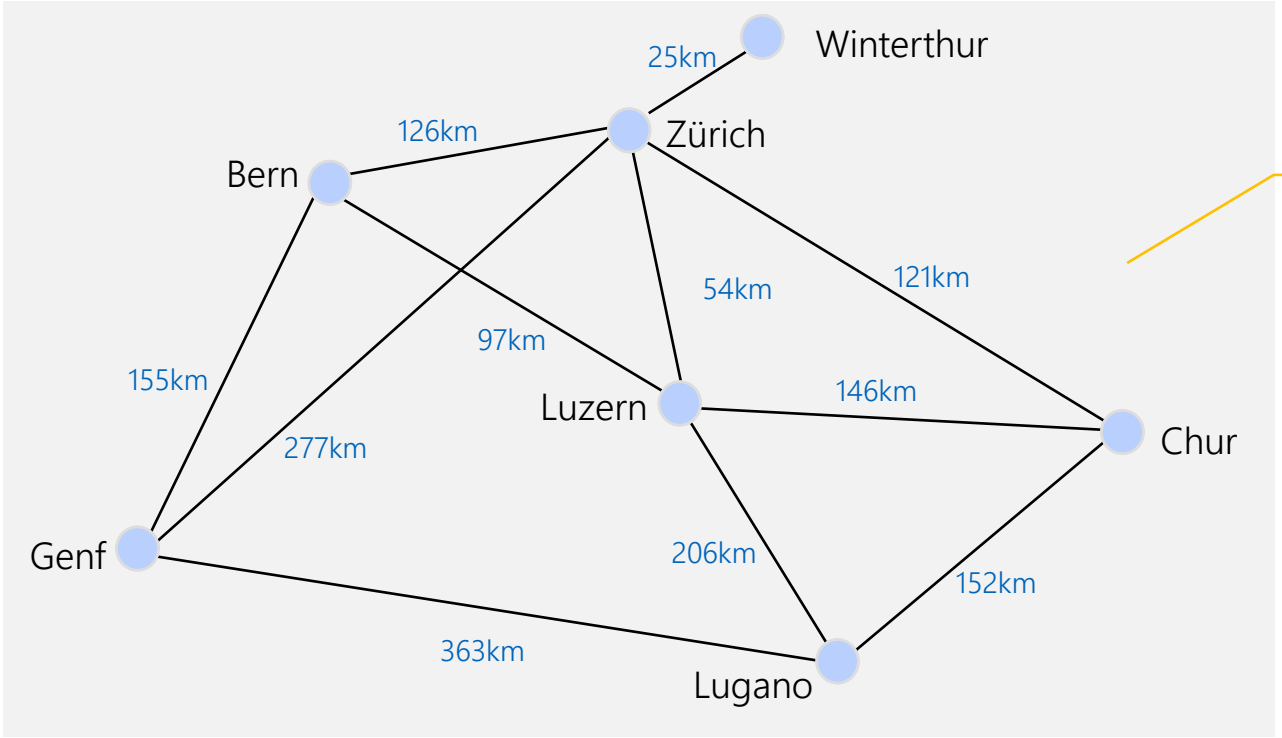
Graphen

- 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

Basiert auf Material von:

Kurt Bleisch
Stephan Neuhaus
Karl Rege
Marcela Ruiz
Jürgen Spielberger

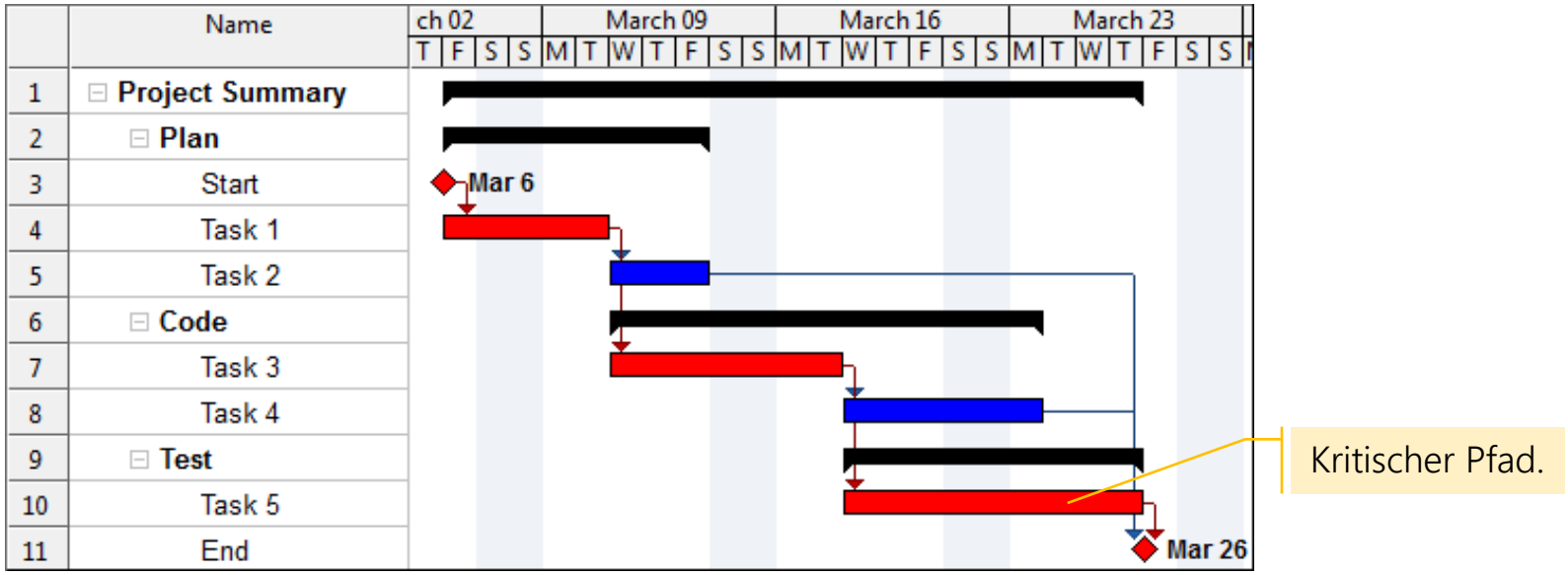
Beispiele: Typische Fragestellungen



Was ist die **kürzeste Verbindung** zwischen Winterthur und Lugano?

1. **Kürzeste Verbindung** (Shortest Path): Verkehr, Postversand von A nach B.
2. **Maximaler Durchsatz** (Maximal Flow): Verkehr, Daten von A nach B.
3. **Kürzester Weg** (Traveling Salesman): um alle Punkte anzufahren.

Beispiele: Typische Fragestellungen



4. Reihenfolge (Topological Sort) von Tätigkeiten aus einem Netzplan erstellen.
5. Minimal benötigte Zeit / Kritischer Pfad (Critical Path): bei optimaler Reihenfolge.

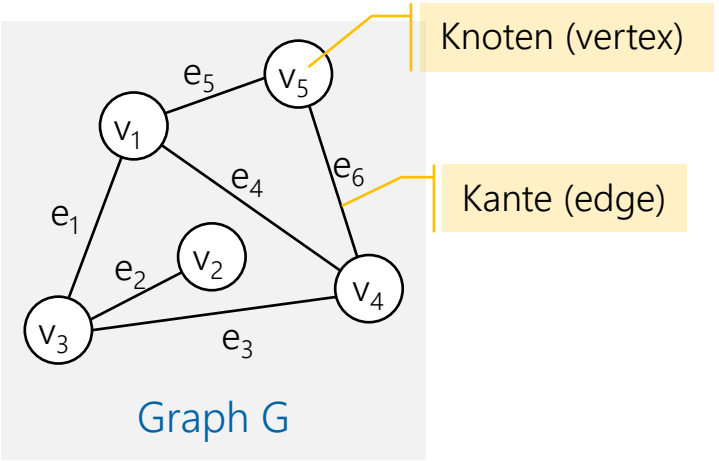
Definition

Graphentheorie

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

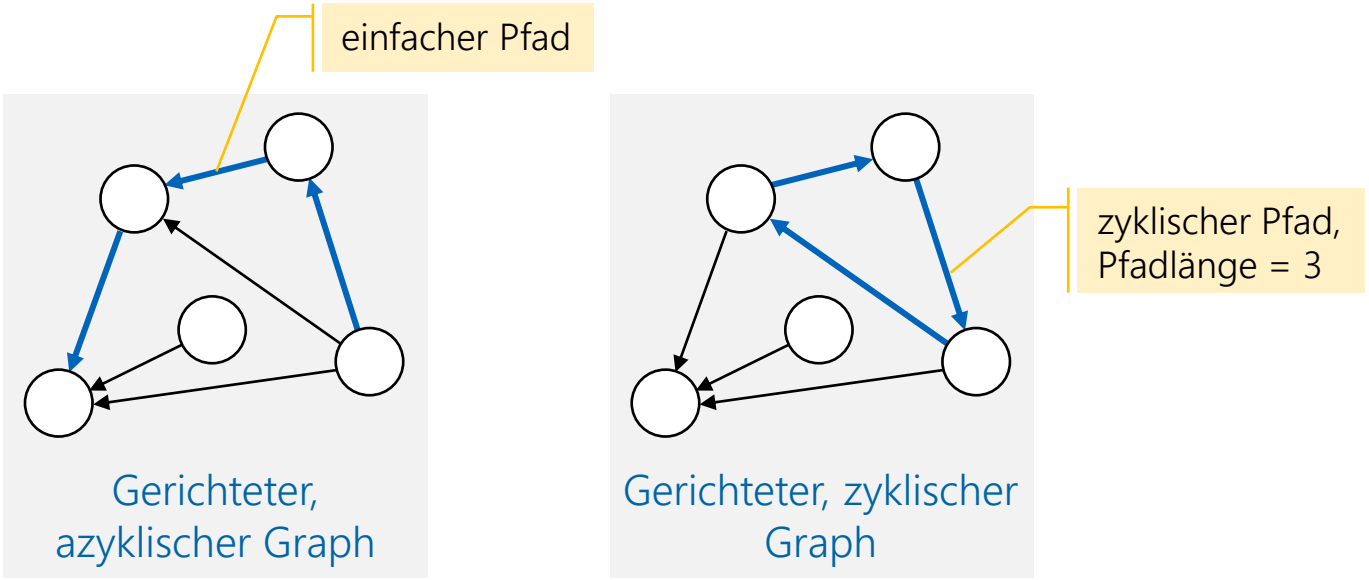
- Implementation:
  - Knoten: Objekte mit Namen und anderen Attributen.
  - Kanten: Gerichtete Verbindungen zwischen zwei Knoten, allenfalls mit Attributen. Es können auch mehrere Kanten zwischen zwei Knoten bestehen und Kanten können einen Knoten mit sich selbst verbinden.

