COM1009
Introduction to Algorithms and Data Structures

Topic 11: Elementary Graph Algorithms

Essential Reading: Chapter 22

Aims for this lecture

- To discuss **breadth-first** and **depth-first** search and trees.
- To show how depth-first search (DFS) can **classify edges** for additional information about graphs. We can use DFS to
  - Check whether a graph contains cycles
  - Put tasks in the right order (topological sorting)
  - Compute strongly connected components in graphs
- To show the **correctness** of some remarkable algorithms.

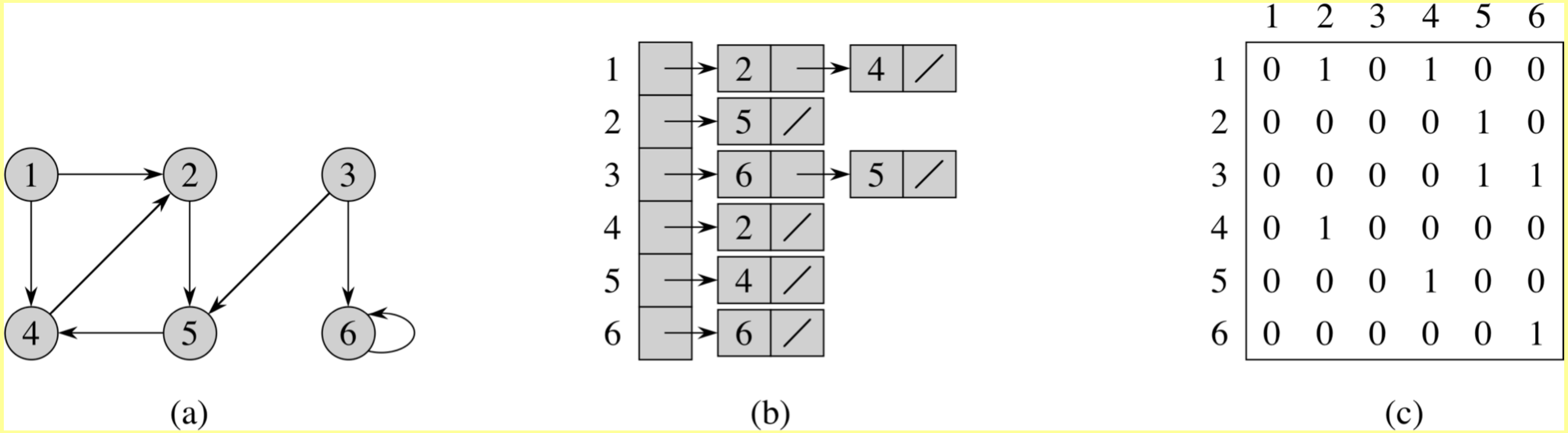
Graphs

- In this context a graph G is a collection of vertices (V) connected by edges (E).
  - Directed: edges allow travel in one direction only
  - Undirected: edges allow travel in both directions
- Formally:
  - $G = (V, E)$ where $\mathrm{E} \subseteq V^2$
  - $(u, v) \in E$ means there's an edge from u to v

Representations of graphs

- Adjacency-relation (edge-relation)
  - Since E is a set of pairs, it is also a *relation* on V.
  - We say that v is adjacent to u if $(u, v) \in E$, i.e. we can travel directly from u to v using exactly one edge.
- Adjacency-list representation
  - An array Adj of lists. The list Adj[u] contains all vertices v adjacent to u in G, i.e. there is an edge from u to v.
- Adjacency matrix representation
  - A matrix where a_{ij}=1 if (i, j) $\in E$ and a_{ij}=0 otherwise.

Example: a directed graph



(a) A directed graph

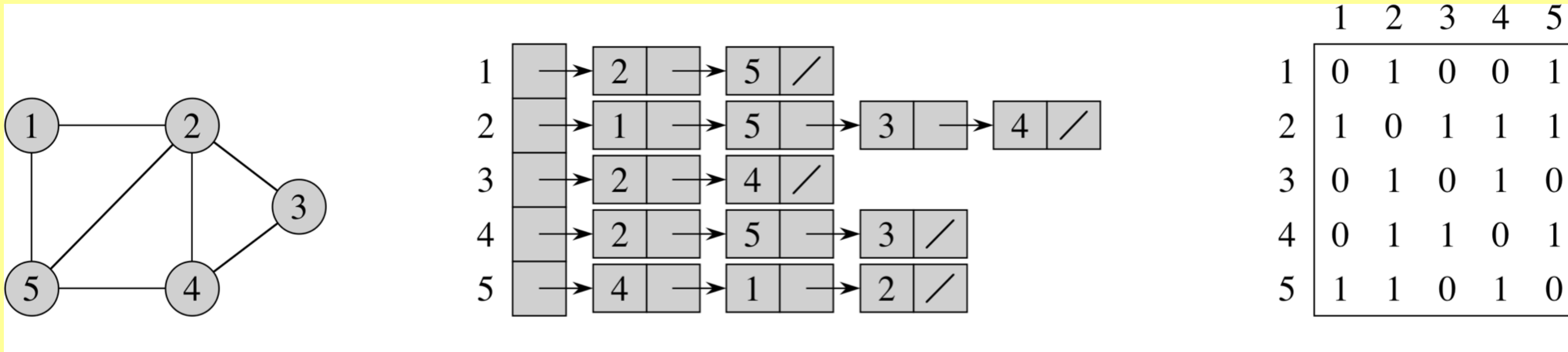
- the numbering of the nodes is arbitrary

(b) Its adjacency list (shown here as an array of linked lists)

- 1 is adjacent to both 2 and 4; 2 is adjacent to 5; and so on ...

(c) Its adjacency matrix

Example: an undirected graph



(a) (b) (c)

- In this case, the adjacency relation is symmetric:
  - u is adjacent to v if and only if v is adjacent to u
  - the adjacency matrix is symmetrical about the main diagonal
  - we only need to store the entries on and above the diagonal.

Adjacency lists vs. adjacency matrix

- The list representation has |V| separate lists, containing at most 2|E| entries in all (one or two for each edge). The matrix has $|V|^2$ entries. So input sizes for algorithms are:
  - $\theta(|V|+|E|)$ for adjacency lists
  - $\theta(|V|^2)$ for adjacency matrices
- Adjacency lists are preferable for **sparse** graphs. A graph is **sparse** if $|E| = o(|V|^2)$ and **dense** if $|E| = \theta(|V|^2)$.
- Testing whether *u* and *v* are adjacent takes time O(1) in an adjacency matrix and can take time $\Omega(|V|)$ with adjacency lists.

Breadth-first search (BFS)

- One of the simplest algorithms for searching graphs.
- Given a graph G = (V, E) and a distinguished **source s**, BFS computes the distance from s to each reachable vertex.
- It also produces a **breadth-first tree** with root s that contains all reachable vertices: the simple path in the breadth-first tree from s to v corresponds to a shortest path from s to v (shortest = smallest number of edges).
- Other problems (e.g., finding shortest paths) use similar ideas.
- In COM1005 BFS is used to search for particular target vertices and stops when a target is reached. Here we explore the whole graph.

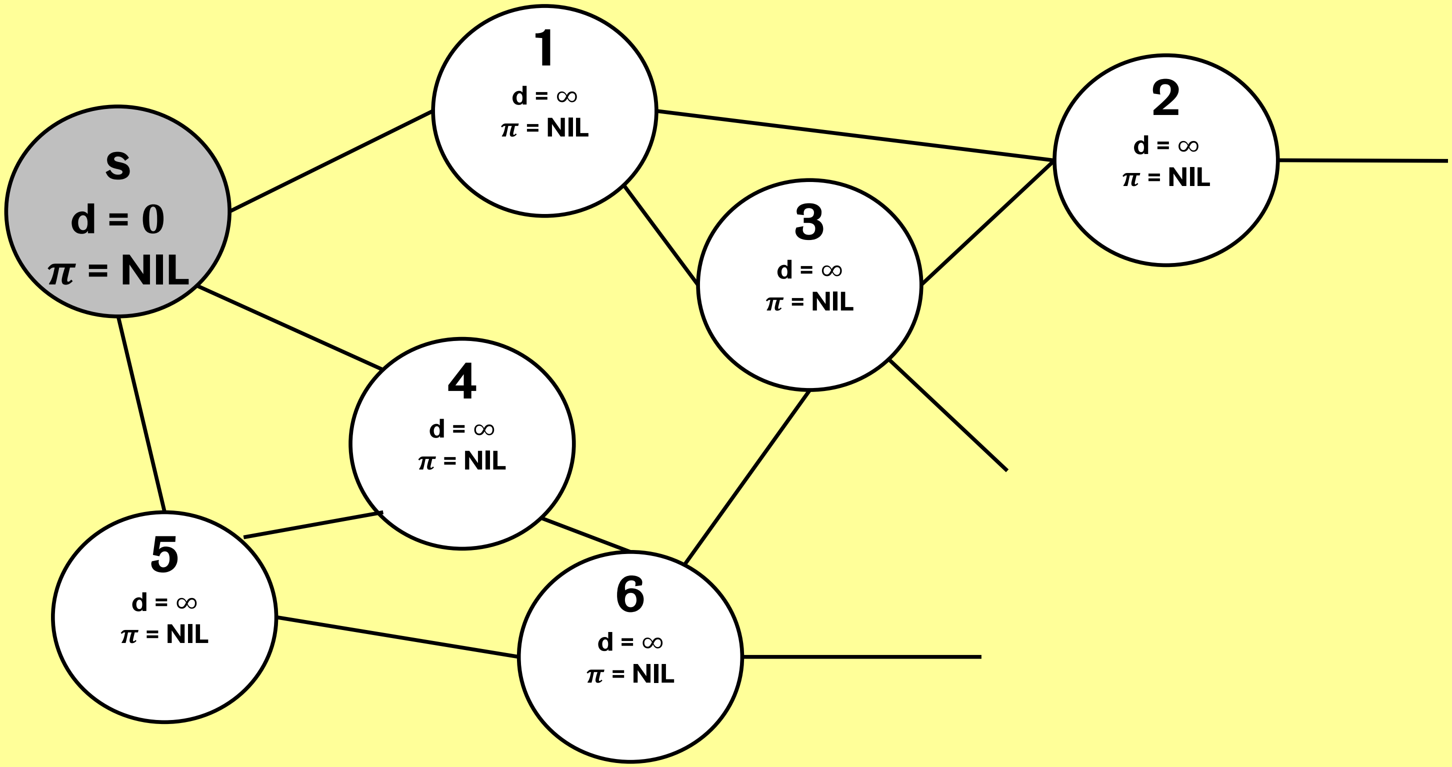
Breadth-first search: Ideas

- Start from the source and then explore the frontier between discovered and undiscovered vertices. BFS explores the whole breadth of this frontier.
- BFS uses a **queue** to store the next vertices to be processed:
  - extract the vertex at the front of the queue
  - add its neighbours to the end of the queue
- We also keeps notes (see next slide) of:
  - which vertices have been checked and what the algorithm discovered
  - other useful information

▶Things to keep track of

- We assign colours to vertices to indicate their status:
  - **White**: vertex has not been discovered yet
  - **Gray**: vertex has been discovered, but needs to be processed.
  - **Black**: vertex has been discovered and processed
- Vertices are also assigned attributes
  - d (distance from the starting node)
  - π (predecessor/ parent in BF tree).
- Following the π pointers gives the shortest path back to the starting node.

