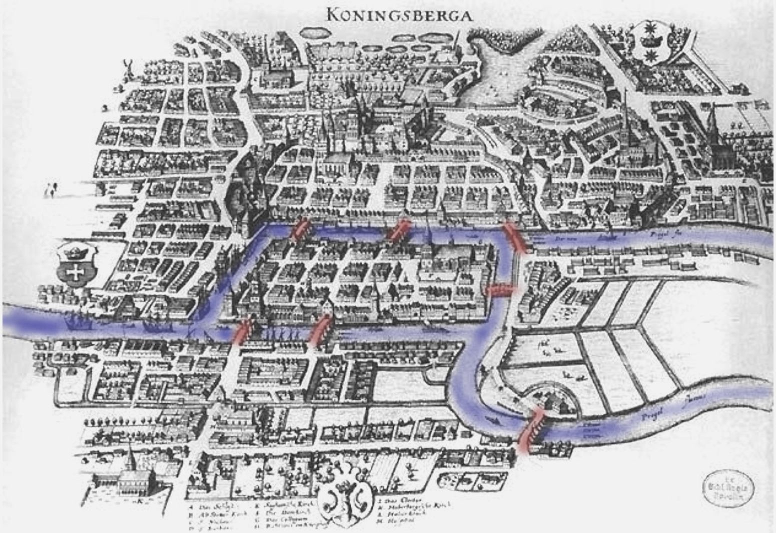
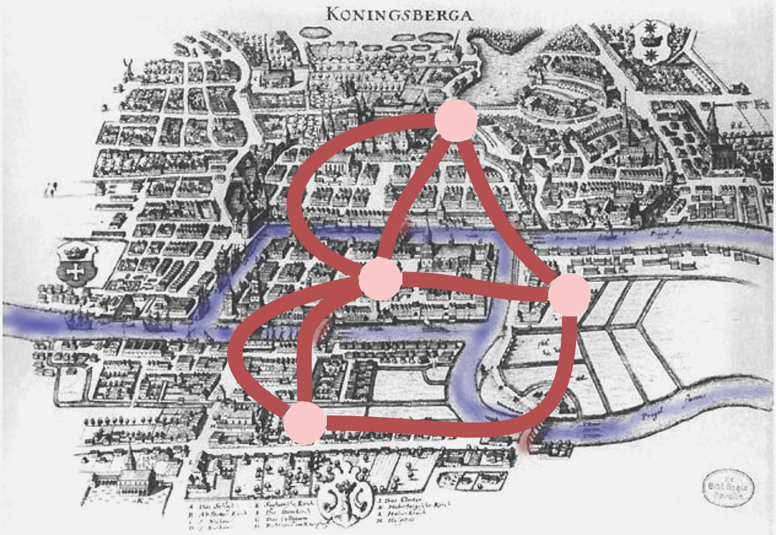
22. Graphen

Reflexive transitive Hülle, Traversieren (DFS, BFS), Zusammenhangskomponenten, Topologisches Sortieren
Ottman/Widmayer, Kap. 9.1 - 9.4,Cormen et al, Kap. 22

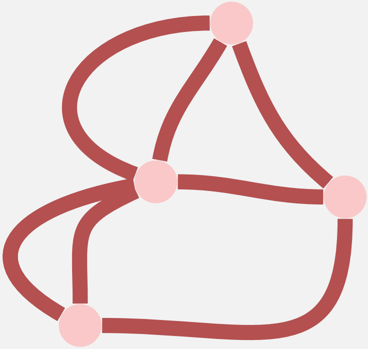
Königsberg 1736



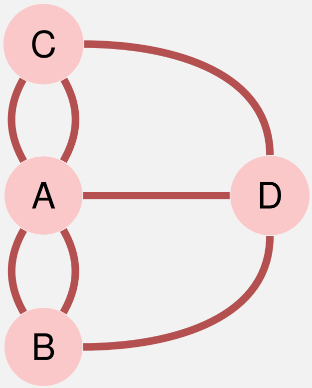
Königsberg 1736



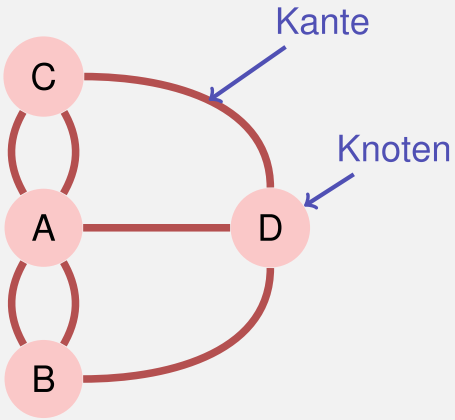
Königsberg 1736



Graph

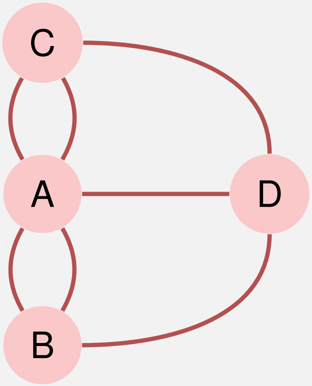


Graph



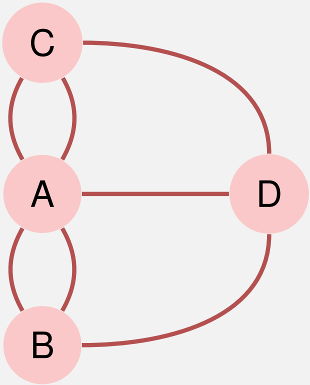
Zyklen

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



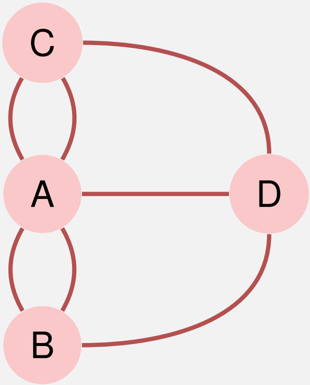
Zyklen

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



Zyklen

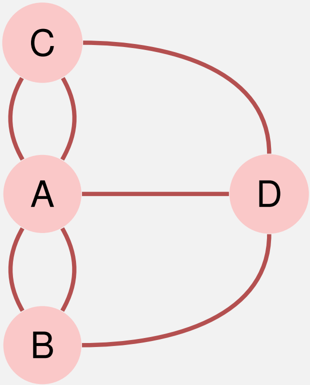
- Gibt es einen Rundweg durch die Stadt (den Graphen), welcher jede Brücke (jede Kante) genau einmal benutzt?
- Euler (1736): nein.
- Solcher Rundweg (*Zyklus*) heisst *Eulerscher Kreis*.



Zyklen

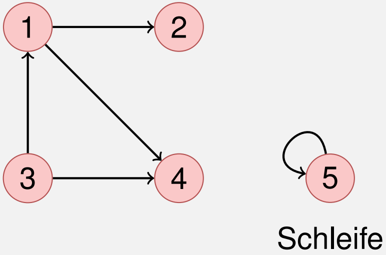
- Gibt es einen Rundweg durch die Stadt (den Graphen), welcher jede Brücke (jede Kante) genau einmal benutzt?
- Euler (1736): nein.
- Solcher Rundweg (*Zyklus*) heisst *Eulerscher Kreis*.
- Eulerzyklus $\Leftrightarrow$ jeder Knoten hat gerade Anzahl Kanten (jeder Knoten hat einen *geraden Grad*).

  "$\Rightarrow$" ist klar, "$\Leftarrow$" ist etwas schwieriger



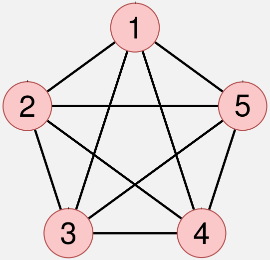
Notation

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 *ungerichteter 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.[31]



ein vollständiger ungerichter Graph

[31] Im Gegensatz zum Eingangsbeispiel – andernfalls Multigraph genannt.

Notation

Ein Graph $G = (V, E)$ in dem E jede mögliche Kante enthält heisst *vollständig*.

Notation

Ein Graph $G = (V, E)$ in dem E jede mögliche Kante enthält heisst *vollständig*.

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 Graph $G = (V, E)$ in dem E jede mögliche Kante enthält heisst *vollständig*.

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*.

