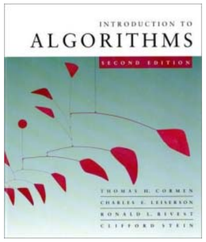


CMPS 2200 – Fall 2012

Graphs

Carola Wenk

Slides courtesy of Charles Leiserson with changes and additions by Carola Wenk

Graphs (review)

Definition. A ***directed graph (digraph)*** $G = (V, E)$ is an ordered pair consisting of

- a set V of ***vertices*** (singular: ***vertex***),
- a set $E \subseteq V \times V$ of ***edges***.

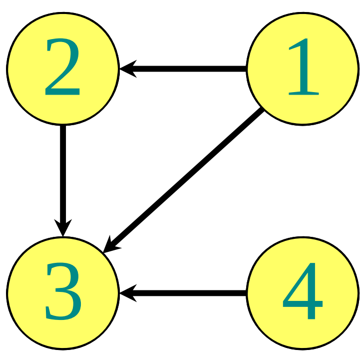
In an ***undirected graph*** $G = (V, E)$, the edge set E consists of *unordered* pairs of vertices.

In either case, we have $|E| = O(|V|^2)$. Moreover, if G is connected, then $|E| \geq |V| - 1$.

Adjacency-matrix representation

The ***adjacency matrix*** of a graph $G = (V, E)$, where $V = \{1, 2, \ldots, n\}$, is the matrix $A[1 \,.\,.\, n, 1 \,.\,.\, n]$ given by

$$A[i, j] = \begin{cases} 1 & \text{if } (i, j) \in E, \\ 0 & \text{if } (i, j) \notin E. \end{cases}$$

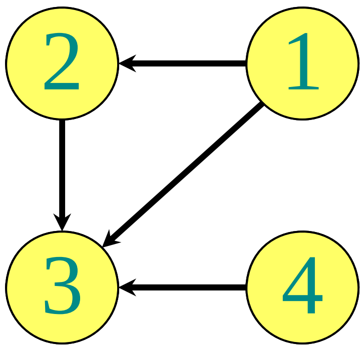


A	1	2	3	4
1	0	1	1	0
2	0	0	1	0
3	0	0	0	0
4	0	0	1	0

$\Theta(|V|^2)$ storage $\Rightarrow$ ***dense*** representation.

Adjacency-list representation

An ***adjacency list*** of a vertex $v \in V$ is the list $Adj[v]$ of vertices adjacent to v.



$Adj[1] = \{2, 3\}$
$Adj[2] = \{3\}$
$Adj[3] = \{\}$
$Adj[4] = \{3\}$

For undirected graphs, $|Adj[v]| = degree(v)$.

For digraphs, $|Adj[v]| = out\text{-}degree(v)$.

Adjacency-list representation

Handshaking Lemma:

Every edge is counted twice

- For undirected graphs:
  $\sum_{v \in V} degree(v) = 2|E|$
- For digraphs:
  $\sum_{v \in V} in\text{-}degree(v) + \sum_{v \in V} out\text{-}degree(v) = 2\,|\,E\,|$

$\Rightarrow$ adjacency lists use $\Theta(|V| + |E|)$ storage
$\Rightarrow$ a ***sparse*** representation
$\Rightarrow$ We usually use this representation, unless stated otherwise

Graph Traversal

Let $G=(V,E)$ be a (directed or undirected) graph, given in adjacency list representation.

$|V| = n$, $|E| = m$

A graph traversal visits every vertex:

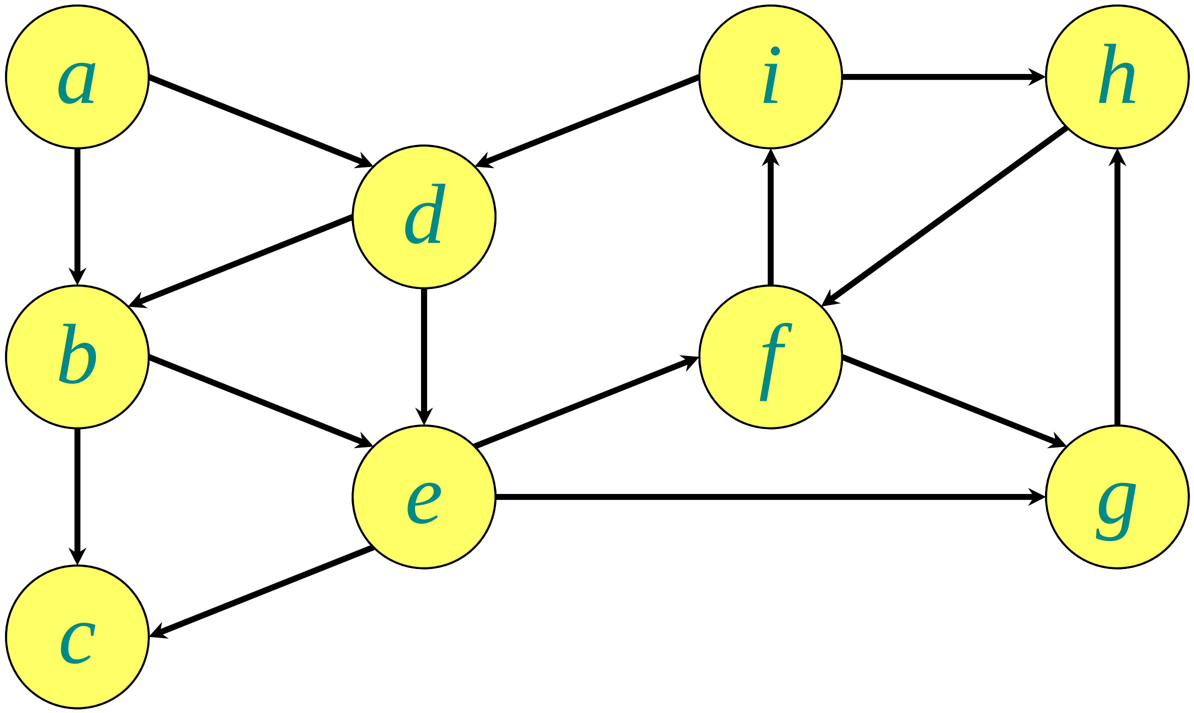
- Breadth-first search (BFS)
- Depth-first search (DFS)

Breadth-First Search (BFS)

```
BFS(G=(V,E))
    Mark all vertices in G as “unvisited” // time=0
    Initialize empty queue Q
    for each vertex v ∈ V do
        if v is unvisited
            visit v // time++
            Q.enqueue(v)
            BFS_iter(G)
```

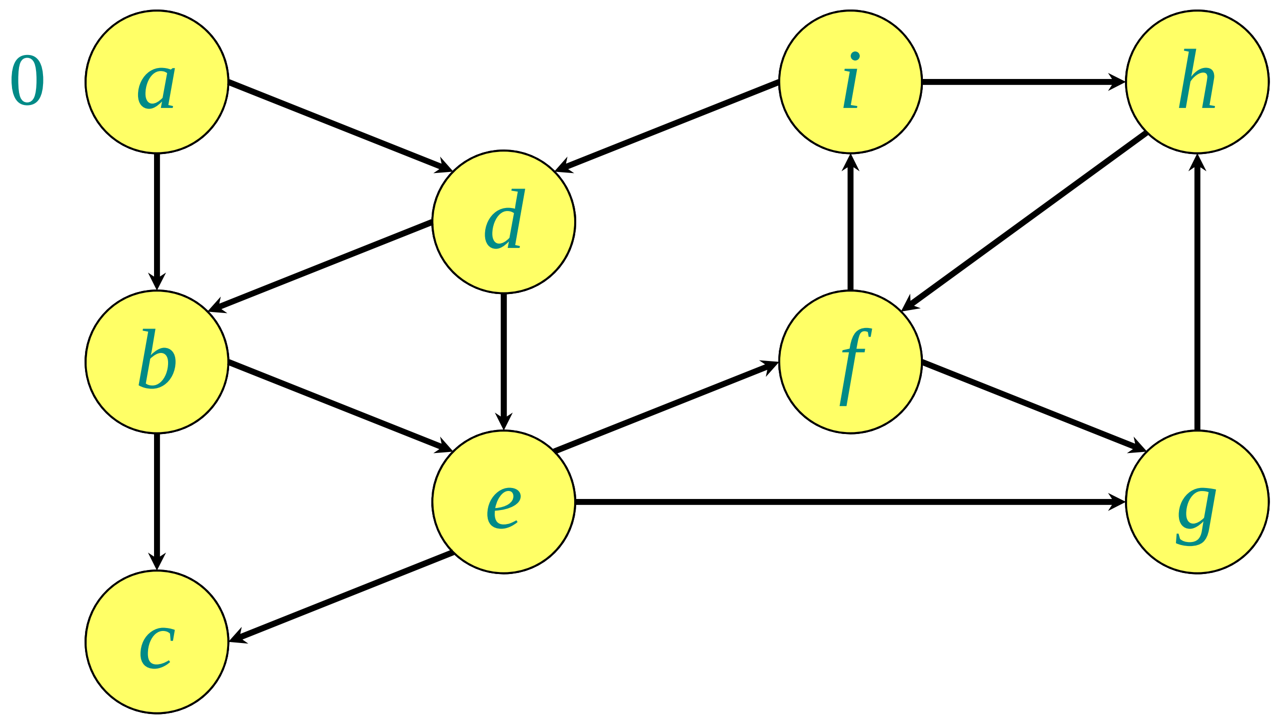
```
BFS_iter(G)
    while Q is non-empty do
        v = Q.dequeue()
        for each w adjacent to v do
            if w is unvisited
                visit w   // time++
                Add edge (v,w) to T
                Q.enqueue(w)
```

Example of breadth-first search



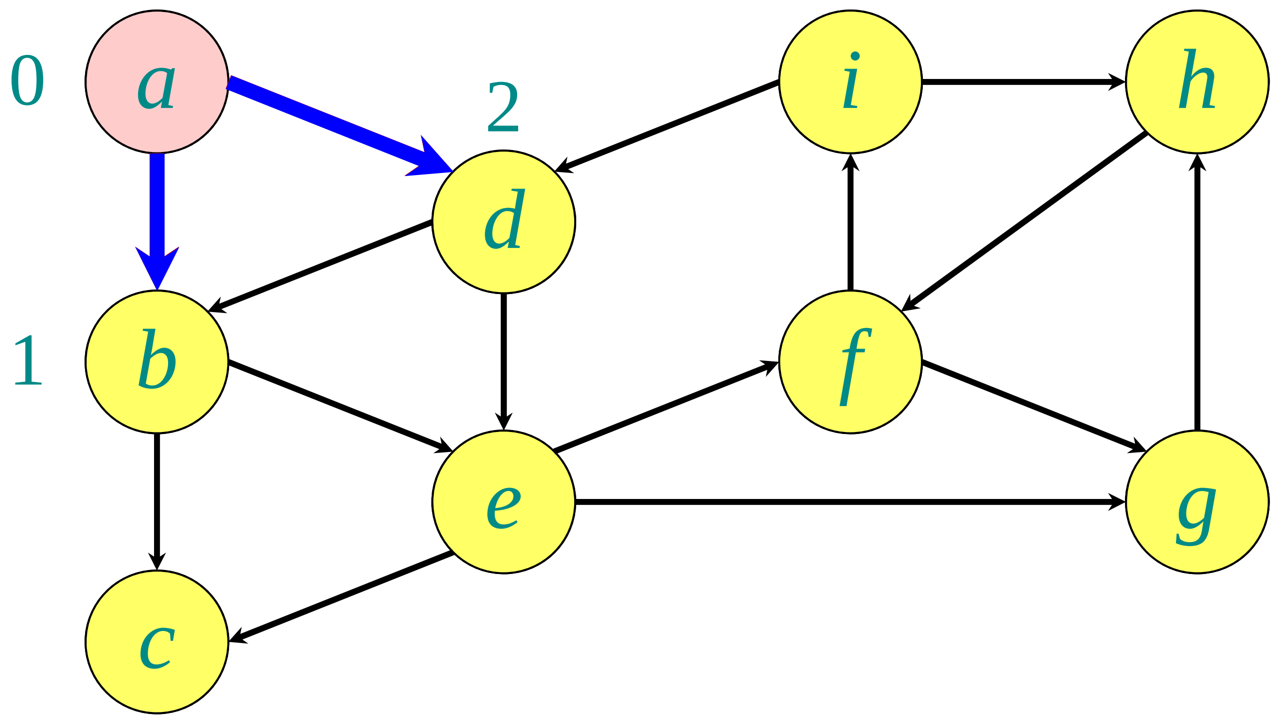
Q:

