Graphs: Representation and Elementary Algorithms

Antonio Carzaniga

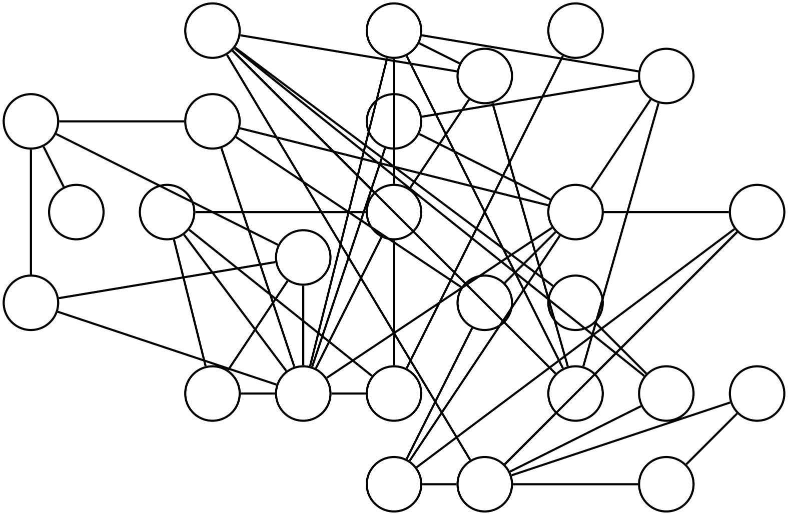
Faculty of Informatics
Università della Svizzera italiana

April 27, 2017

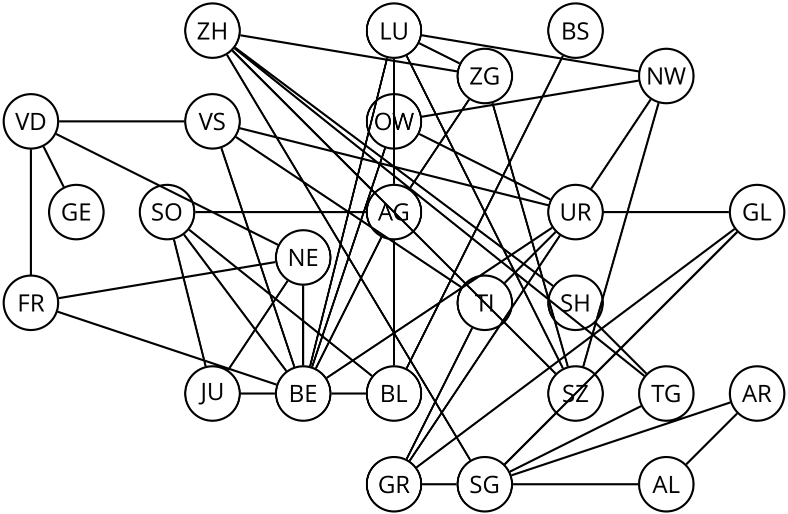
Outline

- Graphs: definitions
- Representations
- Breadth-first search
- Depth-first search

Example

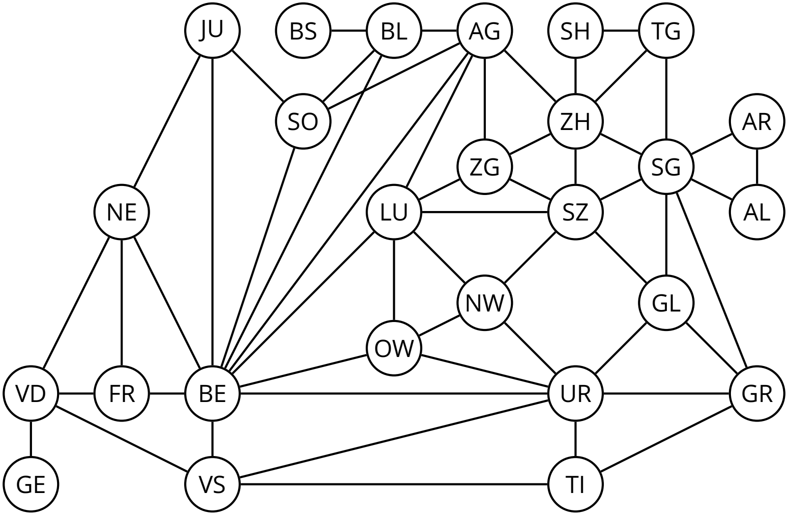


Example



ZH
LU
BS
ZG
NW
VD
VS
OW
GE
SO
AG
UR
GL
NE
FR
TI
SH
JU
BE
BL
SZ
TG
AR
GR
SG
AL

Same Example (Better Layout)



Many Models and Applications

- Social networks: *who knows who*
- The Web graph: *which page links to which*
- The Internet graph: *which router links to which*
- Citation graphs: *who references whose papers*
- Planar graphs: *which country is next to which*
- Well-shaped meshes: *pretty pictures with triangles*
- Geometric graphs: *who is near who*
- Random graphs: *whichever…*

Examples and descriptions taken from Daniel A. Spielman's course "Graphs and Networks."

- A ***graph***

$$G = (V, E)$$

- V is the set of ***vertices*** (also called ***nodes***)
- E is the set of ***edges***

Definitions

- A ***graph***

$$G = (V, E)$$

- V is the set of ***vertices*** (also called ***nodes***)

- E is the set of ***edges***
  - $E \subseteq V \times V$, i.e., E is a ***relation between vertices***
  - an edge $e = (u, v) \in V$ is a pair of vertices $u \in V$ and $v \in V$

