Algorithms and Data Structures

Lecture 8

Graph Algorithms I:
BFS and DFS Traversal

Fabian Kuhn
Algorithms and Complexity

UNI
FREIBURG

Graphs

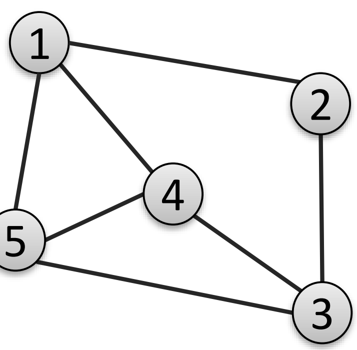
Node set V, typically $n \coloneqq |V|$ (nodes are also called **v**ertices)

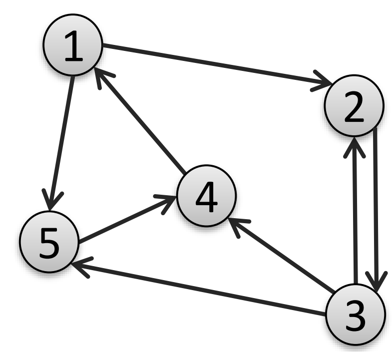
Edge set E, typically $m \coloneqq |E|$

- undirected graph: $E \subseteq \big\{\{u, v\} : u, v \in V\big\}$
- directed graph: $E \subseteq V \times V$

Examples:

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





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

Node Degrees

Graph $\boldsymbol{G = (V, E)}$ undirected:

- Degree of node $u \in V$: Number of edges (neighbors) of u

$$\deg(u) := |\{u, v\} : \{u, v\} \in E|$$

Graph $\boldsymbol{G = (V, E)}$ directed:

- In-degree of node $u \in V$: Number of incoming edges

$$\deg_{in}(u) := |(v, u) : (v, u) \in E|$$

- Out-degree of node $u \in V$: Number of outgoing edges

$$\deg_{out}(u) := |(u, v) : (u, v) \in E|$$

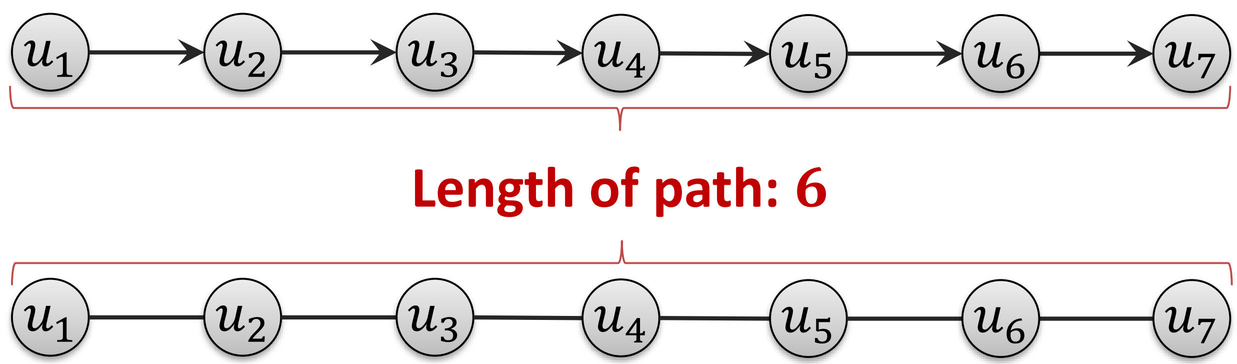
Paths

Paths in a graph $G = (V, E)$

- Path in G : a sequence $u_1, u_2, \ldots, u_k \in V$ with $u_i \neq u_j$ (if $i \neq j$)
  - directed graph: $(u_i, u_{i+1}) \in E$ for all $i \in \{1, \ldots, k-1\}$
  - undirected graph: $\{u_i, u_{i+1}\} \in E$ for all $i \in \{1, \ldots, k-1\}$

Path, directed:

Path, undirected:



Length of a path

- Number of edges on the path
- With edge weights: sum of all edge weights

Pfade

Paths in a graph $G = (V, E)$

- Path in G : a sequence $u_1, u_2, \dots, u_k \in V$ with $u_i \neq u_j$ (if $i \neq j$)
  - directed graph: $(u_i, u_{i+1}) \in E$ for all $i \in \{1, \dots, k-1\}$
  - undirected graph: $\{u_i, u_{i+1}\} \in E$ for all $i \in \{1, \dots, k-1\}$

Length of a path

- Number of edges on the path
- With edge weights: sum of all edge weights

Shortest path between nodes u and v

- Path $u, \dots, v$ of smallest length
- **Distance** $d_G(u, v)$: Length of a shortest path between u and v

Diameter $D := \max_{u,v \in V} d_G(u, v)$

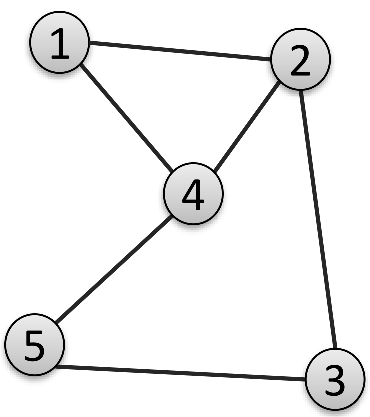
- Length of the longest shortest path

Representation of Graphs

Two classic methods to represent a graph in a computer

- **Adjacency matrix:** Space usage $\Theta(|V|^2) = \Theta(n^2)$
- **Adjacency lists:** Space usage $\Theta(|V| + |E|) = \Theta(n + m) = O(n^2)$

Example:



adjacency matrix

	1	2	3	4	5
1	0	**1**	0	**1**	0
2	**1**	0	**1**	**1**	0
3	0	**1**	0	0	**1**
4	**1**	**1**	0	0	**1**
5	0	0	**1**	**1**	0

adjacency lists

1 → 2 → 4 → None
2 → 1 → 4 → 3 → None
3 → 2 → 5 → None
4 → 5 → 2 → 1 → None
5 → 3 → 4 → None

Adjacency Matrix

Details:

- With **edge weights**, matrix entries are weightes (instead of $0/1$) (implicitly: weight 0 = edge does not exist)
- **Directed graphs:** one entry per directed edge
  - Edge from i to j: entry in row i and column j
- **Undirected graphs:** two entries per edge
  - Matrix in this case is symmetric

Properties Adjacency Matrix:

- Memory-efficient if $|E| = m \in \Theta(|V|^2) = \Theta(n^2)$
  - In particular for unweighted graphs: only one bit per matrix entry
- Not memory-efficient for sparse graphs ($m \in o(n^2)$)
- For certain algorithms, the “right” data structure
- “Edge between u and v” can be answered in time $O(1)$

Adjacency Lists

Structure

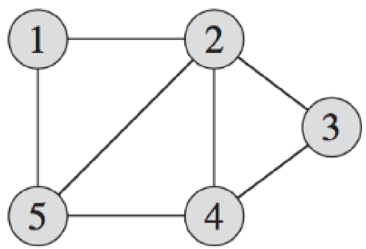
- An array with all the nodes
- Entries in this node array:
  - Linked lists with all edges of the corresponding nodes

