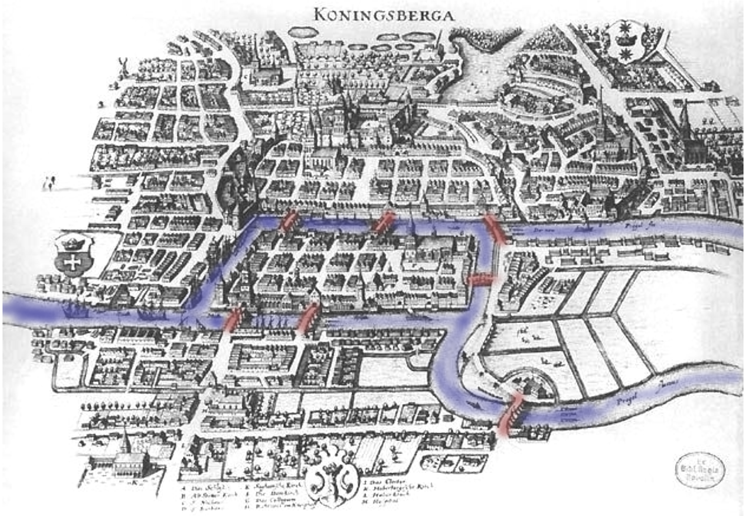
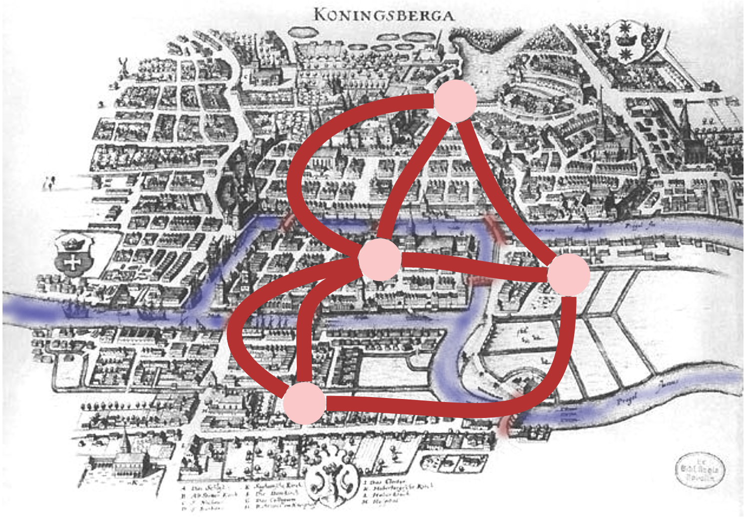
12. Graphen

Notation, Repräsentation, Traversieren (DFS, BFS), Topologisches Sortieren
[Ottman/Widmayer, Kap. 9.1 - 9.4,Cormen et al, Kap. 22]

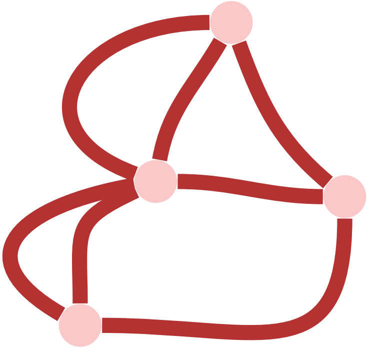
Königsberg 1736



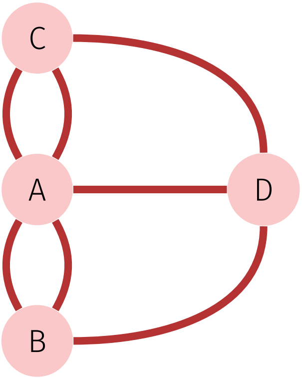
Königsberg 1736



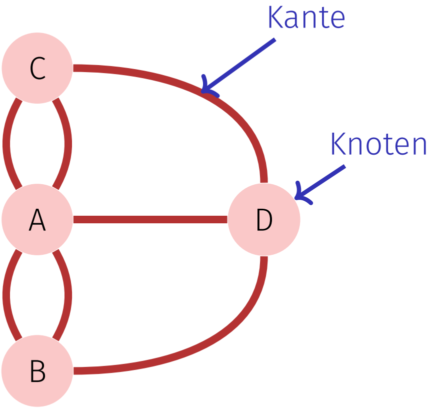
Königsberg 1736



[Multi]Graph

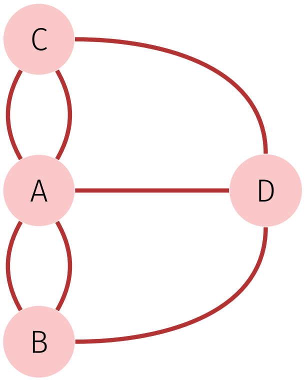


[Multi]Graph



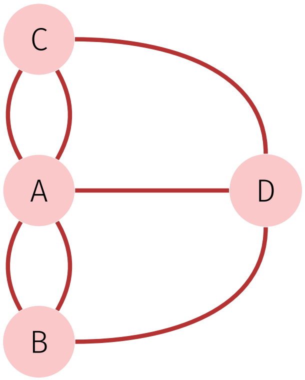
Zyklen

- Gibt es einen Rundweg durch die Stadt (den Graphen), welcher jede Brücke (jede Kante) genau einmal benutzt?



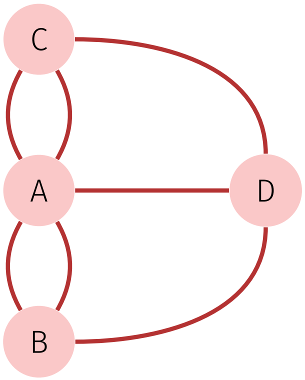
Zyklen

- Gibt es einen Rundweg durch die Stadt (den Graphen), welcher jede Brücke (jede Kante) genau einmal benutzt?
- Euler (1736): nein.



Zyklen

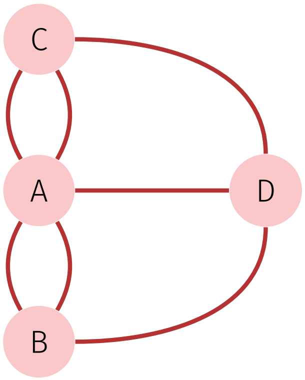
- Gibt es einen Rundweg durch die Stadt (den Graphen), welcher jede Brücke (jede Kante) genau einmal benutzt?
- Euler (1736): nein.
- Solcher Rundweg (*Zyklus*) heisst **Eulerscher Kreis**.



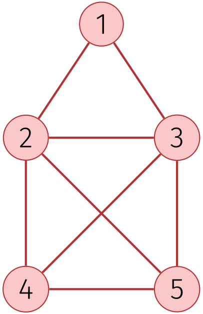
Zyklen

- Gibt es einen Rundweg durch die Stadt (den Graphen), welcher jede Brücke (jede Kante) genau einmal benutzt?
- Euler (1736): nein.
- Solcher Rundweg (*Zyklus*) heisst **Eulerscher Kreis**.
- Eulerzyklus $\Leftrightarrow$ jeder Knoten hat gerade Anzahl Kanten (jeder Knoten hat einen *geraden Grad*).

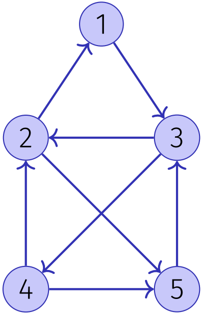
  "$\Rightarrow$" ist sofort klar, "$\Leftarrow$" ist etwas schwieriger, aber auch elementar.



Notation



ungerichtet



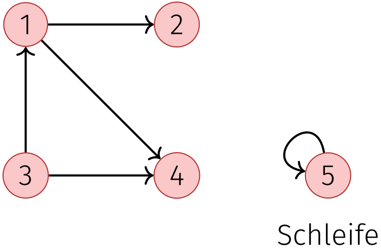
gerichtet

$$V = \{1,2,3,4,5\}$$
$$E = \{\{1,2\},\{1,3\},\{2,3\},\{2,4\},\{2,5\},\{3,4\},\{3,5\},\{4,5\}\}$$

$$V = \{1,2,3,4,5\}$$
$$E = \{(1,3),(2,1),(2,5),(3,2),(3,4),(4,2),(4,5),(5,3)\}$$

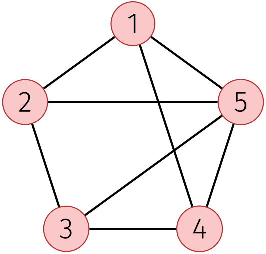
Notation

Ein **gerichteter Graph** besteht aus einer Menge $V = \{v_1, \ldots, v_n\}$ von Knoten (*Vertices*) und einer Menge $E \subseteq V \times V$ von Kanten (*Edges*). Gleiche Kanten dürfen nicht mehrfach enthalten sein.



Notation

Ein **ungerichteteter Graph** besteht aus einer Menge $V = \{v_1, \ldots, v_n\}$ von Knoten und einer Menge $E \subseteq \{\{u, v\} | u, v \in V\}$ von Kanten. Kanten dürfen nicht mehrfach enthalten sein.[15]

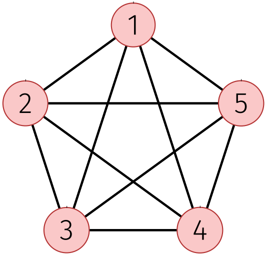


ungerichter Graph

[15] Im Gegensatz zum Eingangsbeispiel – dann Multigraph genannt.

Notation

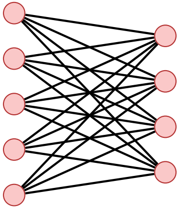
Ein ungerichteter Graph $G = (V, E)$ ohne Schleifen in dem jeder Knoten mit jedem anderen Knoten durch eine Kante verbunden ist, heisst **vollständig**.



ein vollständiger ungerichter Graph

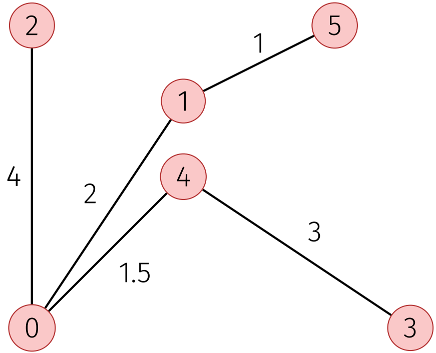
Notation

Ein Graph, bei dem V so in disjunkte U und W aufgeteilt werden kann, dass alle $e \in E$ einen Knoten in U und einen in W haben heisst **bipartit**.



Notation

Ein **gewichteter Graph** $G = (V, E, c)$ ist ein Graph $G = (V, E)$ mit einer **Kantengewichtsfunktion** $c : E \to \mathbb{R}$. $c(e)$ heisst **Gewicht** der Kante e.