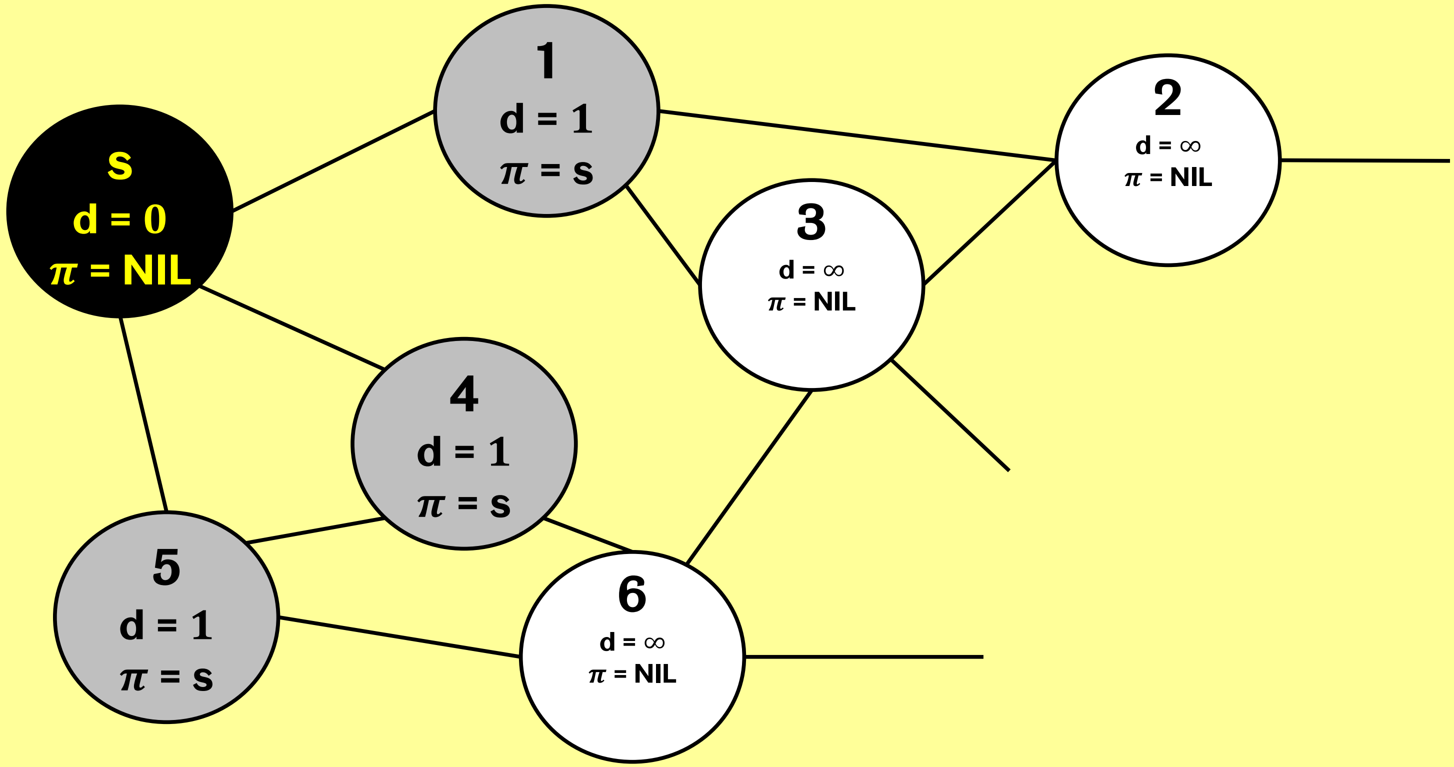
BFS in action (initial configuration)



