Graphs

Giulia Bernardini
giulia.bernardini@units.it

Fundamentals of algorithms
a.y. 2021/2022

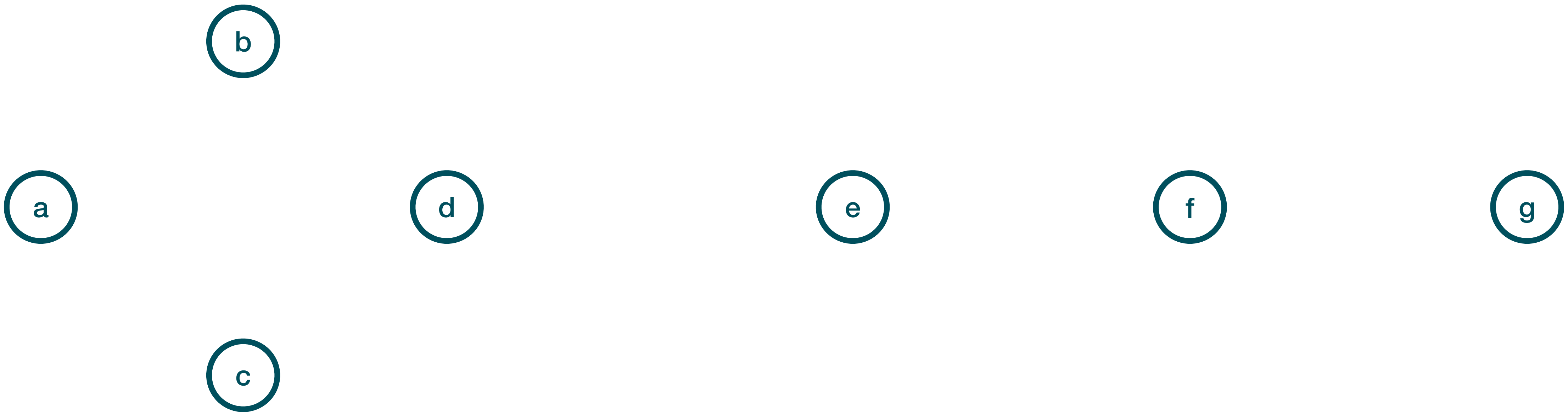
What is a graph?

A graph (V,E) is a collection of vertices and edges:

What is a graph?

A graph (V,E) is a collection of vertices and edges:

V={a,b,c,d,e,f,g} is the set of vertices (aka nodes)

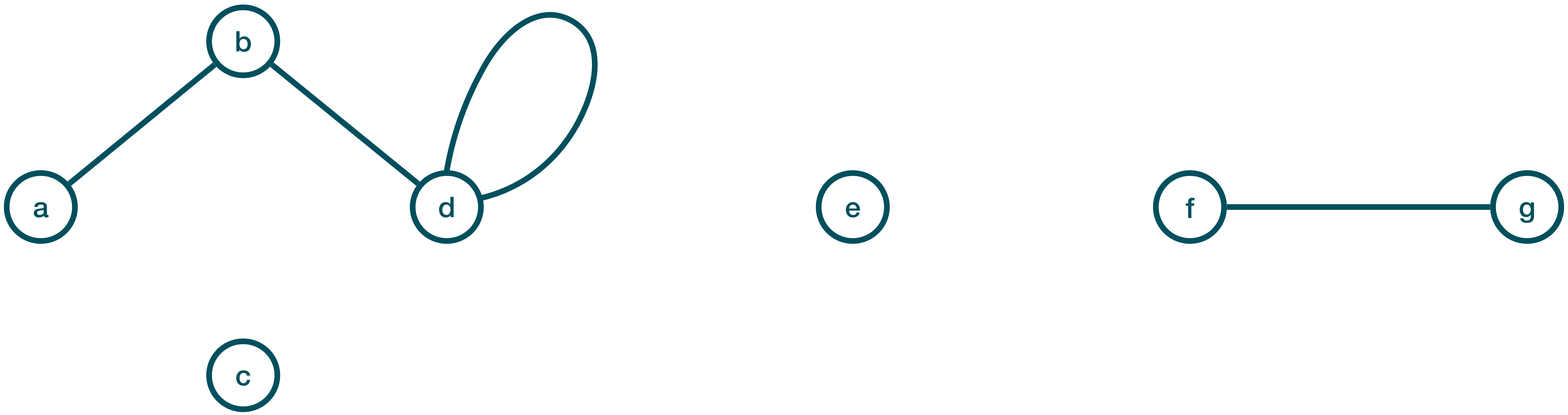


What is a graph?

A graph (V,E) is a collection of vertices and edges:

V={a,b,c,d,e,f,g} is the set of vertices (aka nodes)

E={ {a,b},{b,d},{d,d},{f,g} } is the set of edges



What are graphs for?

In general, they represent relations between objects:

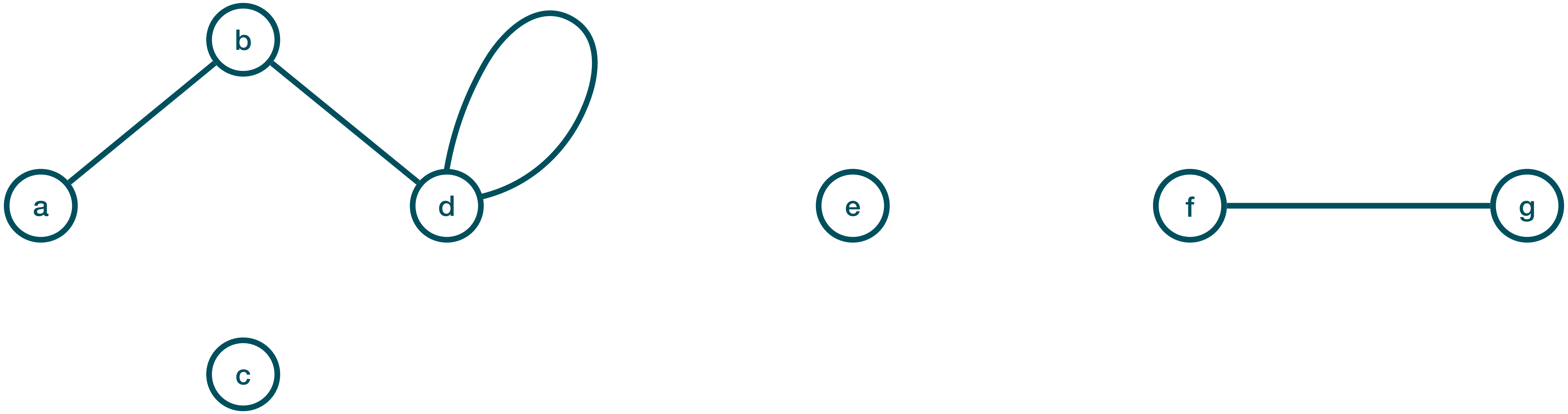
- route systems
- computer networks
- dynamic systems
- information flows
- infectious diseases spread
- dependency relations
- …

Types of graphs

Undirected graphs have undirected edges: {a,b}={b,a}

V={a,b,c,d,e,f,g} is the set of vertices

E={ {a,b},{b,d},{d,d},{f,g} } is the set of undirected edges

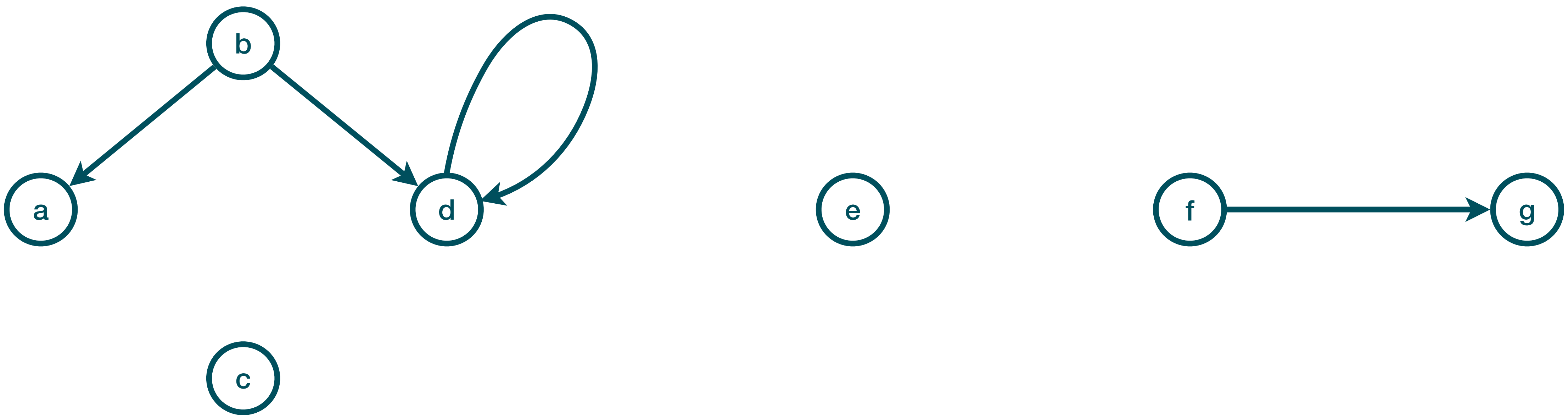


Types of graphs

Directed graphs have directed edges (aka arcs): (a,b)≠(b,a)

V={a,b,c,d,e,f,g} is the set of vertices (aka nodes)

E={ (a,b),(b,d),(d,d),(f,g) } is the set of directed edges (arcs).

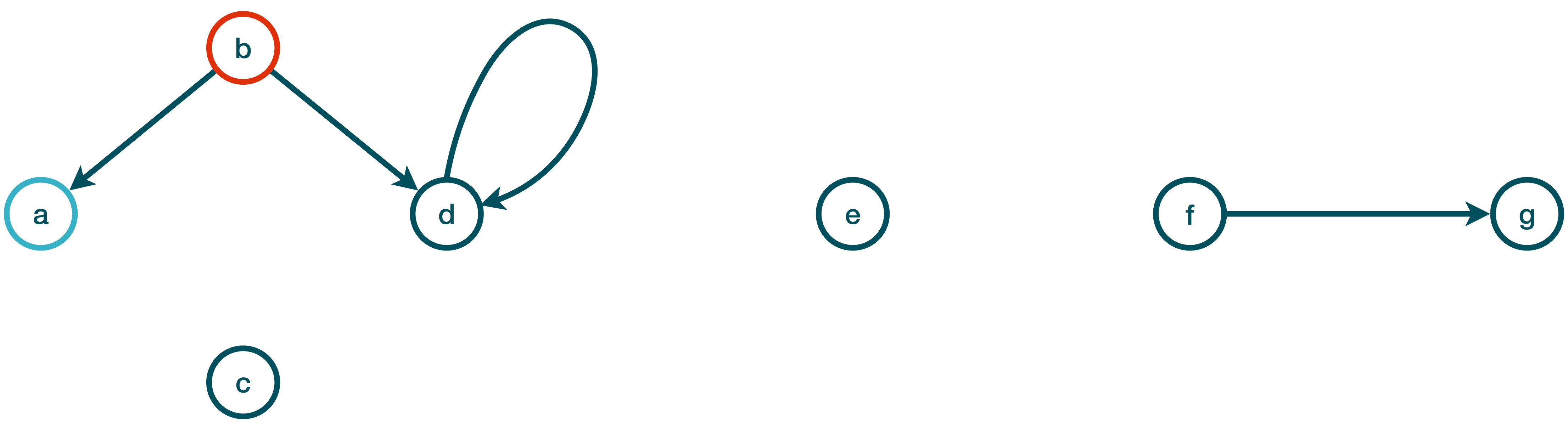


Types of graphs

Directed graphs have directed edges (aka arcs): (a,b)≠(b,a)

V={a,b,c,d,e,f,g} is the set of vertices (aka nodes)

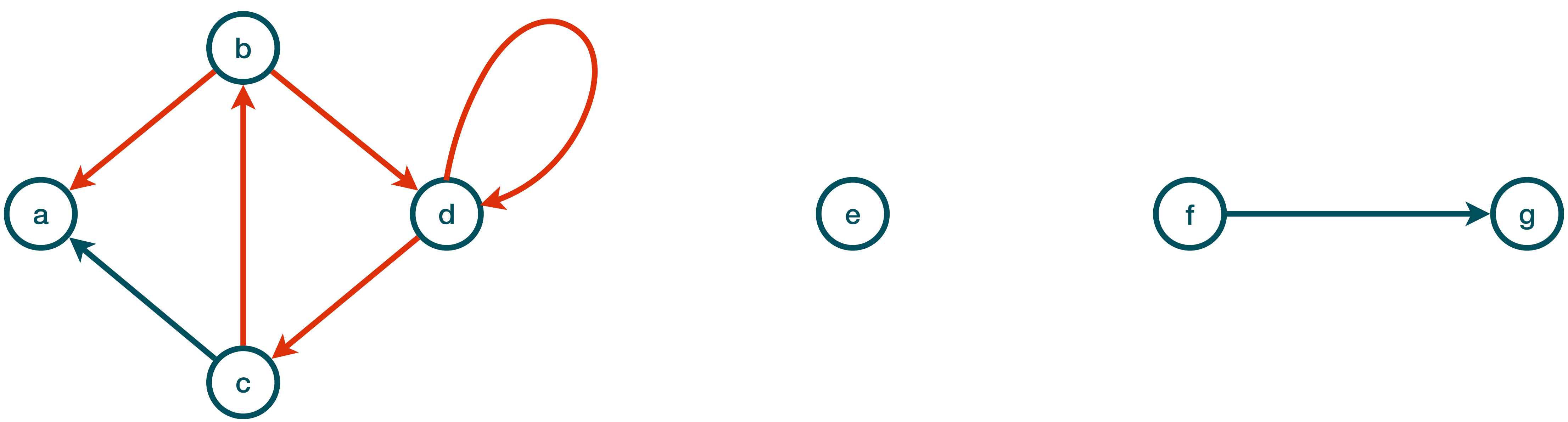
E={ (a,b),(b,d),(d,d),(f,g) } is the set of directed edges (arcs).
The head of an arc (a,b) is a, its tail is b.



Walks, paths, cycles

A walk of length n in G=(V,E) is a sequence of n edges $e_1e_2\ldots e_n$ such that the head of e_i is equal to the tail of e_{i+1}, for all i=1…n-1; equivalently, it is a sequence of n+1 vertices $v_1v_2\ldots v_{n+1}$ such that $(v_i,v_{i+1})\in E$ for all i=1…n

(b,d)(d,d)(d,c)(c,b)(b,a)↔b d d c b a is a walk of length 5 from b to a

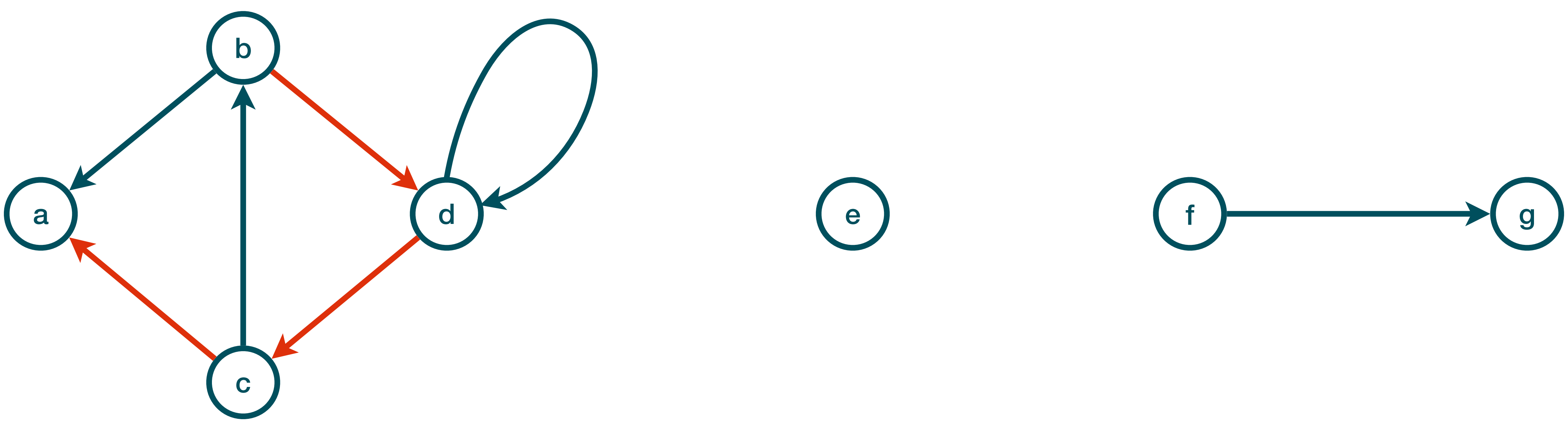


Walks, paths, cycles

A walk of length n in G=(V,E) is a sequence of n edges $e_1e_2\ldots e_n$ such that the head of e_i is equal to the tail of e_{i+1}, for all i=1…n-1; equivalently, it is a sequence of n+1 vertices $v_1v_2\ldots v_{n+1}$ such that $(v_i,v_{i+1})\in E$ for all i=1…n

A path is a walk that does not repeat any vertex

(b,d)(d,c)(c,a)↔b d c a is a path of length 3 from b to a

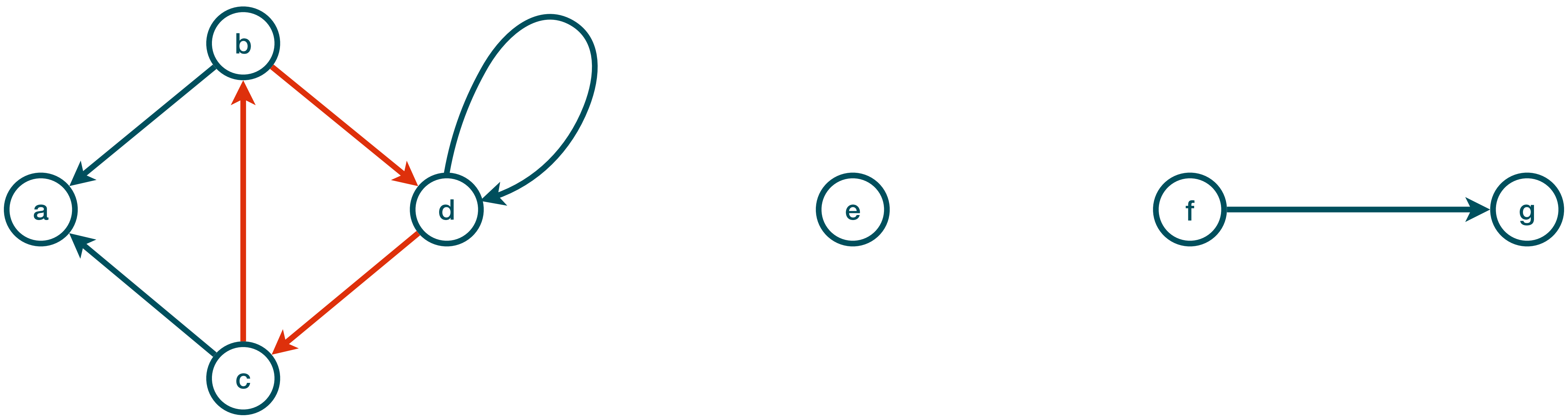


Walks, paths, cycles

A walk of length n in G=(V,E) is a sequence of n edges $e_1e_2 \ldots e_n$ such that the head of e_i is equal to the tail of e_{i+1}, for all i=1…n-1; equivalently, it is a sequence of n+1 vertices $v_1v_2 \ldots v_{n+1}$ such that $(v_i,v_{i+1}) \in E$ for all i=1…n

A path is a walk that does not repeat any vertex. A cycle is a closed path, s.t. the first and the last vertices are the same.

(b,d)(d,c)(c,b)↔b d c b is a cycle of length 3



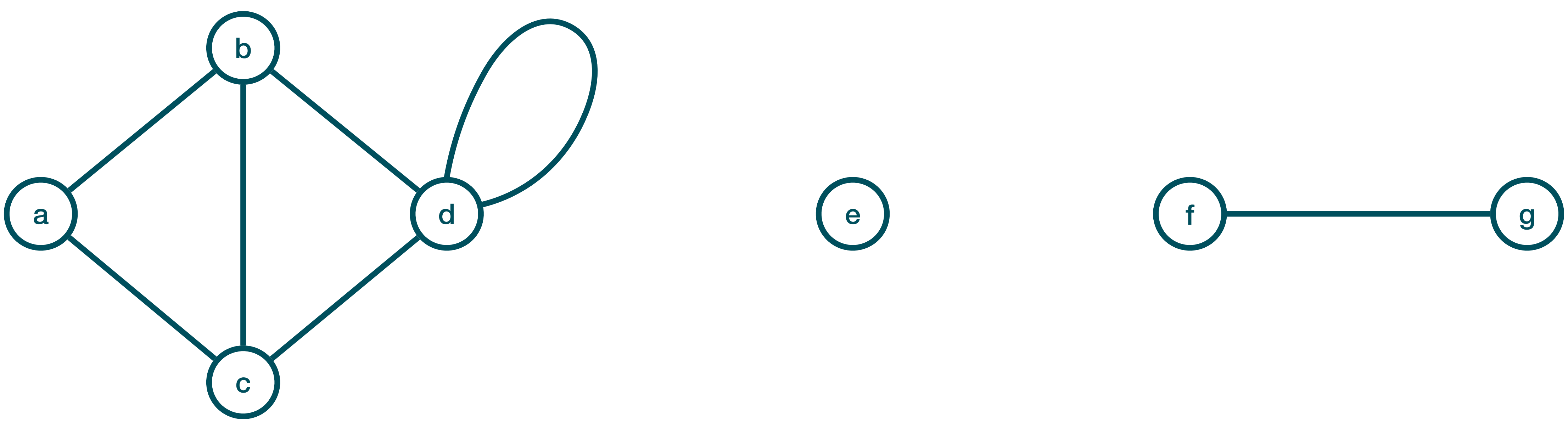
More definitions

An undirected graph G is connected if there is a path between any two vertices

A connected component of G is a maximal connected subgraph of G

Two vertices are adjacent if there is an edge linking the two

The undirected graph below is not connected. It rather has three connected components: C_1={a,b,c,d}; C_2={e}; C_3={f,g}

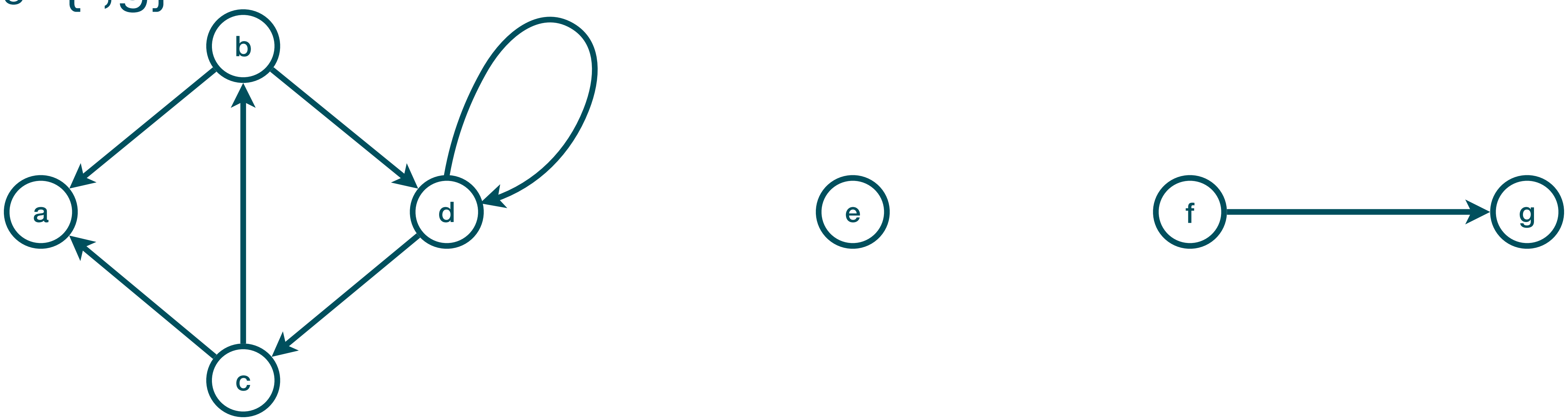


More definitions

A directed graph is strongly connected if there is a path between any two vertices. It is weakly connected if the underlying undirected graph is connected

Two vertices are in the same weakly connected component if they are connected by a path in the underlying unconnected graph

The directed graph below is not even weakly connected. It has three weakly connected components: C_1={a,b,c,d}; C_2={e}; C_3={f,g}

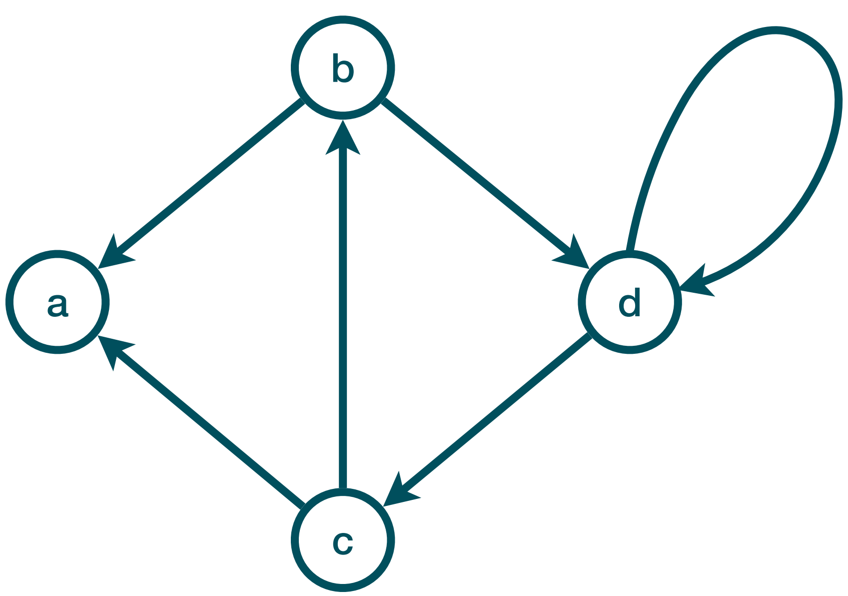


More definitions

A directed graph is strongly connected if there is a path between any two vertices. It is weakly connected if the underlying undirected graph is connected

Two vertices are in the same weakly connected component if they are connected by a path in the underlying unconnected graph

The directed graph below is weakly connected but not strongly connected: for example there is no path from a to b



More definitions

A directed graph is strongly connected if there is a path between any two vertices. It is weakly connected if the underlying undirected graph is connected

Two vertices are in the same weakly connected component if they are connected by a path in the underlying unconnected graph

Is the graph below strongly connected?

