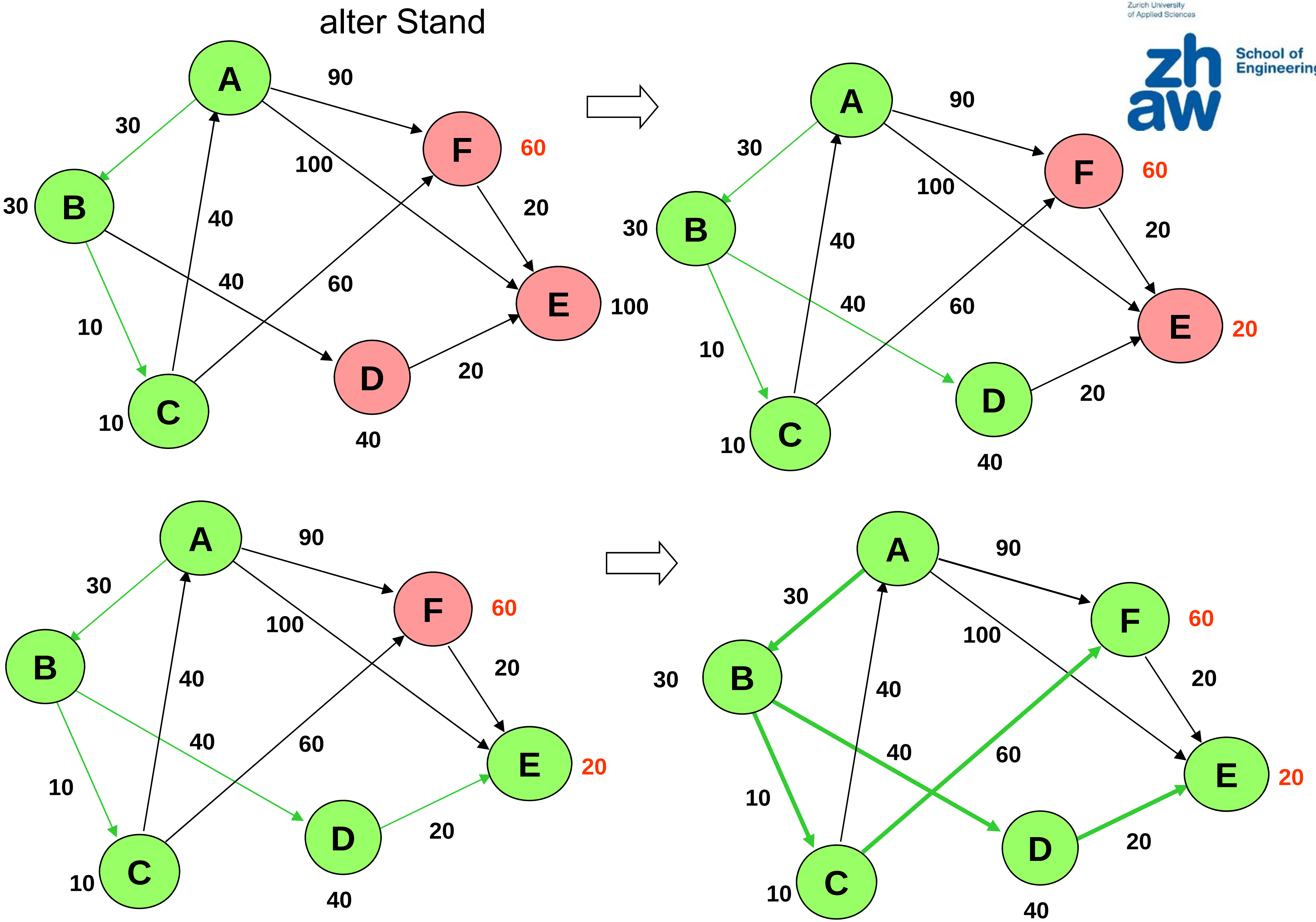
```
void MST() {
    q = new PriorityQueue()
    startNode.dist = 0;
    q.enqueue(startNode,0)
    while (!q.empty()) {
        current = q.dequeue();
        mark current
        for all edges e of current {
            n = e.node;
            if (!(marked(n)) {
                dist = e.dist;
    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

Spanning Tree
(gleicher Graph wie vorher)

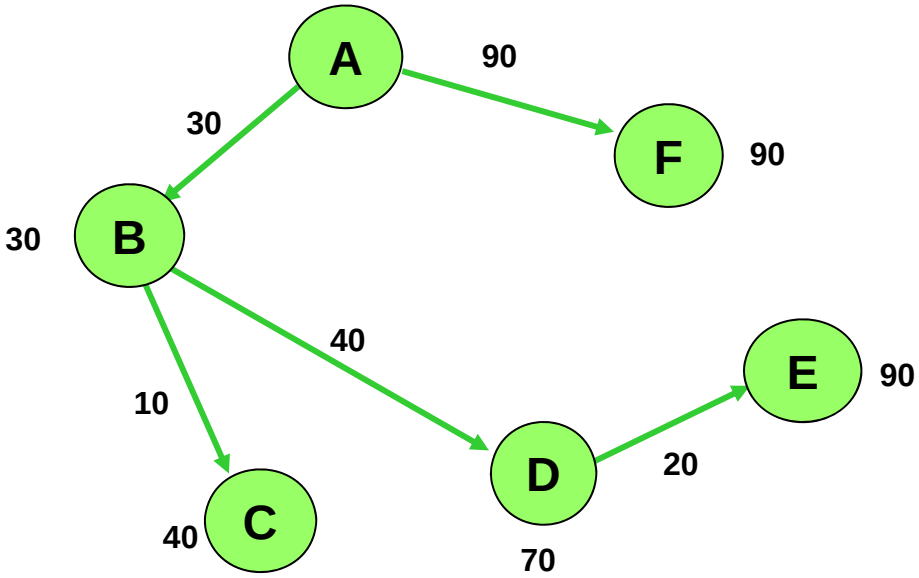
markierte Knoten
Knoten in Queue
unbesuchte Knoten




alter Stand
A
B
C
D
E
F
30
90
100
40
40
60
10
20
20
30
10
40
60
100
A
B
C
D
E
F
30
90
100
40
40
60
10
20
20
30
10
40
60
20
A
B
C
D
E
F