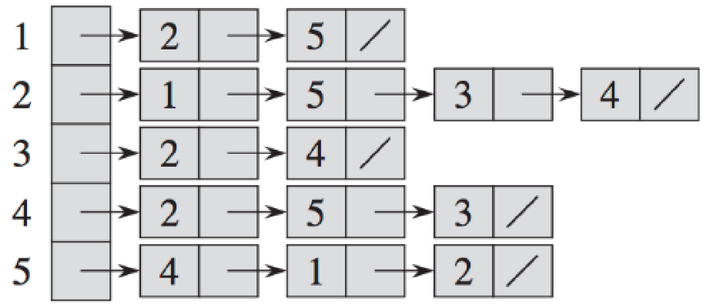
Properties

- Memory-efficient for sparse graphs
- Memory-usage always (almost) asymptotically optimal
  - but for dense graphs, still much worse…
  - To be precise: one actually requires $O(\log n)$ bits per node
- Queries for specific edges not very efficient
  - If necessary, one can use an additional data structure (e.g., a hash table)
- For many algorithms, the “right” data structure
- E.g., for depth first search and breadth first search

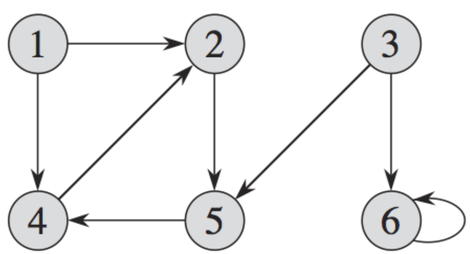
Examples

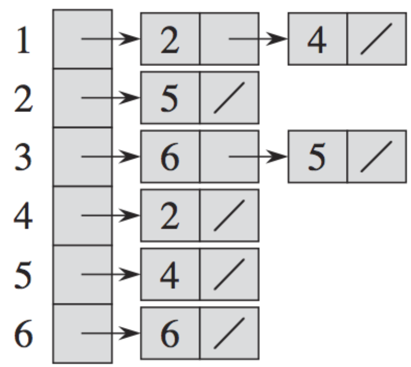
Examples from [CLRS]:





	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0





	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Graph Traversal

Graph Traversal (also: graph exploration) informally

- Given: a graph $G = (V, E)$ and a node $s \in V$, visit all nodes that are reachable from s in a "systematic" way.
- We have already seen this for binary trees.
- As for trees, there are two basic approaches

- **Breadth First Search (BFS)**
  - first "to the breadth" (nodes closer to s first)

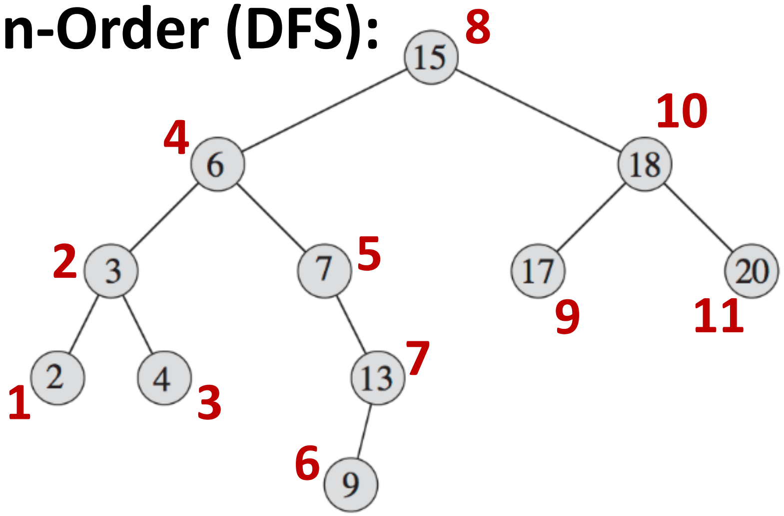
- **Depth First Search (DFS)**
  - first "to the depth" (visit everything that can be reached from some neighbor of the current node, before going to the next neighbor)

- Graph traversal is important as it is often used as a subroutine in other algorithms.
  - E.g., to compute the connected components of a graph
  - We will see a few examples…

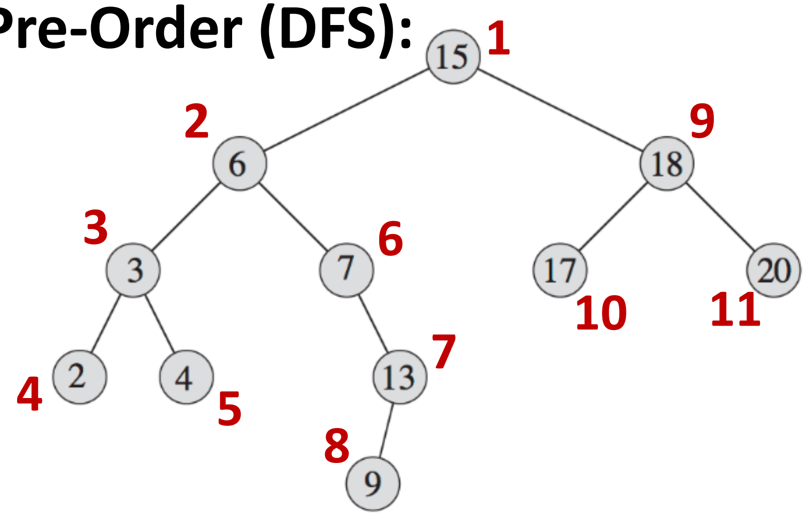
Traversal of a Binary Search Tree

Goal: visit all nodes of a binary search tree once

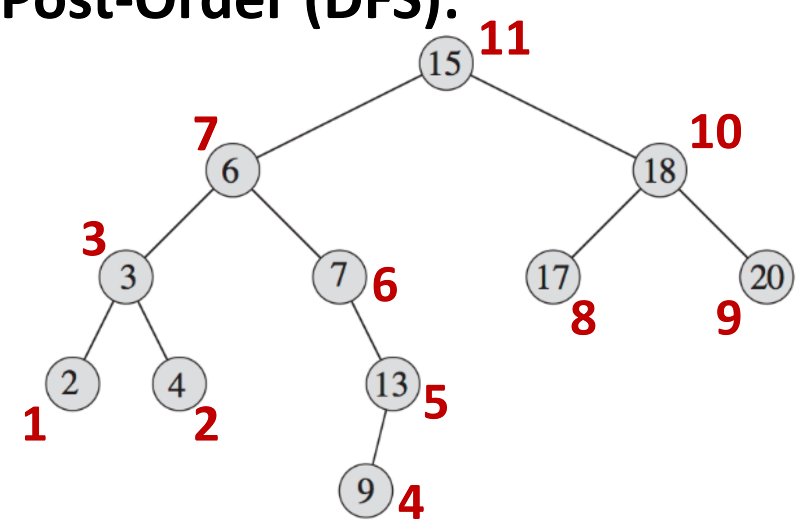
In-Order (DFS):



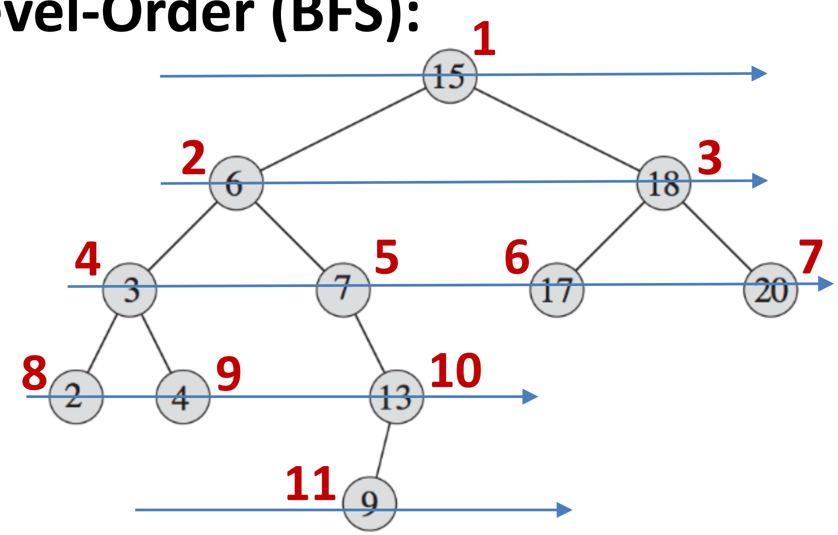
Pre-Order (DFS):