- A ***graph***

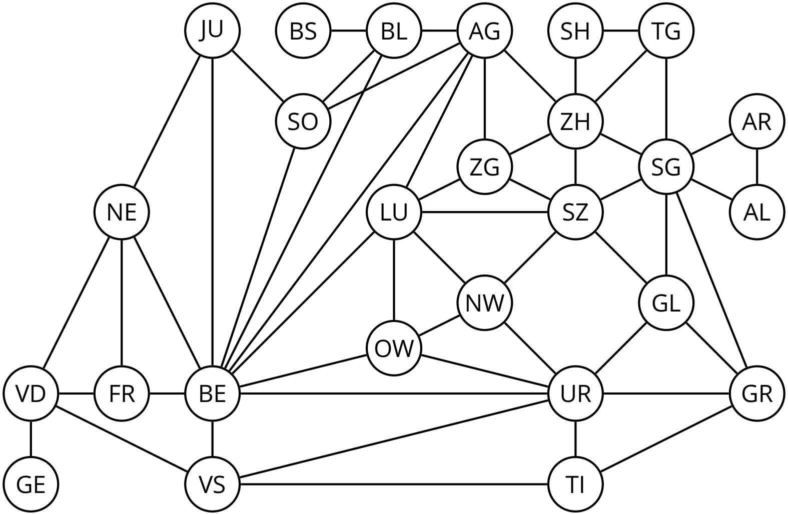
$$G = (V, E)$$

- V is the set of ***vertices*** (also called ***nodes***)

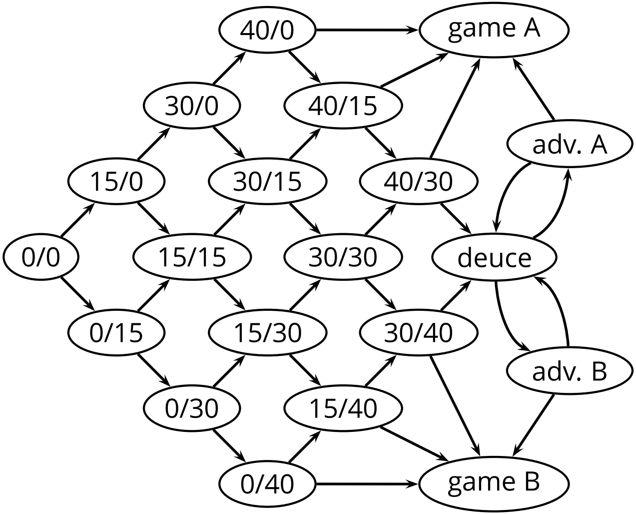
- E is the set of ***edges***
  - $E \subseteq V \times V$, i.e., E is a ***relation between vertices***
  - an edge $e = (u, v) \in V$ is a pair of vertices $u \in V$ and $v \in V$

- An *undirected* graph is characterized by a *symmetric* relation between vertices
  - an edge is a set $e = \{u, v\}$ of two vertices

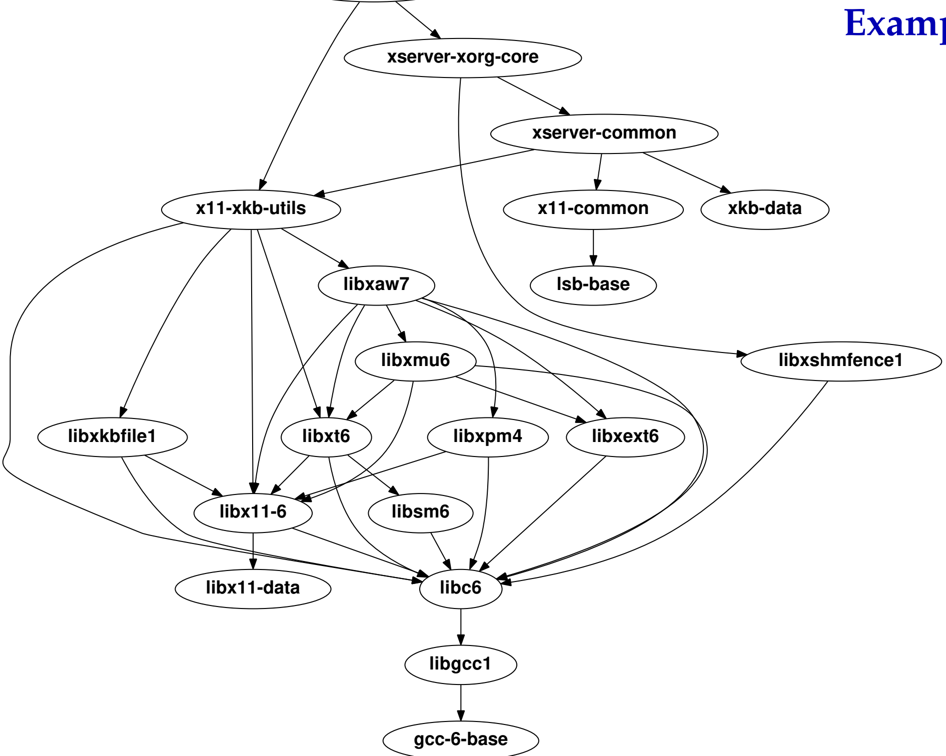
Example (1)


JU
BS
BL
AG
SH
TG
SO
ZH
AR
ZG
SG
NE
LU
SZ
AL
NW
GL
OW
VD
FR
BE
UR
GR
GE
VS
TI

Example (2)


40/0
game A
30/0
40/15
adv. A
15/0
30/15
40/30
0/0
15/15
30/30
deuce
0/15
15/30
30/40
adv. B
0/30
15/40
0/40
game B

Example (3)


xserver-xorg-core
xserver-common
x11-xkb-utils
x11-common
xkb-data
libxaw7
lsb-base
libxmu6
libxshmfence1
libxkbfile1
libxt6
libxpm4
libxext6
libx11-6
libsm6
libx11-data
libc6
libgcc1
gcc-6-base

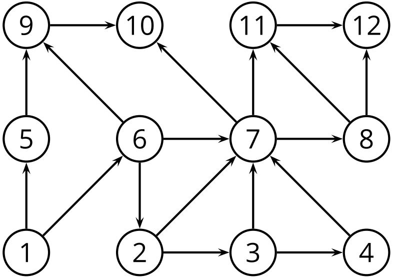
- How do we represent a graph $G = (E, V)$ in a computer?

