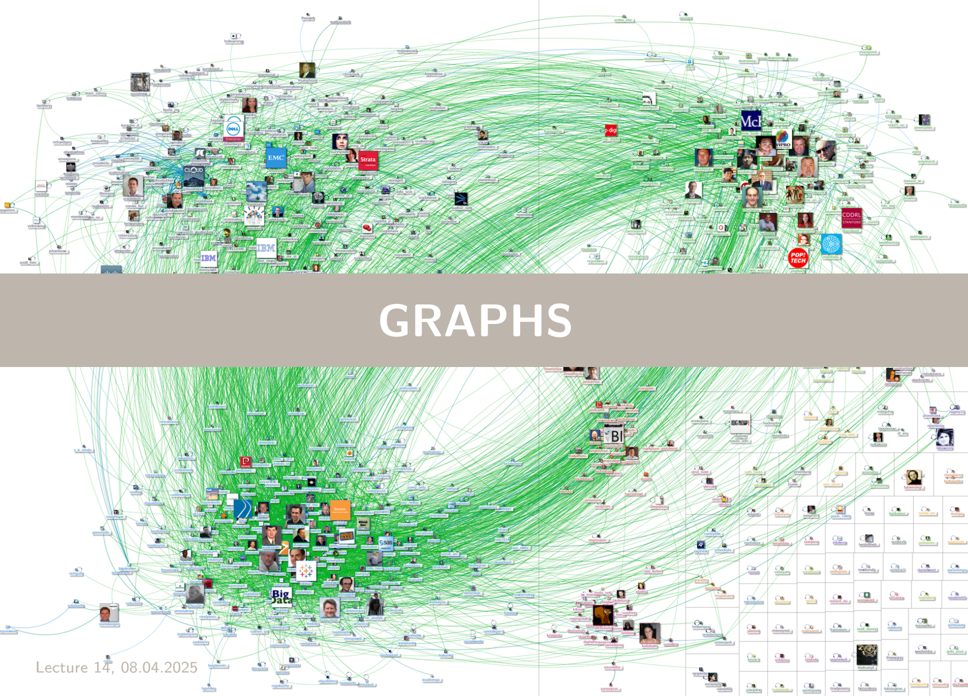
# Algorithms: Elementary Graph Algorithms (BFS, DFS, TOPOLOGICAL SORT)

Ola Svensson

EPFL School of Computer and Communication Sciences

Lecture 14, 08.04.2025



# GRAPHS

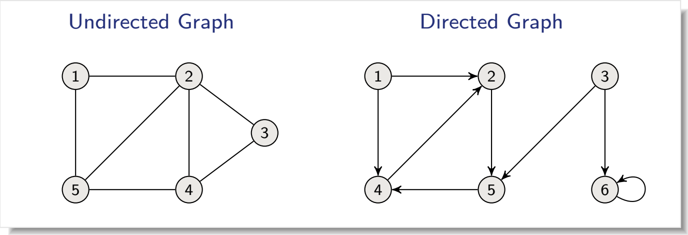
Lecture 14, 08.04.2025

# Graphs

A graph $G = (V, E)$ consists of

- a vertex set $V$
- an edge set $E$ that contain (ordered) pairs of vertices

A graph can be undirected, directed, vertex-weighted, edge-weighted, etc.

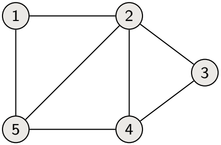



How to represent a graph in the computer?

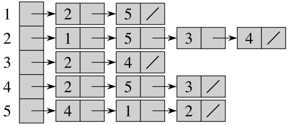
# Adjacency Lists

- Array *Adj* of $|V|$ lists, one per vertex
- Vertex $u$'s list has all vertices $v$ such that $(u, v) \in E$ (works for both undirected and directed graphs)

Undirected Graph




Adjacency list *Adj*

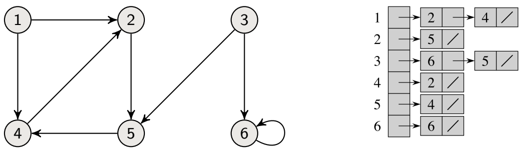


# Adjacency Lists

- Array *Adj* of $|V|$ lists, one per vertex
- Vertex $u$'s list has all vertices $v$ such that $(u, v) \in E$ (works for both undirected and directed graphs)

Directed Graph

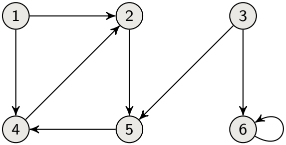
Adjacency list *Adj*



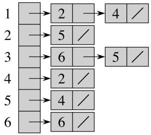
# Adjacency Lists

- Array *Adj* of $|V|$ lists, one per vertex
- Vertex $u$'s list has all vertices $v$ such that $(u, v) \in E$ (works for both undirected and directed graphs)
- In pseudocode, we will denote the array as attribute $G.Adj$, so we will see notation such as $G.Adj[u]$.

Directed Graph




Adjacency list *Adj*

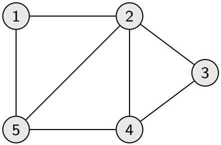


# Adjacency matrix

- A $|V| \times |V|$ matrix $A = (a_{ij})$ where

$$a_{ij} = \begin{cases} 1 & \text{if}(i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

**Undirected Graph**




**Adjacency matrix**

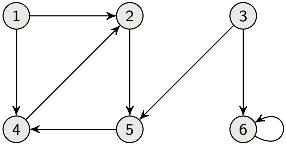
| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 | 1 |
| 2 | 1 | 0 | 1 | 1 | 1 |
| 3 | 0 | 1 | 0 | 1 | 0 |
| 4 | 0 | 1 | 1 | 0 | 1 |
| 5 | 1 | 1 | 0 | 1 | 0 |

# Adjacency matrix

- A $|V| \times |V|$ matrix $A = (a_{ij})$ where

$$a_{ij} = \begin{cases} 1 & \text{if}(i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

**Directed Graph**




**Adjacency matrix**

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 1 | 0 |
| 3 | 0 | 0 | 0 | 0 | 1 | 1 |
| 4 | 0 | 1 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 1 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 0 | 1 |

# Comparison of adjacency list and adjacency matrix

**Adjacency list**

Space $= \Theta(V + E)$

Time: to list all vertices adjacent to $u$: $\Theta(\text{degree}(u))$

Time: to determine whether $(u, v) \in E$: $O(\text{degree}(u))$

**Adjacency matrix**

Space $= \Theta(V^2)$

Time: to list all vertices adjacent to $u$: $\Theta(V)$

Time: to determine whether $(u, v) \in E$: $\Theta(1)$

---

We can extend both representations to include other attributes such as edge weights

# TRAVERSING/SEARCHING A GRAPH

# Breadth-First Search

## Definition

INPUT: Graph $G = (V, E)$, either directed or undirected and source vertex $s \in V$

OUTPUT: $v.d$ = distance (smallest number of edges) from $s$ to $v$, for all $v \in V$

# Breadth-First Search

## Definition

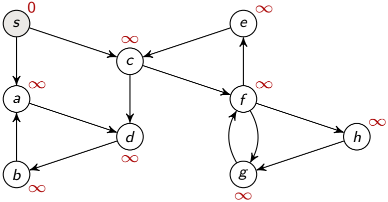
INPUT: Graph $G = (V, E)$, either directed or undirected and source vertex $s \in V$

OUTPUT: $v.d$ = distance (smallest number of edges) from $s$ to $v$, for all $v \in V$

Idea:

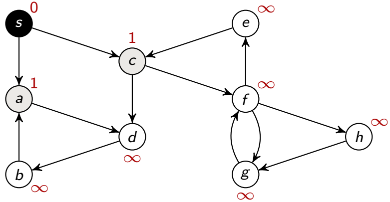
- Send a wave out from $s$
- First hits all vertices 1 edge from $s$
- From there, hits all vertices 2 edges from $s$ ...

