

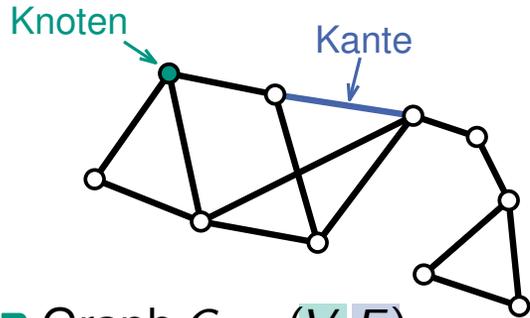
Algorithmen 1

Graphen und Breitensuche



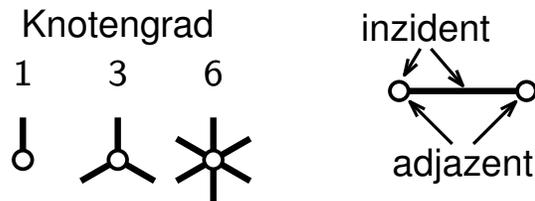
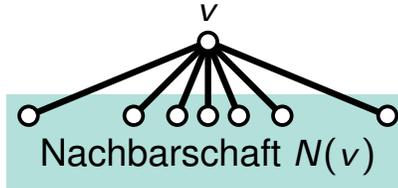
Graphen: Grundbegriffe

Knoten & Kanten

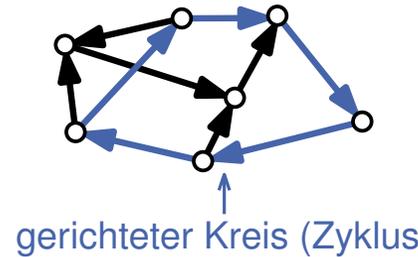
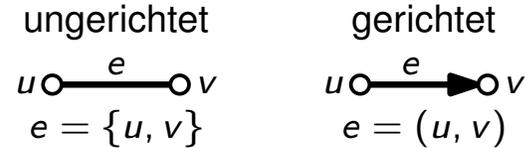


- Graph $G = (V, E)$
- $|V| = n, |E| = m$

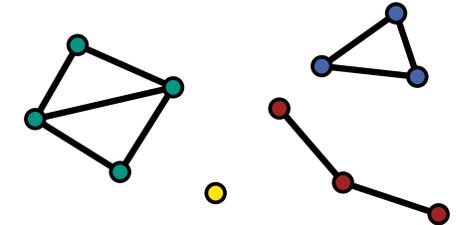
Nachbarschaft



Gerichtete Graphen



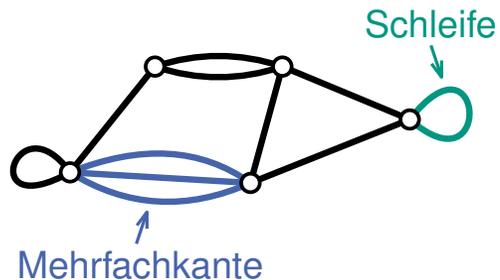
Komponenten



- unzusammenhängend
- 4 Zusammenhangskomp.

