

Informatik II - SS 2014

(Algorithmen & Datenstrukturen)

Vorlesung 14 (25.6.2014)

Graphen: Einführung



**UNI
FREIBURG**

Fabian Kuhn

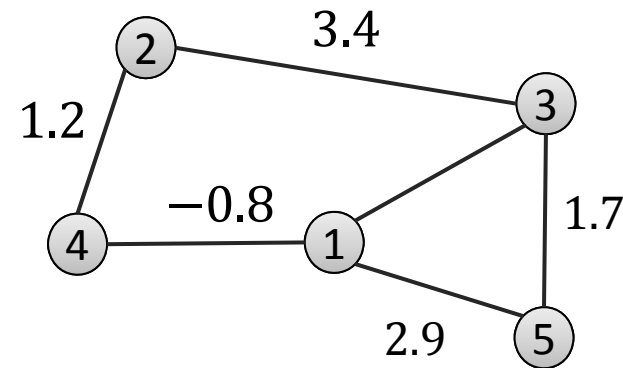
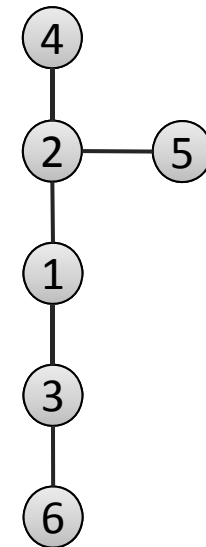
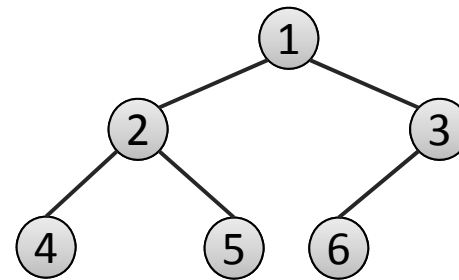
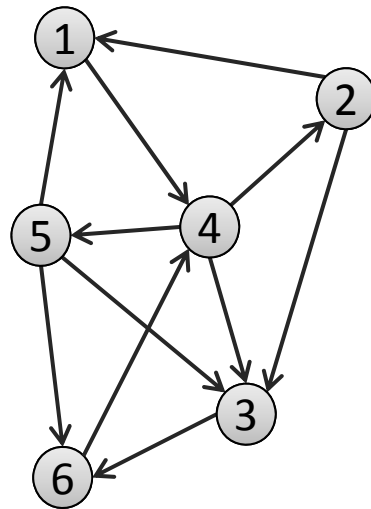
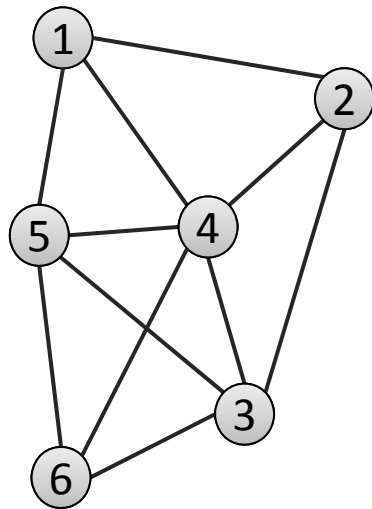
Algorithmen und Komplexität

Graph

Graph $G = (V, E)$:

- Knotenmenge V und Kantenmenge E

Beispiele:



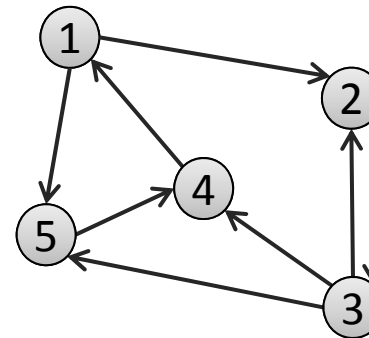
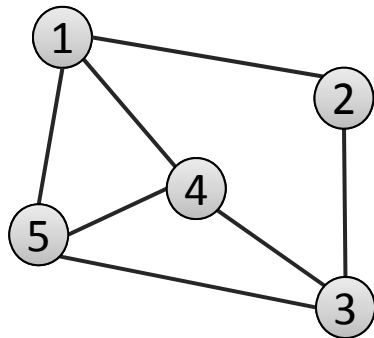
Graph: Notation

