Directed Graphs (Digraphs) and Graphs

Definitions Graph ADT Traversal algorithms DFS

Lecturer: Georgy Gimel'farb

COMPSCI 220 Algorithms and Data Structures

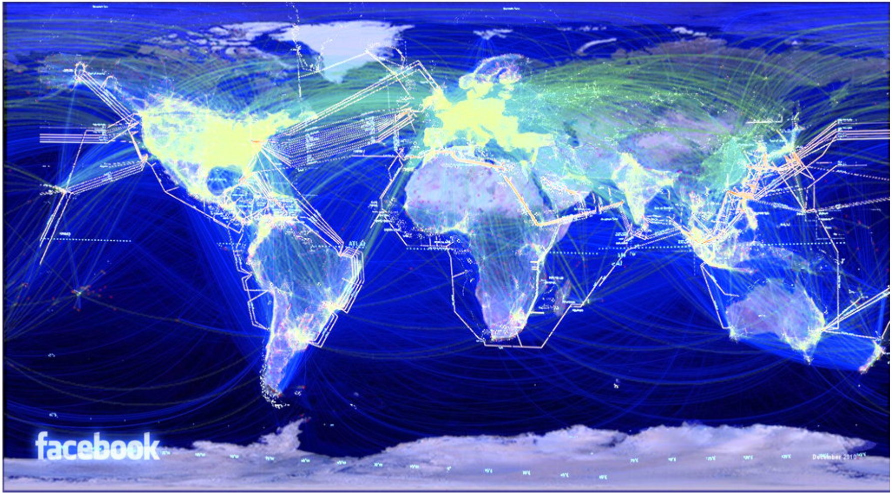
Graphs in Life: World Air Roures

http://milenomics.com/2014/05/partners-alliances-partner-awards/

Graphs in Life: Global Internet Connections

http://www.opte.org/maps/

Graphs in Life: Social Networks (Facebook)

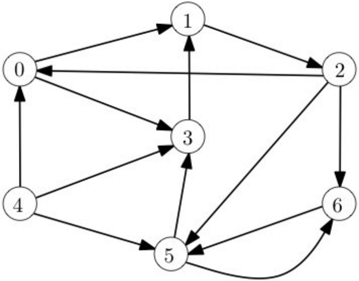


http://robotmonkeys.net/wp-content/uploads/2010/12/social-nets-then-and-now-fb-cities-airlines-data.jpg

Directed Graph, or Digraph: Definition

A **digraph** $G = (V, E)$ is a finite nonempty set V of **nodes** together with a (possibly empty) set E of ordered pairs of nodes° of G called **arcs**.

$$
\begin{aligned}
V &= \{\, 0, 1, 2, 3, 4, 5, 6 \,\} \\
E &= \{\, (0,1),\ (0,3), \\
&\quad (1,2), \\
&\quad (2,0),\ (2,5),\ (2,6), \\
&\quad (3,1), \\
&\quad (4,0),\ (4,3),\ (4,5), \\
&\quad (5,3),\ (5,6), \\
&\quad (6,5)\ \}
\end{aligned}
$$

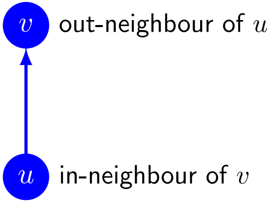


°) Set E is a neighbourhood, or adjacency *relation* on V.

Digraph: Relations of Nodes

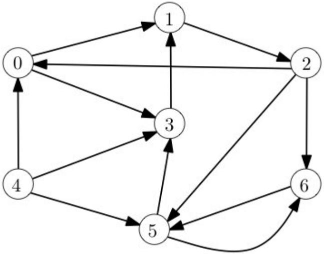
If $(u, v) \in E$,

- v is **adjacent** to u;
- v is an **out-neighbour** of u, and
- u is an **in-neighbour** of v.



Examples:

- Nodes (points) 1 and 3 are adjacent to 0.
- 1 and 3 are out-neighbours of 0.
- 0 is an in-neighbour of 1 and 3.
- Node 1 is adjacent to 3.
- 1 is an out-neighbour of 3.
- 3 is an in-neighbour of 1. ...
- 5 is an out-neighbour of 2, 4, and 6.

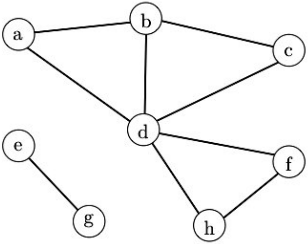


(Undirected) Graph: Definition

A **graph**° $G = (V, E)$ is a finite nonempty set V of **vertices** together with a (possibly empty) set E of unordered pairs of vertices of G called **edges**.

$$V = \{ a, b, c, d, e, f, g, h \}$$

$$E = \{ \{a,b\}, \{a,d\}, \{b,d\}, \{b,c\}, \{c,d\}, \{d,f\}, \{d,h\} \{f,h\}, \{e,g\} \}$$



°) The symmetric digraph: each arc (u, v) has the opposite arc (v, u).

Such a pair is reduced into a single undirected edge that can be traversed in either direction.

Order, Size, and In- / Out-degree

The **order** of a digraph $G = (V, E)$ is the number of nodes, $n = |V|$.

The **size** of a digraph $G = (V, E)$ is the number of arcs, $m = |E|$.

For a given n, $m = 0$ — **Sparse** digraphs: $|E| \in O(n)$ — **Dense** digraphs: $|E| \in \Theta(n^2)$ — $n(n-1)$

The **in-degree** or **out-degree** of a node v is the number of arcs entering or leaving v, respectively.

- A node of in-degree 0 – a **source**.
- A node of out-degree 0 – a **sink**.
- This example: the order $|V| = 6$ and the size $|E| = 9$.

Order, Size, and In- / Out-degree

The **order** of a digraph $G = (V, E)$ is the number of nodes, $n = |V|$.

The **size** of a digraph $G = (V, E)$ is the number of arcs, $m = |E|$.

For a given n, $m = 0$ ... $n(n-1)$

Sparse digraphs: $|E| \in O(n)$ **Dense** digraphs: $|E| \in \Theta(n^2)$

The **in-degree** or **out-degree** of a node v is the number of arcs entering or leaving v, respectively.

- A node of in-degree 0 – a **source**.
- A node of out-degree 0 – a **sink**.
- This example: the order $|V| = 6$ and the size $|E| = 9$.

Order, Size, and In- / Out-degree

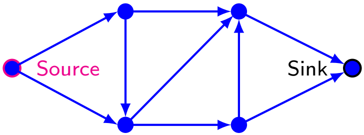
The **order** of a digraph $G = (V, E)$ is the number of nodes, $n = |V|$.

The **size** of a digraph $G = (V, E)$ is the number of arcs, $m = |E|$.

For a given n, $m = 0$ — **Sparse** digraphs: $|E| \in O(n)$ — **Dense** digraphs: $|E| \in \Theta(n^2)$ — $n(n-1)$

The **in-degree** or **out-degree** of a node v is the number of arcs entering or leaving v, respectively.

- A node of in-degree 0 – a **source**.
- A node of out-degree 0 – a **sink**.
- This example: the order $|V| = 6$ and the size $|E| = 9$.



Walk, Path, and Cycle

A walk in a digraph $G = (V, E)$:

a sequence of nodes $v_0\, v_1\, \ldots\, v_n$, such that (v_i, v_{i+1}) is an arc in G, i.e., $(v_i, v_{i+1}) \in E$, for each i; $0 \leq i < n$.

- The **length** of the walk $v_0\, v_1\, \ldots\, v_n$ is the number n of arcs involved.
- A **path** is a walk, in which no node is repeated.
- A **cycle** is a walk, in which $v_0 = v_n$ and no other nodes are repeated.

- By convention, a cycle in a graph is of length at least 3.
- It is easily shown that if there is a walk from u to v, then there is at least one path from u to v.

Walk, Path, and Cycle

A walk in a digraph $G = (V, E)$:

a sequence of nodes $v_0\,v_1\,\ldots\,v_n$, such that (v_i, v_{i+1}) is an arc in G, i.e., $(v_i, v_{i+1}) \in E$, for each i; $0 \leq i < n$.

- The **length** of the walk $v_0\,v_1\,\ldots\,v_n$ is the number n of arcs involved.
- A **path** is a walk, in which no node is repeated.
- A **cycle** is a walk, in which $v_0 = v_n$ and no other nodes are repeated.

- By convention, a cycle in a graph is of length at least 3.
- It is easily shown that if there is a walk from u to v, then there is at least one path from u to v.

Walk, Path, and Cycle

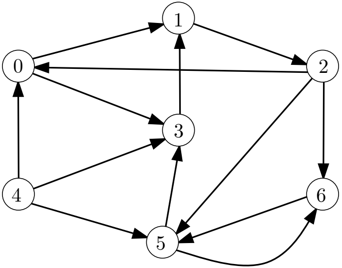
A walk in a digraph $G = (V, E)$:

a sequence of nodes $v_0\,v_1\,\ldots\,v_n$, such that (v_i, v_{i+1}) is an arc in G, i.e., $(v_i, v_{i+1}) \in E$, for each i; $0 \le i < n$.

- The **length** of the walk $v_0\,v_1\,\ldots\,v_n$ is the number n of arcs involved.
- A **path** is a walk, in which no node is repeated.
- A **cycle** is a walk, in which $v_0 = v_n$ and no other nodes are repeated.