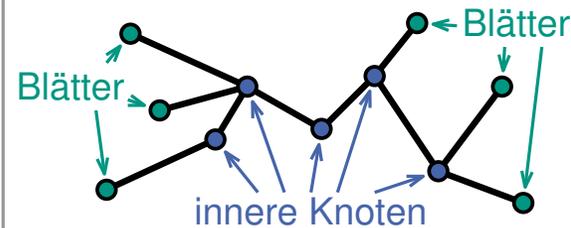
Einfache Graphen

- keine Schleifen
- keine Mehrfachkanten



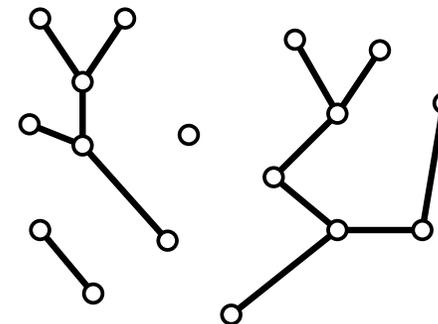
Baum

- kreisfrei
- zusammenhängend

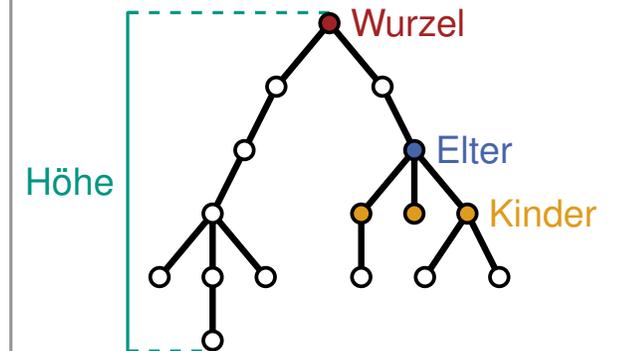


Wald

- kreisfrei



Gewurzelter Baum



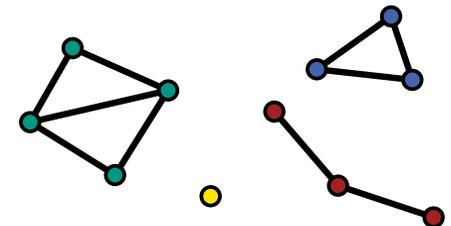
Ein erstes Graphenproblem

Graphen sind allgegenwärtig

- modelliert Beziehungen (Kanten) zwischen Paaren von Objekten (Knoten)
 - Kontakte zwischen Lebewesen
 - Interaktionen zwischen Proteinen
 - Kommunikation zwischen Sensoren
 - ...

Grundlegendes Problem: Zusammenhangskomponenten finden

- gegeben: Graph (z.B. als Liste von Kanten)
- Ziel: färbe die Knoten entsprechend der Komponenten



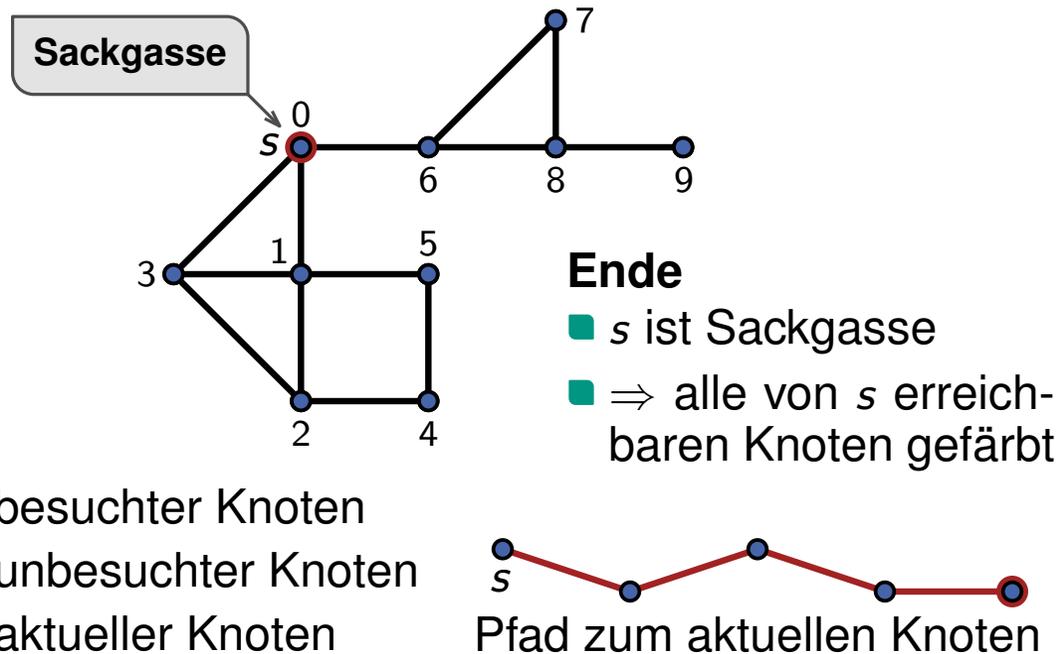
Vorgehen: Graphtraversierung

- laufe bei einem Knoten s los
- finde und färbe alle Knoten, die von s aus erreichbar sind
- wähle neue Farbe und wiederhole mit noch ungefärbtem Knoten (falls vorhanden)