Knotenmenge V , typischerweise $n := |V|$

Kantenmenge E , typischerweise $m := |E|$

- ungerichteter Graph: $E \subseteq \{\{u, v\} : u, v \in V\}$
- gerichteter Graph: $E \subseteq V \times V$

Beispiele:



Graph $G = (V, E)$ ungerichtet:

- Grad eines Knoten $u \in V$: Anzahl Kanten (Nachbarn) von u

$$\deg(u) := |\{u, v\} : \{u, v\} \in E|$$

Graph $G = (V, E)$ gerichtet:

- Eingangsgrad eines Knoten $u \in V$: Anzahl eingehende Kanten

$$\deg_{in}(u) := |(v, u) : (v, u) \in E|$$

- Ausgangsgrad eines Knoten $u \in V$: Anzahl ausgehende Kanten

$$\deg_{out}(u) := |(u, v) : (u, v) \in E|$$

Pfade in einem Graph $G = (V, E)$

- Ein Pfad in G ist eine Folge $u_1, u_2, \dots, u_k \in V$ mit
 - gerichteter Graph: $(u_i, u_{i+1}) \in E$ für alle $i \in \{1, \dots, k-1\}$
 - ungerichteter Graph: $\{u_i, u_{i+1}\} \in E$ für alle $i \in \{1, \dots, k-1\}$

Länge eines Pfades (z. T. auch Kosten eines Pfades)

- ohne Kantengewichte: Anzahl der Kanten
- mit Kantengewichten: Summe der Kantengewichte

Kürzester Pfad (shortest path) zwischen Knoten u und v

- Pfad u, \dots, v mit kleinster Länge
- **Distanz** $d(u, v)$: Länge eines kürzesten Pfades zwischen u und v

Durchmesser $D := \max_{u, v \in V} d(u, v)$

- Länge des längsten kürzesten Pfades

Beispiele: Pfade, Distanzen, Durchmesser

Ungerichteter Graph

Beispiele: Pfade, Distanzen, Durchmesser

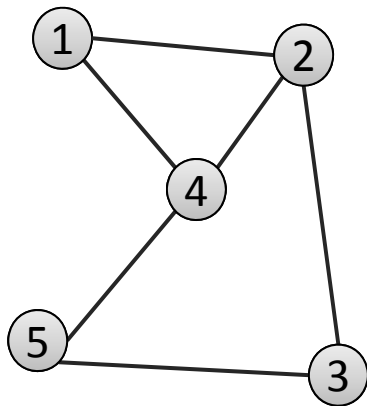
Gerichteter Graph

Repräsentation von Graphen

Zwei klassische Arten, einen Graphen im Rechner zu repräsentieren

- **Adjazenzmatrix:** Platzverbrauch $O(|V|^2) = O(n^2)$
- **Adjazenzlisten:** Platzverbrauch $O(|V| + |E|) = O(n + m)$

Beispiel:



Details:

- Bei **Kantengewichten**, trägt man die Gewichte (statt 0/1) in die Matrix ein (implizit: Gewicht 0 = Kante existiert nicht)
- **Gerichtete Graphen**: Ein Eintrag pro gerichtete Kante
 - Kante von i nach j : Eintrag in Zeile i und Spalte j
- **Ungerichtete Graphen**: Zwei Einträge pro Kante
 - Matrix ist in diesem Fall symmetrisch

Eigenschaften Adjazenzmatrix:

- Speichereffizient, falls $|E| = m \in \Theta(|V|^2) = \Theta(n^2)$
 - speziell für ungewichtete Graphen: nur ein Bit pro Matrixeintrag
- Nicht speichereffizient bei dünn besetzten Graphen ($m \in o(n^2)$)
- Für gewisse Algorithmen, die “richtige” Datenstruktur
- “Kante zwischen u und v ” kann in $O(1)$ Zeit beantwortet werden