More definitions

A directed graph is strongly connected if there is a path between any two vertices. It is weakly connected if the underlying undirected graph is connected

Two vertices are in the same weakly connected component if they are connected by a path in the underlying unconnected graph

Is the graph below strongly connected?

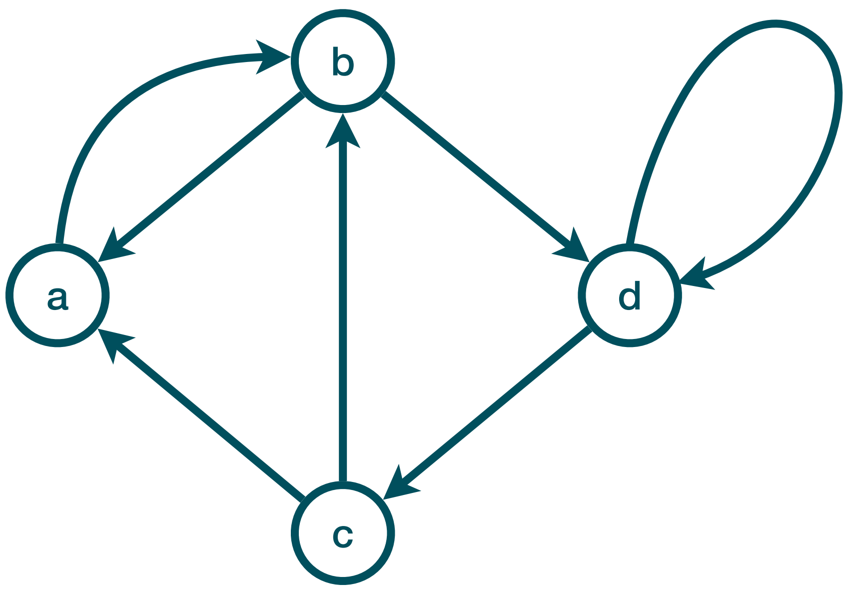
NO: there is no path from g to f

More definitions

A directed graph is strongly connected if there is a path between any two vertices. It is weakly connected if the underlying undirected graph is connected

Two vertices are in the same weakly connected component if they are connected by a path in the underlying unconnected graph

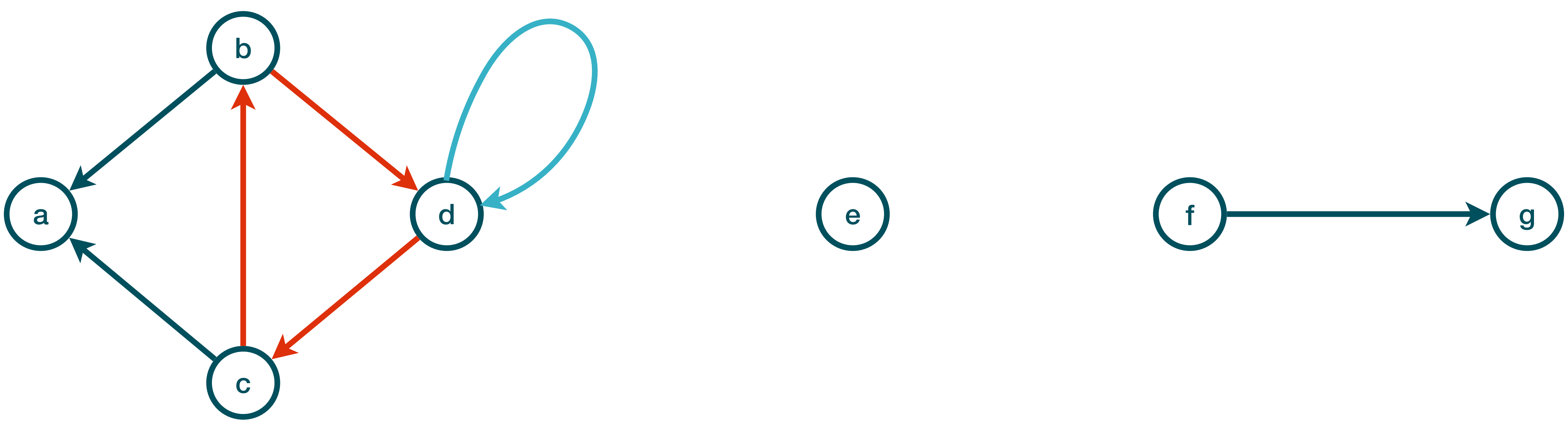
The directed graph below is strongly connected



More definitions

An (un)directed graph is acyclic if it does not contain any cycle

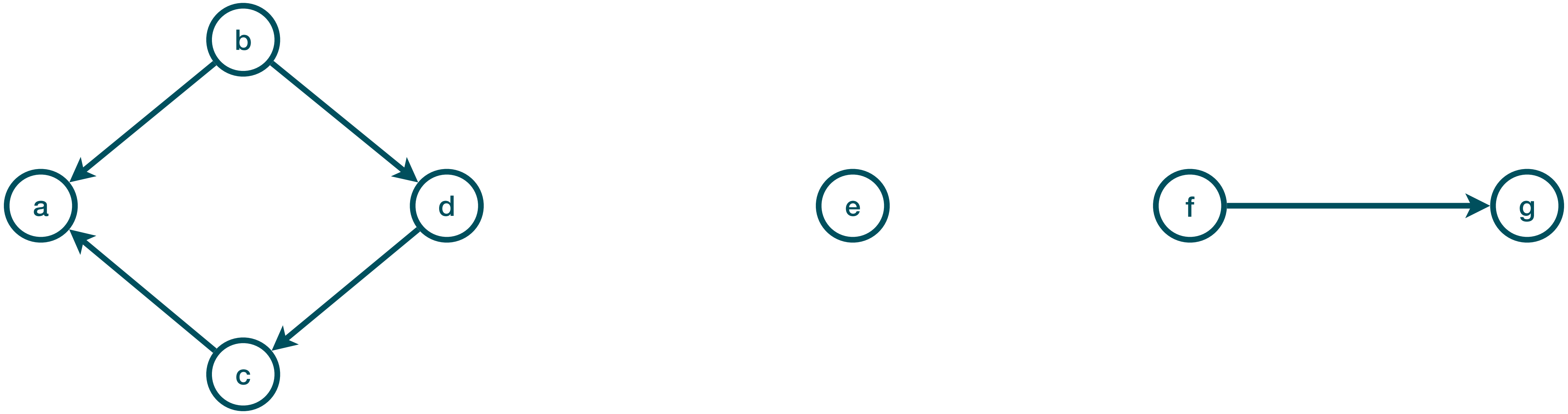
The directed graph below is not acyclic: it contains cycles (d,d) and (b,d)(d,c)(c,b)



More definitions

An (un)directed graph is acyclic if it does not contain any cycle

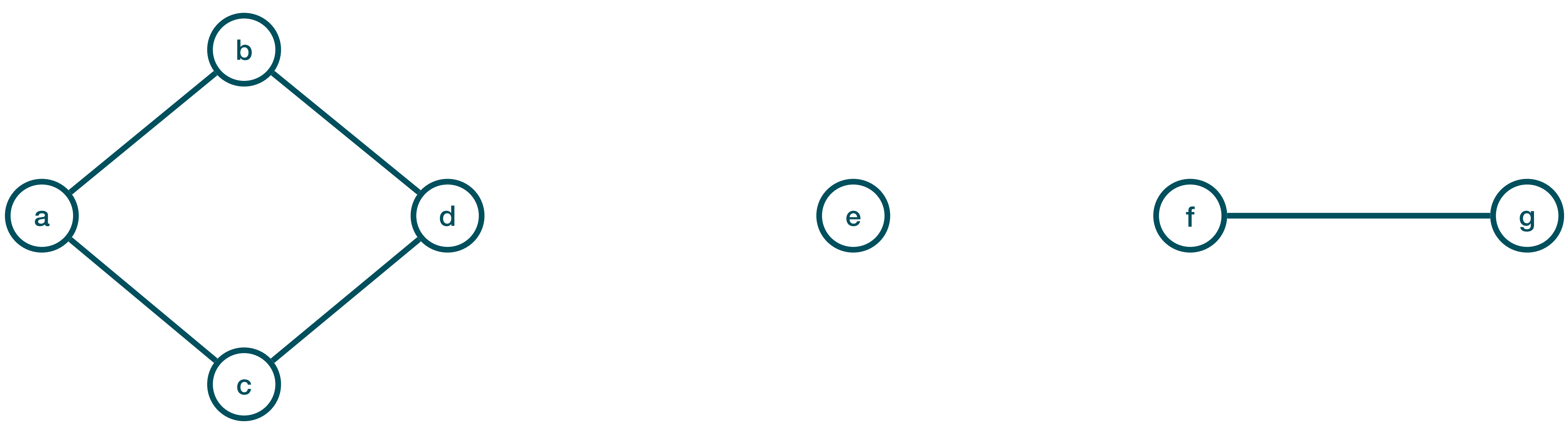
The directed graph below is acyclic: it does not contain any directed cycle



More definitions

An (un)directed graph is acyclic if it does not contain any cycle

The undirected graph below is not acyclic: it contains an undirected cycle {b,d}{d,c}{c,a}{a,b}



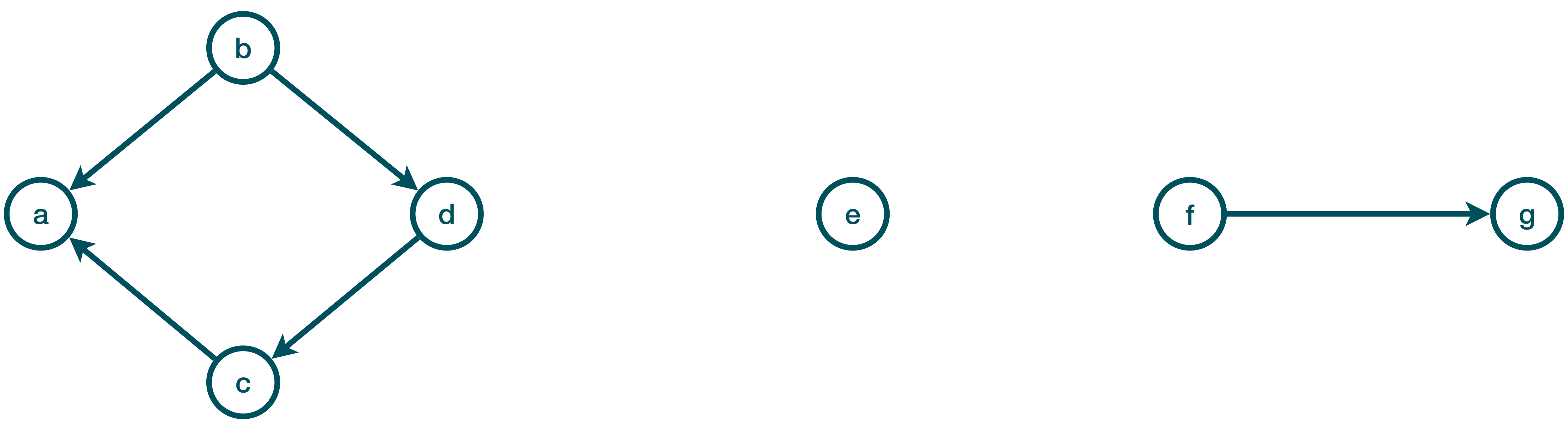
More definitions

An (un)directed graph is acyclic if it does not contain any cycle

Directed Acyclic Graphs are also known as DAGs and enjoy several properties. We will see one of them later.

A graph G=(V,E) is sparse if $|E|=O(|V|)$; is dense if $|E|=O(|V|^2)$

The graph below is a sparse DAG



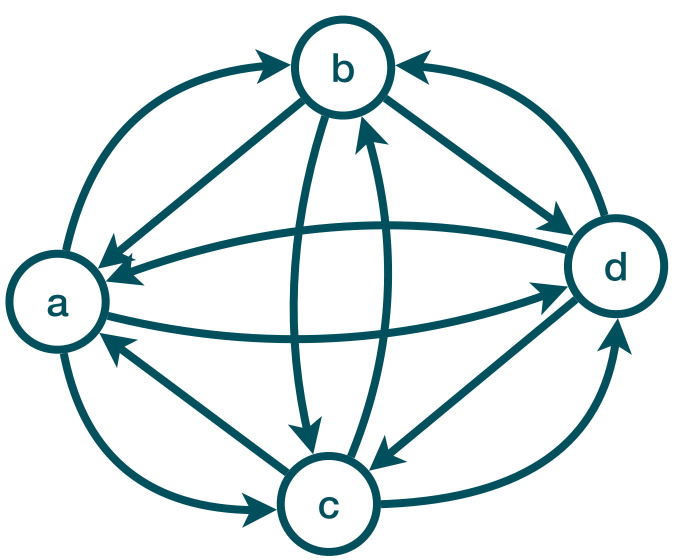
More definitions

An (un)directed graph is acyclic if it does not contain any cycle

Directed Acyclic Graphs are also known as DAGs and enjoy several properties. We will see one of them later.

A graph G=(V,E) is sparse if $|E|=O(|V|)$; is dense if $|E|=O(|V|^2)$

The graph below is dense



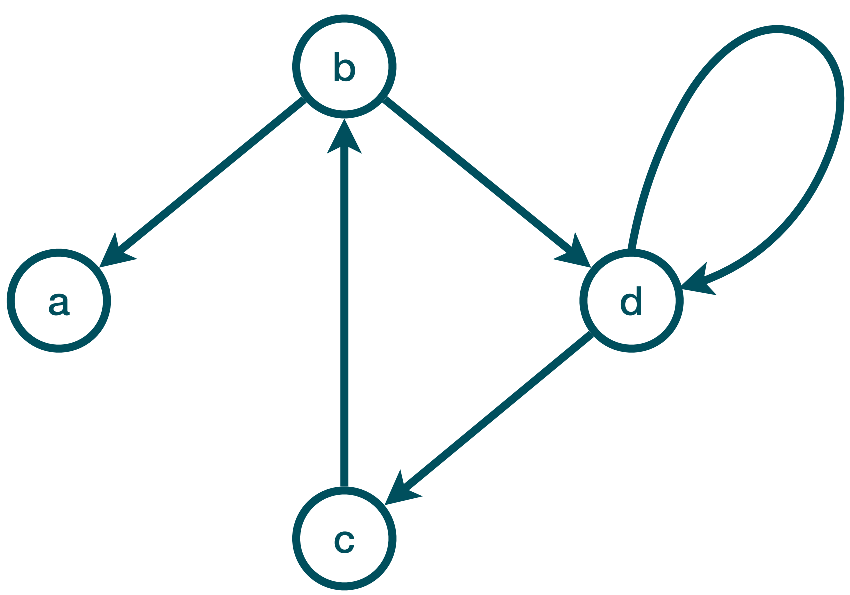
Graph representations

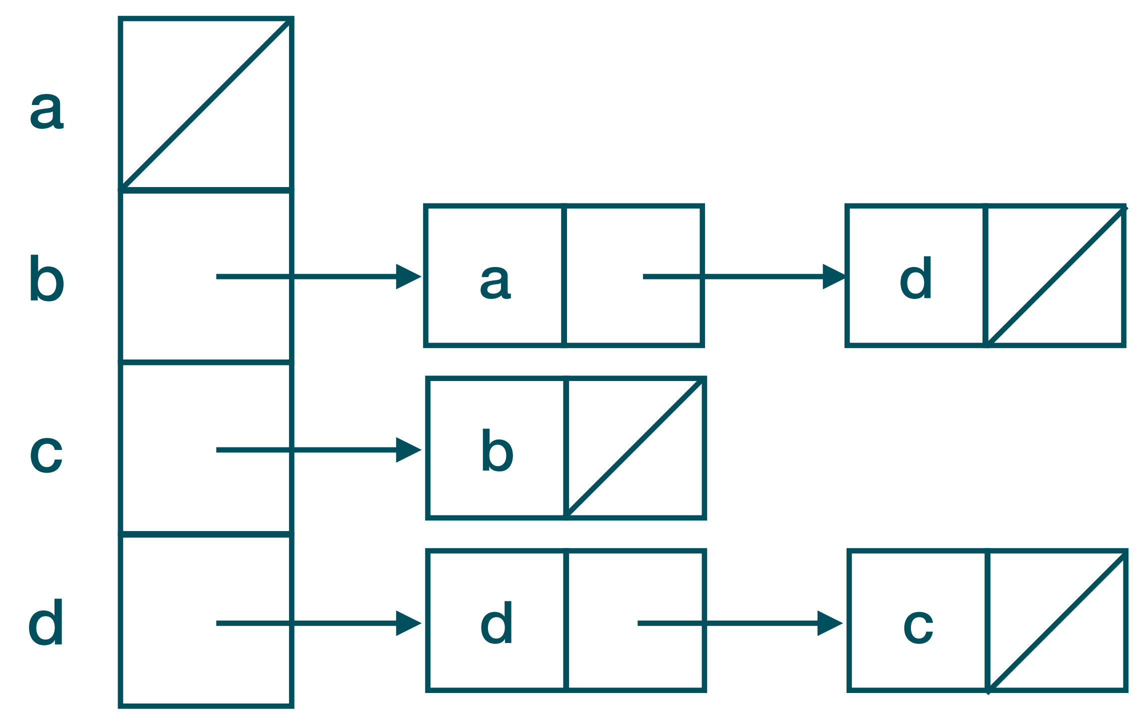
Reference: Chapter “Elementary Graph Algorithms” of: Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. *Introduction to algorithms*. (Chapter 22 of the third edition)

Representing graphs: adjacency lists

Adjacency lists are mostly used for sparse graphs

There is a linked list for each vertex v, containing all vertices adjacent to v

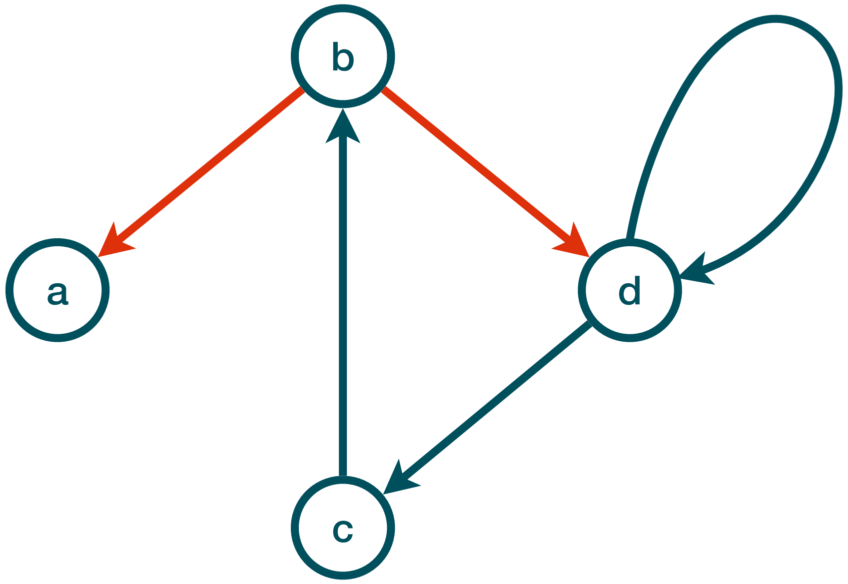


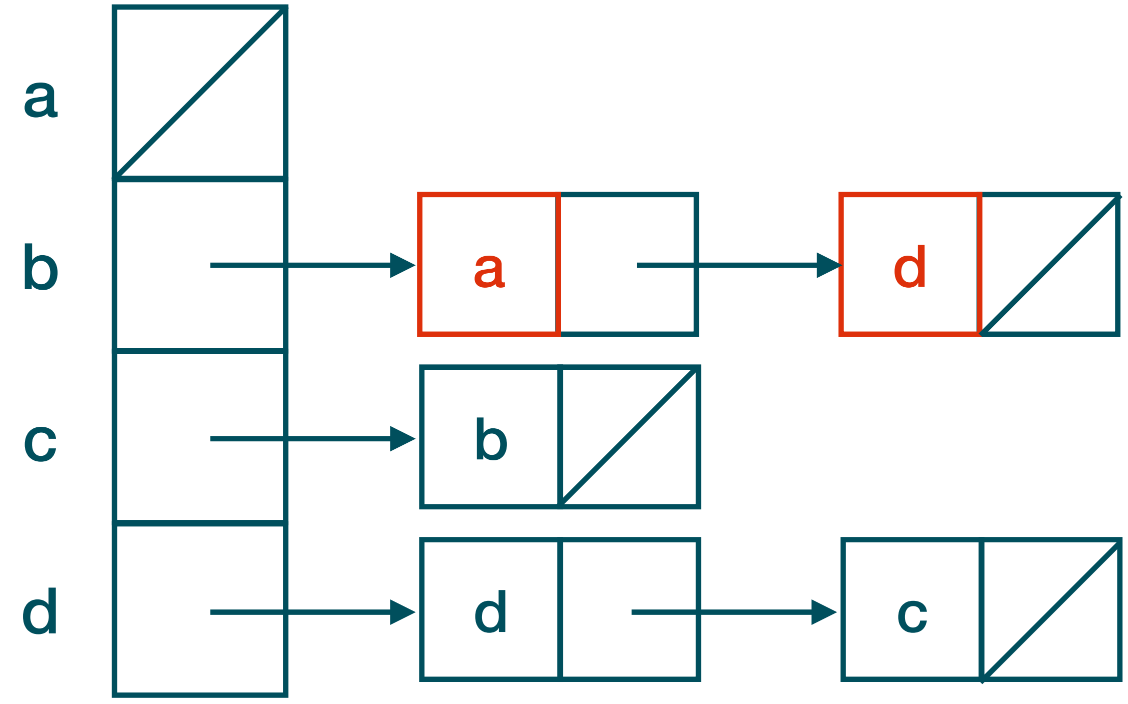


Representing graphs: adjacency lists

Adjacency lists are mostly used for sparse graphs

There is a linked list for each vertex v, containing all vertices adjacent to v



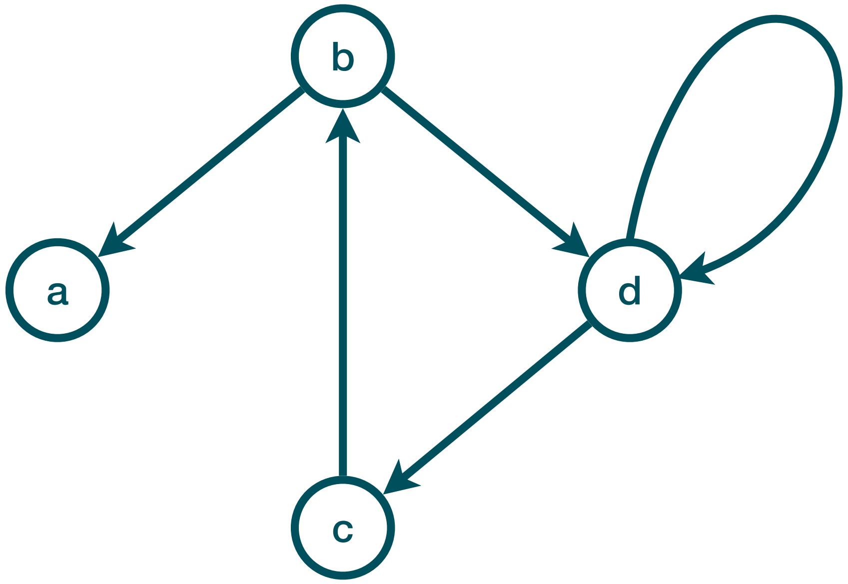


Representing graphs: adjacency matrix

Adjacency matrices are mostly used for dense graphs G=(V,E)

An adjacency matrix A has a row and a column for each vertex.