Graphtraversierung

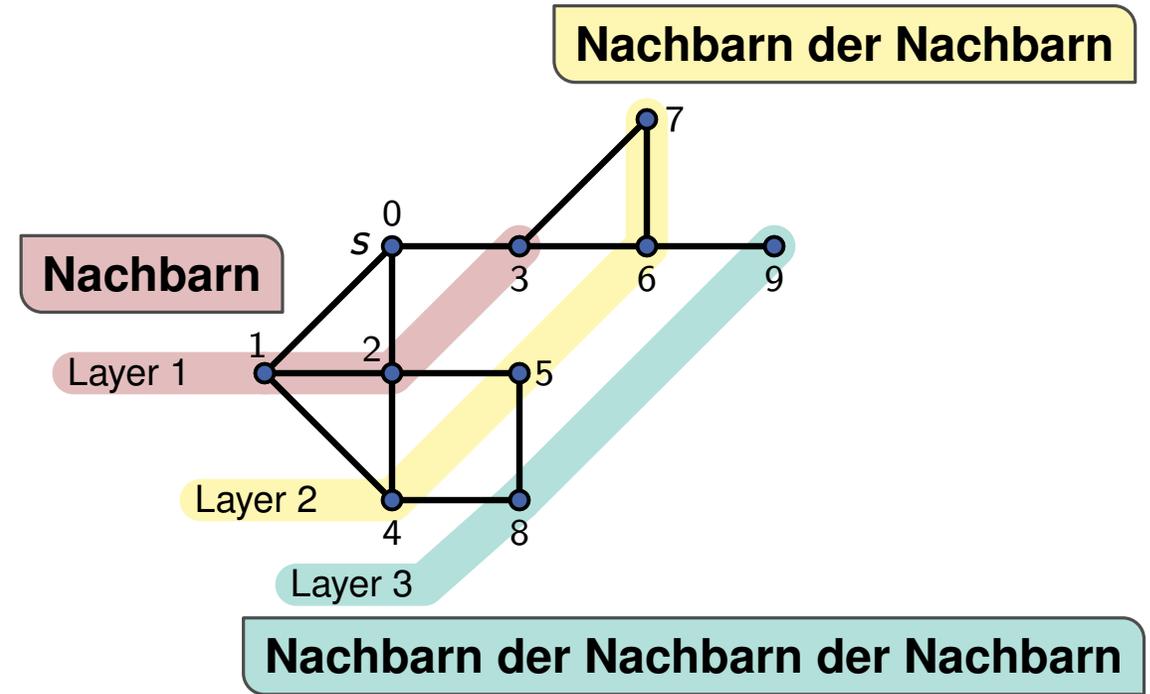
Tiefensuche (DFS)

- starte bei einem Knoten s
- laufe in jedem Schritt zu neuem Nachbarn
- Sackgasse (kein neuer Nachbar)
→ Backtracking: zurück zum Vorgänger



Breitensuche (BFS)

- starte bei einem Knoten s
- besuche alle Nachbarn von s
- dann alle Nachbarn der Nachbarn usw.



Breitensuche

Abstrakte Algorithmenidee

- im Prinzip wissen wir, was die BFS tut
- sind aber noch weit von einer Implementierung entfernt

