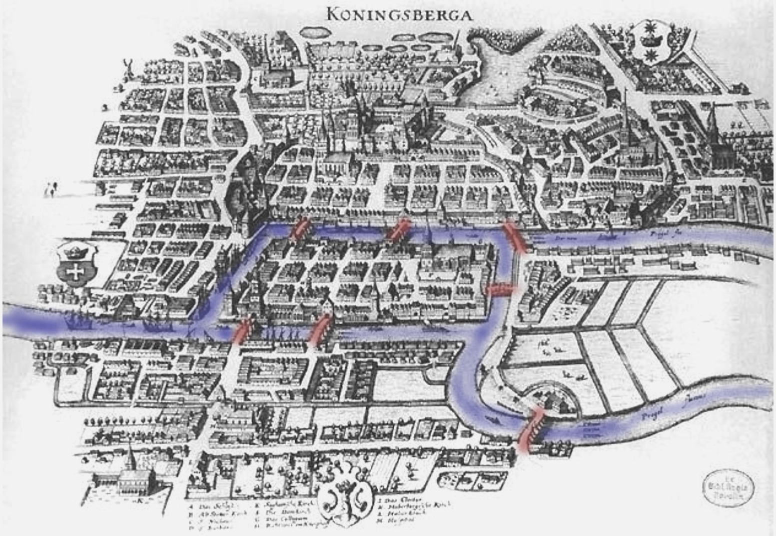
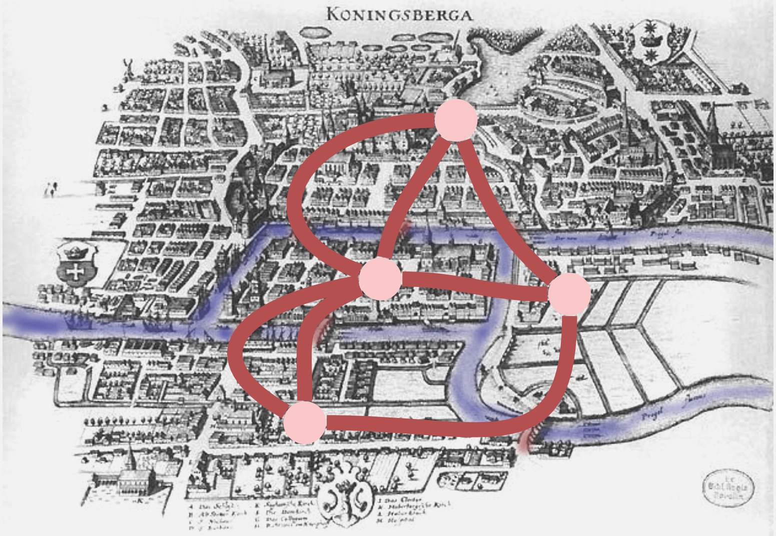
13. Graphs

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

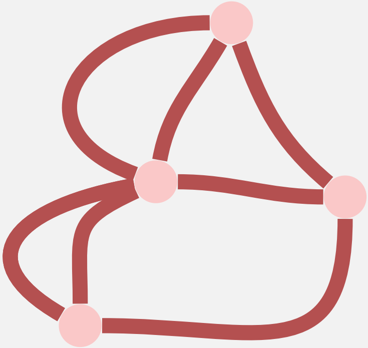
Königsberg 1736



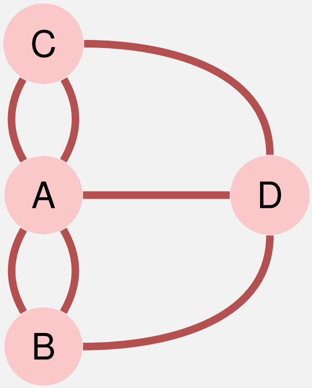
Königsberg 1736



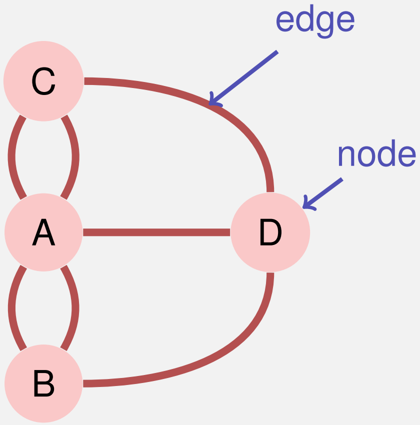
Königsberg 1736



[Multi]Graph

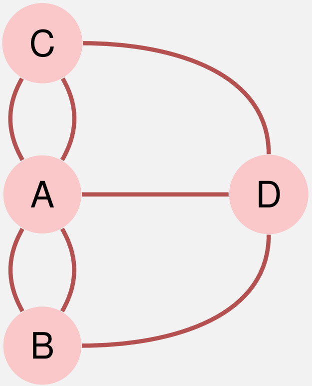


[Multi]Graph



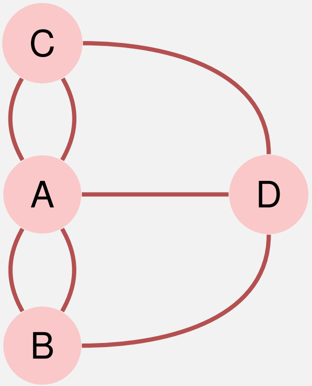
Cycles

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



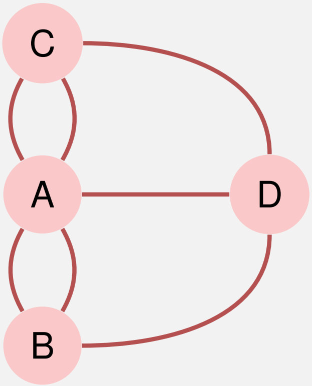
Cycles

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



Cycles

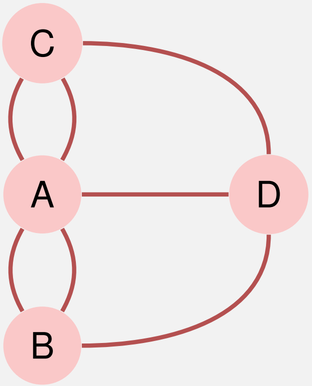
- Is there a cycle through the town (the graph) that uses each bridge (each edge) exactly once?
- Euler (1736): no.
- Such a *cycle* is called *Eulerian path*.



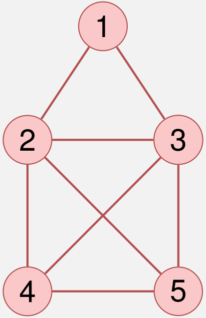
Cycles

- 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*).

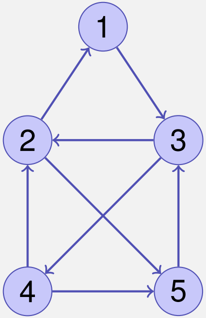
  '$\Rightarrow$" is straightforward, "$\Leftarrow$" ist a bit more difficult but still elementary.