Struktur

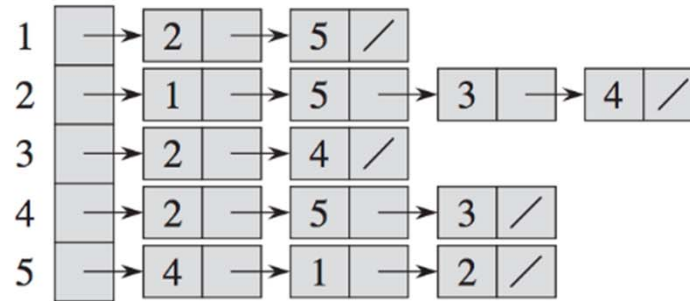
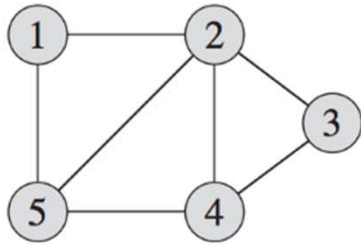
- Ein Array (oder eine verkettete Liste) mit allen Knoten
- Einträge in diesem Knotenarray (oder der Liste):
 - Verkettete Listen (Arrays) mit allen Kanten des Knoten

Eigenschaften

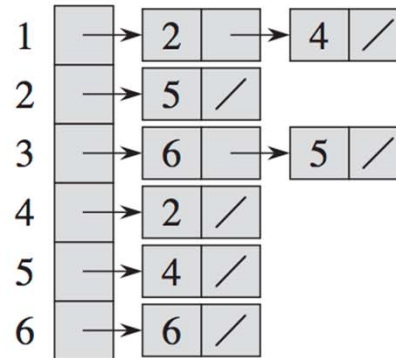
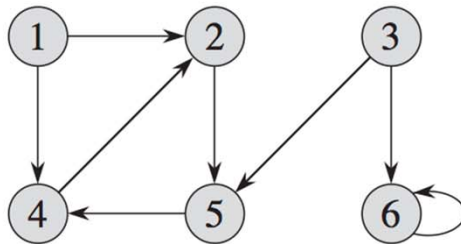
- Speichereffizient bei dünn besetzten Graphen
- Speichereffizienz immer asymptotisch optimal
 - aber bei dicht besetzten Graphen trotzdem deutlich schlechter
- Abfragen nach bestimmten Kanten nicht besonders schnell
 - Falls nötig, eine zusätzliche Datenstruktur (z.B. Hashtabelle) anlegen
- Für viele Algorithmen, die “richtige” Datenstruktur
- Z.B. für Tiefensuche (und Breitensuche)

Beispiele

Beispiele aus [CLRS]:



	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



	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

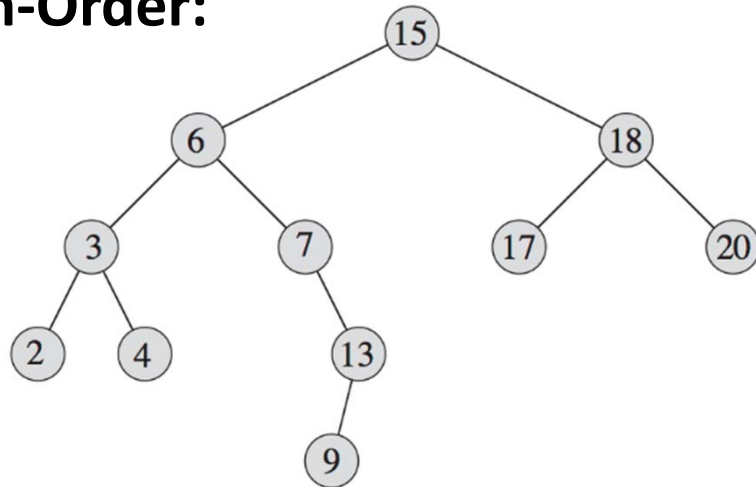
Graph-Traversierung (Graph-Exploration) informell

- Gegeben ein Graph $G = (V, E)$ und ein Knoten $s \in V$, besuche “systematisch” alle Knoten, welche von s aus erreichbar sind.
- Das haben wir bereits bei den Binärbäumen gesehen
- Wie bei den Bäumen gibt es zwei grundsätzliche Verfahren
- **Breitensuche (BFS = breadth first search)**
 - zuerst “in die Breite” (nähere Knoten zu s zuerst)
- **Tiefensuche (DFS = depth first search)**
 - zuerst “in die Tiefe” (besuche alles, was an einem Knoten u “dranhängt”, bevor der nächste Nachbar von u besucht wird)
- Graph-Traversierung ist wichtig, da es oft als Subroutine auftaucht
 - z.B., um die Zusammenhangskomponenten eines Graphen zu finden
 - Wir werden einige Beispiele sehen...