- How do we represent a graph $G = (E, V)$ in a computer?
- *Adjacency-list representation*
- $V = \{1, 2, \ldots |V|\}$
- G consists of an array Adj
- A vertex $u \in V$ is represented by an element in the array Adj

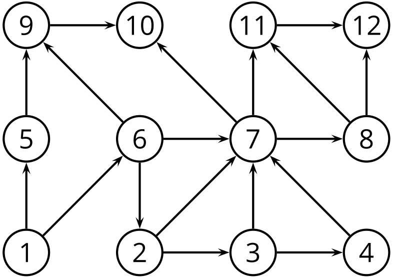
Graph Representation

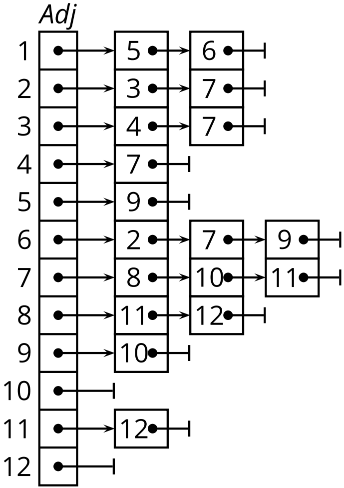
- How do we represent a graph $G = (E, V)$ in a computer?
- *Adjacency-list representation*
- $V = \{1, 2, \ldots |V|\}$
- G consists of an array Adj
- A vertex $u \in V$ is represented by an element in the array Adj
- $Adj[u]$ is the ***adjacency list*** of vertex u
  - the list of the vertices that are adjacent to u
  - i.e., the list of all v such that $(u, v) \in E$

Example

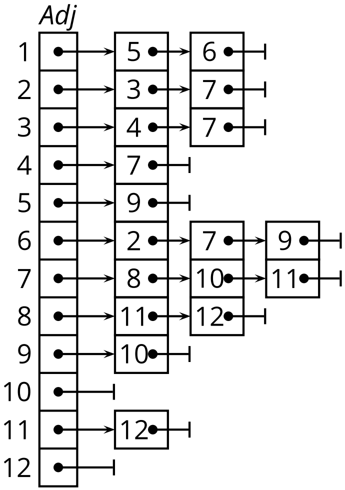


Example

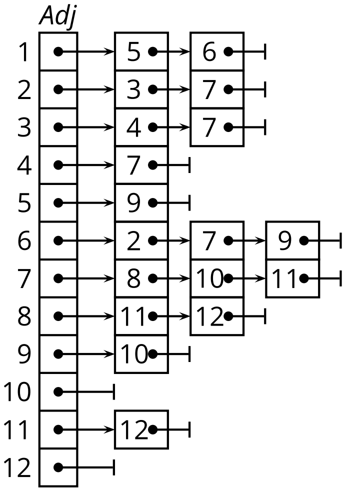

9
10
11
12
5
6
7
8
1
2
3
4


Adj
1 5 6
2 3 7
3 4 7
4 7
5 9
6 2 7 9
7 8 10 11
8 11 12
9 10
10
11 12
12

Using the Adjacency List

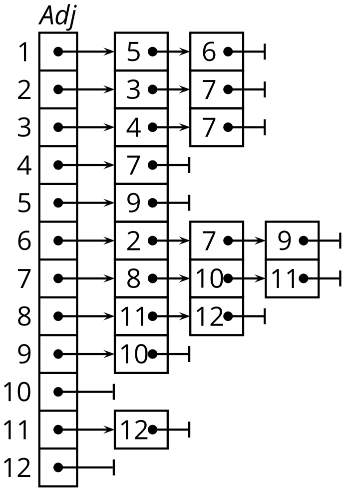


- Accessing a vertex *u*?



Using the Adjacency List

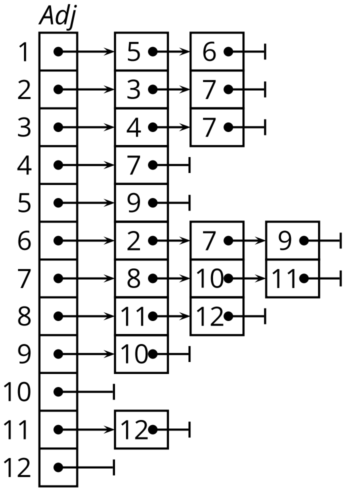
- Accessing a vertex *u*? $O(1)$
  - optimal



Using the Adjacency List

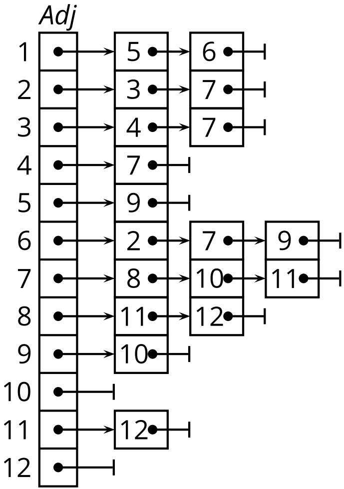
- Accessing a vertex *u*? $O(1)$
  - optimal

- Iteration through *V*?



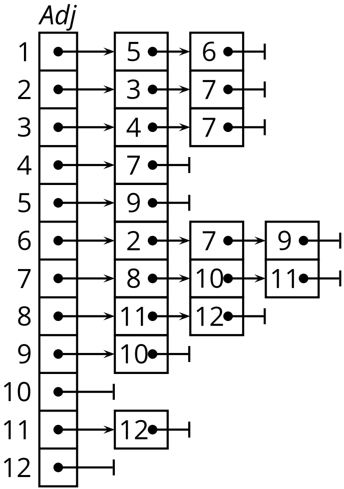
Using the Adjacency List

- Accessing a vertex u? $O(1)$
  - optimal
- Iteration through V? $\Theta(|V|)$
  - optimal



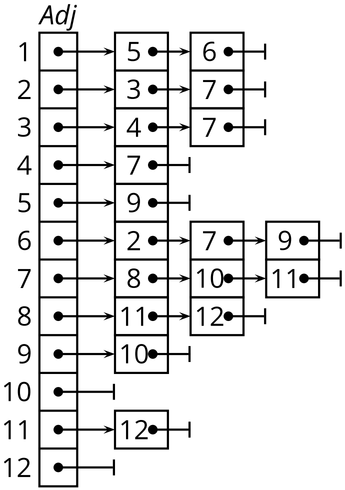
Using the Adjacency List

- Accessing a vertex u? $O(1)$
  - optimal
- Iteration through V? $\Theta(|V|)$
  - optimal
- Iteration through E?

