Institut für Technik der Informationsverarbeitung

Institutsleitung
Prof. Dr.-Ing. J. Becker
Prof. Dr.-Ing. E. Sax
Prof. Dr. rer. nat. W. Stork

4. Suchalgorithmen

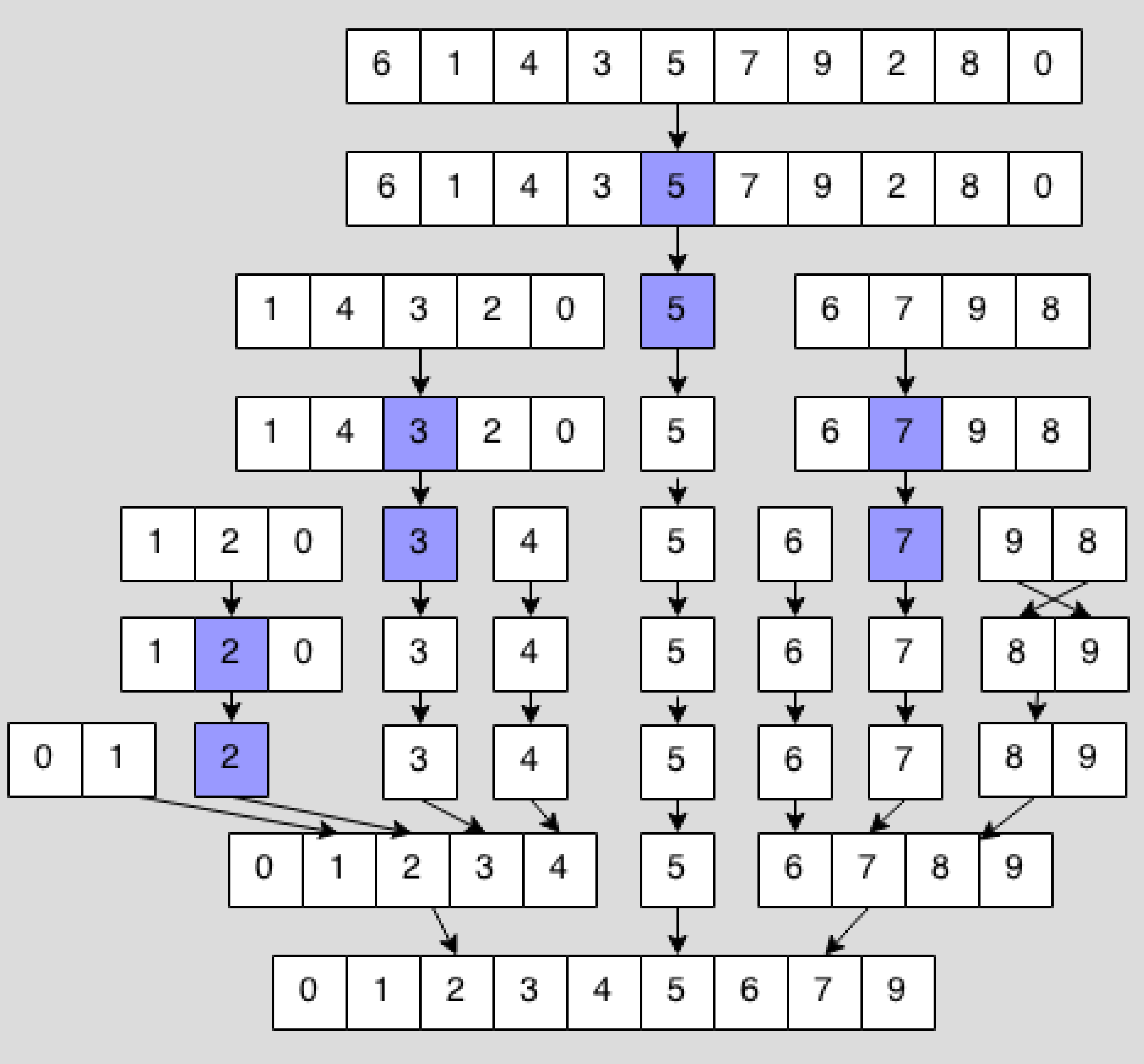
Informationstechnik II

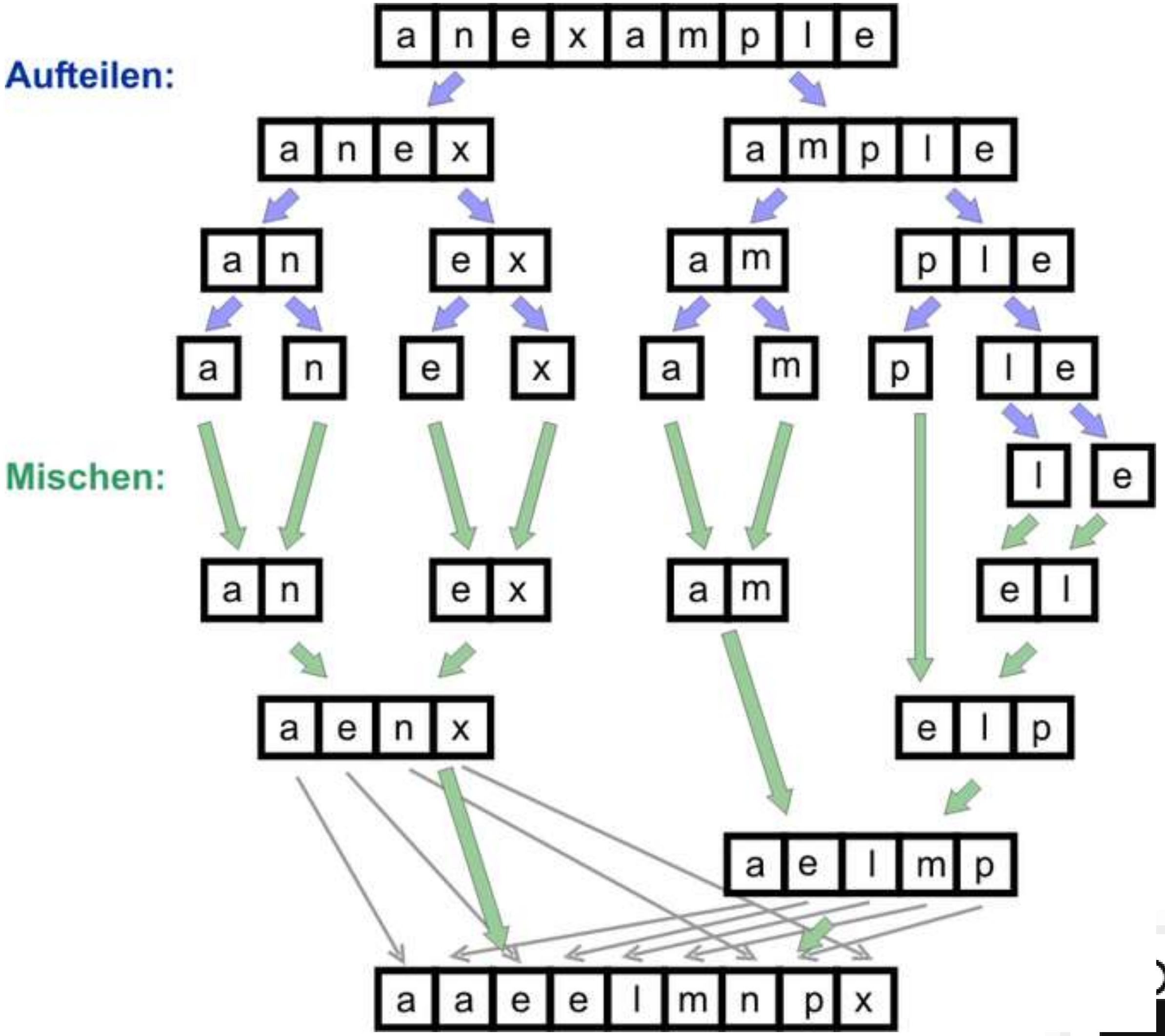
Wir zeichnen auf

Prof. Dr.-Ing. Eric Sax

Quick Sort - Laufzeitanalyse

vs. Mergesort





Institut für Technik der Informationsverarbeitung (ITIV)
Prof. Dr.-Ing. Eric Sax, © 2020

Inhalt IT2

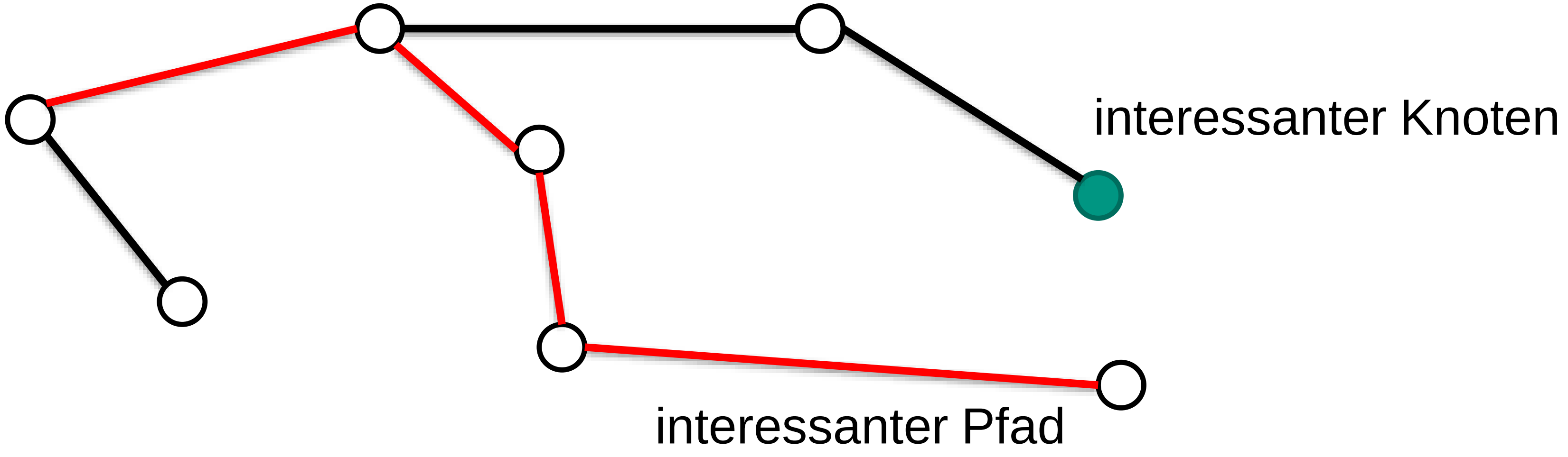
4. Algorithmen auf Graphen

- Graphendefinition
- Binäre Suche
- Breitensuche
- Tiefensuche
- Zyklensuche
- Kürzester und kritischer Pfad

5. Optimierungsalgorithmen

- Partitionierung
- Anlagerungsverfahren
- Random Interchange
- Kernighan-Lin
- Greedy
- Simulated Annealing
- Evolutionäre Algorithmen

Suchaufgabe in Graphen



- Zwei typische Problemstellungen in der Praxis:
  - Finden eines bestimmten Knotens
  - Finden eines bestimmten Pfades im Graph

Lineare Suche – Aufwand (unsortiert)

unsortiert

2	1	4	6	5

Stelle, an der die Suche nach „3“ abgebrochen wird?

- Durchschnittlicher Aufwand: O(n)
  - n/2, falls das Element in der Liste vorhanden ist (Gleichverteilung auf [1..n])
  - n, falls das Element nicht in der Liste steht
- Verbesserungsversuch:
  - Sortierung der Liste
- Eventuell kann entschieden werden ob ein Element in der Liste ist, ohne diese komplett zu durchlaufen

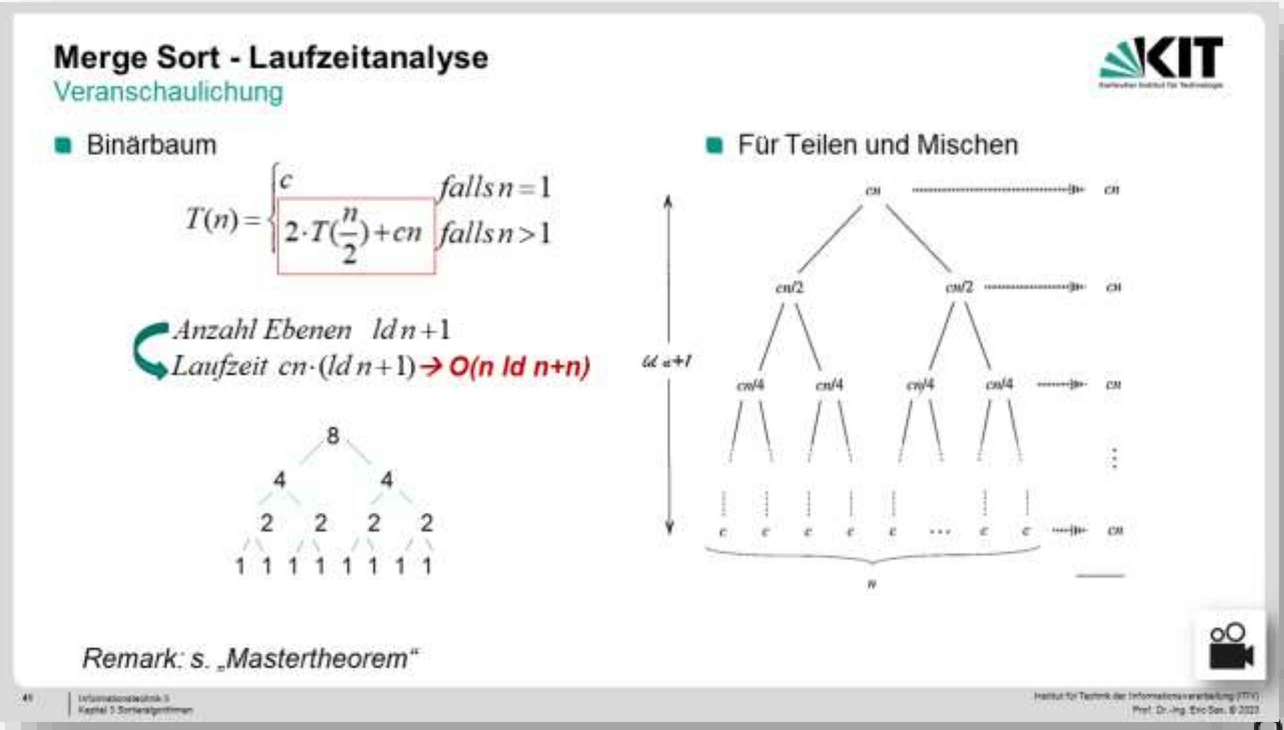
Lineare Suche – Aufwand (sortiert)

Stelle, an der die Suche nach „3“ abgebrochen wird

- Durchschnittlicher Aufwand beim Suchen in sortieren Listen mit linear wachsendem Aufwand: O(n)
  - Gleichverteilung auf [1..n] → n/2, unabhängig davon, ob das Element in der Liste vorhanden ist oder nicht
- Suche mit sub-linearem Aufwand möglich?
- Idee: Verwendung einer geeigneten Kombination aus Datenstruktur und Suchalgorithmus

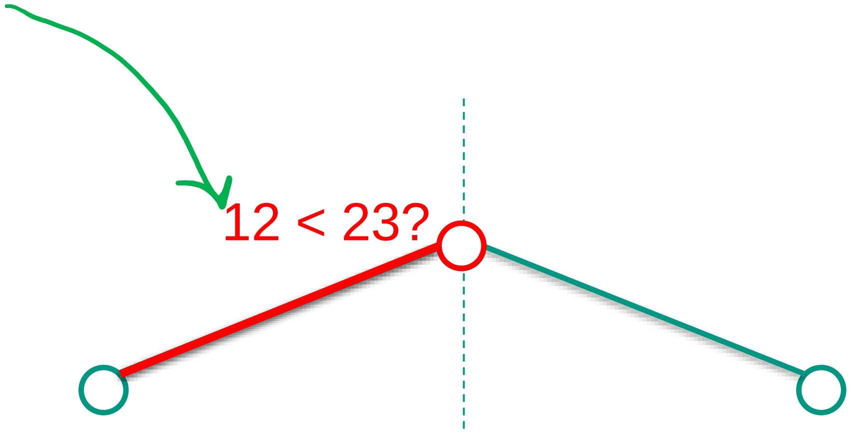
Binäre Suche

- Vorgehen:
  - Die Binärsuche springt rekursiv immer in die Mitte des noch zu durchsuchenden Intervalls und überprüft, ob der gesuchte Wert größer oder kleiner als das so bestimmte Pivotelement ist.
  - Der Suchraum wird bei jedem Schritt halbiert.
- Vorraussetzungen:
  - Sortierte Datenstruktur mit freiem Zugriff auf Elemente (Array)
- Komplexität (worst case):
  - Aufwand der Binärsuche ist O(ld(n)) (→ Suche im Binärbaum der Tiefe ld(n)) und damit sublinear.
- Remark: Bei der Suche bewege ich mich nur auf einem Pfad; beim Mergesort wird jeder Pfad berechnet.



Beispiel: Binäre Suche

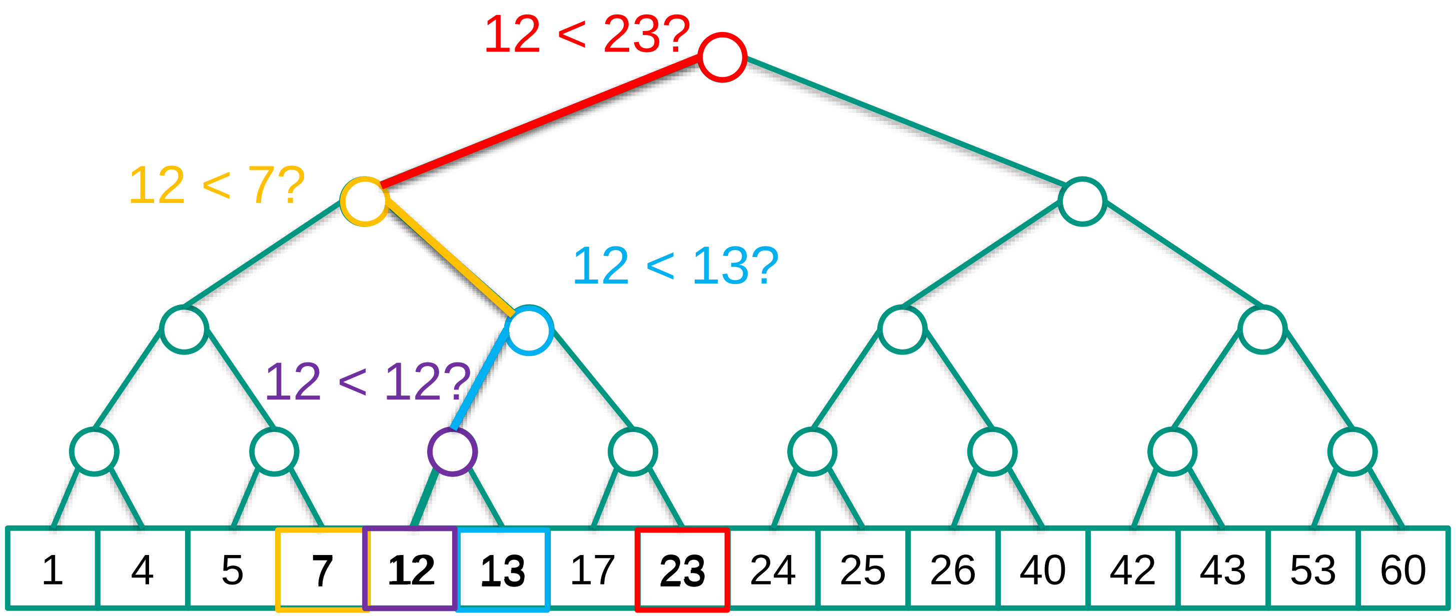
- Vorgehen:
  - Die Binärsuche springt rekursiv immer in die Mitte des noch zu durchsuchenden Intervalls und überprüft, ob der gesuchte Wert größer oder kleiner als das so bestimmte Pivotelement ist
  - Der Suchraum wird bei jedem Schritt halbiert
  - Beispiel: *Suche nach 12*



1	4	5	7	12	13	17	**23**	24	25	26	40	42	43	53	60

Beispiel: Binäre Suche

- Vorgehen:
  - Die Binärsuche springt rekursiv immer in die Mitte des noch zu durchsuchenden Intervalls und überprüft, ob der gesuchte Wert größer oder kleiner als das so bestimmte Pivotelement ist
  - Der Suchraum wird bei jedem Schritt halbiert
  - Beispiel: Suche nach 12

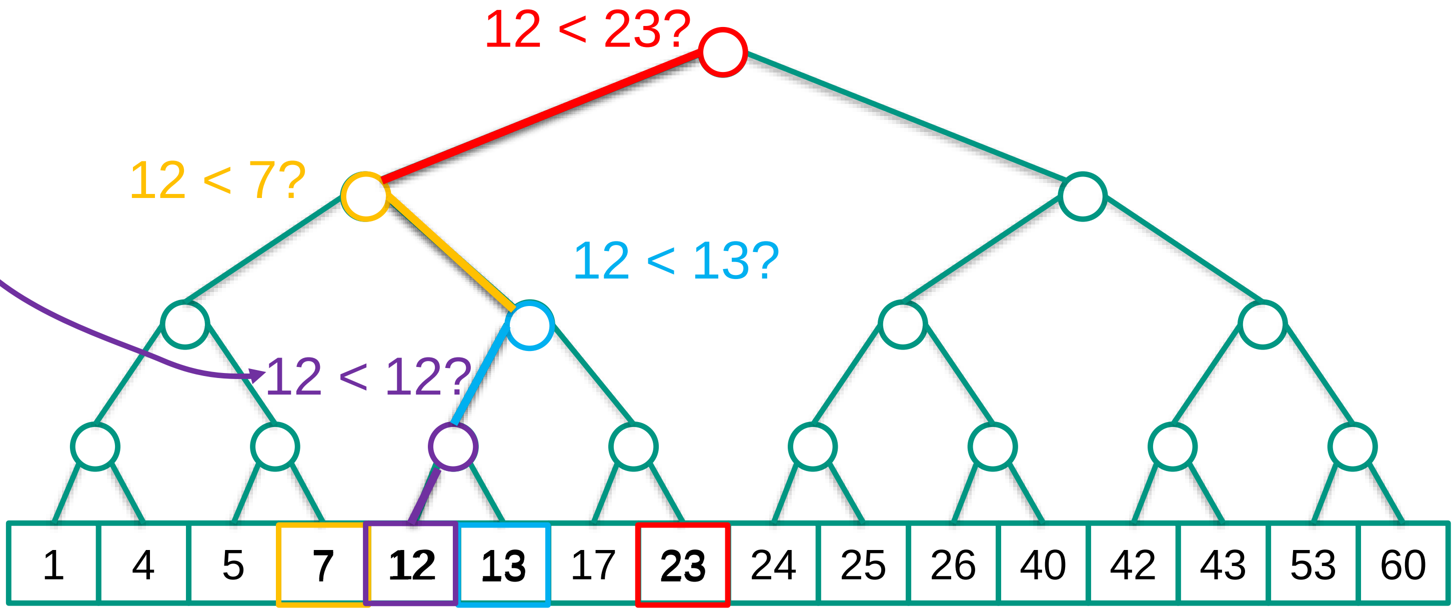


Beispiel: Binäre Suche (rekursive Variante)

```
binarySearch( list, key, bottom, top )
  center = ( bottom + top ) DIV 2
  if list[center] == key
  then return center
  elseif top - bottom > 0
  then if key < list[center]
    then return binarySearch( list, key, bottom, center )
    else return binarySearch( list, key, center + 1, top )
```

→ (list, 12, 1, 16)
→ (1 + 16)/2=8
→ list[8]=12?
→ list[4]=12?
→ list[6]=12?
→ list[5]=12?