# Example of Breadth-first search



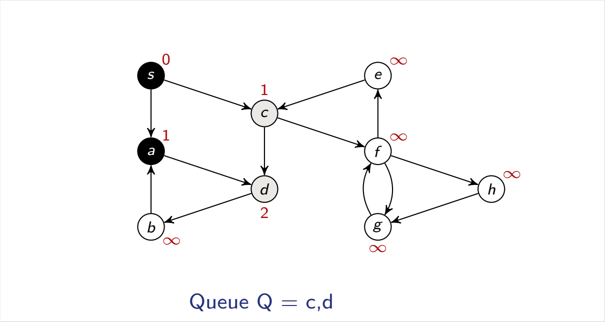

Queue Q = s

# Example of Breadth-first search

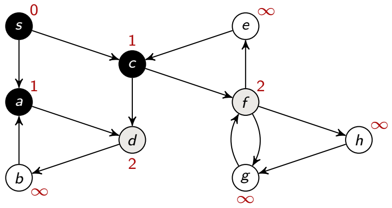



Queue Q = a,c

# Example of Breadth-first search

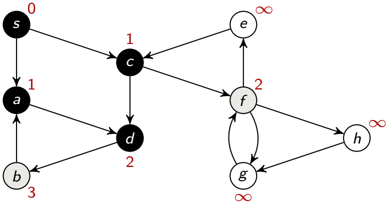


# Example of Breadth-first search



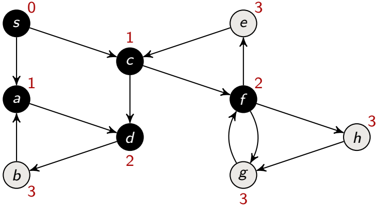

Queue Q = d,f

# Example of Breadth-first search



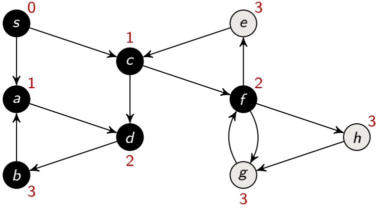

Queue Q = f,b

# Example of Breadth-first search



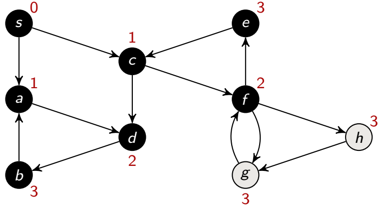

Queue Q = b,e,g,h

# Example of Breadth-first search



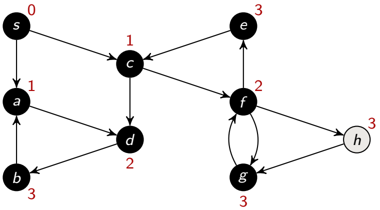

Queue Q = e,g,h

# Example of Breadth-first search



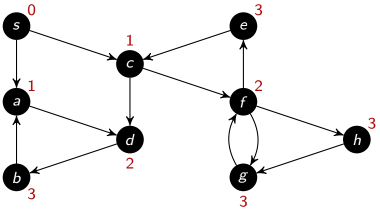

Queue Q = g,h

# Example of Breadth-first search




Queue Q = h

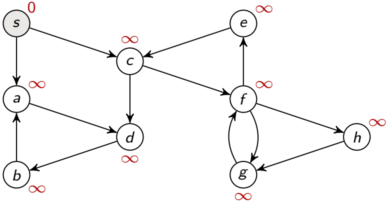
# Example of Breadth-first search




Queue Q = nil

# Pseudocode of Breadth-first search

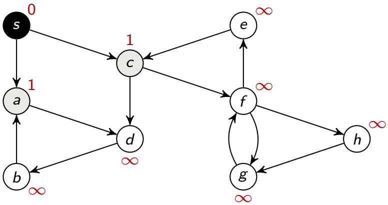
BFS($V, E, s$)
  **for** each $u \in V - \{s\}$
    $u.d = \infty$
  $s.d = 0$
  $Q = \emptyset$
  ENQUEUE($Q, s$)
  **while** $Q \neq \emptyset$
    $u$ = DEQUEUE($Q$)
    **for** each $v \in G.Adj[u]$
      **if** $v.d == \infty$
        $v.d = u.d + 1$
        ENQUEUE($Q, v$)




Queue Q = s

# Pseudocode of Breadth-first search

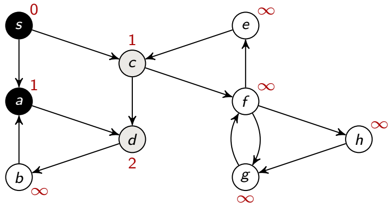
BFS($V, E, s$)
  **for** each $u \in V - \{s\}$
    $u.d = \infty$
  $s.d = 0$
  $Q = \emptyset$
  ENQUEUE($Q, s$)
  **while** $Q \neq \emptyset$
    $u$ = DEQUEUE($Q$)
    **for** each $v \in G.Adj[u]$
      **if** $v.d == \infty$
        $v.d = u.d + 1$
        ENQUEUE($Q, v$)




Queue Q = a,c

# Pseudocode of Breadth-first search

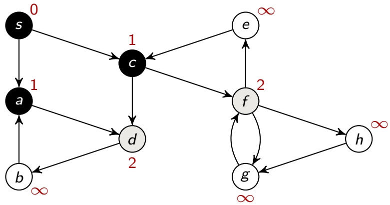
BFS($V, E, s$)
  **for** each $u \in V - \{s\}$
    $u.d = \infty$
  $s.d = 0$
  $Q = \emptyset$
  ENQUEUE($Q, s$)
  **while** $Q \neq \emptyset$
    $u =$ DEQUEUE($Q$)
    **for** each $v \in G.Adj[u]$
      **if** $v.d == \infty$
        $v.d = u.d + 1$
        ENQUEUE($Q, v$)




Queue Q $=$ c,d

# Pseudocode of Breadth-first search

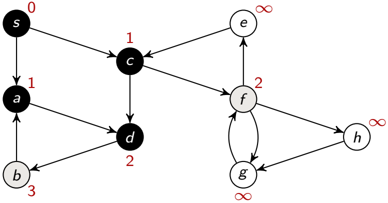
BFS$(V, E, s)$
- **for** each $u \in V - \{s\}$
  - $u.d = \infty$
- $s.d = 0$
- $Q = \emptyset$
- ENQUEUE$(Q, s)$
- **while** $Q \neq \emptyset$
  - $u =$ DEQUEUE$(Q)$
  - **for** each $v \in G.Adj[u]$
    - **if** $v.d == \infty$
      - $v.d = u.d + 1$
      - ENQUEUE$(Q, v)$




Queue Q = d,f

# Pseudocode of Breadth-first search

BFS($V, E, s$)
**for** each $u \in V - \{s\}$
  $u.d = \infty$
$s.d = 0$
$Q = \emptyset$
ENQUEUE($Q, s$)
**while** $Q \neq \emptyset$
  $u$ = DEQUEUE($Q$)
  **for** each $v \in G.Adj[u]$
    **if** $v.d == \infty$
      $v.d = u.d + 1$
      ENQUEUE($Q, v$)

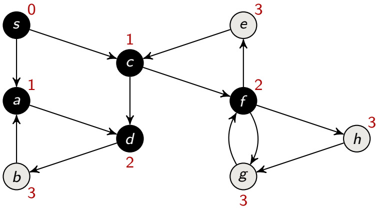



Queue Q = f,b

# Pseudocode of Breadth-first search

BFS$(V, E, s)$

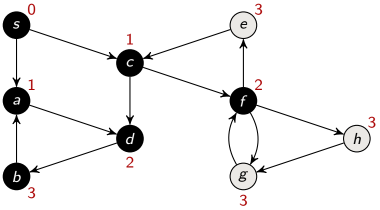
**for** each $u \in V - \{s\}$
  $u.d = \infty$
$s.d = 0$
$Q = \emptyset$
ENQUEUE$(Q, s)$
**while** $Q \neq \emptyset$
  $u =$ DEQUEUE$(Q)$
  **for** each $v \in G.Adj[u]$
    **if** $v.d == \infty$
      $v.d = u.d + 1$
      ENQUEUE$(Q, v)$




Queue Q = b,e,g,h

# Pseudocode of Breadth-first search

BFS($V, E, s$)
  **for** each $u \in V - \{s\}$
    $u.d = \infty$
  $s.d = 0$
  $Q = \emptyset$
  ENQUEUE($Q, s$)
  **while** $Q \neq \emptyset$
    $u =$ DEQUEUE($Q$)
    **for** each $v \in G.Adj[u]$
      **if** $v.d == \infty$
        $v.d = u.d + 1$
        ENQUEUE($Q, v$)

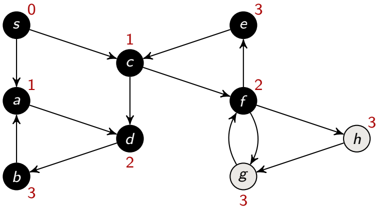



Queue Q = e,g,h

# Pseudocode of Breadth-first search

BFS($V, E, s$)

