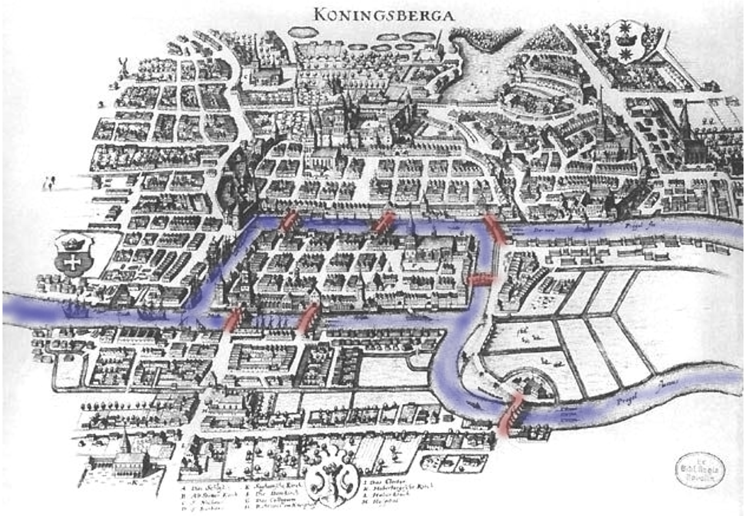
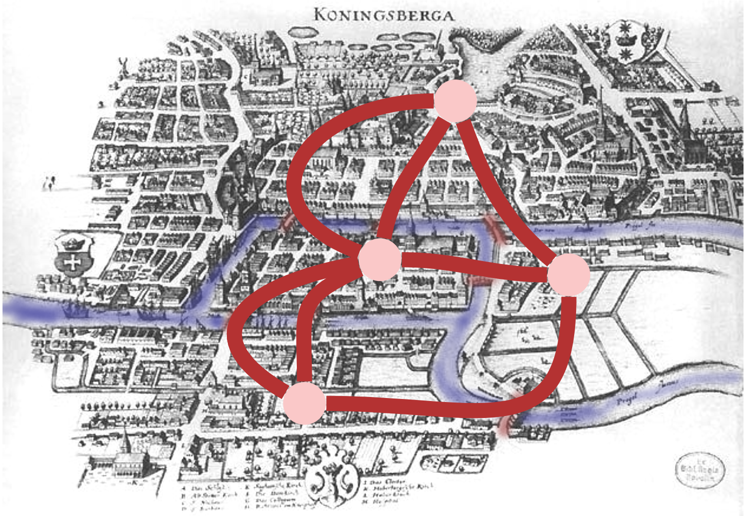
24. Graphs

Notation, Representation, Graph Traversal (DFS, BFS), Topological Sorting , Reflexive transitive closure, Connected components [Ottman/Widmayer, Kap. 9.1 - 9.4,Cormen et al, Kap. 22]

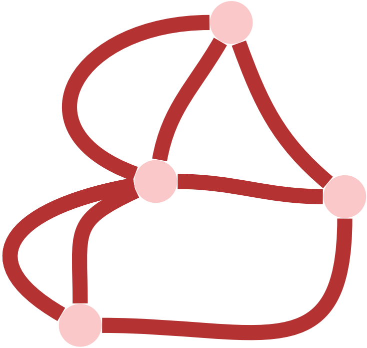
Königsberg 1736



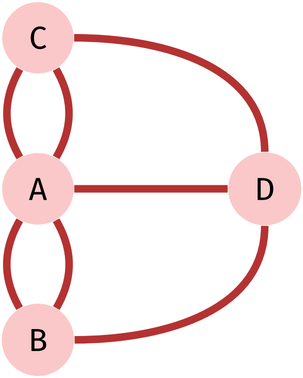
Königsberg 1736



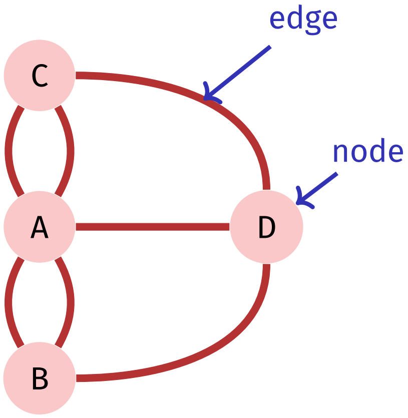
Königsberg 1736



[Multi]Graph

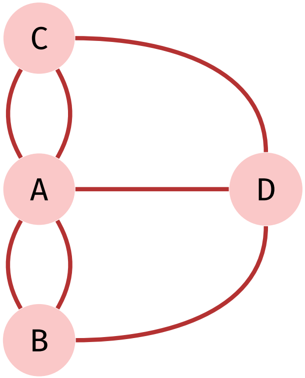


[Multi]Graph



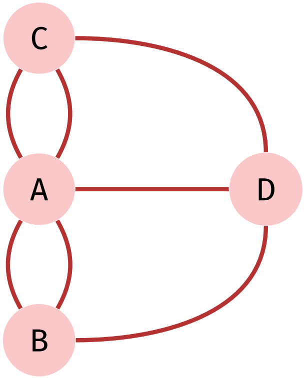
Cycles

- Is there a cycle through the town (the graph) that uses each bridge (each edge) exactly once?



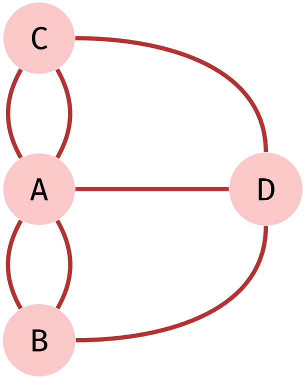
Cycles

- Is there a cycle through the town (the graph) that uses each bridge (each edge) exactly once?
- Euler (1736): no.



Cycles

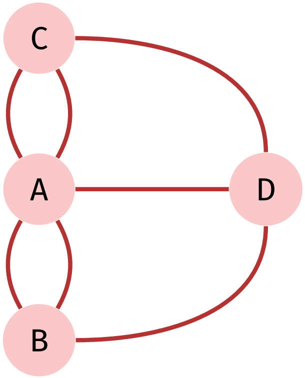
- Is there a cycle through the town (the graph) that uses each bridge (each edge) exactly once?
- Euler (1736): no.
- Such a *cycle* is called *Eulerian path*.



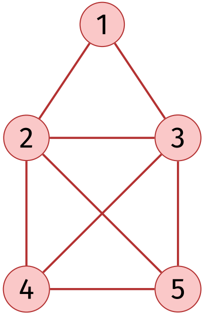
Cycles

- Is there a cycle through the town (the graph) that uses each bridge (each edge) exactly once?
- Euler (1736): no.
- Such a *cycle* is called *Eulerian path*.
- Eulerian path $\Leftrightarrow$ each node provides an even number of edges (each node is of an *even degree*).

  '$\Rightarrow$" is straightforward, "$\Leftarrow$" ist a bit more difficult but still elementary.



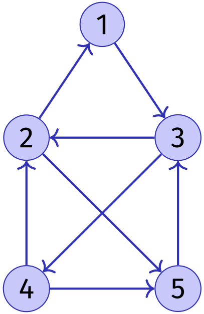
Notation



undirected

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

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



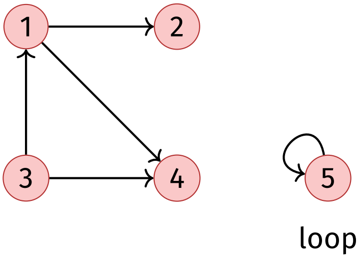
directed

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

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

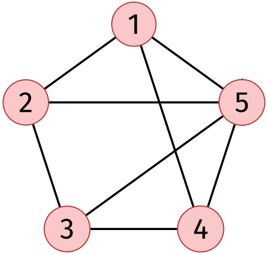
Notation

A directed graph consists of a set $V = \{v_1, \ldots, v_n\}$ of nodes (*Vertices*) *and a set* $E \subseteq V \times V$ *of* Edges. The same edges may not be contained more than once.



Notation

An undirected graph consists of a set $V = \{v_1, \dots, v_n\}$ of nodes a and a set $E \subseteq \{\{u, v\} | u, v \in V\}$ of edges. Edges may not be contained more than once.[37]

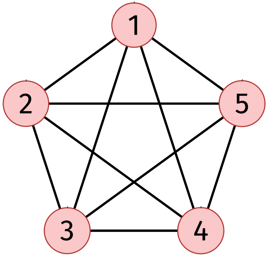


undirected graph

[37] As opposed to the introductory example – it is then called multi-graph.

Notation

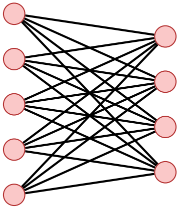
An undirected graph $G = (V, E)$ without loops where E comprises all edges between pairwise different nodes is called complete.



a complete undirected graph

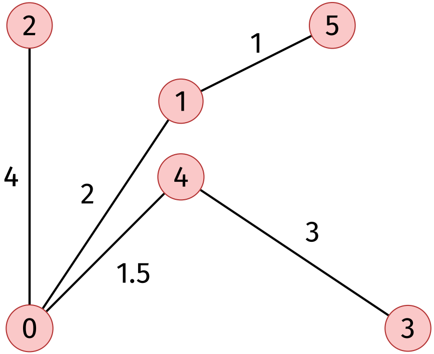
Notation

A graph where V can be partitioned into disjoint sets U and W such that each $e \in E$ provides a node in U and a node in W is called bipartite.



Notation

A weighted graph $G = (V, E, c)$ is a graph $G = (V, E)$ with an edge weight function $c : E \to \mathbb{R}$. $c(e)$ is called weight of the edge e.