next one

s									

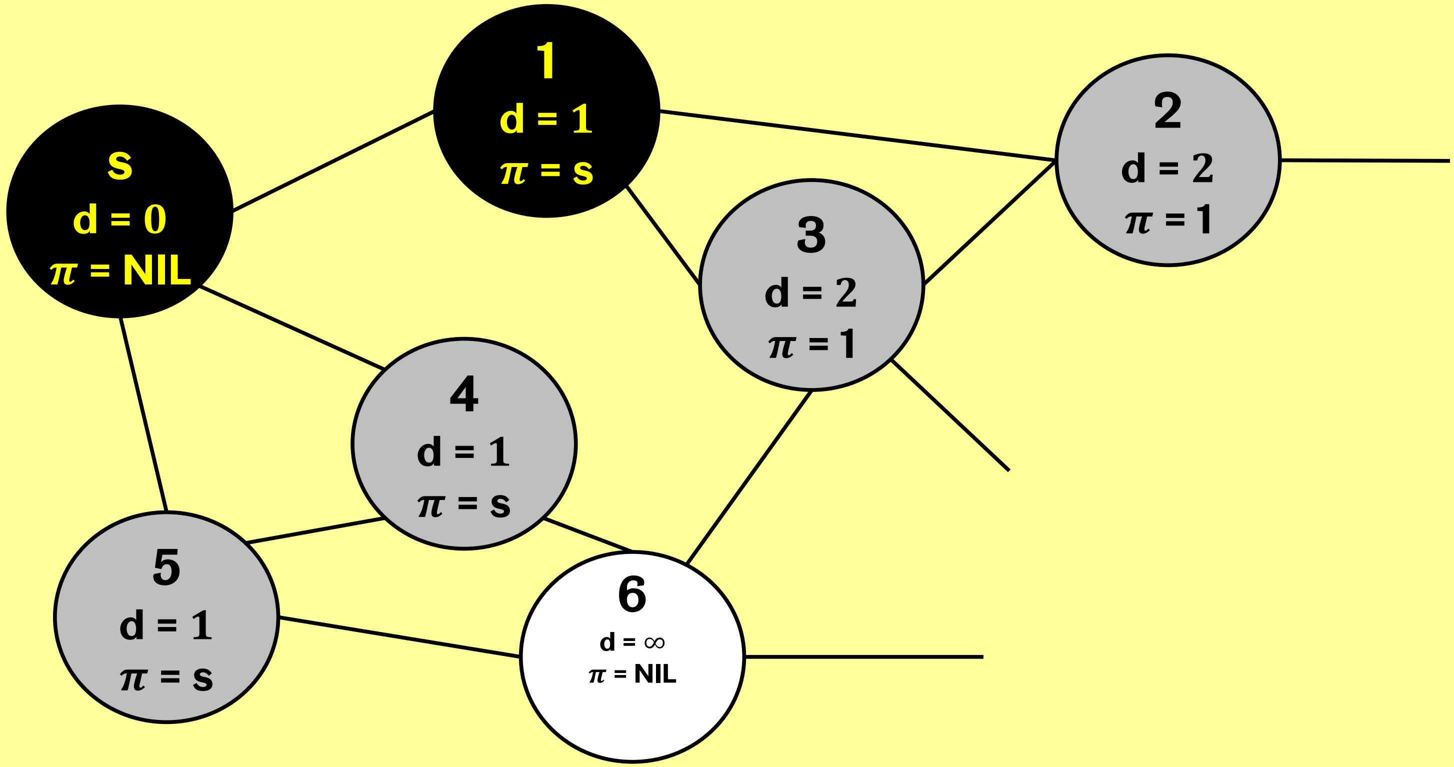
BFS in action: after processing s



next one

removed s	1	4	5							

BFS in action: after processing 1

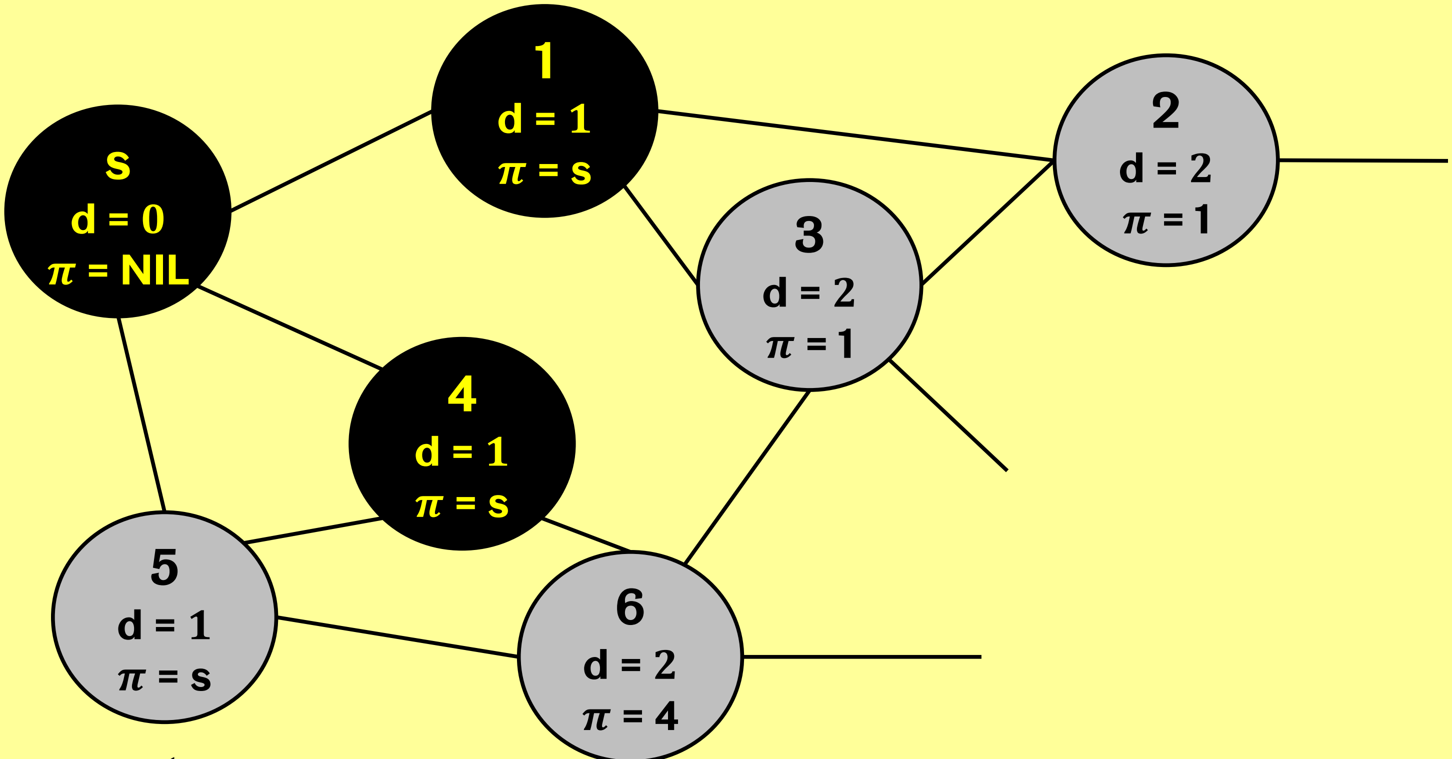


next one

removed 1

4	5	2	3						

BFS in action: after processing 4

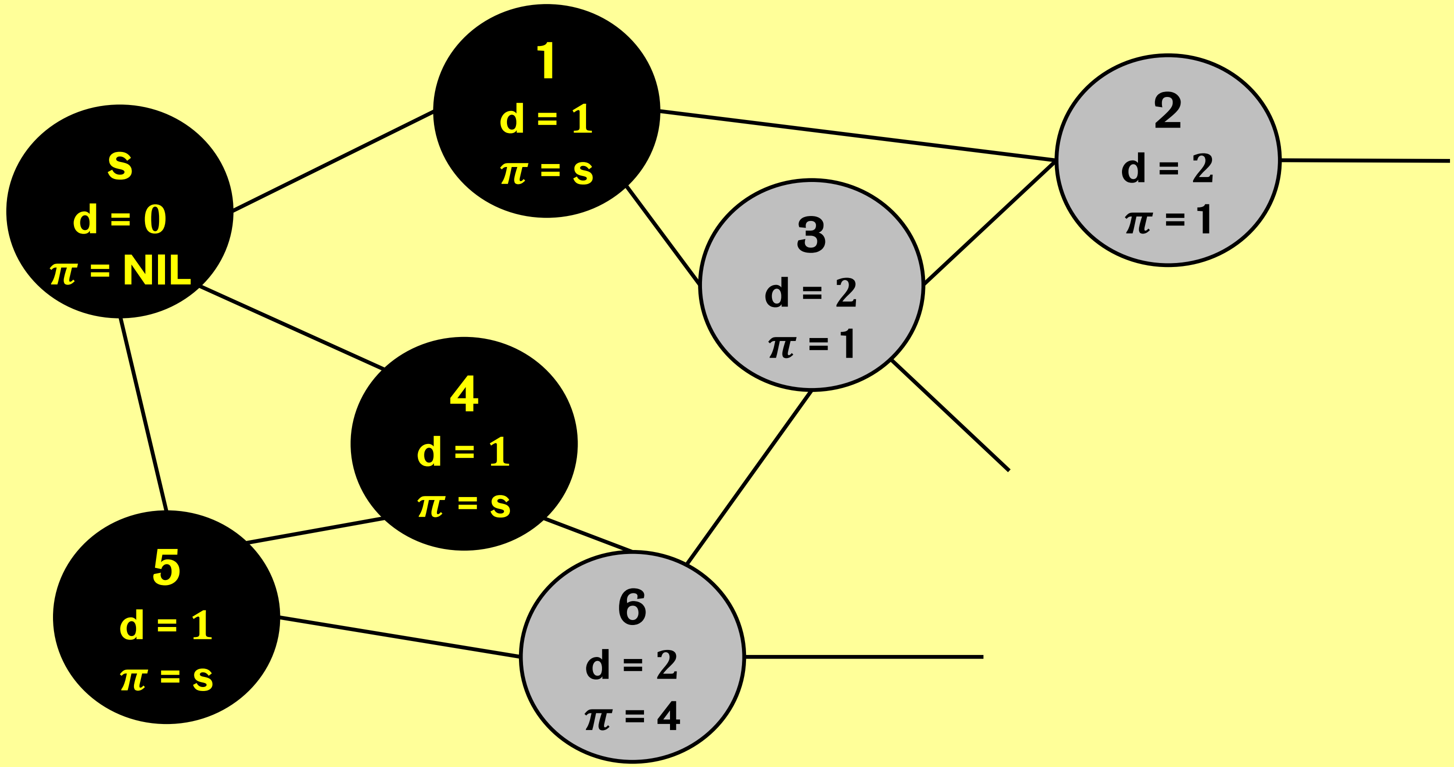


5 is not white (so has already been found) - don't update it

next one

removed 4	5	2	3	6						

BFS in action: after processing 5



6 is not white (so has already been found) - don't update it

next one

removed 5	2	3	6							

BFS

- Lines 1-8: Initially all vertices but s are white.
- While loop: extract front vertex *u* and add all its unseen (white) adjacent vertices *v* to the end of the queue.
- *v*'s distance is one larger than *u*'s, *u* becomes *v*'s predecessor.
- Enqueued vertices become gray, dequeued ones are turned black.

```
BFS(G, s)
1: for each vertex u ∈ V \ {s} do
2:     u.colour = WHITE
3:     u.d = ∞
4:     u.π = NIL
5: s.colour = GRAY
6: s.d = 0
7: s.π = NIL
8: Q = ∅
9: ENQUEUE(Q, s)
10: while Q ≠ ∅ do
11:     u = DEQUEUE(Q)
12:     for each v ∈ Adj[u] do
13:         if v.colour = WHITE then
14:             v.colour = GRAY
15:             v.d = u.d + 1
16:             v.π = u
17:             ENQUEUE(Q, v)
18:     u.colour = BLACK
```

BFS: Runtime (for scanning whole graph)

- No vertex becomes white.
- Test for whiteness is positive only once, as vertices are made grey immediately.
- Hence each vertex is enqueued and dequeued at most once. Time O(|V|) for queue operations.
- Adjacency list of each vertex is scanned at most once, hence total time for scanning all adjacency lists is O(|E|).

```
BFS(G, s)
...
while Q ≠ ∅ do
    u = DEQUEUE(Q)
    for each v ∈ Adj[u] do
        if v.colour = WHITE then
            v.colour = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.colour = BLACK
```

- Overhead before while loop is O(|V|), hence total time is **O(|V| + |E|)**, **linear in the input size.**

Summary for Breadth-First Search

- Breadth-first search searches the breadth of the frontier between discovered and undiscovered vertices.
- It creates a **breadth-first tree** that encodes shortest paths for all vertices. Following predecessors/parents in the tree reconstructs a shortest path from a vertex v to s.
- The running time of BFS is **O(|V| + |E|)**, **linear in the input size.**

Depth-first search (DFS)

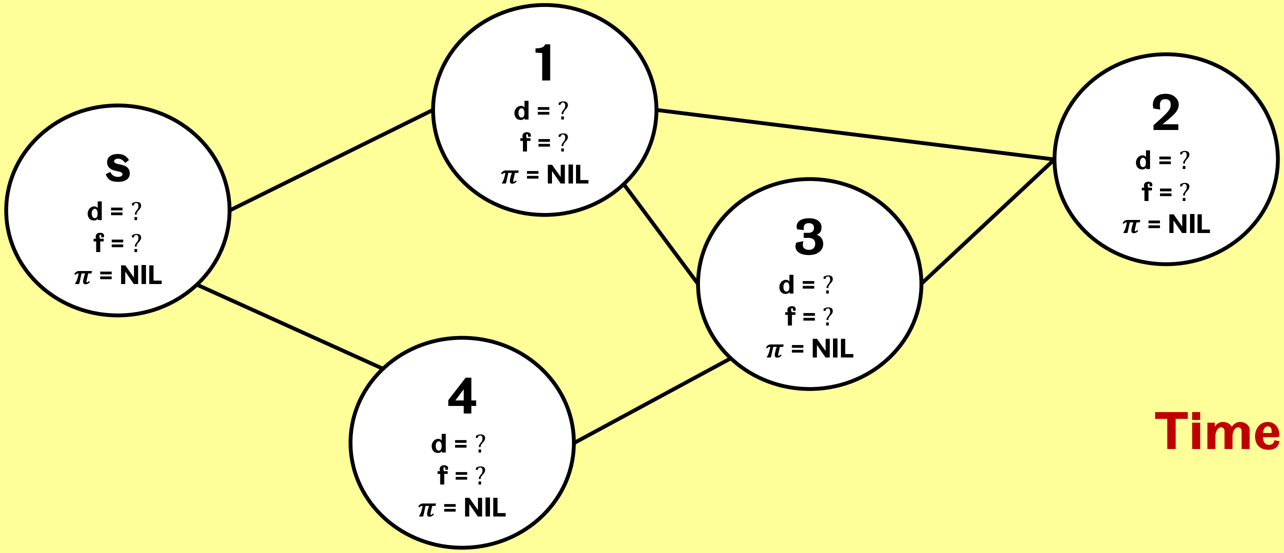
- Works for undirected and directed graphs.
- Ideas:
  - Go into depth by exploring edges out of the most recently discovered vertex and backtrack when stuck.
  - Continue until all vertices reachable from the start vertex are discovered.
  - If any undiscovered vertices remain, continue with one of them as new source.
- As for BFS, define predecessors that represent several **depth-first trees**. These trees form a **depth-first forest.**

DFS: Colours and timestamps

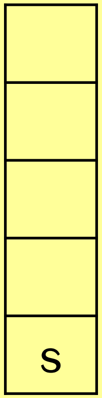
- DFS uses colours white, gray, black as for BFS:
  - **White**: vertex has not been discovered yet
  - **Gray**: vertex has been discovered, but is not finished yet.
  - **Black**: vertex has been finished (finished scan of adjacency list).
- Also uses **timestamps**:
  - **d** is when v is first **discovered** (and grayed), **f** is when v is **finished** (and blackened). Hence for all vertices $v.d < v.f$.
  - Global variable time is incremented with each event

S	1	2	3	4

DFS in action (at start)



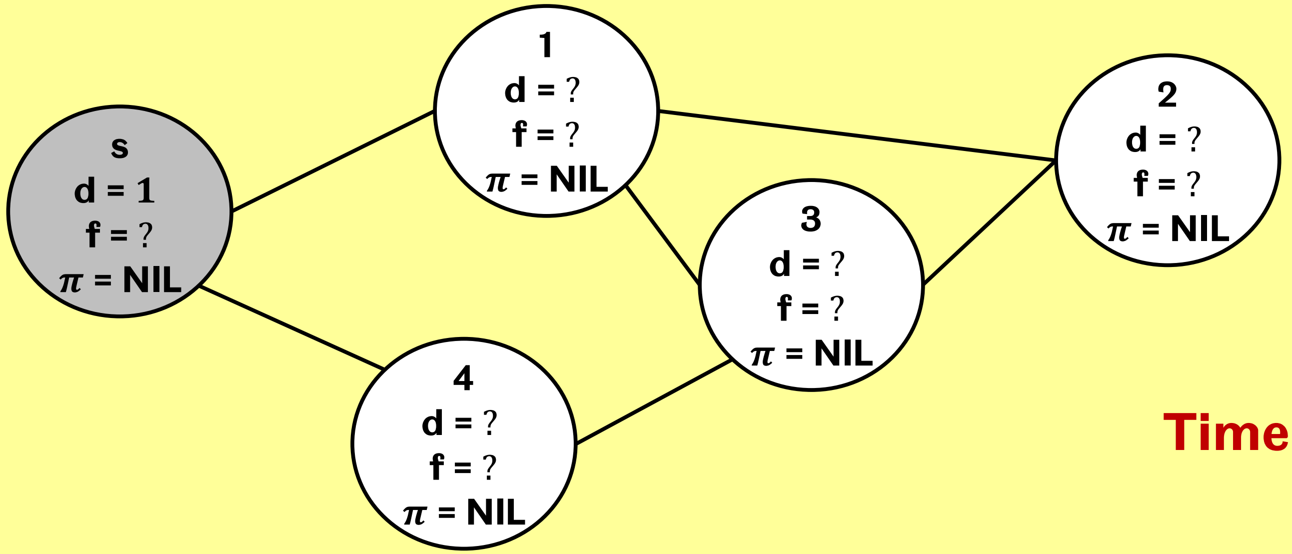
Time = 0



Recursive calls mean that DFS implicitly uses a **stack** to store vertices while exploring the graph (cf. BFS using a queue).

