Fachhochschule
Dortmund
University of Applied Sciences and Arts
Fachbereich Informatik

Algorithmen und Datenstrukturen

VL12 - Graphen 1

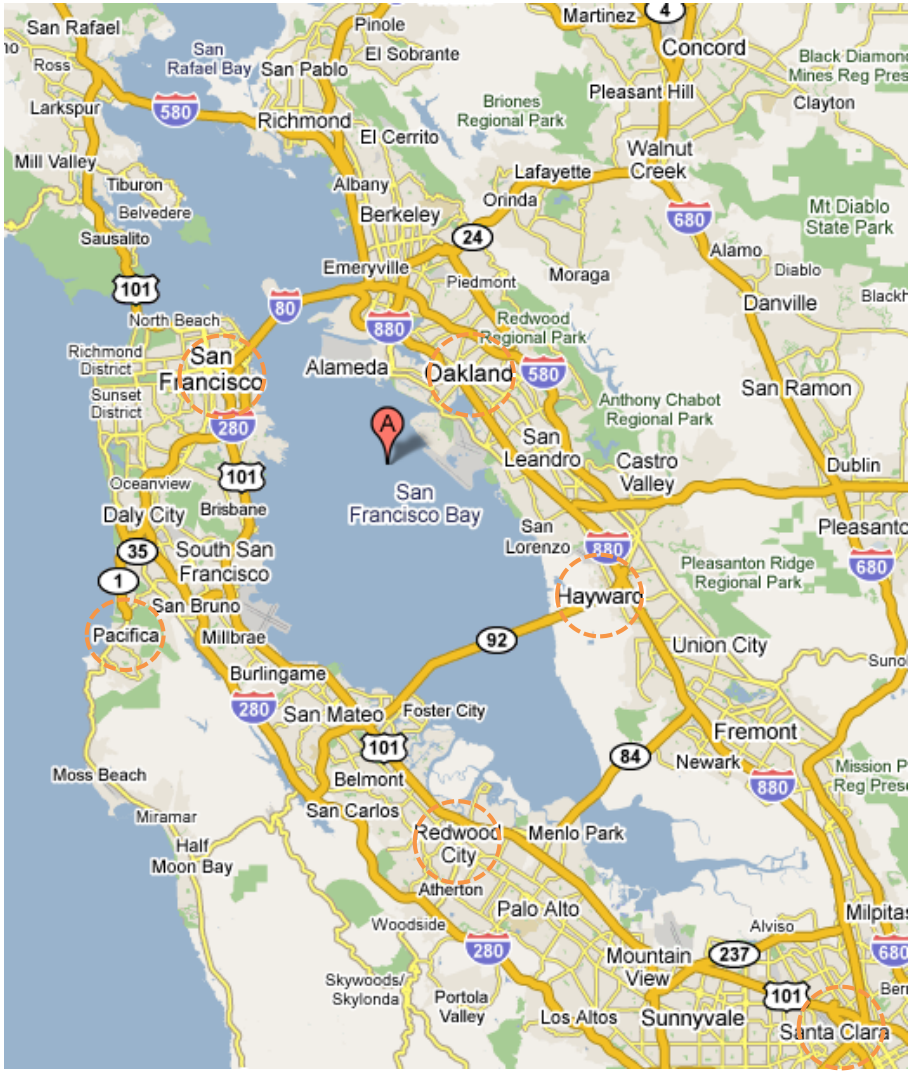
Inhalt

- Graphen
  - Einführung
  - Ungerichtete Graphen
  - Gerichtete Graphen
- Datenstrukturen für Graphen
- Traversierungen
- Topologisches Sortieren

EINFÜHRUNG

Motivation

Wozu Graphen?

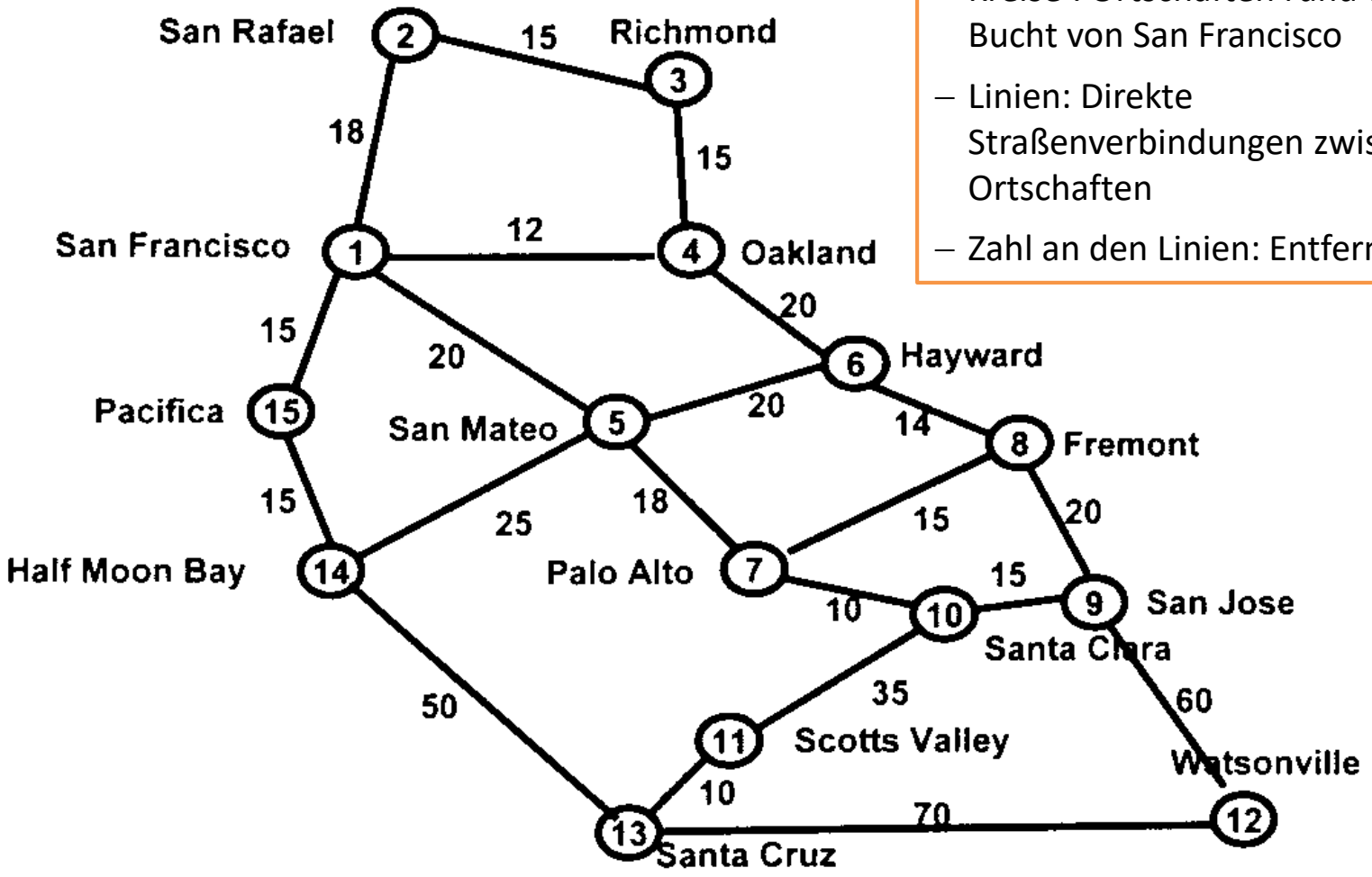


- Ortschaften rund um Bucht von San Francisco
- Gibt es eine Straßenverbindung zwischen Pacifica und Oakland?
- Wie kommt man am schnellsten von San Francisco nach Santa Clara?
- Welches ist die kürzeste Strecke zwischen Redwood City und Hayward?

Quelle: Google

Motivation

Graph für Verkehrsverbindungen



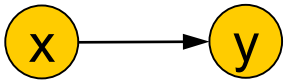
- Kreise : Ortschaften rund um Bucht von San Francisco
- Linien: Direkte Straßenverbindungen zwischen Ortschaften
- Zahl an den Linien: Entfernung

Graphen allgemein

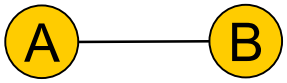
Definition Graph (umgangssprachlich)

Ein **Graph** besteht aus einer Menge von **Knoten (V)** und einer Menge von **Kanten (E)**

- Knoten repräsentieren Objekte
- Kanten sind Verbindungen zwischen zwei Knoten und repräsentieren die Beziehung zwischen Objekten
- Um eine unsymmetrische Beziehung wie "x ist Kind von y" auszudrücken, verwendet man gerichtete Kanten



- Um eine symmetrische Beziehung wie "A und B sind über eine Autobahn verbunden" auszudrücken, verwendet man ungerichtete Kanten



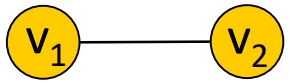
Ungerichtete Graphen

Ungerichtete Graphen

Definition ungerichteter Graph (mathematisch)

Ein **ungerichteter Graph** ist ein Tupel G = (V,E), wobei V die Menge der Knoten und E die Menge der ungerichteten Kanten bezeichnet