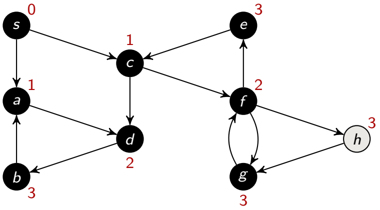
**for** each $u \in V - \{s\}$
    $u.d = \infty$
$s.d = 0$
$Q = \emptyset$
ENQUEUE($Q, s$)
**while** $Q \neq \emptyset$
    $u$ = DEQUEUE($Q$)
    **for** each $v \in G.Adj[u]$
        **if** $v.d == \infty$
            $v.d = u.d + 1$
            ENQUEUE($Q, v$)




Queue Q = g,h

# Pseudocode of Breadth-first search

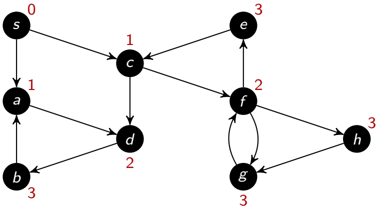
BFS($V, E, s$)
  **for** each $u \in V - \{s\}$
    $u.d = \infty$
  $s.d = 0$
  $Q = \emptyset$
  ENQUEUE($Q, s$)
  **while** $Q \neq \emptyset$
    $u$ = DEQUEUE($Q$)
    **for** each $v \in G.Adj[u]$
      **if** $v.d == \infty$
        $v.d = u.d + 1$
        ENQUEUE($Q, v$)




Queue Q = h

# Pseudocode of Breadth-first search

BFS$(V, E, s)$
  **for** each $u \in V - \{s\}$
    $u.d = \infty$
  $s.d = 0$
  $Q = \emptyset$
  ENQUEUE$(Q, s)$
  **while** $Q \neq \emptyset$
    $u =$ DEQUEUE$(Q)$
    **for** each $v \in G.Adj[u]$
      **if** $v.d == \infty$
        $v.d = u.d + 1$
        ENQUEUE$(Q, v)$




Queue Q = nil

# Analysis

Informal Idea of correctness (formal proof in book):

# Analysis

Informal Idea of correctness (formal proof in book):

- Suppose that $v.d$ is greater than the shortest distance from $s$ to $v$
- but since algorithm repeatedly considers the vertices closest to the root (by adding them to the queue) this cannot happen

# Analysis

Informal Idea of correctness (formal proof in book):

- Suppose that $v.d$ is greater than the shortest distance from $s$ to $v$
- but since algorithm repeatedly considers the vertices closest to the root (by adding them to the queue) this cannot happen

Runtime analysis:

# Analysis

Informal Idea of correctness (formal proof in book):

- Suppose that $v.d$ is greater than the shortest distance from $s$ to $v$
- but since algorithm repeatedly considers the vertices closest to the root (by adding them to the queue) this cannot happen

Runtime analysis: O(V+E)

# Analysis

Informal Idea of correctness (formal proof in book):

- Suppose that $v.d$ is greater than the shortest distance from $s$ to $v$
- but since algorithm repeatedly considers the vertices closest to the root (by adding them to the queue) this cannot happen

Runtime analysis: O(V+E)

- $O(V)$ because each vertex enqueued at most once

# Analysis

Informal Idea of correctness (formal proof in book):

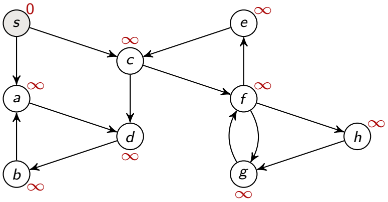
- Suppose that $v.d$ is greater than the shortest distance from $s$ to $v$
- but since algorithm repeatedly considers the vertices closest to the root (by adding them to the queue) this cannot happen

Runtime analysis: O(V+E)

- $O(V)$ because each vertex enqueued at most once
- $O(E)$ because every vertex dequeued at most once and we examine $(u, v)$ only when $u$ is dequeued. Therefore, every edge examined at most once if directed and at most twice if undirected

# Final notes on BFS

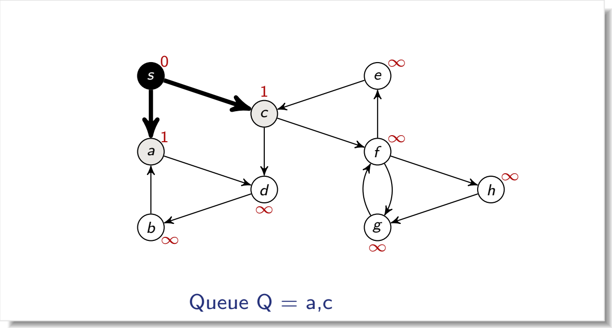
- BFS may not reach all the vertices
- We can save the shortest path tree by keeping track of the edge that discovered the vertex




Queue Q = s

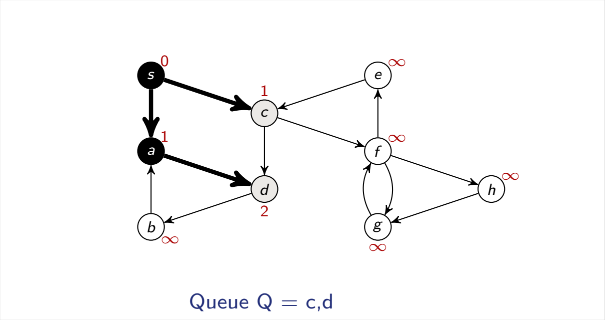
# Final notes on BFS

- BFS may not reach all the vertices
- We can save the shortest path tree by keeping track of the edge that discovered the vertex



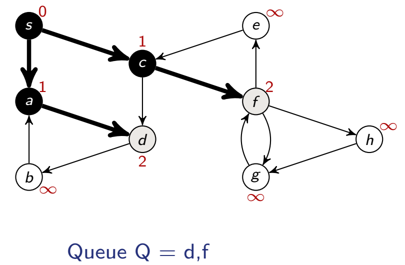
# Final notes on BFS

- BFS may not reach all the vertices
- We can save the shortest path tree by keeping track of the edge that discovered the vertex



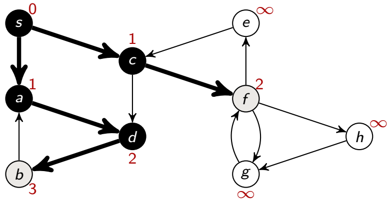
# Final notes on BFS

- BFS may not reach all the vertices
- We can save the shortest path tree by keeping track of the edge that discovered the vertex



# Final notes on BFS

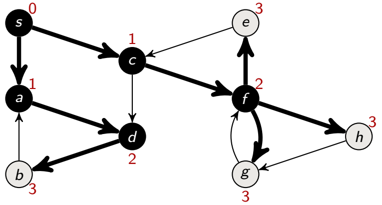
- BFS may not reach all the vertices
- We can save the shortest path tree by keeping track of the edge that discovered the vertex




Queue Q = f,b

# Final notes on BFS

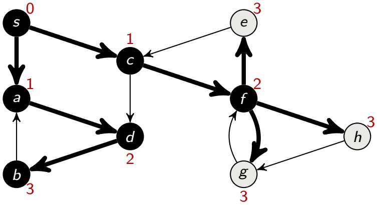
- BFS may not reach all the vertices
- We can save the shortest path tree by keeping track of the edge that discovered the vertex




Queue Q = b,e,g,h

# Final notes on BFS

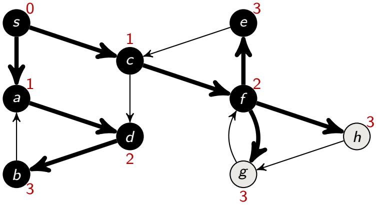
- BFS may not reach all the vertices
- We can save the shortest path tree by keeping track of the edge that discovered the vertex




Queue $Q = e,g,h$

# Final notes on BFS

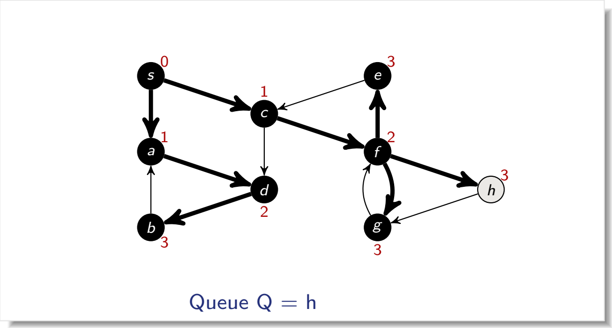
- BFS may not reach all the vertices
- We can save the shortest path tree by keeping track of the edge that discovered the vertex