Example of breadth-first search



$$Q: \overset{0}{a}$$

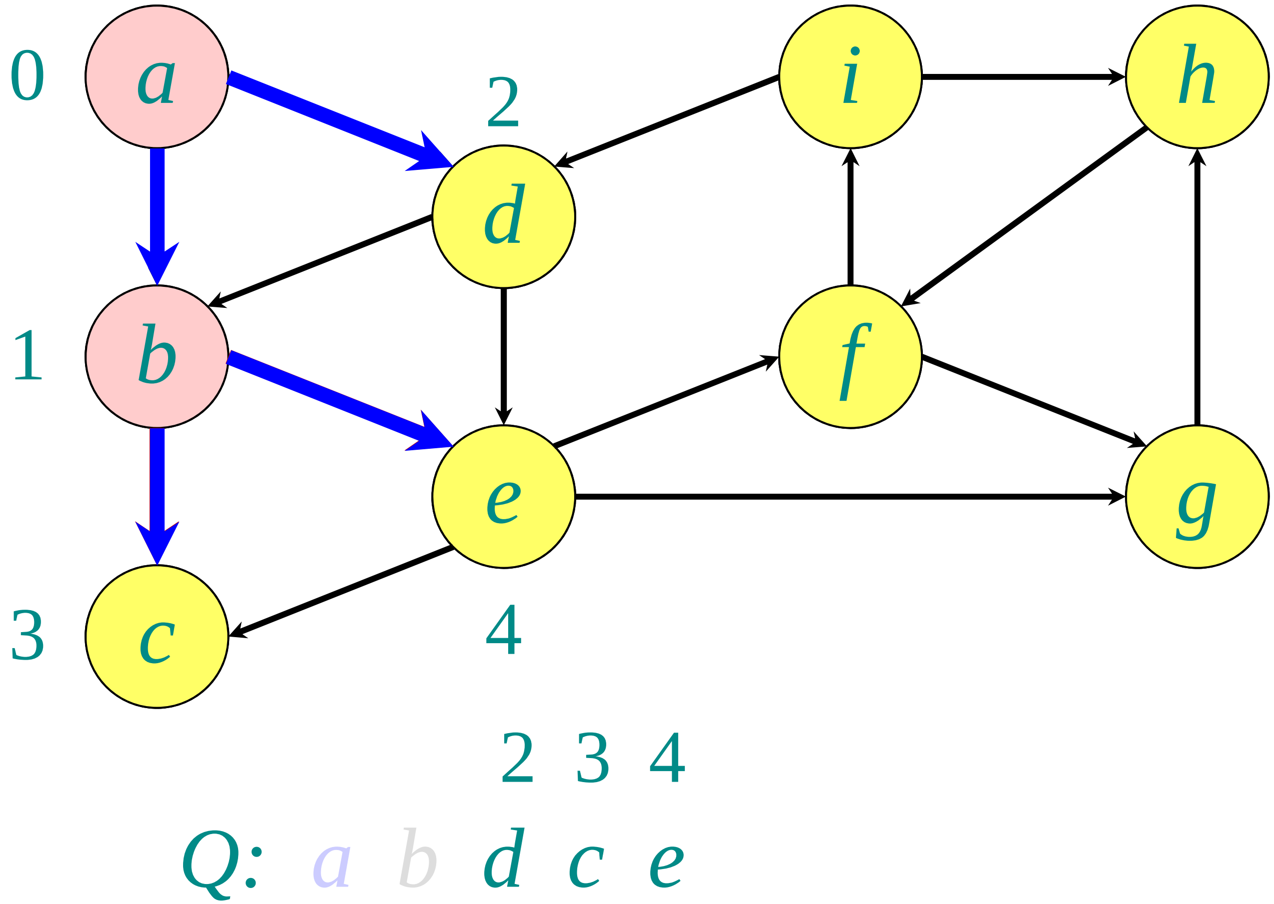
Example of breadth-first search



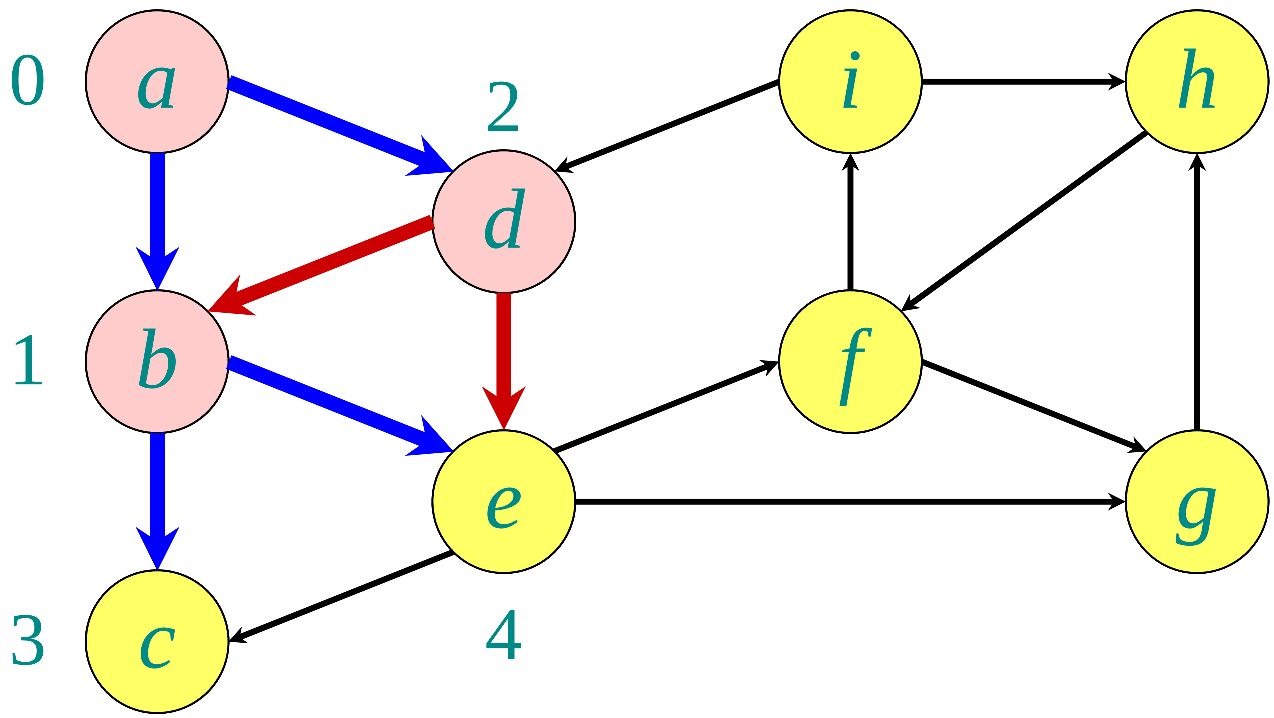
1 2

Q: a b d

Example of breadth-first search



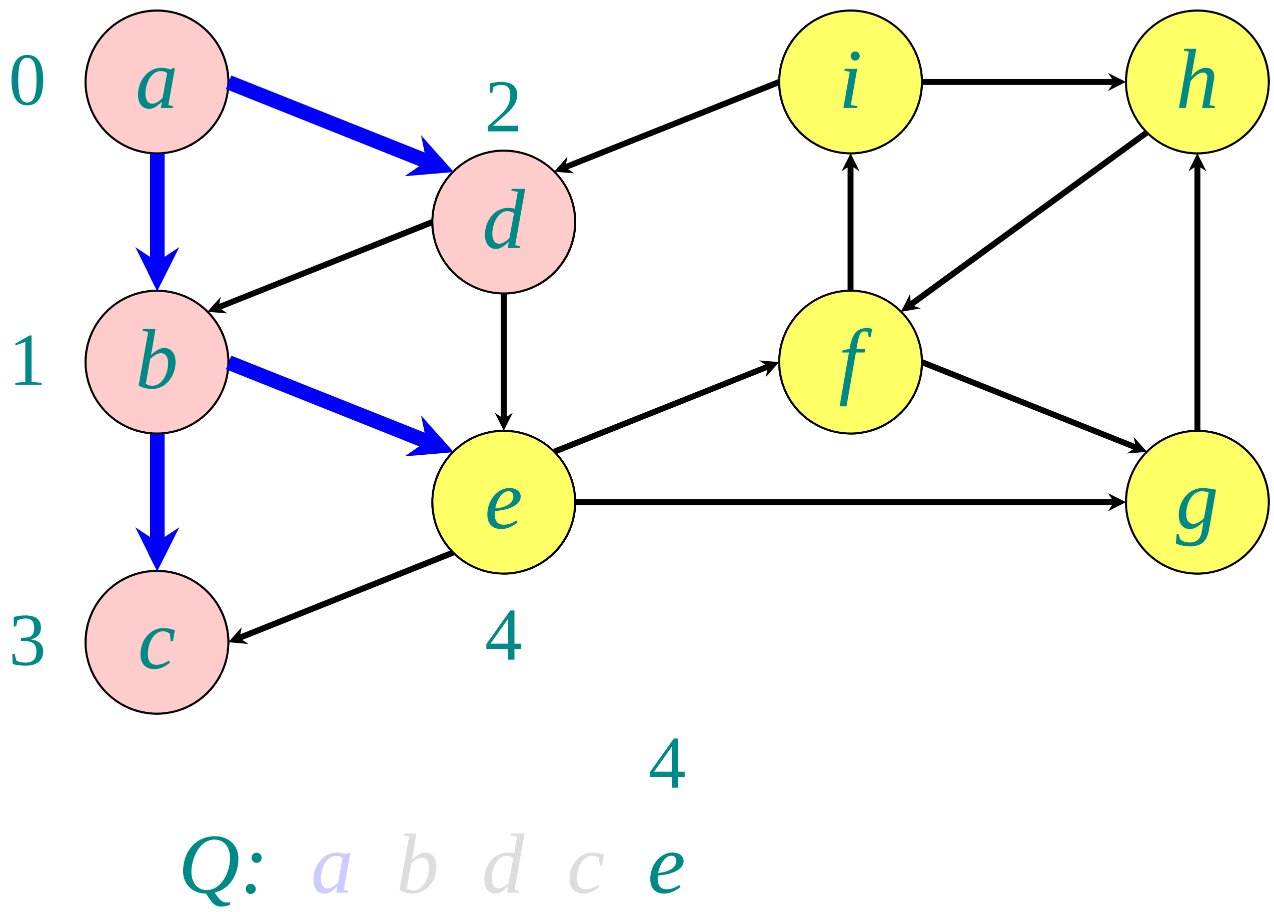
Example of breadth-first search



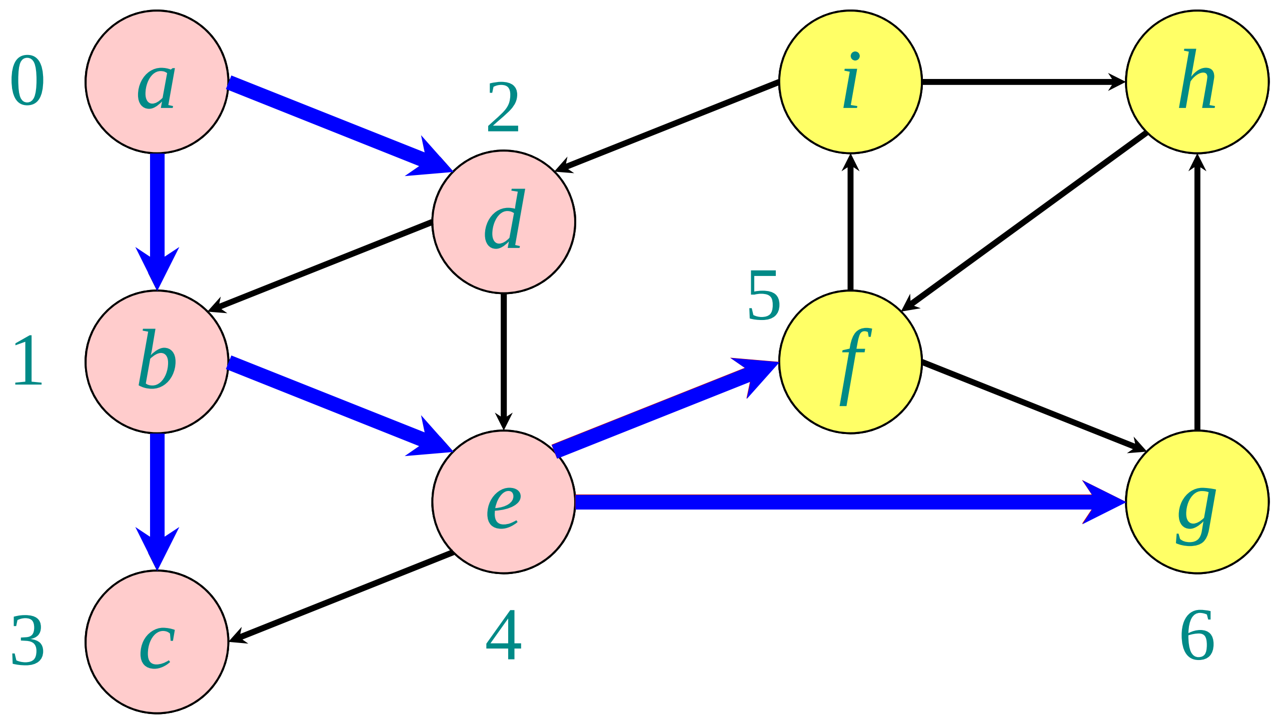
3 4

Q: *a* *b* *d* *c* *e*

Example of breadth-first search



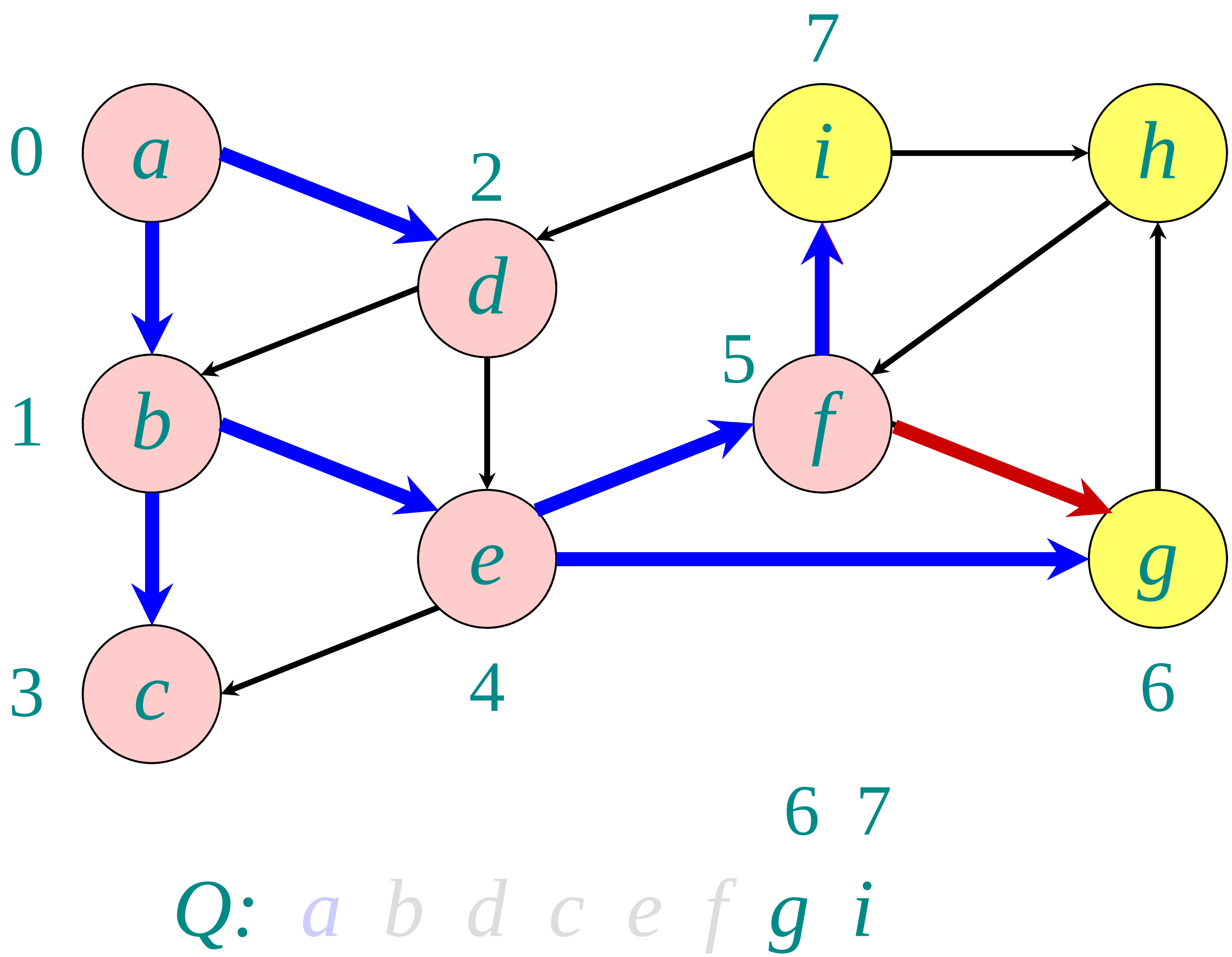
Example of breadth-first search



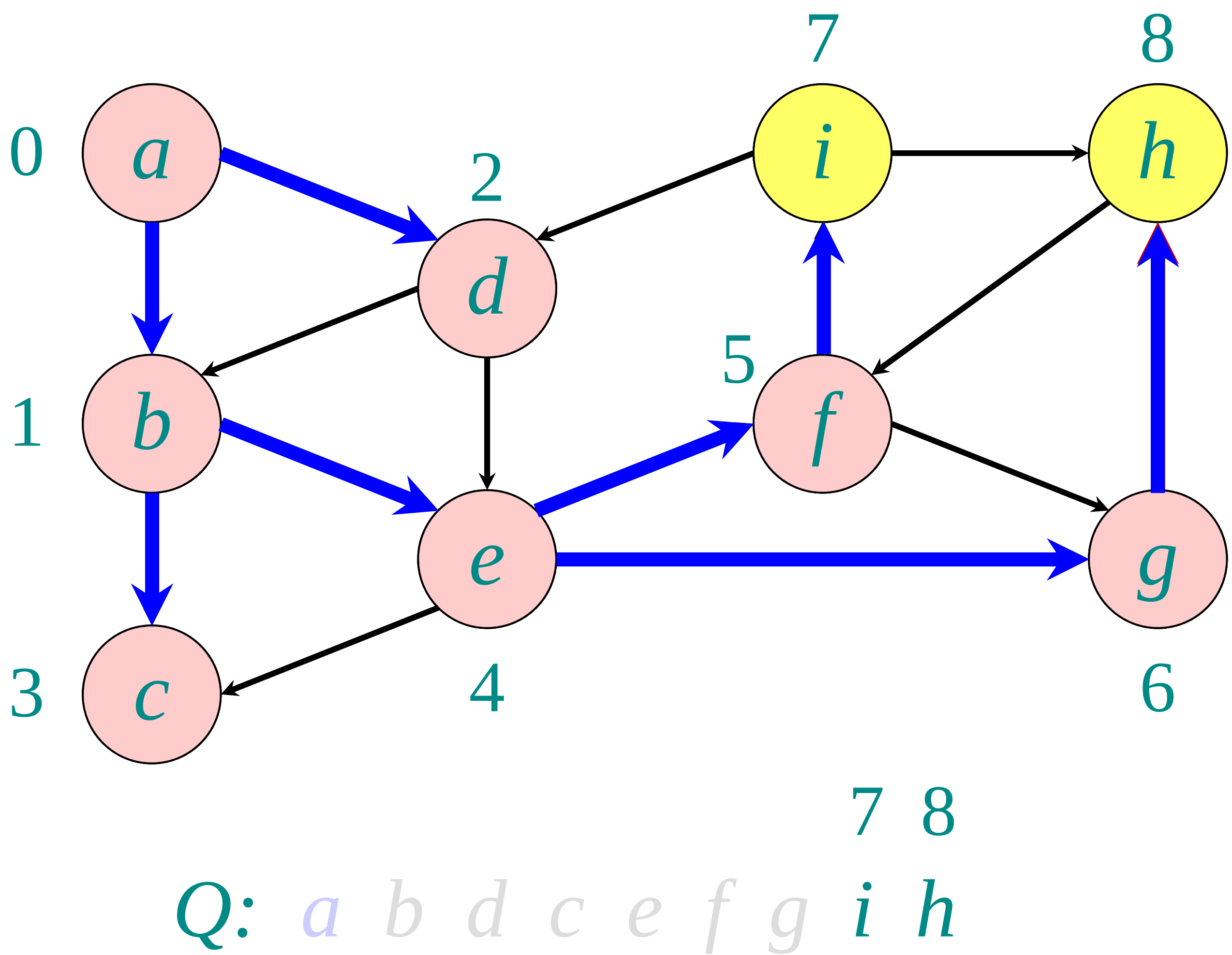
5 6

Q: *a b d c e f g*

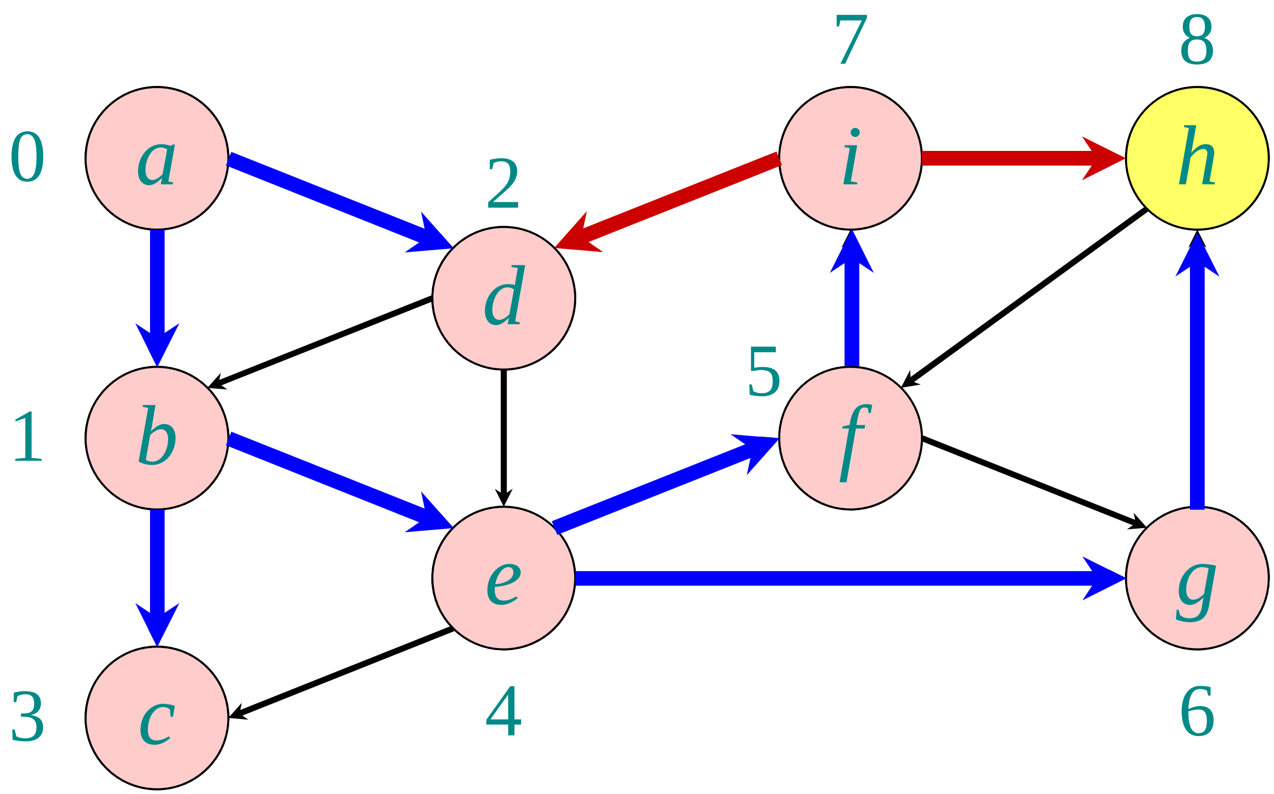
Example of breadth-first search



Example of breadth-first search



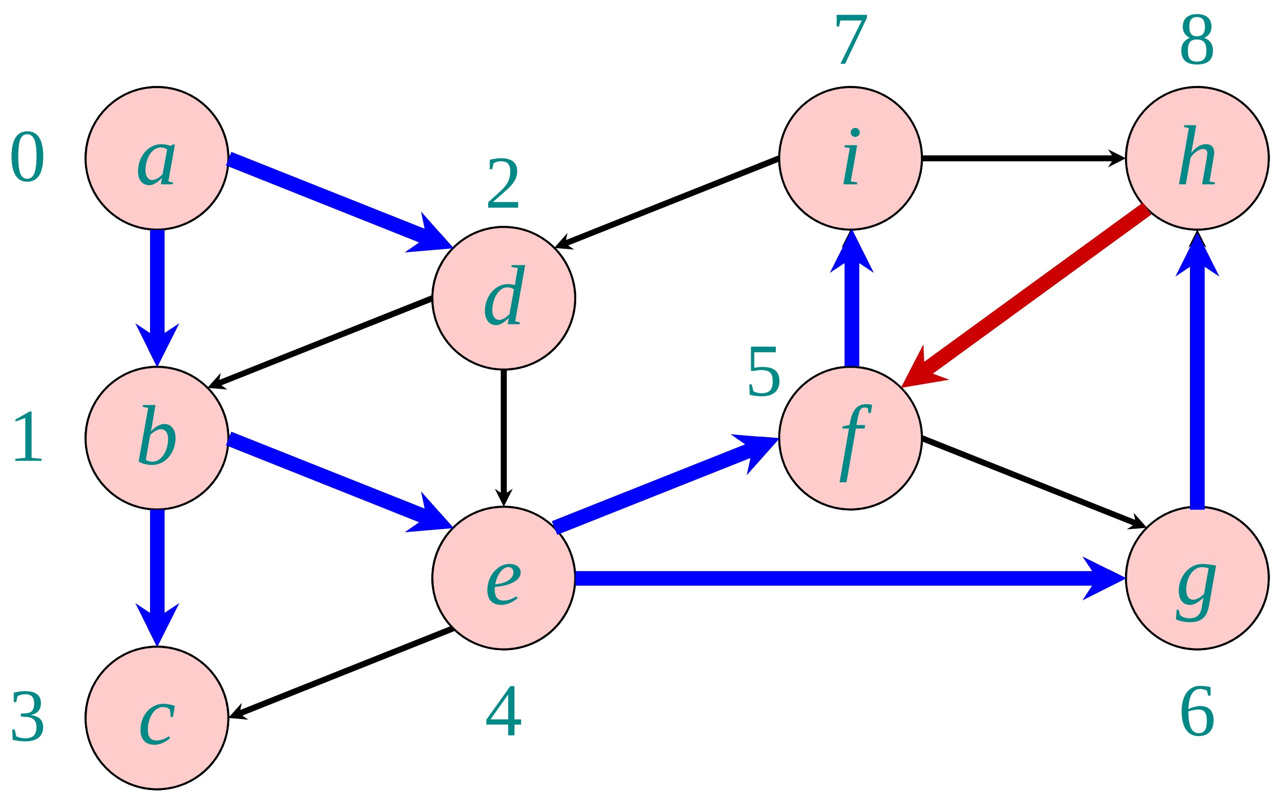
Example of breadth-first search



8

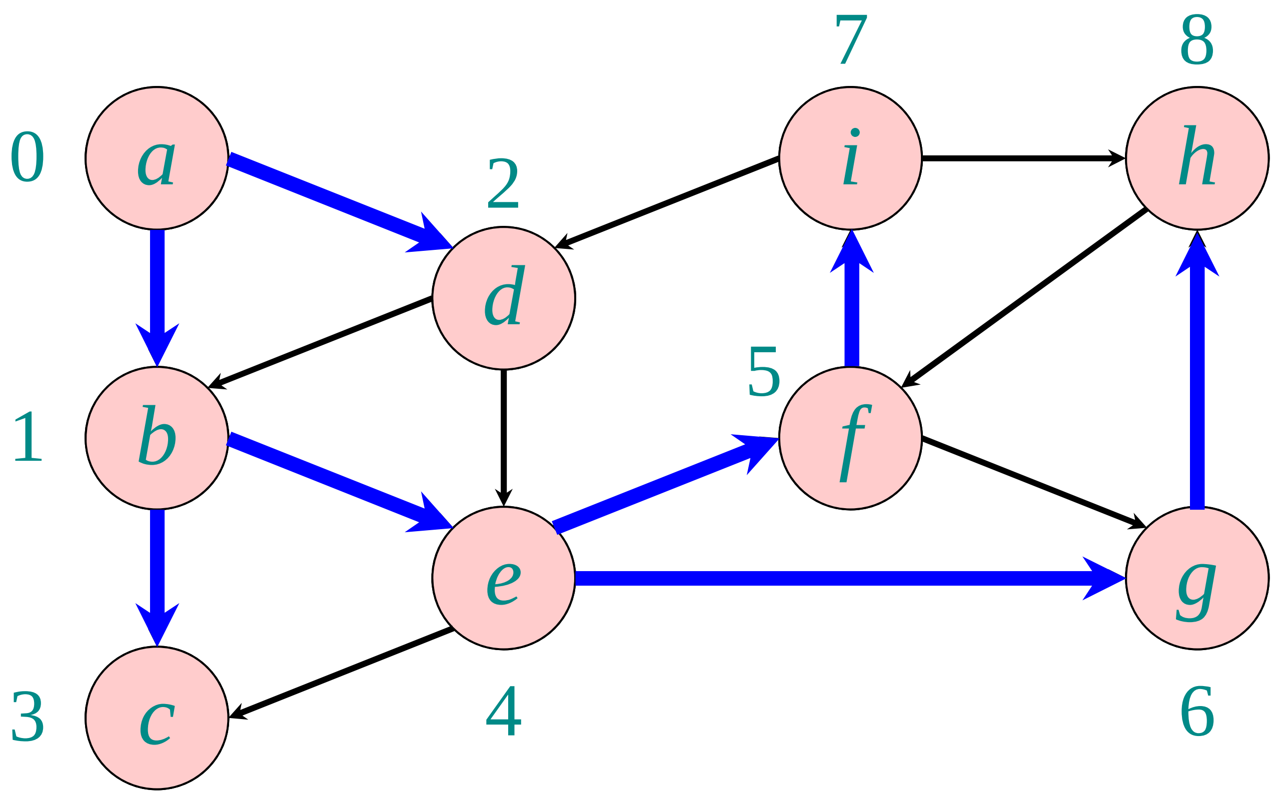
Q: a b d c e f g i h

Example of breadth-first search



Q: *a b d c e f g i h*

Example of breadth-first search



Q: *a b d c e f g i h*