- Hinweise:
  - Knoten, auf Deutsch auch Vertex (Plural: Vertices), heissen auf Englisch vertex/vertices, daher V.
  - Kanten heissen auf Englisch edge/edges, daher E.
  - In der Informatik werden Knoten häufig auch als Node bezeichnet.



Eigenschaften: Pfade, Zyklen

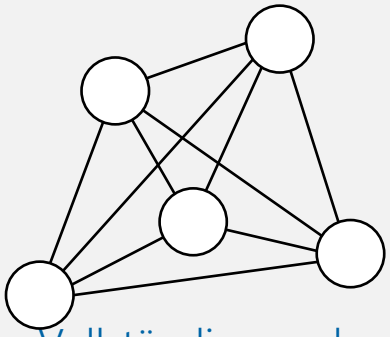
- Eine Sequenz von benachbarten Knoten ist ein **einfacher Pfad**, falls kein Knoten zweimal vorkommt z.B. p = (Zürich, Luzern, Lugano).
- Sind Anfangs- und Endknoten bei einem Pfad gleich, dann ist dies ein **zyklischer Pfad** oder **geschlossener Pfad** bzw. **Zyklus**.
- Die **Pfadlänge** ist die Anzahl der Kanten des Pfads.



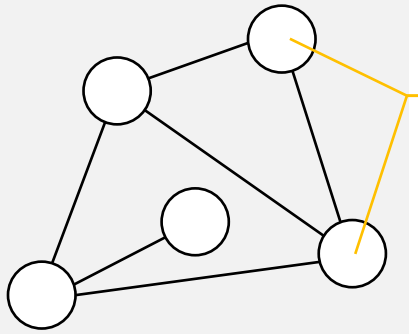
Eigenschaften: vollständig, verbunden

Den Ausdruck Adjazenz werden wir mehrfach wieder sehen.

- **Benachbarte Knoten**: Zwei Knoten x und y sind benachbart (adjacent), falls es eine Kante e_{xy}= (x, y) gibt.
- **Vollständiger (od. kompletter) Graph** (complete graph): Jeder Knoten ist mit jedem anderen Knoten direkt verbunden.
- **Verbundener Graph** (connected graph): Jeder Knoten ist mit jedem anderen Knoten verbunden = es existiert ein Pfad.



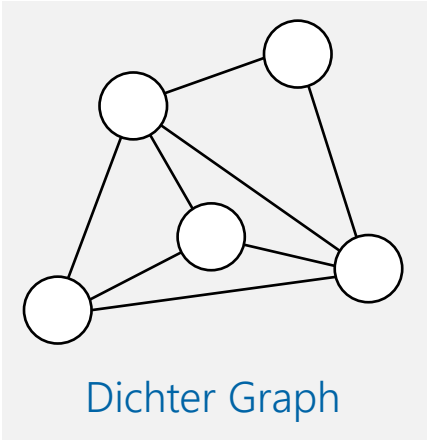
Vollständiger, od. kompletter Graph



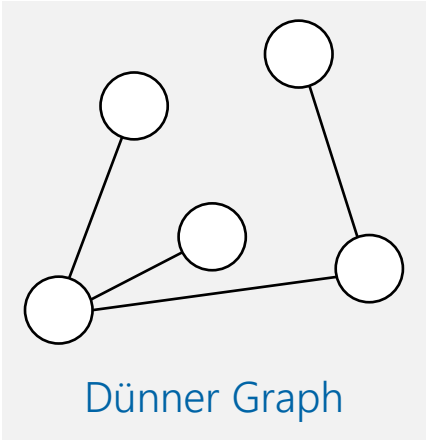
Verbundener Graph

Eigenschaften: dicht, dünn

- **Dichte des Graphen**: Das Verhältnis der Anzahl Kanten zu Anzahl möglicher Kanten (zwischen 0 und 1).
- **Dichter Graph** (dense graph): Nur wenige Kanten im Graph (bezogen auf den kompletten Graphen) fehlen.
- **Dünner oder lichter Graph** (sparse graph): Nur wenige Kanten im Graph (bezogen auf den vollständigen/kompletten Graphen) sind vorhanden.



Dichter Graph

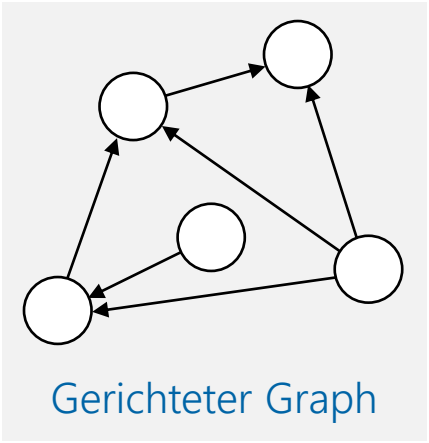


Dünner Graph

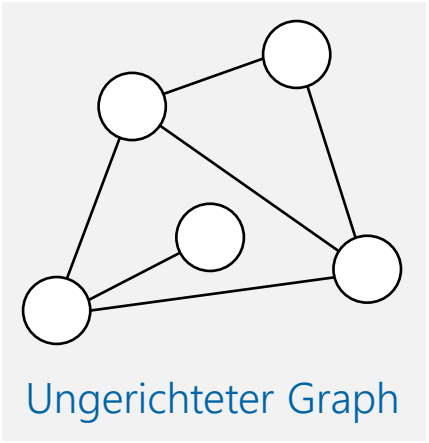
Eigenschaften: ungerichtet / gerichtet

Sei G = (V, E). Falls für jedes e ∈ E mit e = (v1, v2) gilt: e' = (v2, v1) ∈ E,
so heisst G ungerichteter Graph (undirected), ansonsten gerichteter (directed) Graph.

- 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.



Gerichteter Graph



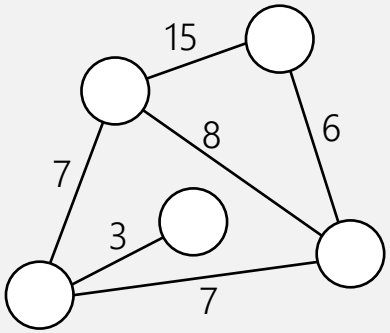
Ungerichteter Graph

Eigenschaften: gewichtet / Netzwerk

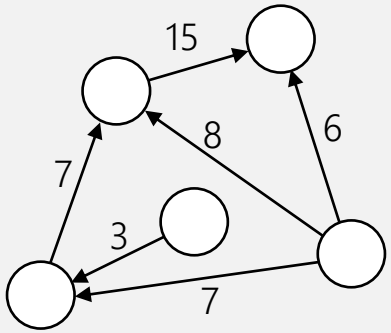
Ein Graph G = (V, E) kann zu einem gewichteten Graphen G = (V, E, g_w(E)) erweitert werden, wenn man eine Gewichtsfunktion g_w: E → $\mathbb{R}$ hinzunimmt, die jeder Kante e ∈ E ein Gewicht g_w(e) zuordnet.

Eigenschaften

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



Gewichteter Graph

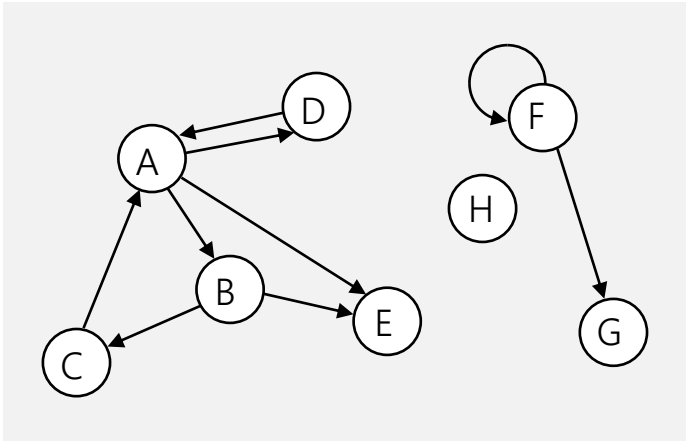


Netzwerk

Graph: Übung

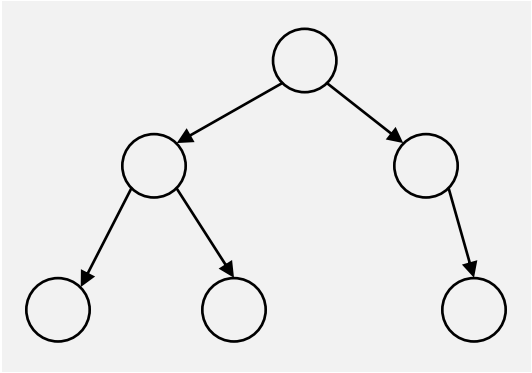
Pfad oder Zyklus?

1. (B, C, A, D, A) ist ein _____ von B nach A.
2. Er enthält einen _____: (A, D, A).
3. (C, A, B, E) ist einfacher _____ von C nach E.
4. (F, F, F, G) ist ein _____.
5. (A, B, C, A) und (D, A, B, C, A, D) und (A, D, A) und (F, F) sind die einzigen _____.
6. (A, B, E, A) ist kein _____ und kein _____.