S	1	2	3	4

DFS in action (visit s)



Time = 1

Adj[s]

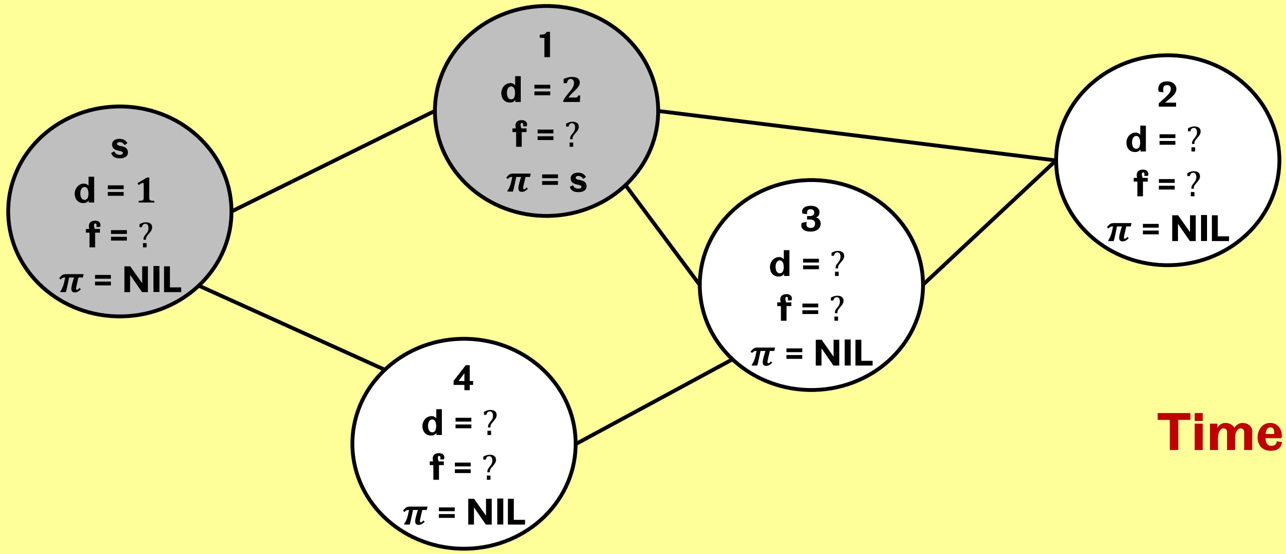
1	4								

1
4

Removed s

S	1	2	3	4

DFS in action (visit 1)



Time = 2

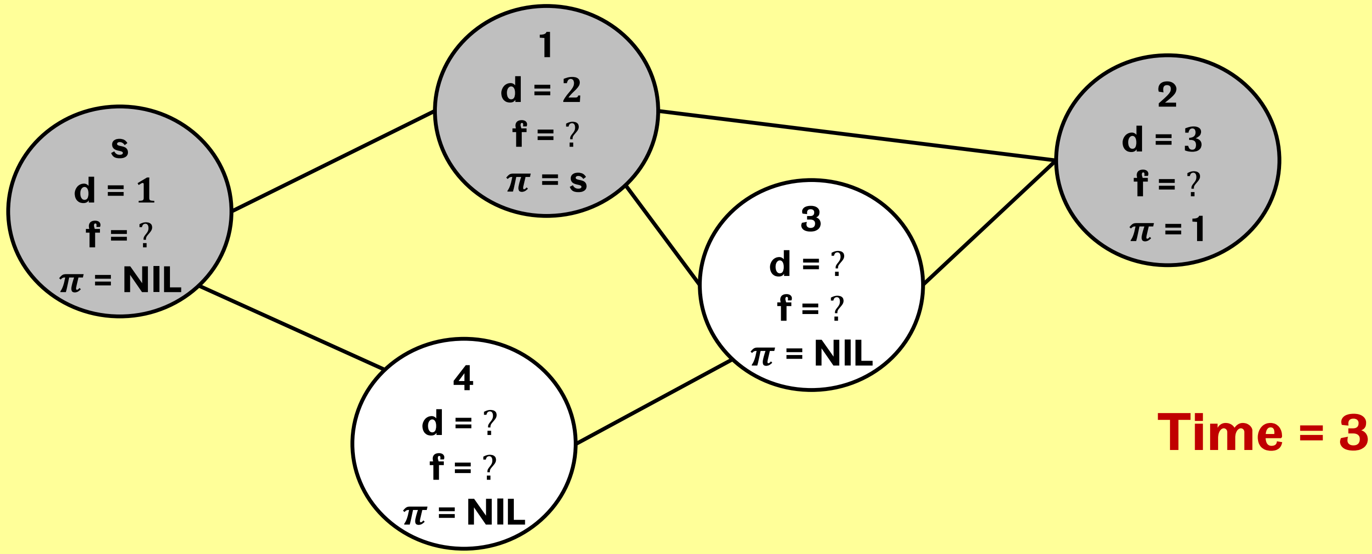
Adj[s]	1	4								
Adj[1]	s	2	3							

2
3
4

Removed 1

S	1	2	3	4

DFS in action (visit 2)



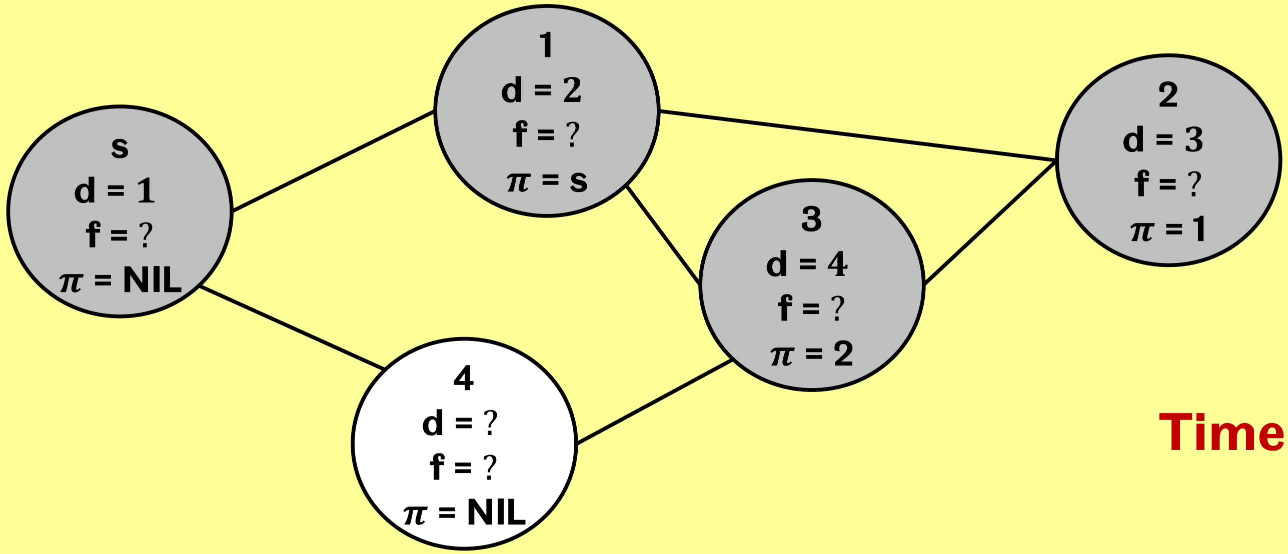
Adj[s]	1	4								
Adj[1]	s	2	3							
Adj[2]	1	3								

3
4

Removed 2

S	1	2	3	4

DFS in action (visit 3)



Time = 4

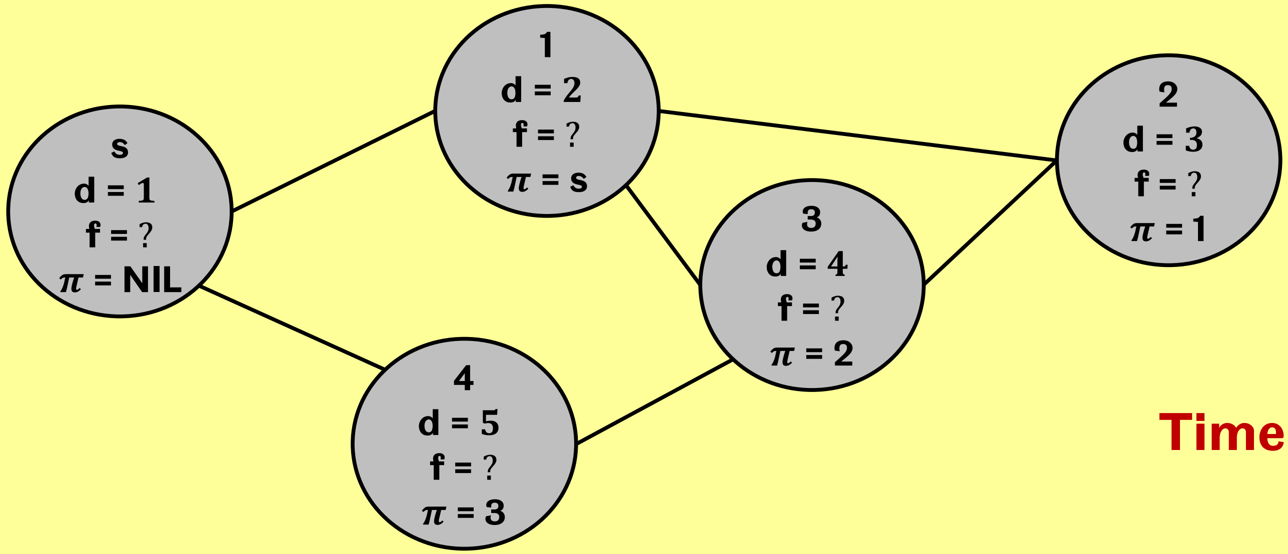
Adj[s]	1	4								
Adj[1]	s	2	3							
Adj[2]	1	3								
Adj[3]	1	2	4							

