

Algorithmen und Datenstrukturen

Vorlesung #07 – Ungewichtete Graphen



Benjamin Blankertz

Lehrstuhl für Neurotechnologie, TU Berlin

benjamin.blankertz@tu-berlin.de

30 · Mai · 2023



Themen der heutigen Vorlesung

- ▶ Grundbegriffe von Graphen
- ▶ Datenstrukturen zur Repräsentation von Graphen
 - ▶ Adjazenzmatrix, AdjazenzListen, Doppelt verkettete Pfeillisten
- ▶ Implementation der Datenstrukturen
- ▶ Graphendurchsuchung, Suchbaum
- ▶ **Tiefensuche** (DFS)
- ▶ Graphen in Zusammenhangskomponenten zerlegen
- ▶ Graphen auf Zyklen prüfen
- ▶ **Breitensuche** (BFS)
- ▶ Pfade mit den wenigsten Kanten finden
- ▶ Klassifikation von Kanten in gerichteten Graphen mit DFS
- ▶ **Topologisches Sortieren**

Begriffe rund um Graphen

Graph

der Ordnung 7

adjazente
Knoten

Knoten
von Grad 3

Kanten

benachbarte
Kante und
Knoten sind
inzident

Spannbaum
alle Knoten,
ohne Zyklen

einfacher Pfad
(kein Knoten doppelt)

Länge 4

gerichteter
Pfad
= gerichteter
Weg

parallele Kanten
(Mehrfachkanten)

reflexive Kante
(Schlinge)

Gerichteter Graph (Digraph)

Quelle
Eingangsgrad=0

Senke
Ausgangsgrad=0

gerichtete Kante
(Pfeil)

gerichteter
Zyklus

Graphen - Einführung der Grundbegriffe

Ein **Graph** bezeichnet eine Vernetzung von Knotenpunkten durch Verbindungen. Damit lassen sich z.B. Computernetzwerke und Straßenkarten modellieren.

- Ein **Graph** ist ein Paar $G = (V, E)$.
- ▶ V ist eine endliche Menge von **Ecken** (*vertices*), bzw. **Knoten** (*nodes*).
- ▶ Jede **Kante** in E verbindet zwei Knoten $e = (v, w)$ mit $v, w \in V$.
- ▶ Man nennt dann v und w **adjacent** (*adjacent*).
- ▶ Und der Knoten v heißt **inzident** (*incident*) mit der Kante e und ebenso heißt e inzident mit v ; ebenso ist natürlich w inzident mit e .
- Wir verwenden $V = |V|$ und $E = |E|$ (nicht fett) für die Anzahl der Knoten bzw. Kanten.
- Als Knoten nehmen wir natürliche Zahlen, normalerweise: $V = \{0, \dots, V-1\}$
Beliebige andere Mengen sind genauso gut möglich (Symbole, Schlüssel, ...).

Definitionen: Teilgraphen, gerichtete und ungerichtete Graphen

- Ein Graph $G' = (V', E')$ ist ein **Teilgraph** von $G = (V, E)$, wenn seine Ecken und Kanten Teilmenge von den Ecken und Kanten von G sind, also $V' \subseteq V$ und $E' \subseteq E$ gilt.
- ▶ In einem 'normalen' Graph haben die Kanten keine Richtung: $(v, w) \equiv (w, v)$. Reflexive Kanten (v, v) sind zugelassen und werden **Schlinge** oder **Schleife** (*self-loop*) genannt.
- Bei einer Kante $e = (v, w)$ in einem **gerichteten Graphen** (*directed graph*), auch **Digraphen** (*digraph*), heißt v Anfangknoten (*head*) und w Endknoten (*tail*). Auch hier sind reflexive Kanten (v, v) zugelassen. Es gibt auch Definitionen von Digraphen, die reflexive Kanten ausschließen.
- ▶ In einem gerichteten Graphen werden die Kanten auch **gerichtete Kanten** oder **Pfeile** (*arcs*) genannt. Wir schreiben auch $v \rightarrow w$ für den Pfeil von v nach w .

Definitionen: Wege und Zyklen in Graphen

- ▶ Ein **Pfad** (*path*) oder **Weg** ist eine Folge von Knoten, die durch Kanten verbunden sind.
- ▶ In einem **einfachen** Pfad kommt jeder Knoten nur einmal vor.
- ▶ Ein **Zyklus** (*cycle*) ist ein Pfad, bei dem Anfangs- und Endknoten übereinstimmen.
- ▶ Bei gerichteten Graphen benutzt man auch die Begriffe **gerichteter Pfad** (*directed path*) und **gerichteter Zyklus**.
- ▶ In einem **einfachen Zyklus** kommen alle Knoten nur einmal vor, bis auf die Übereinstimmung von Anfangs- und Endknoten.
- ▶ Ein Graph, der keine Zyklen enthält, heißt **azyklisch**.
- ▶ Die **Länge** eines Pfads bzw. Zyklus' ist die Anzahl der Kanten.

Formaler: Ein Pfad in $G = (V, E)$ von einem Startknoten s zu einem Zielknoten t ist eine Sequenz v_0, \dots, v_K mit $s = v_0$, $t = v_K$ und $(v_k, v_{k+1}) \in E$ für alle $k < K$.

- ▶ Mit der **Ordnung** eines Graphen bezeichnet man die Anzahl seiner Knoten.
- ▶ Der **Grad eines Knotens** in einem Graphen ist die Anzahl der Kanten, die ihn verbinden.
- ▶ Bei einem Digraphen unterscheidet man **Eingangsgrad** (Anzahl der ankommenden Pfeile) und **Ausgangsgrad** (Anzahl der ausgehenden Pfeile).
- ▶ Eine **Quelle** ist ein Knoten mit Eingangsgrad 0 und Ausgangsgrad > 0 .
- ▶ Eine **Senke** ist ein Knoten mit Ausgangsgrad 0 und Eingangsgrad > 0 .
- ▶ Zwei Kanten heißen **parallel**, wenn sie dieselben beiden Knoten verbinden. Solche Kanten werden auch **Mehrfachkanten** genannt. Bei einem Digraphen muss auch die Richtung übereinstimmen.

Definitionen: Graphen und Zusammenhang

- Ein Graph heißt **zusammenhängend** (*connected*), wenn es zwischen zwei beliebigen Knoten einen Weg gibt.
- Ein nicht-zusammenhängender Graph besteht aus mehreren **Zusammenhangskomponenten** (*connected components*).
- Knoten s und t eines Digraphen heißen **stark verbunden**, wenn es Pfade in beiden Richtungen gibt. Der Pfad von s nach t und der von t nach s darf über dieselben Knoten laufen, wenn es Pfeile in der entsprechenden Richtung gibt.
- Einen *Digraphen* nennt man **stark zusammenhängend** (*strong connected*), wenn alle seine Knoten stark verbunden sind.
- Die Eigenschaft *stark verbunden* ist eine Äquivalenzrelation. Daraus folgt:
- Ein gerichteter Graph, der nicht stark zusammenhängend ist, besteht aus mehreren **starken Zusammenhangskomponenten**.

Definitionen: Graphen und Bäume

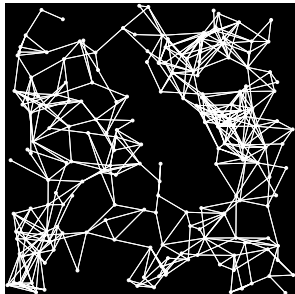
- Ein azyklischer, zusammenhängender Graph ist ein **Baum** (wie im vorigen Semester definiert).
- Eine Menge von Bäumen, die keine gemeinsamen Knoten haben, ergeben einen **Wald**.
- ▶ Ein **Spannbaum** eines Graph G ist ein Teilgraph, der ein Baum ist und alle Knoten enthält.
 - Falls G selbst schon ein Baum ist, so ist er selbst sein einziger Spannbaum. Andernfalls hat jeder Spannbaum von G weniger Kanten als G .

Dichte und dünne Graphen

- Die maximale Anzahl von Kanten E in einem Graphen $G = (V, E)$ ist V^2 , wenn man parallele Kanten ausschließt.
- Man nennt einen Graphen **dünn** (*sparse*), wenn $E \ll V^2$, andernfalls **dicht** (*dense*). Diese Einteilung hat eine Grauzone.

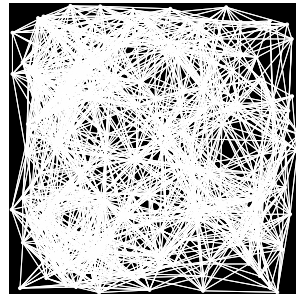
Dünnere Graph

$V = 200$,
 $E = 500$



Dichter Graph

$V = 200$,
 $E = 2000$

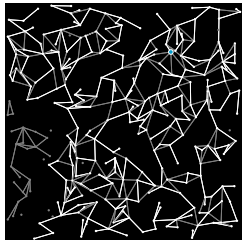


- Die zu erwartende Dichte ist relevant für die Auswahl von Algorithmen und Datenstrukturen in Bezug auf Laufzeit und Speicherbedarf: Für dünne Graphen ist $\mathcal{O}(E)$ deutlich besser als $\mathcal{O}(V^2)$.