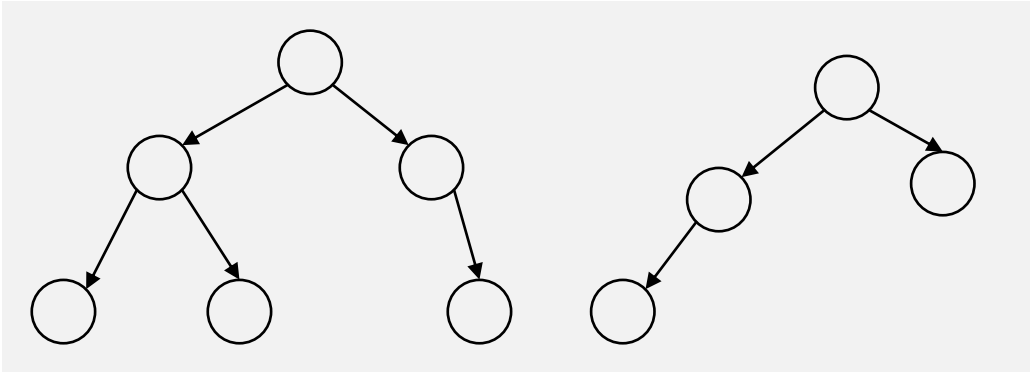


Baum: Spezialfall von Graph

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

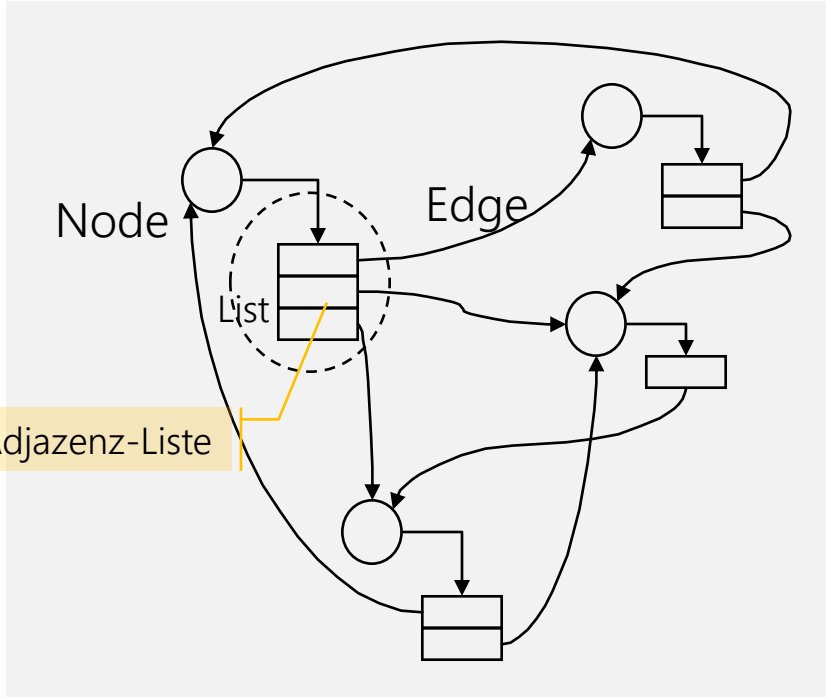


- Eine Gruppe nicht zusammenhängender Bäume heisst Wald (Forest).



Graph: Implementation 1 als Adjazenz-Liste

Jeder Knoten führt eine Adjazenz-Liste, welche alle Kanten zu den benachbarten Knoten enthält.



Dabei wird für jede Kante (Edge) ein Eintrag bestehend aus dem Zielknoten und weiteren Attributen (z.B. Gewicht) erstellt.

Adjazenz-Liste: Interface

Interface!

```
public interface Graph<N, E> {
   // füge Knoten mit Namen name hinzu, tue nichts, falls Knoten schon existiert
   public N addNode (String name);

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

   // Alle Knoten des Graphen mit «möglichem» Iterator
   public Iterable<N> getNodes();

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

Generisches Interface, welches Knoten vom Typ N und Kanten vom Typ E verwaltet.

Iterable, nicht Iterator, ermöglicht «for-each-loop».

Adjazenz-Liste: GraphNode definiert Knoten

```
public class GraphNode<E> {
   protected String name;     // Name des Knoten
   protected List<E> edges;   // Kanten (allenfalls mit Attributen)

   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); }
}
```

Generische Klasse des Knotens, welche Kanten vom Typ E verwaltet.

Iterable, nicht Iterator, ermöglicht «for-each-loop».

Adjazenz-Liste: Edge definiert Kante

Zürcher Hochschule für Angewandte Wissenschaften
School of Engineering

Generische Klasse der Kante, welche Knoten vom Typ N verwendet.

```
public class Edge<N> {
   protected N dest;  // Zielknoten der Kante
   protected double weight;  // Kantengewicht

   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; }
}
```

Adjazenz-Liste: AdjListGraph Implementation

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

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

Interface

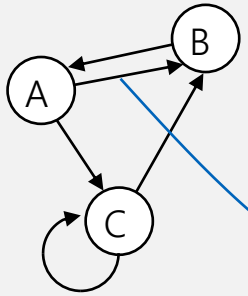
Klassenobjekte der generischen Typen.

Erzeuge Instanz.

newInstance wird hier gebraucht, weil der Typ der Klasse erst zur Laufzeit zugewiesen wird. Somit kann nicht mit new gearbeitet werden.

Das Beispiel zeigt nicht die vollständige Implementation des Interfaces.

Graph: Implementation 2 als Adjazenz-Matrix



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

N x N (N = #Knoten), boolean Matrix.
Dort true, wo Verbindung existiert.

- Falls gewichtete Kanten → Gewichte (z.B. 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 (d.h. speichereffizient), falls es sich um einen dichten Graph handelt.

Adjazenz-Matrix: Beispiel

- Distanzentabelle ist eine Adjazenz-Matrix
- ungerichtet → symmetrisch zur Diagonalen
  - im Prinzip reicht die eine Hälfte → Dreiecksmatrix
- Falls gerichtet → keine Symmetrie, jeder Eintrag ist notwendig

	CALAIS	BRUSSELS	AMESTERDAM	KOLN	HEIDELBERG	STRSBOURG	MANNHEIM	RHINE FALLS	ENGELBERG	LUZERN	INTERLAKEN	GENEVA	BERN	ZURICH	VADUZ	INNSBRUCK	VERONA	VENICE	ROME	FLORENCE	PISA	PARIS
CALAIS		200	370	419	652	621	631	800	887	853	905	821	854	842	950	1141	1245	1360	1682	1412	1358	296
BRUSSELS	200		212	220	563	429	433	708	695	661	714	786	663	650	758	947	1054	1169	1490	1220	1192	311
AMESTERDAM	370	212		267	506	608	499	772	871	838	890	992	839	827	887	994	1230	1345	1667	1397	1368	508
KOLN	419	220	267		254	354	244	520	619	586	638	740	587	575	635	742	978	1093	1414	1144	1116	499
HEIDELBERG	652	563	6	254		139	19	272	371	337	389	491	338	326	386	510	729	844	1166	896	868	542
STRSBOURG	621	429	608	354	139		137	181	268	235	287	389	236	223	331	542	627	742	1064	793	765	487
MANNHEIN	631	433	499	244	19	137		284	382	349	401	503	350	338	398	522	741	856	1178	908	879	522
RHINE FALLS	800	708	772	520	272	181	285		144	110	222	324	171	53	149	287	489	604	926	656	627	702
ENGELBERG	887	695	871	619	371	268	382	144		35	86	298	146	85	165	349	398	513	835	565	536	701
LUZERN	853	661	838	586	337	235	349	110	35		68	265	112	52	131	315	393	508	830	560	531	667
INTERLAKEN	905	714	890	638	389	287	401	222	86	68		215	56	118	198	382	412	527	848	578	550	628
GENEVA	821	786	992	740	491	389	503	324	298	265	215		158	278	386	563	469	584	903	632	544	540
BERN	854	663	839	587	338	236	350	171	146	112	56	158		125	233	410	504	619	941	671	642	572
ZURICH	842	650	827	575	326	223	338	53	85	52	118	278	125		108	291	430	545	867	597	568	655
VADUZ	950	758	887	635	386	331	398	149	165	131	198	386	233	108		169	409	524	845	575	547	763
INNSBRUCK	1141	947	994	742	510	542	522	287	349	315	382	563	410	291	169		274	389	770	500	560	934
VERONA	1245	1054	1230	978	729	627	741	489	398	393	412	469	504	430	409	274		120	517	247	307	1001
VENICE	1360	1169	1345	1093	844	742	856	604	513	508	527	584	619	545	524	389	120		546	276	336	1116
ROME	1682	1490	1667	1414	1166	1064	1178	926	835	830	848	903	941	867	845	770	517	546		291	393	1444
FLORENCE	1412	1220	1397	1144	896	793	908	656	565	560	578	632	671	597	575	500	247	276	291		102	1168
PISA	1358	1192	1368	1116	868	765	879	627	536	531	550	544	642	568	547	560	307	336	393	102		1076
PARIS	296	311	508	499	542	487	522	702	701	667	628	540	572	655	763	934	1001	1116	1444	1168	1076	

Vergleich der Adjazenz-Matrix und -Liste

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

	Vorteile	Nachteile
Adjazenz-Matrix	Feststellen, wer mit wem verbunden ist, ist sehr effizient.	hoher Platzbedarf und teure Initialisierung: wachsen quadratisch mit $O(n^2)$.
Adjazenz-Liste	Es wird kein Speicher «verschwendet», Bedarf 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

Wir betrachten folgende Problemstellungen / 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)
- .....

Traversierungen

Grundformen: «Suchstrategien»

Zürcher Hochschule für Angewandte Wissenschaften
zh aw School of Engineering

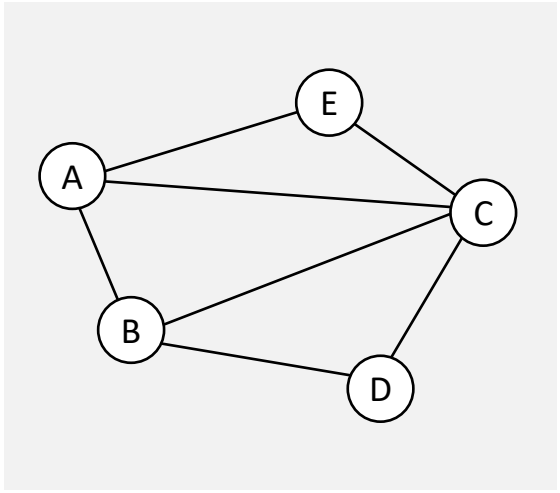
Genau wie bei den Traversierungen bei Bäumen, sollen bei Graphen die Knoten systematisch besucht werden. Es werden im Wesentlichen zwei grundlegende «Suchstrategien» unterschieden:

- **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 schrittweise 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.

Tiefensuche: Übung

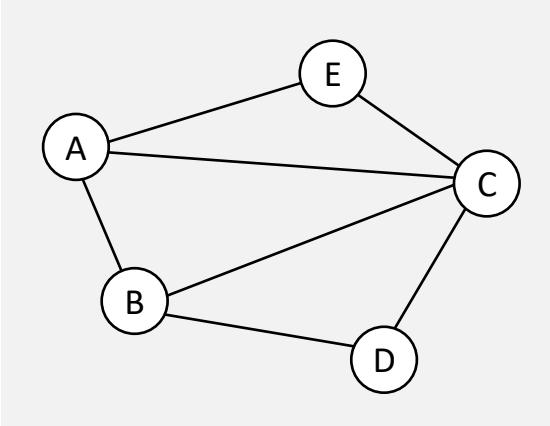
Auf welche Arten kann folgender Graph in Tiefensuche durchsucht werden (Start bei A)?



Tiefensuche: Pseudo Code

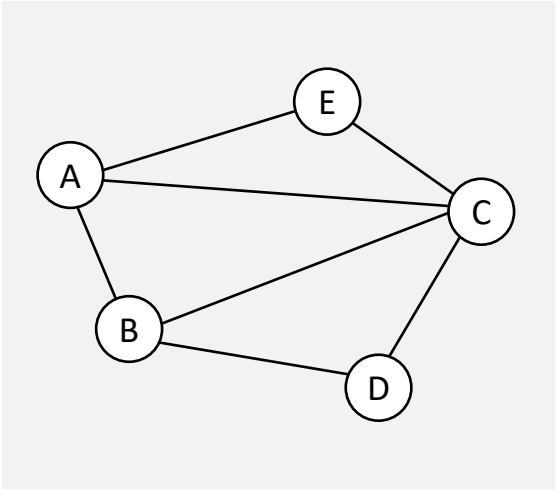
```
void depthFirstSearch()
   s = new Stack();
   mark startNode;
   s.push(startNode)
   while (!s.empty()) {
      currentNode = s.pop()
      print currentNode
      for all nodes n adjacent to currentNode {
         if (!(marked(n)) {
            mark n
            s.push(n)
         }
      }
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

