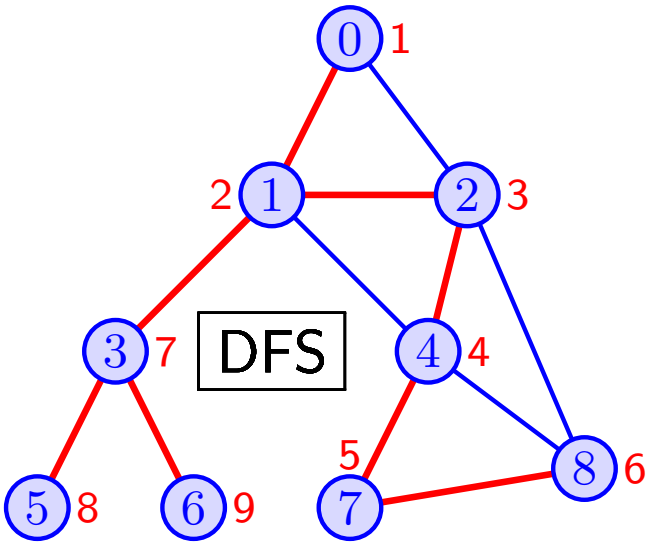
Graphs and Digraphs

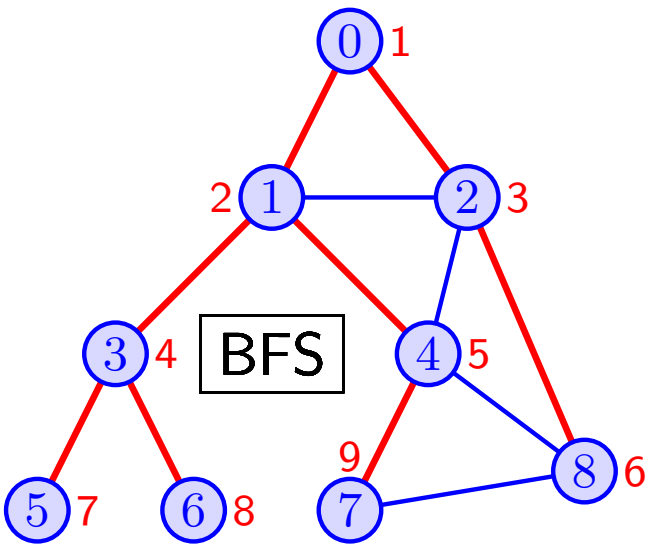
BFS Priority search DAG Connectivity

Lecturer: Georgy Gimel'farb

COMPSCI 220 Algorithms and Data Structures

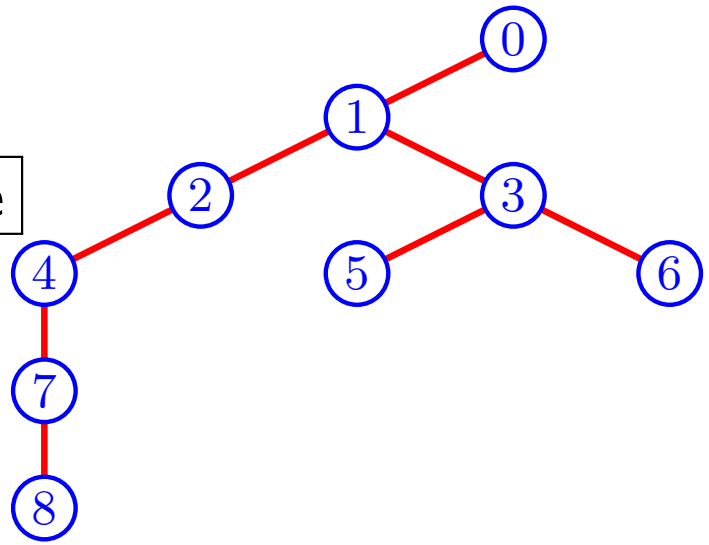
Breadth-First Vs. Depth-first Search



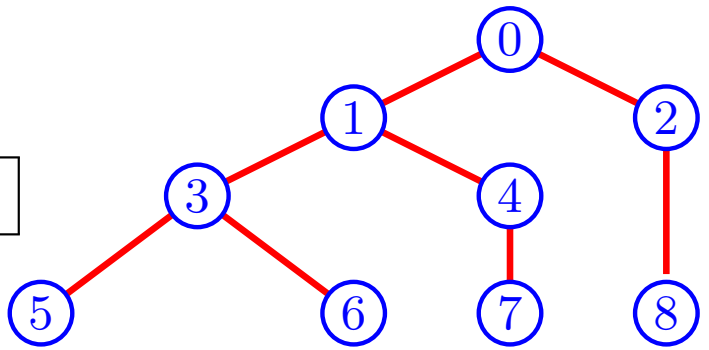


Node choice convention: the lowest index

DFS tree



BFS tree



Breadth-first Search (BFS) Algorithm (Part 1)

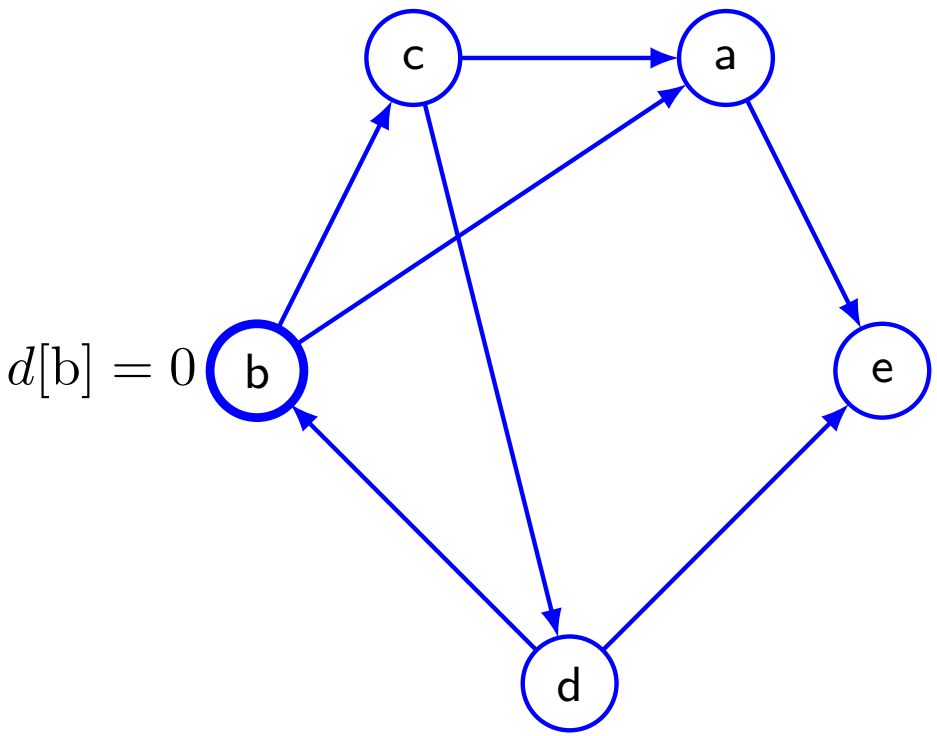
algorithm `bfs`
  Input: digraph $G = \{V(G), E(G)\}$
begin
  queue Q
  array $colour[n], pred[n], d[n]$ (number of steps from the root)
  for $u \in V(G)$ **do**
    $colour[u] \leftarrow$ WHITE; $pred[u] \leftarrow$ NULL
  end for
  for $s \in V(G)$ **do**
    if $colour[s] =$ WHITE **then**
      `bfsvisit`(s)
    end if
  end for
  return $pred, d$
end

Breadth-first Search (BFS) Algorithm (Part 2)

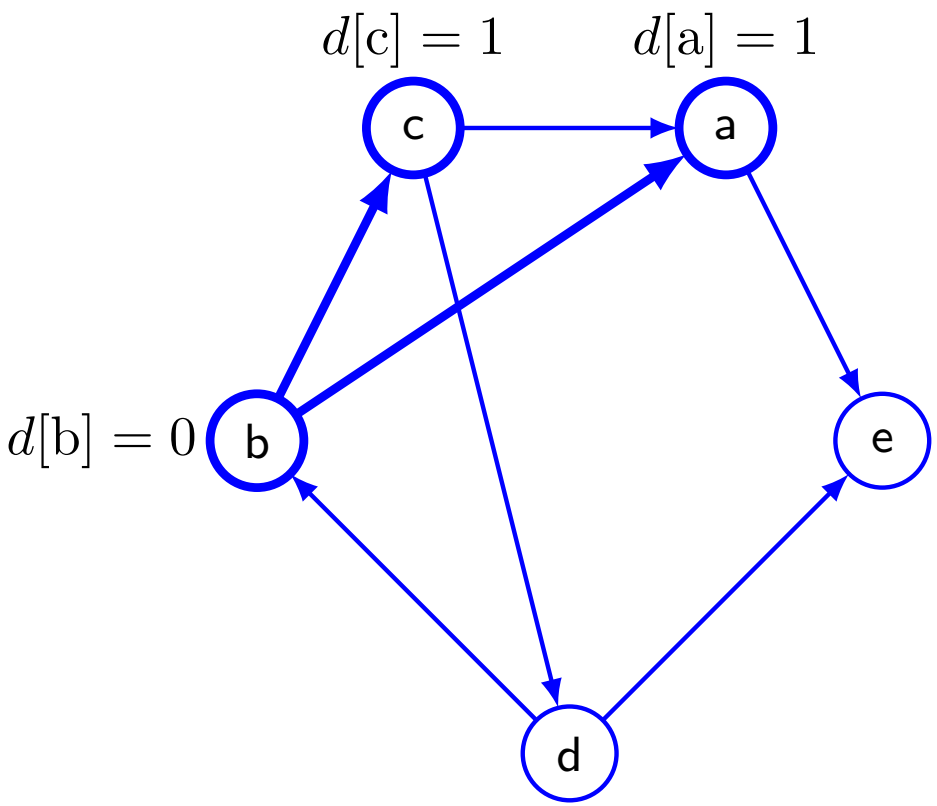
algorithm `bfsvisit`
  Input: node s
begin
  $colour[s] \leftarrow$ GREY; $d[s] \leftarrow 0$; Q.`enqueue`(s)
  while not Q.`isempty`() **do**
    $u \leftarrow$ Q.`get_head`()
    for each v adjacent to u **do**
      if $colour[v] =$ WHITE **then**
        $colour[v] \leftarrow$ GREY; $pred[v] \leftarrow u$; $d[v] \leftarrow d[u] + 1$
        Q.`enqueue`(v)
      end if
    end for
    Q.`dequeue`(); $colour[u] \leftarrow$ BLACK
  end while
end

BFS: Example 1

Queue: b

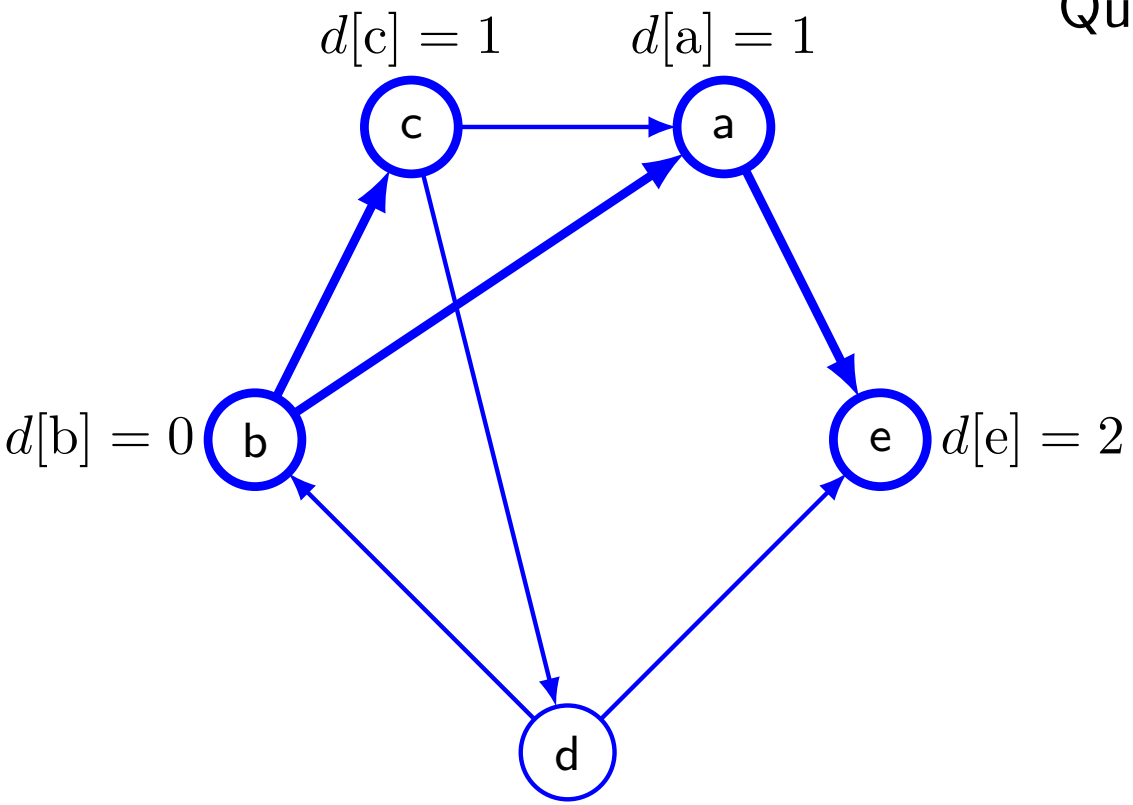


BFS: Example 1



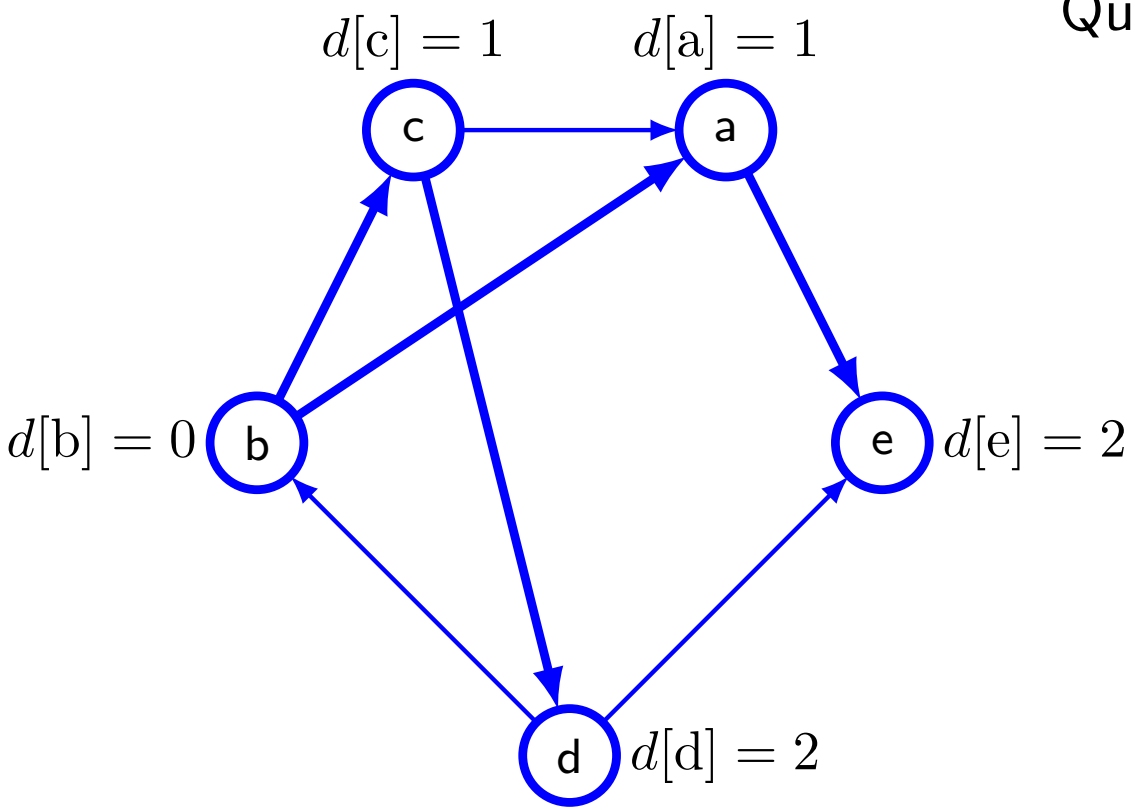
Queue: a c

BFS: Example 1



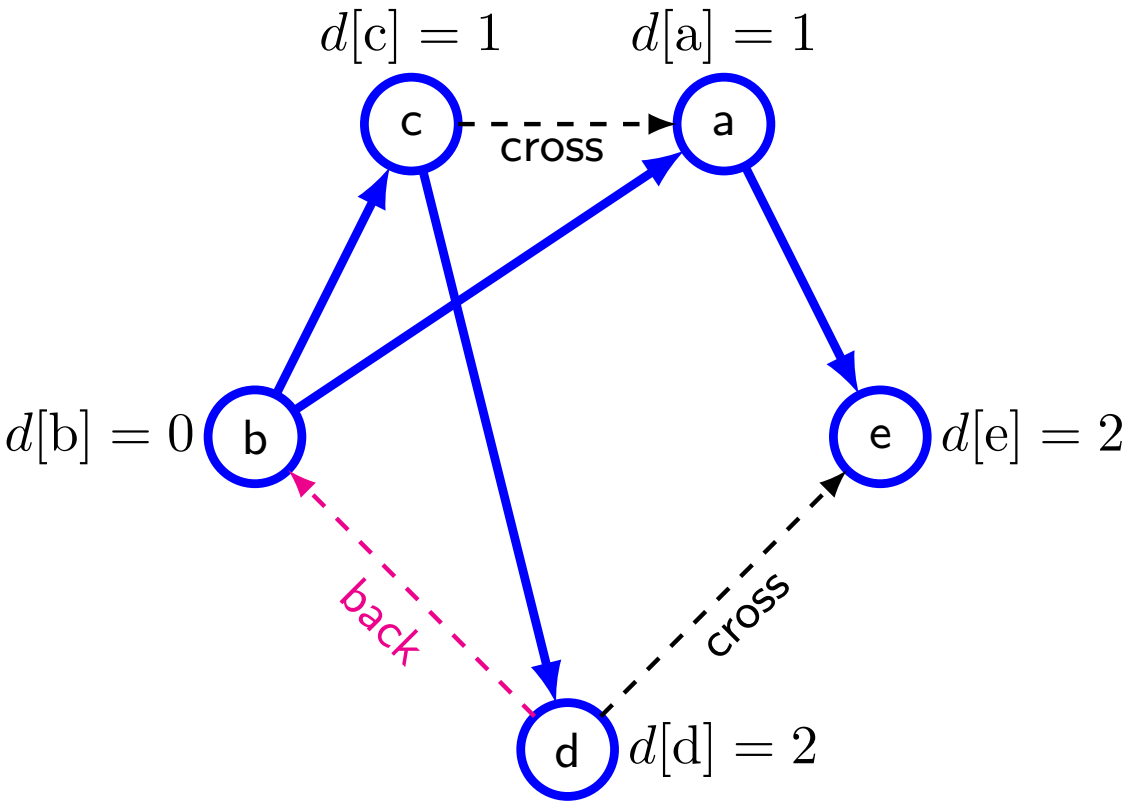
Queue: c e

BFS: Example 1

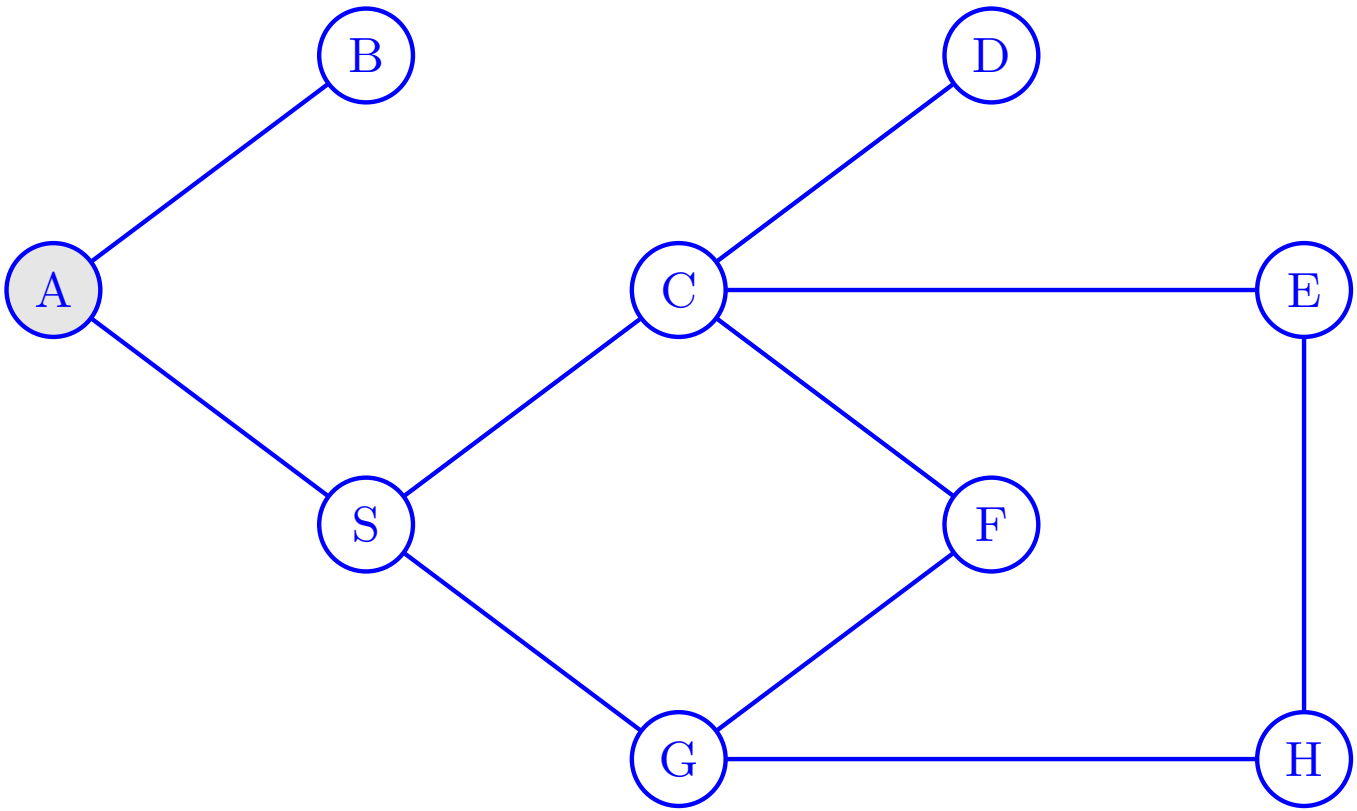


Queue: e d

BFS: Example 1



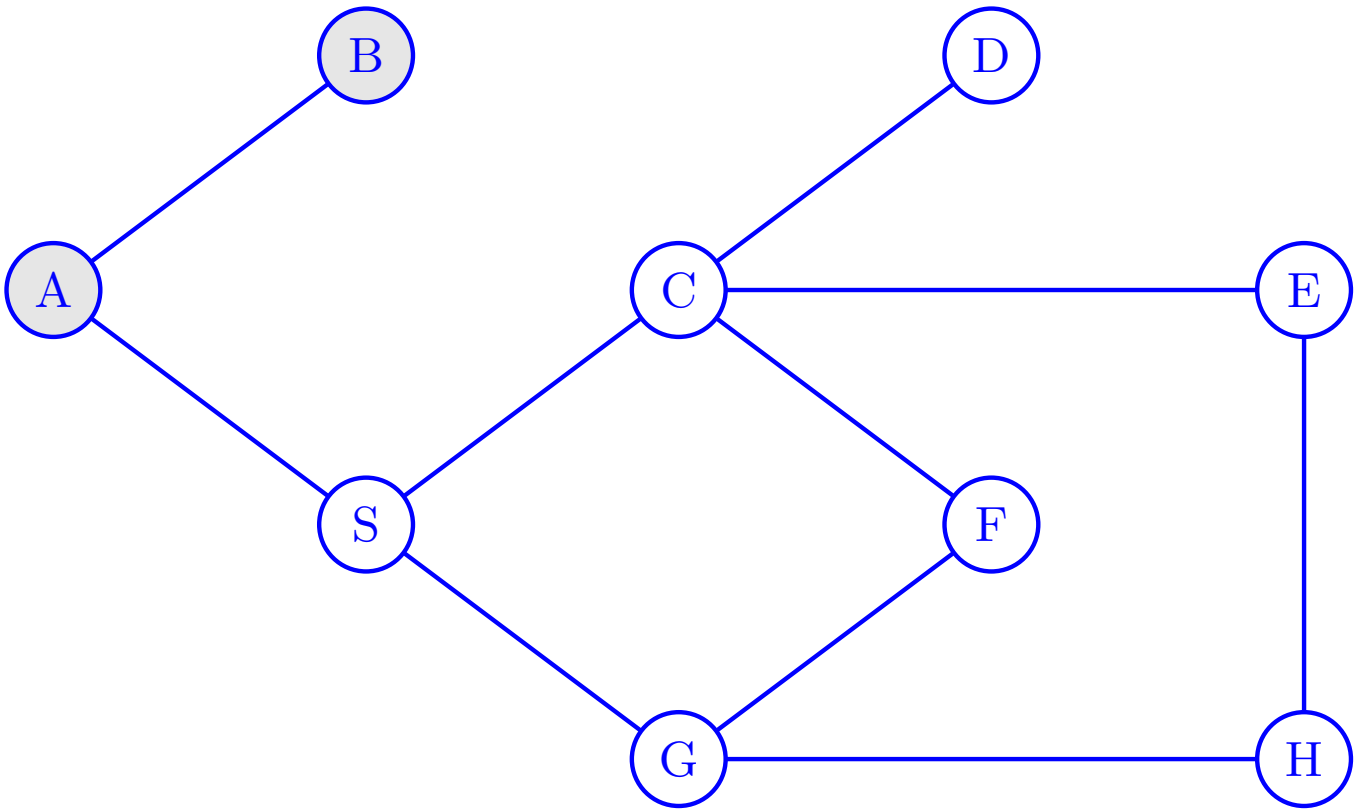
Example 2: BFS (start at A; alphabetical ordering of next nodes) 1