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

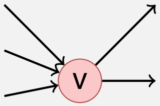
Für gerichtete Graphen $G = (V, E)$

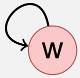
- $w \in V$ heisst *adjazent* zu $v \in V$, falls $(v, w) \in E$
- *Vorgängermenge* von $v \in V$: $N^-(v) := \{u \in V | (u, v) \in E\}$.
  Nachfolgermenge: $N^+(v) := \{u \in V | (v, u) \in E\}$

Notation

Für gerichtete Graphen $G = (V, E)$

- $w \in V$ 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: $N^+(v) := \{u \in V | (v, u) \in E\}$
- *Eingangsgrad*: $\deg^-(v) = |N^-(v)|$,
  Ausgangsgrad: $\deg^+(v) = |N^+(v)|$





$\deg^-(v) = 3, \deg^+(v) = 2$ $\quad$ $\deg^-(w) = 1, \deg^+(w) = 1$

Notation

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

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

Notation

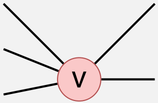
Für ungerichtete Graphen $G = (V, E)$:

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

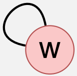
Notation

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 $\langle v_1, \ldots, v_{k+1} \rangle$ 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.

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.
- *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*: Weg der keinen Knoten mehrfach verwendet.

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*: Weg der keinen Knoten mehrfach verwendet.
- *Zusammenhängend*: Ungerichteter Graph, bei dem für jedes Paar $v, w \in V$ ein verbindender Weg existiert.

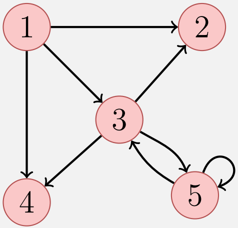
Zyklen

- *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 $v_1, \ldots, v_n$ gespeichert als *Adjazenzmatrix* $A_G = (a_{ij})_{1 \leq i,j \leq n}$ mit Einträgen aus $\{0, 1\}$. $a_{ij} = 1$ genau dann wenn Kante von v_i nach v_j.

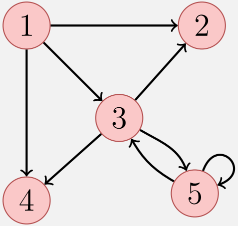


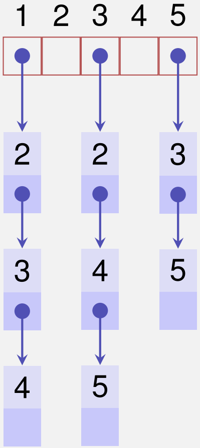
$$\begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \end{pmatrix}$$

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 von $v \in V$ finden		
$v \in V$ ohne Nachbar finden		
$(u, v) \in E$?		
Kante einfügen		
Kante löschen		

Laufzeiten einfacher Operationen

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

Laufzeiten einfacher Operationen

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

Laufzeiten einfacher Operationen

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

Laufzeiten einfacher Operationen

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

Laufzeiten einfacher Operationen

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

Laufzeiten einfacher Operationen

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

Laufzeiten einfacher Operationen

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

Laufzeiten einfacher Operationen

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

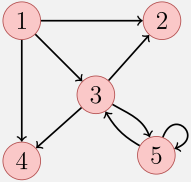
Laufzeiten einfacher Operationen

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

Laufzeiten einfacher Operationen

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

Adjazenzmatrizen multipliziert



$$B := A_G^2 = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \end{pmatrix}^2 = \begin{pmatrix} 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 2 \end{pmatrix}$$

Interpretation

Theorem

Sei $G = (V, E)$ *ein Graph und* $k \in \mathbb{N}$*. Dann gibt das Element* $a_{i,j}^{(k)}$ *der Matrix* $(a_{i,j}^{(k)})_{1 \leq i,j \leq n} = A_G^k$ *die Anzahl der Wege mit Länge* k *von* v_i *nach* v_j *an.*

Beweis

Per Induktion.
Anfang: Klar für $k = 1$. $a_{i,j} = a_{i,j}^{(1)}$.
Hypothese: Aussage wahr für alle $k \leq l$
Schritt ($l \rightarrow l+1$):

$$a_{i,j}^{(l+1)} = \sum_{k=1}^{n} a_{i,k}^{(l)} \cdot a_{k,j}$$

$a_{k,j} = 1$ g.d.w. Kante von k nach j, 0 sonst. Obige Summe zählt somit alle Knoten, die direkte Verbindung haben zu v_j und zu denen Weg der Länge l von v_i existiert, d.h. alle Wege der Länge $l+1$.

Kürzester Weg

Frage: existiert Weg von i nach j? Wie lang ist der kürzeste Weg?

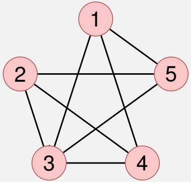
Kürzester Weg

Frage: existiert Weg von i nach j? Wie lang ist der kürzeste Weg?

Antwort: Potenziere A_G bis für ein $k < n$ gilt $a_{i,j}^{(k)} > 0$. k gibt die Weglänge des kürzesten Weges. Wenn $a_{i,j}^{(k)} = 0$ für alle $1 \leq k < n$, so gibt es keinen Weg von i nach j.

Anzahl Dreiecke

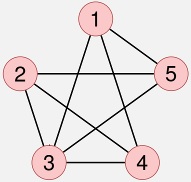
Frage: Wie viele Dreieckswege enthält ein ungerichteter Graph?



Anzahl Dreiecke

Frage: Wie viele Dreieckswege enthält ein ungerichteter Graph?

Antwort: Entferne alle Zyklen (Diagonaleinträge). Berechne A_G^3. $a_{ii}^{(3)}$ bestimmt die Anzahl Wege der Länge 3, die i enthalten.

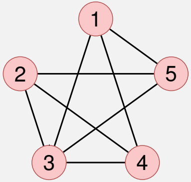


$$\begin{pmatrix} 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \end{pmatrix}^3 = \begin{pmatrix} 4 & 4 & 8 & 8 & 8 \\ 4 & 4 & 8 & 8 & 8 \\ 8 & 8 & 8 & 8 & 8 \\ 8 & 8 & 8 & 4 & 4 \\ 8 & 8 & 8 & 4 & 4 \end{pmatrix}$$

Anzahl Dreiecke

Frage: Wie viele Dreieckswege enthält ein ungerichteter Graph?

Antwort: Entferne alle Zyklen (Diagonaleinträge). Berechne A_G^3. $a_{ii}^{(3)}$ bestimmt die Anzahl Wege der Länge 3, die i enthalten. Es gibt 6 verschiedene Permutationen eines Dreicksweges. Damit Anzahl Dreiecke: $\sum_{i=1}^{n} a_{ii}^{(3)}/6$.



$$\begin{pmatrix} 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \end{pmatrix}^3 = \begin{pmatrix} 4 & 4 & 8 & 8 & 8 \\ 4 & 4 & 8 & 8 & 8 \\ 8 & 8 & 8 & 8 & 8 \\ 8 & 8 & 8 & 4 & 4 \\ 8 & 8 & 8 & 4 & 4 \end{pmatrix}$$

$\Rightarrow 24/6 = 4$ Dreiecke.

Graphen und Relationen

Graph $G = (V, E)$ mit Adjazenzen $A_G \mathrel{\widehat{=}}$ Relation $E \subseteq V \times V$ auf V

Graphen und Relationen

Graph $G = (V, E)$ mit Adjazenzen $A_G \mathrel{\widehat{=}}$ Relation $E \subseteq V \times V$ auf V

- *reflexiv* $\Leftrightarrow a_{i,i} = 1$ für alle $i = 1, \ldots, n$.
- *symmetrisch* $\Leftrightarrow a_{i,j} = a_{j,i}$ für alle $i, j = 1, \ldots, n$ (ungerichtet)
- *transitiv* $\Leftrightarrow (u, v) \in E$, $(v, w) \in E \Rightarrow (u, w) \in E$.

Graphen und Relationen

Graph $G = (V, E)$ mit Adjazenzen $A_G \mathrel{\widehat{=}}$ Relation $E \subseteq V \times V$ auf V

- *reflexiv* $\Leftrightarrow a_{i,i} = 1$ für alle $i = 1, \ldots, n$.
- *symmetrisch* $\Leftrightarrow a_{i,j} = a_{j,i}$ für alle $i, j = 1, \ldots, n$ (ungerichtet)
- *transitiv* $\Leftrightarrow (u, v) \in E$, $(v, w) \in E \Rightarrow (u, w) \in E$.

Äquivalenzrelation $\Leftrightarrow$ Kollektion vollständiger, ungerichteter Graphen, für den jedes Element eine Schleife hat.