A[i,j]=1 if (i,j) ∈ E; A[i,j]=0 otherwise

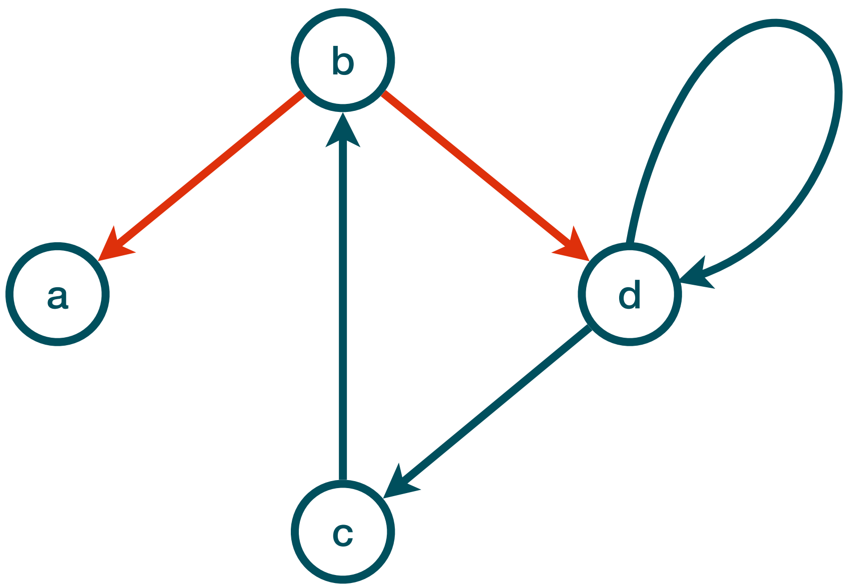


	a	b	c	d
a	0	0	0	0
b	1	0	0	1
c	0	1	0	0
d	0	0	1	1

Representing graphs: adjacency matrix

Adjacency matrices are mostly used for dense graphs G=(V,E)

An adjacency matrix A has a row and a column for each vertex.
A[i,j]=1 if (i,j) $\in$ E; A[i,j]=0 otherwise



	a	b	c	d
a	0	0	0	0
b	1	0	0	1
c	0	1	0	0
d	0	0	1	1

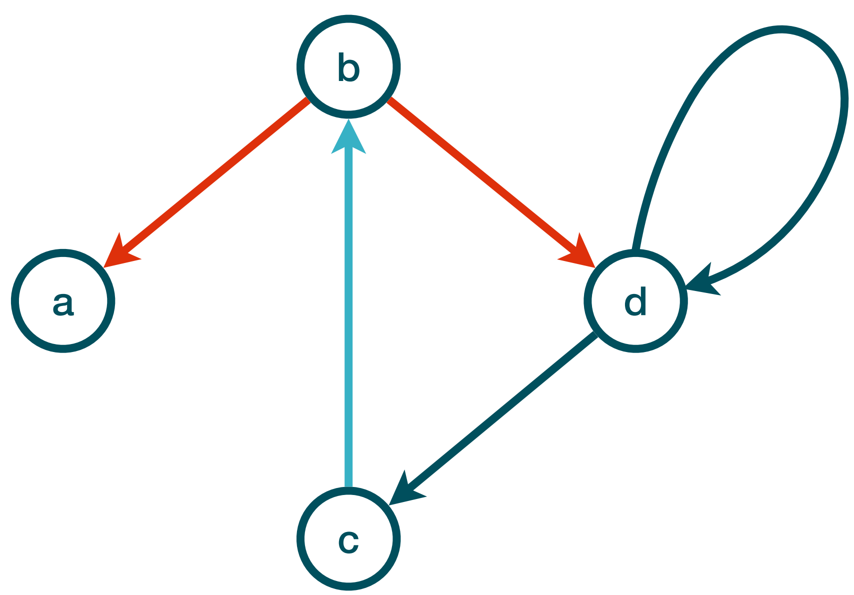
Outgoing from b

Representing graphs: adjacency matrix

Adjacency matrices are mostly used for dense graphs G=(V,E)

An adjacency matrix A has a row and a column for each vertex.

A[i,j]=1 if (i,j) ∈ E; A[i,j]=0 otherwise



	a	b	c	d
a	0	0	0	0
b	1	0	0	1
c	0	1	0	0
d	0	0	1	1

Outgoing from b

Incoming to b

Algorithms on Graphs

Reference: Chapter “Elementary Graph Algorithms” of: Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. *Introduction to algorithms*. (Chapter 22 of the third edition)

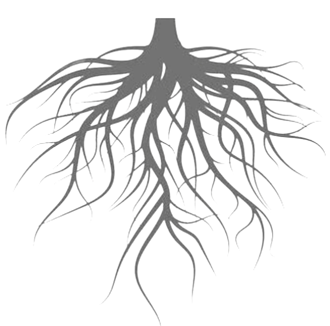
Graph traversals

The most fundamental task on a graph is to traverse it.

Graph traversal = visiting each vertex at least once

Two main ways of traversing both directed and undirected graphs:

1. Breadth-First Search (BFS)



2. Depth-First Search (DFS)

Breadth-First Search

The visiting order is related to the distance from a source node: the closer a node to the source, the sooner it will be visited

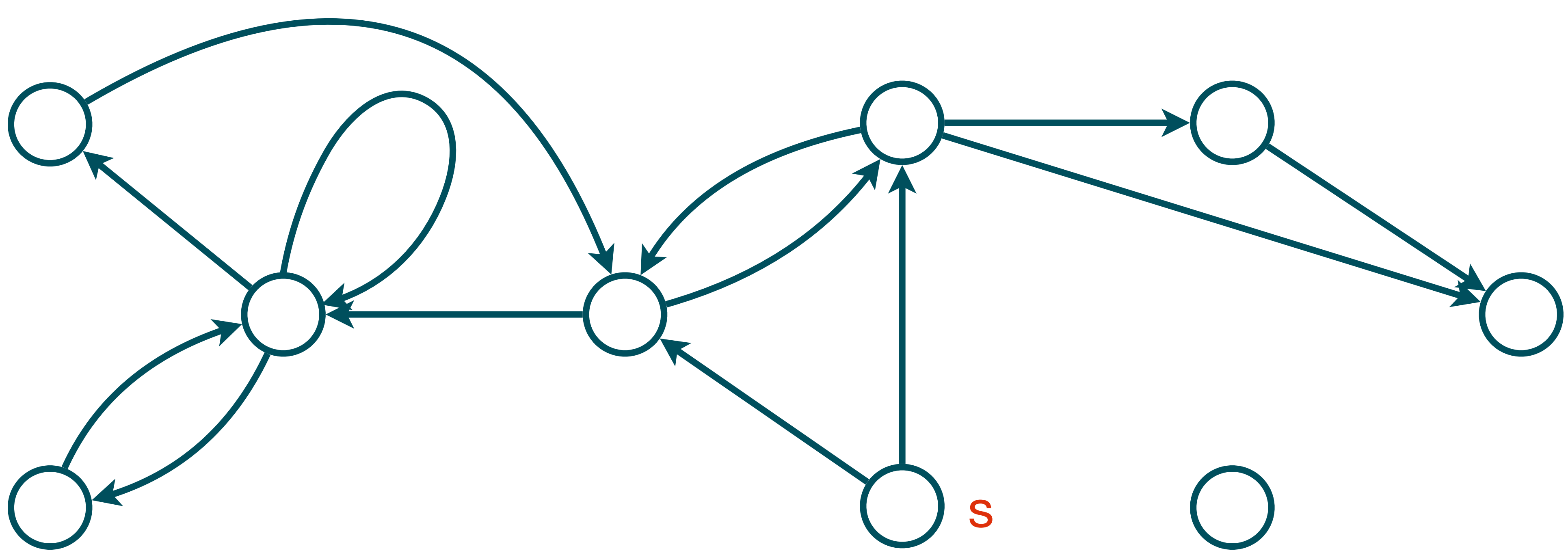
BFS produces a breadth-first tree: the tree consisting of the shortest paths from the source to any reachable node

Breadth-First Search

The visiting order is related to the distance from a source node: the closer a node to the source, the sooner it will be visited

BFS produces a breadth-first tree: the tree consisting of the shortest paths from the source to any reachable node

White nodes have not been discovered yet;

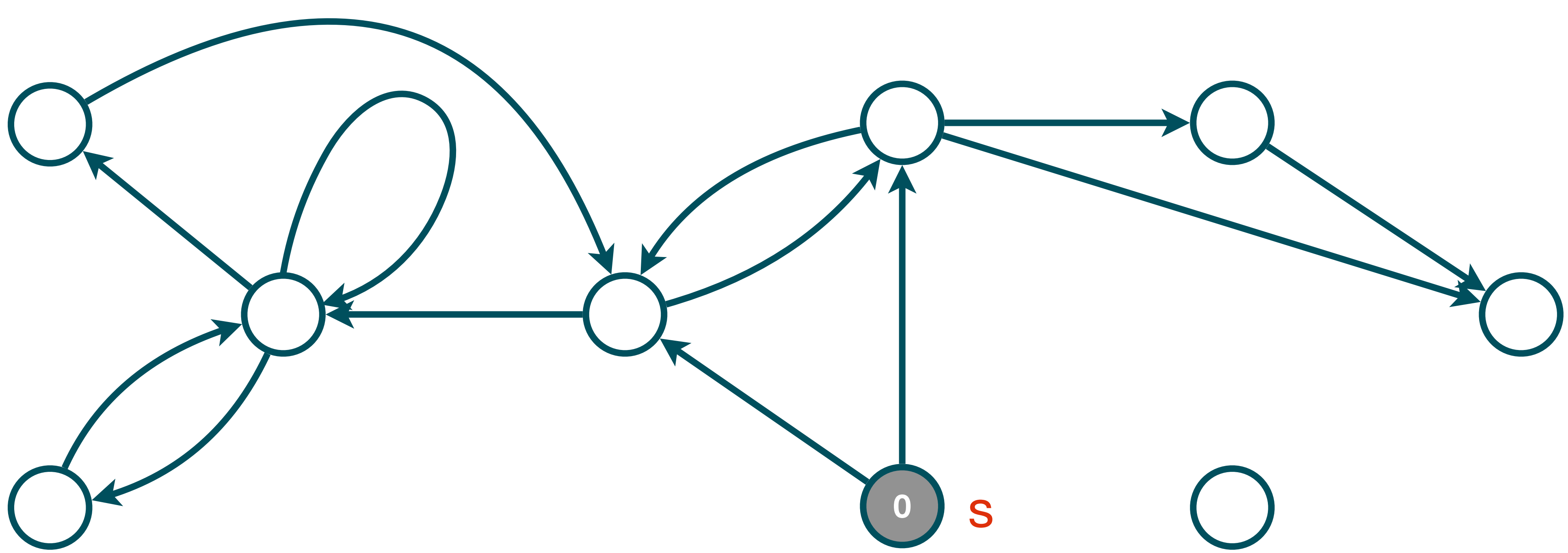


Breadth-First Search

The visiting order is related to the distance from a source node: the closer a node to the source, the sooner it will be visited

BFS produces a breadth-first tree: the tree consisting of the shortest paths from the source to any reachable node

White nodes have not been discovered yet; gray nodes have been discovered but have undiscovered neighbours;

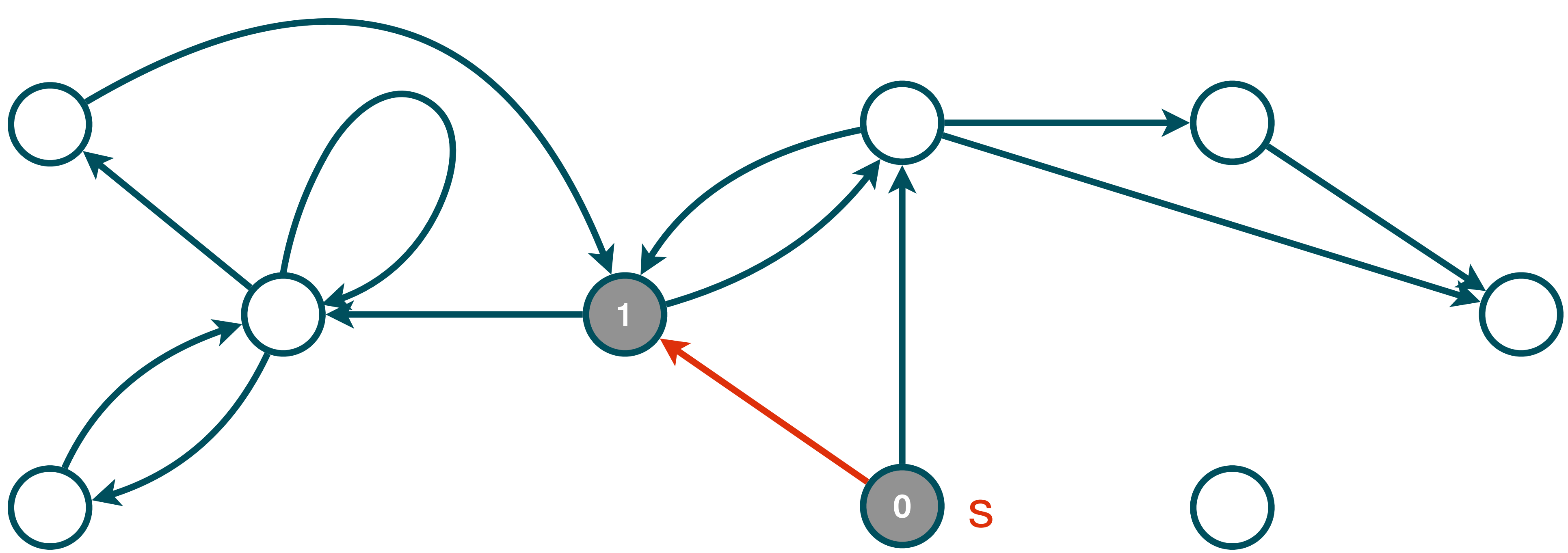


Breadth-First Search

The visiting order is related to the distance from a source node: the closer a node to the source, the sooner it will be visited

BFS produces a breadth-first tree: the tree consisting of the shortest paths from the source to any reachable node

White nodes have not been discovered yet; gray nodes have been discovered but have undiscovered neighbours;

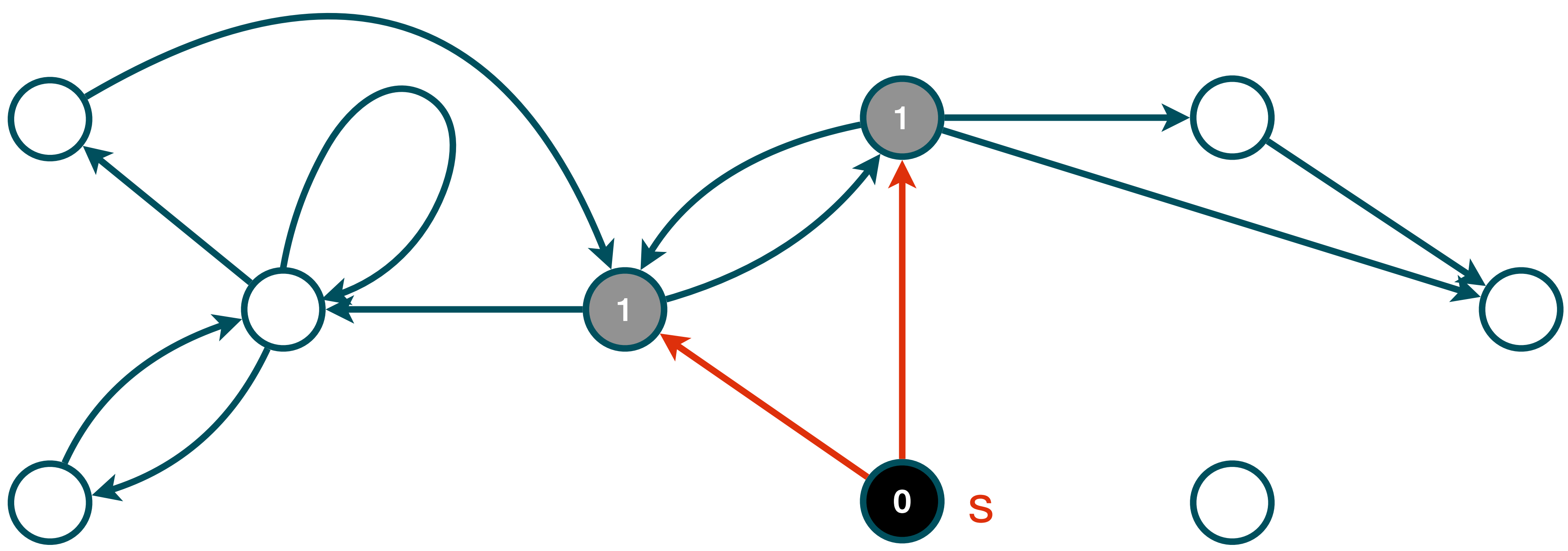


Breadth-First Search

The visiting order is related to the distance from a source node: the closer a node to the source, the sooner it will be visited

BFS produces a breadth-first tree: the tree consisting of the shortest paths from the source to any reachable node

White nodes have not been discovered yet; gray nodes have been discovered but have undiscovered neighbours; black nodes have been discovered and their neighbours too.

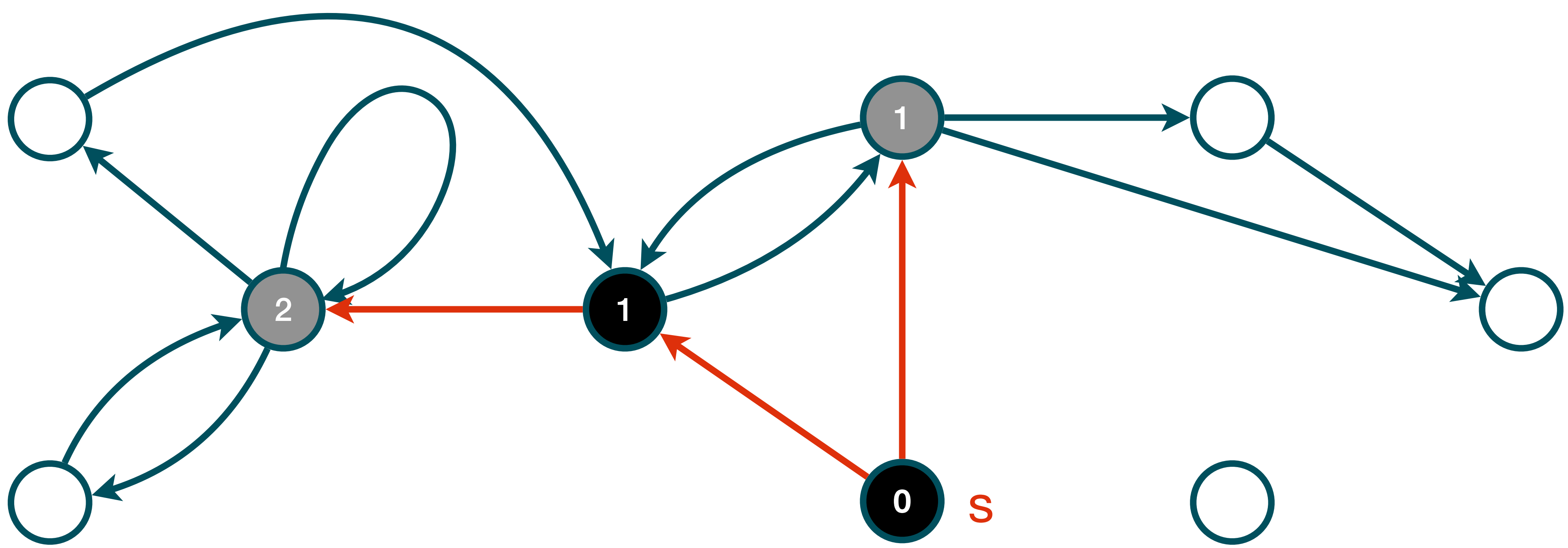


Breadth-First Search

The visiting order is related to the distance from a source node: the closer a node to the source, the sooner it will be visited

BFS produces a breadth-first tree: the tree consisting of the shortest paths from the source to any reachable node

White nodes have not been discovered yet; gray nodes have been discovered but have undiscovered neighbours; black nodes have been discovered and their neighbours too.

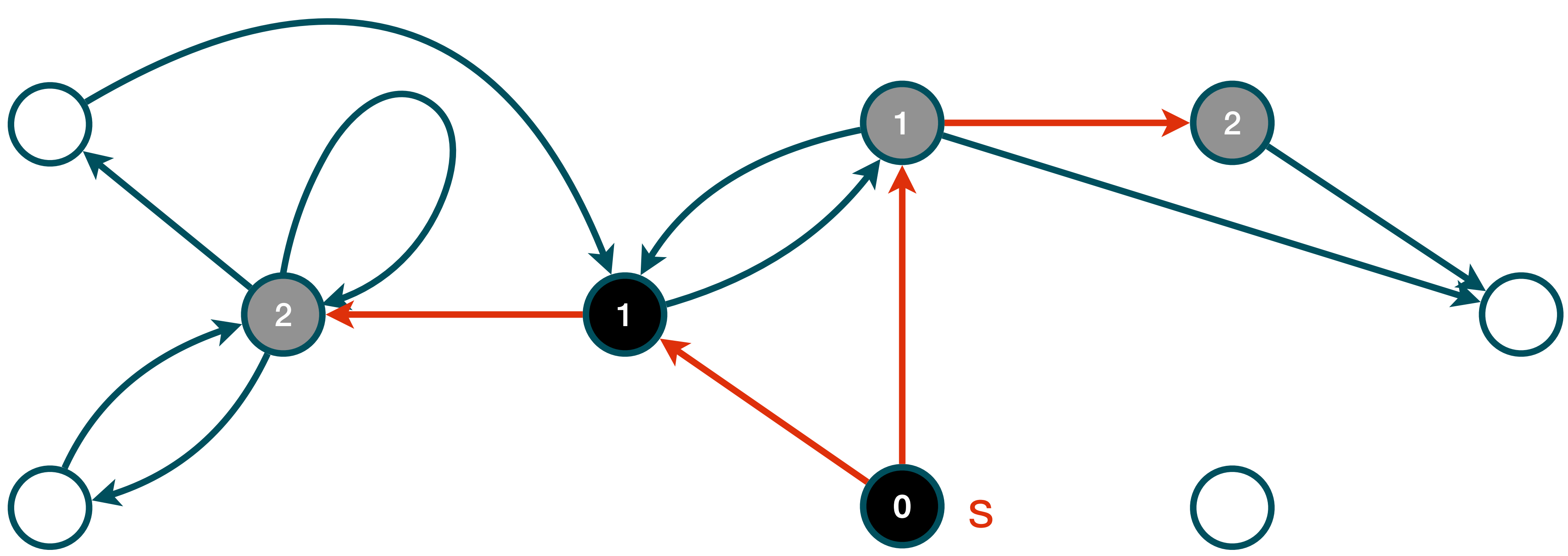


Breadth-First Search

The visiting order is related to the distance from a source node: the closer a node to the source, the sooner it will be visited

BFS produces a breadth-first tree: the tree consisting of the shortest paths from the source to any reachable node

White nodes have not been discovered yet; gray nodes have been discovered but have undiscovered neighbours; black nodes have been discovered and their neighbours too.