s:	A	B	C	D	E	F	G	H	S
$d[s]$:	0								
$pred[s]$:	–								

Queue Q: A

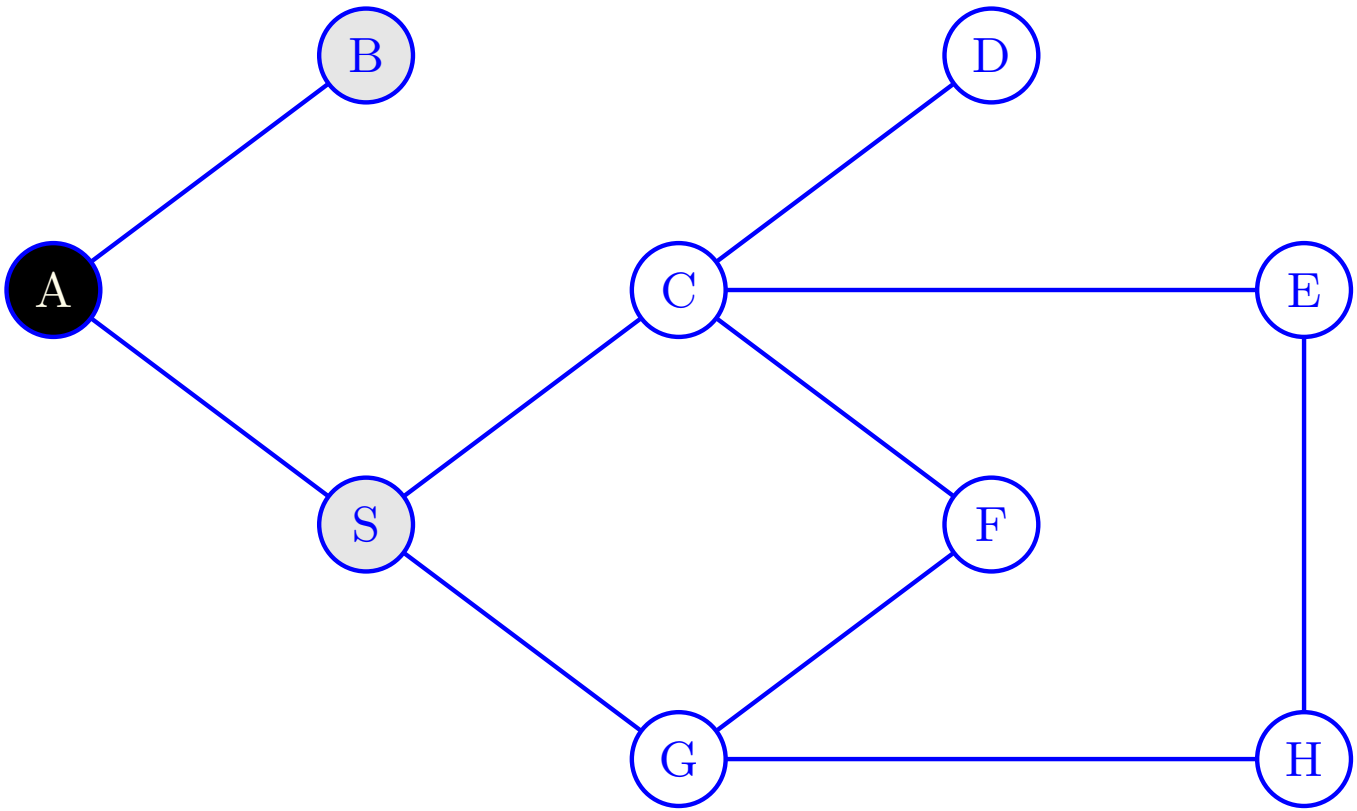
Example 2: BFS (start at A; alphabetical ordering of next nodes) 2



s:	A	B	C	D	E	F	G	H	S
$d[s]$:	0	1							
$pred[s]$:	–	A							

Queue Q: AB

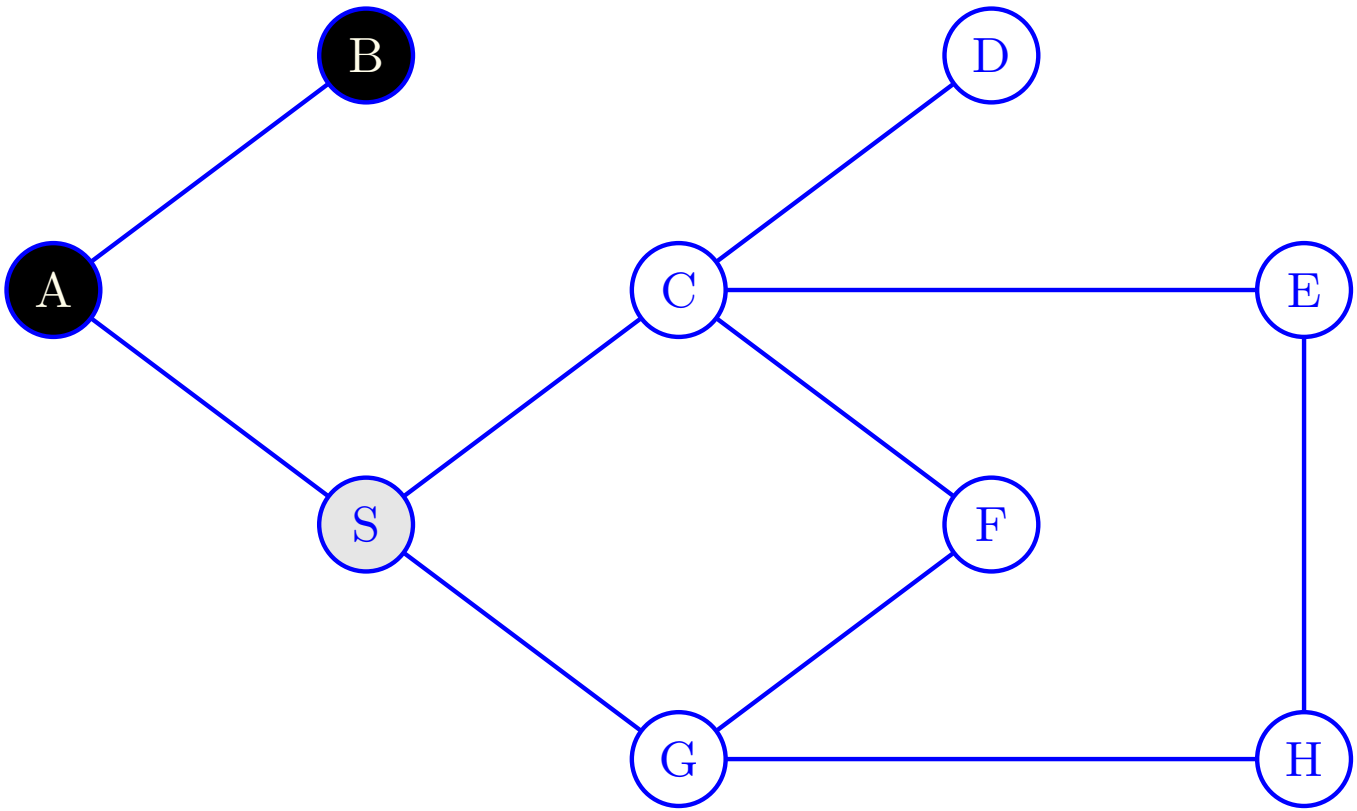
Example 2: BFS (start at A; alphabetical ordering of next nodes) 3



s:	A	B	C	D	E	F	G	H	S
$d[s]$:	0	1							1
$pred[s]$:	–	A							A

Queue Q: BS

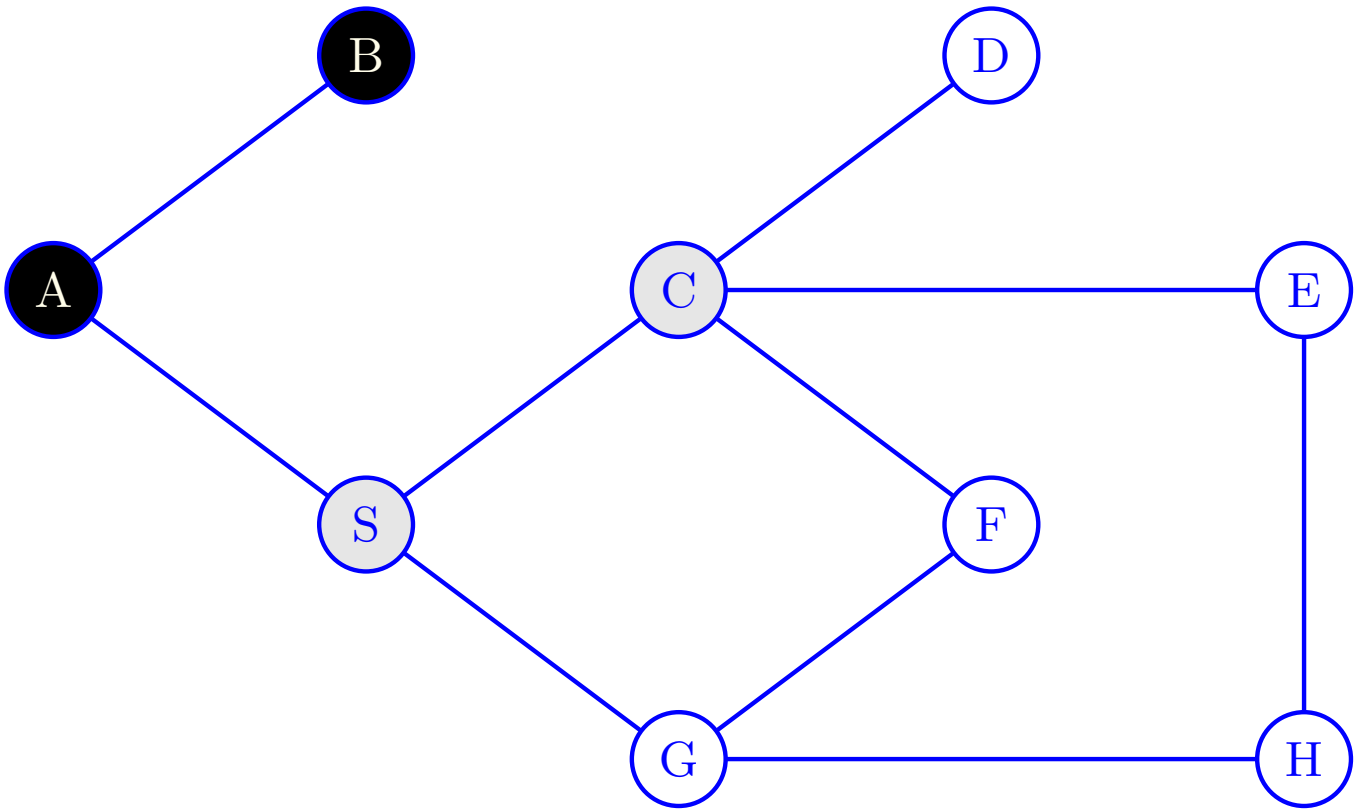
Example 2: BFS (start at A; alphabetical ordering of next nodes) 4



s:	A	B	C	D	E	F	G	H	S
$d[s]$:	0	1							1
$pred[s]$:	–	A							A

Queue Q: S

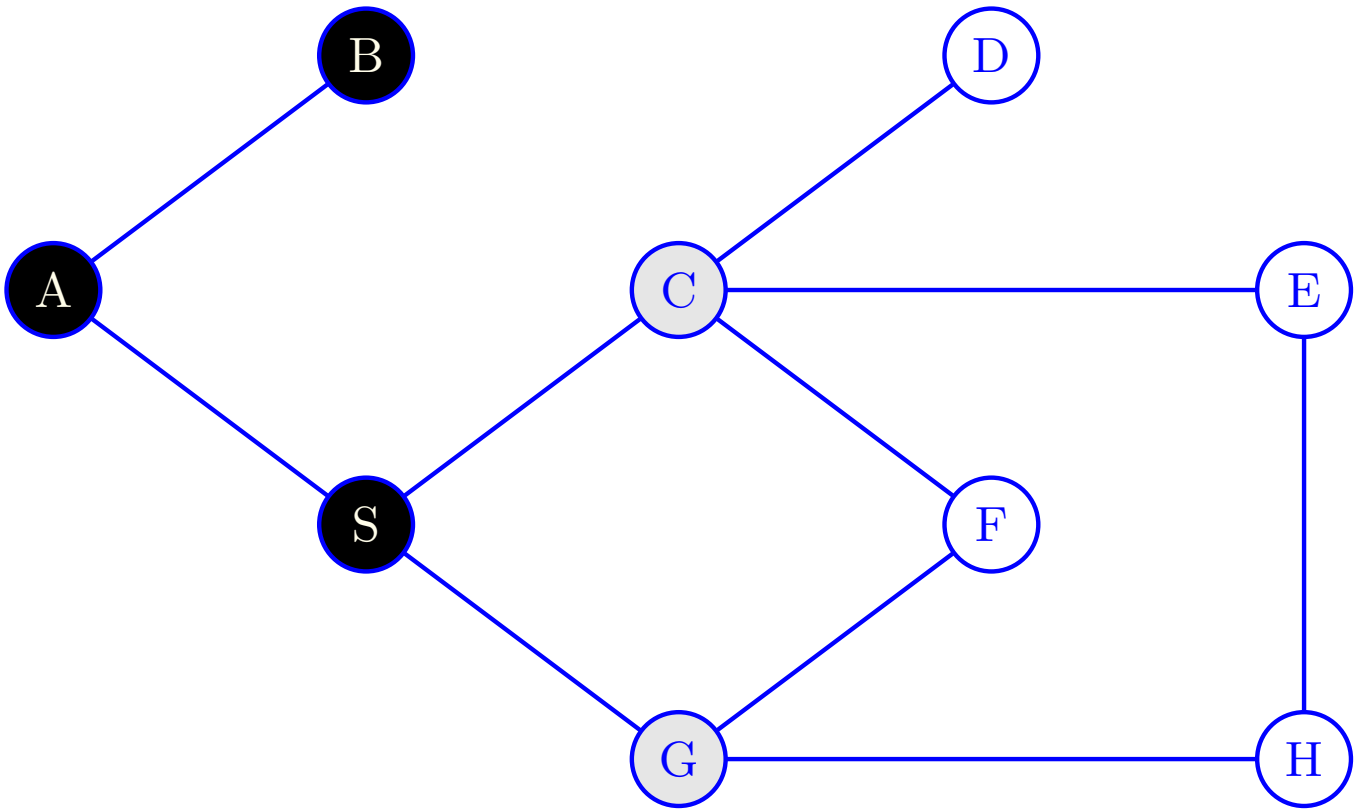
Example 2: BFS (start at A; alphabetical ordering of next nodes) 5



s:	A	B	C	D	E	F	G	H	S
$d[s]$:	0	1	2						1
$pred[s]$:	–	A	S						A

Queue Q: SC

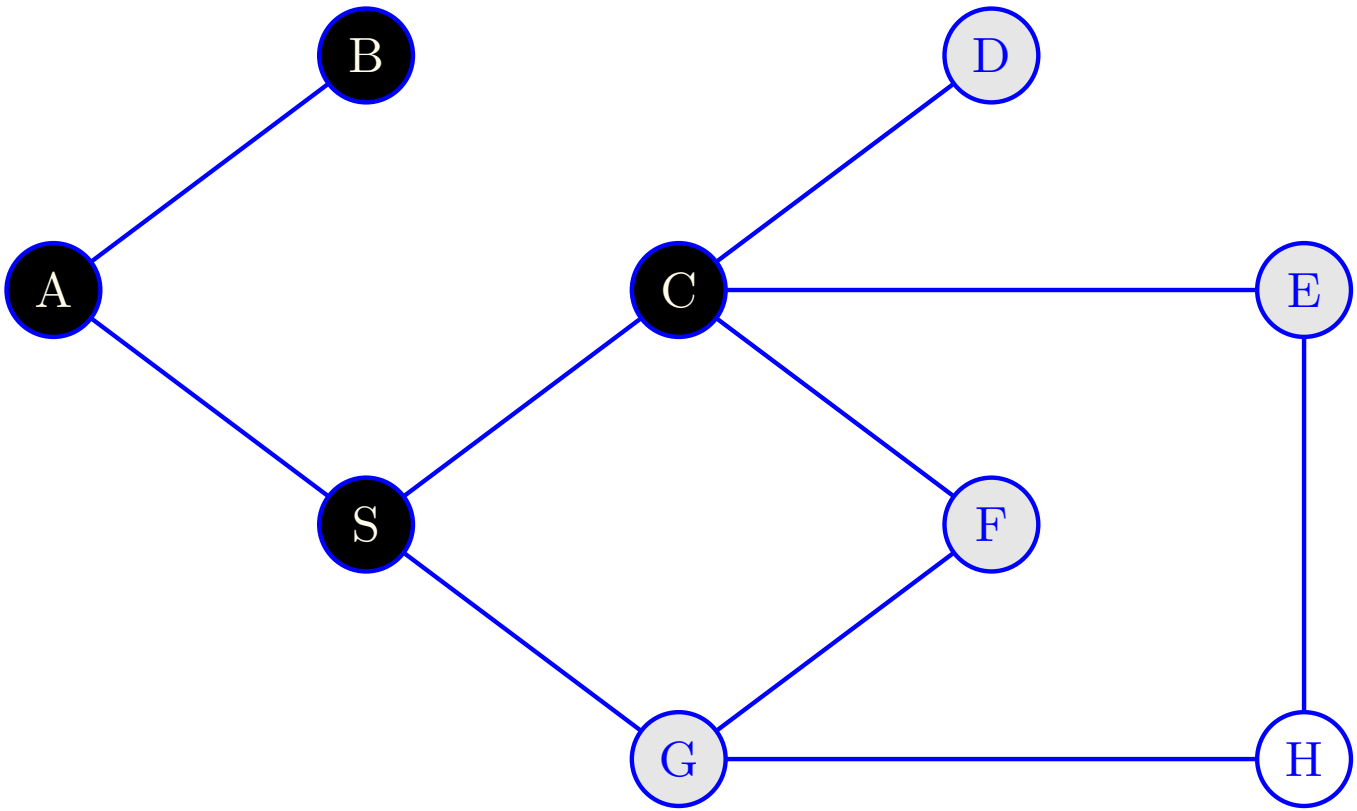
Example 2: BFS (start at A; alphabetical ordering of next nodes) 6



s:	A	B	C	D	E	F	G	H	S
$d[s]$:	0	1	2				2		1
$pred[s]$:	–	A	S				S		A

Queue Q: CG

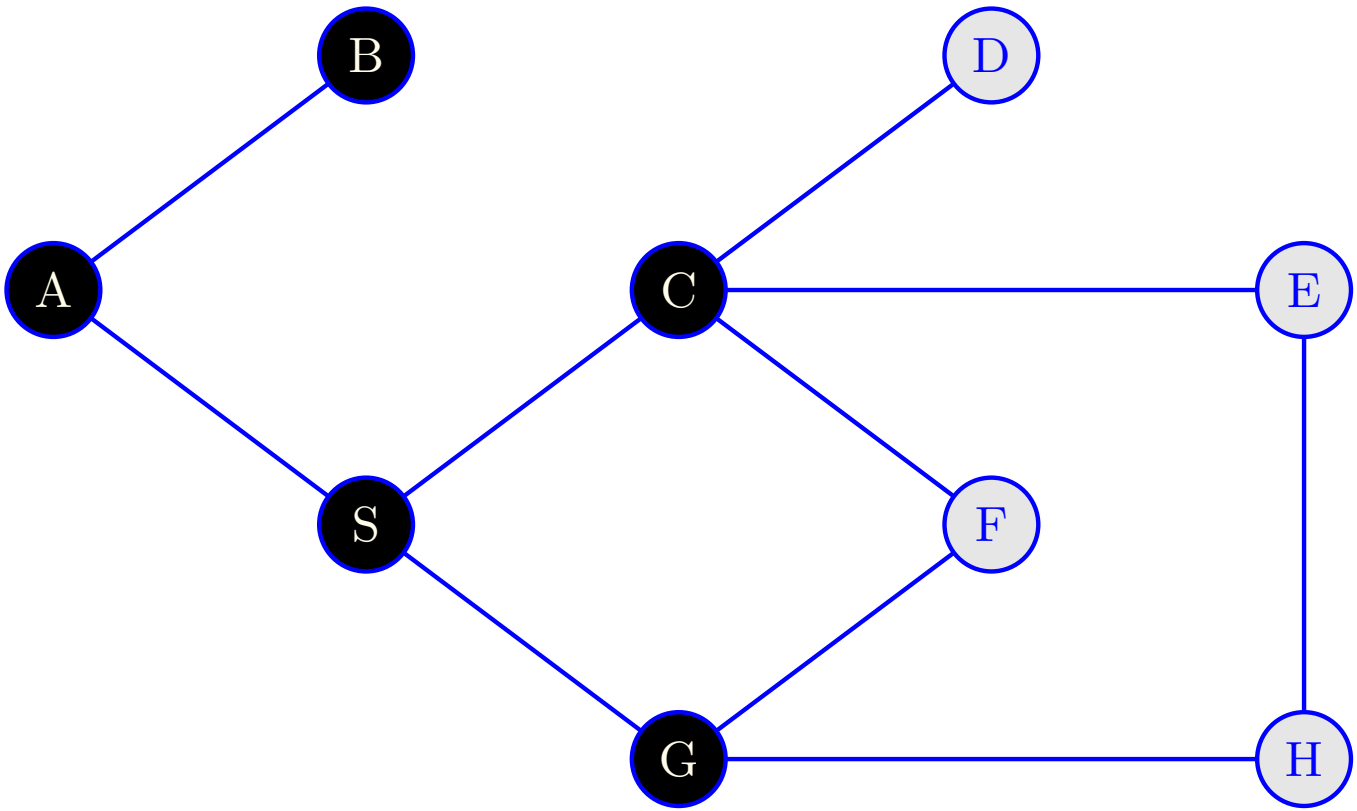
Example 2: BFS (start at A; alphabetical ordering of next nodes) 7-9



s:	A	B	C	D	E	F	G	H	S
$d[s]$:	0	1	2	3	3	3	2		1
$pred[s]$:	–	A	S	C	C	C	S		A

Queue Q: GDEF

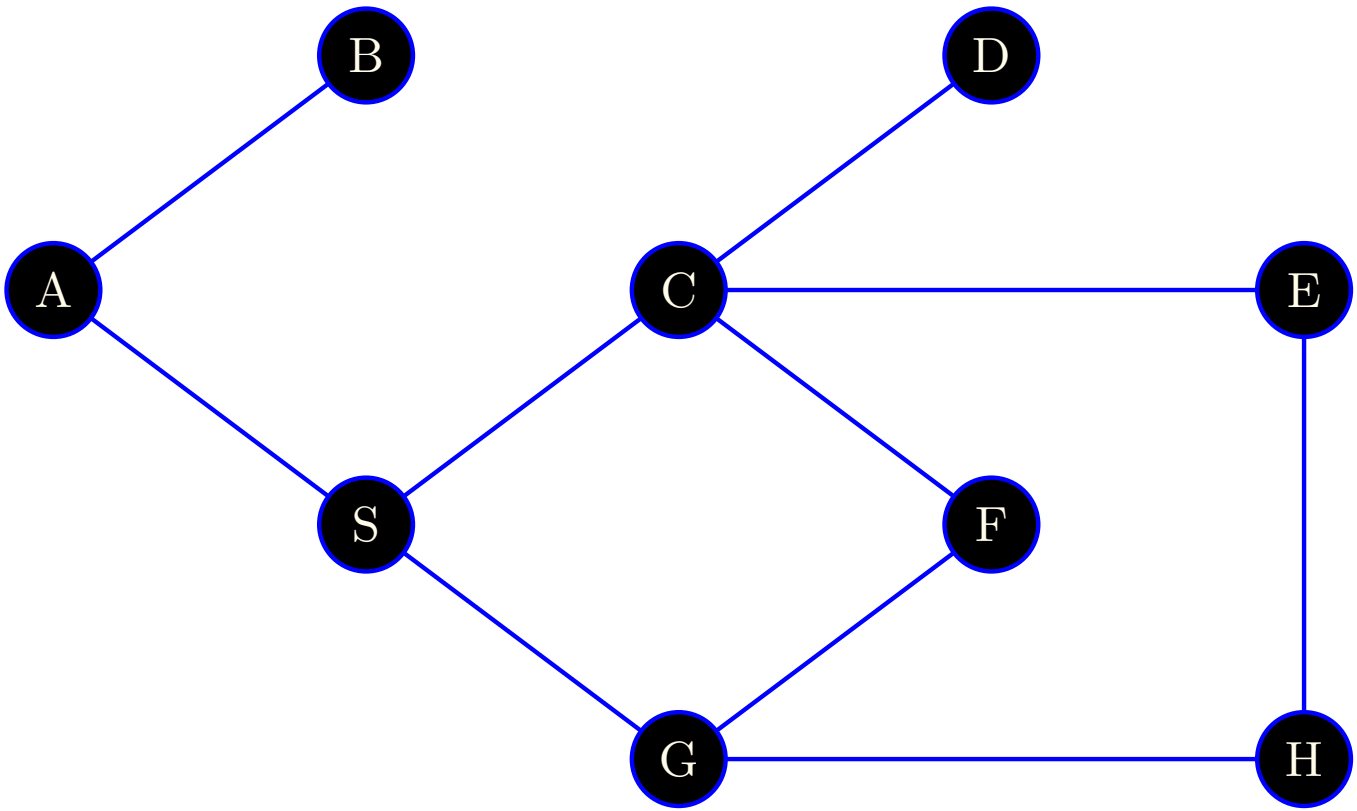
Example 2: BFS (start at A; alphabetical ordering of next nodes) 10



s:	A	B	C	D	E	F	G	H	S
$d[s]$:	0	1	2	3	3	3	2	3	1
$pred[s]$:	–	A	S	C	C	C	S	G	A

Queue Q: DEFH

Example 2: BFS (start at A; alphabetical ordering of next nodes)11-14



s:	A	B	C	D	E	F	G	H	S
$d[s]$:	0	1	2	3	3	3	2	3	1
$pred[s]$:	–	A	S	C	C	C	S	G	A

Queue Q:

Priority-first Search (PFS)

A common generalisation of BFS and DFS:

- Each GREY node is associated with an integer **key**.
- The smaller the key, the higher the priority.
- The rule for selecting a new GREY node: one with the maximum priority (minimum key).
- The keys are fixed in the simplest case, but generally they may be updated.

BFS as PFS: the key of $v \in V \leftarrow$ the time it was first coloured GREY.
DFS as PFS: the key of $v \in V \leftarrow -seen[v]$

PFS is best described via the priority queue ADT

- E.g., using a binary heap.