Queue Q = g,h

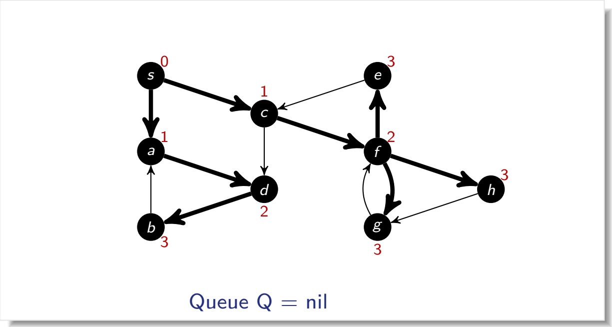
# Final notes on BFS

- BFS may not reach all the vertices
- We can save the shortest path tree by keeping track of the edge that discovered the vertex



# Final notes on BFS

- BFS may not reach all the vertices
- We can save the shortest path tree by keeping track of the edge that discovered the vertex



# Depth-First Search

## Definition

INPUT: Graph $G = (V, E)$, either directed or undirected

OUTPUT: 2 timestamps on each vertex: $v.d =$ **discovery time** and $v.f =$ **finishing time**

# Depth-First Search

## Definition

INPUT: Graph $G = (V, E)$, either directed or undirected

OUTPUT: 2 timestamps on each vertex: $v.d$ = **discovery time** and $v.f$ = **finishing time**

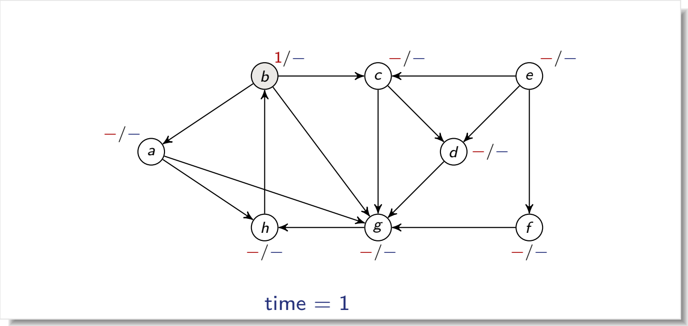
Idea:

- Methodically explore *every* edge
- Start over from different vertices as necessary
- As soon as we discover a vertex explore from it,
  - Unlike BFS, which explores vertices that are close to a source first

# Example of DFS

As DFS progresses, every vertex has a color:

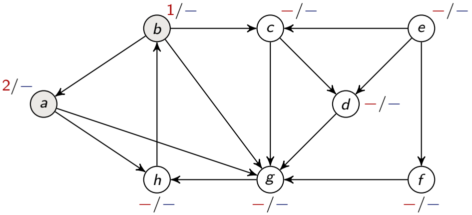
- WHITE = undiscovered
- GRAY = discovered, but not finished (not done exploring from it)
- BLACK = finished (have found everything reachable from it)



# Example of DFS

As DFS progresses, every vertex has a color:

- WHITE = undiscovered
- GRAY = discovered, but not finished (not done exploring from it)
- BLACK = finished (have found everything reachable from it)

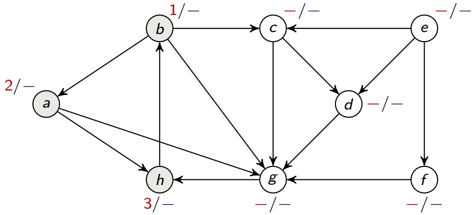



time = 2

# Example of DFS

As DFS progresses, every vertex has a color:

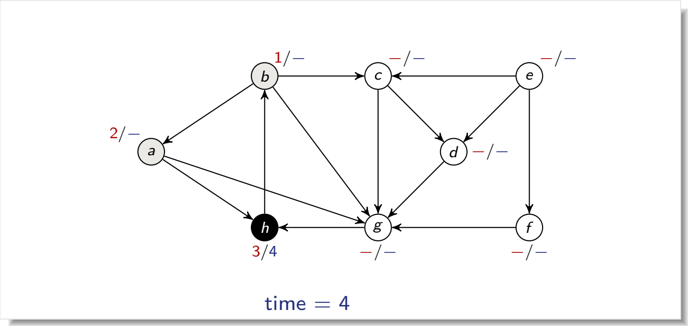
- WHITE = undiscovered
- GRAY = discovered, but not finished (not done exploring from it)
- BLACK = finished (have found everything reachable from it)



# Example of DFS

As DFS progresses, every vertex has a color:

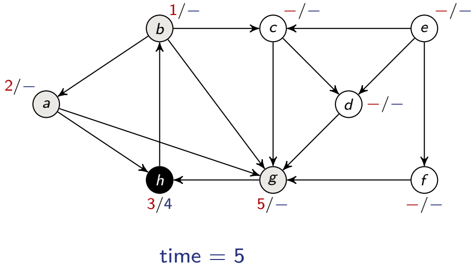
- WHITE = undiscovered
- GRAY = discovered, but not finished (not done exploring from it)
- BLACK = finished (have found everything reachable from it)



# Example of DFS

As DFS progresses, every vertex has a color:

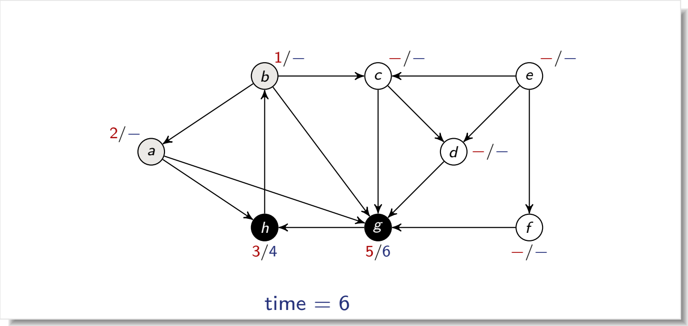
- WHITE = undiscovered
- GRAY = discovered, but not finished (not done exploring from it)
- BLACK = finished (have found everything reachable from it)



# Example of DFS

As DFS progresses, every vertex has a color:

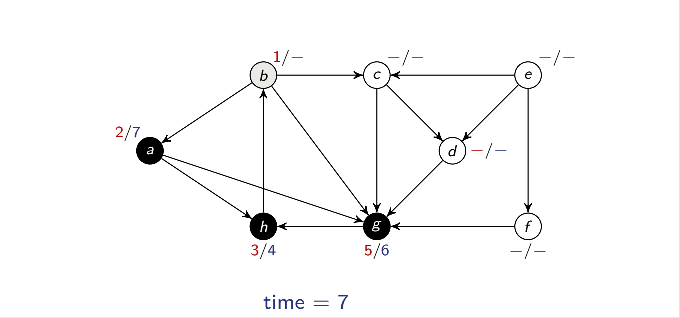
- WHITE = undiscovered
- GRAY = discovered, but not finished (not done exploring from it)
- BLACK = finished (have found everything reachable from it)



# Example of DFS

As DFS progresses, every vertex has a color:

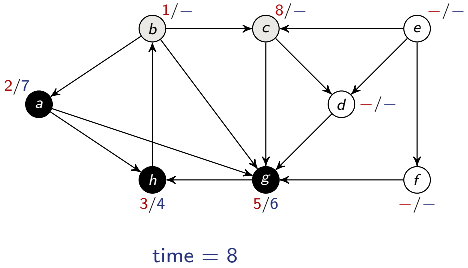
- WHITE = undiscovered
- GRAY = discovered, but not finished (not done exploring from it)
- BLACK = finished (have found everything reachable from it)



# Example of DFS

As DFS progresses, every vertex has a color:

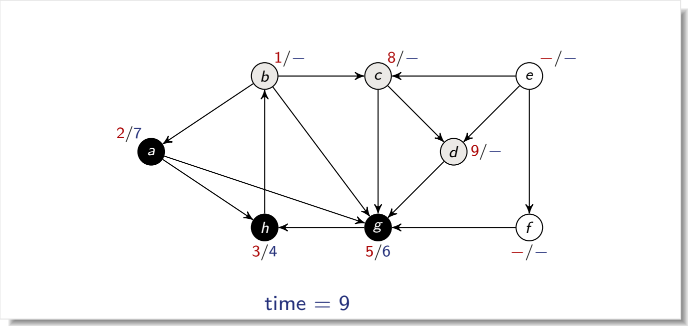
- WHITE = undiscovered
- GRAY = discovered, but not finished (not done exploring from it)
- BLACK = finished (have found everything reachable from it)



# Example of DFS

As DFS progresses, every vertex has a color:

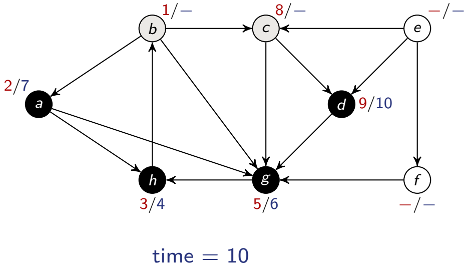
- WHITE = undiscovered
- GRAY = discovered, but not finished (not done exploring from it)
- BLACK = finished (have found everything reachable from it)