Notation



undirected



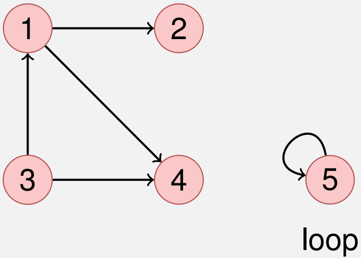
directed

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

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

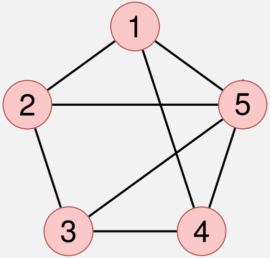
Notation

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 $V = \{v_1, \ldots, v_n\}$ of nodes a and a set $E \subseteq \{\{u, v\} | u, v \in V\}$ of edges. Edges may bot be contained more than once.[22]

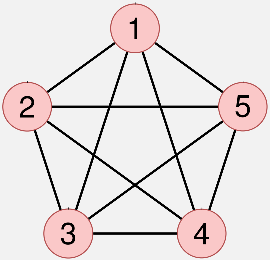


undirected graph

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

Notation

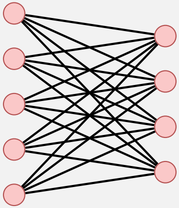
An undirected graph $G = (V, E)$ without loops where E comprises all edges between pairwise different nodes is called *complete*.



a complete undirected graph

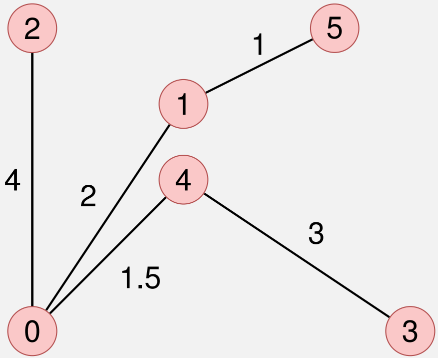
Notation

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

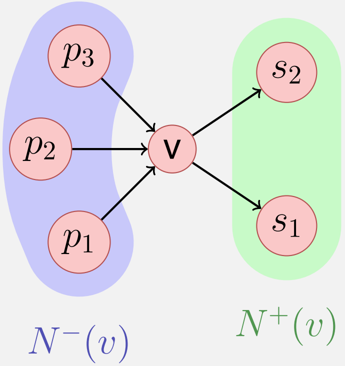
For directed graphs $G = (V, E)$

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

Notation

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

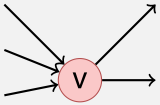
- $w \in V$ is called adjacent to $v \in V$, if $(v, w) \in E$
- *Predecessors* of $v \in V$: $N^-(v) := \{u \in V | (u, v) \in E\}$.
  Successors: $N^+(v) := \{u \in V | (v, u) \in E\}$



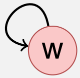
Notation

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

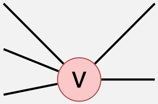
For undirected graphs $G = (V, E)$:

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

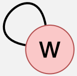
Notation

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.



$\deg(v) = 5$



$\deg(w) = 2$

Relationship between node degrees and 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 eacheach 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)

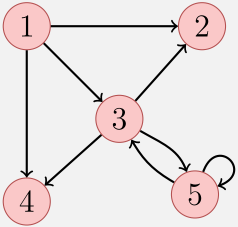
Cycles

- *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.

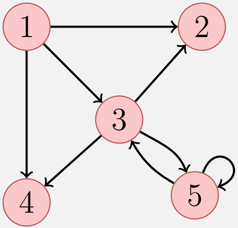


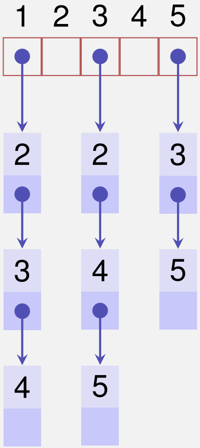
$$\begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \end{pmatrix}$$

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

Representation with a List

Many graphs $G = (V, E)$ with nodes $v_1, \ldots, v_n$ provide much less than n^2 edges. Representation with *adjacency list*: Array $A[1], \ldots, A[n]$, 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		
$(u, v) \in E$?		
Insert edge		
Delete edge		

Runtimes of simple Operations

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

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		
$(u, v) \in E$?		
Insert edge		
Delete edge		

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	$\Theta(n^2)$	
$(u, v) \in E$?		
Insert edge		
Delete edge		

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	$\Theta(n^2)$	$\Theta(n)$
$(u, v) \in E$?		
Insert edge		
Delete edge		

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	$\Theta(n^2)$	$\Theta(n)$
$(u, v) \in E$?	$\Theta(1)$	
Insert edge		
Delete edge		

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	$\Theta(n^2)$	$\Theta(n)$
$(u, v) \in E$?	$\Theta(1)$	$\Theta(\deg^+ v)$
Insert edge		
Delete edge		

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	$\Theta(n^2)$	$\Theta(n)$
$(u, v) \in E$?	$\Theta(1)$	$\Theta(\deg^+ v)$
Insert edge	$\Theta(1)$	
Delete edge		

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	$\Theta(n^2)$	$\Theta(n)$
$(u, v) \in E$?	$\Theta(1)$	$\Theta(\deg^+ v)$
Insert edge	$\Theta(1)$	$\Theta(1)$
Delete edge		

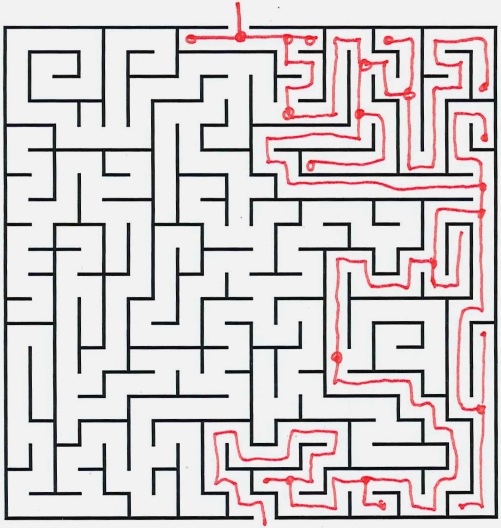
Runtimes of simple Operations

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

Runtimes of simple Operations

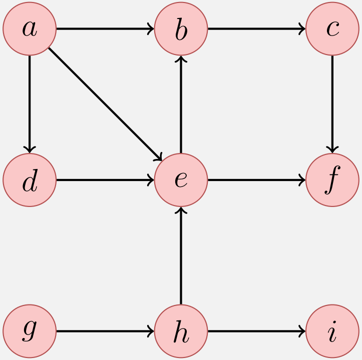
Operation	Matrix	List
Find neighbours/successors of $v \in V$	$\Theta(n)$	$\Theta(\deg^+ v)$
find $v \in V$ without neighbour/successor	$\Theta(n^2)$	$\Theta(n)$
$(u, v) \in E$?	$\Theta(1)$	$\Theta(\deg^+ v)$
Insert edge	$\Theta(1)$	$\Theta(1)$
Delete edge	$\Theta(1)$	$\Theta(\deg^+ v)$

Depth First Search



Graph Traversal: Depth First Search

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

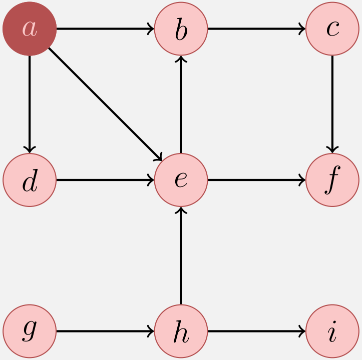


Adjazenzliste

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

Graph Traversal: Depth First Search

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

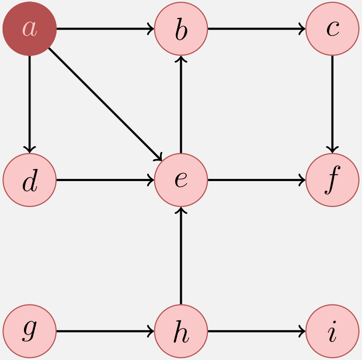


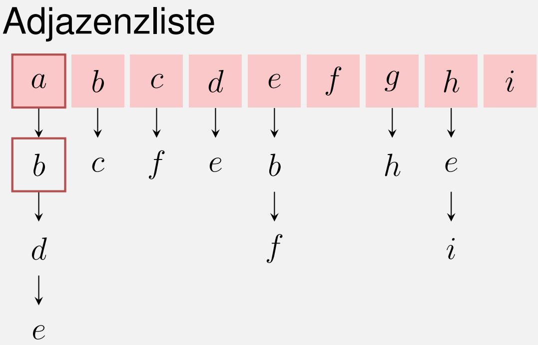
Adjazenzliste

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

Graph Traversal: Depth First Search

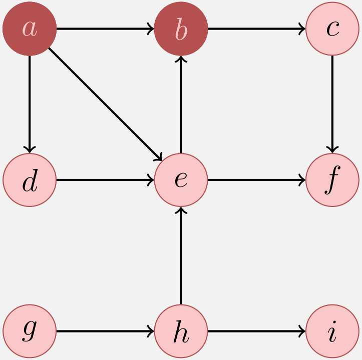
Follow the path into its depth until nothing is left to visit.