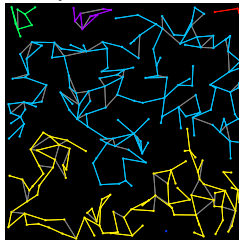
Aufgaben für Graphenalgorithmen

Graphen

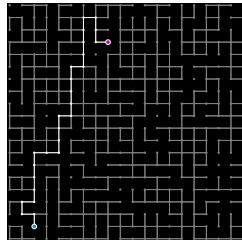
Von s erreichbare
Knoten



Zusammenhangs-
komponenten

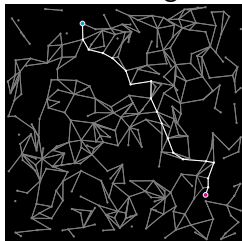


Wege mit wenigen
Kanten

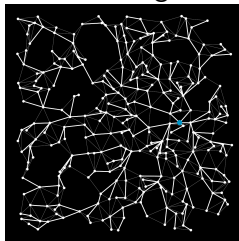


gewichtete
Graphen

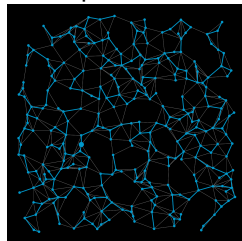
kürzester Weg



kürzeste Wege



Min Spannbaum



API für einen ungerichteten Graphen

public class Graph

| | | |
|--------------------------------------|------------------------------------|-------------------------------------|
| | <code>Graph(int V)</code> | Erzeugt leeren Graphen mit V Knoten |
| <code>int</code> | <code>V()</code> | Anzahl der Knoten |
| <code>int</code> | <code>E()</code> | Anzahl der Kanten |
| <code>void</code> | <code>addEdge(int v, int w)</code> | Füge Kante (v,w) zum Graphen hinzu |
| <code>Iterable<Integer></code> | <code>adj(int v)</code> | Knoten adjazent zu v |

API für einen Digraphen

public class Digraph

| | | |
|--------------------------------------|------------------------------------|---------------------------------------|
| | <code>Digraph(int V)</code> | Erzeugt leeren Digraphen mit V Knoten |
| <code>int</code> | <code>V()</code> | Anzahl der Knoten |
| <code>int</code> | <code>E()</code> | Anzahl der Kanten |
| <code>void</code> | <code>addEdge(int v, int w)</code> | Füge Kante v->w zum Graphen hinzu |
| <code>Iterable<Integer></code> | <code>adj(int v)</code> | Knoten adjazent zu v |

Erinnerung: Kapselung durch Datenabstraktion

- ▶ Die APIs der vorigen Folien sind die Informationen für den Anwender unserer Graphenalgorithmen.
- ▶ Er/sie braucht die Details der Datenstrukturen und Algorithmen nicht zu kennen.
- ▶ Wir als Entwickler hingegen müssen abwägen, welche Datenstrukturen zur Repräsentation von Graphen geeignet sind.

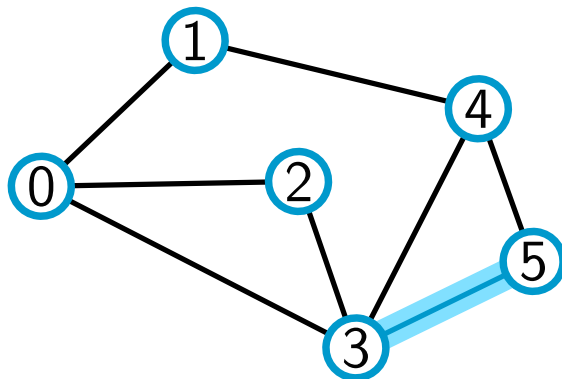
Überlegungen zur Datenstruktur für Graphen

Effiziente Speicherung von Graphen, insbesondere der Kanten E in Hinblick auf Speicherbedarf und Laufzeit der Graphenalgorithmien.

- ▶ Unterschiedliche Möglichkeiten vergleichen und
- ▶ abwägen bezüglich Anwendungsfall (Art der Graphen / Algorithmus).
- ▶ Gebräuchliche Varianten:
 - ▶ Adjazenzmatrix
 - ▶ Adjazenzlisten
 - ▶ Doppelt verkettete Pfeillisten
- ▶ Beliebige Objekten an den Knoten realisiert man mit einem knotenindiziertes Array (Symboltabelle).

Datenstruktur Adjazenzmatrix

Die Menge der Kanten E wird in einer Matrix der Größe $V \times V$ mit Boole'schen Werten gespeichert.



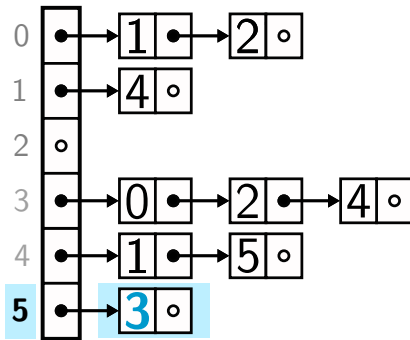
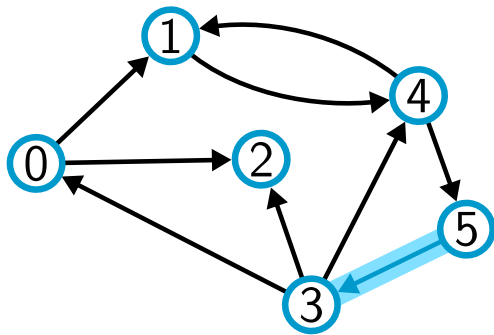
| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| 2 | 1 | 0 | 0 | 1 | 0 | 0 |
| 3 | 1 | 0 | 1 | 0 | 1 | 1 |
| 4 | 0 | 1 | 0 | 1 | 0 | 1 |
| 5 | 0 | 0 | 0 | 1 | 1 | 0 |

- ▶ Bei ungerichteten Graphen ist jede (nicht-reflexive) Kante doppelt gespeichert.
- ▶ Die Matrix ist in diesem Fall also symmetrisch.

- ▶ Speicherbedarf $\Theta(V^2)$, unabhängig von der Anzahl der Kanten.
- ▶ Laufzeit $\mathcal{O}(1)$ zum Prüfen, ob v und w adjazent sind und $\mathcal{O}(V)$ zum Iterieren über die Nachbarknoten von v .
- ▶ Für **dichte** Graphen geeignet, für dünne ineffizient im Speicherbedarf, und meist auch in Bezug auf Laufzeit.

Datenstruktur Adjazenzlisten

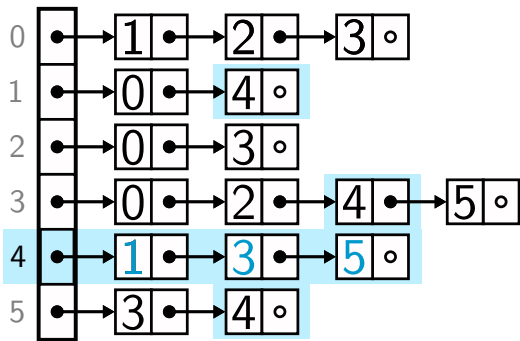
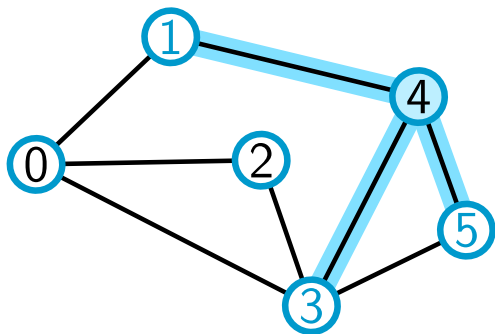
- ▶ Liste $\text{adj}[v]$ mit zu v adjazenten Knoten
- ▶ Gerichtete Graphen: nur ausgehende Pfeile berücksichtigen



- ▶ Listeneinträge: Zielknoten und Zeiger auf folgenden Listeneintrag. Der letzte Eintrag enthält als Zeiger null.
- ▶ Die Reihenfolge in der Liste ist beliebig.
- ▶ Bei ungerichteten Graphen ist jede (nicht-reflexive) Kante doppelt gespeichert.