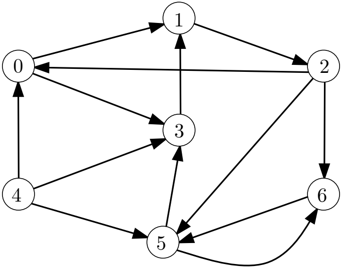
- By convention, a cycle in a graph is of length at least 3.
- It is easily shown that if there is a walk from u to v, then there is at least one path from u to v.

Walks, Paths, and Cycles in a Digraph: an Example



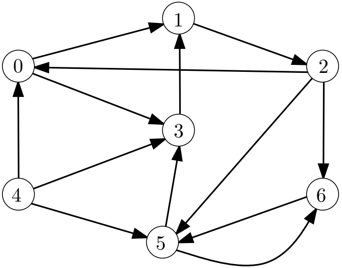
Sequence	Walk?	Path?	Cycle?
0 2 3	no	no	no
3 1 2	yes	yes	no
1 2 6 5 3 1	yes	no	yes
4 5 6 5	yes	no	no
4 3 5	no	no	no

Walks, Paths, and Cycles in a Digraph: an Example



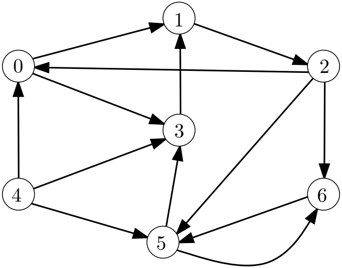
Sequence	Walk?	Path?	Cycle?
0 2 3	no	no	no
3 1 2	yes	yes	no
1 2 6 5 3 1	yes	no	yes
4 5 6 5	yes	no	no
4 3 5	no	no	no

Walks, Paths, and Cycles in a Digraph: an Example



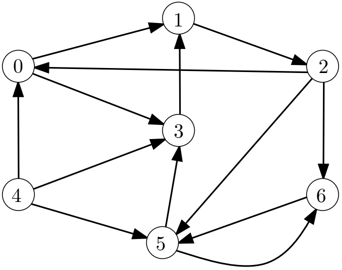
Sequence	Walk?	Path?	Cycle?
0 2 3	no	no	no
3 1 2	yes	yes	no
1 2 6 5 3 1	yes	no	yes
4 5 6 5	yes	no	no
4 3 5	no	no	no

Walks, Paths, and Cycles in a Digraph: an Example



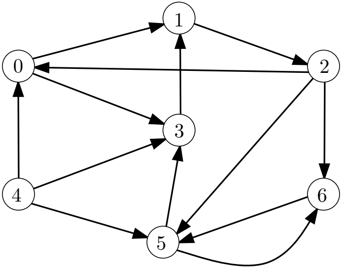
Sequence	Walk?	Path?	Cycle?
0 2 3	no	no	no
3 1 2	yes	yes	no
1 2 6 5 3 1	yes	no	yes
4 5 6 5	yes	no	no
4 3 5	no	no	no

Walks, Paths, and Cycles in a Digraph: an Example



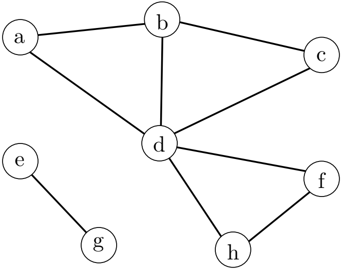
Sequence	Walk?	Path?	Cycle?
0 2 3	no	no	no
3 1 2	yes	yes	no
1 2 6 5 3 1	yes	no	yes
4 5 6 5	yes	no	no
4 3 5	no	no	no

Walks, Paths, and Cycles in a Digraph: an Example



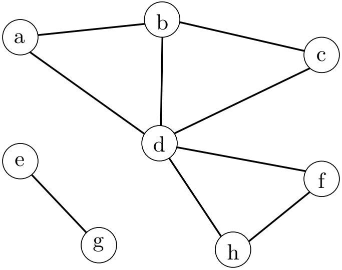
Sequence	Walk?	Path?	Cycle?
0 2 3	no	no	no
3 1 2	yes	yes	no
1 2 6 5 3 1	yes	no	yes
4 5 6 5	yes	no	no
4 3 5	no	no	no

Walks, Paths, and Cycles in a Graph: an Example



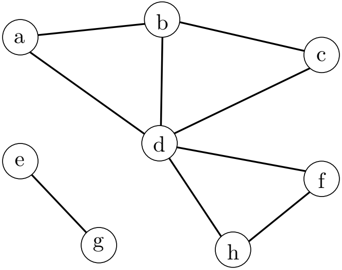
Sequence	Walk?	Path?	Cycle?
$a\,b\,c$	yes	yes	no
$e\,g\,e$	yes	no	no
$d\,b\,c\,d$	yes	no	yes
$d\,a\,d\,f$	yes	no	no
$a\,b\,d\,f\,h$	yes	yes	no

Walks, Paths, and Cycles in a Graph: an Example



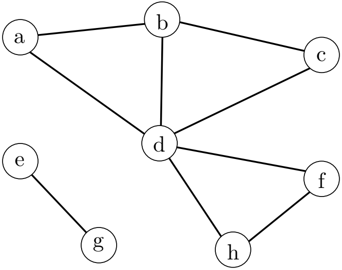
Sequence	Walk?	Path?	Cycle?
$a\,b\,c$	yes	yes	no
$e\,g\,e$	yes	no	no
$d\,b\,c\,d$	yes	no	yes
$d\,a\,d\,f$	yes	no	no
$a\,b\,d\,f\,h$	yes	yes	no

Walks, Paths, and Cycles in a Graph: an Example



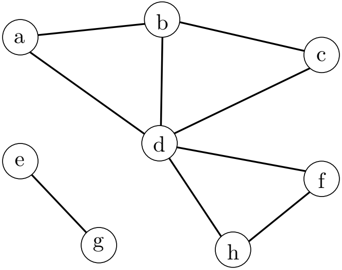
Sequence	Walk?	Path?	Cycle?
$a\,b\,c$	yes	yes	no
$e\,g\,e$	yes	no	no
$d\,b\,c\,d$	yes	no	yes
$d\,a\,d\,f$	yes	no	no
$a\,b\,d\,f\,h$	yes	yes	no

Walks, Paths, and Cycles in a Graph: an Example



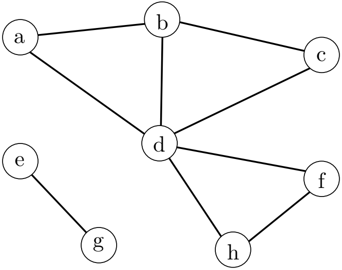
Sequence	Walk?	Path?	Cycle?
$a\,b\,c$	yes	yes	no
$e\,g\,e$	yes	no	no
$d\,b\,c\,d$	yes	no	yes
$d\,a\,d\,f$	yes	no	no
$a\,b\,d\,f\,h$	yes	yes	no

Walks, Paths, and Cycles in a Graph: an Example



Sequence	Walk?	Path?	Cycle?
$a\,b\,c$	yes	yes	no
$e\,g\,e$	yes	no	no
$d\,b\,c\,d$	yes	no	yes
$d\,a\,d\,f$	yes	no	no
$a\,b\,d\,f\,h$	yes	yes	no

Walks, Paths, and Cycles in a Graph: an Example



Sequence	Walk?	Path?	Cycle?
$a\,b\,c$	yes	yes	no
$e\,g\,e$	yes	no	no
$d\,b\,c\,d$	yes	no	yes
$d\,a\,d\,f$	yes	no	no
$a\,b\,d\,f\,h$	yes	yes	no

Digraph $G = (V, E)$: Distances and Diameter

The **distance**, $d(u, v)$, from a node u to a node v in G is the *minimum* length of a path from u to v.

- If no path exists, the distance is undefined or $+\infty$.
- For graphs, $d(u, v) = d(v, u)$ for all vertices u and v.

The **diameter** of G is the *maximum* distance $\max_{u,v \in V}[d(u, v)]$ between any two vertices.

The **radius** of G is $\min_{u \in V} \max_{v \in V}[d(u, v)]$.

Digraph $G = (V, E)$: Distances and Diameter

The **distance**, $d(u, v)$, from a node u to a node v in G is the *minimum* length of a path from u to v.

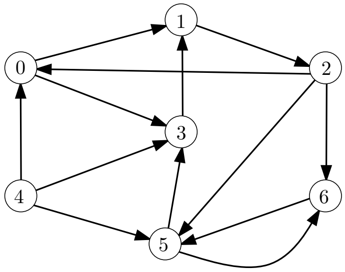
- If no path exists, the distance is undefined or $+\infty$.
- For graphs, $d(u, v) = d(v, u)$ for all vertices u and v.

The **diameter** of G is the *maximum* distance $\max_{u,v \in V}[d(u, v)]$ between any two vertices.

The **radius** of G is $\min_{u \in V} \max_{v \in V}[d(u, v)]$.