Effiziente Umsetzung

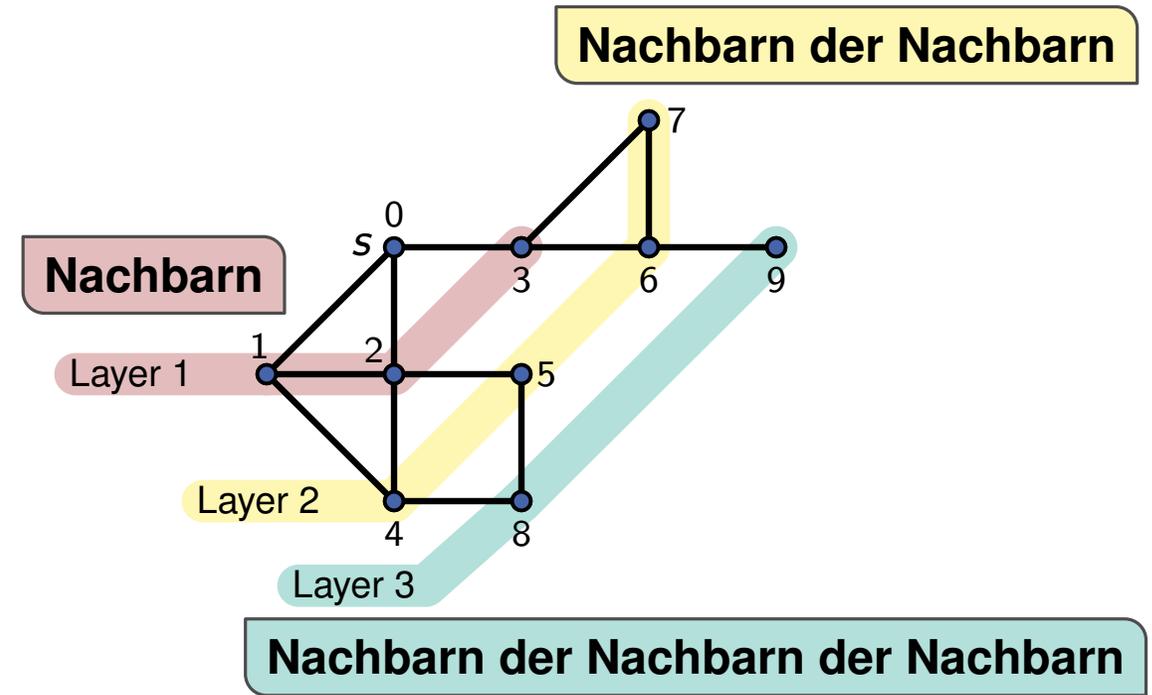
- nicht leicht
- mit etwas Übung aber auch nicht schwer
- für BFS: gleich

Wiederholung: Ziele der Vorlesung

- eigene Algorithmenideen entwickeln
- effiziente Umsetzung der Ideen
- gute Intuition, welche Ideen sich effizient umsetzen lassen

Breitensuche (BFS)

- starte bei einem Knoten s
- besuche alle Nachbarn von s
- dann alle Nachbarn der Nachbarn usw.

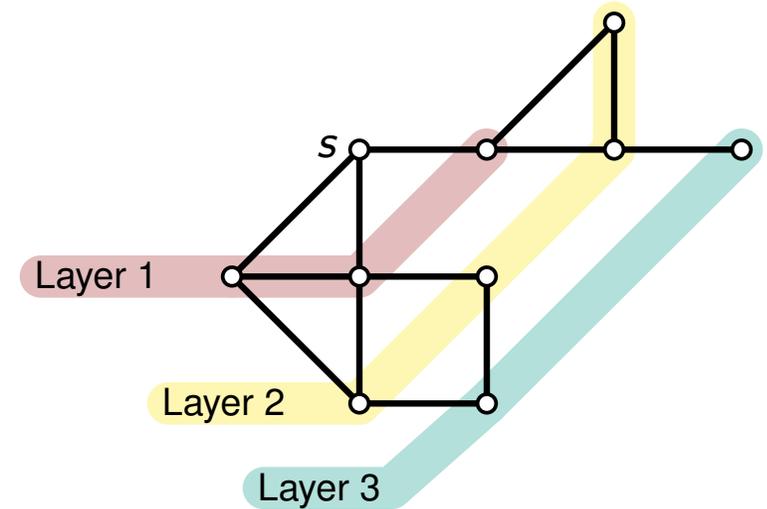


Algorithmus → Pseudocode (erster Versuch)

BFS(*Graph G, Node s*)

```

ℓ := 0 // current layer
color node s
while there is a node in layer ℓ + 1 do
  ℓ := ℓ + 1
  for Node v in layer ℓ do
    color node v
  
```



Zu hohe Abstraktion für Pseudocode

- **while** there is a node in layer $\ell + 1$ **do**
 - Woher wissen wir das?
- **for** Node v in layer ℓ **do**
 - Welche Knoten liegen in Layer ℓ ?