Graphen und Relationen

Graph $G = (V, E)$ mit Adjazenzen $A_G \mathrel{\widehat{=}}$ Relation $E \subseteq V \times V$ auf V

- *reflexiv* $\Leftrightarrow a_{i,i} = 1$ für alle $i = 1, \ldots, n$.
- *symmetrisch* $\Leftrightarrow a_{i,j} = a_{j,i}$ für alle $i, j = 1, \ldots, n$ (ungerichtet)
- *transitiv* $\Leftrightarrow (u, v) \in E$, $(v, w) \in E \Rightarrow (u, w) \in E$.

Äquivalenzrelation $\Leftrightarrow$ Kollektion vollständiger, ungerichteter Graphen, für den jedes Element eine Schleife hat.

Reflexive transitive Hülle von G $\Leftrightarrow$ *Erreichbarkeitsrelation* E^*: $(v, w) \in E^*$ gdw. $\exists$ Weg von Knoten v zu w.

Berechnung Reflexive Transitive Hülle

Ziel: Berechnung von $B = (b_{ij})_{1 \leq i,j \leq n}$ mit $b_{ij} = 1 \Leftrightarrow (v_i, v_j) \in E^*$

Beobachtung: $a_{ij} = 1$ bedeutet bereits $(v_i, v_j) \in E^*$.

Berechnung Reflexive Transitive Hülle

Ziel: Berechnung von $B = (b_{ij})_{1 \leq i,j \leq n}$ mit $b_{ij} = 1 \Leftrightarrow (v_i, v_j) \in E^*$

Beobachtung: $a_{ij} = 1$ bedeutet bereits $(v_i, v_j) \in E^*$.

Erste Idee:

- Starte mit $B \leftarrow A$ und setze $b_{ii} = 1$ für alle i (Reflexivität).
- Iteriere über i, j, k und setze $b_{ij} = 1$, wenn $b_{ik} = 1$ und $b_{kj} = 1$. Dann alle Wege der Länge 1 und 2 berücksichtigt
- Wiederhole Iteration $\Rightarrow$ alle Wege der Länge $1 \ldots 4$ berücksichtigt.
- $\lceil \log_2 n \rceil$ Wiederholungen nötig.

Verbesserung: Algorithmus von Warshall (1962)

Induktiver Ansatz: Alle Wege bekannt über Knoten aus $\{v_i : i < k\}$.
Hinzunahme des Knotens v_k.

Algorithmus ReflexiveTransitiveClosure(A_G)

Input : Adjazenzmatrix $A_G = (a_{ij})_{i,j=1}^n$
Output : Reflexive Transitive Hülle $B = (b_{ij})_{i,j=1}^n$ von G

$B \leftarrow A_G$
for $k \leftarrow 1$ **to** n **do**
    $a_{kk} \leftarrow 1$ // Reflexivität
    for $i \leftarrow 1$ **to** n **do**
        for $j \leftarrow 1$ **to** n **do**
            $b_{ij} \leftarrow \max\{b_{ij}, b_{ik} \cdot b_{kj}\}$ // Alle Wege über v_k

return B

Laufzeit des Algorithmus $\Theta(n^3)$.

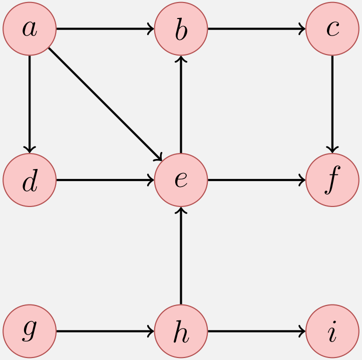
Korrektheit des Algorithmus (Induktion)

Invariante (k): alle Wege über Knoten mit maximalem Index $< k$ berücksichtigt

- Anfang ($k = 1$): Alle direkten Wege (alle Kanten) in A_G berücksichtigt.
- Hypothese: Invariante (k) erfüllt.
- Schritt ($k \to k + 1$): Für jeden Weg von v_i nach v_j über Knoten mit maximalen Index k: nach Hypothese $b_{ik} = 1$ und $b_{kj} = 1$. Somit im k-ten Schleifendurchlauf: $b_{ij} \leftarrow 1$.

Graphen Traversieren: Tiefensuche

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

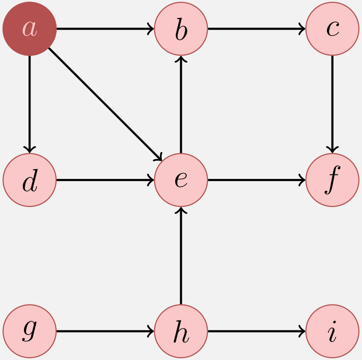


Adjazenzliste

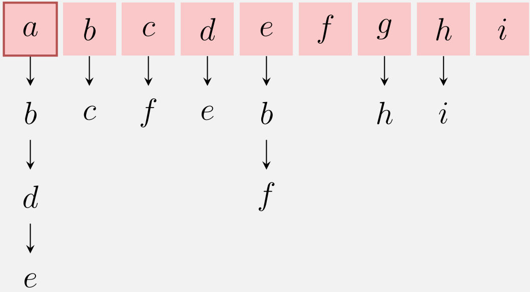
a	b	c	d	e	f	g	h	i
$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$		$\downarrow$	$\downarrow$	
b	c	f	e	b		h	i	
$\downarrow$				$\downarrow$				
d				f				
$\downarrow$								
e								

Graphen Traversieren: Tiefensuche

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

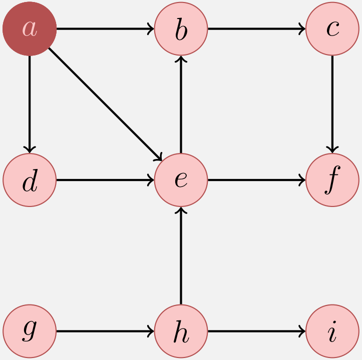


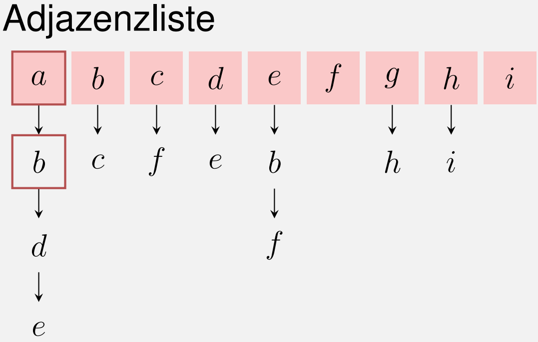
Adjazenzliste



Graphen Traversieren: Tiefensuche

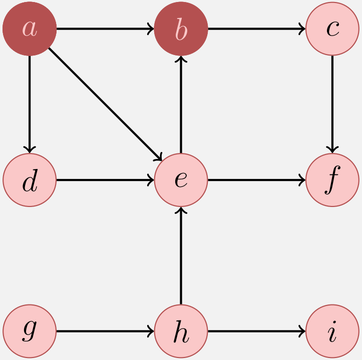
Verfolge zuerst Pfad in die Tiefe, bis nichts mehr besucht werden kann.