Datenstruktur Adjazenzlisten

- ▶ Liste $\text{adj}[v]$ mit zu v adjazenten Knoten
- ▶ Gerichtete Graphen: nur ausgehende Pfeile berücksichtigen



- ▶ Listeneinträge: Zielknoten und Zeiger auf folgenden Listeneintrag. Der letzte Eintrag enthält als Zeiger null.
- ▶ Die Reihenfolge in der Liste ist beliebig.
- ▶ Bei ungerichteten Graphen ist jede (nicht-reflexive) Kante doppelt gespeichert.

- ▶ Speicherbedarf $\Theta(V + E)$.
- ▶ Laufzeit:
 - $\mathcal{O}(\text{grad}(v))$ zum Prüfen, ob v und w adjazent sind und
 - $\mathcal{O}(\text{grad}(v))$ zum Iterieren über die Nachbarknoten von v .
- ▶ Effizient für Graphen mit **wenigen Kanten**, da die Iteration über Nachbarknoten schnell ist.
- ▶ Weniger effizient für das Entfernen und Hinzufügen von **Knoten**.

Datenstruktur doppelt verkettete Pfeillisten

Für Graphen, bei denen dynamisch Knoten hinzugefügt und entfernt werden können, bieten sich **doppelt verkettete Pfeillisten** (*doubly connected arc list; DCAL*) an.

- ▶ Knoten werden als doppelt verkettete Listen gespeichert.
 - ▶ Listenelemente enthalten zwei Zeiger auf Vorgänger und Nachfolger und einen auf eine Pfeilliste.
- ▶ Jedes Element der Pfeilliste enthält drei Einträge:
 - ▶ Rückwärtsverkettung: Zeiger auf vorige Kante bzw. auf Knotenelement
 - ▶ Zeiger zum Endknoten des Pfeils
 - ▶ Vorwärtsverkettung: Zeiger auf nächste Kante bzw. null

Übersicht der Datenstrukturen für Graphen

- ▶ **Adjazenzmatrix**: ggf. geeignet für dichte statische Graphen
- ▶ **Adjazenzlisten**: gut geeignet für dünne statische Graphen
- ▶ **Doppelt verkettete Pfeillisten**: gut für dünne dynamische Graphen
- ▶ **Dynamischer Graph**: Hinzufügen und Entfernen von Knoten möglich, im Gegensatz zu **statischem Graphen**.

Entscheidung: Wir verwenden Adjazenzlisten. Beachte: Für dynamisch veränderbare Graphen sollten DCAL benutzt werden.

Implementation der Datenstruktur Graph

```
public class Graph
{
    private final int V;
    private int E;
    private Bag<Integer>[] adj;

    public Graph(int V)
    {
        this.V = V;
        adj = (Bag<Integer>[]) new Bag[V];
        for (int v = 0; v < V; v++)
            adj[v] = new Bag<Integer>();
    }

    public int V() { return V; }
    public int E() { return E; }
```

```
    public void addEdge(int v, int w)
    {
        adj[v].add(w);
        adj[w].add(v);
        E++;
    }

    public Iterable<Integer> adj(int v)
    {
        return adj[v];
    }
}
```

Bag: LinkedList, die Iterable implementiert und nur Methoden add() und size() hat.

Implementation der Datenstruktur Digraph

```
public class Digraph {  
    private final int V;  
    private int E;  
    private Bag<Integer>[] adj;  
  
    // Alles exakt wie bei Graph (außer Namen des Konstruktors und)  
    // die Methode addEdge:  
  
    public void addEdge(int v, int w) {  
        adj[v].add(w);           // Kante nur in einer Richtung hinzufügen  
        E++;  
    }  
  
    public Digraph reverse() {    // neue Methode nur für Digraph  
        Digraph R = new Digraph(V);  
        for (int v = 0 ; v < V; v++)  
            for (int w : adj[v])  
                R.addEdge(w, v);  
        return R;  
    }  
}
```

Als erstes Beispiel für einen Graphenalgorithmus betrachten wir das

Durchsuchen eines Graphen

Zu einem gegebenen Knoten s eines Graphen oder Digraphen G sollen alle von s in G erreichbaren Knoten markiert werden.

Als Erweiterungen betrachten wir folgende Aufgaben:

- ▶ Gebe zu jedem erreichbaren Knoten einen Weg von s zurück.
- ▶ Suche den kürzesten Weg von s zu allen möglichen Zielknoten.
- ▶ Identifiziere alle Zusammenhangskomponenten von G .
- ▶ Teste, ob G einen Zyklus enthält.
- ▶ Gebe die erreichbaren Knoten in speziellen Reihenfolgen, die noch festzulegen sind, zurück.

Allgemeiner Ansatz zur Graphendurchsuchung

Ganz allgemein kann der Ansatz, um $G = (V, E)$ ausgehend von s aus zu durchsuchen, folgendermaßen formuliert werden:

Listing 1: Graphendurchsuchung

```
1  $M \leftarrow \{s\}$            //  $M$ : markierte Knoten (marked)
2 while there is  $(v, w) \in E$  with  $v \in M$  and  $w \notin M$  do
3   add  $w$  to  $M$ 
4 end
```

- ▶ Die Menge der markierten Knoten M wird von $\{s\}$ schrittweise erweitert.
- ▶ Dabei wird immer eine neue ‘**kreuzende**’ Kante (v, w) benutzt, d. h. eine Kante, die **den markierten Bereich transzendiert**: v ist in M und w ist außerhalb von M .
- ▶ Zeile 3 wird für jeden erreichbaren Knoten einmal ausgeführt. Abgesehen von der Auswahl von v und w zur Erfüllung der **while** Bedingung ist die Laufzeit in $\mathcal{O}(V)$.
- ▶ Kritischer Punkt: (effiziente) Wahl der nächsten Kante (v, w) .

Die benutzten Kanten ergeben einen Baum

Suchbaum

Die von dem Algorithmus in Listing 1 benutzen Kanten (v, w) , also diejenigen, die die Bedingung in Zeile 2 erfüllen, ergeben einen Baum. Er wird **Suchbaum** genannt.

- ▶ **Beweis.** Wir nehmen an, dass es vom Algorithmus benutzte Kanten (v_k, v_{k+1}) für $k < K$ gibt, die einen Zyklus ergeben: $v_0 = v_K$.
- ▶ Sei (v_i, v_{i+1}) die bei der Graphdurchsuchung als letzte hinzugefügte Kante.
- ▶ Damit die Bedingung aus Zeile 2 erfüllt ist, muss $v_i \in M$ und $v_{i+1} \notin M$ gelten.
- ▶ Dann kann aber die von v_{i+1} ausgehende Kante nicht vom Algorithmus benutzt worden sein. \square

Wenn der Graph zusammenhängend ist, also alle Knoten von s erreichbar sind, dann ist der Suchbaum ein **Spannbaum** des Graphen.

Strategien zur Auswahl der Knoten

- ▶ Zwei grundlegende Ansätze zur Auswahl der Knoten:
- ▶ Gehe rekursiv in die Tiefe. Wenn es nicht weiter geht, gehe einen Rekursionsschritt zurück und mache dasselbe für die anderen Nachbarknoten (wie beim *Backtracking*). Dies führt zur **Tiefensuche**.
- ▶ Besuche zunächst alle Knoten mit Abstand 1 vom Startknoten, dann die mit Abstand 2 usw. Dies führt zur **Breitensuche**.

Einfluss der Suchstrategie

- ▶ Die Menge der Knoten A , die der Algorithmus in Listing 1 zurückgibt, ist **unabhängig** von der Suchstrategie. Es sind immer alle von s in G erreichbaren Knoten.
- ▶ Die **Reihenfolge**, in der die Knoten M hinzugefügt werden, hängt sehr wohl von der Suchstrategie ab und von der Reihenfolge in den Adjazenzlisten.
- ▶ Daraus ergibt sich dieselbe Abhängigkeit für den **Suchbaum** (siehe Beweis auf S. 25). Die Menge seiner Knoten ist fix (siehe erster Punkt), aber seine Struktur (welche Knoten durch Kanten verbunden sind) ist es nicht.