- Hierbei enthält jede Kante e in E ein oder zwei Knoten aus V, d.h. $E \subseteq \{ \{ v_1, v_2 \} \mid v_1, v_2 \in V \}$



Ungerichtete Graphen

Beispiel: Graph, Knotenmenge, Kantenmenge

Beispiele:

- $G_1 = (V_1, E_1)$
  V_1 = Städte in Deutschland
  $E_1 = \{ \{x,y\} \mid x, y \in V_1$, es gibt eine Autobahn zwischen x und y$\}$
  (Relation ist symmetrisch ⇒ ungerichteter Graph)

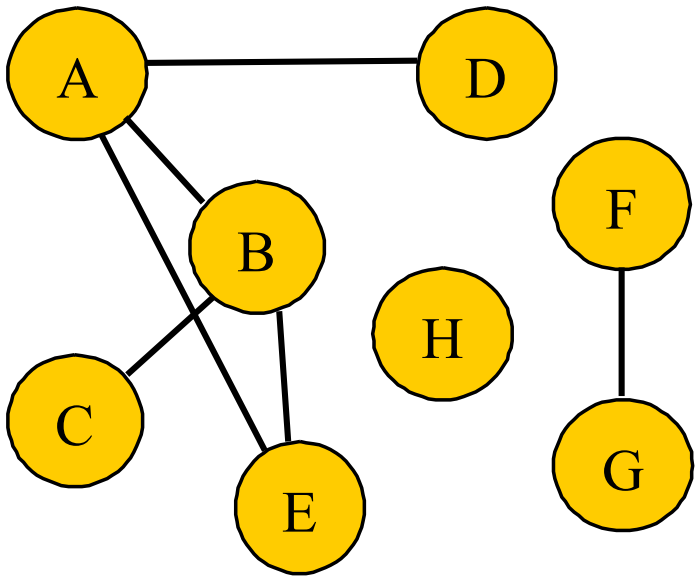
- $G_2 = (V_2, E_2)$
  V_2 = Spieler eines Tennisturniers
  $E_2 = \{ \{x,y\} \mid x, y \in V_2$, x spielt gegen y$\}$
  (Relation ist symmetrisch ⇒ ungerichteter Graph)

Ungerichtete Graphen

Zeichnen von ungerichteten Graphen

Sei G = (V, E) ein ungerichteter Graph. Die Elemente von V (Knoten) werden üblicherweise als Kreise dargestellt. Die Elemente {x, y} ∈ E (Kanten) werden als Linien zwischen x und y dargestellt

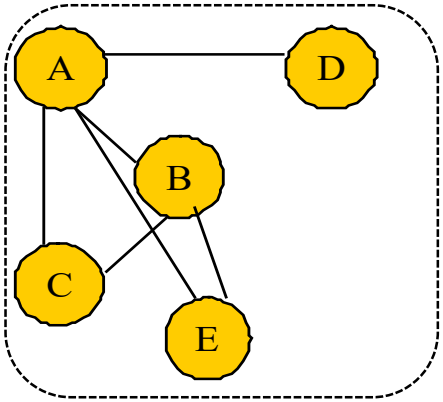
- V = {A, B, C, D, E, F, G, H}
- E = { {A,D}, {A,B}, {B,C}, {B,E}, {A,E}, {F,G} }



Ungerichtete Graphen

Weg und Zyklus bei ungerichteten Graphen

- Ein **Weg** von x nach y ist eine Folge $x = v_1, v_2, \ldots, v_n = y$ von Knoten, in der es jeweils Kanten zwischen v_1 und v_2, v_2 und v_3 usw. bis v_n gibt, d.h. $\{v_i, v_{i+1}\} \in E$ für $1 \leq i < n$
- Auf einem **einfachen Weg** kommt kein Knoten doppelt vor.
- Ein Weg von x nach x, **der mindestens 3 verschiedene Knoten enthält** und in dem bis auf x kein anderer Knoten doppelt vorkommt, heißt **Zyklus**.



B, C, A, B, E	ist ein Weg von B nach E. Er enthält den Zyklus B, C, A, B
C, A, B, E	ist ein einfacher Weg von C nach E
A, D	ist ein Weg, aber **kein** Zyklus
A, B, C, A	ist ein Zyklus
A, B, E, C	ist kein Weg

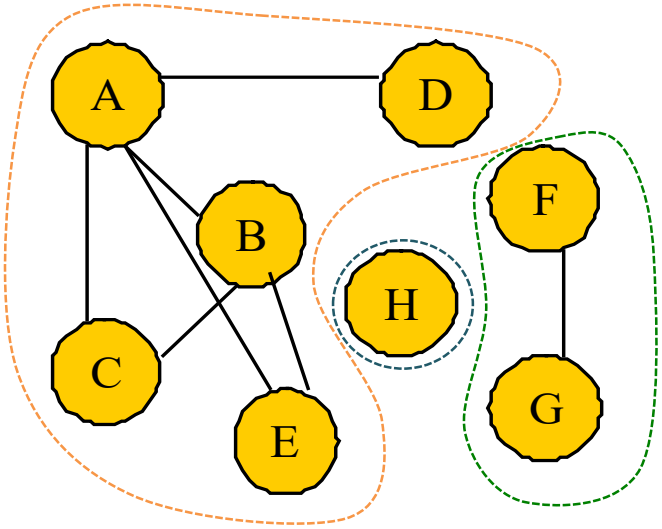
Ungerichtete Graphen

Zusammenhängender, ungerichteter Graph

- Ein ungerichteter Graph G heißt **zusammenhängend**, wenn es zwischen je zwei (verschiedenen) Knoten einen Weg gibt.
- Ist G nicht zusammenhängend, so zerfällt er in eine Vereinigung zusammenhängender Komponenten

- Die zusammenhängenden Komponenten werden auch **Zusammenhangskomponenten** des Graphen genannt

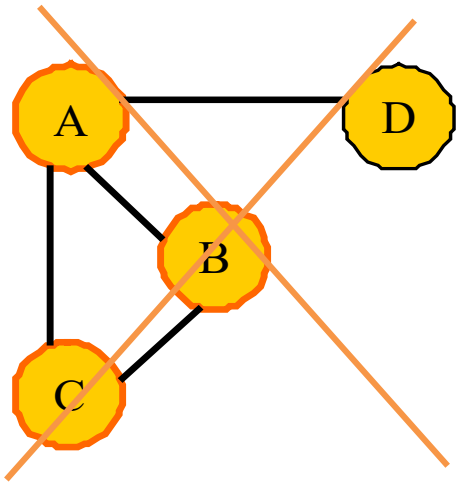
- Der Graph unten besteht aus drei Zusammenhangskomponenten



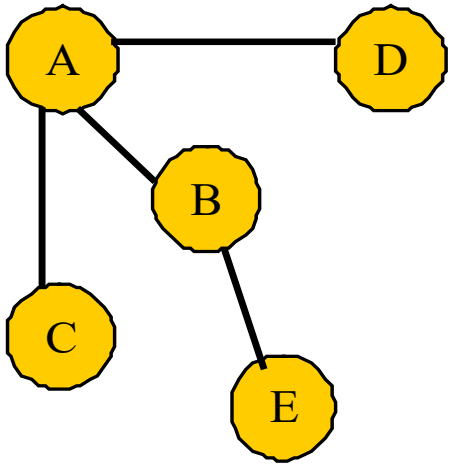
Ungerichtete Graphen

Baum bei ungerichteten Graphen

- Ein ungerichterer Graph heißt **Baum**, wenn er zyklenfrei und zusammenhängend ist.



kein Baum



Baum

Gerichtete Graphen

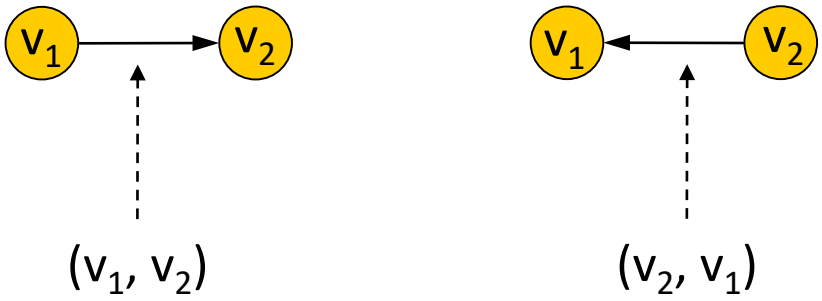
Gerichtete Graphen

Definition gerichteter Graph (mathematisch)

Ein **gerichteter Graph G** ist ein Tupel G=(V, E), wobei V die Menge der Knoten bezeichnet und E die Menge der Kanten

- Dabei ist E eine zweistellige Relation auf der Knotenmenge V, d.h. $E \subseteq V \times V$. Ein Knoten v_1 steht in der Relation E zu einem Knoten v_2, d.h. $(v_1, v_2) \in E$, falls es eine Kante von $v_1 \in V$ nach $v_2 \in V$ gibt

- Die Richtung einer Kante (v_1, v_2) ist durch die Position der Knoten im Tupel vorgegeben. Die Kante beginnt in v_1 und endet in v_2