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



Breitensuche: Übung

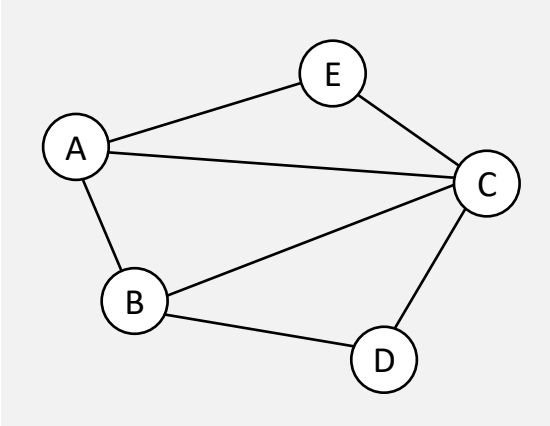
Auf welche Arten kann folgender Graph in Breitensuche durchsucht werden (Start bei A)?



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)
         }
      }
   }
}
```

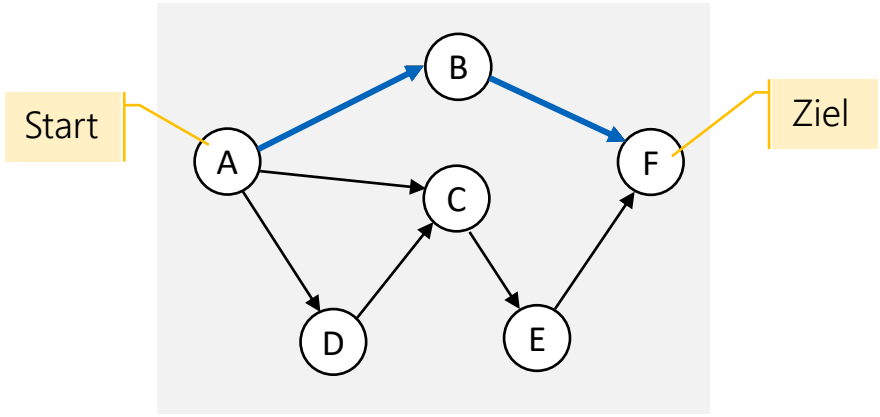
Am Anfang sind alle Knoten nicht markiert. Knoten, die noch nicht besucht wurden, liegen in der Queue.



Kürzester Pfad

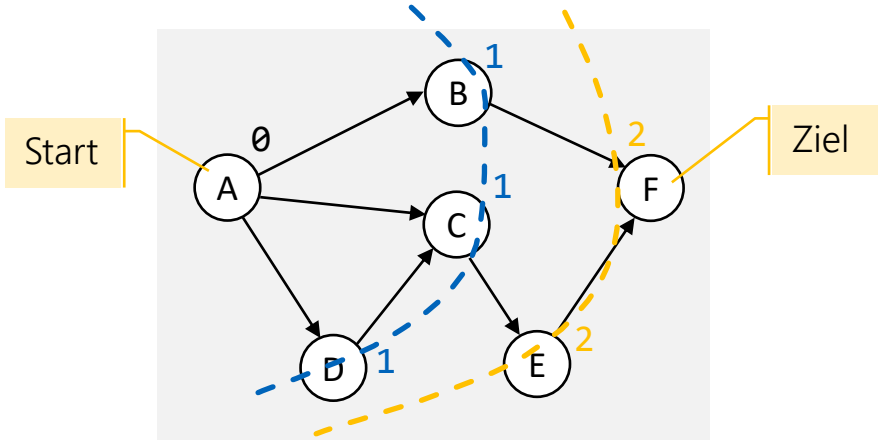
Kürzester, ungewichteter Pfad

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



Kürzester, ungewichteter Pfad: Algorithmus

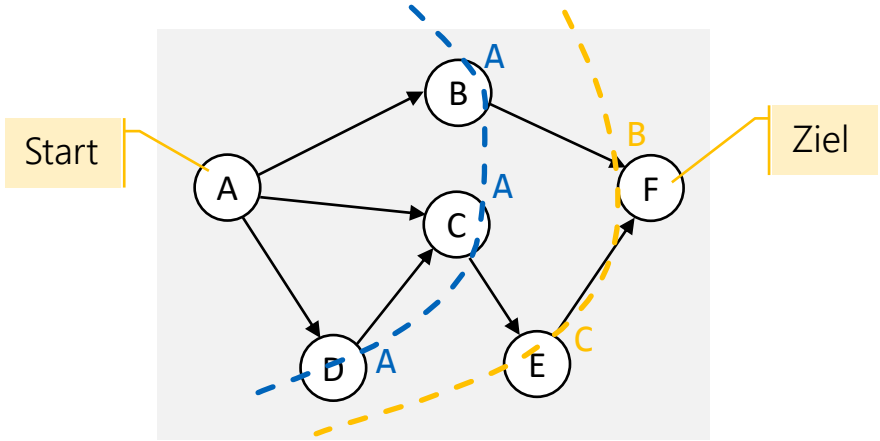
Vom Startpunkt ausgehend werden die Knoten mit ihrer Distanz markiert:



Der Graph wird hier mit Breitensuche traversiert.

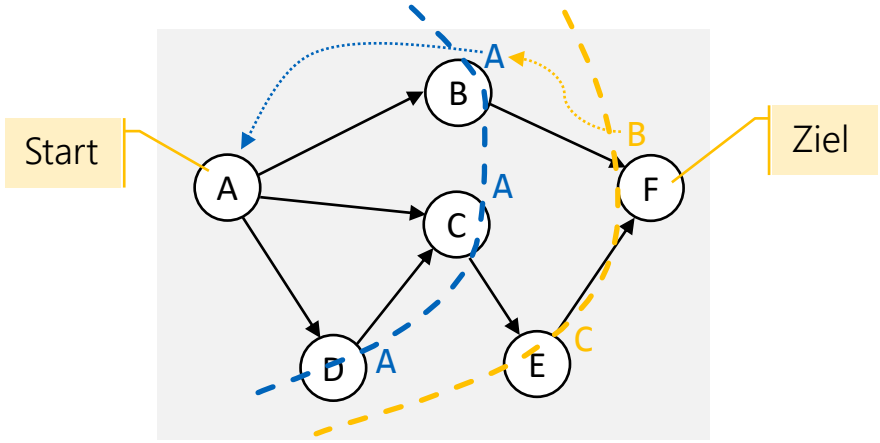
Kürzester, ungewichteter Pfad: Algorithmus

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



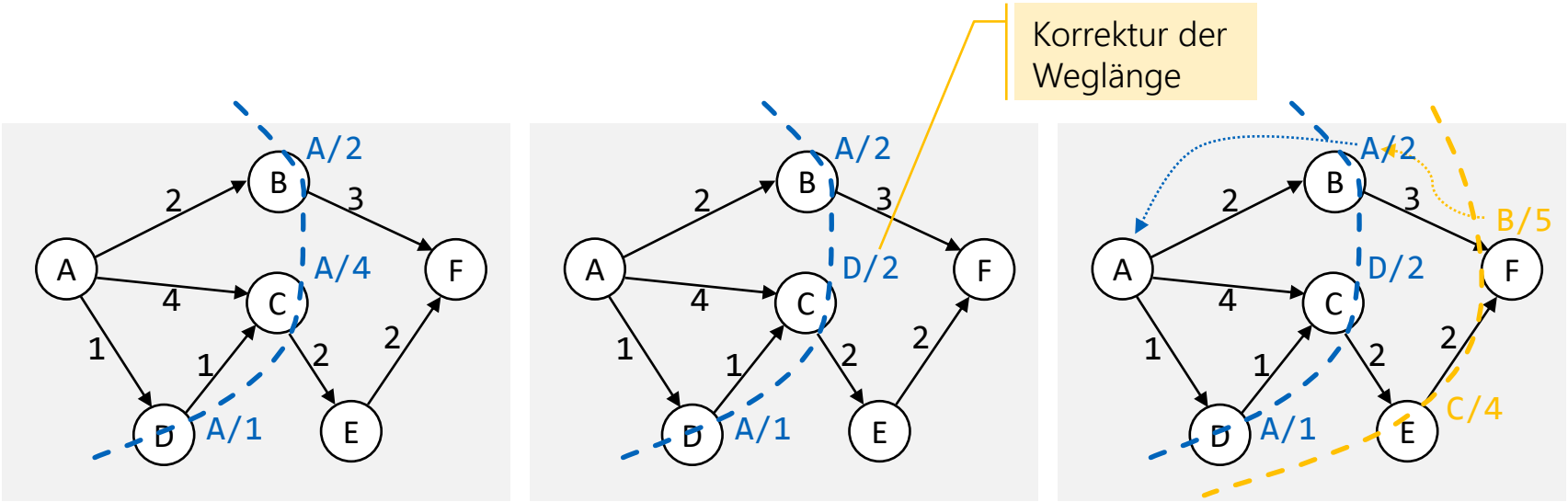
Kürzester, ungewichteter Pfad: Algorithmus

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



Kürzester, gewichteter Pfad: Algorithmus

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

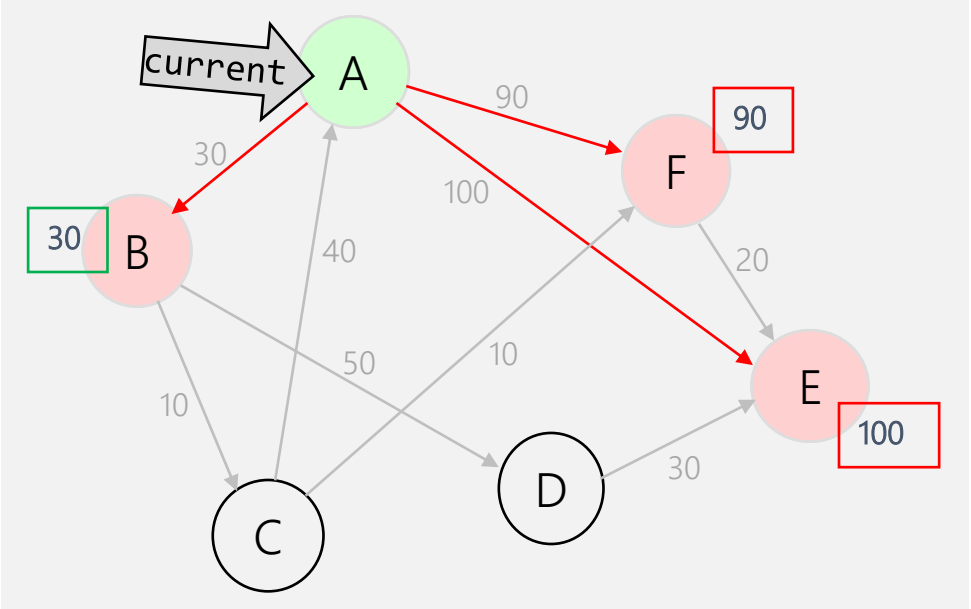


Der Eintrag für C wird auf den neuen Wert gesetzt; statt «markiert» gehe so lange weiter, bis der neue Weg länger als der angetroffene ist.