Post-Order (DFS):



Level-Order (BFS):



BFS Traversal

- Solution with a FIFO queue:
  - If a node is visited, its children are inserted into the queue.

```
BFS-Traversal:
    Q = new Queue()
    Q.enqueue(root)
    while not Q.empty() do
        node = Q.dequeue()
        visit(node)
        if node.left != null
            Q.enqueue(node.left)
        if node.right != null
            Q.enqueue(node.right)
```

BFS Traversal of General Graphs

Differences binary tree $T \Leftrightarrow$ general graph G

- Graph G can contain cycles
- In T, we have a root and every node know the direction towards the root.
  - Such trees are very often also called “rooted trees”

BFS Traversal in graph G (start at node $s \in V$)

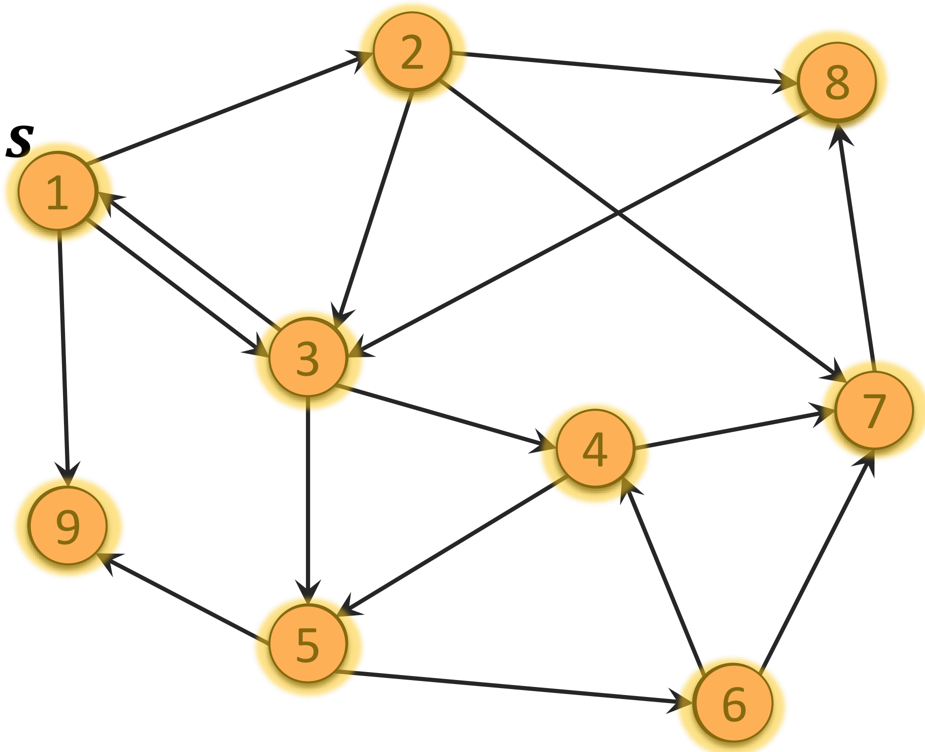
- **Cycles: mark** nodes that we have already seen
- **Mark** node s, then insert s into the queue
- As before, take first node u from the queue:
  - **visit node u**
  - Go through the neighbors v of u
    If v is not marked, mark v and insert v into the queue
    If v is marked, there is nothing to be done

BFS Traversal of General Graphs

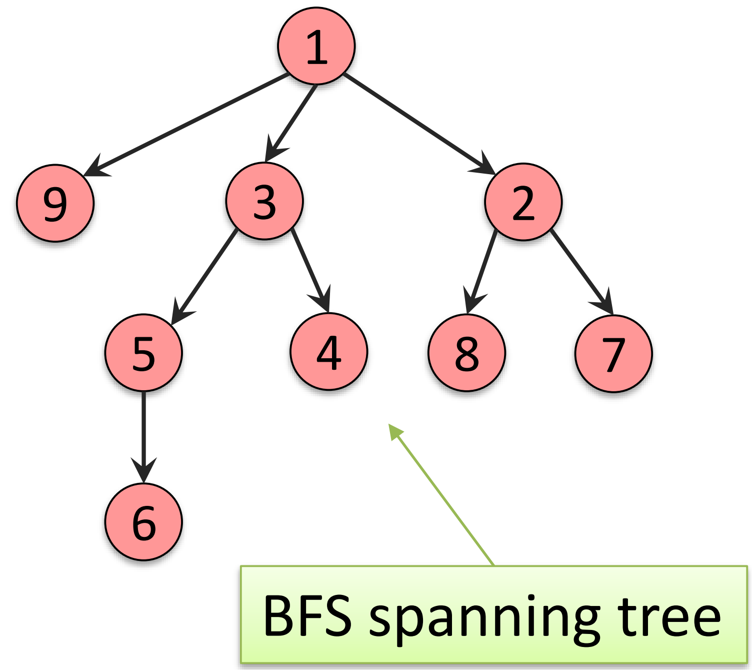
- At the beginning v. marked is set to **false** for all nodes v

```
BFS-Traversal(s):
    for all u in V: u.marked = false;
    Q = new Queue()
    s.marked = true
    Q.enqueue(s)
    while not Q.empty() do
        u = Q.dequeue()
        visit(u)
        for v in u.neighbors do
            if not v.marked then
                v.marked = true;
                Q.enqueue(v)
```

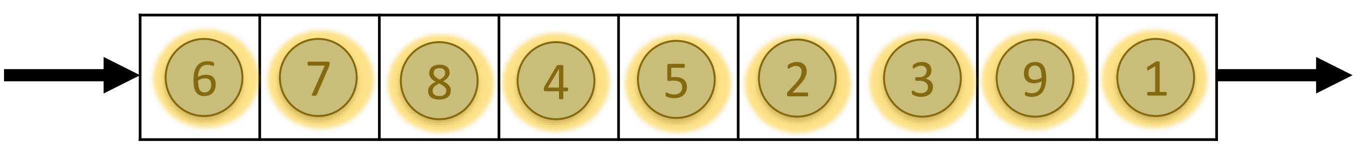
BFS Traversal Exmaple



Visiting Order:



FIFO Queue:



Analysis BFS Traversal

In the following, we label nodes as follows

- white nodes: Knoten, welche der Alg. noch nicht gesehen hat
- gray (before: blue) nodes: marked nodes
  - Nodes become gray when they are inserted into the queue.
  - Nodes are gray, as long as they are in the queue.
- black (before: red) nodes: visited nodes
  - Nodes become black when they are removed from the queue.

Analysis BFS Traversal

The running time of a BFS traversal is $\boldsymbol{O(n+m)}$.