Notation

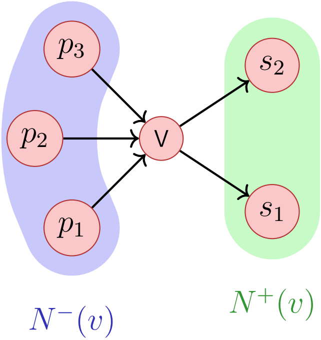
Für gerichtete Graphen $G = (V, E)$

- $w \in V$ heisst **adjazent** zu $v \in V$, falls $(v, w) \in E$

Notation

Für gerichtete Graphen $G = (V, E)$

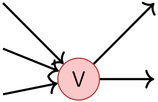
- $w \in V$ heisst **adjazent** zu $v \in V$, falls $(v, w) \in E$
- **Vorgängermenge** von $v \in V$: $N^-(v) := \{u \in V | (u, v) \in E\}$.
  Nachfolgermenge: $N^+(v) := \{u \in V | (v, u) \in E\}$



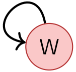
Notation

Für gerichtete Graphen $G = (V, E)$

- **Eingangsgrad**: $\deg^-(v) = |N^-(v)|$,
  Ausgangsgrad: $\deg^+(v) = |N^+(v)|$



$\deg^-(v) = 3$, $\deg^+(v) = 2$



$\deg^-(w) = 1$, $\deg^+(w) = 1$

Notation

Für ungerichtete Graphen $G = (V, E)$:

- $w \in V$ heisst **adjazent** zu $v \in V$, falls $\{v, w\} \in E$

Notation

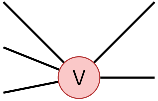
Für ungerichtete Graphen $G = (V, E)$:

- $w \in V$ heisst **adjazent** zu $v \in V$, falls $\{v, w\} \in E$
- **Nachbarschaft** von $v \in V$: $N(v) = \{w \in V | \{v, w\} \in E\}$

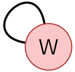
Notation

Für ungerichtete Graphen $G = (V, E)$:

- $w \in V$ heisst **adjazent** zu $v \in V$, falls $\{v, w\} \in E$
- **Nachbarschaft** von $v \in V$: $N(v) = \{w \in V | \{v, w\} \in E\}$
- **Grad** von v: $\deg(v) = |N(v)|$ mit Spezialfall Schleifen: erhöhen Grad um 2.



$\deg(v) = 5$



$\deg(w) = 2$

Beziehung zwischen Knotengraden und Kantenzahl

In jedem Graphen $G = (V, E)$ gilt

1. $\sum_{v \in V} \deg^-(v) = \sum_{v \in V} \deg^+(v) = |E|$, falls G gerichtet
2. $\sum_{v \in V} \deg(v) = 2|E|$, falls G ungerichtet.

Wege

- **Weg**: Sequenz von Knoten $\langle v_1, \ldots, v_{k+1} \rangle$ so dass für jedes $i \in \{1 \ldots k\}$ eine Kante von v_i nach v_{i+1} existiert.

Wege

- **Weg**: Sequenz von Knoten $\langle v_1, \dots, v_{k+1} \rangle$ so dass für jedes $i \in \{1 \dots k\}$ eine Kante von v_i nach v_{i+1} existiert.
- **Länge** des Weges: Anzahl enthaltene Kanten k.

Wege

- **Weg**: Sequenz von Knoten $\langle v_1, \ldots, v_{k+1} \rangle$ so dass für jedes $i \in \{1 \ldots k\}$ eine Kante von v_i nach v_{i+1} existiert.
- **Länge** des Weges: Anzahl enthaltene Kanten k.
- **Gewicht** des Weges (in gewichteten Graphen): $\sum_{i=1}^{k} c((v_i, v_{i+1}))$ (bzw. $\sum_{i=1}^{k} c(\{v_i, v_{i+1}\})$)

Wege

- **Weg**: Sequenz von Knoten $\langle v_1, \ldots, v_{k+1} \rangle$ so dass für jedes $i \in \{1 \ldots k\}$ eine Kante von v_i nach v_{i+1} existiert.
- **Länge** des Weges: Anzahl enthaltene Kanten k.
- **Gewicht** des Weges (in gewichteten Graphen): $\sum_{i=1}^{k} c((v_i, v_{i+1}))$ (bzw. $\sum_{i=1}^{k} c(\{v_i, v_{i+1}\})$)
- **Pfad** (auch: einfacher Pfad): Weg der keinen Knoten mehrfach verwendet.

Zusammenhang

- Ungerichteter Graph heisst **zusammenhängend**, wenn für jedes Paar $v, w \in V$ ein verbindender Weg existiert.
- Gerichteter Graph heisst **stark zusammenhängend**, wenn für jedes Paar $v, w \in V$ ein verbindender Weg existiert.
- Gerichteter Graph heisst **schwach zusammenhängend**, wenn der entsprechende ungerichtete Graph zusammenhängend ist.

Einfache Beobachtungen

- Allgemein: $0 \leq |E| \in \mathcal{O}(|V|^2)$
- Zusammenhängender Graph: $|E| \in \Omega(|V|)$
- Vollständiger Graph: $|E| = \frac{|V| \cdot (|V|-1)}{2}$ (ungerichtet)
- Maximal $|E| = |V|^2$ (gerichtet),$|E| = \frac{|V| \cdot (|V|+1)}{2}$ (ungerichtet)

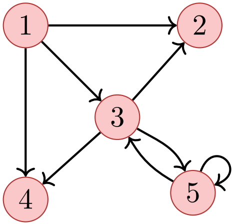
Zyklen

- **Zyklus**: Weg $\langle v_1, \ldots, v_{k+1} \rangle$ mit $v_1 = v_{k+1}$
- **Kreis**: Zyklus mit paarweise verschiedenen $v_1, \ldots, v_k$, welcher keine Kante mehrfach verwendet.
- **Kreisfrei (azyklisch)**: Graph ohne jegliche Kreise.

Eine Folgerung: Ungerichtete Graphen können keinen Kreis der Länge 2 enthalten (Schleifen haben Länge 1).

Repräsentation mit Matrix

Graph $G = (V, E)$ mit Knotenmenge $v_1, \dots, v_n$ gespeichert als **Adjazenzmatrix** $A_G = (a_{ij})_{1 \le i,j \le n}$ mit Einträgen aus $\{0, 1\}$. $a_{ij} = 1$ genau dann wenn Kante von v_i nach v_j.

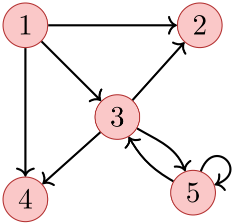


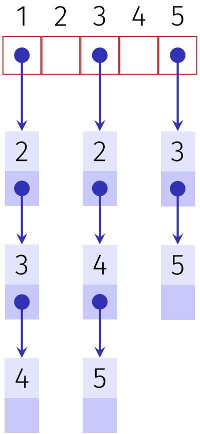
$$\begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \end{pmatrix}$$

Speicherbedarf $\Theta(|V|^2)$. A_G ist symmetrisch, wenn G ungerichtet.

Repräsentation mit Liste

Viele Graphen $G = (V, E)$ mit Knotenmenge $v_1, \ldots, v_n$ haben deutlich weniger als n^2 Kanten. Repräsentation mit **Adjazenzliste**: Array $A[1], \ldots, A[n]$, A_i enthält verkettete Liste aller Knoten in $N^+(v_i)$.





Speicherbedarf $\Theta(|V| + |E|)$.

Laufzeiten einfacher Operationen

Operation	Matrix	Liste
Nachbarn/Nachfolger von $v \in V$ finden		
$v \in V$ ohne Nachbar/Nachfolger finden		
$(u, v) \in E$?		
Kante einfügen		
Kante löschen		

Laufzeiten einfacher Operationen

Operation	Matrix	Liste
Nachbarn/Nachfolger von $v \in V$ finden	$\Theta(n)$	
$v \in V$ ohne Nachbar/Nachfolger finden		
$(u, v) \in E$?		
Kante einfügen		
Kante löschen		

Laufzeiten einfacher Operationen

Operation	Matrix	Liste
Nachbarn/Nachfolger von $v \in V$ finden	$\Theta(n)$	$\Theta(\deg^+ v)$
$v \in V$ ohne Nachbar/Nachfolger finden		
$(u, v) \in E$?		
Kante einfügen		
Kante löschen		

Laufzeiten einfacher Operationen

Operation	Matrix	Liste
Nachbarn/Nachfolger von $v \in V$ finden	$\Theta(n)$	$\Theta(\deg^+ v)$
$v \in V$ ohne Nachbar/Nachfolger finden	$\Theta(n^2)$	
$(u, v) \in E$?		
Kante einfügen		
Kante löschen		

Laufzeiten einfacher Operationen

Operation	Matrix	Liste
Nachbarn/Nachfolger von $v \in V$ finden	$\Theta(n)$	$\Theta(\deg^+ v)$
$v \in V$ ohne Nachbar/Nachfolger finden	$\Theta(n^2)$	$\Theta(n)$
$(u, v) \in E$?		
Kante einfügen		
Kante löschen		

Laufzeiten einfacher Operationen

Operation	Matrix	Liste
Nachbarn/Nachfolger von $v \in V$ finden	$\Theta(n)$	$\Theta(\deg^+ v)$
$v \in V$ ohne Nachbar/Nachfolger finden	$\Theta(n^2)$	$\Theta(n)$
$(u, v) \in E$?	$\Theta(1)$	
Kante einfügen		
Kante löschen		

Laufzeiten einfacher Operationen

Operation	Matrix	Liste
Nachbarn/Nachfolger von $v \in V$ finden	$\Theta(n)$	$\Theta(\deg^+ v)$
$v \in V$ ohne Nachbar/Nachfolger finden	$\Theta(n^2)$	$\Theta(n)$
$(u, v) \in E$?	$\Theta(1)$	$\Theta(\deg^+ v)$
Kante einfügen		
Kante löschen		

Laufzeiten einfacher Operationen

Operation	Matrix	Liste
Nachbarn/Nachfolger von $v \in V$ finden	$\Theta(n)$	$\Theta(\deg^+ v)$
$v \in V$ ohne Nachbar/Nachfolger finden	$\Theta(n^2)$	$\Theta(n)$
$(u, v) \in E$?	$\Theta(1)$	$\Theta(\deg^+ v)$
Kante einfügen	$\Theta(1)$	
Kante löschen		

Laufzeiten einfacher Operationen

Operation	Matrix	Liste
Nachbarn/Nachfolger von $v \in V$ finden	$\Theta(n)$	$\Theta(\deg^+ v)$
$v \in V$ ohne Nachbar/Nachfolger finden	$\Theta(n^2)$	$\Theta(n)$
$(u, v) \in E$?	$\Theta(1)$	$\Theta(\deg^+ v)$
Kante einfügen	$\Theta(1)$	$\Theta(1)$
Kante löschen		

