Algorithmen I
Tutorium 32

Eine Lehrveranstaltung im SS 2017 (mit Folien von Christopher Hommel)

Daniel Jungkind (ufesa@kit.edu) | 30. Juni 2017

INSTITUT FÜR THEORETISCHE INFORMATIK

GRAPHEN

Der Plural, nicht der Kohlenstoff

Graphen

Wir erinnern uns ...

- Graph $G = (V, E)$ mit **Knoten**menge $V \neq \emptyset$ und **Kanten**menge E
- *Gerichteter* Graph: $E \subseteq V \times V$
- *Ungerichteter* Graph: $E \subseteq \Big\{\{u, v\} \;\Big|\; u, v \in V\Big\}$
- Umwandlung *ungerichtet* $\rightsquigarrow$ *gerichtet* trivial
  $\Rightarrow$ Im Folgenden stets *gerichtete* Graphen
- $n := |V|$
- $m := |E|$
- Betrachten üblicherweise $V = \{1...n\}$

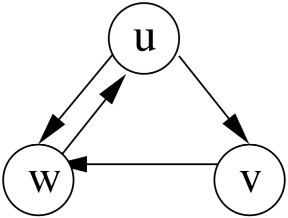
Repräsentationen von Graphen

Kantenfolge

- (Zusammenhängender) Graph eindeutig definiert durch **Menge aller Kanten** (Reihenfolge egal)
- Knoten v existiert in G
  $\Leftrightarrow \exists\ (v, x)$ oder $(x, v) \in$ *Kantenliste* (x beliebig)

+ **Kompakt** $\Rightarrow$ Gut handhabbar (im Speicher oder bei I/O)

− Einzige effiziente Operation: **Durchlaufen** aller Kanten

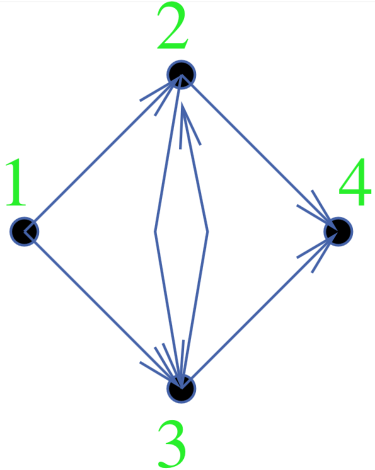


$$\Leftrightarrow \langle (u, v), (v, w), (w, u), (u, w) \rangle$$

Repräsentationen von Graphen

Adjazenzmatrix

Verwende Matrix $A \in \{0,1\}^{n \times n}$ mit $a_{ij} = 1 \;\Leftrightarrow\; (i,j) \in E$



$$\begin{array}{c} \text{Von} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \text{Nach} \;\; 1 \quad 2 \quad 3 \quad 4 \\ \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

Repräsentationen von Graphen

Adjazenzmatrix

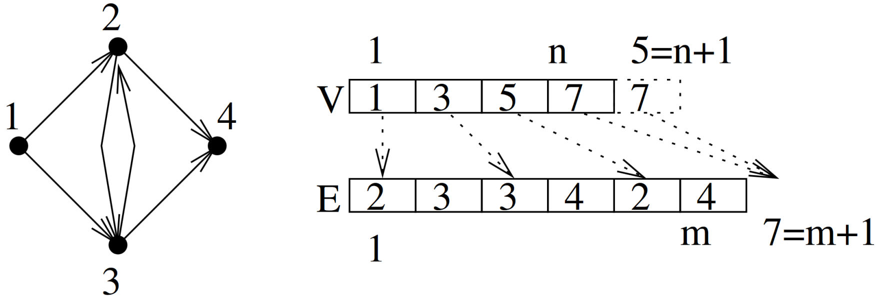
- \+ recht **platzeffizient**, falls Graph dicht
- \+ **Einfügen**, **Löschen** und **Testen** von Kanten in $O(1)$ und simpel
- \+ $\{0, 1\} \rightsquigarrow \mathbb{R}$ erweitern für **Kantengewichte**
- \+ LA, yay! :D

- − platz**in**effizient bei dünnbesetzten Graphen
  (also durchschn. Knotengrad $\ll n$)
- − **langsame** Navigation
- − LA **kotz**

Repräsentationen von Graphen

Adjazenzfeld (aka Adjazenzarray)