4

Removed 3

S	1	2	3	4

DFS in action (visit 4)



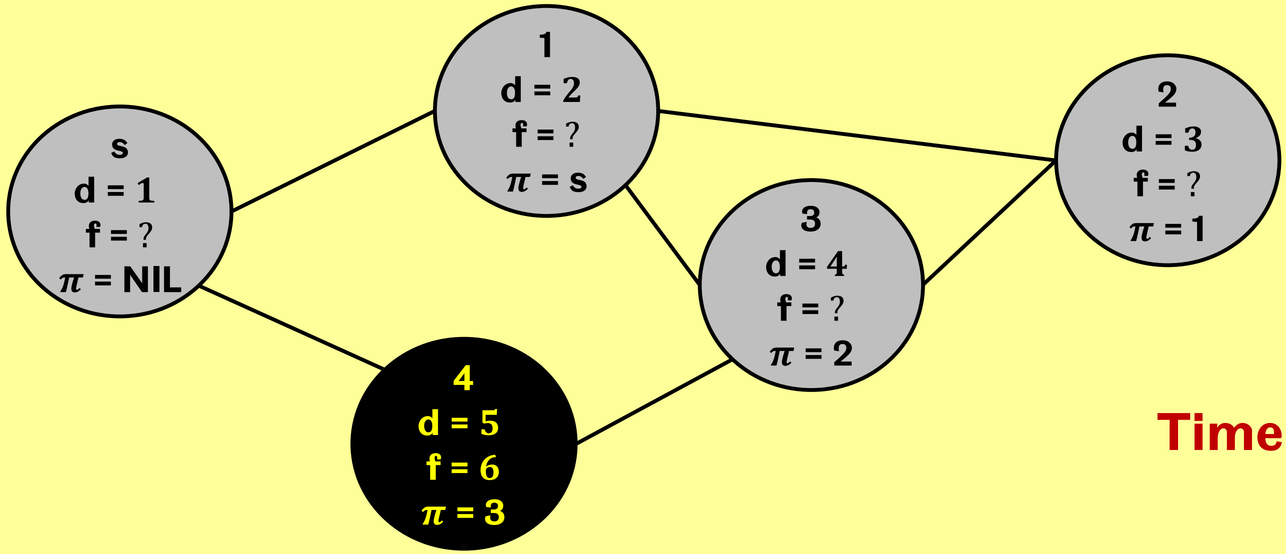
Time = 5

Adj[s]	1	4								
Adj[1]	s	2	3							
Adj[2]	1	3								
Adj[3]	1	2	4							
Adj[4]	s	3								

Nothing left to check

S	1	2	3	4

DFS in action (finish visit to 4)



Time = 6

Adj[s]

1	4								

Adj[1]

s	2	3							

Adj[2]

1	3								

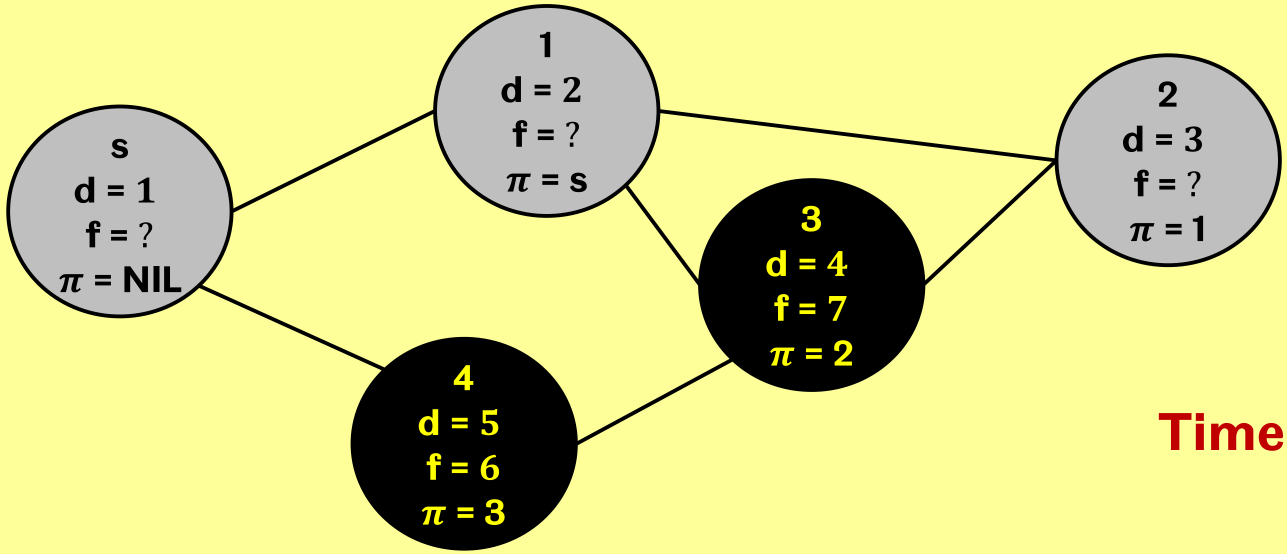
Adj[3]

1	2	4							

Nothing left to check

S	1	2	3	4

DFS in action (finish visit to 3)



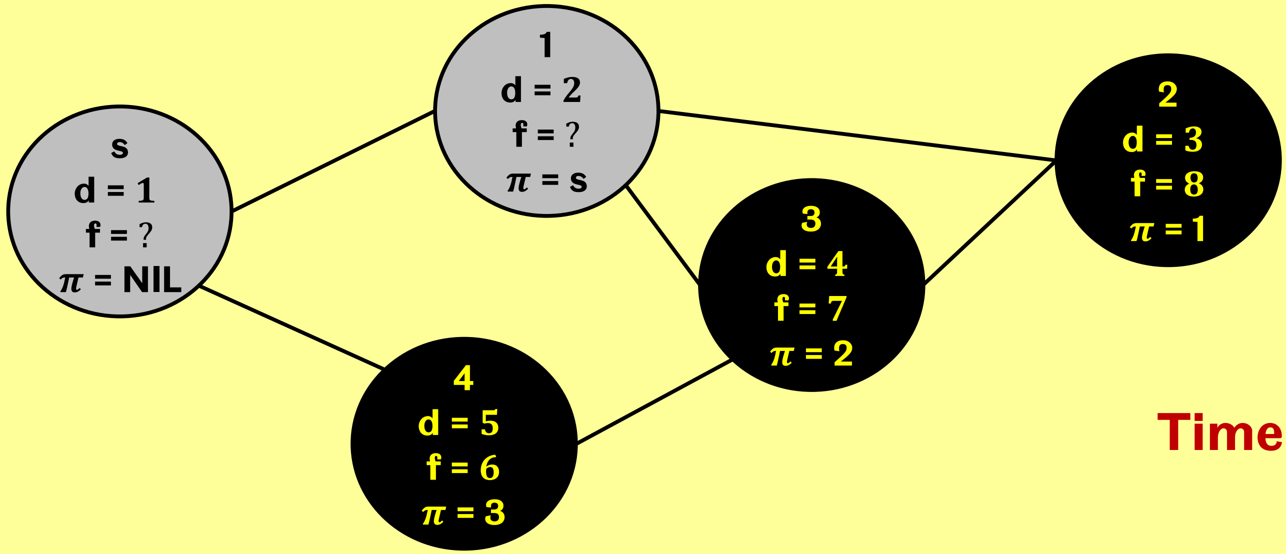
Time = 7

Adj[s]	1	4								
Adj[1]	s	2	3							
Adj[2]	1	3								

Nothing left to check

S	1	2	3	4

DFS in action (finish visit to 2)



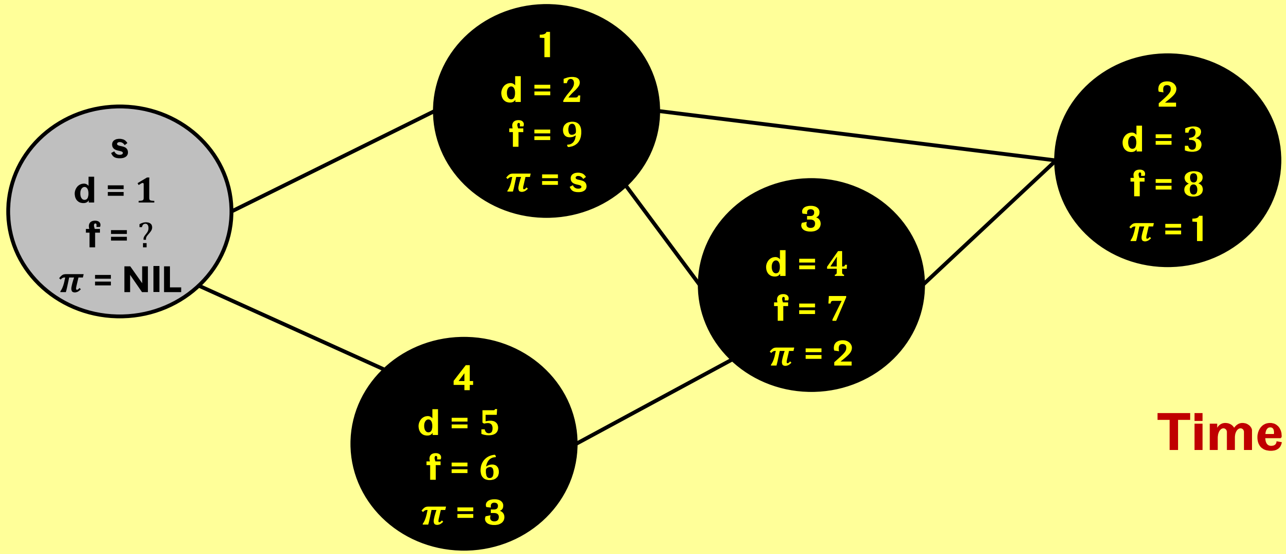
Time = 8

Adj[s]	1	4								
Adj[1]	s	2	3							

Nothing left to check

S	1	2	3	4

DFS in action (finish visit to 1)