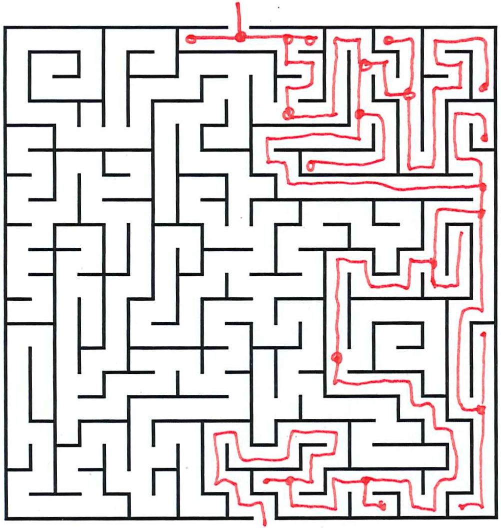
Laufzeiten einfacher Operationen

Operation	Matrix	Liste
Nachbarn/Nachfolger von $v \in V$ finden	$\Theta(n)$	$\Theta(\deg^+ v)$
$v \in V$ ohne Nachbar/Nachfolger finden	$\Theta(n^2)$	$\Theta(n)$
$(u, v) \in E$?	$\Theta(1)$	$\Theta(\deg^+ v)$
Kante einfügen	$\Theta(1)$	$\Theta(1)$
Kante löschen	$\Theta(1)$	

Laufzeiten einfacher Operationen

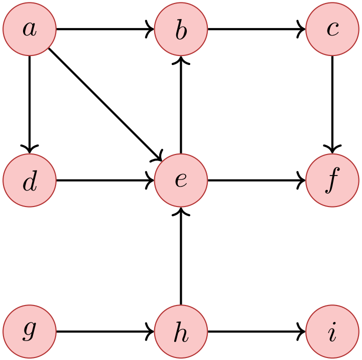
Operation	Matrix	Liste
Nachbarn/Nachfolger von $v \in V$ finden	$\Theta(n)$	$\Theta(\deg^+ v)$
$v \in V$ ohne Nachbar/Nachfolger finden	$\Theta(n^2)$	$\Theta(n)$
$(u, v) \in E$?	$\Theta(1)$	$\Theta(\deg^+ v)$
Kante einfügen	$\Theta(1)$	$\Theta(1)$
Kante löschen	$\Theta(1)$	$\Theta(\deg^+ v)$

Tiefensuche



Graphen Traversieren: Tiefensuche

Verfolge zuerst Pfad in die Tiefe, bis nichts mehr besucht werden kann.

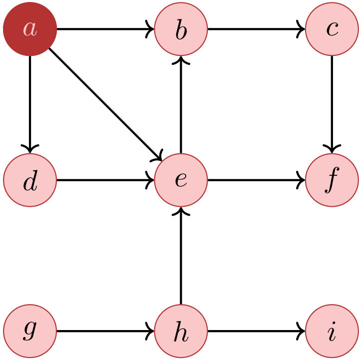


Adjazenzliste

a	b	c	d	e	f	g	h	i
$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$		$\downarrow$	$\downarrow$	
b	c	f	e	b		h	e	
$\downarrow$				$\downarrow$			$\downarrow$	
d				f			i	
$\downarrow$								
e								

Graphen Traversieren: Tiefensuche

Verfolge zuerst Pfad in die Tiefe, bis nichts mehr besucht werden kann.

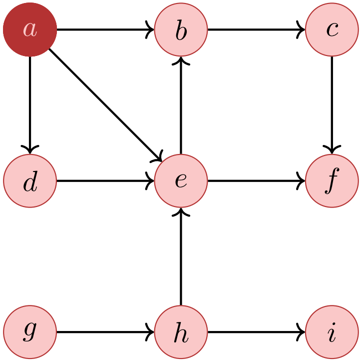


Adjazenzliste

a	b	c	d	e	f	g	h	i
↓	↓	↓	↓	↓		↓	↓	
b	c	f	e	b		h	e	
↓				↓			↓	
d				f			i	
↓								
e								

Graphen Traversieren: Tiefensuche

Verfolge zuerst Pfad in die Tiefe, bis nichts mehr besucht werden kann.

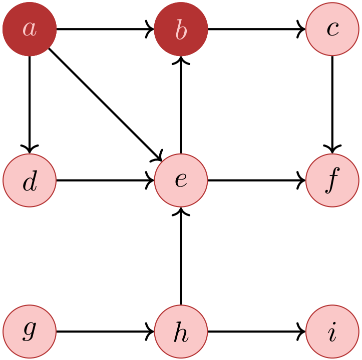


Adjazenzliste

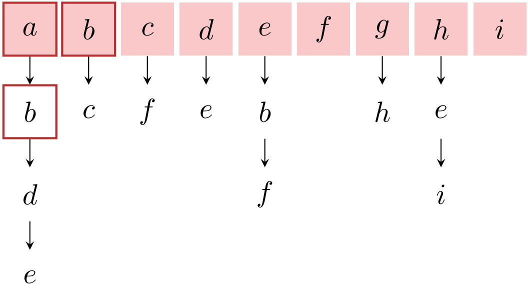
a	b	c	d	e	f	g	h	i
$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$		$\downarrow$	$\downarrow$	
b	c	f	e	b		h	e	
$\downarrow$				$\downarrow$			$\downarrow$	
d				f			i	
$\downarrow$								
e								

Graphen Traversieren: Tiefensuche

Verfolge zuerst Pfad in die Tiefe, bis nichts mehr besucht werden kann.

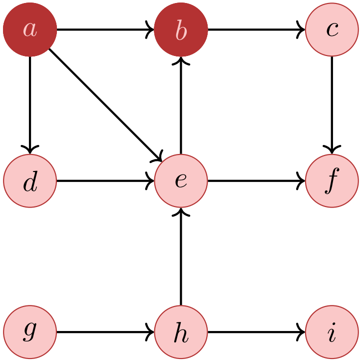


Adjazenzliste

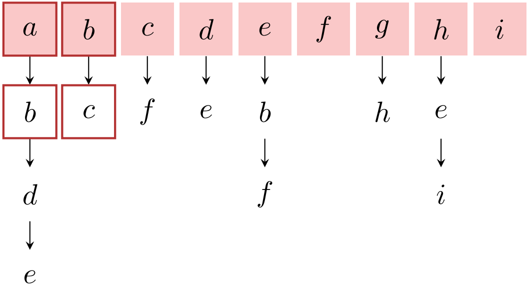


Graphen Traversieren: Tiefensuche

Verfolge zuerst Pfad in die Tiefe, bis nichts mehr besucht werden kann.

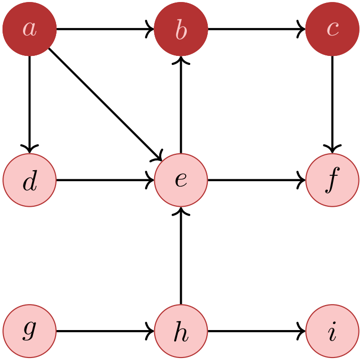


Adjazenzliste

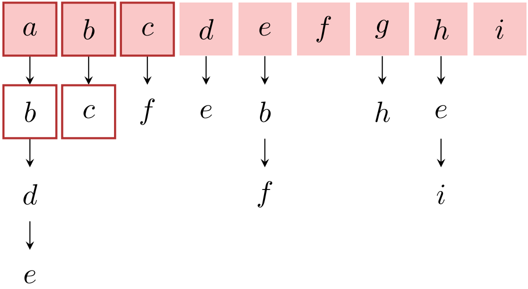


Graphen Traversieren: Tiefensuche

Verfolge zuerst Pfad in die Tiefe, bis nichts mehr besucht werden kann.

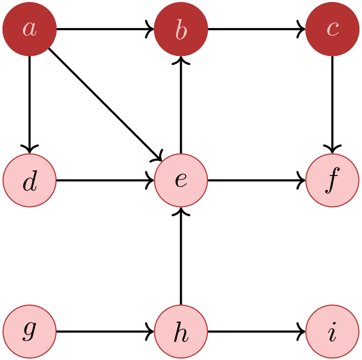


Adjazenzliste

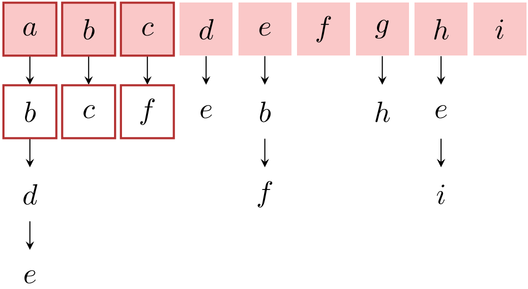


Graphen Traversieren: Tiefensuche

Verfolge zuerst Pfad in die Tiefe, bis nichts mehr besucht werden kann.

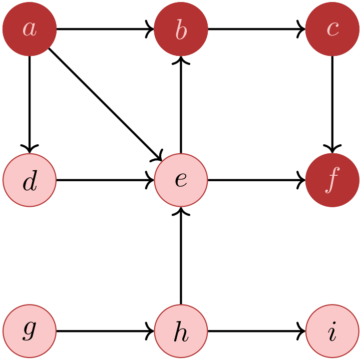


Adjazenzliste

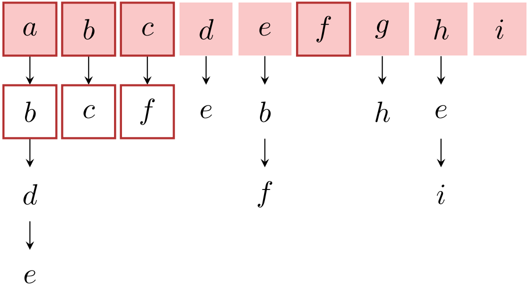


Graphen Traversieren: Tiefensuche

Verfolge zuerst Pfad in die Tiefe, bis nichts mehr besucht werden kann.

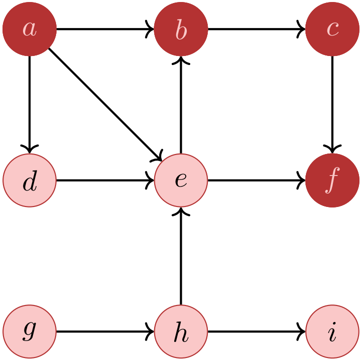


Adjazenzliste

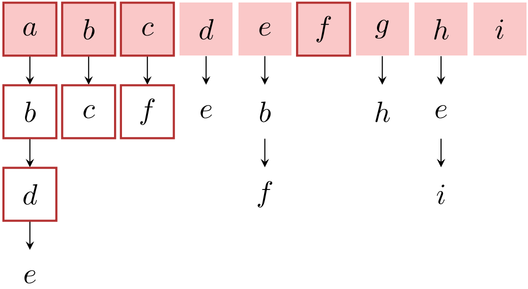


Graphen Traversieren: Tiefensuche

Verfolge zuerst Pfad in die Tiefe, bis nichts mehr besucht werden kann.

