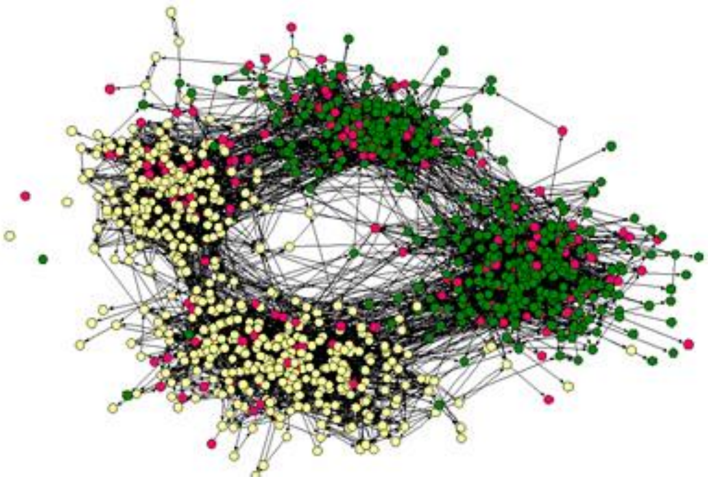
Algorithmen und Datenstrukturen
Kapitel 7A: Graphen – Tiefensuche, Breitensuche

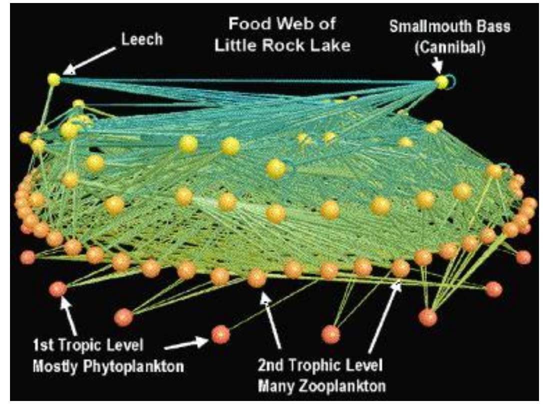
Prof. Dr. Wolfgang Mühlbauer
Fakultät für Informatik
wolfgang.muehlbauer@th-rosenheim.de

Wintersemester 2019/2020

Graphen

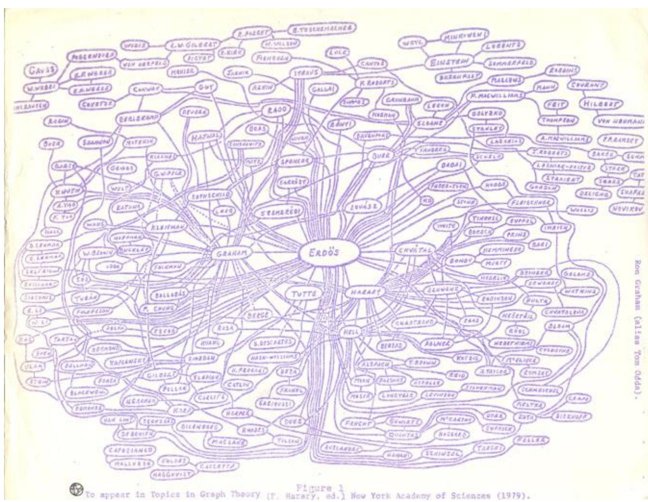


friendship network

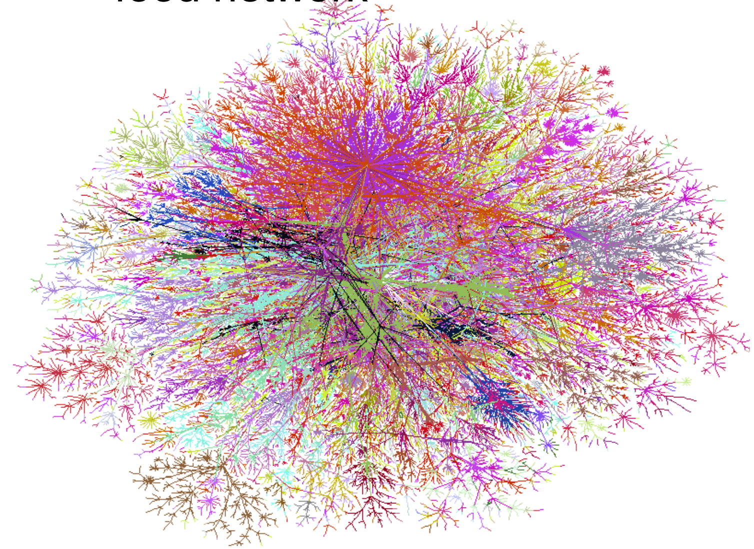


food network

Quelle: [3]



Internet network



collaboration network

Übersicht

- **Graphen als Datenstruktur**
  - **Definition**
  - Speichern von Graphen: Adjazenzliste und Adjazenzmatrix
  - Durchlaufen von Graphen: Breitensuche
  - Durchlaufen von Graphen: Tiefensuche
  - Topologische Sortierung von gerichteten Graphen

- Kürzeste Wege
  - Siehe Kapitel 7B

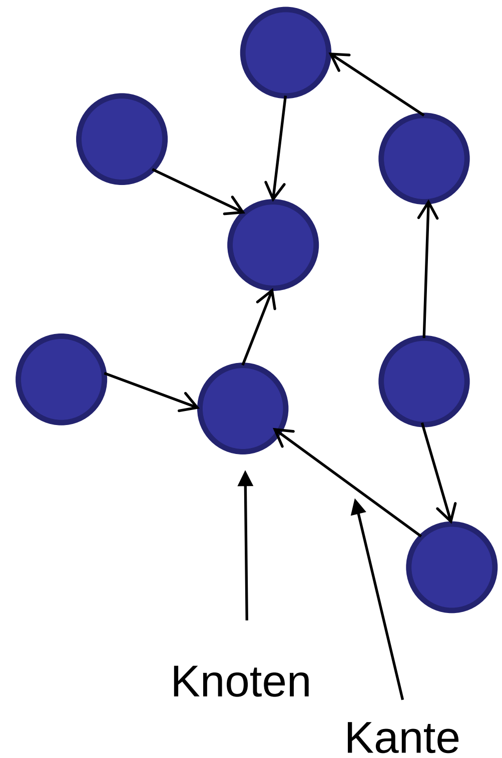
Anwendungen von Graphen

- Navigation zwischen 2 Orten in einem Straßennetz
  - Kürzeste Route
  - Schnellste Route
- Wegefindung (Routing) im Internet
- Rundreise
  - Wie besucht man alle Knoten mit einer kürzest möglichen Rundreise?
  - *Eulerkreis*: Rundweg durch Königsberg, so dass jede Brücke genau einmal überquert wird und man am Schluss beim Ausgangspunkt ist?
- Wie viele Farben benötigt man, um die Länder einer Karte einzufärben?
- Abstimmung von Arbeitsabläufen
  - Welche Aufgaben können parallel erledigt werden?