Priority-first Search (PFS) Algorithm (Part 1)

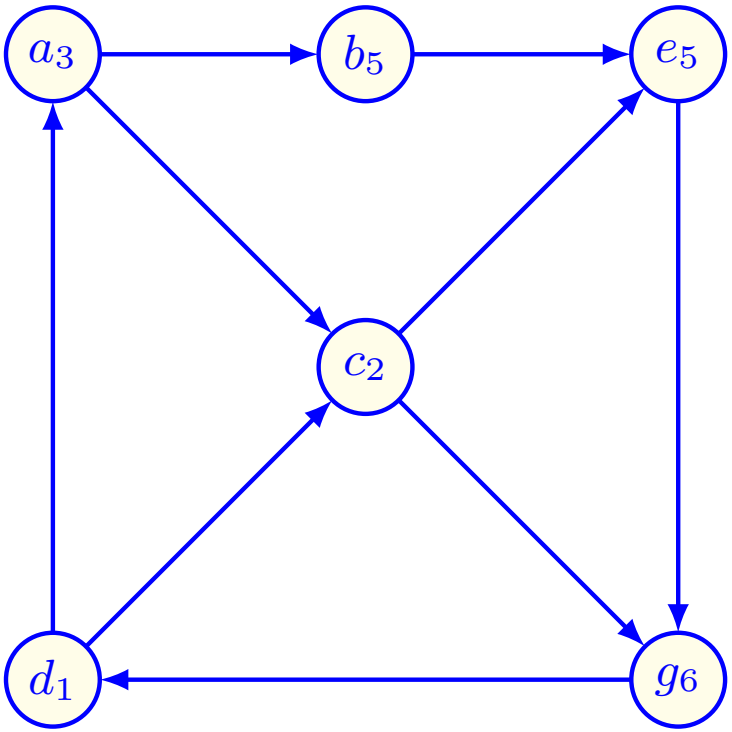
algorithm `pfs`
    Input: digraph $G = (V(G), E(G))$
begin
    priority queue Q;
    array $colour[n], pred[n]$
    for $u \in V(G)$ **do**
        $colour[u] \leftarrow$ WHITE; $pred[u] \leftarrow$ NULL
    end for
    for $s \in V(G)$ **do**
        if $colour[s] =$ WHITE **then**
            `pfsvisit`(s)
        end if
    end for
    return $pred$
end

Priority-first Search (PFS) Algorithm (Part 2)

algorithm `pfsvisit`
  Input: node s
begin
  $colour[s] \leftarrow$ GREY; Q.`insert`(s, `setkey` (s))
  while not Q.`is_empty`() **do**
    $u \leftarrow$ Q.`get_min`()
    if v adjacent to u and $colour[v] =$ WHITE **then**
      $colour[v] \leftarrow$ GREY; $pred[v] \leftarrow u$;
      Q.`insert`(v, `setkey` (v))
    else
      Q.`del_min`(); $colour[u] \leftarrow$ BLACK
    end if
  end while
end

`setkey`(s) – the rule of assigning a priority key to the node s.

PFS: Example: Initialisation

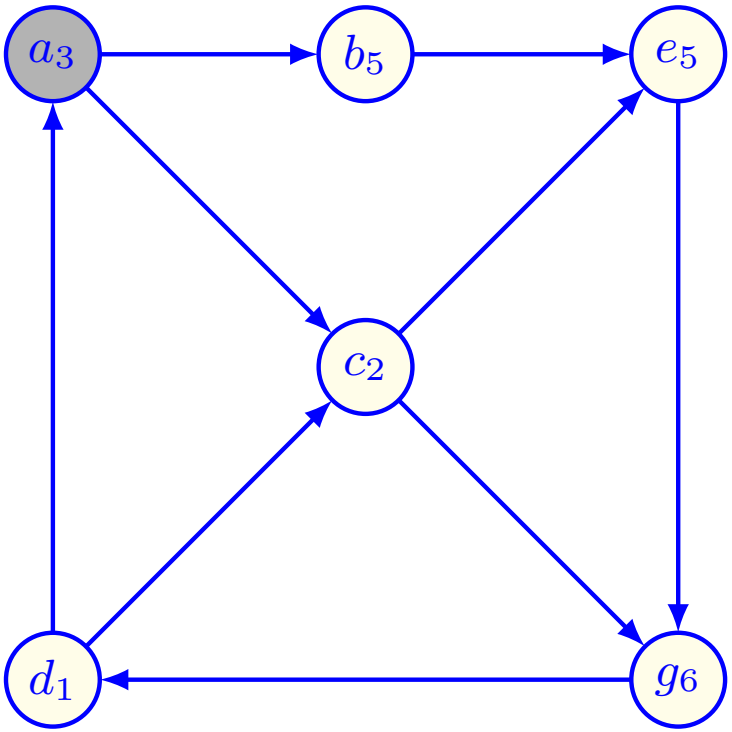


$v \in V$	a	b	c	d	e	g
key_v	3	5	2	1	5	6
$pred[v]$	–	–	–	–	–	–

Priority queue $Q = \{\ \}$

PFS: Example:

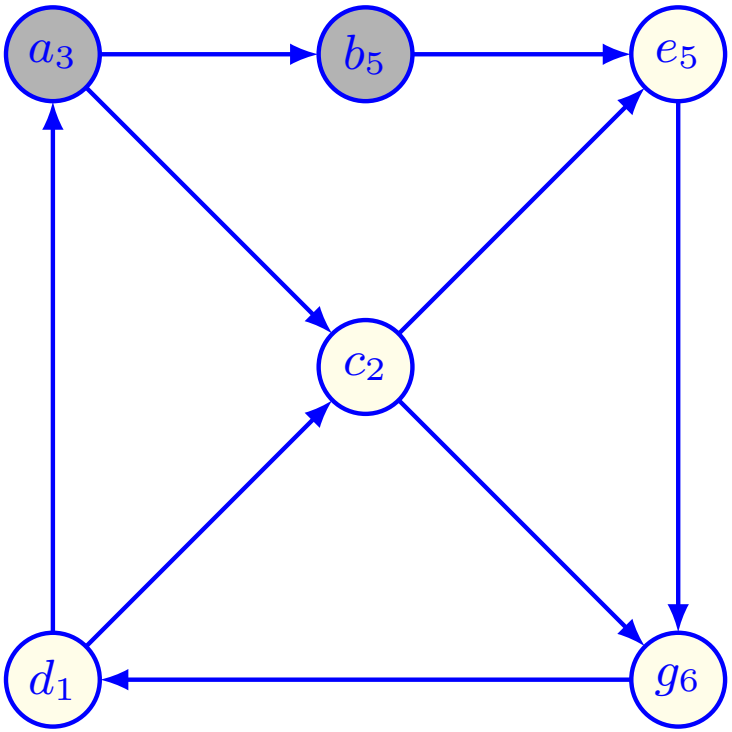
1: $s \leftarrow a$



$v \in V$	a	b	c	d	e	g
key_v	3	5	2	1	5	6
$pred[v]$	–	–	–	–	–	–

Priority queue $Q = \{a_3\}$

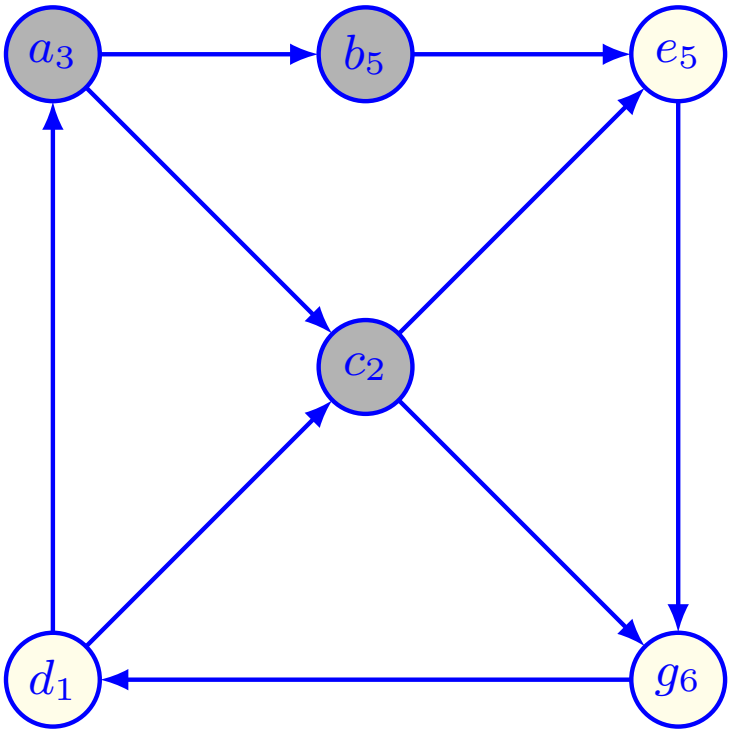
PFS: Example: 2: $u \leftarrow Q.\texttt{get\_min}() = a$; $v \leftarrow b$; $Q.\texttt{insert}(b, 5)$



$v \in V$	a	b	c	d	e	g
key_v	3	5	2	1	5	6
$pred[v]$	–	a	–	–	–	–

Priority queue $Q = \{a_3, b_5\}$

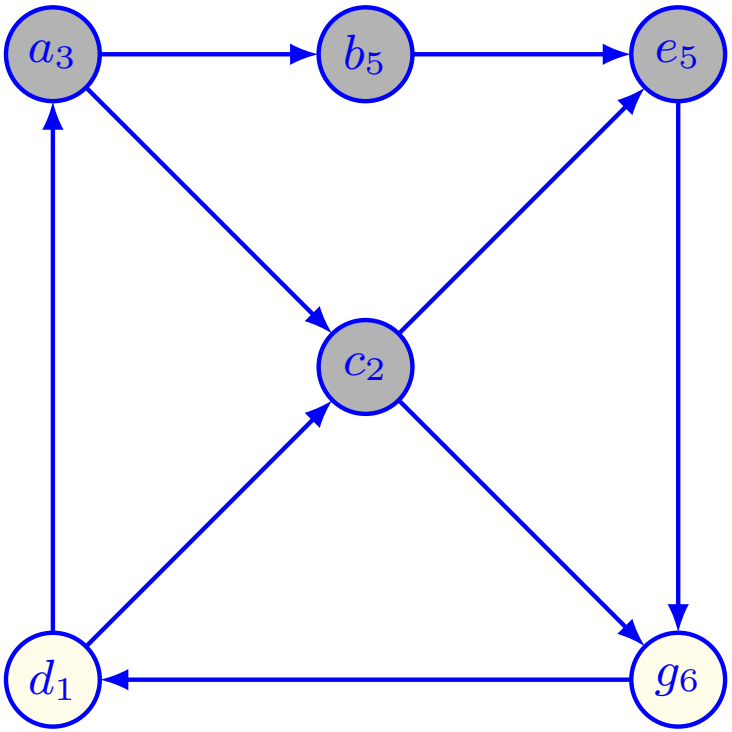
PFS: Example: 3: $u \leftarrow Q.\texttt{get\_min}() = a$; $v \leftarrow c$; $Q.\texttt{insert}(c, 2)$



$v \in V$	a	b	c	d	e	g
key_v	3	5	2	1	5	6
$pred[v]$	–	a	a	–	–	–

Priority queue $Q = \{a_3, b_5, c_2\}$

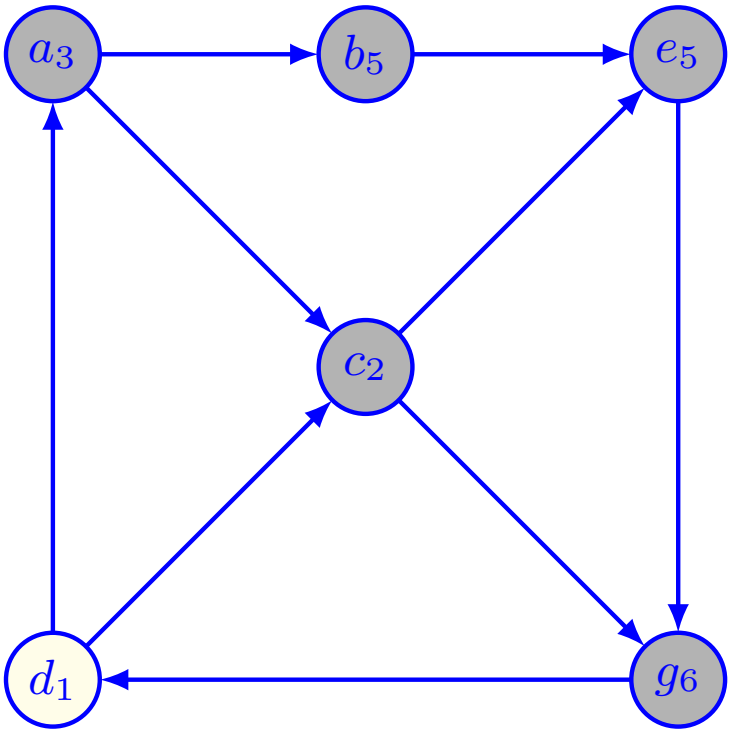
PFS: Example: 4: $u \leftarrow Q.\texttt{get\_min}() = c$; $v \leftarrow e$; $Q.\texttt{insert}(e, 5)$



$v \in V$	a	b	c	d	e	g
key_v	3	5	2	1	5	6
$pred[v]$	$-$	a	$-a$	$-$	c	$-$

Priority queue $Q = \{a_3, b_5, c_2, e_5\}$

PFS: Example: 5: $u \leftarrow Q.\texttt{get\_min}() = c$; $v \leftarrow g$; $Q.\texttt{insert}(g, 6)$

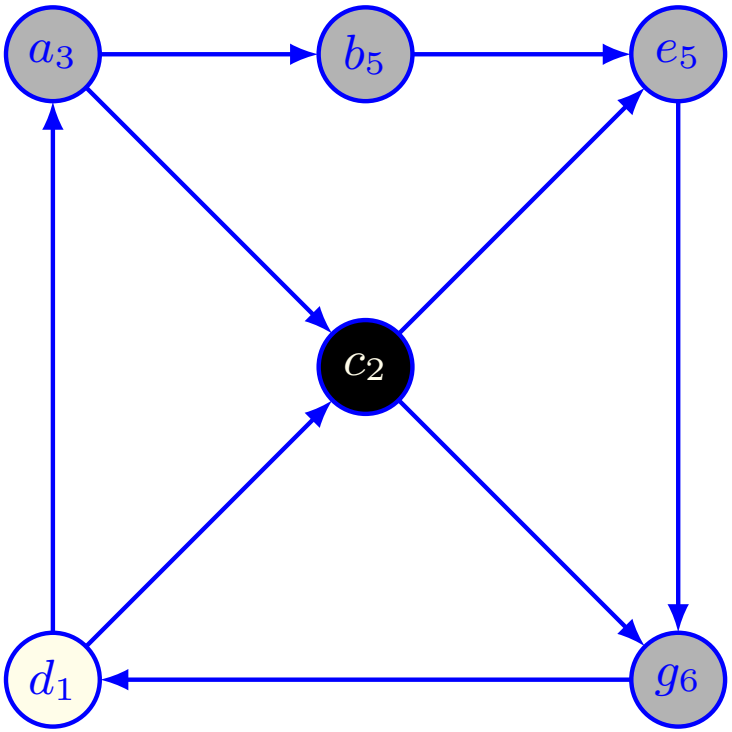


$v \in V$	a	b	c	d	e	g
key_v	3	5	2	1	5	6
$pred[v]$	–	a	a	–	c	c

Priority queue $Q = \{a_3, b_5, c_2, e_5, g_6\}$

PFS: Example:

6: $u \leftarrow Q.\texttt{get\_min}() = c$; $Q.\texttt{del\_min}()$

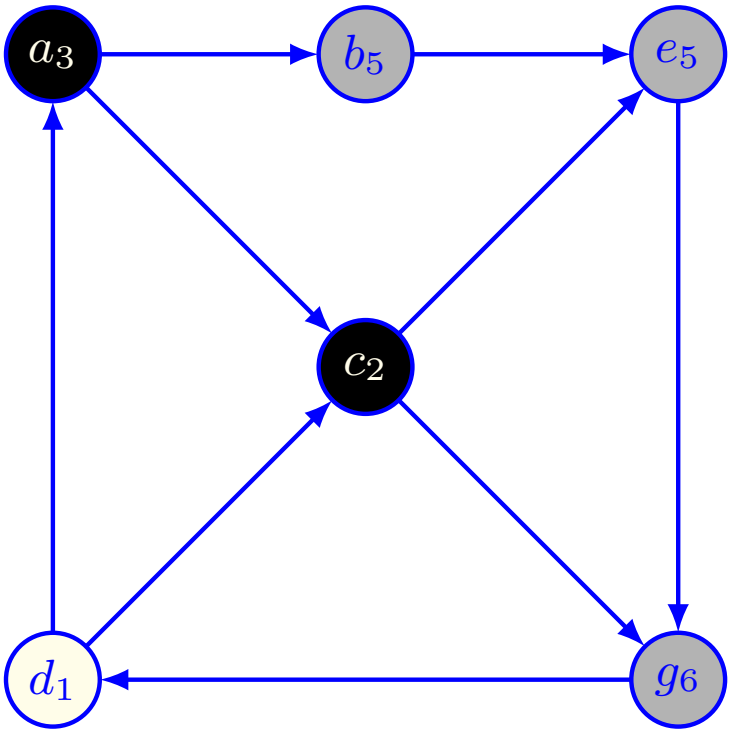


$v \in V$	a	b	c	d	e	g
key_v	3	5	2	1	5	6
$pred[v]$	$-$	a	a	$-$	c	c

Priority queue $Q = \{a_3, b_5, e_5, g_6\}$

PFS: Example:

7: $u \leftarrow Q.\texttt{get\_min}() = a$; $Q.\texttt{del\_min}()$

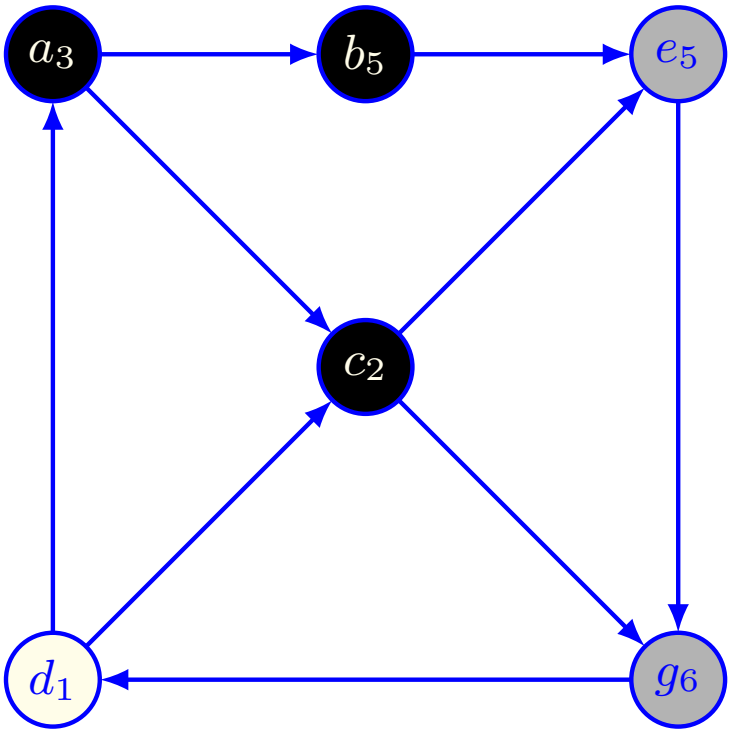


$v \in V$	a	b	c	d	e	g
key_v	3	5	2	1	5	6
$pred[v]$	$-$	a	a	$-$	c	c

Priority queue $Q = \{b_5, e_5, g_6\}$

PFS: Example:

8: $u \leftarrow Q.\texttt{get\_min}() = b$; $Q.\texttt{del\_min}()$

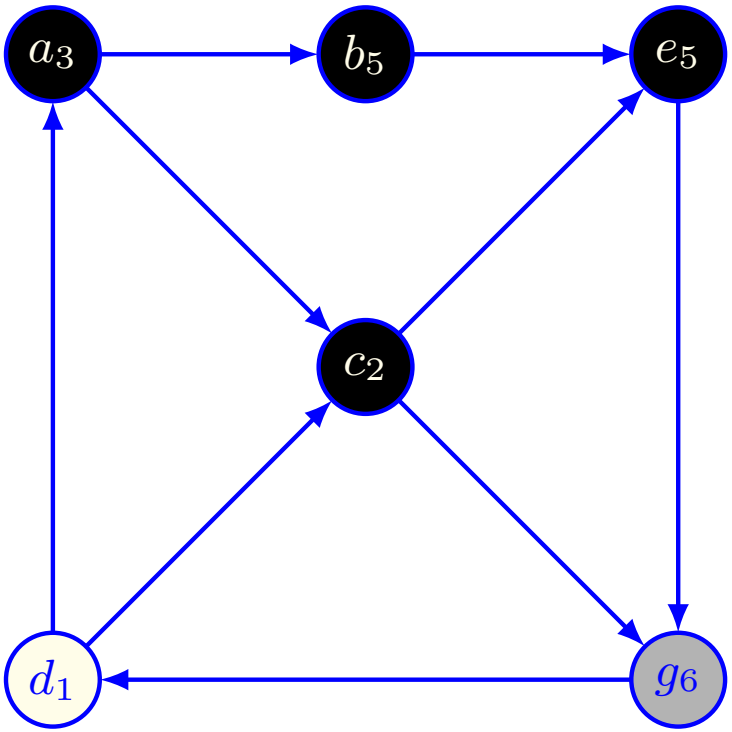


$v \in V$	a	b	c	d	e	g
key_v	3	5	2	1	5	6
$pred[v]$	$-$	a	a	$-$	c	c

Priority queue $Q = \{e_5, g_6\}$

PFS: Example:

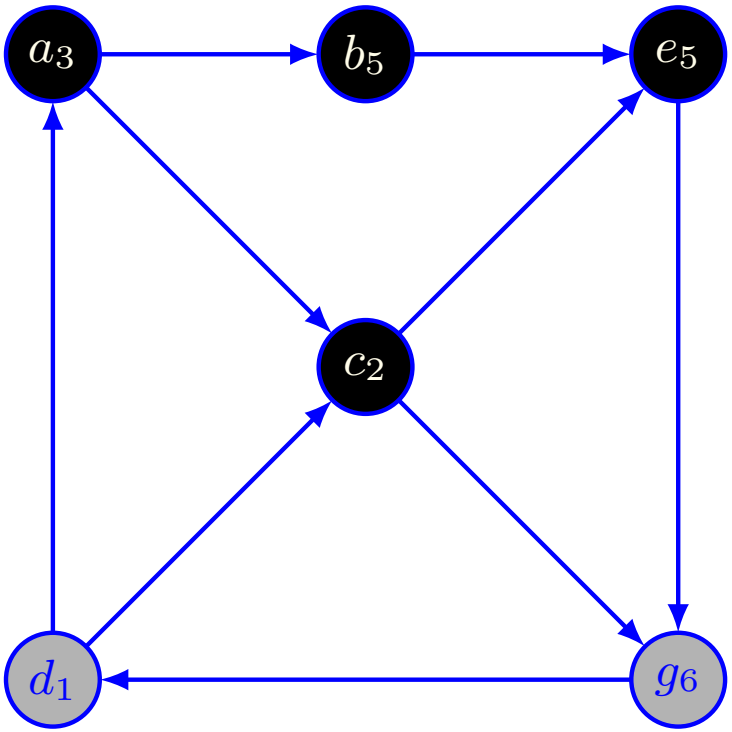
9: $u \leftarrow Q.\texttt{get\_min}() = e$; $Q.\texttt{del\_min}()$



$v \in V$	a	b	c	d	e	g
key_v	3	5	2	1	5	6
$pred[v]$	$-$	a	a	$-$	c	c

Priority queue $Q = \{g_6\}$

PFS: Example: 10: $u \leftarrow Q.\texttt{get\_min}() = g$; $v \leftarrow d$; $Q.\texttt{insert}(d, 1)$

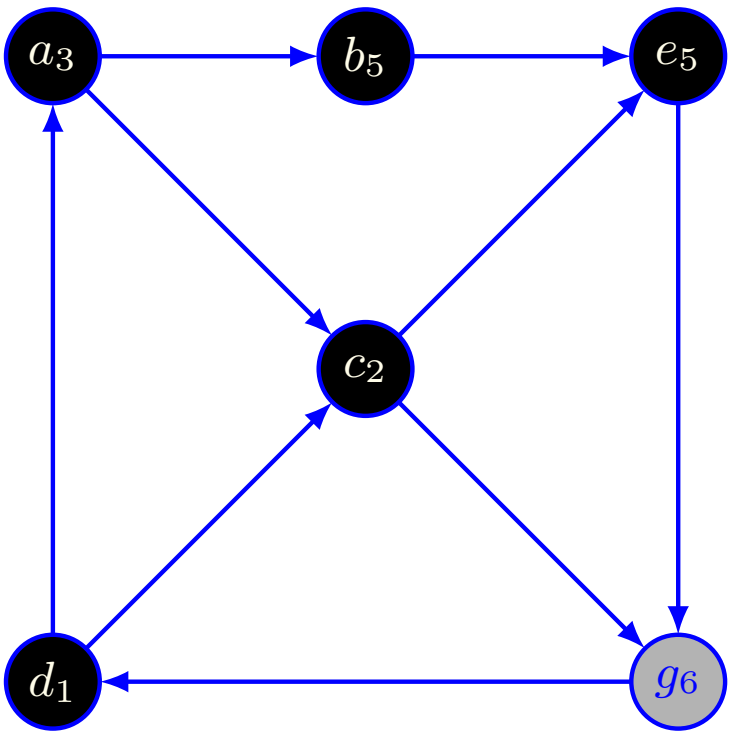


$v \in V$	a	b	c	d	e	g
key_v	3	5	2	1	5	6
$pred[v]$	$-$	a	a	g	c	c

Priority queue $Q = \{g_6, d_1\}$

PFS: Example:

11: $u \leftarrow Q.\texttt{get\_min}() = d$; $Q.\texttt{del\_min}()$

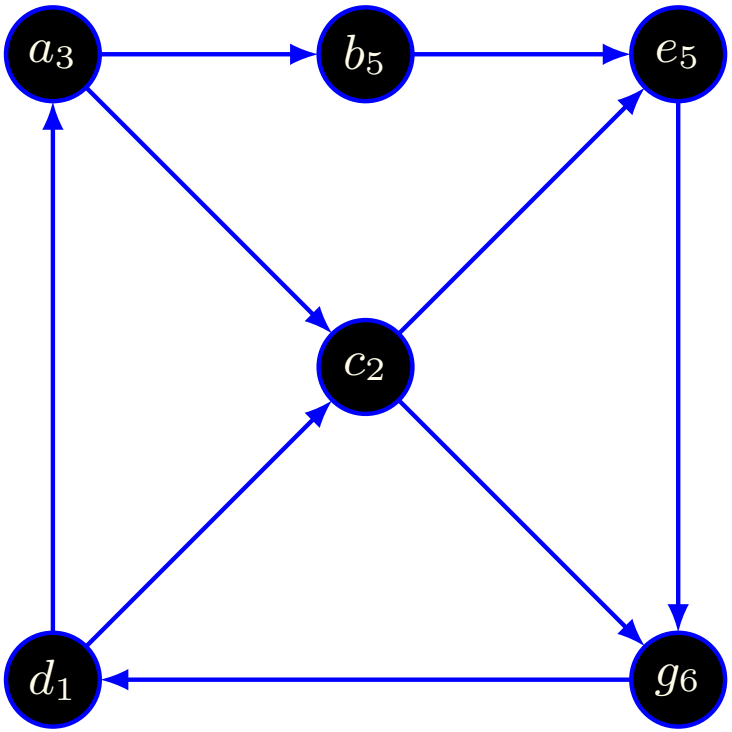


$v \in V$	a	b	c	d	e	g
key_v	3	5	2	1	5	6
$pred[v]$	$-$	a	a	g	c	c

Priority queue $Q = \{g_6\}$

PFS: Example:

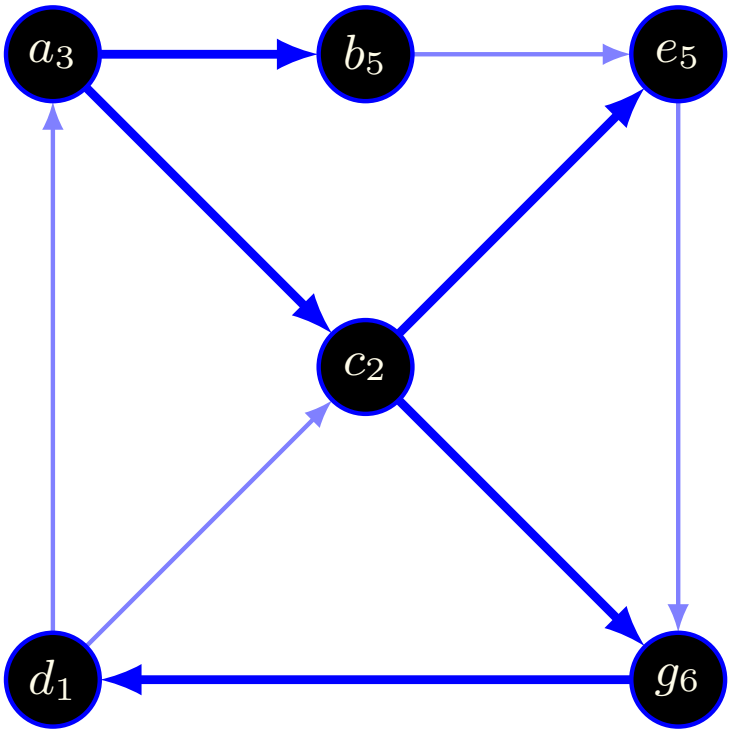
12: $u \leftarrow Q.\texttt{get\_min}() = g$; $Q.\texttt{del\_min}()$



$v \in V$	a	b	c	d	e	g
key_v	3	5	2	1	5	6
$pred[v]$	$-$	a	a	g	c	c

Priority queue $Q = \{\,\}$

PFS: Example: Output

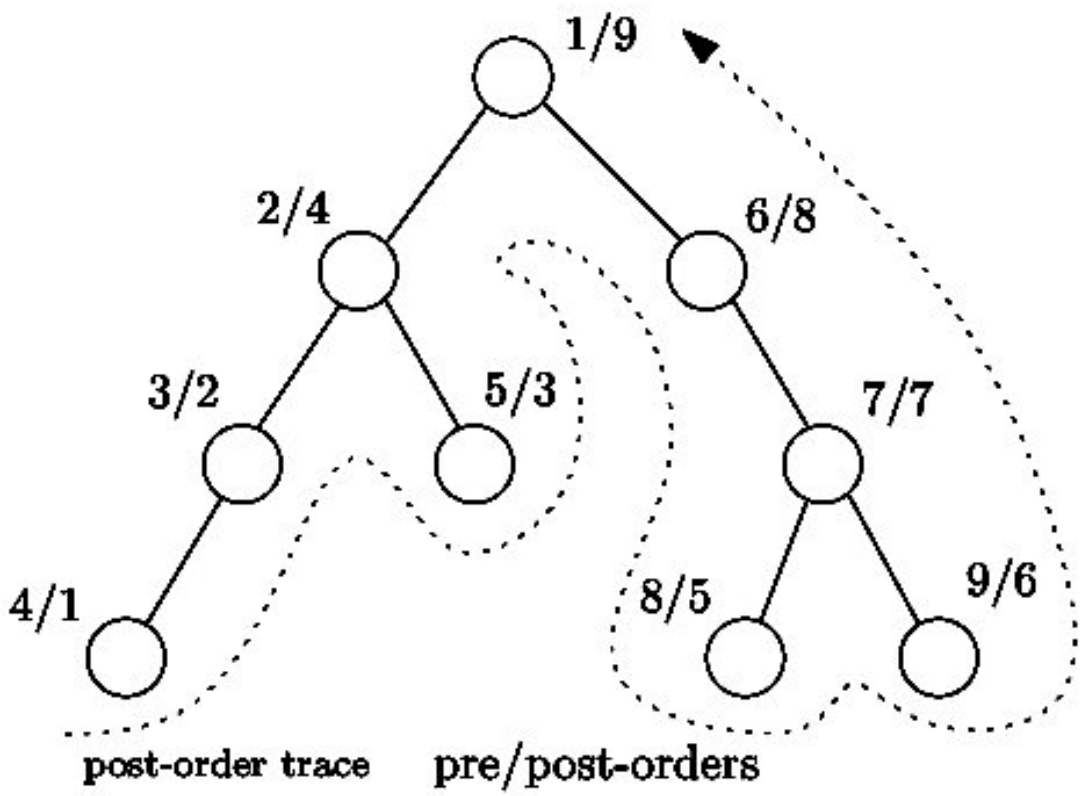


$v \in V$	a	b	c	d	e	g
key_v	3	5	2	1	5	6
$pred[v]$	$-$	a	a	g	c	c

Pre-order and Post-order Labelings

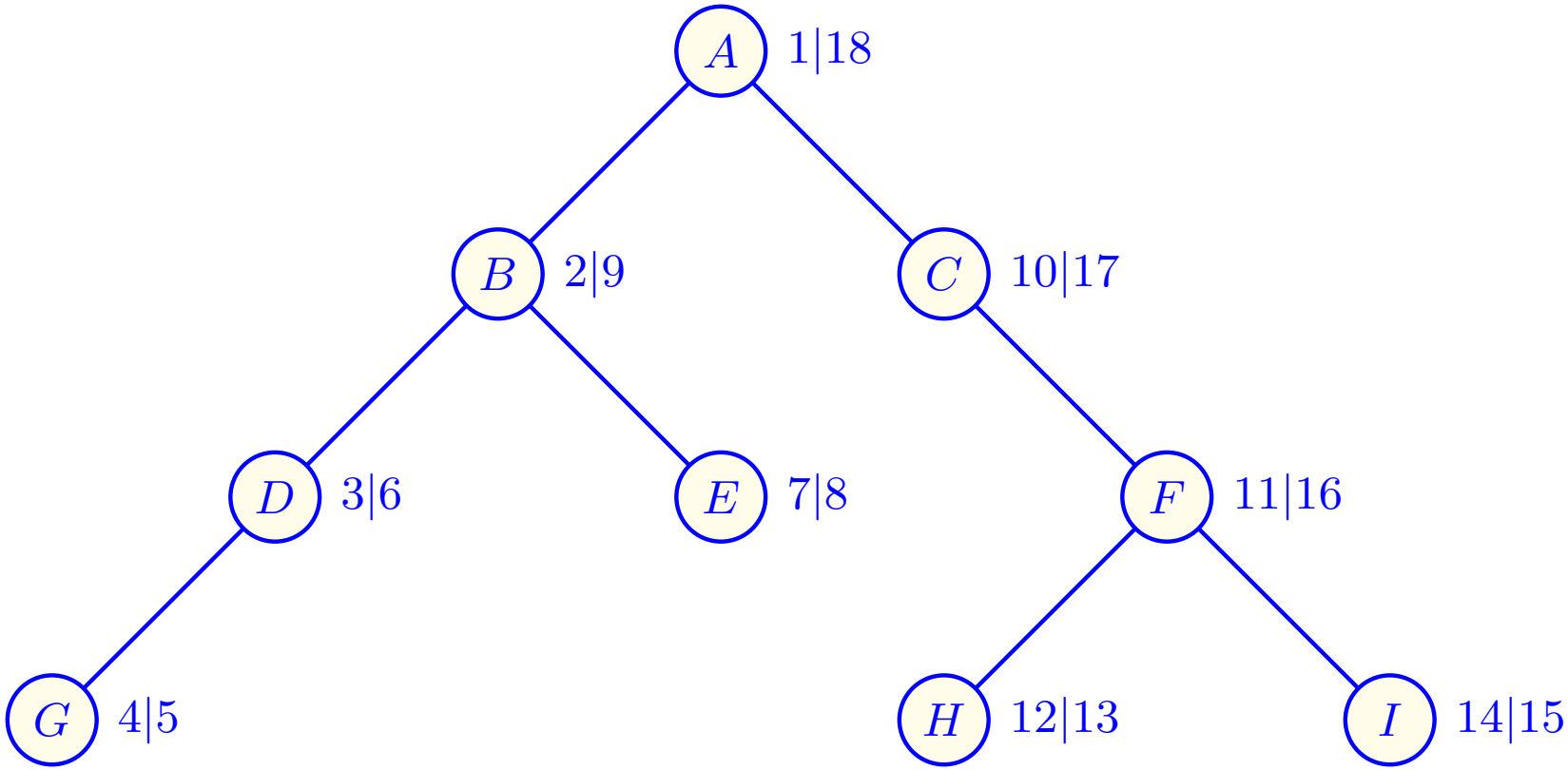
DFS: gives pre-order and post-order labellings to a digraph:

- Pre-order label – the order in which the nodes were turned GREY.
- Post-order label – the order in which the nodes were turned BLACK.



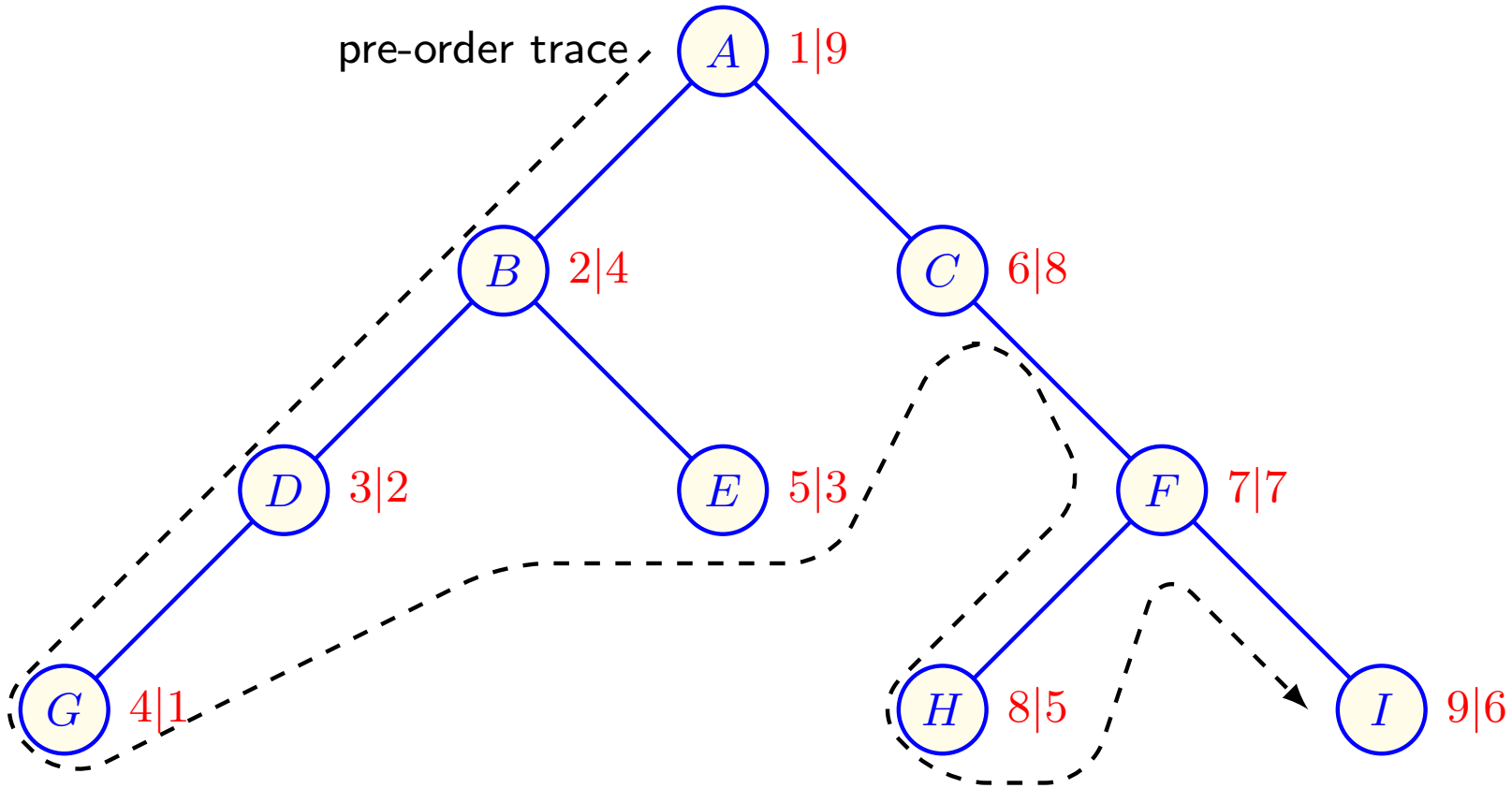
Pre-order: visit a current node, then traverse its subtrees.
Post-order: traverse subtrees, then visit their parent node.

Pre-order and Post-order Labelings



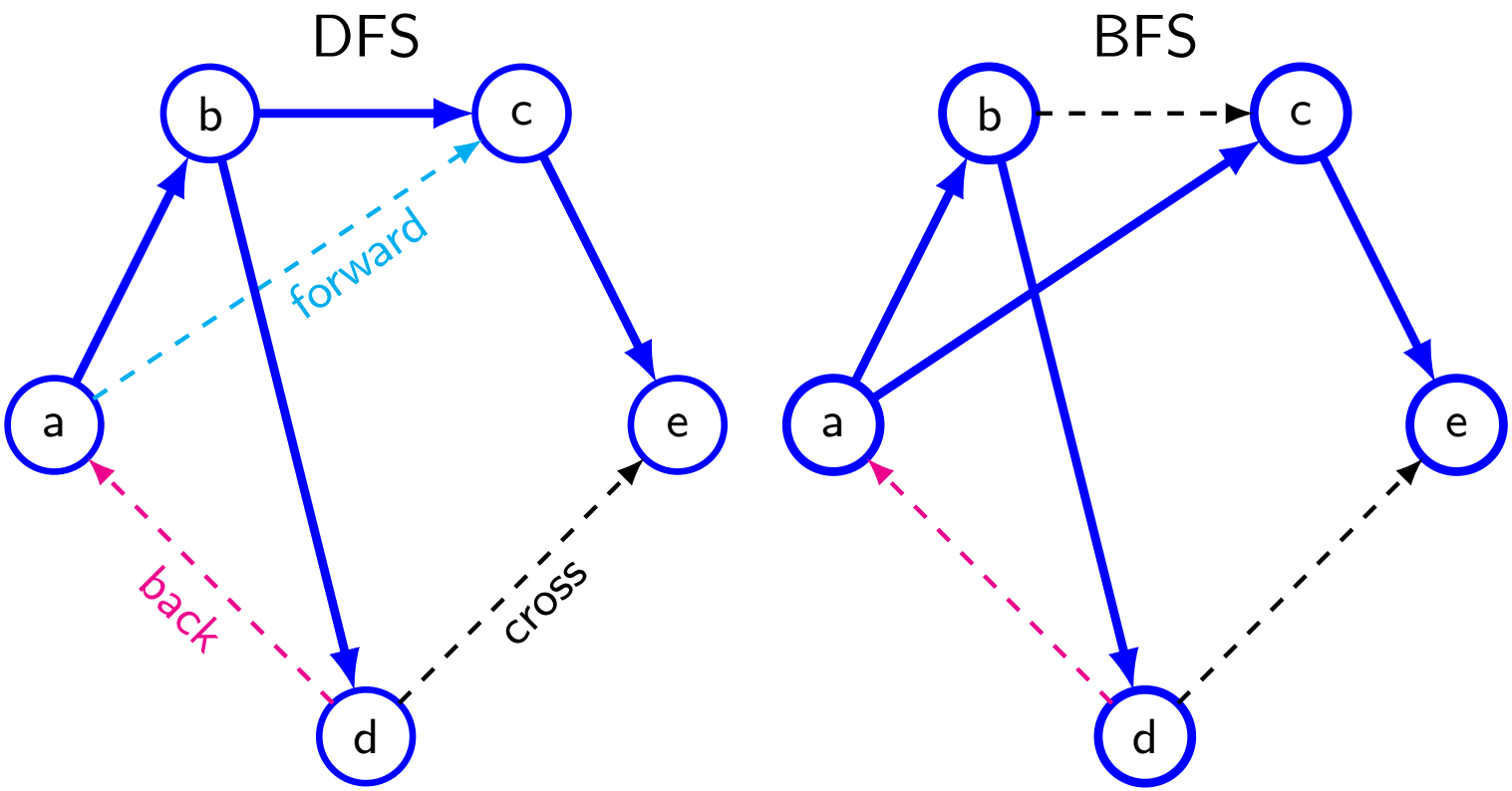
v	A	B	C	D	E	F	G	H	I
$seen[v]$/pre-	1/1	2/2	10/6	3/3	7/5	11/7	4/4	12/8	14/9
$done[v]$/post-	18/9	9/4	17/8	6/2	8/3	16/7	5/1	13/5	15/6

Pre-order and Post-order Labelings



v	A	B	C	D	E	F	G	H	I
$seen[v]$/pre-	1/1	2/2	10/6	3/3	7/5	11/7	4/4	12/8	14/9
$done[v]$/post-	18/9	9/4	17/8	6/2	8/3	16/7	5/1	13/5	15/6

Cycle Detection



Cycle: a – b – d – a

Cycle Detection

Suppose that there is a cycle in G and let v be the node in the cycle visited first by DFS.

- If (u, v) is an arc in the cycle then it must be a back arc.
- Conversely if there is a back arc, we must have a cycle.
- So a digraph is acyclic iff there are no back arcs from DFS.

An acyclic digraph is called a directed acyclic graph (DAG).

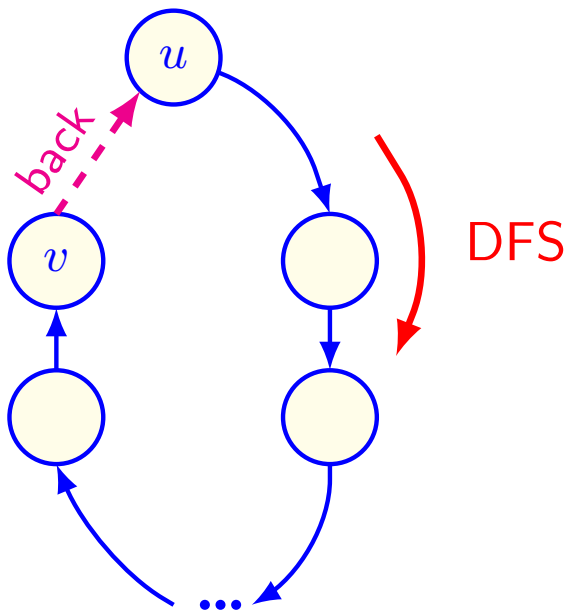
- An acyclic graph is a tree or a forest.

Cycles can also be easily detected in a graph using BFS.

- Finding a cycle of minimum length in a graph is not difficult using BFS (better than DFS).

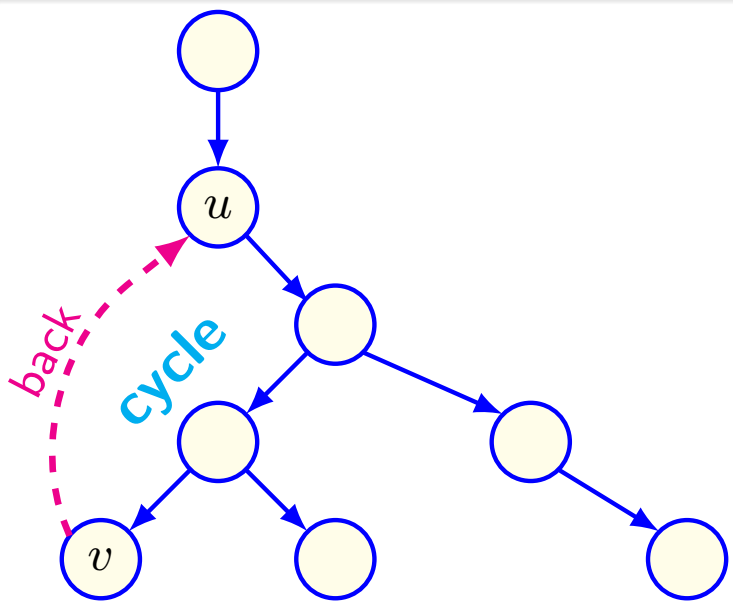
Back Arcs in Cycle Detection

Search forest

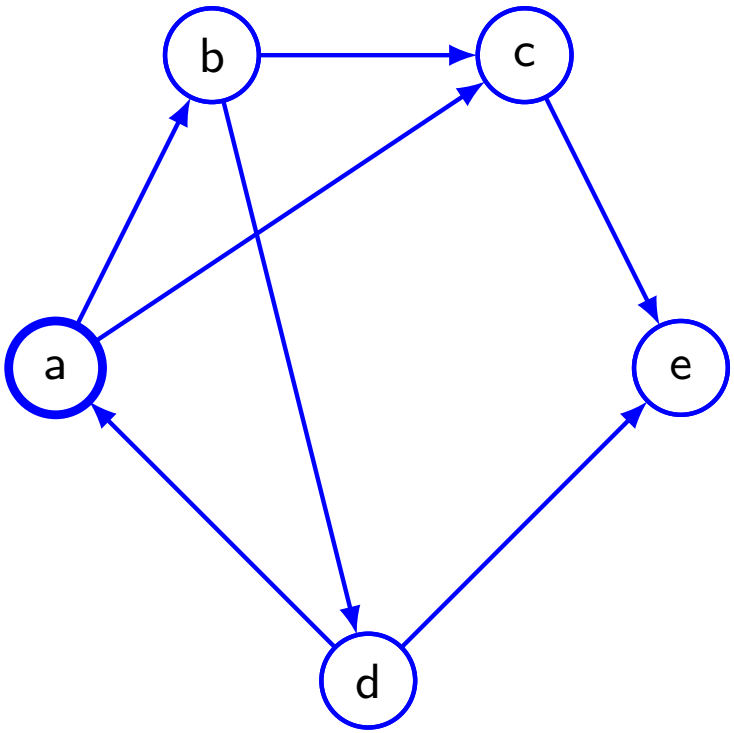


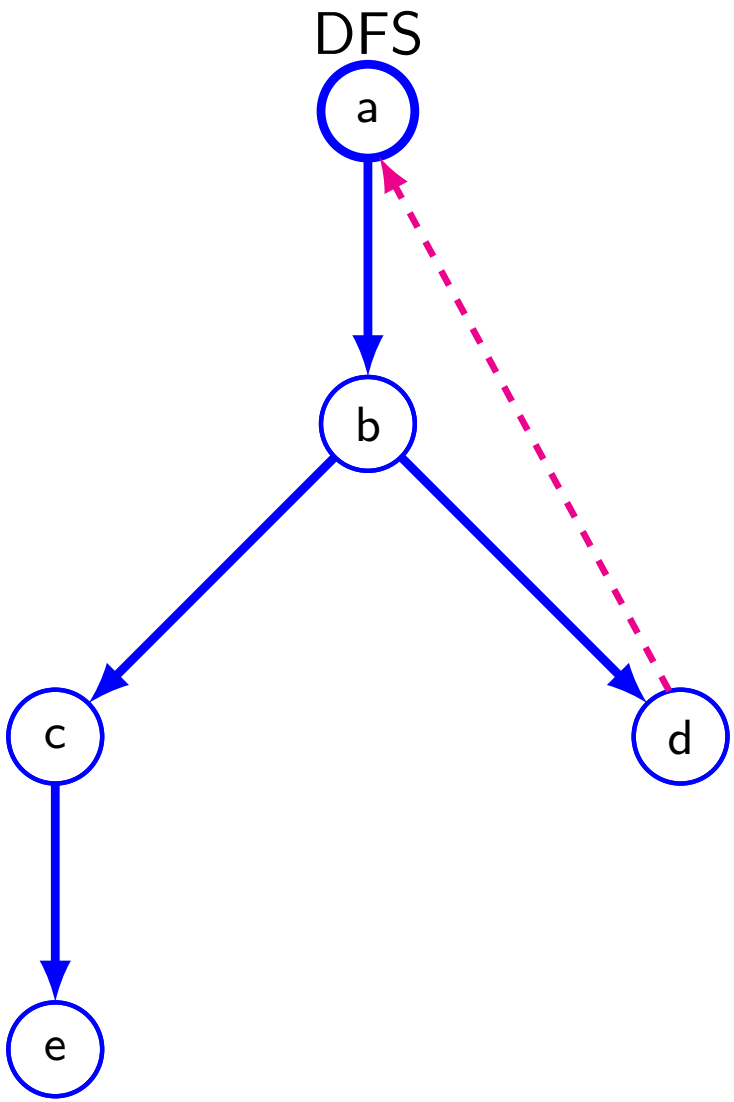
Implications of the back arc (v, u):

- A directed path from u to v exists in a tree of the search forest.
- Hence, there is a directed cycle containing both u and v.