- Definiere V : **array**$[1 \dots n+1]$ **of** $\{1 \dots m+1\}$ und
  E : **array**$[1 \dots m]$ **of** $\{1 \dots n\}$
- Von v erreichbare Knoten: $\left\{\, E[i] \mid V[v] \leqslant i < V[v+1] \,\right\}$
- *„Dummy-Eintrag“*:
  $V[n+1] := m+1$, damit oben $v = n$ nicht knallt



Repräsentationen von Graphen

Adjazenzfeld (aka Adjazenzarray)

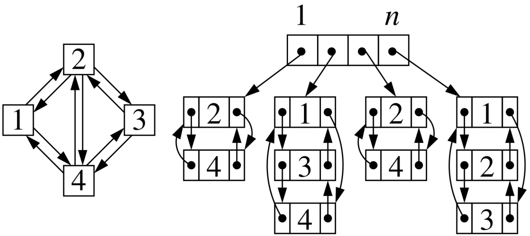
- ➕ **Navigation** gut möglich
- ➕ **Zusatzinfos** (z. B. Kantengewichte) durch weitere Arrays leicht aufrüstbar
- ➕ **Cachefreundlich**
- ➕ Nachrüstbar: Kanten **löschen**, **rückwärts** laufen
- ➖ Hinzufügen von Kanten scheiße (#ArraysHalt...)

Repräsentationen von Graphen

Adjazenzliste

- Verwende **array** $A[1 \ldots n]$ von verketteten Listen
- $A[v]$: Liste aller von $v \in V$ aus **erreichbaren Knoten**

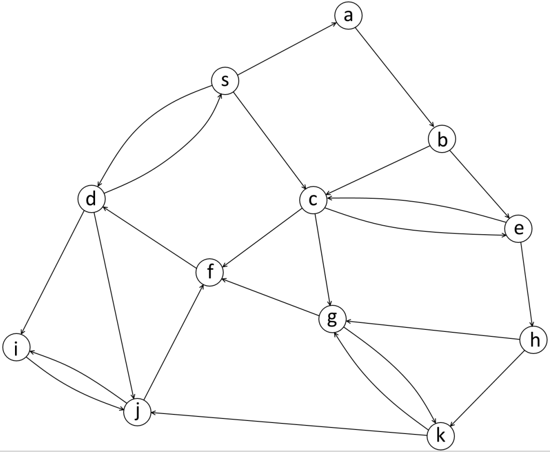
+ Alle Features vom Adjazenzfeld
+ ...und noch mehr: **Einfügen**, **Löschen** von Kanten
− Benötigt mehr **Platz** (für Zeiger)
− **Cachefeindlicher**



Repräsentationen von Graphen

Aufgabe 1: Malen nach Zahlen

Stellt diesen Graphen als Adjazenzfeld, Adjazenzmatrix und Kantenfolge dar (alphabetisch geordnet).



Graphen durchlaufen

- **Geg.**: Startknoten $s \in V$
- **Ziel**: Von s aus alle weiteren Knoten besuchen
- **Aber**: Keine **Doppel**besuche/Endlosschleifen $\Rightarrow$ **Merke** besuchte Knoten

Graphen durchlaufen – Tiefensuche

Intuitive Implementierung: Tiefensuche

procedure DFS($G = (V, E),\ s \in V$)
  DFSrec($G, s,$ (**false**, ..., **false**))

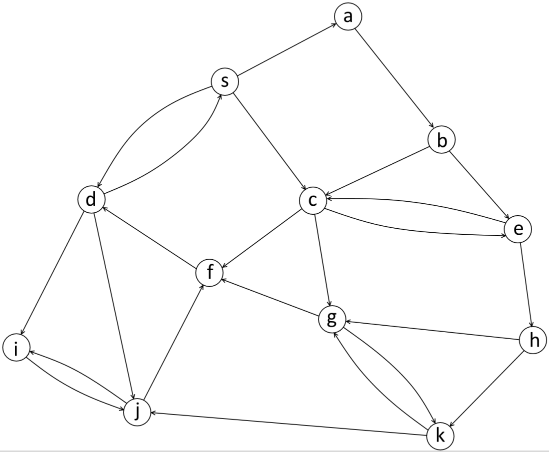
procedure DFSrec($G = (V, E),\ u \in V, visited$: **array of** *Boolean*)
  if $\neg visited[u]$ **then**
    visit(u)   *// Do something with u*
    $visited[u] :=$ **true**

    foreach $(u, v) \in E$ **do**
      DFSrec($G, v, visited$)

Graphen durchlaufen – Tiefensuche

Aufgabe 2: Tiefe in freier Wildbahn

Führt auf diesem Graphen Tiefensuche von *s* ausgehend aus. Nachbarn werden in alphabetischer Reihenfolge besucht.