Graph Traversal: Depth First Search

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



Adjazenzliste

$a \rightarrow \boxed{b} \rightarrow d \rightarrow e$

$b \rightarrow c$

$c \rightarrow f$

$d \rightarrow e$

$e \rightarrow b \rightarrow f$

f

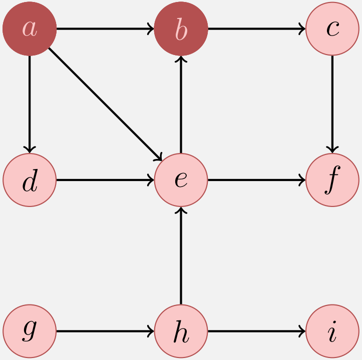
$g \rightarrow h$

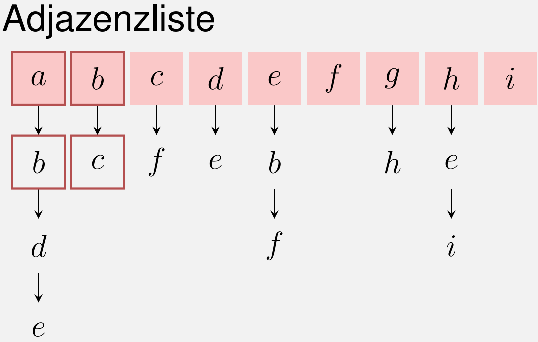
$h \rightarrow e \rightarrow i$

i

Graph Traversal: Depth First Search

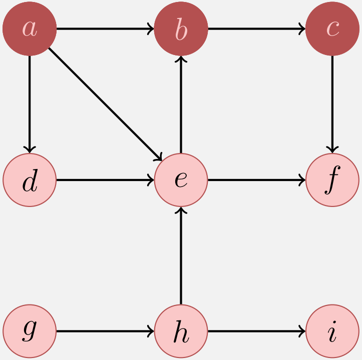
Follow the path into its depth until nothing is left to visit.



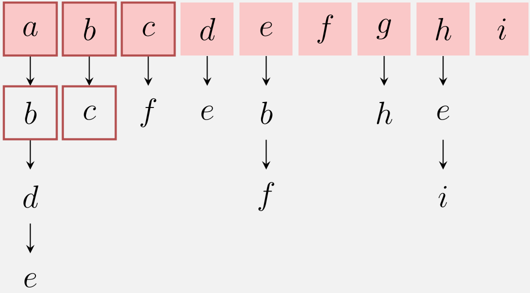


Graph Traversal: Depth First Search

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

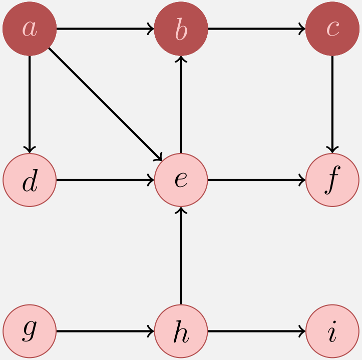


Adjazenzliste



Graph Traversal: Depth First Search

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

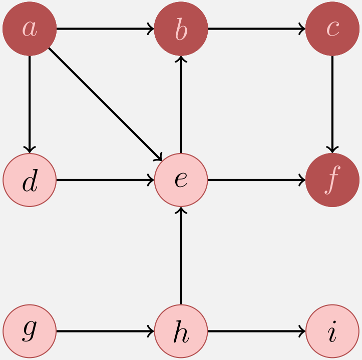


Adjazenzliste

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

Graph Traversal: Depth First Search

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

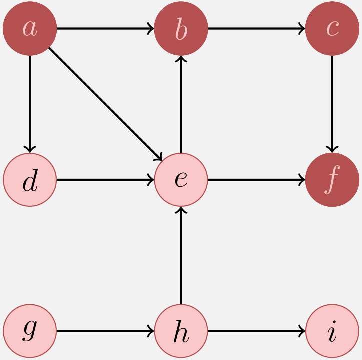


Adjazenzliste

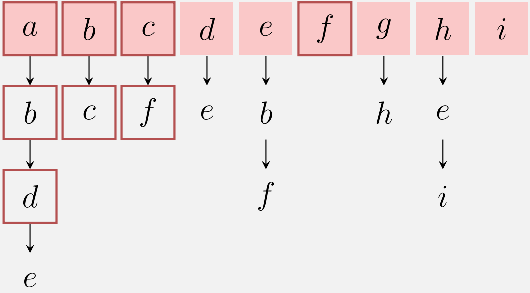
a	b	c	d	e	f	g	h	i
b	c	f	e	b		h	e	
d				f			i	
e								

Graph Traversal: Depth First Search

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

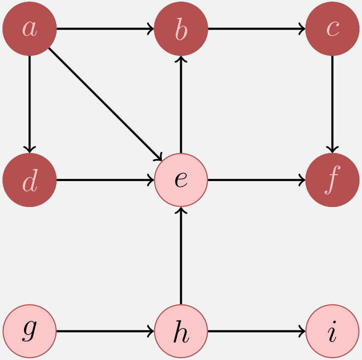


Adjazenzliste

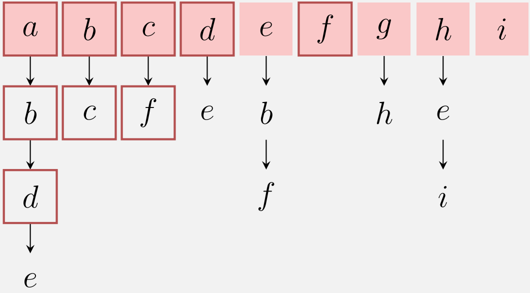


Graph Traversal: Depth First Search

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

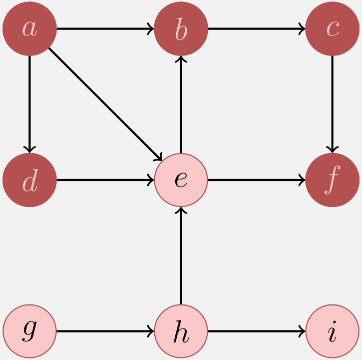


Adjazenzliste

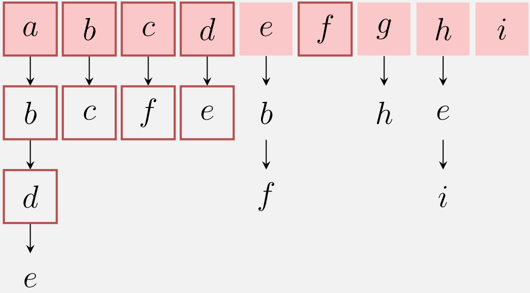


Graph Traversal: Depth First Search

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

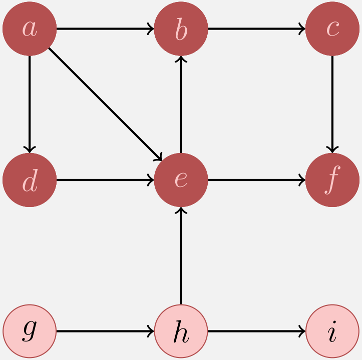


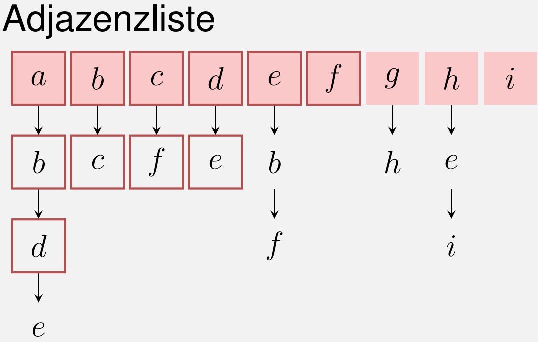
Adjazenzliste



Graph Traversal: Depth First Search

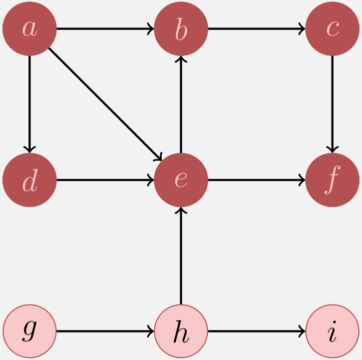
Follow the path into its depth until nothing is left to visit.