Pseudocode für Tiefensuche (DFS) mit Rekursion

```
1  $M \leftarrow \emptyset$ 
2  $dfs(G, 0)$ 
3
4 procedure  $dfs(G, v)$ 
5   add  $v$  to  $M$  //  $v$  wurde entdeckt
6   for each  $w$  with  $(v, w) \in E$  and  $w \notin M$ 
7      $dfs(G, w)$  // folge Kante  $(v, w)$ 
8   end //  $v$  fertig bearbeitet
```

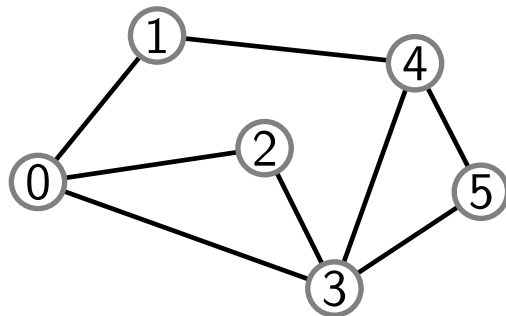
$v =$

$w =$

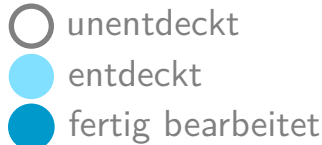
$M = \{\}$

$(v, w) =$

recursion stack:



Entdeckte Knoten
als Baum:



Pseudocode für Tiefensuche (DFS) mit Rekursion

```
1  $M \leftarrow \emptyset$   
2  $dfs(G, 0)$   
3  
4 procedure  $dfs(G, v)$   
5   add  $v$  to  $M$  //  $v$  wurde entdeckt  
6   for each  $w$  with  $(v, w) \in E$  and  $w \notin M$   
7      $dfs(G, w)$  // folge Kante  $(v, w)$   
8 end //  $v$  fertig bearbeitet
```

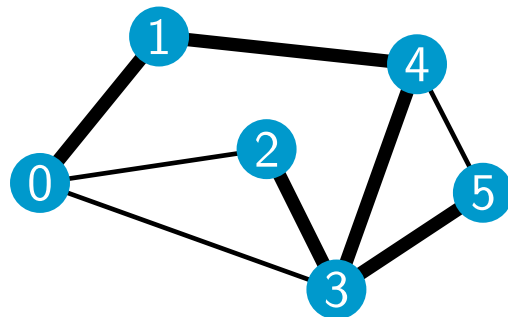
$v = 0$

$w = 5$

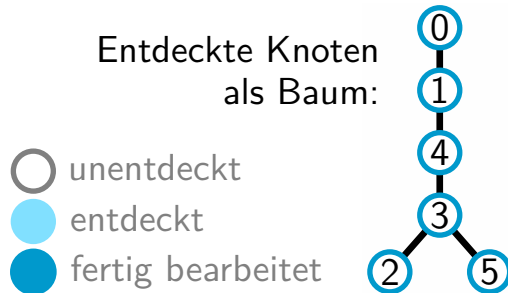
$M = \{0, 1, 4, 3, 2, 5\}$

$(v, w) = (0, 1), (1, 4), (4, 3), (3, 2), (3, 5)$

recursion stack: $dfs(G, 0), dfs(G, 1),$
 $dfs(G, 4), dfs(G, 3), dfs(G, 2), dfs(G, 5)$



Entdeckte Knoten
als Baum:



Anforderungen an M (*markierte Knoten*):

- ▶ $(1 \times)$ Leer initialisieren (Zeile 1) $O(V)$
 - ▶ $(V \times)$ Elemente hinzufügen (Zeile 5) jeweils $O(1)$
 - ▶ $(E \times)$ Prüfen, ob ein Element zu M gehört (Zeile 6) jeweils $O(1)$
- ⇒ Implementiere M als **Boole'sches Array** der Größe V

Listing 2: Tiefensuche: markiert von s erreichbare Knoten

```
1 public class DepthFirstPaths {  
2     private boolean[] marked;  
3  
4     public DepthFirstPaths(Graph G, int s) {  
5         marked = new boolean[G.V()];  
6         dfs(G, s);  
7     }  
8  
9     public void dfs(Graph G, int v) {  
10        marked[v] = true;  
11        for (int w : G.adj(v)) {  
12            if (!marked[w]) {  
13                dfs(G, w);  
14            }  
15        }  
16    }  
17 }
```

(Demo: animierte Tiefensuche)

Korrektheit des Programms zur Tiefensuche

Das Programm in Listing 2 markiert genau die von s aus erreichbaren Knoten.

- ▶ Es werden nur Knoten über Pfade ausgehend von s markiert. (klar)
- ▶ *Zu zeigen:* Wenn es einen Pfad von s zu einem Knoten t gibt, so wird t markiert.
 - Nehmen wir an, t würde nicht markiert. Auf dem Pfad von s nach t muss es dann eine Kante (v, w) geben, für die v markiert ist, w aber nicht (Übergang vom markierten Bereich in den nicht markierten Bereich).
 - In diesem Fall hätte die Tiefensuche nachdem v markiert wurde (Zeile 10) auch w markiert (w ist in der Adjazenzliste von v , Zeile 11, und nach unserer Annahme nicht markiert, also (Zeile 12) würde $\text{dfs}(G, w)$ in Zeile 13 aufgerufen und damit w markiert.)
 - So führt die Annahme zu einem Widerspruch. \square

Die Terminierung wird von der folgenden Laufzeitbetrachtung impliziert.

Betrachtung der Laufzeit von DFS

- ▶ Da die Methode `dfs` nur für unmarkierte Knoten aufgerufen wird, wird jeder Knoten höchstens einmal markiert.
- ▶ Die `for` Schleife in Zeile 11 wird also höchstens für jeden Knoten v einmal aufgerufen und läuft dann über alle seine Nachbarn w . Der Rumpf (Zeile 12-14) wird also höchstens einmal für jede Kante $(v, w) \in E$ aufgerufen.
- ▶ Dabei kann eine Kante eines ungerichteten Graphen zweimal betrachtet werden, von jeder Seite einmal. Der Faktor 2 spielt für die Wachstumsordnung keine Rolle.
- ▶ Somit ist die Laufzeit von `dfs` in $\mathcal{O}(E)$.
- ▶ Die Initialisierung von `marked` in Zeile 5 benötigt eine Laufzeit in $\mathcal{O}(V)$.
- ▶ Insgesamt ist die Laufzeit der Tiefensuche in $\mathcal{O}(V + E)$.

Pfade mit Tiefensuche finden

- ▶ Das Programm markiert alle erreichbaren Knoten.
- ▶ Es sollte zu jedem erreichbaren Zielknoten einen Pfad zurück geben können.

Pfade mit einem Startknoten *single-source paths*

Zu einem gegebenen Graphen G und Startknoten s , soll für alle Knoten v entschieden werden, ob sie von s erreichbar sind und falls ja, soll ein Pfad zurückgegeben werden.

- ▶ Alle benutzen Pfade ergeben einen Baum (siehe S. 25).
- ▶ Also können sie gemeinsam in einem Knoten-indizierten Array gespeichert werden (Jeder Knoten verweist auf Elternknoten).
- ▶ Pfad abzulesen: vom Zielknoten zum rückwärts zum Startknoten.

Pfade mit Tiefensuche finden

```
public class DepthFirstPaths {  
    private boolean[] marked;  
    private int[] parent;  
    private final int s;  
  
    public DepthFirstPaths(Graph G, int s) {  
        marked = new boolean[G.V()];  
        parent = new int[G.V()];  
        this.s = s;  
        dfs(G, s);  
    }  
  
    public void dfs(Graph G, int v) {  
        marked[v] = true;  
        for (int w : G.adj(v))  
            if (!marked[w]) {  
                parent[w] = v;  
                dfs(G, w);  
            }  
    }  
}
```

Neue Methoden:

```
public boolean hasPathTo(int v) {  
    return marked[v];  
}  
  
// Um den Pfad von s nach v zurückzugeben,  
// geht man von v solange zum Vorgänger bis  
// man bei s ankommt. Durch Benutzung eines  
// Stapels ist der Pfad in richtiger  
// Reihenfolge im Iterator.  
public Iterable<Integer> pathTo(int v) {  
    if (!hasPathTo(v))  
        return null;  
    Stack<Integer> path = new Stack<>();  
    for (int w = v; w != s; w = parent[w])  
        path.push(w);  
    path.push(s);  
    return path;  
}
```

- ▶ Mit einer anderen kleinen Erweiterung, kann mit Tiefensuche die folgende Aufgabe gelöst werden:

Graphen in Zusammenhangskomponenten zerlegen

Ein gegebener Graph soll in seine Zusammenhangskomponenten zerlegt werden. Dazu soll jedem Knoten die Nummer seiner Komponente zugeordnet werden.

- ▶ **Lösung:**
 - Durch Tiefensuche mit beliebigem Startknoten werden alle erreichbaren Knoten mit ID #1 gekennzeichnet.
 - Dann wird iterativ die ID erhöht und der Vorgang wiederholt, ausgehend von dem nächsten Knoten, der noch ohne ID ist.
- ▶ Die Zerlegung in **starke Zusammenhangskomponenten** in einem gerichteten Graphen ist deutlich schwieriger, siehe Bemerkung auf S. 56.