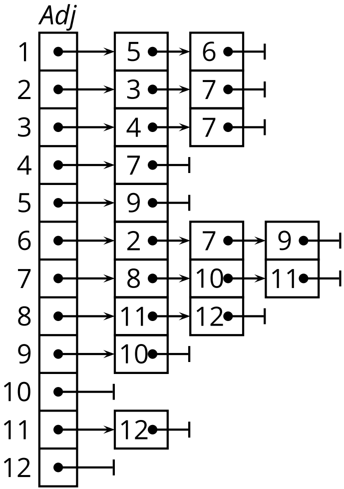


Using the Adjacency List

- Accessing a vertex u? $O(1)$
  - optimal
- Iteration through V? $\Theta(|V|)$
  - optimal
- Iteration through E? $\Theta(|V| + |E|)$
  - okay (not optimal)

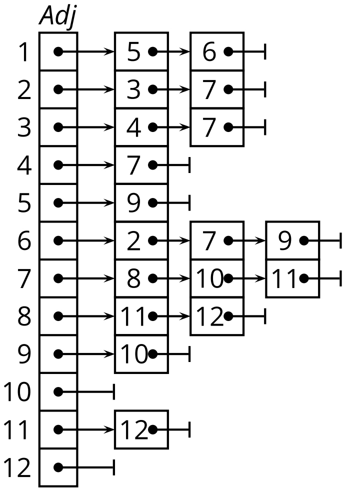


- Accessing a vertex u? $O(1)$
  - optimal
- Iteration through V? $\Theta(|V|)$
  - optimal
- Iteration through E? $\Theta(|V| + |E|)$
  - okay (not optimal)
- Checking $(u, v) \in E$?



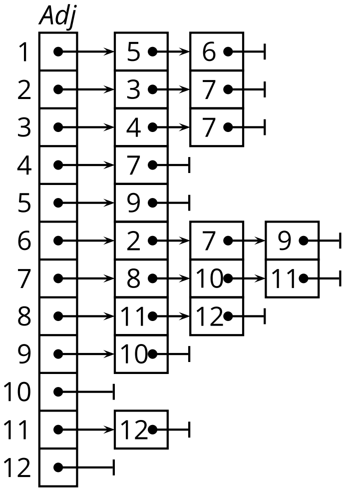
Using the Adjacency List

- Accessing a vertex u? $O(1)$
  - optimal
- Iteration through V? $\Theta(|V|)$
  - optimal
- Iteration through E? $\Theta(|V| + |E|)$
  - okay (not optimal)
- Checking $(u, v) \in E$? $O(|V|)$



Using the Adjacency List

- Accessing a vertex u? $O(1)$
  - optimal
- Iteration through V? $\Theta(|V|)$
  - optimal
- Iteration through E? $\Theta(|V| + |E|)$
  - okay (not optimal)
- Checking $(u, v) \in E$? $O(|V|)$
  - bad

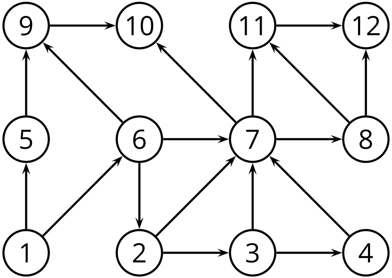


Graph Representation (2)

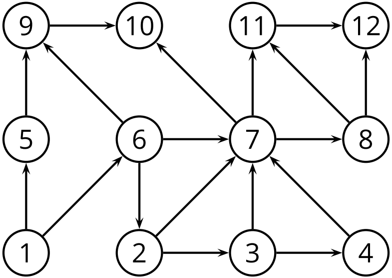
- *Adjacency-matrix representation*
- $V = \{1, 2, \ldots |V|\}$
- G consists of a $|V| \times |V|$ matrix A
- $A = (a_{ij})$ such that

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

Example

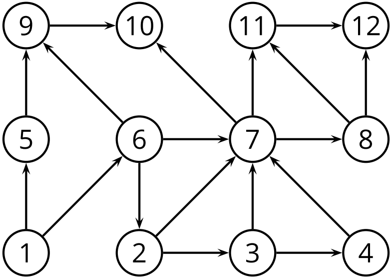


Example



	1	2	3	4	5	6	7	8	9	10	11	12
1												
2												
3												
4												
5												
6												
7												
8												
9												
10												
11												
12												

Example



	1	2	3	4	5	6	7	8	9	10	11	12
1					1	1						
2			1				1					
3				1			1					
4							1					
5									1			
6		1					1		1			
7								1		1	1	
8											1	1
9										1		
10												
11												1
12												

Using the Adjacency Matrix

	1	2	3	4	5	6	7	8	9	10	11	12
1					1	1						
2			1				1					
3				1			1					
4							1					
5									1			
6		1					1		1			
7								1		1	1	
8											1	1
9										1		
10												
11												1
12												

Using the Adjacency Matrix

- Accessing a vertex *u*?

	1	2	3	4	5	6	7	8	9	10	11	12
1					1	1						
2			1				1					
3				1			1					
4							1					
5									1			
6		1					1		1			
7								1		1	1	
8											1	1
9										1		
10												
11												1
12												

Using the Adjacency Matrix

- Accessing a vertex u? $O(1)$
  - optimal

	1	2	3	4	5	6	7	8	9	10	11	12
1					1	1						
2			1				1					
3				1			1					
4							1					
5									1			
6		1					1		1			
7								1		1	1	
8											1	1
9										1		
10												
11												1
12												

Using the Adjacency Matrix

- Accessing a vertex *u*? $O(1)$
  - optimal

- Iteration through *V*?

	1	2	3	4	5	6	7	8	9	10	11	12
1					1	1						
2			1				1					
3				1			1					
4							1					
5									1			
6		1					1		1			
7								1		1	1	
8											1	1
9										1		
10												
11												1
12												

Using the Adjacency Matrix

- Accessing a vertex u? $O(1)$
  - optimal
- Iteration through V? $\Theta(|V|)$
  - optimal

	1	2	3	4	5	6	7	8	9	10	11	12
1					1	1						
2			1				1					
3				1			1					
4							1					
5									1			
6		1					1		1			
7								1		1	1	
8											1	1
9										1		
10												
11												1
12												

Using the Adjacency Matrix

- Accessing a vertex u? $O(1)$
  - optimal
- Iteration through V? $\Theta(|V|)$
  - optimal
- Iteration through E?