Ziele für die Laufzeit

- Layer $\ell + 1$ aus Layer ℓ : betrachte nur Kanten inzident zu Knoten aus Layer ℓ
→ insgesamt $\Theta(m)$
- über Knoten in Layer ℓ iterieren: linear in Anzahl Knoten in Layer ℓ
→ insgesamt $\Theta(n)$

Algorithmus → Pseudocode (zweiter Versuch)

BFS(*Graph G, Node s*)

currLayer := {s}

color node s

nextLayer := $N(s)$ and color $N(s)$

while nextLayer $\neq \emptyset$ **do**

currLayer := nextLayer

// determine next layer

nextLayer := \emptyset

for Node u in currLayer **do**

for Node v in $N(u)$ **do**

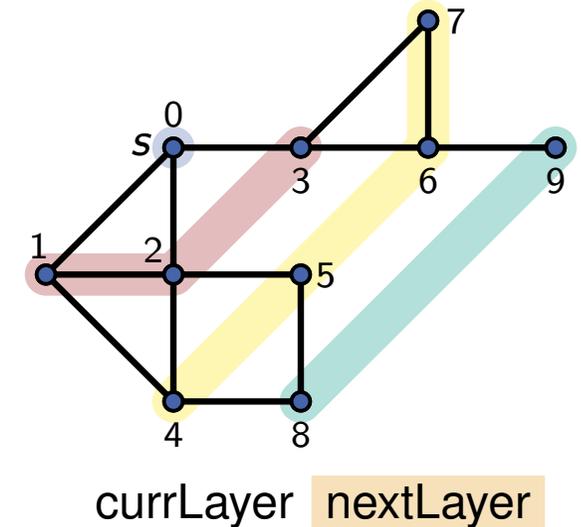
if Node v is uncolored **then**

 insert v into nextLayer

 color node v

Beobachtung

- neue Knoten v werden hinten eingefügt
- Knoten u läuft von vorne nach hinten
- Verhalten einer Queue (First In, First Out)



Färbung beim Einfügen in nextLayer

- stellt sicher, dass jeder Knoten v nur einmal eingefügt wird
- auch dann, wenn v mehrere Kanten vom vorherigen Layer hat

Algorithmus → Pseudocode (Endergebnis)

BFS(*Graph G, Node s*)

Queue $Q :=$ empty queue

$Q.\text{push}(s)$

color node s

while $Q \neq \emptyset$ **do**

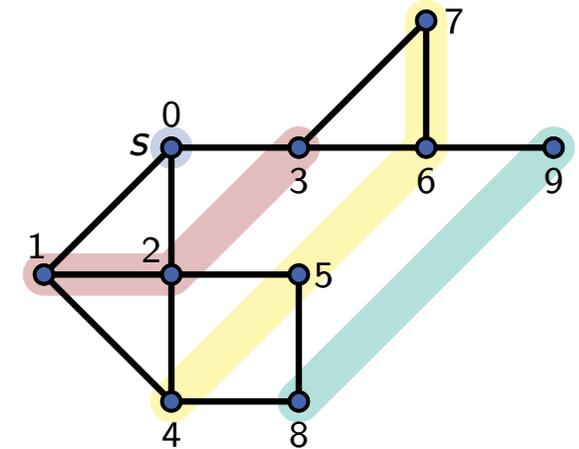
$u := Q.\text{pop}()$

for Node v in $N(u)$ **do**

if v is uncolored **then**

$Q.\text{push}(v)$

 color node v



Q: [0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9]

Erinnerung: Queue

- Operationen: **pushBack**, **popFront**
- FIFO: First In – First Out



Breitensuche: Laufzeit

BFS(*Graph G, Node s*)

Queue $Q :=$ empty queue

$Q.$ **push**(s)

color node s

while $Q \neq \emptyset$ **do**

$u := Q.$ **pop**()

for *Node v* in $N(u)$ **do**

if v is uncolored **then**

$Q.$ **push**(v)

color node v

Intuition von vorhin: pro Layer ...

- betrachte Knoten in dem Layer \rightarrow insgesamt $\Theta(n)$
- betrachte nur Kanten inzident zu Knoten aus dem Layer \rightarrow insgesamt $\Theta(m)$

Etwas formaler für unsere finale Umsetzung

- jeder Knoten wird **nur einmal** in die Queue eingefügt
- **while-Schleife**: ein Durchlauf für jeden Knoten $u \in V$
- **for-Schleife**: $\deg(u) = |N(u)|$ Durchläufe
- Gesamtkosten: $\sum_{u \in V} \deg(u) = 2m$

(jede Kante $\{u, v\}$ wird doppelt gezählt:
einmal für u in $\deg(u)$ und einmal für v in $\deg(v)$)