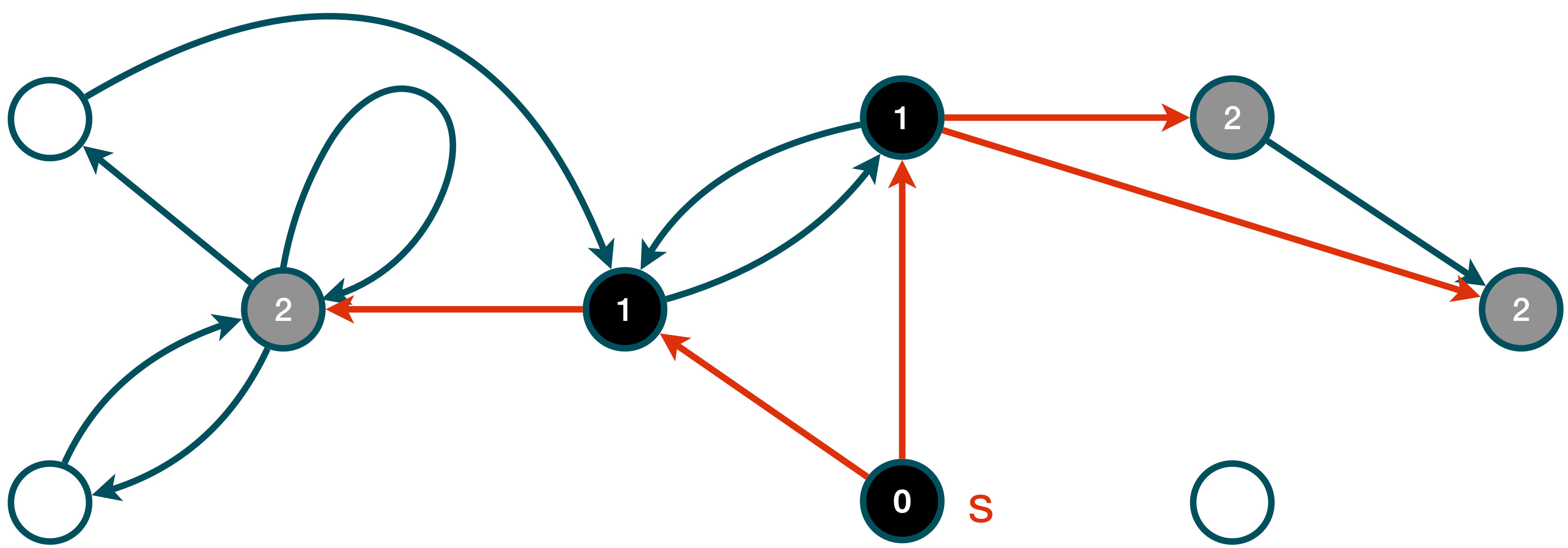


Breadth-First Search

The visiting order is related to the distance from a source node: the closer a node to the source, the sooner it will be visited

BFS produces a breadth-first tree: the tree consisting of the shortest paths from the source to any reachable node

White nodes have not been discovered yet; gray nodes have been discovered but have undiscovered neighbours; black nodes have been discovered and their neighbours too.

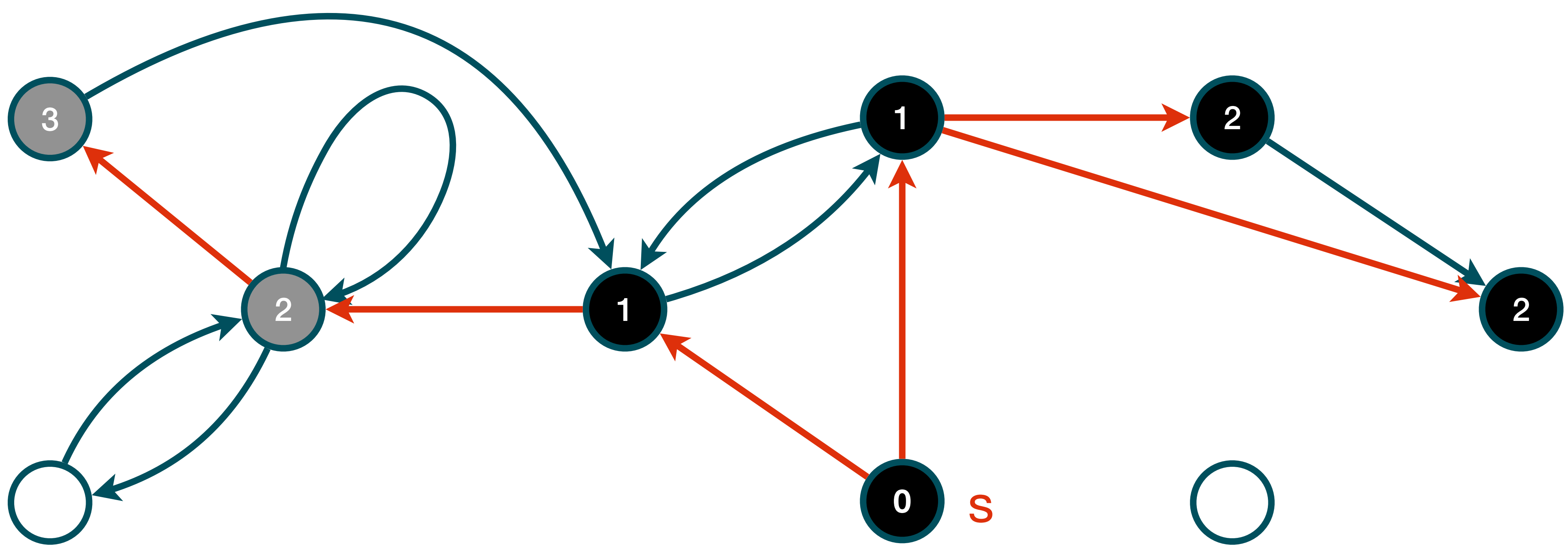


Breadth-First Search

The visiting order is related to the distance from a source node: the closer a node to the source, the sooner it will be visited

BFS produces a breadth-first tree: the tree consisting of the shortest paths from the source to any reachable node

White nodes have not been discovered yet; gray nodes have been discovered but have undiscovered neighbours; black nodes have been discovered and their neighbours too.

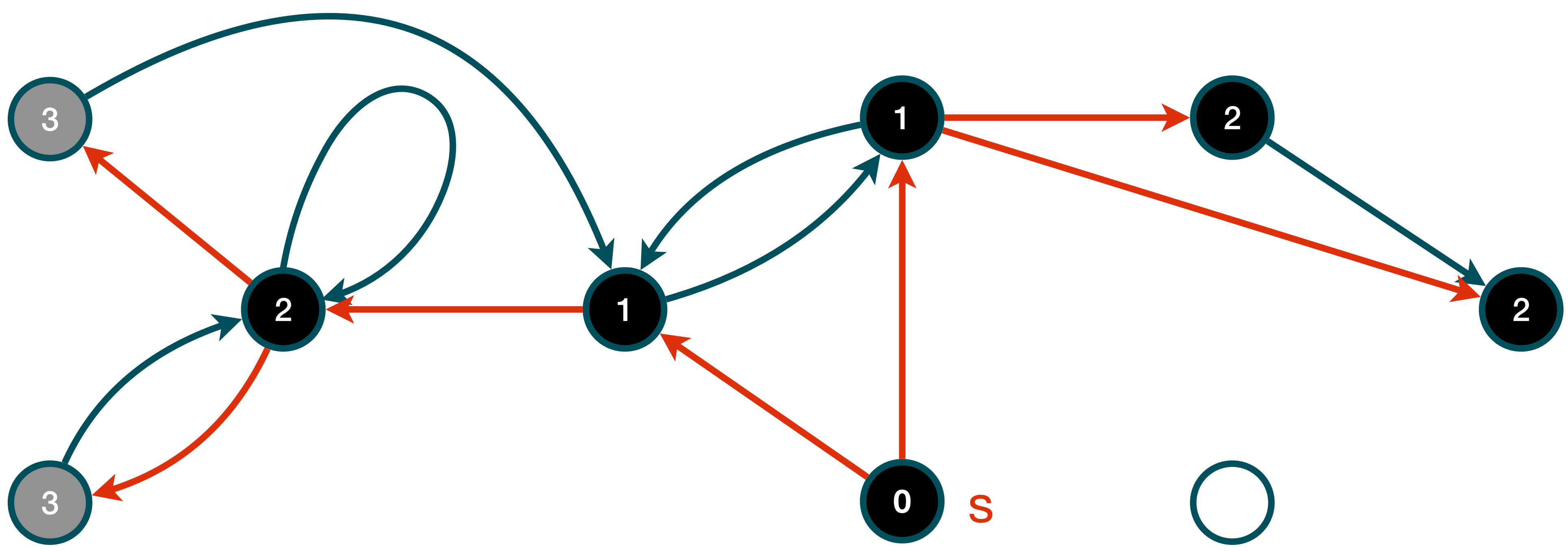


Breadth-First Search

The visiting order is related to the distance from a source node: the closer a node to the source, the sooner it will be visited

BFS produces a breadth-first tree: the tree consisting of the shortest paths from the source to any reachable node

White nodes have not been discovered yet; gray nodes have been discovered but have undiscovered neighbours; black nodes have been discovered and their neighbours too.

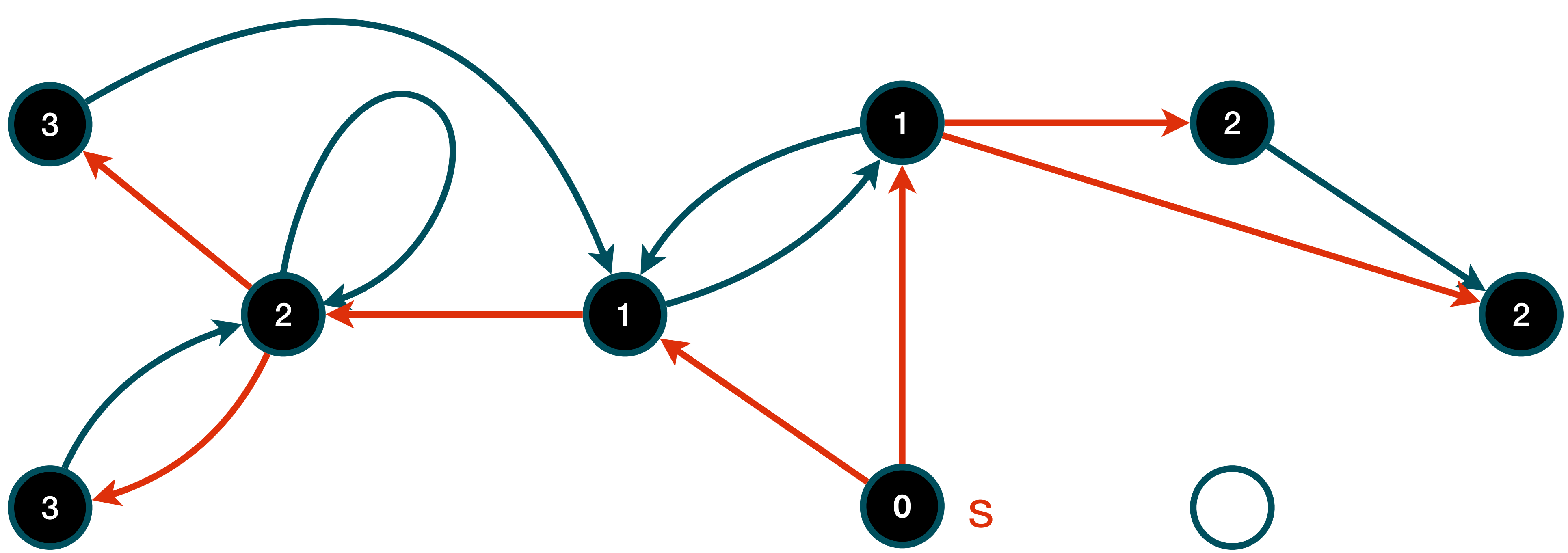


Breadth-First Search

The visiting order is related to the distance from a source node: the closer a node to the source, the sooner it will be visited

BFS produces a breadth-first tree: the tree consisting of the shortest paths from the source to any reachable node

White nodes have not been discovered yet; gray nodes have been discovered but have undiscovered neighbours; black nodes have been discovered and their neighbours too.

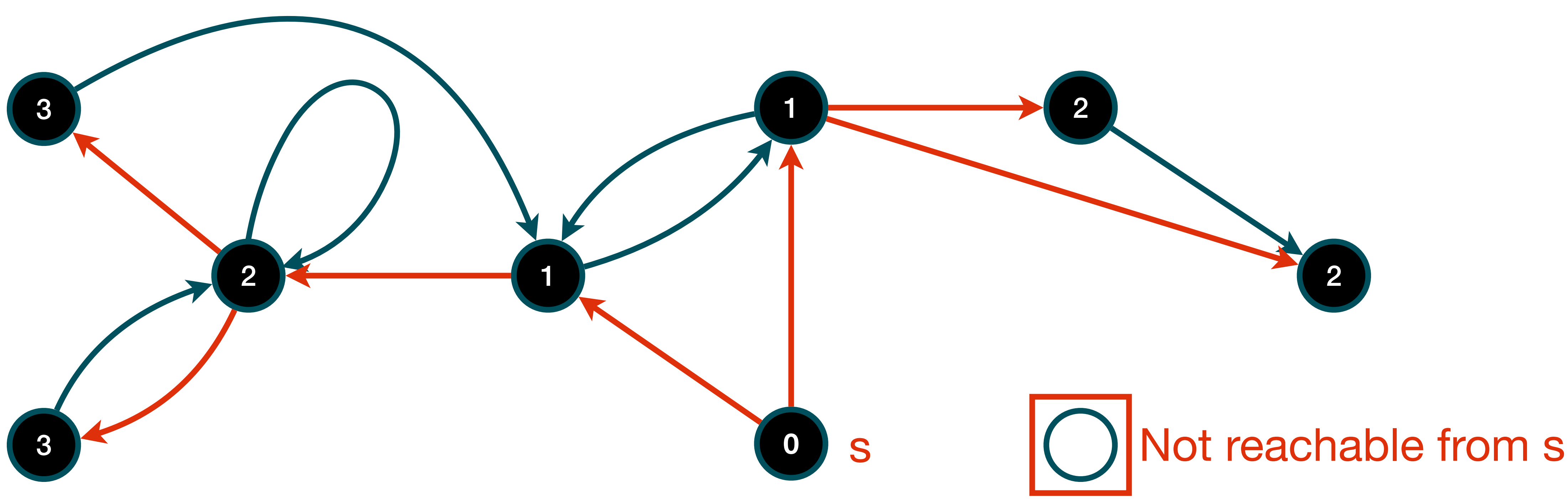


Breadth-First Search

The visiting order is related to the distance from a source node: the closer a node to the source, the sooner it will be visited

BFS produces a breadth-first tree: the tree consisting of the shortest paths from the source to any reachable node

White nodes have not been discovered yet; gray nodes have been discovered but have undiscovered neighbours; black nodes have been discovered and their neighbours too.



BFS: Pseudocode

BFS(G,s) - G is represented by the adjacency lists Adj[·] of its vertices

```
for each u ∈ V∖{s}
        u.color←white;
        u.distance←∞;
s.color←gray;
s.distance←0;
Q←∅;
enqueue(Q,s);
while Q ≠ ∅
        u←dequeue(Q);
        for each v ∈ Adj[u]
                if v.color = white
                    v.color←gray;
                    v.distance←u.distance + 1;
                    enqueue(Q,v);
        u.color←black;
```

Initialisation

Visit of the source

Visit of the other vertices

BFS: Complexity

BFS(G,s) - G is represented by the adjacency lists $\mathrm{Adj}[\cdot]$ of its vertices

for each $u \in V \setminus \{s\}$
    $u.color \leftarrow$ white;
    $u.distance \leftarrow \infty$;
$s.color \leftarrow$ gray;
$s.distance \leftarrow 0$;
$Q \leftarrow \emptyset$;
enqueue(Q,s);
while $Q \neq \emptyset$
    $u \leftarrow$ dequeue(Q);
    for each $v \in \mathrm{Adj}[u]$
        if $v.color =$ white
            $v.color \leftarrow$ gray;
            $v.distance \leftarrow u.distance + 1$;
            enqueue(Q,v);
    $u.color \leftarrow$ black;

Initialisation: $O(|V|)$

Visit of the source: $O(1)$

$O(1)$

$O(|\mathrm{Adj}[u]|)$

Visit of the other vertices: each iteration of the **for** loop enqueues $v \in \mathrm{Adj}[u]$ only if it is white, and it immediately turns its color to gray $\Longrightarrow$ each vertex is inserted in Q at most once.
Cost of the **while** loop:

$$O\left(\sum_{u \in V} |\mathrm{Adj}[u]|\right) = O(|E|)$$

BFS: Properties

Lemma 1. The time complexity of BFS is $O(|V|+|E|)$ (linear in the size of the adjacency-list representation of G)

Lemma 2. Let $Q=[v_1,\ldots,v_n]$ be the queue at any iteration of BFS. Then $v_i.distance \leq v_{i+1}.distance$ and $v_n.distance \leq v_1.distance+1$, for all $i=1,\ldots,n-1$

Lemma 2 tells us that, at any iteration, if the head node of Q is at distance d from s, Q only contains nodes at distance d or $d+1$ from s; possible nodes at distance $d+2$ will be only enqueued after all nodes at distance d have been dequeued.

Lemma 3. Let $d(v,s)$ be the distance between v and s, for any $v \in V$. Then:

(i) $v.distance \neq \infty \iff v$ is reachable from s

(ii) if $v.distance \neq \infty \implies v.distance = d(v,s)$

BFS: Exercise

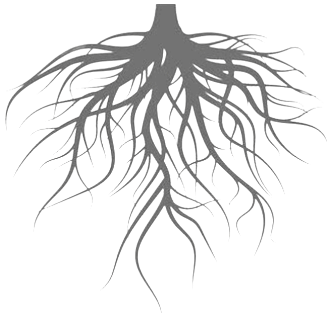
We said that BFS can produce a breadth-first tree (the tree consisting of the shortest paths from the source to any reachable node). More precisely, a breadth-first tree is defined as follows:

Definition 1. The root of the tree is the source *s* of BFS; its nodes are the nodes of G reachable from *s*; its edges are the edges of G traversed during BFS; the unique path from the root to a node *v* is the shortest path from *s* to *v* in G.

Exercise 1. Our pseudocode computes all the information needed to construct the breadth-first tree. Can you complement it so that it explicitly construct and output such tree?

Hint: it suffices to store the correct predecessor (ancestor in the tree) for each node. The BF tree consists of the red edges in our example.

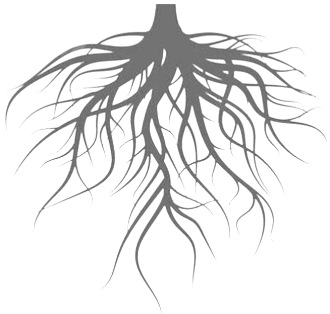
Depth-First Search



DFS searches “deeper” in G whenever possible:

- It selects a source node *s* and follows a path from *s* as long as possible, by adding only non-visited nodes
- It repeats the same process on each of the branches deviating from the path of the previous step
- If some nodes remain non-visited, a node among them is selected as new source and the whole procedure is repeated until every node has been visited

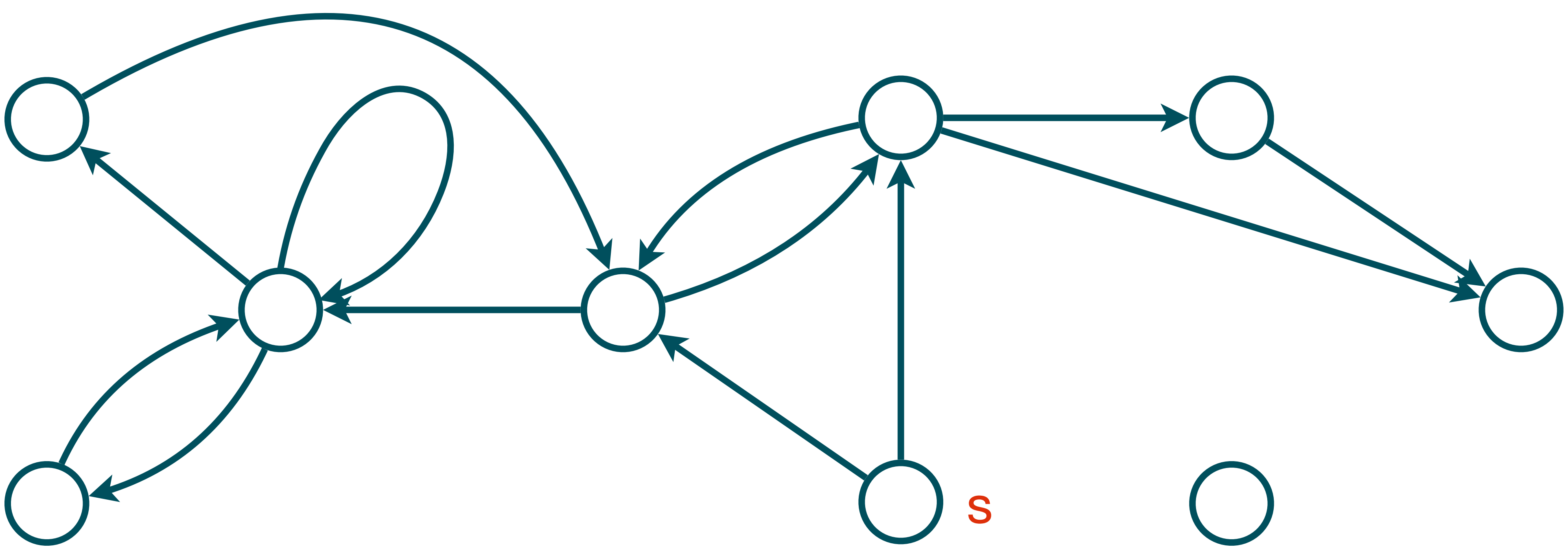
Depth-First Search



Much like BFS, DFS colors the nodes of G during the visit.

Again, white nodes have not been visited yet; gray nodes have been discovered but have undiscovered neighbours; black nodes have been discovered and their neighbours too.

DFS assigns two timestamps to each node *v*: *v.d* records when *v* becomes gray, *v.f* records when it becomes black.



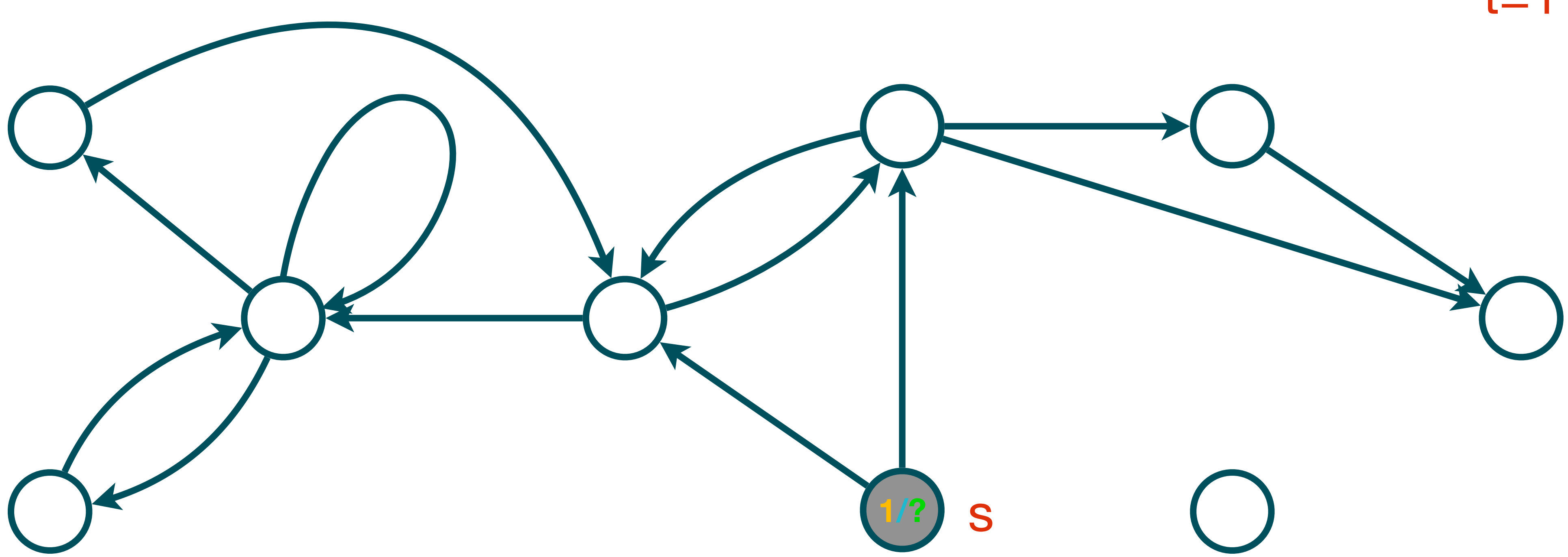
Depth-First Search

Much like BFS, DFS colors the nodes of G during the visit.