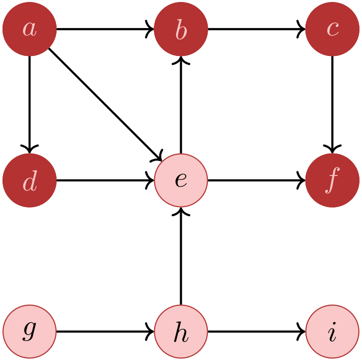


Adjazenzliste

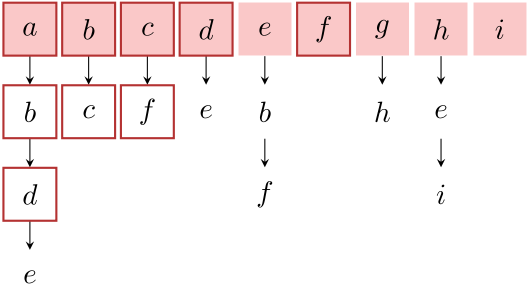


Graphen Traversieren: Tiefensuche

Verfolge zuerst Pfad in die Tiefe, bis nichts mehr besucht werden kann.

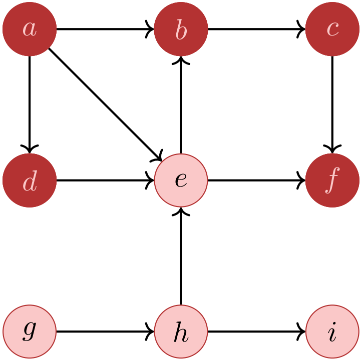


Adjazenzliste

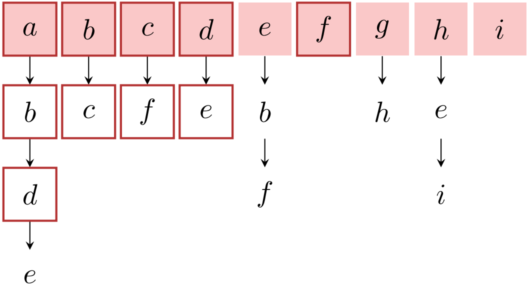


Graphen Traversieren: Tiefensuche

Verfolge zuerst Pfad in die Tiefe, bis nichts mehr besucht werden kann.

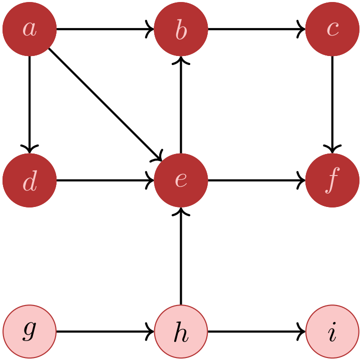


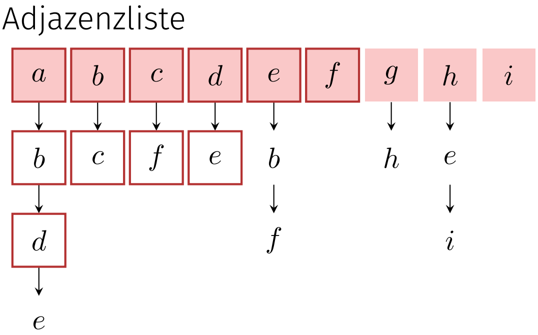
Adjazenzliste



Graphen Traversieren: Tiefensuche

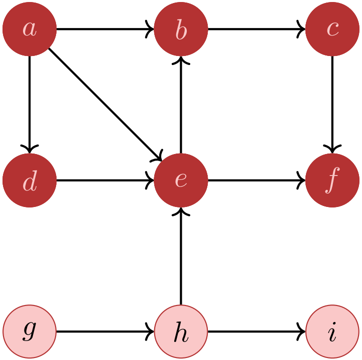
Verfolge zuerst Pfad in die Tiefe, bis nichts mehr besucht werden kann.



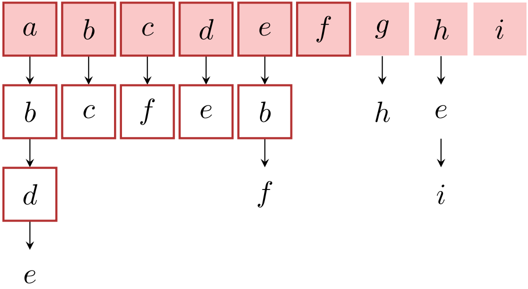


Graphen Traversieren: Tiefensuche

Verfolge zuerst Pfad in die Tiefe, bis nichts mehr besucht werden kann.

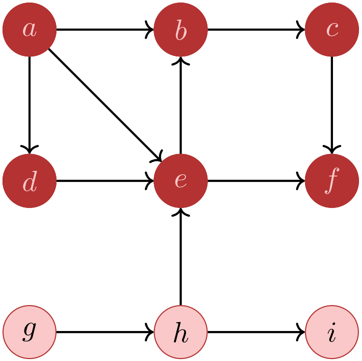


Adjazenzliste

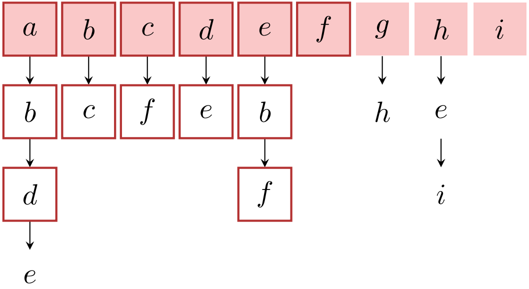


Graphen Traversieren: Tiefensuche

Verfolge zuerst Pfad in die Tiefe, bis nichts mehr besucht werden kann.

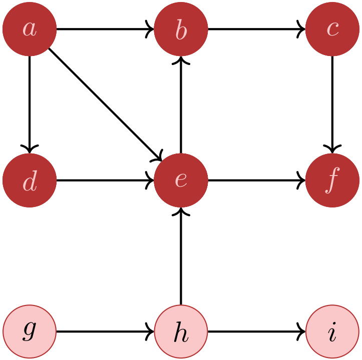


Adjazenzliste



Graphen Traversieren: Tiefensuche

Verfolge zuerst Pfad in die Tiefe, bis nichts mehr besucht werden kann.

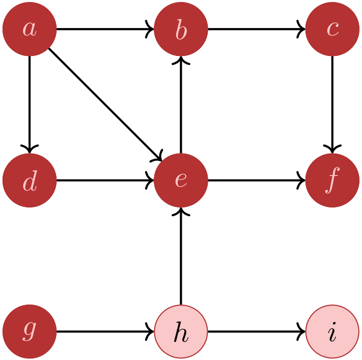


Adjazenzliste

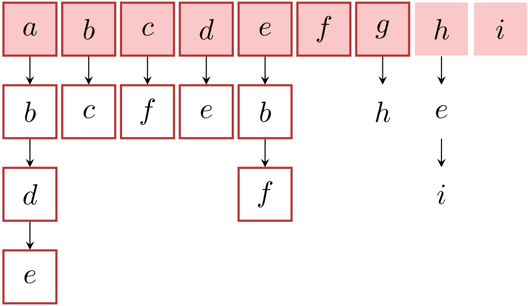
- $a \rightarrow b \rightarrow d \rightarrow e$
- $b \rightarrow c$
- $c \rightarrow f$
- $d \rightarrow e$
- $e \rightarrow b \rightarrow f$
- f
- $g \rightarrow h$
- $h \rightarrow e \rightarrow i$
- i

Graphen Traversieren: Tiefensuche

Verfolge zuerst Pfad in die Tiefe, bis nichts mehr besucht werden kann.

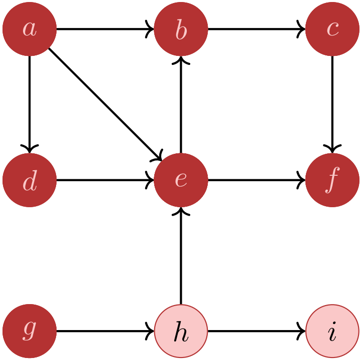


Adjazenzliste

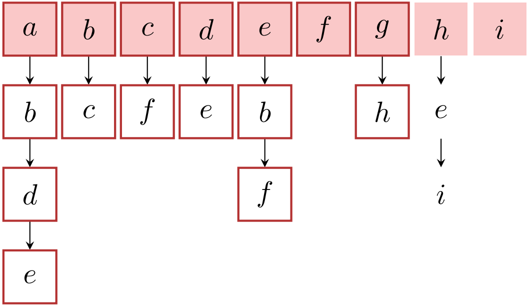


Graphen Traversieren: Tiefensuche

Verfolge zuerst Pfad in die Tiefe, bis nichts mehr besucht werden kann.

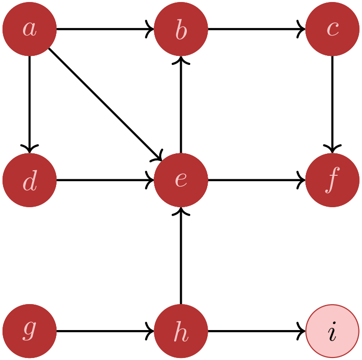


Adjazenzliste

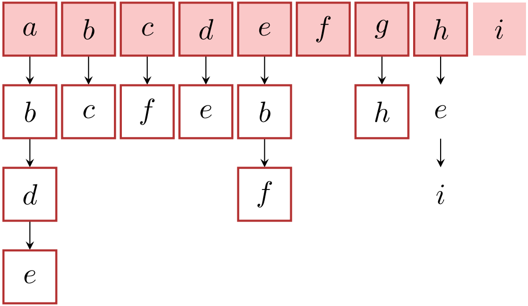


Graphen Traversieren: Tiefensuche

Verfolge zuerst Pfad in die Tiefe, bis nichts mehr besucht werden kann.