Time = 9

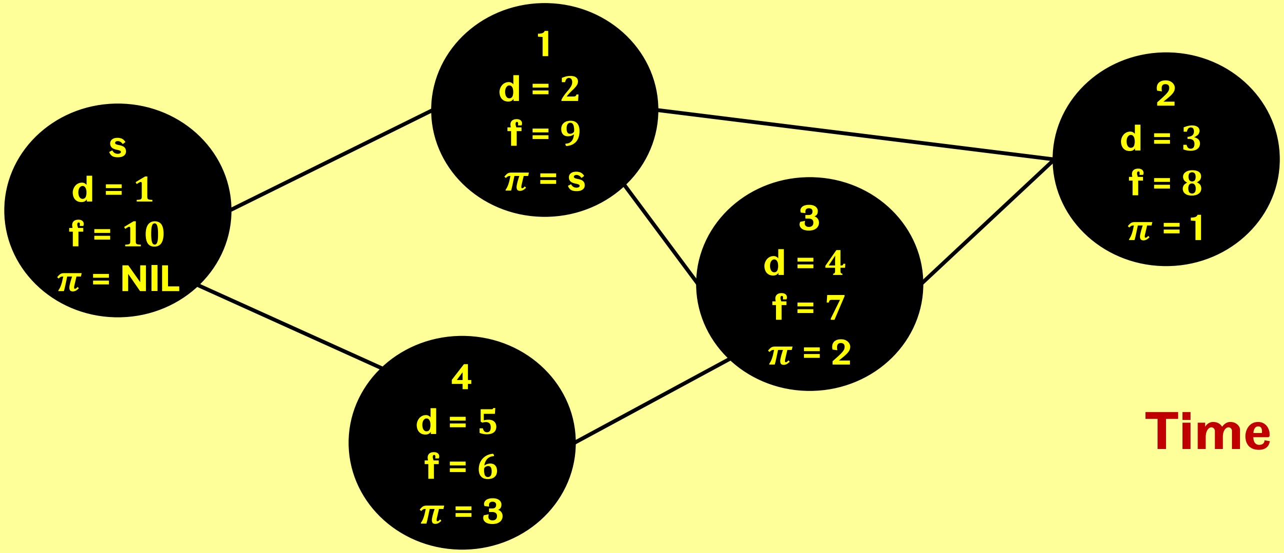
Adj[s]

1	4								

Nothing left to check

Nothing left to check: S 1 2 3 4

DFS in action (finish visit to s)



Time = 10

If there are vertices we can't reach from s, we continue with the next undiscovered node, starting at time = 11.

DFS: Pseudocode and runtime

DFS(G)

```
for each vertex u ∈ V do
    u.colour = white
    u.π =NIL
time = 0
for each vertex u ∈ V do
    if u.colour == white then
        DFS-VISIT(G, u)
```

DFS-VISIT(G, u)

```
time = time+1
u.d = time
u.colour = gray
for each v ∈ Adj[u] do
    if v.colour == white then
        v.π = u
        DFS-VISIT(G, v)
u.colour = black
time = time+1
u.f = time
```

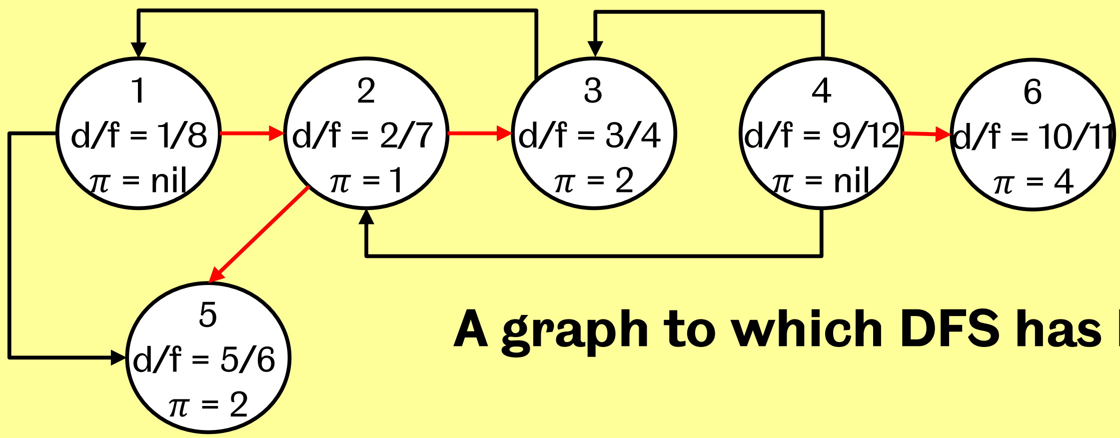
- Runtime:
  - DFS does θ(|V|) work setting things up, then starts the visits.
  - Between them, all the calls to DFS-Visit account for each outgoing edge exactly once. DFS-Visit itself does constant extra work.
  - So the total cost for DFS is θ(|V|+|E|).

DFS: Parenthesis structure

In any DFS of a (directed or undirected) graph, for any two vertices $u \neq v$, either

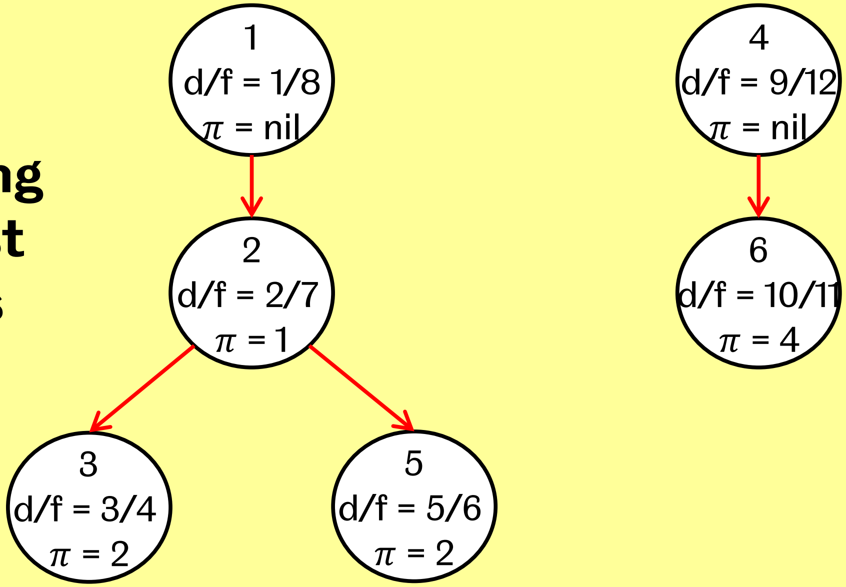
1. DFS-Visit(v) is called during DFS-Visit(u)
   - then v is a descendant of u and DFS-Visit(v) finishes earlier than u.
     So: $u.d < v.d < v.f < u.f$
2. DFS-Visit(u) is called during DFS-Visit(v)
   - So: $v.d < u.d < u.f < v.f$
3. the intervals [u.d, u.f] and [v.d, v.f] are entirely disjoint, and neither u nor v is a descendant of the other.

This means the DFS search effectively generates a depth-first forest (collection of trees) showing which visits called which others.



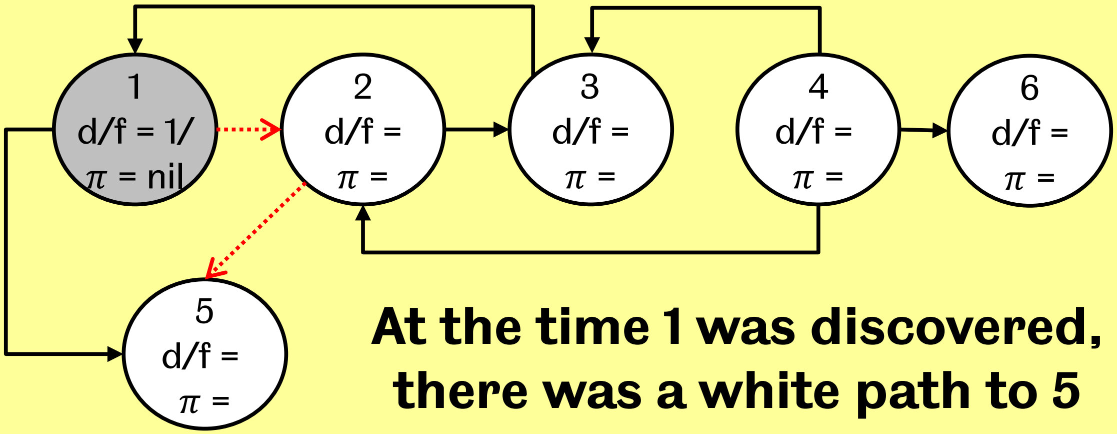
A graph to which DFS has been applied

The corresponding depth-first forest contains 2 trees



White-path theorem

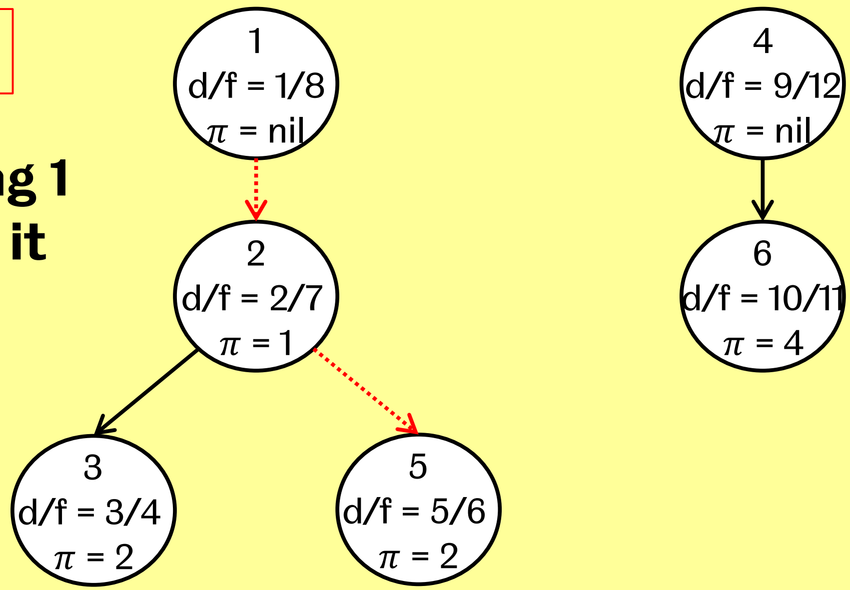
Theorem 22.9: In a depth-first forest of a (directed or undirected) graph, vertex *v* is a descendant of a vertex *u* if and only if at the time u.d that the search discovers u, there is a path from *u* to *v* consisting entirely of white vertices.



At the time 1 was discovered, there was a white path to 5

if and only if

The tree containing 1 contains 5 below it



▶Proving the white-path theorem (1)

Theorem 22.9: In a depth-first forest of a (directed or undirected) graph, vertex *v* is a descendant of a vertex *u* if and only if at the time u.d that the search discovers u, there is a path from *u* to *v* consisting entirely of white vertices.