Again, white nodes have not been visited yet; gray nodes have been discovered but have undiscovered neighbours; black nodes have been discovered and their neighbours too.

DFS assigns two timestamps to each node *v*: *v.d* records when *v* becomes gray, *v.f* records when it becomes black.

t=1

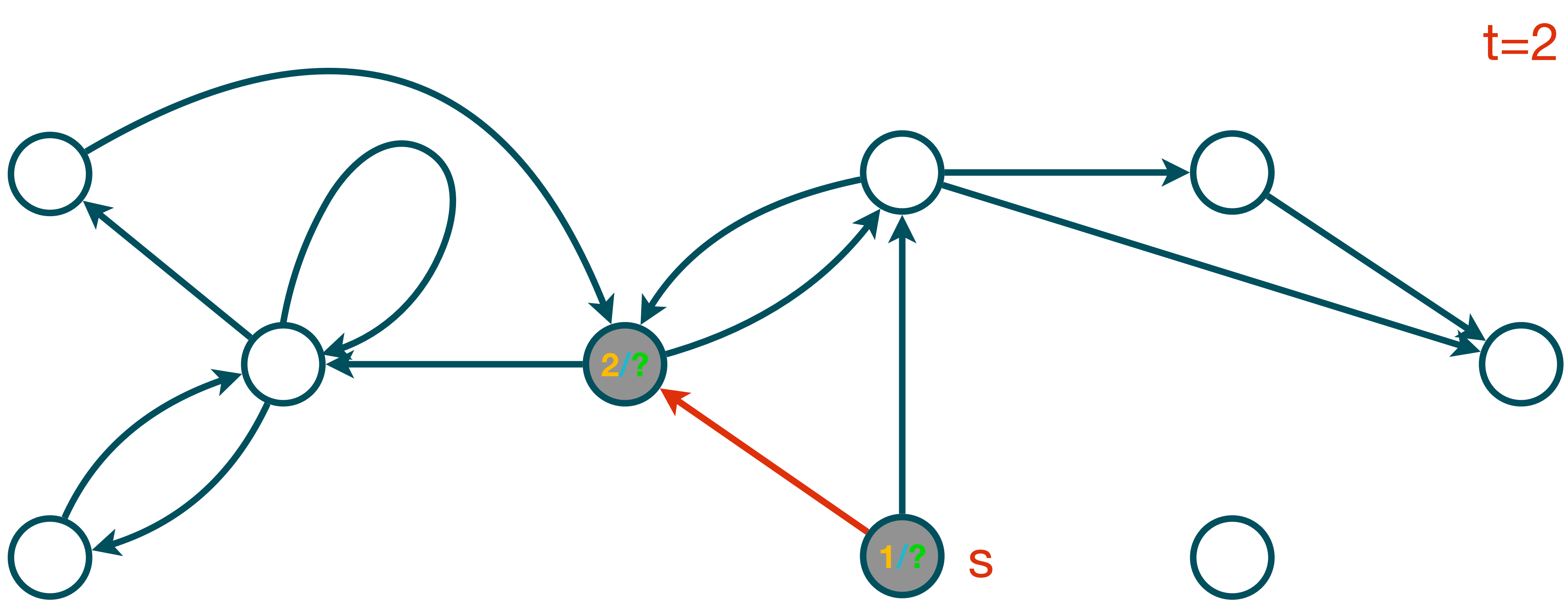


Depth-First Search

Much like BFS, DFS colors the nodes of G during the visit.

Again, white nodes have not been visited yet; gray nodes have been discovered but have undiscovered neighbours; black nodes have been discovered and their neighbours too.

DFS assigns two timestamps to each node *v*: *v.d* records when *v* becomes gray, *v.f* records when it becomes black.

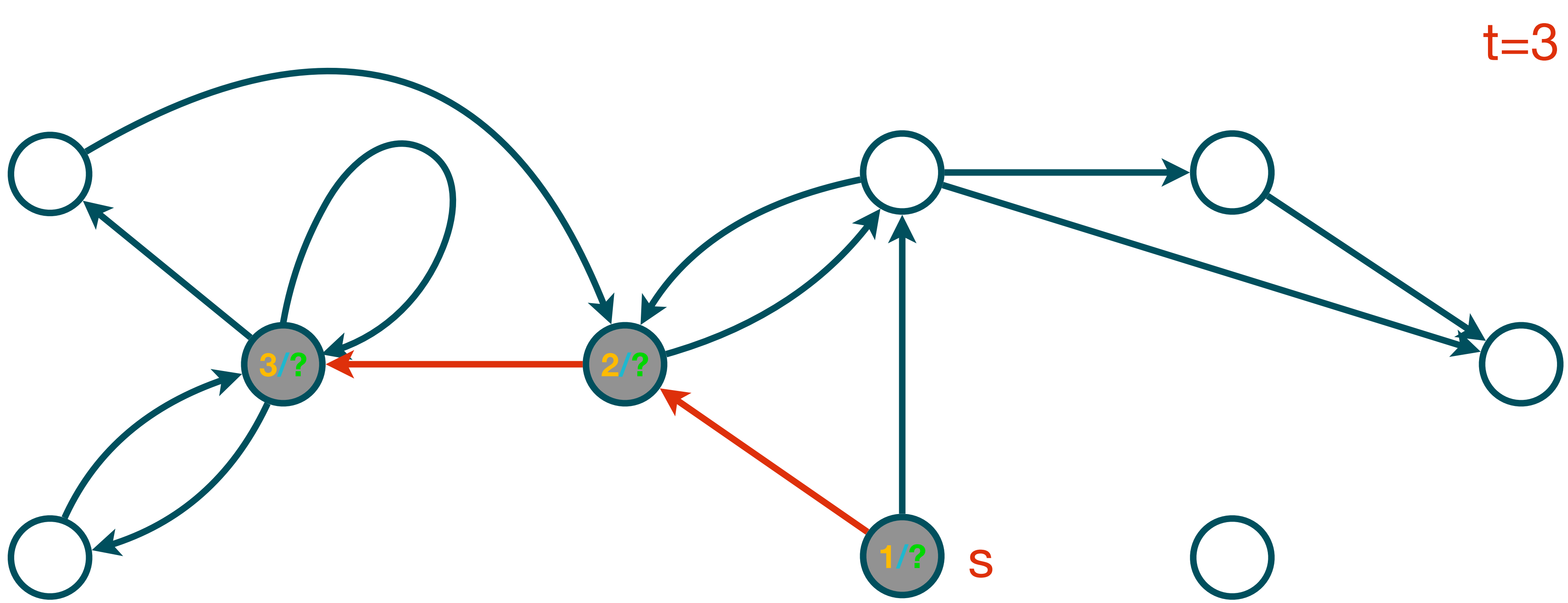


Depth-First Search

Much like BFS, DFS colors the nodes of G during the visit.

Again, white nodes have not been visited yet; gray nodes have been discovered but have undiscovered neighbours; black nodes have been discovered and their neighbours too.

DFS assigns two timestamps to each node *v*: *v.d* records when *v* becomes gray, *v.f* records when it becomes black.

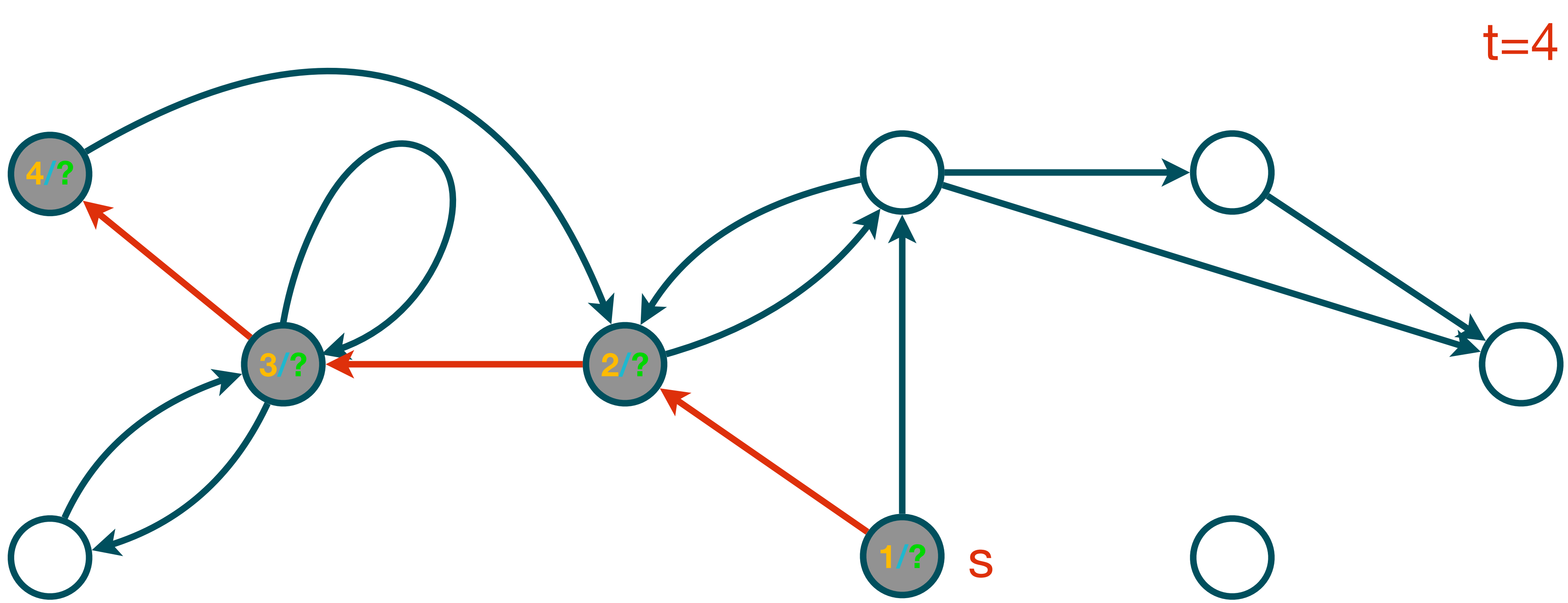


Depth-First Search

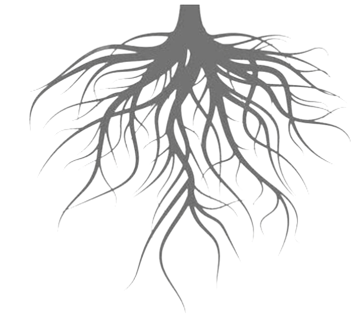
Much like BFS, DFS colors the nodes of G during the visit.

Again, white nodes have not been visited yet; gray nodes have been discovered but have undiscovered neighbours; black nodes have been discovered and their neighbours too.

DFS assigns two timestamps to each node *v*: *v.d* records when *v* becomes gray, *v.f* records when it becomes black.



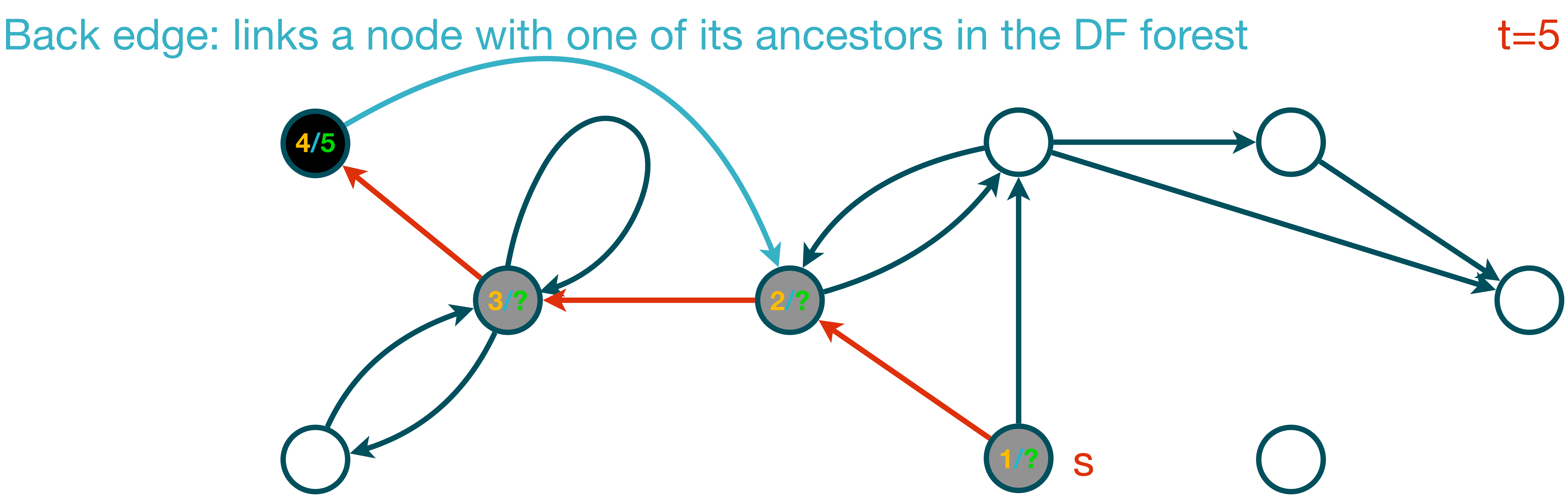
Depth-First Search



Much like BFS, DFS colors the nodes of G during the visit.

Again, white nodes have not been visited yet; gray nodes have been discovered but have undiscovered neighbours; black nodes have been discovered and their neighbours too.

DFS assigns two timestamps to each node *v*: *v.d* records when *v* becomes gray, *v.f* records when it becomes black.

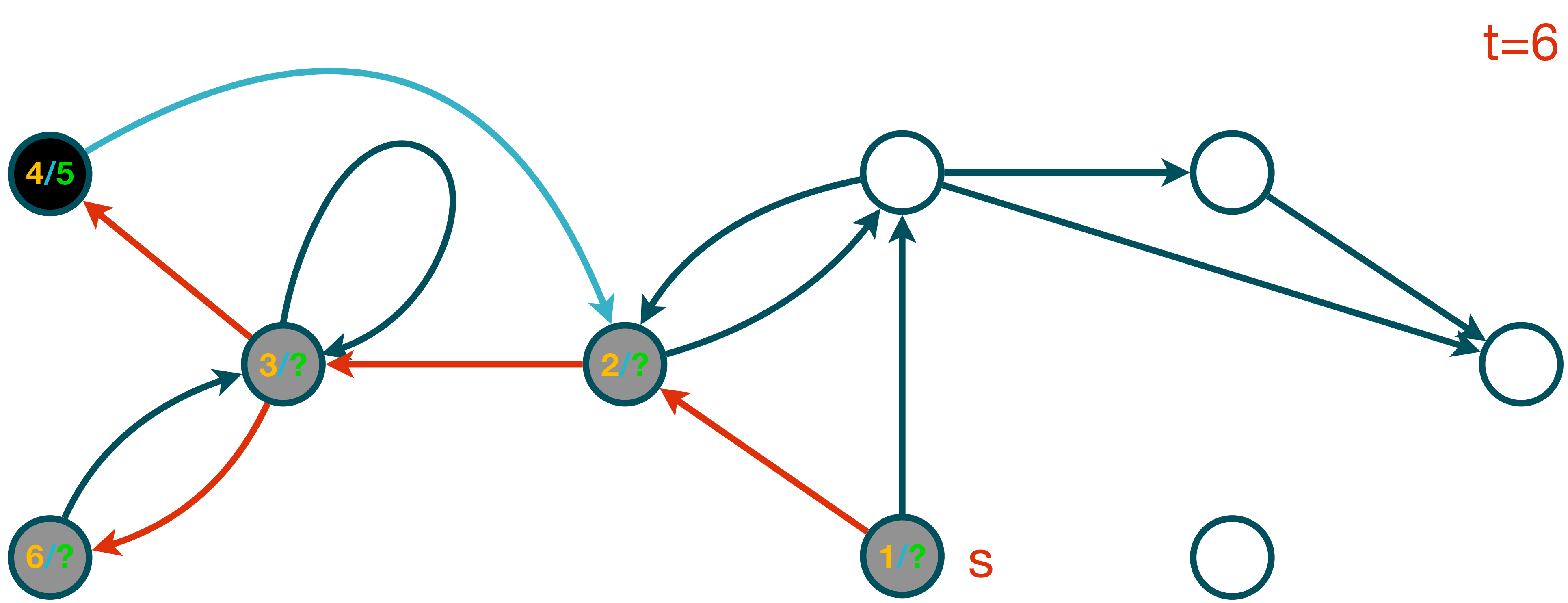


Depth-First Search

Much like BFS, DFS colors the nodes of G during the visit.

Again, white nodes have not been visited yet; gray nodes have been discovered but have undiscovered neighbours; black nodes have been discovered and their neighbours too.

DFS assigns two timestamps to each node *v*: *v.d* records when *v* becomes gray, *v.f* records when it becomes black.



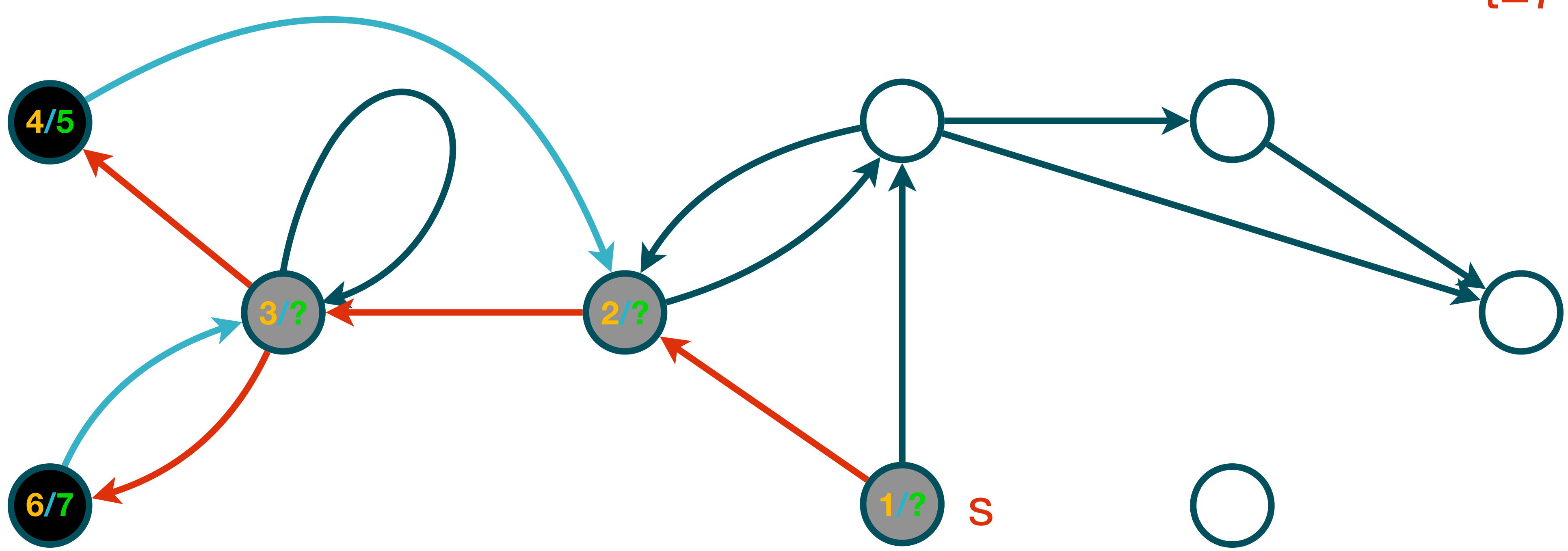
Depth-First Search

Much like BFS, DFS colors the nodes of G during the visit.

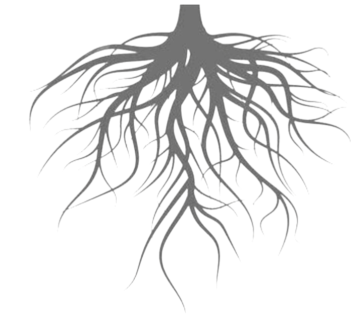
Again, white nodes have not been visited yet; gray nodes have been discovered but have undiscovered neighbours; black nodes have been discovered and their neighbours too.

DFS assigns two timestamps to each node *v*: *v.d* records when *v* becomes gray, *v.f* records when it becomes black.

t=7



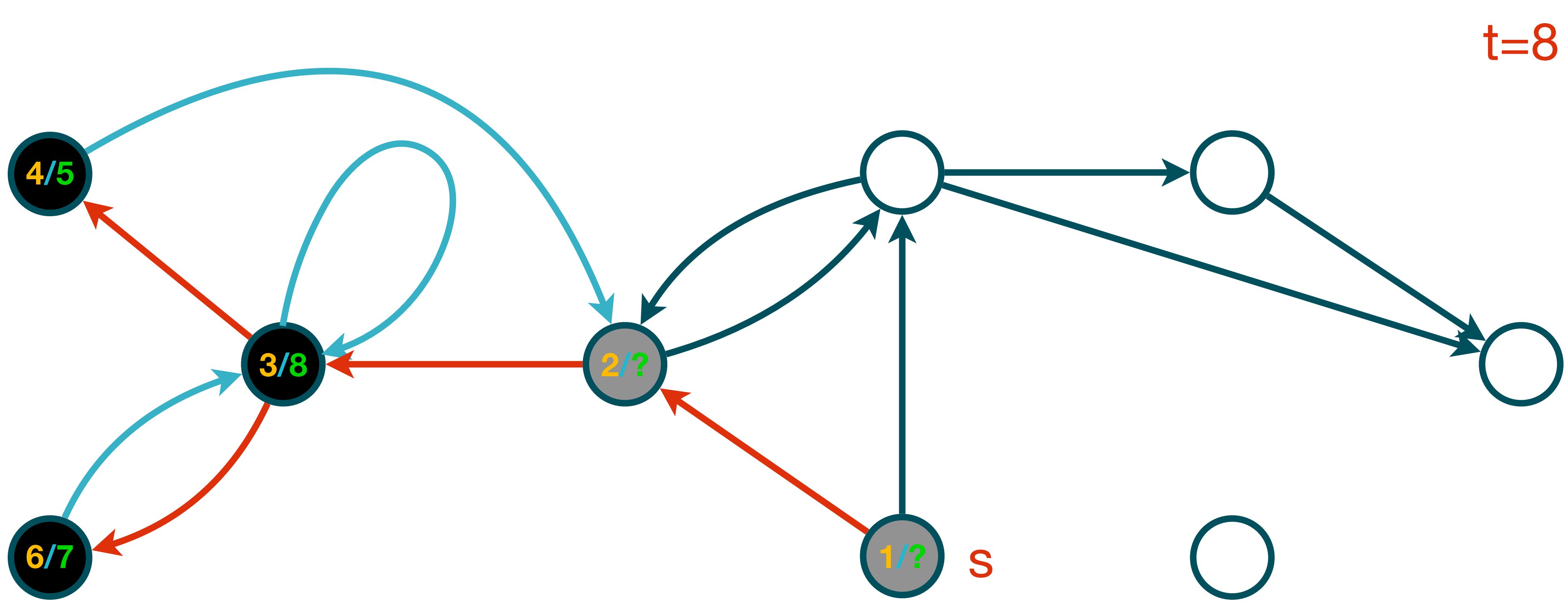
Depth-First Search



Much like BFS, DFS colors the nodes of G during the visit.

Again, white nodes have not been visited yet; gray nodes have been discovered but have undiscovered neighbours; black nodes have been discovered and their neighbours too.

DFS assigns two timestamps to each node *v*: *v.d* records when *v* becomes gray, *v.f* records when it becomes black.