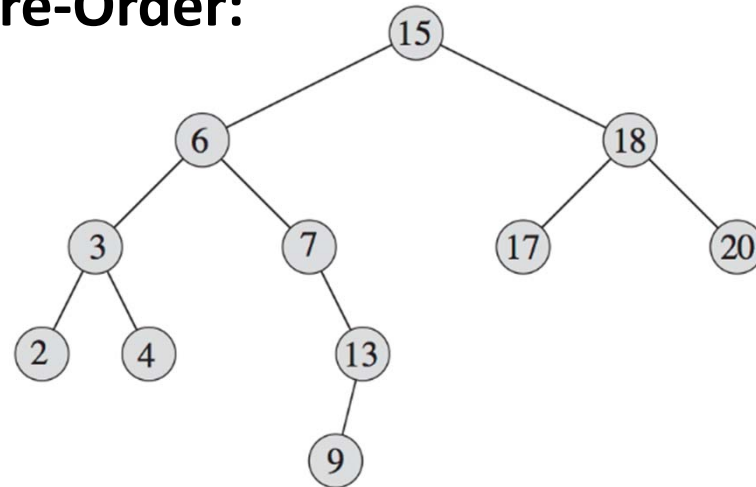
Traversieren eines binären Suchbaums

Ziel: Besuche alle Knoten eines binären Suchbaums einmal

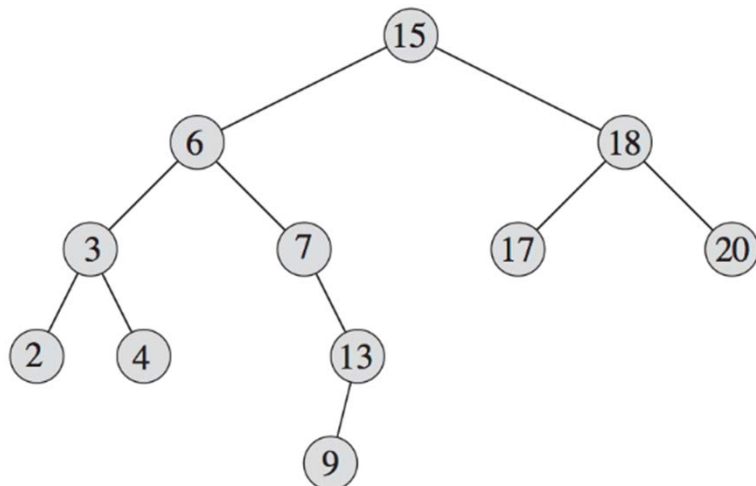
In-Order:



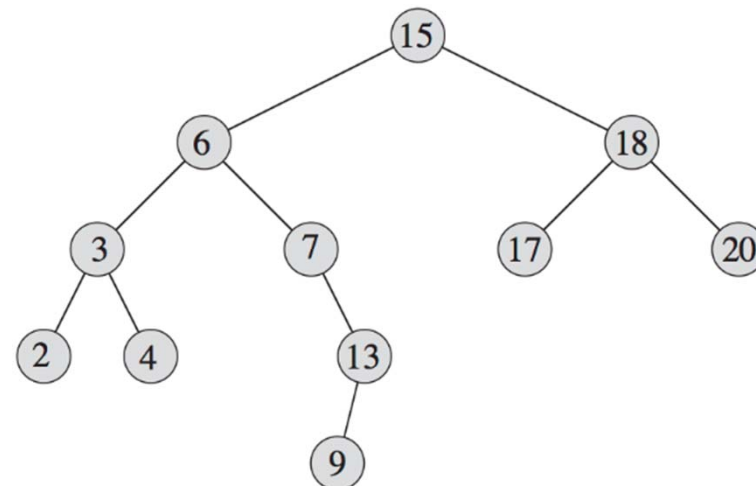
Pre-Order:



Post-Order:



Level-Order:



Tiefensuche / DFS Traversierung

preorder(node):

```
if node != null
    visit(node)
    preorder(node.left)
    preorder(node.right)
```

inorder(node):

```
if node != null
    inorder(node.left)
    visit(node)
    inorder(node.right)
```

postorder(node):

```
if node != null
    postorder(node.left)
    postorder(node.right)
    visit(node)
```