- This is a statement of the form "A $\Leftrightarrow$ B "
- To prove this kind of statement, we split it into two parts:
  1. Prove that A $\Rightarrow$ B
  2. Prove that B $\Rightarrow$ A

Proof of "⇒"

Theorem 22.9: In a depth-first forest of a (directed or undirected) graph: if vertex *v* is a descendant of a vertex *u* then at the time u.d that the search discovers *u*, there is a path from *u* to *v* consisting entirely of white vertices.

Proof of "⇒" (being descendant implies white path):

- If *u*=*v* then u is still white when u.d is set, thus a white path from u to v exists (just one vertex *u*=*v*).
- If *v* is a proper descendant of *u*, then u.d < v.d and therefore *v* is white at time u.d. This holds for all descendants of *u*, hence a white path from *u* to *v* exists at time u.d.

▶Proof of "⇐"

Theorem 22.9: In a depth-first forest of a (directed or undirected) graph: if at the time the search discovers u, there is a path from u to v consisting entirely of white vertices then v is a descendant of u.

Proof of "⇐" (by contradiction):

- Suppose there is a white path from *u* to *v* when u is discovered (time = u.d). Assume *v* is the first vertex on the path which is not a descendant of *u* (otherwise we consider this first vertex instead). Let *w* be the predecessor of *v* on the path (could be *w*=*u*).
  - *w* must be a descendant of *u* (by above assumption). Thus $w.f < u.f$.
  - *v* is discovered after *u* but before *w* finishes (since there is an edge from *w* to *v*), so we get: $u.d < v.d < w.f$.
- It follows that $u.d < v.d < u.f$. Now parenthesis structure tells us that $u.d < v.d < v.f < u.f$
- So *v* must be a descendant of *u* after all. [this is the desired contradiction - QED]

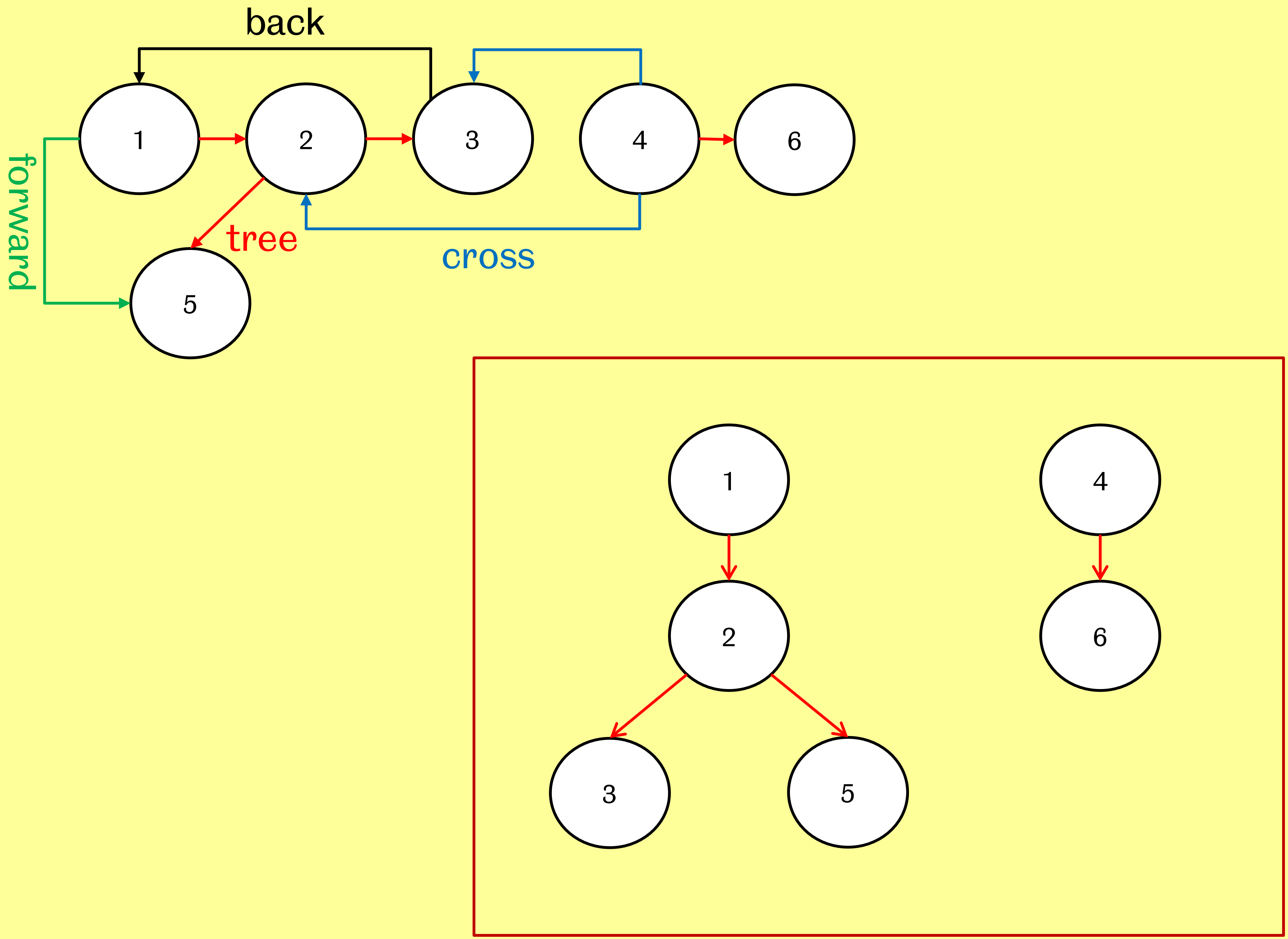
Classification of edges in directed graphs

1. **Tree edges** are edges in the depth-first forest. Edge (u, v) is a tree edge if v was first discovered by exploring edge (u, v).
   An edge (u, v) is a tree edge if at the time of exploration v is white.

2. **Back edges** are edges (u, v) connecting a vertex u to an ancestor v in a depth-first tree (or self-loops in directed graphs).
   An edge (u, v) is a back edge if at the time of exploration v is grey.

3. **Forward edges** are nontree edges (u, v) connecting a vertex u to a descendant v in a depth-first tree (pointing forward in the tree).
   (u, v) is a forward edge if v is black and was discovered later: u.d < v.d.

4. **Cross edges** are all other edges: either leading to a subtree constructed earlier or leading to a different (earlier) depth-first tree.
   (u, v) is a cross edge if v is black and was discovered earlier: u.d > v.d.


back
1
2
3
4
6
forward
tree
cross
5
1
4
2
6
3
5

▶Edge classification in undirected graphs

Theorem 22.10: In a depth-first search of an undirected graph, every edge is either a tree edge or a back edge.

→ There are no forward or cross edges in undirected graphs.

Proof. Suppose (u,v) is an edge in the graph, and suppose we are just discovering u.

- If u is discovered before v, then *v* is still white, so this becomes a tree edge (because it's a white path from u to v).

- If v was already discovered, then the same reasoning says that (v,u) must be a tree edge. So (u,v) must be a back edge.

QED.

Precedence graphs

- Graphs have many applications. One of them is modelling precedences:
  - Vertices represent tasks
  - A edge (u, v) means that task u has to be executed before task *v*.
- Coming up: how to order tasks such that all precedence constraints are respected.
  - This is only feasible if the precedence graph does not contain any cycles (paths from a node back to itself)
  - A graph with no cycles in it is called **acyclic**.

Application of DFS: testing for cycles

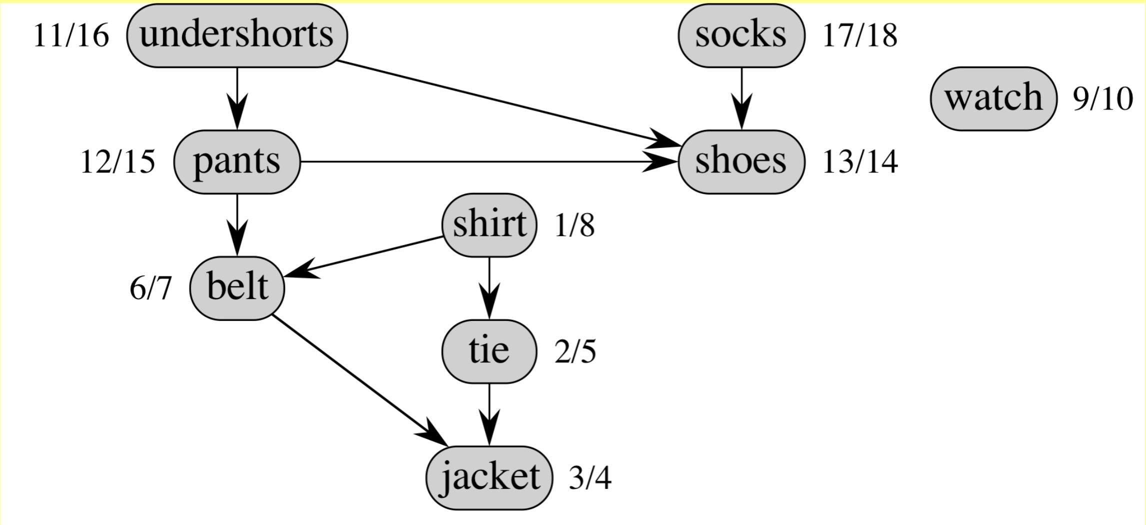
Theorem (adapted from Lemma 22.11): A directed graph G contains a cycle if and only if DFS finds at least one back edge.

Proof (for directed graphs):

- "⇐": Suppose DFS produces a back edge (*u, v*). Then *v* is an ancestor of *u* in the depth-first tree. Thus, G contains a path (of tree edges) from *v* to *u*, and the back edge completes a cycle.
- "⇒": Suppose that G contains a cycle C. We show that DFS yields a back edge. Let v be the first vertex to be discovered in C, and let (*u, v*) be the edge on C going into *v*. At time v.d, the vertices of C form a path of white vertices from *v* to *u*. By the white-path theorem, *u* becomes a descendant of *v*. Therefore, (*u, v*) is a back edge.

▶Topological sorting

- Consider a directed acyclic graph ("dag") showing precedence between tasks. We want to sort them into a list that respects the precedence requirements.
  - A topological sort of a dag is a linear ordering of all its vertices such that for each edge (*u*, *v*), *u* appears before *v*.
  - If vertices are arranged on a horizontal line, all edges go from left to right.