Notation

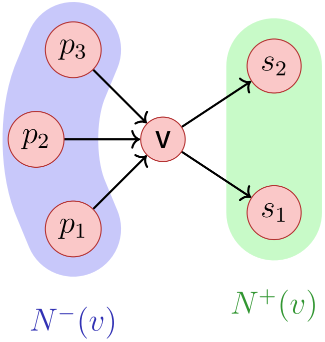
For directed graphs $G = (V, E)$

- $w \in V$ is called adjacent to $v \in V$, if $(v, w) \in E$

Notation

For directed graphs $G = (V, E)$

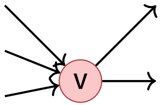
- $w \in V$ is called adjacent to $v \in V$, if $(v, w) \in E$
- Predecessors of $v \in V$: $N^-(v) := \{u \in V | (u, v) \in E\}$.
  Successors: $N^+(v) := \{u \in V | (v, u) \in E\}$



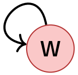
Notation

For directed graphs $G = (V, E)$

- **In-Degree:** $\deg^-(v) = |N^-(v)|$,
  Out-Degree: $\deg^+(v) = |N^+(v)|$



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



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

Notation

For undirected graphs $G = (V, E)$:

- $w \in V$ is called adjacent to $v \in V$, if $\{v, w\} \in E$

Notation

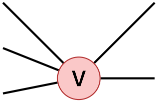
For undirected graphs $G = (V, E)$:

- $w \in V$ is called adjacent to $v \in V$, if $\{v, w\} \in E$
- Neighbourhood of $v \in V$: $N(v) = \{w \in V | \{v, w\} \in E\}$

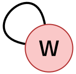
Notation

For undirected graphs $G = (V, E)$:

- $w \in V$ is called adjacent to $v \in V$, if $\{v, w\} \in E$
- Neighbourhood of $v \in V$: $N(v) = \{w \in V | \{v, w\} \in E\}$
- Degree of v: $\deg(v) = |N(v)|$ with a special case for the loops: increase the degree by 2.



$\deg(v) = 5$



$\deg(w) = 2$

Node Degrees $\leftrightarrow$ Number of Edges

For each graph $G = (V, E)$ it holds

1. $\sum_{v \in V} \deg^-(v) = \sum_{v \in V} \deg^+(v) = |E|$, for G directed
2. $\sum_{v \in V} \deg(v) = 2|E|$, for G undirected.

Paths

- Path: a sequence of nodes $\langle v_1, \ldots, v_{k+1} \rangle$ such that for each $i \in \{1 \ldots k\}$ there is an edge from v_i to v_{i+1} .

Paths

- Path: a sequence of nodes $\langle v_1, \ldots, v_{k+1} \rangle$ such that for each $i \in \{1 \ldots k\}$ there is an edge from v_i to v_{i+1} .
- Length of a path: number of contained edges k.

Paths

- Path: a sequence of nodes $\langle v_1, \ldots, v_{k+1} \rangle$ such that for each $i \in \{1 \ldots k\}$ there is an edge from v_i to v_{i+1} .
- Length of a path: number of contained edges k.
- Weight of a path (in weighted graphs): $\sum_{i=1}^{k} c((v_i, v_{i+1}))$ (bzw. $\sum_{i=1}^{k} c(\{v_i, v_{i+1}\})$)

Paths

- Path: a sequence of nodes $\langle v_1, \ldots, v_{k+1} \rangle$ such that for each $i \in \{1 \ldots k\}$ there is an edge from v_i to v_{i+1} .
- Length of a path: number of contained edges k.
- Weight of a path (in weighted graphs): $\sum_{i=1}^{k} c((v_i, v_{i+1}))$ (bzw. $\sum_{i=1}^{k} c(\{v_i, v_{i+1}\})$)
- Simple path: path without repeating vertices

Connectedness

- An undirected graph is called connected, if for each pair $v, w \in V$ there is a connecting path.
- A directed graph is called strongly connected, if for each pair $v, w \in V$ there is a connecting path.
- A directed graph is called weakly connected, if the corresponding undirected graph is connected.

Simple Observations

- generally: $0 \leq |E| \in \mathcal{O}(|V|^2)$
- connected graph: $|E| \in \Omega(|V|)$
- complete graph: $|E| = \frac{|V| \cdot (|V|-1)}{2}$ (undirected)
- Maximally $|E| = |V|^2$ (directed),$|E| = \frac{|V| \cdot (|V|+1)}{2}$ (undirected)

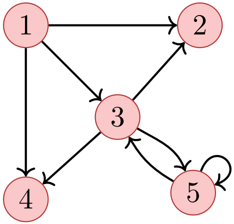
Cycles

- Cycle: path $\langle v_1, \ldots, v_{k+1} \rangle$ with $v_1 = v_{k+1}$
- Simple cycle: Cycle with pairwise different $v_1, \ldots, v_k$, that does not use an edge more than once.
- Acyclic: graph without any cycles.

Conclusion: undirected graphs cannot contain cycles with length 2 (loops have length 1)

Representation using a Matrix

Graph $G = (V, E)$ with nodes $v_1 \ldots, v_n$ stored as adjacency matrix $A_G = (a_{ij})_{1\le i,j\le n}$ with entries from $\{0,1\}$. $a_{ij} = 1$ if and only if edge from v_i to v_j.

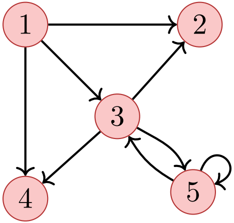


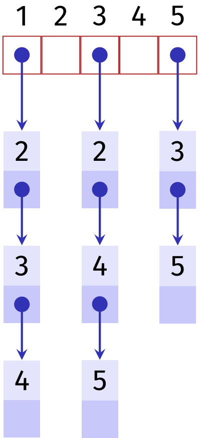
$$\begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \end{pmatrix}$$

Memory consumption $\Theta(|V|^2)$. A_G is symmetric, if G undirected.

Representation with a List

Many graphs $G = (V, E)$ with nodes $v_1, \dots, v_n$ provide much less than n^2 edges. Representation with adjacency list: Array $A[1], \dots, A[n]$, A_i comprises a linked list of nodes in $N^+(v_i)$.





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

Runtimes of simple Operations

Operation	Matrix	List
Find neighbours/successors of $v \in V$		
find $v \in V$ without neighbour/successor		
$(v, u) \in E$?		
Insert edge		
Delete edge (v, u)		

Runtimes of simple Operations

Operation	Matrix	List
Find neighbours/successors of $v \in V$	$\Theta(n)$	
find $v \in V$ without neighbour/successor		
$(v, u) \in E$?		
Insert edge		
Delete edge (v, u)		

Runtimes of simple Operations

Operation	Matrix	List
Find neighbours/successors of $v \in V$	$\Theta(n)$	$\Theta(\deg^+ v)$
find $v \in V$ without neighbour/successor		
$(v, u) \in E$?		
Insert edge		
Delete edge (v, u)		

Runtimes of simple Operations

Operation	Matrix	List
Find neighbours/successors of $v \in V$	$\Theta(n)$	$\Theta(\deg^+ v)$
find $v \in V$ without neighbour/successor	$\Theta(n^2)$	
$(v, u) \in E$?		
Insert edge		
Delete edge (v, u)		

Runtimes of simple Operations

Operation	Matrix	List
Find neighbours/successors of $v \in V$	$\Theta(n)$	$\Theta(\deg^+ v)$
find $v \in V$ without neighbour/successor	$\Theta(n^2)$	$\Theta(n)$
$(v, u) \in E$?		
Insert edge		
Delete edge (v, u)		

Runtimes of simple Operations

Operation	Matrix	List
Find neighbours/successors of $v \in V$	$\Theta(n)$	$\Theta(\deg^+ v)$
find $v \in V$ without neighbour/successor	$\Theta(n^2)$	$\Theta(n)$
$(v, u) \in E$?	$\Theta(1)$	
Insert edge		
Delete edge (v, u)		

Runtimes of simple Operations

Operation	Matrix	List
Find neighbours/successors of $v \in V$	$\Theta(n)$	$\Theta(\deg^+ v)$
find $v \in V$ without neighbour/successor	$\Theta(n^2)$	$\Theta(n)$
$(v, u) \in E$?	$\Theta(1)$	$\Theta(\deg^+ v)$
Insert edge		
Delete edge (v, u)		

Runtimes of simple Operations

Operation	Matrix	List
Find neighbours/successors of $v \in V$	$\Theta(n)$	$\Theta(\deg^+ v)$
find $v \in V$ without neighbour/successor	$\Theta(n^2)$	$\Theta(n)$
$(v, u) \in E$?	$\Theta(1)$	$\Theta(\deg^+ v)$
Insert edge	$\Theta(1)$	
Delete edge (v, u)		

Runtimes of simple Operations

Operation	Matrix	List
Find neighbours/successors of $v \in V$	$\Theta(n)$	$\Theta(\deg^+ v)$
find $v \in V$ without neighbour/successor	$\Theta(n^2)$	$\Theta(n)$
$(v, u) \in E$?	$\Theta(1)$	$\Theta(\deg^+ v)$
Insert edge	$\Theta(1)$	$\Theta(1)$
Delete edge (v, u)		

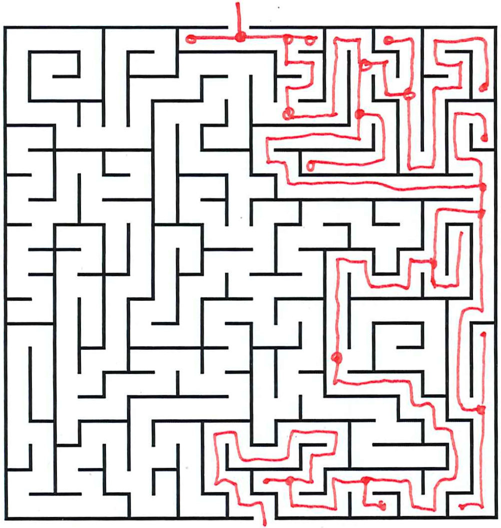
Runtimes of simple Operations

Operation	Matrix	List
Find neighbours/successors of $v \in V$	$\Theta(n)$	$\Theta(\deg^+ v)$
find $v \in V$ without neighbour/successor	$\Theta(n^2)$	$\Theta(n)$
$(v, u) \in E$?	$\Theta(1)$	$\Theta(\deg^+ v)$
Insert edge	$\Theta(1)$	$\Theta(1)$
Delete edge (v, u)	$\Theta(1)$	

Runtimes of simple Operations

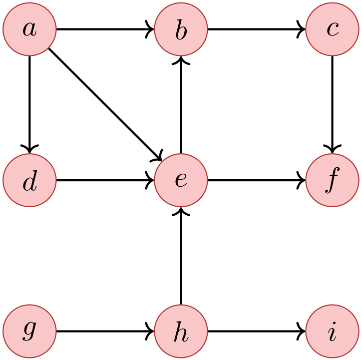
Operation	Matrix	List
Find neighbours/successors of $v \in V$	$\Theta(n)$	$\Theta(\deg^+ v)$
find $v \in V$ without neighbour/successor	$\Theta(n^2)$	$\Theta(n)$
$(v, u) \in E$?	$\Theta(1)$	$\Theta(\deg^+ v)$
Insert edge	$\Theta(1)$	$\Theta(1)$
Delete edge (v, u)	$\Theta(1)$	$\Theta(\deg^+ v)$

Depth First Search



Graph Traversal: Depth First Search

Follow the path into its depth until nothing is left to visit.