Zusammenhangskomponenten mit Tiefensuche identifizieren

```
public class ConnectedComponents {  
    private int count;  
    private int[] id;  
  
    public ConnectedComponents(Graph G) {  
        id = new int[G.V()];  
        for (int v = 0; v < G.V(); v++) {  
            if (id[v] == 0) {  
                count++;  
                dfs(G, v);  
            }  
        }  
    }  
}
```

```
public void dfs(Graph G, int v) {  
    id[v] = count;  
    for (int w : G.adj(v)) {  
        if (id[w] == 0)  
            dfs(G, w);  
    }  
}  
  
public int count() { return count; }  
  
public int id(int v) { return id[v]; }  
  
public boolean connected(int v, int w)  
{ return id[v] == id[w]; }  
}
```

- id übernimmt auch die Rolle von marked via $\text{marked} \Leftrightarrow (\text{id} > 0)$.

Graph auf Zyklen prüfen

- ▶ Mit einer anderen kleinen Änderung kann man mit Tiefensuche prüfen, ob ein Graph Zyklen enthält.

Zykluserkennung

Zu einem gegebenen Graphen (ohne Schleifen und Mehrfachkanten) soll entschieden werden, ob er einen Zyklus enthält.

▶ Lösung:

- ▶ Falls Zyklus vorhanden, trifft Tiefensuche auf einen schon markierten Knoten.
- ▶ Bei einem ungerichteten Graphen sind alle Kanten doppelt vorhanden.
- ▶ Daher prüfe, ob der markierte Knoten nicht derjenige ist, von dem man gekommen ist.
- ▶ Für gerichtete Graphen ist die Prüfung auf gerichtete Zyklen etwas aufwendiger.

Zyklen mit Tiefensuche erkennen

```
1 public class Cycle
2 {
3     private boolean[] marked;
4     private int[] parent;
5     public boolean hasCycle;
6
7     public Cycle(Graph G)
8     {
9         marked = new boolean[G.V()];
10        parent = new int[G.V()];
11        for (int v = 0; v < G.V(); v++)
12            if (!marked[v])
13                dfs(G, v);
14    }
15    public void dfs(Graph G, int v)
16    {
17        marked[v] = true;
18        for (int w : G.adj(v))
19            if (!marked[w]) {
20                parent[w] = v;
21                dfs(G, w);
22            } else if (parent[v] != w) {
23                hasCycle = true;
24            }
25    }
26 }
```

Um den gefundenen Zyklus zurückzugeben, kann der folgende Code Zeile 23 ersetzen:

```
cycle = new Stack<Integer>();
cycle.push(w);
for (int u = v; u != w; u = parent[u])
    cycle.push(u);
cycle.push(w);
```

- ▶ In diesem Fall sollte die Bedingung der if Anweisung in Zeile 22 durch `&& cycle != null` ergänzt werden und
- ▶ `cycle` als Objektvariable deklariert werden.
- ▶ Die Variable `hasCycle` wird überflüssig, da dies durch `cycle != null` geprüft werden kann.

Zweiter Ansatz zur Graphendurchsuchung

- ▶ Die **Breitensuche** durchläuft den Graphen schichtweise.
- ▶ Zunächst werden alle Knoten besucht, die den Abstand 1 zum Startknoten s haben, dann die Knoten mit Abstand 2 usw.
- ▶ Allen erreichbaren Knoten kann so ihr Abstand zu s zugeordnet werden.

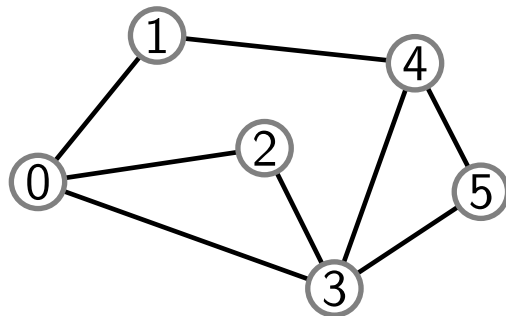
■ Implementierung:

- ▶ Ausgehend von s markiere alle Nachbarknoten, die noch nicht markiert sind, und schiebe sie in eine Warteschlange.
- ▶ Mach so weiter mit den Knoten in der Warteschlange (FIFO), bis sie leer ist.

Programmablauf Breitensuche (BFS)

```
1 //  $Q$ : Queue,  $M$ : Array der Größe  $V$   
2  $s \leftarrow 0$   
3  $M \leftarrow \{s\}$   
4  $\text{Enqueue}(Q, s)$   
5 while  $Q \neq \emptyset$  do  
6    $v \leftarrow \text{Dequeue}(Q)$   
7   for each  $w$  with  $(v, w) \in E$  and  $w \notin M$   
8     add  $w$  to  $M$  //  $w$  wurde entdeckt  
9      $\text{Enqueue}(Q, w)$   
10  end //  $v$  fertig bearbeitet  
11 end
```

$v =$
 $w =$
 $M = \{\}$
 $Q = \langle \rangle$
 $(v, w) =$



Entdeckte Knoten
als Baum:

Programmablauf Breitensuche (BFS)

```
1 // Q: Queue, M: Array der Größe V
2  $s \leftarrow 0$ 
3  $M \leftarrow \{s\}$ 
4 Enqueue(Q, s)
5 while  $Q \neq \emptyset$  do
6    $v \leftarrow \text{Dequeue}(Q)$ 
7   for each  $w$  with  $(v, w) \in E$  and  $w \notin M$ 
8     add  $w$  to  $M$  //  $w$  wurde entdeckt
9     Enqueue(Q, w)
10  end //  $v$  fertig bearbeitet
11 end
```

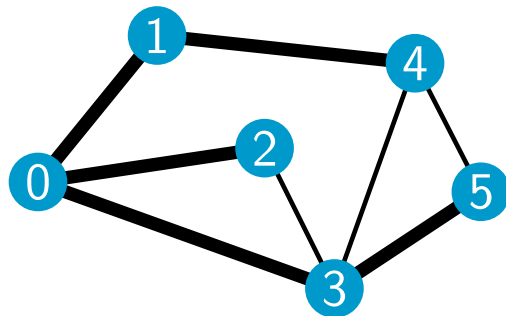
$v = 5$

$w = 5$

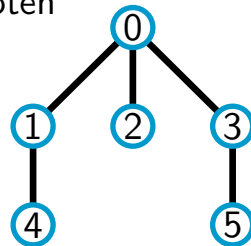
$M = \{0, 1, 2, 3, 4, 5\}$

$Q = \langle \rangle$

$(v, w) = (0, 1), (0, 2), (0, 3), (1, 4), (3, 5)$



Entdeckte Knoten
als Baum:

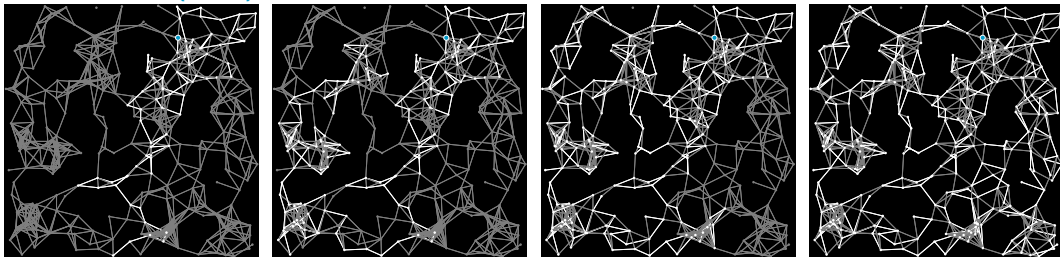


Betrachtung der Laufzeit von BFS

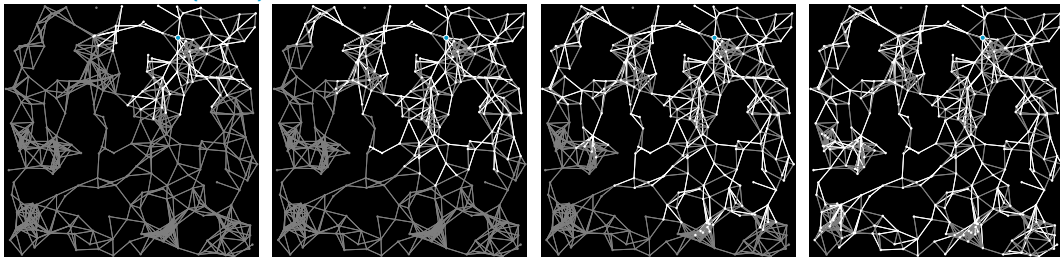
- ▶ Jeder Knoten wird **nur einmal** zu M hinzugefügt, denn Zeile 8 wird nur ausgeführt, wenn $w \notin M$ erfüllt ist.
- ▶ Jeder Knoten wird **einmal** in die Schlange Q eingefügt und herausgeholt.
- ▶ Die *for* Schleife in Zeile 7 wird also für jeden Knoten v einmal aufgerufen.
- ▶ Für jeden Knoten v werden alle benachbarten Knoten w durchlaufen und auf $w \notin M$ geprüft.
- ▶ Dies geschieht für jede Kante nur einmal, also ist die Laufzeit dieses Programmtails in $\mathcal{O}(E)$. (Die Queue Operationen haben konstante Laufzeit!)
- ▶ Bei einem ungerichteten Graphen kann jede Kante von beiden Seiten benutzt werden. Aber der konstante Faktor 2 spielt für die Wachstumsordnung keine Rolle.
- ▶ Die Initialisierung des Arrays M ist in $\mathcal{O}(V)$.
- ▶ Insgesamt ist die Laufzeit der Breitensuche also in $\mathcal{O}(V + E)$.