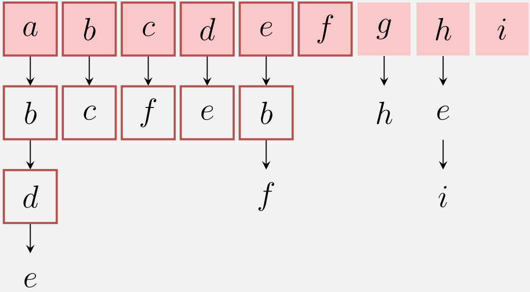


Graph Traversal: Depth First Search

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

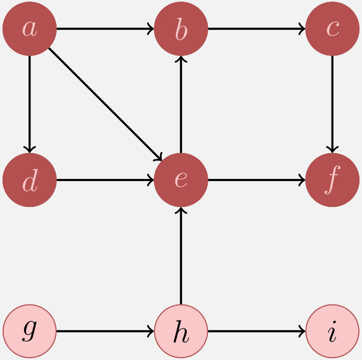


Adjazenzliste

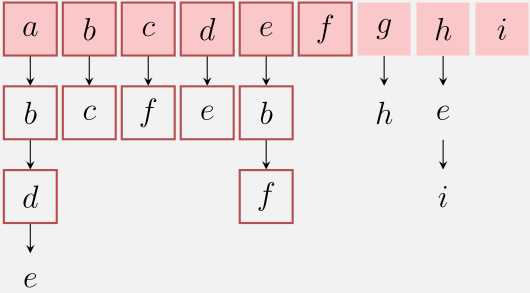


Graph Traversal: Depth First Search

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

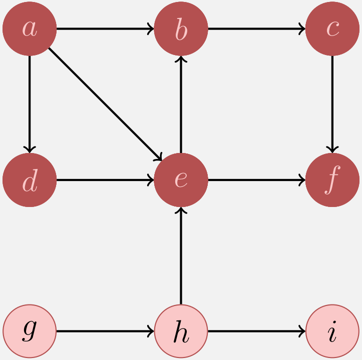


Adjazenzliste

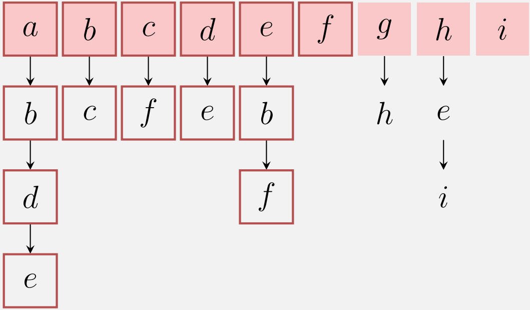


Graph Traversal: Depth First Search

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

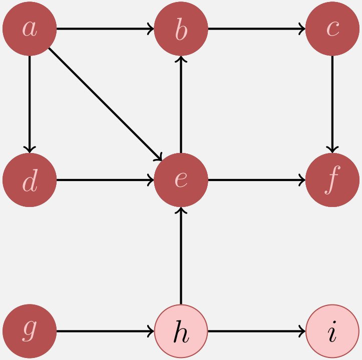


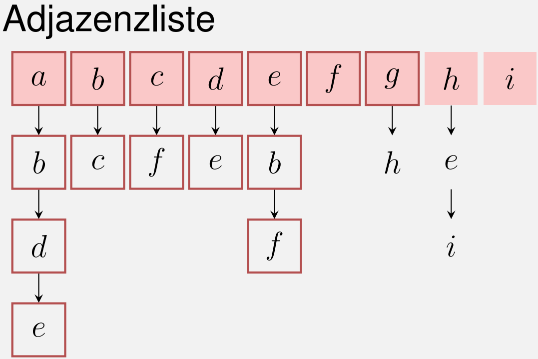
Adjazenzliste



Graph Traversal: Depth First Search

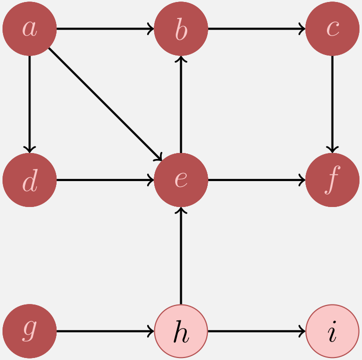
Follow the path into its depth until nothing is left to visit.