Notation

- V : Knotenmenge
- E : Kantenmenge
- $n = |V|, m = |E|$
- $\deg(u)$: Grad von u

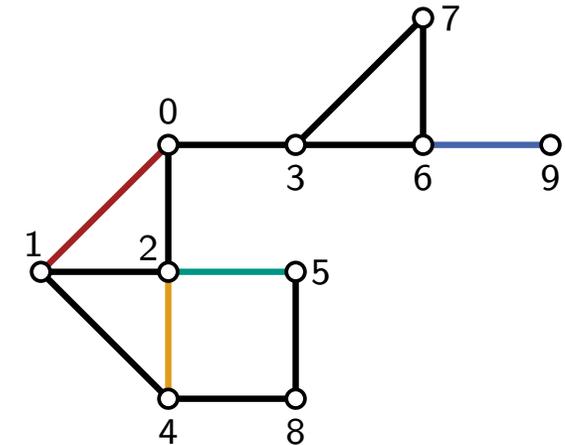
\Rightarrow BFS hat lineare Laufzeit $\Theta(m)$

(vorausgesetzt, unsere Graphdatenstruktur erlaubt es uns in $O(\deg(u))$ über $N(u)$ zu iterieren)

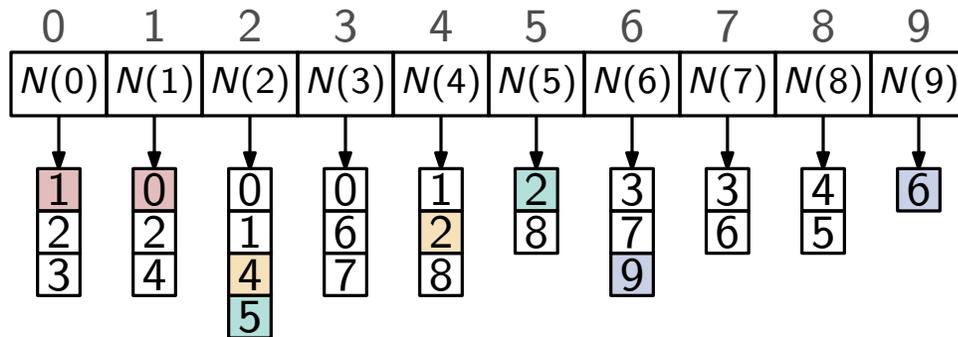
Einschub: Graphrepräsentation

Grundsätzliches

- repräsentiere jeden Knoten durch eine Zahl in $\{0, \dots, n - 1\}$
- Zusatzinfos für Knoten (z.B. Farbe): Array der Länge n
 - Array erstellen: einmalig $\Theta(n)$
 - Eigenschaft eines Knotens ändern/abfragen: $\Theta(1)$



Adjazenzliste



- über $N(v)$ iterieren: $\Theta(\text{deg}(v))$
- Detailumsetzung hängt von Anwendung ab
(mit Engineering-Potential, z.B. für bessere Cache-Effizienz)

Adjazenzmatrix

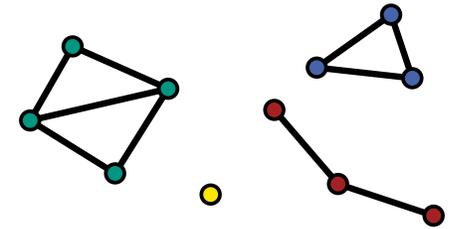
	0	1	2	3	4	5	6	7	8	9
0	0	1	1	1	0	0	0	0	0	0
1	1	0	1	0	1	0	0	0	0	0
2	1	1	0	0	1	1	0	0	0	0
3	1	0	0	0	0	0	1	1	0	0
4	0	1	1	0	0	0	0	0	1	0
5	0	0	1	0	0	0	0	0	1	0
6	0	0	0	1	0	0	0	1	0	1
7	0	0	0	1	0	0	1	0	0	0
8	0	0	0	0	1	1	0	0	0	0
9	0	0	0	0	0	0	1	0	0	0

- testen ob $\{u, v\} \in E$ geht in $\Theta(1)$
- $\Theta(n^2)$ Speicher
- $N(v)$ iterieren: $\Theta(n)$