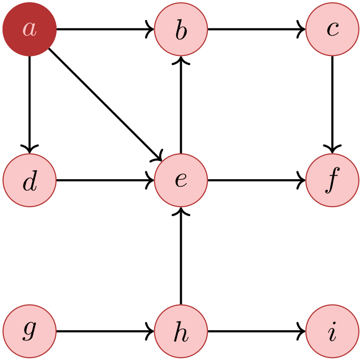


adjacency list

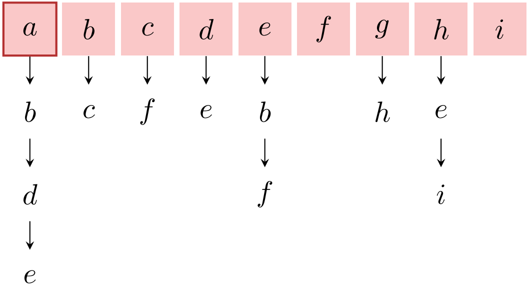
a	b	c	d	e	f	g	h	i
$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$		$\downarrow$	$\downarrow$	
b	c	f	e	b		h	e	
$\downarrow$				$\downarrow$			$\downarrow$	
d				f			i	
$\downarrow$								
e								

Graph Traversal: Depth First Search

Follow the path into its depth until nothing is left to visit.

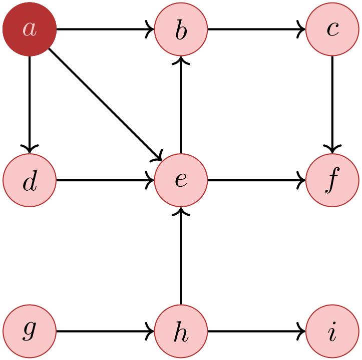


adjacency list

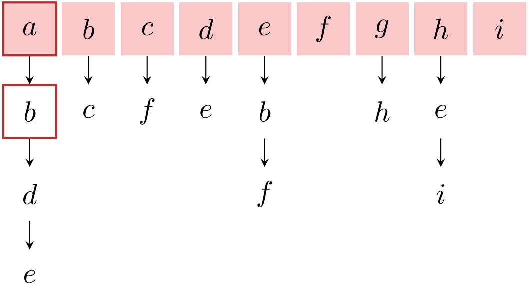


Graph Traversal: Depth First Search

Follow the path into its depth until nothing is left to visit.

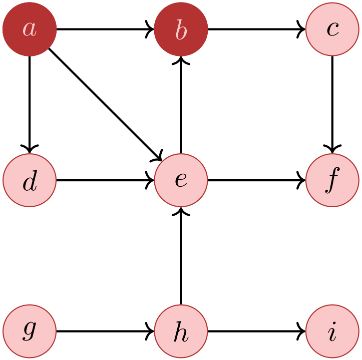


adjacency list

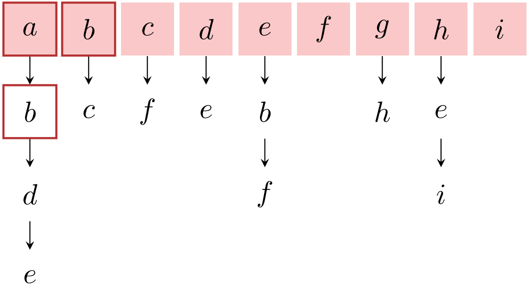


Graph Traversal: Depth First Search

Follow the path into its depth until nothing is left to visit.

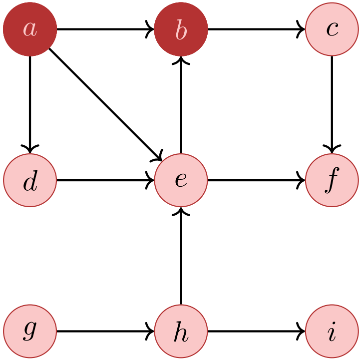


adjacency list

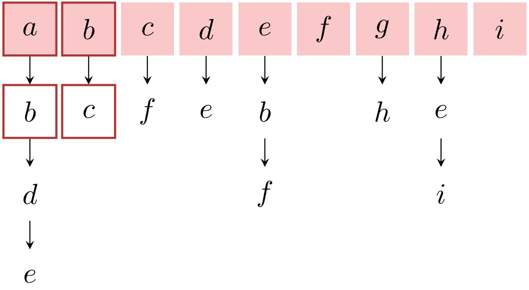


Graph Traversal: Depth First Search

Follow the path into its depth until nothing is left to visit.

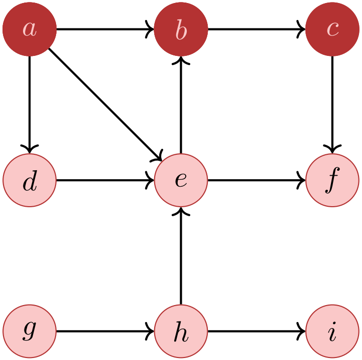


adjacency list

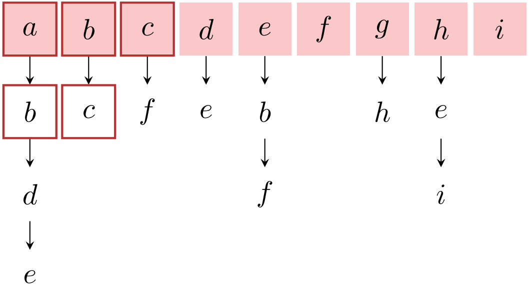


Graph Traversal: Depth First Search

Follow the path into its depth until nothing is left to visit.

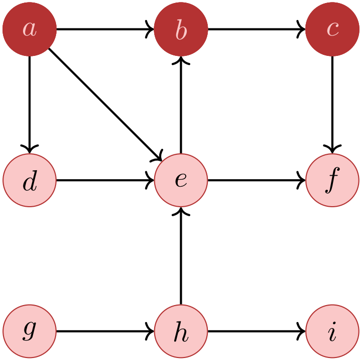


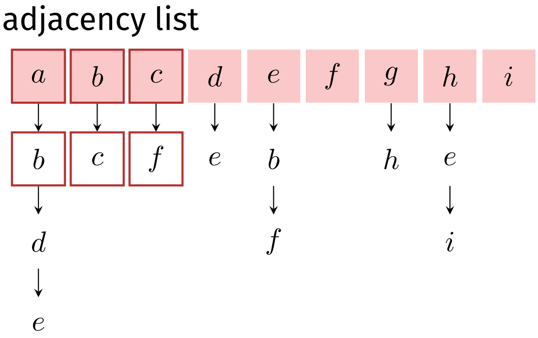
adjacency list



Graph Traversal: Depth First Search

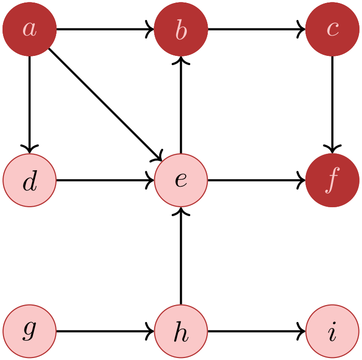
Follow the path into its depth until nothing is left to visit.



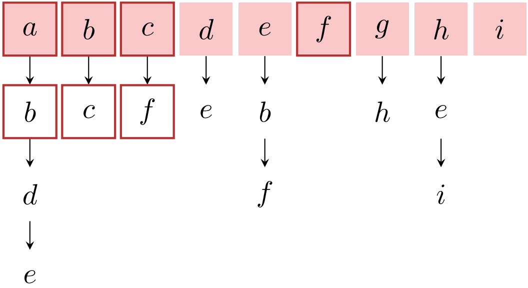


Graph Traversal: Depth First Search

Follow the path into its depth until nothing is left to visit.

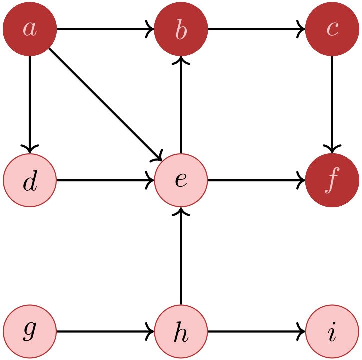


adjacency list

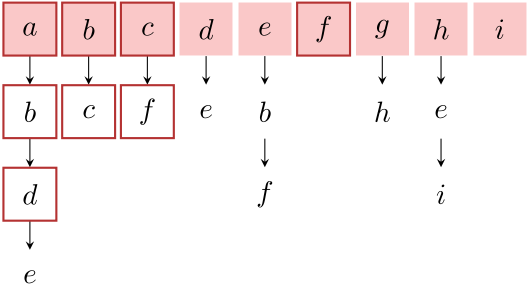


Graph Traversal: Depth First Search

Follow the path into its depth until nothing is left to visit.