Gerichteter Graph: Definition

- Nützliche Abstraktion für zahlreiche Anwendungen

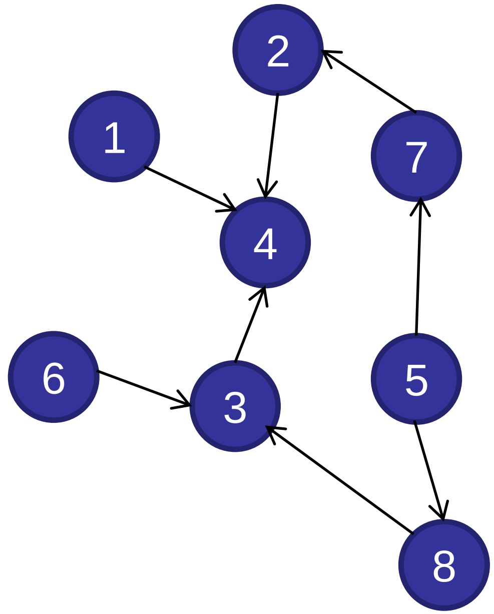
- **Definition:** Ein ***gerichteter Graph G* (V,E)** *(directed graph, digraph)* besteht aus:
  - Menge von ***Knoten*** *(nodes, vertices)*
    - $V = \{0, 1, 2, .., |V| - 1\}$
    - Annahme im Folgenden:
      - Knotennamen sind Integer.
      - Vorteile für Implementierung: Über Arrayindizes schneller Zugriff auf Knoteninformation.
      - Falls nötig: zusätzliche Abbildungstabelle (Array) für die Zuordnung Integer und Knotennamen.
  - Menge von *gerichteten* ***Kanten*** *(Edges)*, die Knoten miteinander verbinden.
    - $E \subseteq V \times V$.



- Falls eine Kante von *v* zu *v‘* zeigt, so nennt man *v* und *v‘* ***adjazent (=benachbart).***

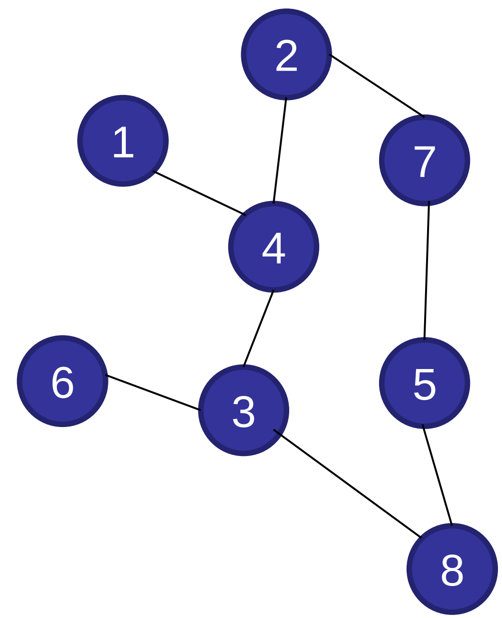
Gerichteter Graph: Begriffe

- ***Anzahl der Nachbarn*** von *v*
  - ***Eingangsgrad*** *indeg(v)*: # einmündender Pfeile
  - ***Ausgangsgrad*** *outdeg(v)*: # ausgehender Pfeile
- Graph $G'(V', E')$ ist ***Teilgraph*** von G falls:
  - $V' \subseteq V$
  - $E' \subseteq E$
- ***Pfade***, Wege zwischen 2 Knoten
  - Jeder Knoten darf nur einmal besucht werden
  - > 1 Weg möglich
- ***Zyklus / Kreis***
  - Weg der Länge > 1, der am Ausgangspunkt endet.
- Falls nicht anders erwähnt: Nur 1 Kante zwischen 2 Knoten erlaubt.



Ungerichteter Graph

- Definition analog zu gerichtetem Graphen
  - Unterschied: Kanten haben keine Richtung
- ***Grad*** eines Knoten *v*: *deg(v)*
  - Anzahl der Kanten eines Knoten
  - Sprechweise: „v ist mit *deg(v)* Kanten ***inzident***“.
- Alle weiteren Definitionen analog zu gerichteten Graphen.



Übersicht

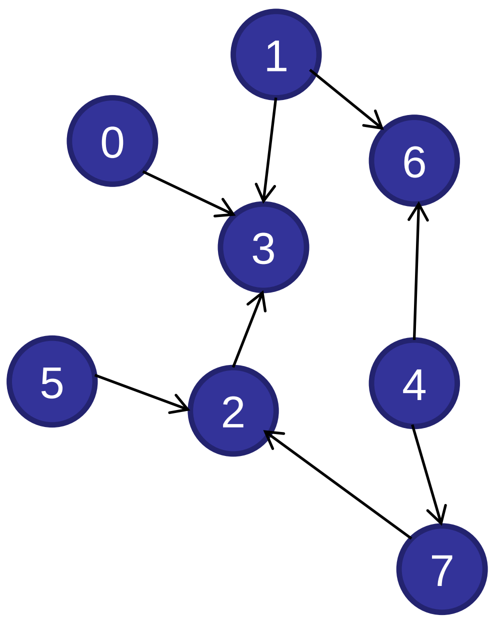
- Graphen als Datenstruktur
  - Definition
  - **Speichern von Graphen**
  - Durchlaufen von Graphen: Breitensuche
  - Durchlaufen von Graphen: Tiefensuche
  - Topologische Sortierung von gerichteten Graphen