30
90
100
40
40
60
10
20
20
30
10
40
60
20
A
B
C
D
E
F
30
90
100
40
40
60
10
20
20
30
10
40
60
20

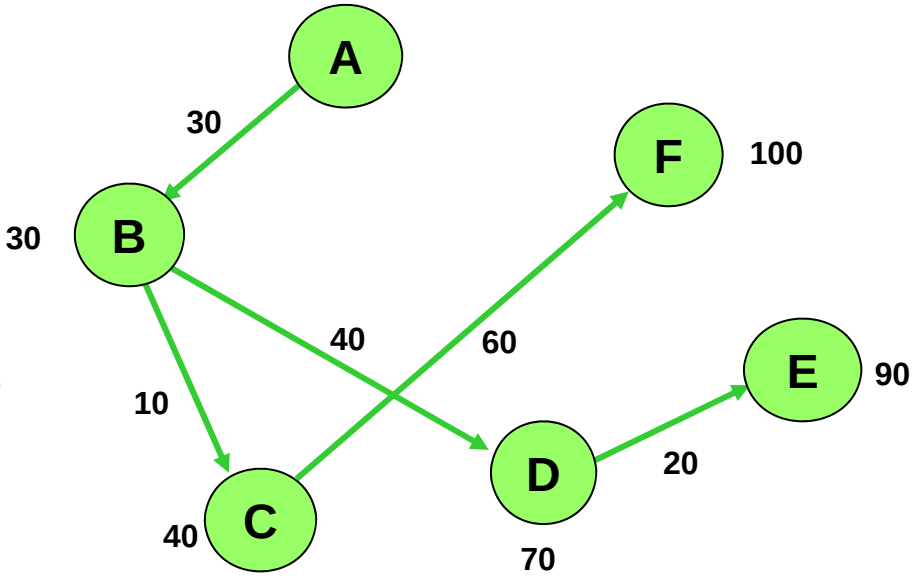
Shortest Path vs. Minimum Spanning Tree

Dijkstra: Shortest path



Total = 190
Path A-F = 90

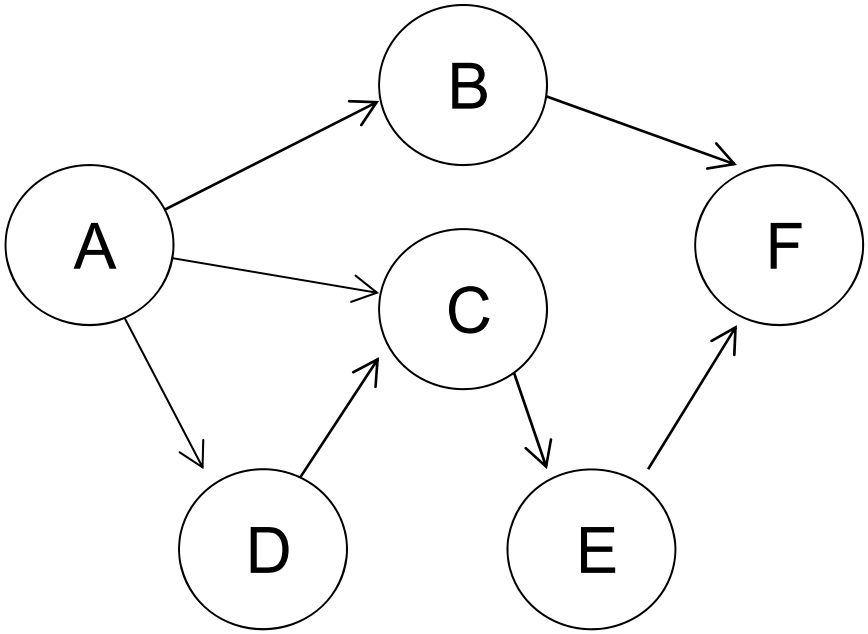
MST: Minimum sum



Total = 160
Path A-F = 100

Sortierung eines gerichteten Graphen: topologisches Sortieren

Die Knoten eines gerichteten, azyklischen Graphs in einer korrekten Reihenfolge auflisten.

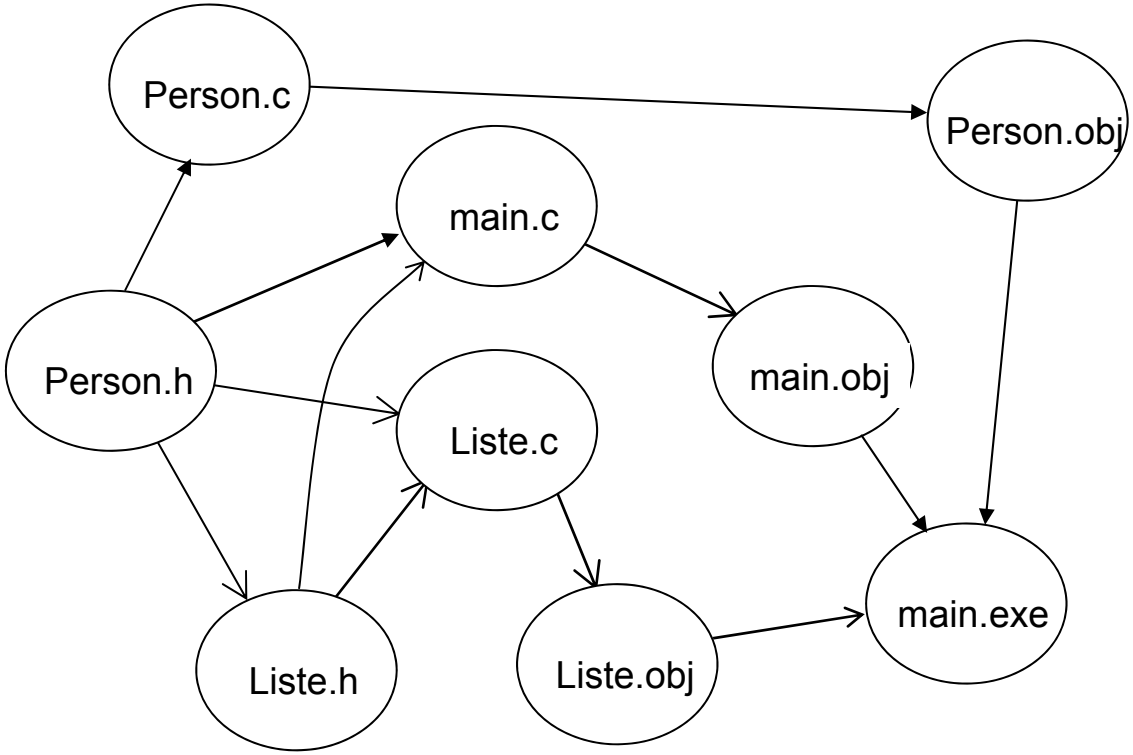


Anwendung:
Die Kanten geben Abhängigkeiten zwischen Modulen in einem Programm an. Topologisches Sortieren zeigt eine mögliche Compilationsreihenfolge.