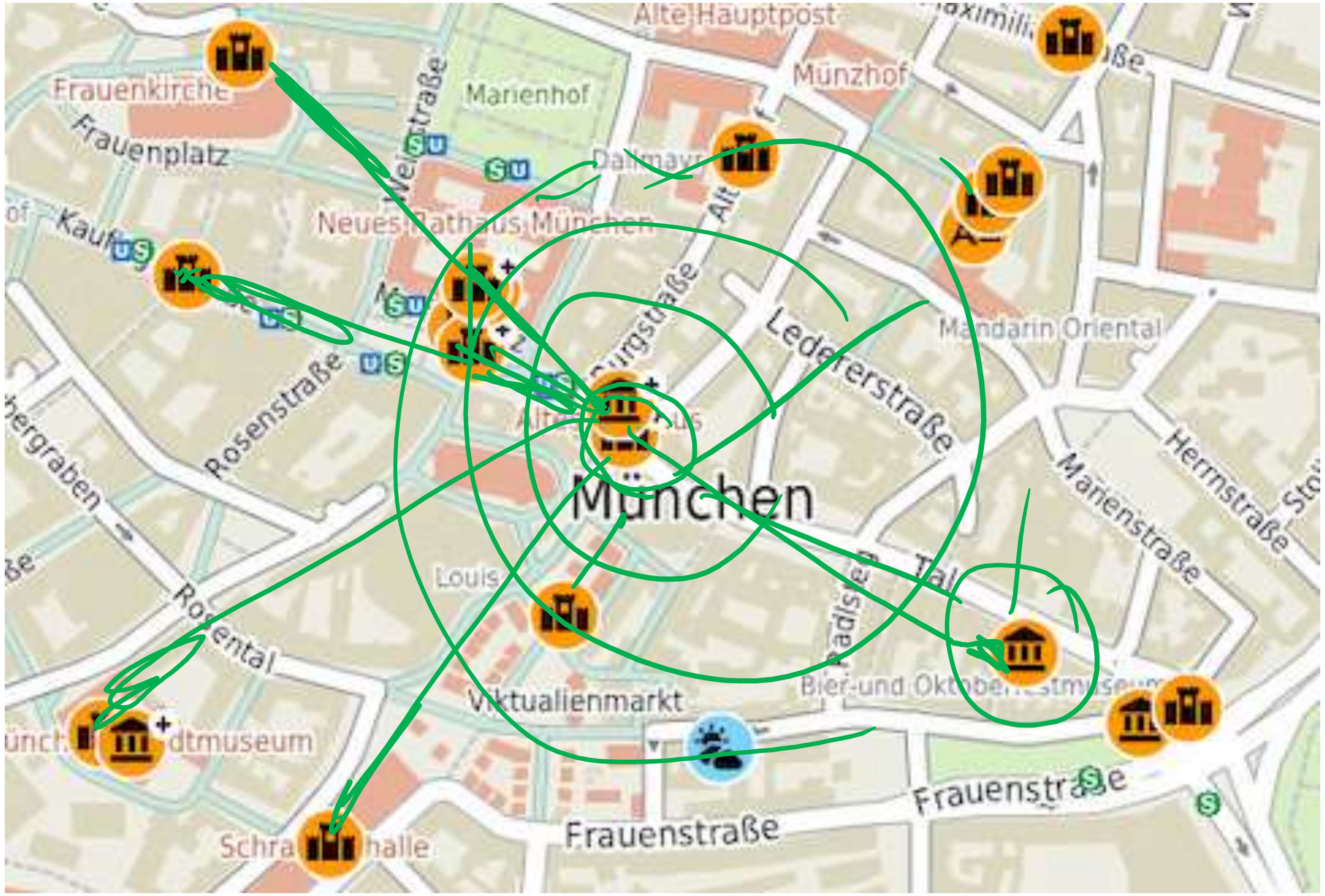
→ 12<list[8]? list[4]? list[6]?
→ (list, 12, 1, 8)
→ (list, 12, 5, 8)



- Binäre Suche

Alternative Suchmethoden

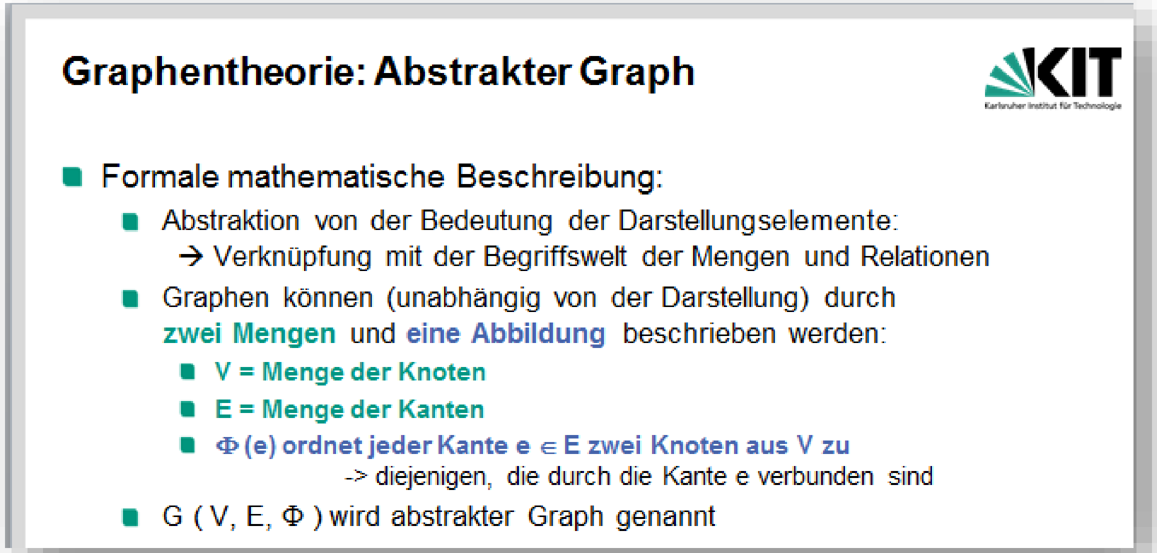
München: Kürzester Weg vom Marienplatz zur Frauenkirche



Institut für Technik der Informationsverarbeitung (ITIV)
Prof. Dr.-Ing. Eric Sax, © 2020

Breitensuche

- Breitensuche (Breadth First Search – BFS) ist ein grundlegender Algorithmus der Graphentheorie
- Resultate des Algorithmus:
  - Erzeugt einen Breitensuchbaum mit Wurzel s, der alle erreichbaren Knoten auf dem kürzesten Weg mit s verbindet
  - Findet alle von einem vordefinierten Startknoten s erreichbaren Knoten
- Eigenschaften:
  - Hat einen Laufzeitaufwand von O(V+E)
  - Funktioniert für gerichtete und ungerichtete Graphen



Graphentheorie: Abstrakter Graph

- Formale mathematische Beschreibung:
  - Abstraktion von der Bedeutung der Darstellungselemente:
    → Verknüpfung mit der Begriffswelt der Mengen und Relationen
  - Graphen können (unabhängig von der Darstellung) durch **zwei Mengen** und **eine Abbildung** beschrieben werden:
    - **V = Menge der Knoten**
    - **E = Menge der Kanten**
    - **Φ (e) ordnet jeder Kante e $\in$ E zwei Knoten aus V zu**
      -> diejenigen, die durch die Kante e verbunden sind
  - G (V, E, Φ) wird abstrakter Graph genannt

s. IT1

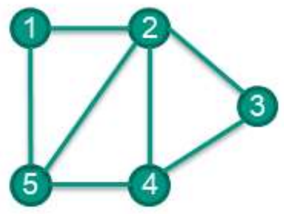
Graphentheorie: Adjazent
(Knoten orientiert)

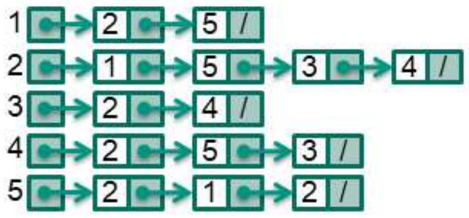
- Zwei durch eine Kante e verbundene Knoten g und h heißen adjazent zur Kante e
- **Adjazenzmatrix (Nachbarschaftsmatrix)**
  - speichert, welche Knoten des Graphen durch eine Kante verbunden sind.
  - besitzt für jeden Knoten eine Zeile und eine Spalte, woraus sich für *n* Knoten eine n x n-Matrix ergibt.
- Gerichteter Graph
  - Gibt es eine Kante von Knoten g zu Knoten h, wird in der Matrix in der g-ten Zeile an der h-ten Stelle eine 1 eingetragen.
- Ungerichteter Graph
  - Es muss eigentlich nur die Hälfte gespeichert werden, da sich die andere Hälfte durch Spiegelung ergibt.

s. IT1

Graphen: Darstellung Adjazenzmatrix
(Knoten orientiert)

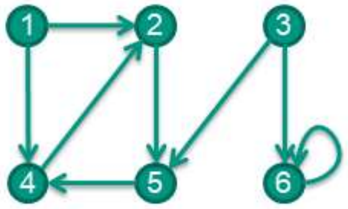
- Adjazenzdarstellung für einen ungerichteten Graphen:

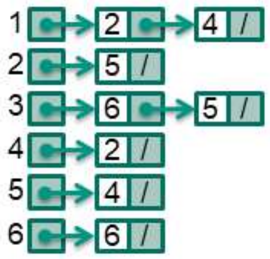




	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

- Adjazenzdarstellung für einen gerichteten Graphen:



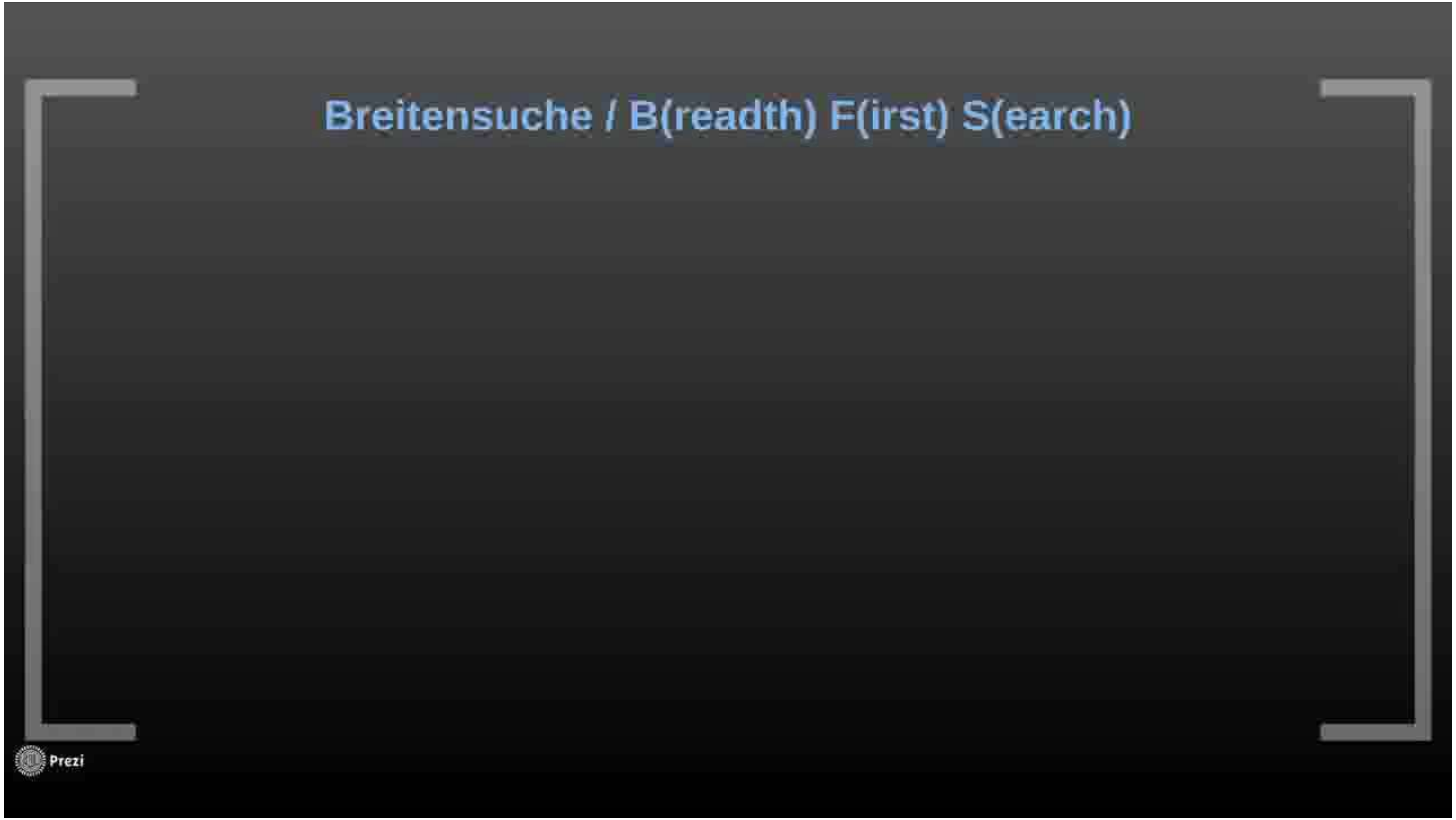


	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

9 | Informationstechnik
Kapitel 7: Datenstrukturen

Institut für Technik der Informationsverarbeitung (ITIV)
Prof. Dr.-Ing. Eric Sax, © 2017

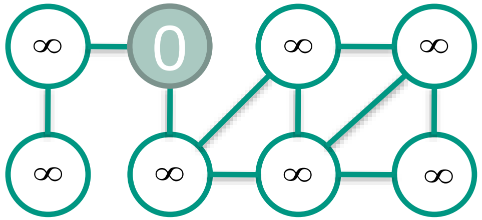
Breitensuche



Ende VL4

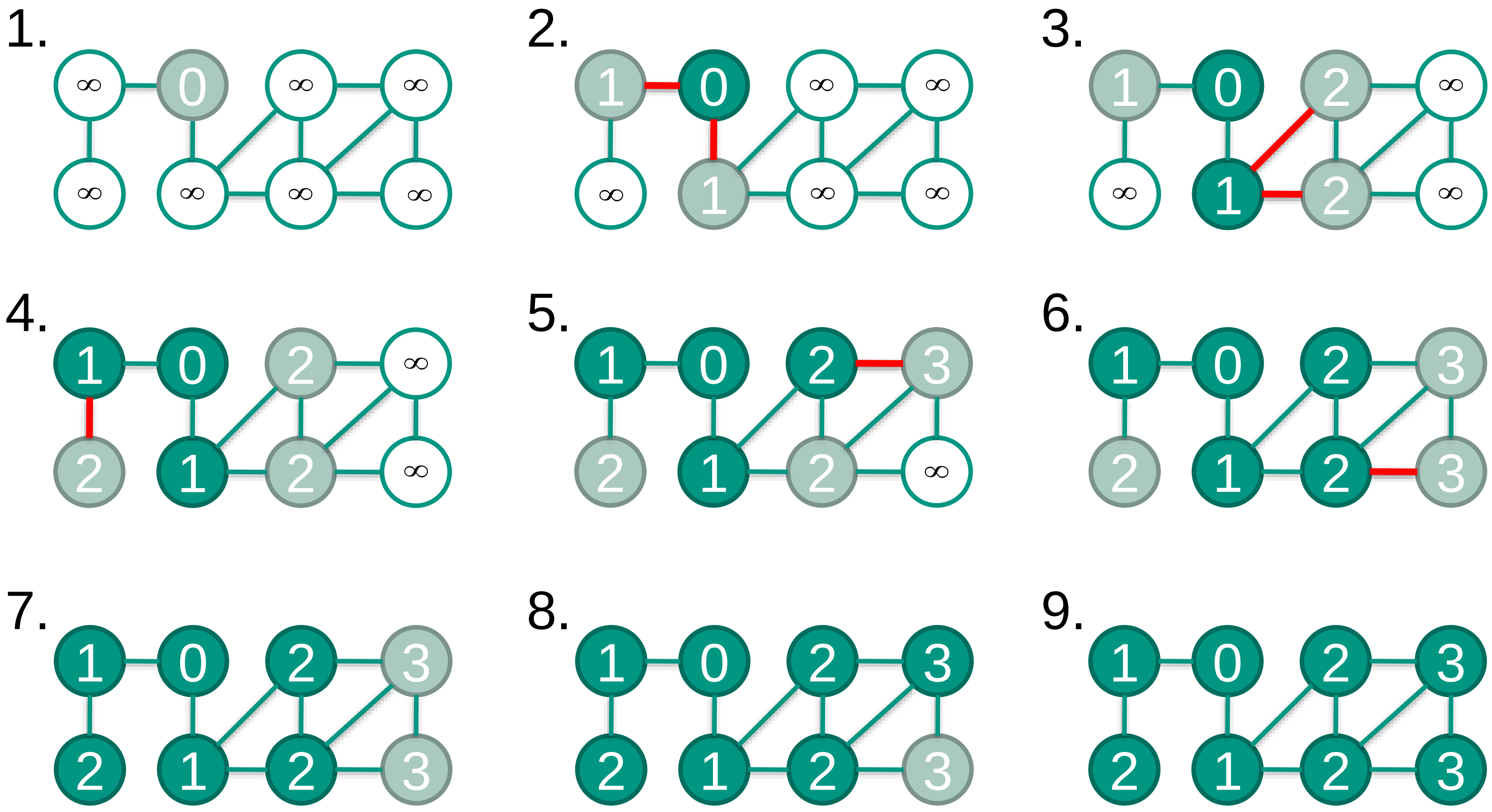
Ablauf: Breitensuche

1.



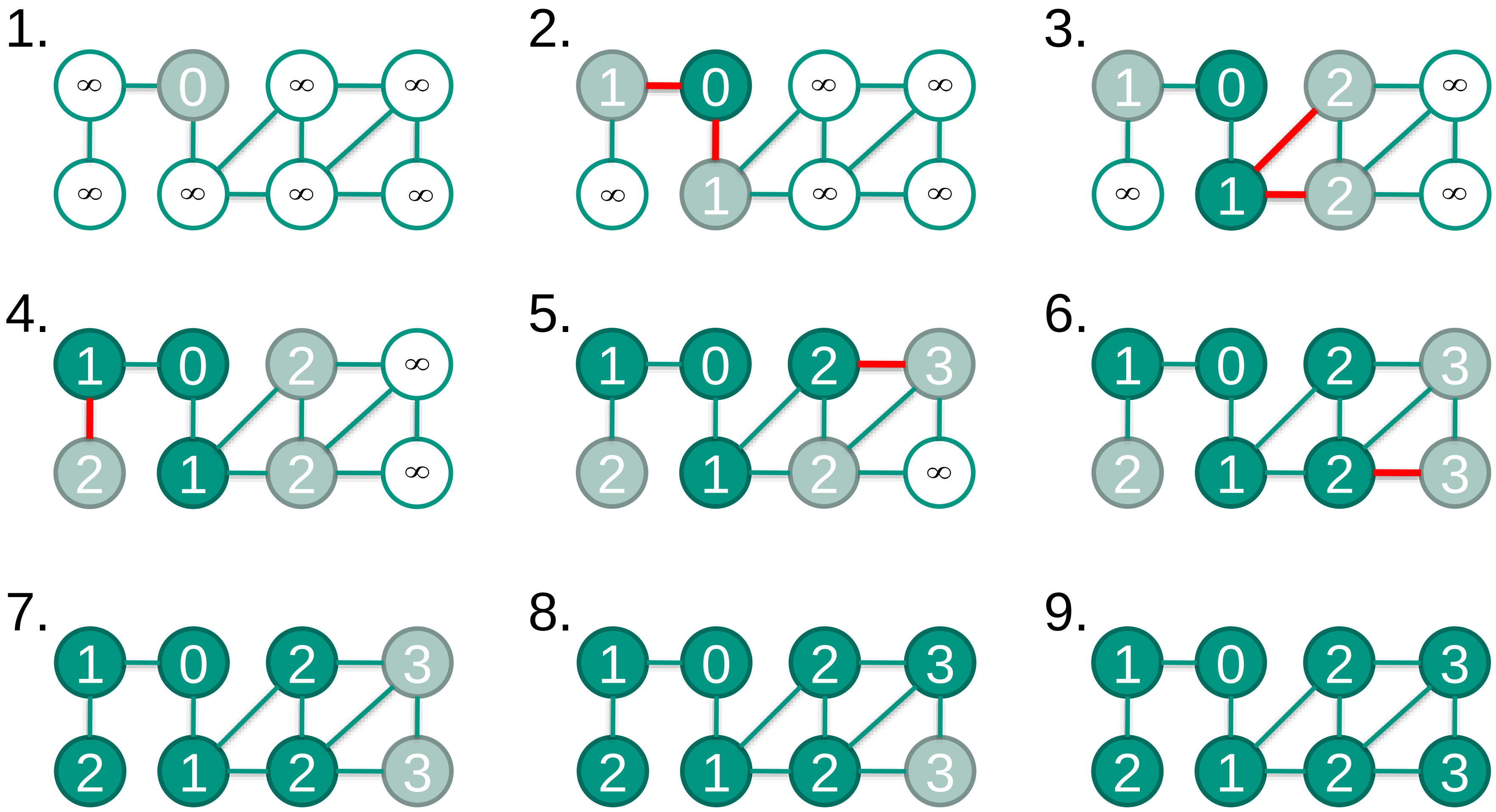
Am Anfang sind alle Knoten weiß, entdeckte Knoten werden grau, schon abgearbeitete Knoten dunkelgrün.