	1	2	3	4	5	6	7	8	9	10	11	12
1					1	1						
2			1				1					
3				1			1					
4							1					
5									1			
6		1					1		1			
7								1		1	1	
8											1	1
9										1		
10												
11												1
12												

Using the Adjacency Matrix

- Accessing a vertex u? $O(1)$
  - optimal
- Iteration through V? $\Theta(|V|)$
  - optimal
- Iteration through E? $\Theta(|V|^2)$
  - possibly very bad

	1	2	3	4	5	6	7	8	9	10	11	12
1					1	1						
2			1				1					
3				1			1					
4							1					
5									1			
6		1					1		1			
7								1		1	1	
8											1	1
9										1		
10												
11												1
12												

Using the Adjacency Matrix

- Accessing a vertex u? $O(1)$
  - optimal
- Iteration through V? $\Theta(|V|)$
  - optimal
- Iteration through E? $\Theta(|V|^2)$
  - possibly very bad
- Checking $(u, v) \in E$?

	1	2	3	4	5	6	7	8	9	10	11	12
1					1	1						
2			1				1					
3				1			1					
4							1					
5									1			
6		1					1		1			
7								1		1	1	
8											1	1
9										1		
10												
11												1
12												

Using the Adjacency Matrix

- Accessing a vertex u? $O(1)$
  - optimal
- Iteration through V? $\Theta(|V|)$
  - optimal
- Iteration through E? $\Theta(|V|^2)$
  - possibly very bad
- Checking $(u, v) \in E$? $O(1)$

	1	2	3	4	5	6	7	8	9	10	11	12
1					1	1						
2			1				1					
3				1			1					
4							1					
5									1			
6		1					1		1			
7								1		1	1	
8											1	1
9										1		
10												
11												1
12												

Using the Adjacency Matrix

- Accessing a vertex u? — $O(1)$
  - optimal
- Iteration through V? — $\Theta(|V|)$
  - optimal
- Iteration through E? — $\Theta(|V|^2)$
  - possibly very bad
- Checking $(u, v) \in E$? — $O(1)$
  - optimal

	1	2	3	4	5	6	7	8	9	10	11	12
1					1	1						
2			1				1					
3				1			1					
4							1					
5									1			
6		1					1		1			
7								1		1	1	
8											1	1
9										1		
10												
11												1
12												

Space Complexity

- Adjacency-list representation

Space Complexity

- Adjacency-list representation

$$\boxed{\Theta(|V| + |E|)}$$

Space Complexity

- Adjacency-list representation

$$\boxed{\Theta(|V| + |E|)}$$

optimal

Space Complexity

- Adjacency-list representation

$$\boxed{\Theta(|V| + |E|)}$$

  optimal

- Adjacency-matrix representation

- Adjacency-list representation

$$\boxed{\Theta(|V| + |E|)}$$

optimal

- Adjacency-matrix representation

$$\boxed{\Theta(|V|^2)}$$

Space Complexity

- Adjacency-list representation

$$\boxed{\Theta(|V| + |E|)}$$

optimal

- Adjacency-matrix representation

$$\boxed{\Theta(|V|^2)}$$

possibly very bad

- Adjacency-list representation

$$\Theta(|V| + |E|)$$

  optimal

- Adjacency-matrix representation

$$\Theta(|V|^2)$$

  possibly very bad

- When is the adjacency-matrix "very bad"?

Choosing a Graph Representation

- Adjacency-list representation
  - generally good, especially for its optimal space complexity
  - bad for ***dense*** graphs and algorithms that require random access to edges
  - preferable for ***sparse*** graphs or graphs with ***low degree***

Choosing a Graph Representation

- Adjacency-list representation
  - generally good, especially for its optimal space complexity
  - bad for ***dense*** graphs and algorithms that require random access to edges
  - preferable for ***sparse*** graphs or graphs with ***low degree***

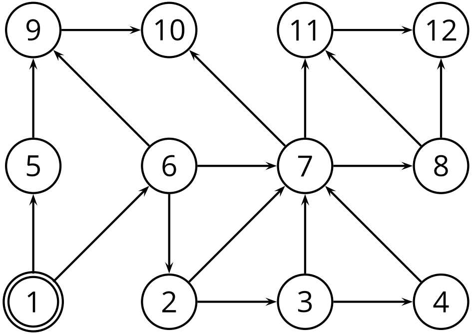
- Adjacency-matrix representation
  - suffers from a bad space complexity
  - good for algorithms that require random access to edges
  - preferable for ***dense*** graphs

Breadth-First Search

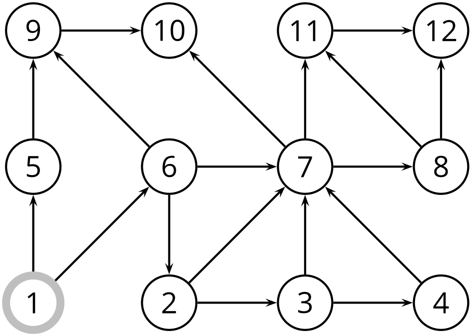
- One of the simplest but also a fundamental algorithm

- One of the simplest but also a fundamental algorithm
- *Input:* $G = (V, E)$ and a vertex $s \in V$
  - explores the graph, touching all vertices that are reachable from s
  - iterates through the vertices at increasing distance (edge distance)
  - computes the distance of each vertex from s
  - produces a ***breadth-first tree*** rooted at s
  - works on both *directed* and *undirected* graphs