Lösung : A B D C E F oder A D C E B F

Topologisches Sortieren, Beispiel

- Programm für eine Liste von Personen



Zurich University of Applied Sciences

School of Engineering

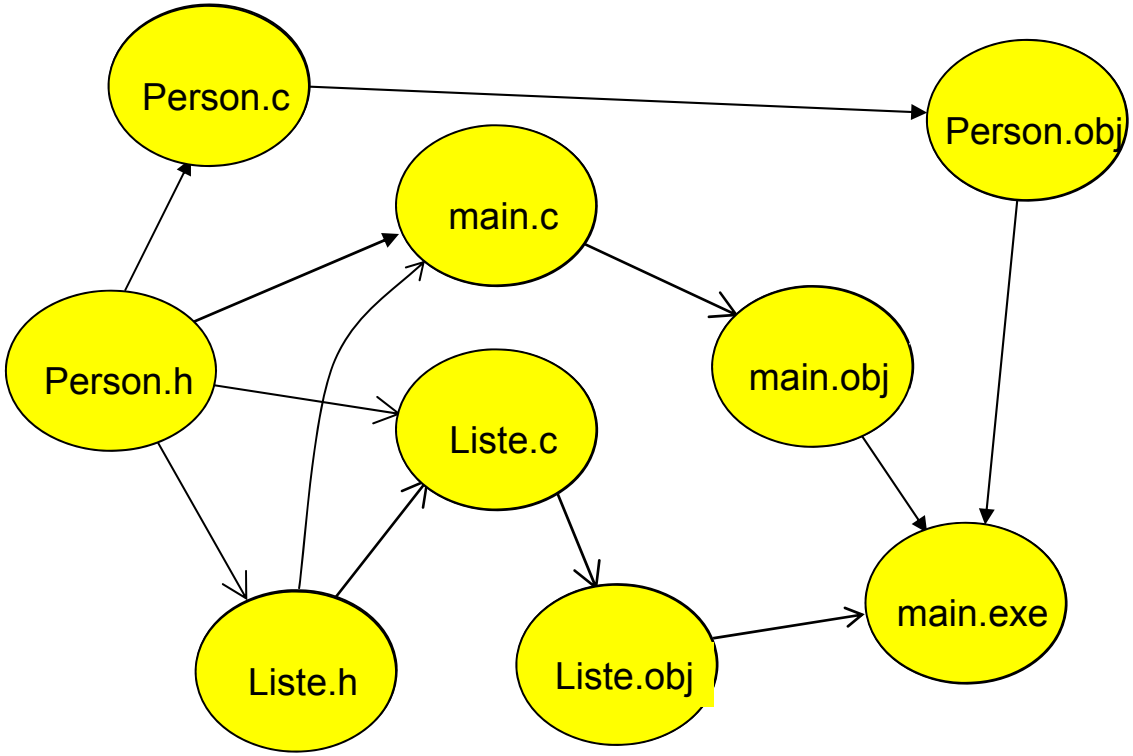
Topologisches Sortieren, Übung

- Entwerfen Sie einen Algorithmus in Pseudocode, der die Knoten eines gerichteten azyklischen Graphen in der richtigen Reihenfolge ausgibt.
- Hinweise:
  - Ein Knoten der keine Eingangskanten hat, kann sofort ausgegeben werden.
  - Ausgangskanten können (gedanklich) entfernt werden.

- Wie kann dieser Algorithmus verwendet werden, um festzustellen, ob ein Graph Zyklen hat?