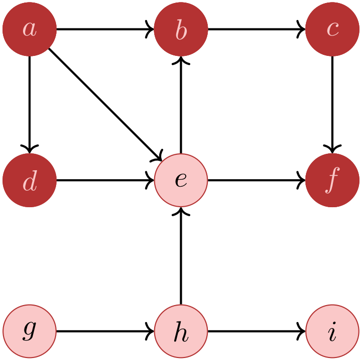


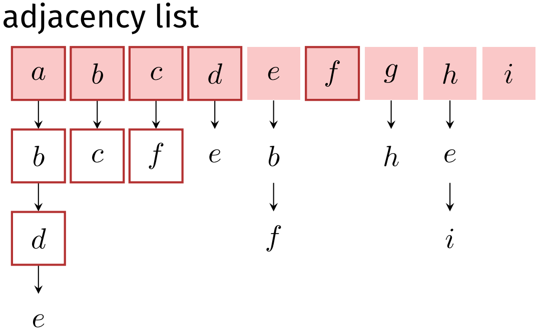
adjacency list



Graph Traversal: Depth First Search

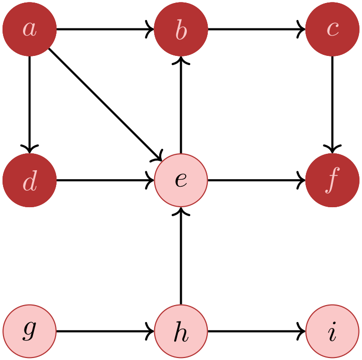
Follow the path into its depth until nothing is left to visit.



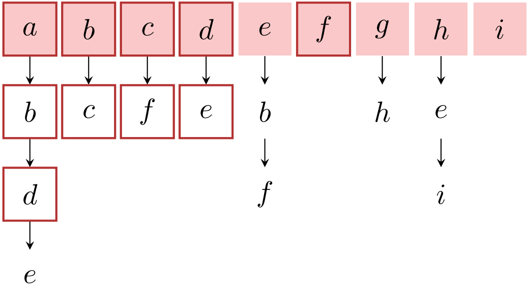


Graph Traversal: Depth First Search

Follow the path into its depth until nothing is left to visit.

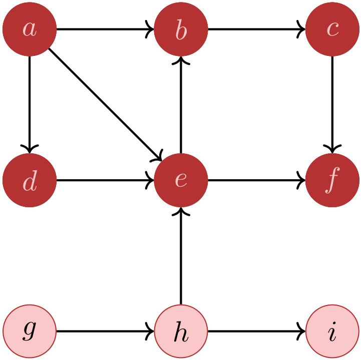


adjacency list



Graph Traversal: Depth First Search

Follow the path into its depth until nothing is left to visit.

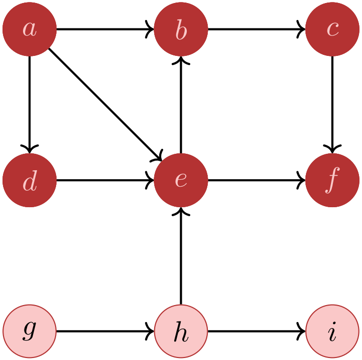


adjacency list

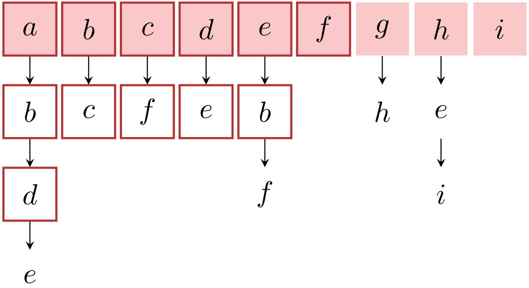
- $a \rightarrow b \rightarrow d \rightarrow e$
- $b \rightarrow c$
- $c \rightarrow f$
- $d \rightarrow e$
- $e \rightarrow b \rightarrow f$
- f
- $g \rightarrow h$
- $h \rightarrow e \rightarrow i$
- i

Graph Traversal: Depth First Search

Follow the path into its depth until nothing is left to visit.

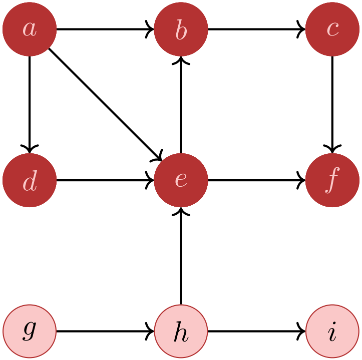


adjacency list

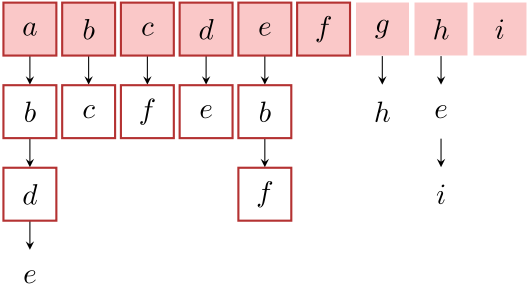


Graph Traversal: Depth First Search

Follow the path into its depth until nothing is left to visit.

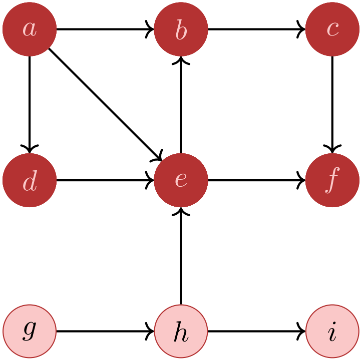


adjacency list

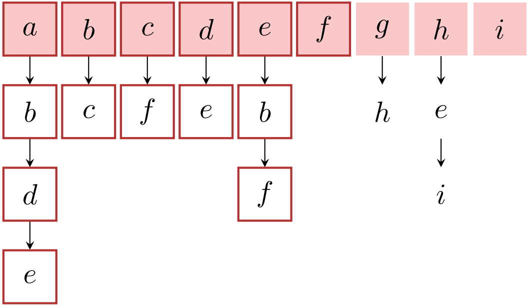


Graph Traversal: Depth First Search

Follow the path into its depth until nothing is left to visit.

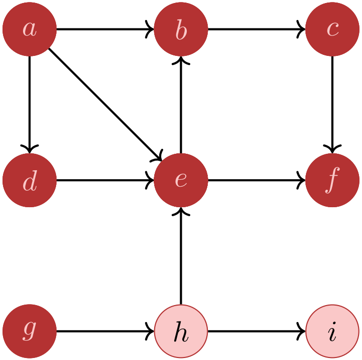


adjacency list

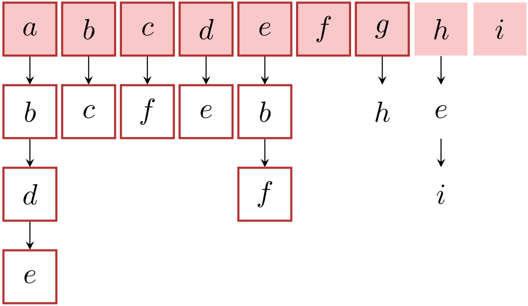


Graph Traversal: Depth First Search

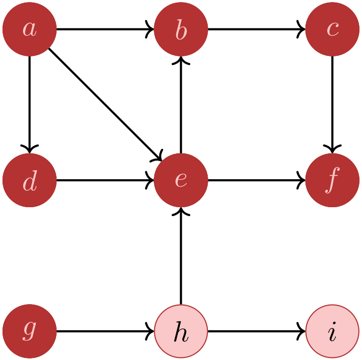
Follow the path into its depth until nothing is left to visit.



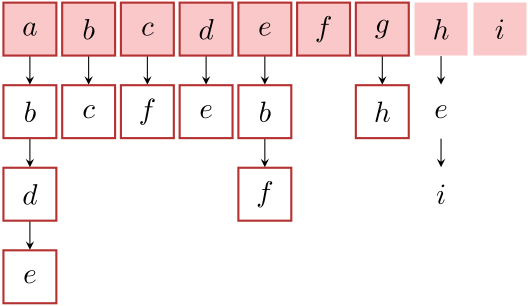
adjacency list



Follow the path into its depth until nothing is left to visit.

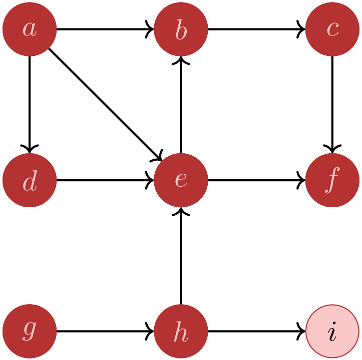


adjacency list

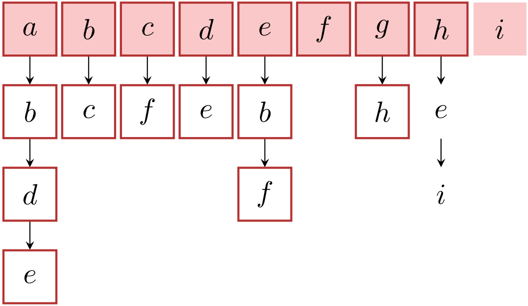


Graph Traversal: Depth First Search

Follow the path into its depth until nothing is left to visit.