Zusammenhangskomponenten

Grundlegendes Problem: Zusammenhangskomponenten finden

- gegeben: Graph
- Ziel: färbe die Knoten entsprechend der Komponenten



Gerade gesehen

- **BFS**(G, s) färbt von s aus erreichbaren Knoten (Komponente $C(s)$)
- Laufzeit: $\Theta(m_{C(s)})$

Färbung aller Komponenten

- iteriere über alle Knoten $s \in V \rightarrow$ **BFS**(G, s) wenn s noch ungefärbt
- wähle für jeden Aufruf der BFS eine neue Farbe
- für jede Komponente wird die BFS einmal ausgeführt \rightarrow eine Farbe pro Komponenten
- Laufzeit: $\Theta(n + m)$
 - über alle Knoten iterieren: $\Theta(n)$
 - BFS in Summe (jede Kante gehört zu nur einer Komponente): $\Theta(m)$

Notation

- $G = (V, E)$: Graph
- $n = |V|, m = |E|$
- $C(s)$: Komponente von s
- $m_{C(s)}$: #Kanten in $C(s)$

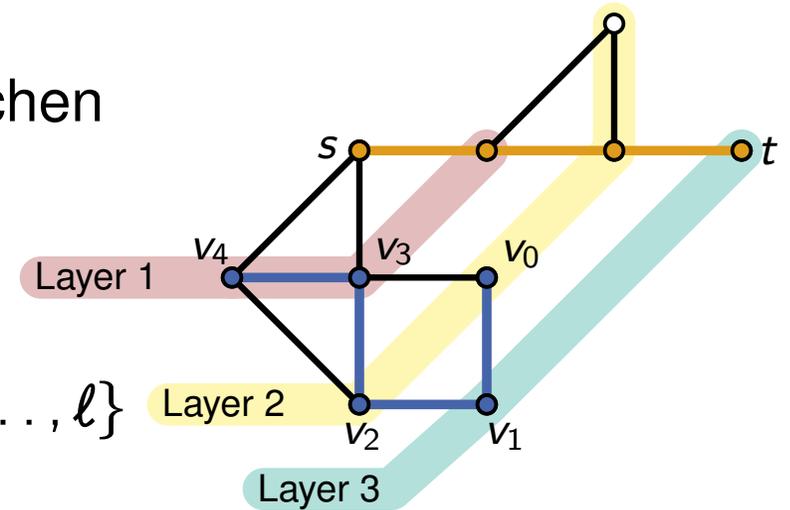
BFS: Layer, Distanzen, Kürzeste Pfade

Beobachtung

- Knoten in Layer ℓ kann man mit ℓ Schritten von s aus erreichen
- BFS berechnet also die Distanz von s zu anderen Knoten

Etwas formaler: ein paar Grundbegriffe

- **Pfad**: Knotenfolge $\langle v_0, \dots, v_\ell \rangle$, mit $\{v_{i-1}, v_i\} \in E$ für $i \in \{1, \dots, \ell\}$
- **Länge** des Pfades $\langle v_0, \dots, v_\ell \rangle$: ℓ
- v_0 und v_ℓ sind **Start-** bzw. **Endknoten** von $\langle v_0, \dots, v_\ell \rangle$
- **kürzester Pfad** von $s \in V$ nach $t \in V$: Pfad minimaler Länge mit Start s und Ende t
- **Distanz** $\text{dist}(s, t)$: Länge des kürzesten st -Pfades



Theorem

Liegt ein Knoten v bei der Breitensuche von s aus in Layer ℓ dann gilt $\text{dist}(s, v) = \ell$.

BFS kann Distanzen berechnen

Theorem

Liegt ein Knoten v bei der Breitensuche von s aus in Layer ℓ dann gilt $\text{dist}(s, v) = \ell$.