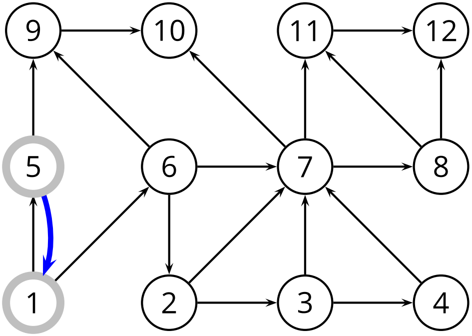
Example



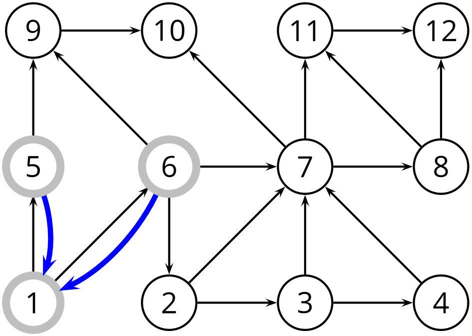
Example



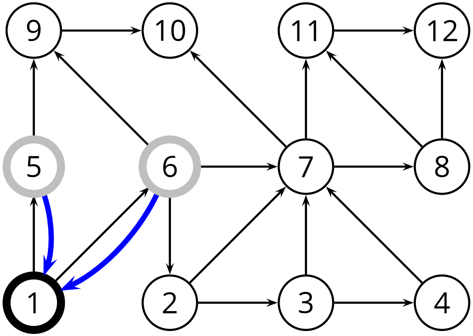
Example



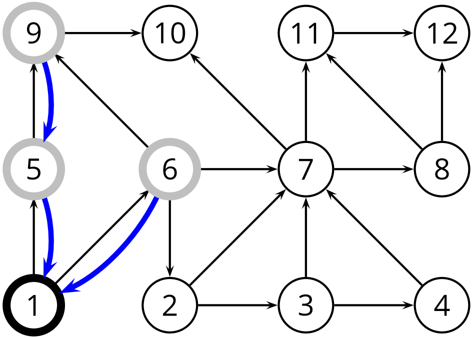
Example



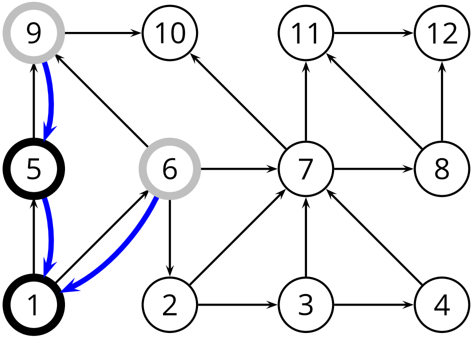
Example



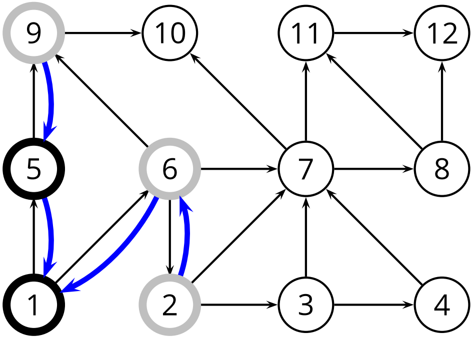
Example



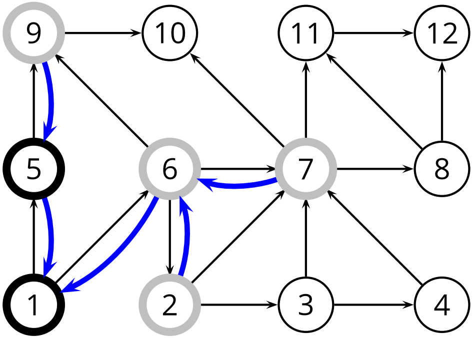
Example



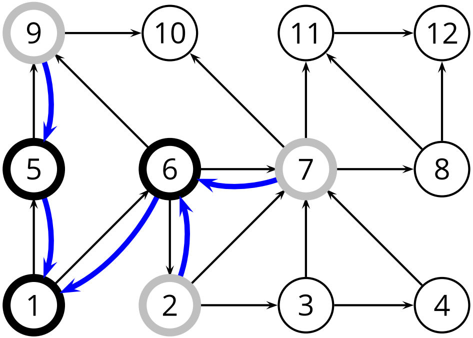
Example

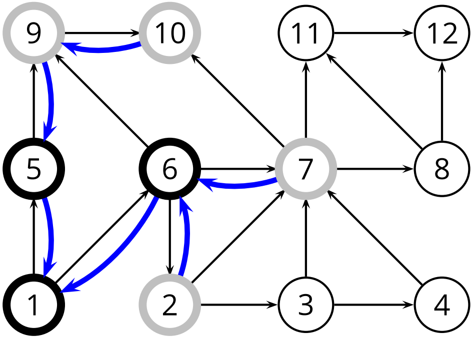


Example

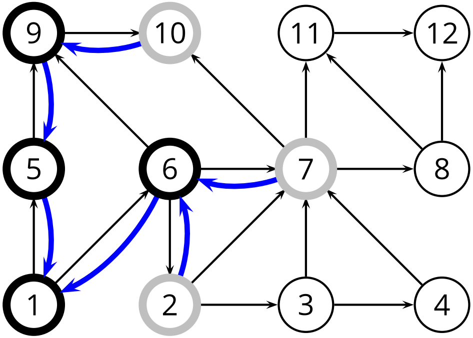


Example

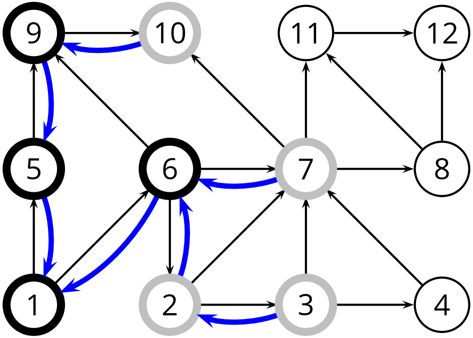



9
10
11
12
5
6
7
8
1
2
3
4

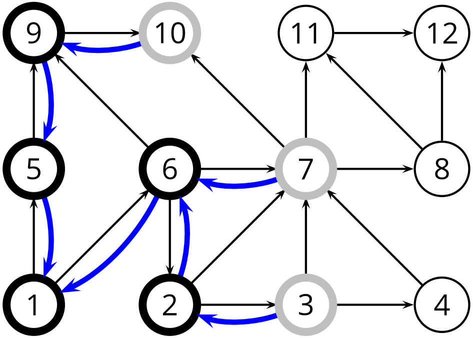
Example



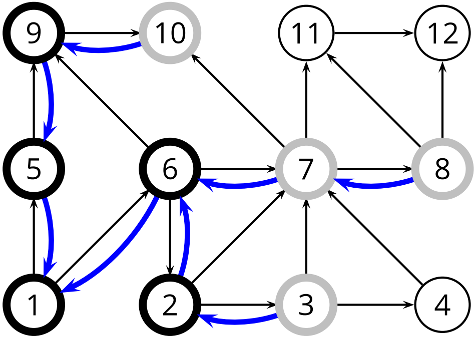
Example

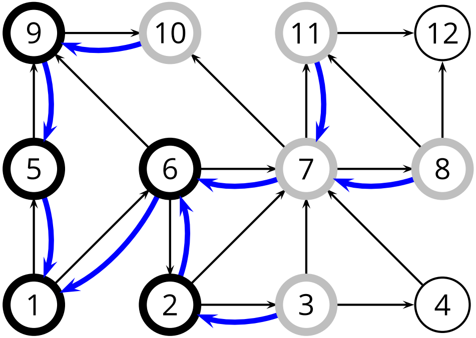