Breadth-First Search (BFS)

BFS(G=(V,E))
  Mark all vertices in G as "unvisited" // time=0 — O(n)
  Initialize empty queue Q — O(1)
  for each vertex $v \in V$ **do**
    if v is unvisited
      visit v // time++
      Q.enqueue(v)
      BFS_iter(G)

O(n) without BFS_iter

BFS_iter(G)
  while Q is non-empty **do**
    v = Q.dequeue()
    for each w adjacent to v **do**
      if w is unvisited
        visit w // time++
        Add edge (v,w) to T
        Q.enqueue(w)

O($deg(v)$)

O(m)

BFS runtime

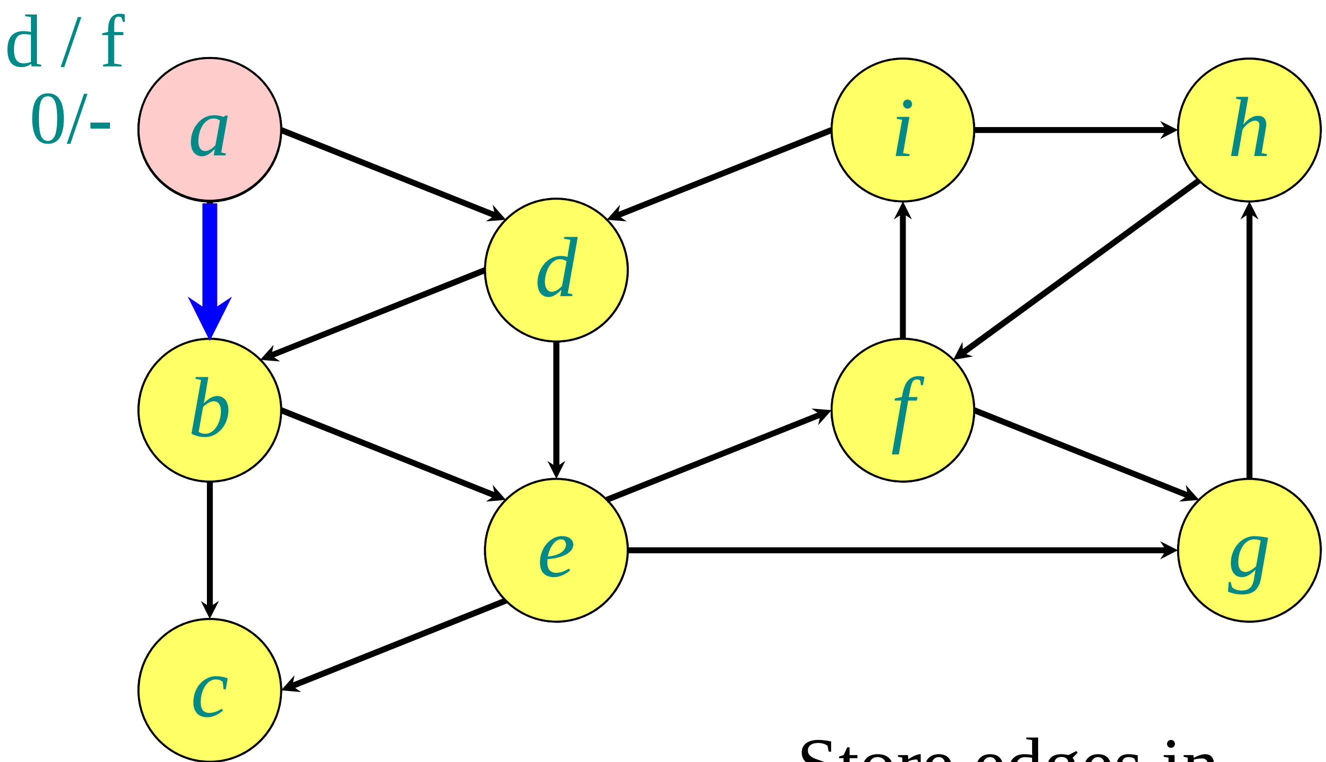
• Each vertex is marked as unvisited in the beginning $\Rightarrow$ $O(n)$ time
• Each vertex is marked at most once, enqueued at most once, and therefore dequeued at most once
• The time to process a vertex is proportional to the size of its adjacency list (its degree), since the graph is given in adjacency list representation
$\Rightarrow$ $O(m)$ time
• Total runtime is $O(n+m) = O(|V| + |E|)$