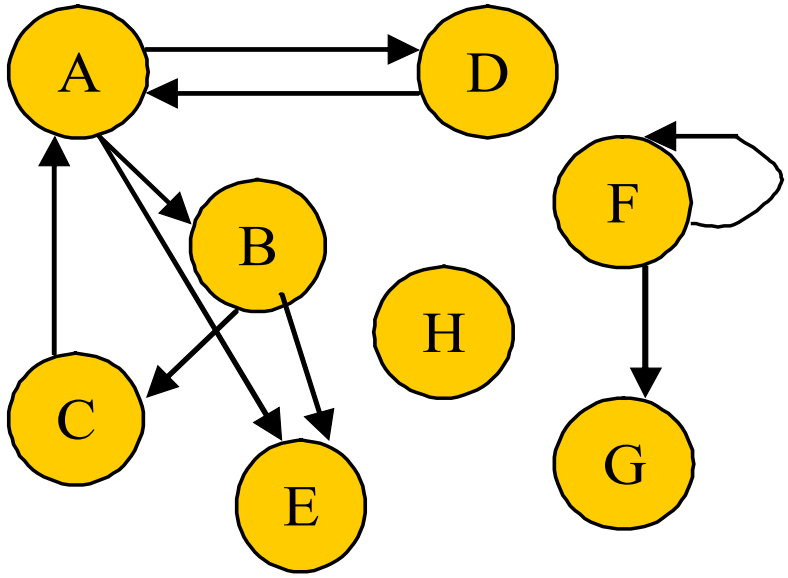


Gerichtete Graphen

Zeichnen von gerichteten Graphen

Sei G = (V, E) ein gerichteter Graph. Die Elemente von V (Knoten) werden üblicherweise als Kreise dargestellt. Die Elemente (x, y) $\in$ E (Kanten) werden als Pfeile von x nach y dargestellt

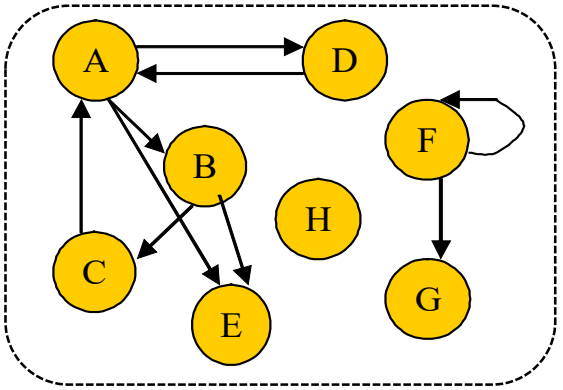
- V = {A, B, C, D, E, F, G, H}
- E = {(A,D), (D,A), (A,B), (B,C), (C,A), (B,E), (A,E), (F,G), (F,F)}



Gerichtete Graphen

Weg und Zyklus bei gerichteten Graphen

- Ein **Weg** von x nach y ist eine Folge $x = v_1, v_2, \ldots, v_n = y$ von Knoten, in der es jeweils Kanten von v_1 nach v_2, nach v_3 usw. bis nach v_n gibt, d.h. $(v_i, v_{i+1}) \in E$ für $1 \leq i < n$
- Auf einem **einfachen Weg** kommt kein Knoten doppelt vor.
- Ein Weg von x nach x, in dem bis auf x kein anderer Knoten doppelt vorkommt, heißt **Zyklus**.



Beispiel:

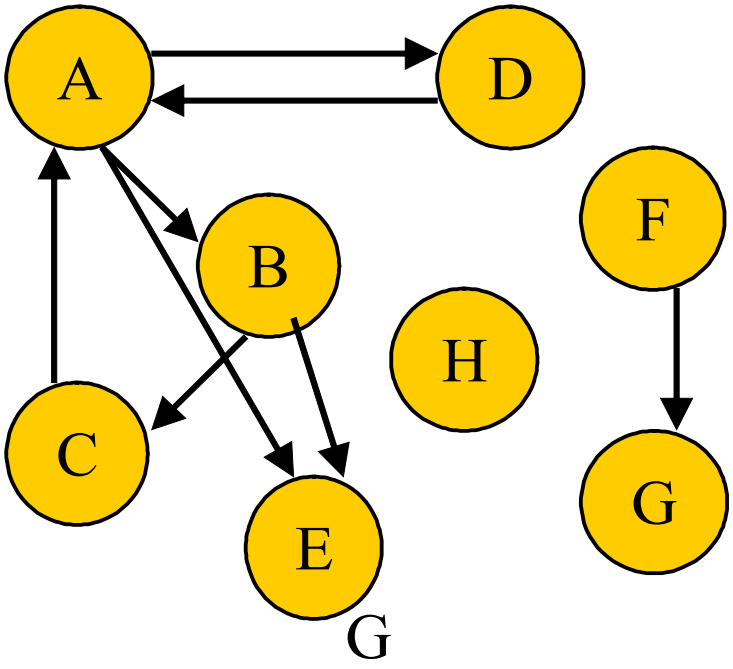
B, C, A, D, A	ist ein Weg von B nach A. Er enthält den Zyklus A, D, A
C, A, B, E	ist ein einfacher Weg von C nach E
F, F, F, G	ist ein Weg (aber kein einfacher)
A, B, C, A	ist ein Zyklus (ebenso A, D, A und F, F)
A, B, E, A	ist kein Weg und kein Zyklus

Gerichtete Graphen

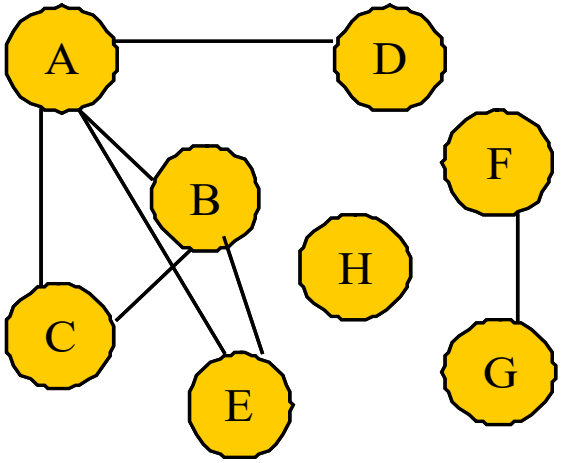
Gerichtete und ungerichtete Graphen

Jedem gerichteten Graph G = (V, E) kann man einen ungerichteten Graphen G' = (V,E') zuordnen, den **G zu Grunde liegenden ungerichteten Graphen**: E' enthält alle Kanten aus E, ohne die Richtung zu beachten,

$$E' = \{ \{u,v\} \mid (u,v) \in E \}$$



G



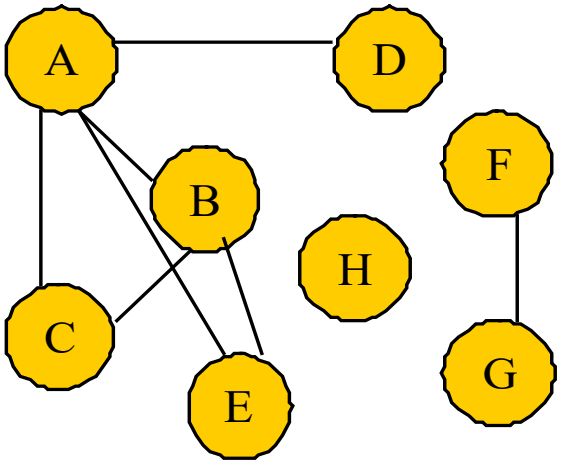
G'

Gerichtete Graphen

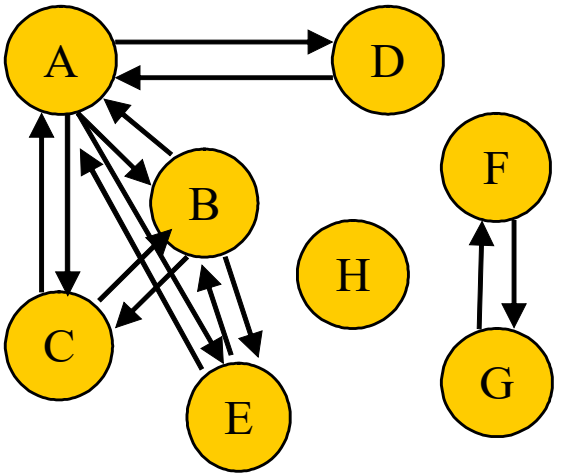
Gerichtete und ungerichtete Graphen

Jeden ungerichteten Graph G = (V, E) kann man mit einem gerichteten Graphen G' = (V,E') identifizieren, dem **G zugeordneten gerichteten Graph G'**: für jede Kante e in E zwischen Knoten x und y enthält E' die Kanten von x nach y und y nach x, d.h.

$$E' = \{ (u,v), (v,u) \mid \{u,v\} \in E \}$$



G



G'