Ablauf: Breitensuche



Am Anfang sind alle Knoten weiß, entdeckte Knoten werden grau, schon abgearbeitete Knoten dunkelgrün.

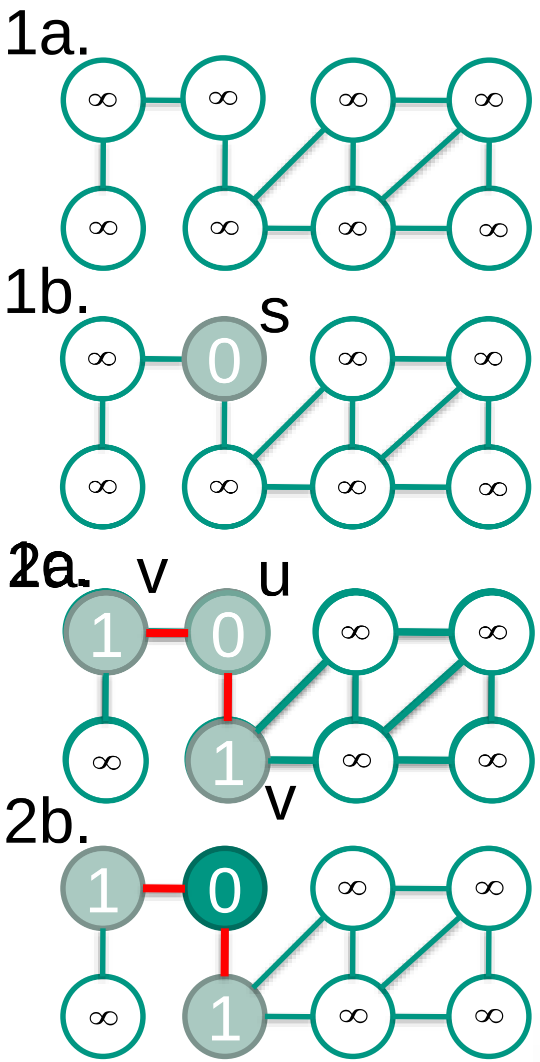
Ablauf: Breitensuche



Am Anfang sind alle Knoten weiß, entdeckte Knoten werden grau, schon abgearbeitete Knoten dunkelgrün.

Breitensuche: Pseudocode

```
BreadthFirstSearch( G, s )
  for alle Knoten u in G          //
  do farbe[u] = weiss             //
     d[u] = ∞                     //
     vater[u] = NIL               //

  create queue Q                  //
  farbe[s] = grau                 //
  d[s] = 0                        //
  enqueue( Q, s )                 //

  while Q not empty               //
  do u = dequeue( Q )             //
    for alle Knoten v aus Adj[u]  //
    do if farbe[v] == weiss       //
       then farbe[v] = grau       //
            d[v] = d[u] + 1       //
            vater[v] = u          //
            enqueue( Q, v )       //
    farbe[u] = dunkelgrün         //
```



Am Anfang sind alle Knoten weiß, entdeckte Knoten werden grau, schon abgearbeitete Knoten dunkelgrün.

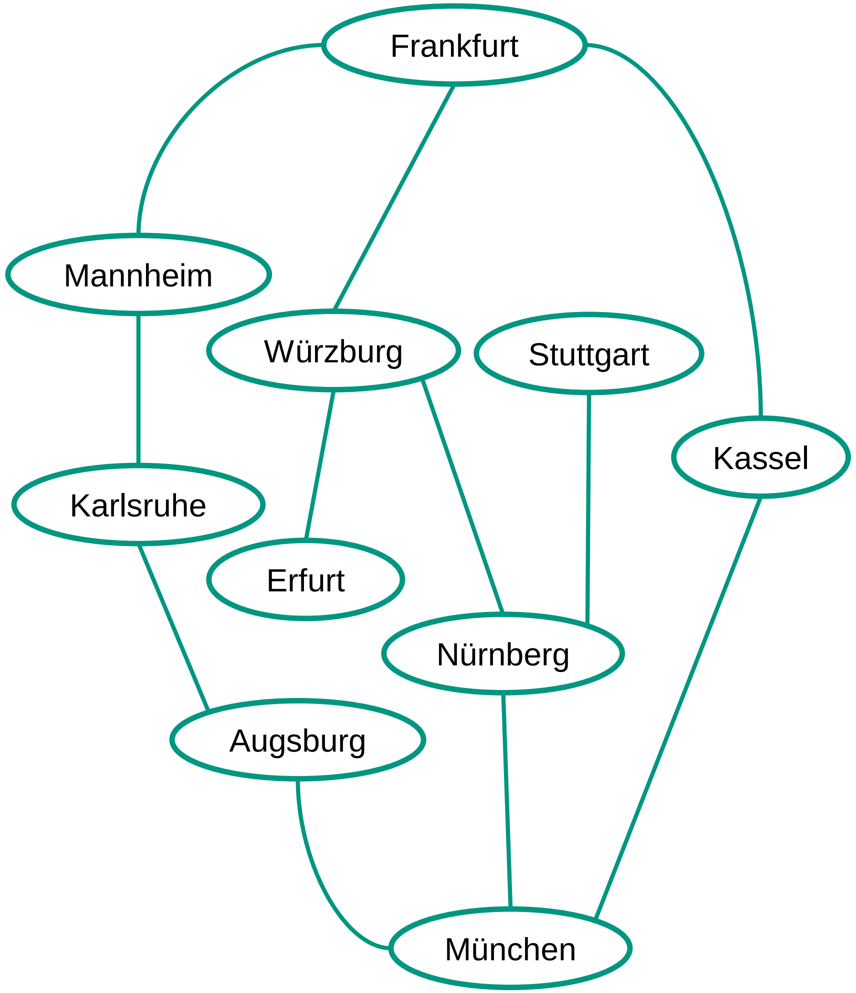
Breitensuche: Pseudocode

```
BreadthFirstSearch( G, s )
  for alle Knoten u in G               // Initialisiere alle Knoten im Graph
  do farbe[u] = weiss                  // Alle Knoten wurden noch nicht gefunden
     d[u] = ∞                          // Alle Knoten haben noch keinen Abstand
     vater[u] = NIL                    // Alle Knoten haben noch keinen Vater

  create queue Q                       // Erzeuge eine Warteschlange Q
  farbe[s] = grau                      // Starte mit Startknoten s
  d[s] = 0                             // Abstand zum Startknoten d
  enqueue( Q, s )                      // Hänge s an die Warteschlange

  while Q not empty                    // Solange noch Knoten abgearbeitet werden
  do u = dequeue( Q )                  // hole den nächsten Knoten u
    for alle Knoten v aus Adj[u]       // Für alle zu u adjazente Knoten v
    do if farbe[v] == weiss            // prüfe ob v schon gefunden wurde
       then farbe[v] = grau            // wenn nicht, dann setze v auf gefunden
            d[v] = d[u] + 1            // Weise Knoten v seinen Abstand zu
            vater[v] = u               // Der Vater von Knoten v ist der Knoten u
            enqueue( Q, v )            // Merke v für die weitere Verarbeitung
    farbe[u] = dunkelgrün              // Markiere Knoten u als abgearbeitet
```

Am Anfang sind alle Knoten weiß, entdeckte Knoten werden grau, schon abgearbeitete Knoten dunkelgrün.

Beispiel: Breitensuche

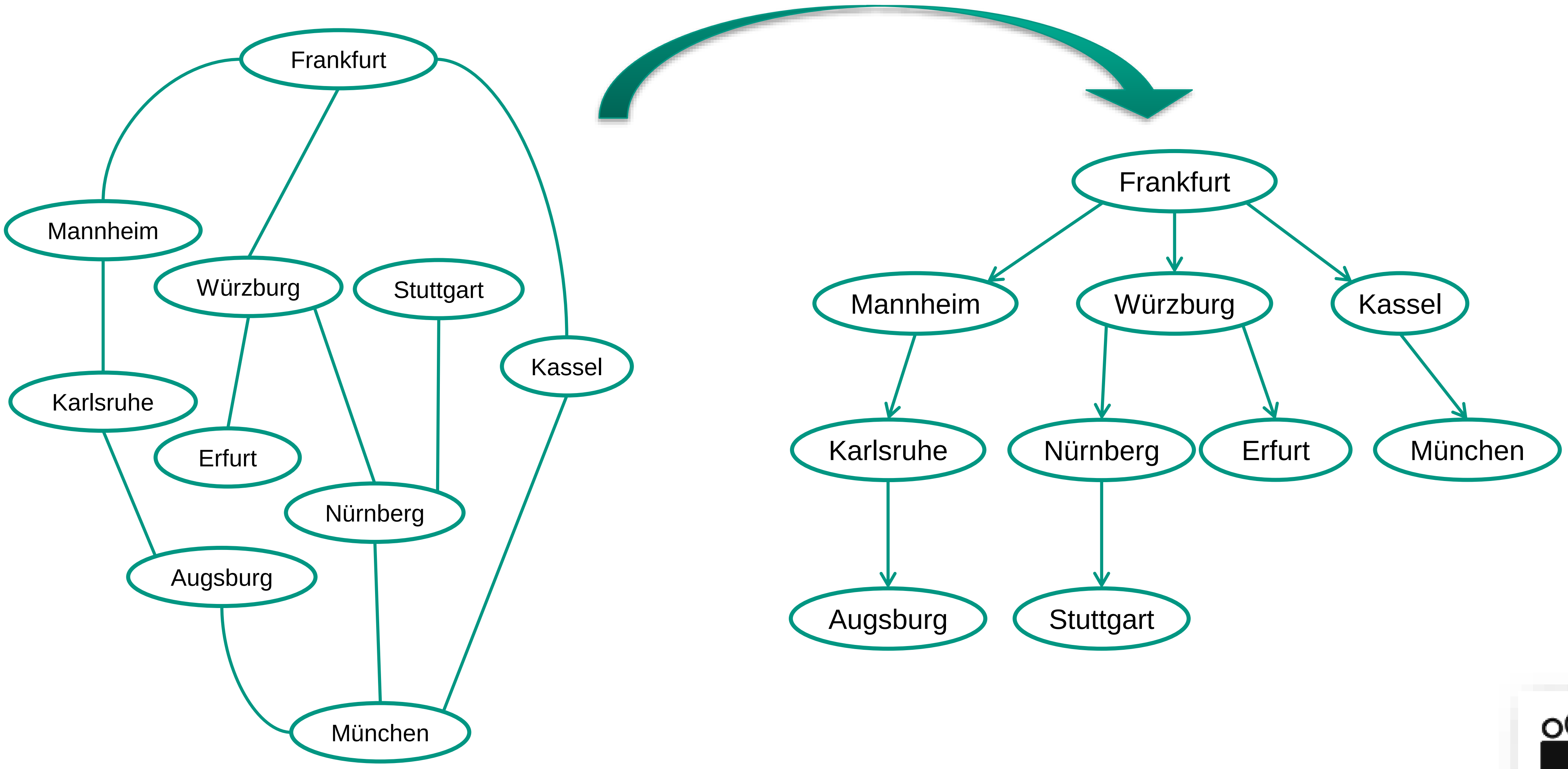
Start in Frankfurt



- Von Frankfurt aus werden die Nachbarn betrachtet:
  - Mannheim, Würzburg und Kassel.
- Von diesen aus werden auch jeweils die Nachbarn betrachtet:
  - Karlsruhe, Nürnberg, Erfurt, München.
- Gehen wir nun eine Stufe weiter, finden wir
  - als Nachbar von Karlsruhe → Augsburg
  - als Nachbar von Nürnberg → Stuttgart und München.
- Da München jedoch auf anderem Wege schon besucht wurde, wird es nicht mehr in den Graphen aufgenommen.
- Augsburg hat nun als einzigen Nachbarn noch das bereits besuchte München,
  - Stuttgart hat keinen Nachbarn außer Nürnberg, wo wir aber hergekommen sind.
- Die Breitensuche ist also komplett.
- Es entsteht ein Breitensuchbaum.

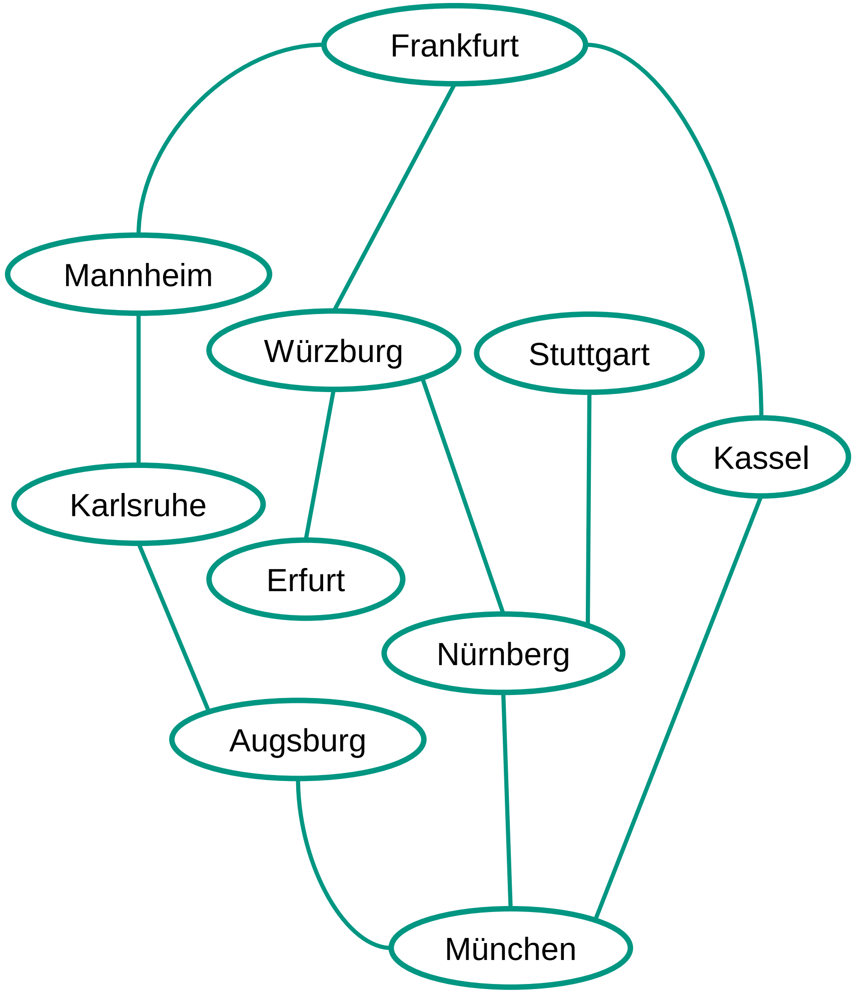
Beispiel: Breitensuche

Start in Frankfurt



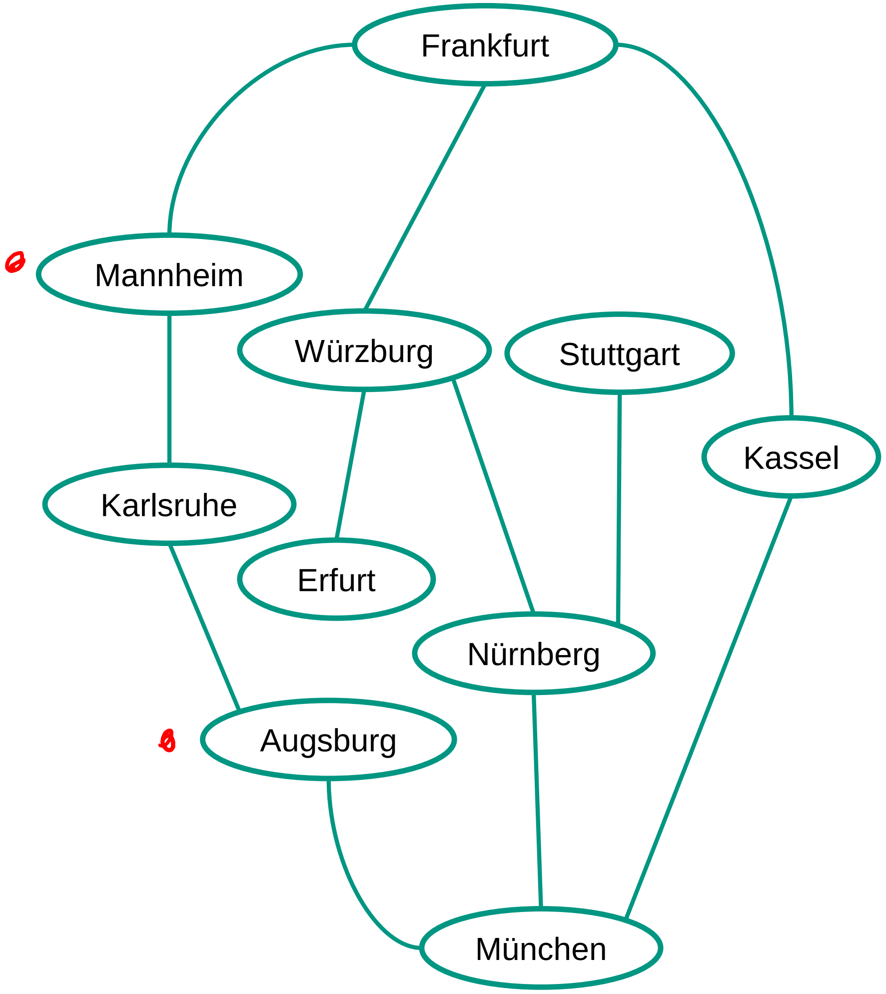
Übung: Breitensuche

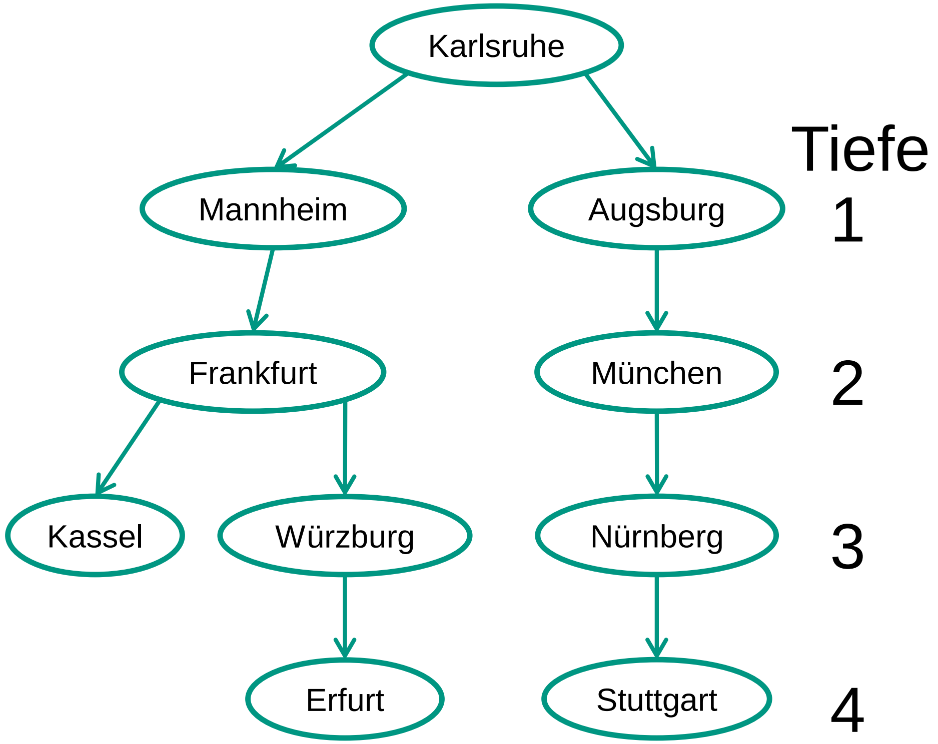
- Erzeugen Sie den Breitensuchbaum, wenn Sie in Karlsruhe starten.



Übung: Breitensuche

- Erzeugen Sie den Breitensuchbaum, wenn Sie in Karlsruhe starten.





Beispiel Anwendung Breitensuche

LEE-Algorithmus (maze routing)

- Der Lee-Algorithmus ist eine von mehreren Lösungen zur Breitensuche/Pathfinding
  - zum Finden eines Weges von einem Ausgangspunkt zu einem Zielpunkt.
- Zuerst wurde er bei der automatischen Erstellung von Platinen angewandt, kann aber auch bei Computerspielen zum Einsatz kommen, um die Figuren innerhalb der Spielwelt zu bewegen.