Depth-First Search (DFS)

```
DFS(G=(V,E))
    Mark all vertices in G as "unvisited"  // time=0
    for each vertex v ∈ V do
        if v is unvisited
            DFS_rec(G,v)
```

```
DFS_rec(G, v)
    mark v as "visited" // d[v]=++time
    for each w adjacent to v do
        if w is unvisited
            Add edge (v,w) to tree T
            DFS_rec(G,w)
    mark v as "finished" // f[v]=++time
```

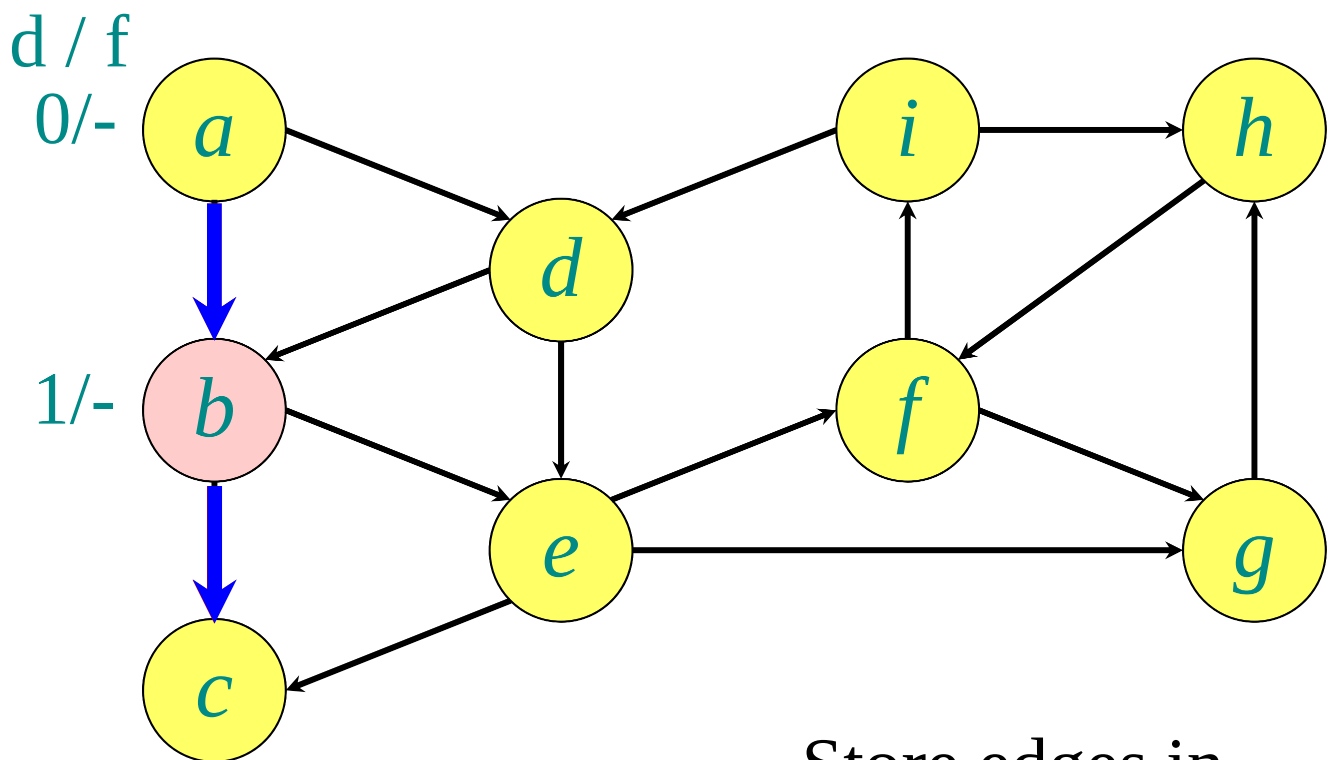
Example of depth-first search



Store edges in
predecessor array

π:	a	b	c	d	e	f	g	h	i
	-	a							

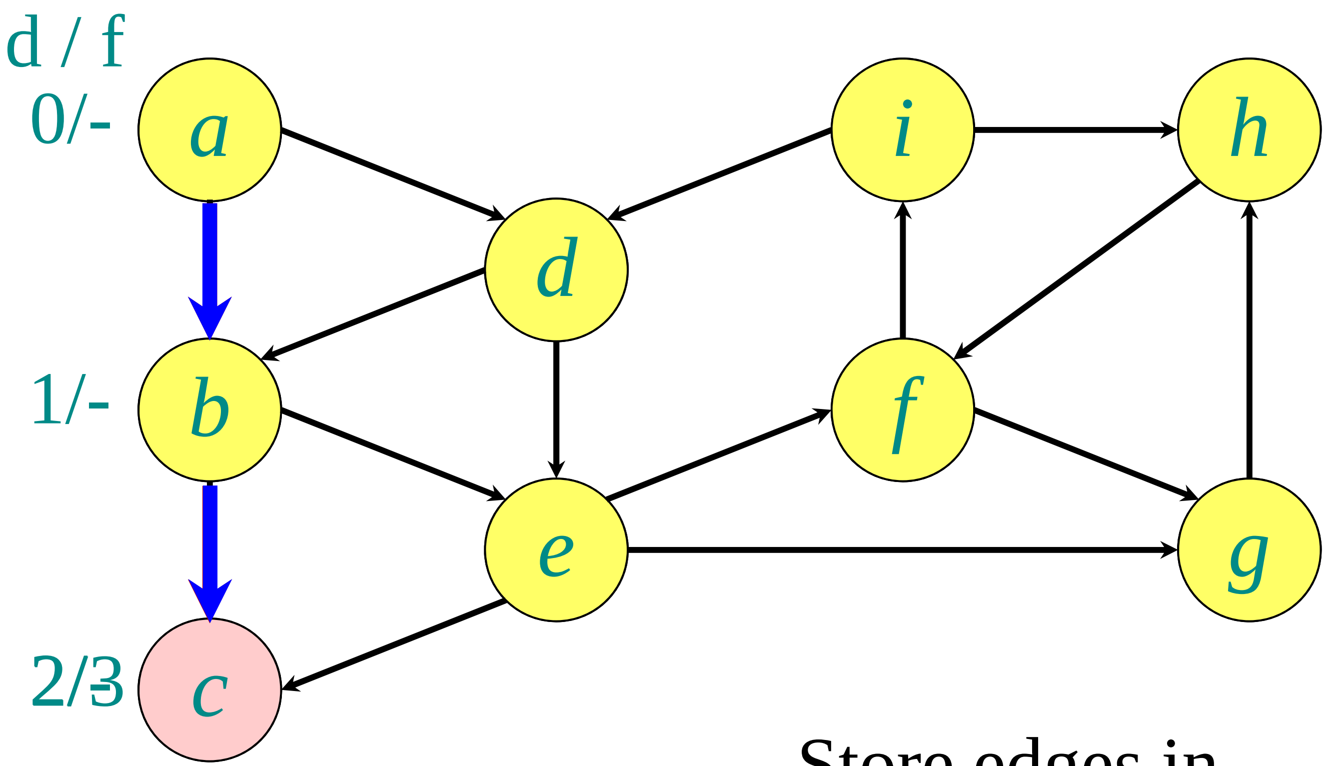
Example of depth-first search



Store edges in
predecessor array

π:	a	b	c	d	e	f	g	h	i
	-	a	b						

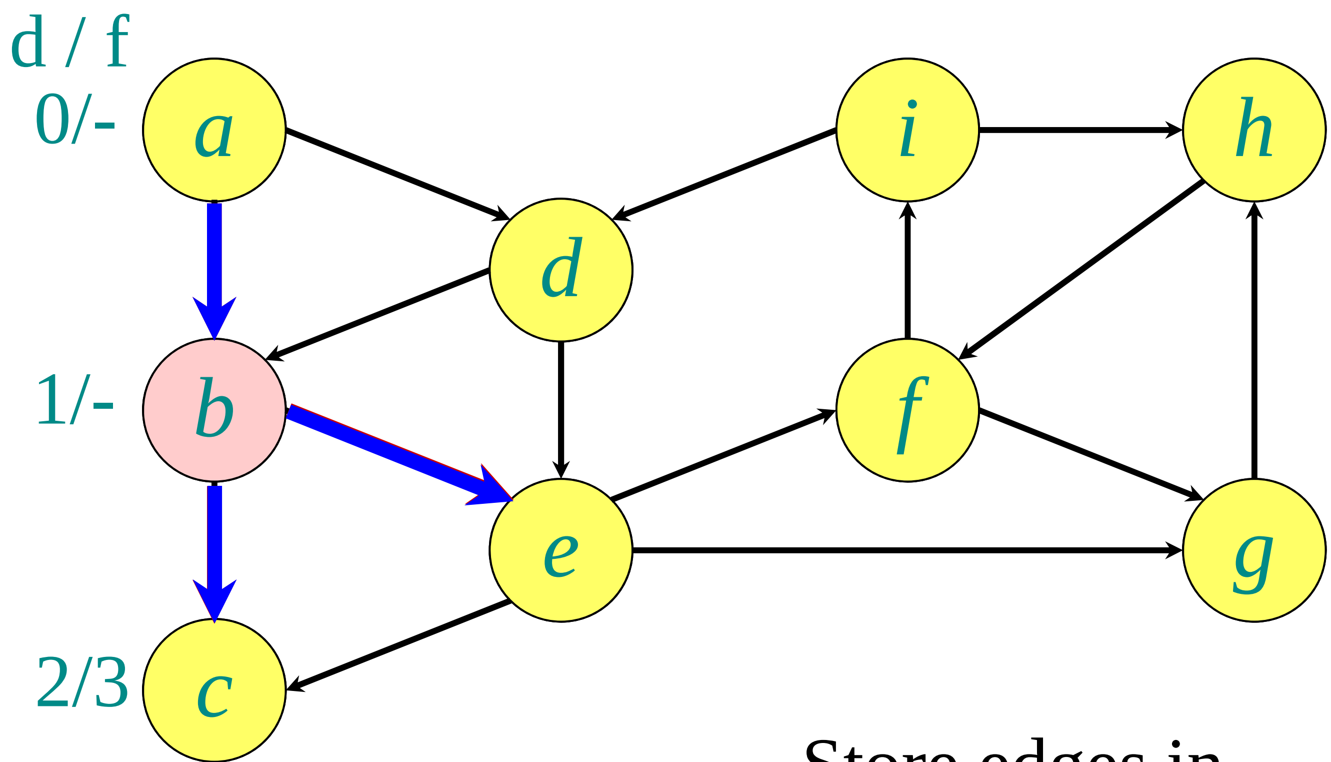
Example of depth-first search