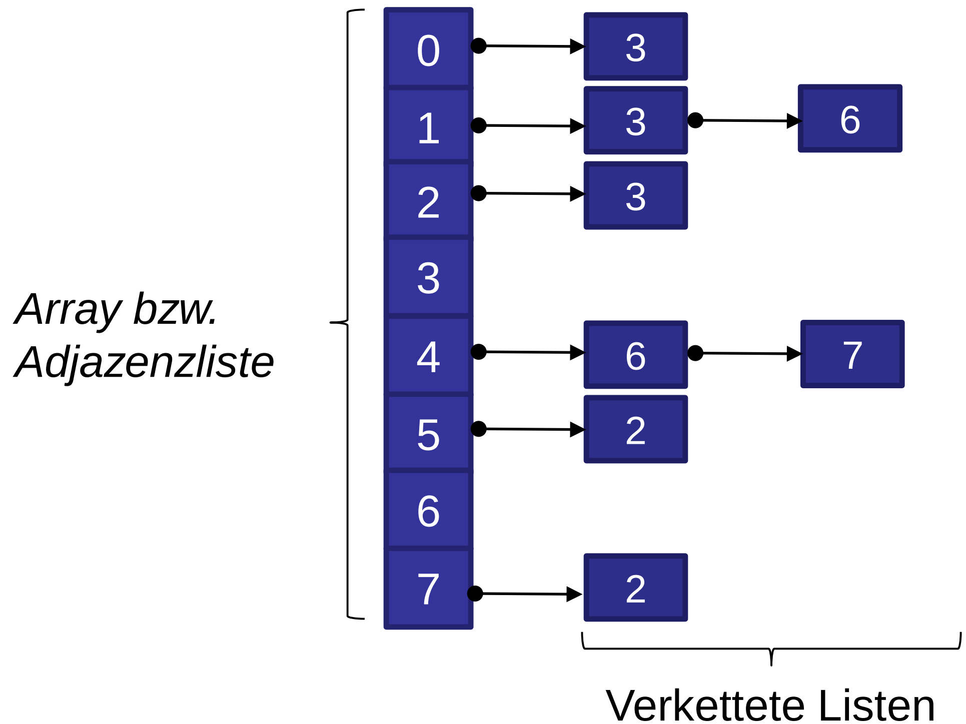
- Kürzeste Wege
  - Siehe Kapitel 7B

Adjazenzliste

- ***Array von*** $|V|$ ***Listen***, eine Liste für jeden Knoten *u*
- Liste von Knoten *u* enthält alle (benachbarten) Knoten v, so dass $(u, v) \in E$
- Falls Kanten Gewichte haben, so kann man diese in der Adjazenzliste mitspeichern.

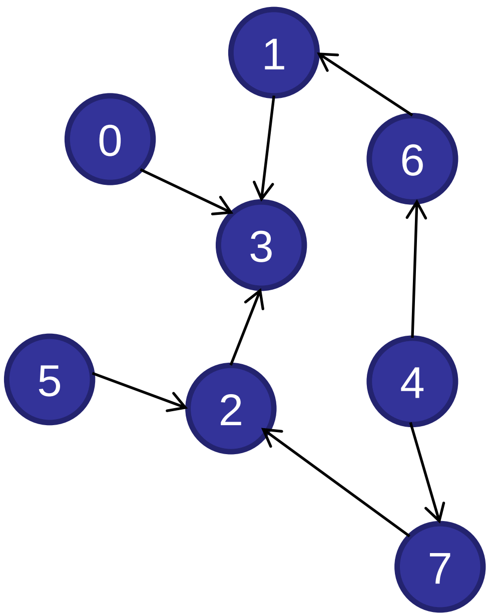
Quellcode: UndirectedGraph.java





Adjazenzmatrix

- $|V| \times |V|$ Matrix
- Eintrag a_{ij} gibt an, ob zwischen dem Knoten *i* und Knoten *j* eine Kante existiert.



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

Diskussion

	Adjazenzliste	Adjazenzmatrix
Speicher	$\Theta(\lvert V\rvert + \lvert E\rvert)$	$\Theta(\lvert V\rvert^2)$
Laufzeit, um alle Knoten zu finden, die zu einem Knoten u benachbart sind.	$\Theta(degree(u))$	$\Theta(\lvert V\rvert)$
Laufzeit, um zu entscheiden, ob Kante (u,v) existiert.	$O(degree(u))$	$\Theta(1)$

- Adjazenzliste meist effizienter, da Graph nie alle möglichen Kanten enthält.
  - Prüfen ob bestimmte Kante in Graphen enthalten ist, dauert aber etwas länger.

- Falls nicht anders erwähnt wird im Folgenden immer eine Adjazenzliste verwendet.

Quellcode: UndirectedGraph.java

Publikums-Joker:

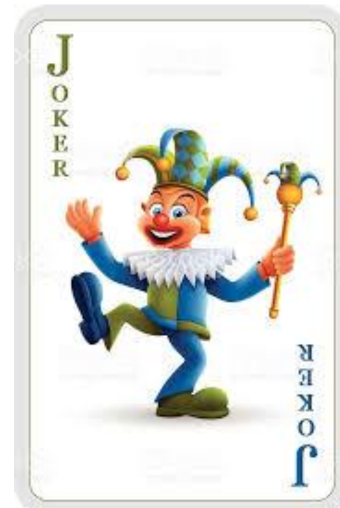
Was ist die kleinste obere Schranke für die Worst-Case Laufzeit, um bei einem Graphen, der als Adjazenzmatrix abgespeichert ist, die Anzahl der Kanten zu ermitteln?

A. $O(|V|)$

B. $O(|E|^2)$

C. $O(|E|)$

D. $O(|V|^2)$



Übersicht

- Graphen als Datenstruktur
  - Definition
  - Speichern von Graphen: Adjazenzliste und Adjazenzmatrix
  - **Durchlaufen von Graphen: Breitensuche**
  - Durchlaufen von Graphen: Tiefensuche
  - Topologische Sortierung von gerichteten Graphen

- Kürzeste Wege
  - Siehe Kapitel 8B

Durchlaufen von Graphen

- ***Motivation:*** Labyrinth
  - Person ist in Labyrinth und beginnt ausgehend von einer Kreuzung alle Kreuzungen zu besuchen.
  - Unterschied zu Pledge-Algorithmus: Möglichkeit z.B. mit Kreide zu markieren.
  - Mögliche Ansätze
    - Man geht so lange wie möglich geradeaus („***Suche in der Tiefe***“)
    - Man besucht erst alle nächstgelegenen Kreuzungen („***Suche in der Breite***“)

- 2 Verfahren zum Besuchen aller Knoten:
  - ***Breitensuche*** (jetzt)
  - ***Tiefensuche*** (im Anschluss)

- ***Annahmen***
  - Jeder Knoten kennt seine Nachbarn (Adjazenzliste!)
  - Manchmal wird zusätzlich ein fester Startknoten vorgegeben.

- Grundlage für zahlreiche Algorithmen

Breitensuche (engl. Breadth-First Search)

- ***Eingabe***
  - Gerichteter oder ungerichteter Graph *G(V,E)*
  - Startknoten
- ***Ausgabe***
  - ***v.d*****:** Entfernung ("distance") von Startknoten *s* zu Knoten *v*
    - Kürzester Pfad = Pfad mit minimaler Kantenanzahl zwischen wischen *s* und *v*
  - ***v.π*****:** Vorgängerknoten *u* auf kürzestem Weg von Startknoten *s* zu Knoten *v*.
    - *(u,v)* ist die ***letzte Kante*** auf kürzestem Pfad.
    - *u* ist Vorgänger/Predecessor im "Baum der kürzesten Wege".
- ***Idee***
  - Schicke von *s* eine Welle aus.
  - Welle trifft zunächst alle Knoten, die 1 Kante entfernt sind.
  - Dann alle Knoten, die 2 Kanten entfernt sind, usw.
  - Abspeichern und Abarbeiten von Tasks in der FIFO Reihenfolge
    - Welche Datenstruktur?