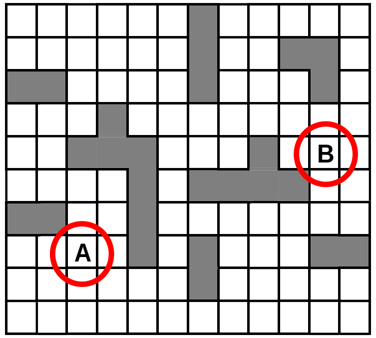
1. Schritt
- Auswahl des Startpunktes

N Schritte
- Fortlaufende Nummerierung aller Nachbarpunkte in Achsenrichtung

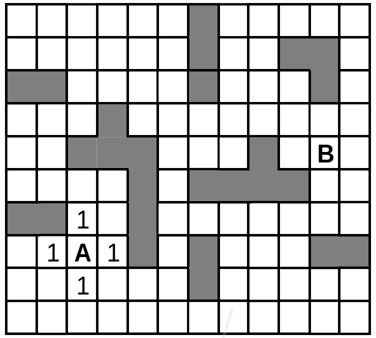
Abbruch
- Rückwärtssuche des kürzesten Weges entlang der Nummerierung

Beispiel für LEE-Algorithmus

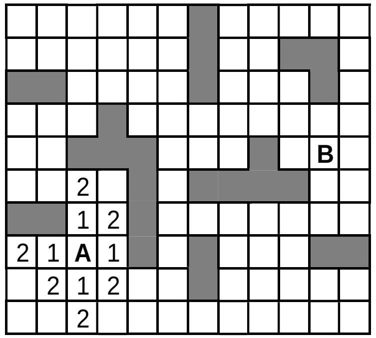
- Start in A → Gesucht wird B



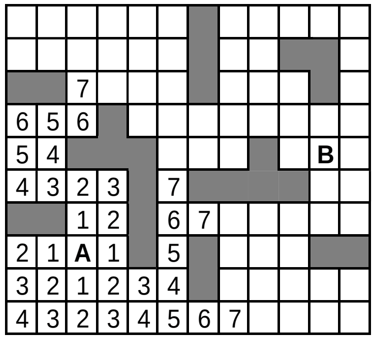
Verbindungspunkte



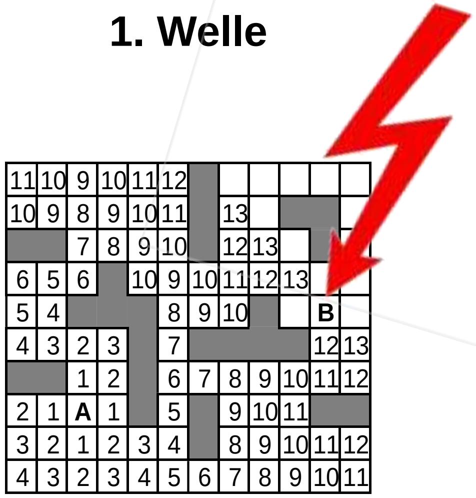
1. Welle



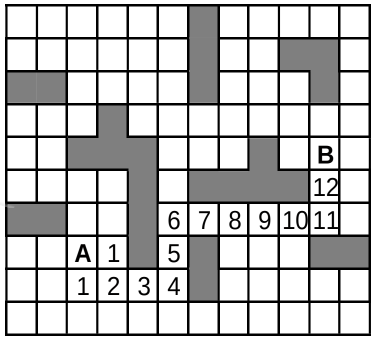
2. Welle



3. bis 7. Welle



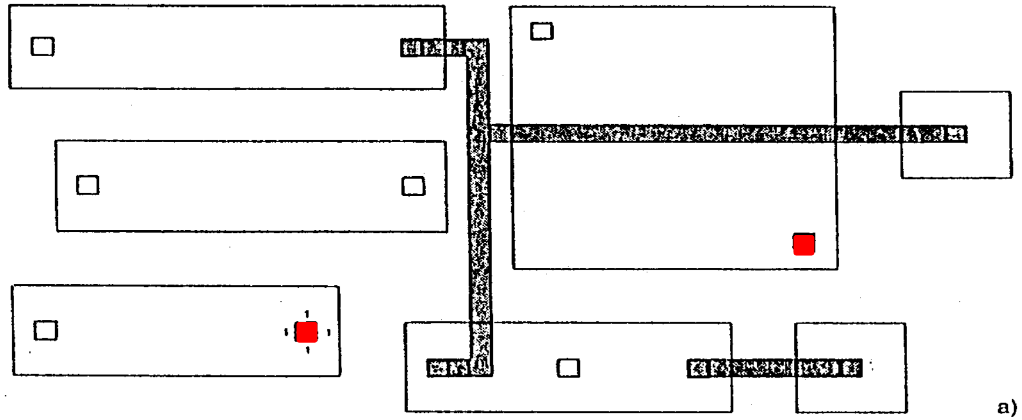
8. bis 13. Welle



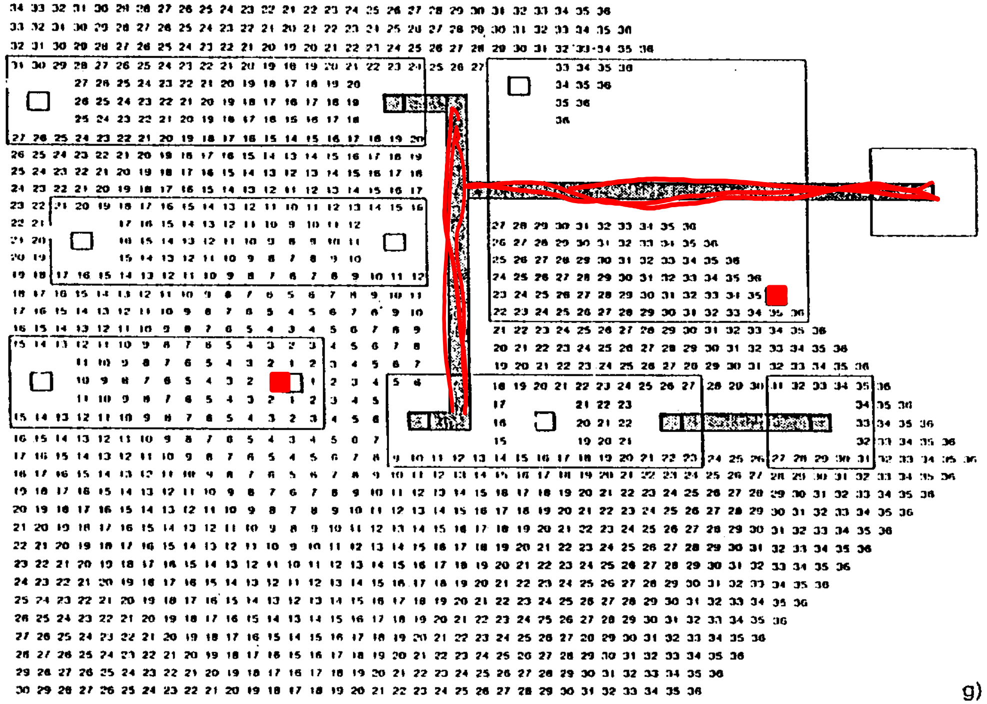
Rückverfolgung

Institut für Technik der Informationsverarbeitung (ITIV)
Prof. Dr.-Ing. Eric Sax, © 2020

Beispiel: LEE-Algorithmus

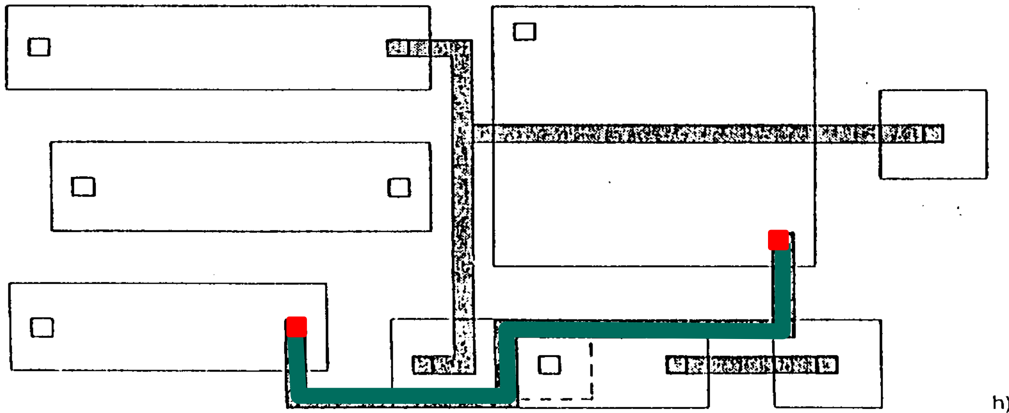


Beispiel: LEE-Algorithmus



g)

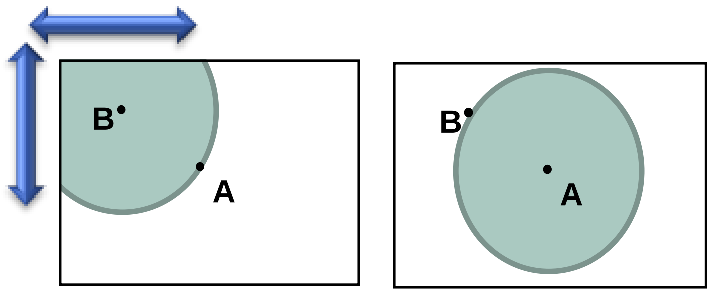
Beispiel: LEE-Algorithmus



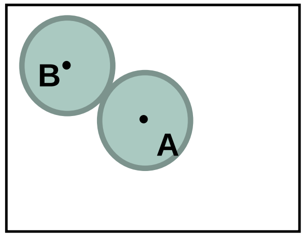
35. Welle → resultierende Verbindung (hier kürzeste Leiterbahn)

Modifikation des LEE-Algorithmus

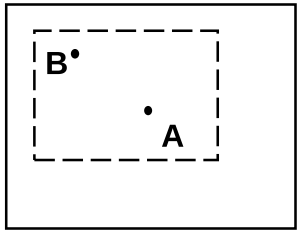
- Günstige Wahl des Startpunktes:



- Gleichzeitige Wegsuche von
  Start- und Zielpunkt:



- Einführung eines Suchrahmens:



- Übergang von Zweipunkt- zu Mehrpunktverbindungen
- Dreidimensionale Wegesuche möglich (Mehrlagenverbindungen)

Institut für Technik der Informationsverarbeitung (ITIV)
Prof. Dr.-Ing. Eric Sax, © 2020

- Breitensuche

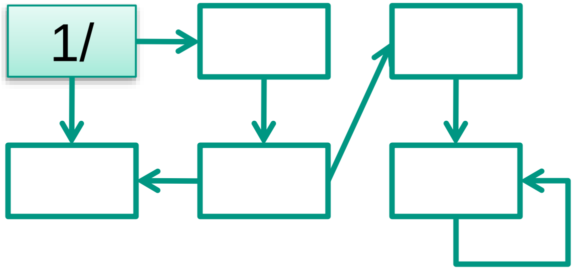
Tiefensuche

- Tiefensuche (*Depth First Search – DFS*) ist der nächste grundlegende Algorithmus der Graphentheorie.
- Im Gegensatz zur Breitensuche wird bei der Tiefensuche zunächst ein Pfad vollständig in die Tiefe beschritten, bevor abzweigende Pfade untersucht werden.
- Resultate des Algorithmus:
  - Findet alle von Startknoten s erreichbaren Knoten
  - Bearbeitet immer den zuletzt gefundenen Knoten
  - Erst nach Abarbeitung aller Nachbar-Knoten kehrt die Suche zum Vaterknoten zurück, so dass vor allem der Entdeckungszeitpunkt interessant ist.
  - Tiefensuchwald:
    - Zusammengesetzt aus mehreren Bäumen, je nach Zusammenhang des Graphen.
- Eigenschaften:
  - Hat einen Laufzeitaufwand von O(V+E)
  - Funktioniert für gerichtete und ungerichtete Graphen


Tiefensuche / D(epth) F(irst) S(earch)
Prezi

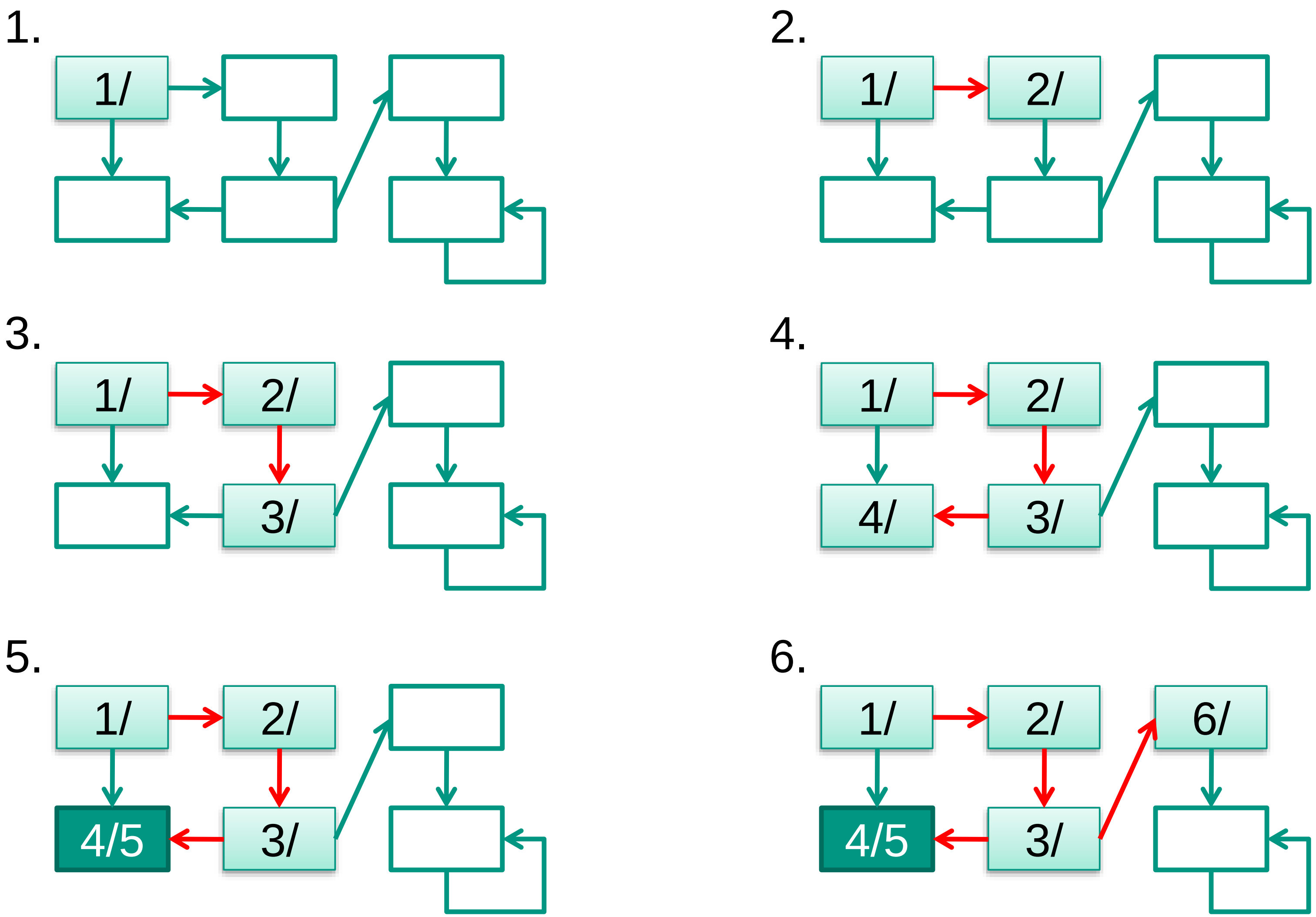
Ablauf: Tiefensuche

1.



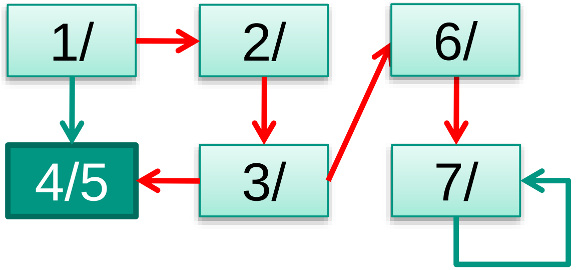
3/

Ablauf: Tiefensuche



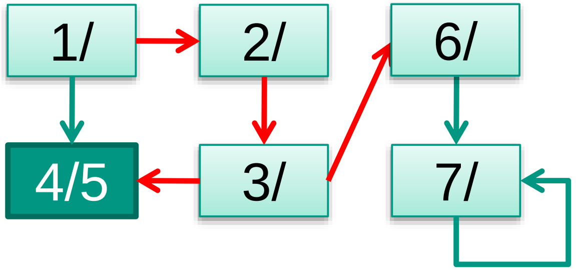
Ablauf: Tiefensuche

7.

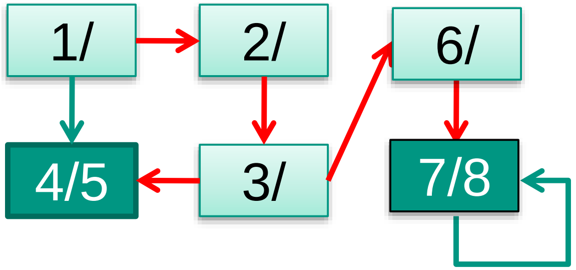


Ablauf: Tiefensuche

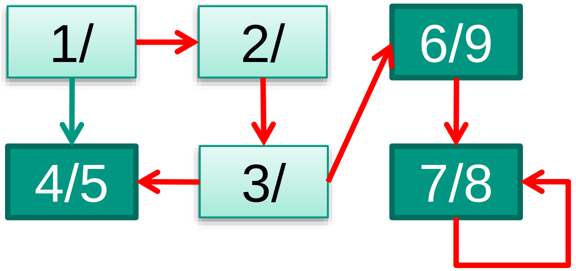
7.



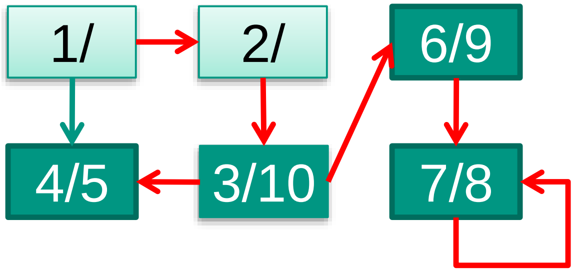
8.



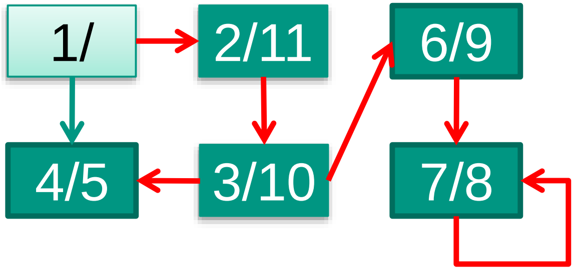
9.



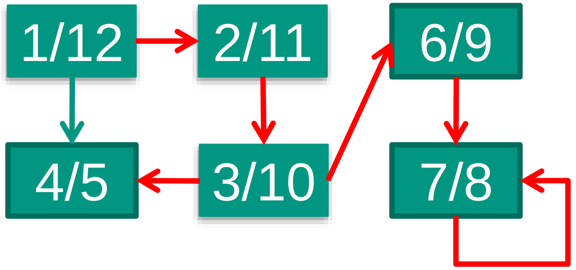
10.



11.



12.



Tiefensuche: Pseudocode

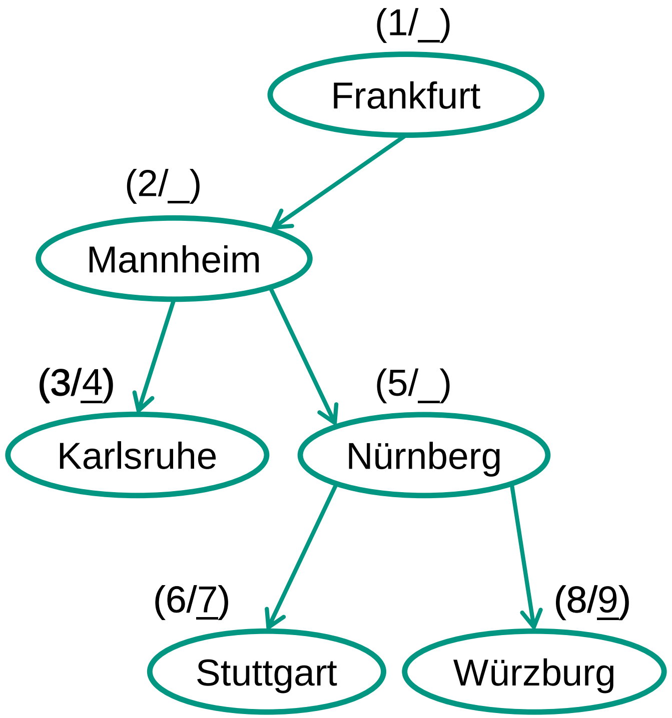
```
DepthFirstSearch( G )
  for alle Knoten u in G                //
  do farbe[u] = weiss                   //
     vater[u] = NIL                     //

  zeit = 0                              //
  for alle Knoten u in G                //
  do if farbe[u] == weiss               //
     then DFS-VISIT( u )                //

DFS-VISIT( u )
  farbe[u] = grau; zeit = zeit + 1      //
  startTime[u] = zeit                   //
  for alle Knoten v aus Adj[u]          //
  do if farbe[v] == weiss               //
     then vater[v] = u                  //
          DFS-VISIT( v )                //

  farbe[u] = grün; zeit = zeit + 1      //
  endTime[u] = zeit                     //
```

Tiefensuche: Pseudocode

```
DepthFirstSearch( G )
  for alle Knoten u in G             // Initialisiere alle Knoten im Graph
  do farbe[u] = weiss                // Alle Knoten wurden noch nicht gefunden
     vater[u] = NIL                  // Alle Knoten haben noch keinen Vater

  zeit = 0                           // Zum Messen der Zeitschritte
  for alle Knoten u in G             // Für alle Knoten im Graph,
  do if farbe[u] == weiss            // welche noch nicht bearbeitet wurden
     then DFS-VISIT( u )             // Besuch registrieren

DFS-VISIT( u )
  farbe[u] = grau; zeit = zeit + 1   // Als besucht setzen; Zeitschritt erhöhen
  startTime[u] = zeit                // Startzeitpunkt des Knotens u abspeichern
  for alle Knoten v aus Adj[u]       // Für alle zu u adjunken Knoten v
  do if farbe[v] == weiss            // prüfe ob v schon gefunden wurde
     then vater[v] = u               // Der Vater vom Knoten v ist der Knoten u
          DFS-VISIT( v )             // Weiter in die Tiefe vom Knoten v gehen

  farbe[u] = grün; zeit = zeit + 1   // Knoten u ist abgearbeitet; Zeitschritt erhöhen
  endTime[u] = zeit                  // Endzeitpunkt des Knotens u abspeichern
```

Übung: Tiefensuche

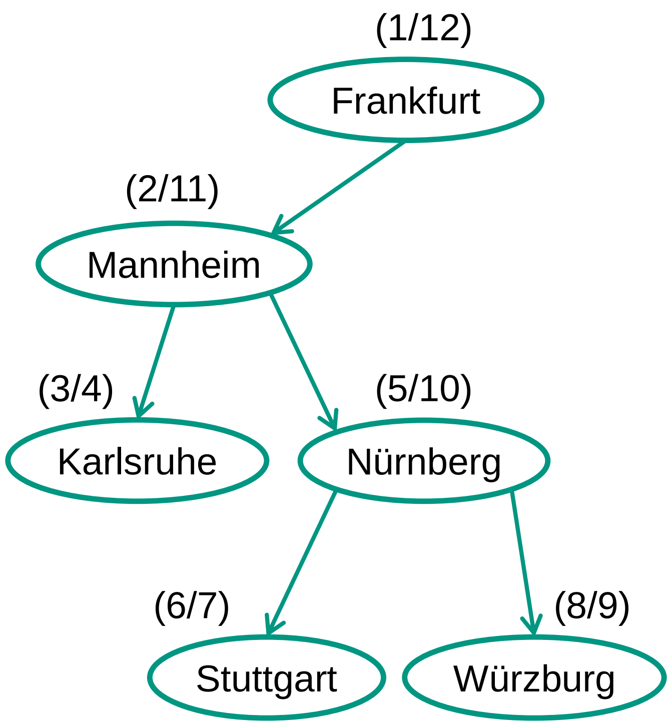
- Finden Sie mit Hilfe der Tiefensuche einen Weg von Frankfurt nach Würzburg.