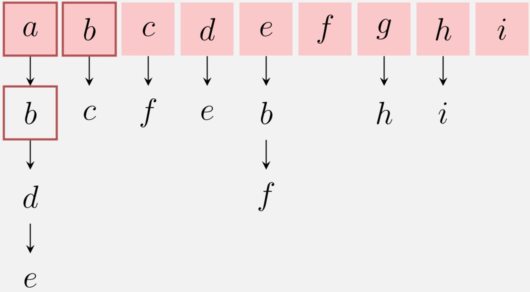


Graphen Traversieren: Tiefensuche

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

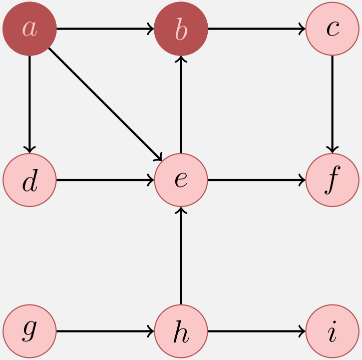


Adjazenzliste

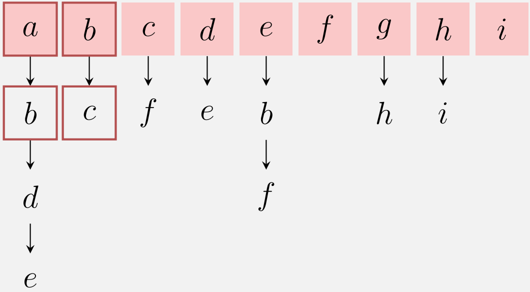


Graphen Traversieren: Tiefensuche

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

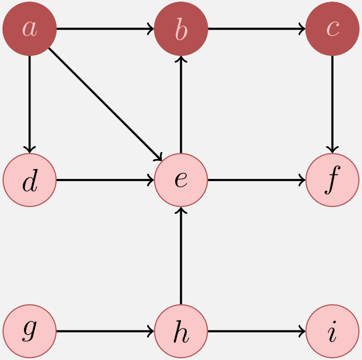


Adjazenzliste

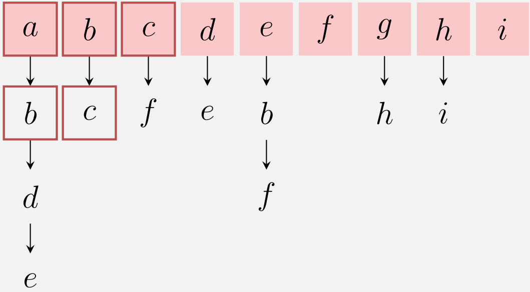


Graphen Traversieren: Tiefensuche

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

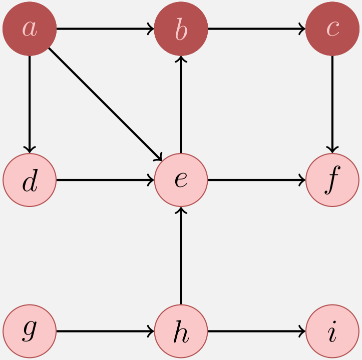


Adjazenzliste

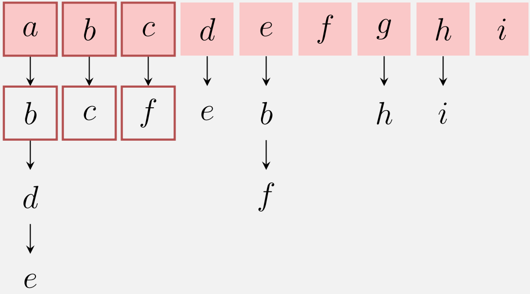


Graphen Traversieren: Tiefensuche

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

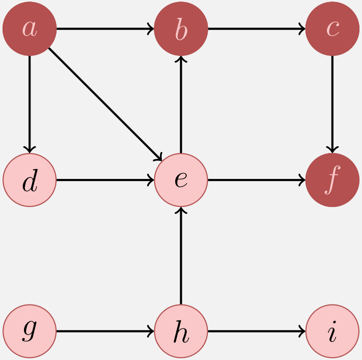


Adjazenzliste

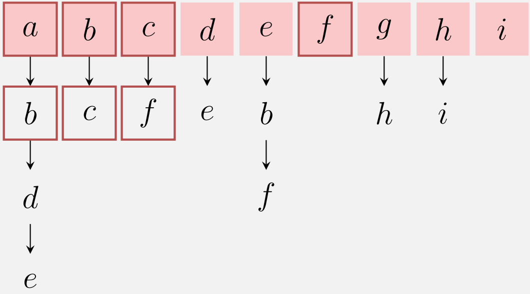


Graphen Traversieren: Tiefensuche

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

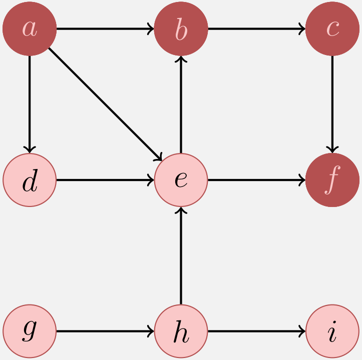


Adjazenzliste

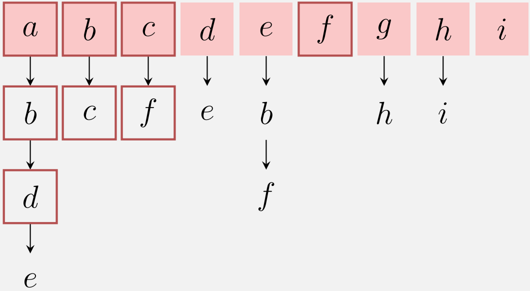


Graphen Traversieren: Tiefensuche

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

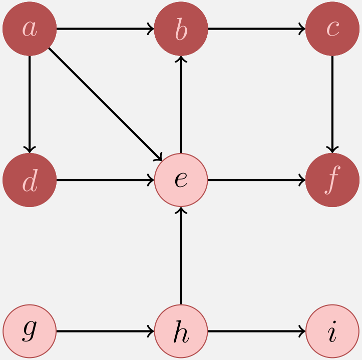


Adjazenzliste

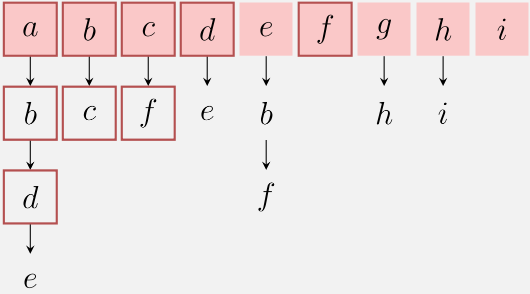


Graphen Traversieren: Tiefensuche

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

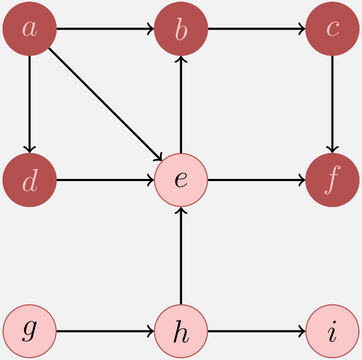


Adjazenzliste

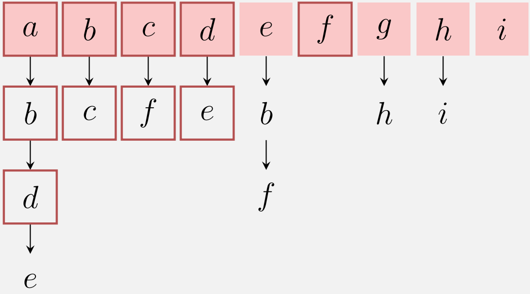


Graphen Traversieren: Tiefensuche

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

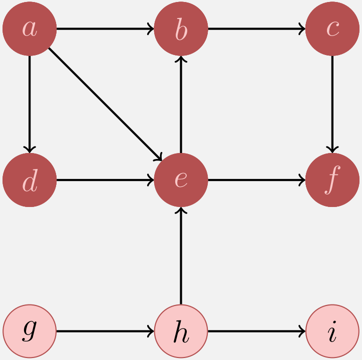


Adjazenzliste

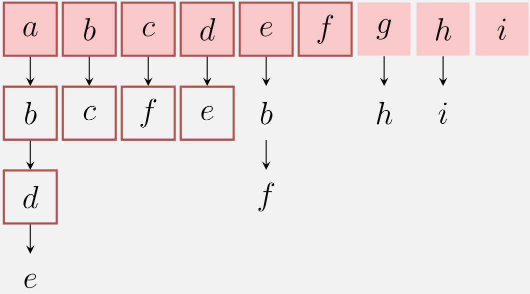


Graphen Traversieren: Tiefensuche

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

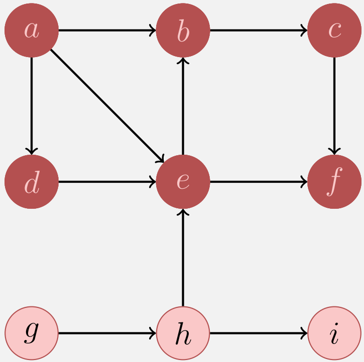


Adjazenzliste

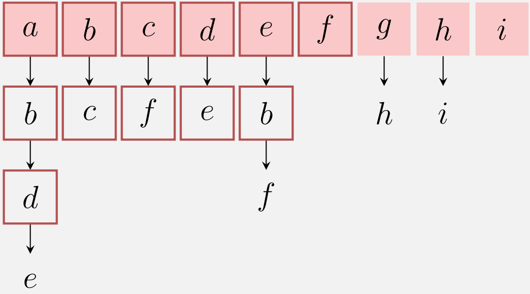


Graphen Traversieren: Tiefensuche

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

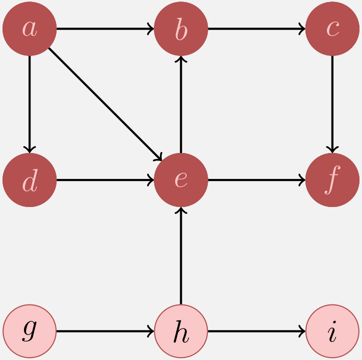


Adjazenzliste

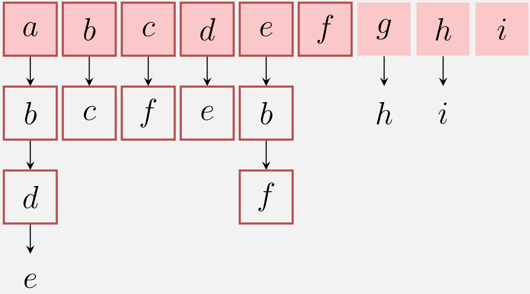


Graphen Traversieren: Tiefensuche