Reihenfolge und Baumstruktur bei Breiten- und Tiefensuche

Tiefensuche (DFS)



Breitensuche (BFS)



Pfade mit wenigsten Kanten mit Breitensuche finden

- ▶ Die Breitensuche auf S. 40 markiert alle erreichbaren Knoten.
- ▶ Auf Grund der schichtweisen Durchsuchung gemäß Abstand von s , eignet sich die Breitensuche für die folgende Aufgabe:

Pfade mit minimaler Kantenanzahl von einem Startknoten

Zu einem gegebenen Graphen G und Startknoten s , soll für alle erreichbaren Knoten v ein Pfad mit minimaler Kantenanzahl zurückgegeben werden.

- ▶ Vorsicht bei dem Begriff **kürzeste Pfade**: Er hat bei *gewichteten* Graphen eine spezielle Bedeutung.
- ▶ Zum Speichern der Pfade benutzen wir dieselbe Strategie wie bei der Tiefensuche.

BFS zum Finden von Pfaden mit den wenigsten Kanten

```
1 //Q: Queue
2 //parent: Array of size V
3  $M \leftarrow \{s\}$ 
4  $dist[s] \leftarrow 0$ 
5 Enqueue(Q, s)
6 while  $Q \neq \emptyset$  do
7      $v \leftarrow Dequeue(Q)$ 
8     for each  $w$  with  $(v, w) \in E$  and  $w \notin M$ 
9         add  $w$  to  $M$ 
10         $parent[w] \leftarrow v$ 
11         $dist[w] \leftarrow dist[v] + 1$ 
12        Enqueue(Q, w)
13 end
14 end
```

- ▶ Der Pfad zu einem gegebenen Zielknoten kann genau wie bei der Tiefensuche aus `parent` konstruiert werden, siehe S. 34, Methode `pathTo`.
- ▶ Die Distanz (Anzahl der Kanten im kürzesten Weg) von s zu v wird gespeichert in `dist[v]`.

Korrektheit des Programms zur Breitensuche für Pfade mit wenigsten Kanten

Der Algorithmus zur Breitensuche auf S. 44 bestimmt zu allen erreichbaren Knoten den Pfad von s mit den wenigsten Kanten.

- ▶ In die Warteschlange kommen zunächst alle Knoten mit Abstand 1 vom Startknoten.
- ▶ Wenn in der Warteschlange alle Knoten mit Abstand n von s sind, werden bei der Abarbeitung der Warteschlange alle Knoten mit Abstand $n + 1$ in die Schlange eingereiht.
- ▶ Dies ergibt per Induktion den Beweis. \square

Die Terminierung wird von der vorangehenden Laufzeitbetrachtung impliziert.

Gegenüberstellung von Tiefensuche und Breitensuche

Tiefensuche

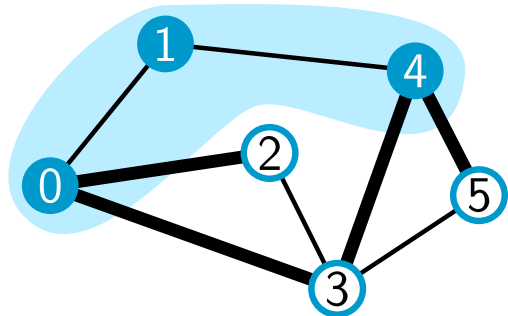
- ▶ Markiert erreichbare Knoten in Laufzeit $\mathcal{O}(V + E)$
- ▶ Zusammenhangskomponenten
- ▶ nicht geeignet für kürzeste Pfade

Breitensuche

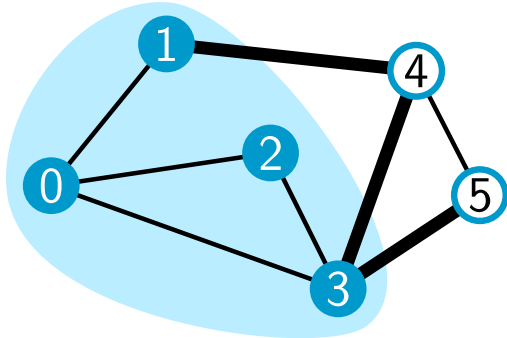
- ▶ Markiert erreichbare Knoten in Laufzeit $\mathcal{O}(V + E)$
- ▶ kürzeste Pfade
- ▶ Zusammenhangskomponenten allerdings nicht für *starke* Zusammenhangskomponenten in Digraphen geeignet

Gegenüberstellung von Tiefensuche und Breitensuche

Tiefensuche



Breitensuche

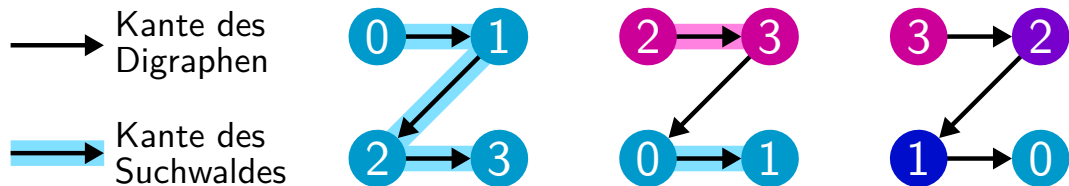


- ▶ In beiden Fällen werden vom Startknoten ausgehend erreichbare Knoten markiert.
- ▶ Es wird immer ein Knoten über eine Kante ausgewählt, die den markierten Bereich transzendiert (kreuzende Kante).
- ▶ Tiefensuche fährt bei frisch entdeckten Knoten fort, während Breitensuche die schon am längsten entdeckten Kanten auswählt.
- ▶ Weitere Auswahlvarianten führen zu wichtigen Algorithmen ... (tbc)

- ▶ Bisherige Algorithmen waren gleichermaßen für ungerichtete und gerichtete Graphen.
- ▶ Im Folgenden: Algorithmen, speziell für **gerichtete Graphen** (Digraphen)
- ▶ Dabei eignet sich insbesondere Tiefensuche dazu, **Strukturinformation** zu gewinnen:
- ▶ Wir teilen Kanten eines Digraphen in vier Kategorien ein.
- ▶ Dabei spielen zwei Reihenfolgen eine wichtige Rolle:
 - ▶ **Nebenreihenfolge** (*preorder*) in der Knoten entdeckt werden (Färbung hellblau)
 - ▶ **Hauptreihenfolge** (*postorder*) in der Knoten fertig abgearbeitet sind (Färbung dunkelblau)
- ▶ Mit dieser Strukturinformation können **gerichtete Zyklen** erkannt und **topologische Sortierungen** von Digraphen hergestellt werden.

Vorbemerkung: Besonderheit in gerichteten Graphen

- ▶ Um einen Digraphen mit Tiefensuche zu untersuchen, werden Tiefensuchdurchläufe mit unterschiedlichen Startknoten gestartet, bis alle Knoten erreicht wurden wie bei der Analyse von Zusammenhangskomponenten und Zyklen, Seite 36 und 38.
- ▶ Man erhält also einen Tiefensuchwald.
- ▶ Dies ist noch wie bei ungerichteten Graphen.
- ▶ Je nach Reihenfolge der Startknoten, kann das Ergebnis unerwartet sein:



- ▶ In **stark** zusammenhängenden Digraphen passiert diese Merkwürdigkeit nicht.

Kategorisierung von Kanten in Digraphen

Tiefensuche: Kanten $v \rightarrow w$ werden von bereits markierten Knoten v besucht.

- ▶ Ist w noch nicht markiert, so wird $v \rightarrow w$ eine **Baumkante**, also Teil des Suchwalds.
- ▶ Andernfalls, wenn w auch schon markiert ist:
 - ▶ **Vorwärtskante** (VK), wenn es einen gerichteten Pfad im Suchwald von v nach w gibt. Sie stellen Abkürzungen dar.
 - ▶ **Rückwärtskante** (RK), wenn es einen gerichteten Pfad im Suchwald von w nach v gibt. Sie schließen einen gerichteten Zyklus.
- ▶ Andernfalls: **Seitenkante** (SK), Verbindungen zwischen Pfaden eines Suchbaums oder zwischen Bäumen des Suchwaldes.