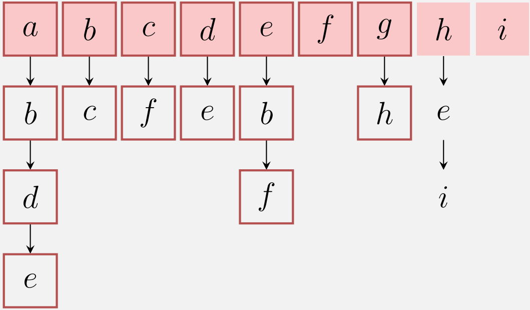


Graph Traversal: Depth First Search

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

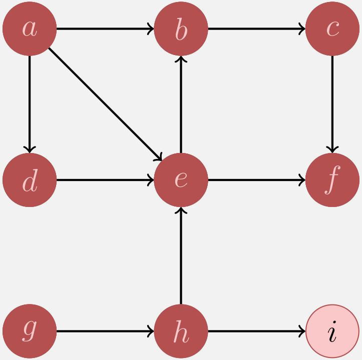


Adjazenzliste

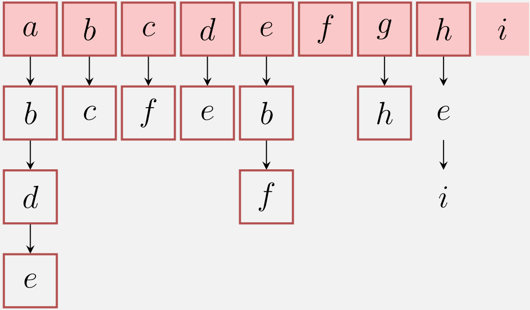


Graph Traversal: Depth First Search

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

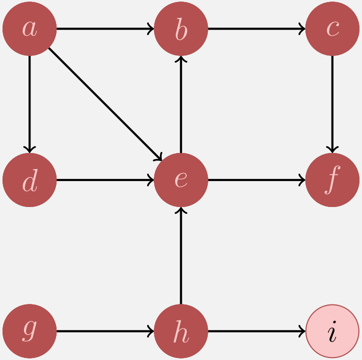


Adjazenzliste

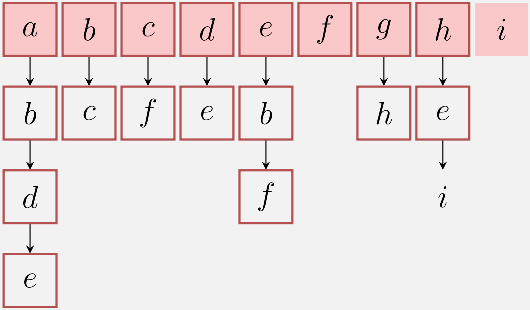


Graph Traversal: Depth First Search

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

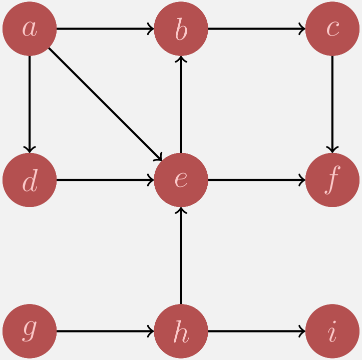


Adjazenzliste



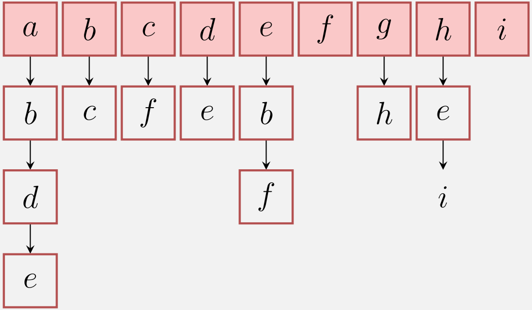
Graph Traversal: Depth First Search

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



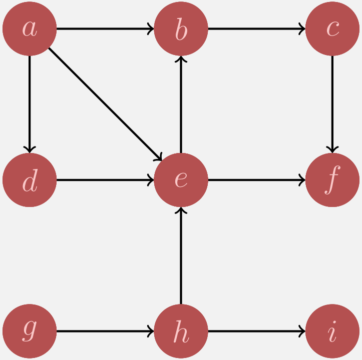
Order $a, b, c, f, d, e, g, h, i$

Adjazenzliste



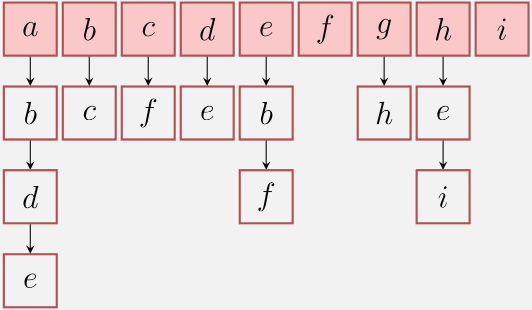
Graph Traversal: Depth First Search

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



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

Adjazenzliste



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)

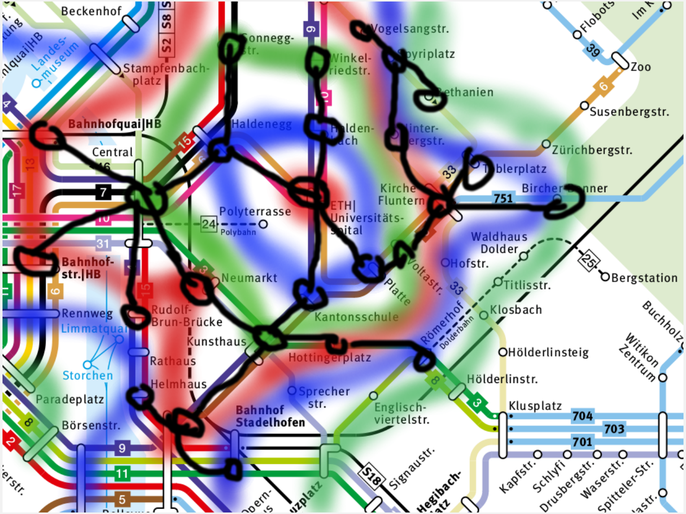
Depth First Search for all nodes of a graph. Running time:
$\Theta(|V| + \sum_{v \in V}(\deg^+(v) + 1)) = \Theta(|V| + |E|)$.

Interpretation of the Colors

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

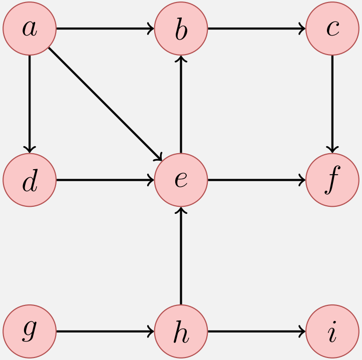
- White node: new tree edge
- Grey node: Zyklus (“back-egde”)
- Black node: forward- / cross edge

Breadth First Search



Graph Traversal: Breadth First Search

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

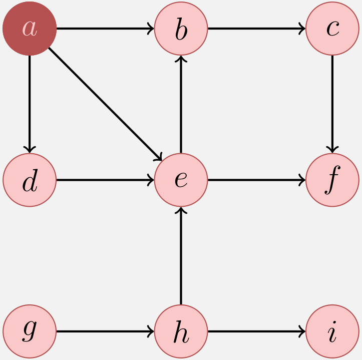


Adjazenzliste

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

Graph Traversal: Breadth First Search

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

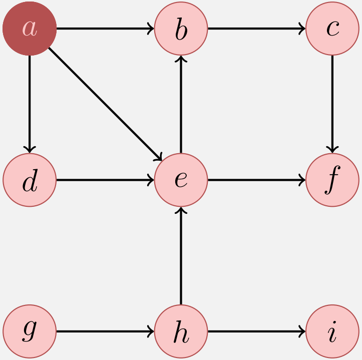


Adjazenzliste

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

Graph Traversal: Breadth First Search

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

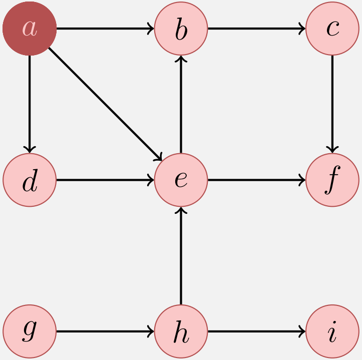


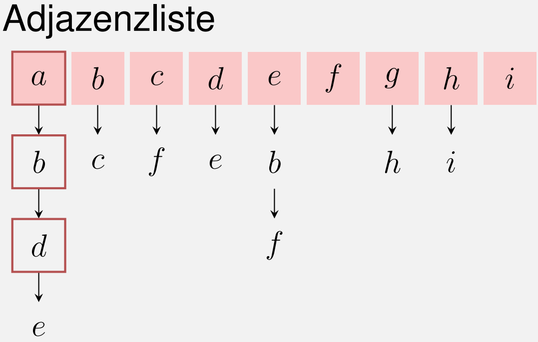
Adjazenzliste

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

Graph Traversal: Breadth First Search

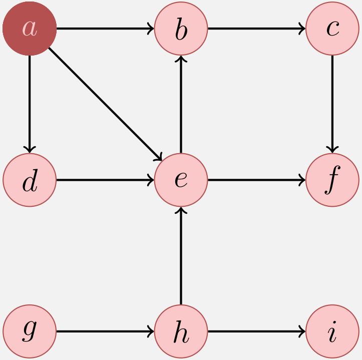
Follow the path in breadth and only then descend into depth.