Geht das performanter? Der bekannteste Algorithmus zur Lösung des Kürzeste-Wege-Problems in Graphen (mit positiven Kantengewichten) ist der Algorithmus von Dijkstra.

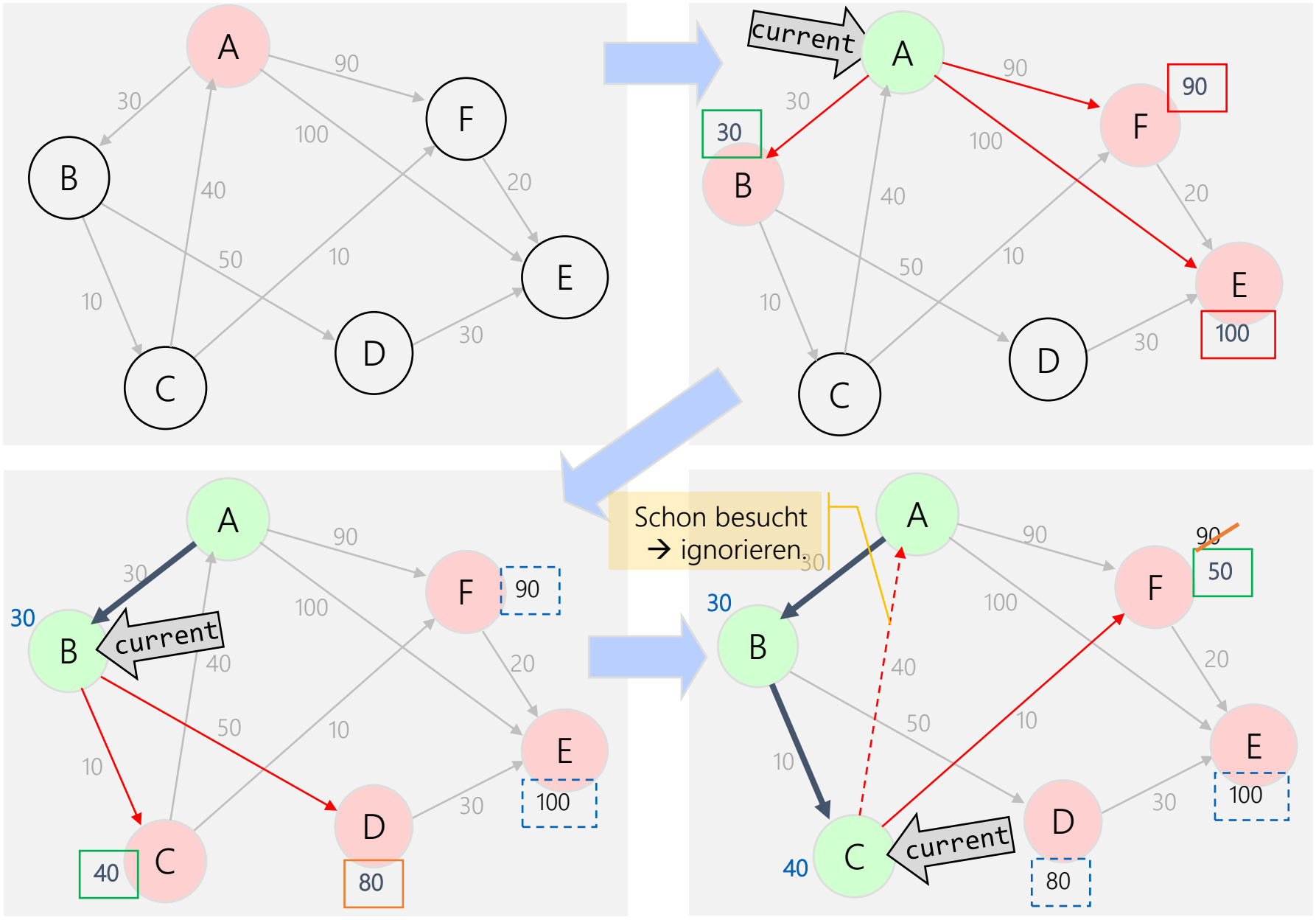
Dijkstras Algorithmus

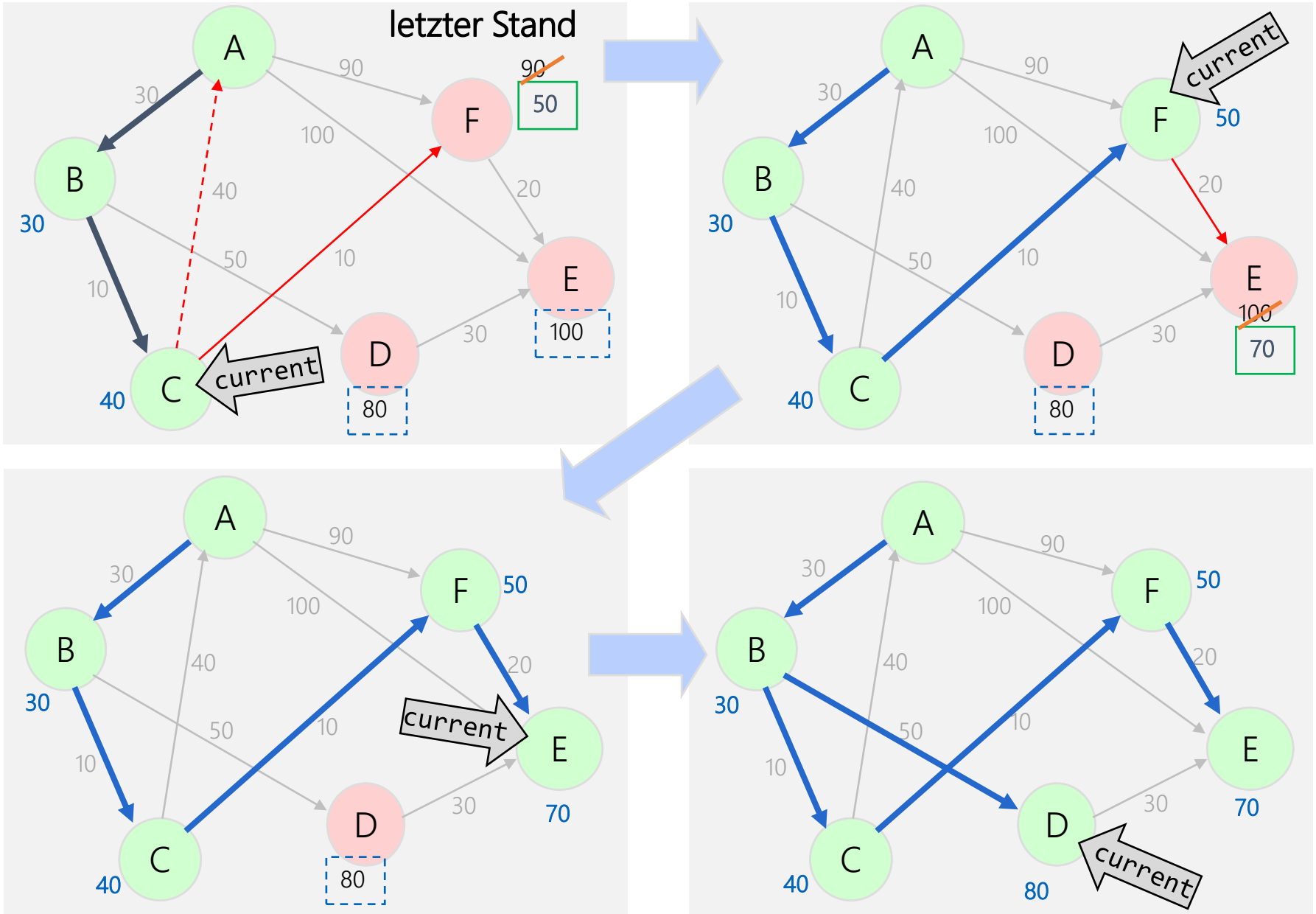
- Teilt die Knoten in 3 Gruppen auf
  - besuchte Knoten (kleinste Distanz bekannt)
  - benachbart zu allen bereits besuchten Knoten
  - unbesehene Knoten (der Rest)



- Solange nicht alle Knoten besucht wurden (grün sind):
  1. Berechne für alle benachbarten, unbesuchten Knoten des aktuell besuchten Knotens (current) die neuen Gewichte (rote Pfeile).
  2. Suche unter **allen** benachbarten (nicht nur jene des aktuellen Knotens), unbesuchten Knoten denjenigen, dessen Pfad zum Startknoten das **kleinste Gewicht** (= kürzeste Distanz) hat (grüner Rahmen um Zahl).
  3. Besuche diesen (neuer aktueller Knoten und kürzester Pfad zu diesem Knoten bekannt).

  Soll nur der minimale Pfad für einen bestimmten Knoten gesucht werden, kann die Suche abgebrochen werden, sobald dieser «current» wird.


A
B
C
D
E
F
30
90
100
40
20
50
10
10
30
current
30
90
100
40
20
50
10
10
30
90
100
current
30
40
80
Schon besucht
→ ignorieren.
30
90
50
100
40
20
50
10
10
30
40
current
80
100


letzter Stand
A
B
C
D
E
F
30
40
10
80
100
70
50
90
100
40
50
10
30
20
current

Dijkstras Algorithmus: Pseudocode

Initialisierung