1
2
3
4
5
6
7
8
9
10
11
12

Übung: Tiefensuche

- Finden Sie mit Hilfe der Tiefensuche einen Weg von Frankfurt nach Würzburg.



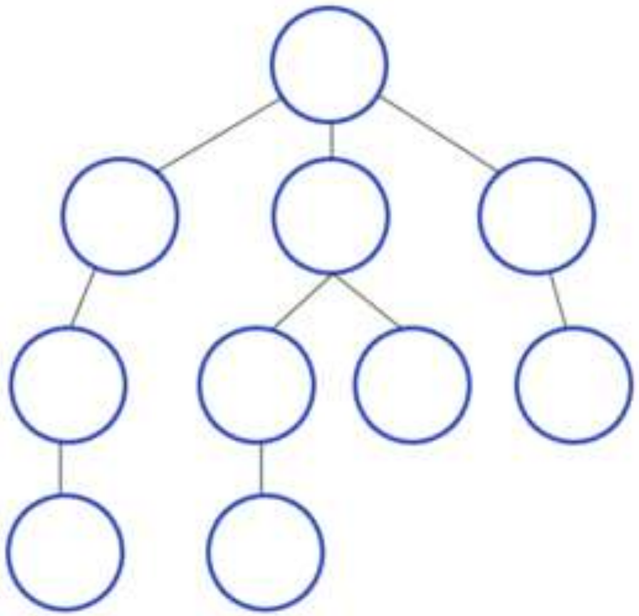

```
1   DFS(Frankfurt)
      // Frankfurt in Bearbeitung
2     DFS(Mannheim)
        // Mannheim in Bearbeitung
3       DFS(Karlsruhe)
          // Karlsruhe in Bearbeitung
4         Karlsruhe abgeschlossen
5       DFS(Nürnberg)
          // Nürnberg in Bearbeitung
6         DFS(Stuttgart)
            // Stuttgart in Bearbeitung
7           Stuttgart abgeschlossen
8         DFS(Würzburg)
            // Würzburg in Bearbeitung
                // Pfad gefunden
9           Würzburg abgeschlossen
10      Nürnberg abgeschlossen
11    Mannheim abgeschlossen
12  Frankfurt abgeschlossen
```

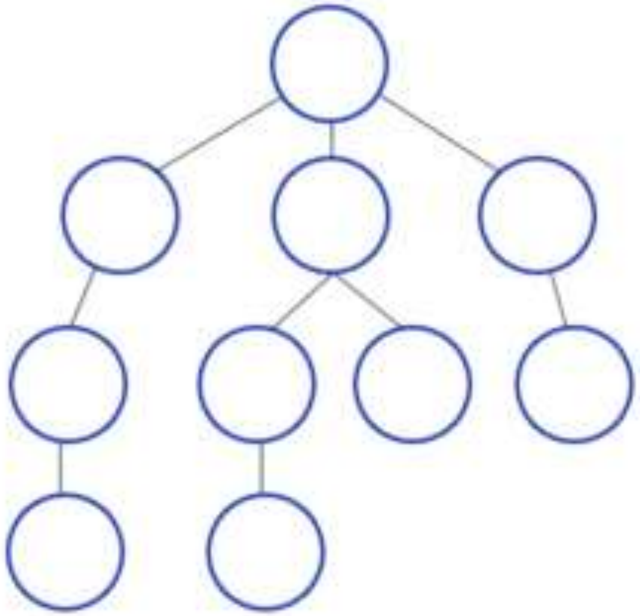
- Tiefensuche

Vergleich

- Breitensuche
  - Bevorzugt bei Knotensuche
  - Bei mehreren möglichen Lösungen, schneller alternative, kostengünstige Lösung

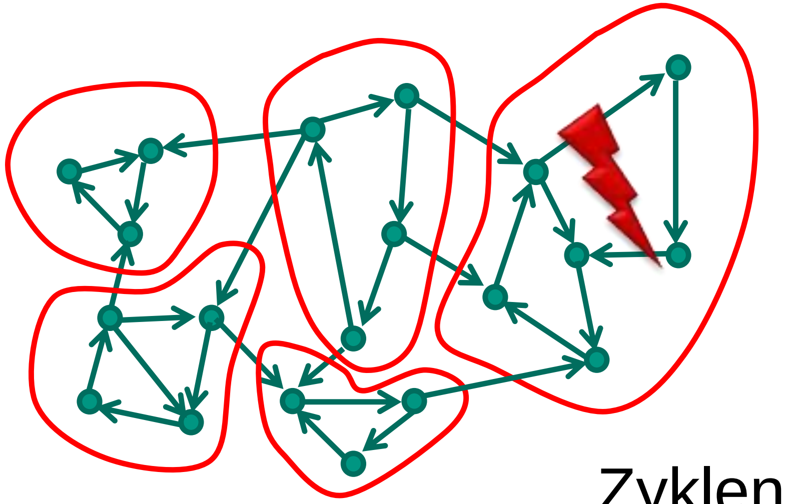
- Tiefensuche
  - Bevorzugt bei Blattsuche
  - Tiefenbeschränkung möglich
  - Kosten eskalieren bei monoton steigenden Pfad-Suchkosten





Zyklensuche

- Anwendung der Tiefensuche zum Finden von Zyklen in Graphen

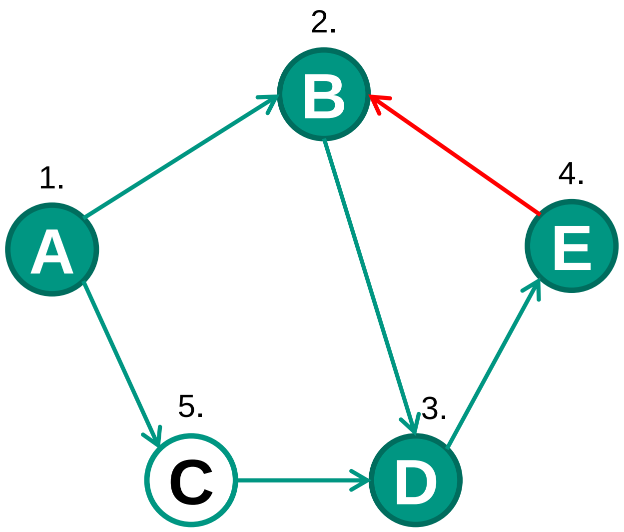


Zyklen

Zyklensuche

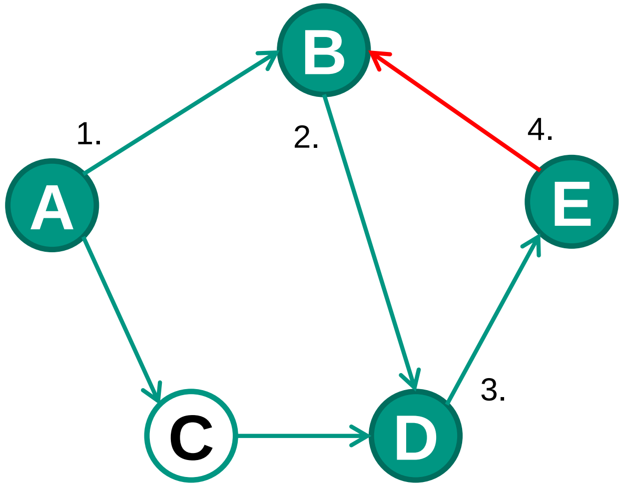
- Anwendung der Tiefensuche zum Finden von Zyklen in Graphen

- Nutzung der verschiedenen Kantenarten bei der DFS:
  - Vorwärtskanten, z.B.: von A nach C
  - Querkanten, z.B.: von C nach D
  - Rückwärtskanten, z.B.: von E nach B



Zyklensuche

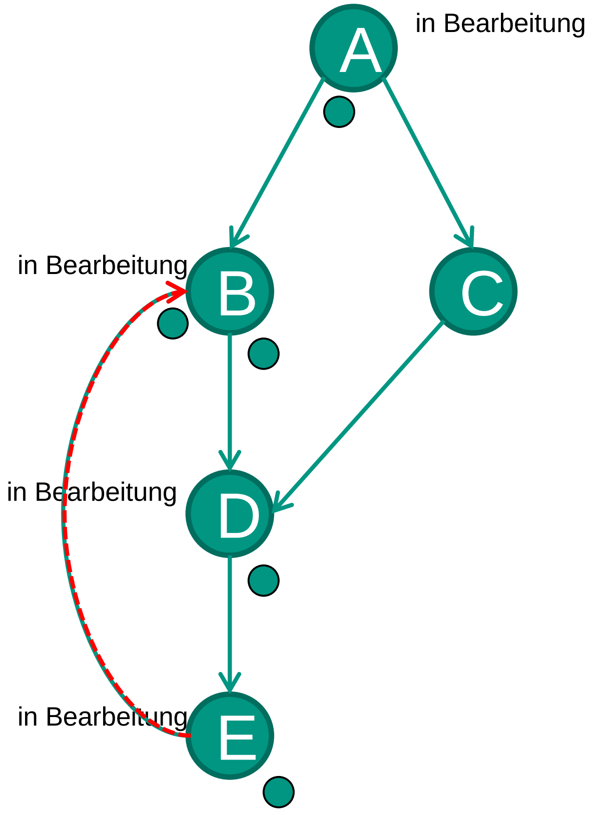
- Anwendung der Tiefensuche zum Finden von Zyklen in Graphen
- Nutzung der verschiedenen Kantenarten bei der DFS:
  - Vorwärtskanten, z.B.: von A nach C
  - Querkanten, z.B.: von C nach D
  - Rückwärtskanten, z.B.: von E nach B
- Nur Rückwärtskanten können Zyklen verursachen
- Vorgehen zur Zyklensuche:
  - Führe Tiefensuche durch
  - Sobald die Tiefensuche auf eine Rückwärtskante stößt, die auf einen Knoten mit Status „in Bearbeitung“ verweist, ist ein Zyklus gefunden



Im 4. Schritt stößt die DFS von E auf B
Ein Zyklus ist gefunden!

Beispiel: Zyklensuche

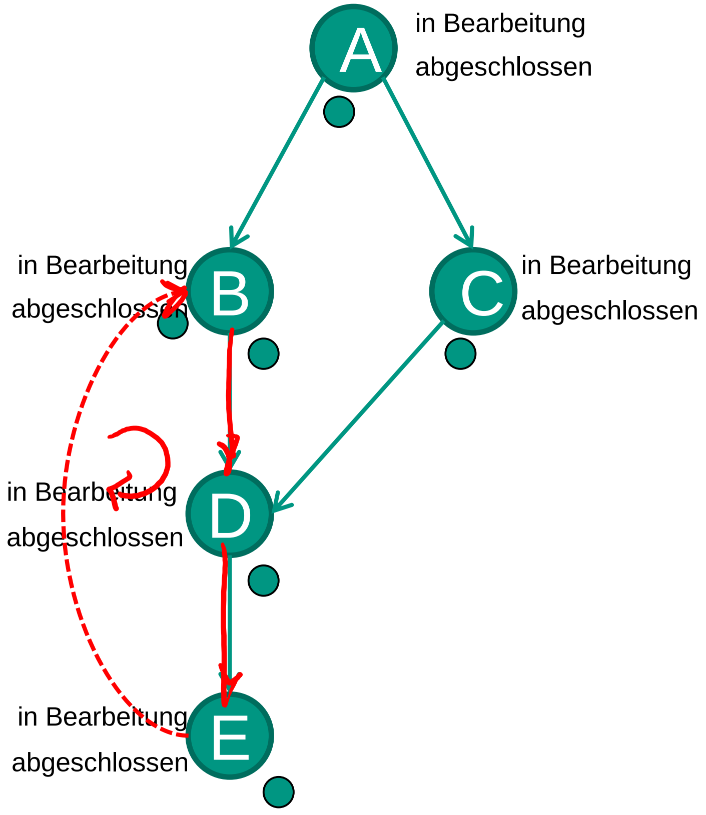
```
Zyklensuche (A) // A noch nicht begonnen
  A in Bearbeitung
  Zyklensuche (B) // B noch nicht begonnen
    B in Bearbeitung
    Zyklensuche (D) // D noch nicht begonnen
      D in Bearbeitung
      Zyklensuche (E) // E noch nicht begonnen
        E in Bearbeitung
        Zyklensuche (B) // B in Bearbeitung
               Zyklus gefunden!
```



Beispiel: Zyklensuche

```
Zyklensuche (A) // A noch nicht begonnen
  A in Bearbeitung
  Zyklensuche (B) // B noch nicht begonnen
    B in Bearbeitung
    Zyklensuche (D) // D noch nicht begonnen
      D in Bearbeitung
      Zyklensuche (E) // E noch nicht begonnen
        E in Bearbeitung
        Zyklensuche (B) // B in Bearbeitung
          Zyklus gefunden!
        E abgeschlossen
      D abgeschlossen
    B abgeschlossen
  Zyklensuche (C) // C noch nicht begonnen
    C in Bearbeitung
    Zyklensuche (D) // D abgeschlossen
    C abgeschlossen
  A abgeschlossen
```

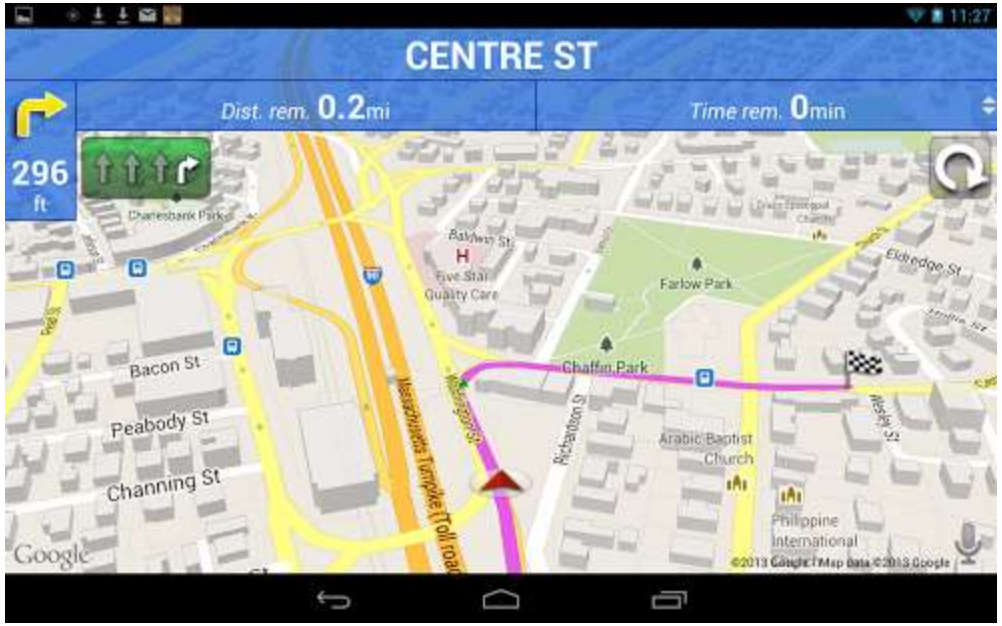
Fertig!



- Zyklensuche

Zyklen

Kürzeste Pfade



- Finden eines kürzesten Pfades in einem Graphen ist ein sehr altes, sehr bekanntes Problem
  → Distanzberechnung
- Praktische Anwendung in heutigen Navigationssystemen
- Löst das Problem des kürzesten Pfades bei einem Startknoten in einem positiv gewichteten, gerichteten Graphen G(V, E)

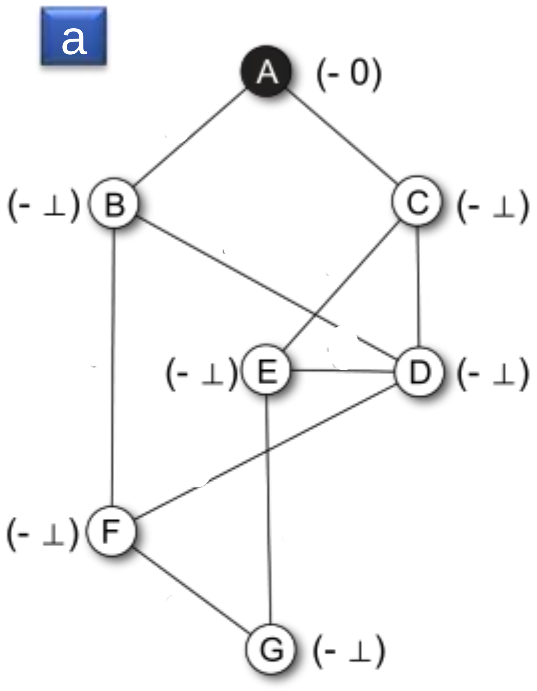
Dijkstra-Algorithmus

- Die Grundidee des Algorithmus ist es, immer derjenigen Kante zu folgen, die den kürzesten Streckenabschnitt vom Startknoten darstellt.
- Es wird die Distanz eines Knotens zu einem Zielknoten berechnet (*single-pair, shortest path*)
  oder die Distanzen aller Knoten zum Startknoten (*single-source, shortest path*).
- Es werden sowohl die kürzesten Wegstrecken als auch deren Knotenfolgen berechnet.

- **Edsger Wybe Dijkstra** (* 11. Mai 1930 in Rotterdam; † 6. August 2002 in Nuenen, Niederlande) war ein niederländischer Informatiker.
- Er war der Wegbereiter der strukturierten Programmierung

Beispiel zu Dijkstra-Routingverfahren

Vorgehen

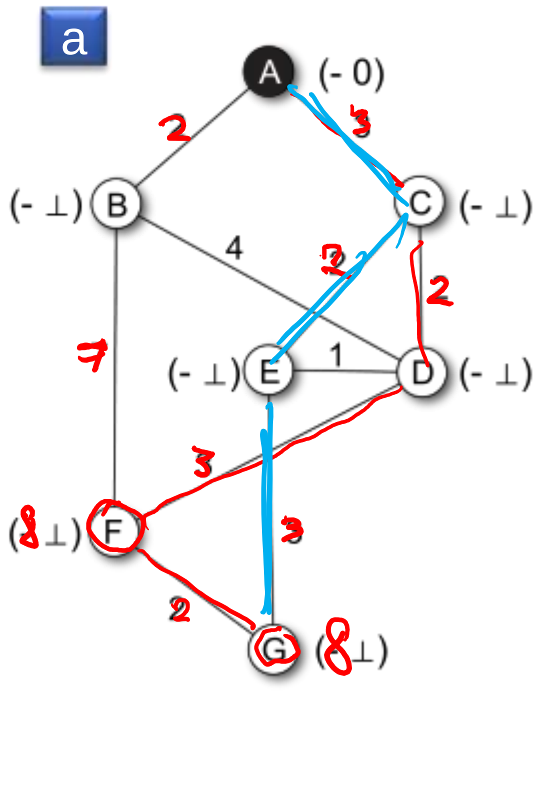


(a)

- Initialisiere die Distanz im Startknoten mit 0 und in allen anderen Knoten mit ∞ (oder hier ⊥).
- Solange es noch unbesuchte Knoten gibt, wähle darunter denjenigen mit minimaler Distanz aus und speichere, dass dieser Knoten schon besucht wurde.
- Hier: *Netzwerk mit Knoten A, B, C, D, E, F, G und den jeweiligen Verbindungsgewichten*

Beispiel zu Dijkstra-Routingverfahren

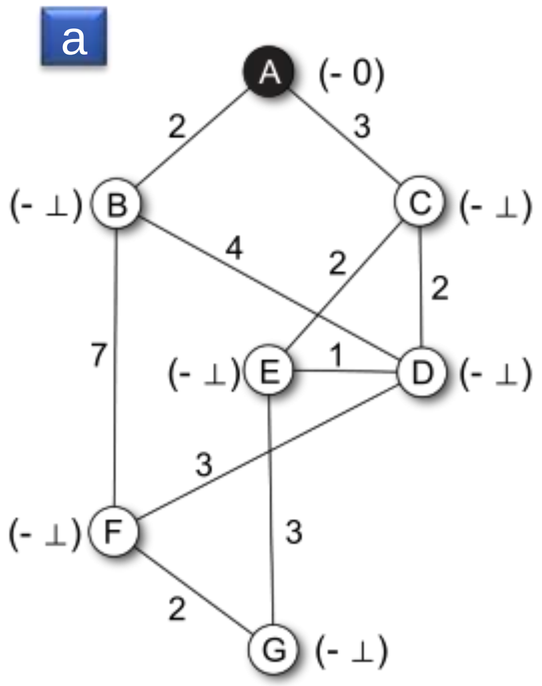
Vorgehen



(a)

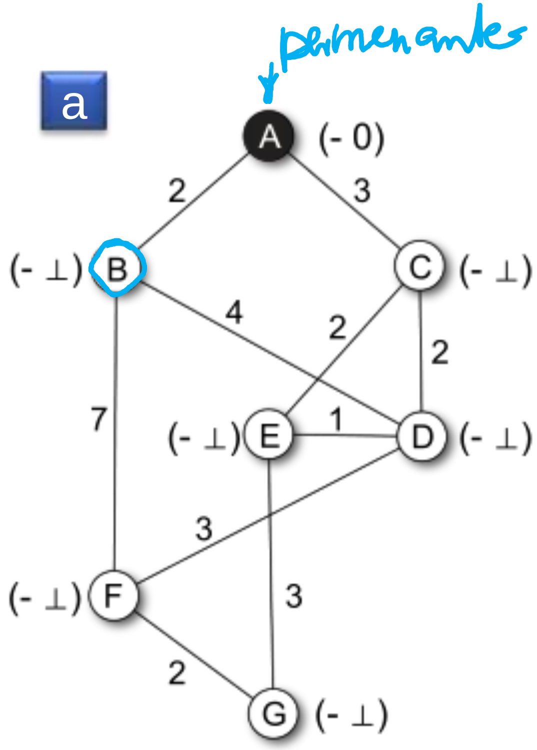
- Berechne für alle noch unbesuchten Nachbarknoten die Summe aus dem jeweiligen Kantengewicht und der Distanz vom Startknoten zum aktuellen Knoten.
- Ist dieser Wert für einen Knoten kleiner als die dort gespeicherte Distanz, aktualisiere sie und setze den aktuellen Knoten als Vorgänger.
  - Dieser Schritt wird auch als Update oder Relaxieren bezeichnet.