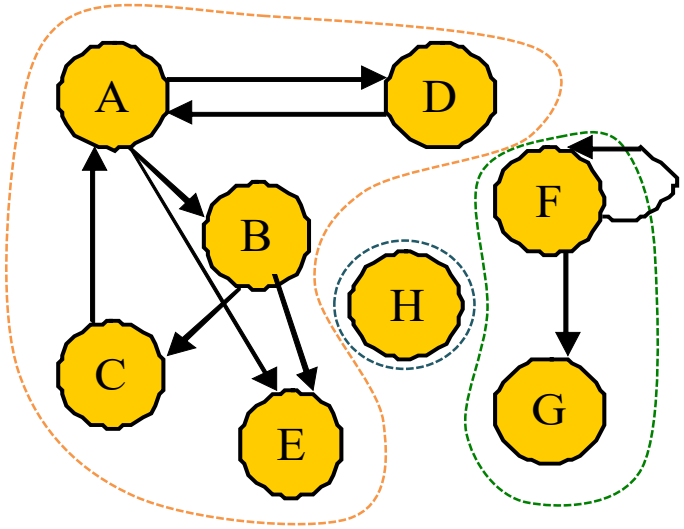
Gerichtete Graphen

Zusammenhängender, gerichteter Graph

- Ein gerichteter Graph G heißt **zusammenhängend**, wenn der G zu Grunde liegende ungerichtete Graph zusammenhängend ist, d.h. wenn es zwischen je zwei (verschiedenen) Knoten einen ungerichteten Weg gibt
  - **In diesem Kontext wird bei der Bestimmung eines Weges die Richtung der Kanten ausnahmsweise außer Acht gelassen!**
- Ist G nicht zusammenhängend, zerfällt er in eine Vereinigung zusammenhängender Komponenten

- Die zusammenhängenden Komponenten werden auch **Zusammenhangskomponenten** des Graphen genannt

- Der Graph rechts besteht aus drei Zusammenhangskomponenten



Gerichtete Graphen

Zusammenhängender, gerichteter Graph

- Ein gerichteter Graph G heißt **stark zusammenhängend**, wenn es zwischen je zwei (verschiedenen) Knoten einen (gerichteten) Weg gibt

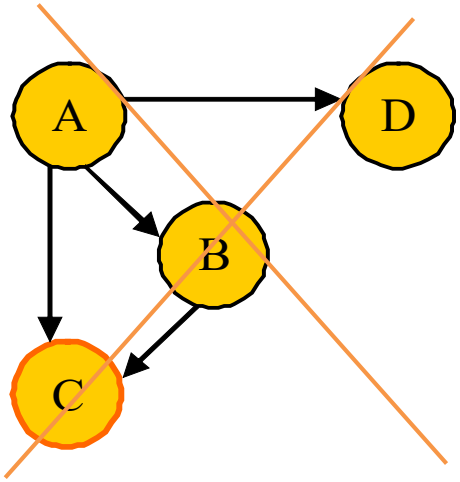
- Ist G nicht stark zusammenhängend, zerfällt er in eine Vereinigung **stark zusammenhängender Komponenten** (auch **starke Zusammenhangskomponenten** des Graphen genannt)

- Identifiziert man ungerichtete Graphen mit ihren zugeordneteten gerichteten Graphen, so ist Zusammenhang das gleiche wie starker Zusammenhang in diesen Graphen

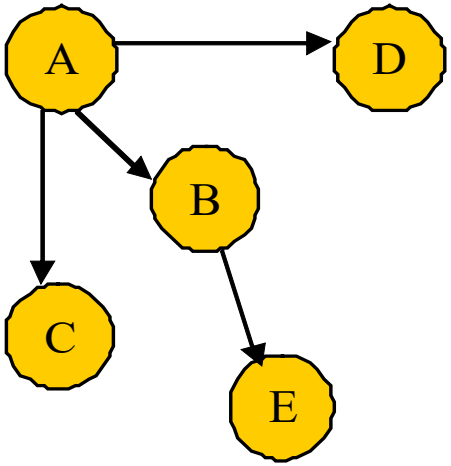
Gerichtete Graphen

Baum bei gerichteten Graphen

Ein gerichteter Graph heißt **Baum**, wenn er zyklenfrei und zusammenhängend ist und wenn jeder Knoten höchstens eine einlaufende Kante besitzt.



kein Baum



Baum

Anmerkung: Alle Bäume, die wir bisher kennengelernt haben, waren **gerichtete Bäume**

Gerichtete Graphen

Beispiel: Graph, Knotenmenge, Kantenmenge

Beispiele:

- $G_1 = (V_1, E_1)$
  V_1 = Städte in Deutschland
  $E_1 = \{ \{x,y\} \mid x, y \in V_1$, es gibt eine Autobahn zwischen x und y$\}$
  (Relation ist symmetrisch => ungerichteter Graph)

- $G_2 = (V_2, E_2)$
  V_2 = Spieler eines Tennisturniers
  $E_2 = \{ \{x,y\} \mid x, y \in V_2$, x spielt gegen y$\}$
  (Relation ist symmetrisch => ungerichteter Graph)

- $G_3 = (V_3, E_3)$
  V_3 = Einwohner von Dortmund
  $E_3 = \{(x,y) \in V_3 \times V_3 \mid$ x ist ein Kind von y$\}$
  (Relation ist unsymmetrisch => gerichteter Graph)

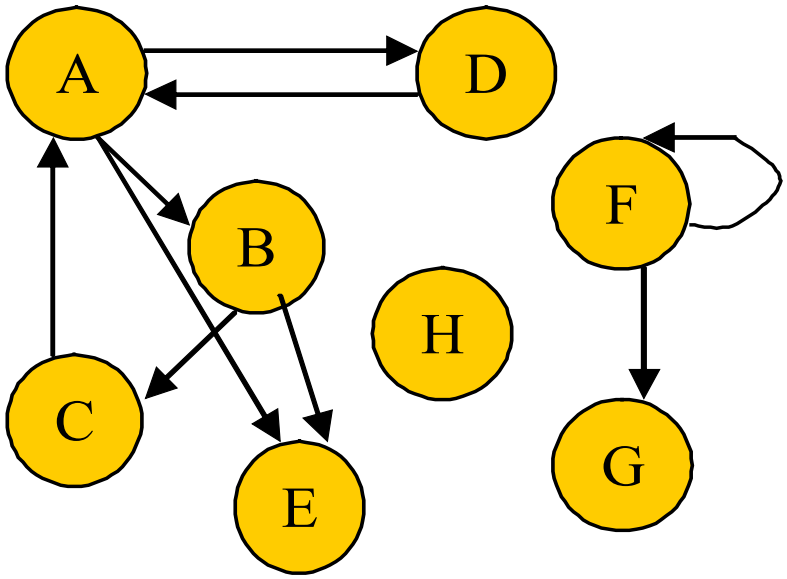
Datenstrukturen für Graphen

Datenstrukturen für Graphen

Adjazenzmatrix

Die Kanten eines Graphen kann man durch eine boolsche Matrix, die sogenannte Adjazenzmatrix darstellen:

```
boolean[][] kanten
```



⇔

	0	1	2	3	4	5	6	7
0		1		1	1			
1			1		1			
2	1							
3	1							
4								
5						1	1	
6								
7								

True = 1, False = " "

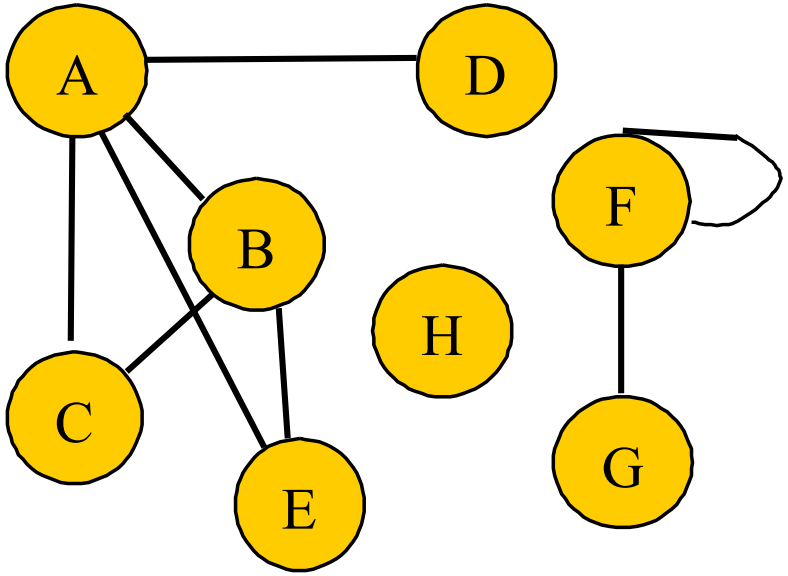
0	1	2	3	4	5	6	7
A	B	C	D	E	F	G	H

Datenstrukturen für Graphen

Adjazenzmatrix

Die Adjazenzmatrix für einen ungerichteten Graphen erhält man mittels des zugeordneten gerichteten Graphen. Die Matrix ist in diesem Fall symmetrisch:

```
boolean[][] kanten
```



⇔

	0	1	2	3	4	5	6	7
0		1	1	1	1			
1	1		1		1			
2	1	1						
3	1							
4	1	1						
5						1	1	
6						1		
7								