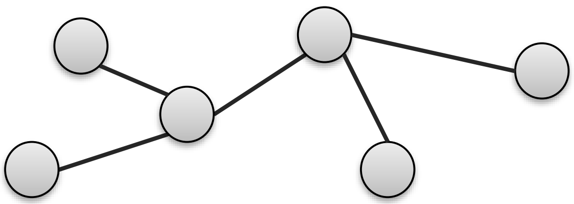
- **Assumption:** graph given as adjacency lists
  - If the graph is given by an adjacency matrix, the running time is $\Theta(n^2)$.
- white nodes: nodes that the algorithm has not seen yet
- gray (before: blue) nodes: marked nodes
- black (before: red) nodes: visited nodes

- Every node is inserted at most once into the queue.
  - In total, there are therefore $O(n)$ queue operations.
- If node v gets removed from the queue, the algorithm looks at all its nighbors.
  - Every directed edge is considered once.
  - Adjacency lists: time cost per node $= O(\#\text{neighbors})$
    - One has to go through the neighbor list once.
  - Adjacency matrix: time cost per node $= O(n)$
    - One has to go through a whole row of the matrix.

Trees, Spanning Trees

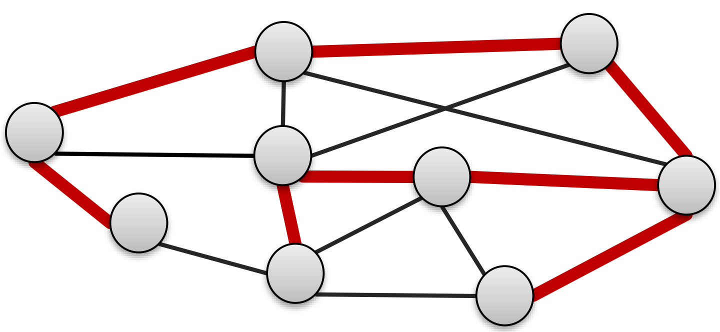
Tree:

- A connected, undirected graph without cycles
  - potentially also a directed graph, but then the graph must not have cycles, even when ignoring the directions.



Spanning Tree of a Graph G:

- A subgraph T such that T is a tree containing all nodes of G
  - Subgraph: Subset of the nodes and edges of G such that they together define a graph.

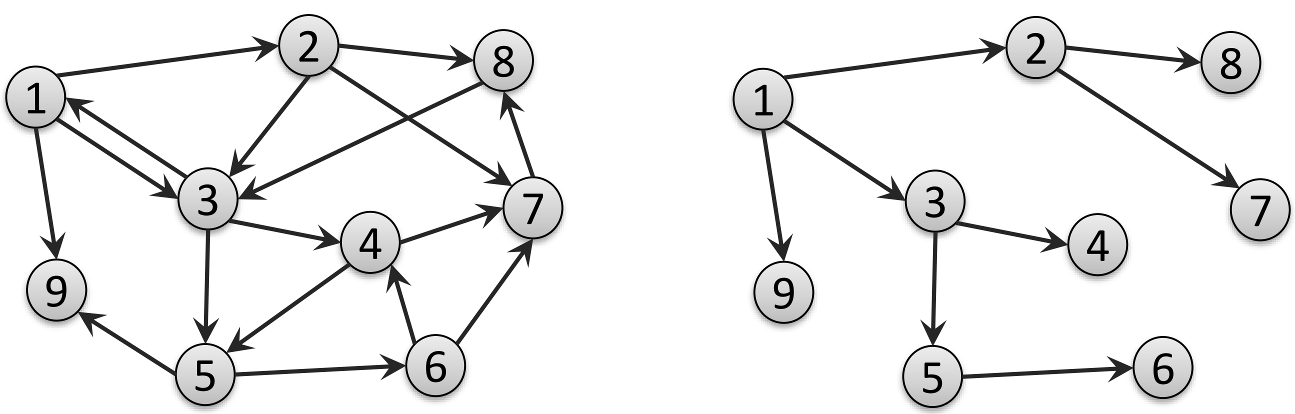


BFS Tree

If G is directed: all nodes are reachable from s

In a BFS traversal, we can construct a spanning tree as follows (if G is connected):

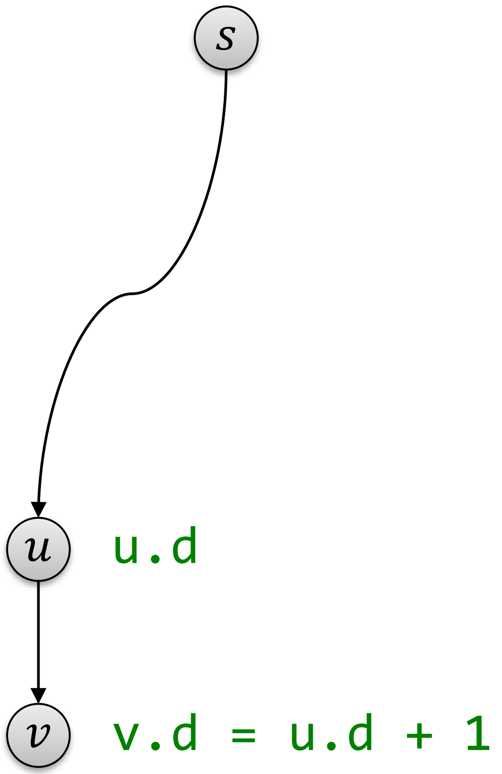
- Every node u stores, from which node v it was marked
- Node v then becomes the parent node of u
  - Because every node is marked exactly once, the parent of each node is defined in a unique way, s is the root and has no parent.



BFS Tree: Pseudocode

- We additionaly store the distance from s in the tree.

```
BFS-Tree(s):
    Q = new Queue();
    for all u in V: u.marked = false;
    s.marked = true;
    s.parent = NULL;
    s.d = 0
    Q.enqueue(s)
    while not Q.empty() do
        u = Q.dequeue()
        visit(u)
        for v in u.neighbors do
            if not v.marked then
                v.marked = true;
                v.parent = u;
                v.d = u.d + 1;
                Q.enqueue(v)
```



Analysis BFS Traversal

In the BFS tree of an unweighted graph, the distance from the root $\boldsymbol{s}$ to each node $\boldsymbol{u}$ is equal to $\boldsymbol{d_G(s,u)}$.

- Tree distance from the root: $d_T(s,u) = u.d$
- We therefore need to show that $u.d = d_G(s,u)$
- It definitely holds that $\boldsymbol{u.d \geq d_G(s,u)}$
  - Because $u.d = d_T(s,u)$, this is equivalent to $d_T(s,u) \geq d_G(s,u)$
  - This of course holde because every path in T is also a path in G, the distance in T can therefore not be smaller than the distance in G.

Analysis BFS Traversal

Lemma: Assume that during BFS traversal, the state of the queue is

$$Q = \langle v_1, v_2, \dots, v_r \rangle \;\; (v_1\text{: head}, v_r\text{: tail})$$

Then, $v_r.d \leq v_1.d + 1$ and $v_i.d \leq v_{i+1}.d$ (for $i = 1, \dots, r-1$)

Proof: By induction on the queue operations

- **Base:** At the beginning, only s with $s.d = 0$ is in the queue.
- **Step:**
  - dequeue operation: $Q = \langle v_1, v_2, \dots, v_r \rangle$, $\underbrace{v_r.d \leq v_1.d + 1 \leq v_2.d + 1}_{\text{induction hypothesis}}$
  - enqueue operation: $\boldsymbol{u}$, $\langle v_1, v_2, \dots, v_r \rangle$, $\boldsymbol{v}$

    most recently deleted node

    new node in the queue

    $u \rightarrow v$

    When v is inserted into the queue, the last removed node u is getting processed (v is a neighbor of u).