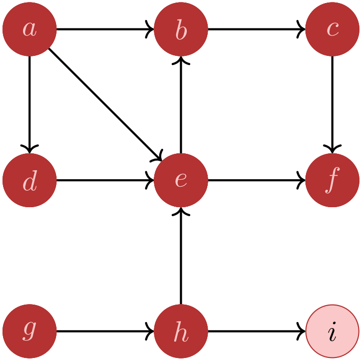


adjacency list

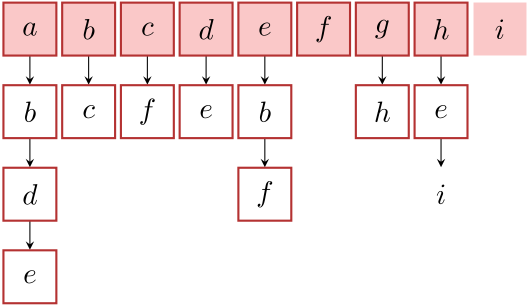


Graph Traversal: Depth First Search

Follow the path into its depth until nothing is left to visit.

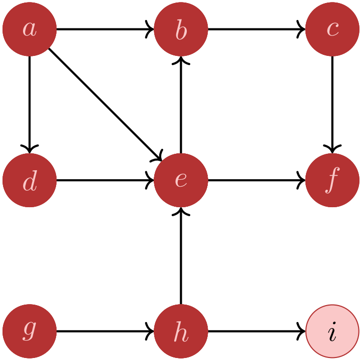


adjacency list

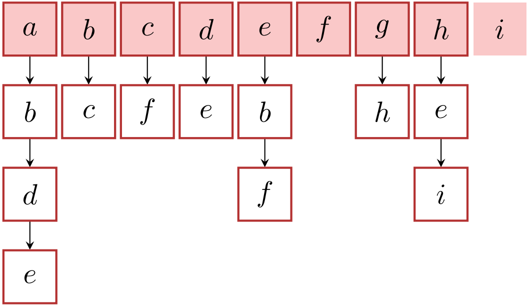


Graph Traversal: Depth First Search

Follow the path into its depth until nothing is left to visit.

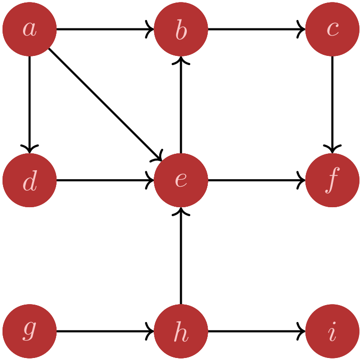


adjacency list



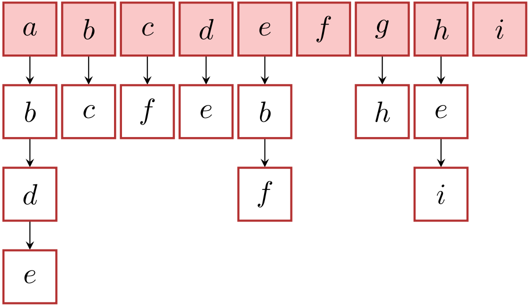
Graph Traversal: Depth First Search

Follow the path into its depth until nothing is left to visit.



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

adjacency list



Colors

Conceptual coloring of nodes

- **white:** node has not been discovered yet.
- **grey:** node has been discovered and is marked for traversal / being processed.
- **black:** node was discovered and entirely processed.

Algorithm Depth First visit 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

Depth First Search starting from node v. Running time (without recursion): $\Theta(\deg^+ v)$

Algorithm Depth First visit 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)

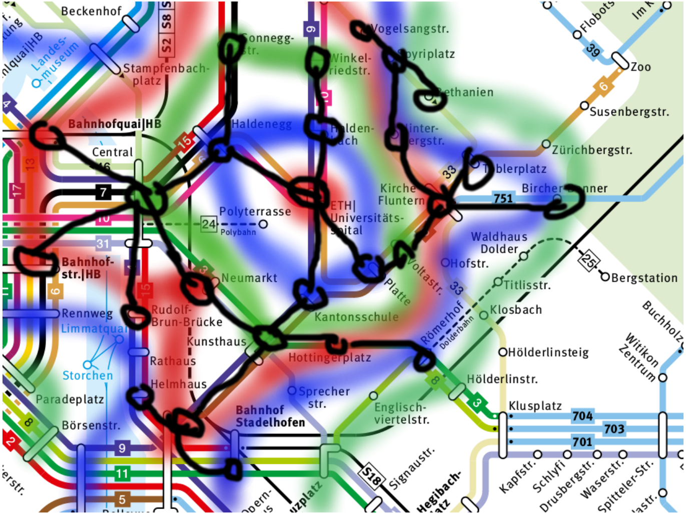
Depth First Search for all nodes of a graph. Running time:
$\Theta(|V| + \sum_{v \in V} (\deg^+(v) + 1)) = \Theta(|V| + |E|)$.

Interpretation of the Colors

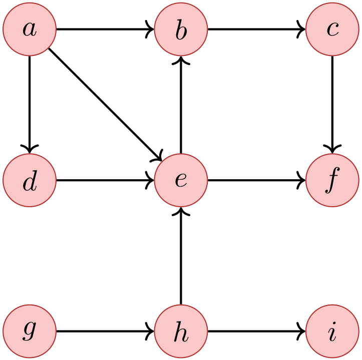
When traversing the graph, a tree (or Forest) is built. When nodes are discovered there are three cases

- White node: new tree edge
- Grey node: cycle (“back-edge”)
- Black node: forward- / cross edge

Breadth First Search



Follow the path in breadth and only then descend into depth.

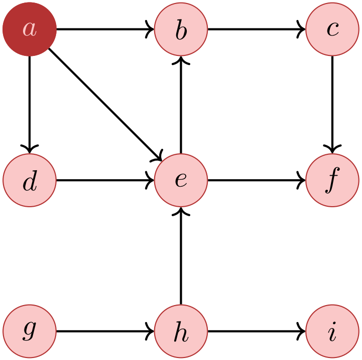


Adjazenzliste

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

Graph Traversal: Breadth First Search

Follow the path in breadth and only then descend into depth.

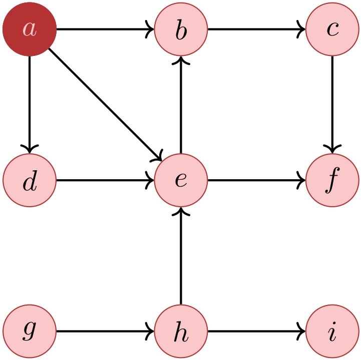


Adjazenzliste

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

Graph Traversal: Breadth First Search

Follow the path in breadth and only then descend into depth.

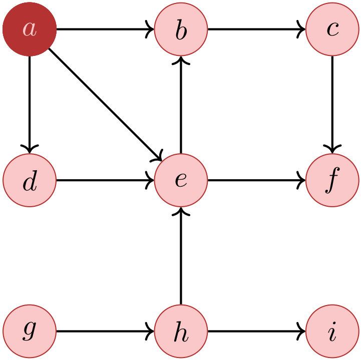


Adjazenzliste

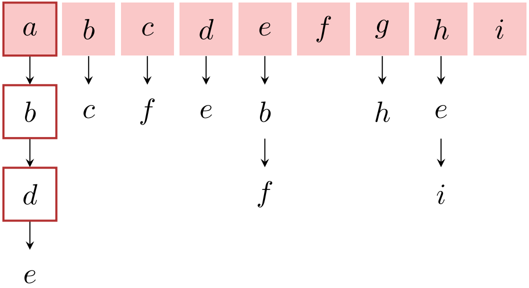
a	b	c	d	e	f	g	h	i
b	c	f	e	b		h	e	
d				f			i	
e								

Graph Traversal: Breadth First Search

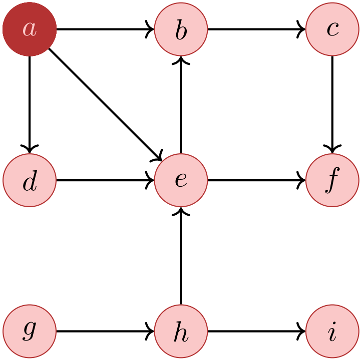
Follow the path in breadth and only then descend into depth.



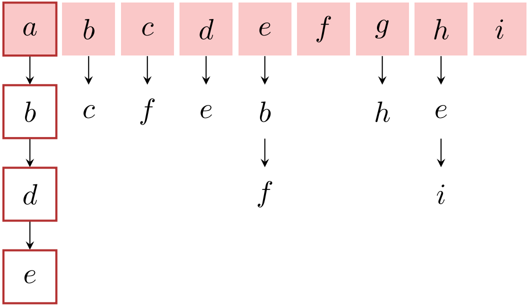
Adjazenzliste



Follow the path in breadth and only then descend into depth.

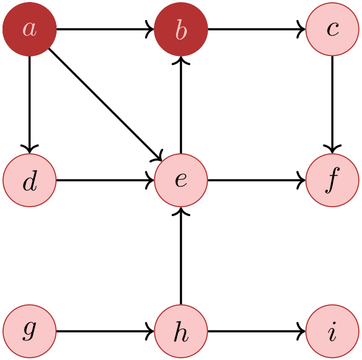


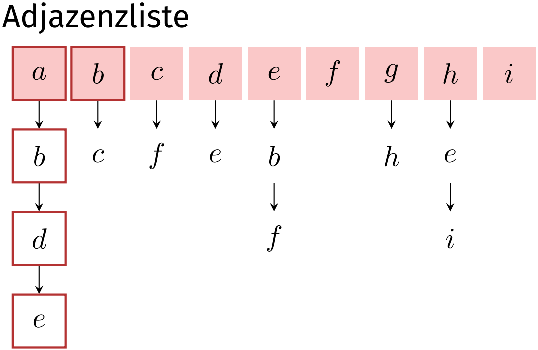
Adjazenzliste



Graph Traversal: Breadth First Search

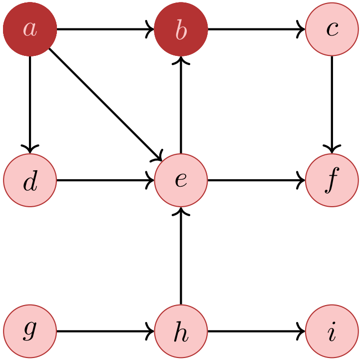
Follow the path in breadth and only then descend into depth.