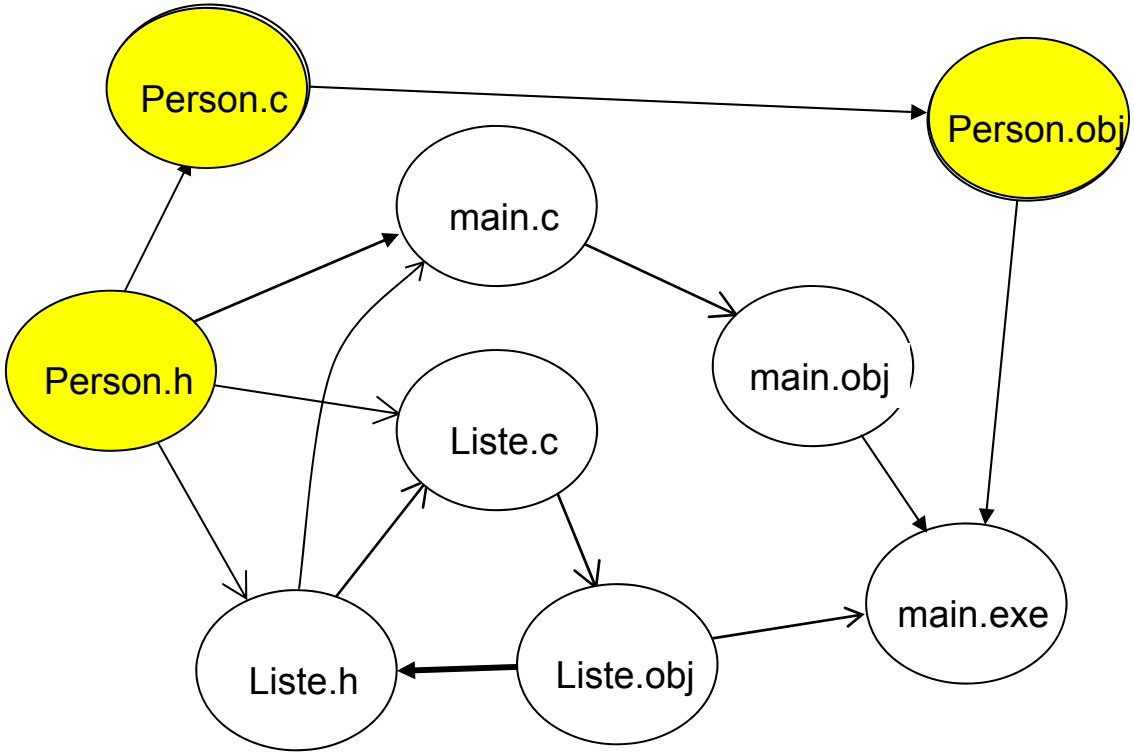
Topologisches Sortieren, Beispiel

- Programm für eine Liste von Personen



Topologisches Sortieren, Beispiel

- **Zyklus**



Maximaler Fluss

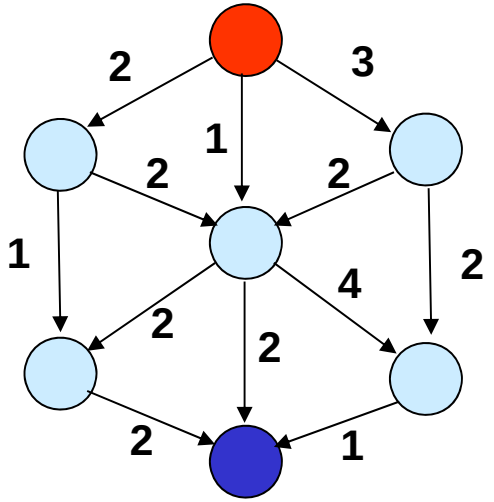
(aus www.cs.brown.edu/courses/cs016/book/slides/MaximumFlow2x2.pdf)

Fluss-Netzwerke:

- Ein gerichteter Graph. Die Werte der Kanten geben den maximalen Fluss durch diese Kante an (Kapazität).
- Zwei ausgezeichnete Knoten:
  - s (Source). Knoten hat nur Ausgangskanten.
  - t (Target). Knoten hat nur Eingangskanten.
- Gesucht: Maximaler Fluss von s nach t.

Anwendungen:

- Strassenverkehr
- Hydraulische Probleme
- Datennetzwerke
- Elektrische Netzwerke

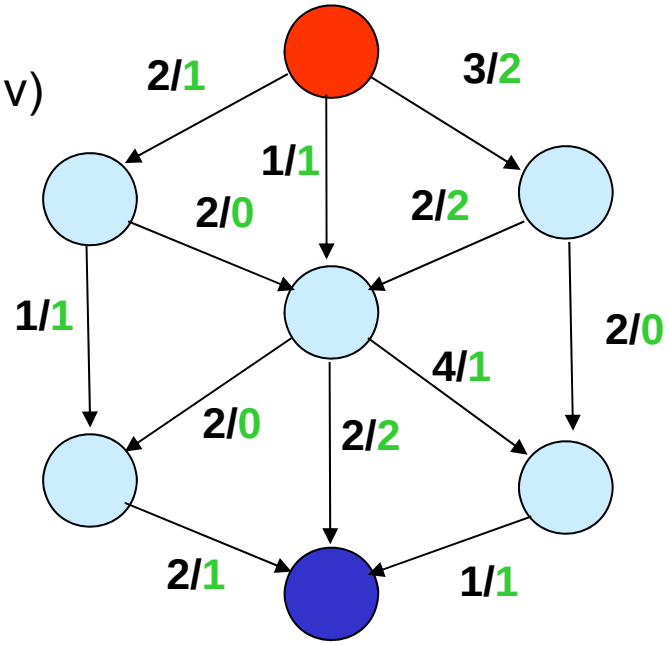


Maximaler Fluss

Fluss

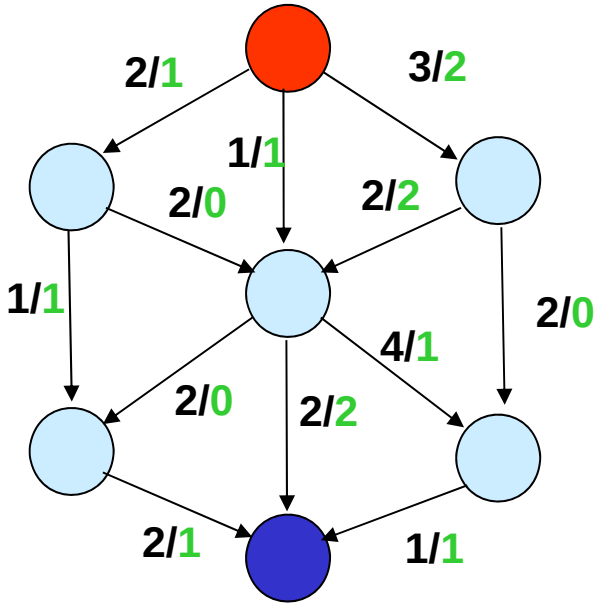
Zwei Bedingungen:

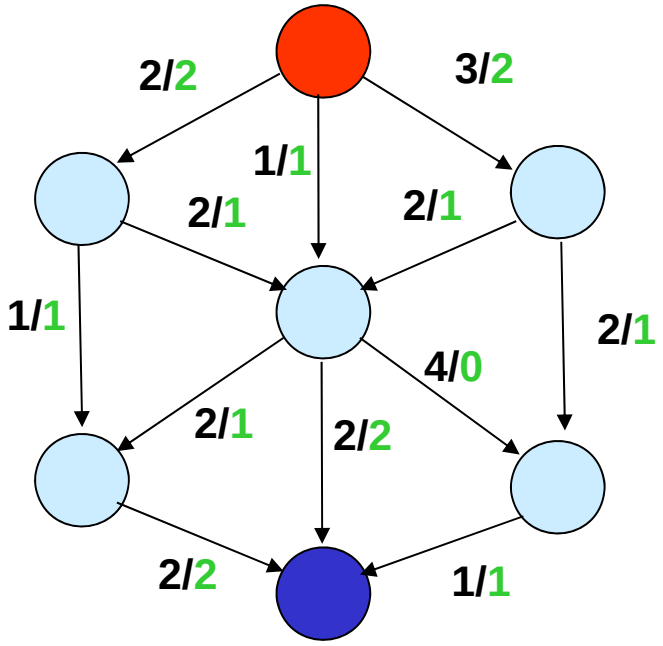
1. Der Fluss f(u,v) durch eine Kante e(u,v) darf nicht grösser als deren Kapazität c(u,v) sein (Kapazitätsregel).
   $0 \leq f(u,v) \leq c(u,v) \quad \forall\ e(u,v)$
2. Was in einen Knoten hineinfliesst, muss auch wieder herausfliessen (Kontinuitätsregel).
   $\Sigma f(u,v) = \Sigma f(v,w) \quad \forall\ v \neq s,t,$
   $u \in in(v),\ w \in out(v)$



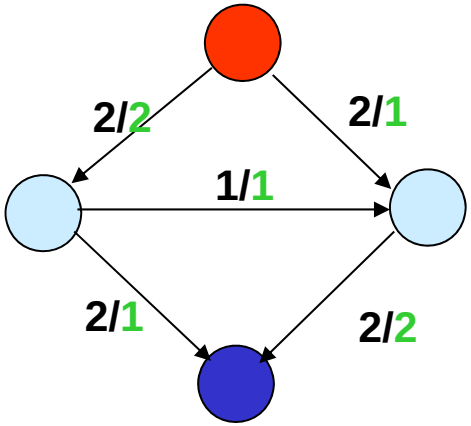
Maximaler Fluss

- Der Fluss aus der Quelle ist gleich dem Fluss in die Senke.
- Der Graph auf der rechten Seite zeigt nochmals das gleiche Netzwerk, aber diesmal mit maximalem Fluss.
- Beachte: Auf einzelnen Kanten ist der Fluss reduziert worden.

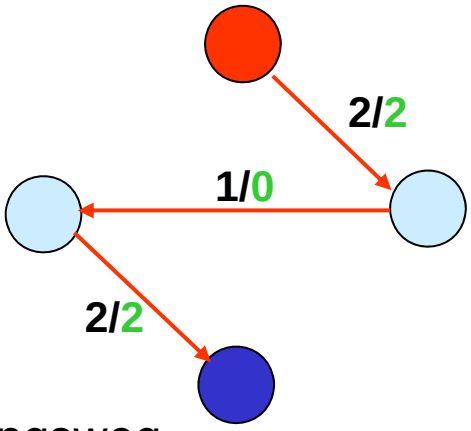




Maximaler Fluss

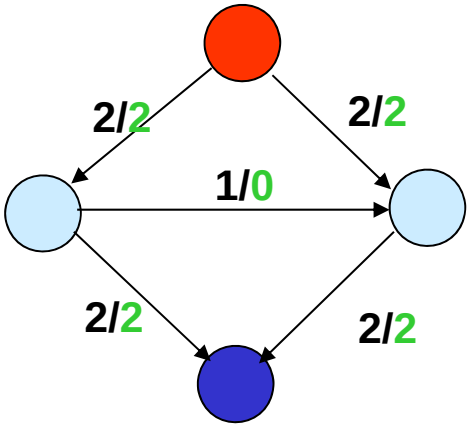


Ist dieser Fluss maximal (Wert=3)?



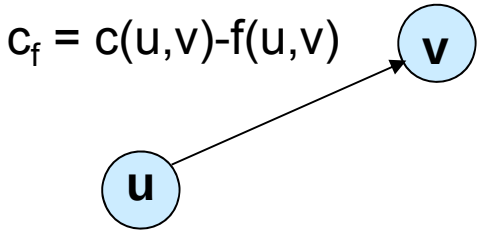
Erweiterungsweg
(Augmenting Path)

Nein. Fluss mit Wert 4

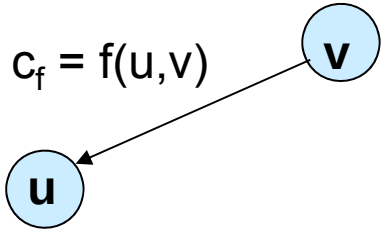


Maximaler Fluss

Erweiterungsweg



Vorwärtskante
f(u,v) < c(u,v)
Fluss kann erhöht werden.
Kantenwert: c(u,v)-f(u,v)



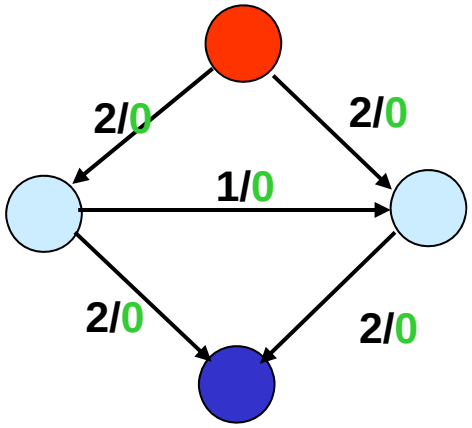
Rückwärtskante
f(u,v) > 0
Fluss kann verkleinert werden.
Kantenwert: f(u,v)

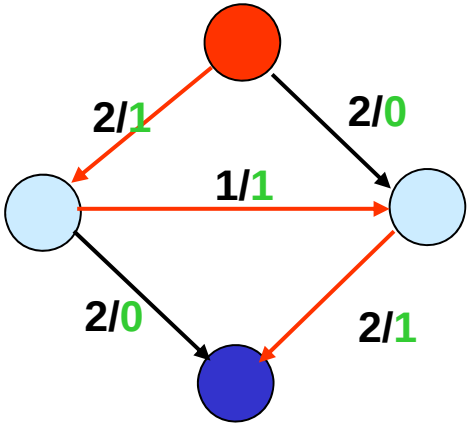
Ein *Erweiterungsnetzwerk* (Residual Network) ist ein Graph mit denselben Knoten wie das ursprüngliche Netz, aber mit den nebenstehenden Kantenwerten c_f. Kanten mit Wert 0 werden entfernt.