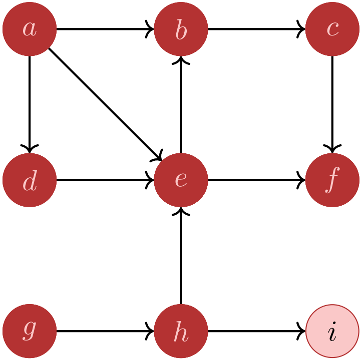


Adjazenzliste

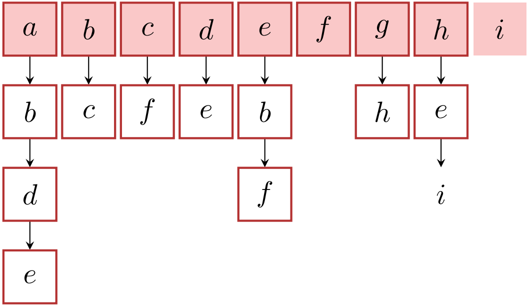


Graphen Traversieren: Tiefensuche

Verfolge zuerst Pfad in die Tiefe, bis nichts mehr besucht werden kann.

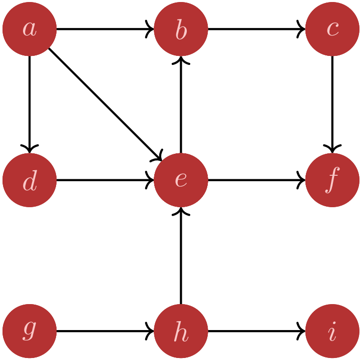


Adjazenzliste



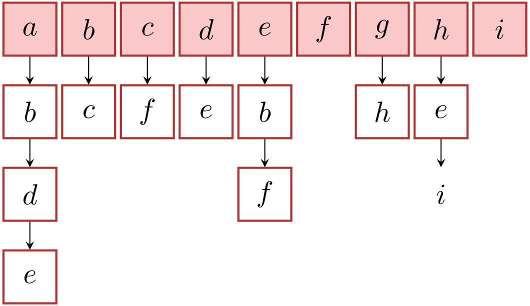
Graphen Traversieren: Tiefensuche

Verfolge zuerst Pfad in die Tiefe, bis nichts mehr besucht werden kann.



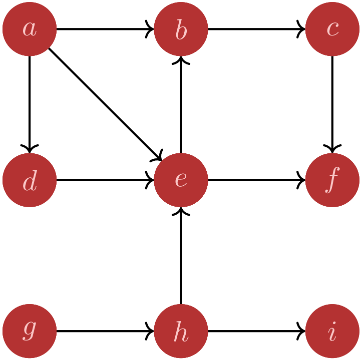
Reihenfolge $a, b, c, f, d, e, g, h, i$

Adjazenzliste



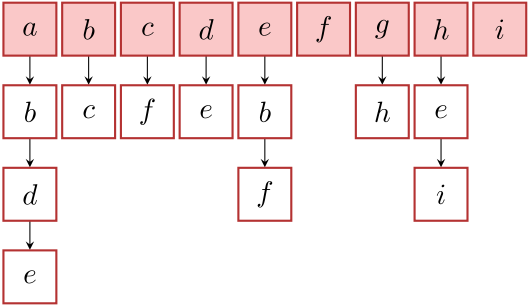
Graphen Traversieren: Tiefensuche

Verfolge zuerst Pfad in die Tiefe, bis nichts mehr besucht werden kann.



Reihenfolge $a, b, c, f, d, e, g, h, i$

Adjazenzliste



Farben

Konzeptuelle Färbung der Knoten

- **Weiss:** Knoten wurde noch nicht entdeckt.
- **Grau:** Knoten wurde entdeckt und zur Traversierung vorgemerkt / in Bearbeitung.
- **Schwarz:** Knoten wurde entdeckt und vollständig bearbeitet

Algorithmus Tiefensuche DFS-Visit(G, v)

Input: Graph $G = (V, E)$, Knoten v.

$v.color \leftarrow$ grey
foreach $w \in N^+(v)$ **do**
  if $w.color =$ white **then**
    DFS-Visit(G, w)

$v.color \leftarrow$ black

Tiefensuche ab Knoten v. Laufzeit (ohne Rekursion): $\Theta(\deg^+ v)$

Algorithmus Tiefensuche DFS-Visit(G)

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

foreach $v \in V$ **do**
  $v.color \leftarrow$ white

foreach $v \in V$ **do**
  if $v.color =$ white **then**
    DFS-Visit(G,v)

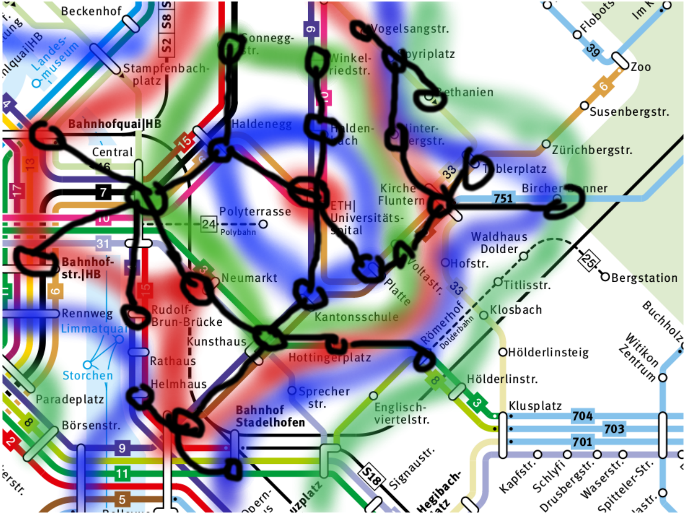
Tiefensuche für alle Knoten eines Graphen. Laufzeit $\Theta(|V| + \sum_{v \in V}(\deg^+(v) + 1)) = \Theta(|V| + |E|)$.

Interpretation der Farben

Beim Traversieren des Graphen wird ein Baum (oder Wald) aufgebaut.
Beim Entdecken von Knoten gibt es drei Fälle

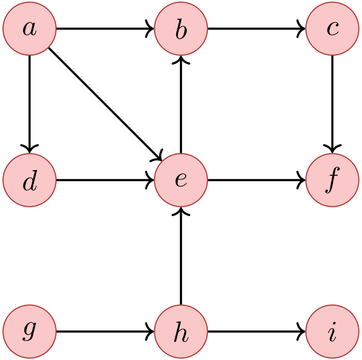
- Weisser Knoten: neue Baumkante
- Grauer Knoten: Zyklus („Rückwärtskante“)
- Schwarzer Knoten: Vorwärts-/Seitwärtskante

Breitensuche

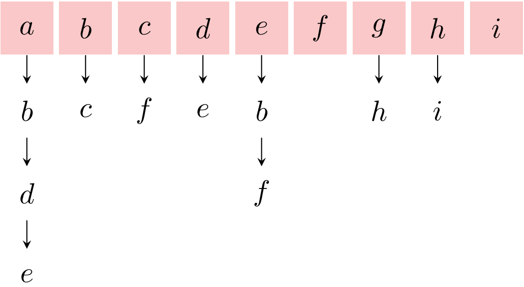


Graphen Traversieren: Breitensuche

Verfolge zuerst Pfad in die Breite, gehe dann in die Tiefe.

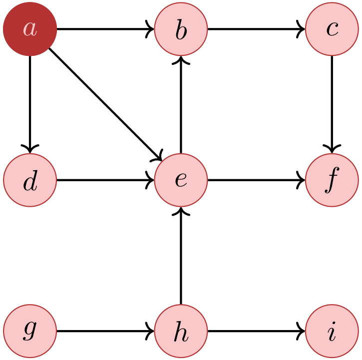


Adjazenzliste

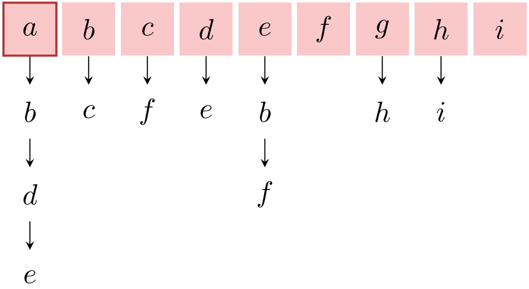


Graphen Traversieren: Breitensuche

Verfolge zuerst Pfad in die Breite, gehe dann in die Tiefe.

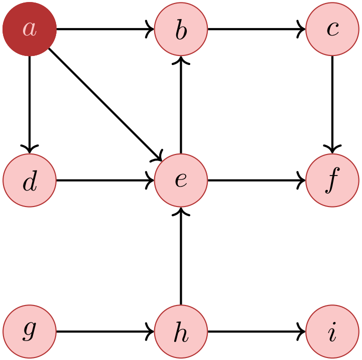


Adjazenzliste

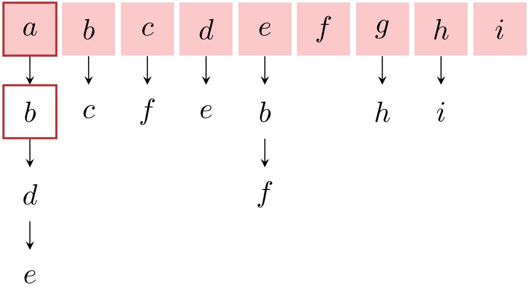


Graphen Traversieren: Breitensuche

Verfolge zuerst Pfad in die Breite, gehe dann in die Tiefe.

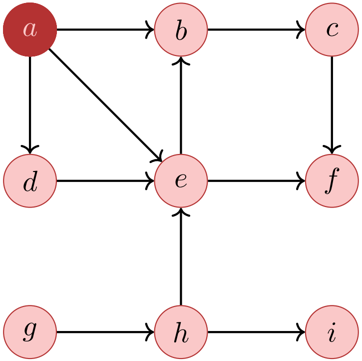


Adjazenzliste

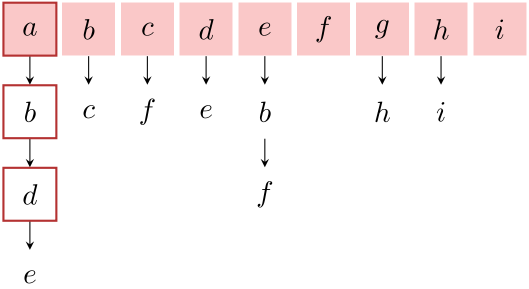


Graphen Traversieren: Breitensuche

Verfolge zuerst Pfad in die Breite, gehe dann in die Tiefe.

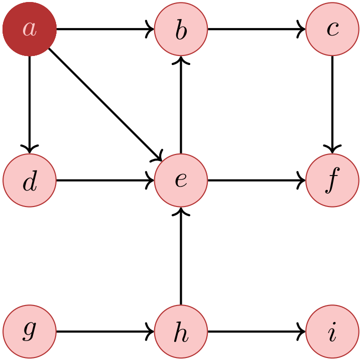


Adjazenzliste

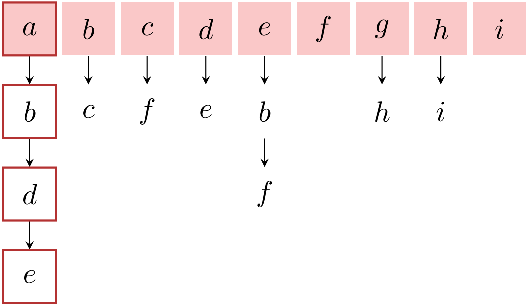


Graphen Traversieren: Breitensuche

Verfolge zuerst Pfad in die Breite, gehe dann in die Tiefe.