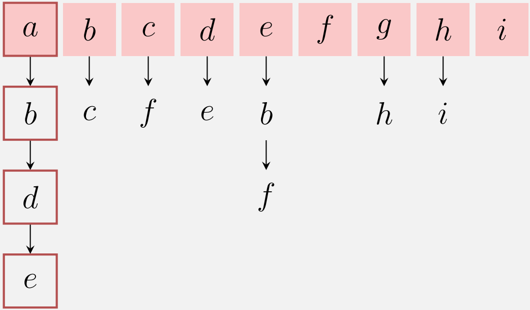


Graph Traversal: Breadth First Search

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

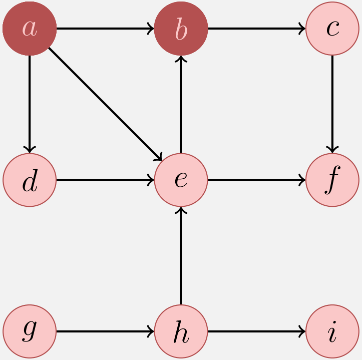


Adjazenzliste

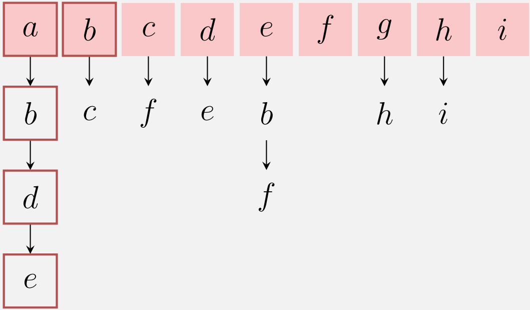


Graph Traversal: Breadth First Search

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

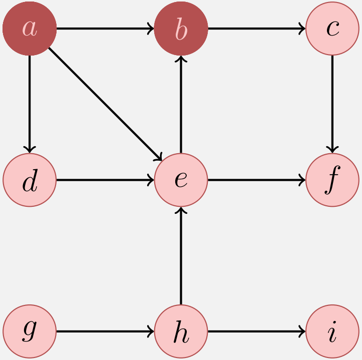


Adjazenzliste

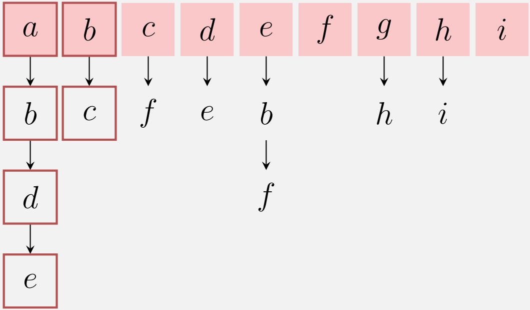


Graph Traversal: Breadth First Search

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

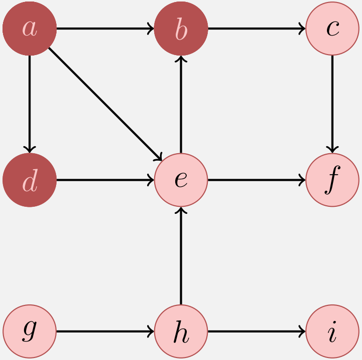


Adjazenzliste

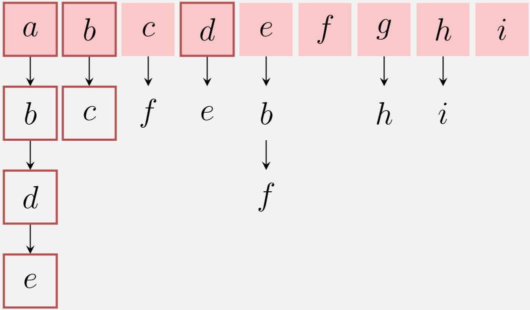


Graph Traversal: Breadth First Search

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

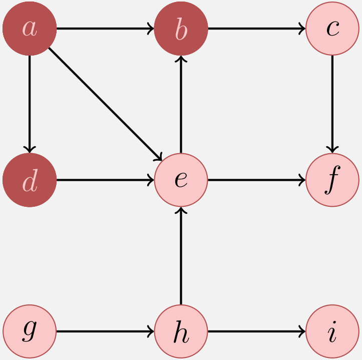


Adjazenzliste

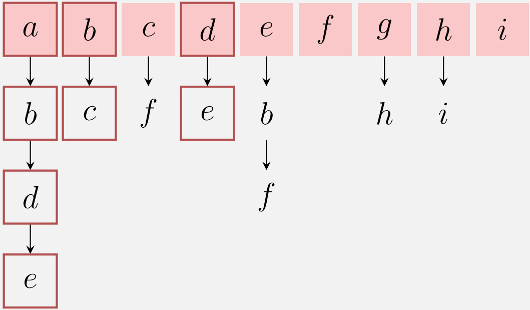


Graph Traversal: Breadth First Search

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

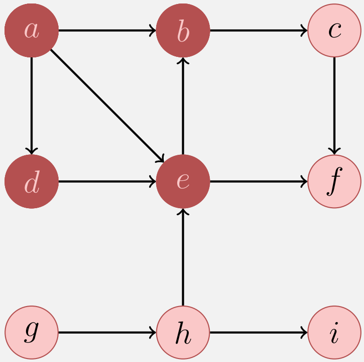


Adjazenzliste

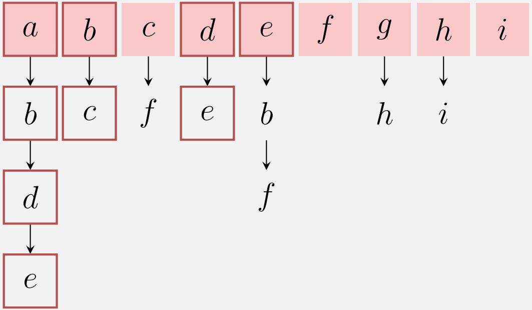


Graph Traversal: Breadth First Search

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

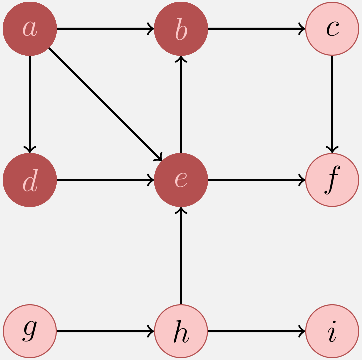


Adjazenzliste

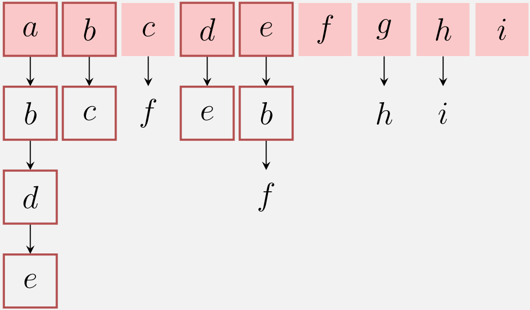


Graph Traversal: Breadth First Search

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

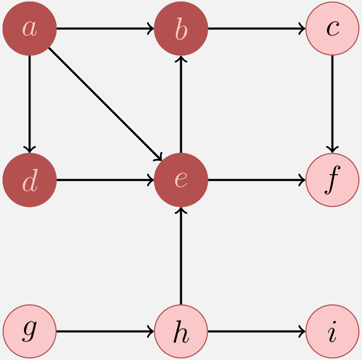


Adjazenzliste

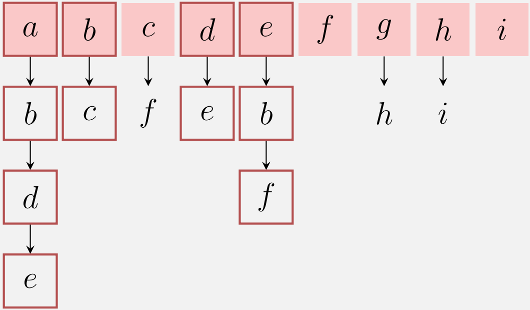


Graph Traversal: Breadth First Search

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

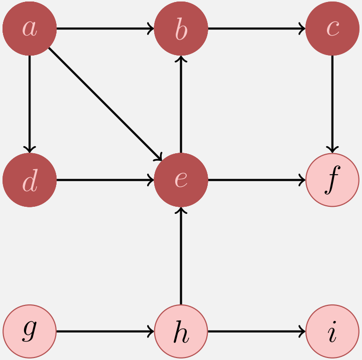


Adjazenzliste

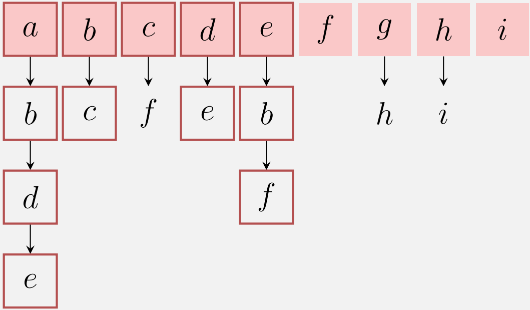


Graph Traversal: Breadth First Search

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

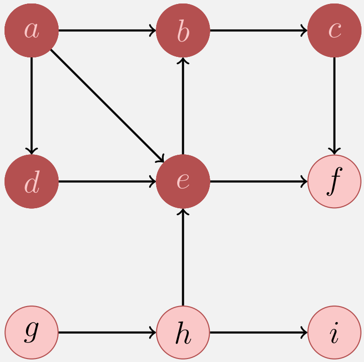


Adjazenzliste