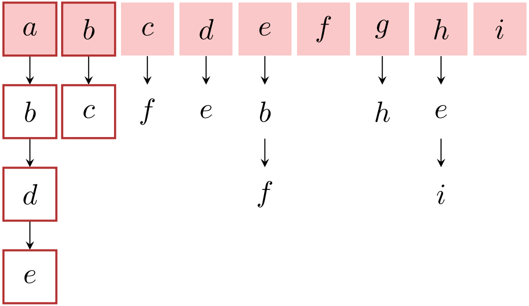


Graph Traversal: Breadth First Search

Follow the path in breadth and only then descend into depth.

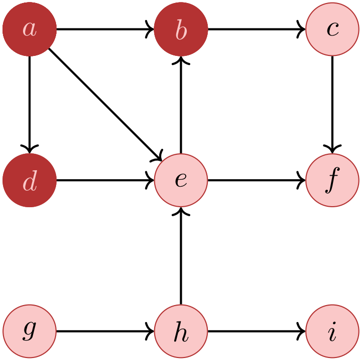


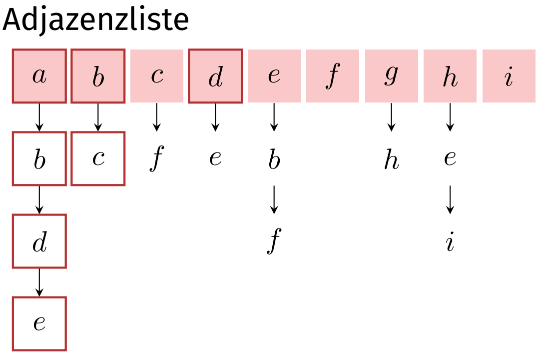
Adjazenzliste



Graph Traversal: Breadth First Search

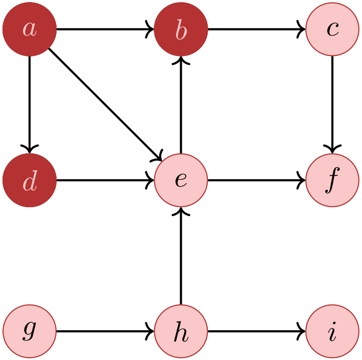
Follow the path in breadth and only then descend into depth.



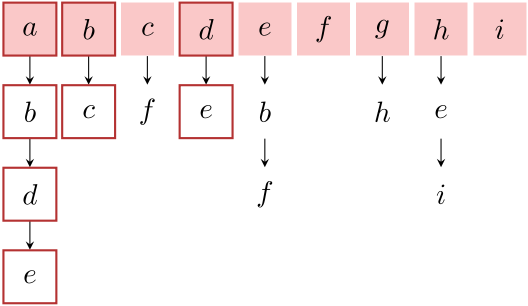


Graph Traversal: Breadth First Search

Follow the path in breadth and only then descend into depth.

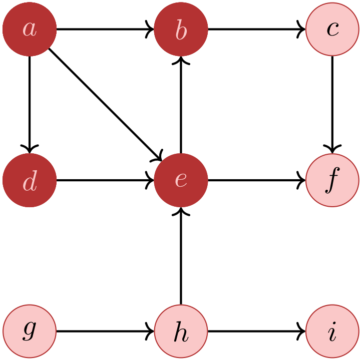


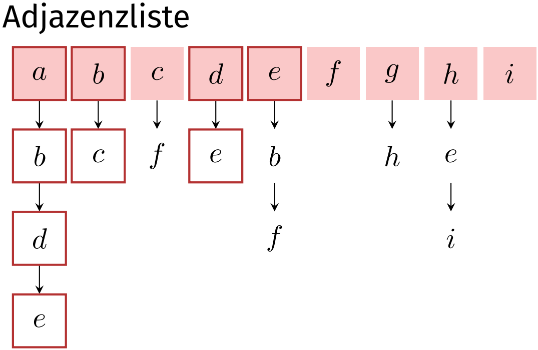
Adjazenzliste



Graph Traversal: Breadth First Search

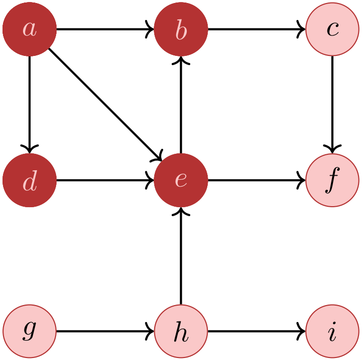
Follow the path in breadth and only then descend into depth.



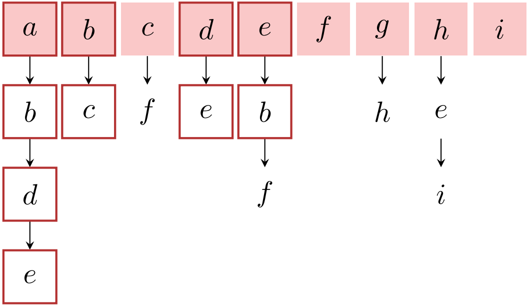


Graph Traversal: Breadth First Search

Follow the path in breadth and only then descend into depth.

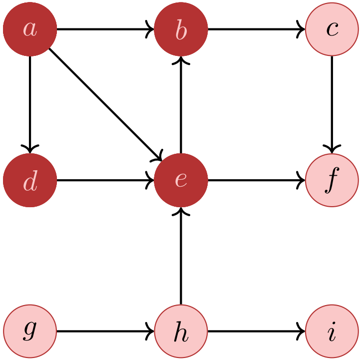


Adjazenzliste

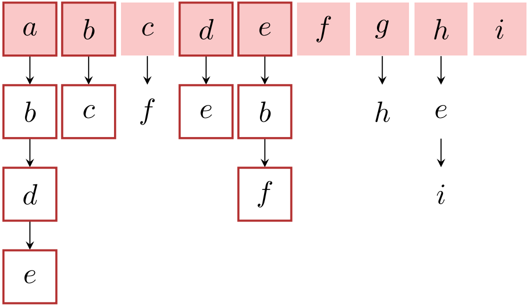


Graph Traversal: Breadth First Search

Follow the path in breadth and only then descend into depth.

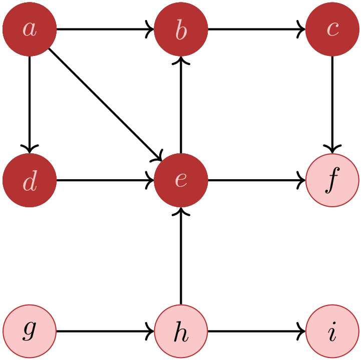


Adjazenzliste

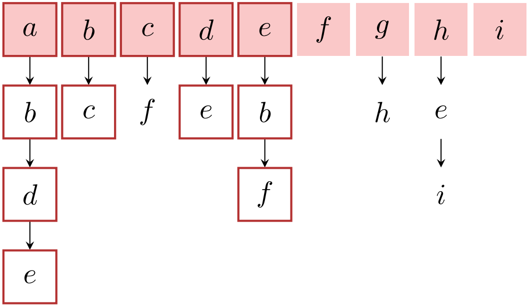


Graph Traversal: Breadth First Search

Follow the path in breadth and only then descend into depth.

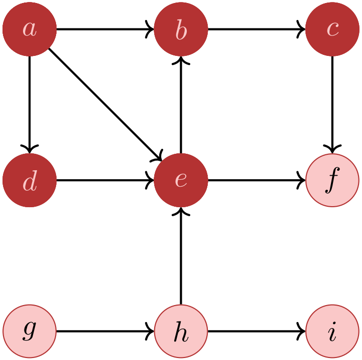


Adjazenzliste

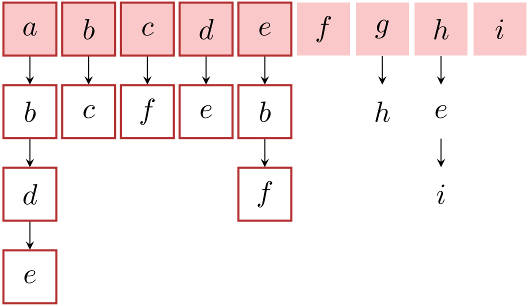


Graph Traversal: Breadth First Search

Follow the path in breadth and only then descend into depth.

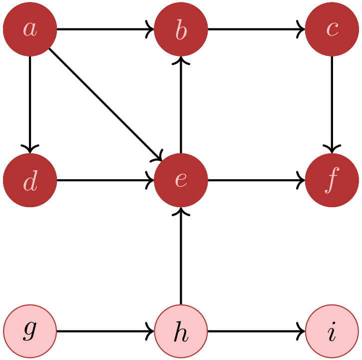


Adjazenzliste

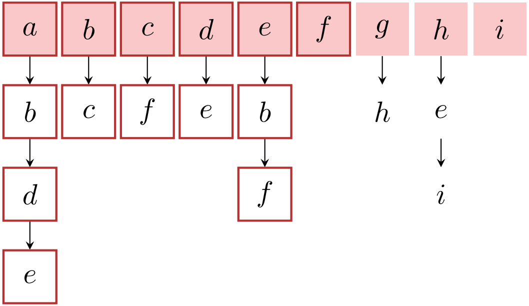


Graph Traversal: Breadth First Search

Follow the path in breadth and only then descend into depth.