# Example of DFS

As DFS progresses, every vertex has a color:

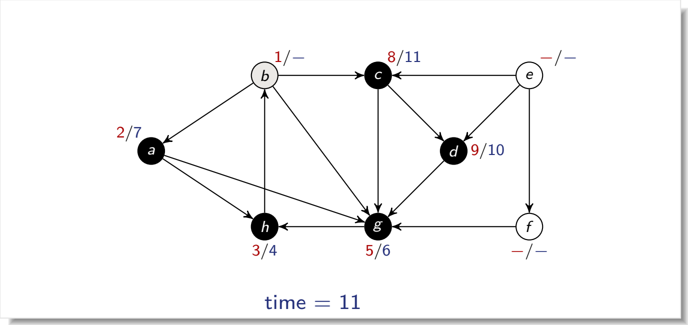
- WHITE = undiscovered
- GRAY = discovered, but not finished (not done exploring from it)
- BLACK = finished (have found everything reachable from it)



# Example of DFS

As DFS progresses, every vertex has a color:

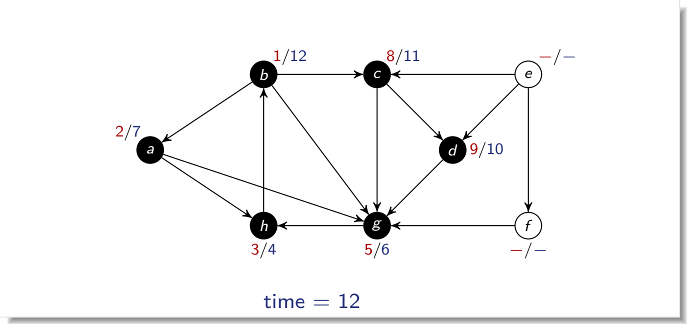
- WHITE = undiscovered
- GRAY = discovered, but not finished (not done exploring from it)
- BLACK = finished (have found everything reachable from it)



# Example of DFS

As DFS progresses, every vertex has a color:

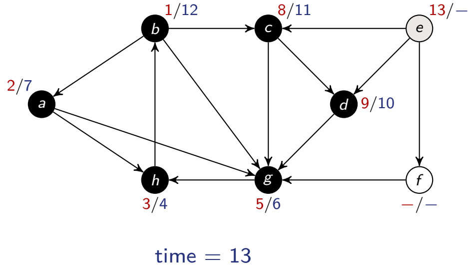
- WHITE = undiscovered
- GRAY = discovered, but not finished (not done exploring from it)
- BLACK = finished (have found everything reachable from it)



# Example of DFS

As DFS progresses, every vertex has a color:

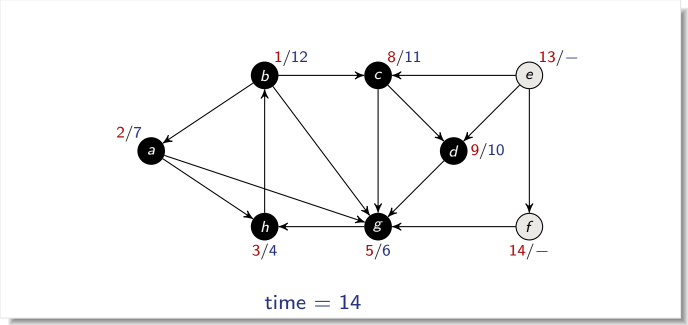
- WHITE = undiscovered
- GRAY = discovered, but not finished (not done exploring from it)
- BLACK = finished (have found everything reachable from it)



# Example of DFS

As DFS progresses, every vertex has a color:

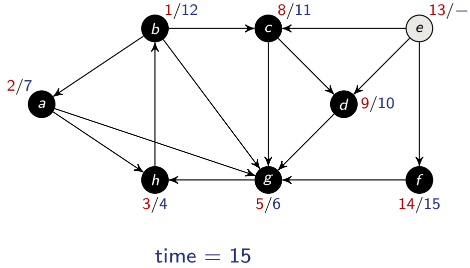
- WHITE = undiscovered
- GRAY = discovered, but not finished (not done exploring from it)
- BLACK = finished (have found everything reachable from it)



# Example of DFS

As DFS progresses, every vertex has a color:

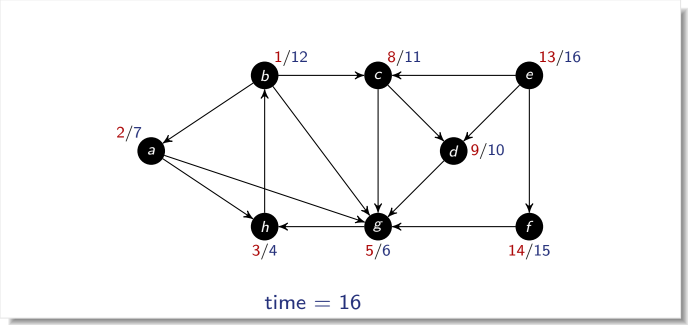
- WHITE = undiscovered
- GRAY = discovered, but not finished (not done exploring from it)
- BLACK = finished (have found everything reachable from it)



# Example of DFS

As DFS progresses, every vertex has a color:

- WHITE = undiscovered
- GRAY = discovered, but not finished (not done exploring from it)
- BLACK = finished (have found everything reachable from it)



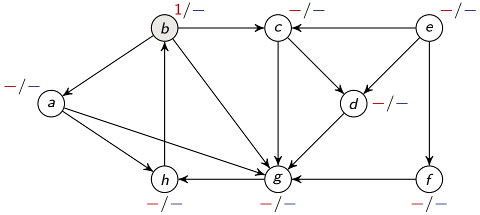
# Pseudocode of DFS

```
DFS(G)
  for each u ∈ G.V
      u.color = WHITE
  time = 0
  for each u ∈ G.V
      if u.color == WHITE
          DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
  time = time + 1
  u.d = time
  u.color = GRAY                  // discover u
  for each v ∈ G.Adj[u]           // explore (u, v)
      if v.color == WHITE
          DFS-VISIT(v)
  u.color = BLACK
  time = time + 1
  u.f = time                      // finish u
```

# Pseudocode of DFS

```
DFS-VISIT(G, u)
  time = time + 1
  u.d = time
  u.color = GRAY                   // discover u
  for each v ∈ G.Adj[u]            // explore (u, v)
       if v.color == WHITE
            DFS-VISIT(v)
  u.color = BLACK
  time = time + 1
  u.f = time                       // finish u
```

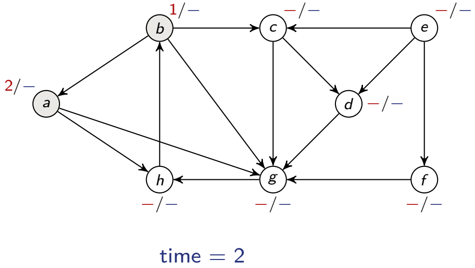



time = 1

# Pseudocode of DFS

DFS-VISIT$(G, u)$

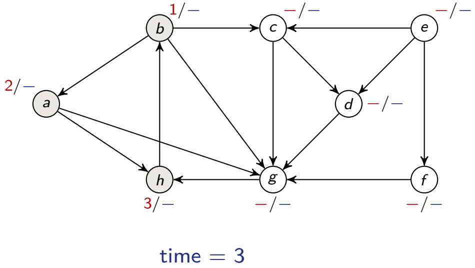
$time = time + 1$
$u.d = time$
$u.color =$ GRAY // discover $u$
**for** each $v \in G.Adj[u]$ // explore $(u, v)$
  **if** $v.color ==$ WHITE
    DFS-VISIT$(v)$
$u.color =$ BLACK
$time = time + 1$
$u.f = time$ // finish $u$



# Pseudocode of DFS

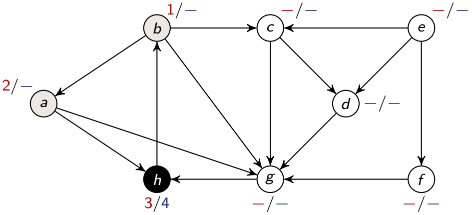
DFS-VISIT$(G, u)$

```
time = time + 1
u.d = time
u.color = GRAY                      // discover u
for each v ∈ G.Adj[u]               // explore (u, v)
    if v.color == WHITE
        DFS-VISIT(v)
u.color = BLACK
time = time + 1
u.f = time                          // finish u
```