    $\Longrightarrow v.d = u.d + 1$
  - **From the induction hypothesis:**

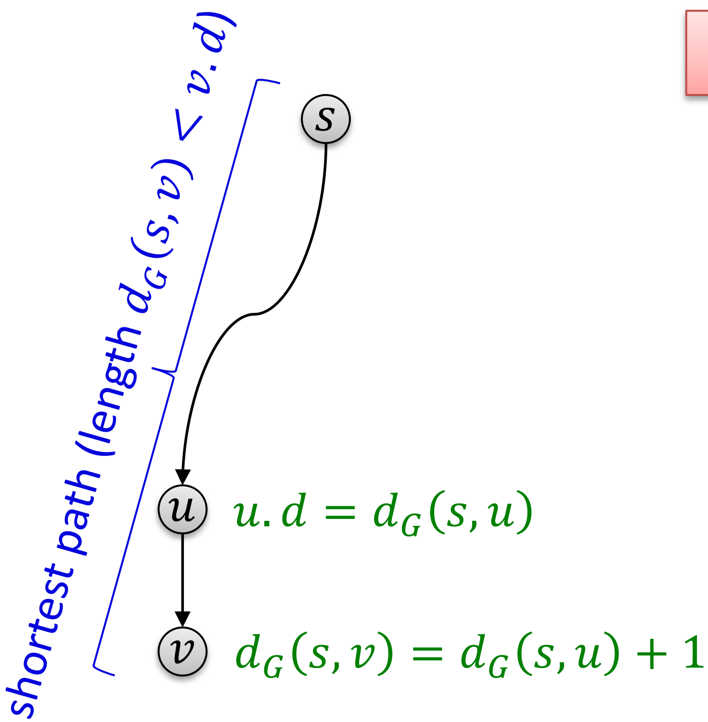
    $\boldsymbol{v.d \leq v_1.d + 1}$**:** $\quad v.d = u.d + 1 \leq v_1.d + 1$

    $\boldsymbol{v_r.d \leq v.d}$**:** $\quad v_r.d \leq u.d + 1 = v.d$

Analysis BFS Traversal

In the BFS tree of an unweighted graph, the distance from the root s to each node u is equal to $d_G(s,u)$.

- Proof by contradiction:
  - Assumption: v is node with smallest $d_G(s,v)$ for which $\boldsymbol{v.d > d_G(s,v)}$



$$\boldsymbol{v.d} > d_G(s,v) = d_G(s,u) + 1 = \boldsymbol{u.d + 1}$$

Consider dequeue of u:

- v will be considered as neighbor of u
- v is white $\Longrightarrow v.d = u.d + 1$
- v is black $\Longrightarrow v.d \leq u.d$
- v is gray $\Longrightarrow v$ is in the queue
  Lemma: $v.d \leq u.d + 1$

Depth-First Search in General Graphs

Basic idea DFS traversal in G (start at node $s \in V$)

- **Mark node v** (at the beginning $v = s$)
- Visit the neighbors of v one after the other *recursively*
- After all neighbors are visited, **visit v**

- **Recursively:** While visiting the neighbors, visit their neighbors and while doing this their neighbors, etc.
- **Cycles in G:** Only visit nodes that have not been marked

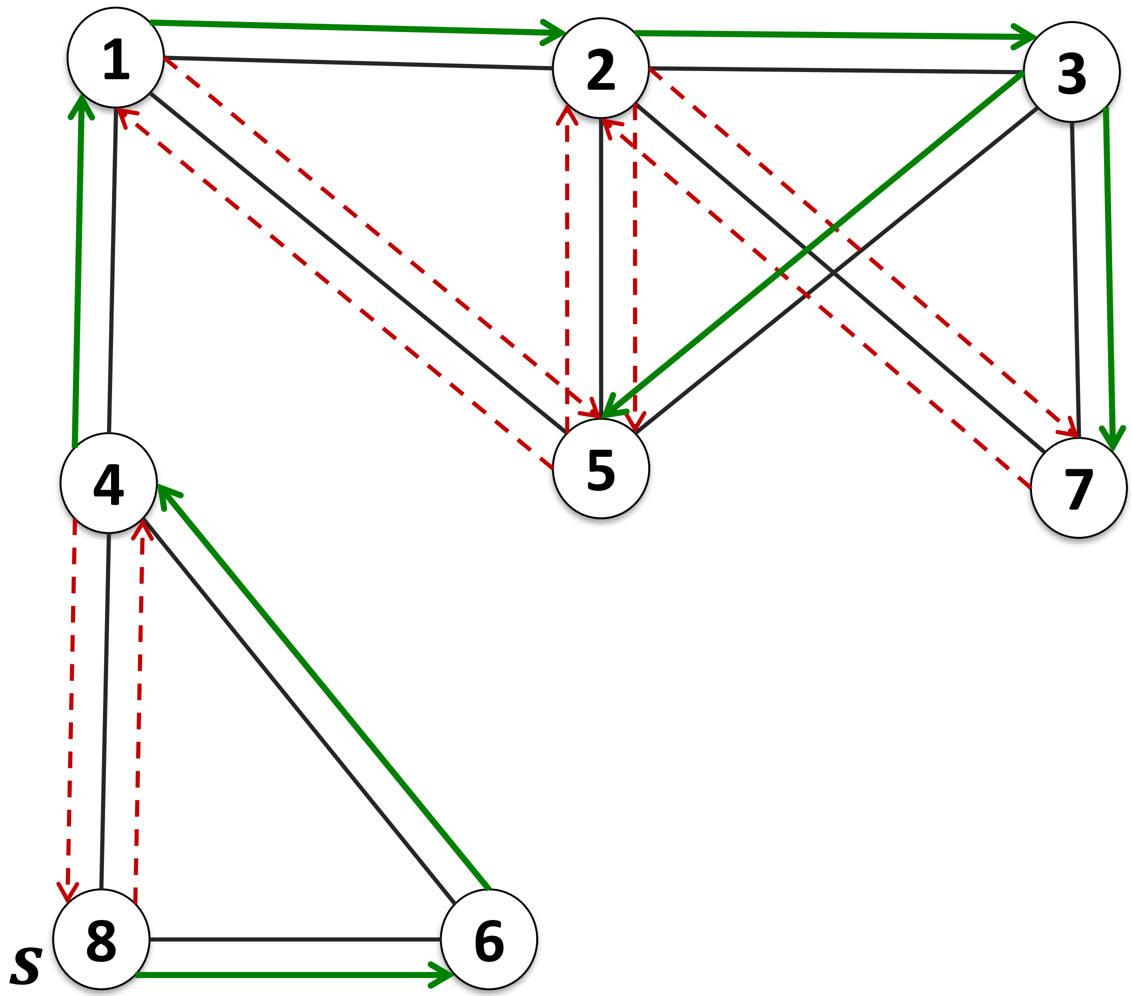
- Corresponds to the post-order traversal in trees.
- The order in which the nodes are marked corresponds to the pre-order traversal in trees.

DFS Traversal: Pseusocode

```
DFS-Traversal(s):
    for all u in V: u.color = white;
    DFS-visit(s, NULL)

DFS-visit(u, p):
    u.color = gray;
    u.parent = p;
    for all v in u.neighbors do
      if v.color = white
        DFS-visit(v, u)
    visit node u;
    u.color = black;
```

DFS Traversal: Example



DFS Traversal: Analysis

In the same way as for a BFS traversal, one can also construct a spanning tree when doing a DFS traversal.