Farben

- Attribute eines Knoten, siehe vorherige Folie
  - ***v.d*** und
  - ***v.π***

- **Weiteres Attribut: Farbe** eines Knoten
  - ***v.color***: `WHITE, BLACK und GRAY`
  - Erlaubt es Knoten zu markieren und sich z.B. zu merken ob man den Knoten schon mal besucht hat ("Kreide").
  - Farbe hilfreich für Verständnis des Algorithmus.

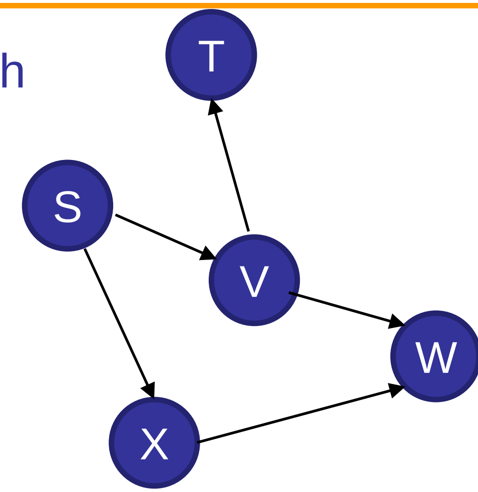
- **Bedeutung:**
  - **WHITE**: Knoten noch nie besucht, Knoten noch unentdeckt.
  - **BLACK**: Knoten besucht und ***alle*** Nachbarknoten entdeckt.
  - **GRAY**: Knoten bereits entdeckt, hat aber möglicherweise noch unentdeckte (weiße) Nachbarknoten.
    - Grenze zwischen entdeckten und unentdeckten Knoten.

Breitensuche: Algorithmus

```
BFS(V, E, s)
// s ist der Startknoten
    for each u ∈ V \ {s}
        u.d = ∞
        u.π = NIL
        u.color = WHITE
    s.d = 0
    s.π = NIL
    s.color = GRAY
    Q = Ø
    ENQUEUE(Q,s)
    while Q ≠ Ø
        u = DEQUEUE(Q)
        for each v ∈ G.Adj[u]
            if v.d == ∞
                v.color = GRAY
                v.d = u.d + 1
                v.π = u
                ENQUEUE(Q,v)
        u.color = BLACK
```

Quellcode: UndirectedGraph.java / bfs

Breitensuche auf Graph *G(V,E)* beginnend bei Startknoten *s*

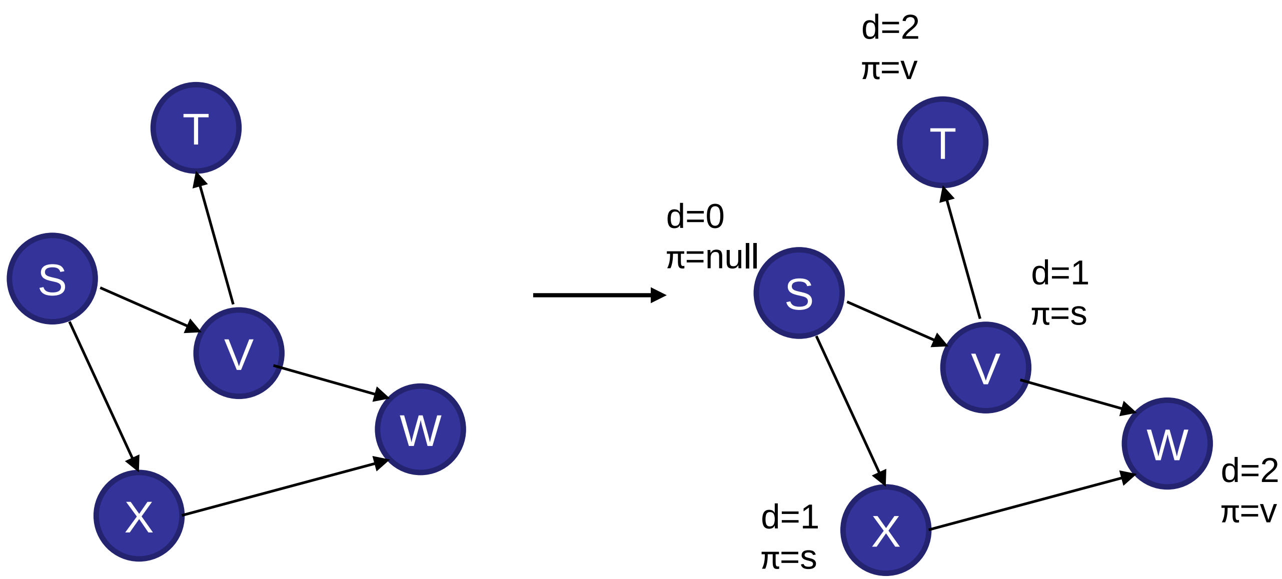


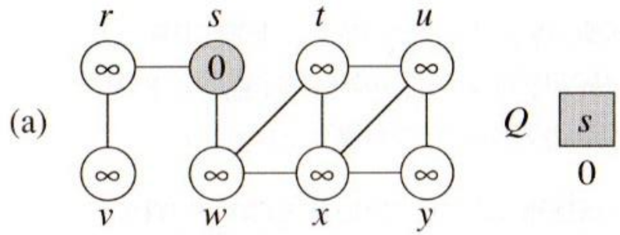
Queue: enthält zu Beginn Startknoten *s*, der als GRAY markiert wird. Im weiteren Verlauf enthält Queue stets „graue“ Knoten

While-Schleife iteriert solange es noch graue Knoten gibt (Knoten, von denen noch nicht alle Nachbarn entdeckt wurden).