Path Distances in Digraphs: Examples

$$d(0,3) = \min\{\text{length}_{\text{of } 0,3}; \text{length}_{\text{of } 0,1,2,6,5,3}; \text{length}_{\text{of } 0,1,2,5,3}\} = \min\{1;5;4\} = 1$$



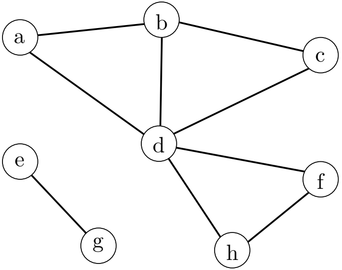
v	0	1	2	3	4	5	6
u=0	–	1	2	1	∞	3	3
u=1	2	–	1	3	∞	2	2
u=2	1	3	–	2	∞	1	1
u=3	3	1	2	–	∞	3	3
u=4	1	2	3	1	–	1	2
u=5	4	2	3	1	∞	–	1
u=6	5	3	4	2	∞	1	–

$d(0,1)=1,\ d(0,2)=2,\ d(0,5)=3,\ d(0,4)=\infty,\ d(5,5)=0,\ d(5,2)=3,$
$d(5,0)=4,\ d(4,6)=2,\ d(4,1)=2,\ d(4,2)=3$

Diameter: $\max\{1,2,1,\infty,3,\ldots,4,\ldots,5,\ldots,1\}=\infty$

Raduis: $\min\{\infty,\infty,\ldots,3,\infty,\infty\}=3$

Path Distances in Graphs: Examples



v

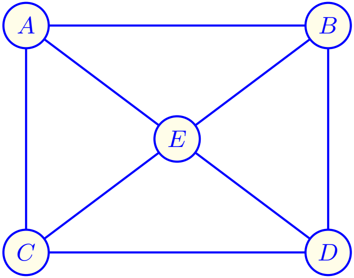
	a	b	c	d	e	f	g	h
u=a	0	1	2	1	∞	2	∞	2
u=b	1	0	1	1	∞	2	∞	2
u=c	2	1	0	1	∞	2	∞	2
u=d	1	1	1	0	∞	1	∞	1
u=e	∞	∞	∞	∞	0	∞	1	∞
u=f	2	2	2	1	∞	0	∞	1
u=g	∞	∞	∞	∞	1	∞	0	∞
u=h	2	2	2	1	∞	1	∞	0

$d(\mathrm{a},\mathrm{b}) = d(\mathrm{b},\mathrm{a}) = 1,\ d(\mathrm{a},\mathrm{c}) = d(\mathrm{c},\mathrm{a}) = 2,\ d(\mathrm{a},\mathrm{f}) = d(\mathrm{f},\mathrm{a}) = 2,$
$d(\mathrm{a},\mathrm{e}) = d(\mathrm{e},\mathrm{a}) = \infty,\ d(\mathrm{e},\mathrm{e}) = 0,\ d(\mathrm{e},\mathrm{g}) = d(\mathrm{g},\mathrm{e}) = 1,\ d(\mathrm{h},\mathrm{f}) = d(\mathrm{f},\mathrm{h}) = 1,$
$d(\mathrm{d},\mathrm{h}) = d(\mathrm{h},\mathrm{d}) = 1$

Diameter: $\max\{0, 1, 2, 1, \infty, 2, \ldots, 2, \ldots, 2, \ldots, 0\} = \infty$

Radius: $\min\{\infty, \ldots, \infty\} = \infty$

Diameter / Radius of an Unweighted Graph

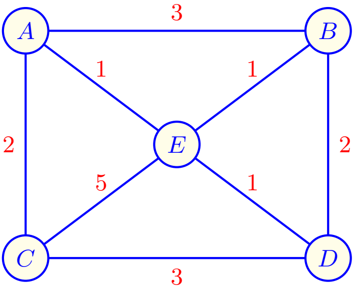


	A	B	C	D	E	$\max_v d(u,v)$
A	0	1	1	2	1	2
B	1	0	2	1	1	2
C	1	2	0	1	1	2
D	2	1	1	0	1	2
E	1	1	1	1	0	1

$$\begin{aligned} d(C,E) &= d(E,C) \\ &= \min\{1, 1+1, 1+1, 1+1+1, 1+1+1\} = 1 \\ d(B,C) &= d(C,B) \\ &= \min\{1+1, 1+1+1, 1+1, 1+1+1, 1+1, 1+1+1\} = 2 \end{aligned}$$

Radius $= 1$; diameter $= 2$.

Diameter / Radius of a Weighted Graph



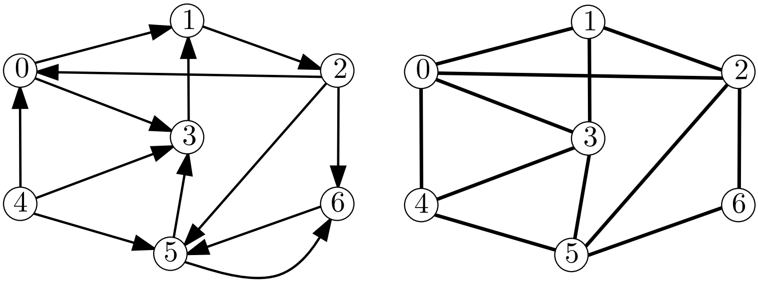
	A	B	C	D	E	$\max_v d(u,v)$
A	0	2	2	2	1	2
B	2	0	4	2	1	4
C	2	4	0	3	3	4
D	2	2	3	0	1	3
E	1	1	3	1	0	3

$$\begin{aligned} d(C,E) &= d(E,C) \\ &= \min\{5, 2+1, 3+1, 2+3+1, 3+2+1\} = 3 \\ d(B,C) &= d(C,B) \\ &= \min\{3+2, 1+1+2, 1+5, 1+1+3, 2+3, 2+1+5\} = 4 \end{aligned}$$

Radius $= 2$; diameter $= 4$.

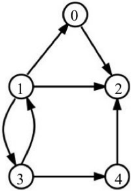
Underlying Graph of a Digraph

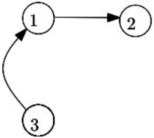
The underlying graph of a digraph $G = (V, E)$ is the graph $G' = (V, E')$ where $E' = \{\{u, v\} \mid (u, v) \in E\}$.



Sub(di)graphs

A **subdigraph** of a digraph $G = (V, E)$ is a digraph $G' = (V', E')$ where $V' \subseteq V$ and $E' \subseteq E$.

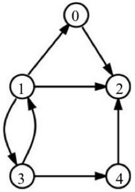


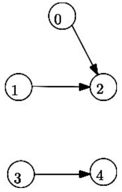


$$G = \left(\begin{array}{l} V = \{0, 1, 2, 3, 4\}, \\ E = \left\{ \begin{array}{l} (0,2), (1,0), (1,2), \\ (1,3), (3,1), (4,2), \\ (3,4) \end{array} \right\} \end{array} \right) \quad G' = \left(\begin{array}{l} V' = \{1, 2, 3\}, \\ E' = \{(1,2),\ (3,1)\} \end{array} \right)$$

Spanning Sub(di)graphs

A **spanning** subdigraph contains all nodes, that is, $V' = V$.

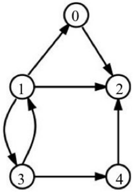


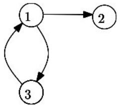


$$G = \begin{pmatrix} V = \{0,1,2,3,4\}, \\ E = \begin{Bmatrix} (0,2), (1,0), (1,2), \\ (1,3), (3,1), (4,2), \\ (3,4) \end{Bmatrix} \end{pmatrix} \quad G' = \begin{pmatrix} V' = \{0,1,2,3,4\}, \\ E' = \begin{Bmatrix} (0,2), \ (1,2), \\ (3,4) \end{Bmatrix} \end{pmatrix}$$

Induced Sub(di)graphs

The subdigraph **induced** by a subset V' of V is the digraph $G' = (V', E')$ where $E' = \{(u, v) \in E \mid u \in V' \text{ and } v \in V'\}$.





$$G = \begin{pmatrix} V = \{0, 1, 2, 3, 4\}, \\ E = \left\{ \begin{matrix} (0,2), (1,0), (1,2), \\ (1,3), (3,1), (4,2), \\ (3,4) \end{matrix} \right\} \end{pmatrix} \quad G' = \begin{pmatrix} V' = \{1, 2, 3\}, \\ E' = \left\{ \begin{matrix} (1,2), \ (1,3), \\ (3,1) \end{matrix} \right\} \end{pmatrix}$$

Digraphs: Computer Representation

For a digraph G of order n with the vertices, V, labelled $0, 1, \ldots, n-1$:

The adjacency matrix of G:

The $n \times n$ boolean matrix (often encoded with 0's and 1's) such that its entry (i, j) is true if and only if there is an arc (i, j) from the node i to node j.

An adjacency list of G:

A sequence of n sequences, $L_0, \ldots, L_{n-1}$, such that the sequence L_i contains all nodes of G that are adjacent to the node i.

Each sequence L_i may not be sorted! But we usually sort them.

Digraphs: Computer Representation

For a digraph G of order n with the vertices, V, labelled $0, 1, \ldots, n-1$:

The adjacency matrix of G:

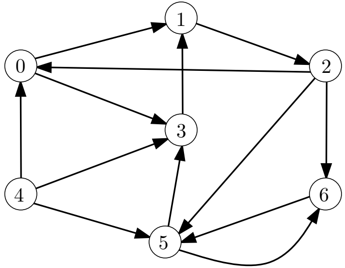
The $n \times n$ boolean matrix (often encoded with 0's and 1's) such that its entry (i, j) is true if and only if there is an arc (i, j) from the node i to node j.

An adjacency list of G:

A sequence of n sequences, $L_0, \ldots, L_{n-1}$, such that the sequence L_i contains all nodes of G that are adjacent to the node i.

Each sequence L_i may not be sorted! But we usually sort them.