Beispiel zu Dijkstra-Routingverfahren



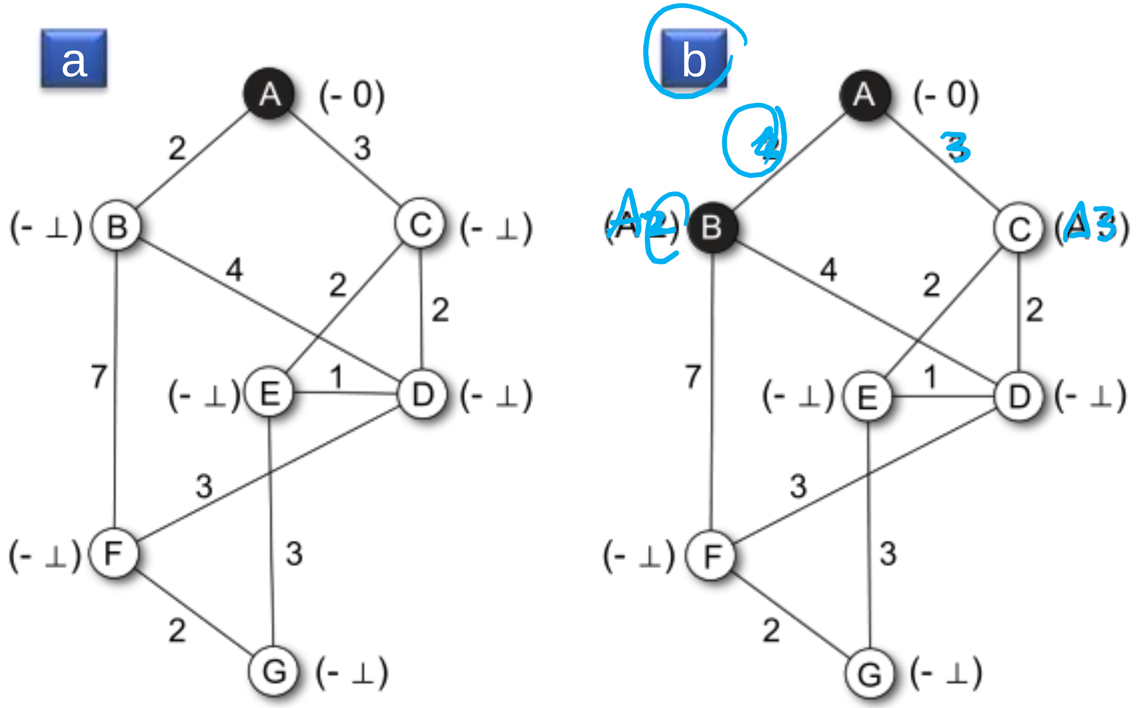
- Es wird ein **kürzester Weg** von A nach G gesucht (single pair, shortest path)
- Zu jedem Knoten wird dazu der jeweilige Vorgängerknoten entlang des kürzesten Weges und die aktuell ermittelte Entfernung dieses Knotens vom Startknoten bereitgestellt.
- Zu Beginn ist für jeden Knoten der Vorgängerknoten unbesetzt und die aktuelle Entfernung wird mit Unendlich angegeben.

Beispiel zu Dijkstra-Routingverfahren



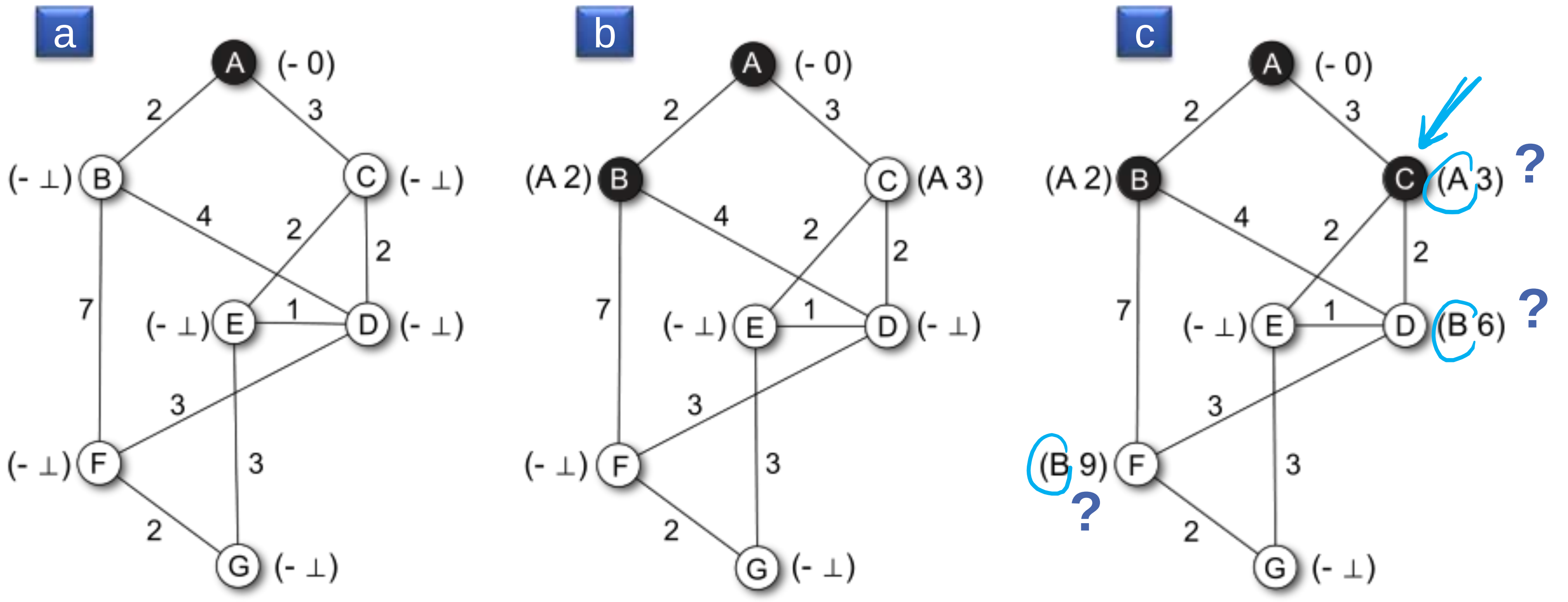
- Wir unterscheiden „**permanente**“ bzw. „besuchte“ Knoten von „noch in der Berechnung“ befindlichen Knoten.
  - Def: Für **permanente** Knoten wurde bereits die kürzeste Entfernung zum Startknoten bestimmt (unveränderlich)
  - Sie können neuer Ausgangspunkt für Folgeberechnungen werden.
  - **Permanente** Knoten sind durch ausgefüllte Kreise kenntlich gemacht.

Beispiel zu Dijkstra-Routingverfahren (2)



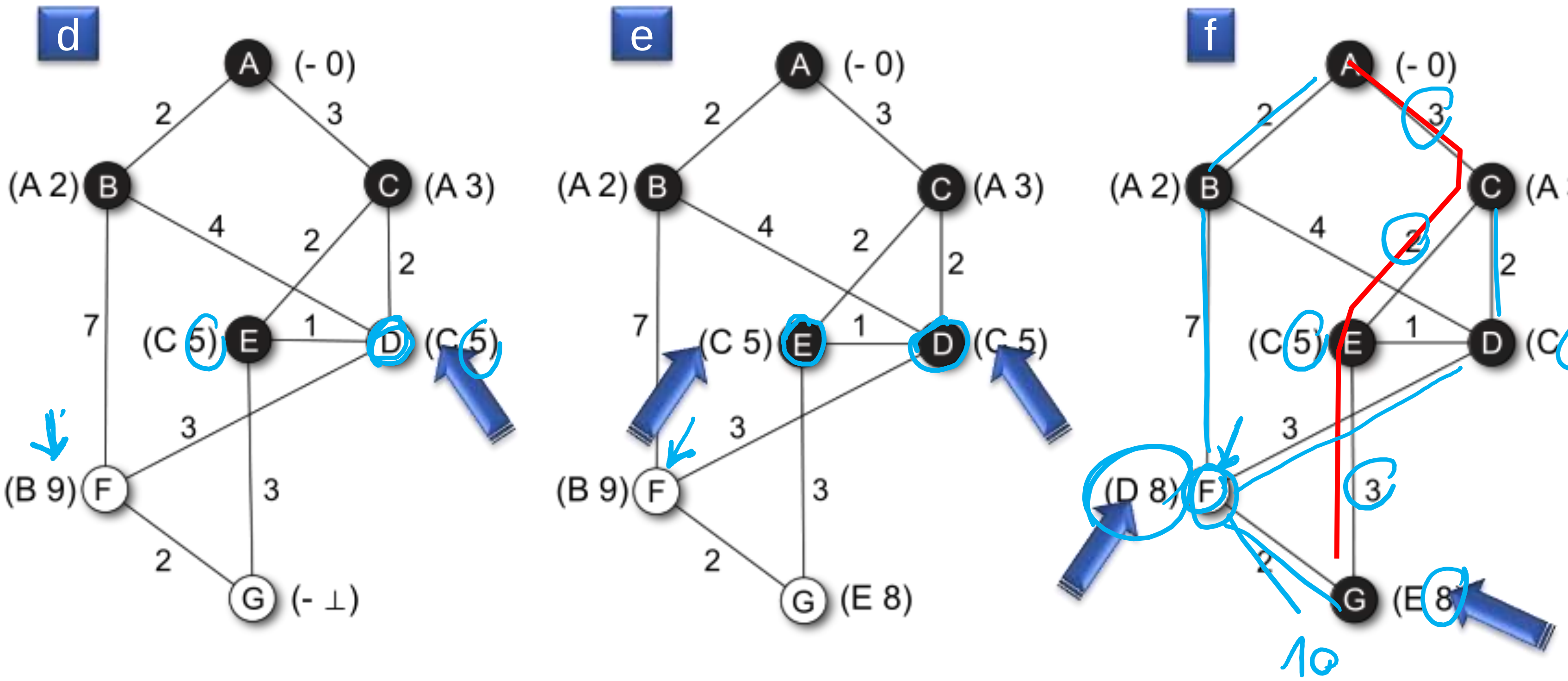
- Zu Beginn der Berechnung wird der Startknoten A als permanent markiert (a).
- Es werden dann die zu A benachbarten Knoten B und C betrachtet,
- Da diese noch nicht als permanent markiert sind, erhalten sie die Belegung (A,2) für B und (A,3) für C entsprechend der (gewichteten) Entfernung von A.
- Jetzt werden alle nicht permanenten Knoten des Graphs untersucht und derjenige Knoten mit der kleinsten Belegung, hier B, wird zum neuen Ausgangsknoten gewählt und als permanent markiert (b).

Beispiel zu Dijkstra-Routingverfahren (2)



- Dann werden die zu B benachbarten, nicht permanenten Knoten D und F betrachtet. Entsprechend ihrer Distanz von A, die sich ergibt aus der Distanz vom Ausgangsknoten B plus dessen Belegung, erhalten die Knoten die Belegungen (B,9) für F und (B,6) für D.
- Zur Ermittlung des nächsten Ausgangsknotens für den kürzesten Weg werden wieder alle nicht permanenten Knoten betrachtet und derjenige mit der kürzesten Distanz zu A, jetzt C, herausgenommen und als permanent markiert (c).
- Dann werden die nicht permanenten Nachbarn von C betrachtet (D und E) und deren Distanz von A entlang der Route über C ermittelt.

Beispiel zu Dijkstra-Routingverfahren (2)

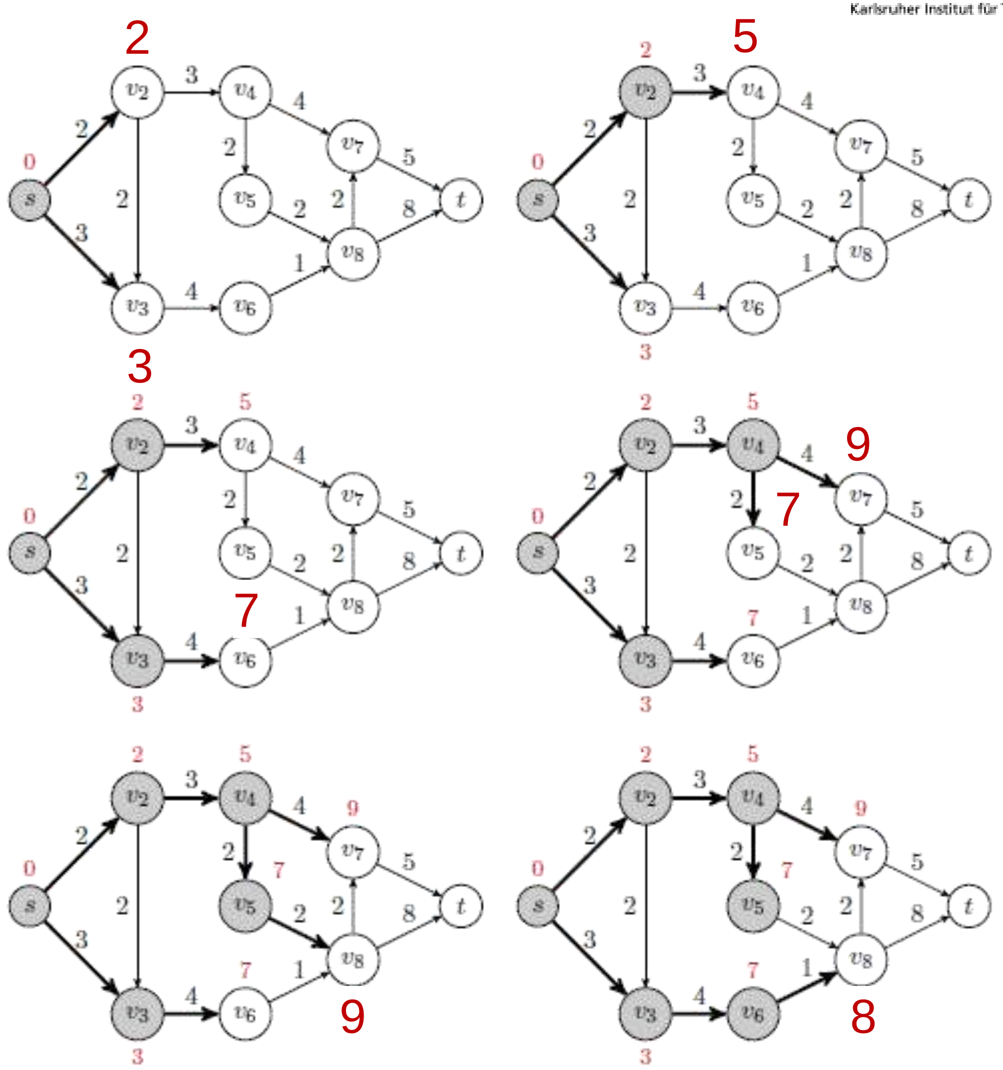


- Da die Route von A nach D über C kürzer ist, als die bisherige, wird die Belegung von D auf (C,5) gesetzt (d) → Relaxation!
- Haben, wie jetzt E und D, zwei oder mehrere nicht permanenten Knoten dieselbe Distanz zur Quelle, so wird **zufällig** einer ausgewählt, als permanent markiert und zum Ausgangspunkt für die nächste Runde gewählt.
- Auf diese Weise wird solange fortgefahren, bis der eigentliche Zielpunkt (G) als permanent markiert worden ist.
- Die Entfernung vom Startpunkt A ergibt sich dann aus der Belegung von G (8), und der kürzeste Weg kann in entgegengesetzter Richtung von G nach A, der Belegung der jeweiligen Knoten folgend, rekonstruiert werden → G,E,C,A

Beispiel

Zum Selbststudium

- Gesucht: kürzester s-t-Weg
- Wir schreiben hier die Distanz(v) jeweils außen an den Knoten und kennzeichnen die Menge aller bereits bearbeiteten Knoten durch Einfärben der Knoten.
- Außerdem kennzeichnen wir jeweils die Kante, von der aus ein Knoten sein aktuelles Label erhalten hat, um am Ende den kürzesten Weg rekonstruieren zu können.
- Es ergeben sich dann die folgenden Berechnungsschritte

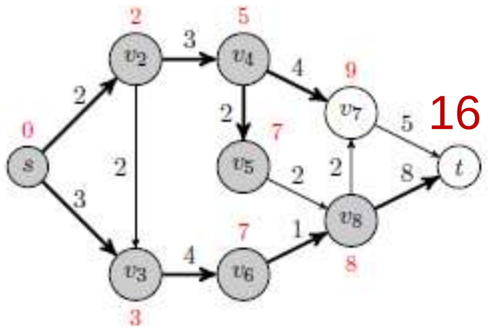


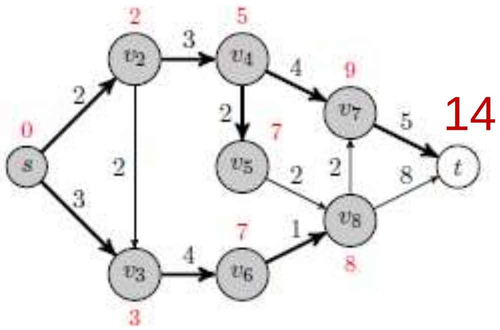
Beispiel

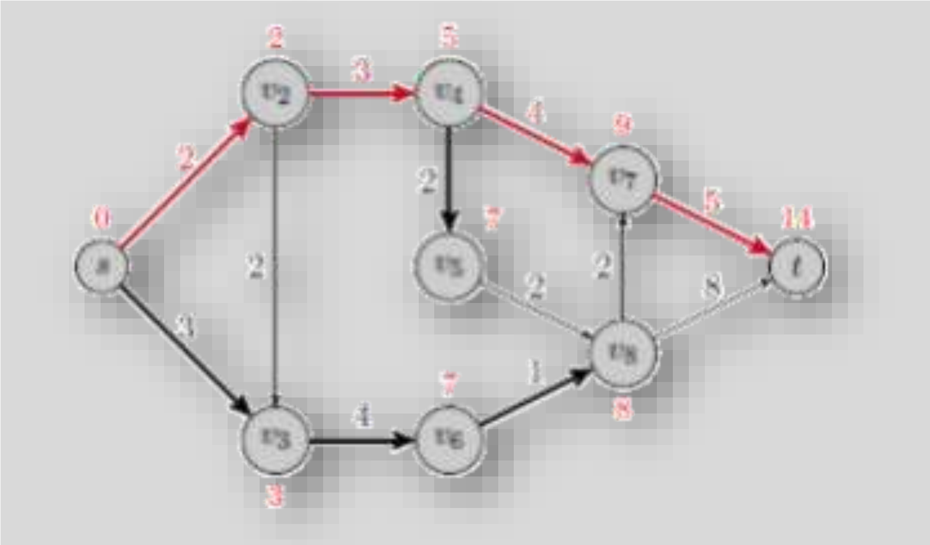
Zum Selbststudium

Karlsruher Institut für Technologie

- Gesucht: kürzester s-t-Weg
- Wir schreiben hier die Distanz(v) jeweils außen an den Knoten und kennzeichnen die Menge aller bereits bearbeiteten Knoten durch Einfärben der Knoten.
- Außerdem kennzeichnen wir jeweils die Kante, von der aus ein Knoten sein aktuelles Label erhalten hat, um am Ende den kürzesten Weg rekonstruieren zu können.
- Es ergeben sich dann die folgenden Berechnungsschritte



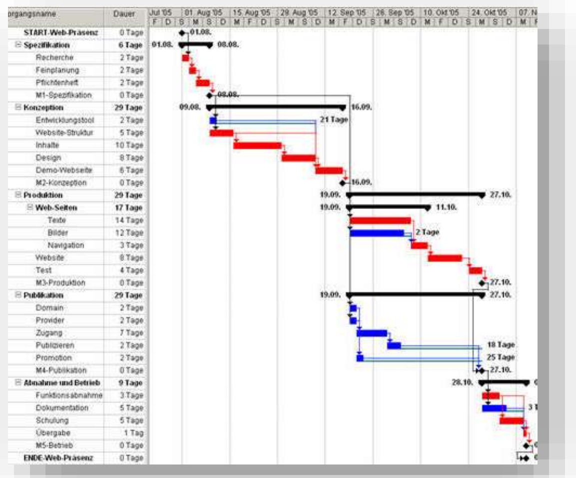




Kritische Pfad Methode

Critical Path Method (CPM)