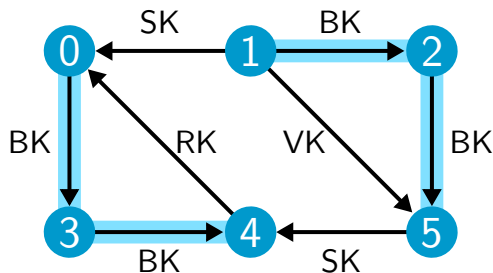
Bemerkungen zur Kategorisierung

- ▶ Die Unterscheidung zwischen VK und RK ist eindeutig: Jeder Knoten gehört zu genau einem Baum des Suchwaldes. Einen Pfad kann es nur geben, wenn v und w in demselben Suchbaum liegen. In einem Baum kann es nicht gerichtete Pfade in beiden Richtungen geben, also entweder VK oder RK.
- ▶ VK sind **Abkürzungen**: Eine VK ist nicht Teil des Suchwaldes. Sie ist ein Pfad von v nach w der Länge 1. Der im Suchbaum existierende Pfad muss mindestens die Länge 2 haben.
- ▶ RK schließen einen gerichteten **Zyklus**: Im Suchbaum existiert ein gerichteter Pfad von w nach v . Die gerichtete Kante $v \rightarrow w$ schließt also den Zyklus (ist aber nicht Teil des Suchwaldes.).
- ▶ SK verbinden zwei markierte Knoten. Also liegen v und w auf Pfaden des Suchwaldes. Da es aber weder einen Pfad von v nach w noch einen Pfad von w nach v im Suchwald gibt, sind diese Pfade getrennt, ggf. sogar in unterschiedlichen Bäumen.

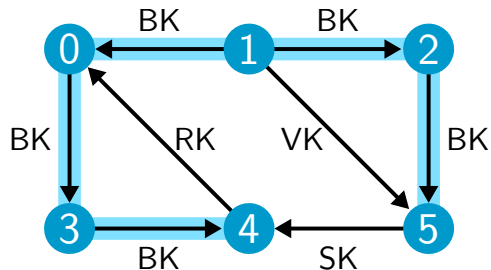
Quiz: Zu welchen Kategorien gehören die Kanten dieses Graphen?

→ Kante des Digraphen

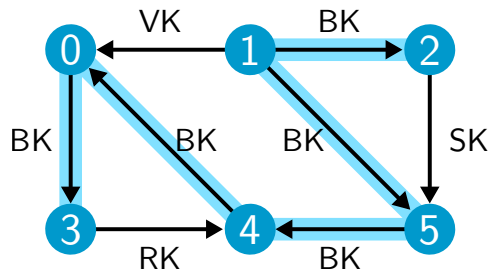
→ Kante des Suchwaldes



Start von Knoten 1



Start von 1 in Richtung 5



Effiziente Zuordnung der Kategorien

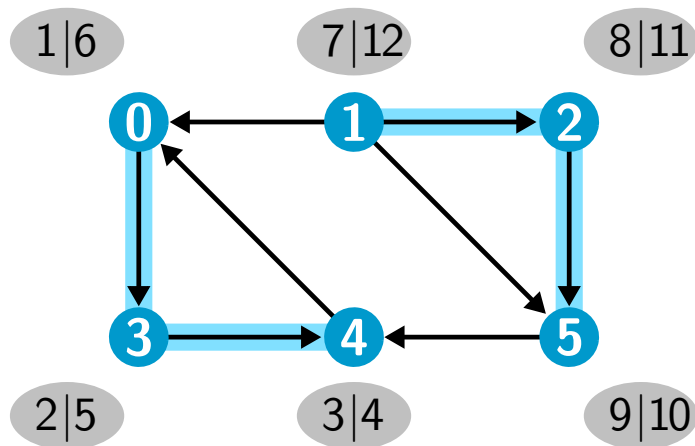
- ▶ Überprüfung, ob $v \rightsquigarrow w$ im Suchbaum ist, durch Iteration entlang der Baumkanten: nicht effizient
- ▶ Effizienter Ansatz mit zwei Zeitstempeln pro Knoten:
 - **Eingangsstempel** wenn der Knoten entdeckt wird (Färbung hellblau)
 - **Ausgangsstempel** wenn der Knoten fertig abgearbeitet ist (Färbung dunkelblau)
- ▶ Zeit kann getrennt oder gemeinsam für Ein- und Ausgangsstempel hochgezählt werden.
- ▶ **Nebenreihenfolge** (*preorder*): Sortierung aufsteigend nach Eingangsstempel
- ▶ **Hauptreihenfolge** (*postorder*): Sortierung aufsteigend nach Ausgangsstempel
- ▶ **Umgekehrte Hauptreihenfolge** (*reverse postorder*): absteigend nach Ausgangsstempel

Zeitstempel der Tiefensuche

```
1 public class NodeOrder {
2     public int[] dt; // discovery time
3     public int[] ft; // finalizing time
4     private int time;
5
6     public NodeOrder(Digraph G) {
7         dt = new int[G.V()];
8         ft = new int[G.V()];
9         for (int v = 0; v < G.V(); v++)
10             if (dt[v] == 0)
11                 dfs(G, v);
12     }
13
14     public void dfs(Digraph G, int v) {
15         dt[v] = ++time;
16         for (int w : G.adj(v))
17             if (dt[w] == 0)
18                 dfs(G, w);
19         ft[v] = ++time;
20     }
21 }
```

- ▶ Der Eingangszeitstempel wird als Markierung genutzt via `marked[v]` $\Leftrightarrow (dt[v] > 0)$.
- ▶ Die Methode `dfs()` wird in Zeile 11 für jeden Knoten `v` aufgerufen, damit auch bei Graphen, die **nicht stark zusammenhängend** sind alle Knoten erreicht werden.

Beispieldurchlauf DFS mit Zeitstempeln



Nebenordnung: 0, 3, 4, 1, 2, 5

Hauptordnung: 4, 3, 0, 5, 2, 1

Effiziente Zuordnung der Kategorien

- ▶ Wenn v und w innerhalb desselben Pfades im Suchwald liegen, rahmen sich die Zeiten ein. Wurde v zuerst entdeckt (also $dt[v] < dt[w]$), dann wurde w zuerst fertig bearbeitet ($ft[w] < ft[v]$).
- ▶ Damit haben wir eine effizient zu prüfende Bedingung zur Unterscheidung der Knotentypen VK, RK und SK:

Erreichbarkeit im Suchwald der Tiefensuche

Es gibt einen gerichteten Pfad von v nach w im Suchwald, genau dann, wenn $dt(v) \leq dt(w)$ und $ft(w) \leq ft(v)$ gilt.

- ▶ Mit Hilfe der Zeitstempel kann mit einer Zusatzüberlegung eine Methode `isDAG()` implementiert werden, die in einer Laufzeit in $O(V + E)$ entscheiden kann, ob ein gegebener Digraph ein **zyklenfreier Digraph** (*directed acyclic graph*, DAG) ist.

Topologische Sortierung

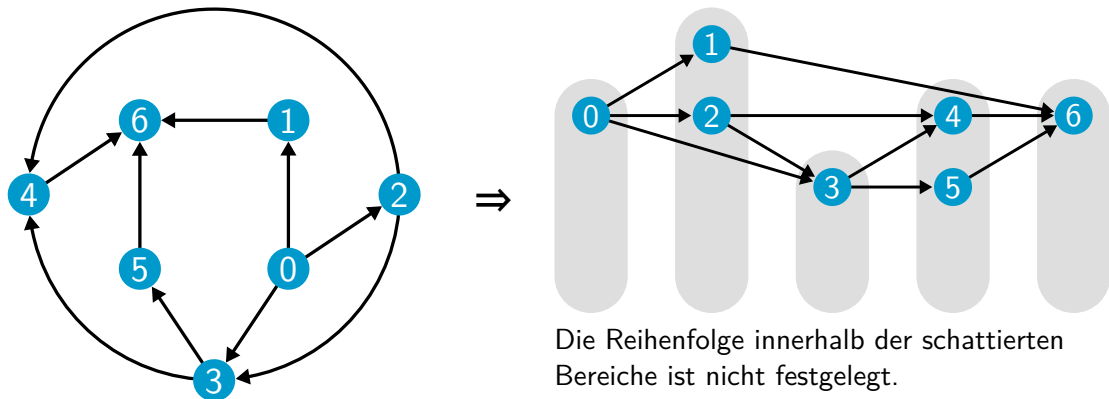
- Unter einer **topologischen Sortierung** eines Digraphen versteht man eine Reihenfolge der Knoten, die mit seinen Pfeilen verträglich ist:
 - ▶ In dieser Sortierung zeigen alle Pfeile auf zukünftige Knoten.
 - ▶ Graphen mit topologischer Sortierung haben mindestens eine **Quelle** (Knoten am Anfang der Ordnung) und mindestens eine **Senke** (Knoten am Ende der Ordnung).
 - ▶ Wenn die Pfeile **Vorrangbedingungen** (*precedence constraints*) darstellen (v muss vor w geschehen), dann liefert die topologische Sortierung eine Ablaufreihenfolge.
 - ▶ Die Aufgabe des topologischen Sortierens kommen in vielen Anwendungsbereichen vor, z.B. Vererbungshierarchien, Ablaufplanung, Tabellenkalkulation.