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

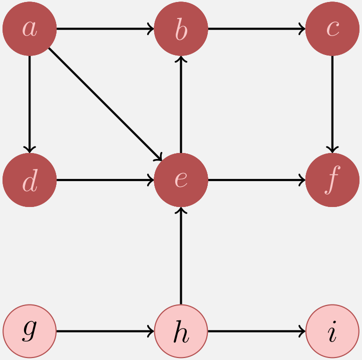


Adjazenzliste

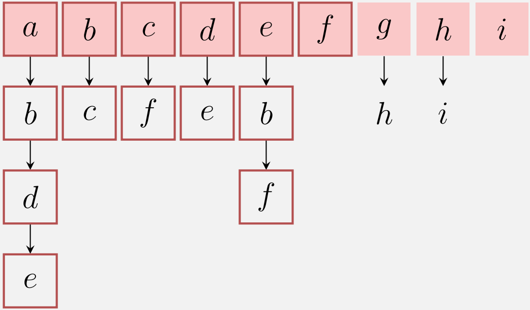


Graphen Traversieren: Tiefensuche

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

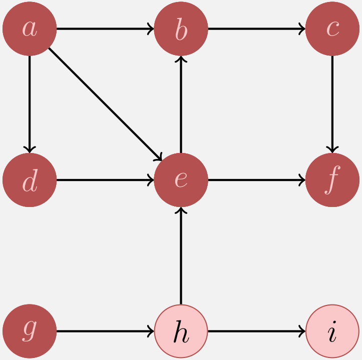


Adjazenzliste

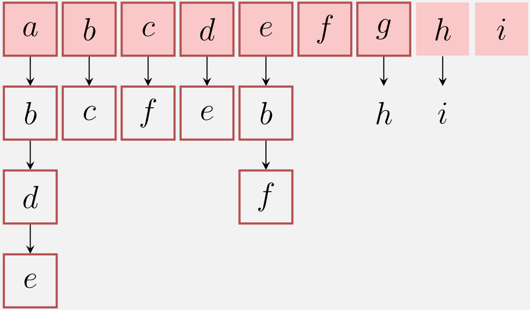


Graphen Traversieren: Tiefensuche

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

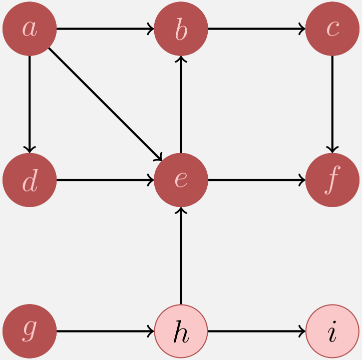


Adjazenzliste

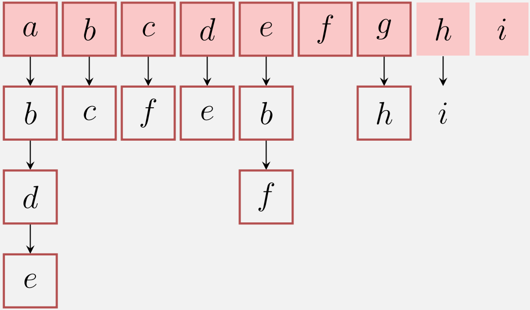


Graphen Traversieren: Tiefensuche

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

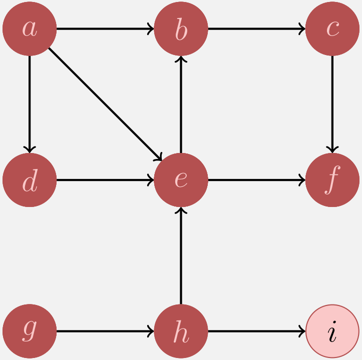


Adjazenzliste

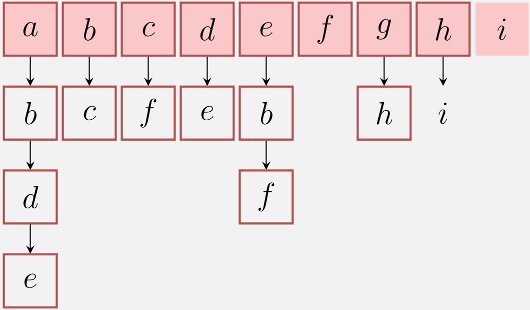


Graphen Traversieren: Tiefensuche

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

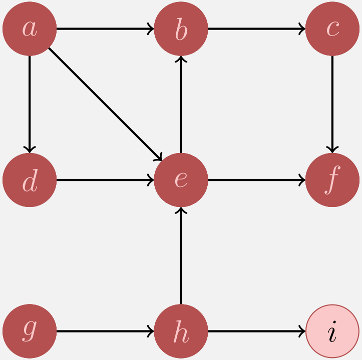


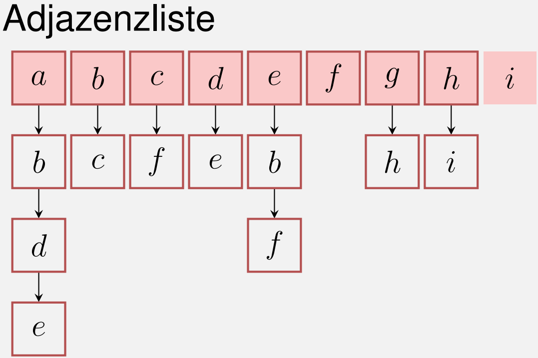
Adjazenzliste



Graphen Traversieren: Tiefensuche

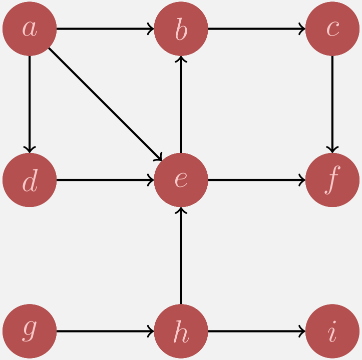
Verfolge zuerst Pfad in die Tiefe, bis nichts mehr besucht werden kann.





Graphen Traversieren: Tiefensuche

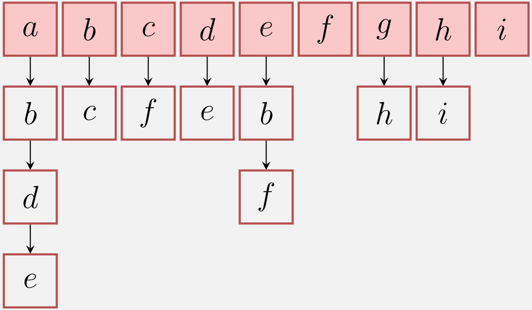
Verfolge zuerst Pfad in die Tiefe, bis nichts mehr besucht werden kann.



Reihenfolge

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

Adjazenzliste



Algorithmus Tiefensuche DFS-Visit(G, v)

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

Markiere v als besucht
foreach $(v, w) \in E$ **do**
    if $\neg$(w besucht) **then**
        DFS-Visit(w)

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**
  if $\neg(v$ besucht$)$ **then**
    DFS-Visit(G,v)

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

Problem der Rekursion?

Algorithmus Tiefensuche DFS-Visit(G)

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

foreach $v \in V$ **do**
  if $\neg(v$ besucht$)$ **then**
    DFS-Visit(G,v)

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

Problem der Rekursion?

Sehr tiefe Graphen können Stack-Überlauf erzeugen.