- Grundlagen
  - Bei CPM-Plänen werden die Vorgänge als gerichtete Kanten und die Ereignisse/Ergebnisse als Knoten dargestellt.
  - Sie werden speziell in Netzplänen bzw. Gantt-Charts eingesetzt
  - Voraussetzung ist, dass alle Vorgänge des Projekts mit der jeweiligen individuellen Dauer richtig zueinander in Beziehung gesetzt werden.
- Mehrwert und Ziele
  - Zwang zum exakten Durchdenken des Projektablaufs
  - übersichtliche Darstellung der Abhängigkeiten
  - Möglichkeit der Minimierung der Projektdauer
  - höhere Sicherheit in der Termineinhaltung
  - deutliche Hervorhebung von Projektengpässen
  - Rechtzeitiges Erkennen möglicher Verzögerungen
    → Abschätzen von Konsequenzen

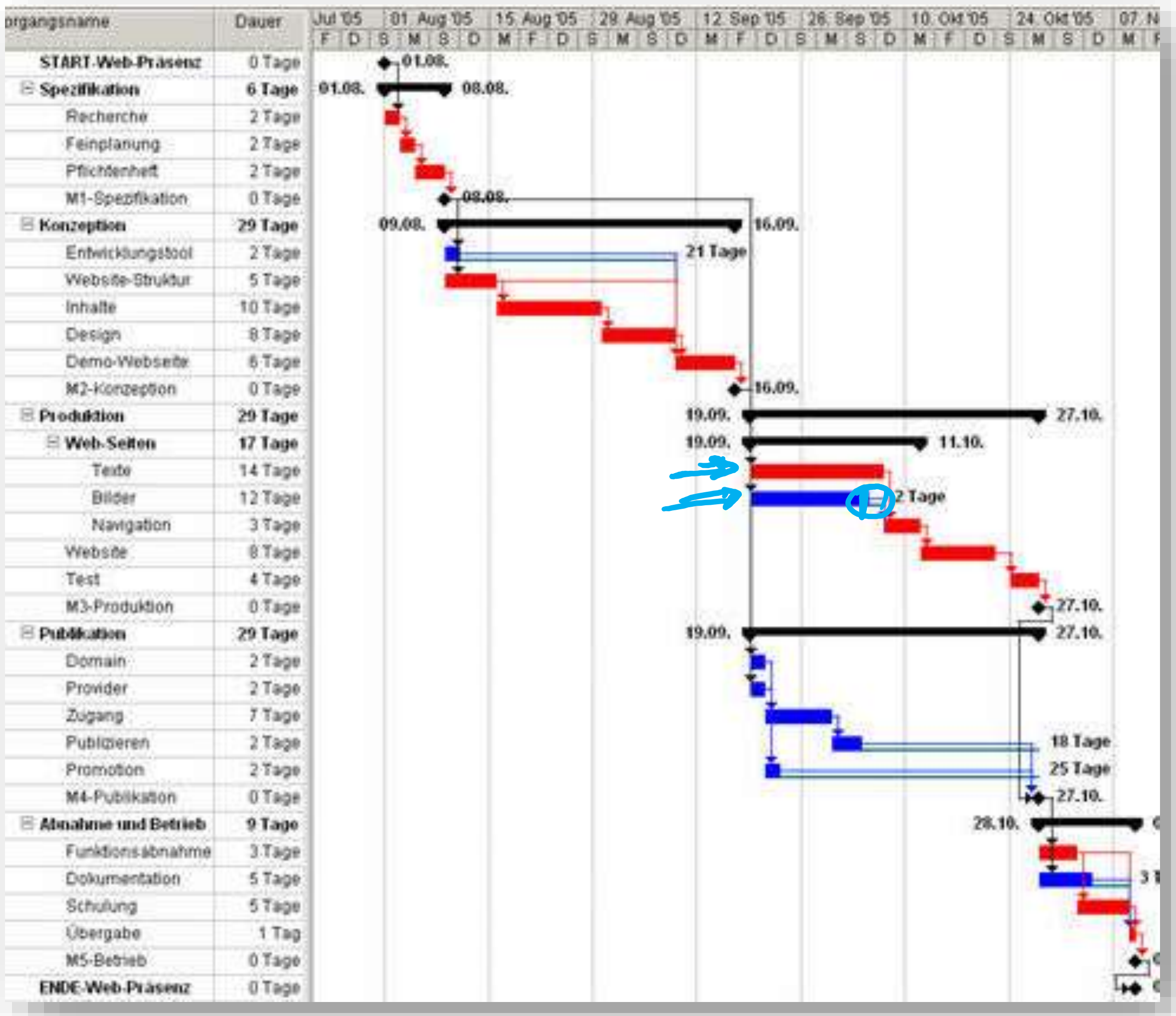


Kritische Pfad Methode (CPM)

(s. IT I → Projektmanagement)

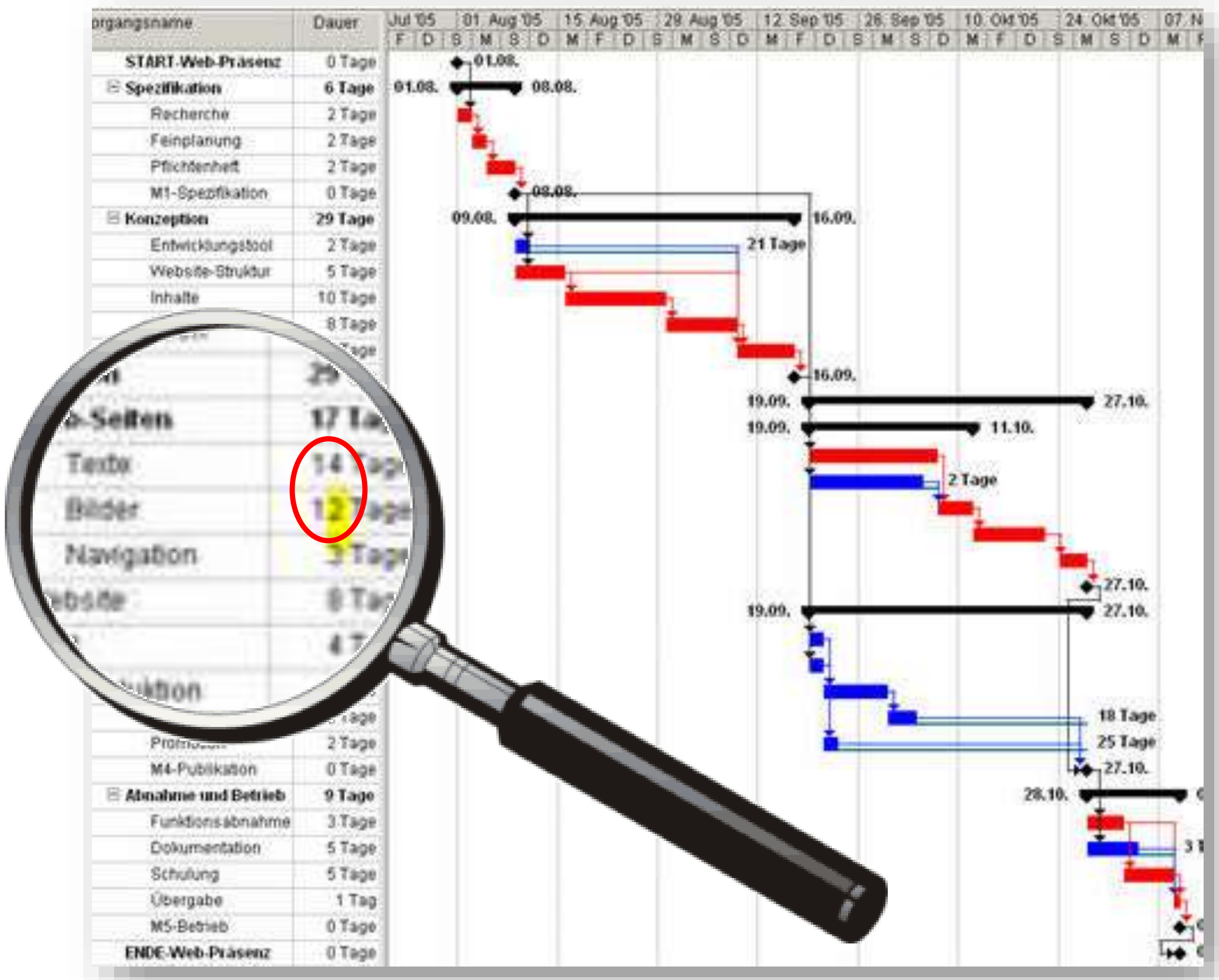
- Herausforderung: Kritischer Pfad in Gantt-Chart
  - Hier: Internet-Auftritt erstellen

- *Remark: Im Gegensatz zur Pfadsuche muss jeder Pfad gegangen werden*



Beispiel: Internet-Auftritt

- Die rot hervorgehobenen Vorgänge bilden den kritischen Weg.
  - Wenn hier was schiefgeht, ist der Endtermin des Gesamtprojektes gefährdet.
- Die blauen Vorgänge dagegen sind unkritisch
  - sie haben keinen Einfluss auf das Projektende.

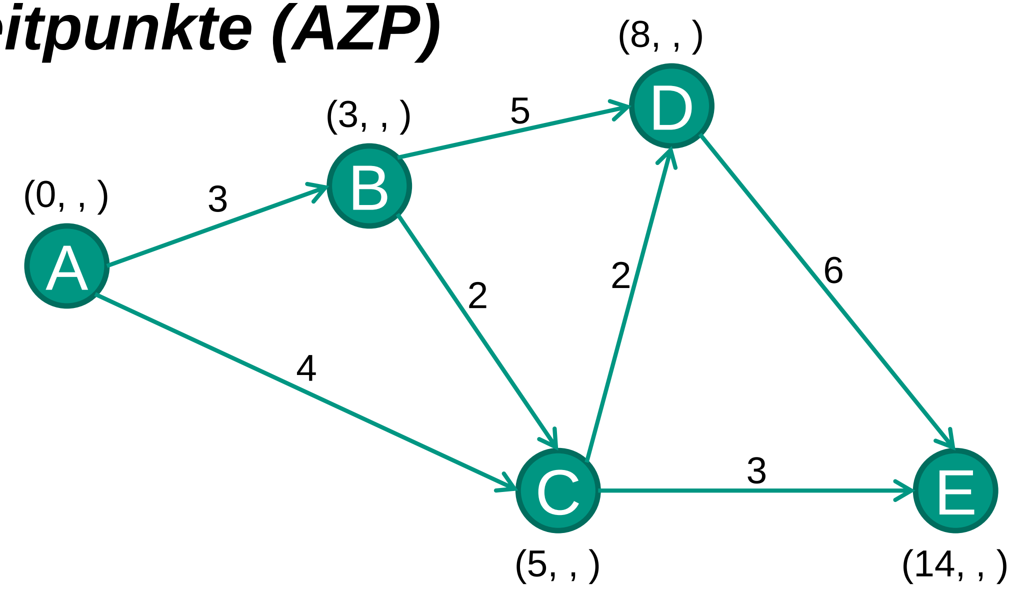


Berechnung des kritischen Pfades in 2 Schritten

- Vorwärtsterminierung
  - Ausgehend vom geplanten frühesten Startzeitpunkt des Projekts wird nacheinander für alle Vorgänge in die Zukunft hinein gerechnet.
  - Damit ergeben sich die frühesten Anfangszeitpunkte aller Vorgänge.
- Rückwärtsterminierung
  - Umgekehrtes Vorgehen:
    Vom Zeitpunkt des geplanten Projektendes wird nun in Richtung der Gegenwart gerechnet.
  - Es ergeben sich die spätesten Endzeitpunkte der einzelnen Vorgänge.
- Besteht für einen Vorgang zwischen dem Ergebnis der Vor- und der Rückwärtsrechnung eine Differenz, so liegt eine Pufferzeit vor.
  - Ein Vorgang mit Pufferzeit kann zwischen dem frühesten Anfangstermin und seinem spätesten Endtermin um diese Pufferzeit verschoben werden, ohne dass dadurch die Projektdauer negativ beeinflusst wird.
- Ein Weg durch den Netzplan vom Start- zum Zielknoten, bei dem kein Ereignis Pufferzeiten hat, wird als kritischer Pfad bezeichnet.
  - Eine Verzögerung bei einem der Ereignisse des kritischen Pfades hat immer eine Verzögerung des Endtermins des Projektes zur Folge.

CPM: *Früheste Anfangszeitpunkte (AZP)*

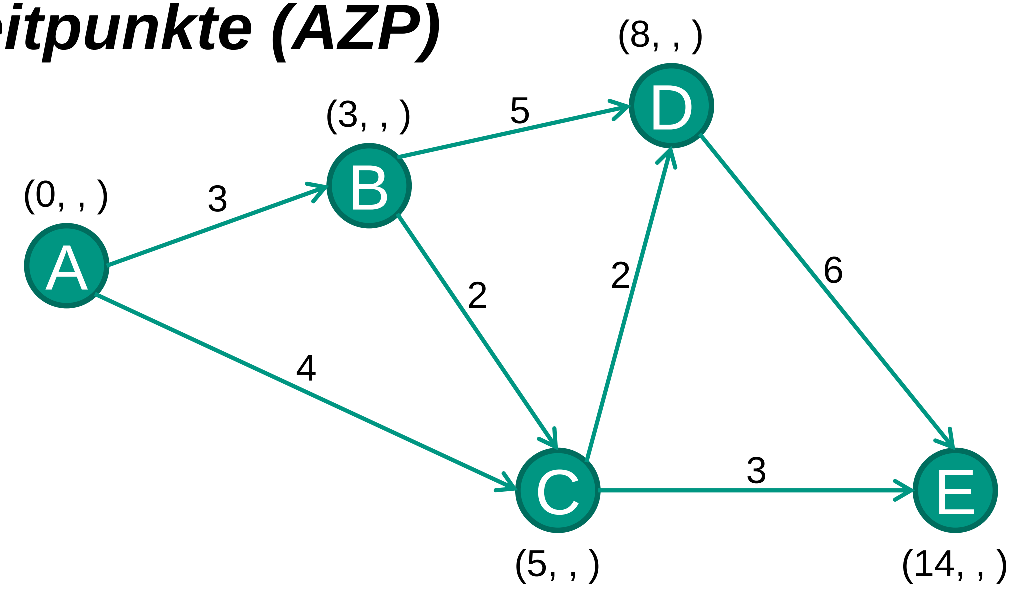
Vorwärtsterminierung



Fort-schritt	Vorgänger	AZP Vorgänger plus Zeit des Arbeitsschrittes	Größte Summe = Frühester Anfangszeitpunkt (eines Knotens)
A			
B			
C			
D			
E			

CPM: *Früheste Anfangszeitpunkte (AZP)*

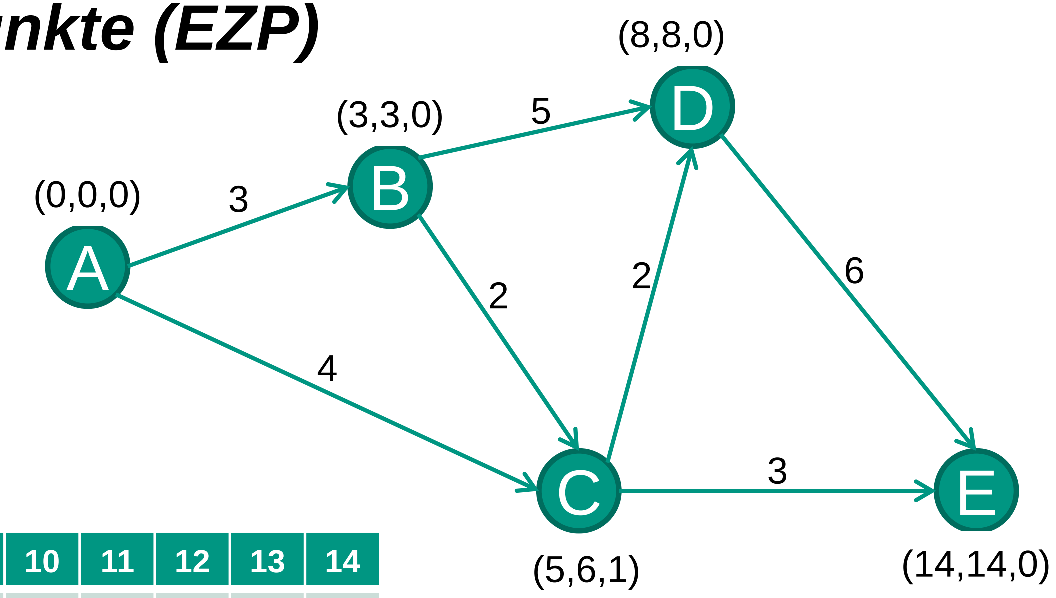
Vorwärtsterminierung



Fort-schritt	Vorgänger	AZP Vorgänger plus Zeit des Arbeitsschrittes	Größte Summe = Frühester Anfangszeitpunkt (eines Knotens)
A	-	0 + 0 = 0	0
B	A	0 + 3 = 3	3
C	A	0 + 4 = 4	
	B	3 + 2 = 5	5
D	B	3 + 5 = 8	8
	C	5 + 2 = 7	
E	C	5 + 3 = 8	
	D	8 + 6 = 14	14

CPM: *Späteste Endzeitpunkte (EZP)*

Rückwärtsterminierung



„Nuller“ zusammenstellen = CP

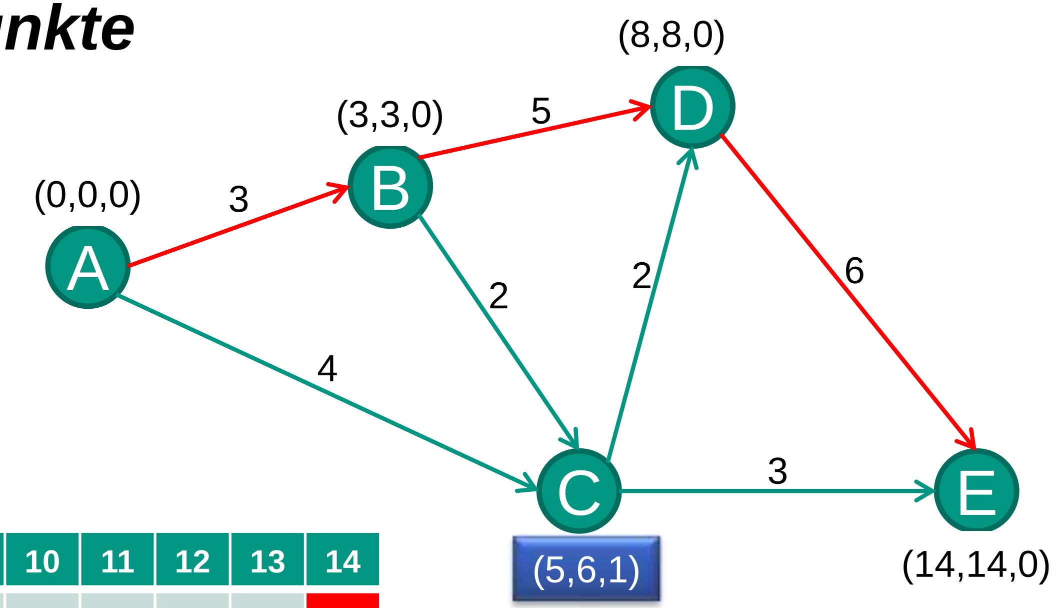
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A			B					D						E

Keine Pufferzeit

Fort-schritt	Nachfolger	EZP für Nachfolger minus Zeit des Arbeitsschrittes	Kleinste Differenz = Spätester Endtermin
E			
D			
C			
B			
A			

CPM: *Späteste Endzeitpunkte*

Rückwärtsterminierung



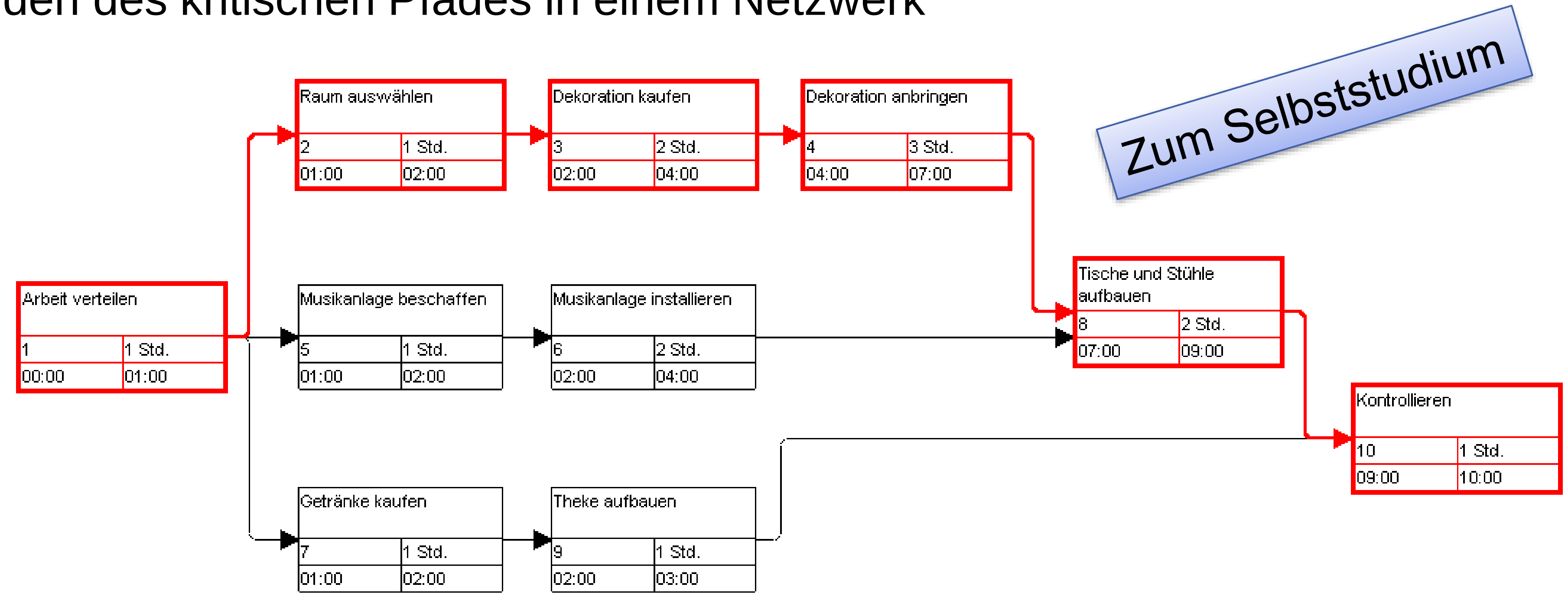
„Puffer" darstellen

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A			B		C	C		D						E

Fort-schritt	Nachfolger	EZP für Nachfolger minus Zeit des Arbeitsschrittes	Kleinste Differenz = Spätester Endtermin
E	-	14 – 0 = 14	14
D	E	14 – 6 = 8	8
C	D	8 – 2 = 6	6
	E	14 – 3 = 11	
B	C	6 – 2 = 4	4
	D	8 – 5 = 3	3
A	B	3 – 3 = 0	0
	C	6 – 4 = 2	2

Beispiel: Betriebsfest organisieren

- Ziel: Finden des kritischen Pfades in einem Netzwerk



- Ein Vorgang kann erst beginnen, wenn alle vorhergehenden Vorgänge (Vorgänger) abgeschlossen sind.
- Müssen mehrere Vorgänge beendet sein, bevor ein folgender Vorgang (Nachfolger) beginnen kann, so enden sie dort.