Example

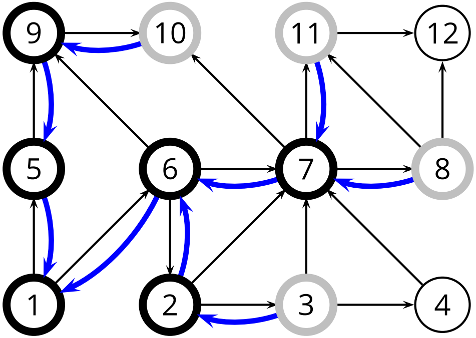


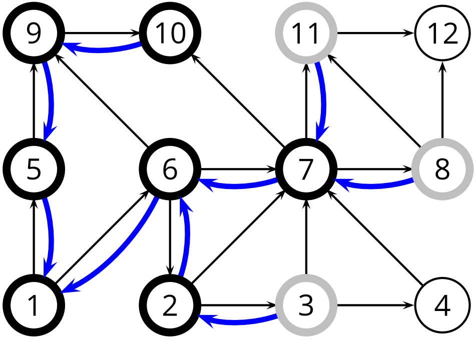
Example



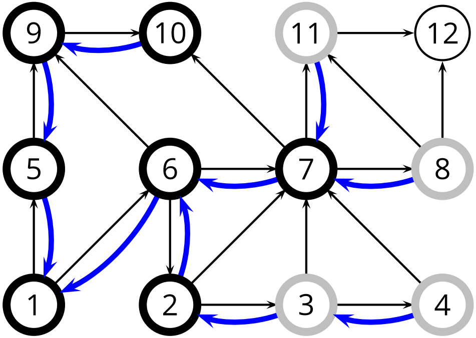

9
10
11
12
5
6
7
8
1
2
3
4

Example

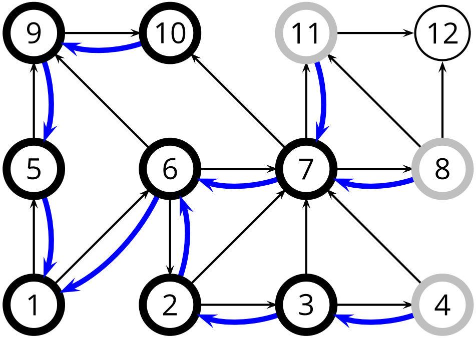



9
10
11
12
5
6
7
8
1
2
3
4

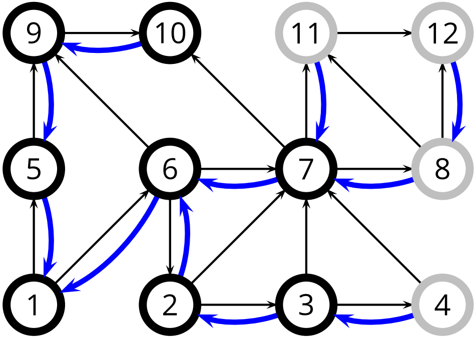
Example



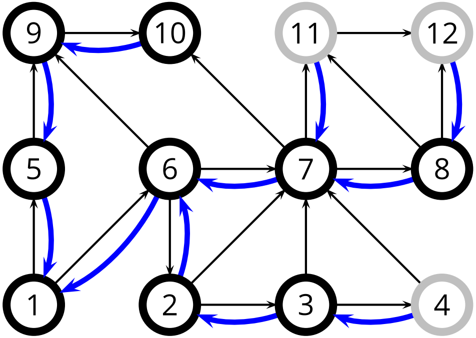
Example



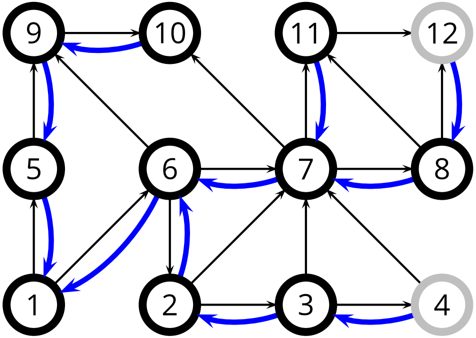
Example



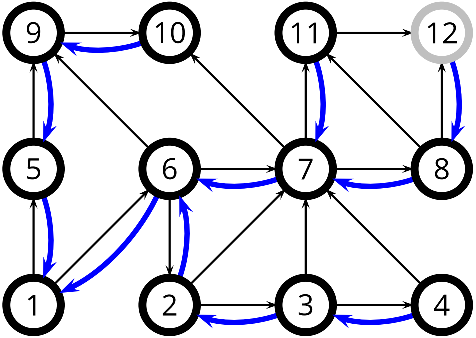
Example



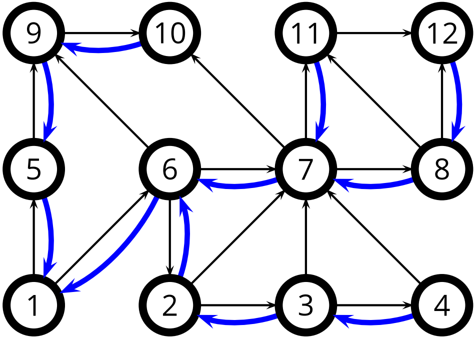
Example



Example

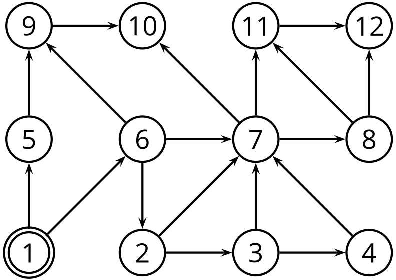


Example



BFS Algorithm

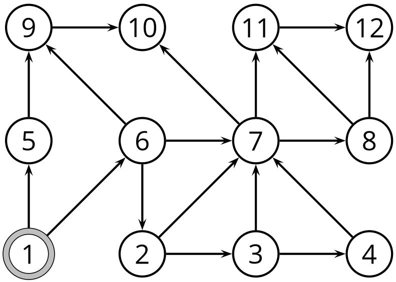
```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```



BFS Algorithm

BFS(G, s)