Adjacency Matrix of a Digraph



Digraph $G = (V, E)$

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

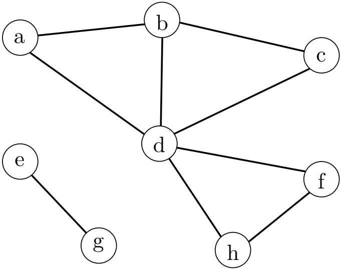
Adjacency matrix of G:

0 – a non-adjacent pair of vertices: $(i, j) \notin E$

1 – an adjacent pair of vertices: $(i, j) \in E$

The number of 1's in a row (column) is the out-(in-) degree of the related node.

Adjacency Lists of a Graph



Graph $G = (V, E)$

	symbolic	numeric
		8
$0 =$	a: b d	1 3
$1 =$	b: a c d	0 2 3
$2 =$	c: b d	1 3
$3 =$	d: a b c f h	0 1 2 5 7
$4 =$	e: g	6
$5 =$	f: d h	3 7
$6 =$	g: e	4
$7 =$	h: d f	3 5

Special cases can be stored more efficiently:

- A complete binary tree or a heap: in an array.
- A general rooted tree: in an array `pred` of size n;
  - `pred[i]` – a pointer to the parent of node i.

Digraph Operations w.r.t. Data Structures

Operation	Adjacency Matrix	Adjacency Lists
arc (i, j) exists?	is entry (i, j) 0 or 1	find j in list i
out-degree of i	scan row and sum 1's	size of list i
in-degree of i	scan column and sum 1's	for $j \neq i$, find i in list j
add arc (i, j)	change entry (i, j)	insert j in list i
delete arc (i, j)	change entry (i, j)	delete j from list i
add node	create new row/column	add new list at end
delete node i	delete row/column i and shuffle other entries	delete list i and for $j \neq i$, delete i from list j

Adjacency Lists / Matrices: Comparative Performance

$$G = (V, E) \quad \longrightarrow \quad n = |V|; \quad m = |E|$$

Operation	**array/array**	**list/list**
arc (i, j) exists?	$\Theta(1)$	$\Theta(\alpha)^{\circ)}$
out-degree of i	$\Theta(n)$	$\Theta(1)$
in-degree of i	$\Theta(n)$	$\Theta(n + m)$
add arc (i, j)	$\Theta(1)$	$\Theta(1)$
delete arc (i, j)	$\Theta(1)$	$\Theta(\alpha)$
add node	$\Theta(n)$	$\Theta(1)$
delete node i	$\Theta(n^2)$	$\Theta(n + m)$

°) Here, α denotes size of the adjacency list for vertex i.

General Graph Traversal Algorithm (Part 1)

algorithm `traverse`
  Input: digraph $G = (V, E)$
begin
  array $colour[n]$, $pred[n]$
  for $u \in V(G)$ **do**
    $colour[u] \leftarrow$ WHITE
  end for
  for $s \in V(G)$ **do**
    if $colour[s] =$ WHITE **then**
      `visit`(s)
    end if
  end for
  return $pred$
end

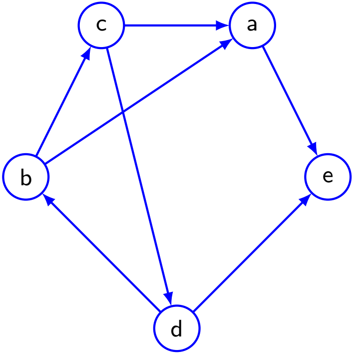
Three types of nodes each stage:

- WHITE – unvisited yet.
- GREY – visited, but some adjacent nodes are WHITE.
- BLACK – visited; only GREY adjacent nodes

General Graph Traversal Algorithm (Part 2)

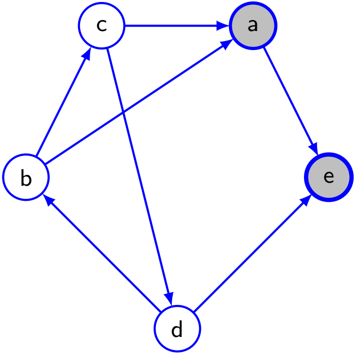
algorithm `visit`
    Input: node s of digraph G
begin
    $colour[s] \leftarrow$ GREY; $pred[s] \leftarrow$ NULL
    while there is a grey node **do**
        choose a grey node u
        if there is a white neighbour of u
            choose such a neighbour v
            $colour[v] \leftarrow$ GREY; $pred[v] \leftarrow u$
        else $colour[u] \leftarrow$ BLACK
        end if
    end while
end

Illustrating the General Traversal Algorithm



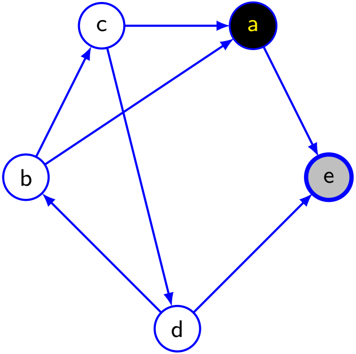
initialising all nodes WHITE

Illustrating the General Traversal Algorithm



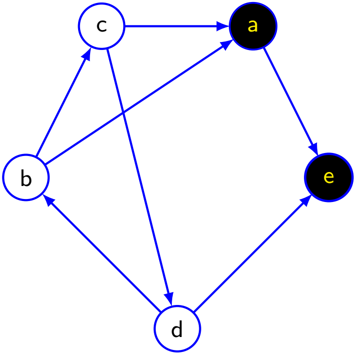
visit(a); $colour$[a] ← GREY
e is WHITE neighbour of a:
$colour$[e] ← GREY; $pred$[e] ← a

Illustrating the General Traversal Algorithm



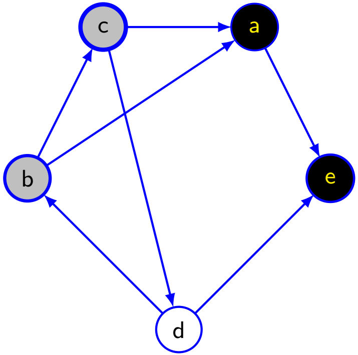
visit(a); $colour$[a] ← GREY
e is WHITE neighbour of a
  $colour$[e] ← GREY; $pred[e]$ ← a
choose GREY a: no WHITE neighbour:
  $colour$[a] ← BLACK

Illustrating the General Traversal Algorithm



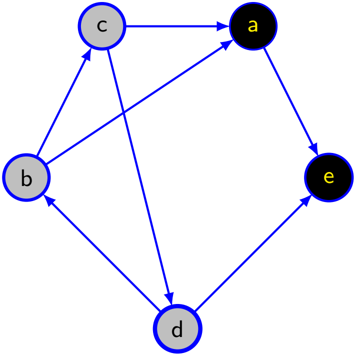
`visit`(a); $colour$[a] ← GREY
e is WHITE neighbour of a
$colour$[e] ← GREY; $pred$[e] ← a
choose GREY a: no WHITE neighbour:
$colour$[a] ← BLACK
choose GREY e: no WHITE neighbour:
$colour$[e] ← BLACK

Illustrating the General Traversal Algorithm



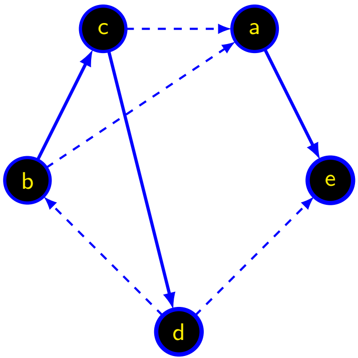
visit(b); $colour$[b] ← GREY
c is WHITE neighbour of b
$colour$[c] ← GREY; $pred$[c] ← b

Illustrating the General Traversal Algorithm



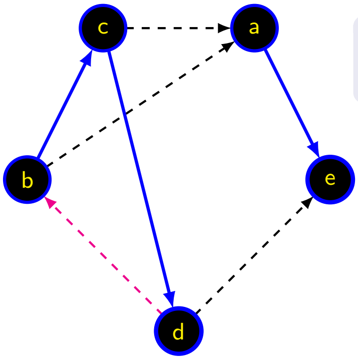
`visit`(b); $colour$[b] ← GREY
c is WHITE neighbour of b
  $colour$[c] ← GREY; $pred[c]$ ← b
d is WHITE neighbour of c
  $colour$[d] ← GREY; $pred[d]$ ← c

Illustrating the General Traversal Algorithm



`visit`(b); $colour$[b] ← GREY
c is WHITE neighbour of b
    $colour$[c] ← GREY; $pred[c]$ ← b
d is WHITE neighbour of c
    $colour$[d] ← GREY; $pred[d]$ ← c
no more WHITE nodes:
    $colour$[d] ← BLACK
    $colour$[c] ← BLACK
    $colour$[b] ← BLACK

Classes of Traversal Arcs



Search forest F: a set of disjoint trees spanning a digraph G after its traversal.

An arc $(u, v) \in E(G)$ is called a **tree arc** if it belongs to one of the trees of F

The arc (u, v), which is not a tree arc, is called:

- a **forward arc** if u is an ancestor of v in F;
- a **back arc** if u is a descendant of v in F, and
- a **cross arc** if neither u nor v is an ancestor of the other in F.

Basic Facts about Traversal Trees (for further analyses)

Theorem 5.2: Suppose we have run `traverse` on G, resulting in a search forest F.

1. If T_1 and T_2 are different trees in F and T_1 was explored before T_2, then there are no arcs from T_1 to T_2.
2. If G is a graph, then there can be no edges joining different trees of F.
3. If $v, w \in V(G)$; v is visited before w, and w is reachable from v in G, then v and w belong to the same tree of F.
4. If $v, w \in V(G)$ and v and w belong to the same tree T in F, then any path from v to w in G must have all nodes in T.