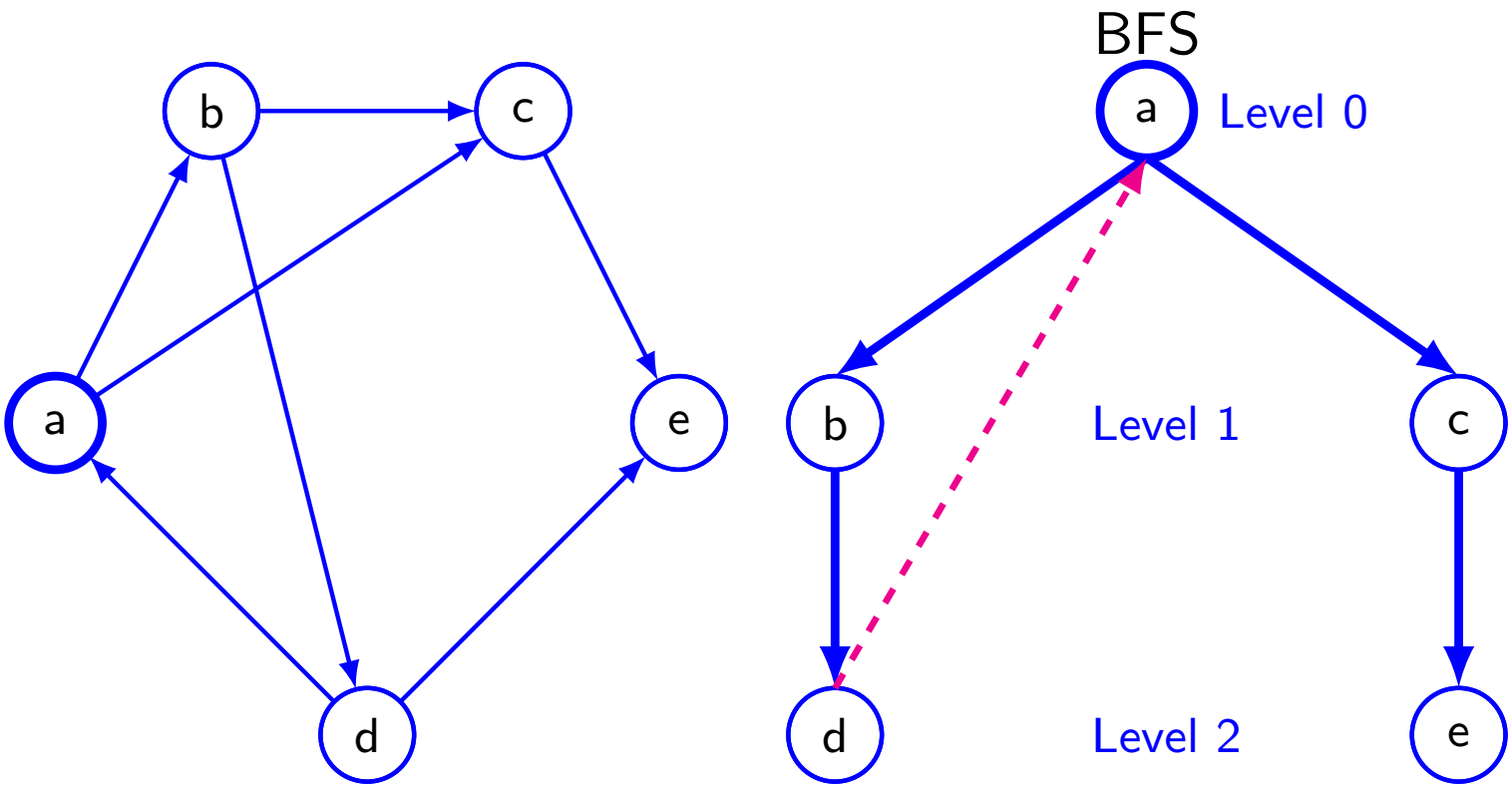


Using DFS to Find Cycles in Digraphs

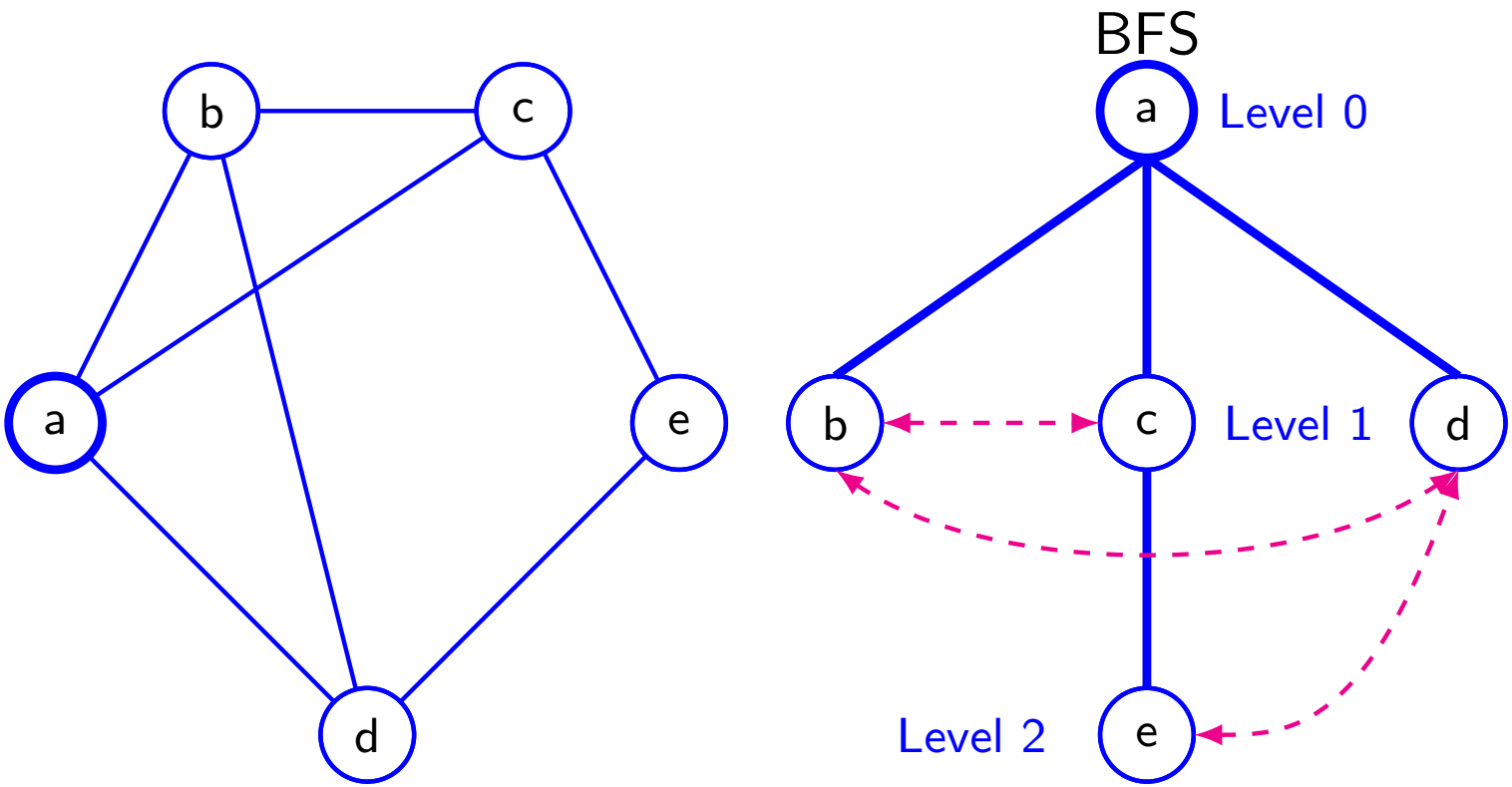




Using BFS to Find Cycles in Digraphs



Using BFS to Find Cycles in Graphs



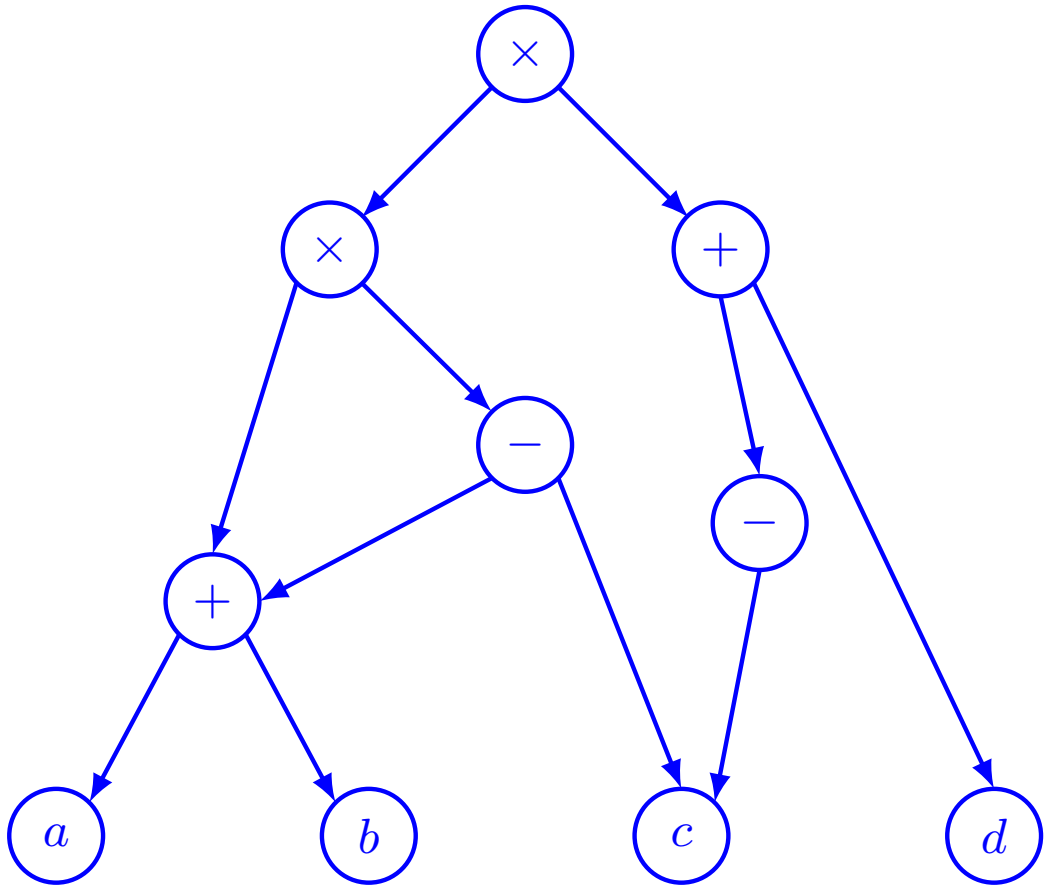
Digraph to Describe Structure of Arithmetic Expression

Evaluating expressions like

$(a+b)\times(c-(a+b))\times(-c+d)$

in a compiler by describing precedence order:

- Compute $(a+b)$ and c before $(c-(a+b))$.
- Compute $-c$ before $(-c+d)$.
- Compute $(a+b)\times(c-(a+b))$.
- Compute the expression.



Topological Sorting

Definition 5.9: Topological sort (order), or linear order

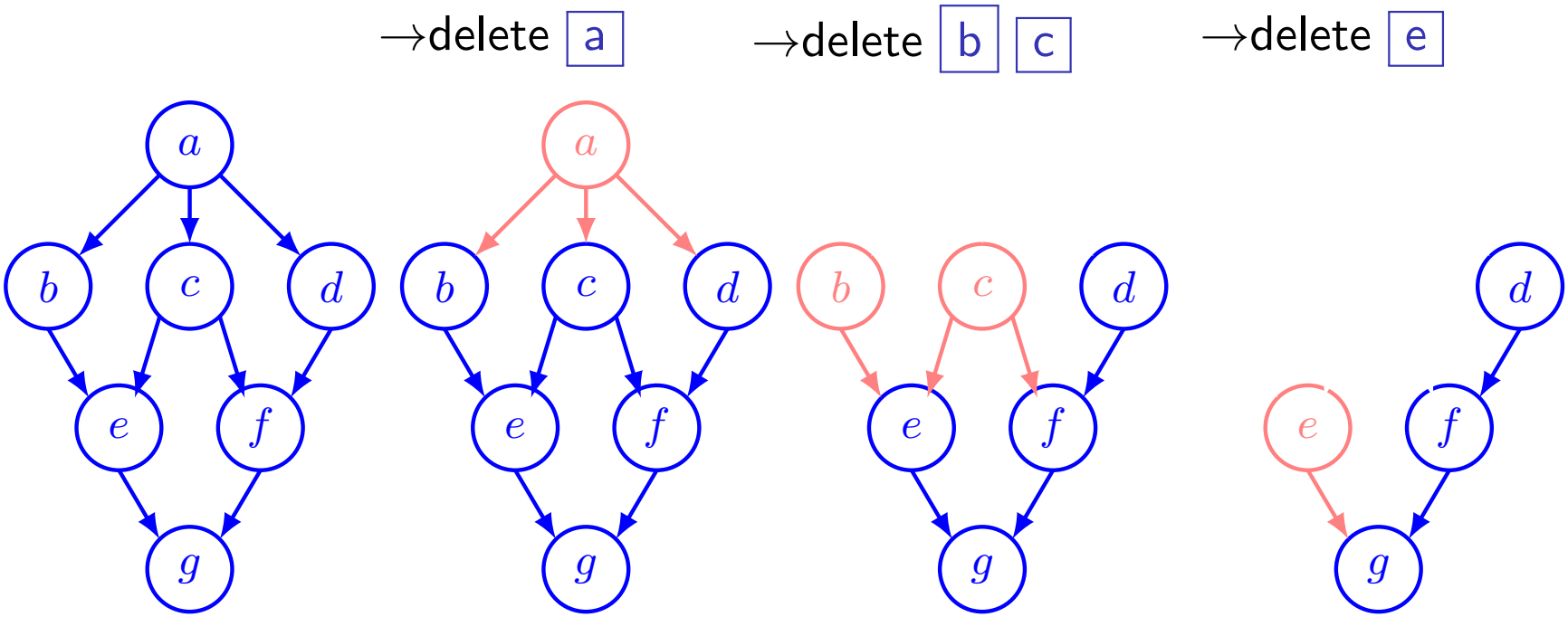
of a digraph G is a linear ordering of all its vertices, such that if $(u, v) \in E(G)$, then u appears before v in the ordering.

- **[Theorem 5.11]** Topological sort is possible iff G is a DAG.
- Main application: scheduling events (arithmetic expressions, university prerequisites, etc).

List of finishing times for DFS, in reverse order, solves the problem: due to no back arcs, each node finishes before anything pointing to it.

- *Another solution*: zero in-degree sorting: find node of in-degree zero, delete it, and repeat until all nodes listed.
- Is it less efficient than sorting via DFS?

Zero In-degree Topological Sorting



Topological sorting: a b c e d f g

Note that topological sorting is not unique and depends on a selection rule for multiple zero in-degree (i.e., source) nodes.

Zero In-degree Topological Sorting

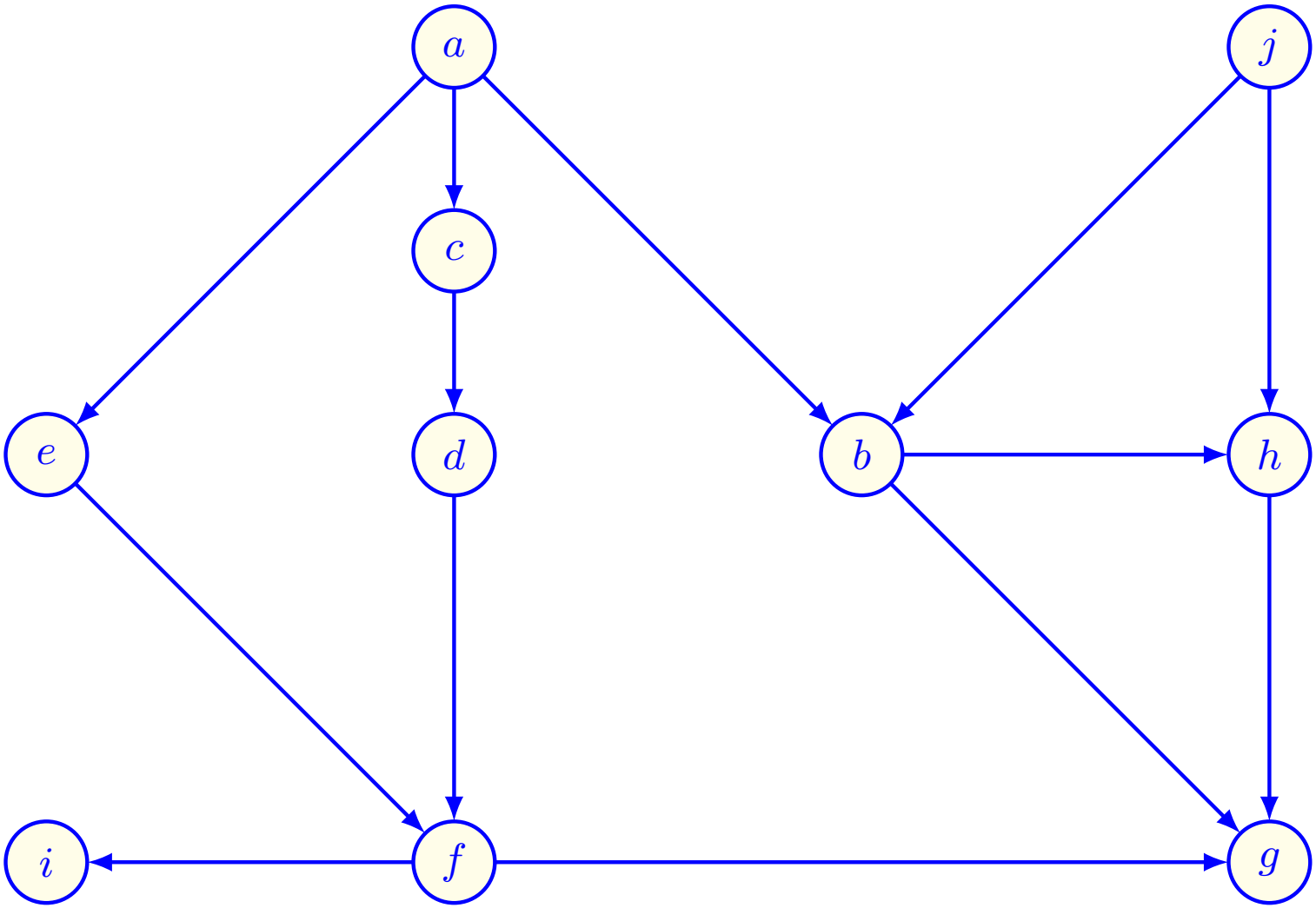
Goals:

- To decide whether a digraph G has a directed cycle or not.
  - If no zero in-degree node is found at any step, then G has a directed cycle.
- To find topological sorting of a DAG.

Running time $\mathrm{O}(n^2)$:

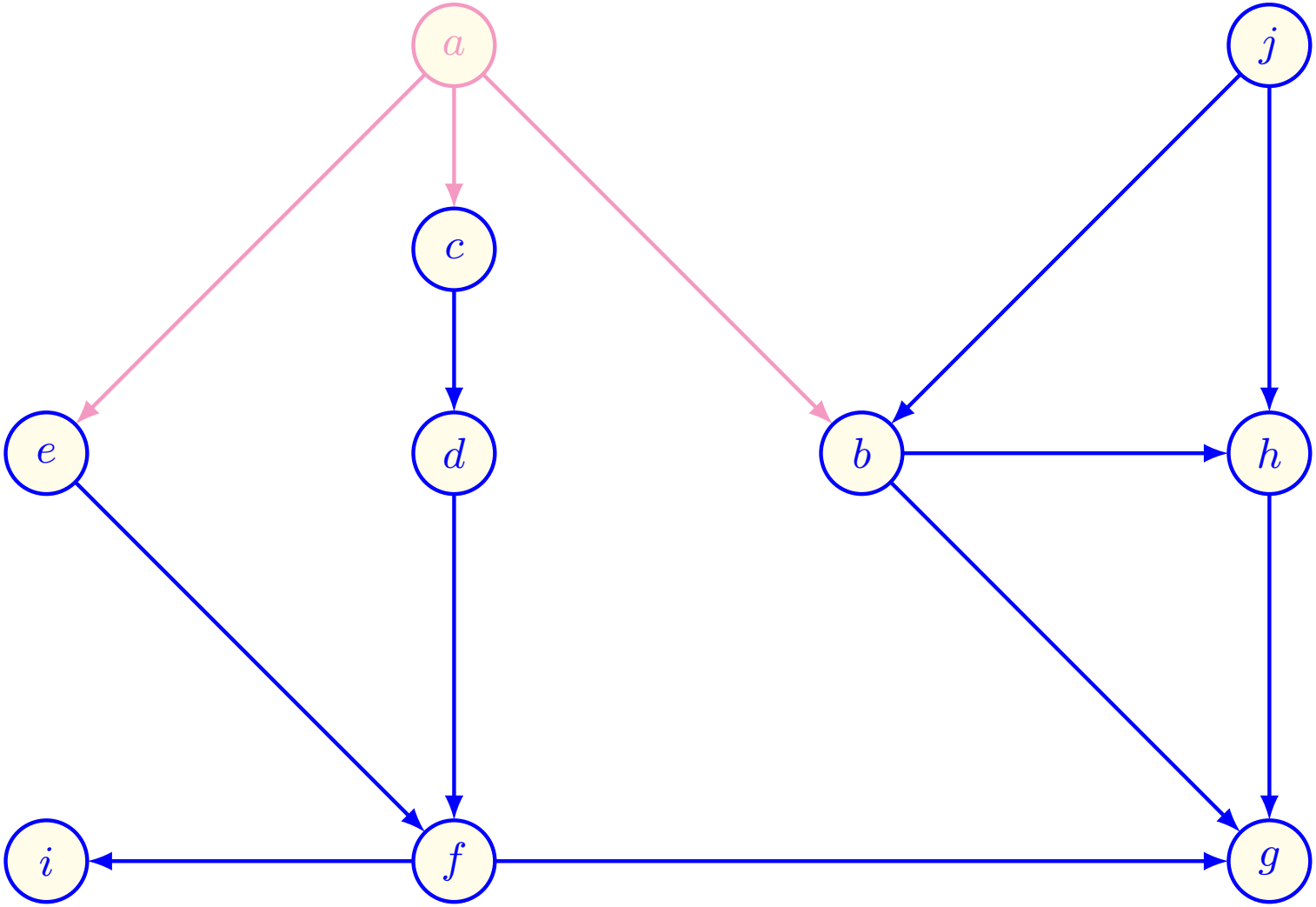
- n steps to delete ongoing zero in-degree nodes one-by-one.
- Searching at each step k through all the remaining $n - k$ nodes to find the first node with zero in-degree.
  - $\mathrm{O}(n)$ time complexity of the single step in the average and worst cases.