Übung: Kundenauftrag durchführen

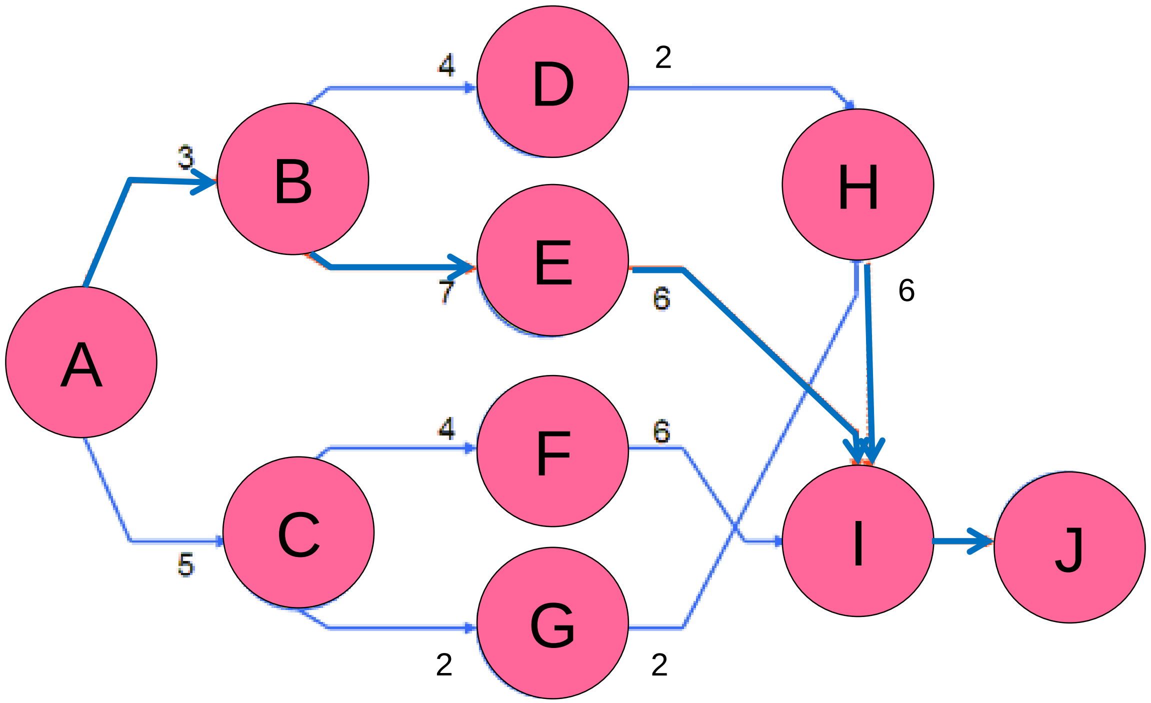
Vorgangsliste *mit* Zeitangaben und Zeitanalyse

Vorgangs-Nr.	Vorgangsbezeichnung	Vorgänger	Nachfolger	Dauer/ Tage
A	Auftragseingang			-
B	Material bestellen			3
C	Arbeitspläne erstellen			5
D	Materialkosten errechnen			4
E	Lieferzeit des Materials			7
F	Arbeitskräfte einweisen			4
G	Lohnkosten ermitteln			2
H	Selbstkosten kalkulieren			2
I	Arbeit ausführen			6
J	Auslieferung			-

Übung: Kundenauftrag durchführen

Vorgangsliste *mit* Zeitangaben und Zeitanalyse

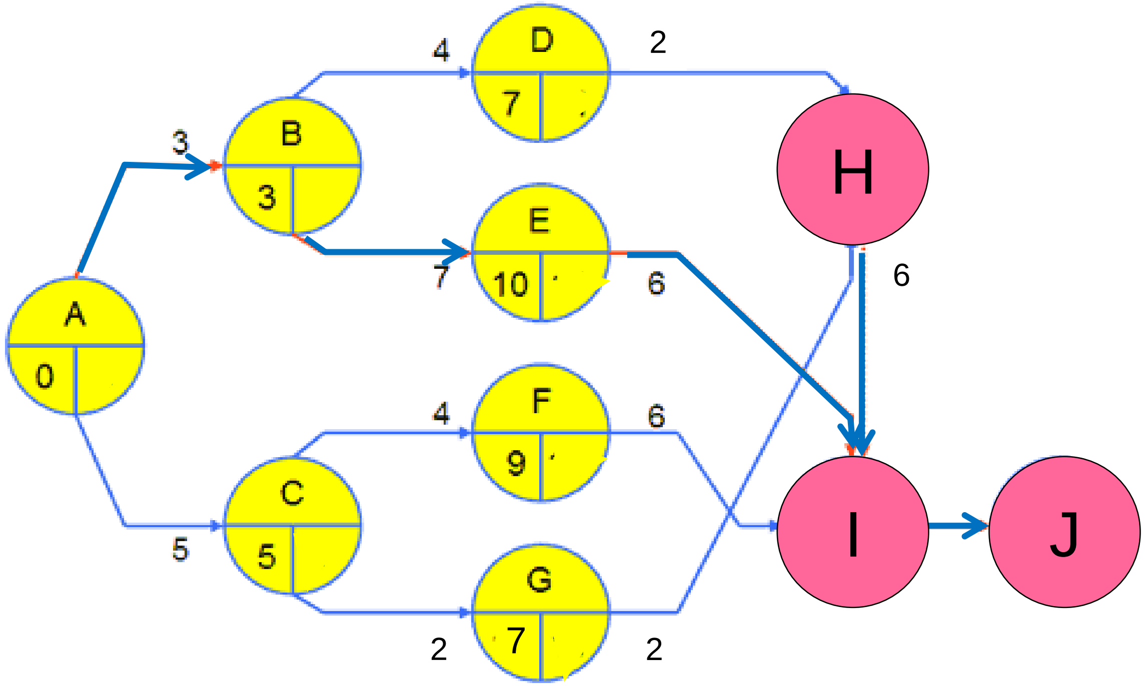
Vorgangs-Nr.	Vorgangsbezeichnung	Vorgänger	Nachfolger	Dauer/ Tage
A	Auftragseingang	-	B, C	-
B	Material bestellen	A	D, E	3
C	Arbeitspläne erstellen	A	F, G	5
D	Materialkosten errechnen	B	H	4
E	Lieferzeit des Materials	B	I	7
F	Arbeitskräfte einweisen	C	I	4
G	Lohnkosten ermitteln	C	H	2
H	Selbstkosten kalkulieren	D, G	I	2
I	Arbeit ausführen	E, F, H	J	6
J	Auslieferung	I	-	-



Übung: Kundenauftrag durchführen

Vorgangsliste *mit* Zeitangaben und Zeitanalyse

Vorgangs-Nr.	Vorgangsbezeichnung	Vorgänger	Nachfolger	Dauer/Tage
A	Auftragseingang	-	B, C	-
B	Material bestellen	A	D, E	3
C	Arbeitspläne erstellen	A	F, G	5
D	Materialkosten errechnen	B	H	4
E	Lieferzeit des Materials	B	I	7
F	Arbeitskräfte einweisen	C	I	4
G	Lohnkosten ermitteln	C	H	2
H	Selbstkosten kalkulieren	D, G	I	2
I	Arbeit ausführen	E, F, H	J	6
J	Auslieferung	I	-	-

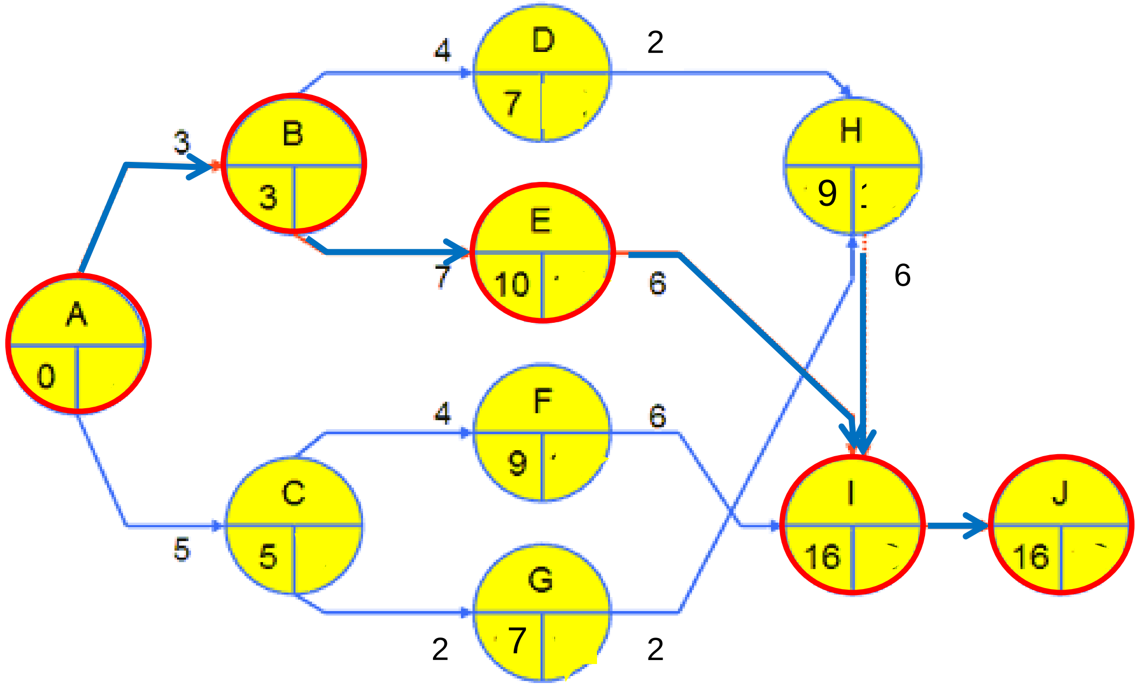


Übung: Kundenauftrag durchführen

Karlsruher Institut für Technologie

Vorgangsliste *mit* Zeitangaben und Zeitanalyse

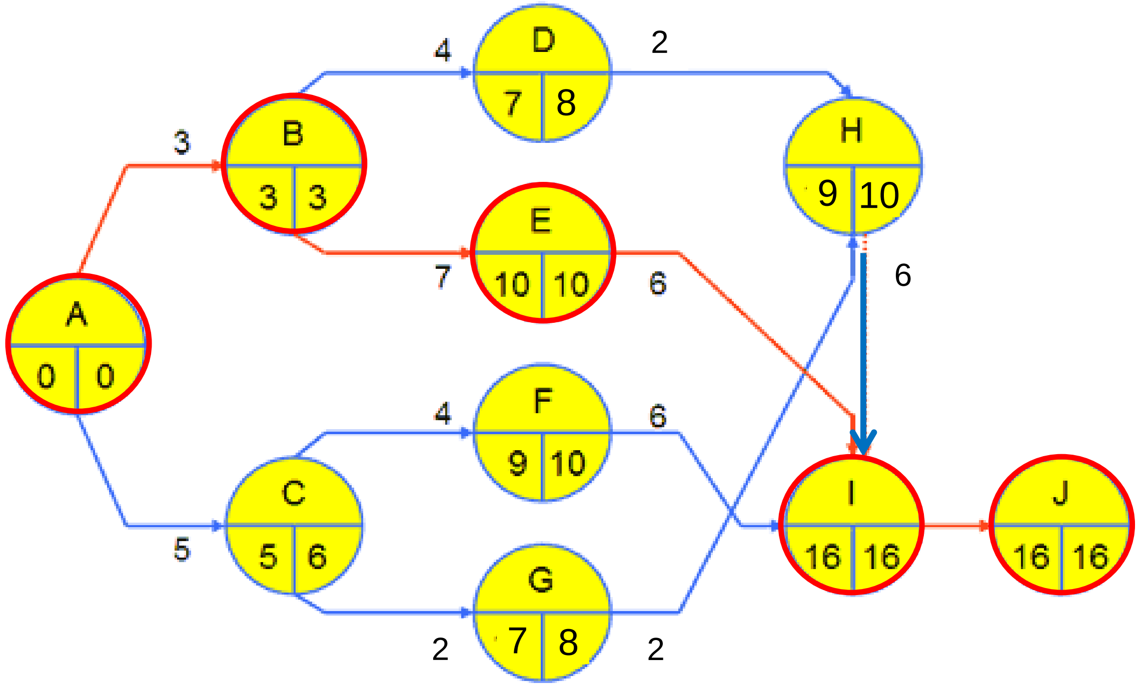
Vorgangs-Nr.	Vorgangsbezeichnung	Vorgänger	Nachfolger	Dauer/Tage
A	Auftragseingang	-	B, C	-
B	Material bestellen	A	D, E	3
C	Arbeitspläne erstellen	A	F, G	5
D	Materialkosten errechnen	B	H	4
E	Lieferzeit des Materials	B	I	7
F	Arbeitskräfte einweisen	C	I	4
G	Lohnkosten ermitteln	C	H	2
H	Selbstkosten kalkulieren	D, G	I	2
I	Arbeit ausführen	E, F, H	J	6
J	Auslieferung	I	-	-



Übung: Kundenauftrag durchführen

Vorgangsliste *mit* Zeitangaben und Zeitanalyse

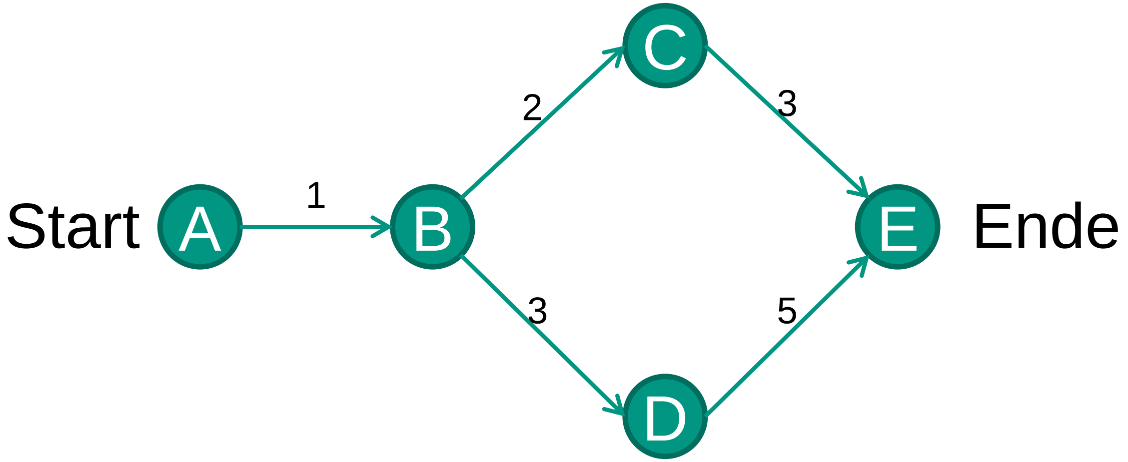
Vorgangs-Nr.	Vorgangsbezeichnung	Vorgänger	Nachfolger	Dauer/ Tage
A	Auftragseingang	-	B, C	-
B	Material bestellen	A	D, E	3
C	Arbeitspläne erstellen	A	F, G	5
D	Materialkosten errechnen	B	H	4
E	Lieferzeit des Materials	B	I	7
F	Arbeitskräfte einweisen	C	I	4
G	Lohnkosten ermitteln	C	H	2
H	Selbstkosten kalkulieren	D, G	I	2
I	Arbeit ausführen	E, F, H	J	6
J	Auslieferung	I	-	-



Übung: CPM

Zum Selbststudium

- Gegeben sei folgender Netzplan:

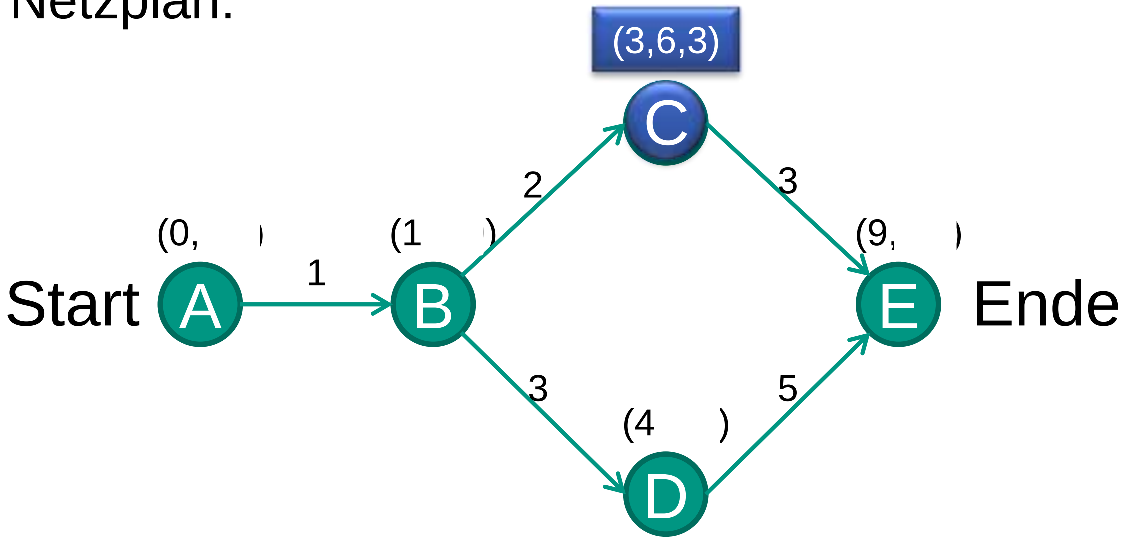


- Ermitteln Sie mit Hilfe der Critical Path Method, welcher Prozess später gestartet werden kann und in welchem Zeitraum er gestartet werden muss.

Übung: CPM – Lsg.

Zum Selbststudium

- Gegeben sei folgender Netzplan:

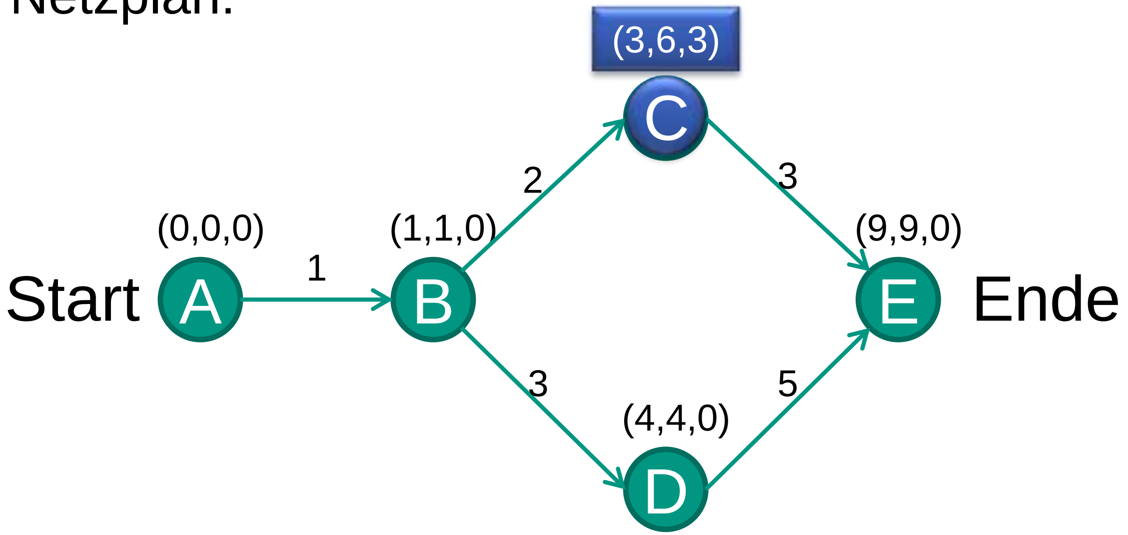


- Ermitteln Sie mit Hilfe der Critical Path Method, welcher Prozess später gestartet werden kann und in welchem Zeitraum er gestartet werden muss.

Übung: CPM – Lsg.

Zum Selbststudium

- Gegeben sei folgender Netzplan:



- Prozess C kann zwischen der 3. und 6. Woche gestartet werden.

- Kürzester Pfad
- Dijkstra-Algorithmus
- Critical Path Method


Backup
Shift

Ablauf: Dijkstra-Algorithmus