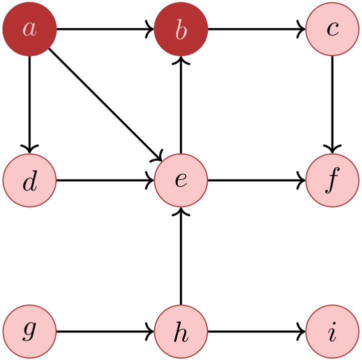


Adjazenzliste

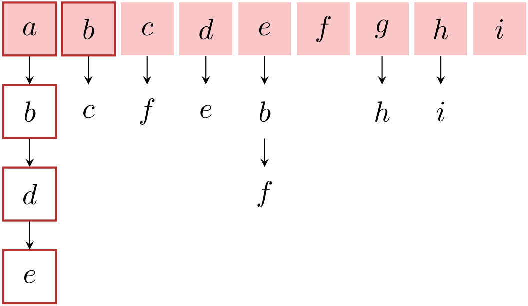


Graphen Traversieren: Breitensuche

Verfolge zuerst Pfad in die Breite, gehe dann in die Tiefe.

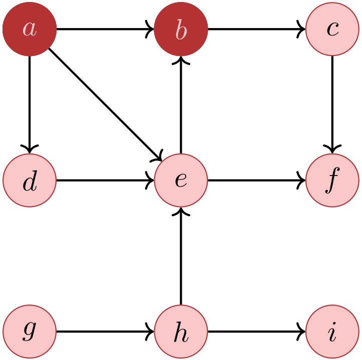


Adjazenzliste

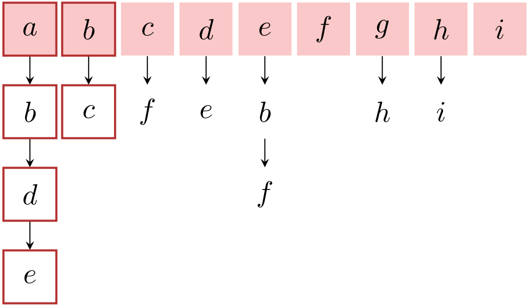


Graphen Traversieren: Breitensuche

Verfolge zuerst Pfad in die Breite, gehe dann in die Tiefe.

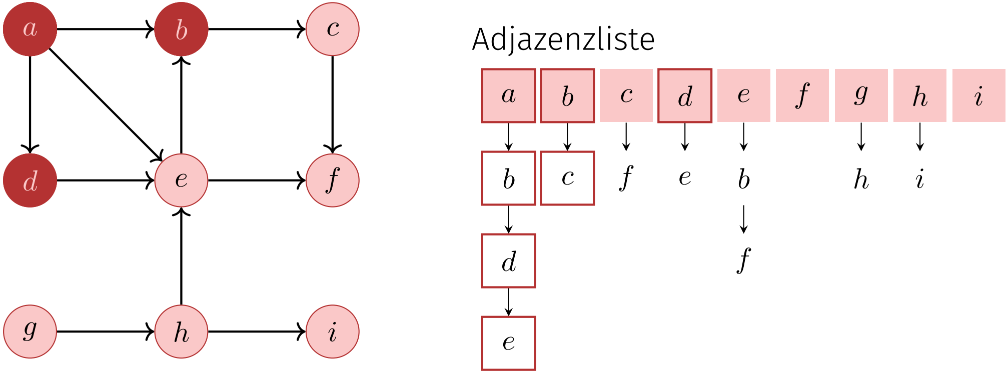


Adjazenzliste



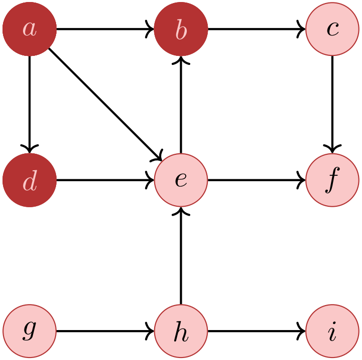
Graphen Traversieren: Breitensuche

Verfolge zuerst Pfad in die Breite, gehe dann in die Tiefe.

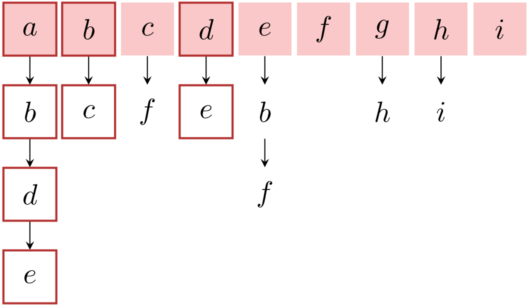


Graphen Traversieren: Breitensuche

Verfolge zuerst Pfad in die Breite, gehe dann in die Tiefe.

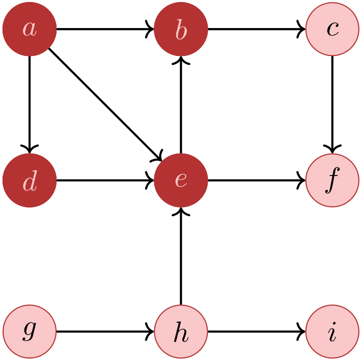


Adjazenzliste

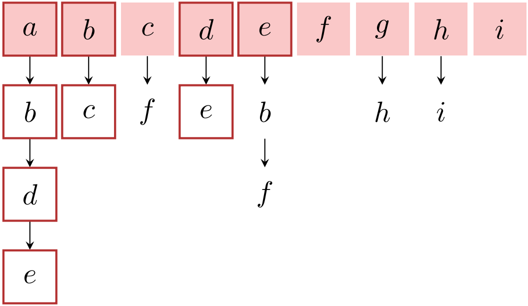


Graphen Traversieren: Breitensuche

Verfolge zuerst Pfad in die Breite, gehe dann in die Tiefe.

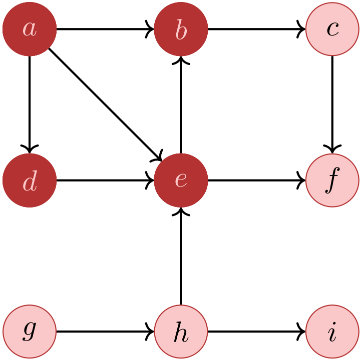


Adjazenzliste

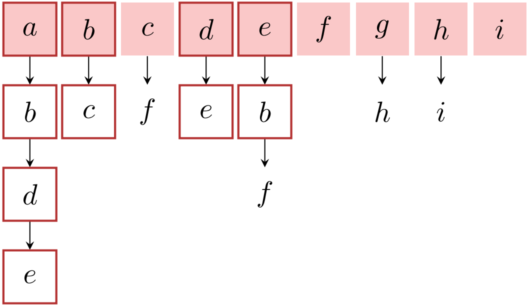


Graphen Traversieren: Breitensuche

Verfolge zuerst Pfad in die Breite, gehe dann in die Tiefe.

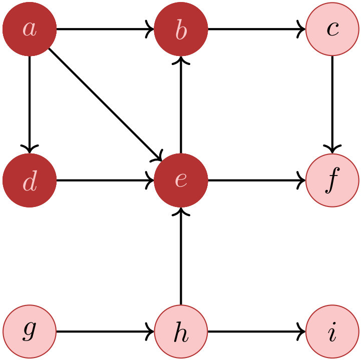


Adjazenzliste

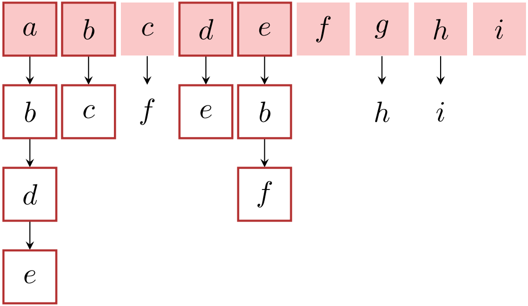


Graphen Traversieren: Breitensuche

Verfolge zuerst Pfad in die Breite, gehe dann in die Tiefe.

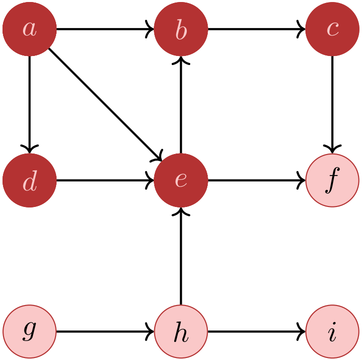


Adjazenzliste

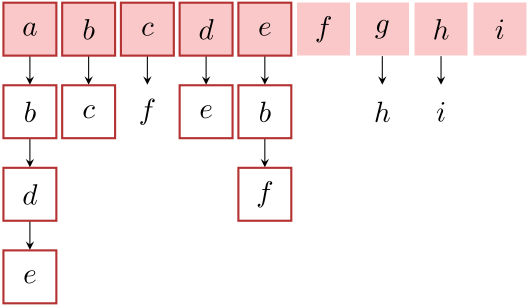


Graphen Traversieren: Breitensuche

Verfolge zuerst Pfad in die Breite, gehe dann in die Tiefe.

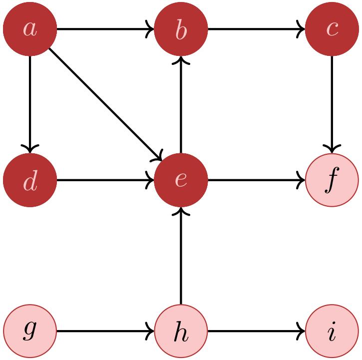


Adjazenzliste

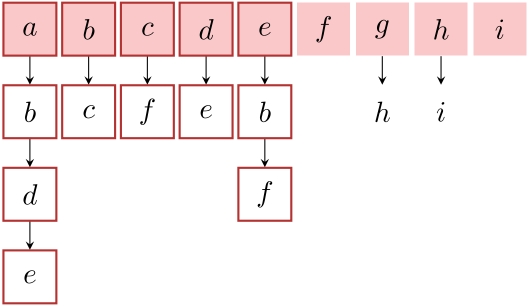


Graphen Traversieren: Breitensuche

Verfolge zuerst Pfad in die Breite, gehe dann in die Tiefe.