Zero In-degree Topological Sorting: Example 2



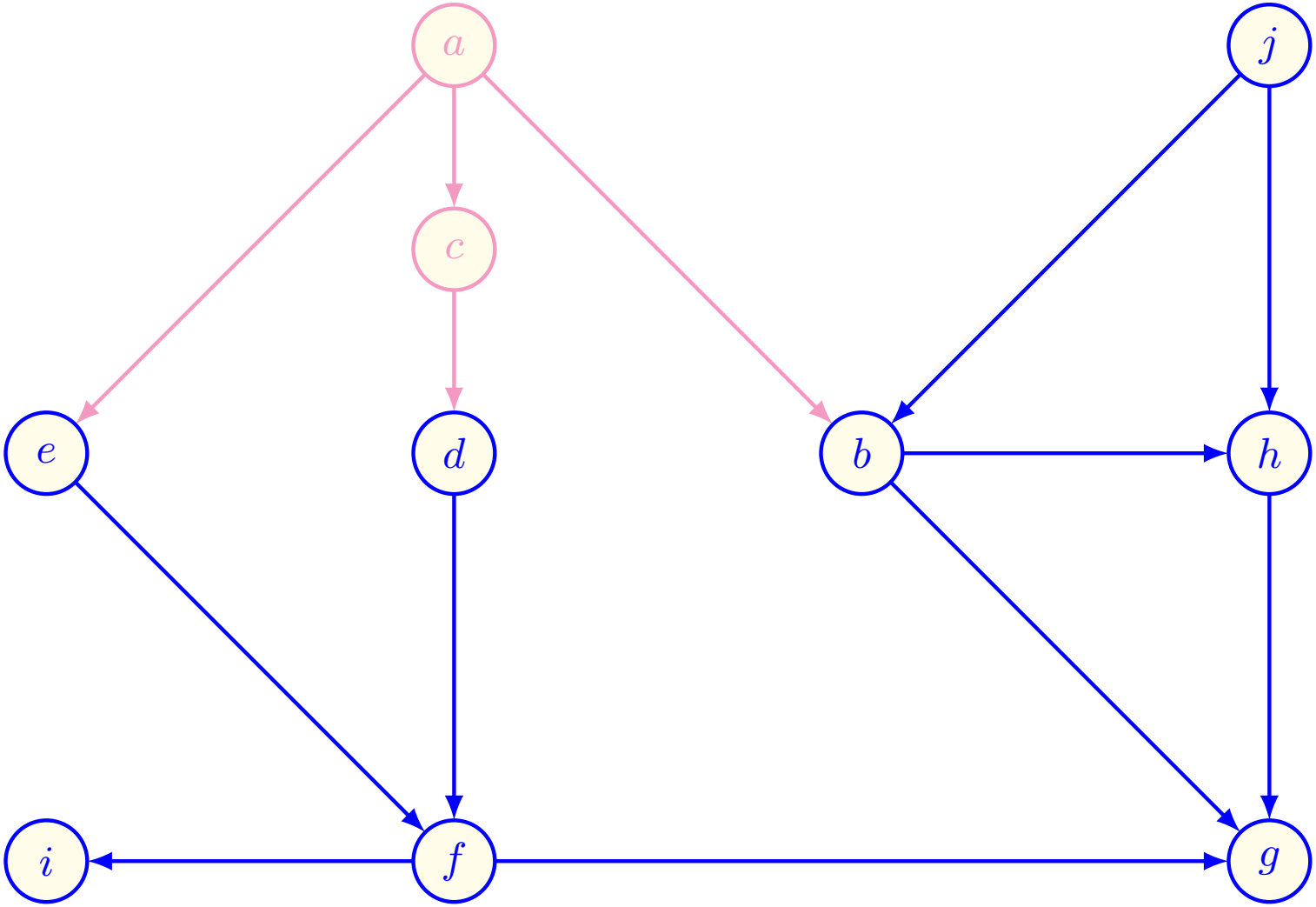
Topological sorting: –

Zero In-degree Topological Sorting: Example 2



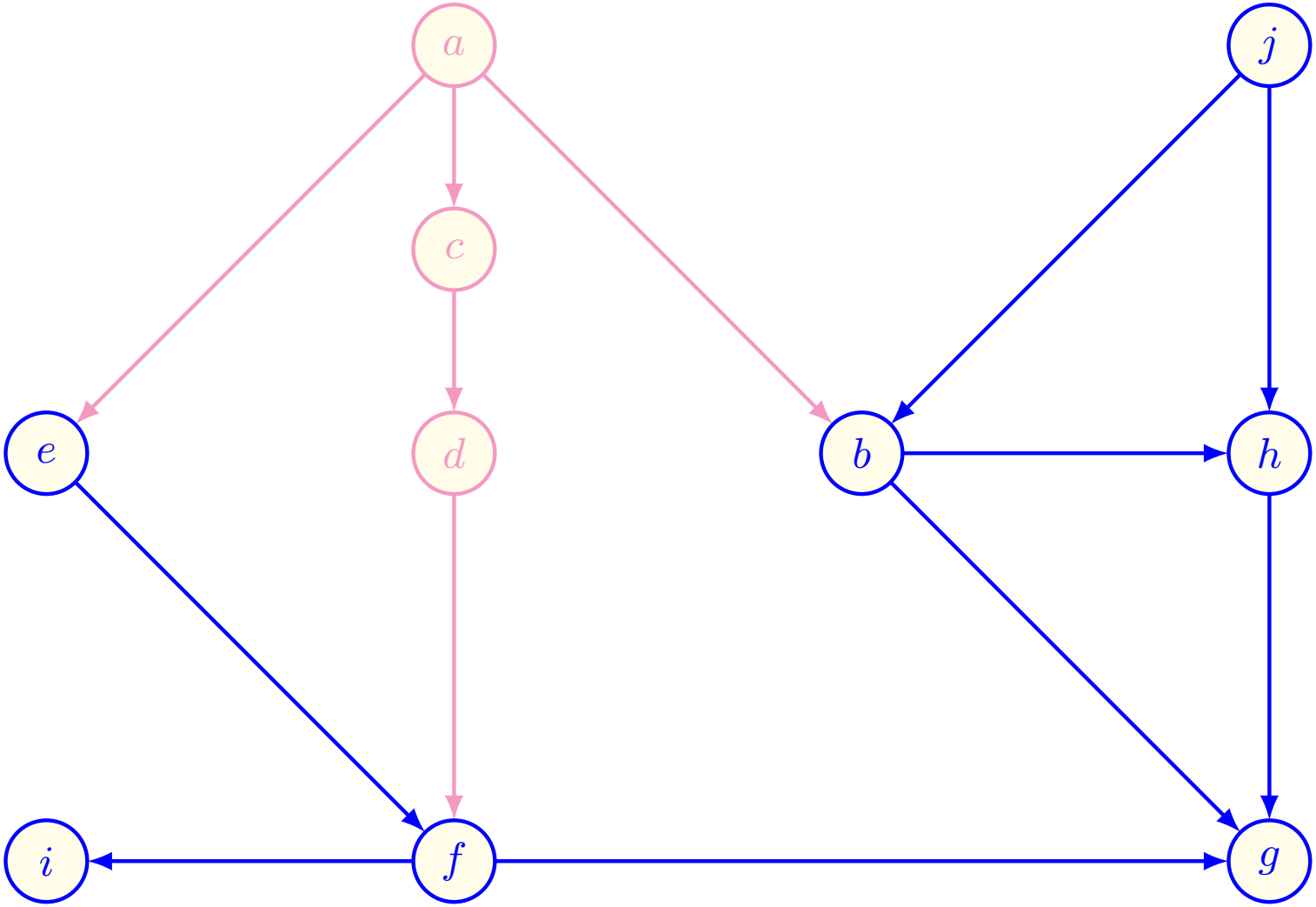
Topological sorting: a

Zero In-degree Topological Sorting: Example 2



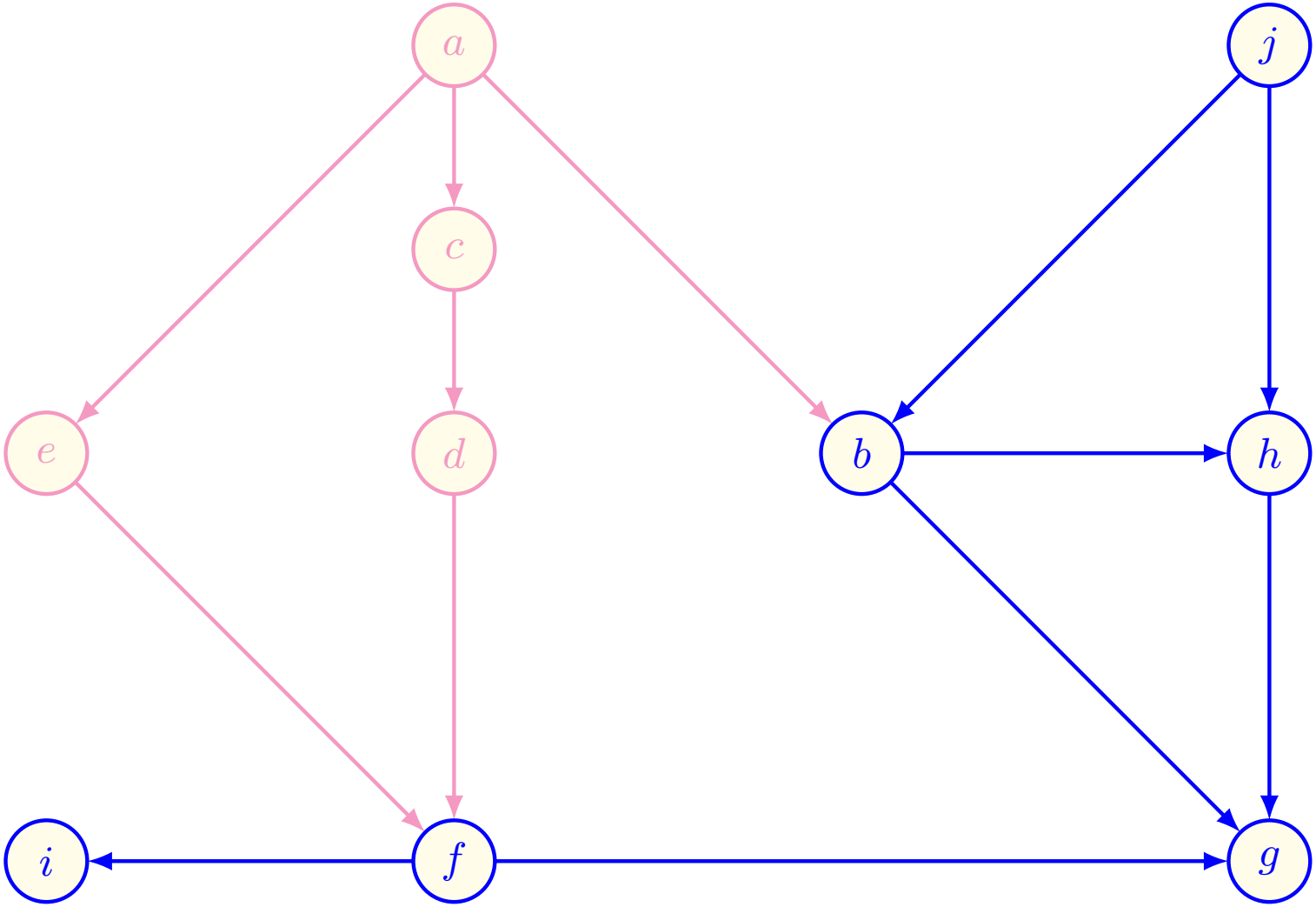
Topological sorting: $a\ c$

Zero In-degree Topological Sorting: Example 2



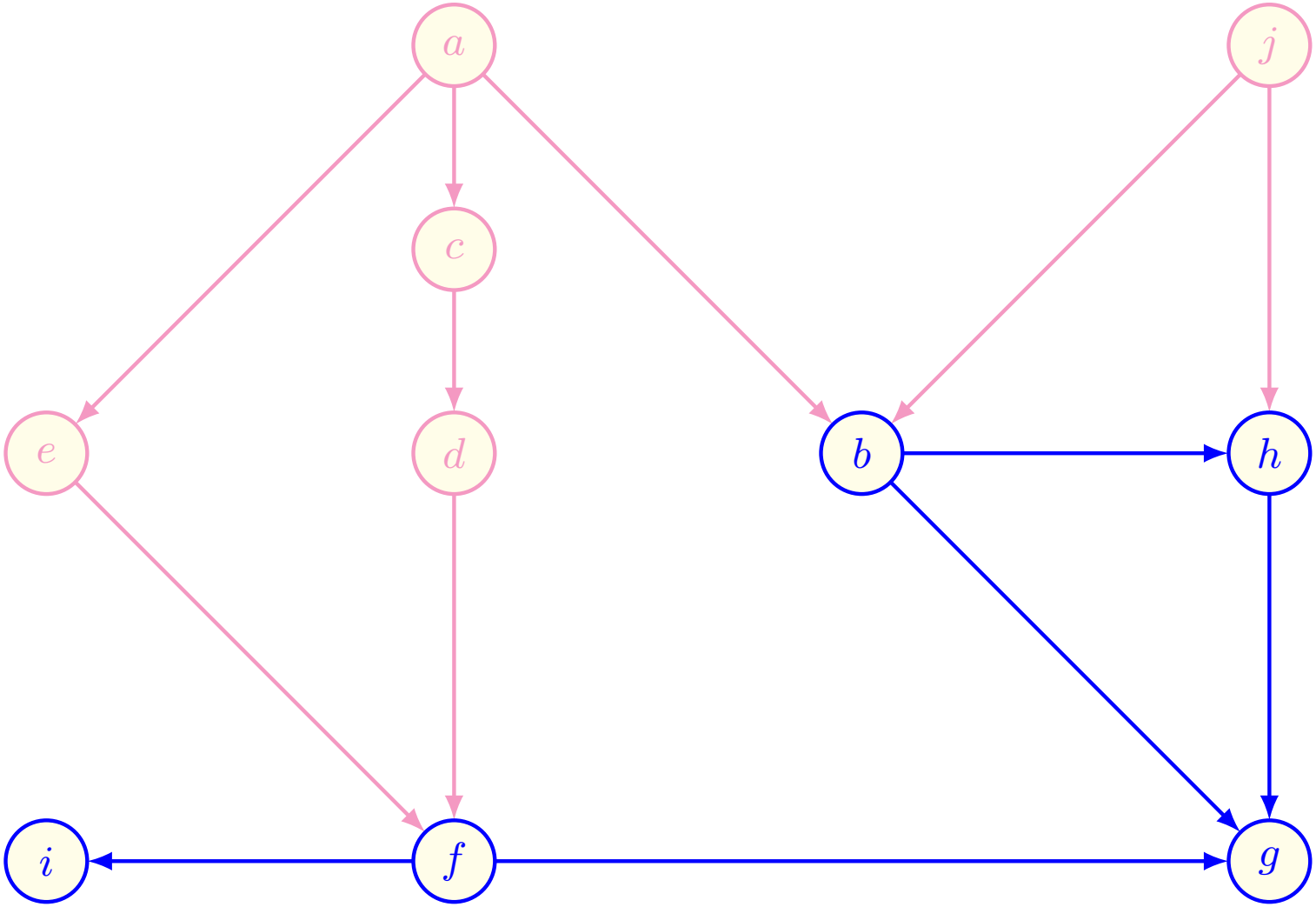
Topological sorting: $a\ c\ d$

Zero In-degree Topological Sorting: Example 2



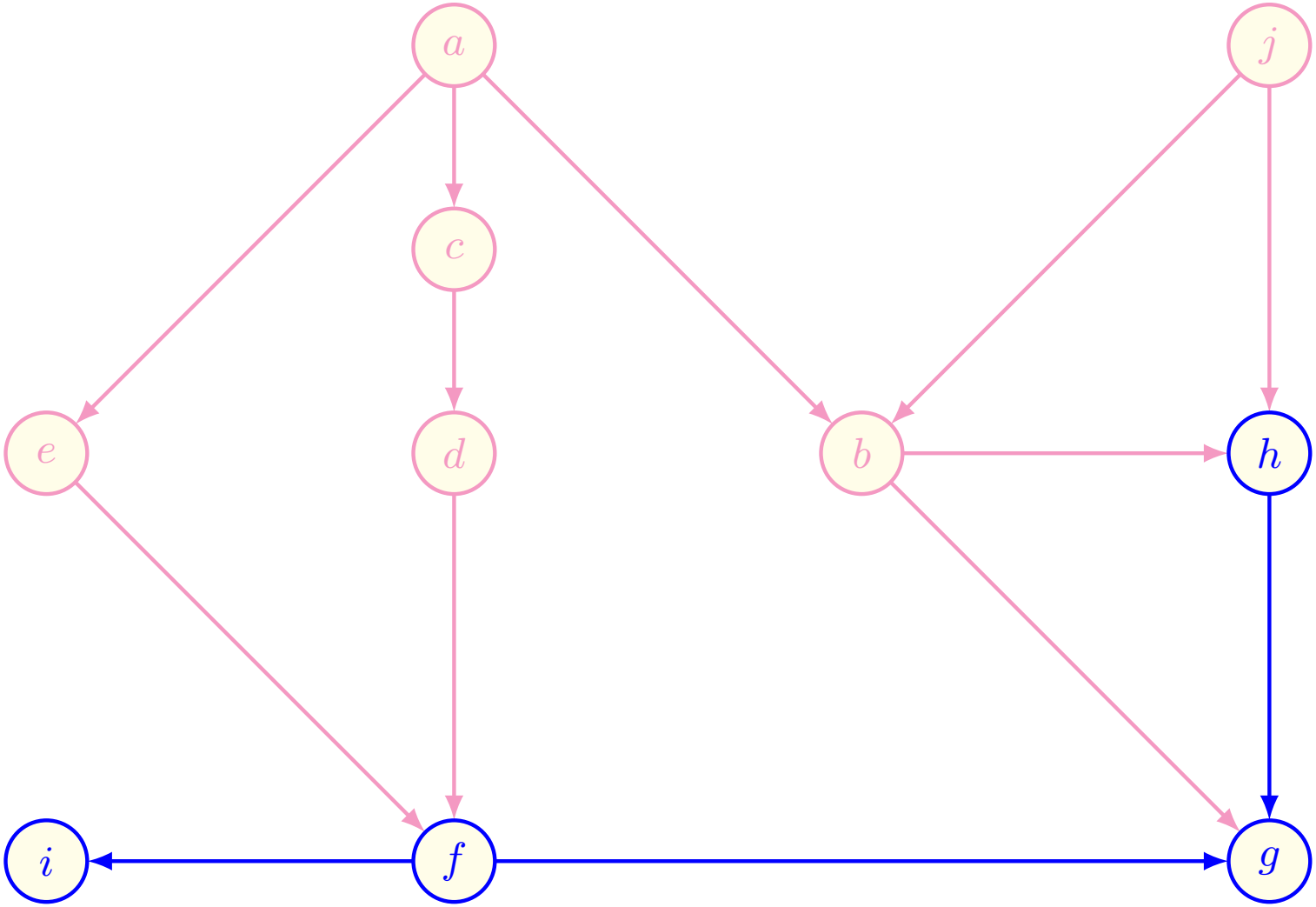
Topological sorting: $a\, c\, d\, e$

Zero In-degree Topological Sorting: Example 2



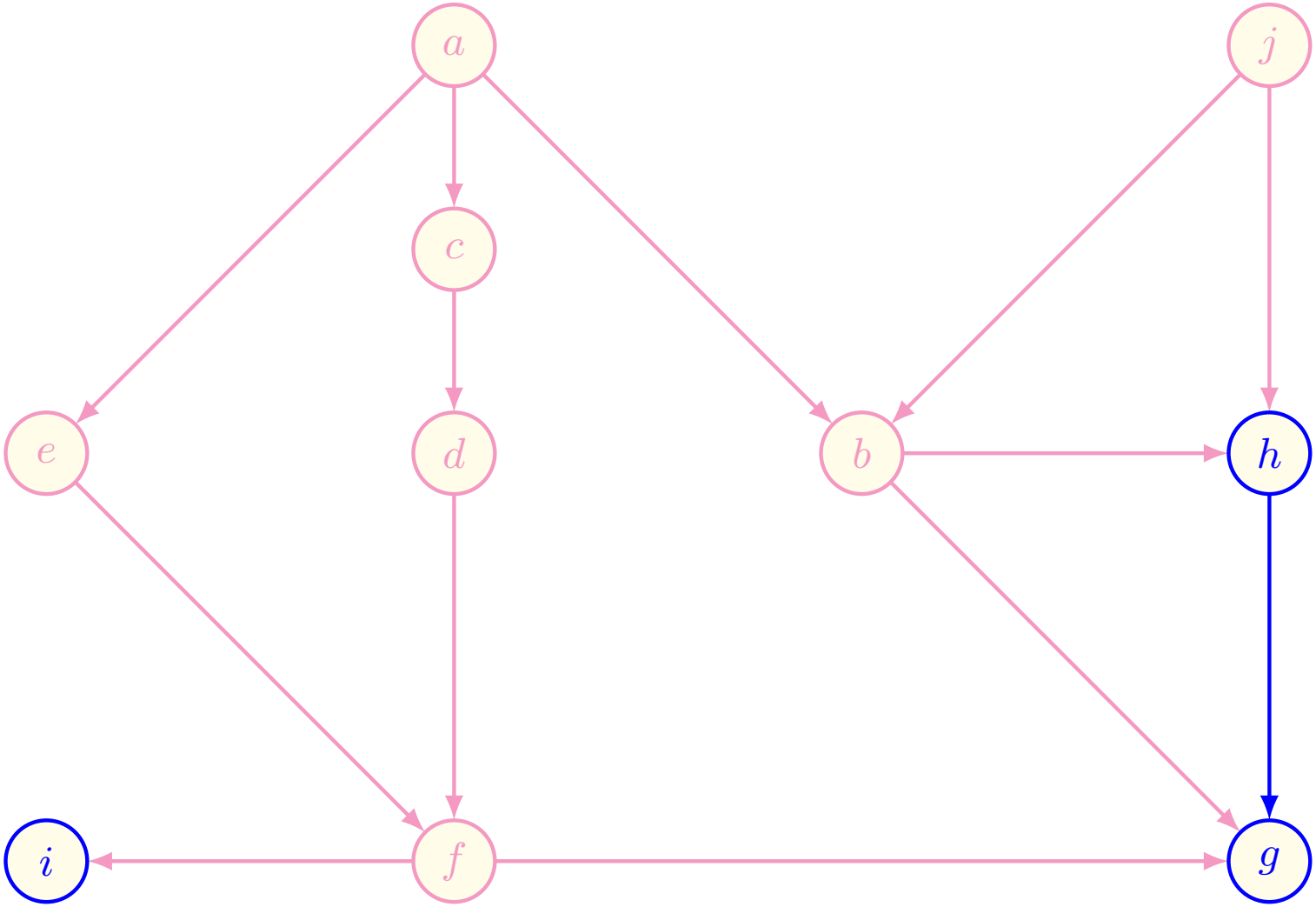
Topological sorting: $a\ c\ d\ e\ j$

Zero In-degree Topological Sorting: Example 2



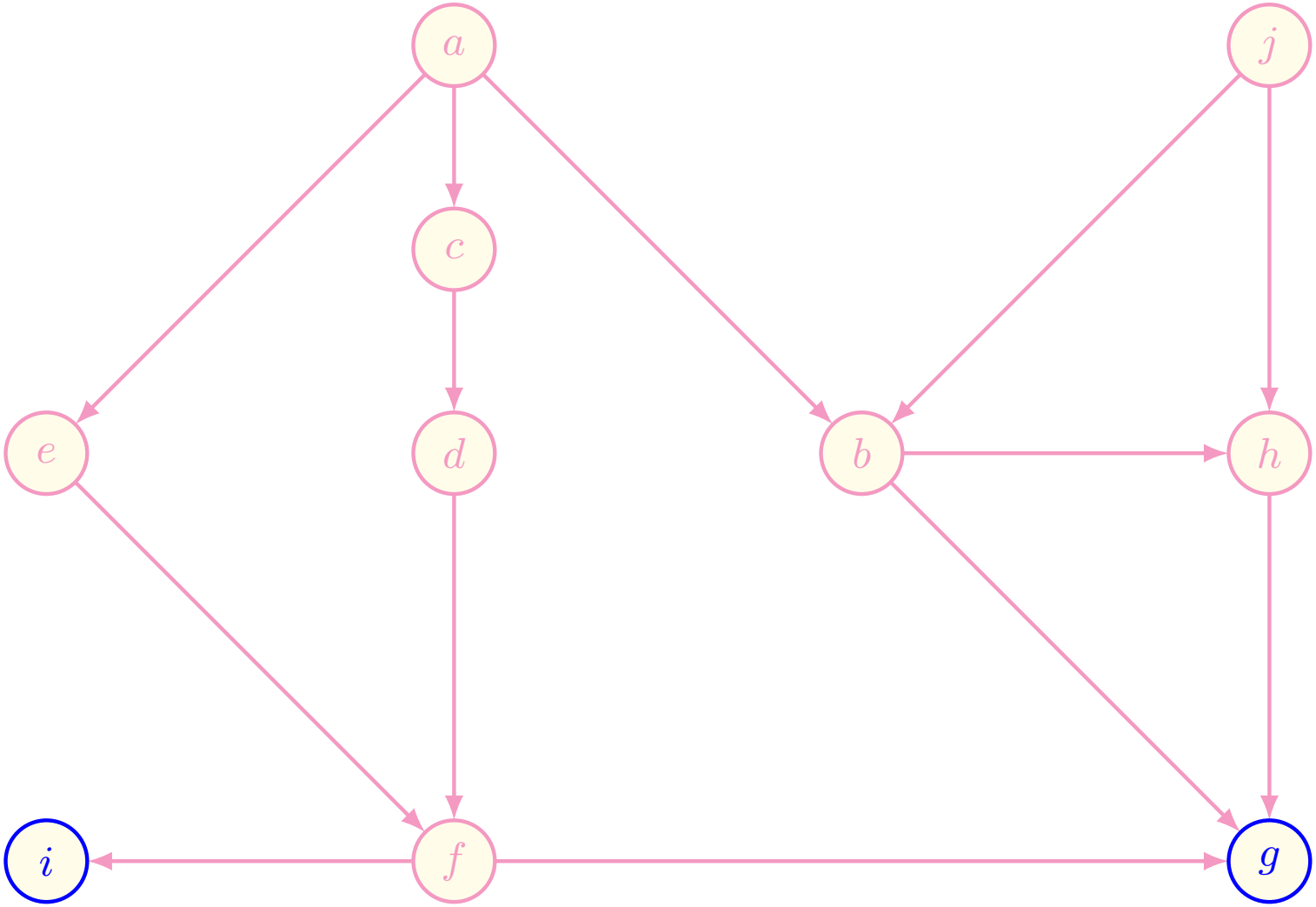
Topological sorting: $a\ c\ d\ e\ j\ b$

Zero In-degree Topological Sorting: Example 2



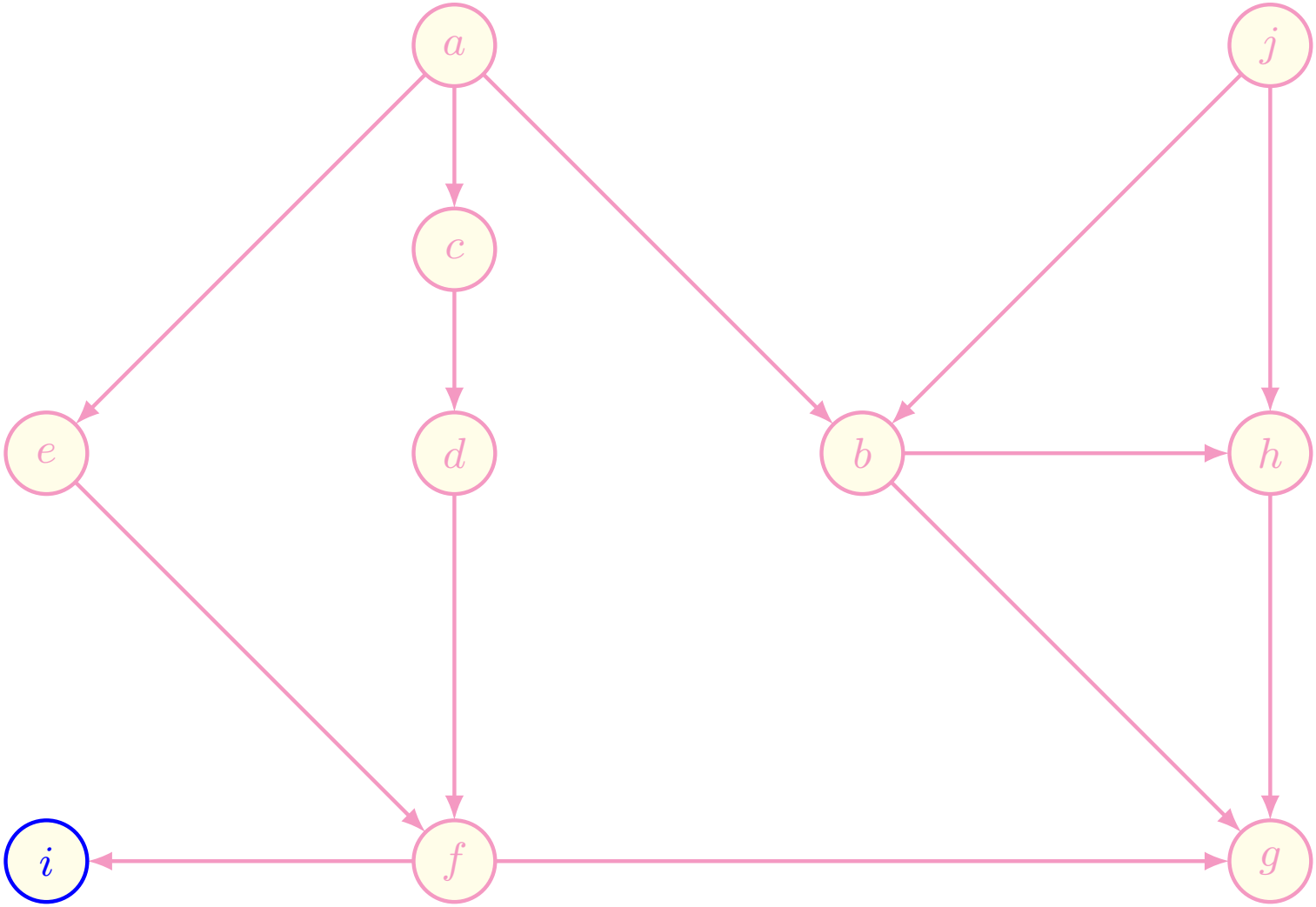
Topological sorting: $a\ c\ d\ e\ j\ b\ f$

Zero In-degree Topological Sorting: Example 2



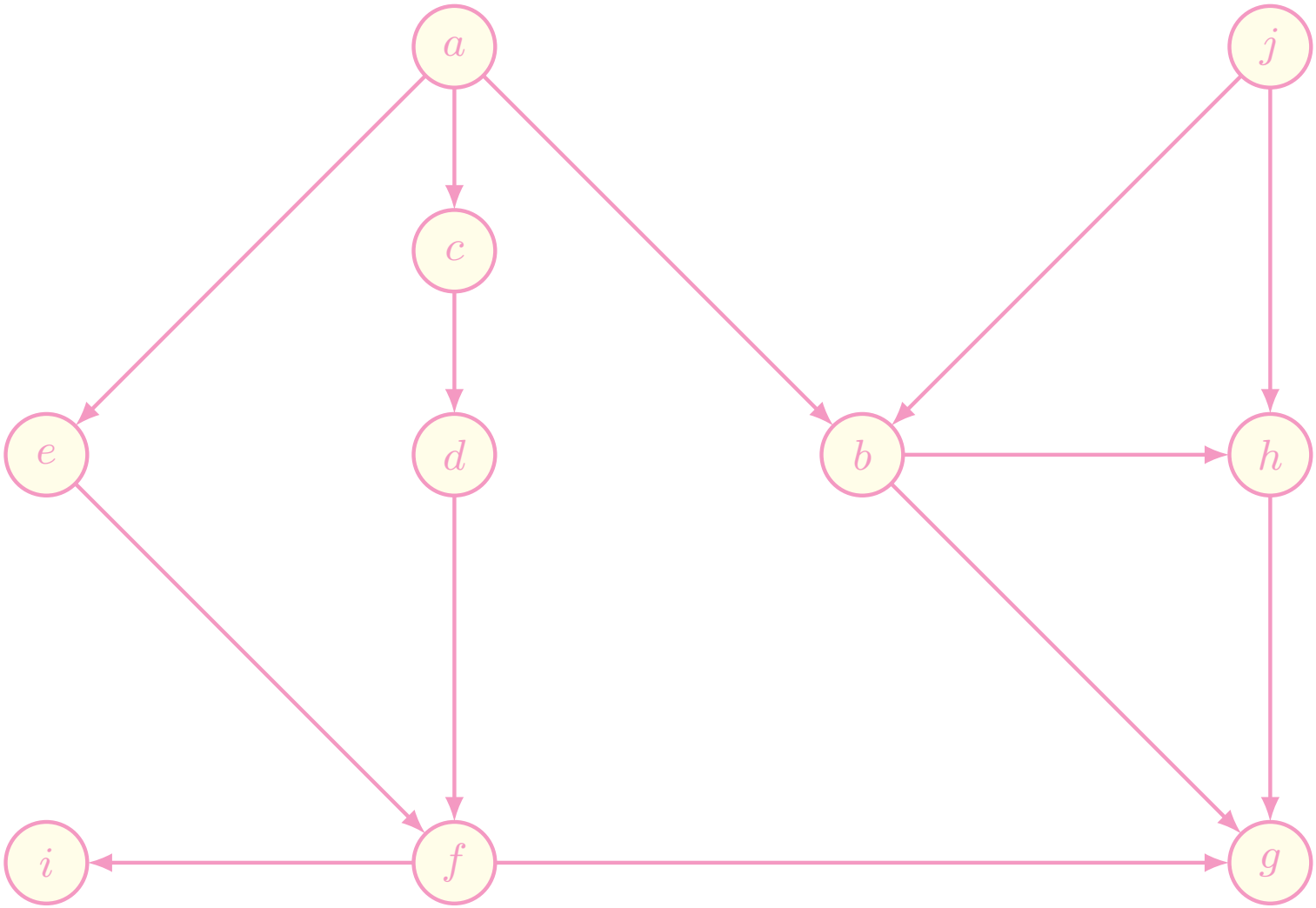
Topological sorting: $a\ c\ d\ e\ j\ b\ f\ h$

Zero In-degree Topological Sorting: Example 2



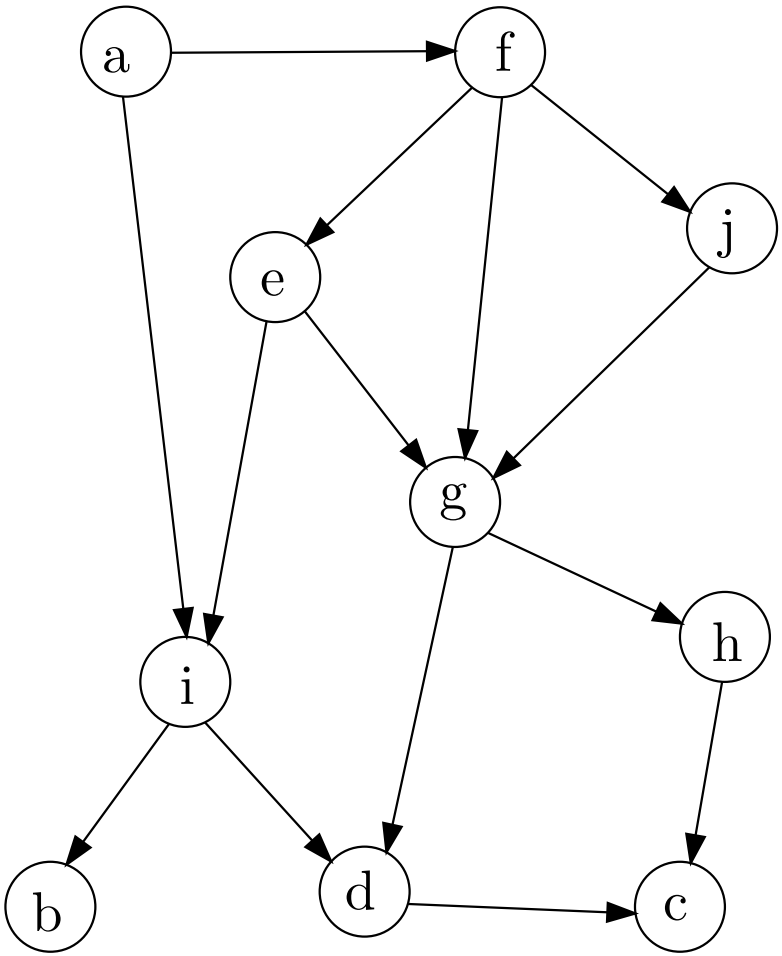
Topological sorting: $a\ c\ d\ e\ j\ b\ f\ h\ g$

Zero In-degree Topological Sorting: Example 2



Topological sorting: $a\ c\ d\ e\ j\ b\ f\ h\ g\ i$ (it is not unique!)