Topologische Sortierung durch Rangordnung

- Eine injektive (rechtseindeutige) Funktion

$$r : V \rightarrow \mathbb{R}$$

definiert eine topologische Ordnung eines Digraph $G = (V, E)$, wenn für alle Kanten $v \rightarrow w \in E$ gilt: $r(v) < r(w)$. Man kann auch $\{0, \dots, V-1\}$ als Zielbereich nehmen.



Existenz einer Topologischen Ordnung

Ein Digraph kann genau dann topologisch geordnet werden, wenn er zyklensfrei ist.

- ▶ Offensichtlich kann es bei Existenz eines Zyklus keine topologische Ordnung geben.
- ▶ Dass jeder DAG eine topologische Sortierung besitzt, zeigen wir an Hand eines Algorithmus, der diese Aufgabe löst.
- ▶ Die Reihenfolge des Entdeckens bei der Tiefensuche liefert eine gute Ausgangsbasis. Für Baumkanten und Vorwärtskanten ist die Vorrangbedingung erfüllt.
- ▶ Da es keine Rückwärtskanten gibt (Zyklenfreiheit), bleiben die Seitenkanten als einziger kritischer Punkt, siehe Kante $1 \rightarrow 0$ auf S. 55.
- ▶ Die **umgekehrte Reihenfolge** des fertig Bearbeitens (**umgekehrte Hauptordnung**, *reverse postorder*) erfüllt in allen Fällen die topologische Sortierung.

Beweis der Existenz einer Topologische Sortierung

Definition einer Topologischen Ordnung durch umgekehrte Hauptordnung

Die umgekehrte Hauptordnung, die durch die Rangordnung $r(v) = -ft(v)$ definiert ist, ergibt eine topologische Ordnung für jeden zyklenfreien Digraphen.

- ▶ Zum Beweis sei $v \rightarrow w$ eine Kante des Digraphen G . Wir betrachten die Situation, in der $\text{dfs}(G, v)$ aufgerufen wird und zeigen $ft(w) < ft(v)$. Es gibt folgende Möglichkeiten:
 - ▶ w ist noch nicht markiert.
 - ▶ Dann liegen v und w in demselben Pfad des Suchwaldes und v wird zuerst entdeckt. Folglich gilt $ft(w) < ft(v)$, siehe S. 56.
 - ▶ w ist markiert.
 - ▶ Dann muss w schon vollständig abgearbeitet sein, also gilt $ft(w) < dt(v) < ft(v)$. Wäre w noch nicht abgearbeitet, dann gäbe es einen Pfad von w nach v , was zusammen mit der Kante $v \rightarrow w$ ein Zyklus wäre.
- ▶ Für jeden Pfeil $v \rightarrow w$ gilt also $ft(w) < ft(v)$. \square

Implementierung der Topologischen Sortierung

- ▶ Wir benutzen einen Stack für die **umgekehrte** Hauptordnung.
- ▶ Achtung: Bei dem Stack aus `java.util` funktioniert das nur, wenn man die Elemente per `pop()` aus dem Stack holt, nicht bei der Benutzung als `Iterable`.

```
// Only for DAG! Check for cycle before
public class Topological {
    private boolean[] marked;
    private Stack<Integer> reversePost;

    public Topological(Digraph G) {
        reversePost = new Stack<>();
        marked = new boolean[G.V()];
        for (int v = 0; v < G.V(); v++) {
            if (!marked[v]) {
                dfs(G, v);
            }
        }
    }
}
```

```
public void dfs(Digraph G, int v) {
    marked[v] = true;
    for (int w : G.adj(v)) {
        if (!marked[w]) {
            dfs(G, w);
        }
    }
    reversePost.push(v);
}

public Stack<Integer> order() {
    return reversePost;
}
}
```

- ▶ Topologische Sortierung hat wie Tiefensuche eine Laufzeit in $O(V + E)$.

Herausforderungen für Graphenalgorithmen

- ▶ **Bipartit:** Können die Knoten mit zwei Farben eingefärbt werden, so dass keine Kante Knoten derselben Farbe verbindet?
Einfach mit Kenntnis dieser Vorlesung
- ▶ **Euler Zyklus:** Finde einen Zyklus, der jede Kante genau einmal benutzt. Gibt es immer, wenn alle Knoten eine grade Ordnung haben.
Relativ einfach mit Kenntnis dieser Vorlesung
- ▶ **Hamilton Zyklus:** Finde einen Zyklus, der jeden Knoten genau einmal benutzt.
NP-schweres Problem
- ▶ **Isomorphismus:** Prüfe, ob zwei Graphen bis auf Nummerierung der Knoten übereinstimmen.
Offenes Problem, ob dies in P lösbar ist
- ▶ **Ebenes Layout:** Finde eine Anordnung der Knoten, so dass sich keine Kanten kreuzen (falls möglich).
Komplexer Algorithmus mit DFS und linearer Laufzeit [Hopcroft & Tarjan 1974]

- ▶ Dünne Graphen werden am besten mit Adjazenzlisten implementiert.
- ▶ Graphen können mit Breiten- und Tiefensuche in $\mathcal{O}(V + E)$ durchsucht werden.
- ▶ Ausgehend von s wird ein Suchbaum erstellt, der alle von s erreichbaren Knoten enthält (Spannbaum).
- ▶ Breitensuche erstellt Wege, die über möglichst wenige Kanten zum Ziel führen.
- ▶ Tiefensuche liefert Strukturinformation, u.a. eine topologische Sortierung für DAG.

- ▶ Ottmann T & Widmayer P. *Algorithmen und Datenstrukturen*. Springer Verlag, 5. Auflage; 2011. ISBN: 978-3827428042
- ▶ Sedgwick R & Wayne K, *Algorithmen: Algorithmen und Datenstrukturen*, Pearson Studium, 4. Auflage, 2014. ISBN: 978-3868941845; in Teilen auch auf <http://www.cs.princeton.edu/IntroAlgsDS>
- ▶ TH Cormen, CE Leiserson, R Rivest, C Stein, *Algorithmen - Eine Einführung* . De Gruyter Oldenbourg, 4. Auflage; 2013. ISBN: 978-3486748611
- ▶ Hopcroft J & Tarjan R. *Efficient planarity testing*. Journal of the ACM (JACM). 1974 Oct 1;21(4):549-68.
- ▶ <https://idea-instructions.com/>

Index

adjacent, 4
adjazent, 4
Adjazenzliste, 17
Adjazenzmatrix, 15
API
 Digraph, 12
 Graph, 12
arcs, 5
Ausgangsgrad, 7
azyklisch, 6
Baum, 9
Baumkante, 50
Breitensuche, 39
 Laufzeit, 41
connected, 8
connected components, 8
cycle, 6
DCAL, 19
Digraph, 5
Digraph, 12

digraph, 5
directed acyclic graph, 56
directed path, 6
Ecken, 4
Eingangsgrad, 7
gerichteter Pfad, 6
gerichteter Zyklus, 6
Grad eines Knotens, 7
Graph, 4
 gerichteter, 5
Graph, 12
Hauptreihenfolge, 53
incident, 4
inzident, 4
Kante, 4
 gerichtete, 5
Knoten, 4
Laufzeit
 BFS, 41

Breitensuche, 41
DFS, 32
Tiefensuche, 32
Länge eines Pfades, 6
Mehrfachkanten, 7
Nebenreihenfolge, 53
nodes, 4
Ordnung, 7
parallel, 7
path, 6
Pfad, 6
 einfacher, 6
 gerichteter, 6
Pfeile, 5
Pfeilliste
 doppelt verkettet, 19
postorder, 53
preorder, 53
Quelle, 7

reverse postorder, 53, 59

Rückwärtskante, 50

Seitenkante, 50

Senke, 7

Spannbaum, 9

stark verbunden, 8

stark zusammenhängend, 8

strong connected, 8

Suchbaum, 25

Tiefensuche

Laufzeit, 32

rekursive Implementation, 30

Topologische Sortierung, 57

umgekehrte Hauptordnung, 59

Umgekehrte Hauptreihenfolge,
53

vertices, 4

Vorwärtskante, 50

Wald, 9

Weg, 6

Zusammenhangskomponente, 8

starke, 8

Zusammenhangskomponenten

identifizieren, 36

zusammenhängend, 8

zyklenfreier Digraph, 56

Zyklus, 6

Zyklus Erkennung, 38