Iteratives DFS-Visit(G, v)

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

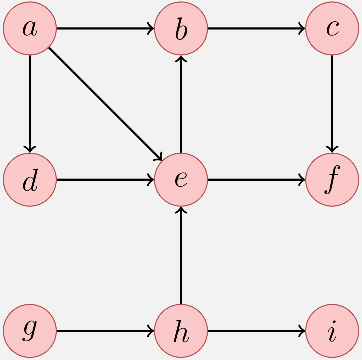
Stack $S \leftarrow \emptyset$; push(S, v)
while $S \neq \emptyset$ **do**
    $w \leftarrow$ pop(S)
    if $\neg(w$ besucht$)$ **then**
        Markiere w besucht
        foreach $(w, c) \in E$ **do** // (ggfs umgekehrt einfügen)
            if $\neg(c$ besucht$)$ **then**
                push(S, x)

Stapelgrösse bis zu $|E|$, für jeden Knoten maximal Extraaufwand $\Theta(\deg^+(w) + 1)$. Gesamt: $\mathcal{O}(|V| + |E|)$

Mit Aufruf aus obigem Rahmenprogramm: $\Theta(|V| + |E|)$

Graphen Traversieren: Breitensuche

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

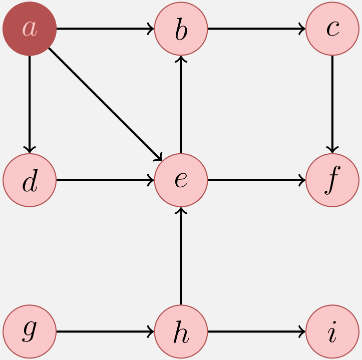


Adjazenzliste

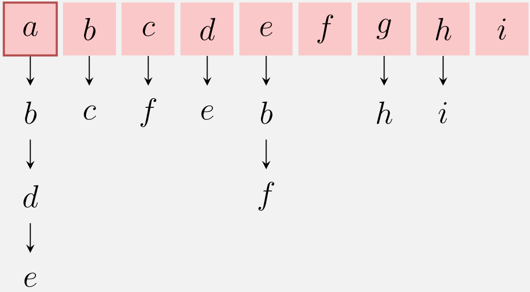
- $a \rightarrow b \rightarrow d \rightarrow e$
- $b \rightarrow c$
- $c \rightarrow f$
- $d \rightarrow e$
- $e \rightarrow b \rightarrow f$
- f
- $g \rightarrow h$
- $h \rightarrow i$
- i