Ein *Erweiterungsweg* (Augmenting Path) ist ein Pfad im Erweiterungsnetzwerk von der Quelle bis zum Ziel.

Satz (Ford, Fulkerson):
Ein Fluss hat maximalen Wert dann und nur dann, wenn es keinen Erweiterungsweg gibt.

Maximaler Fluss





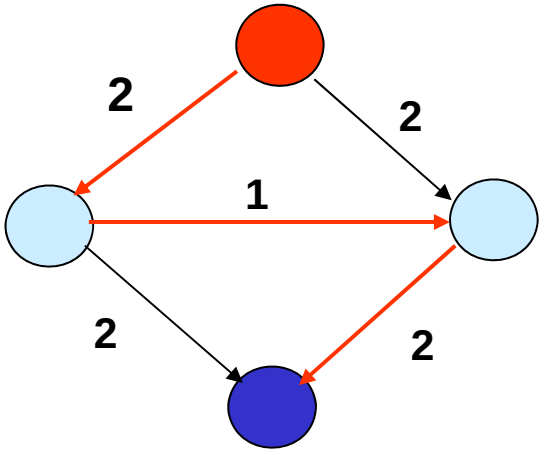
0: Initialisiere Fluss mit 0

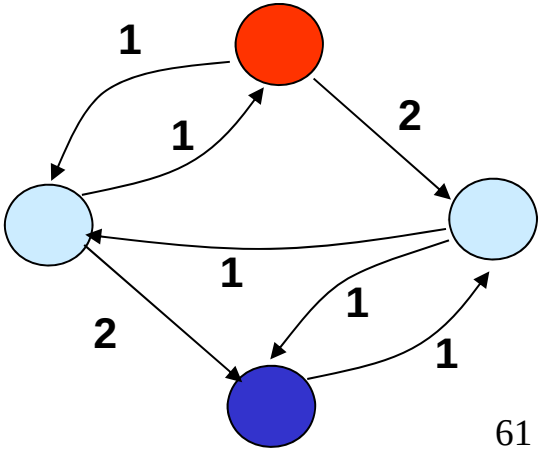
1.1: Finde einen Erweiterungsweg (es gäbe auch andere)

1.2: Finde den maximal möglichen Fluss Δf auf diesem Weg ➔ 1.

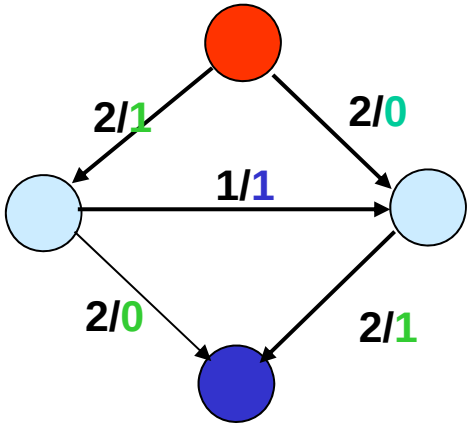
1.3: Korrigiere die Kantenwerte im Erweiterungsnetzwerk um Δf.

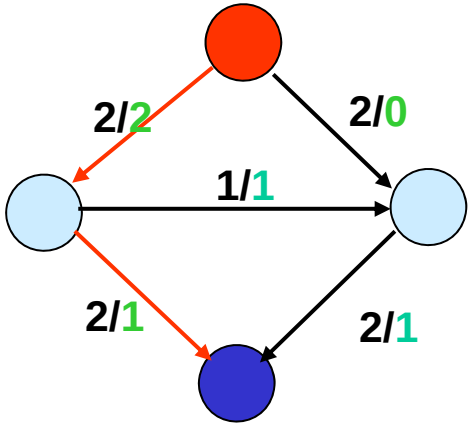
Erweiterungsnetz





Maximaler Fluss



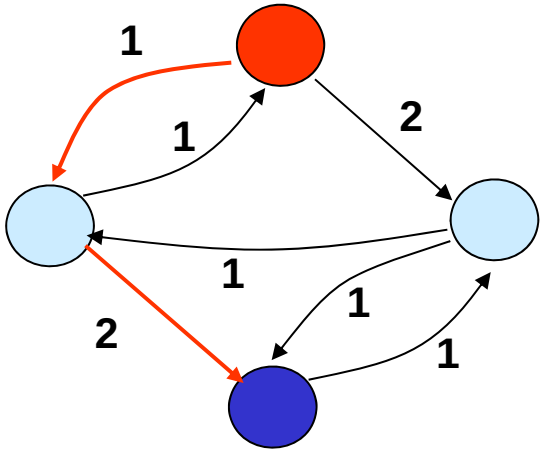


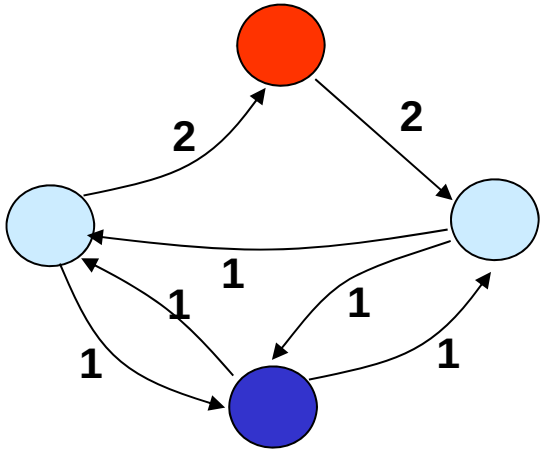
2.1: Finde einen Erweiterungsweg

2.2: Finde den maximal möglichen Fluss Δf auf diesem Weg ➔ 1.