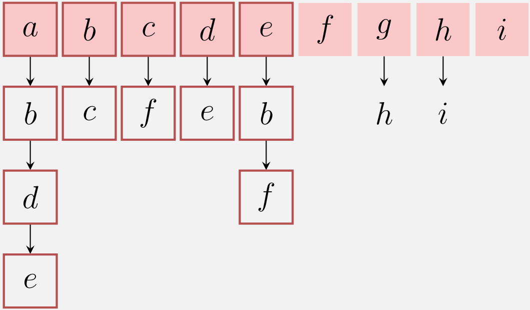


Graph Traversal: Breadth First Search

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

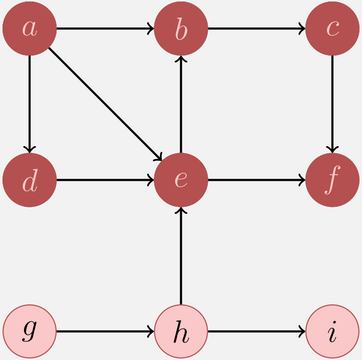


Adjazenzliste

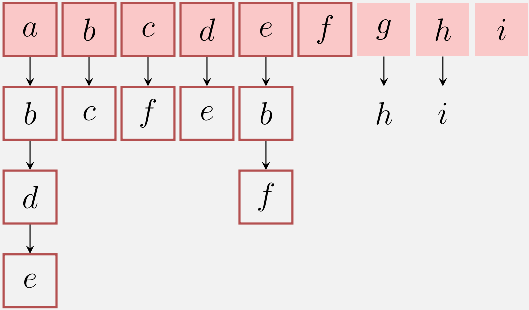


Graph Traversal: Breadth First Search

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

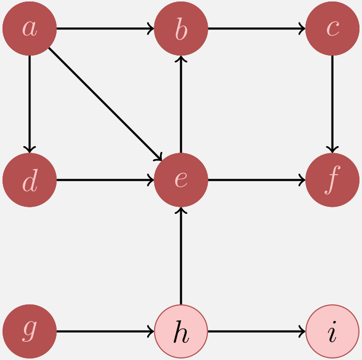


Adjazenzliste

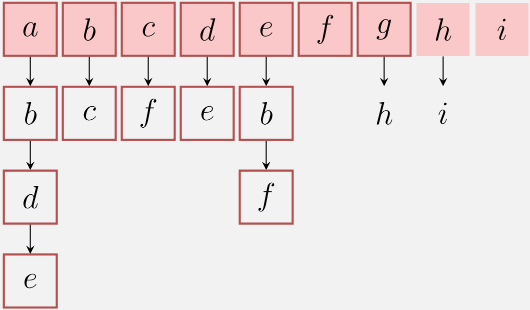


Graph Traversal: Breadth First Search

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

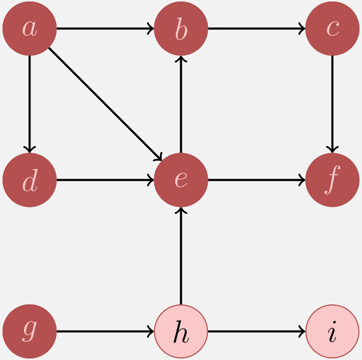


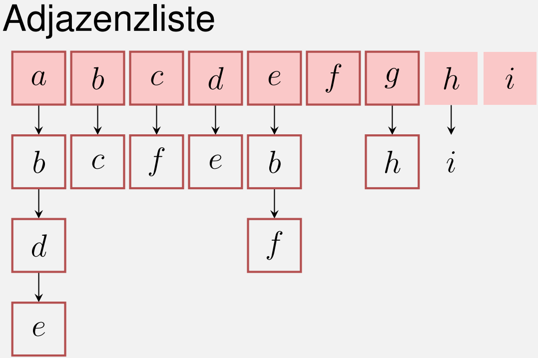
Adjazenzliste



Graph Traversal: Breadth First Search

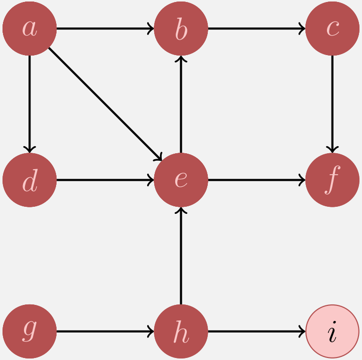
Follow the path in breadth and only then descend into depth.

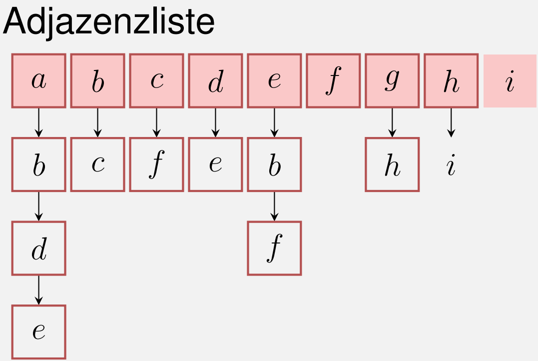




Graph Traversal: Breadth First Search

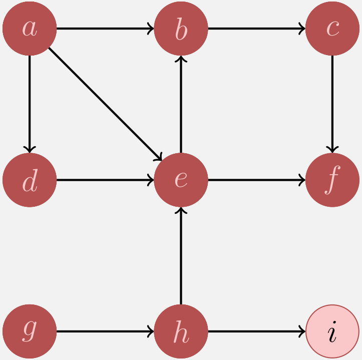
Follow the path in breadth and only then descend into depth.



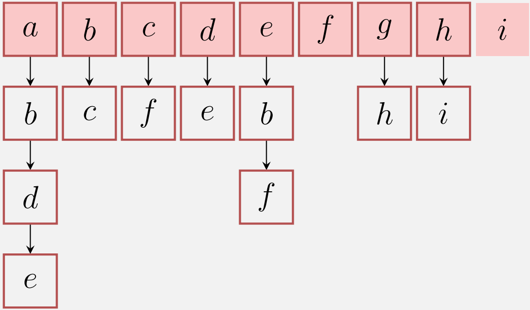


Graph Traversal: Breadth First Search

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

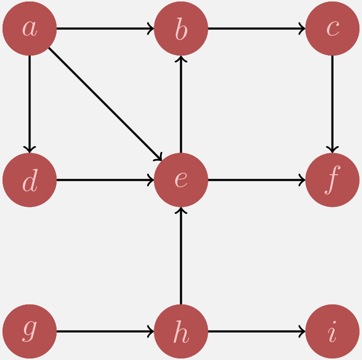


Adjazenzliste



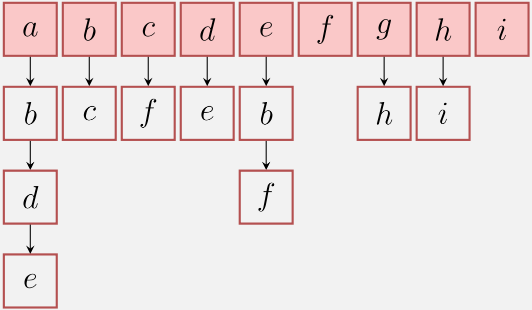
Graph Traversal: Breadth First Search

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



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

Adjazenzliste



(Iterative) BFS-Visit(G, v)

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

```
Queue Q ← ∅
v.color ← grey
enqueue(Q, v)
while Q ≠ ∅ do
    w ← dequeue(Q)
    foreach c ∈ N⁺(w) do
        if c.color = white then
            c.color ← grey
            enqueue(Q, c)
    w.color ← black
```

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

Main program BFS-Visit(G)