Breitensuche: Übung

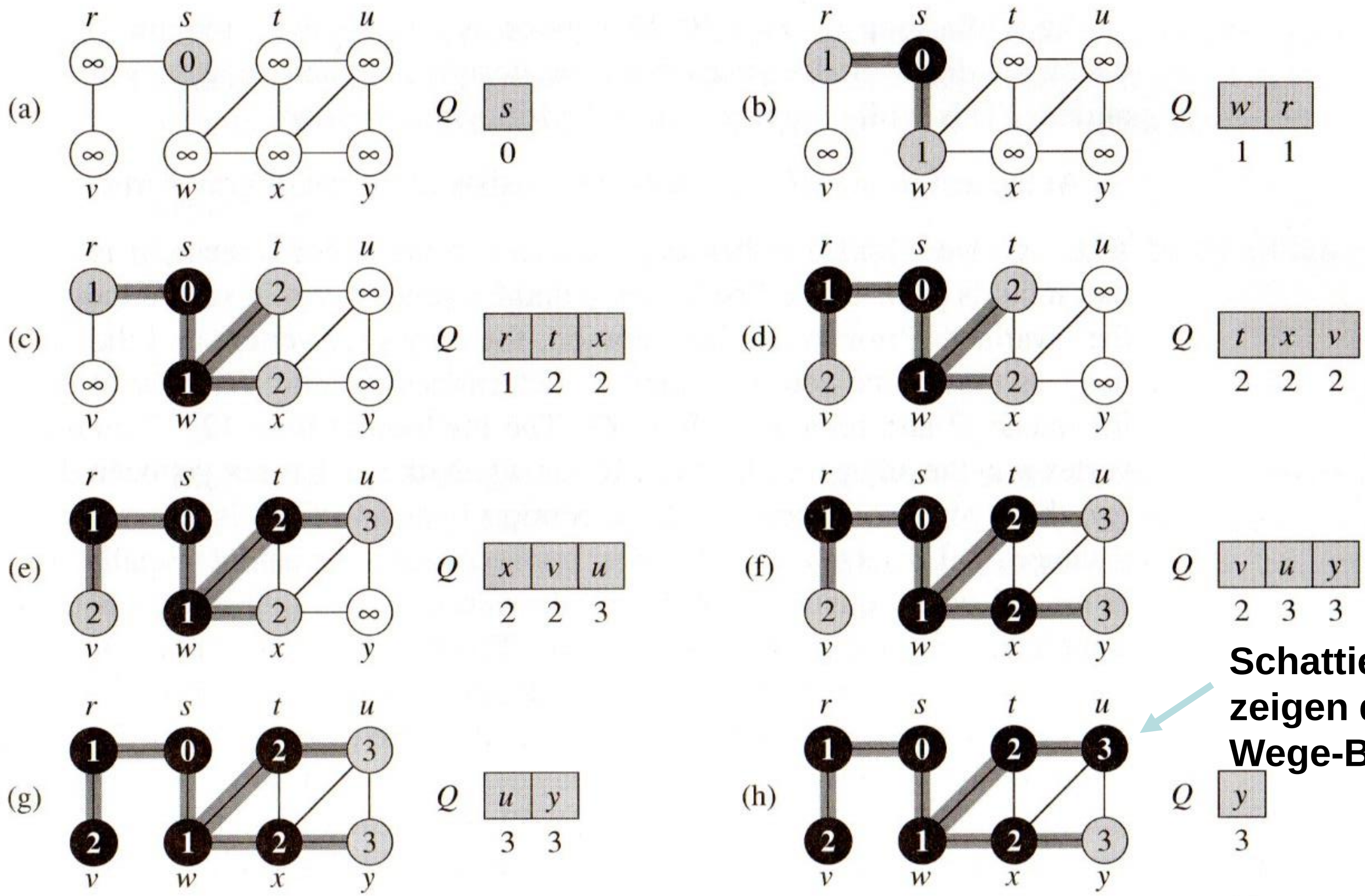
- Breitensuche ausgehend vom Startknoten *s*.
- **Annahme**
  - Es wird immer zunächst der alphabetisch kleinere Nachbar besucht.
- **Frage**
  - Mögliche Besuchsreihenfolge?
  - Werte von $v.d$ und $v.\pi$ für jeden Knoten v nach Beendigung der Breitensuche?





- Innerhalb jedes Knotens *u* steht der berechnete Wert von *u.d*.
- Die verwendeten „Vorgängerkanten“ sind schattiert.
- *Q* zeigt jeweils den Inhalt der Queue am Anfang der Iteration

Quelle[1]



Schattierte Kanten zeigen den "Kürzeste-Wege-Baum"

- Innerhalb jedes Knotens *u* steht der berechnete Wert von *u.d.*
- Die verwendeten „Vorgängerkanten“ sind schattiert.
- *Q* zeigt jeweils den Inhalt der Queue am Anfang der Iteration

Quelle[1]

Breitensuche: Diskussion

- Queue Q enthält zu jedem Zeitpunkt Menge der grau gefärbten Knoten.

- **Laufzeit**: O(|V|+|E|)
  - Jeder Knoten wird höchstens einmal in die Queue eingetragen. Warum?
  - Es finden höchstens O(|V|) Operationen auf der Queue statt.
  - Für jeden Knoten wird die Adjazenzliste durchlaufen → insgesamt werden alle Kanten einmal "besucht": O(|E|):

- Breitensuche findet von Start- zu jedem Zielknoten die kürzeste Entfernung
  - Aber ***nur unter der Annahme:*** Alle Kantengewichte sind 1.
  - "BFS-Tree": Der ***Kürzeste-Wege-Baum*** kann über die Vorgänger ***`v.π`*** rekonstruiert werden (siehe Vorgängerfolie)

- Implementierung
  - `UndirectedGraph.java`, Methode `bfs`

- Animation: https://www.cs.usfca.edu/~galles/visualization/BFS.html

Publikums-Joker:

Was ist die kleinste obere Schranke für die Worst-Case Laufzeit bei der Breitensuche, falls der Graph n Knoten und $n^{1,25}$ Kanten hat?

A. $O(n)$

B. $O(n^{1,25})$

C. $O(n^{2,25})$

D. $O(n*n)$

Übersicht

- Graphen als Datenstruktur
  - Definition
  - Speichern von Graphen: Adjazenzliste und Adjazenzmatrix
  - Durchlaufen von Graphen: Breitensuche
  - **Durchlaufen von Graphen: Tiefensuche**
  - Topologische Sortierung von gerichteten Graphen

- Kürzeste Wege
  - Siehe Kapitel 7B

Tiefensuche (engl. Depth-First Search)

- ***Eingabe***
  - Gerichteter oder ungerichteter Graph *G(V,E)*
  - Zur Abwechslung: Dieses Mal kein Startknoten vorgegeben!
  - Funktioniert auch, falls Graph *G* nicht zusammenhängend ist.

- ***Ausgabe:*** 2 "Zeitstempel" für jeden Knoten
  - ***Discovery Time*** $\boldsymbol{v.d}$ = Zeitpunkt, an dem Knoten entdeckt wird.
    - D.h. "grau" eingefärbt wird (wie bei Breitensuche)
  - ***Finish Time*** $\boldsymbol{v.f}$ = Zeitpunkt, an dem alle Nachbarn eines Knotens entdeckt
    - D.h. "schwarz" eingefärbt sind (wie bei Breitensuche)
  - Zeitstempel des Pseudocodes so gewählt, dass: $1 \leq v.d < v.f \leq 2|V|$
  - Zeitstempel nützlich für einige Anwendungen, siehe topologische Sortierung!
  - $\boldsymbol{v.\pi}$ "Vorgänger", über den ein Knoten entdeckt wurde.
  - Entfernung wird nicht gespeichert!

- **Idee**
  - Sobald neuer Knoten entdeckt wird, setze zunächst Erforschung vom ***neuen*** Knoten fort ("LIFO"-Strategie)
  - Vergleich Breitensuche: Dort eher "FIFO"-Strategie.