Store edges in predecessor array

π:	a	b	c	d	e	f	g	h	i
	-	a	b						

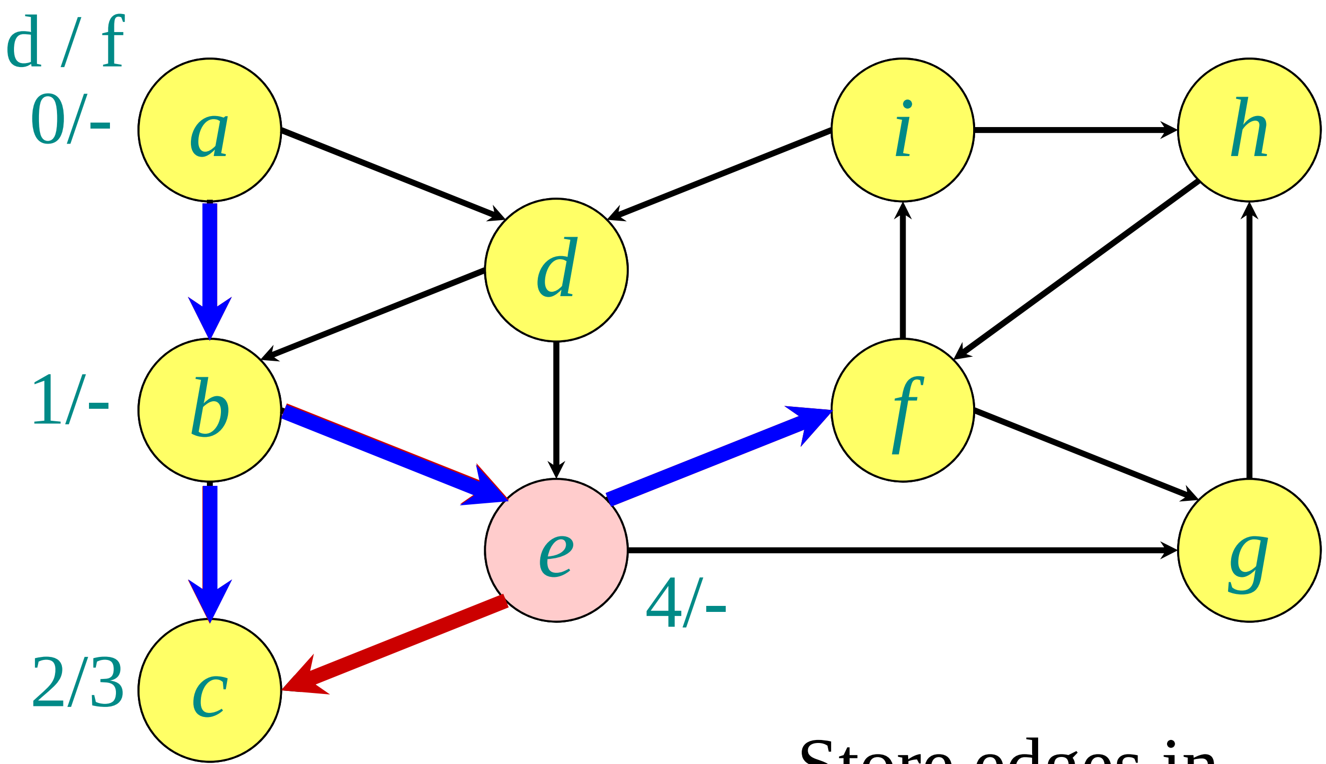
Example of depth-first search



Store edges in predecessor array

π:	a	b	c	d	e	f	g	h	i
	-	a	b		b				

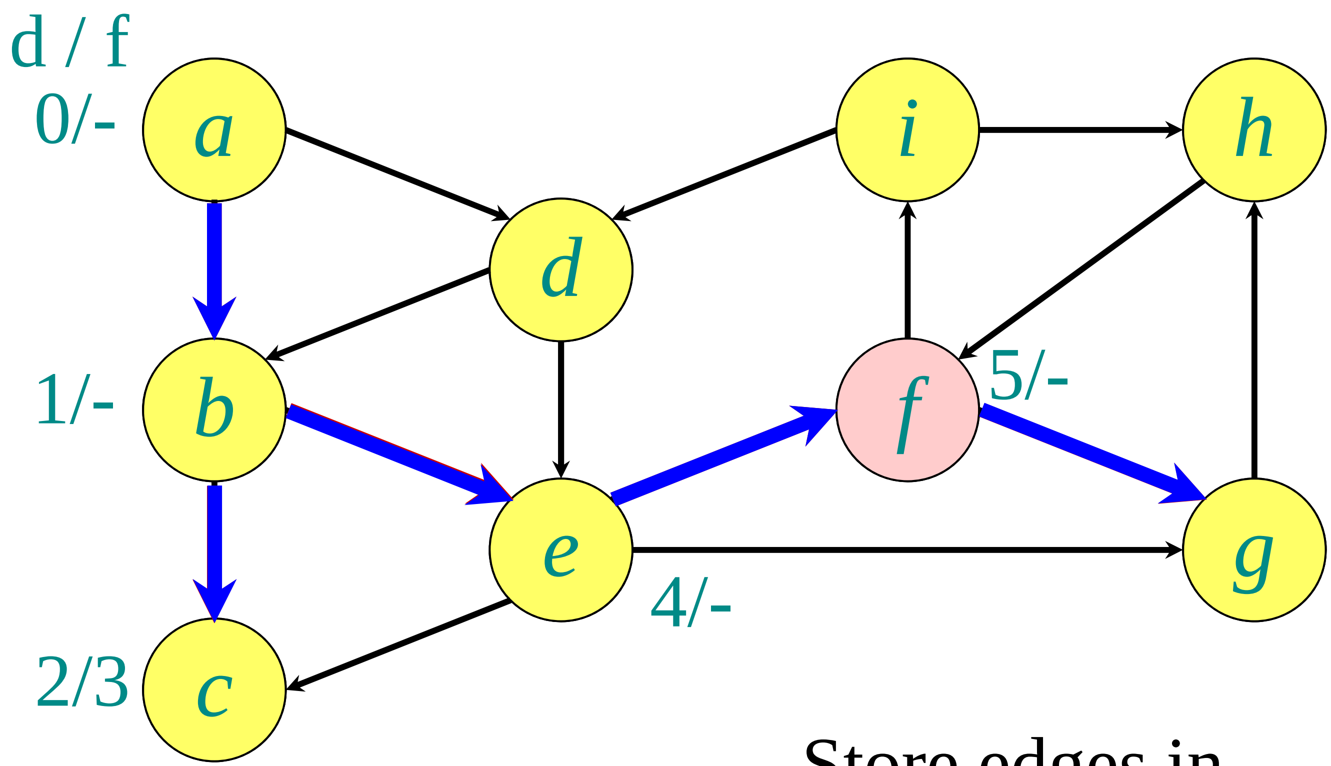
Example of depth-first search



Store edges in
predecessor array

π:	a	b	c	d	e	f	g	h	i
	-	a	b		b	e			

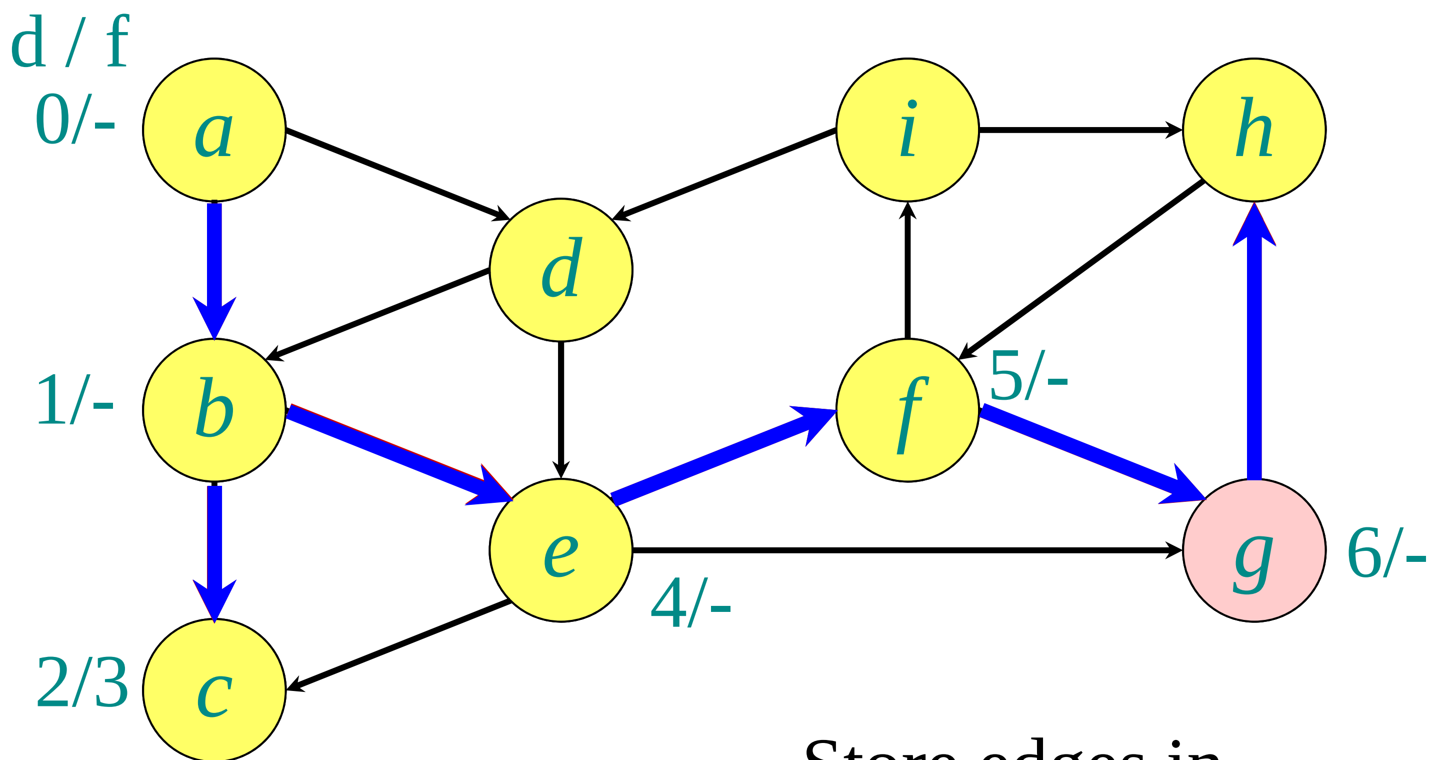
Example of depth-first search



Store edges in
predecessor array

π:	a	b	c	d	e	f	g	h	i
	-	a	b		b	e	f		

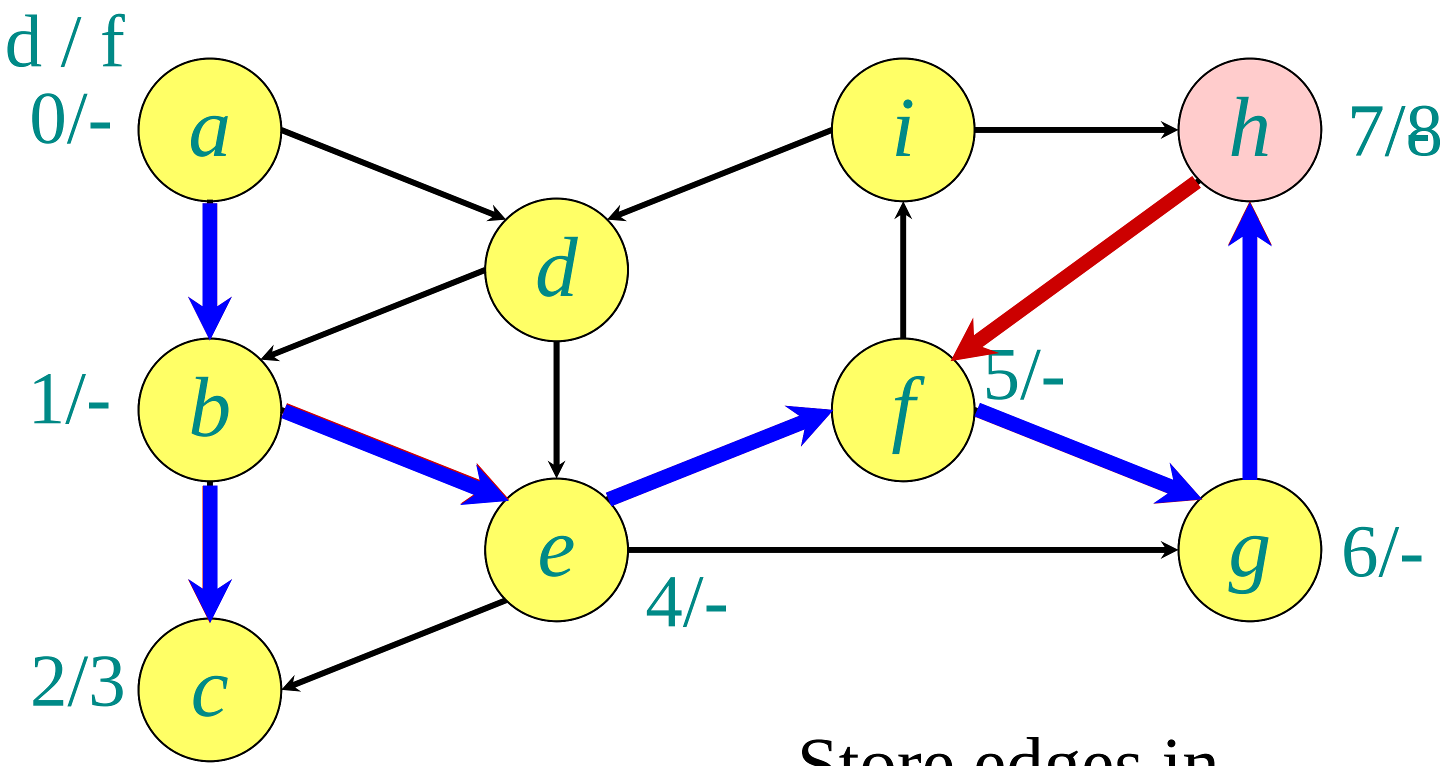
Example of depth-first search



Store edges in
predecessor array

π:	a	b	c	d	e	f	g	h	i
	-	a	b		b	e	f	g	

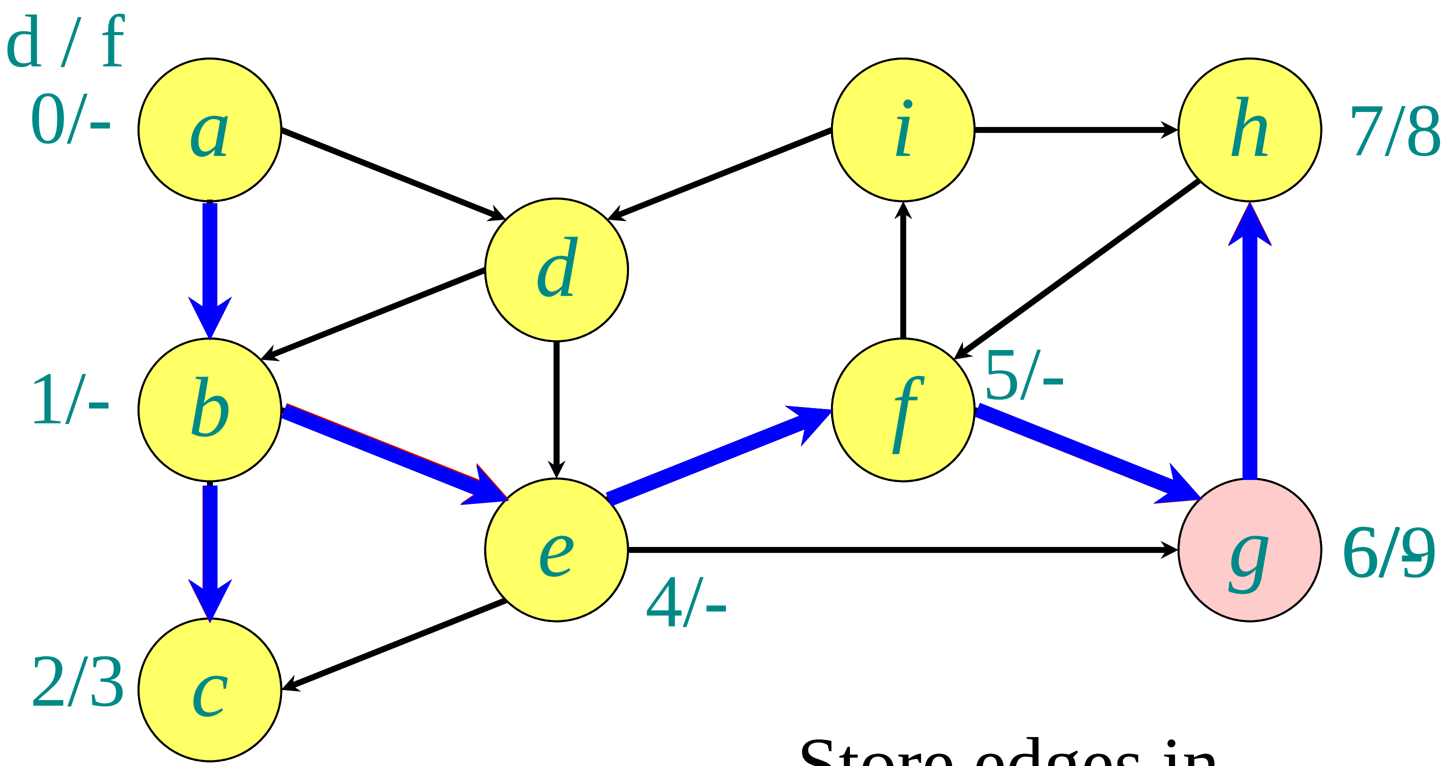
Example of depth-first search



Store edges in
predecessor array

π:	a	b	c	d	e	f	g	h	i