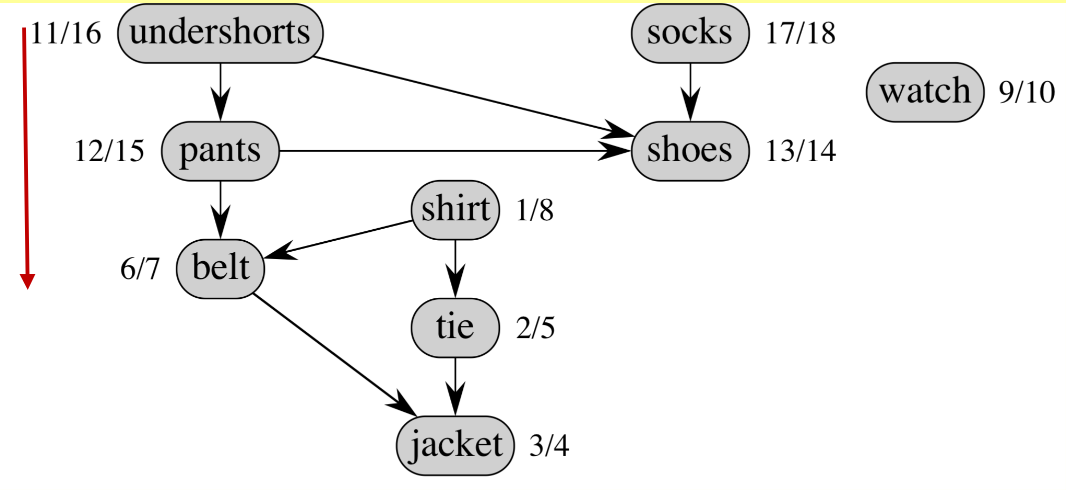
Computing a topological sort

- Here's how to use DFS to compute a topological sort:

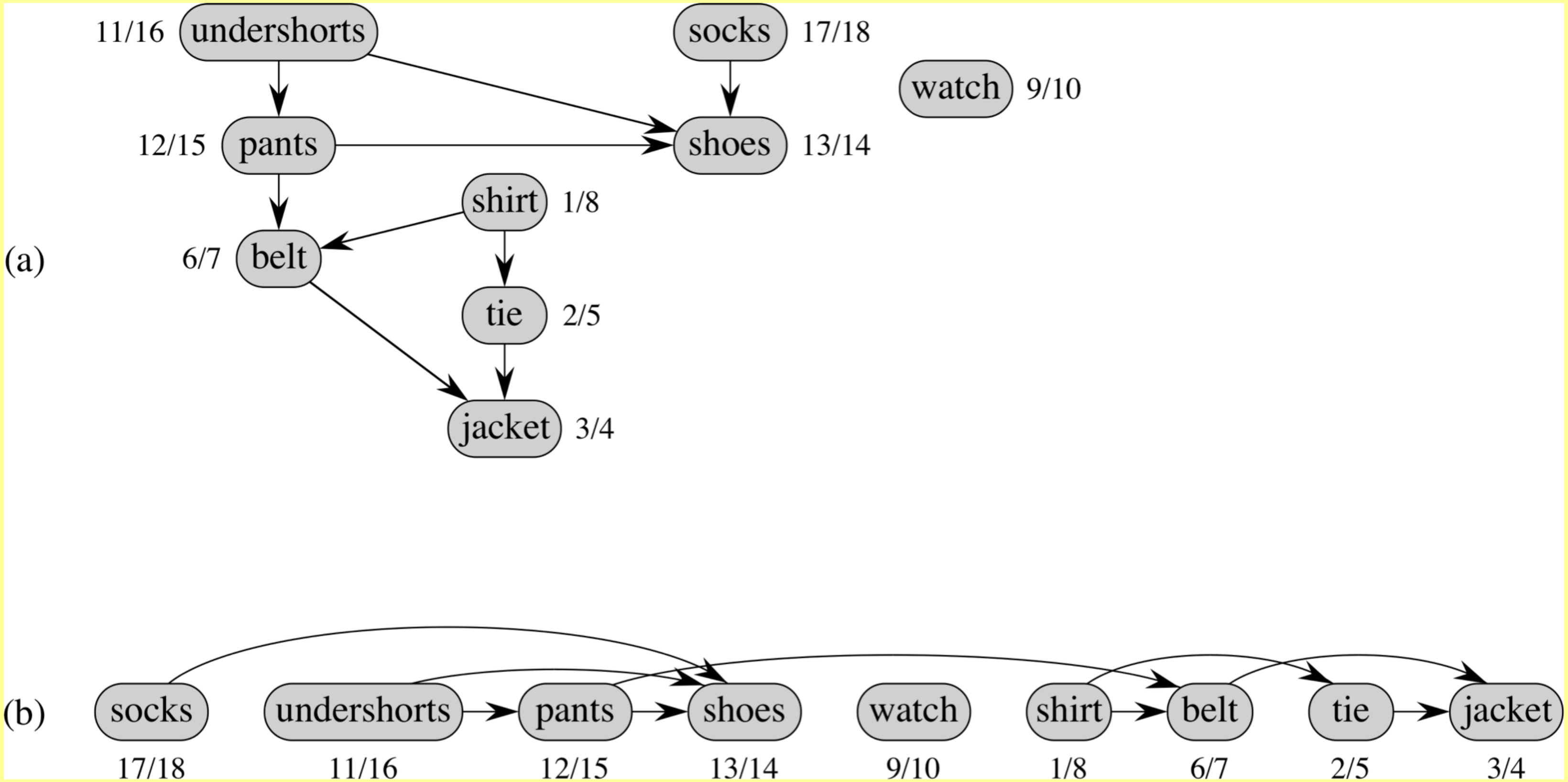
TOPOLOGICAL-SORT(G)

1: call DFS(G) to compute finishing times $v.f$ for each vertex v
2: as each vertex is finished, insert it onto the front of a linked list
3: **return** the linked list of vertices

The first thing we need to do has the latest DFS finishing time



Getting dressed



Topological sort: Running time

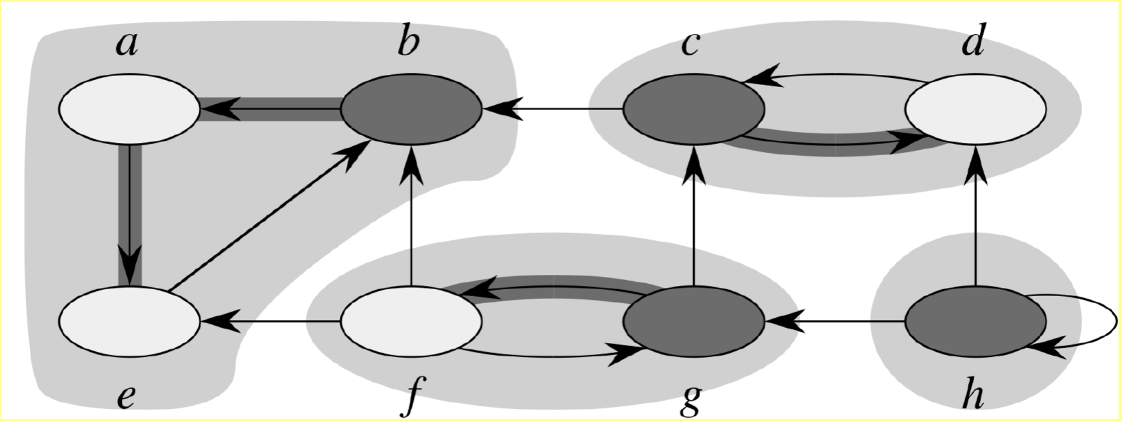
TOPOLOGICAL-SORT(G)

1: call DFS(G) to compute finishing times $v.f$ for each vertex v
2: as each vertex is finished, insert it onto the front of a linked list
3: **return** the linked list of vertices

- Running time:
  - time for DFS = $\theta(|V|+|E|)$
  - + O(1) for each vertex inserted into the linked list O(|V|)
  - Total time $\theta(|V|+|E|)$

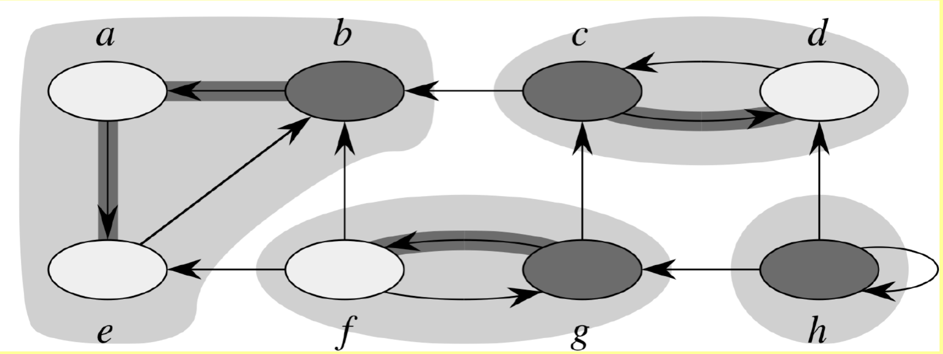
▶Strongly connected components

- A directed graph is called **strongly connected** if every two vertices are reachable from each other.
- The **strongly connected components (SCCs)** of a directed graph are the equivalence classes under the "mutually reachable" relation. In other words, they are maximal sets of vertices where all vertices in every set are mutually reachable.



▶Strongly connected components

- Applications:
  - Finding groups of friends in social network graphs.
  - Many algorithms working on directed graphs decompose the graph into its SCCs, run separately on all of them, and then combine solutions for all SCCs to one overall solution.



Computing SCCs with DFS

- Let G^T be the transpose of G, i. e. the graph where all edges have their direction reversed.
- Note that G and G^T have the same SCC as *u* and *v* are reachable in G^T if and only if they are reachable in G.
- G^T can be computed in time O(|V| + |E|).

STRONGLY-CONNECTED-COMPONENTS(G)

1: call DFS(G) to compute finishing times $v.f$ for each vertex v
2: compute $G^\top$
3: call DFS($G^\top$), but in the main loop of DFS, consider the vertices in order of decreasing $u.f$ (as computed in line 1)
4: output the vertices of the tree in the depth-first forest formed in line 3 as a separate SCC

Correctness of the SCC algorithm

- Why on earth does this work? It's a miracle!
- Proof in the book is not very intuitive.
- There's a simpler and more intuitive proof by Ingo Wegener:

  A simplified correctness proof for a well-known algorithm computing strongly connected components, Information Processing Letters 83(1), pages 17–19.

- Copy available here.

Summary for Depth-First Search

- Depth-first search explores the graph going into depth and using backtracking in time $\theta(|V|+|E|)$.
- DFS classifies edges into **tree, back, forward**, and **cross edges**.
- DFS is used
  - to test whether a graph is **acyclic** in time $\theta(|V|+|E|)$. DFS is used for **topological sorting** in directed acyclic graphs in time $\theta(|V|+|E|)$.
  - to determine **strongly connected components** in graphs in time $\theta(|V|+|E|)$.

▶ And finally …

- There are many other uses for graphs and tree algorithms
  - How can we supply *n* newly built houses with electricity, using the minimum length of ?
  - What is the shortest road-route from Sheffield to Liverpool that doesn't use motorways?
  - If our main goods depot is in Manchester, and each lorry can carry at most n tonnes of goods, how many lorries do we need, and what routes should they use, to deliver all of today's deliveries before 10pm while minimising delivery costs?
- See you next year!