Tiefensuche: Algorithmus

```
DFS()
    for each u ∈ V
        u.color = WHITE
        u.π = NIL
    time = 0
    for each u ∈ V
        if u.color == WHITE
            DFS-VISIT(u)

DFS-VISIT(u)
    time = time + 1
    u.d = time
    u.color = GRAY
    for each v ∈ G.Adj[u]
            if v.color == WHITE
                v.π = u
                DFS-VISIT(v)
    u.color = BLACK
    time = time + 1
    u.f = time
```

Quellcode: UndirectedGraph.java / dfs

Tiefensuche (hier: kein Startknoten vorgegeben)

Alle Knoten am Anfang weiß

Globale Zeit initialisieren

Für alle weißen Knoten rekursive Methode aufrufen; ggfs. ist Graph nicht zusammenhängend

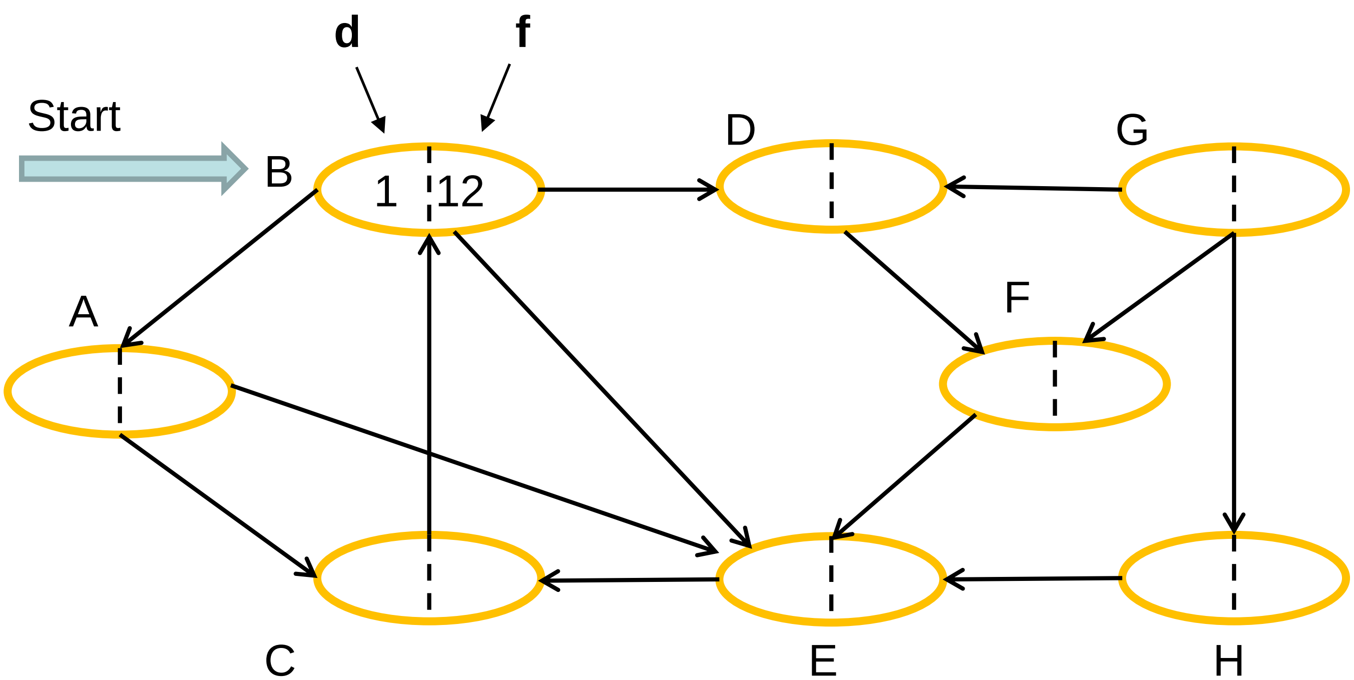
Neuer Knoten entdeckt: Grau einfärben und Discovery Time setzen

Ist Nachbar des Knotens noch unentdeckt? Ggfs. rekursiver Aufruf.

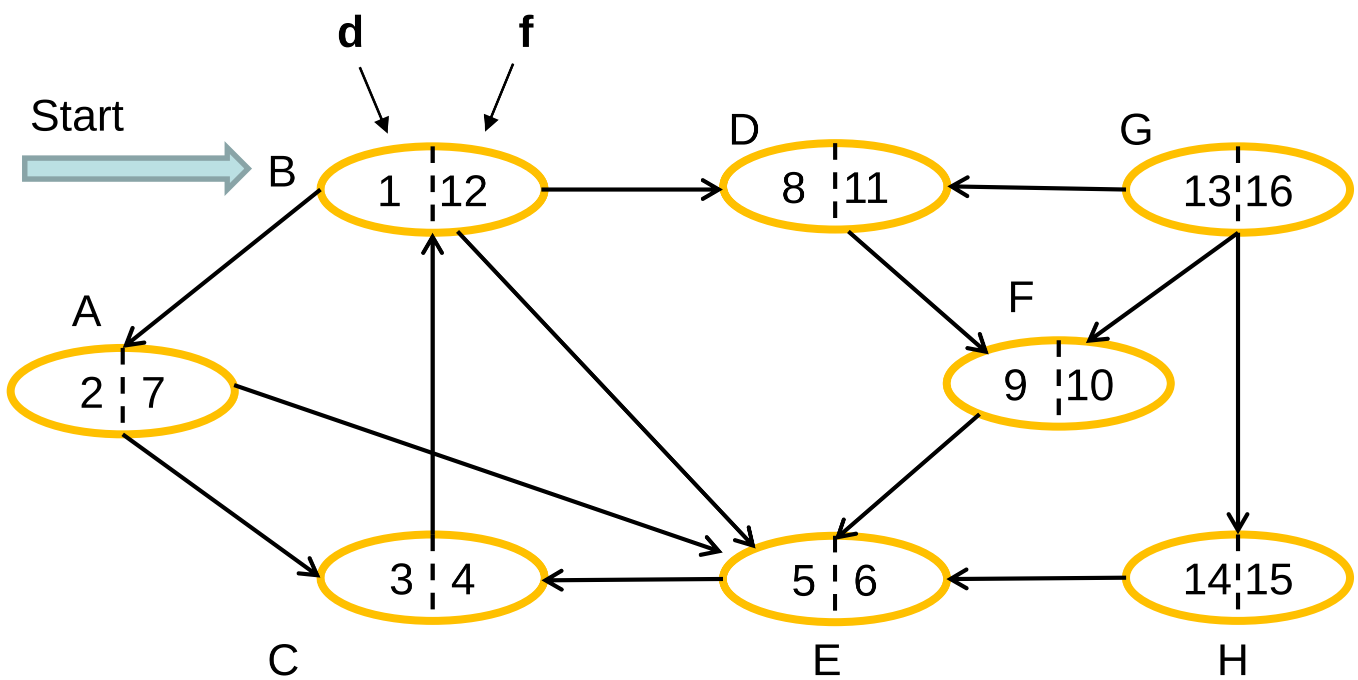
Erst wenn alle Nachbarn entdeckt sind, Knoten schwarz einfärben und Finish Time setzen.