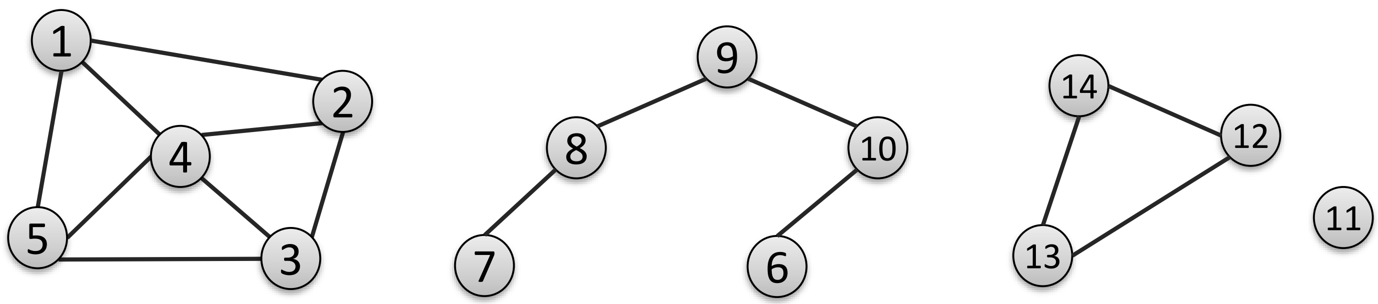
The running time of a DFS traversal is $\boldsymbol{O(n+m)}$.

- We color the nodes white, gray, and black as before
  - not marked = white, marked = gray, visited = schwarz

- The recursive DFS traversal function is called at most once for every node.
- The time to process a node v is proportional to the number of (outgoing) edges of v

Connected Components

- The connected components (or simply components) of a graph are its connected parts.



Goal: Find all components of a graph.

```
for u in V do
    if not u.marked then
        start new component
        explore with DFS/BFS starting at u
```

- The connected components of a graph can be identified in time $O(n + m)$. (by using a DFS or a BFS traversal)

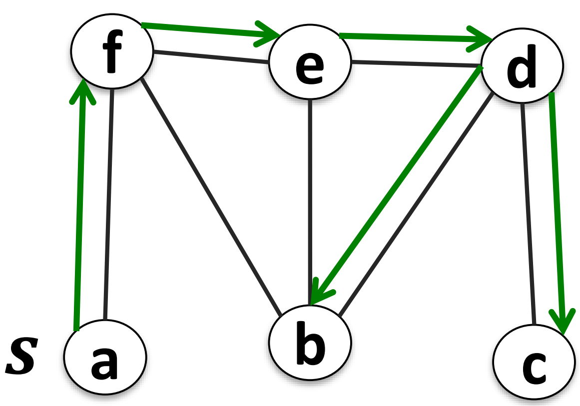
DFS "Parenthesis" Theorem

We define the following two times for each node v

- $t_{v,1}$: time, when v is colored gray in the DFS traversal
- $t_{v,2}$: time, when v is colored black in the DFS traversal

Theorem: In the DFS tree, a node v is in the subtree of a node u, if and only if the interval $[t_{v,1}, t_{v,2}]$ is completely contained in the interval $[t_{u,1}, t_{u,2}]$.

Example:



$t_{a,1}\ t_{f,1}\ t_{e,1}\ t_{d,1}\ t_{c,1}\ t_{c,2}\ t_{b,1}\ t_{b,2}\ t_{d,2}\ t_{e,2}\ t_{f,2}\ t_{a,2}$

[[[[[ ] [ ]]]]]

DFS "Parenthesis" Theorem

UNI FREIBURG

Theorem: In the DFS tree, a node v is in the subtree of a node u, if and only if the interval $[t_{v,1}, t_{v,2}]$ is completely contained in the interval $[t_{u,1}, t_{u,2}]$.

Why is this useful?

- Improves our understanding of the structure of the resulting DFS tree
- We need the theorem, e.g., to prove the correctness of the algorithm for computing a topological sort.

DFS “Parenthesis” Theorem

Theorem: In the DFS tree, a node v is in the subtree of a node u, if and only if the interval $[t_{v,1}, t_{v,2}]$ is completely contained in the interval $[t_{u,1}, t_{u,2}]$.

Proof:

- Gray nodes always form a path that starts at node s.
  - Path starts at s, currently active node at the end of the path
  - New node w becomes gray $\Longrightarrow$ w neighbor of active node
  - Node becomes black $\Longrightarrow$ active node ends recursion

$$s \to b \to c \to d \text{ - - - } \cdots \text{ - - - } \to u \to w$$

- Node v is in the subtree of u u if and only if u is part of the path, when v becomes gray and thus iff $t_{u,1} < t_{v,1} < t_{u,2}$.
- NOde v in this case is further to the end in the path than u and has to become black before node u, hence $t_{v,2} < t_{u,2}$

DFS "Parenthesis" Theorem

Theorem: In the DFS tree, a node v is in the subtree of a node u, if and only if the interval $[t_{v,1}, t_{v,2}]$ is completely contained in the interval $[t_{u,1}, t_{u,2}]$.

Implications

- Two intervals are always either disjoint or one of the intervals is contained in the other.
- Why "Parenthesis" Theorem?
  If for each $t_{v,1}$ we write an open parenthesis and for each $t_{v,2}$ we write a close parenthesis, one gets an expression in which the parentheses are nested properly.
- A white node v, which is discovered in the recursive traversal started at u becomes black before the recursion gets back to u.

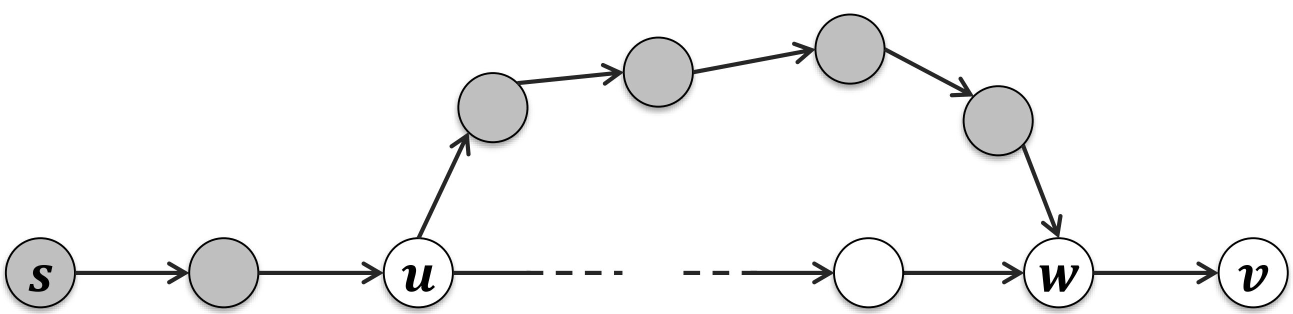
White Paths

Theorem: In a DFS tree, a node v is in the subtree of a node u, if and only if immediately before marking node u, a completely white path from u to v exists.

at time $t_{u,1}$

Proof:

- **Proof by contradiction:** Assume that there is a node v to which there is a white path, but that node v is not in the subtree of u.
- Assume that v is such a node with the additional property that immediately before marking u, v has the shortest white path from u among all such nodes.