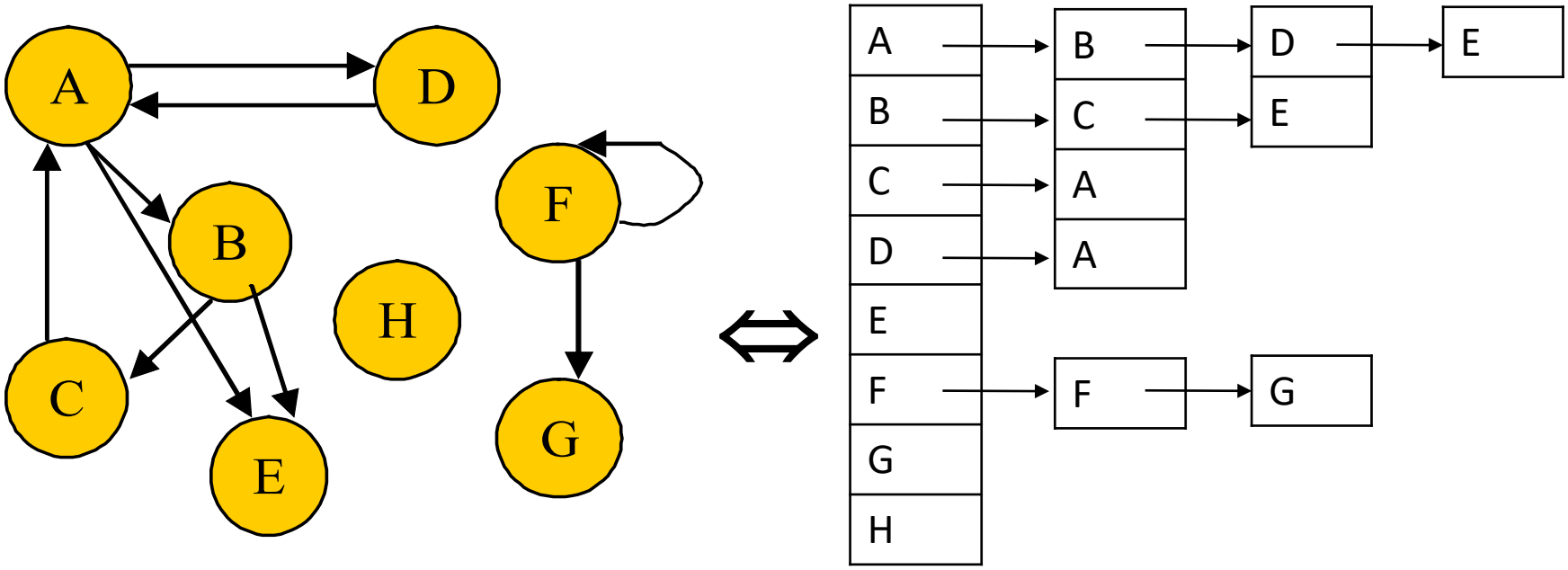
True = 1, False = " "

0	1	2	3	4	5	6	7
A	B	C	D	E	F	G	H

Datenstrukturen für Graphen

Adjazenzlisten

- Die von einem Knoten abgehenden Kanten eines Graph kann man durch eine lineare Liste darstellen:

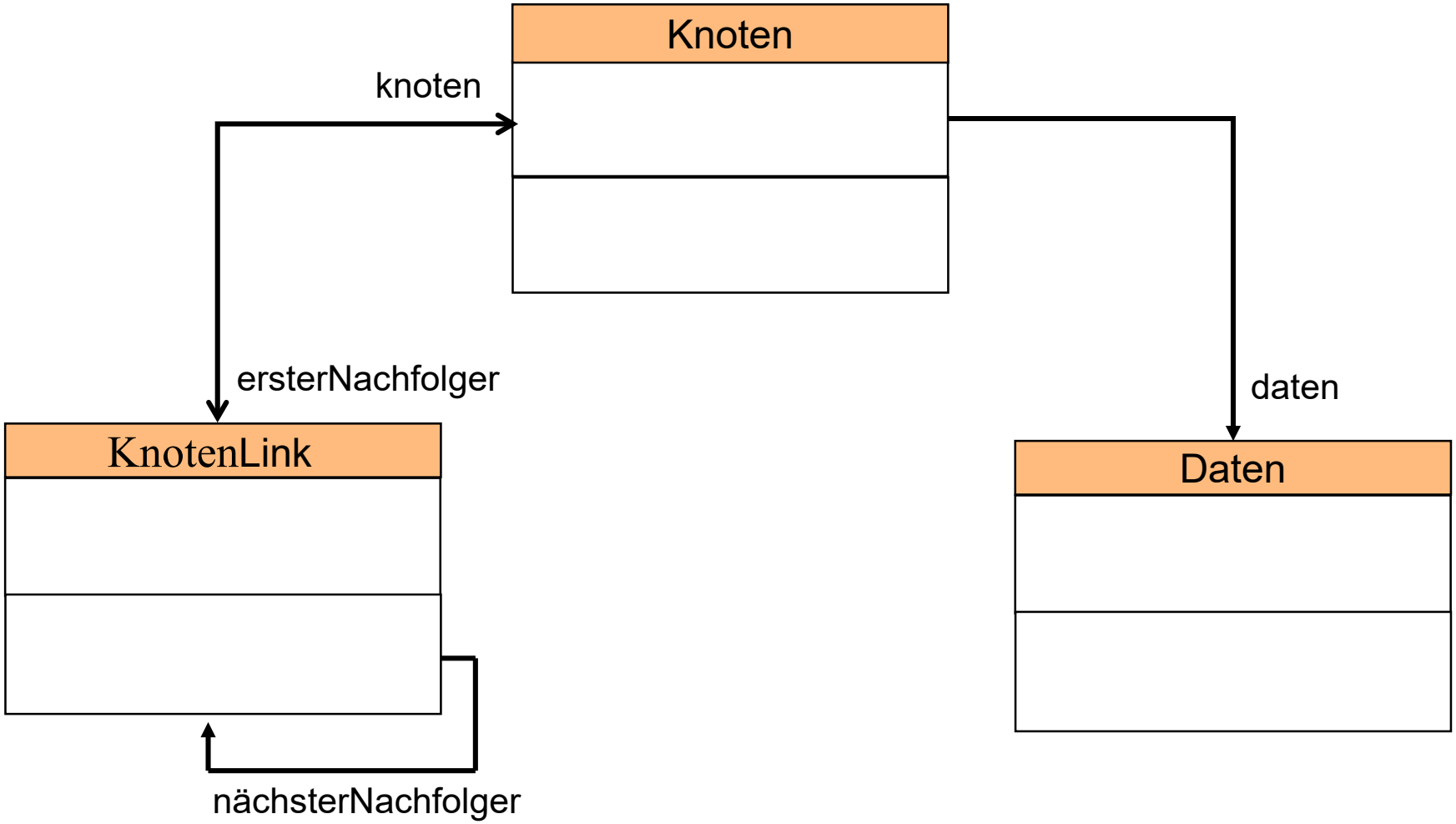


- Ungerichtete Graphen werden wiederum durch den zugeordneten gerichteten Graph dargestellt

Datenstrukturen für Graphen

Adjazenzlisten

Die von einem Knoten abgehenden Kanten eines Graph kann man durch eine lineare Liste darstellen:



Datenstrukturen für Graphen

Darstellung: Vergleich

- **Vorteile Adjazenzliste:**
  - Spart Speicherplatz gegenüber der Adjazenzmatrix, wenn diese nur dünn besetzt ist
  - Schnellerer Zugriff auf Nachfolger eines Knotens
- **Nachteile Adjazenzliste:**
  - Hoher Aufwand für den direkten Zugriff auf eine Kante von x nach y: Suche in der Liste aller Nachfolger erforderlich
- Man kann aus einer Adjazenzlistendarstellung eine Adjazenzmatrixdarstellung in Zeit $O(|V|^2)$ erzeugen
- Man kann aus einer Adjazenzmatrixdarstellung eine Adjazenzlistendarstellung in Zeit $O(|V|^2)$ erzeugen
- Siehe Graph.java in BSP12-Graph.zip

TRAVERSIERUNGEN

Traversierungen

Allgemeines

- **Traversieren** heißt alle Knoten eines Graphen durchwandern
- Ähnlich den Baum-Traversierungen
- Zyklen bergen die Gefahr von Endlosschleifen. Gegenmaßnahme beim Programmieren: besuchte Knoten markieren
- Traversierungen meist nur auf zusammenhängenden Graphen
- Traversierungsstrategien: **Tiefensuche** (depth first), **Breitensuche** (breadth first)
- **Voraussetzung** der in diesem Abschnitt vorgestellten Algorithmen:

  Der Graph G = (V,E) ist in Adjazenzlistendarstellung gegeben

Traversierungen

Tiefensuche

- Besucht rekursiv alle Knoten, die vom Ausgangsknoten k aus erreichbar sind, und markiert jeweils die besuchten Knoten
- Zu Beginn sind alle Markierungen gelöscht