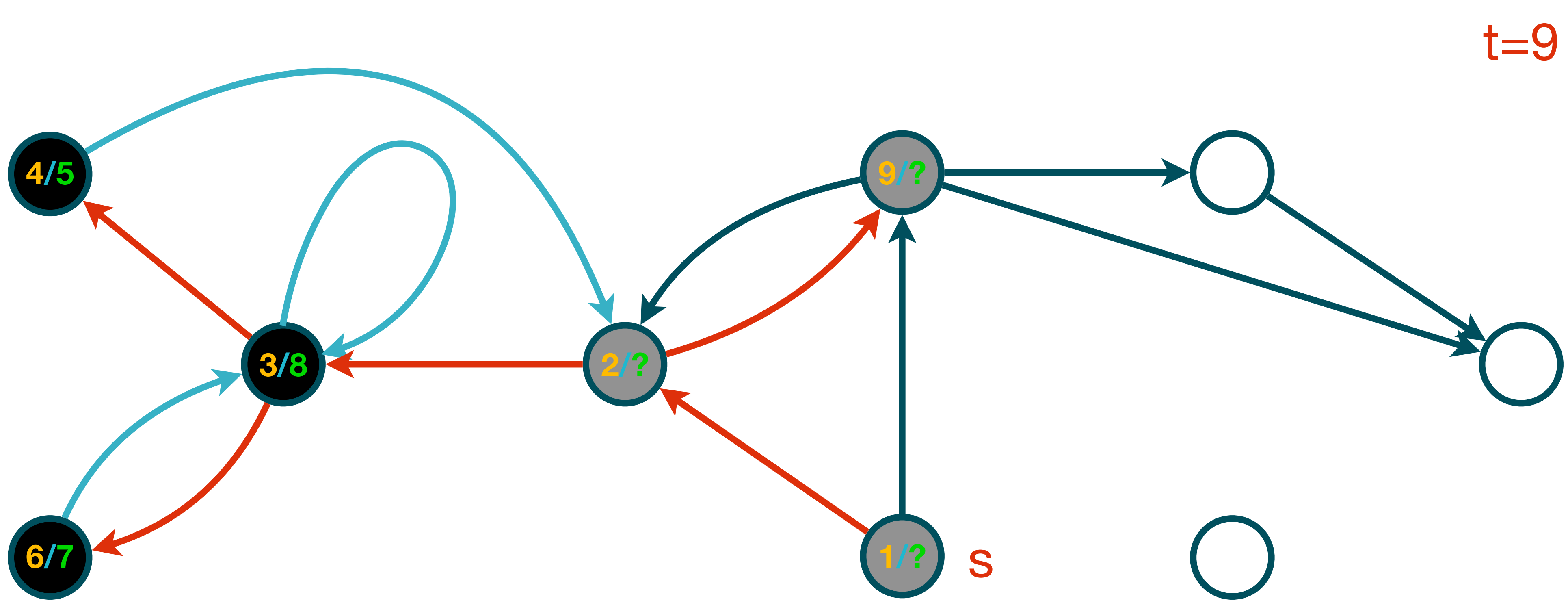


Depth-First Search

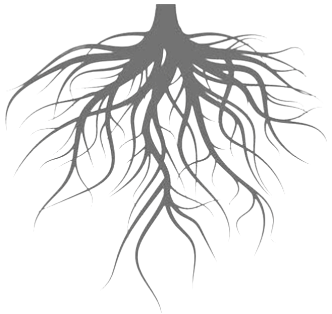
Much like BFS, DFS colors the nodes of G during the visit.

Again, white nodes have not been visited yet; gray nodes have been discovered but have undiscovered neighbours; black nodes have been discovered and their neighbours too.

DFS assigns two timestamps to each node *v*: *v.d* records when *v* becomes gray, *v.f* records when it becomes black.



Depth-First Search

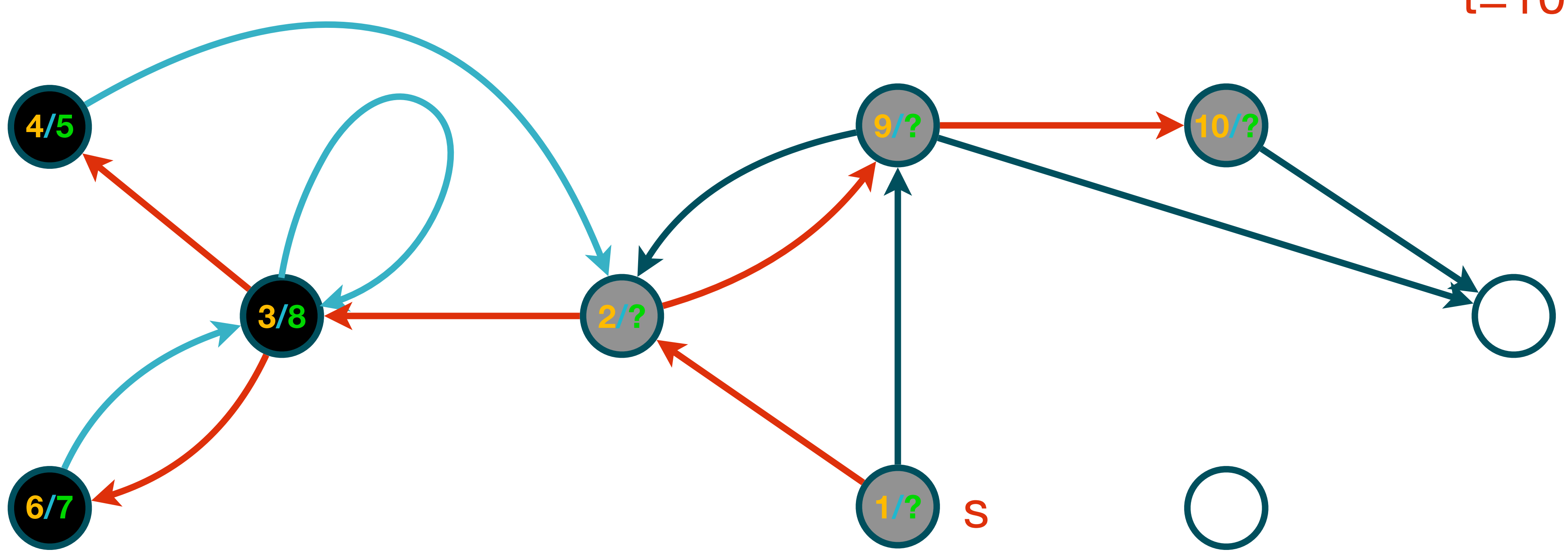


Much like BFS, DFS colors the nodes of G during the visit.

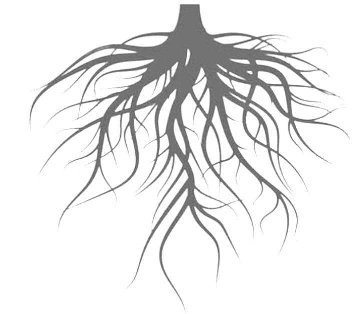
Again, white nodes have not been visited yet; gray nodes have been discovered but have undiscovered neighbours; black nodes have been discovered and their neighbours too.

DFS assigns two timestamps to each node *v*: *v.d* records when *v* becomes gray, *v.f* records when it becomes black.

t=10



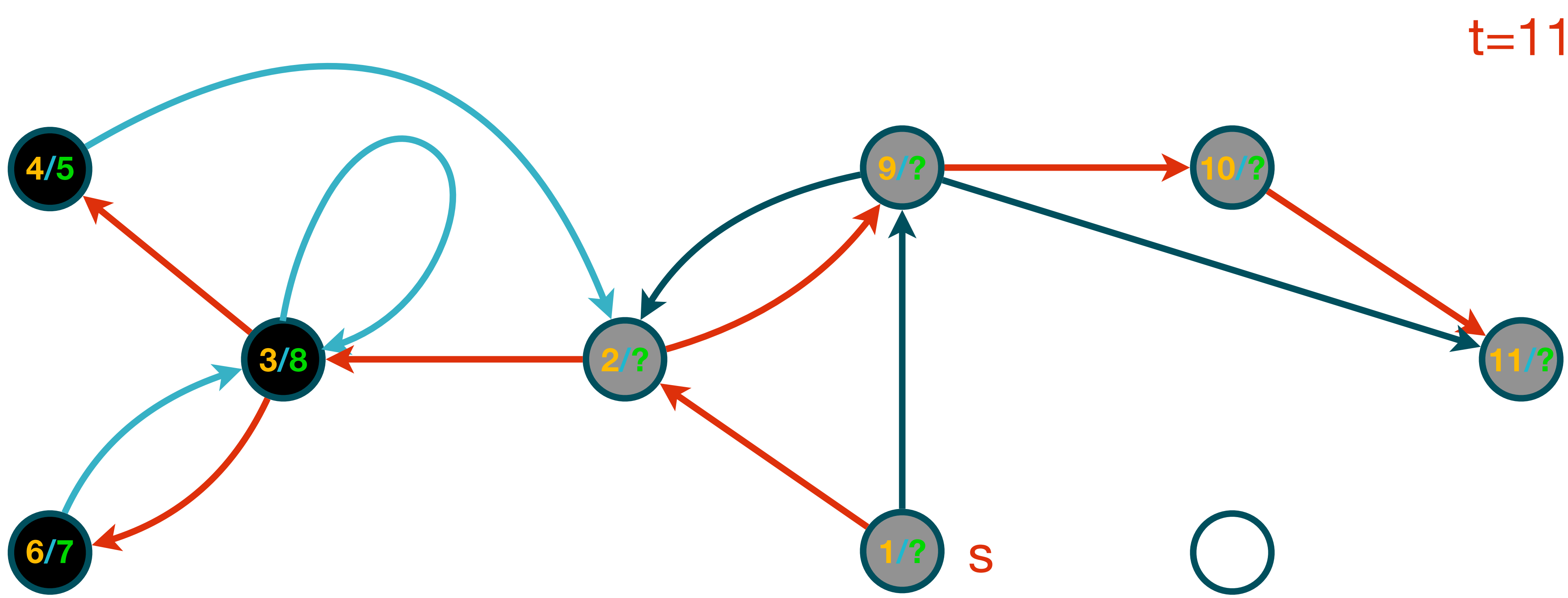
Depth-First Search



Much like BFS, DFS colors the nodes of G during the visit.

Again, white nodes have not been visited yet; gray nodes have been discovered but have undiscovered neighbours; black nodes have been discovered and their neighbours too.

DFS assigns two timestamps to each node *v*: *v.d* records when *v* becomes gray, *v.f* records when it becomes black.

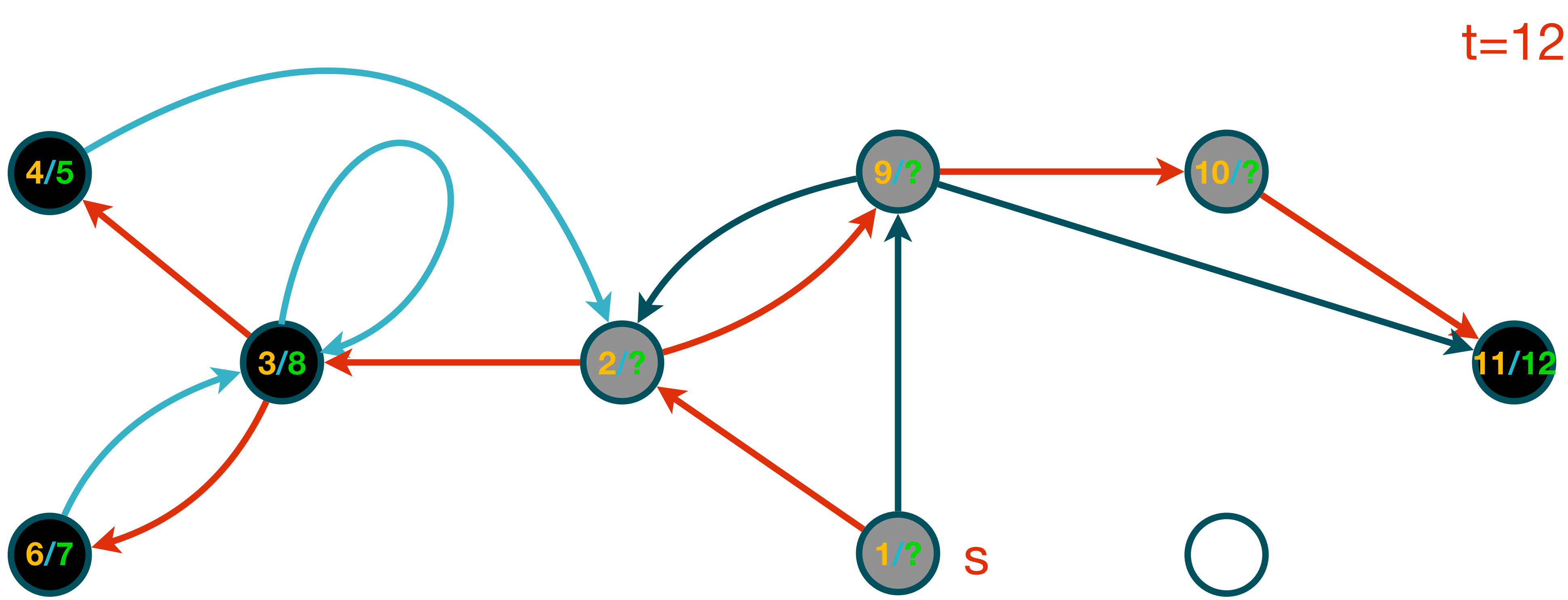


Depth-First Search

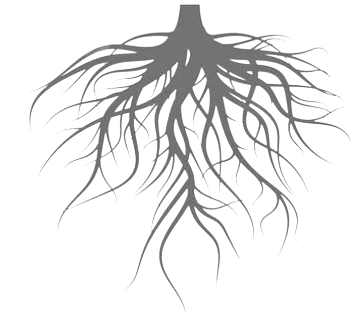
Much like BFS, DFS colors the nodes of G during the visit.

Again, white nodes have not been visited yet; gray nodes have been discovered but have undiscovered neighbours; black nodes have been discovered and their neighbours too.

DFS assigns two timestamps to each node *v*: *v.d* records when *v* becomes gray, *v.f* records when it becomes black.



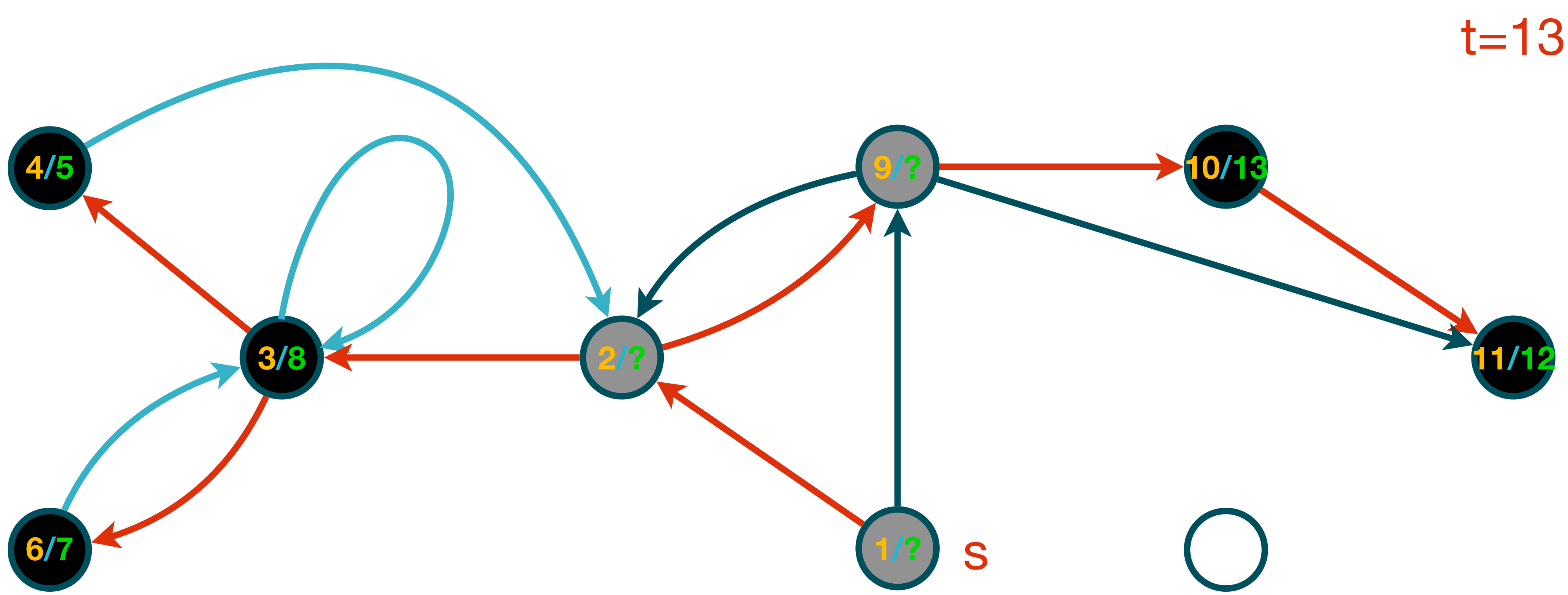
Depth-First Search



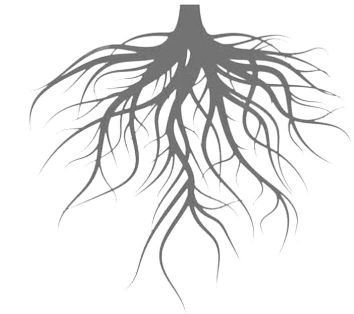
Much like BFS, DFS colors the nodes of G during the visit.

Again, white nodes have not been visited yet; gray nodes have been discovered but have undiscovered neighbours; black nodes have been discovered and their neighbours too.

DFS assigns two timestamps to each node *v*: *v.d* records when *v* becomes gray, *v.f* records when it becomes black.



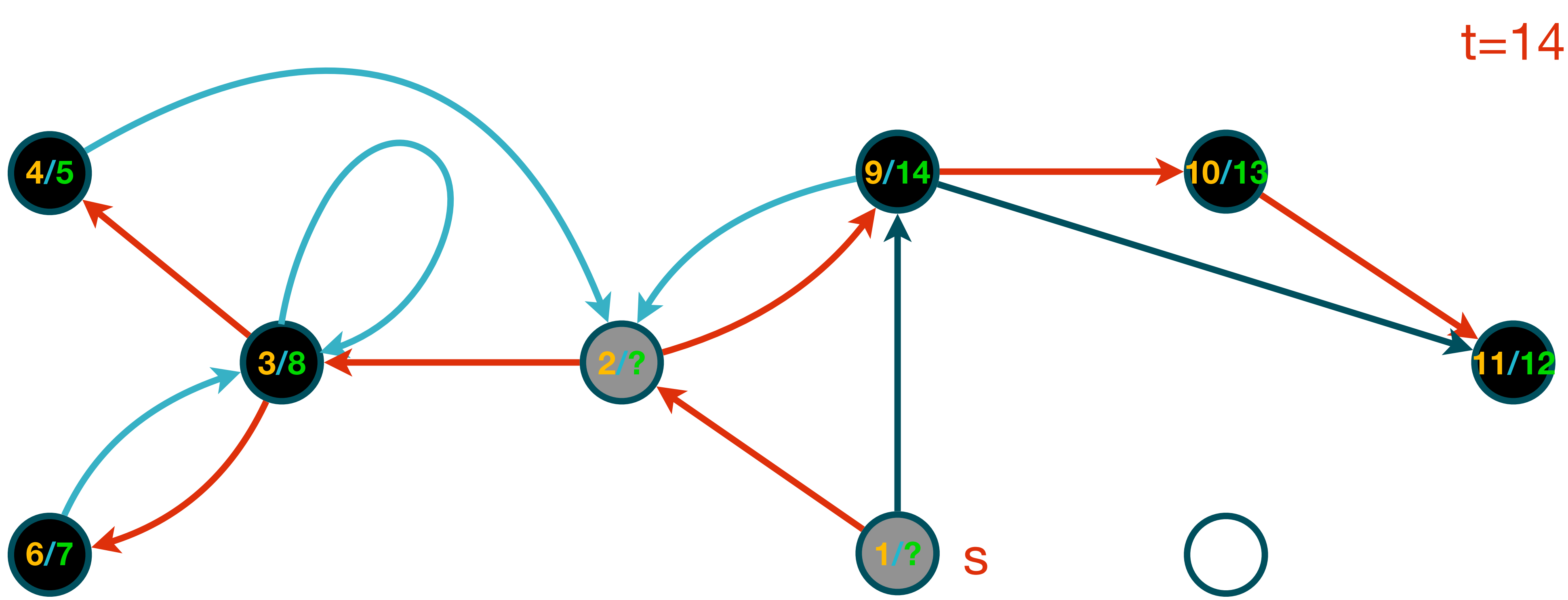
Depth-First Search



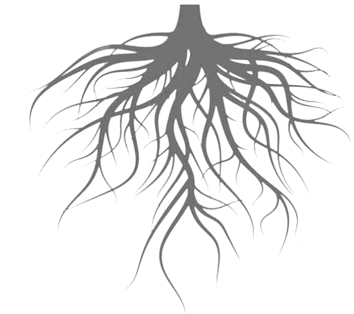
Much like BFS, DFS colors the nodes of G during the visit.

Again, white nodes have not been visited yet; gray nodes have been discovered but have undiscovered neighbours; black nodes have been discovered and their neighbours too.

DFS assigns two timestamps to each node *v*: *v.d* records when *v* becomes gray, *v.f* records when it becomes black.



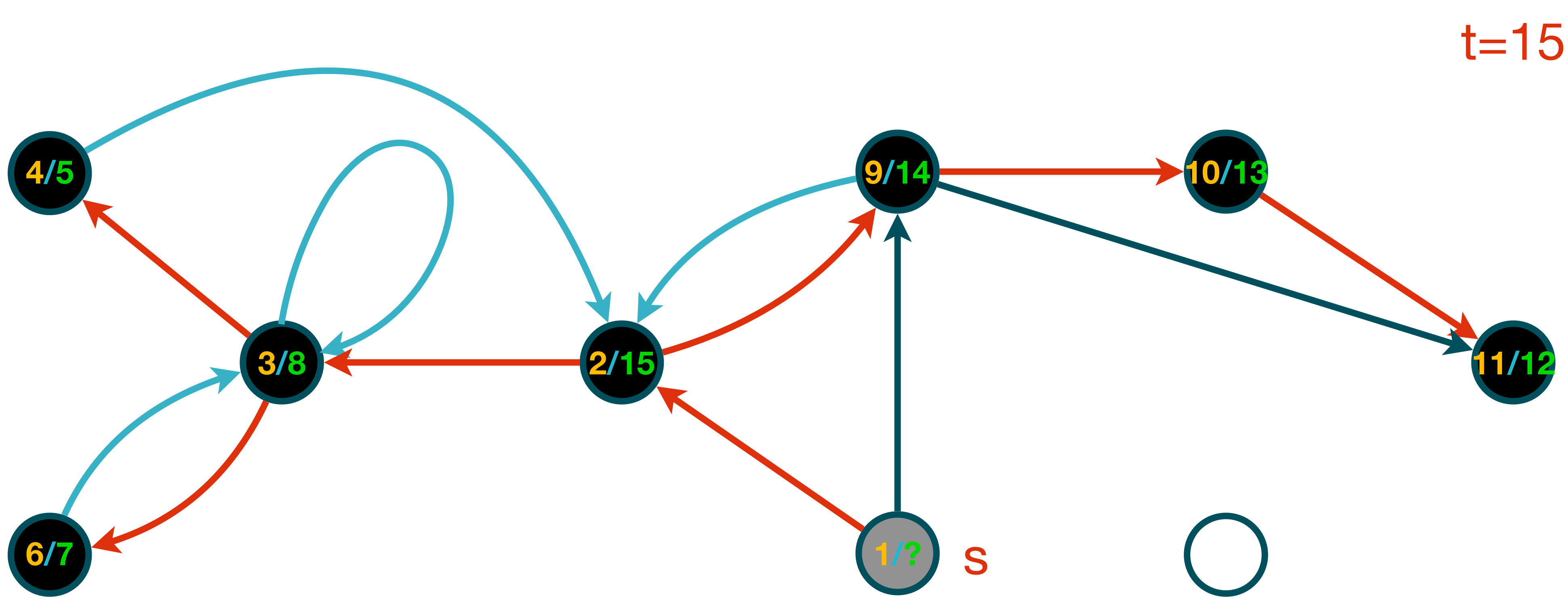
Depth-First Search



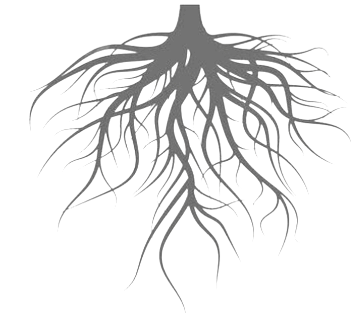
Much like BFS, DFS colors the nodes of G during the visit.