	-	a	b		b	e	f	g	

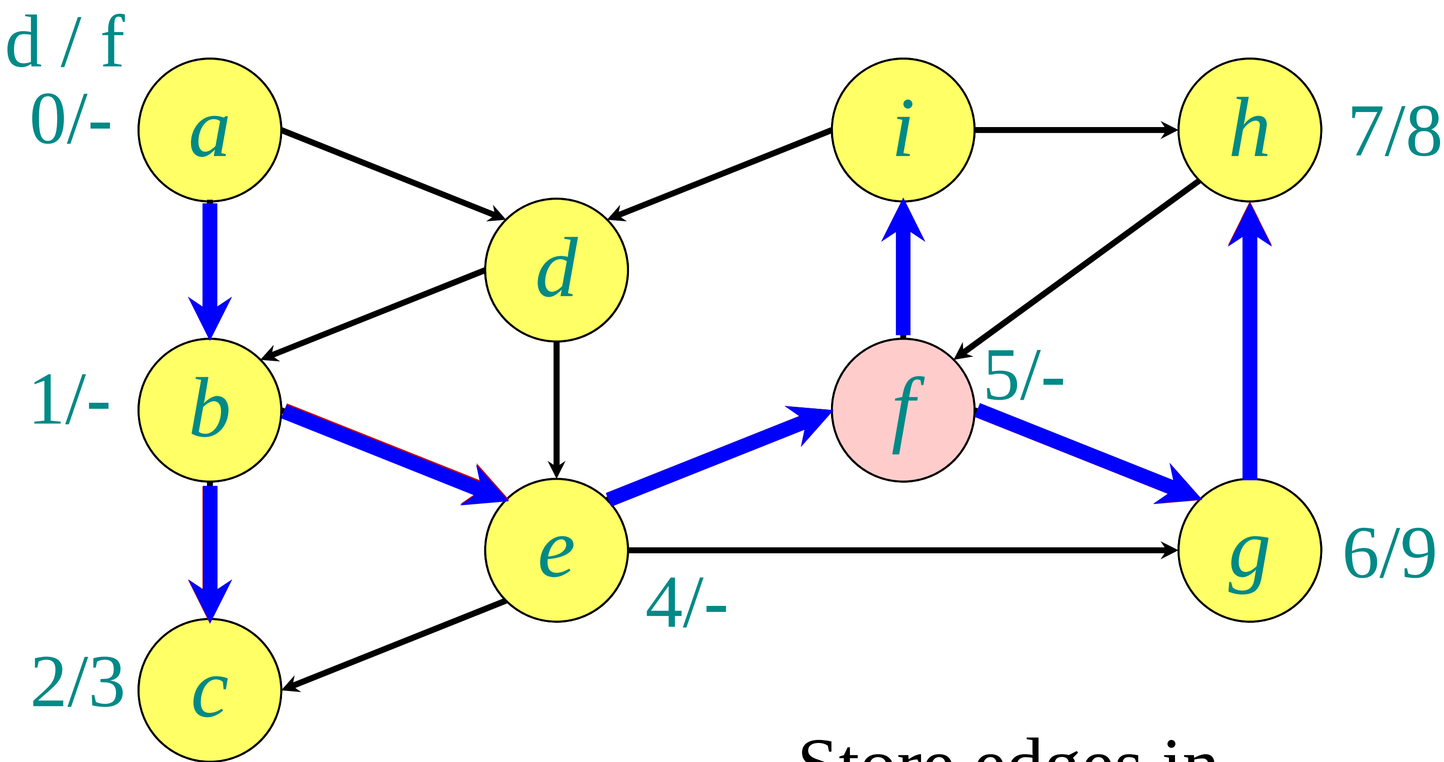
Example of depth-first search



Store edges in
predecessor array

π:	a	b	c	d	e	f	g	h	i
	-	a	b		b	e	f	g	

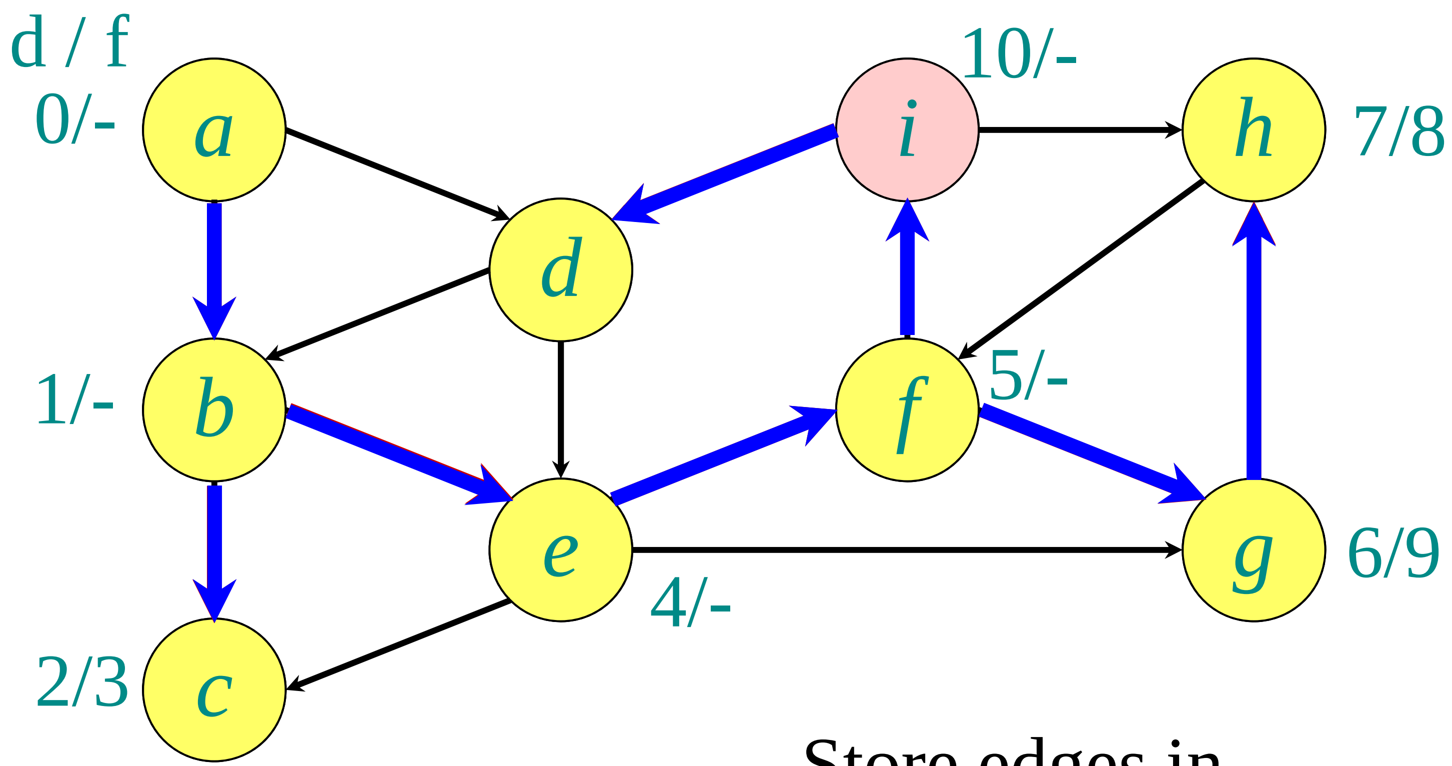
Example of depth-first search



π:	a	b	c	d	e	f	g	h	i
	-	a	b		b	e	f	g	f

Store edges in predecessor array

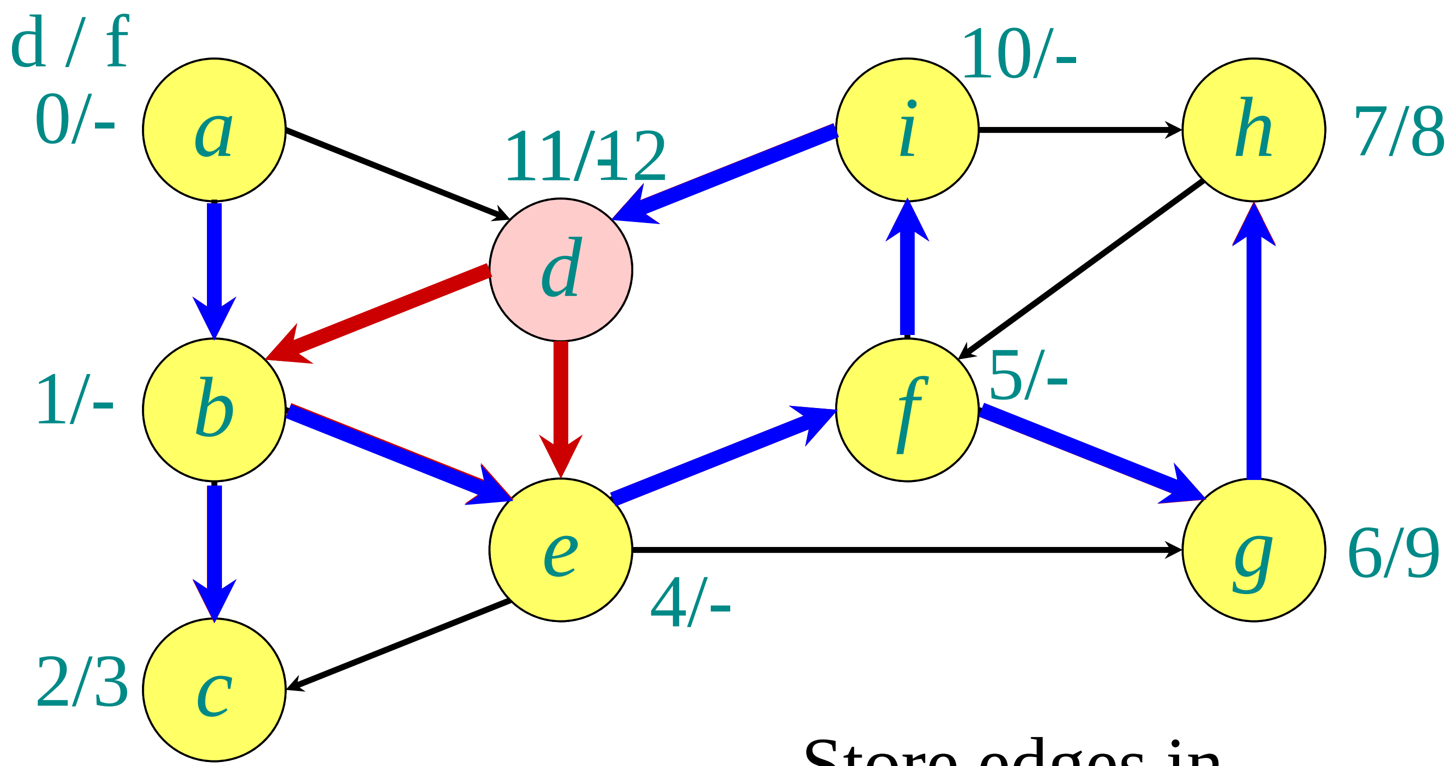
Example of depth-first search



π:	a	b	c	d	e	f	g	h	i
	-	a	b	i	b	e	f	g	f

Store edges in predecessor array

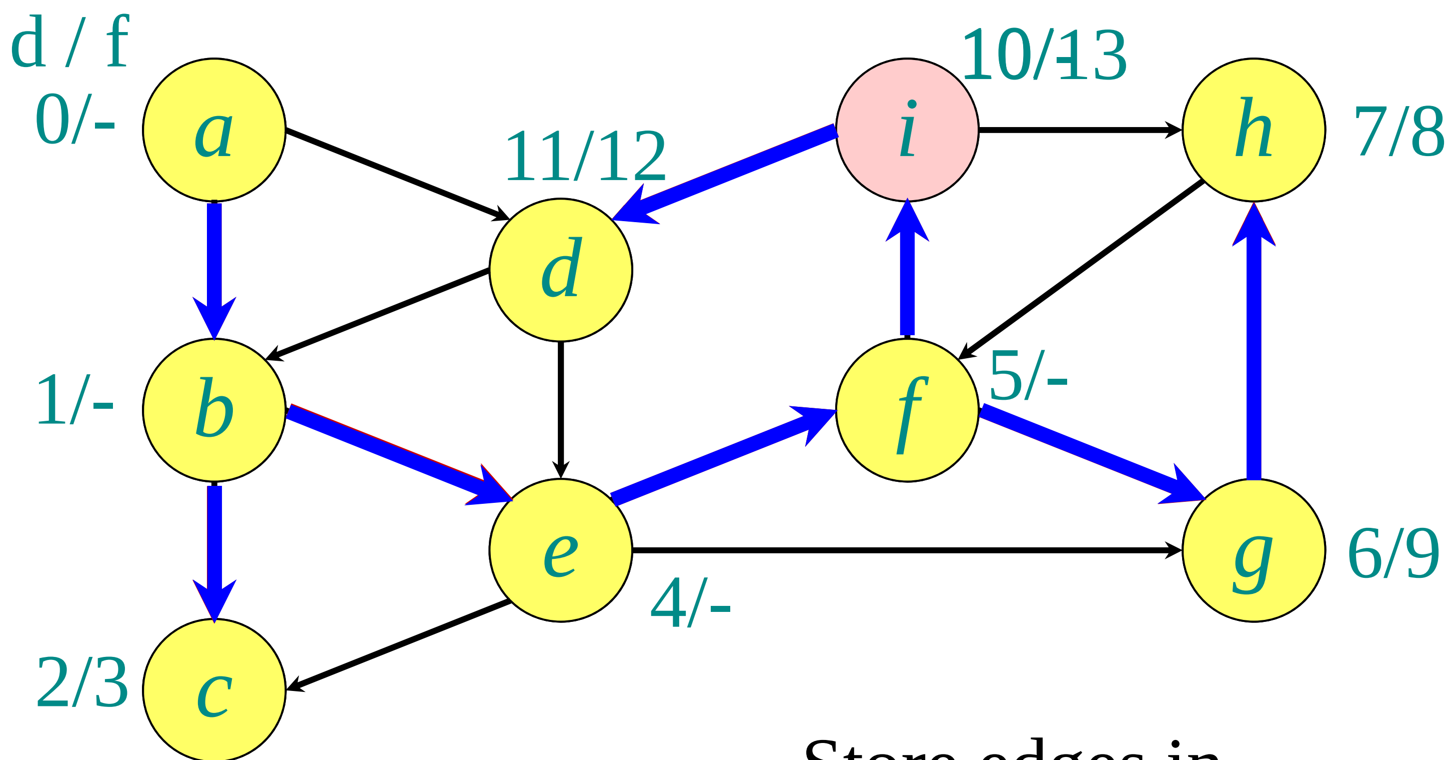
Example of depth-first search



π:	a	b	c	d	e	f	g	h	i
	-	a	b	i	b	e	f	g	f

Store edges in predecessor array

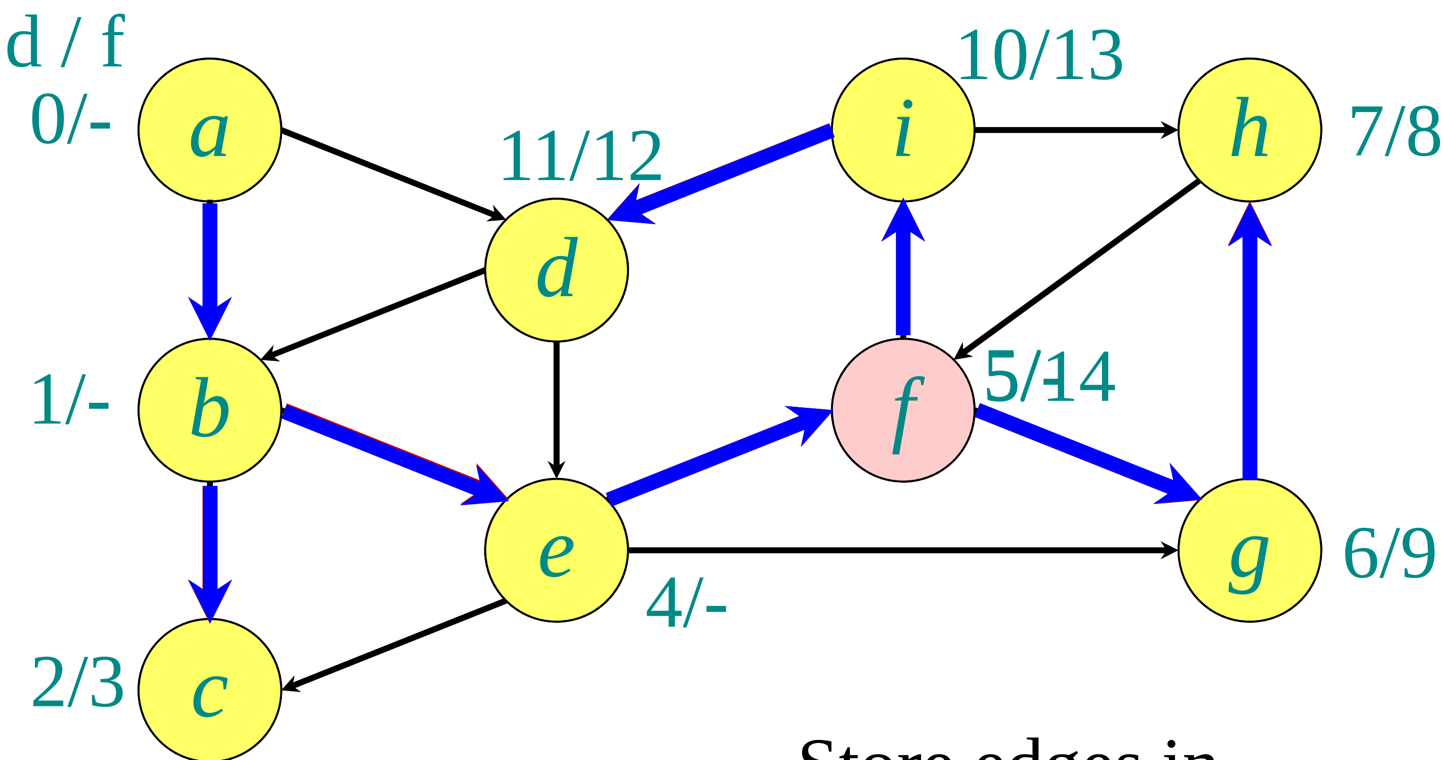
Example of depth-first search



Store edges in predecessor array

π:	a	b	c	d	e	f	g	h	i
	-	a	b	i	b	e	f	g	f