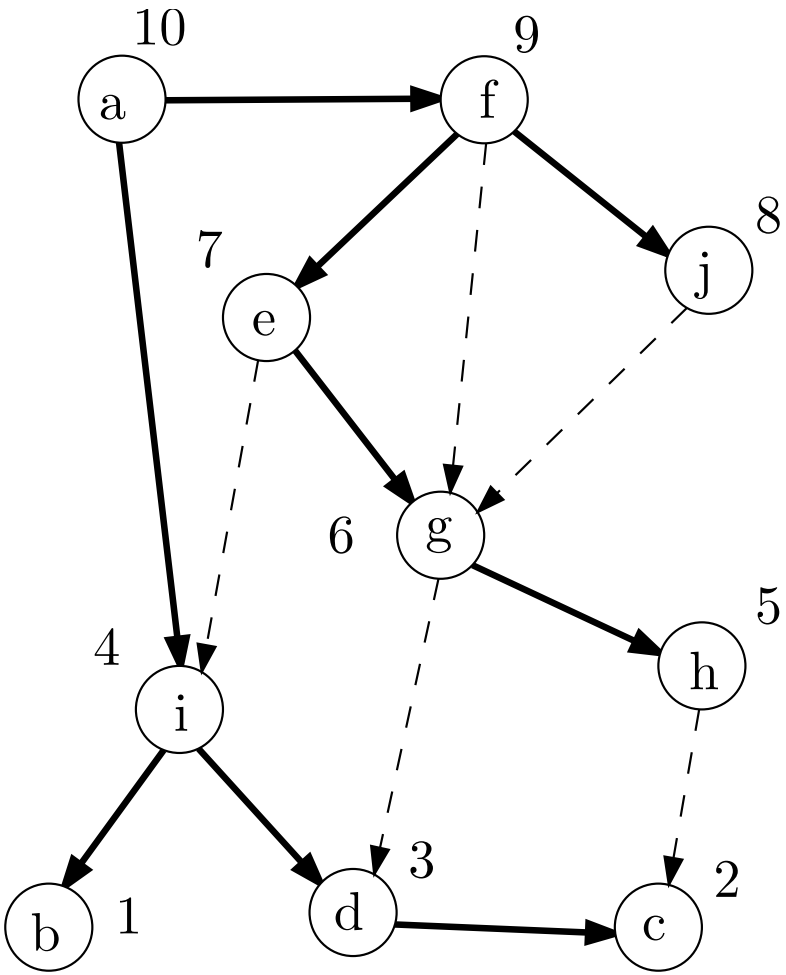
Topological Sorting via DFS



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

$$\begin{aligned} E = \{ & (a, i), (a, f), \\ & (i, b), (i, d), \\ & (f, e), (f, g), (f, j), \\ & (e, i), (e, g), \\ & (j, g), \\ & (g, d), (g, h), \\ & (d, c), \\ & (h, c)\} \end{aligned}$$

Topological Sorting via DFS



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

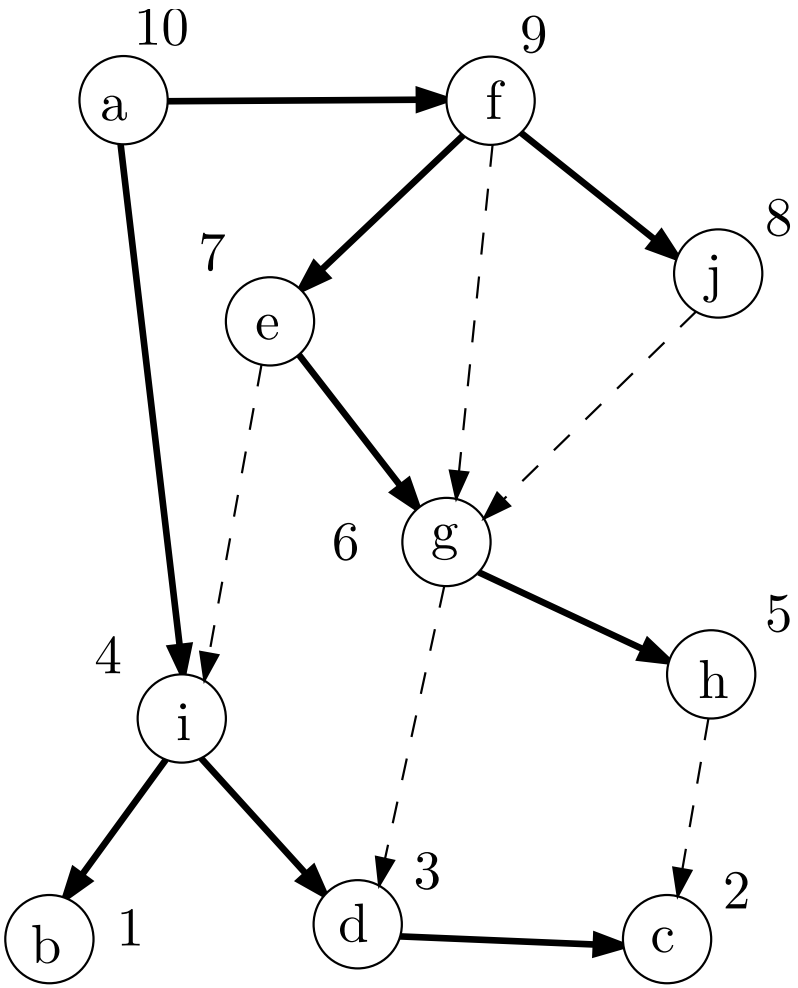
$$E = \{ (a, i), (a, f), (i, b), (i, d), (f, e), (f, g), (f, j), (e, i), (e, g), (j, g), (g, d), (g, h), (d, c), (h, c)\}$$

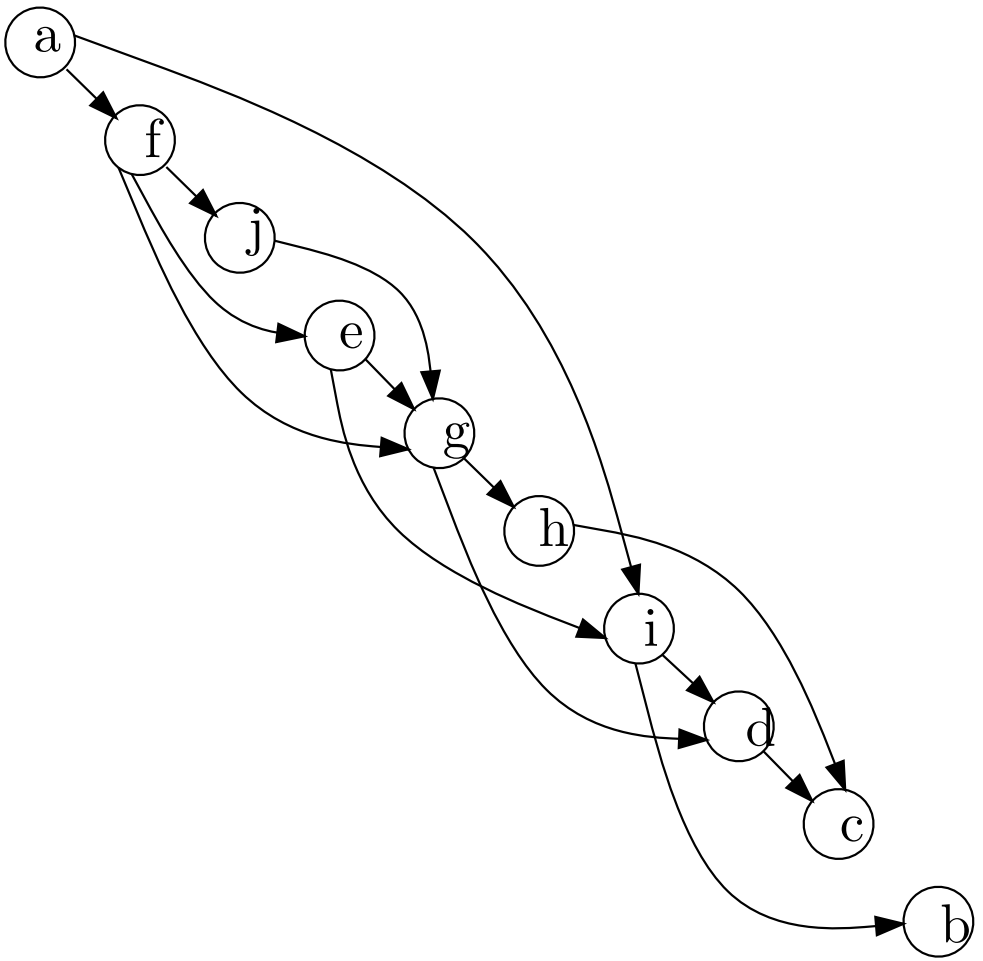
Theorem 5.13: Listing the nodes of a DAG G in reverse order of DFS finishing times yields a topological order of G.

Proof.

No arc $(u, v) \in E$ is a back arc in a DAG G, and it holds that $done[u] > done[v]$ for all other arcs, so u will be listed before v. □

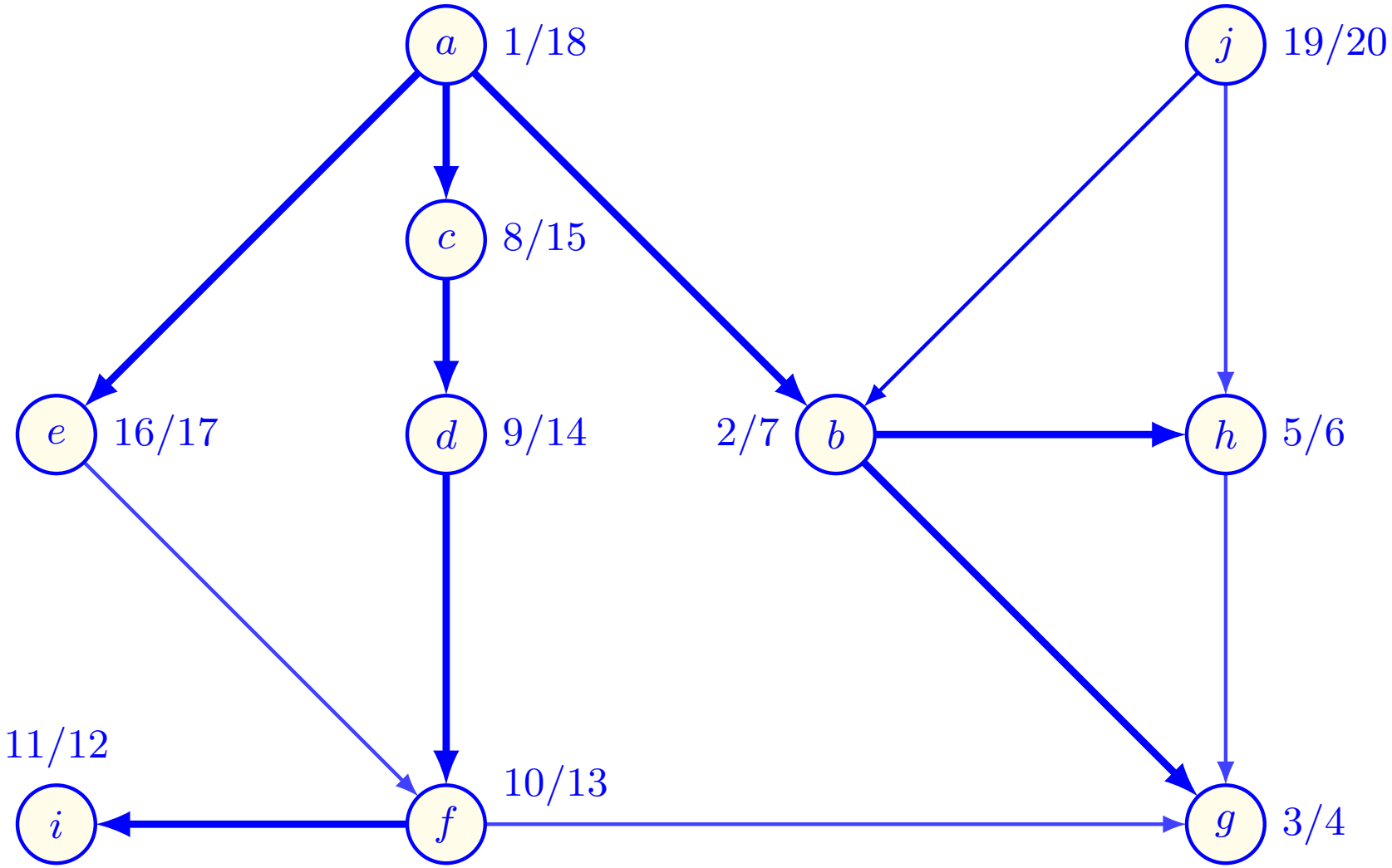
Topological Sorting via DFS





Topological Sorting via DFS: Example 2

$seen[v]/done[v]$



Topological sorting: $j\ a\ e\ c\ d\ f\ i\ b\ h\ g$

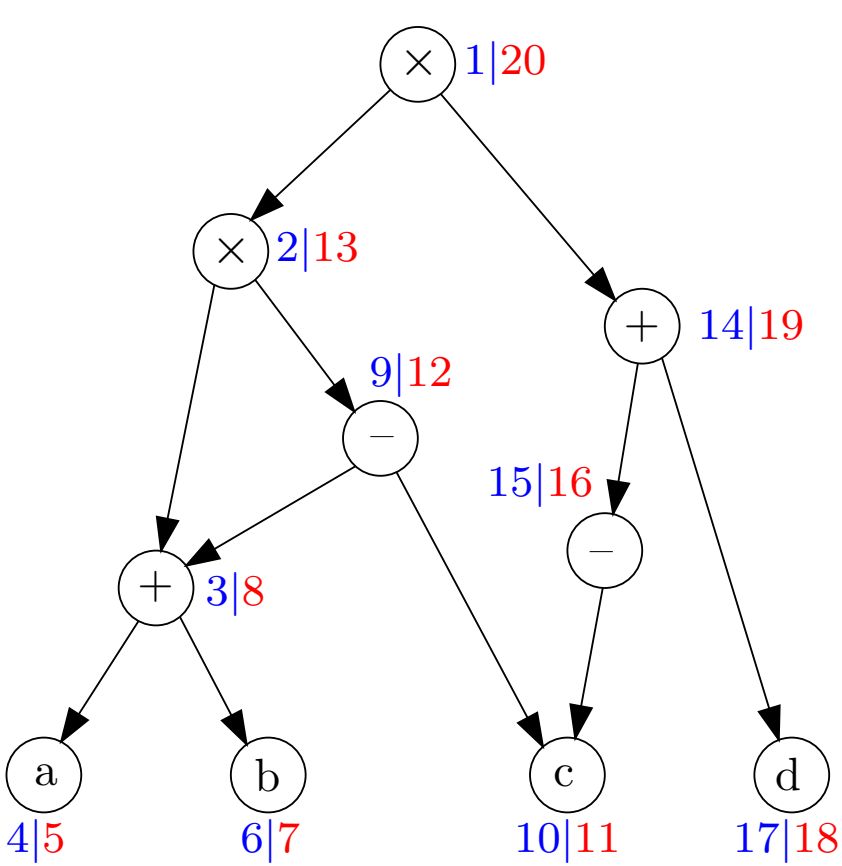
Topological Sorting via DFS: Example 2

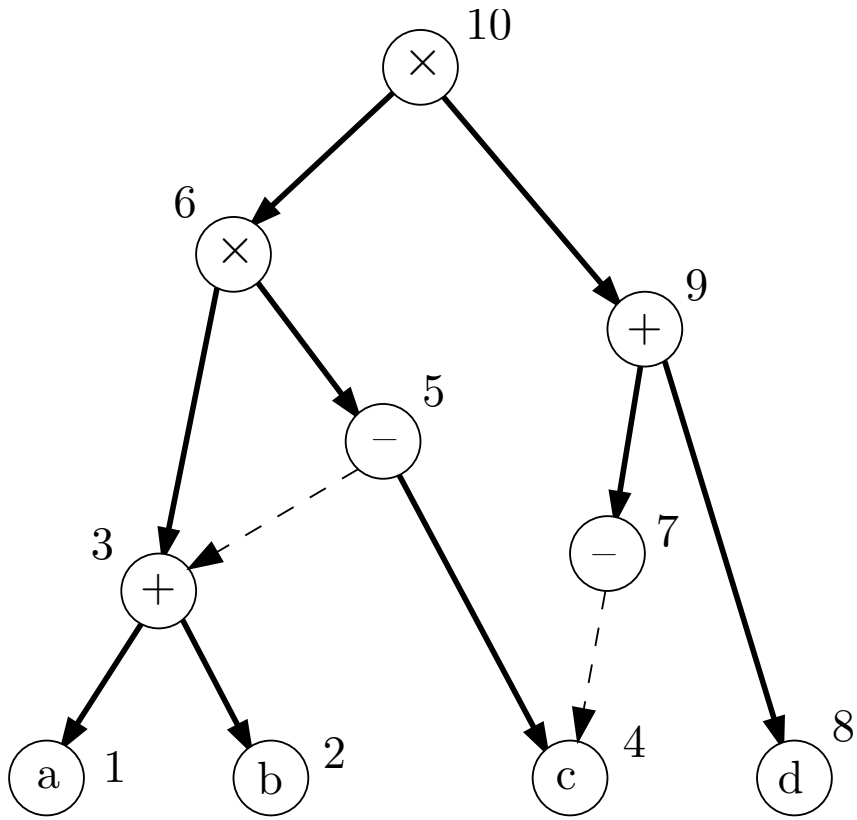
$v \in V$	a	b	c	d	e	f	g	h	i	j
$seen[v]$	1	2	8	9	16	10	3	5	11	19
$done[v]$	18	7	15	14	17	13	4	6	12	20
$pred[v]$	$-$	a	a	c	a	d	b	b	f	$-$

$time$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
stack Q	a	b	g	b	h	b	a	c	d	f	i	f	d	c	a	e	a	$-$	j	$-$
		a	b	a	b	a		a	c	d	f	d	c	a		a				
			a		a				a	c	d	c	a							
										a	c	a								
											a									

$done[v]$	20	18	17	15	14	13	12	7	6	4
sorted v	j	a	e	c	d	f	i	b	h	g

Arithmetic Expression Evaluation Order





$$(a+b)\times(c-(a+b))\times(-c+d)$$

Arithmetic Expression Evaluation Order

Parentheses-free normal Polish, or prefix mathematical notation:

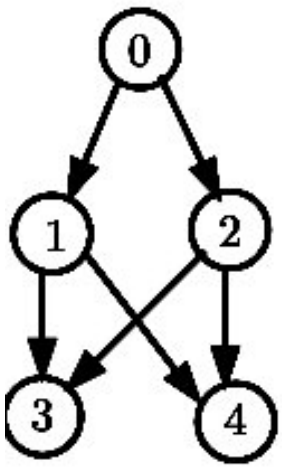
each operator is placed before its operands, e,g., $+yx$, $-yx$, or $\times yx$, in contrast to the common infix notation $x + y$, $x - y$, or $x \times y$, respectively.

See, e.g., https://en.wikipedia.org/wiki/Polish_notation

$done[v]$	5	7	8	11	12	13	16	18	19	20
Topological order:	$\times$	$+$	d	$-$	$\times$	$-$	c	$+$	b	a
Computing:								$a+b$		
								$c-(a+b)$		
								$(c-(a+b))\times(a+b)$		
								$-c$		
								$-c+d$		
								$(-c+d)\times(c-(a+b))\times(a+b)$		

Interpreters of the LISP and other programming languages use the prefix (or the equally parentheses-free postfix) notation as a syntax for math expressions.

Multiple Topological Orders of a DAG

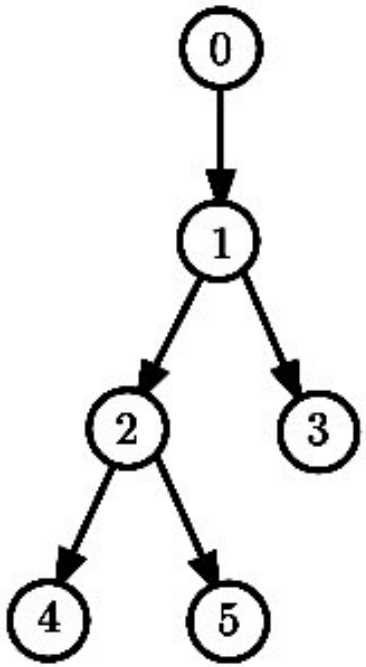


0,1,2,3,4

0,1,2,4,3

0,2,1,3,4

0,2,1,4,3



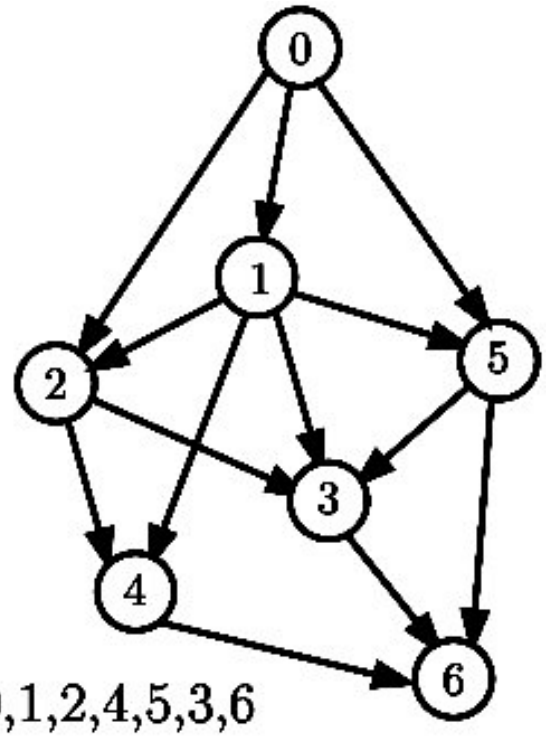
0,1,3,2,4,5

0,1,3,2,5,4

0,1,2,3,4,5

0,1,2,3,5,4

0,1,2,5,4,3



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

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

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

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

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

Graph Connectivity

Definition 5.14: A graph $G = (V, E)$ is **connected** if there is a path between each pair of its vertices $u, v \in V$.

- A graph G is **disconnected** if it is not connected.
- The maximum induced connected subgraphs are called the **components** of G.

Theorem 5.17:

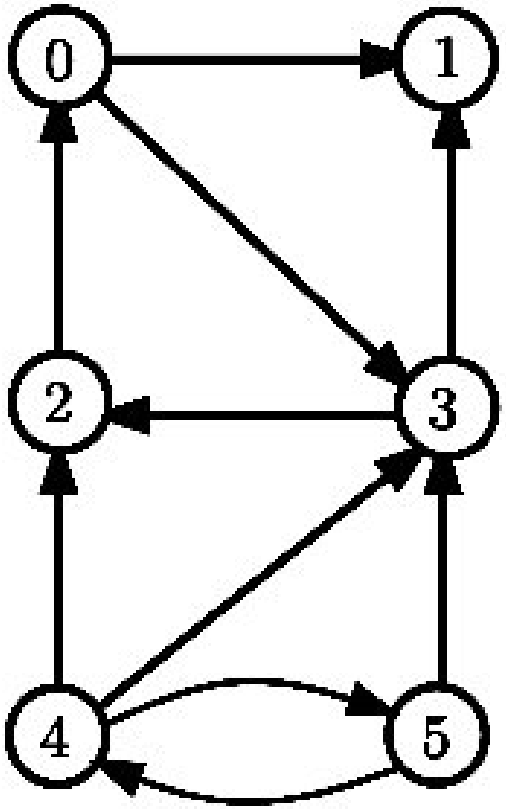
The connected components of a graph G are precisely the subgraphs spanned by the trees in the search forest obtained after DFS or BFS is run on G.

The number of components $\leftarrow$ the number of times the search chooses a root.