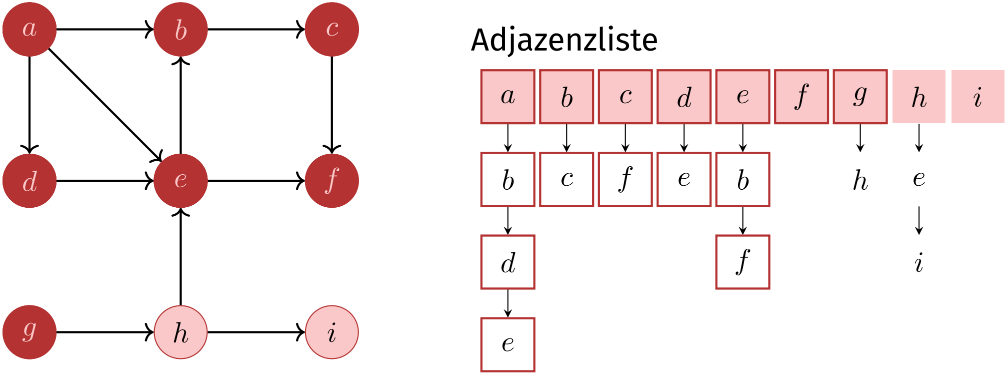


Adjazenzliste



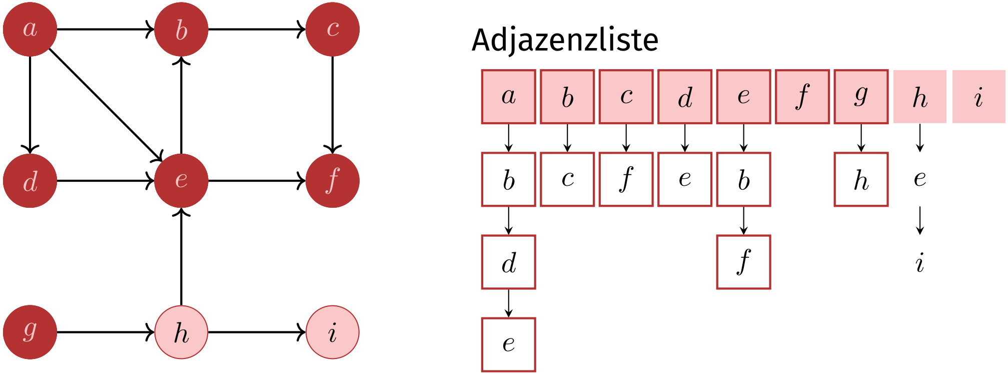
Graph Traversal: Breadth First Search

Follow the path in breadth and only then descend into depth.



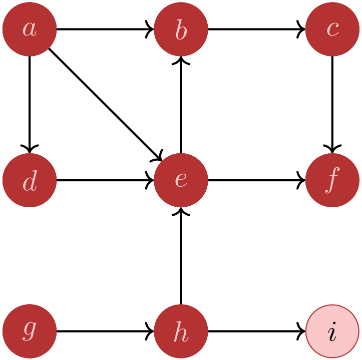
Graph Traversal: Breadth First Search

Follow the path in breadth and only then descend into depth.

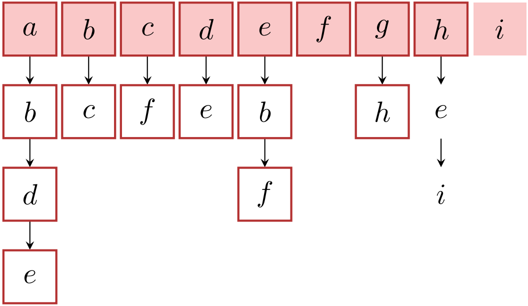


Graph Traversal: Breadth First Search

Follow the path in breadth and only then descend into depth.

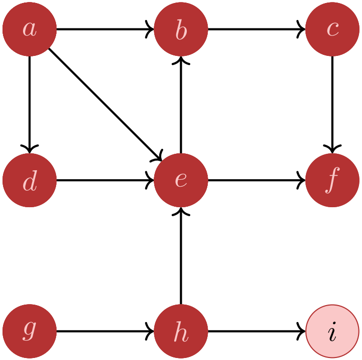


Adjazenzliste

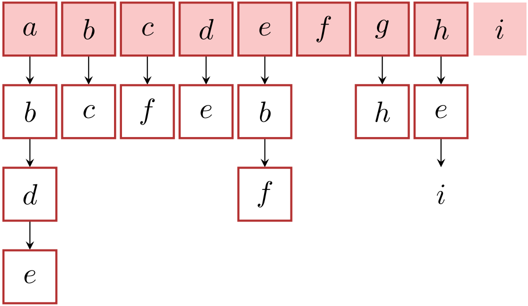


Graph Traversal: Breadth First Search

Follow the path in breadth and only then descend into depth.

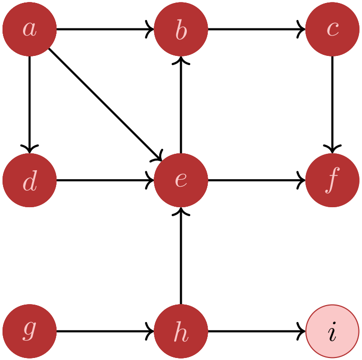


Adjazenzliste

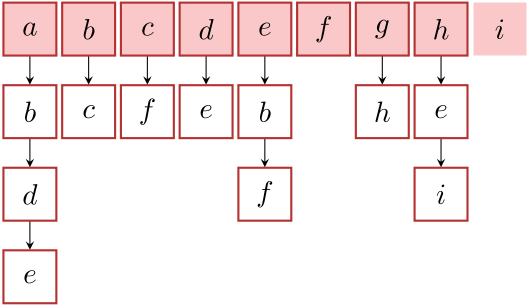


Graph Traversal: Breadth First Search

Follow the path in breadth and only then descend into depth.

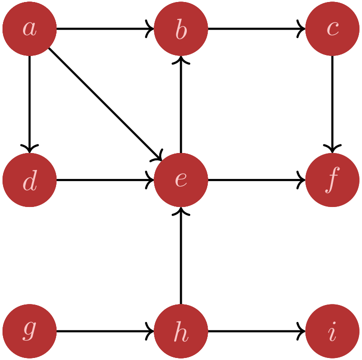


Adjazenzliste



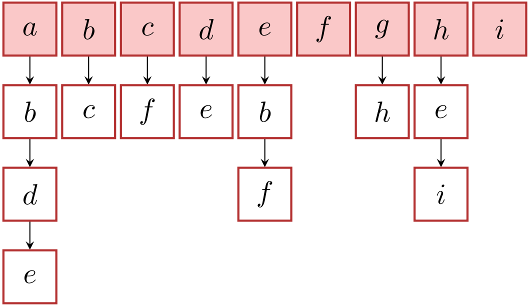
Graph Traversal: Breadth First Search

Follow the path in breadth and only then descend into depth.



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

Adjazenzliste



(Iterative) 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

Algorithm requires extra space of $\mathcal{O}(|V|)$.

Main program BFS-Visit(G)

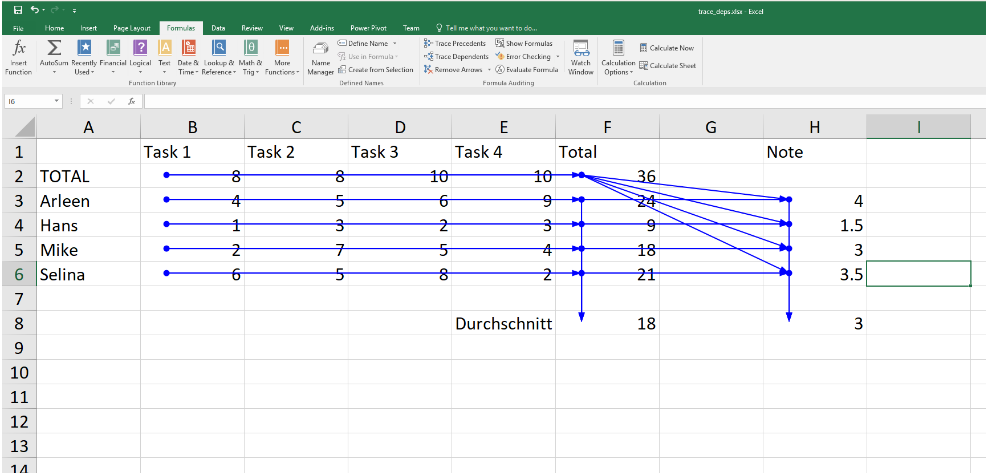
Input: graph $G = (V, E)$

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

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

Breadth First Search for all nodes of a graph. Running time: $\Theta(|V| + |E|)$.

Topological Sorting



Evaluation Order?

Topological Sorting

Topological Sorting of an acyclic directed graph $G = (V, E)$:
Bijective mapping

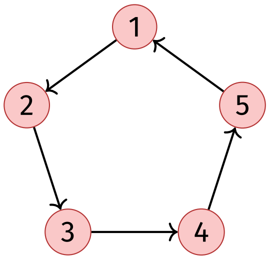
$$\mathrm{ord} : V \to \{1, \dots, |V|\}$$

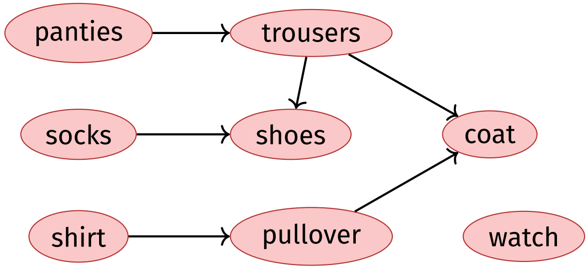
such that

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

Identify i with Element $v_i := \mathrm{ord}^1(i)$. Topological sorting $\hat{=} \langle v_1, \dots, v_{|V|} \rangle$.

(Counter-)Examples





Cyclic graph: cannot be sorted topologically.

A possible toplogical sorting of the graph:
shirt, pullover, panties, watch, trousers, coat, socks, shoes

Observation

Theorem 20

A directed graph $G = (V, E)$ permits a topological sorting if and only if it is acyclic.