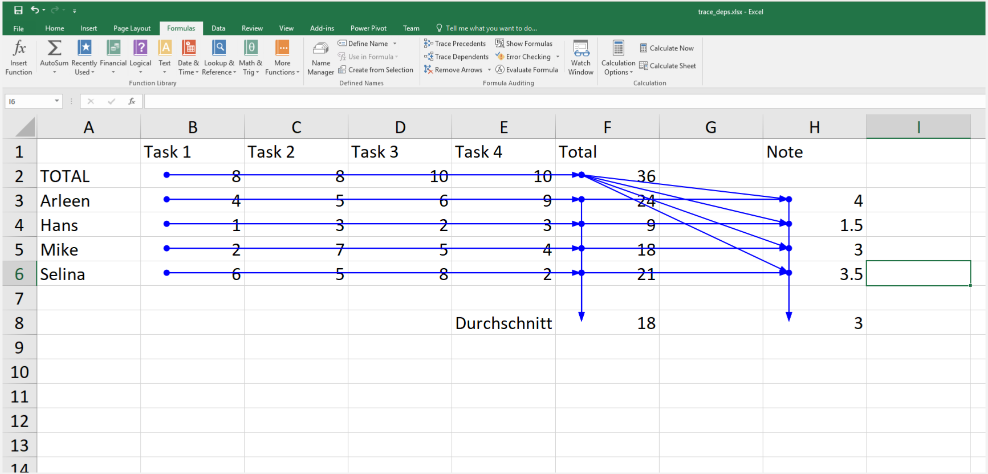
Input: graph $G = (V, E)$

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

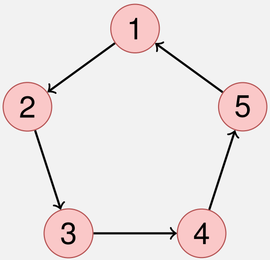
$$\mathrm{ord} : V \to \{1, \ldots, |V|\}$$

such that

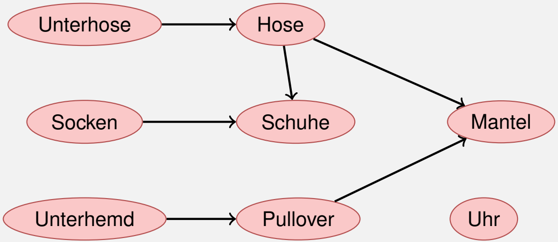
$$\mathrm{ord}(v) < \mathrm{ord}(w) \;\forall\; (v, w) \in E.$$

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

(Counter-)Examples



Cyclic graph: cannot be sorted topologically.



A possible toplogical sorting of the graph:
Unterhemd,Pullover,Unterhose,Uhr,Hose,Mantel,Socken,Schuhe

Observation

Theorem

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

Observation

Theorem

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

Proof "$\Rightarrow$": If G contains a cycle it cannot permit a topological sorting, because in a cycle $\langle v_{i_1}, \dots, v_{i_m} \rangle$ it would hold that $v_{i_1} < \cdots < v_{i_m} < v_{i_1}$.

Inductive Proof Opposite Direction

- Base case ($n = 1$): Graph with a single node without loop can be sorted topologically, set$\mathrm{ord}(v_1) = 1$.

Inductive Proof Opposite Direction

- Base case ($n = 1$): Graph with a single node without loop can be sorted topologically, set$\mathrm{ord}(v_1) = 1$.
- Hypothesis: Graph with n nodes can be sorted topologically

Inductive Proof Opposite Direction

- Base case ($n = 1$): Graph with a single node without loop can be sorted topologically, set$\mathrm{ord}(v_1) = 1$.
- Hypothesis: Graph with n nodes can be sorted topologically
- Step ($n \to n + 1$):

Inductive Proof Opposite Direction

- Base case ($n = 1$): Graph with a single node without loop can be sorted topologically, set $\mathrm{ord}(v_1) = 1$.
- Hypothesis: Graph with n nodes can be sorted topologically
- Step ($n \to n + 1$):

  1. G contains a node v_q with in-degree $\deg^-(v_q) = 0$. Otherwise iteratively follow edges backwards – after at most $n + 1$ iterations a node would be revisited. Contradiction to the cycle-freeness.

Inductive Proof Opposite Direction

- Base case ($n = 1$): Graph with a single node without loop can be sorted topologically, set$\mathrm{ord}(v_1) = 1$.
- Hypothesis: Graph with n nodes can be sorted topologically
- Step ($n \to n + 1$):
  1. G contains a node v_q with in-degree $\deg^-(v_q) = 0$. Otherwise iteratively follow edges backwards – after at most $n + 1$ iterations a node would be revisited. Contradiction to the cycle-freeness.
  2. Graph without node v_q and without its edges can be topologically sorted by the hypothesis. Now use this sorting and set $\mathrm{ord}(v_i) \leftarrow \mathrm{ord}(v_i) + 1$ for all $i \neq q$ and set $\mathrm{ord}(v_q) \leftarrow 1$.

Preliminary Sketch of an Algorithm

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

1. Traverse backwards starting from any node until a node v_q with in-degree 0 is found.

Worst case runtime:

Preliminary Sketch of an Algorithm

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

1. Traverse backwards starting from any node until a node v_q with in-degree 0 is found.
2. If no node with in-degree 0 found after n stepsm, then the graph has a cycle.

Worst case runtime:

Preliminary Sketch of an Algorithm

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

1. Traverse backwards starting from any node until a node v_q with in-degree 0 is found.
2. If no node with in-degree 0 found after n stepsm, then the graph has a cycle.
3. Set $\mathrm{ord}(v_q) \leftarrow d$.

Worst case runtime:

Preliminary Sketch of an Algorithm

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

1. Traverse backwards starting from any node until a node v_q with in-degree 0 is found.
2. If no node with in-degree 0 found after n stepsm, then the graph has a cycle.
3. Set $\operatorname{ord}(v_q) \leftarrow d$.
4. Remove v_q and his edges from G.

Worst case runtime:

Preliminary Sketch of an Algorithm

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

1. Traverse backwards starting from any node until a node v_q with in-degree 0 is found.
2. If no node with in-degree 0 found after n stepsm, then the graph has a cycle.
3. Set $\mathrm{ord}(v_q) \leftarrow d$.
4. Remove v_q and his edges from G.
5. If $V \neq \emptyset$, then $d \leftarrow d + 1$, go to step 1.

Worst case runtime:

Preliminary Sketch of an Algorithm

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

1. Traverse backwards starting from any node until a node v_q with in-degree 0 is found.
2. If no node with in-degree 0 found after n stepsm, then the graph has a cycle.
3. Set $\operatorname{ord}(v_q) \leftarrow d$.
4. Remove v_q and his edges from G.
5. If $V \neq \emptyset$, then $d \leftarrow d + 1$, go to step 1.

Worst case runtime:

Preliminary Sketch of an Algorithm

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

1. Traverse backwards starting from any node until a node v_q with in-degree 0 is found.
2. If no node with in-degree 0 found after n stepsm, then the graph has a cycle.
3. Set $\operatorname{ord}(v_q) \leftarrow d$.
4. Remove v_q and his edges from G.
5. If $V \neq \emptyset$, then $d \leftarrow d + 1$, go to step 1.

Worst case runtime: $\Theta(|V|^2)$.

Improvement

Idea?

Improvement

Idea?

Compute the in-degree of all nodes in advance and traverse the nodes with in-degree 0 while correcting the in-degrees of following nodes.

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

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

Algorithm Correctness

Theorem

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

Proof: follows from previous theorem:

1. Decreasing the in-degree corresponds with node removal.
2. In the algorithm it holds for each node v with $A[v] = 0$ that either the node has in-degree 0 or that previously all predecessors have been assigned a value $\mathrm{ord}[u] \leftarrow i$ and thus $\mathrm{ord}[v] > \mathrm{ord}[u]$ for all predecessors u of v. Nodes are put to the stack only once.
3. Runtime: inspection of the algorithm (with some arguments like with graph traversal)

Algorithm Correctness

Theorem

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

Algorithm Correctness

Theorem

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

Proof: let $\langle v_{i_1}, \dots, v_{i_k} \rangle$ be a cycle in G. In each step of the algorithm remains $A[v_{i_j}] \geq 1$ for all $j = 1, \dots, k$. Thus k nodes are never pushed on the stack und therefore at the end it holds that $i \leq V + 1 - k$.

The runtime of the second part of the algorithm can become shorter. But the computation of the in-degree costs already $\Theta(|V| + |E|)$.