Nice DFS Application: Strong Components

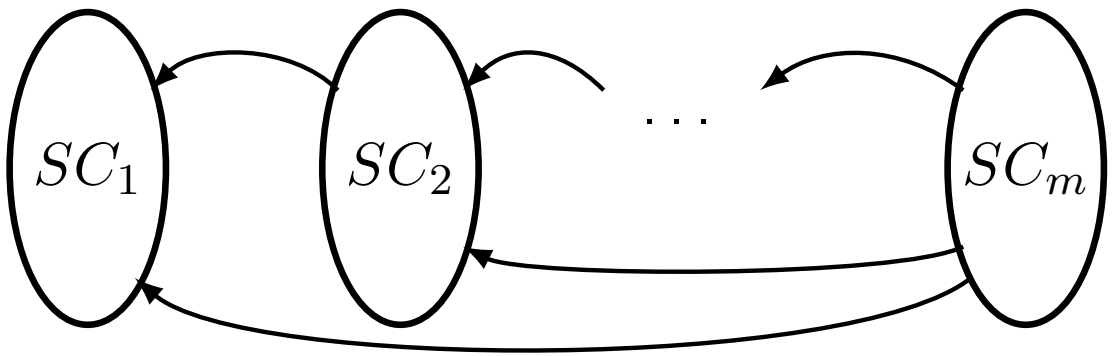
- Nodes v and w are **mutually reachable** if there is a path from v to w and a path from w to v.
- Nodes of a digraph divide up into disjoint subsets of mutually reachable nodes, which induce **strong components**.
  - For a graph, a strong component is called just a component.
  - (Strong) components are precisely the equivalence classes under the mutual reachability relation.
- A digraph is **strongly connected** if it has only one strong component.
  - Components of a graph are found easily by BFS or DFS: each tree spans a component.
  - However, this does not work well for a digraph, which may have a connected underlying graph yet not be strongly connected.
  - A new idea is needed.

Strongly Connected Components of a Digraph



Digraph G

$\Longrightarrow$ Three strong components of G



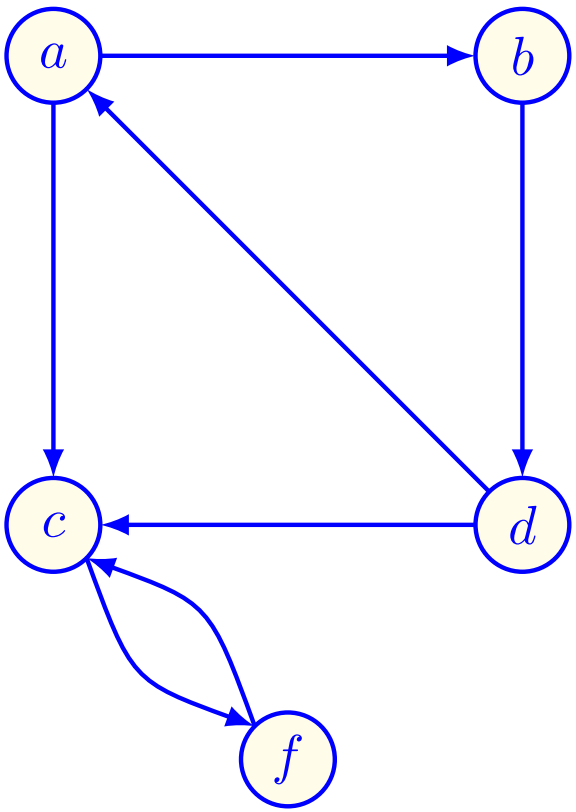
Overall digraph's structure in terms of its strong components (SC)

Strongly Connected Components Algorithm

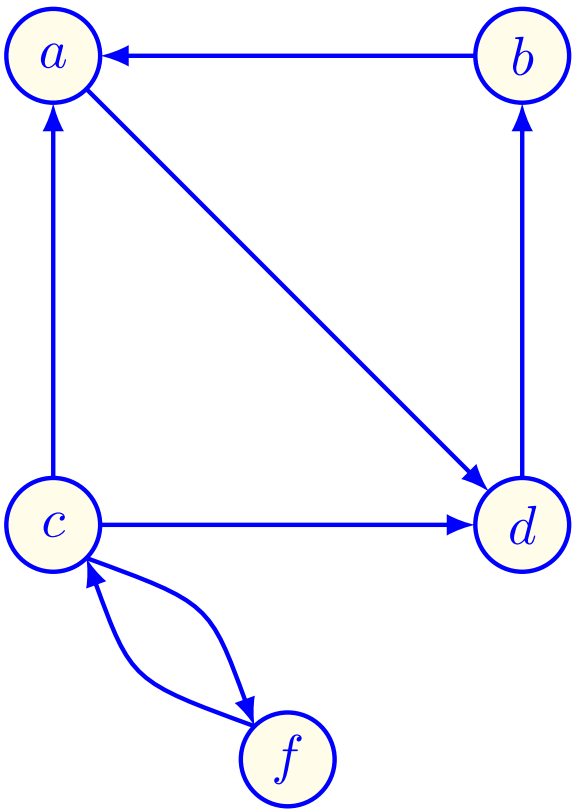
- Run DFS on G, to get depth-first forest F.
- Run DFS on reverse digraph G_{r} (with all reversed arcs).
  - Get forest F_{r} by choosing root from unseen nodes finishing latest in F.
- Suppose a node v in tree of F_{r} with root w.
  - Consider the four possibilities in F:
    1. $seen[w] < seen[v] < done[v] < done[w]$
    2. $seen[w] < done[w] < seen[v] < done[v]$
    3. $seen[v] < seen[w] < done[w] < done[v]$
    4. $seen[v] < done[v] < seen[w] < done[w]$
  - By root choice, 2nd and 3rd impossible; by root choice and since w reachable from v in G, 4th impossible.
- So v is descendant of w in F, and v, w are in the same strong component.
- The converse is easy.

Strongly Connected Components: Example 1

Digraph G

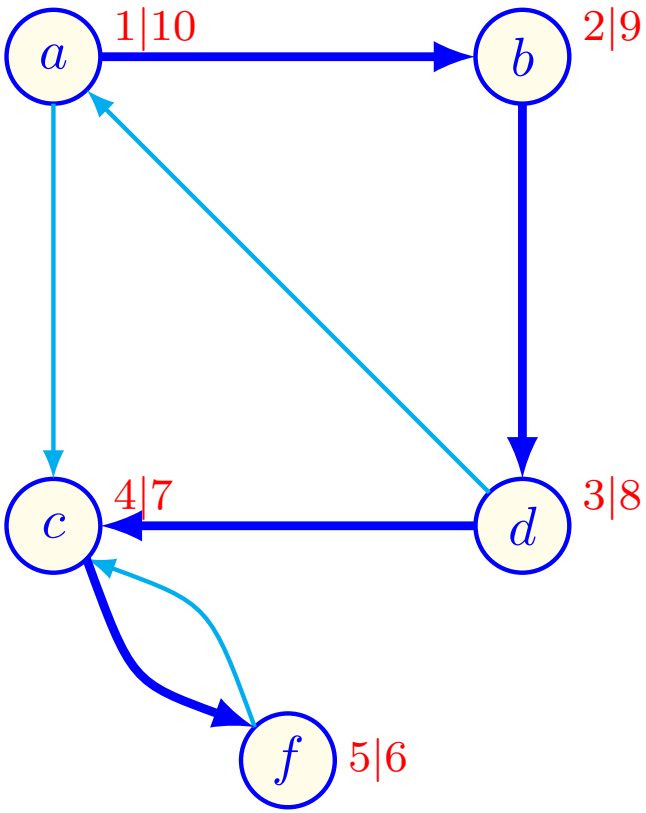


Digraph G_{r}

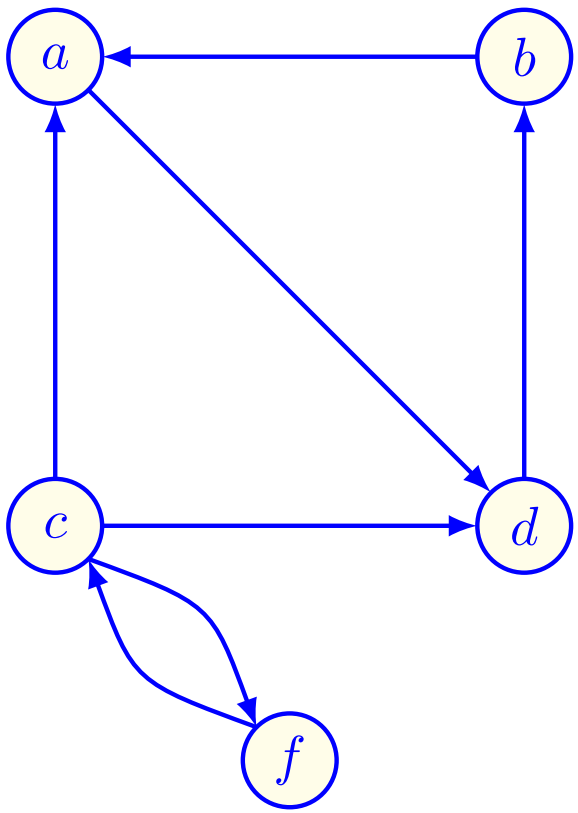


Strongly Connected Components: Example 1

Digraph G: DFS $seen[v]|done[v]$



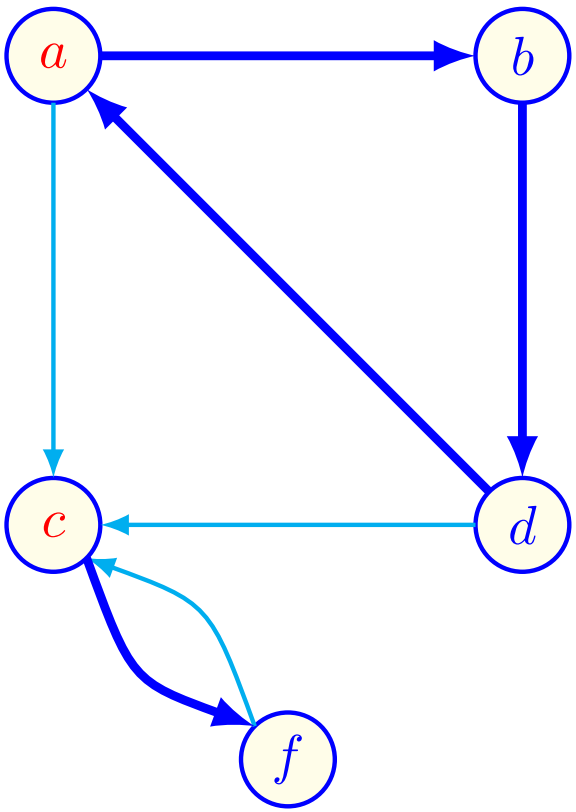
Digraph G_{r}



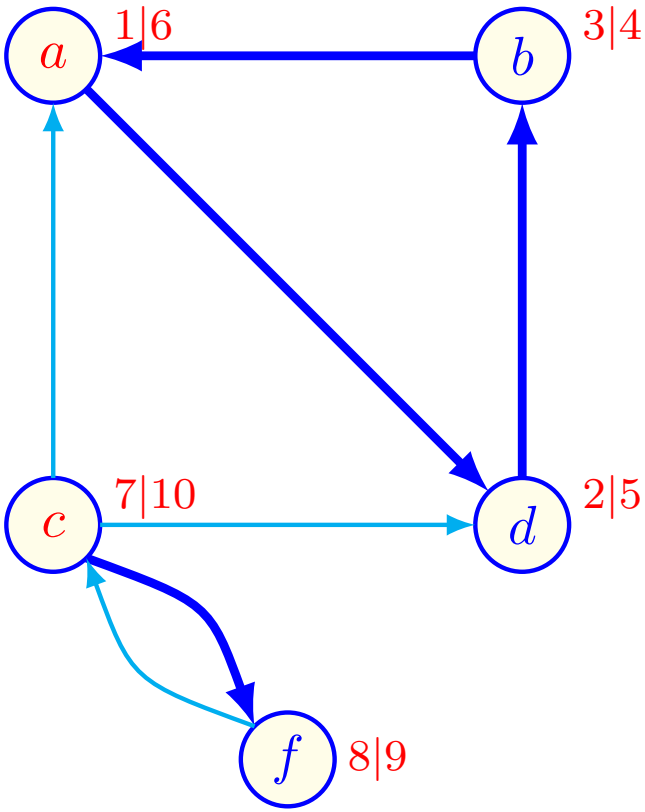
$v_{done[v]} \Leftrightarrow \{a_{10};\ b_9,;\ d_8;\ c_7;\ f_6\}$

Strongly Connected Components: Example 1

Digraph G



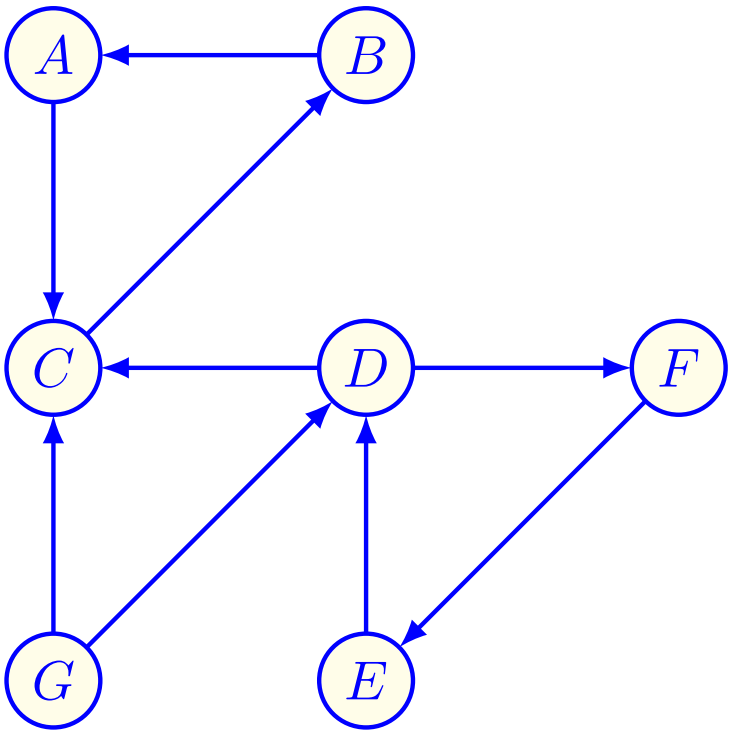
Digraph G_{r}: DFS



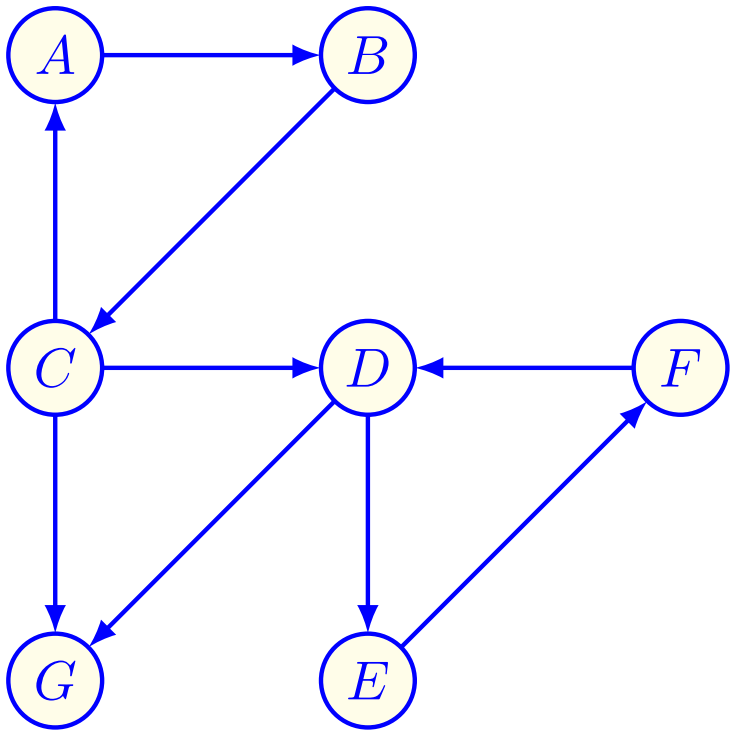
Two components: $\{\{a, d, b\}, \{c, f\}\}$

Strongly Connected Components: Example 2

Digraph G

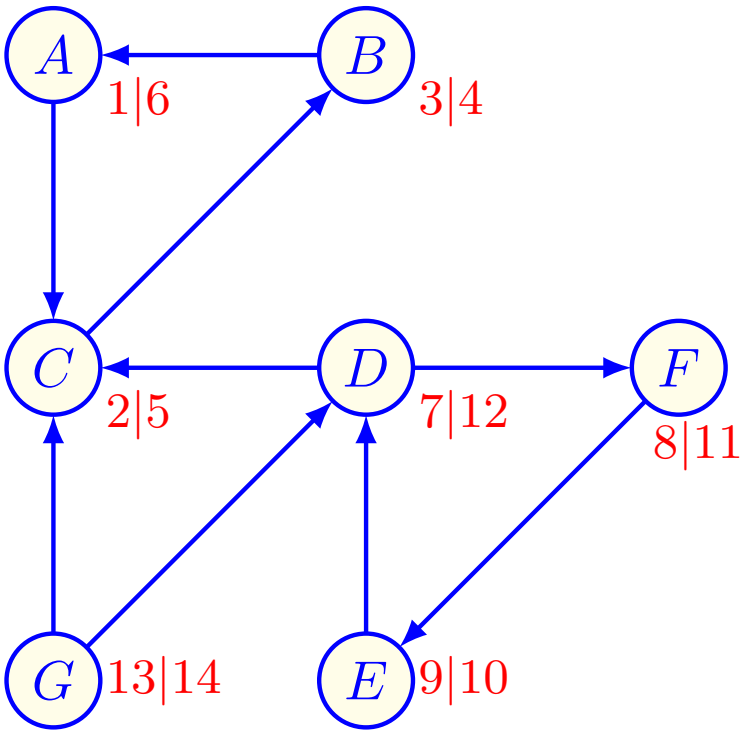


Digraph G_{r}

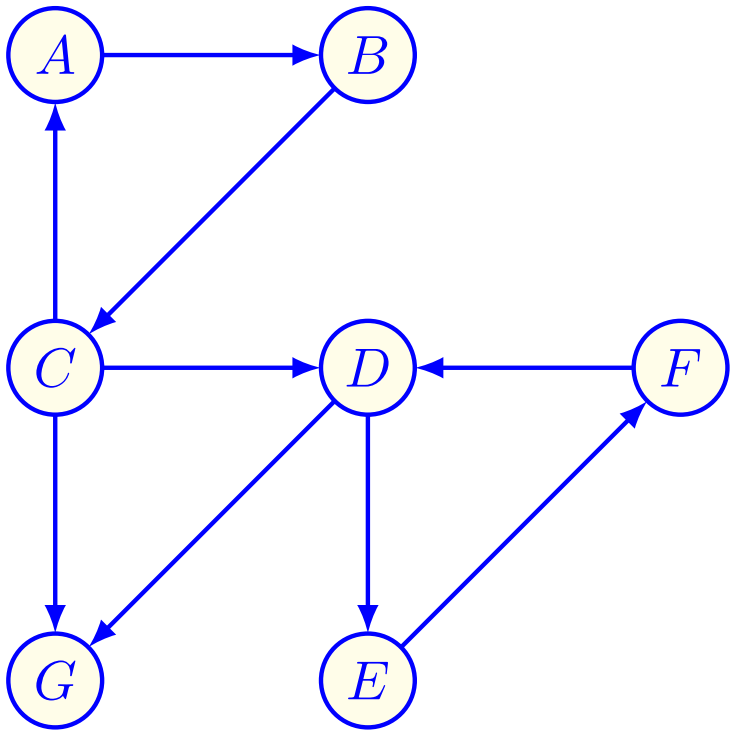


Strongly Connected Components: Example 2

Digraph G: DFS



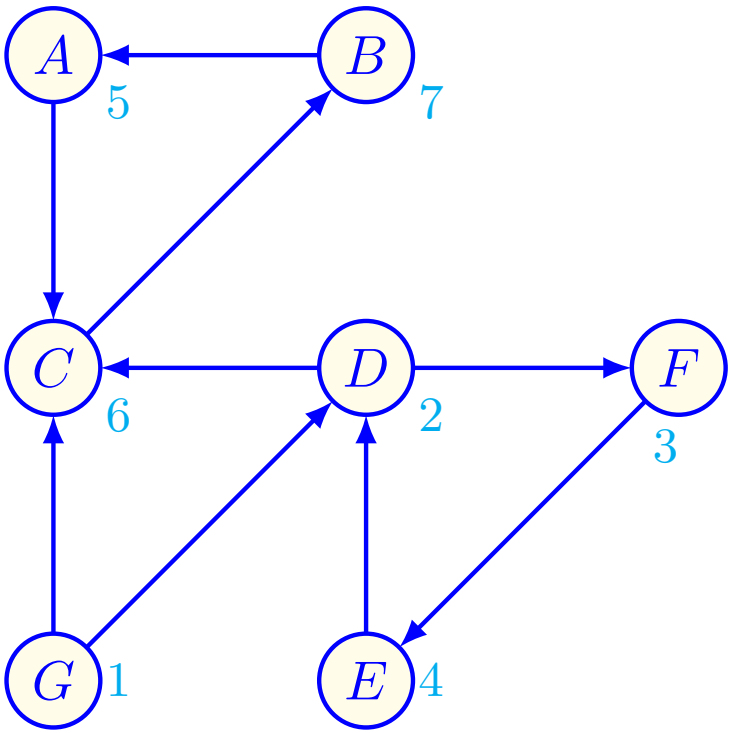
Digraph G_{r}



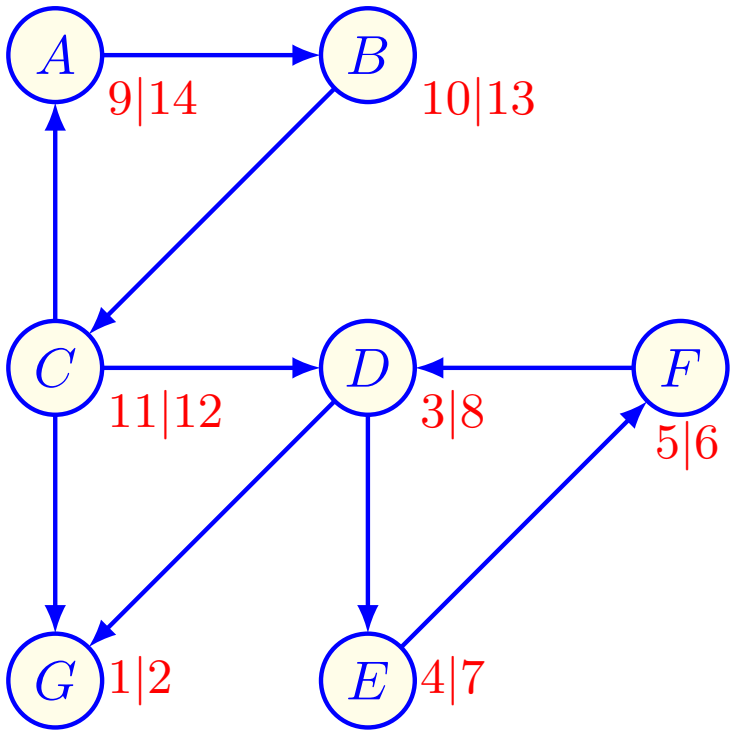
$v_{done[v]} \Leftrightarrow \{G_{14}; D_{12}, F_{11}, E_{10}, A_6, C_5, B_4\}$

Strongly Connected Components: Example 2

Digraph G



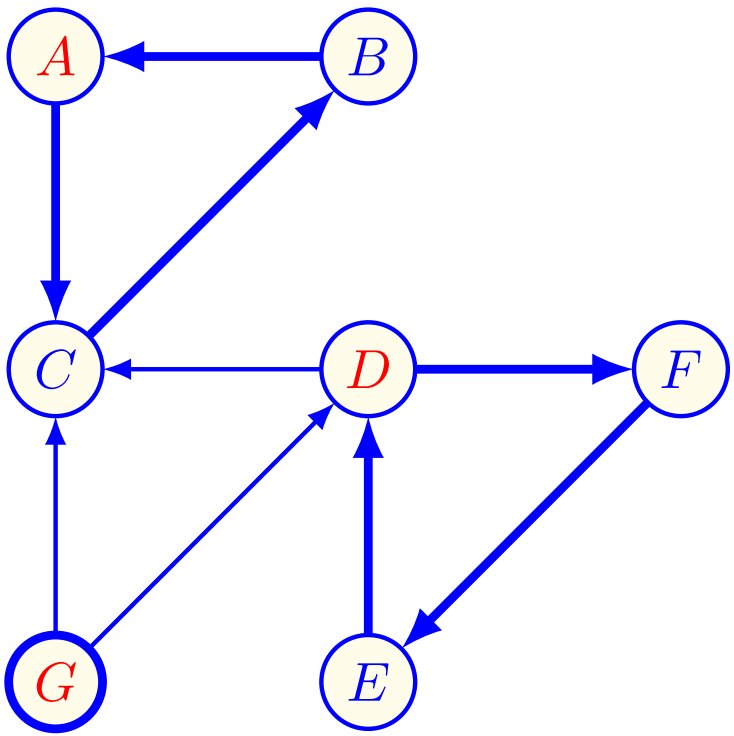
Digraph G_{r}: DFS



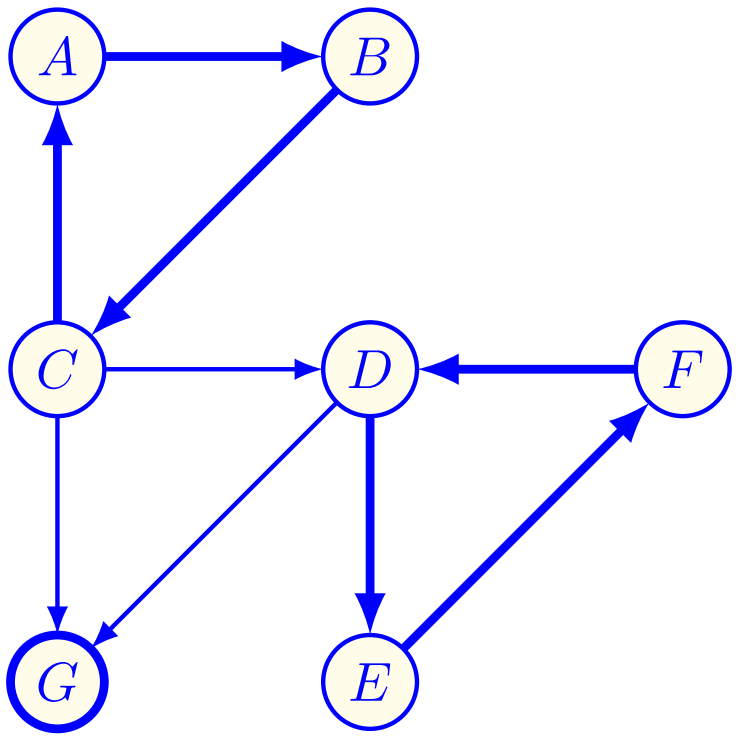
Three components: $\{\{G\}, \{D, F, E\}, \{A, B, C\}\}$

Strongly Connected Components: Example 2

Digraph G



Digraph G_{r}



Three components: $\{\{G\}, \{D, F, E\}, \{A, B, C\}\}$