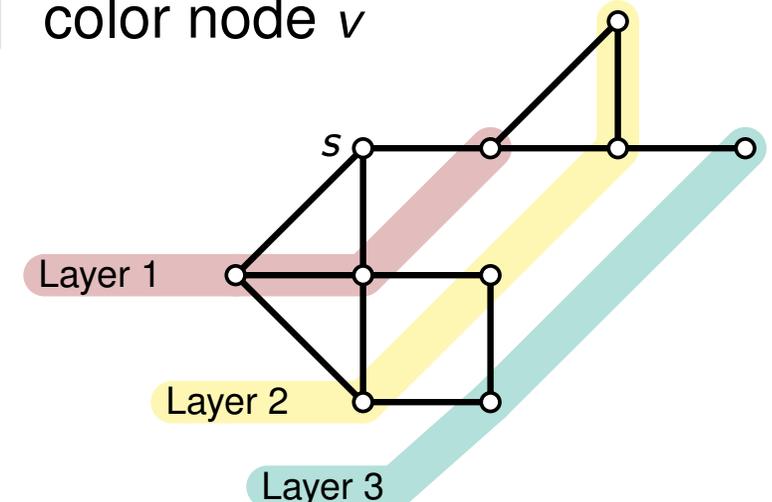
Beweis

- es gibt einen Pfad der Länge ℓ von s nach v
 - Induktion über ℓ : klar für $\ell = 0$
 - v in Layer $\ell \Rightarrow v$ hat Nachbarn u in Layer $\ell - 1$
 - Ind.-Voraussetzung $\Rightarrow su$ -Pfad der Länge $\ell - 1$
 - daher: sv -Pfad der Länge ℓ

- es gibt keinen kürzeren Pfad von s nach v
 - Kanten überspringen keine Layer
 - jeder Pfad von v (Layer ℓ) nach s (Layer 0) hat Länge mindestens ℓ

```

...
for Node v in N(u) do
  if v is uncolored then
    Q.push(v)
    color node v
  
```



Distanzen und kürzeste Wege: tatsächliche Berechnung

Folgerungen aus dem Beweis

- gerade bewiesen: die Layer geben die Distanzen von s
- im Beweis sieht man, wie man den kürzesten Weg erhält:
 - Knoten v wird durch Knoten u gefunden
 - definiere u als Elternknoten von v
 - Zurückverfolgung der Eltern bis s liefert kürzesten Pfad

Beobachtung

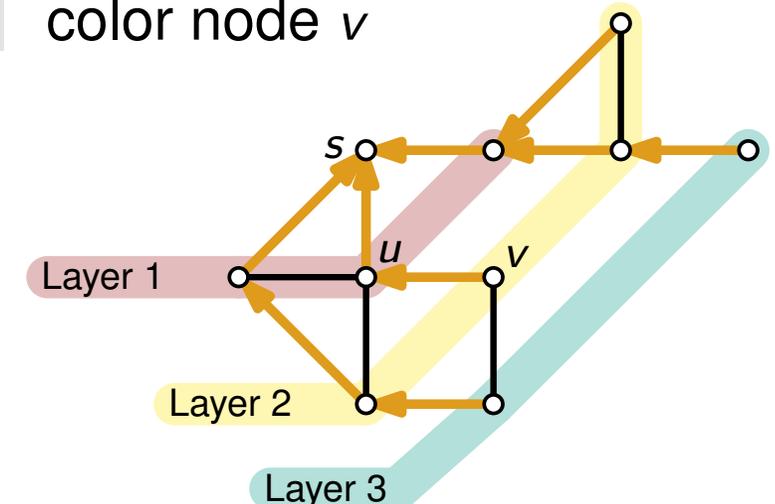
- Eltern-Beziehung liefert **Baum** mit Wurzel s
- diesen Baum nennt man auch **BFS-Baum**

Anmerkung

- auch hier: wie das geht ist im Prinzip klar
- für die tatsächliche Umsetzung muss man aber nochmal tätig werden

```

...
for Node v in N(u) do
  if v is uncolored then
    Q.push(v)
    color node v
  
```



Breitensuche (BFS)

Graph Traversierung

- zwei Vorgehensweisen: BFS und DFS (DFS kommt später nochmal im Detail)
- damit kann man zum Beispiel die Zusammenhangskomponenten finden

Algorithmenidee → Pseudocode am Beispiel BFS

- auch bei einem so einfachen Algorithmus kann man viel falsch machen
- meine Hoffnung:
 - wenn du die Idee der Breitensuche kennst
 - du aber **nicht** den Pseudocode dazu auswendig kannst
 - dann kannst du eine Breitensuche im Schlaf implementieren

Kürzeste Wege

- BFS besucht Knoten in aufsteigender Distanz vom Start
- daher: Berechnung von Distanzen und kürzesten Wegen