Breitensuche (BFS Traversierung)

- Funktioniert nicht so einfach rekursiv wie die Tiefensuche
- Lösung mit einer Warteschlange:
 - Wenn ein Knoten besucht wird, werden seine Kinder in die Queue eingereiht

BFS-Traversal:

```
Q = new Queue()
Q.enqueue(root)
while not Q.empty() do
    node = Q.dequeue()
    visit(node)
    if node.left != null
        Q.enqueue(node.left)
    if node.right != null
        Q.enqueue(node.right)
```

Unterschiede Binärbaum $T \Leftrightarrow$ allg. Graph G

- Graph G kann Zyklen haben
- In T haben wir eine Wurzel und kennen von jedem Knoten die Richtung zur Wurzel
 - etwas allgemeiner bezeichnen wir solche Bäume auch als gewurzelte Bäume

Breitensuche in Graph G (Start bei Knoten $s \in V$)

- **Zyklen:** markiere Knoten, welche man schon gesehen hat
- **Markiere** Knoten s , hänge s in die Queue
- Wie bisher, nehme immer den ersten Knoten u aus der Queue:
 - **besuche Knoten u**
 - Gehe durch die Nachbarn v von u
 - Falls v nicht markiert, markiere v und hänge v in Queue
 - Falls v markiert ist, muss nichts getan werden

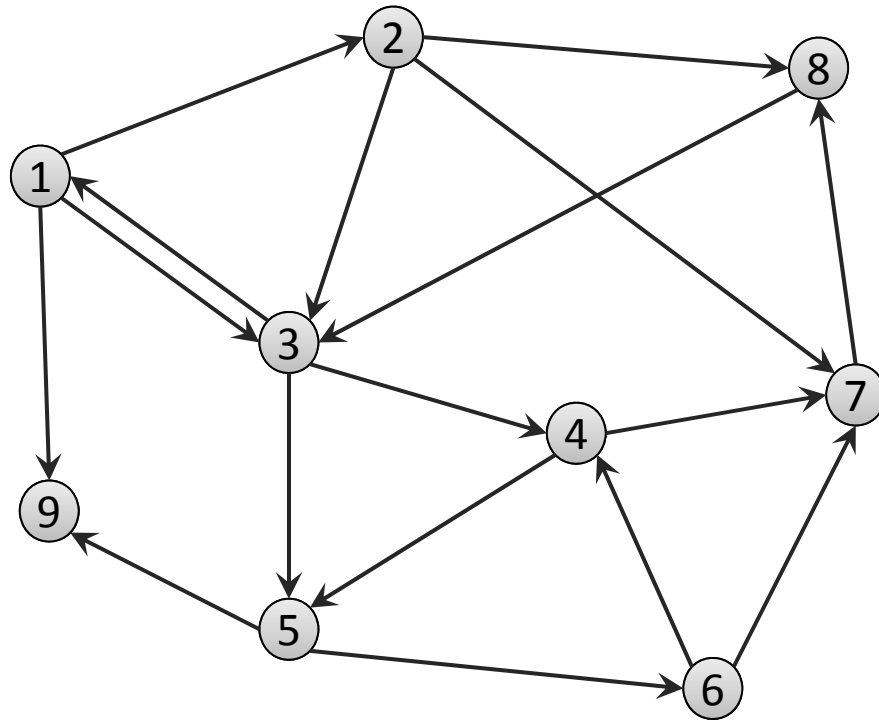
Breitensuche (BFS Traversierung)

- Am Anfang ist v .marked bei allen Knoten v auf **false** gesetzt

BFS-Traversal:

```
for all u in V: u.marked = false;
Q = new Queue()
root.marked = true
Q.enqueue(root)
while not Q.empty() do
    u = Q.dequeue()
    visit(u)
    for v in u.neighbors do
        if not v.marked then
            v.marked = true;
            Q.enqueue(v)
```

Breitensuche Beispiel



Baum:

- Ein zusammenhängender, ungerichteter Graph ohne Zyklen
 - bzw. auch ein gerichteter Graph, aber dann darf es auch ohne Berücksichtigung der Richtungen keine Zyklen haben