```
for all nodes n in G {
   n.mark = black; // Alle Knoten noch unbesehen
   n.dist = inf;   // Distanz zu Beginn unendlich
   n.prev = null;  // Bester Vorgängerknoten in Richtung Start
}
start.dist = 0; start.mark = red;
current = start;
```

Start-Knoten

Z.B. mittels Priority-Queue implementieren

```
for all nodes in RED {
   current = findNodeWithSmallestDist(); // suche den besten roten Knoten
   if (current == goal) return; // kürzester Pfad gefunden -> Abbruch
   current.mark = green;
   for all neighbour in successors(current) {
      if (neighbour.mark != green) { // dieser Knoten wurde nicht besucht
         neighbour.mark = red;
         dist = current.dist + edge(current, neighbour);
         if (dist < neighbour.dist) {
            neighbour.dist = dist;
            neighbour.prev = current;
         }
      }
   }
}
```

Neue unbesuchte Nachbarn suchen

Es wurde ein neuer, kürzerer Pfad gefunden.

Dijkstras Algorithmus: Java & PriorityQueue

```
void breadthFirstSearch()
   redNodes = new PriorityQueue()
   startNode.dist = 0;
   redNodes.add(startNode)
   while (!redNodes.empty()) {
      current = redNodes.remove() // roter Knoten mit aktuell kürzester Distanz
      if (current == goal) return; // kürzester Pfad gefunden -> Abbruch
      current.marked = true; // Knoten wird 'grün' markiert
      for (Edge edge: current.getEdges()) {
         neighbour = edge.getDestination();
         if (!neighbour.marked) { // dieser Knoten wurde nicht besucht
            dist = current.dist + edge.dist;
            if (dist < neighbour.dist) {
               neighbour.dist = dist;
               neighbour.prev = current;
               redNodes.remove(neighbour)
               redNodes.add(neighbour)
            }
         }
      }
   }
}
```

Enthält die Liste der roten Knoten sortiert nach Distanz (Nachbarn).

Fügt den Startknoten in die Queue, sortiert gemäss «compareTo», ein.

Initialisiert mit Integer.MAX_VALUE.

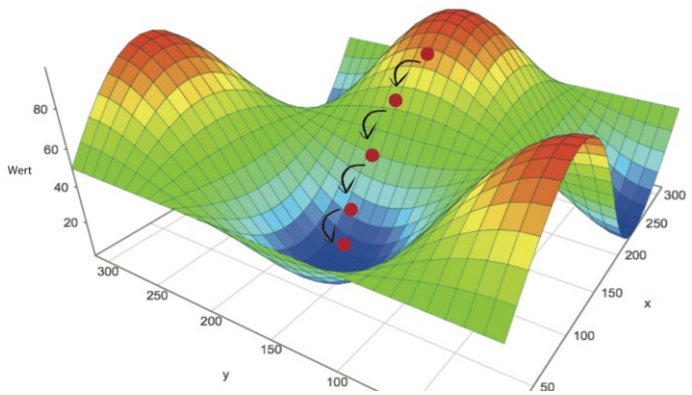
Allfälligen Eintrag mit alter, längerer Distanz in PriorityQueue löschen.

Greedy Algorithmen

Gierige Algorithmen (Greedy)

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

Wir sehen heute noch einen weiteren Greedy-Algorithmus.

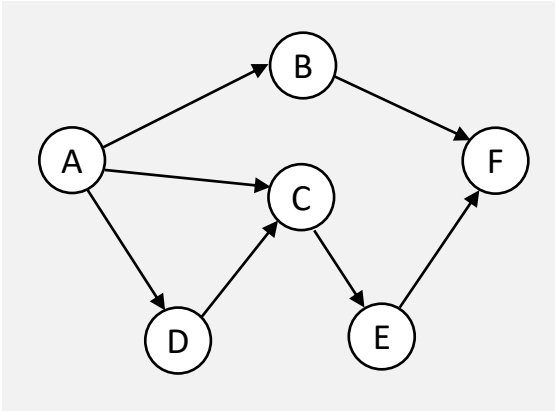


Gradientenverfahren

Topologisches Sortieren

Topologisches Sortieren

Die Knoten eines gerichteten, unzyklischen Graphs in einer «natürlichen» Reihenfolge (Sortierung) auflisten. In anderen Worten: Die Knoten eines gerichteten, azyklischen Graphen so in eine Reihe bringen, dass Pfeile immer nach rechts zeigen.



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

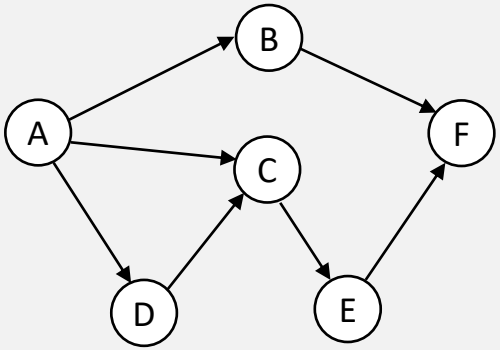
Die Topologisches Sortierung zeigt eine mögliche Compilationsreihenfolge.

Was sind mögliche Compilationsreihenfolgen für den Graphen oben?

Topologische Sortierung: Übung

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

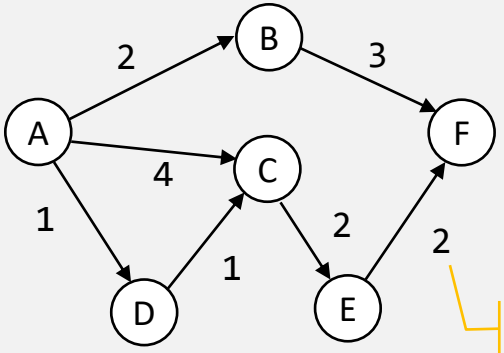