Again, white nodes have not been visited yet; gray nodes have been discovered but have undiscovered neighbours; black nodes have been discovered and their neighbours too.

DFS assigns two timestamps to each node *v*: *v.d* records when *v* becomes gray, *v.f* records when it becomes black.



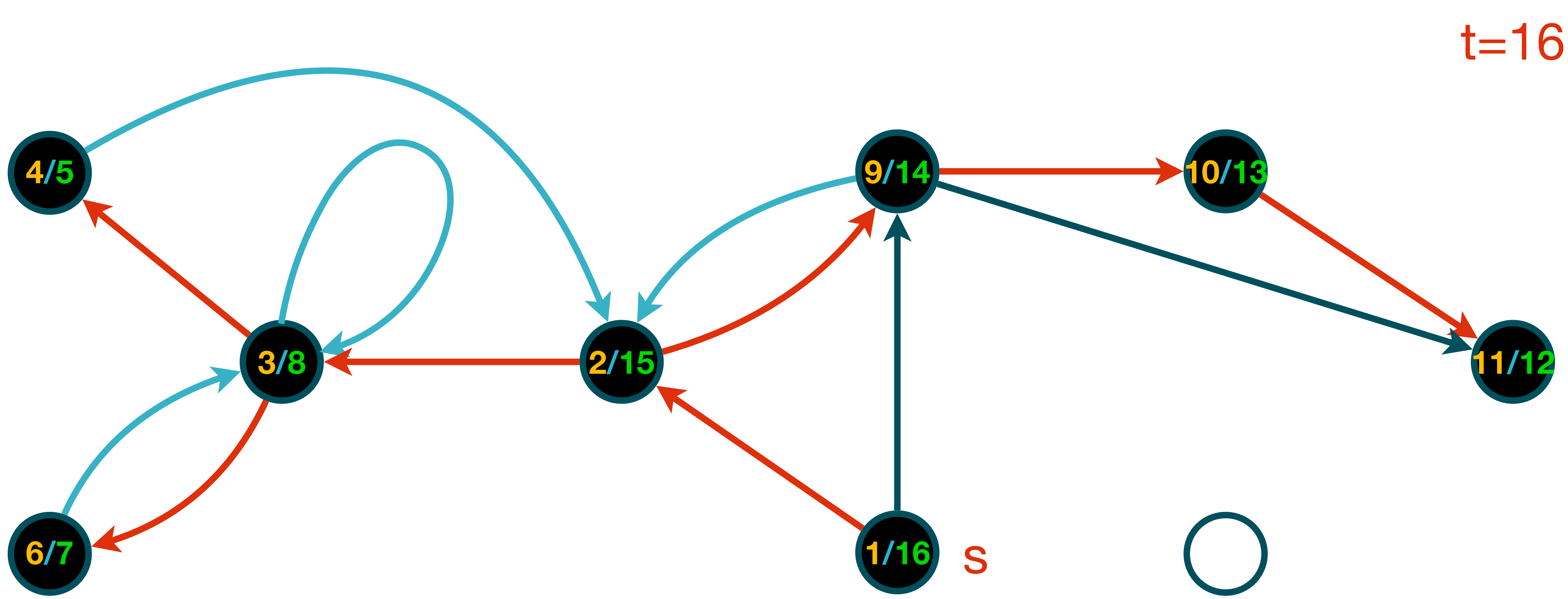
Depth-First Search



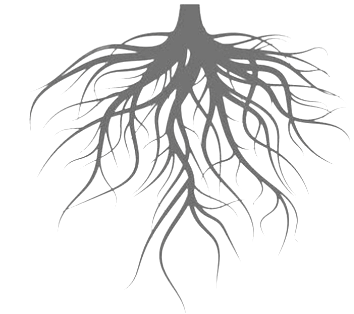
Much like BFS, DFS colors the nodes of G during the visit.

Again, white nodes have not been visited yet; gray nodes have been discovered but have undiscovered neighbours; black nodes have been discovered and their neighbours too.

DFS assigns two timestamps to each node *v*: *v.d* records when *v* becomes gray, *v.f* records when it becomes black.



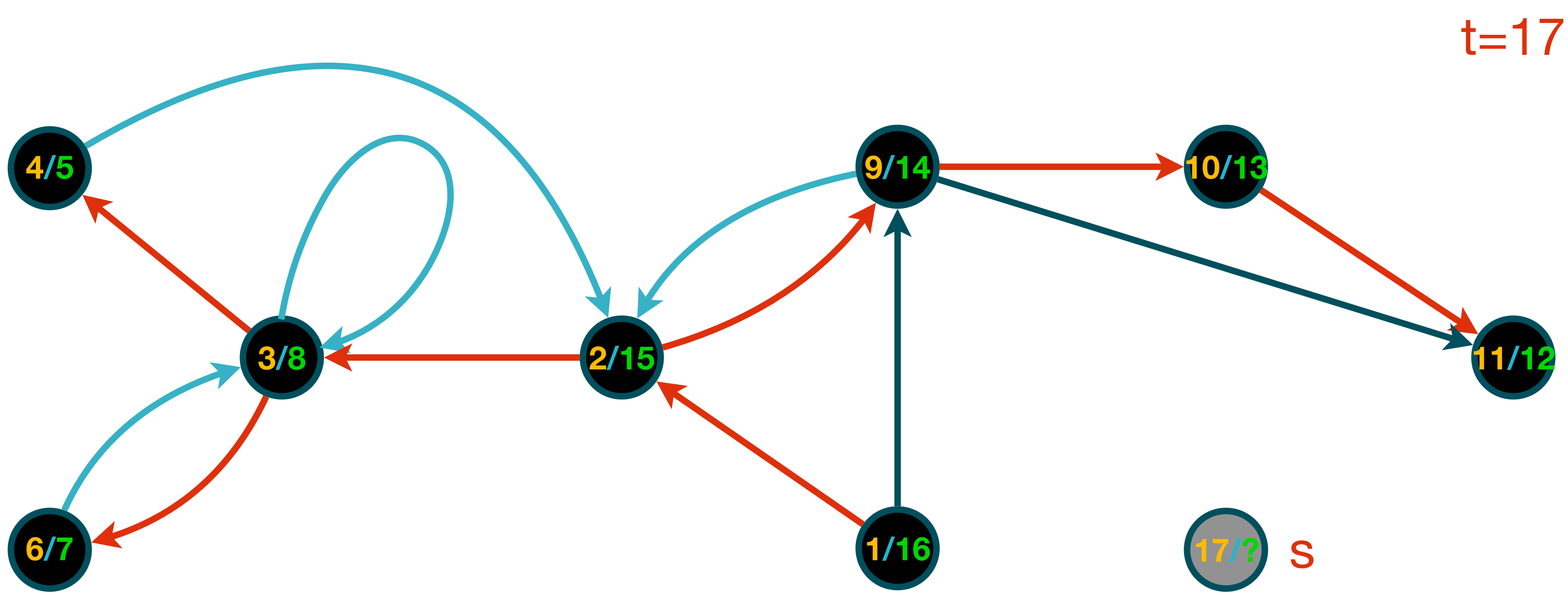
Depth-First Search



Much like BFS, DFS colors the nodes of G during the visit.

Again, white nodes have not been visited yet; gray nodes have been discovered but have undiscovered neighbours; black nodes have been discovered and their neighbours too.

DFS assigns two timestamps to each node *v*: *v.d* records when *v* becomes gray, *v.f* records when it becomes black.

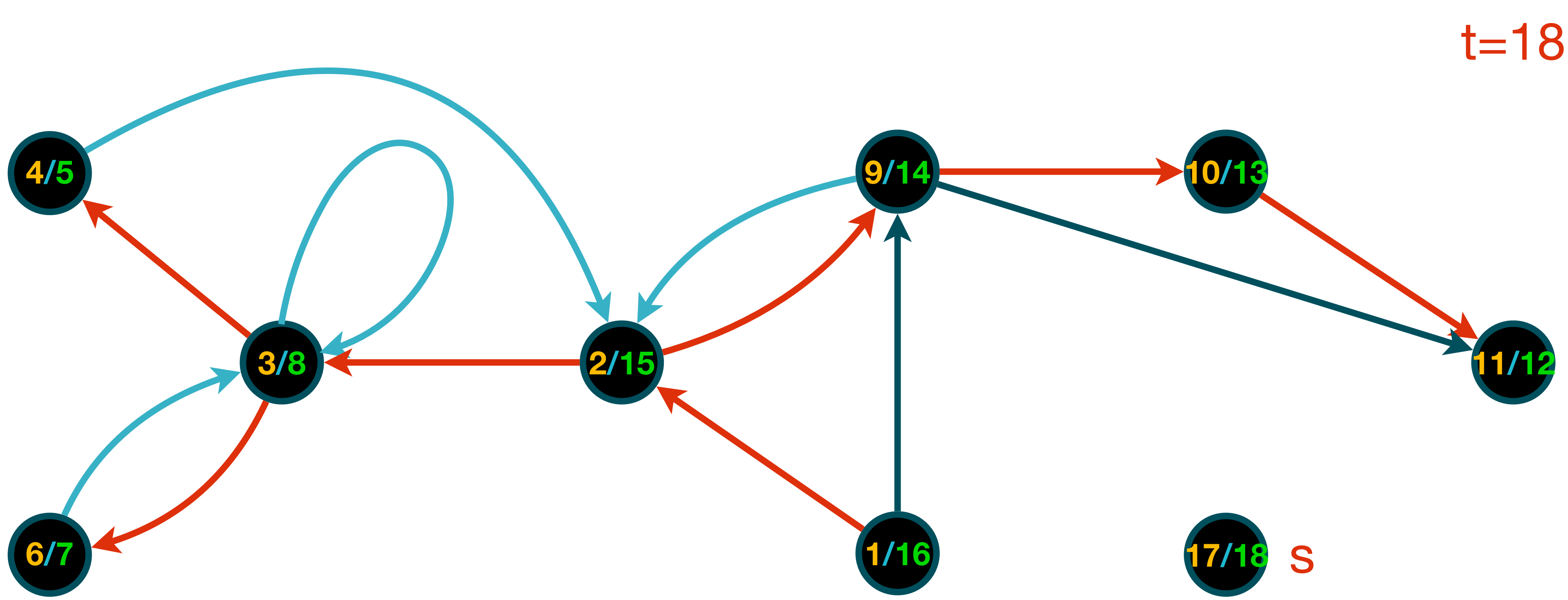


Depth-First Search

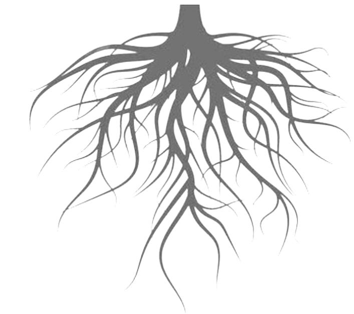
Much like BFS, DFS colors the nodes of G during the visit.

Again, white nodes have not been visited yet; gray nodes have been discovered but have undiscovered neighbours; black nodes have been discovered and their neighbours too.

DFS assigns two timestamps to each node *v*: *v.d* records when *v* becomes gray, *v.f* records when it becomes black.



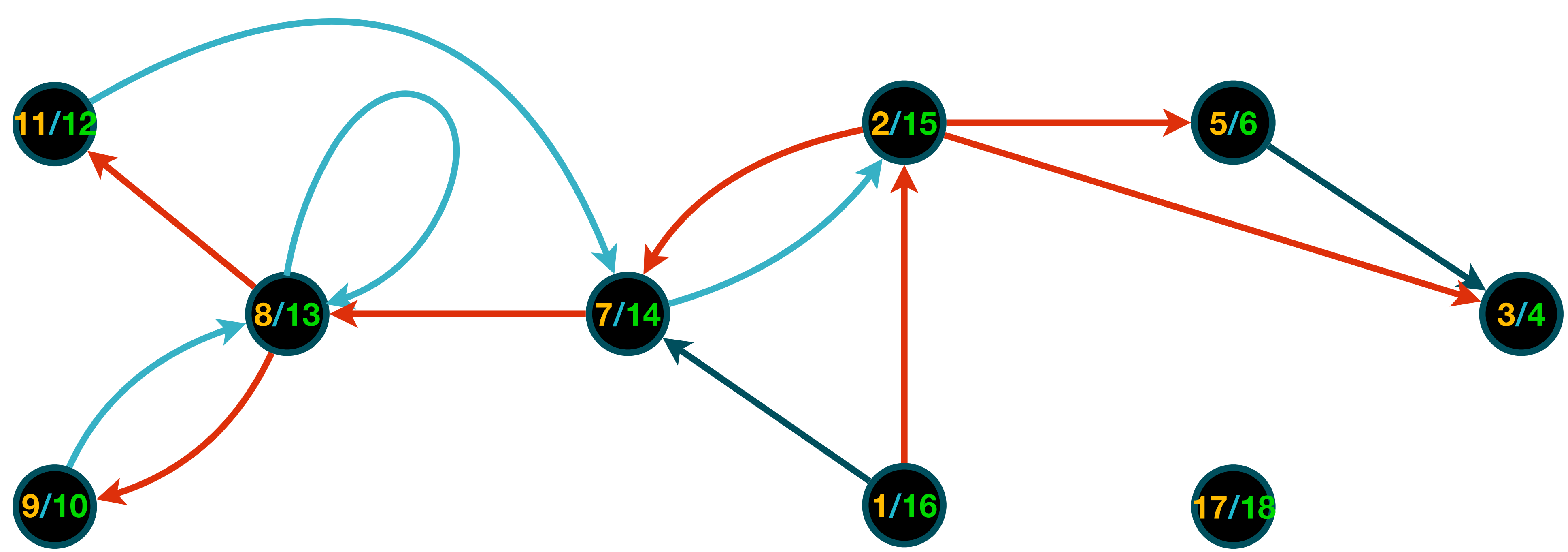
Depth-First Search



DFS produces a depth-first (DF) forest (a different tree for each source). Even for the same sources, this forest is not unique: it depends from the order in which the edges outgoing from each node are traversed. All the results are essentially equivalent.

The red edges are tree edges; the light blue edges are back edges, linking a node with one of its ancestors in the DF forest.

You can verify yourself that the result below is another possible outcome of DFS with the same two sources.



DFS: Pseudocode

DFS(G) - G is represented by the adjacency lists Adj[·] of its vertices

Initialisation:

for each $u \in V$
    $u.color \leftarrow$ white;
$t \leftarrow 0$;

Start the search from a new source:

for each $u \in V$
    if $u.color$ = white
        DFS_visit(G,u)

Visit the graph recursively:

DFS_visit(G,u)
    $t \leftarrow t+1$;
    $u.d \leftarrow t$;
    $u.color \leftarrow$ gray;
    for each $v \in$ Adj[u]
        if $v.color$ = white
            DFS_visit(G,u);
    $v.color \leftarrow$ black;
    $t \leftarrow t+1$;
    $u.f \leftarrow t$;

DFS: Complexity

DFS(G) - G is represented by the adjacency lists Adj[$\cdot$] of its vertices

for each $u \in V$
    $u.color \leftarrow$ white;
$t \leftarrow 0$;
for each $u \in V$
    if $u.color$ = white
        DFS_visit(G,u)

Initialisation: $O(|V|)$

Start the search from a new source: this only happens when a vertex is white $\Longrightarrow O(|V|)$ calls

DFS_visit(G,u)
    $t \leftarrow t+1$;
    $u.d \leftarrow t$;
    $u.color \leftarrow$ gray;
    for each $v \in$ Adj[u]
        if $v.color$ = white
            DFS_visit(G,u);
    $v.color \leftarrow$ black;
    $t \leftarrow t+1$;
    $u.f \leftarrow t$;

Visit the graph recursively: this procedure is only called on white vertices, which are immediately painted gray

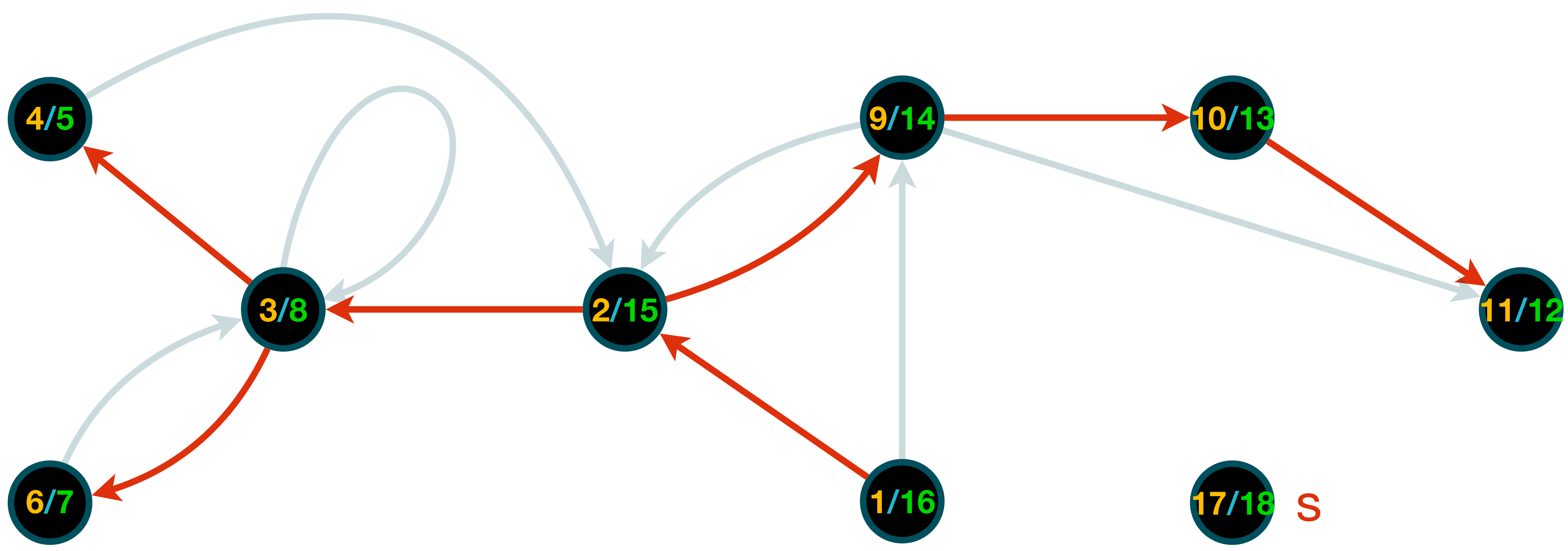
$$\Longrightarrow O\left(\sum_{u \in V} |\mathrm{Adj}[u]|\right) = O(|E|)$$

DFS: Properties

Lemma 4. The time complexity of DFS is $O(|V|+|E|)$ (linear in the size of the adjacency-list representation of G)

Parenthesis Theorem. For any two nodes $u,v \in V$, either:

- $[u.d,u.f] \cap [v.d,v.f] = \emptyset$ and neither *u* is a descendant of *v* nor *v* is a descendant of *u*
- $[u.d,u.f] \subsetneq [v.d,v.f]$ and *u* is a descendant of *v*
- $[v.d,v.f] \subsetneq [u.d,u.f]$ and *v* is a descendant of *u*

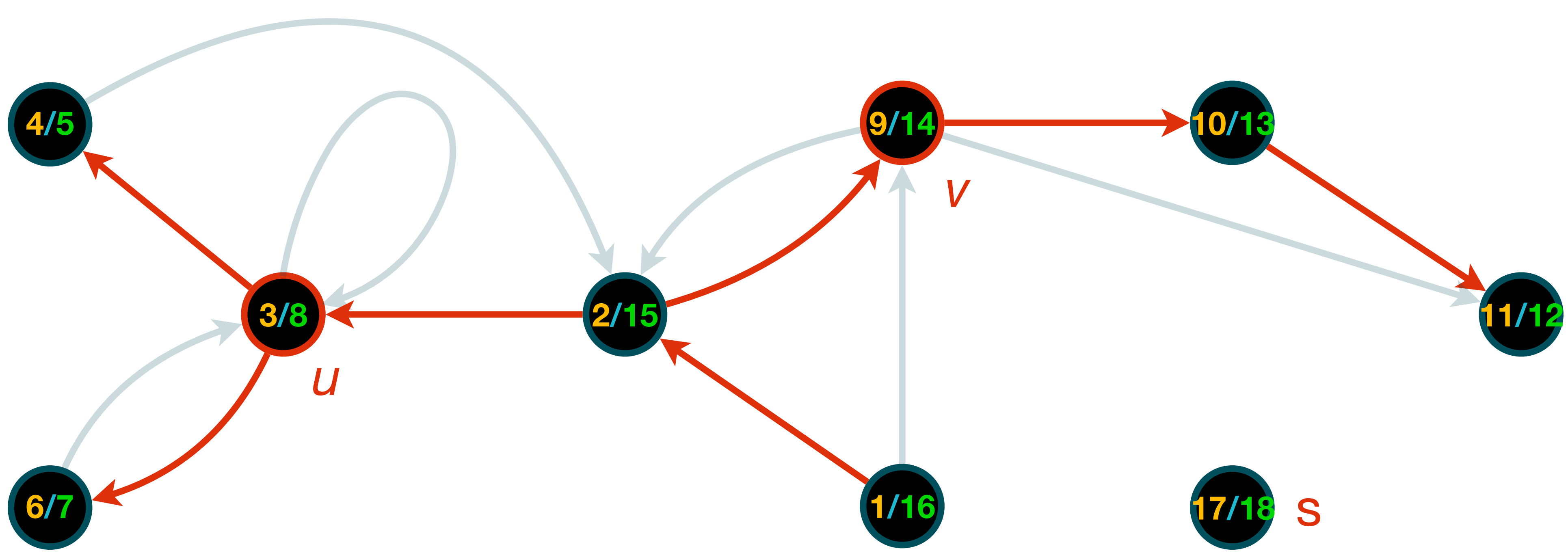


DFS: Properties

Lemma 4. The time complexity of DFS is $O(|V|+|E|)$ (linear in the size of the adjacency-list representation of G)

Parenthesis Theorem. For any two nodes $u,v \in V$, either:

- $[3,8] \cap [9,14] = \varnothing$ and neither u is a descendant of v nor v is a descendant of u
- $[u.d, u.f] \subsetneq [v.d, v.f]$ and u is a descendant of v
- $[v.d, v.f] \subsetneq [u.d, u.f]$ and v is a descendant of u

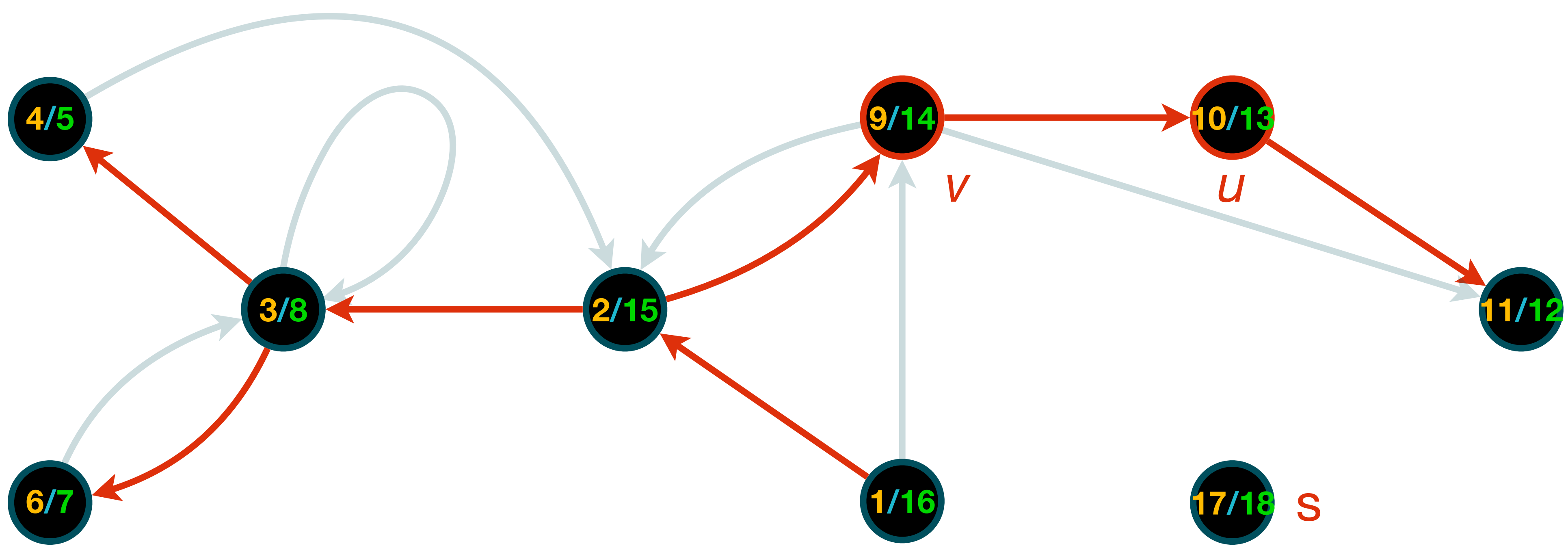


DFS: Properties

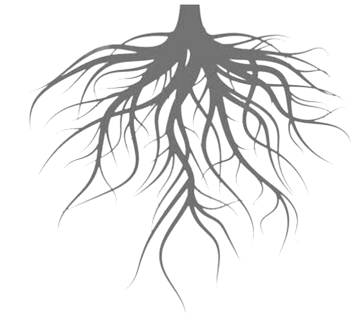
Lemma 4. The time complexity of DFS is $O(|V|+|E|)$ (linear in the size of the adjacency-list representation of G)

Parenthesis Theorem. For any two nodes $u,v \in V$, either:

- $[u.d, u.f] \cap [v.d, v.f] = \emptyset$ and neither u is a descendant of v nor v is a descendant of u
- $[10,13] \subsetneq [9,14]$ and u is a descendant of v
- $[v.d, v.f] \subsetneq [u.d, u.f]$ and v is a descendant of u



DFS: Properties



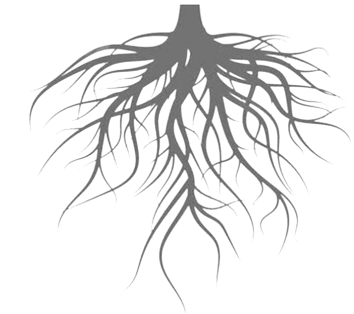
Lemma 4. The time complexity of DFS is $O(|V|+|E|)$ (linear in the size of the adjacency-list representation of G)

Parenthesis Theorem. For any two nodes $u,v \in V$, either:

- $[u.d, u.f] \cap [v.d, v.f] = \emptyset$ and neither u is a descendant of v nor v is a descendant of u
- $[u.d, u.f] \subsetneq [v.d, v.f]$ and u is a descendant of v
- $[v.d, v.f] \subsetneq [u.d, u.f]$ and v is a descendant of u

White-Path Theorem. For any two nodes $u,v \in V$, u is a descendant of v $\Longleftrightarrow$ at time $v.d$-1 there exists a path of white nodes from v to u.