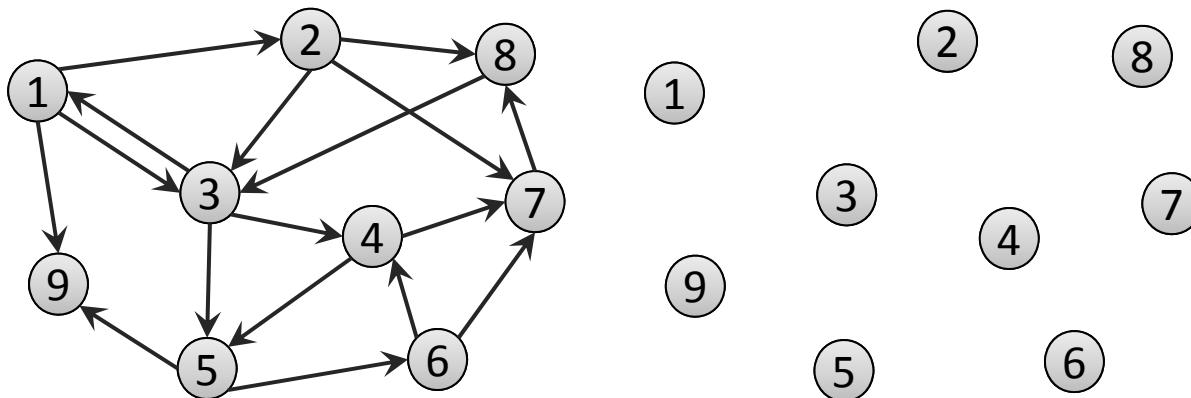
Spannbaum eines Graphen G :

- Ein Teilgraph T , so dass T ein Baum ist, welcher alle Knoten von G enthält
 - Teilgraph: Teilmenge der Knoten und Kanten, welche einen Graph ergeben

BFS Baum

Mit der BFS-Traversierung können wir folgendermaßen einen Spannbaum generieren (falls G zusammenhängend ist):

- Jeder Knoten u merkt sich, von welchem Knoten v aus er markiert wurde
- Der Knoten v ist dann der Parentknoten von u
 - Da jeder Knoten genau einmal markiert wird, ist der Parent von jedem Knoten eindeutig definiert, s ist die Wurzel und hat keinen Parent.



BFS Baum: Pseudocode

- Wir merken uns zusätzlich die Distanz zu s im Baum

BFS-Tree:

```
Q = new Queue();
for all u in V: u.marked = false;
root.marked = true;
root.parent = NULL;
root.d = 0
Q.enqueue(root)
while not Q.empty() do
    u = Q.dequeue()
    visit(u)
    for v in u.neighbors do
        if not v.marked then
            v.marked = true;
            v.parent = u;
            v.d = u.d + 1;
            Q.enqueue(v)
```

Analyse Breitensuche

In der Folge benennen wir die Knoten folgendermassen

- weiße Knoten: Knoten, welche der Alg. noch nicht gesehen hat
- graue Knoten: markierte Knoten
- schwarze Knoten: besuchte Knoten

Die Laufzeit der BFS-Traversierung ist $O(n)$.

Analyse Breitensuche

Im BFS-Baum eines ungewichteten Graphen ist die Distanz von jedem Knoten u zur Wurzel s gleich $d_G(s, u)$.

Grundidee Tiefensuche in G (Start bei Knoten $s \in V$)

- **Markiere Knoten v** (am Anfang ist $v = s$)
- Besuche die Nachbarn von v der Reihe nach *rekursiv*
- Nachdem alle Nachbarn besucht sind, **besuche s**
- **rekursiv:** Beim Besuchen der Nachbarn werden deren Nachbarn besucht, und dabei deren Nachbarn, etc.
- **Zyklen in G :** Besuche jeweils nur Knoten, welche noch nicht markiert sind
- entspricht der Postorder-Traversierung in Bäumen
- Fall man gleich beim Markieren den Knoten besucht, entspricht es der Preorder-Traversierung

Tiefensuche: Pseudocode

DFS-Traversal:

```
for all u in V: u.color = white;  
DFS-visit(root, NULL)
```

DFS-visit(u, p):

```
u.color = gray;  
u.parent = p;  
for all v in u.neighbors do  
    if v.color = white  
        DFS-visit(v, u)  
visit node u;  
u.color = black;
```

Tiefensuche: Beispiel