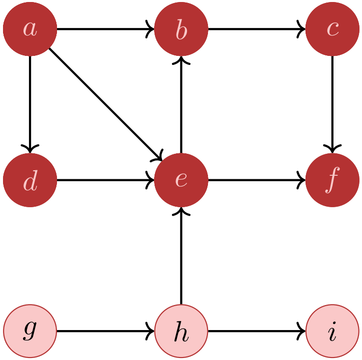


Adjazenzliste

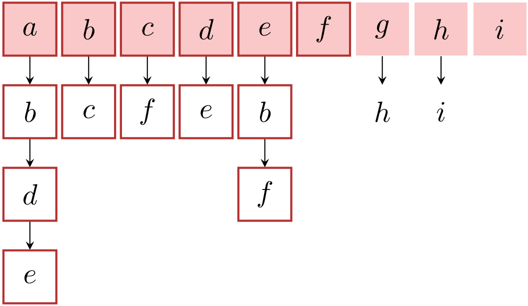


Graphen Traversieren: Breitensuche

Verfolge zuerst Pfad in die Breite, gehe dann in die Tiefe.

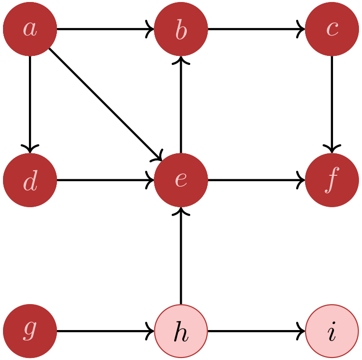


Adjazenzliste

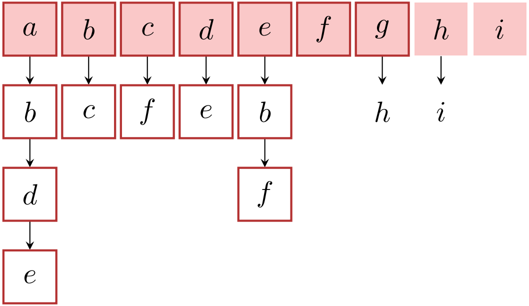


Graphen Traversieren: Breitensuche

Verfolge zuerst Pfad in die Breite, gehe dann in die Tiefe.

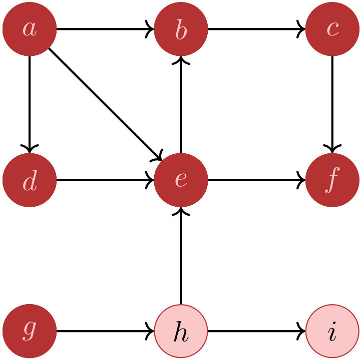


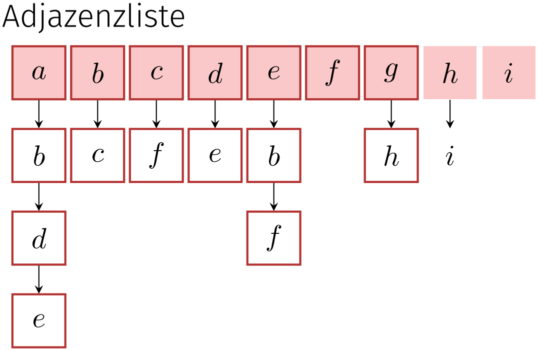
Adjazenzliste



Graphen Traversieren: Breitensuche

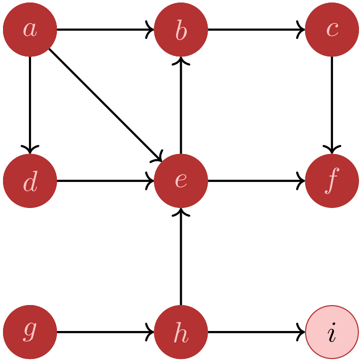
Verfolge zuerst Pfad in die Breite, gehe dann in die Tiefe.



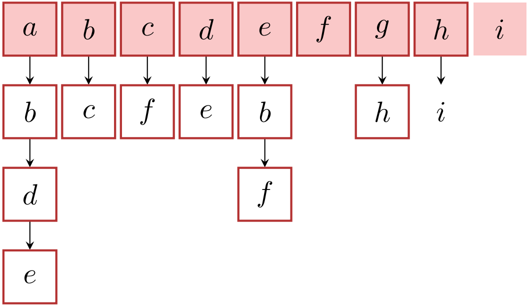


Graphen Traversieren: Breitensuche

Verfolge zuerst Pfad in die Breite, gehe dann in die Tiefe.

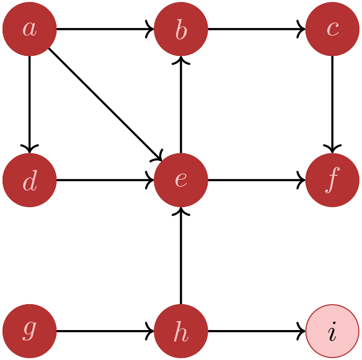


Adjazenzliste

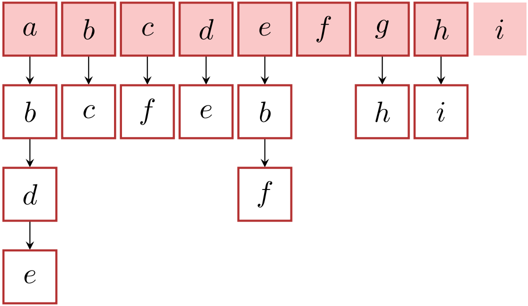


Graphen Traversieren: Breitensuche

Verfolge zuerst Pfad in die Breite, gehe dann in die Tiefe.

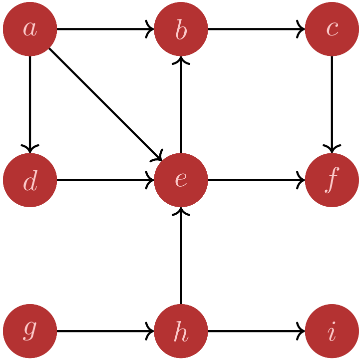


Adjazenzliste



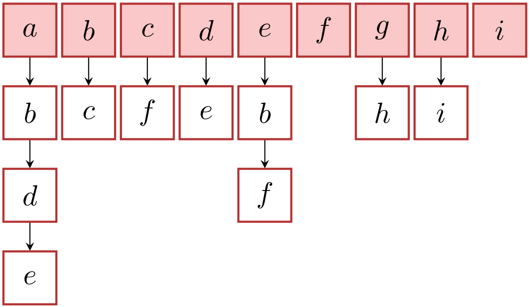
Graphen Traversieren: Breitensuche

Verfolge zuerst Pfad in die Breite, gehe dann in die Tiefe.



Reihenfolge $a, b, d, e, c, f, g, h, i$

Adjazenzliste



(Iteratives) BFS-Visit(G, v)

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

Queue $Q \leftarrow \emptyset$
$v.color \leftarrow$ grey
enqueue(Q, v)
while $Q \neq \emptyset$ **do**
  $w \leftarrow$ dequeue(Q)
  foreach $c \in N^+(w)$ **do**
    if $c.color =$ white **then**
      $c.color \leftarrow$ grey
      enqueue(Q, c)
  $w.color \leftarrow$ black

Algorithmus kommt mit $\mathcal{O}(|V|)$ Extraplatz aus.

Rahmenprogramm BFS-Visit(G)

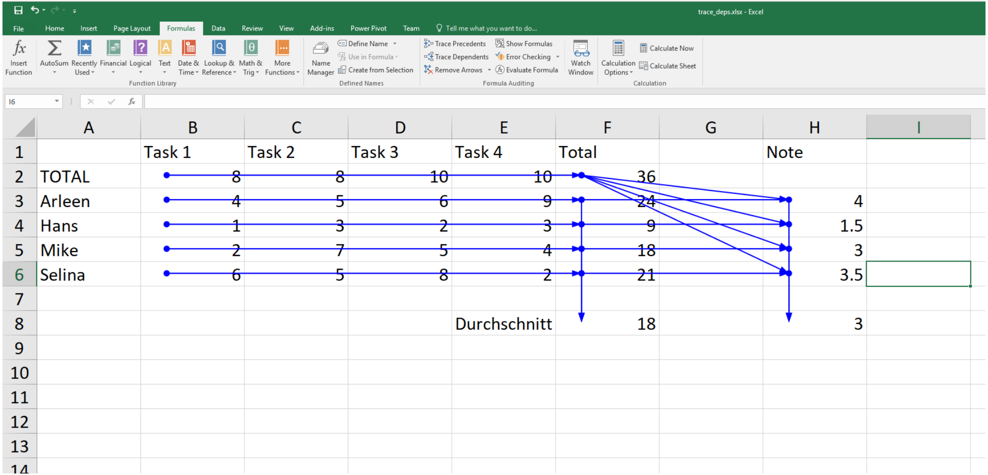
Input: Graph $G = (V, E)$

foreach $v \in V$ **do**
  $v.color \leftarrow$ white

foreach $v \in V$ **do**
  if $v.color =$ white **then**
    BFS-Visit(G,v)

Breitensuche für alle Knoten eines Graphen. Laufzeit $\Theta(|V| + |E|)$.

Topologisches Sortieren



	A	B	C	D	E	F	G	H	I
1		Task 1	Task 2	Task 3	Task 4	Total		Note	
2	TOTAL	8	8	10	10	36			
3	Arleen	4	5	6	9	24		4	
4	Hans	1	3	2	3	9		1.5	
5	Mike	2	7	5	4	18		3	
6	Selina	6	5	8	2	21		3.5	
7									
8					Durchschnitt	18		3	
9									
10									
11									
12									
13									
14									

Auswertungsreihenfolge?

Topologische Sortierung

Topologische Sortierung eines azyklischen gerichteten Graphen $G = (V, E)$:

Bijektive Abbildung

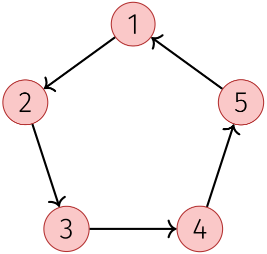
$$\mathrm{ord} : V \to \{1, \ldots, |V|\}$$

so dass

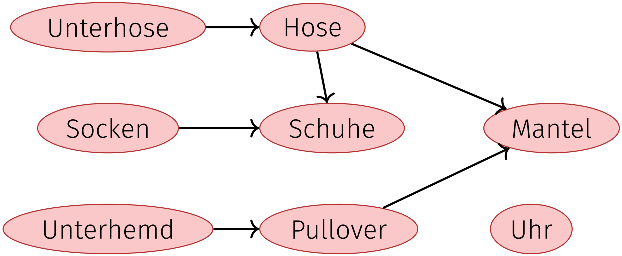
$$\mathrm{ord}(v) < \mathrm{ord}(w) \;\forall\; (v, w) \in E.$$

Identifizieren Wert i mit dem Element $v_i := \mathrm{ord}^{-1}(i)$. Topologische Sortierung $\widehat{=} \langle v_1, \ldots, v_{|V|} \rangle$.