Graphen Traversieren: Breitensuche

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

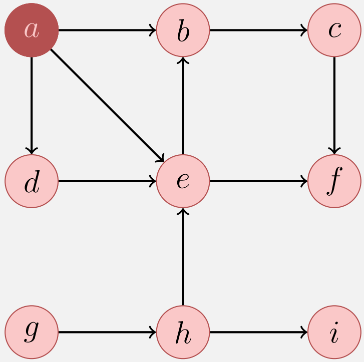


Adjazenzliste

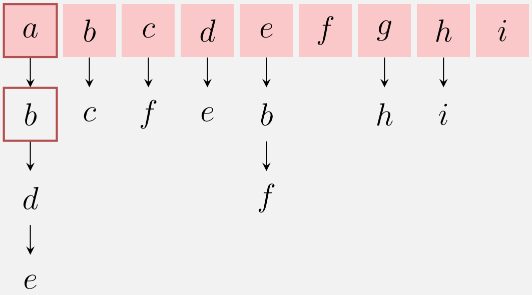


Graphen Traversieren: Breitensuche

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

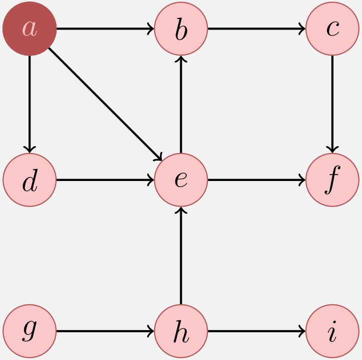


Adjazenzliste

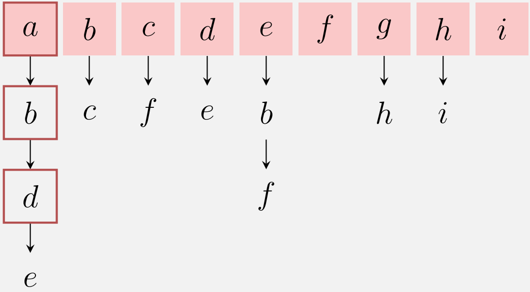


Graphen Traversieren: Breitensuche

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

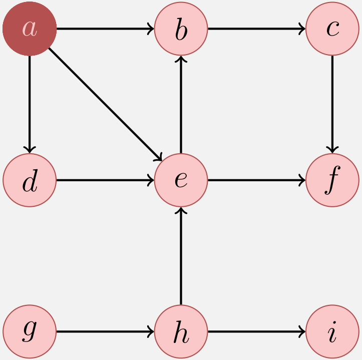


Adjazenzliste

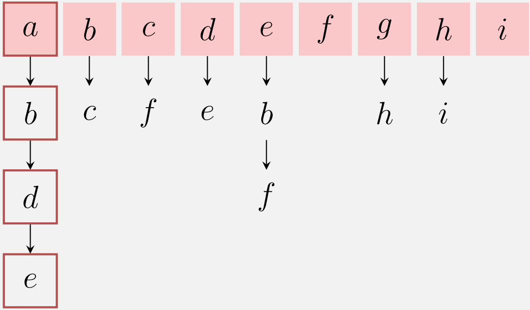


Graphen Traversieren: Breitensuche

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

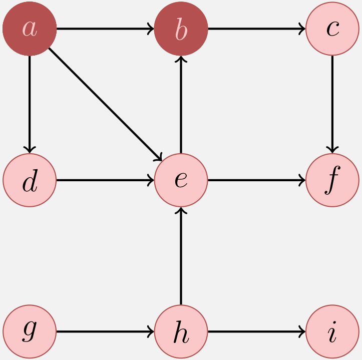


Adjazenzliste

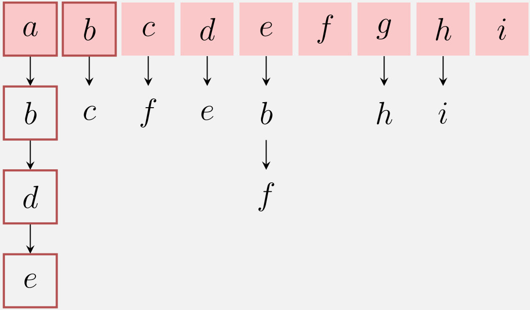


Graphen Traversieren: Breitensuche

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

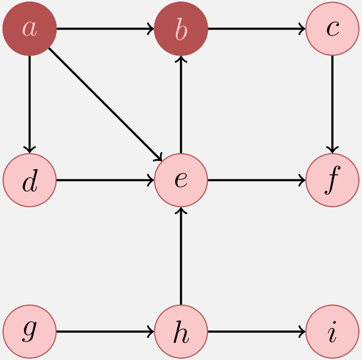


Adjazenzliste

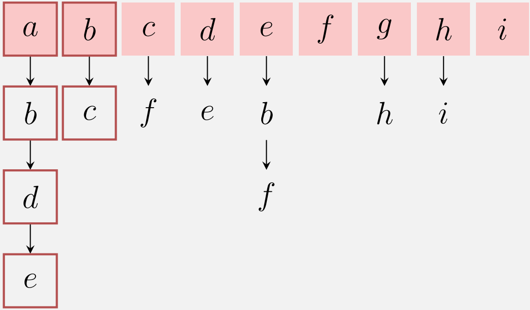


Graphen Traversieren: Breitensuche

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

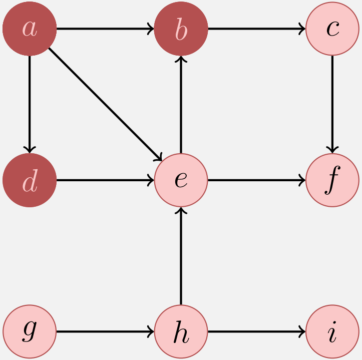


Adjazenzliste

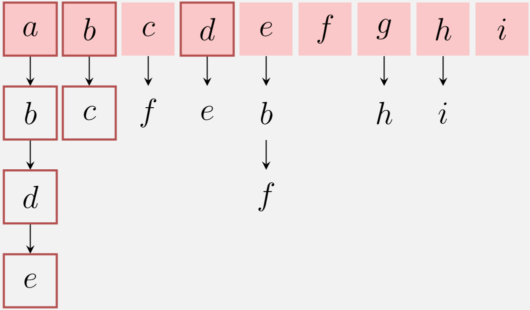


Graphen Traversieren: Breitensuche

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

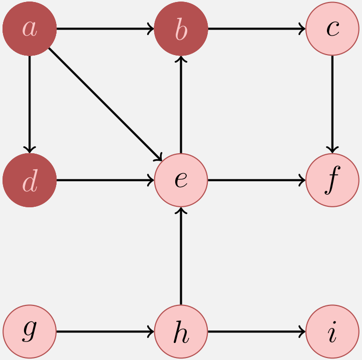


Adjazenzliste

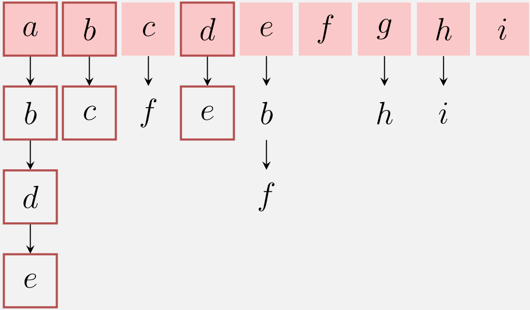


Graphen Traversieren: Breitensuche

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

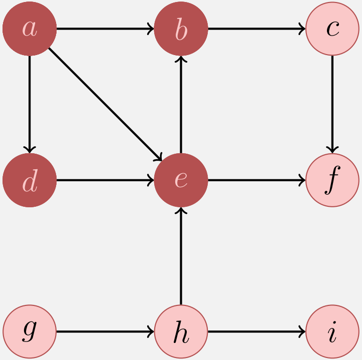


Adjazenzliste

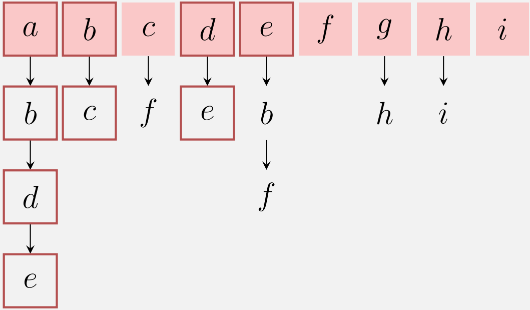


Graphen Traversieren: Breitensuche

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

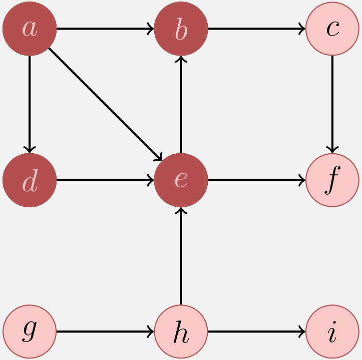


Adjazenzliste

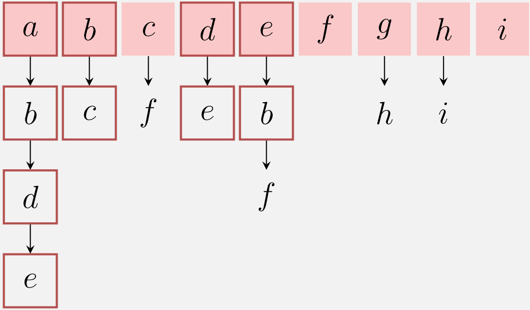


Graphen Traversieren: Breitensuche

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

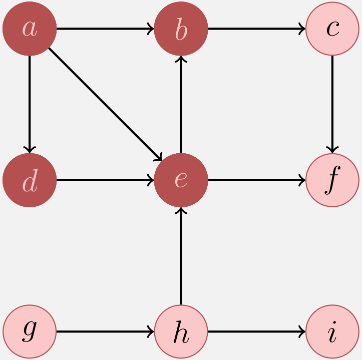


Adjazenzliste

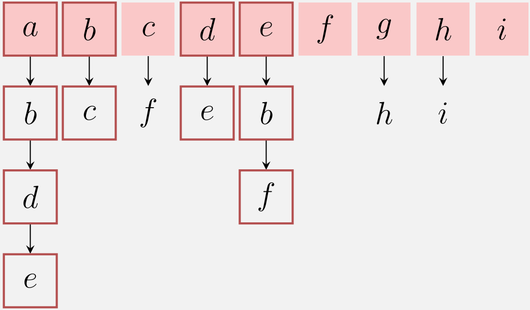


Graphen Traversieren: Breitensuche

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

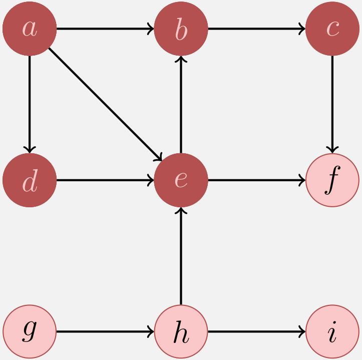


Adjazenzliste

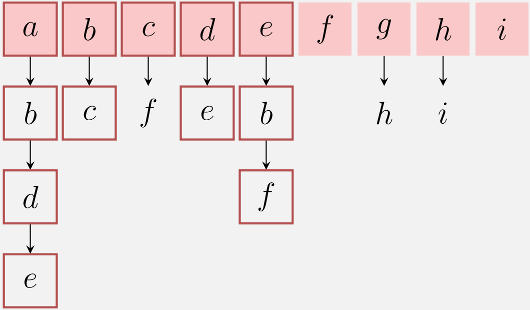


Graphen Traversieren: Breitensuche

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

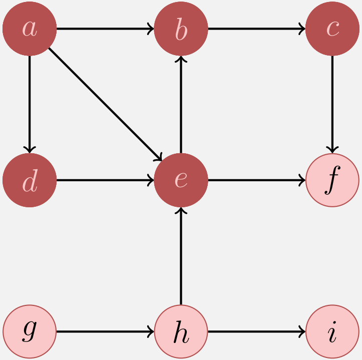


Adjazenzliste

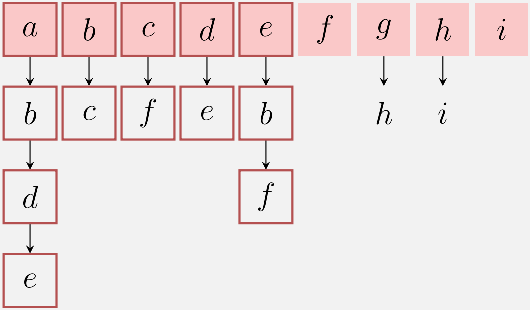


Graphen Traversieren: Breitensuche

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

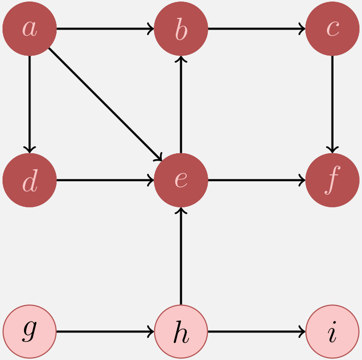


Adjazenzliste

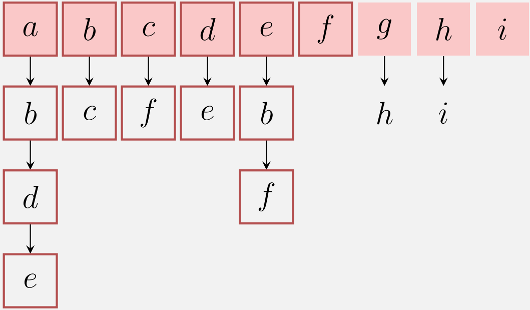


Graphen Traversieren: Breitensuche

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

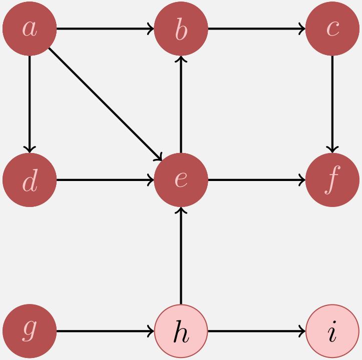


Adjazenzliste

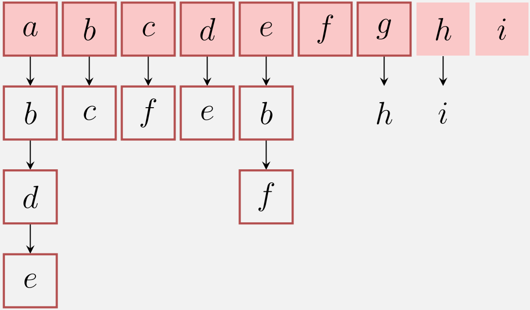


Graphen Traversieren: Breitensuche

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

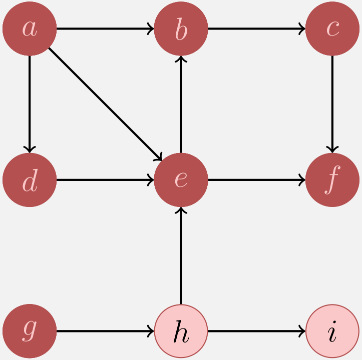


Adjazenzliste

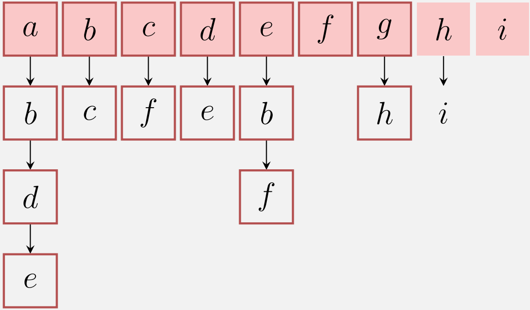


Graphen Traversieren: Breitensuche

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

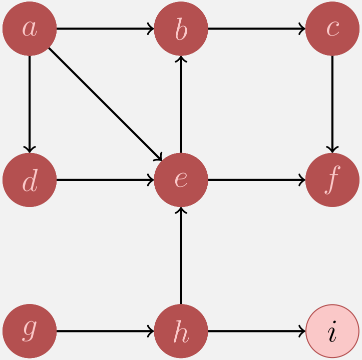


Adjazenzliste

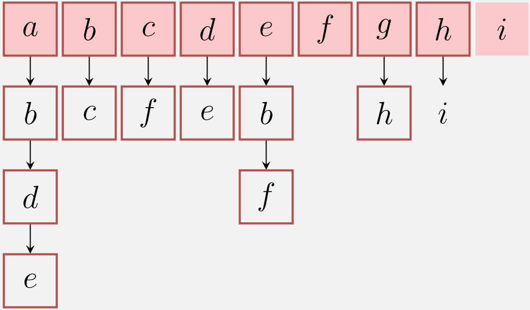


Graphen Traversieren: Breitensuche

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

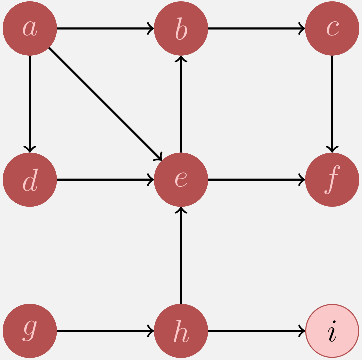


Adjazenzliste

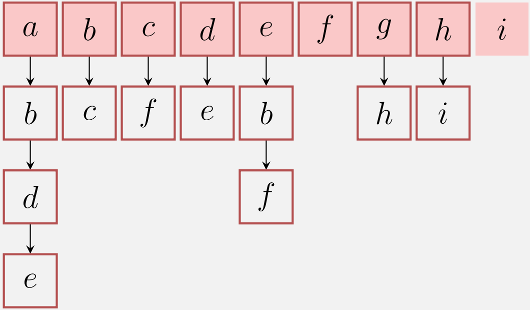


Graphen Traversieren: Breitensuche