# Pseudocode of DFS

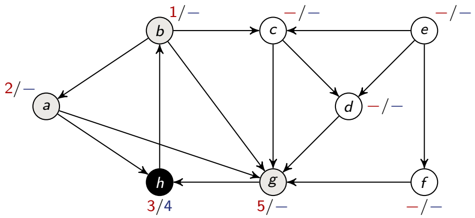
DFS-VISIT$(G, u)$
```
time = time + 1
u.d = time
u.color = GRAY                         // discover u
for each v ∈ G.Adj[u]                  // explore (u, v)
    if v.color == WHITE
        DFS-VISIT(v)
u.color = BLACK
time = time + 1
u.f = time                             // finish u
```



# Pseudocode of DFS

DFS-VISIT$(G, u)$

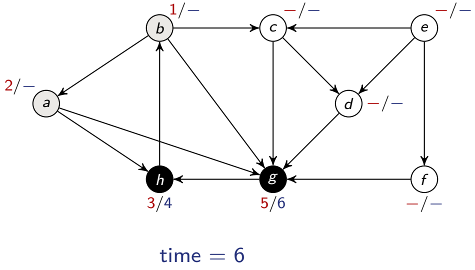
```
time = time + 1
u.d = time
u.color = GRAY                  // discover u
for each v ∈ G.Adj[u]           // explore (u, v)
    if v.color == WHITE
        DFS-VISIT(v)
u.color = BLACK
time = time + 1
u.f = time                      // finish u
```



# Pseudocode of DFS

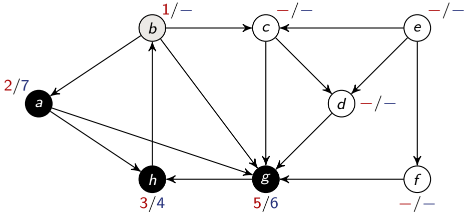
DFS-VISIT$(G, u)$

$time = time + 1$
$u.d = time$
$u.color =$ GRAY // discover $u$
**for** each $v \in G.Adj[u]$ // explore $(u, v)$
  **if** $v.color ==$ WHITE
    DFS-VISIT$(v)$
$u.color =$ BLACK
$time = time + 1$
$u.f = time$ // finish $u$



# Pseudocode of DFS

```
DFS-VISIT(G, u)
  time = time + 1
  u.d = time
  u.color = GRAY                 // discover u
  for each v ∈ G.Adj[u]          // explore (u, v)
      if v.color == WHITE
          DFS-VISIT(v)
  u.color = BLACK
  time = time + 1
  u.f = time                     // finish u
```

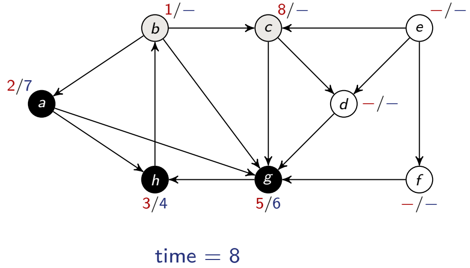



time = 7

# Pseudocode of DFS

DFS-VISIT$(G, u)$

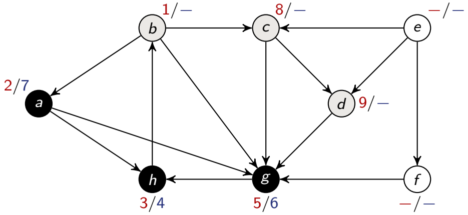
```
time = time + 1
u.d = time
u.color = GRAY                    // discover u
for each v ∈ G.Adj[u]             // explore (u, v)
    if v.color == WHITE
        DFS-VISIT(v)
u.color = BLACK
time = time + 1
u.f = time                        // finish u
```



# Pseudocode of DFS

DFS-VISIT$(G, u)$

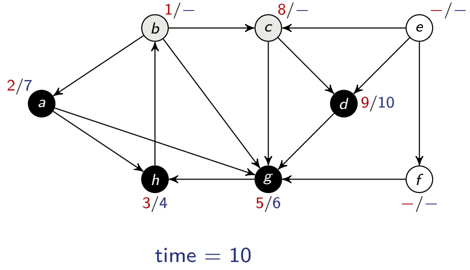
```
time = time + 1
u.d = time
u.color = GRAY                      // discover u
for each v ∈ G.Adj[u]               // explore (u, v)
    if v.color == WHITE
        DFS-VISIT(v)
u.color = BLACK
time = time + 1
u.f = time                          // finish u
```




time = 9

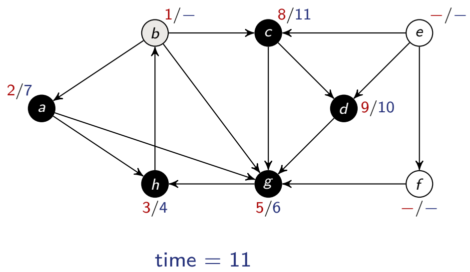
# Pseudocode of DFS

```
DFS-VISIT(G, u)
  time = time + 1
  u.d = time
  u.color = GRAY                     // discover u
  for each v ∈ G.Adj[u]              // explore (u, v)
      if v.color == WHITE
          DFS-VISIT(v)
  u.color = BLACK
  time = time + 1
  u.f = time                         // finish u
```



# Pseudocode of DFS

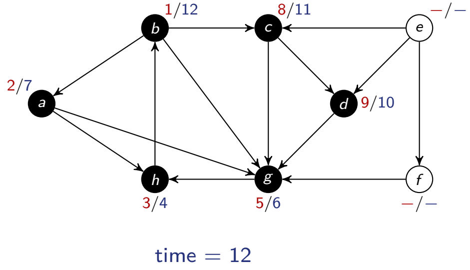
```
DFS-VISIT(G, u)
  time = time + 1
  u.d = time
  u.color = GRAY                       // discover u
  for each v ∈ G.Adj[u]                // explore (u, v)
      if v.color == WHITE
          DFS-VISIT(v)
  u.color = BLACK
  time = time + 1
  u.f = time                           // finish u
```



# Pseudocode of DFS

DFS-VISIT$(G, u)$

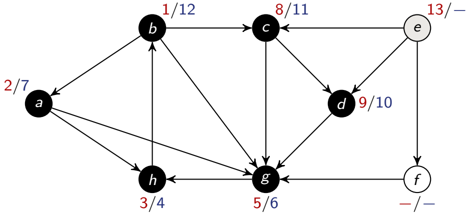
```
time = time + 1
u.d = time
u.color = GRAY                     // discover u
for each v ∈ G.Adj[u]              // explore (u, v)
    if v.color == WHITE
        DFS-VISIT(v)
u.color = BLACK
time = time + 1
u.f = time                         // finish u
```



# Pseudocode of DFS

DFS-VISIT$(G, u)$

```
time = time + 1
u.d = time
u.color = GRAY                          // discover u
for each v ∈ G.Adj[u]                   // explore (u, v)
    if v.color == WHITE
        DFS-VISIT(v)
u.color = BLACK
time = time + 1
u.f = time                              // finish u
```

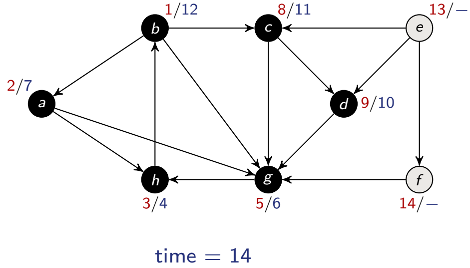



time = 13

# Pseudocode of DFS

DFS-VISIT$(G, u)$

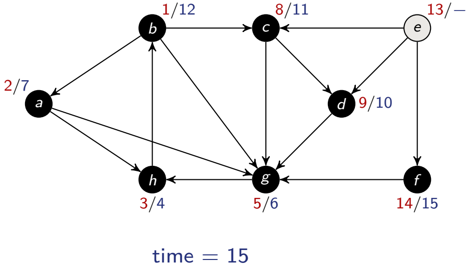
```
time = time + 1
u.d = time
u.color = GRAY                      // discover u
for each v ∈ G.Adj[u]               // explore (u, v)
    if v.color == WHITE
        DFS-VISIT(v)
u.color = BLACK
time = time + 1
u.f = time                          // finish u
```



# Pseudocode of DFS

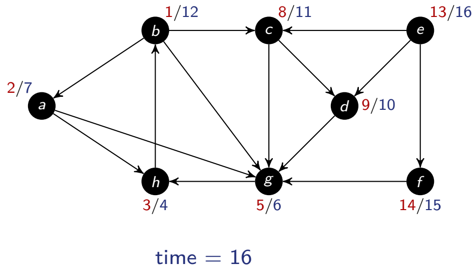
DFS-VISIT$(G, u)$

```
time = time + 1
u.d = time
u.color = GRAY                      // discover u
for each v ∈ G.Adj[u]               // explore (u, v)
    if v.color == WHITE
        DFS-VISIT(v)
u.color = BLACK
time = time + 1
u.f = time                          // finish u
```



# Pseudocode of DFS

```
DFS-VISIT(G, u)
  time = time + 1
  u.d = time
  u.color = GRAY                  // discover u
  for each v ∈ G.Adj[u]           // explore (u, v)
      if v.color == WHITE
          DFS-VISIT(v)
  u.color = BLACK
  time = time + 1
  u.f = time                      // finish u
```



# Analysis

DFS forms a depth-first forest comprised of $\geq 1$ depth-first trees. Each tree is made of edges $(u, v)$ such that $u$ is gray and $v$ is white when $(u, v)$ is explored.