Basic Facts about Traversal Trees (for further analyses)

Theorem 5.2: Suppose we have run `traverse` on G, resulting in a search forest F.

1. If T_1 and T_2 are different trees in F and T_1 was explored before T_2, then there are no arcs from T_1 to T_2.
2. If G is a graph, then there can be no edges joining different trees of F.
3. If $v, w \in V(G)$; v is visited before w, and w is reachable from v in G, then v and w belong to the same tree of F.
4. If $v, w \in V(G)$ and v and w belong to the same tree T in F, then any path from v to w in G must have all nodes in T.

Basic Facts about Traversal Trees (for further analyses)

Theorem 5.2: Suppose we have run `traverse` on G, resulting in a search forest F.

1. If T_1 and T_2 are different trees in F and T_1 was explored before T_2, then there are no arcs from T_1 to T_2.
2. If G is a graph, then there can be no edges joining different trees of F.
3. If $v, w \in V(G)$; v is visited before w, and w is reachable from v in G, then v and w belong to the same tree of F.
4. If $v, w \in V(G)$ and v and w belong to the same tree T in F, then any path from v to w in G must have all nodes in T.

Basic Facts about Traversal Trees (for further analyses)

Theorem 5.2: Suppose we have run `traverse` on G, resulting in a search forest F.

1. If T_1 and T_2 are different trees in F and T_1 was explored before T_2, then there are no arcs from T_1 to T_2.
2. If G is a graph, then there can be no edges joining different trees of F.
3. If $v, w \in V(G)$; v is visited before w, and w is reachable from v in G, then v and w belong to the same tree of F.
4. If $v, w \in V(G)$ and v and w belong to the same tree T in F, then any path from v to w in G must have all nodes in T.

Basic Facts about Traversal Trees (for further analyses)

Theorem 5.2: Suppose we have run `traverse` on G, resulting in a search forest F.

1. If T_1 and T_2 are different trees in F and T_1 was explored before T_2, then there are no arcs from T_1 to T_2.
2. If G is a graph, then there can be no edges joining different trees of F.
3. If $v, w \in V(G)$; v is visited before w, and w is reachable from v in G, then v and w belong to the same tree of F.
4. If $v, w \in V(G)$ and v and w belong to the same tree T in F, then any path from v to w in G must have all nodes in T.

Run-time Analysis of Algorithm `traverse`

In the **while-loop** of subroutine `visit` let:

- a (A) be lower (upper) time bound to choose a GREY node.
- b (B) be lower (upper) time bound to choose a WHITE neighbour.

Given a (di)graph $G = (V, E)$ of order $n = |V|$ and size $m = |E|$, the running time of `traverse` is:

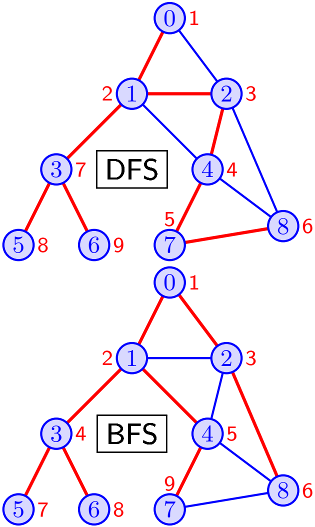
- $\mathrm{O}(An + Bm)$ and $\Omega(an + bm)$ with adjacency lists, and
- $\mathrm{O}(An + Bn^2)$ and $\Omega(an + bn^2)$ with an adjacency matrix.

Time to find a GREY node:	$\mathrm{O}(An)$ and $\Omega(an)$	
Time to find a WHITE neighbour:	$\mathrm{O}(Bm)$ and $\Omega(bm)$	(adjacency lists)
	$\mathrm{O}(Bn^2)$ and $\Omega(bn^2)$	(an adjacency matrix)

- Generally, A, B, a, b may depend on n.
- A more detailed analysis depends on the rules used.

Main Rules for Choosing Next Nodes

- Depth-first search (DFS):
  - Starting at a node v.
  - Searching as far away from v as possible via neighbours.
  - Continue from the next neighbour until no more new nodes.
- Breadth-first search (BFS):
  - Starting at a node v.
  - Searching through all its neighbours, then through all their neighbours, etc.
  - Continue until no more new nodes.
- More complicated priority-first search (PFS).



Depth-first Search (DFS) Algorithm (Part 1)

algorithm `dfs`
    Input: digraph $G = (V(G), E(G))$
begin
    stack S; array $colour[n], pred[n], seen[n], done[n]$
    for $u \in V(G)$ **do**
        $colour[u] \leftarrow$ WHITE; $pred[u] \leftarrow$ NULL
    end for
    $time \leftarrow 0$
    for $s \in V(G)$ **do**
        if $colour[s] =$ WHITE **then**
            `dfsvisit`(s)
        end if
    end for
    return $pred, seen, done$
end

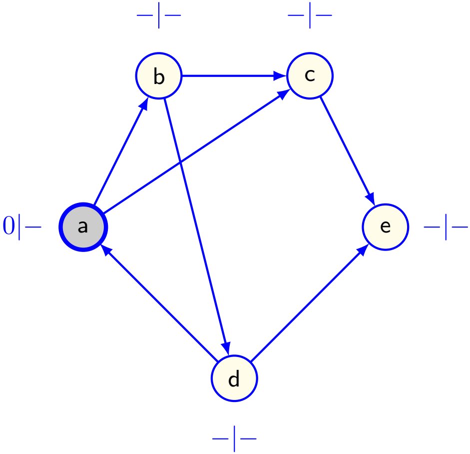
Depth-first Search (DFS) Algorithm (Part 2)

algorithm `dfsvisit`
    Input: node s
begin
    $colour[s] \leftarrow$ GREY; $seen[s] \leftarrow time++$;
    S.`push_top`(s)
    while not S.`isempty`() **do**
        $u \leftarrow S$.`get_top`()
        if there is a v adjacent to u **and** $colour[v] =$ WHITE **then**
            $colour[v] \leftarrow$ GREY; $pred[v] \leftarrow u$
            $seen[v] \leftarrow time++$; S.`push_top`(v)
        else S.`del_top`();
            $colour[u] \leftarrow$ BLACK; $done[u] \leftarrow time++$;
        end if
    end while
end

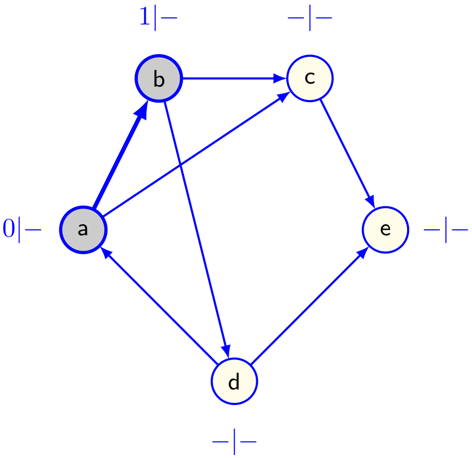
Recursive View of DFS Algorithm

algorithm `rec_dfs_visit`
    Input: node s
begin
    $colour[s] \leftarrow$ GREY
    $seen[s] \leftarrow time++$
    for each v adjacent to s **do**
        if $colour[v] =$ WHITE **then**
            $pred[v] \leftarrow s$
            `rec_dfs_visit`(v)
        end if
    end for
    $colour[s] \leftarrow$ BLACK
    $done[s] \leftarrow time++$
end

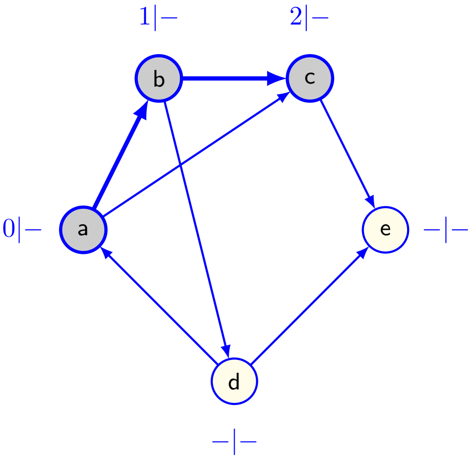
DFS: An Example $(seen[v] \mid done[v])$: $time = 0; 1$



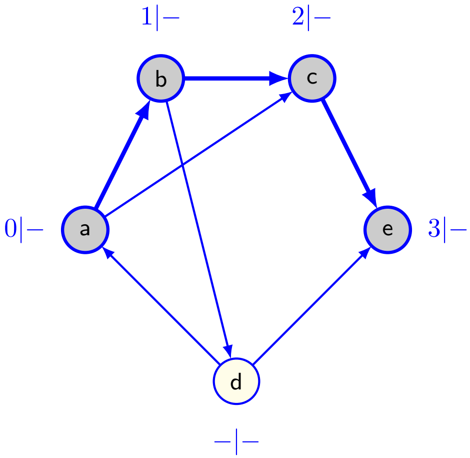
DFS: An Example $(seen[v] \mid done[v])$: $time = 1; 2$



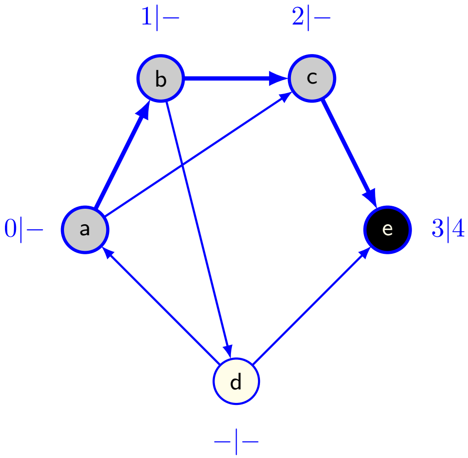
DFS: An Example ($seen[v] \mid done[v]$): $time = 2, 3$