Hinweis: Zähle die eingehenden Kanten; welche können ausgegeben werden?



Maximaler Fluss

Maximaler Fluss

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



Es fliessen max. 2 Liter Wasser pro Stunde von E nach F.

Graf mit maximalem Fluss.

Text zu maximalem Fluss.

Hinweise:

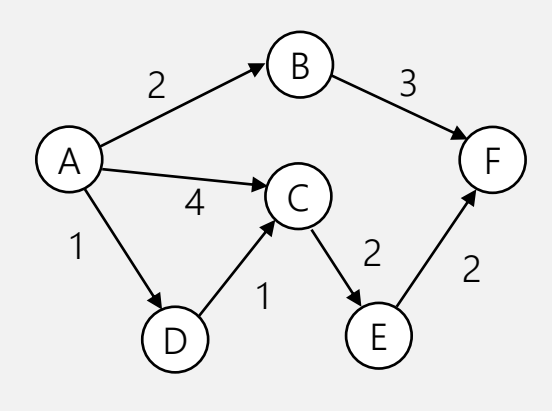
1. Was in einen Knoten hinein fliesst, muss auch wieder heraus.
2. Es sind allenfalls mehrere Lösungen mit demselben Fluss möglich.

Maximaler Fluss: Lösungsidee

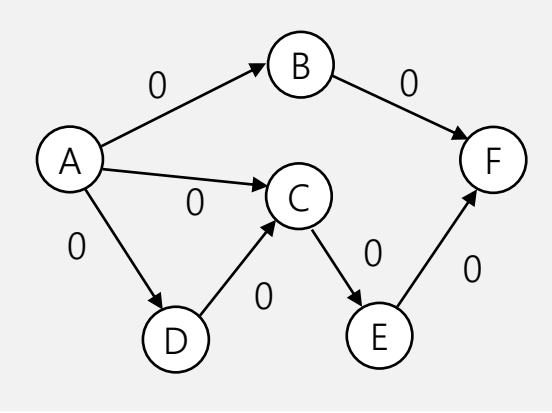
Noch zwei zusätzliche Versionen des Graphen führen so dass gilt:

Original = Vorläufiger Fluss + Rest-Fluss

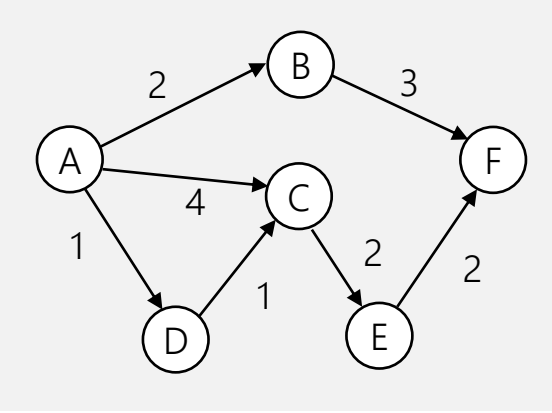
Startkonfiguration:



Original



Vorläufiger Fluss



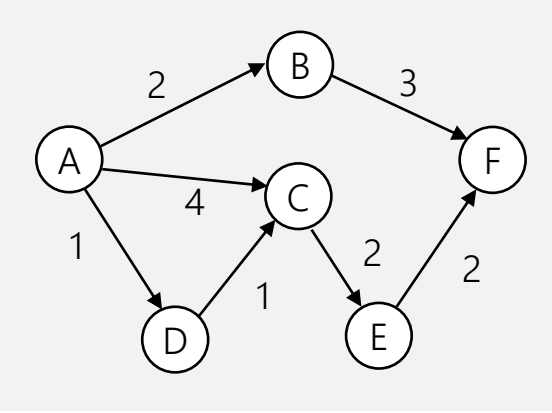
Rest-Fluss oder Residualgraph

Dies ist der Algorithmus von Ford und Fulkerson.

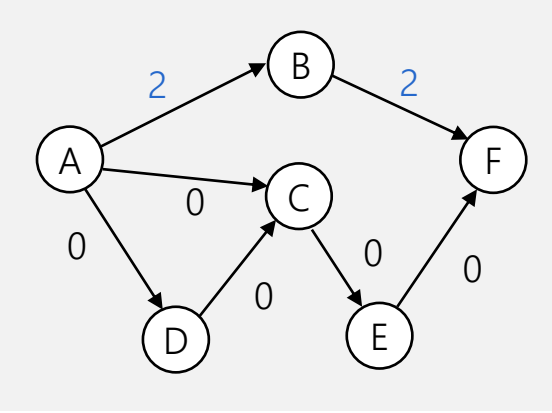
Maximaler Fluss: Lösungsidee

1. Vorläufiger Fluss: Pfad A-B-F mit Fluss 2 einfügen (2 ist hier das Maximum).
2. Rest-Fluss: Pfad A-B löschen (da 2 – 2 = 0), B-F auf 1 (da 3 - 2 = 1) setzen.

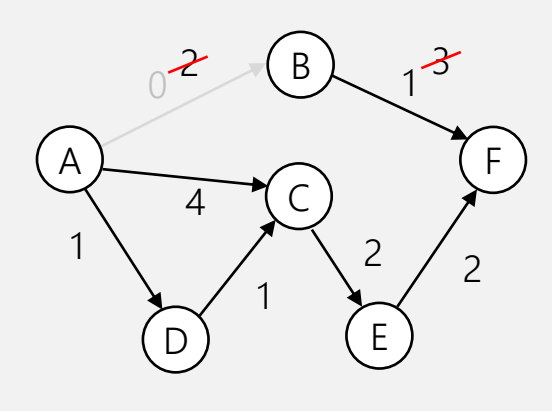
Erste Iteration:



Original



Vorläufiger Fluss

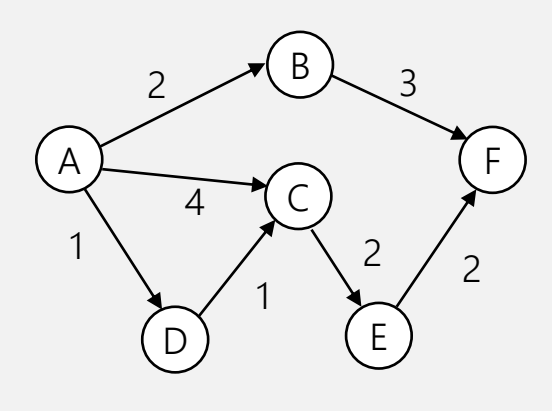


Rest-Fluss oder Residualgraph

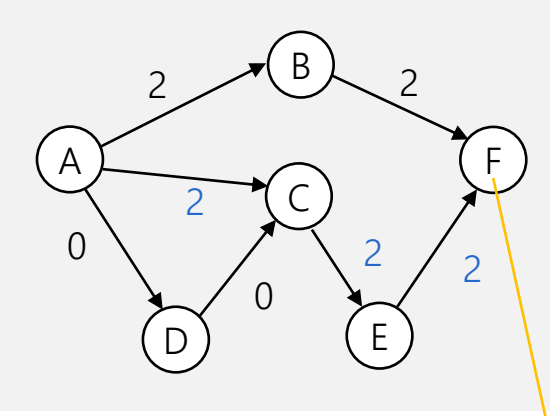
Maximaler Fluss: Lösungsidee

1. Vorläufiger Fluss: Pfad A-C-E-F mit Fluss 2 einfügen.
2. Rest-Fluss: Pfad C-E-F löschen (da Fluss = 0), A-C auf 2 setzen.
3. Kein weiterer Fluss möglich, da A nicht mehr mit F verbunden ist.

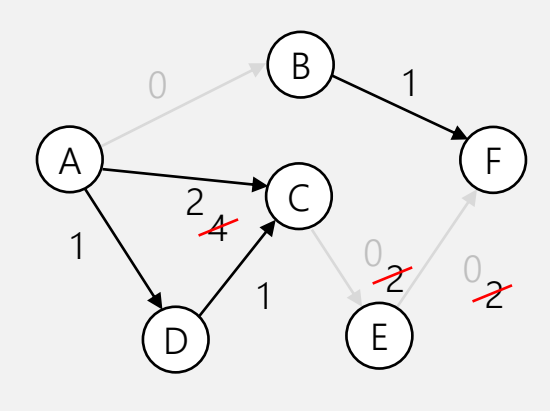
Zweite Iteration:



Original



Vorläufiger Fluss



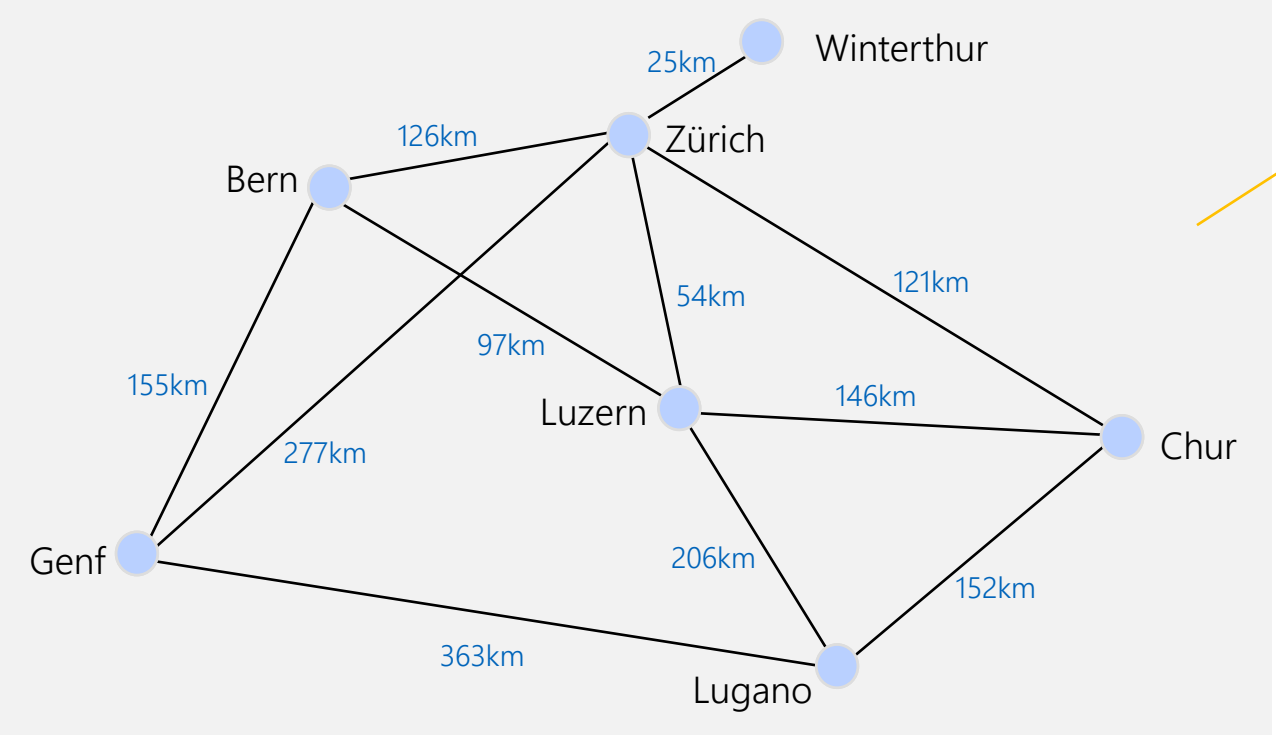
Knoten F erhält 2 über A-B-F plus 2 über A-C-E-F, insgesamt 4.

Der vorläufige Fluss zeigt jetzt eine mögliche Lösung für den maximalen Fluss von 4 an (es gibt noch andere).

Traveling Salesman

Traveling Salesman Problem: TSP

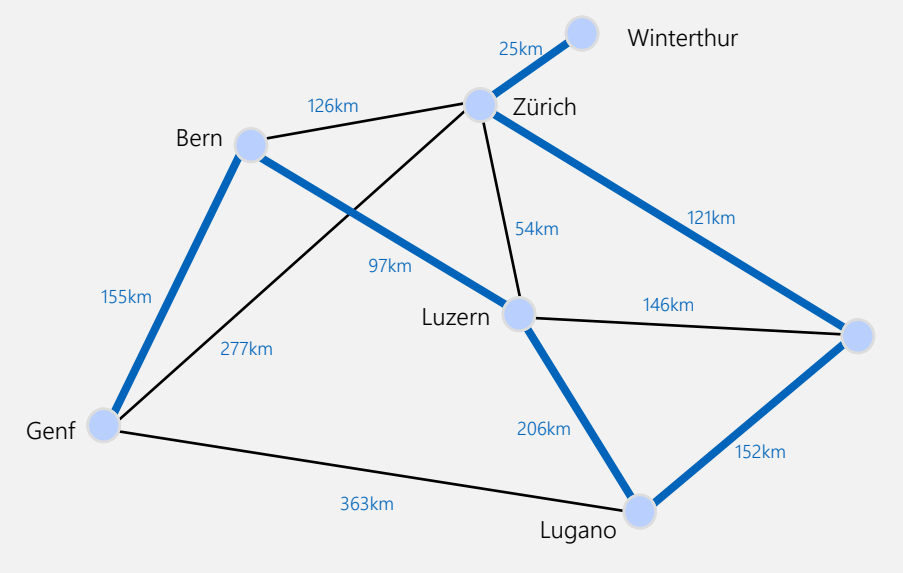
Finden Sie die kürzeste **Reiseroute**, in der jede Stadt genau einmal besucht wird.
Option: am Schluss wieder am Ursprungsort



Typisches Problem von Spediteuren.

Traveling Salesman Problem: Eigenschaften

- Es ist relativ einfach eine Lösung im Beispiel zu finden. Ist das der kürzeste Weg?
- 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!). Der Aufwand wächst faktoriell (pro Memoria: $50! > 3 \cdot 10^{64}$).



Traveling Salesman Problem: Lösungen

Um schnell zu brauchbaren Lösungen zu kommen, sind meist durch Heuristiken motivierte Näherungsverfahren notwendig, die aber in der Regel keine Güteabschätzung für die gefundenen Lösungen liefern.

Je nachdem, ob eine Heuristik eine neue Tour konstruiert oder ob sie versucht, eine bestehende Rundreise zu verbessern, wird sie als Eröffnungs- oder Verbesserungsverfahren bezeichnet.

Die Kunst, mit begrenztem Wissen und wenig Zeit dennoch zu praktikablen Lösungen zu kommen.

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