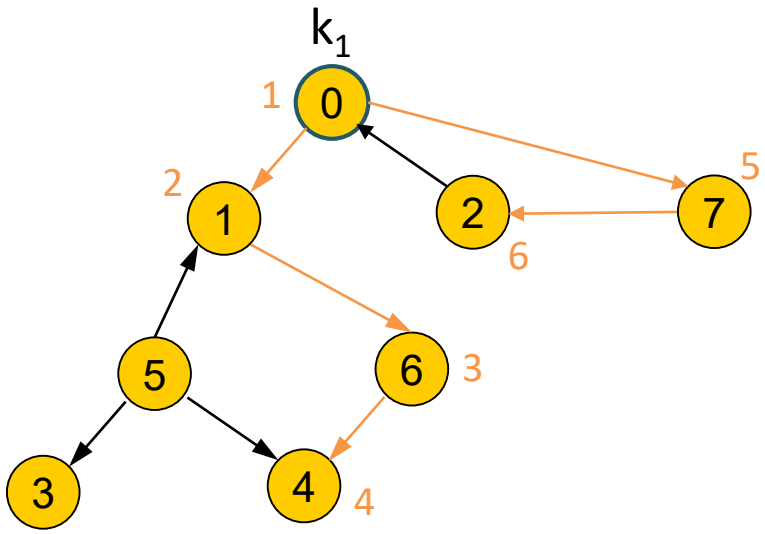
```
dfsRekursiv(knoten)
   initialisiere Knotenmarkierungen // keine Knoten markiert
   rekDfs(knoten)

rekDfs(knoten)
   markiere knoten
   // hier kann je nach Anwendung Code ergänzt werden
   nachfolger ← knoten.ersterNachfolger
   while nachfolger existiert do
     if  nachfolger.knoten nicht markiert then
        rekDfs(nachfolger.knoten)
     end if
     nachfolger ← nachfolger.nächsterNachfolger
   end while
```

- Zeitaufwand: O(|V| + |E|)

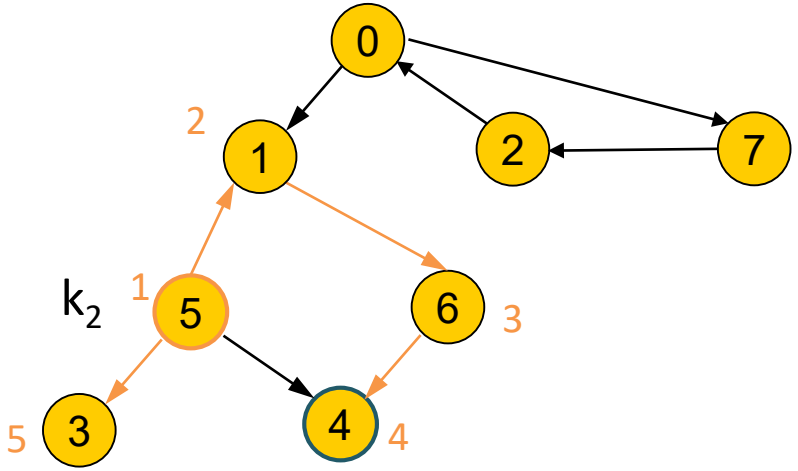
Traversierungen

Beispiel Tiefensuche

Die Reihenfolge der besuchten Knoten ist durch Zahlen neben den Knoten angegeben



Einstiegspunkt k_1



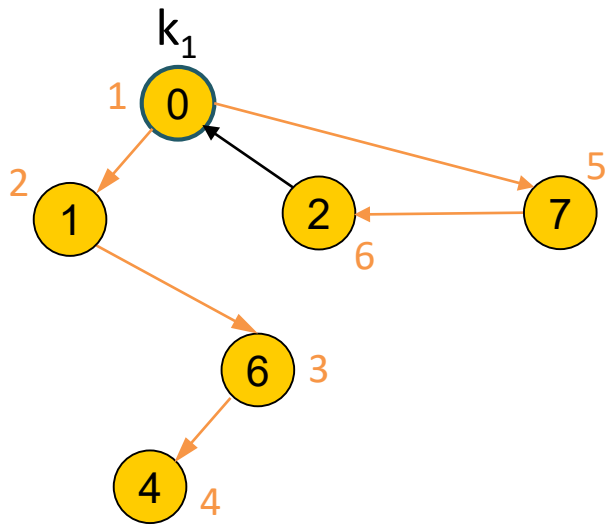
Einstiegspunkt k_2

Anmerkung: Blau umrandete Knoten würden 2 x besucht. Dies wird durch die Markierungen im Algorithmus verhindert!

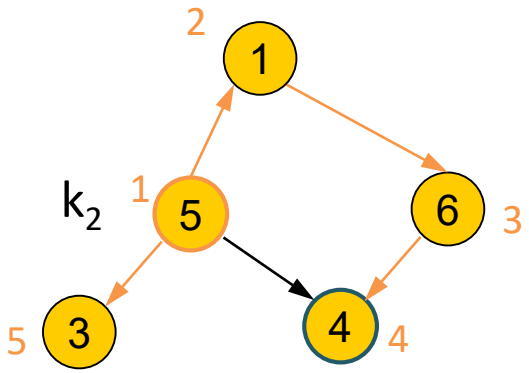
Traversierungen

Beispiel Tiefensuche

- Die besuchten Knoten mit den genutzten Kanten bilden einen Baum



Einstiegspunkt k_1



Einstiegspunkt k_2

- Die Kanten in diesem Baum (orange) werden **Baumkanten**, die anderen Kanten (schwarz) werden **Rückwärtskanten** genannt

Traversierungen

Anwendungen der Tiefensuche: Zusammenhang

- Durch Aufruf von Tiefensuche auf einem beliebigen Knoten kann getestet werden, ob ein ungerichteter Graph G=(V,E) zusammenhängend ist:

```
istZusammenhängend()
   initialisiere Knotenmarkierungen // keine Knoten markiert
   knoten ← beliebiger Knoten von G
   rekDfs(knoten)
   if alle Knoten von G markiert then
     return true
   else
     return false
   end if
```

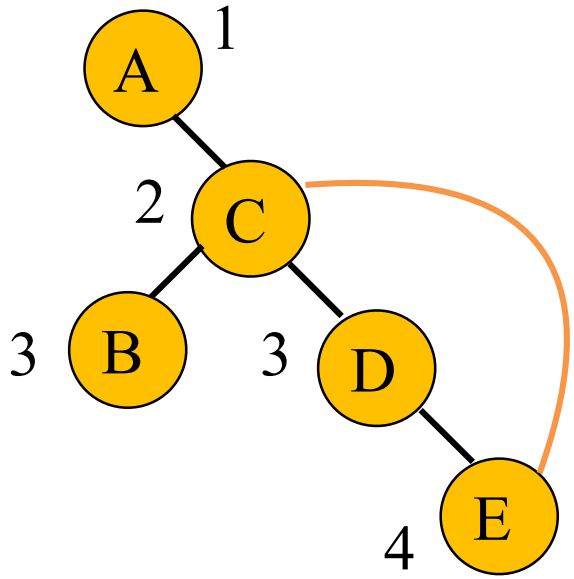
- Zeitaufwand: O(|V| + |E|)

Traversierungen

Anwendungen der Tiefensuche: Zyklenfreiheit

Durch Aufruf von Tiefensuche auf einem beliebigen Knoten kann getestet werden, ob ein ungerichteter, zusammenhängender Graph G=(V,E) zyklenfrei ist:

- Schritt 1: Nummeriere jeden Knoten mit seinem Niveau im entstehenden Baum
- Schritt 2: Teste, ob es Rückwärtskanten gibt, die ein Niveau überspringen



Traversierungen

Anwendungen der Tiefensuche: Zyklenfreiheit

- Durch Aufruf von Tiefensuche auf einem beliebigen Knoten kann getestet werden, ob ein ungerichteter, zusammenhängender Graph G=(V,E) zyklenfrei ist:

```
istZyklenfrei()
   initialisiere Knotennummerierung // keine Knoten nummeriert
   knoten ← beliebiger Knoten von G
   return rekDfsZyklenfrei(knoten, 1)

rekDfsZyklenfrei(knoten, niveau)
   ergebnis ← true
   nummeriere knoten mit niveau
   nachfolger ← knoten.ersterNachfolger
   while (ergebnis = true) and nachfolger existiert do
    if nachfolger.knoten nicht nummeriert then
     ergebnis ← ergebnis and rekDfsZyklenfrei(nachfolger.knoten, niveau+1)
    else if Nummer von nachfolger.knoten < niveau - 1 then
     ergebnis ← false
    end if
    nachfolger ← nachfolger.nächsterNachfolger
   end while
   return ergebnis
```

- Zeitaufwand: O(|V| + |E|)

Traversierungen

Tiefensuche iterativ

- Iterative Tiefensuche:

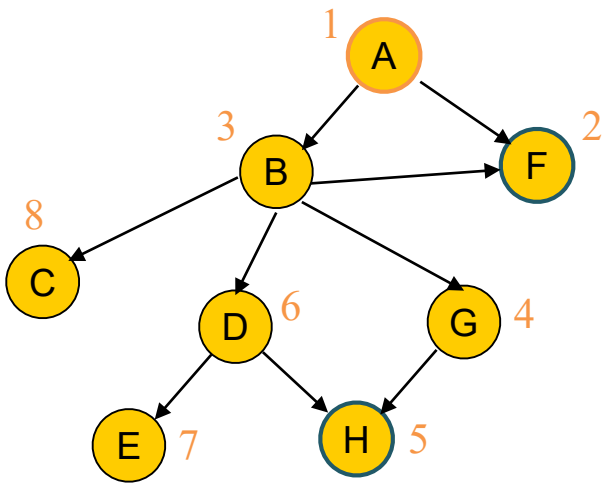
```
dfsIterativ(knoten)
   initialisiere Knotenmarkierungen // keine Knoten markiert
   initialisiere Stack S
   S.push(knoten)
   while S nicht leer do
     knoten ← S.pop()
     if knoten nicht markiert then
        markiere knoten
        nachfolger ← knoten.ersterNachfolger
        while nachfolger existiert do
          if nachfolger.knoten nicht markiert then
            S.push(nachfolger.knoten)
          end if
          nachfolger ← nachfolger.nächsterNachfolger
        end while
     end if
   end while
```

- Zeitaufwand: O(|V| + |E|)
- Vorsicht: Will man die gleiche Reihenfolge wie in der rekursiven Variante erhalten, so müssen die Nachfolger in umgekehrter Reihenfolge auf den Stack gelegt werden.