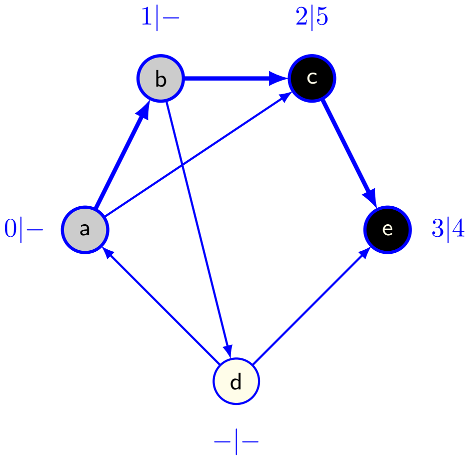
DFS: An Example ($seen[v] \mid done[v]$): $time = 3; 4$



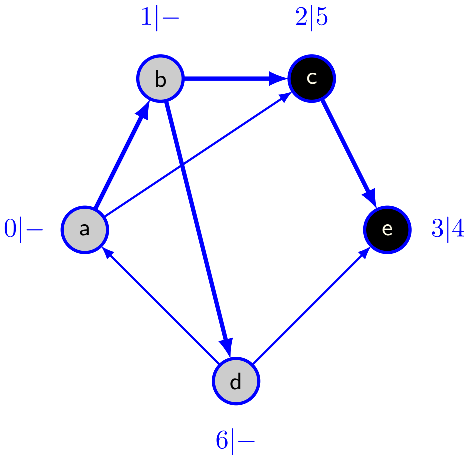
DFS: An Example ($seen[v] \mid done[v]$): $time = 4; 5$



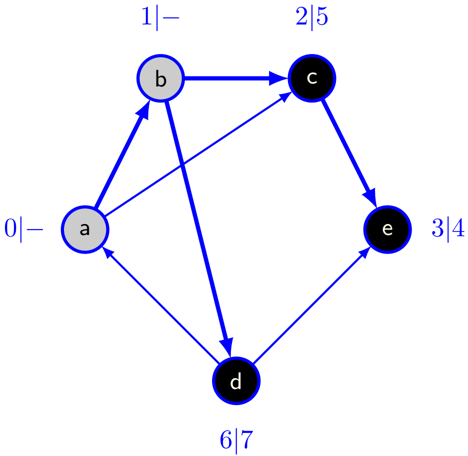
DFS: An Example ($seen[v] \mid done[v]$: $time = 5, 6$)



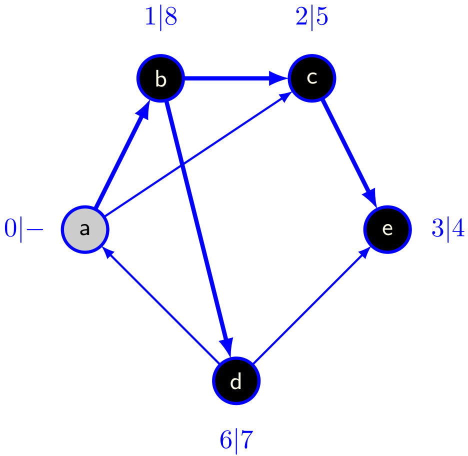
DFS: An Example ($seen[v] \mid done[v]$): $time = 6, 7$



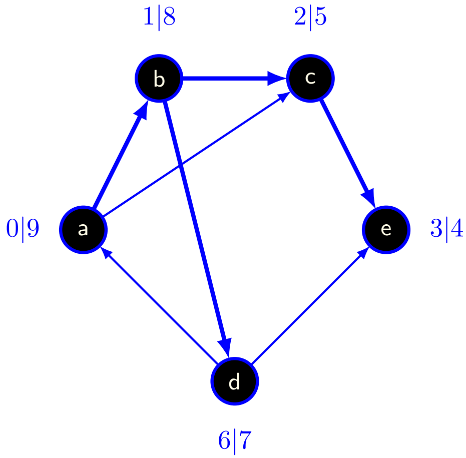
DFS: An Example $(seen[v] \mid done[v])$: $time = 7, 8$



DFS: An Example $(seen[v] \mid done[v])$: $time = 8, 9$



DFS: An Example ($seen[v] \mid done[v]$): $time = 9, 10$



Basic Properties of Depth-first Search

Next GREY node chosen $\leftarrow$ the last one coloured GREY thus far.

- Data structure for this "last in, first out" order – a **stack**.

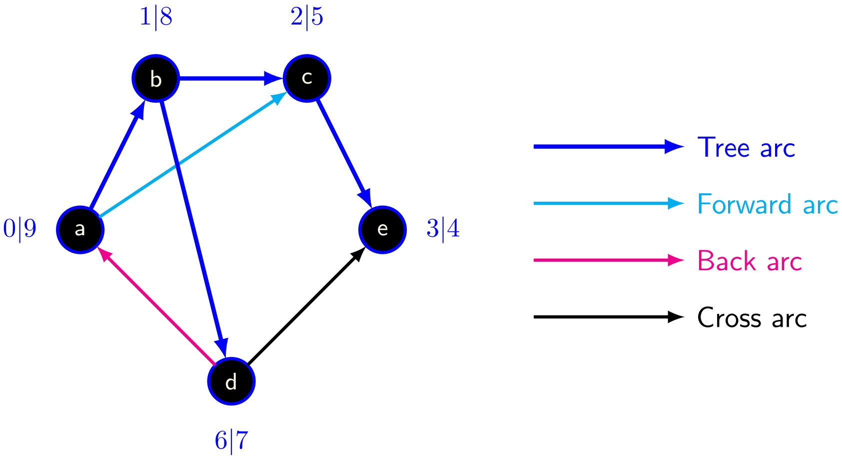
Each call to `dfs_visit`(v) terminates only when all nodes reachable from v via a path of WHITE nodes have been seen.

If (v, w) is an arc, then for a

- tree or forward arc: $seen[v] < seen[w] < done[w] < done[v]$
  - Example in Slide 52: $(a, b) : 0 < 1 < 8 < 9$; $(b, c) : 1 < 2 < 5 < 8$; $(a, c) : 0 < 2 < 5 < 9$;
- back arc: $seen[w] < seen[v] < done[v] < done[w]$:
  - Example in Slide 52: $(d, a) : 0 < 6 < 7 < 9$;
- cross arc: $seen[w] < done[w] < seen[v] < done[v]$.
  - Example in Slide 52: $(d, e) : 3 < 4 < 6 < 7$;

Hence, there are no cross edges on a graph.

Tree, Forward, Back, and Cross Arcs (Example in Slide 52)



Using DFS to Determine Ancestors of a Tree

Theorem 5.5

Suppose that DFS on a digraph G results in a search forest F. Let $v, w \in V(G)$ and $seen[v] < seen[w]$.

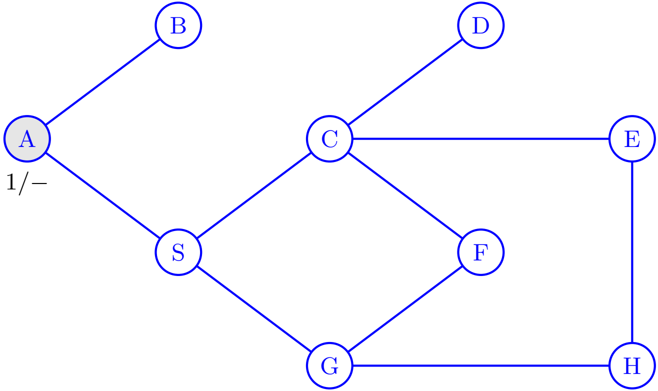
1. If v is an ancestor of w in F, then
$$seen[v] < seen[w] < done[w] < done[v].$$
2. If v is not an ancestor of w in F, then
$$seen[v] < done[v] < seen[w] < done[w].$$

Proof.

1. This part follows from the recursive nature of DFS.
2. If v is not an ancestor of w in F, then w is also not an ancestor v.
   - Thus v is in a subtree, which was completely explored before the subtree of w.