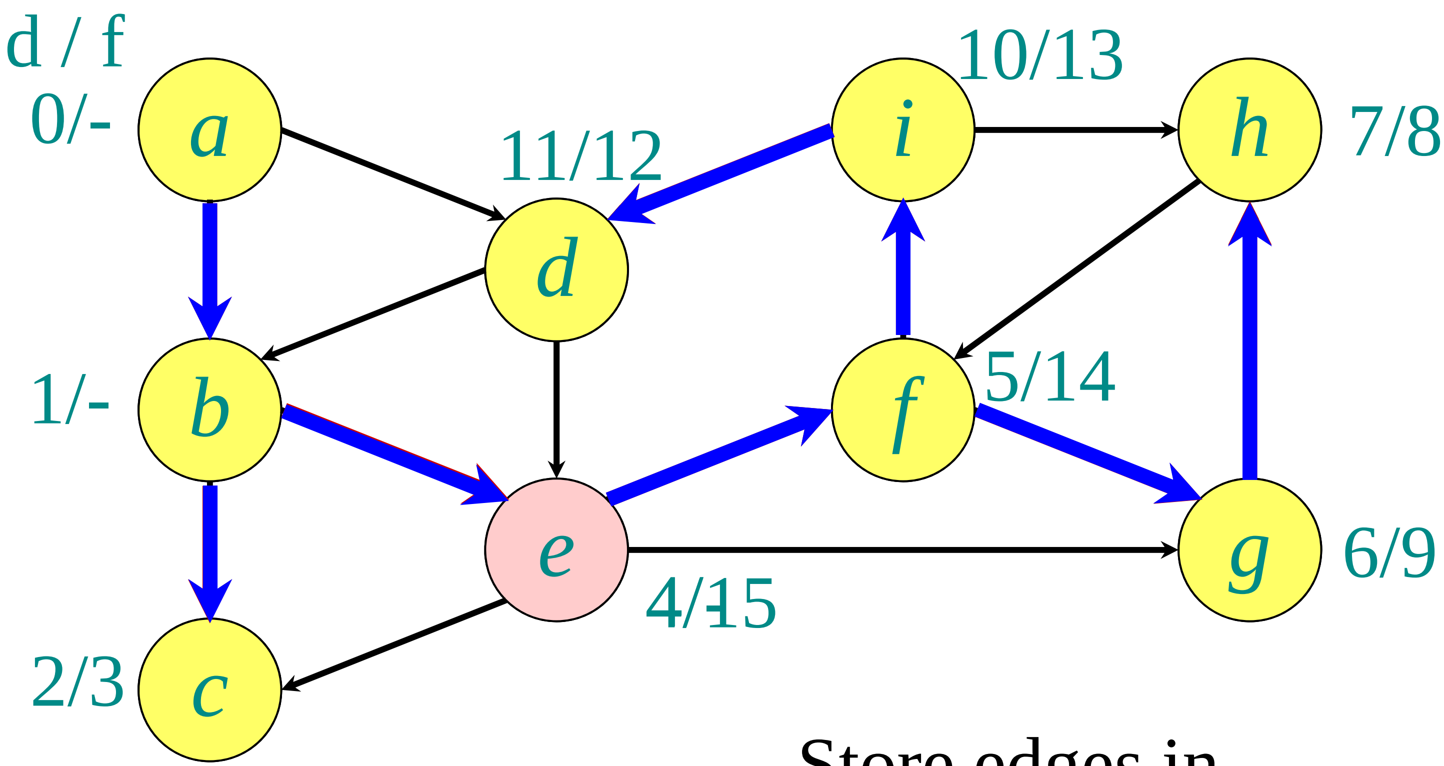
Example of depth-first search



π:	a	b	c	d	e	f	g	h	i
	-	a	b	i	b	e	f	g	f

Store edges in predecessor array

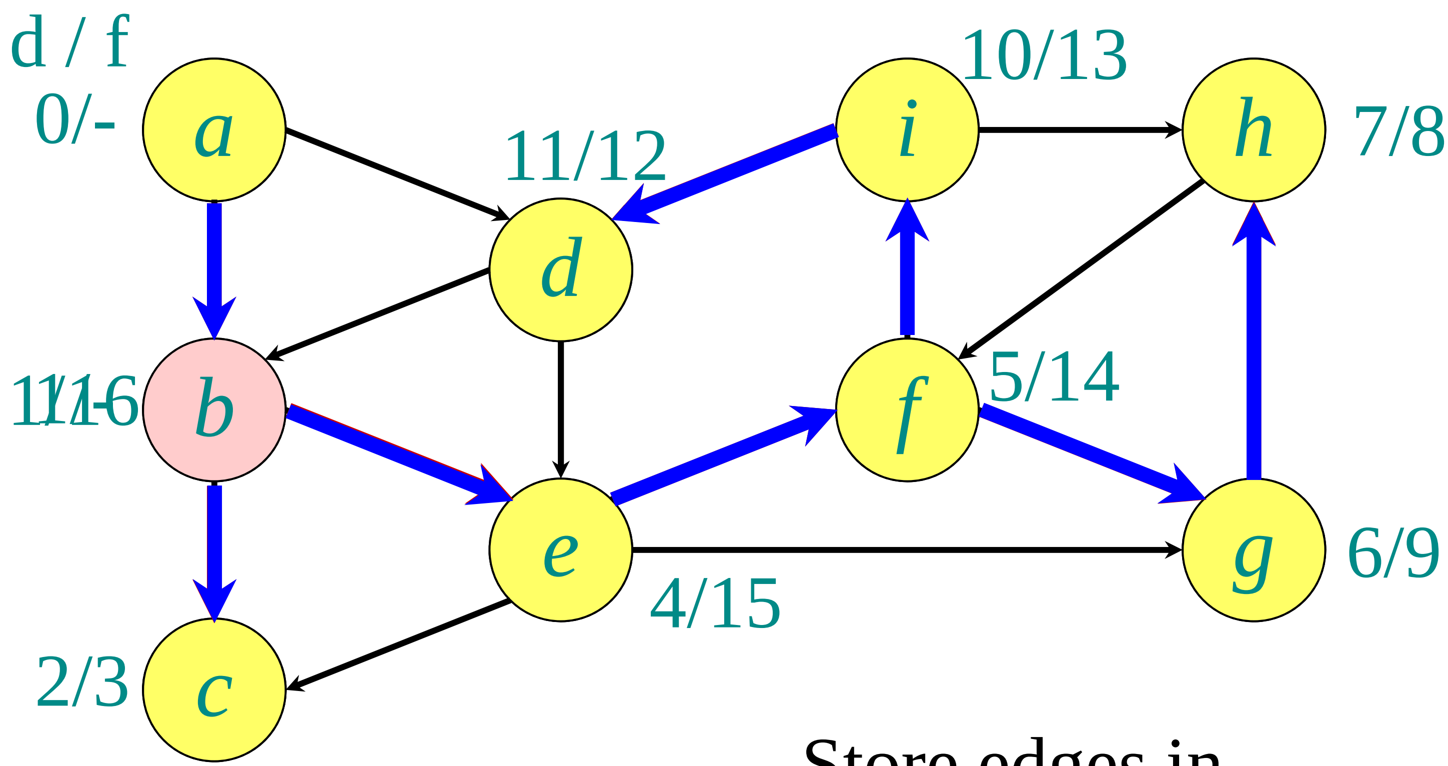
Example of depth-first search



Store edges in
predecessor array

π:	a	b	c	d	e	f	g	h	i
	-	a	b	i	b	e	f	g	f

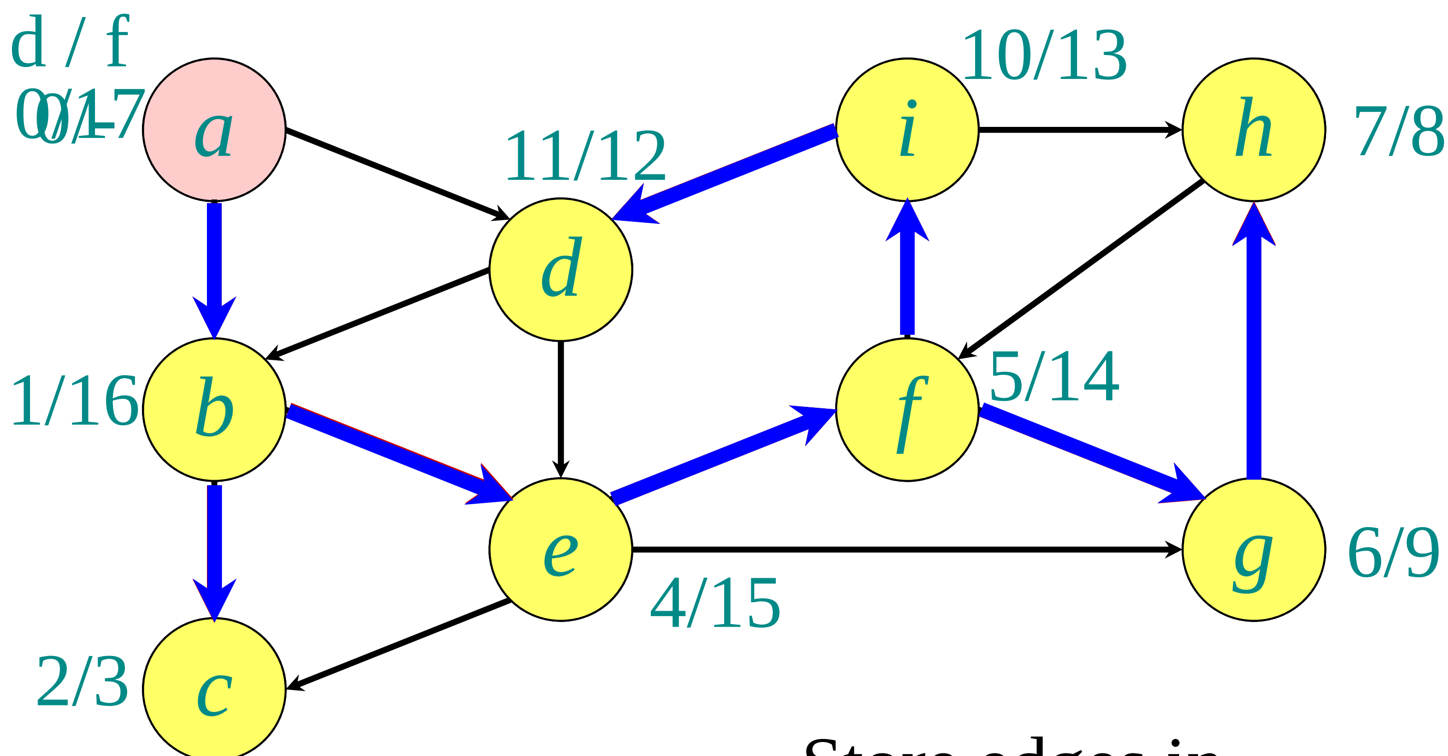
Example of depth-first search



Store edges in
predecessor array

π:	a	b	c	d	e	f	g	h	i
	-	a	b	i	b	e	f	g	f

Example of depth-first search



Store edges in
predecessor array

π:	a	b	c	d	e	f	g	h	i
	-	a	b	i	b	e	f	g	f

Depth-First Search (DFS)

$O(n)$

$O(n)$ without DFS_rec

```
DFS(G=(V,E))
    Mark all vertices in G as "unvisited"  // time=0
    for each vertex v ∈ V do
        if v is unvisited
            DFS_rec(G,v)
```