(Gegen-)Beispiele



Zyklischer Graph: kann nicht topologisch sortiert werden.



Eine mögliche Topologische Sortierung des Graphen:
Unterhemd,Pullover,Unterhose,Uhr,Hose,Mantel,Socken,S

Beobachtung

Theorem 7

Ein gerichteter Graph $G = (V, E)$ besitzt genau dann eine topologische Sortierung, wenn er kreisfrei ist

Beobachtung

Theorem 7

Ein gerichteter Graph $G = (V, E)$ besitzt genau dann eine topologische Sortierung, wenn er kreisfrei ist

Beweis "$\Rightarrow$": Wenn G einen Kreis besitzt, so besitzt er keine topologische Sortierung. Denn in einem Kreis $\langle v_{i_1}, \ldots, v_{i_m} \rangle$ gälte $v_{i_1} < \cdots < v_{i_m} < v_{i_1}$.

Induktiver Beweis Gegenrichtung

- Anfang ($n = 1$): Graph mit einem Knoten ohne Schleife ist topologisch sortierbar. Setze $\mathrm{ord}(v_1) = 1$.

Induktiver Beweis Gegenrichtung

- Anfang ($n = 1$): Graph mit einem Knoten ohne Schleife ist topologisch sortierbar. Setze $\mathrm{ord}(v_1) = 1$.
- Hypothese: Graph mit n Knoten kann topologisch sortiert werden.

Induktiver Beweis Gegenrichtung

- Anfang ($n = 1$): Graph mit einem Knoten ohne Schleife ist topologisch sortierbar. Setze $\operatorname{ord}(v_1) = 1$.
- Hypothese: Graph mit n Knoten kann topologisch sortiert werden.
- Schritt ($n \to n + 1$):

Induktiver Beweis Gegenrichtung

- Anfang ($n = 1$): Graph mit einem Knoten ohne Schleife ist topologisch sortierbar. Setze $\mathrm{ord}(v_1) = 1$.
- Hypothese: Graph mit n Knoten kann topologisch sortiert werden.
- Schritt ($n \to n + 1$):

  1. G enthält einen Knoten v_q mit Eingangsgrad $\deg^-(v_q) = 0$. Andernfalls verfolge iterativ Kanten rückwärts – nach spätestens $n + 1$ Iterationen würde man einen Knoten besuchen, welcher bereits besucht wurde. Widerspruch zur Zyklenfreiheit.

Induktiver Beweis Gegenrichtung

- Anfang ($n = 1$): Graph mit einem Knoten ohne Schleife ist topologisch sortierbar. Setze $\mathrm{ord}(v_1) = 1$.
- Hypothese: Graph mit n Knoten kann topologisch sortiert werden.
- Schritt ($n \to n + 1$):

  1. G enthält einen Knoten v_q mit Eingangsgrad $\deg^-(v_q) = 0$. Andernfalls verfolge iterativ Kanten rückwärts – nach spätestens $n + 1$ Iterationen würde man einen Knoten besuchen, welcher bereits besucht wurde. Widerspruch zur Zyklenfreiheit.
  2. Graph ohne Knoten v_q und ohne dessen Eingangskanten kann nach Hypothese topologisch sortiert werden. Verwende diese Sortierung, setze $\mathrm{ord}(v_i) \leftarrow \mathrm{ord}(v_i) + 1$ für alle $i \neq q$ und setze $\mathrm{ord}(v_q) \leftarrow 1$.

Algorithmus, vorläufiger Entwurf

Graph $G = (V, E)$. $d \leftarrow 1$

1. Traversiere von beliebigem Knoten rückwärts bis ein Knoten v_q mit Eingangsgrad 0 gefunden ist.

Laufzeit im schlechtesten Fall:

Algorithmus, vorläufiger Entwurf

Graph $G = (V, E)$. $d \leftarrow 1$

1. Traversiere von beliebigem Knoten rückwärts bis ein Knoten v_q mit Eingangsgrad 0 gefunden ist.
2. Wird kein Knoten mit Eingangsgrad 0 gefunden (n Schritte), dann Zyklus gefunden.

Laufzeit im schlechtesten Fall:

Algorithmus, vorläufiger Entwurf

Graph $G = (V, E)$. $d \leftarrow 1$

1. Traversiere von beliebigem Knoten rückwärts bis ein Knoten v_q mit Eingangsgrad 0 gefunden ist.
2. Wird kein Knoten mit Eingangsgrad 0 gefunden (n Schritte), dann Zyklus gefunden.
3. Setze $\mathrm{ord}(v_q) \leftarrow d$.

Laufzeit im schlechtesten Fall:

Algorithmus, vorläufiger Entwurf

Graph $G = (V, E)$. $d \leftarrow 1$

1. Traversiere von beliebigem Knoten rückwärts bis ein Knoten v_q mit Eingangsgrad 0 gefunden ist.
2. Wird kein Knoten mit Eingangsgrad 0 gefunden (n Schritte), dann Zyklus gefunden.
3. Setze $\mathrm{ord}(v_q) \leftarrow d$.
4. Entferne v_q und seine Kanten von G.

Laufzeit im schlechtesten Fall:

Algorithmus, vorläufiger Entwurf

Graph $G = (V, E)$. $d \leftarrow 1$

1. Traversiere von beliebigem Knoten rückwärts bis ein Knoten v_q mit Eingangsgrad 0 gefunden ist.
2. Wird kein Knoten mit Eingangsgrad 0 gefunden (n Schritte), dann Zyklus gefunden.
3. Setze $\operatorname{ord}(v_q) \leftarrow d$.
4. Entferne v_q und seine Kanten von G.
5. Wenn $V \neq \emptyset$, dann $d \leftarrow d + 1$, gehe zu Schritt 1.

Laufzeit im schlechtesten Fall:

Algorithmus, vorläufiger Entwurf

Graph $G = (V, E)$. $d \leftarrow 1$

1. Traversiere von beliebigem Knoten rückwärts bis ein Knoten v_q mit Eingangsgrad 0 gefunden ist.
2. Wird kein Knoten mit Eingangsgrad 0 gefunden (n Schritte), dann Zyklus gefunden.
3. Setze $\mathrm{ord}(v_q) \leftarrow d$.
4. Entferne v_q und seine Kanten von G.
5. Wenn $V \neq \emptyset$, dann $d \leftarrow d + 1$, gehe zu Schritt 1.

Laufzeit im schlechtesten Fall:

Algorithmus, vorläufiger Entwurf

Graph $G = (V, E)$. $d \leftarrow 1$

1. Traversiere von beliebigem Knoten rückwärts bis ein Knoten v_q mit Eingangsgrad 0 gefunden ist.
2. Wird kein Knoten mit Eingangsgrad 0 gefunden (n Schritte), dann Zyklus gefunden.
3. Setze $\mathrm{ord}(v_q) \leftarrow d$.
4. Entferne v_q und seine Kanten von G.
5. Wenn $V \neq \emptyset$, dann $d \leftarrow d + 1$, gehe zu Schritt 1.

Laufzeit im schlechtesten Fall: $\Theta(|V|^2)$.

Verbesserung

Idee?

Verbesserung

Idee?

Berechne die Eingangsgrade der Knoten im Voraus und durchlaufe dann jeweils die Knoten mit Eingangsgrad 0 die Eingangsgrade der Nachfolgeknoten korrigierend.

Algorithmus Topological-Sort(G)

Input: Graph $G = (V, E)$.
Output: Topologische Sortierung ord

Stack $S \leftarrow \emptyset$
foreach $v \in V$ **do** $A[v] \leftarrow 0$
foreach $(v, w) \in E$ **do** $A[w] \leftarrow A[w] + 1$ // Eingangsgrade berechnen
foreach $v \in V$ with $A[v] = 0$ **do** push(S, v) // Merke Nodes mit Eingangsgrad 0
$i \leftarrow 1$
while $S \neq \emptyset$ **do**
    $v \leftarrow$ pop(S); $\mathrm{ord}[v] \leftarrow i$; $i \leftarrow i + 1$ // Wähle Knoten mit Eingangsgrad 0
    foreach $(v, w) \in E$ **do** // Verringere Eingangsgrad der Nachfolger
        $A[w] \leftarrow A[w] - 1$
        if $A[w] = 0$ **then** push(S, w)

if $i = |V| + 1$ **then return** ord **else return** “Cycle Detected”

Algorithmus Korrektheit

Theorem 8

Sei $G = (V, E)$ ein gerichteter, kreisfreier Graph. Der Algorithmus `TopologicalSort`(G) *berechnet in Zeit* $\Theta(|V| + |E|)$ *eine topologische Sortierung* ord *für* G.

Algorithmus Korrektheit

Theorem 8

Sei $G = (V, E)$ ein gerichteter, kreisfreier Graph. Der Algorithmus **`TopologicalSort`**(G) *berechnet in Zeit $\Theta(|V| + |E|)$ eine topologische Sortierung* ord *für G.*

Beweis: folgt im wesentlichen aus vorigem Theorem:

1. Eingangsgrad verringern entspricht Knotenentfernen.
2. Im Algorithmus gilt für jeden Knoten v mit $A[v] = 0$ dass entweder der Knoten Eingangsgrad 0 hat oder dass zuvor alle Vorgänger einen Wert $\mathrm{ord}[u] \leftarrow i$ zugewiesen bekamen und somit $\mathrm{ord}[v] > \mathrm{ord}[u]$ für alle Vorgänger u von v. Knoten werden nur einmal auf den Stack gelegt.
3. Laufzeit: Inspektion des Algorithmus (mit Argumenten wie beim Traversieren).

Algorithmus Korrektheit

Theorem 9

Sei $G = (V, E)$ ein gerichteter, nicht kreisfreier Graph. Der Algorithmus `TopologicalSort`(G) *terminiert in Zeit* $\Theta(|V| + |E|)$ *und detektiert Zyklus.*

Algorithmus Korrektheit

Theorem 9

Sei $G = (V, E)$ ein gerichteter, nicht kreisfreier Graph. Der Algorithmus `TopologicalSort`(G) *terminiert in Zeit* $\Theta(|V|+|E|)$ *und detektiert Zyklus.*

Beweis: Sei $\langle v_{i_1}, \dots, v_{i_k} \rangle$ ein Kreis in G. In jedem Schritt des Algorithmus bleibt $A[v_{i_j}] \geq 1$ für alle $j = 1, \dots, k$. Also werden k Knoten nie auf den Stack gelegt und somit ist zum Schluss $i \leq V + 1 - k$.

Die Laufzeit des zweiten Teils des Algorithmus kann kürzer werden, jedoch kostet die Berechnung der Eingangsgrade bereits $\Theta(|V| + |E|)$.