□

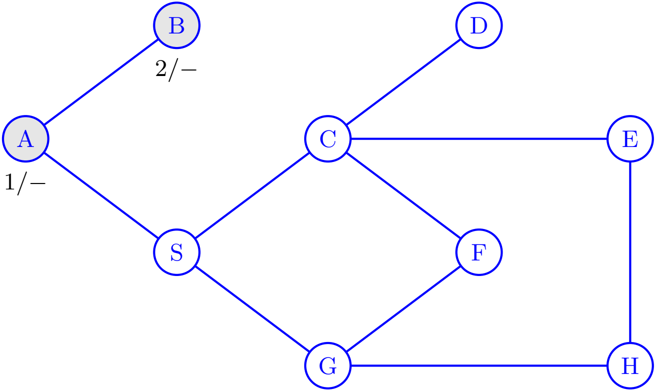
DFS: *seen/done*: step 1



Preorder (WHITE to GREY): *seen* A
1

Postorder (GREY to BLACK) *done*

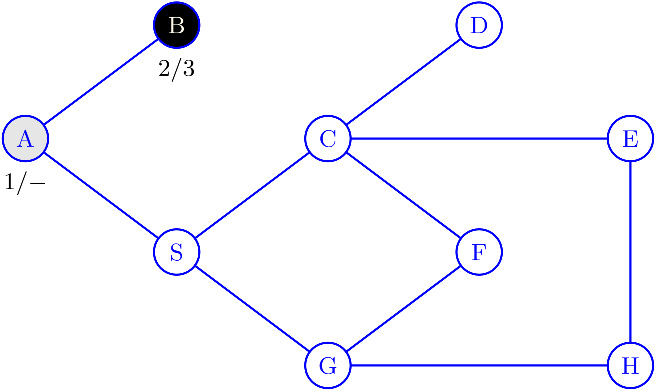
DFS: *seen/done*: step 2



Preorder (WHITE to GREY): *seen* A B
1 2

Postorder (GREY to BLACK) *done*

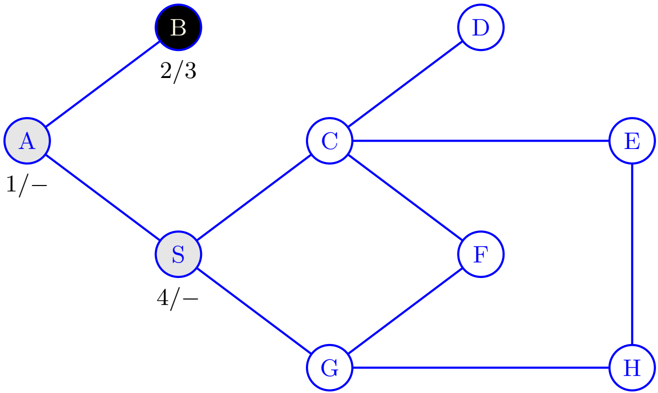
DFS: *seen/done*: step 3



Preorder (WHITE to GREY): *seen* A B
1 2

Postorder (GREY to BLACK) *done* B
3

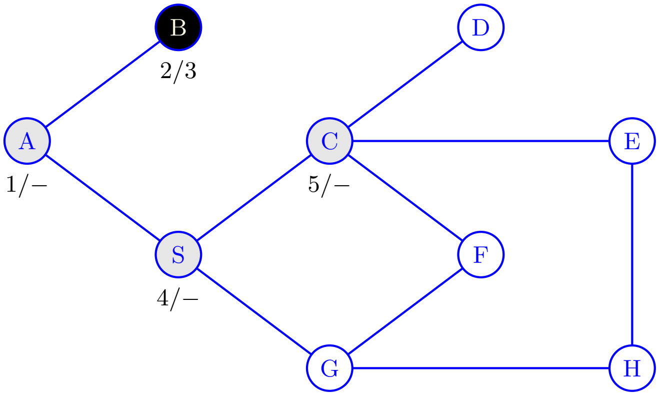
DFS: *seen*/*done*: step 4



Preorder (WHITE to GREY): *seen* A B S
1 2 4

Postorder (GREY to BLACK) *done* B
3

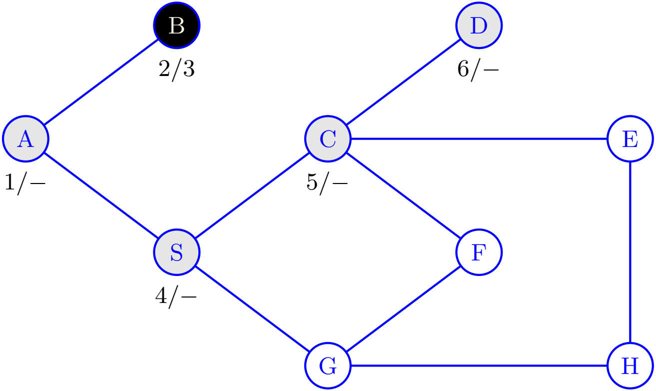
DFS: *seen/done*: step 5



Preorder (WHITE to GREY): *seen* A B S C
1 2 4 5

Postorder (GREY to BLACK) *done* B
3

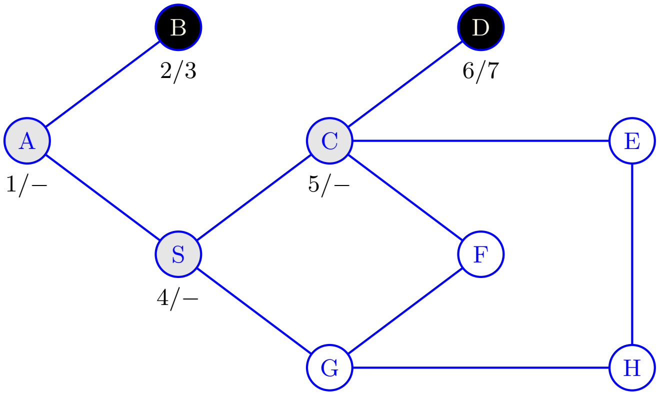
DFS: *seen/done*: step 6



Preorder (WHITE to GREY): *seen* A B S C D
1 2 4 5 6

Postorder (GREY to BLACK) *done* B
3

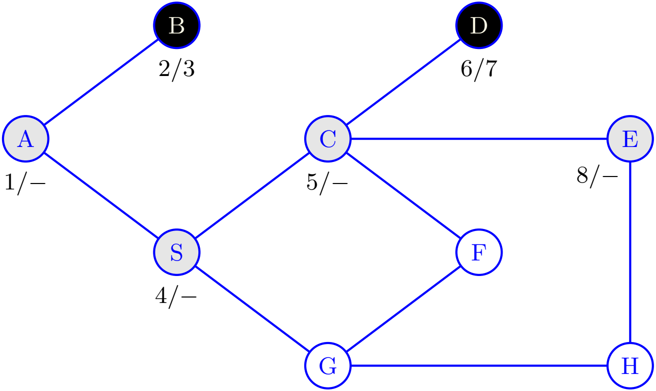
DFS: *seen/done*: step 7



Preorder (WHITE to GREY): *seen* A B S C D
1 2 4 5 6

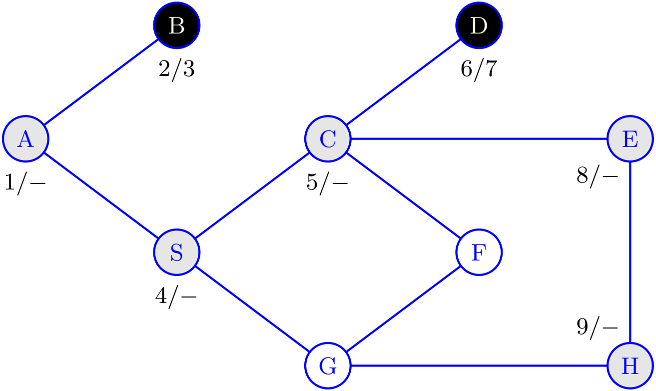
Postorder (GREY to BLACK) *done* B D
3 7

DFS: *seen/done*: step 8



Preorder (WHITE to GREY): *seen*	A	B	S	C	D	E
	1	2	4	5	6	8
Postorder (GREY to BLACK) *done*	B	D				
	3	7				