$O(1)$

$O(deg(v))$ without recursive call

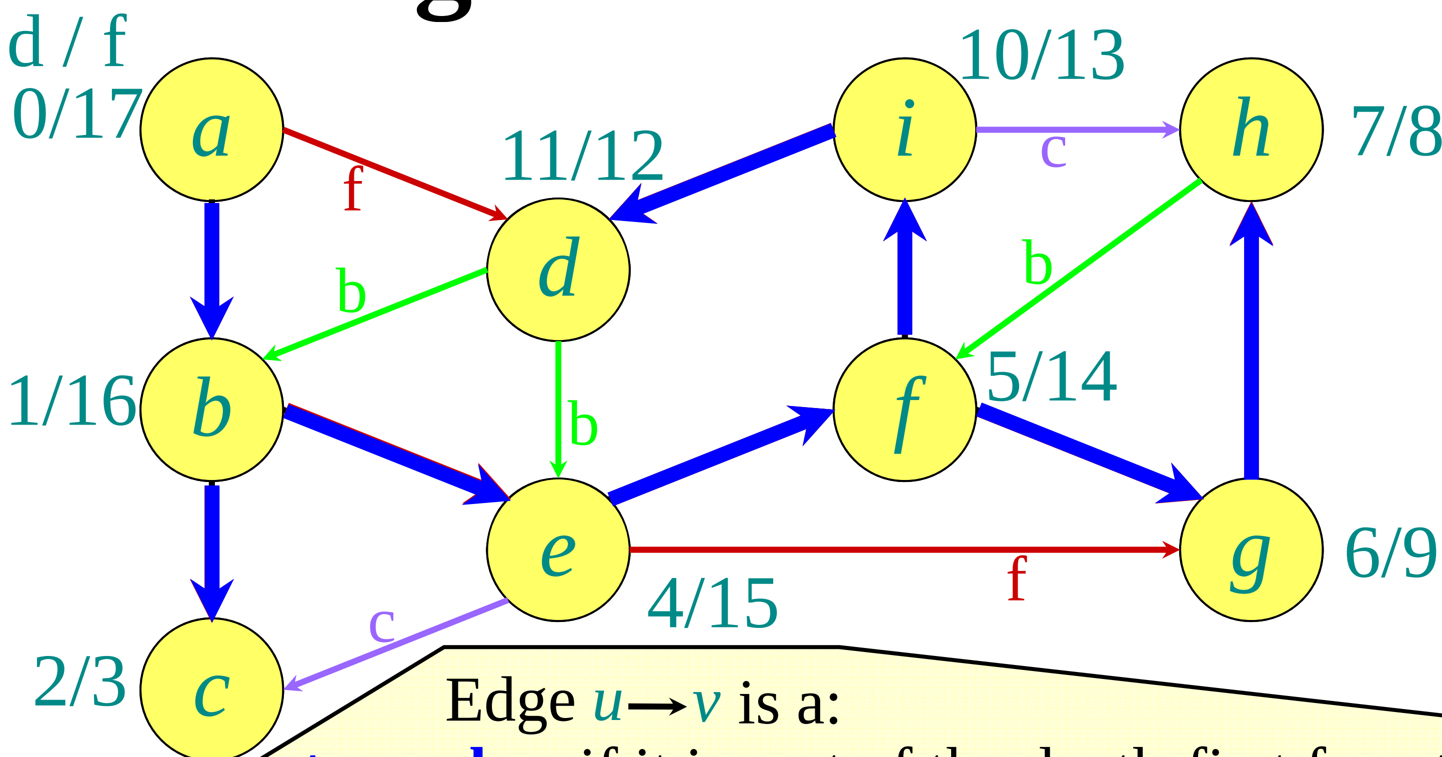
```
DFS_rec(G, v)
    mark v as "visited" // d[v]=++time
    for each w adjacent to v do
        if w is unvisited
            Add edge (v,w) to tree T
            DFS_rec(G,w)
    mark v as "finished" // f[v]=++time
```

$\Rightarrow$ With Handshaking Lemma, all recursive calls are $O(m)$, for a total of $O(n + m)$ runtime

DFS runtime

- Each vertex is visited at most once $\Rightarrow$ O(n) time
- The body of the **for** loops (except the recursive call) take constant time per graph edge
- All **for** loops take O(m) time
- Total runtime is O(n+m) = O(|V| + |E|)

DFS edge classification



Edge $u \longrightarrow v$ is a:

- **tree edge**, if it is part of the depth-first forest.
- back edge, if u connects to an ancestor v in a depth-first tree. It holds $d(u)>d(v)$ and $f(u)<f(v)$.
- forward edge, if it connects u to a descendant v in a depth-first tree. It holds $d(u)<d(v)$.
- cross edge, if it is any other edge. It holds $d(u)>d(v)$ and $f(u)>f(v)$.

Paths, Cycles, Connectivity

Let $G=(V,E)$ be a directed (or undirected) graph

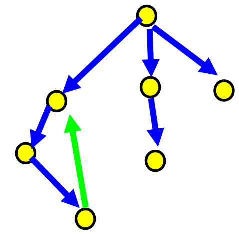
- A **path** from v_1 to v_k in G is a sequence of vertices $v_1, v_2,\ldots,v_k$ such that $(v_i,v_{\{i+1\}})\in E$ (or $\{v_i,v_{\{i+1\}}\} \in E$ if G is undirected) for all $i\in\{1,\ldots,k\text{-}1\}$.
- A path is **simple** if all vertices in the path are distinct.
- A path $v_1, v_2,\ldots,v_k$ forms a **cycle** if $v_1=v_k$.
- A graph with no cycles is **acyclic**.
    - An undirected acyclic graph is called a **tree**. (Trees do not have to have a root vertex specified.)
    - A directed acyclic graph is a **DAG**. (A DAG can have undirected cycles if the direction of the edges is not considered.)
- An undirected graph is **connected** if every pair of vertices is connected by a path. A directed graph is **strongly connected** if for every pair $u,v\in V$ there is a path from u to v and there is a path from v to u.
- The **(strongly) connected components** of a graph are the equivalence classes of vertices under this reachability relation.

DAG Theorem

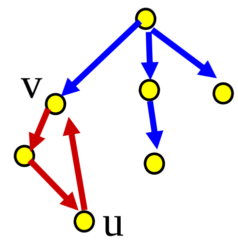
Theorem: A directed graph G is acyclic
 ⇔ a depth-first search of G yields no back edges.

Proof:

"⇒": Suppose there is a back edge (u,v). Then by definition of a back edge there would be a cycle.



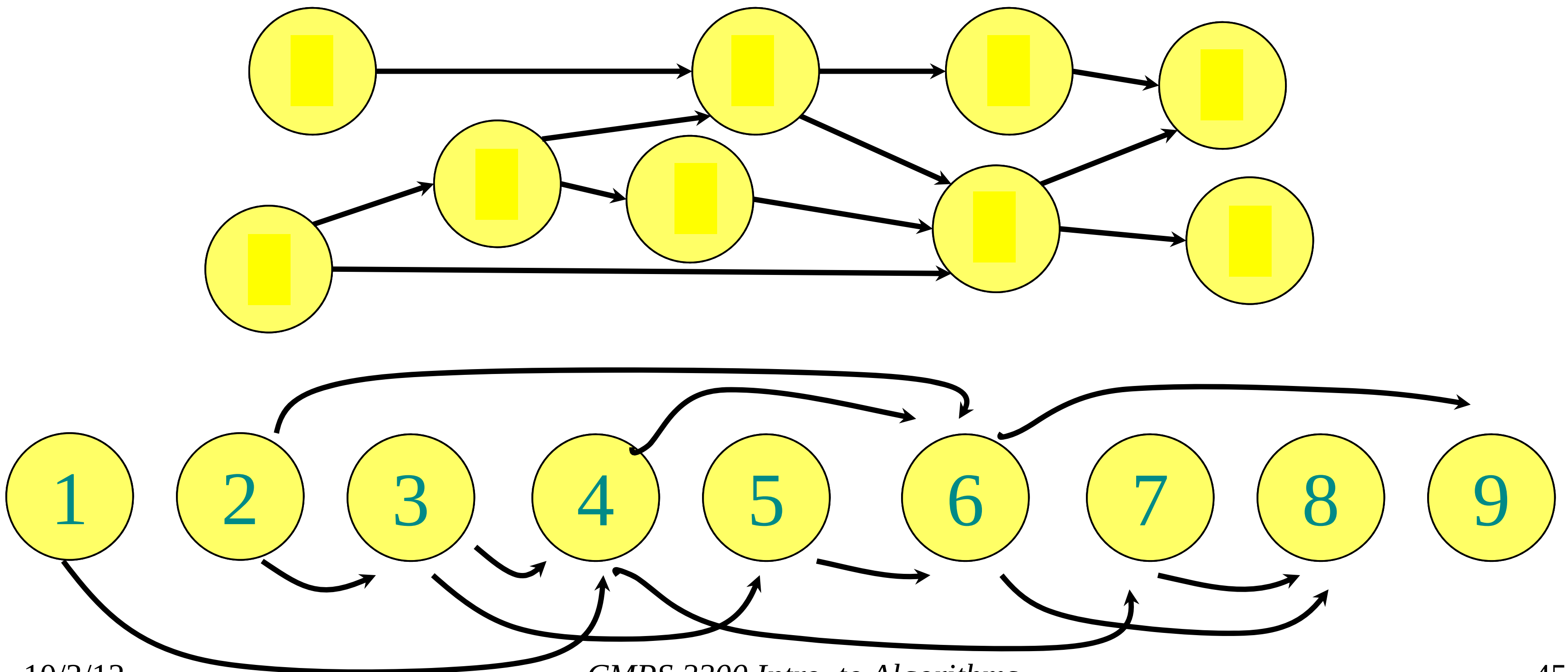
"⇐": Suppose G contains a cycle c. Let v be the first vertex to be discovered in c, and let u be the preceding vertex in c. v is an ancestor of u in the depth-first forest, hence (u,v) is a back edge.



Topological Sort

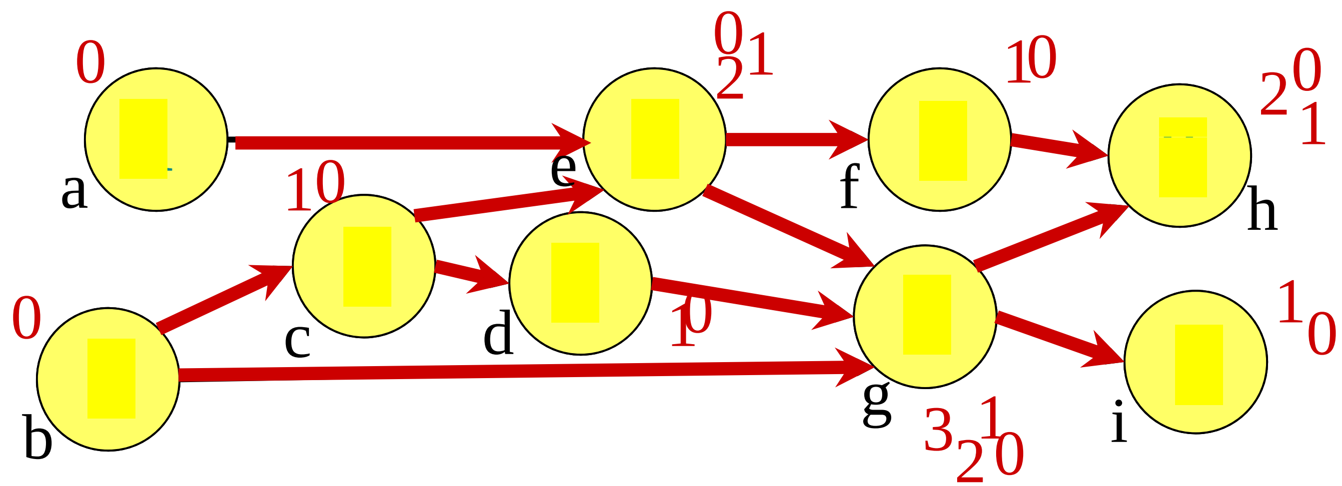
Topologically sort the vertices of a ***directed acyclic graph*** (***DAG***):

- Determine $f : V \rightarrow \{1, 2, \ldots, |V|\}$ such that $(u, v) \in E$ $\Rightarrow f(u) < f(v)$.



Topological Sort Algorithm

- Store vertices with in-degree 0 in a queue Q.
- While Q is not empty
  - Dequeue vertex v, and give it the next number
  - Decrease in-degree of all adjacent vertices by 1
  - Enqueue all vertices with in-degree 0



Q: a , b , c, e, d, f, g, i , h

Topological Sort Runtime

Runtime:

- $O(|V|+|E|)$ because every edge is touched once, and every vertex is enqueued and dequeued exactly once

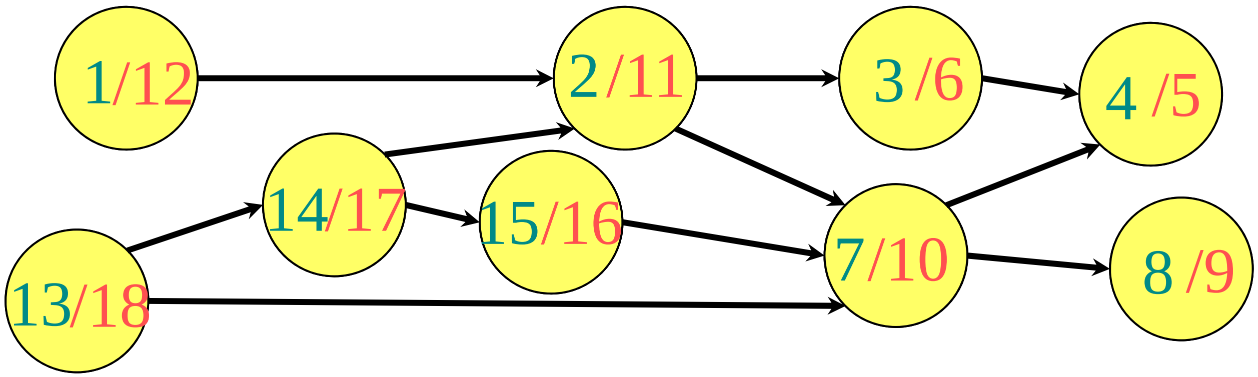
DFS-Based Topological Sort Algorithm

- Call DFS on the directed acyclic graph $G=(V,E)$
  $\Rightarrow$ Finish time for every vertex
- Reverse the finish times (highest finish time becomes the lowest finish time,…)
  $\Rightarrow$ Valid function $f\,'\colon V \to \{1, 2, \ldots, |\,V\,|\}$ such that
  $(u, v) \in E \Rightarrow f\,'(u) < f\,'(v)$

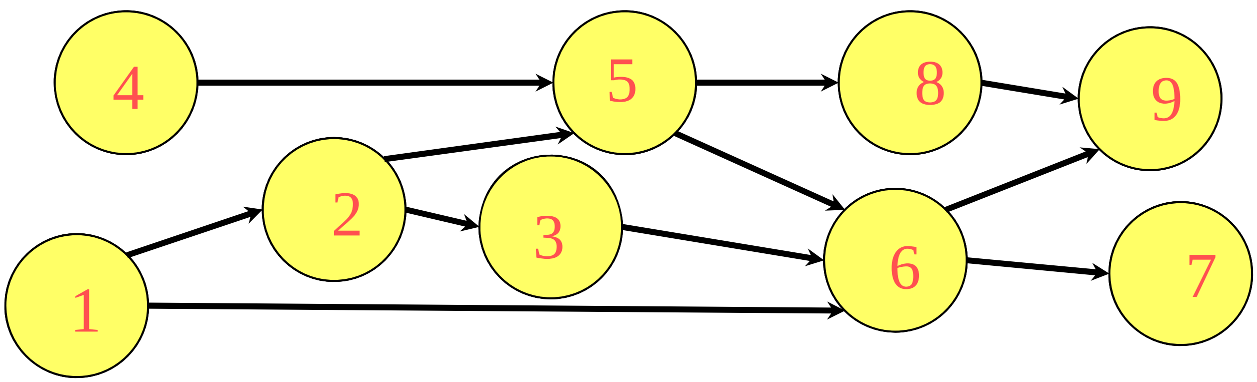
Runtime: $O(|V|+|E|)$

DFS-Based Topological Sort

- Run DFS:



- Reverse finish times:



DFS-Based Top. Sort Correctness

- Need to show that for any $(u, v) \in E$ holds $f(v) < f(u)$.
  (since we consider reversed finish times)
- Consider exploring edge (u, v) in DFS:
  - v cannot be visited and unfinished (and hence an ancestor in the depth first tree), since then (u,v) would be a back edge (which by the DAG lemma cannot happen).
  - If v has not been visited yet, it becomes a descendant of u, and hence $f(v)<f(u)$. (tree edge)
  - If v has been finished, $f(v)$ has been set, and u is still being explored, hence $f(u)>f(v)$ (forward edge, cross edge) .

Topological Sort Runtime

Runtime:

- O(|V|+|E|) because every edge is touched once, and every vertex is enqueued and dequeued exactly once

- DFS-based algorithm: O(|V| + |E|)