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

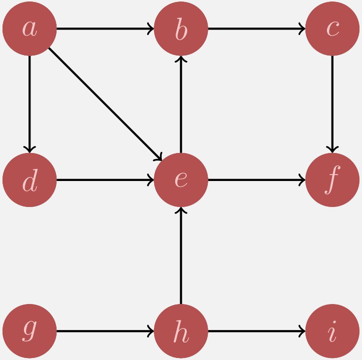


Adjazenzliste



Graphen Traversieren: Breitensuche

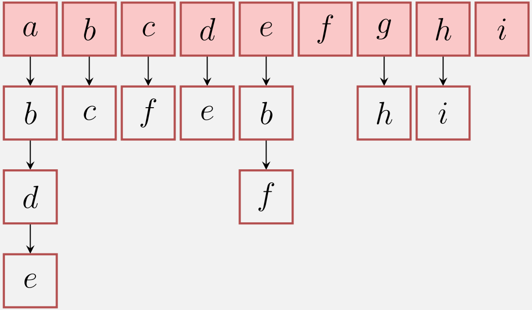
Verfolge zuerst Pfad in die Breite, gehe dann in die Tiefe.



Reihenfolge

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

Adjazenzliste



Iteratives BFS-Visit(G, v)

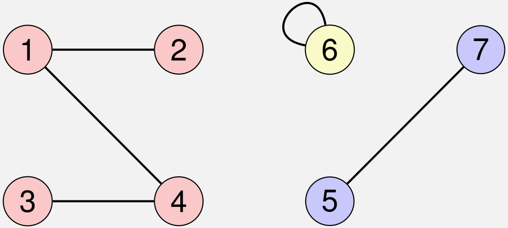
Input : Graph $G = (V, E)$

Queue $Q \leftarrow \emptyset$
Markiere v aktiv
enqueue(Q, v)
while $Q \neq \emptyset$ **do**
    $w \leftarrow$ dequeue(Q)
    Markiere w besucht
    foreach $(w, c) \in E$ **do**
        if $\neg$(c besucht $\vee$ c aktiv) **then**
            Markiere c aktiv
            enqueue(Q, c)

- Algorithmus kommt mit $\mathcal{O}(|V|)$ Extraplatz aus.(Warum funktioniert dieser simple Trick nicht beim DFS?)
- Gesamtlaufzeit mit Rahmenprogramm: $\Theta(|V| + |E|)$.

Zusammenhangskomponenten

Zusammenhangskomponenten eines ungerichteten Graphen G:
Äquivalenzklassen der reflexiven, transitiven Hülle von G.
Zusammenhangskomponente = Teilgraph $G' = (V', E')$,
$E' = \{\{v, w\} \in E | v, w \in V'\}$ mit
$\{\{v, w\} \in E | v \in V' \vee w \in V'\} = E = \{\{v, w\} \in E | v \in V' \wedge w \in V'\}$



Graph mit Zusammenhangskomponenten $\{1, 2, 3, 4\}$, $\{5, 7\}$, $\{6\}$.

Berechnung der Zusammenhangskomponenten

- Berechnung einer Partitionierung von V in paarweise disjnkte Teilmengen $V_1, \ldots, V_k$
- so dass jedes V_i die Knoten einer Zusammenhangskomponente enhält.
- Algorithmus: Tiefen- oder Breitensuche. Bei jedem Neustart von DFSSearch(G, v) oder BFSSearch(G, v) neue leere Zusammenhangskomponte erstellen und alle traversierten Knoten einfügen.

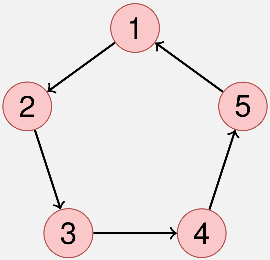
Topologische Sortierung

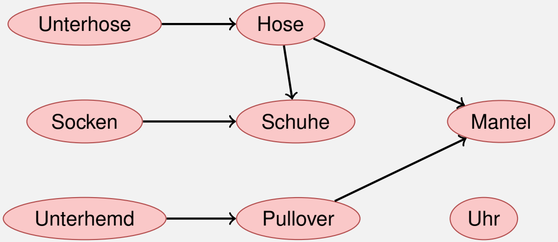
Topologische Sortierung eines azyklischen gerichteten Graphen $G = (V, E)$: Bijektive Abbildung

$$\mathrm{ord} : V \to \{1, \ldots, |V|\} \quad | \quad \mathrm{ord}(v) < \mathrm{ord}(w) \; \forall \; (v, w) \in E.$$

Können Wert i auch identifizieren mit v_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,Schuhe

Beobachtung

Theorem

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

Beobachtung

Theorem

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 $\mathrm{ord}(v_1) = 1$.
- 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 $\mathrm{ord}(v_q) \leftarrow d$.
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:

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: $\Omega(|V|^2)$.

Verbeserung

Idee?

Verbeserung

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 ← ∅
foreach v ∈ V do A[v] ← 0
foreach (v,w) ∈ E do A[w] ← A[w] + 1 // Eingangsgrade berechnen
foreach v ∈ V with A[v] = 0 do push(S, v) // Merke Nodes mit Eingangsgrad 0
i ← 1
while S ≠ ∅ do
    v ← pop(S); ord[v] ← i; i ← i + 1 // Wähle Knoten mit Eingangsgrad 0
    foreach (v,w) ∈ E do // Verringere Eingangsgrad der Nachfolger
        A[w] ← 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

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

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

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

Algorithmus Korrektheit

Theorem

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