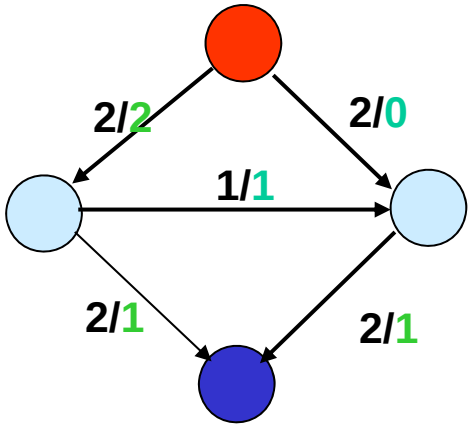
2.3: Korrigiere die Kantenwerte im Erweiterungsnetzwerk um Δf.

Erweiterungsnetz





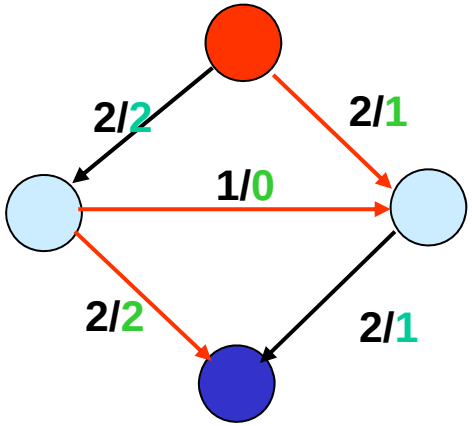
Maximaler Fluss



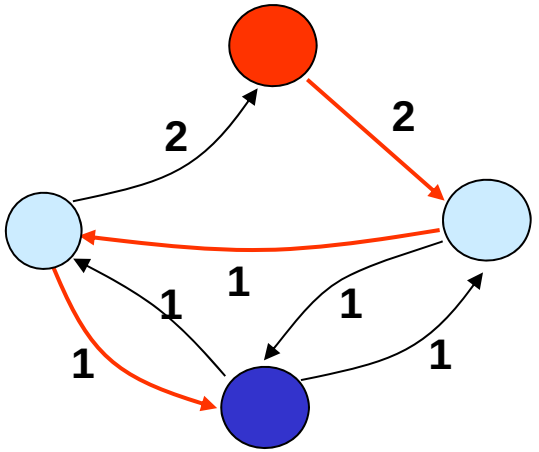
3.1: Finde einen Erweiterungsweg

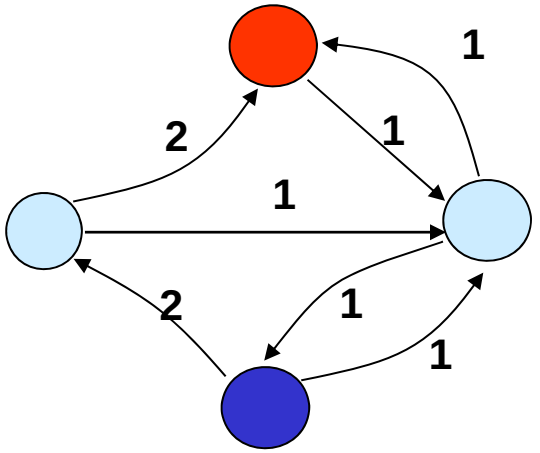
3.2: Finde den maximal möglichen Fluss Δf auf diesem Weg ➔ 1.

3.3: Korrigiere die Kantenwerte im Erweiterungsnetzwerk um Δf.

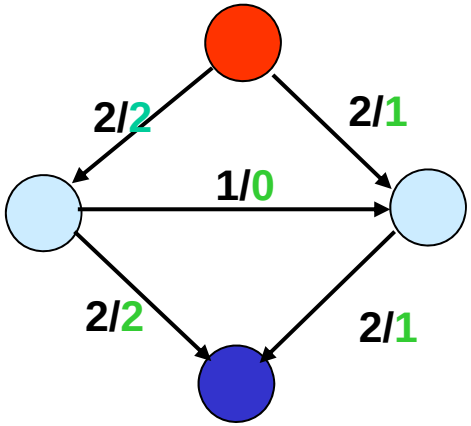


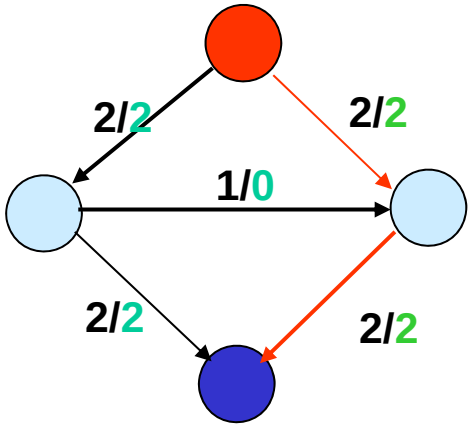
Erweiterungsnetz





Maximaler Fluss





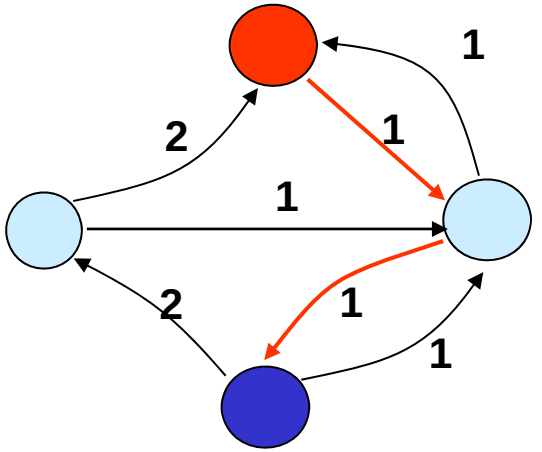
4.1: Finde einen Erweiterungsweg

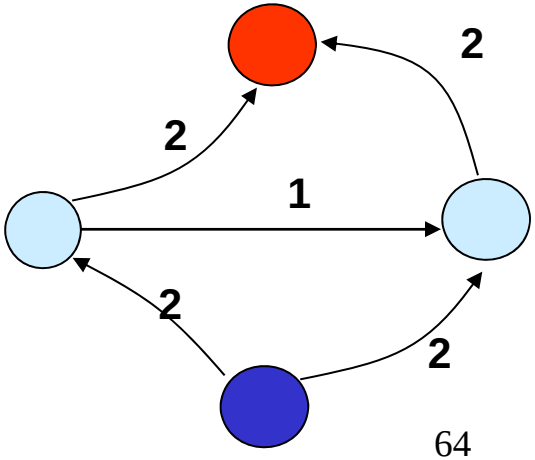
4.2: Finde den maximal möglichen Fluss Δf auf diesem Weg ➔ 1.