Classification of Edges (in DFS)

Tree Edges:

- (u, v) is a tree edge, if node v is discovered from node u
  - When considering (u, v), v is white

Backward Edges:

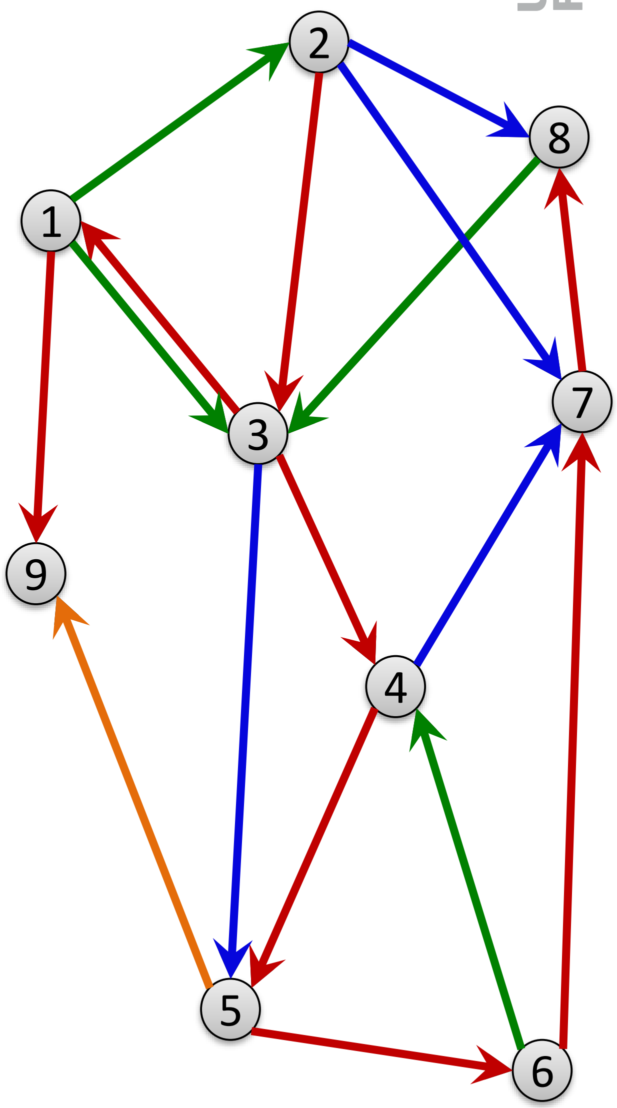
- (u, v) is a backward edge if v is a predecessor node of u
  - When considering (u, v), v is gray

Forward Edges:

- (u, v) is a forward edges if v is a successor node of u
  - When considering (u, v), v is black

Cross Edges:

- All other edges
  - When considering (u, v), v is black



Classification of Edges (in DFS)

Tree Edge (u, v):

- Node v is “discovered” as white neighbor of u
  - If when considering (u, v), **v is white** $\Longrightarrow$ (u, v) **tree edge**

Backward Edge (u, v):

- Subtree of u will be completely visited, before v becomes black
  - If when considering (u, v) **v is gray** $\Longrightarrow$ (u, v) **backward edge**

Forward Edge (u, v):

$$t_{u,1} < t_{v,1} < t_{v,2} < t_{u,2}$$

- v is in a subtree of u that has already been visited completely
  - Since v is in a subtree of u, **v is schwarz and $t_{v,1} > t_{u,1}$**

Cross Edge (u, v):

$$t_{v,1} < t_{v,2} < t_{u,1} < t_{u,2}$$

- As long as u is gray, all newly visited nodes are in the subtree of u, v was therefore visited before u: **v is black and $t_{v,1} < t_{u,1}$**.

DFS – Undirected Graphs

- In undirected graphs, every edge $\{u, v\}$ is considered twice (once from u and once from v)
- We classify the edge according to the first consideration.

Theorem: In a DFS traversal in an undirected graph, every edge is either a tree edge or a backward edge.

Proof:

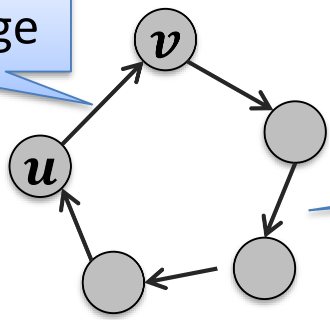
- W.l.o.g., we assume that u becomes gray before v becomes gray.
- From the theorem about white paths, we know that v is visited as long as u is still gray (v is in the subtree of u).
- If the edge $\{u, v\}$ is first considered from u, node v is still white $\Longrightarrow \{u, v\}$ is a tree edge.
- If the edge $\{u, v\}$ is first considered from v, node u is still gray $\Longrightarrow \{u, v\}$ is a backward edge.

DFS – Directed Graphs

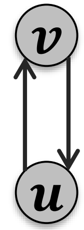
Theorem: A directed graph has no cycles if and only if during a DFS traversal, there are no backward edges.

backward edge $\Longrightarrow$ cycle:

backward edge



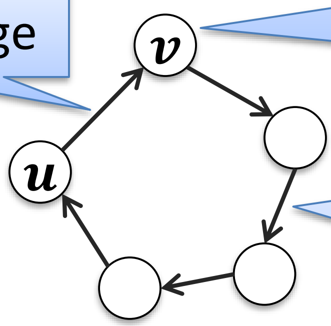
u must be in the subtree of v.



u can also be a child of v.

cycle $\Longrightarrow$ backward edge:

backward edge



assumption: v is the first considered node in the cycle .

white path: u is in the subtree of v.

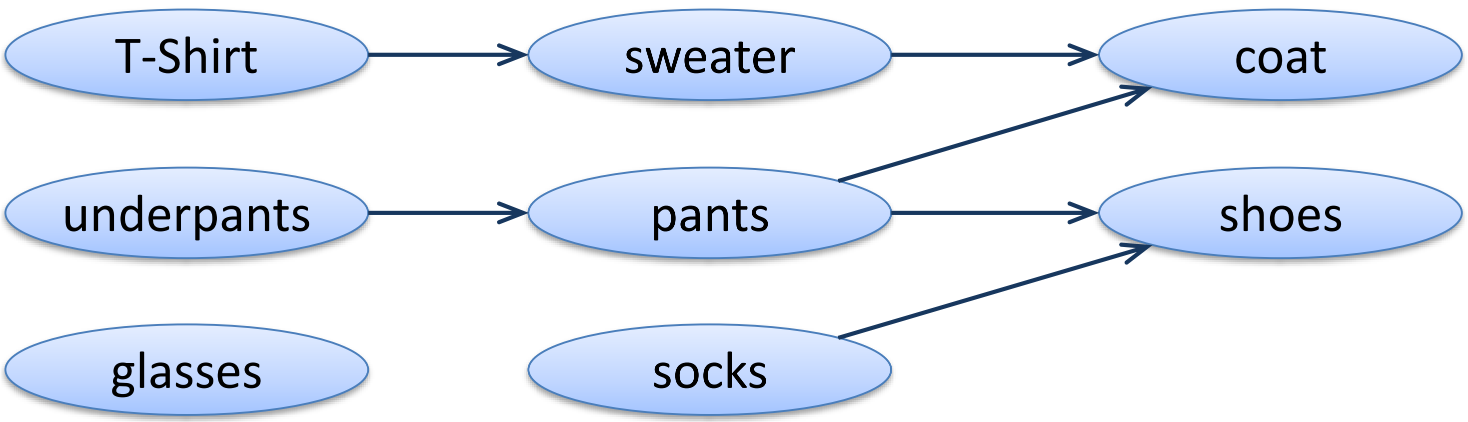
Implication:

In directed graphs, one can test in time $O(n+m)$ if the graph is acyclic.