Tiefensuche: Übung

- Führe den Pseudocode auf folgendem Graphen aus.
- Beginne beim markierten Knoten.
- Ergänze die **Discovery *d*** und **Finish Times *f*** ein.
- Die Adjazenzlisten jedes Knoten seien alphabetisch sortiert.



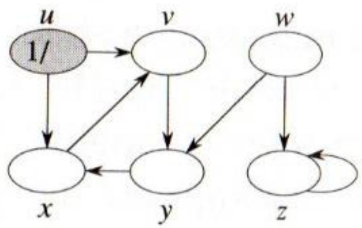
Tiefensuche: Ergebnis der Übung



Diskussion

- **Laufzeit**: $\Theta(|V|+|E|)$
  - Zeile 1-3 benötigt $\Theta(|V|)$
  - DFS-VISIT wird für jeden Knoten genau 1mal aufgerufen.
  - Jede inzidente Kante wird innerhalb dieser Methode besucht.
  - Insgesamt werden alle Kanten besucht.

- Der Beispielgraph ist **nicht zusammenhängend**!
  - ***Bsp***: Knoten *G* von z.B. Knoten *A* und *B* aus **nicht** erreichbar!
  - Der vorgestellte Pseudocode besucht dennoch alle Knoten.
  - Allerdings besteht der Graph der Vorgängerkanten ($v.\pi$) aus mehreren nicht zusammenhängenden Bäumen (=Wald)



(a)

(e)

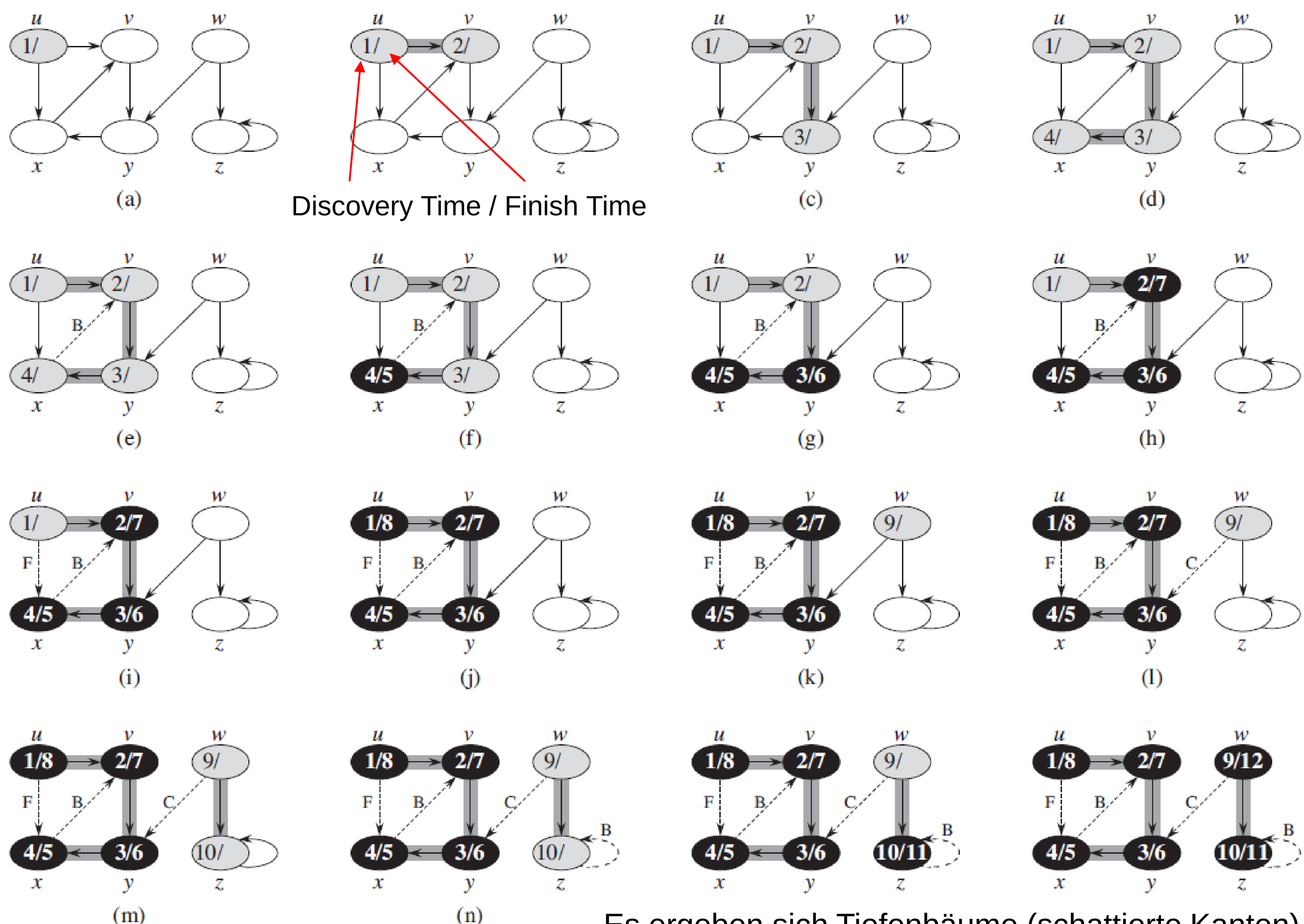
(i)

(m)

(n)

(o)

Schwarz, graue und weiße Knoten: Bedeutung wie bei BFS



Es ergeben sich Tiefenbäume (schattierte Kanten)

Tiefensuche: Diskussion

- Laufzeit: $\Theta(|V|+|E|)$

- Implementierung auch per Stack möglich
  - Jedoch ist die Speicherung der „Finish Time“ etwas komplizierter

- Tiefensuche ist Bestandteil von vielen Algorithmen
  - Auffinden von Zusammenhangskomponenten eines Graphen.
  - Testen eines Graphen auf Kreise.
  - Auflösung von Abhängigkeiten → siehe topologische Sortierung
  - …