```
Erzeuge alle möglichen Routen;
Berechne die Kosten (Weglänge) für jede Route;
Wähle die kostengünstigste Route aus.
```

Traveling Salesman Problem: Algorithmus

Der folgende Greedy-Algorithmus führt zu einer Näherungslösung:

1. Die Kanten werden nach ihren Kosten sortiert → Liste.
2. Wähle billigste Kante unter folgenden Bedingungen (ungültige Kanten ebenfalls aus Liste entfernen):
   - Es darf kein Zyklus entstehen (eventuell am Ende erlauben, wenn Rundreise möglich)
   - Kein Knoten darf mit mehr als zwei Kanten verbunden sein

- Laufzeit liegt bei $O(n^2 \log n^2)$ – wir verzichten auf die Herleitung.
- Das Verfahren führt nicht immer zu einer optimalen Lösung. Trotzdem wird es in der Praxis erfolgreich eingesetzt.

Zusammenfassung

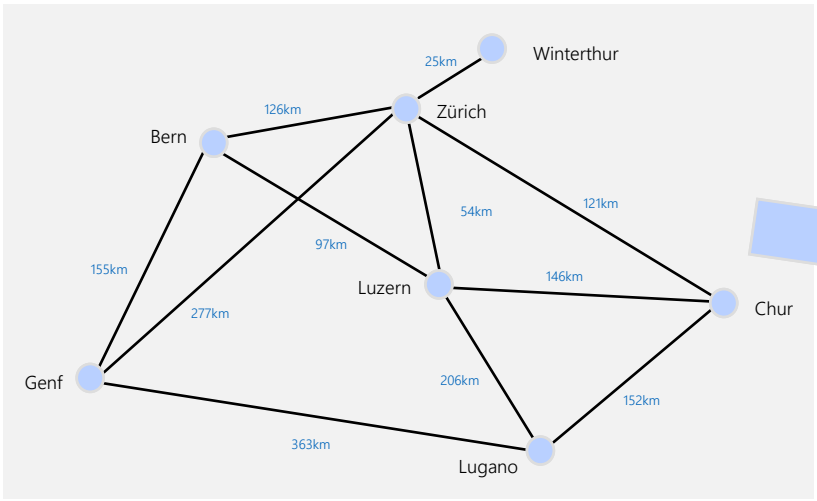
- Graphen
  - gerichtet, ungerichtet
  - zyklisch, azyklisch
  - gewichtet, ungewichtet
- Implementationen von Graphen
  - Adjazenz-Liste, Adjazenz-Matrix
- Algorithmen
  - Grundformen: Tiefensuche/ Breitensuche
  - Kürzester Pfad (ungewichtet/gewichtet)
  - Greedy-Algorithmen
  - Topologisches Sortieren
  - Maximaler Fluss
  - Traveling Salesman Problem

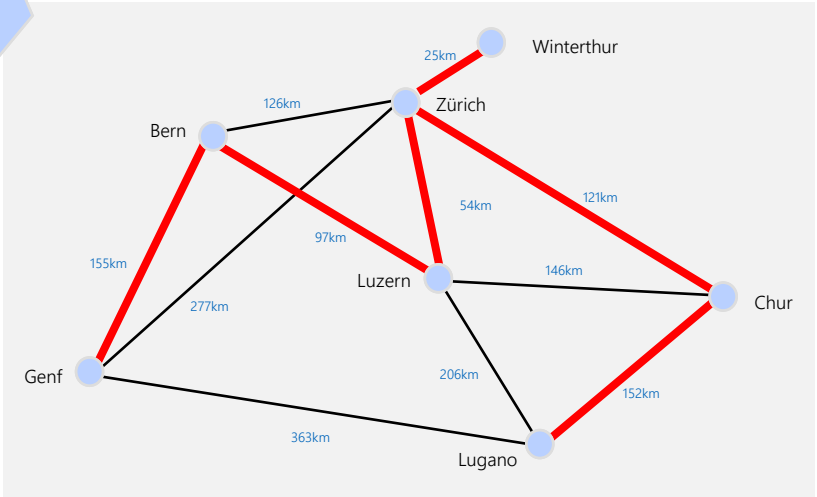
Nerd-Zone

Spannbaum

Minimum Spanning Tree: Beispiel

Wie kann ich alle Orte mit Strom versorgen, so dass die Kabellänge minimal ist?





Minimum Spanning Tree: Definition, Algorithmus

Definitionen:

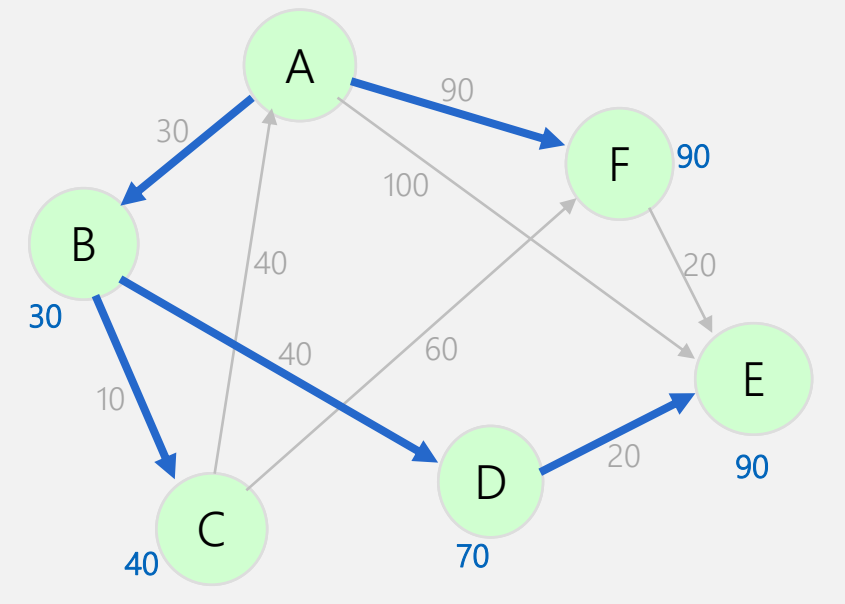
- Ein Spannbaum eines Graphen ist ein Baum, der alle Knoten des Graphen enthält und dessen Kanten im Graphen vorhanden waren.
- Ein minimaler Spannbaum (minimum spanning tree) ist ein Spannbaum eines gewichteten Graphen, sodass die Summe aller Kanten minimal ist.

Algorithmus:

- z.B. Prim-Jarnik, Prim-Jarnik ist ähnlich wie Dijkstras Algorithmus:
  - Wähle einen beliebigen Knoten v als Startgraph G.
  - Solange G noch nicht alle Knoten enthält:
    - Wähle eine Kante e mit minimalem Gewicht aus, die einen noch nicht in G enthaltenen Knoten v mit G verbindet.
    - Füge e und v dem Graphen G hinzu.

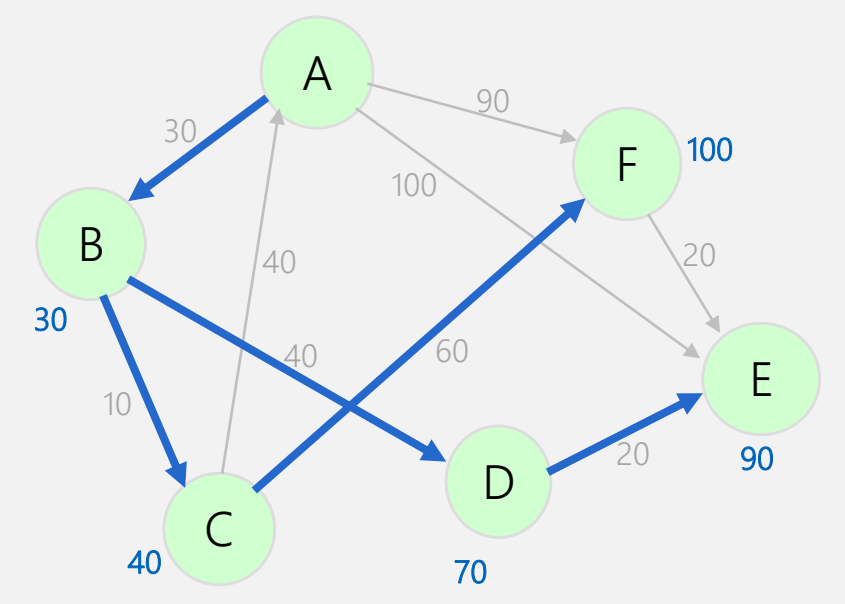
Shortest Path vs. Minimum Spanning Tree

Dijkstra: Shortest path



Total = 190
Weglänge A-F = 90

Prim-Jarnik: Min. Spanning Tree



Total = 160
Weglänge A-F = 100