# Analysis

DFS forms a depth-first forest comprised of $\geq 1$ depth-first trees. Each tree is made of edges $(u, v)$ such that $u$ is gray and $v$ is white when $(u, v)$ is explored.

Runtime analysis:

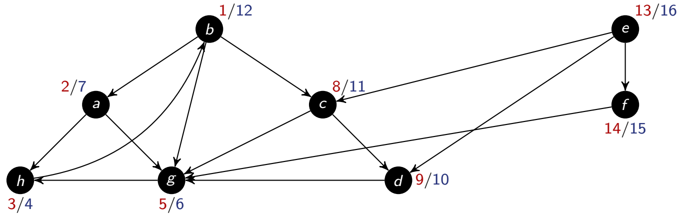
# Analysis

DFS forms a depth-first forest comprised of $\geq 1$ depth-first trees. Each tree is made of edges $(u, v)$ such that $u$ is gray and $v$ is white when $(u, v)$ is explored.

Runtime analysis: $\Theta(V + E)$

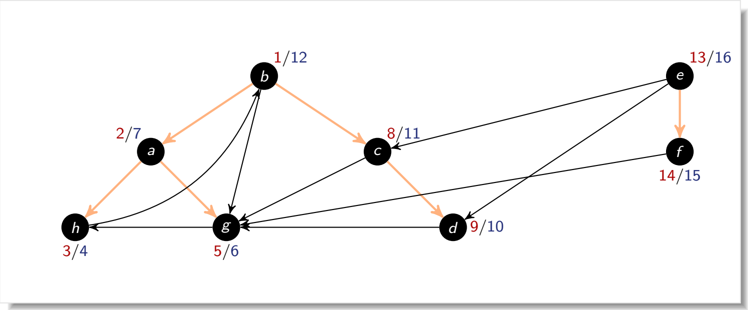
- $\Theta(V)$ because each vertex is discovered once
- $\Theta(E)$ because each edge is examined once if directed graph and twice if undirected graph.

# Classification of edges



# Classification of edges

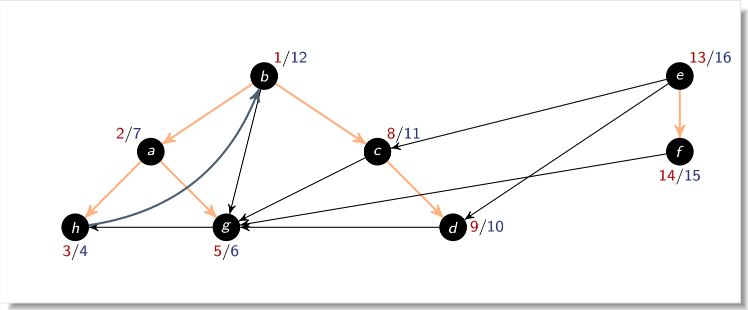
Tree edge: In the depth-first forest, found by exploring $(u, v)$



# Classification of edges

Tree edge: In the depth-first forest, found by exploring $(u, v)$

Back edge: $(u, v)$ where $u$ is a descendant of $v$

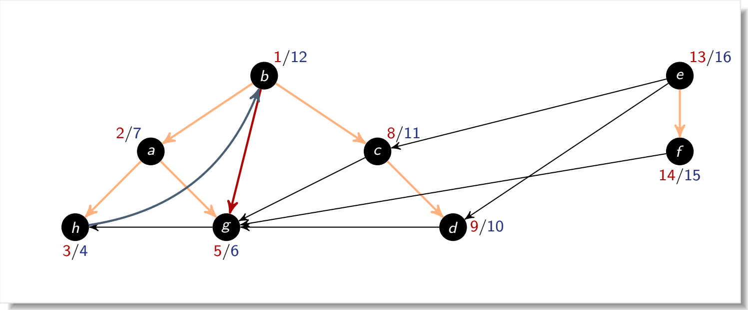


# Classification of edges

Tree edge: In the depth-first forest, found by exploring $(u, v)$

Back edge: $(u, v)$ where $u$ is a descendant of $v$

Forward edge: $(u, v)$ where $v$ is a descendant of $u$, but not a tree edge



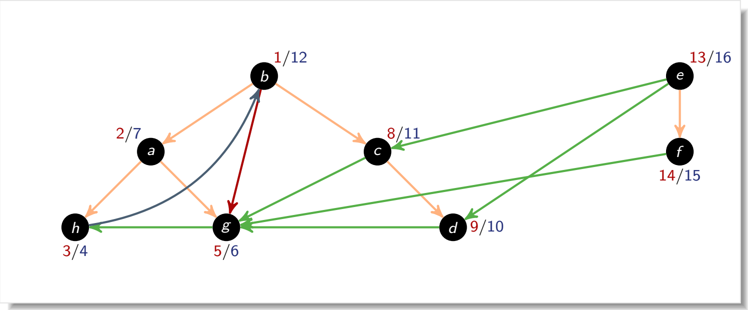
# Classification of edges

Tree edge: In the depth-first forest, found by exploring $(u, v)$

Back edge: $(u, v)$ where $u$ is a descendant of $v$

Forward edge: $(u, v)$ where $v$ is a descendant of $u$, but not a tree edge

Cross edge: any other edge



# Classification of edges

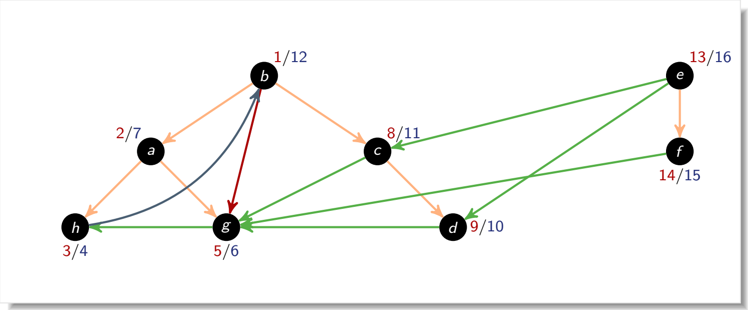
**Tree edge:** In the depth-first forest, found by exploring $(u, v)$

**Back edge:** $(u, v)$ where $u$ is a descendant of $v$

**Forward edge:** $(u, v)$ where $v$ is a descendant of $u$, but not a tree edge

**Cross edge:** any other edge

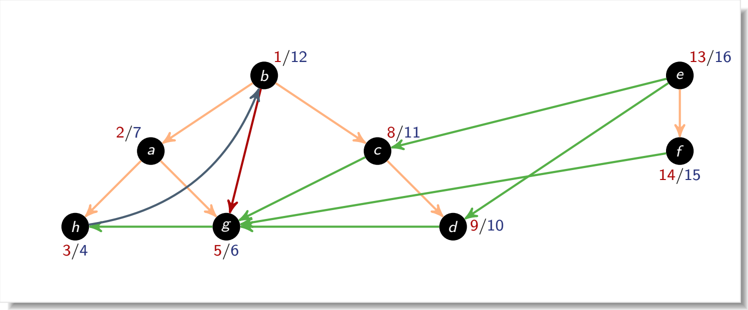
In DFS of an undirected graph we get only tree and back edges, no forward or cross-edges. Why?



# Parenthesis theorem

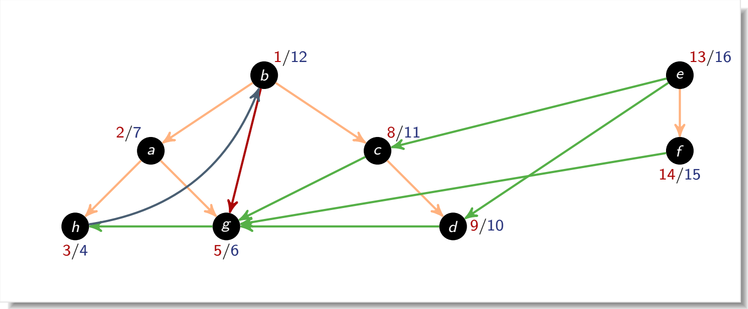
For all $u, v$ exactly one of the following holds

1. $u.d < u.f < v.d < v.f$ or $v.d < v.f < u.d < u.f$ and neither of $u$ and $v$ are descendant of each other
2. $u.d < v.d < v.f < u.f$ and $v$ is a descendant of $u$
3. $v.d < u.d < u.f < v.f$ and $u$ is a descendant of $v$.