- Animation
  - https://www.cs.usfca.edu/~galles/visualization/DFS.html

Übersicht

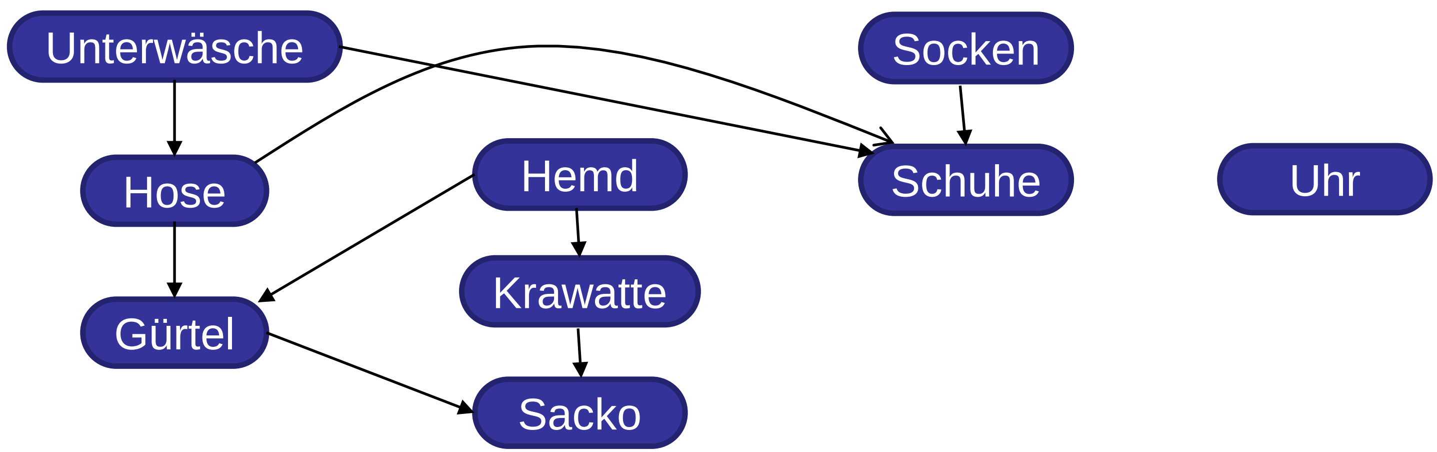
- Graphen als Datenstruktur
  - Definition
  - Speichern von Graphen: Adjazenzliste und Adjazenzmatrix
  - Durchlaufen von Graphen: Breitensuche
  - Durchlaufen von Graphen: Tiefensuche
  - **Topologische Sortierung von gerichteten**

- Kürzeste Wege
  - Siehe Kapitel 8B

Beispiel: Informatikstudent „Genau“

- Student *Genau* überlegt in welcher Reihenfolge er sich ankleiden muss, z.B.
  - erst Socken, dann Schuhe
  - erst Hemd, dann Krawatte

- Modellierung als **gerichteter** Graph



- In welcher zeitlichen Reihenfolge kann er die verschiedenen Kleidungsstücke anziehen?

Topologische Sortierung

- **Idee**
  - Verwende Tiefensuche
  - Knoten, die spät schwarz eingefärbt werden (hohe „Finish Time“), müssen am Anfang der Ordnung stehen
  - „Kleidungsstücke“, die spät schwarz eingefärbt werden, müssen zu einem frühen Zeitpunkt angezogen werden.

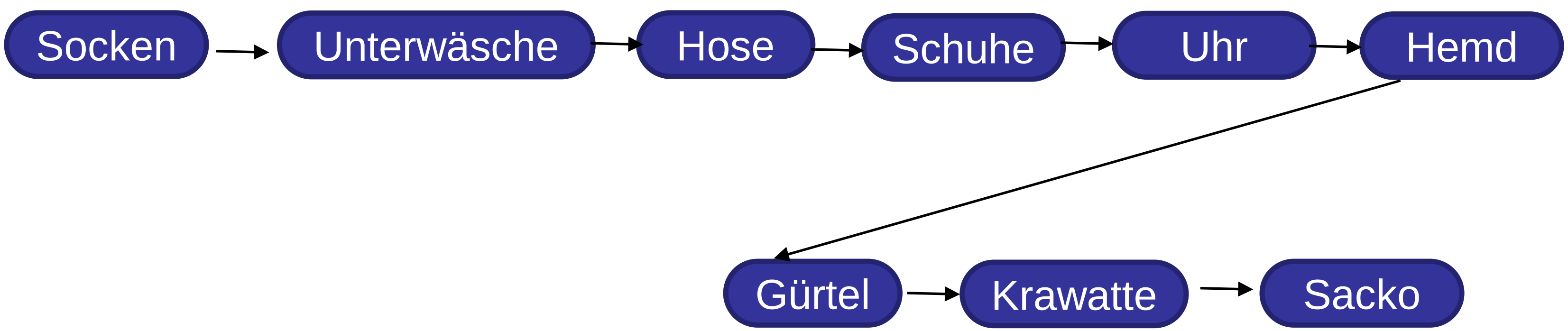
- In der Regel gibt es mehrere, mögliche topologische Sortierungen

- Topologische Sortierung nur möglich, falls gerichteter Graph ***zyklenfrei*** ist.

```
TOPOLOGICAL-SORT(G)
○  Rufe DFS(G) auf
○  Jedes Mal wenn ein Knoten fertig ist (schwarz eingefärbt wird),
   füge ihn vorne in eine Ergebnisliste ein
○  Gib die Liste zurück
```

Lösung: Informatikstudent „Genau“

- Es gibt mehrere mögliche Lösungen!
- Absteigend bzgl. Finish Time durchlaufen ergibt topologische Sortierung.
- Topologische Sortierung nur bei gerichteten Graphen sinnvoll.
- Mögliche Lösung:



Publikums-Joker:

Wie oft wird jeder Knoten *v* bei der Tiefensuche "gesehen"?

A. Einmal

B. Zweimal.

C. indeg(v)-mal.

D. |E|-mal.

Graphen in Java

- Leider bietet die Java-Standard Library keine Graphalgorithmen!

- Mögliche Bibliotheken, z.B.
  - JGraph: http://jgrapht.org/
  - JUNG: http://jung.sourceforge.net/

Zusammenfassung

- Graphen als Datenstruktur
  - Definition
  - Speichern von Graphen: Adjazenzliste und Adjazenzmatrix
  - Durchlaufen von Graphen: Breitensuche
  - Durchlaufen von Graphen: Tiefensuche
  - Topologische Sortierung von gerichteten Graphen
- Kürzeste Wege
  - Definitionen
  - Algorithmus von Bellman-Ford
  - Algorithmus von Dijkstra
  - ADT: Priority Queue

Graphalgorithmen

Quellenverzeichnis

[1] Cormen, Leiserson, Rivest and Stein. *Introduction to Algorithms*, Third Edition, The MIT Press, 2009.

[2] Ottmann, Widmayer. *Algorithmen und Datenstrukturen*, Kapitel 9, 5. Auflage, Spektrum Akademischer Verlag, 2012.

[3] Quelle: http://people.seas.harvard.edu/~babis/amazing.html (abgerufen am 03.12.2016)

[4] Rubik's Cube, *Introduction to Algorithms*, https://courses.csail.mit.edu/6.006/fall11/rec/rec16.pdf (abgerufen am 11.12.2016)

[5] https://commons.wikimedia.org/wiki/File%3ARubiks-Cube.gif (abgerufen am 11.12.2016)