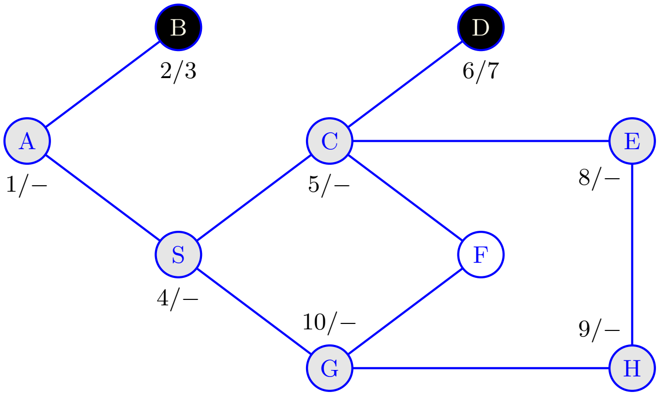
DFS: *seen*/*done*: step 9



Preorder (WHITE to GREY): *seen* A B S C D E H
1 2 4 5 6 8 9

Postorder (GREY to BLACK) *done* B D
3 7

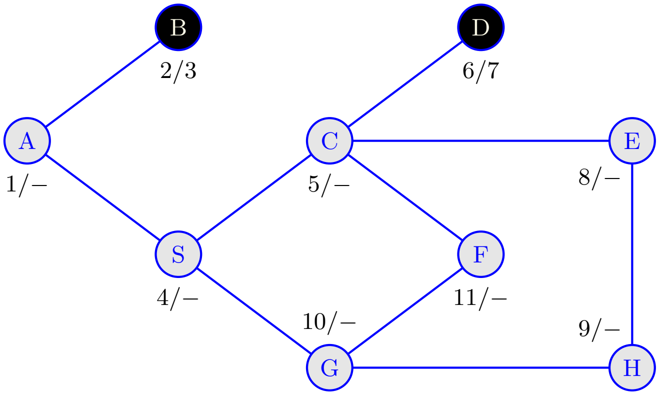
DFS: *seen*/*done*: step 10



Preorder (WHITE to GREY): *seen* A B S C D E H G
1 2 4 5 6 8 9 10

Postorder (GREY to BLACK) *done* B D
3 7

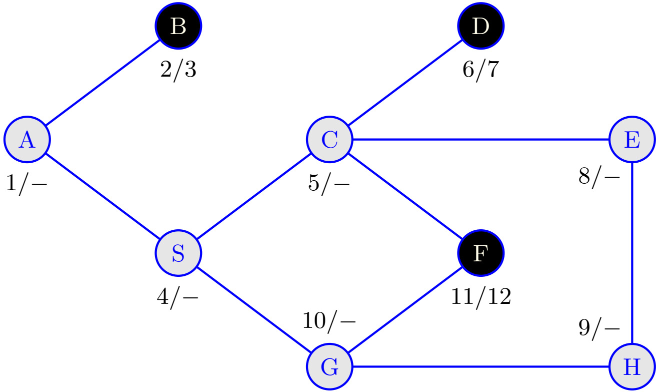
DFS: *seen*/*done*: step 11



Preorder (WHITE to GREY): *seen* A B S C D E H G F
1 2 4 5 6 8 9 10 11

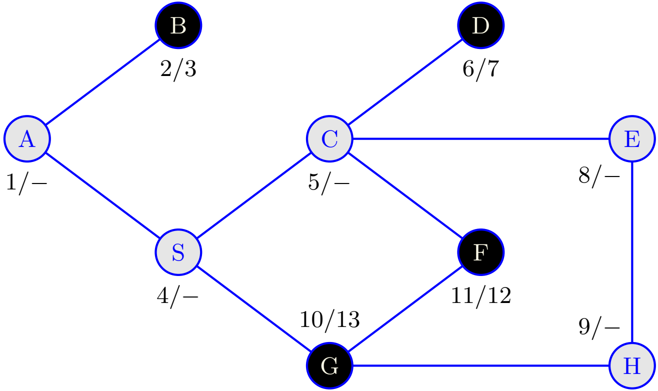
Postorder (GREY to BLACK) *done* B D
3 7

DFS: *seen/done*: step 12



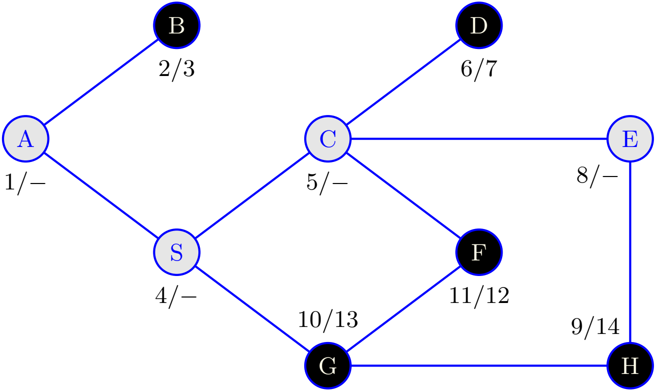
Preorder (WHITE to GREY):	*seen*	A	B	S	C	D	E	H	G	F
		1	2	4	5	6	8	9	10	11
Postorder (GREY to BLACK)	*done*	B	D	F						
		3	7	12						

DFS: *seen/done*: step 13



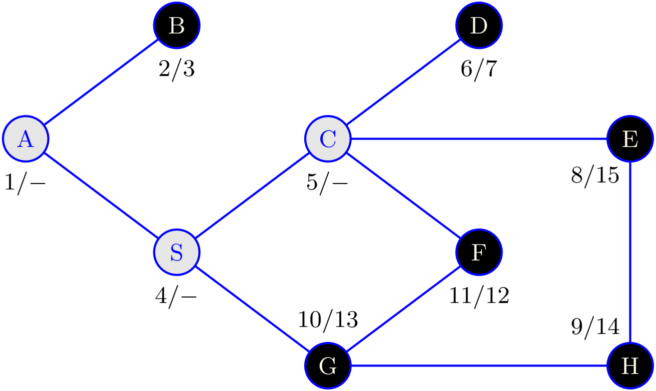
Preorder (WHITE to GREY): *seen*	A	B	S	C	D	E	H	G	F
	1	2	4	5	6	8	9	10	11
Postorder (GREY to BLACK) *done*	B	D	F	G					
	3	7	12	13					

DFS: *seen*/*done*: step 14



Preorder (WHITE to GREY):	*seen*	A	B	S	C	D	E	H	G	F
		1	2	4	5	6	8	9	10	11
Postorder (GREY to BLACK)	*done*	B	D	F	G	H				
		3	7	12	13	14				

DFS: *seen/done*: step 15



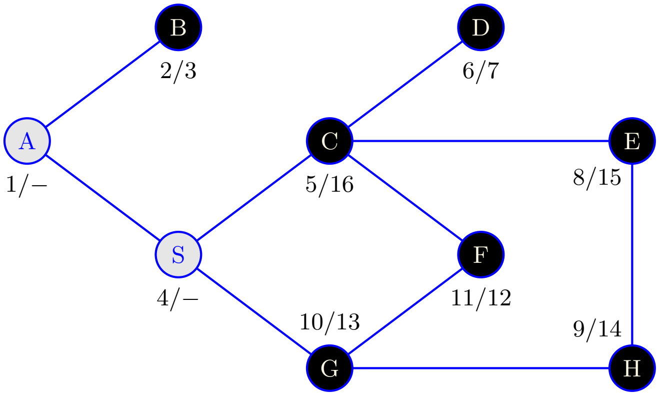
Preorder (WHITE to GREY): *seen*

A	B	S	C	D	E	H	G	F
1	2	4	5	6	8	9	10	11

Postorder (GREY to BLACK) *done*

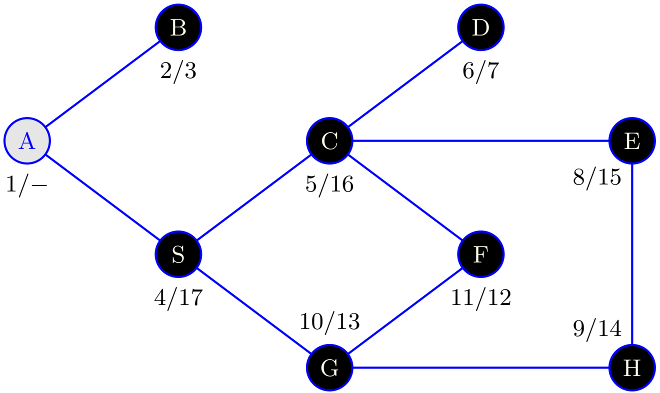
B	D	F	G	H	E
3	7	12	13	14	15

DFS: *seen*/*done*: step 16



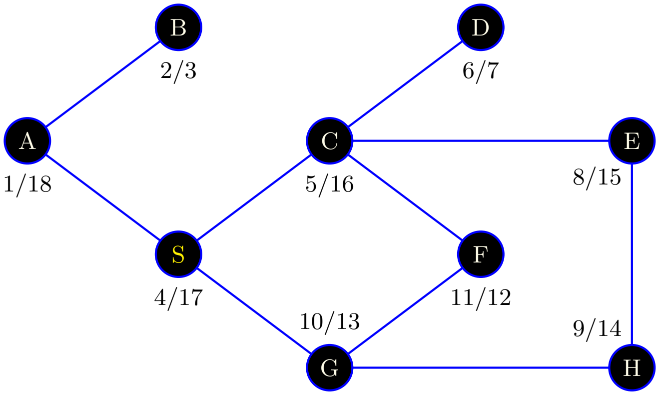
Preorder (WHITE to GREY): *seen*	A	B	S	C	D	E	H	G	F
	1	2	4	5	6	8	9	10	11
Postorder (GREY to BLACK) *done*	B	D	F	G	H	E	C		
	3	7	12	13	14	15	16		

DFS: *seen*/*done*: step 17



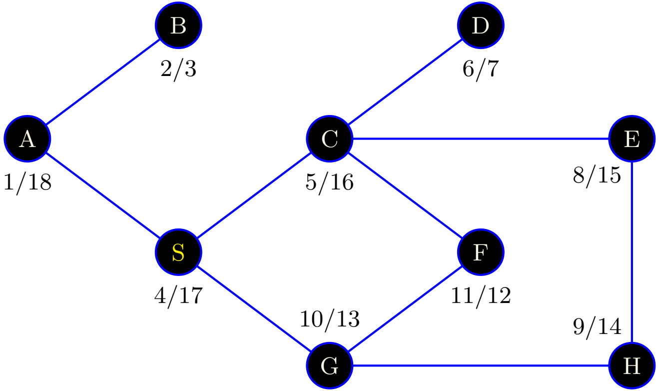
Preorder (WHITE to GREY):	*seen*	A	B	S	C	D	E	H	G	F
		1	2	4	5	6	8	9	10	11
Postorder (GREY to BLACK)	*done*	B	D	F	G	H	E	C	S	
		3	7	12	13	14	15	16	17	

DFS: *seen*/*done*: step 18



Preorder (WHITE to GREY): *seen*	A	B	S	C	D	E	H	G	F
	1	2	4	5	6	8	9	10	11
Postorder (GREY to BLACK) *done*	B	D	F	G	H	E	C	S	A
	3	7	12	13	14	15	16	17	18

Determining Ancestors of a Tree: Examples



$$\begin{aligned}
A \rightarrow B: &\quad seen[A] = 1 < seen[B] = 2 < done[B] = 3 < done[A] = 18 \\
S \rightarrow H: &\quad seen[S] = 4 < seen[H] = 9 < done[H] = 14 < done[S] = 17 \\
B \nrightarrow D: &\quad seen[B] = 2 < done[B] = 3 < seen[D] = 6 < done[D] = 7 \\
D \nrightarrow G: &\quad seen[D] = 6 < done[D] = 7 < seen[G] = 10 < done[G] = 13
\end{aligned}$$