DFS: Properties



Lemma 4. The time complexity of DFS is $O(|V|+|E|)$ (linear in the size of the adjacency-list representation of G)

Parenthesis Theorem. For any two nodes $u,v \in V$, either:

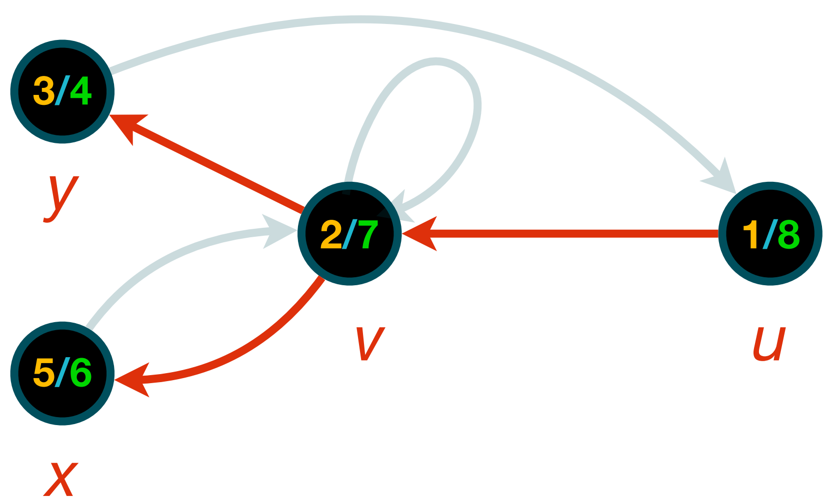
- $[u.d, u.f] \cap [v.d, v.f] = \varnothing$ and neither u is a descendant of v nor v is a descendant of u
- $[u.d, u.f] \subsetneq [v.d, v.f]$ and u is a descendant of v
- $[v.d, v.f] \subsetneq [u.d, u.f]$ and v is a descendant of u

White-Path Theorem. For any two nodes $u,v \in V$, u is a descendant of $v \iff$ at time $v.d$-1 there exists a path of white nodes from v to u.

Observation 5. In DFS, when we explore an edge (u,v), this is a tree edge if v is white; it is a back edge if v is gray.

DFS: Exercises

Observation. The parenthesis theorem tells us that the intervals determined by the discovery and finishing time of every vertex are either nested or disjoint. If we represent the discovery of a vertex *u* with a left parenthesis "(*u*" and its finishing with a right parenthesis "*u*)", then the sequence of discoveries and finishings makes an expression whose parentheses are properly nested. Inspect the graph below and its expression:



Time: 1 2 3 4 5 6 7 8

Expression: (u (v (y y) (x x) v) u)

Exercise 2. Can you find the right parentheses expression for the examples of DFS we have seen?

DFS: Exercises

Exercise 3. Can you rewrite the pseudocode for DFS using a stack to avoid recursion?

Exercise 4. DFS can be used to identify the connected components of a graph. Can you modify the pseudocode so that it assigns to every vertex *v* a label *v.cc* between 1 and *k*, where *k* is the number of connected components, such that *v.cc* = *u.cc* if and only if *u* and *v* are in the same connected component?

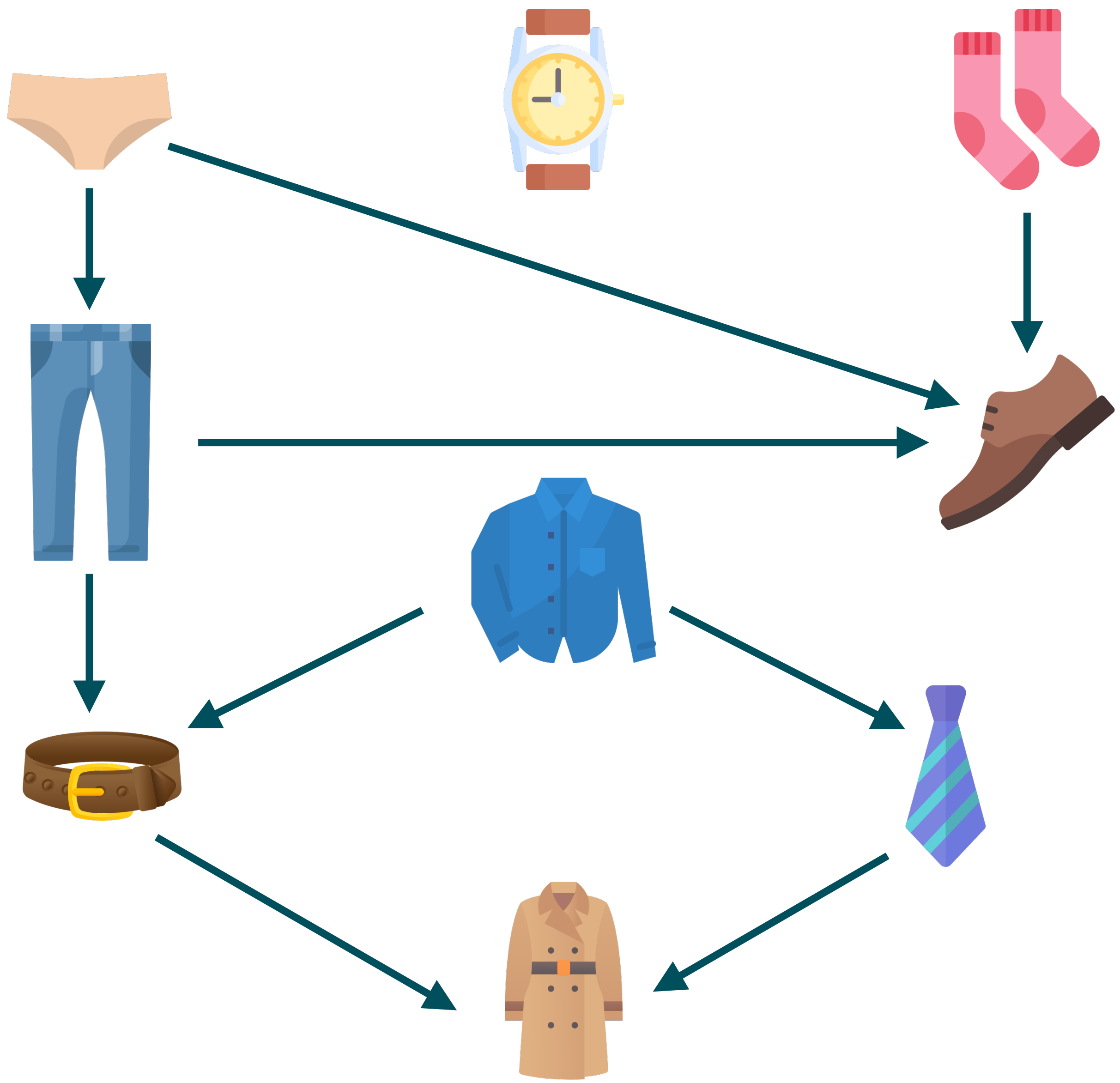
An application: Topological Sort

Important property of directed acyclic graphs (DAGs): they admit a topological sort.

A topological sort of a graph is an ordering of its vertices such that if the graph has an edge (u,v) then u comes before v in the ordering.

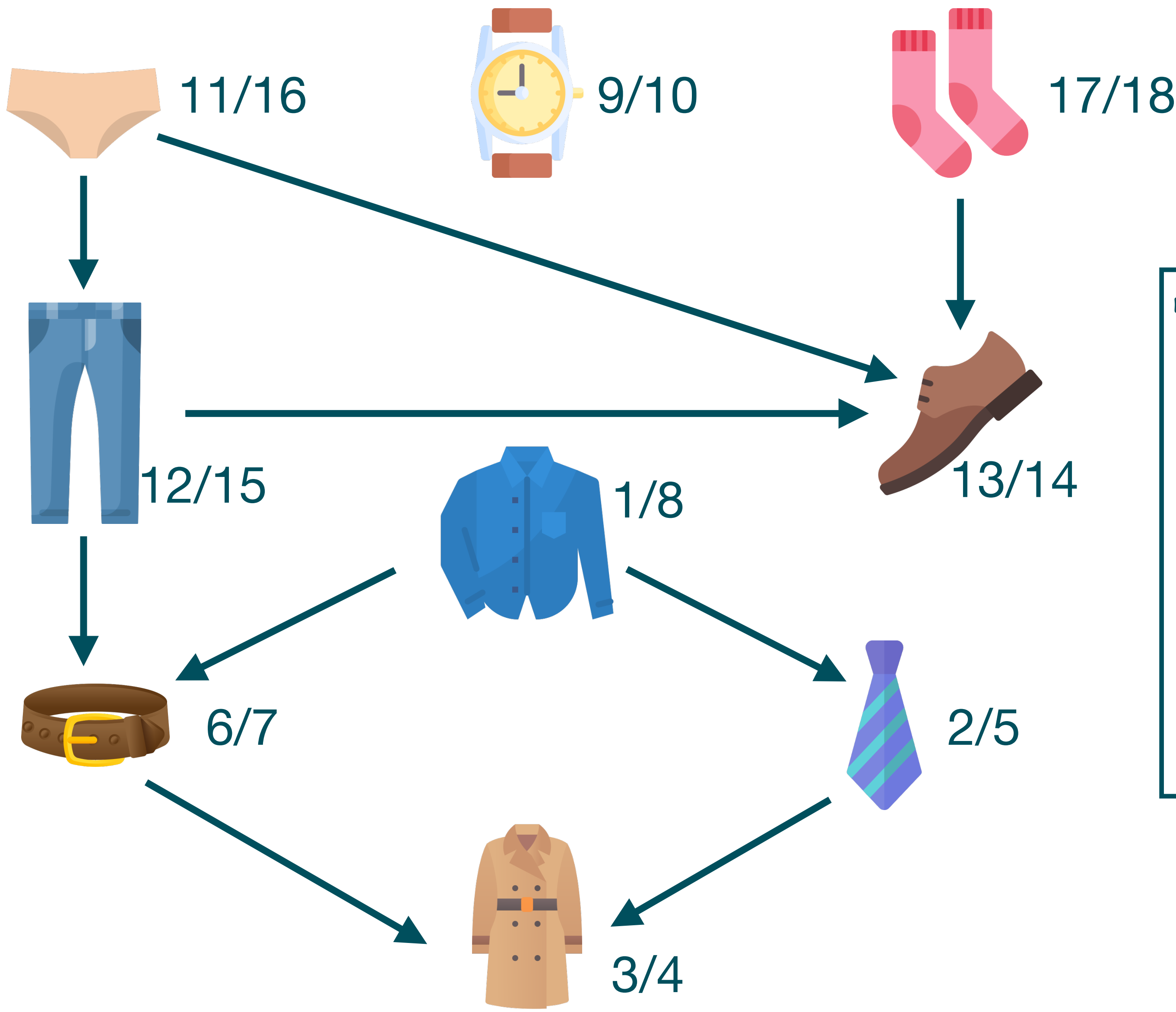
It is useful, for examples, in cases where DAGs indicate precedences among events.

An application: Topological Sort



An edge (*u*,*v*) indicates that item *u* must be worn before item *v*.

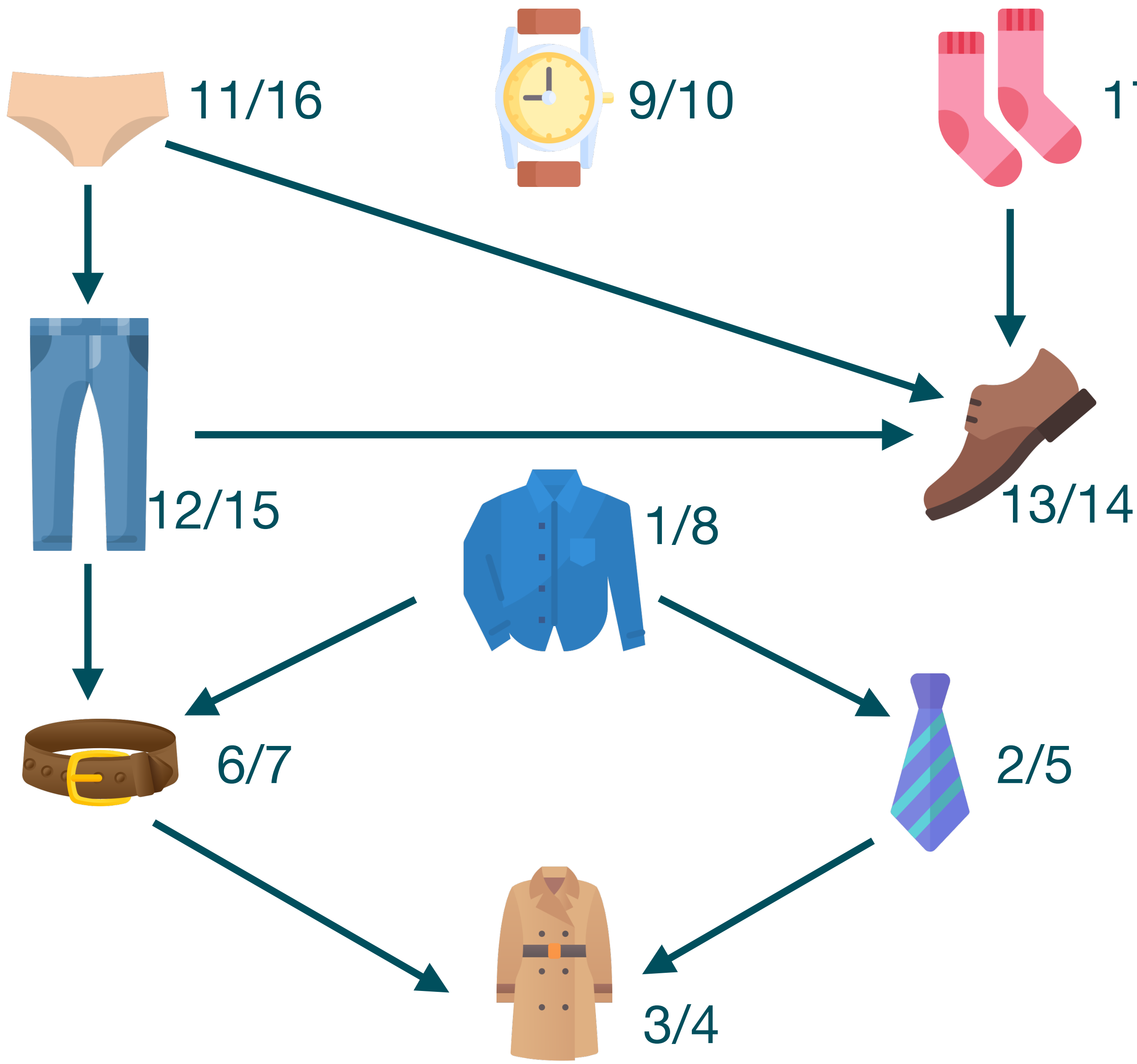
An application: Topological Sort



TopologicalSort(G)
  DFS(G);
  $\tilde{V}[1,\ldots,|V|] \leftarrow V$ sorted w.r.t finishing time
  TopOrder←empty_stack;
  for $i = 1\ldots|V|$
    TopOrder.push($\tilde{V}[i]$);
  return TopOrder;

An edge (*u*,*v*) indicates that item *u* must be worn before item *v*.

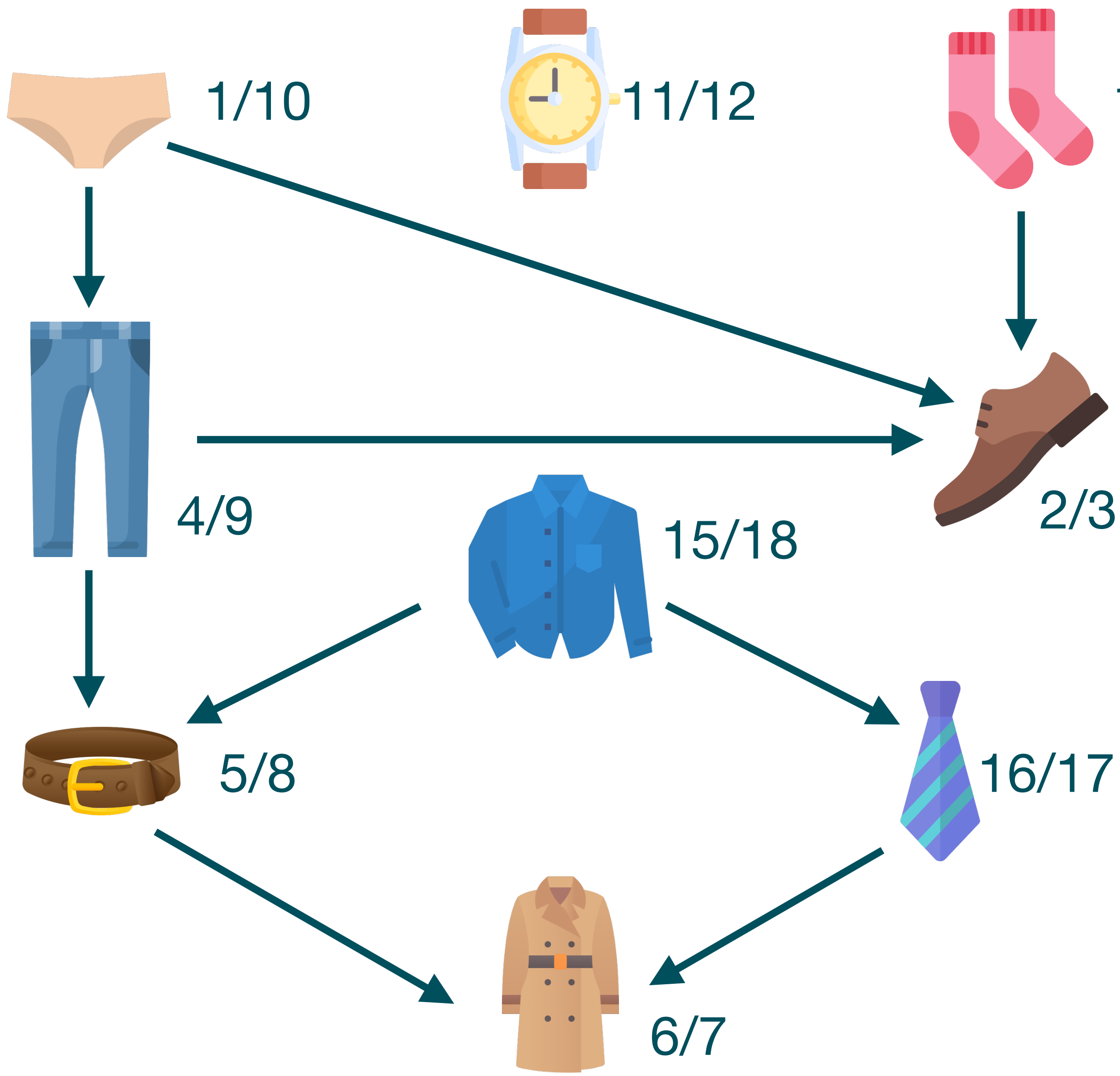
An application: Topological Sort



TopologicalSort(G)
  DFS(G);
  $\tilde{V}[1,\ldots,|V|] \leftarrow V$ sorted w.r.t finishing time
  TopOrder←empty_stack;
  for $i = 1\ldots|V|$
    TopOrder.push($\tilde{V}[i]$);
  return TopOrder;

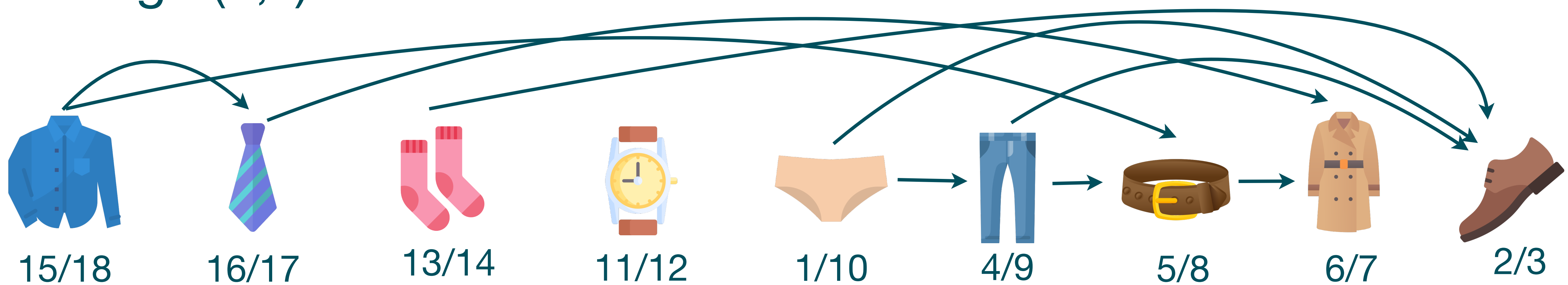
An edge (*u*,*v*) indicates that item *u* must be worn before item *v*.

An application: Topological Sort



TopologicalSort(G)
  DFS(G);
  $\tilde{V}[1,\ldots,|V|] \leftarrow V$ sorted w.r.t finishing time
  TopOrder←empty_stack;
  for $i = 1\ldots|V|$
    TopOrder.push($\tilde{V}[i]$);
  return TopOrder;

An edge (*u*,*v*) indicates that item *u* must be worn before item *v*.



Correctness of Topological Sort

Lemma 6. A directed graph G is acyclic $\Longleftrightarrow$ DFS(G) yields no back edges

Theorem 7. Algorithm TopologicalSort is correct.

Proof (sketch). It suffices to show that for two distinct vertices $u,v \in V$, if G contains an edge from u to v, then $v.f < u.f$.
It is easy to prove it by using Lemma 6 and Observation 5: consider an edge (u,v) explored by DFS(G) and consider all possible cases for the color of v.