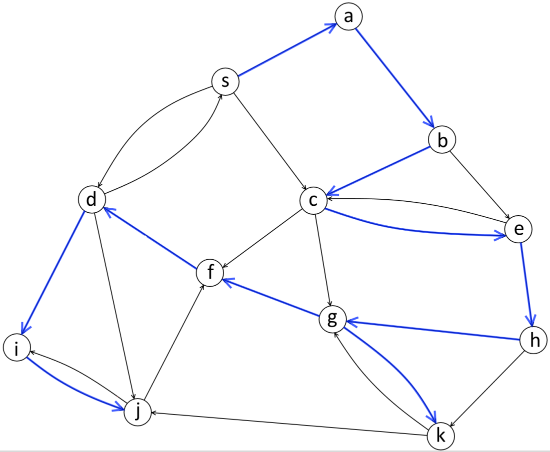


Graphen durchlaufen – Tiefensuche

Lösung von Aufgabe 2

Besuchsreihenfolge:

s, a, b, c, e, h, g, f, d, i, j, k



Graphen durchlaufen – Tiefensuche

- **Beobachtung**: Dringt schnell **tief** in den Graphen ein, anstatt sich „auszubreiten“ (daher der Name)
- **Laufzeit**? $\Theta(n + m)$
  In-place? **Nein.** (wegen *visited* und Rekursion)
- Etwas chaotische Laufwege – geht’s auch organisierter?

Graphen durchlaufen – Breitensuche

Organisierte Reihenfolge: Breitensuche

procedure BFS($G = (V, E),\ s \in V$)
- $visited := ($**false**$, ..., $**false**$)$
- $Q := \{s\}$
- **while** $Q \neq \varnothing$ **do**
  - $u := Q$.popFront()
  - **if** $\neg visited[u]$ **then**
    - visit(u) *// Do something with u*
    - $visited[u] :=$ **true**
    - **foreach** $(u, v) \in E$ **do**
      - Q.pushBack(v)

Graphen durchlaufen – Breitensuche

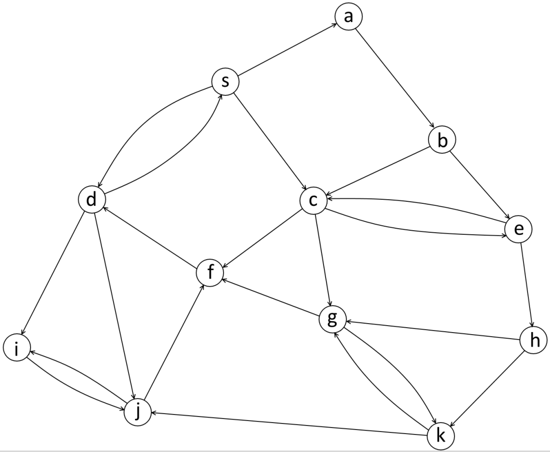
Organisierte Reihenfolge: Breitensuche **mit Layer-Counter**

procedure BFS-with-LayerCounter($G = (V, E),\ s \in V$)
- *visited* := (**false**, ..., **false**)
- $Q := \{s\}, \quad Q' := \varnothing$ *// Extra queue Q'*
- $layer := 0$ *// For counting the layers of traversal*
- **while** $Q \neq \varnothing$ **do**
  - $u := Q$.popFront()
  - **if** $\neg visited[u]$ **then**
    - visit($u, layer$)   *// Do something with u and layer*
    - $visited[u] :=$ **true**
    - **foreach** $(u, v) \in E$ **do**
      - Q'.pushBack(v)   *// Append to next-queue Q'*
  - **if** $Q = \varnothing$ **then**
    - $(Q, Q') := (Q', Q)$ *// New layer, so swap queues*
    - $layer{++}$

Graphen durchlaufen – Breitensuche

Aufgabe 3: Volle Breitseite

Führt auf diesem Graphen Breitensuche von *s* ausgehend aus. Nachbarn werden in alphabetischer Reihenfolge besucht.