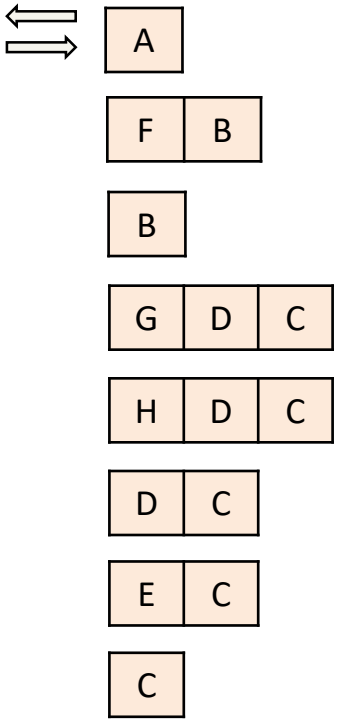
Traversierungen

Beispiel Stack bei Tiefensuche

Die Reihenfolge der besuchten Knoten ist durch Zahlen neben den Knoten angegeben; Besuch direkter Nachbarn in lexikografischer Reihenfolge



Einstiegspunkt A



Veränderung des Stacks während des Durchlaufs durch den Graphen

Traversierungen

Breitensuche

- Besucht zuerst alle Knoten, die direkt vom Ausgangsknoten k aus erreichbar sind, markiert sie und besucht erst dann die Knoten, die direkt mit den neu markierten Knoten verbunden sind

- Zu Beginn sind alle Markierungen gelöscht

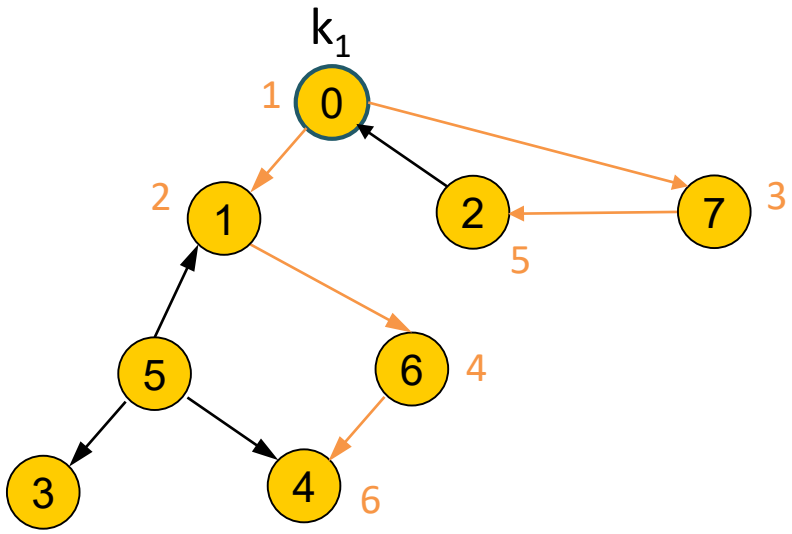
```
bfsIterativ(knoten)
   initialisiere Knotenmarkierungen // keine Knoten markiert
   initialisiere Queue Q
   Q.enqueue(knoten)
   markiere knoten
   while Q nicht leer do
     knoten ← Q.dequeue()
     nachfolger ← knoten.ersterNachfolger
     while nachfolger existiert do
       if nachfolger.knoten nicht markiert then
         Q.enqueue(nachfolger.knoten)
         markiere nachfolger.knoten
       end if
       nachfolger ← nachfolger.nächsterNachfolger
     end while
   end while
```

- Zeitaufwand: O(|V| + |E|)

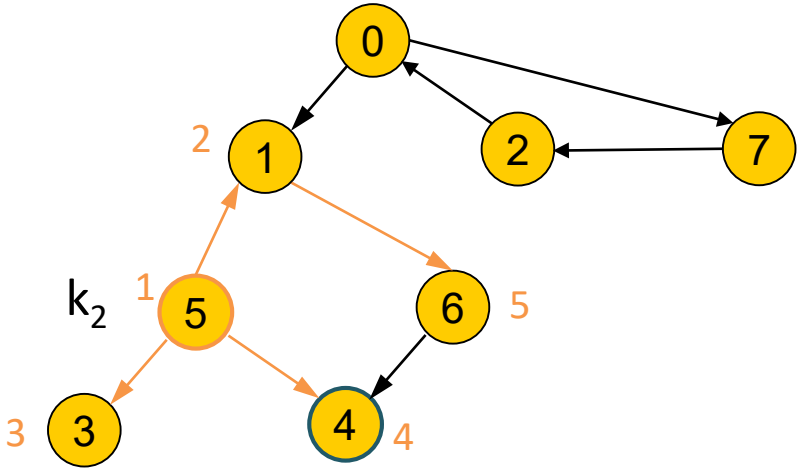
Traversierungen

Beispiel Breitensuche

- Die Reihenfolge der besuchten Knoten ist durch Zahlen neben den Knoten angegeben



Einstiegspunkt k_1



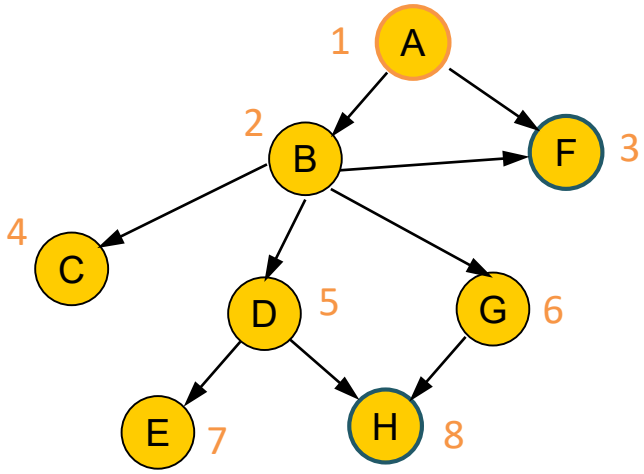
Einstiegspunkt k_2

Anmerkung: Blau umrandete Knoten würden 2 x besucht. Dies wird durch die Markierungen im Algorithmus verhindert!

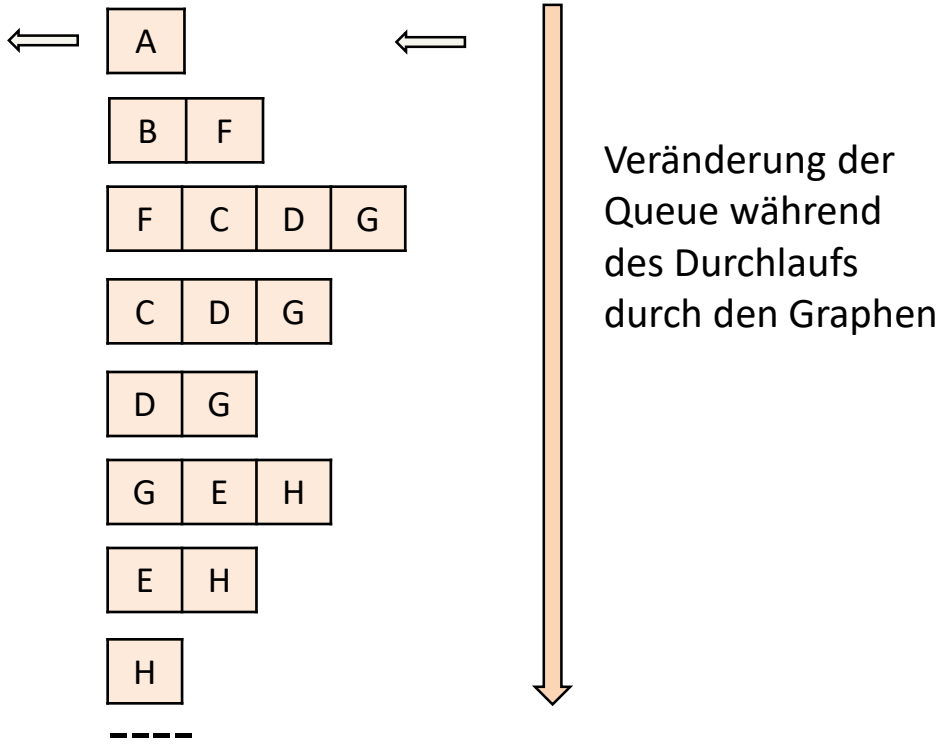
Traversierungen

Beispiel Queue bei Breitensuche

- Die Reihenfolge der besuchten Knoten ist durch Zahlen neben den Knoten angegeben; Besuch direkter Nachbarn in lexikografischer Reihenfolge