4.3: Korrigiere die Kantenwerte im Erweiterungsnetzwerk um Δf.

5: Abbruch: Kein Erweiterungsweg mehr.
Maximaler Fluss erreicht.

Erweiterungsnetz





Maximaler Fluss

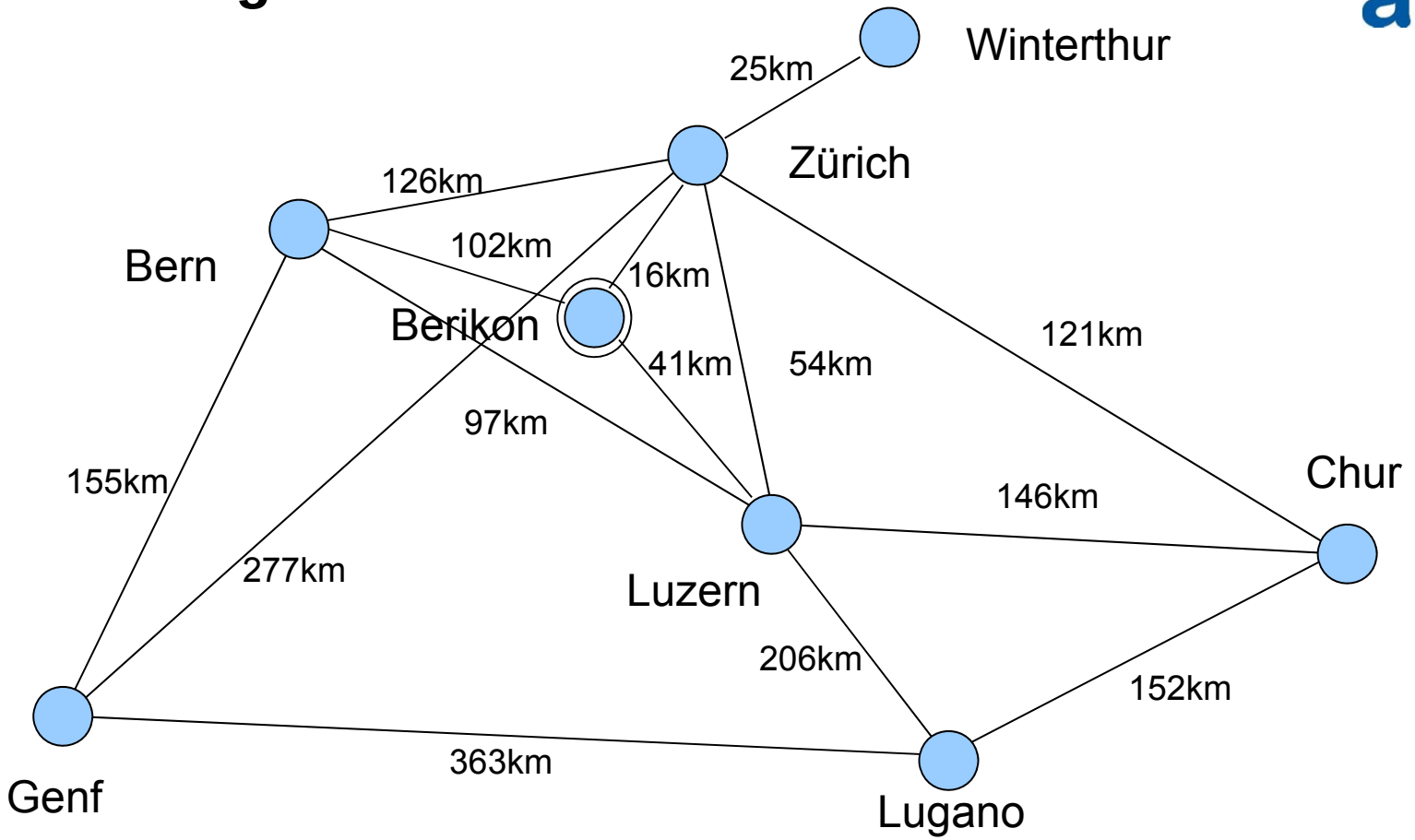
Pseudocode

Start with null flow
$f(u,v) = 0 \ \forall (u,v) \in G$
Initialize residual network
$N_f = N$

```
do {
    find path p in N_f from s to t
    if (found) {
        Δf = min (f(u,v),  (u,v)∈p)
        for each (u,v)∈p {
            if (forward(u,v)) f(u,v) = f(u,v) - Δf;
            if (backward(u,v)) f(u,v) = f(u,v) + Δf;
        }
        update N_f
} while (augmenting path exists)
```

Zeitkomplexität:
Verwende Breadth-first Strategie um den Pfad zu finden.
$O(n^5)$

Traveling Salesman Problem: TSP



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

Eigenschaften des TSP

Es ist einfach eine Lösung im Beispiel zu finden, aber:

(a) Oft gibt es **überhaupt keine** Lösung.

Beispiel: Wenn Winterthur besucht werden soll.

(b) Oft (wie auch hier) gibt es mehrere Zyklen, die alle zu besuchenden Knoten enthalten. Dann kann der **kürzeste Zyklus** nur durch Ausprobieren gefunden werden.

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 $O(2^N)$, wobei N die Anzahl der Kanten ist. → Exponentieller 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

Zurich University of Applied Sciences

zh aw School of Engineering

Zusammenfassung

- Graphen
  - gerichtete, ungerichtete
  - zyklische, azyklische
  - gewichtete, ungewichtete
- Implementationen von Graphen
  - Adjazenz-Liste, Adjazenz-Matrix
- Decorator Pattern

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