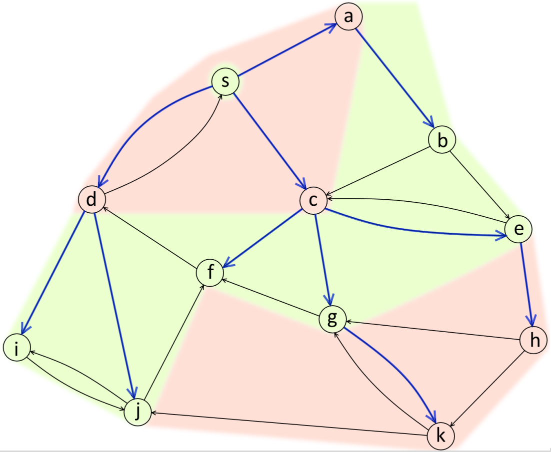


Graphen durchlaufen – Breitensuche

Lösung von Aufgabe 3

Besuchsreihenfolge:

s, a, c, d, b, e, f, g, i, j, h, k



Graphen durchlaufen – Breitensuche

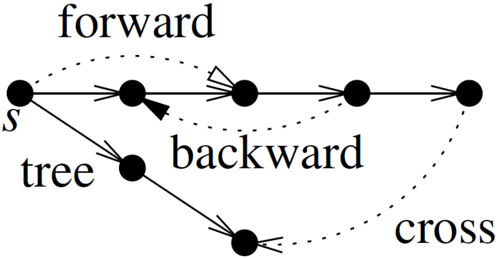
- **Beobachtung**: **Breitet** sich schnell stark **aus** (daher der Name)
- Offensichtlich: Findet **kürzeste Pfade** (bei **ungewichteten** Kanten)
- **Laufzeit**? $\Theta(n + m)$
  In-place **Nein.** (wegen *visited*, Q und Q')

Graphen durchlaufen – Kantentypen

- Bei BFS/DFS „entlanggelaufene“ Kanten **bilden Baum** (da kein Knoten zweimal besucht!)

⇒ Teile Kanten ein:

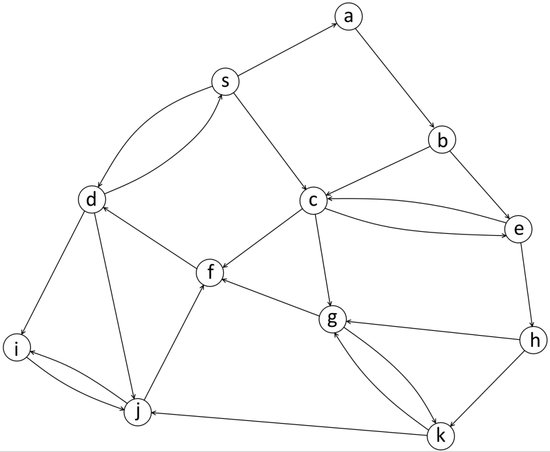
tree-:	„Entlanggelaufene“ Kanten des Baumes
cross-:	Kanten **zwischen** versch. **„Ästen“** im Baum
backward-:	Kanten, die **rückwärts** zu (einer/mehreren) *tree*-Kanten laufen
forward-:	Kanten, die **mehrere** *tree*-Kanten „**überholen**“



Graphen durchlaufen – Kantentypen

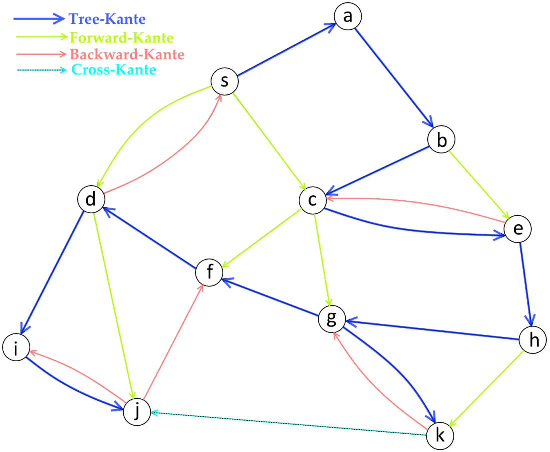
Aufgabe 4: Die Graphschaft besichtigen

Betrachtet die vorhin durchgespielte Tiefen- und Breitensuche und klassifiziert jeweils alle Kanten entsprechend.