Application: Topological Sort

Directed Acyclic Graphs:

- **DAG**: directed acyclic graph
- E.g., models time dependencies between tasks
- Example: putting on pieces of clothes



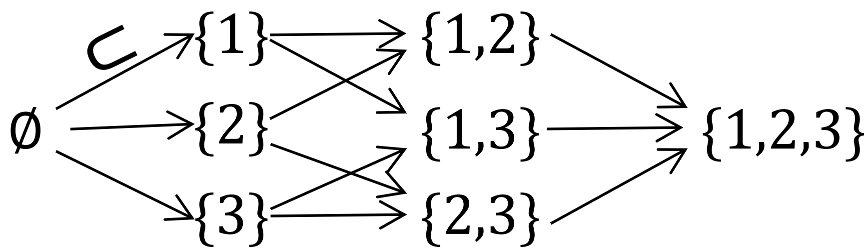
Topological sort:

- Sort the nodes of a DAG in such a way that u appears before v if a directed path from u to v exists.
- In the example: Find a possible dressing order

Topological Sort: A bit more formally…

Directed Acyclic Graphs:

- represent partial orders
  - asymmetric: $a < b \implies \neg(b < a)$
    transitive: $a < b \,\wedge\, b < c \implies a < c$
  - partial order: not all pairs need to be comparable
- Example: subset relation for sets



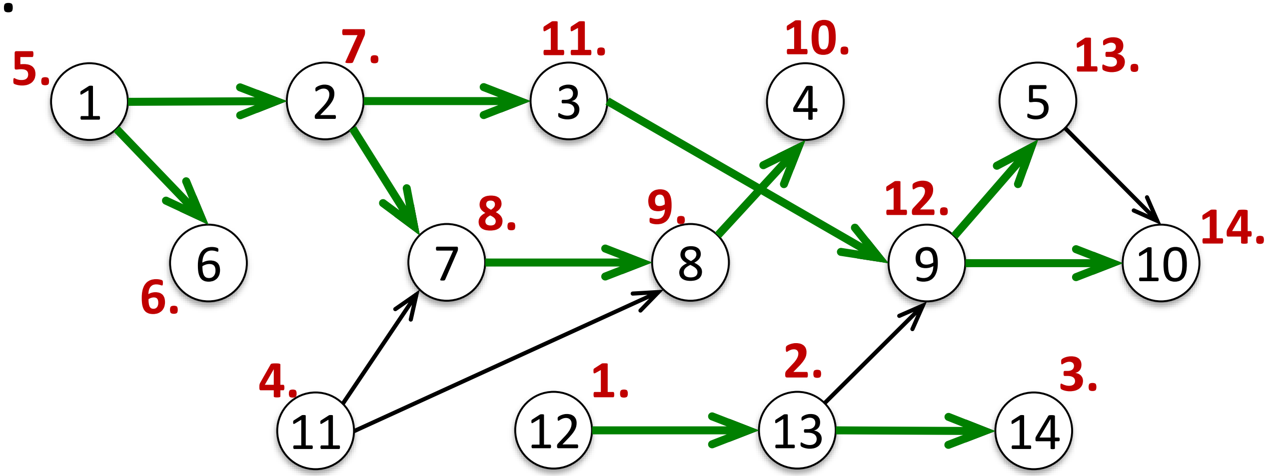
Topological Sort:

- Sort the nodes of a DAG in such a way that u appears before v if a directed path from u to v exists.
- Extend a partial order to a total order:

$$\emptyset, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}$$

Topological Sort: Algorithm

Do a DFS …



$t_{1,1}$ $t_{2,1}$ $t_{3,1}$ $t_{9,1}$ $t_{10,1}$ $t_{10,2}$ $t_{5,1}$ $t_{5,2}$ $t_{9,2}$ $t_{3,2}$ $t_{7,1}$ $t_{8,1}$ $t_{4,1}$ $t_{4,2}$ $t_{8,2}$

$t_{7,2}$ $t_{2,2}$ $t_{6,1}$ $t_{6,2}$ $t_{1,2}$ $t_{11,1}$ $t_{11,2}$ $t_{12,1}$ $t_{13,1}$ $t_{14,1}$ $t_{14,2}$ $t_{13,2}$ $t_{12,2}$

Observation:

- Nodes without successor are visited first (colored black)
- Visiting order is a reverse topological sort order

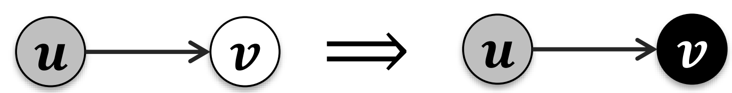
Topological Sort: Algorithm

Theorem: The reversed "visit" order (coloring black) of the nodes in a DFS traversal gives a topological sort of a directed acyclic graph.

Proof:

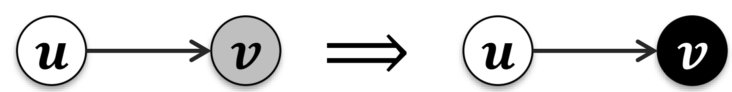
- We must show that for every edge (u, v), node v becomes black before node u.

- **Case 1: u becomes gray before v:**



  - Then, v is in the subtree of u and therefore $t_{u,1} < t_{v,1} < t_{u,2}$.
    From the parenthesis theorem, we then also get $t_{v,2} < t_{u,2}$.

- **Case 2: v becomes gray before u:**

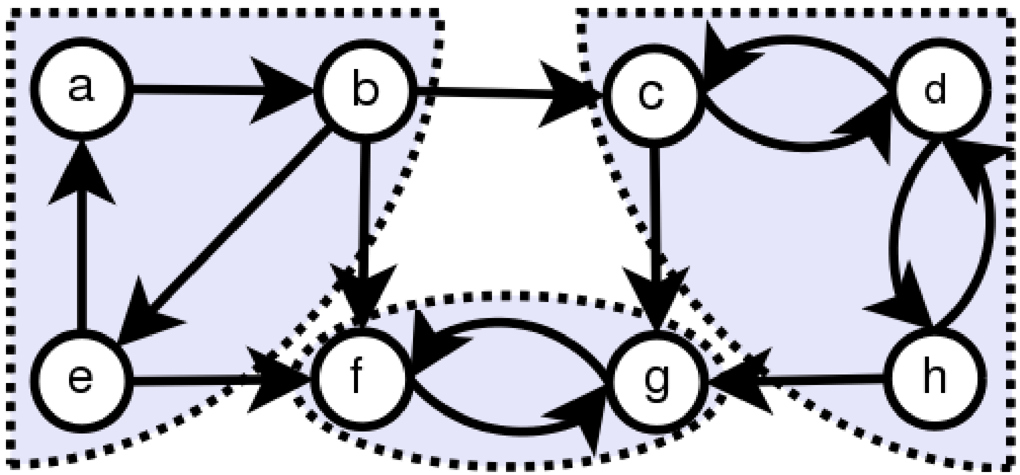


  - u can only become gray before v becomes black, if u is in the subtree of v.
    Then we would have a directed path from v to $u \Longrightarrow$ cycle!

DFS Traversal: Further Application

Strongly Connected Components

- Strongly connected component of a directed graph:
  "Maximal subset of nodes s. t. every node can reach every other node"



Picture: Wikipedia

- Requires 2 DFS traversals (time $= O(m+n)$)
  - on G and on G^T (all edges reversed)
  - G and G^T have the same strongly connected components
- Details, e.g., in [CLRS]