Algorithm Topological-Sort(G)

Input: graph $G = (V, E)$.
Output: Topological sorting ord

Stack $S \leftarrow \emptyset$
foreach $v \in V$ **do** $A[v] \leftarrow 0$
foreach $(v, w) \in E$ **do** $A[w] \leftarrow A[w] + 1$ // Compute in-degrees
foreach $v \in V$ with $A[v] = 0$ **do** push(S, v) // Memorize nodes with in-degree 0
$i \leftarrow 1$
while $S \neq \emptyset$ **do**
    $v \leftarrow$ pop(S); $\mathrm{ord}[v] \leftarrow i$; $i \leftarrow i + 1$ // Choose node with in-degree 0
    foreach $(v, w) \in E$ **do** // Decrease in-degree of successors
        $A[w] \leftarrow A[w] - 1$
        if $A[w] = 0$ **then** push(S, w)

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

Algorithm Correctness

Theorem 21

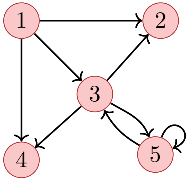
Let $G = (V, E)$ be a directed acyclic graph. Algorithm `TopologicalSort`(G) *computes a topological sorting* ord *for* G *with run-time* $\Theta(|V| + |E|)$.

Algorithm Correctness

Theorem 22

Let $G = (V, E)$ be a directed graph containing a cycle. Algorithm TopologicalSort terminates within $\Theta(|V| + |E|)$ steps and detects a cycle.

Adjacency Matrix Product



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

Interpretation

Theorem 23

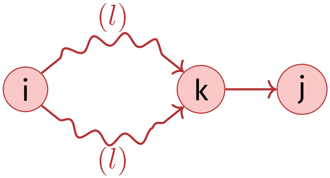
Let $G = (V, E)$ *be a graph and* $k \in \mathbb{N}$*. Then the element* $a_{i,j}^{(k)}$ *of the matrix* $(a_{i,j}^{(k)})_{1 \leq i,j \leq n} = (A_G)^k$ *provides the number of paths with length* k *from* v_i *to* v_j *.*

Proof

By Induction.

Base case: straightforward for $k = 1$. $a_{i,j} = a_{i,j}^{(1)}$.
Hypothesis: claim is true for all $k \leq l$
Step ($l \to l+1$):

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



$a_{k,j} = 1$ iff egde k to j, 0 otherwise. Sum counts the number paths of length l from node v_i to all nodes v_k that provide a direct direction to node v_j, i.e. all paths with length $l+1$.

Relation

Given a finite set V

(Binary) **Relation** R on V: Subset of the cartesian product $V \times V = \{(a,b) | a \in V, b \in V\}$

Relation $R \subseteq V \times V$ is called

- reflexive, if $(v,v) \in R$ for all $v \in V$
- symmetric, if $(v,w) \in R \Rightarrow (w,v) \in R$
- transitive, if $(v,x) \in R$, $(x,w) \in R \Rightarrow (v,w) \in R$

The (Reflexive) Transitive Closure R^* of R is the smallest extension $R \subseteq R^* \subseteq V \times V$ such that R^* is reflexive and transitive.

Graphs and Relations

Graph $G = (V, E)$
adjacencies $A_G \mathrel{\widehat{=}}$ Relation $E \subseteq V \times V$ over V

Graphs and Relations

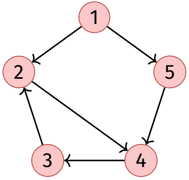
Graph $G = (V, E)$
adjacencies $A_G \mathrel{\widehat{=}}$ Relation $E \subseteq V \times V$ over V

- reflexive $\Leftrightarrow a_{i,i} = 1$ for all $i = 1, \ldots, n$. (loops)
- symmetric $\Leftrightarrow a_{i,j} = a_{j,i}$ for all $i, j = 1, \ldots, n$ (undirected)
- transitive $\Leftrightarrow (u, v) \in E$, $(v, w) \in E \Rightarrow (u, w) \in E$. (reachability)

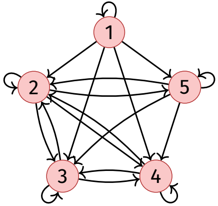
Reflexive Transitive Closure

Reflexive transitive closure of G $\Leftrightarrow$ Reachability relation E^*: $(v, w) \in E^*$ iff $\exists$ path from node v to w.

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \qquad \Rightarrow \qquad \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$



$G = (V, E)$



$G^* = (V, E^*)$

Algorithm $A \cdot A$

Input: (Adjacency-)Matrix $A = (a_{ij})_{i,j=1\dots n}$
Output: Matrix Product $B = (b_{ij})_{i,j=1\dots n} = A \cdot A$

$B \leftarrow 0$
for $r \leftarrow 1$ **to** n **do**
    for $c \leftarrow 1$ **to** n **do**
        for $k \leftarrow 1$ **to** n **do**
            $b_{rc} \leftarrow b_{rc} + a_{rk} \cdot a_{kc}$ // Number of Paths

return B

Counts number of paths of length 2

Algorithm $A \otimes A$

Input: Adjacency-Matrix $A = (a_{ij})_{i,j=1\dots n}$
Output: Modified Matrix Product $B = (b_{ij})_{i,j=1\dots n} = A \otimes A$

```
B ← A                                          // Keep paths
for r ← 1 to n do
    for c ← 1 to n do
        for k ← 1 to n do
            b_rc ← max{b_rc, a_rk · a_kc}      // Path: yes/no
return B
```

$B \leftarrow A$ // Keep paths
for $r \leftarrow 1$ **to** n **do**
  for $c \leftarrow 1$ **to** n **do**
    for $k \leftarrow 1$ **to** n **do**
      $b_{rc} \leftarrow \max\{b_{rc}, a_{rk} \cdot a_{kc}\}$ // Path: yes/no
return B

Computes which paths of length 1 and 2 exist

Computation of the Reflexive Transitive Closure

Goal: computation of $B = (b_{ij})_{1 \leq i,j \leq n}$ with $b_{ij} = 1 \Leftrightarrow (v_i, v_j) \in E^*$

Computation of the Reflexive Transitive Closure

Goal: computation of $B = (b_{ij})_{1 \leq i,j \leq n}$ with $b_{ij} = 1 \Leftrightarrow (v_i, v_j) \in E^*$ First idea:

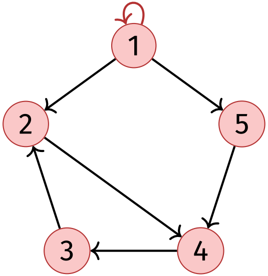
- Start with $B \leftarrow A$ and set $b_{ii} = 1$ for each i (Reflexivity.).
- Compute

$$B_n = \bigotimes_{i=1}^{n} B$$

with powers of 2 $B_2 := B \otimes B$, $B_4 := B_2 \otimes B_2$, $B_8 = B_4 \otimes B_4$ ...
$\Rightarrow$ running time $n^3 \lceil \log_2 n \rceil$

Improvement: Algorithm of Warshall (1962)

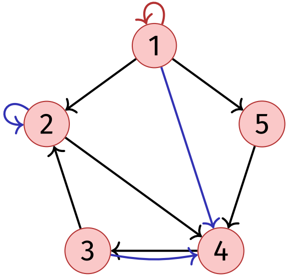
Inductive procedure: all paths known over nodes from $\{v_i : i < k\}$. Add node v_k.



$$\begin{bmatrix} \color{red}{1} & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Improvement: Algorithm of Warshall (1962)

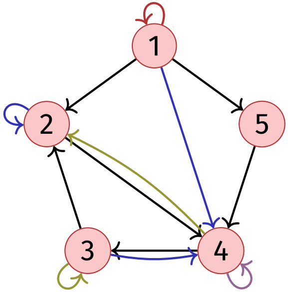
Inductive procedure: all paths known over nodes from $\{v_i : i < k\}$. Add node v_k.



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

Improvement: Algorithm of Warshall (1962)

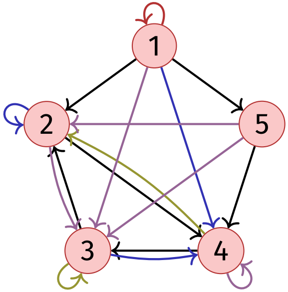
Inductive procedure: all paths known over nodes from $\{v_i : i < k\}$. Add node v_k.



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

Improvement: Algorithm of Warshall (1962)

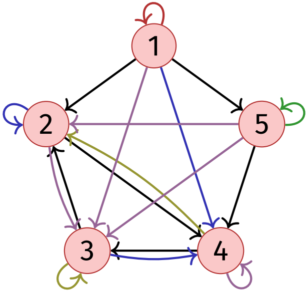
Inductive procedure: all paths known over nodes from $\{v_i : i < k\}$. Add node v_k.



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

Improvement: Algorithm of Warshall (1962)

Inductive procedure: all paths known over nodes from $\{v_i : i < k\}$. Add node v_k.



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

Algorithm TransitiveClosure(A_G)

Input: Adjacency matrix $A_G = (a_{ij})_{i,j=1\dots n}$
Output: Reflexive transitive closure $B = (b_{ij})_{i,j=1\dots n}$ of G

```
B ← A_G
for k ← 1 to n do
    b_kk ← 1                                          // Reflexivity
    for r ← 1 to n do
        for c ← 1 to n do
            b_rc ← max{b_rc, b_rk · b_kc}             // All paths via v_k
return B
```

Runtime $\Theta(n^3)$.

Correctness of the Algorithm (Induction)

Invariant (k): all paths via nodes with maximal index $< k$ considered.

- **Base case ($k = 1$)**: All directed paths (all edges) in A_G considered.
- **Hypothesis**: invariant (k) fulfilled.
- **Step** ($k \to k + 1$): For each path from v_i to v_j via nodes with maximal index k: by the hypothesis $b_{ik} = 1$ and $b_{kj} = 1$. Therefore in the k-th iteration: $b_{ij} \leftarrow 1$.