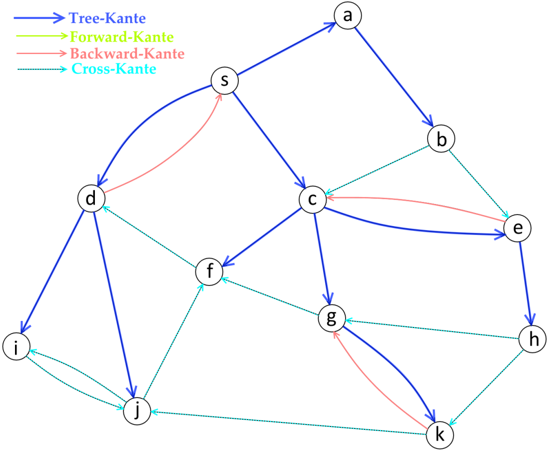
Graphen durchlaufen – Kantentypen

Lösung zu Aufgabe 4.1: für Tiefensuche



Graphen durchlaufen – Kantentypen

Lösung zu Aufgabe 4.2: für Breitensuche



Graphen durchlaufen – Kantentypen

- Gibt es eine Art von Kante, die bei Breitensuche nicht auftreten kann? Falls ja, warum?

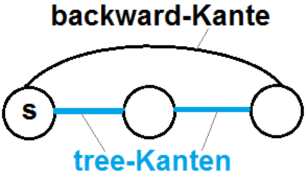
$\Rightarrow$ *forward*-Kanten können **nicht** auftreten
(BFS bestimmt schon den Pfad mit kleinster Kantenanzahl)

Graphen durchlaufen – Kantentypen

- Gibt es eine Art von Kante, die bei Tiefensuche nicht auftreten kann? Falls ja, warum?
- Bei Tiefensuche können **alle** Arten von Kanten auftreten.

Graphen durchlaufen – Kantentypen

- Gibt es eine Art von Kante, die bei Tiefensuche **auf ungerichteten Graphen** nicht auftreten kann? Falls ja, warum?
- *cross*-Kanten können nicht auftreten:
  Wäre nämlich schon **vorher** entlanggelaufen worden (da ungerichtet!). Die einzigen Kanten, die hier das Ende eines Tiefensuch-Astes markieren können, sind *backward-*/*forward*-Kanten. (Ob man die jetzt *backward*- oder *forward*- nennt, ist wurscht, sind ja faktisch **beides**.) Bsp. dazu:



Graphen durchlaufen – Kantentypen

- Sind *cross*-Kanten eindeutig? Falls ja, warum?
- *cross*-Kanten sind genau dann eindeutig, wenn der zugehörige Baum eindeutig ist. $\Rightarrow$ I. A. **nicht** der Fall (da Nachbarn i. A. nicht in bestimmter Reihenfolge gewählt).

Graphen durchlaufen – Kantentypen

- Nach welcher Strategie muss bei Tiefensuche die Reihenfolge der rekursiven Abstiege (also die Reihenfolge der Nachbarn) gewählt werden, damit keine *forward*-Kanten auftreten?
- **Fangfrage**! :P
  Es gibt **keine** solche Strategie; *forward*-Kanten bei DFS in manchen Fällen unvermeidbar

Graphenbeweise

Aufgabe 5: I Wanna Ride My Acycle!

Es sei $G = (V, E)$ ein gerichteter azyklischer Graph (DAG) mit endlich vielen und mindestens einem Knoten. Zeige, dass G mindestens einen Knoten mit Eingangsgrad 0 besitzt.

Lösung zu Aufgabe 5

Angenommen, jeder Knoten hat Eingangsgrad $\geqslant 1$. ($=$ Gegenteil.)
Nehme irgendeinen Knoten v. Auf diesen zeigt also garantiert ne **Kante**.
Laufe sie **rückwärts** $\rightsquigarrow$ neuer Knoten. **Wiederhole** beliebig oft (das geht dank Voraussetzung!).
Also geht das öfter, als G Knoten hat $\Rightarrow$ Irgendwann ein Knoten 2x besucht $\Rightarrow$ Wir laufen im Kreis $\Rightarrow$ $\lightning$ G kreisfrei.

Schönes Wochenende! ☺