Einstiegspunkt A



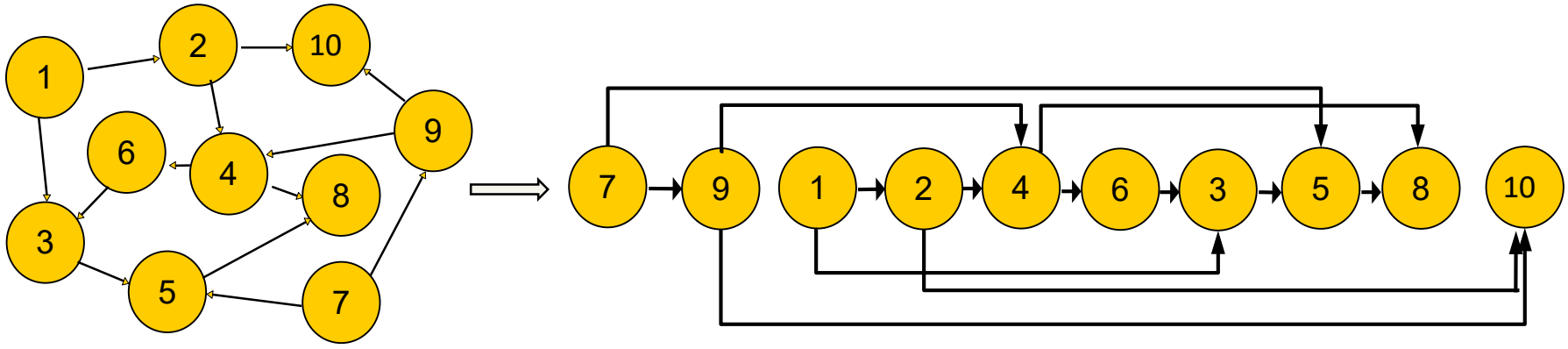
Queue: rechts einfügen
links entfernen

Topologisches Sortieren

Topologisches Sortieren

- Ein häufig im Kontext von Graphen auftretendes Problem ist das des **topologischen Sortierens**:
  In einem gerichteten Graphen sollen die Knoten, wenn möglich, so linear angeordnet werden, dass alle Kanten nur in eine Richtung zeigen

- Beispiel:



Topologisches Sortieren

Beispiel Projektplan eines größeren Projektes

Eigenschaft: bestimmte Teilprojekte müssen vor anderen abgeschlossen sein

- So ist es beim Hausbau in der Regel erforderlich, dass das Fundament fertig ist, bevor die Mauern hochgezogen werden
- Die Mauern müssen fertig sein, bevor an ihnen elektrische oder sanitäre Installationen angebracht werden können
- Elektrische und sanitäre Installationen sind aber (weitestgehend) voneinander unabhängig, zwischen ihnen besteht keine solche Reihenfolge-Beziehung

Topologisches Sortieren

Beispiel Projektplan

- Darstellung: gerichteter Graph
  - Die Knoten repräsentieren die einzelnen Teilprojekte
  - Eine Kante von Knoten x zu Knoten y gibt an, dass das Teilprojekt x vor dem Beginn des Teilprojekts y fertig gestellt werden muss
- Problemstellung
  - Die Knoten sollen so sortiert werden, dass sich eine lineare Reihenfolge ergibt, in der die Teilprojekte bearbeitet werden können

Topologisches Sortieren

Mehrere Lösungen möglich

- Eine lineare Reihenfolge der Knoten (auch **lineare Einbettung**) gibt es natürlich nur dann, wenn der Graph keine **Zyklen** enthält.
  - Diese dürfen bei einer korrekten Vorher-Nachher-Beziehung aber auch nicht vorkommen

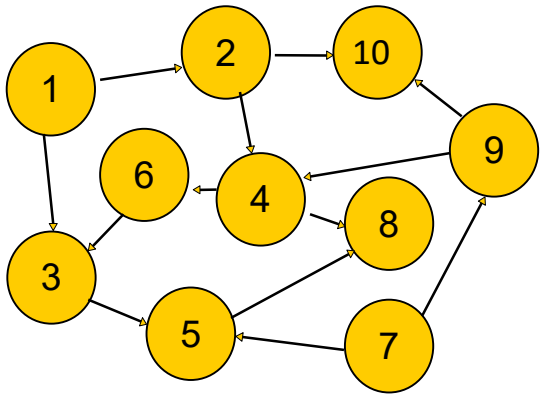
- Durchaus möglich ist aber, dass es mehrere lineare Einbettungen gibt
  - So hätte man im Beispiel zum topologischen Sortieren statt der Reihenfolge

    7 - 9 - 1 - 2 - 4 - 6 - 3 - 5 - 8 - 10

    etwa auch die Reihenfolge

    1 - 2 - 7 - 9 - 4 - 6 - 3 - 5 - 8 - 10

    wählen können



Topologisches Sortieren

Erzeugen einer Einbettung

- Da der Graph keine Zyklen enthält, muss es mindestens einen Knoten ohne Vorgänger geben
- Wir nehmen diesen Knoten, entfernen ihn (mit allen von ihm ausgehenden Kanten) aus dem Graphen und stellen ihn an den Anfang der zu erzeugenden Liste
- Der Restgraph muss nun wiederum einen Knoten ohne Vorgänger enthalten, so dass wir den Algorithmus wiederholt anwenden können, bis der Graph vollständig abgearbeitet wurde
- Die Kanten der so erhaltenen linearen Einbettung entsprechen natürlich noch genau den Kanten des Ursprungsgraphen

Topologisches Sortieren

Zahl der Vorgänger merken

- Um diesen Algorithmus ausführen zu können, benötigt man für jeden Knoten die Zahl seiner Vorgänger
  - Da nur die Anzahl benötigt wird ohne genaue Kenntnis darüber, welche Vorgänger dies sind, erweitert man die Klasse `KnotenTyp` um eine Attribut `anzVorg` vom Typ `int`
  - Die Anzahl der Vorgänger kann vor dem Topologischen Sortieren oder schon bei der Eingabe oder Initialisierung des Graphen (d.h. beim Anlegen einer neuen Kante) für jeden Knoten berechnet werden
- Die Knoten, die keinen Vorgänger besitzen, werden in einer Liste abgelegt

Topologisches Sortieren

Prinzip des Algorithmus

- Siehe Graph.java in BSP12-TopologischesSortieren.zip
  - Der jeweils erste gefundene Knoten ohne Vorgänger wird aus der Liste der Knoten ohne Vorgänger entfernt
  - Für alle Kanten, die von ihm ausgehen, muss in den Zielknoten der Vorgängerzähler um 1 verringert werden
  - Wird hierbei der Vorgängerzähler auf 0 verringert, so wird dieser Knoten in die Liste der Knoten ohne Vorgänger aufgenommen
  - ein globaler Durchlaufzähler wird um 1 erhöht, um am Ende prüfen zu können, ob alle Knoten abgearbeitet wurden
  - Ist die Liste der Knoten ohne Vorgänger leer so endet der Algorithmus; Entspricht zu diesem Zeitpunkt der Durchlaufzähler nicht der Anzahl der Knoten im Graph, so existiert ein Zyklus und der Graph ist nicht topologisch sortierbar

Lernziele

- Sie können wichtige Begriffe in der Welt der Graphen aufzählen, identifizieren und beschreiben
- Sie können einen Graphen in zwei unterschiedlichen Datenstrukturen darstellen, beschreiben und hierzu die Objektdiagramme aufzeichnen oder Klassen angeben
- Sie wissen, wie der Tiefen- und Breitensuchedurchlauf durch einen Graphen implementiert ist und können ihn anwenden
- Sie wissen, wie topologisches Sortieren funktioniert und können es anwenden

Literatur

Quellen

- Günther Saake: Algorithmen und Datenstrukturen
  - Kapitel 16 Graphen
- Karsten Weicker: Algorithmen und Datenstrukturen
  - Kapitel 1 (Ein Anwendungsbeispiel),
    4.5 (Strukturierte Verarbeitung von Graphen)

Vertiefend

- **[Güting_2018]**
  Güting, Ralf Hartmut, Dieker, Stefan : Datenstrukturen und Algorithmen, Springer Vieweg, 4. Auflage, 2018 (https://link.springer.com/book/10.1007%2F978-3-658-04676-7)
- **[Ottmann_Widmayer_2017]**
  Ottmann, Thomas, Widmayer, Peter: Algorithmen und Datenstrukturen, Springer Vieweg, 6. Auflage, 2017 (https://link.springer.com/book/10.1007%2F978-3-662-55650-4)
- **[Sedgewick_2014]**
  Sedgewick, Robert, Wayne, Kevin: Algorithmen und Datenstrukturen, Pearson, 4. Auflage, 2014