# White-path theorem

Vertex $v$ is a descendant of $u$ if and only if at time $u.d$ there is a path from $u$ to $v$ consisting of only white vertices (except for $u$, which was just colored gray)



# TOPOLOGICAL SORT

# Topological sort

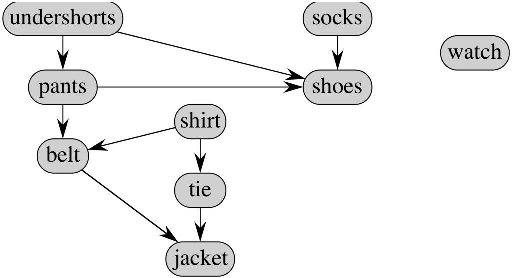
## Definition

INPUT: A directed acyclic graph (DAG) $G = (V, E)$

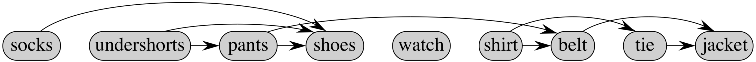
OUTPUT: a linear ordering of vertices such that if $(u, v) \in E$, then $u$ appears somewhere before $v$

# Example

Getting dressed in the morning:




in which order?



# When is a directed graph acyclic?

# When is a directed graph acyclic?

## Lemma

*A directed graph $G$ is acyclic if and only if a DFS of $G$ yields no back edges*

# When is a directed graph acyclic?

## Lemma

*A directed graph $G$ is acyclic if and only if a DFS of $G$ yields no back edges*

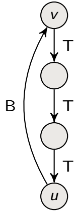
**Proof.** First show that back-edge implies cycle

# When is a directed graph acyclic?

## Lemma

*A directed graph $G$ is acyclic if and only if a DFS of $G$ yields no back edges*

**Proof.** First show that back-edge implies cycle
Suppose there is a back edge $(u, v)$. Then $v$ is ancestor of $u$ in depth-first forest. Therefore there is a path from $v$ to $u$, which creates a cycle.



# When is a directed graph acyclic?

## Lemma

*A directed graph $G$ is acyclic if and only if a DFS of $G$ yields no back edges*

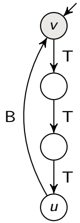
**Proof.** Second show that cycle implies back-edge

# When is a directed graph acyclic?

## Lemma

*A directed graph $G$ is acyclic if and only if a DFS of $G$ yields no back edges*

**Proof.** Second show that cycle implies back-edge
Let $v$ be the first vertex discovered in the cycle $C$ and let $(u, v)$ be the preceding edge in $C$. At time $v.d$ vertices in $C$ form a white-path from $v$ to $u$ and hence $u$ is a descendant of $v$.



# Algorithm for topological sort

# Algorithm for topological sort

TOPOLOGICAL-SORT($G$):

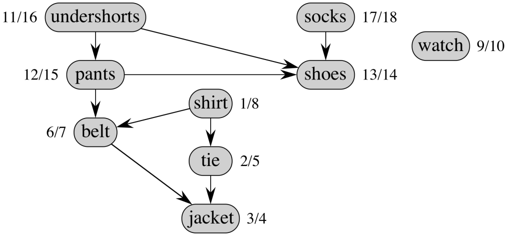
1. Call $DFS(G)$ to compute finishing times $v.f$ for all $v \in G.V$
2. Output vertices in order of *decreasing* finishing times

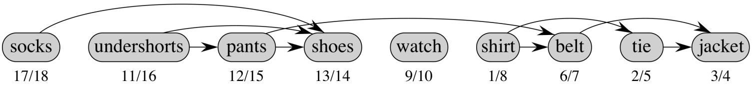
# Algorithm for topological sort

TOPOLOGICAL-SORT($G$):

1. Call $DFS(G)$ to compute finishing times $v.f$ for all $v \in G.V$
2. Output vertices in order of *decreasing* finishing times

## Example



# Time Analysis

TOPOLOGICAL-SORT($G$):

1. Call $DFS(G)$ to compute finishing times $v.f$ for all $v \in G.V$
2. Output vertices in order of *decreasing* finishing times

# Time Analysis

TOPOLOGICAL-SORT($G$):
1. Call $DFS(G)$ to compute finishing times $v.f$ for all $v \in G.V$
2. Output vertices in order of *decreasing* finishing times

Do not need to sort by finishing times

- Can just output vertices as they are finished and understand that we want the reverse of the list

# Time Analysis

TOPOLOGICAL-SORT($G$):
1. Call $DFS(G)$ to compute finishing times $v.f$ for all $v \in G.V$
2. Output vertices in order of *decreasing* finishing times

Do not need to sort by finishing times

- Can just output vertices as they are finished and understand that we want the reverse of the list
- Or put them onto the front of a linked list as they are finished. When done, the list contains vertices in topologically sorted order.

Time:

# Time Analysis

TOPOLOGICAL-SORT($G$):
1. Call $DFS(G)$ to compute finishing times $v.f$ for all $v \in G.V$
2. Output vertices in order of *decreasing* finishing times

Do not need to sort by finishing times

- Can just output vertices as they are finished and understand that we want the reverse of the list
- Or put them onto the front of a linked list as they are finished. When done, the list contains vertices in topologically sorted order.

Time: $\Theta(V + E)$ (same as DFS)

# Correctness

**Need to show that if $(u, v) \in E$ then $v.f < u.f$**

When we explore $(u, v)$ what are the colors of $u$ and $v$?

# Correctness

**Need to show that if $(u, v) \in E$ then $v.f < u.f$**

When we explore $(u, v)$ what are the colors of $u$ and $v$?

- $u$ is gray

# Correctness

**Need to show that if $(u, v) \in E$ then $v.f < u.f$**

When we explore $(u, v)$ what are the colors of $u$ and $v$?

- $u$ is gray
- Is $v$ gray, too?

# Correctness

**Need to show that if $(u, v) \in E$ then $v.f < u.f$**

When we explore $(u, v)$ what are the colors of $u$ and $v$?

- $u$ is gray
- Is $v$ gray, too?
  - No, because then $v$ would be ancestor of $u$ which implies that there is a back edge so the graph is not acyclic (by previous Lemma)

# Correctness

**Need to show that if $(u, v) \in E$ then $v.f < u.f$**

When we explore $(u, v)$ what are the colors of $u$ and $v$?

- $u$ is gray
- Is $v$ gray, too?
  - No, because then $v$ would be ancestor of $u$ which implies that there is a back edge so the graph is not acyclic (by previous Lemma)
- Is $v$ white?
  - Then becomes descendant of $u$. By parenthesis theorem, $u.d < v.d < v.f < u.f$

# Correctness

**Need to show that if $(u, v) \in E$ then $v.f < u.f$**

When we explore $(u, v)$ what are the colors of $u$ and $v$?

- $u$ is gray
- Is $v$ gray, too?
  - No, because then $v$ would be ancestor of $u$ which implies that there is a back edge so the graph is not acyclic (by previous Lemma)
- Is $v$ white?
  - Then becomes descendant of $u$. By parenthesis theorem, $u.d < v.d < v.f < u.f$
- Is $v$ black?
  - Then $v$ is already finished. Since we are exploring $(u, v)$, we have not yet finished $u$. Therefore, $v.f < u.f$.

# Correctness

**Need to show that if $(u, v) \in E$ then $v.f < u.f$**

When we explore $(u, v)$ what are the colors of $u$ and $v$?

- $u$ is gray
- Is $v$ gray, too?
  - No, because then $v$ would be ancestor of $u$ which implies that there is a back edge so the graph is not acyclic (by previous Lemma)
- Is $v$ white?
  - Then becomes descendant of $u$. By parenthesis theorem, $u.d < v.d < v.f < u.f$
- Is $v$ black?
  - Then $v$ is already finished. Since we are exploring $(u, v)$, we have not yet finished $u$. Therefore, $v.f < u.f$.

# Summary

- Graphs fundamental object to study
- Representation either by adjacency list or adjacency matrix
- Two natural ways of traversing a graph: breadth-first search and depth-first search
- Topological sort of acyclic graphs by applying DFS and then order according to decreasing finishing times