```
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```

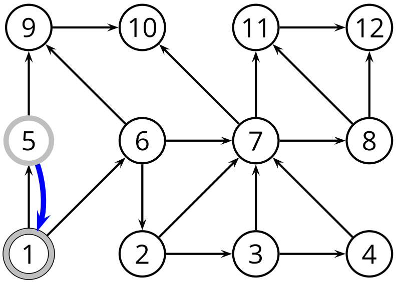


$u = 1$

$Q = \emptyset$

BFS Algorithm

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```

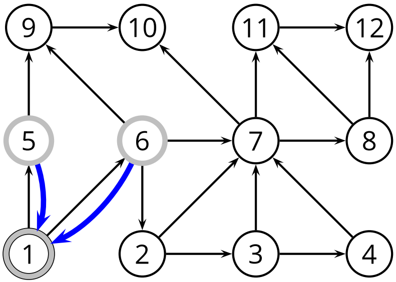


$u = 1$

$Q = \{5\}$

BFS Algorithm

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```

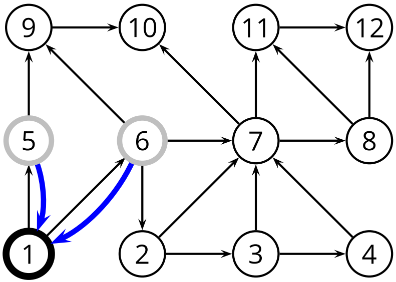


$u = 1$

$Q = \{5, 6\}$

BFS Algorithm

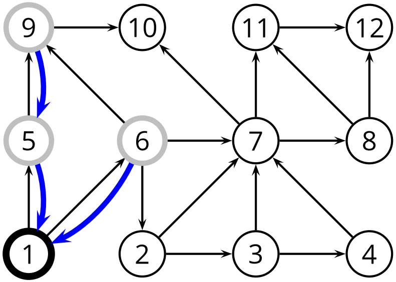
```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```



$u = 5$

$Q = \{6\}$

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
        color[u] = WHITE
        d[u] = ∞
        π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
        u = DEQUEUE(Q)
        for each v ∈ Adj[u]
                if color[v] == WHITE
                        color[v] = GRAY
                        d[v] = d[u] + 1
                        π[v] = u
                        ENQUEUE(Q, v)
        color[u] = BLACK
```

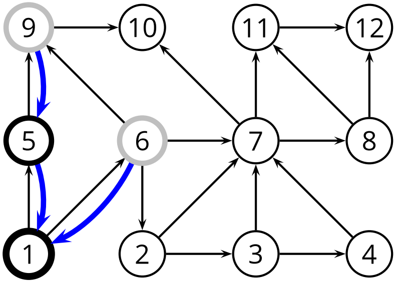


$u = 5$

$Q = \{6, 9\}$

BFS Algorithm

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```

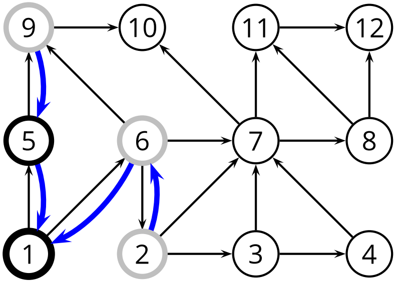


$u = 6$

$Q = \{9\}$

BFS Algorithm

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```

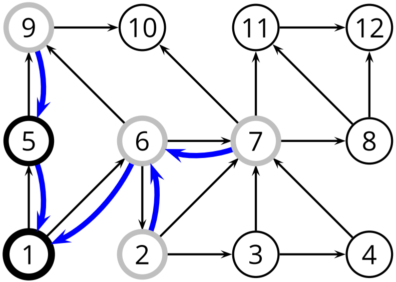


$u = 6$

$Q = \{9, 2, 7\}$

BFS Algorithm

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
        color[u] = WHITE
        d[u] = ∞
        π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
        u = DEQUEUE(Q)
        for each v ∈ Adj[u]
                if color[v] == WHITE
                        color[v] = GRAY
                        d[v] = d[u] + 1
                        π[v] = u
                        ENQUEUE(Q, v)
        color[u] = BLACK
```

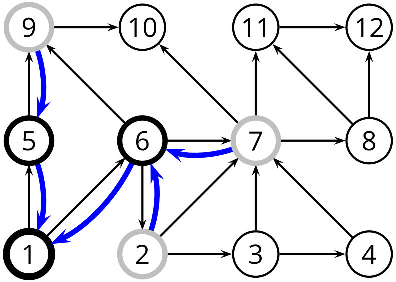


$u = 6$

$Q = \{9, 2, 7\}$

BFS Algorithm

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
        color[u] = WHITE
        d[u] = ∞
        π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
        u = DEQUEUE(Q)
        for each v ∈ Adj[u]
                if color[v] == WHITE
                        color[v] = GRAY
                        d[v] = d[u] + 1
                        π[v] = u
                        ENQUEUE(Q, v)
        color[u] = BLACK
```

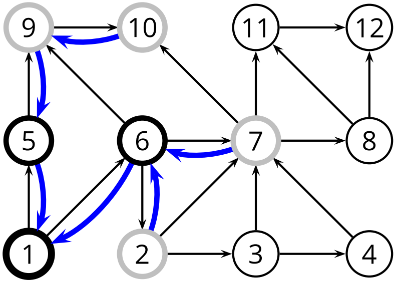


$u = 9$

$Q = \{2, 7\}$

BFS Algorithm

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```

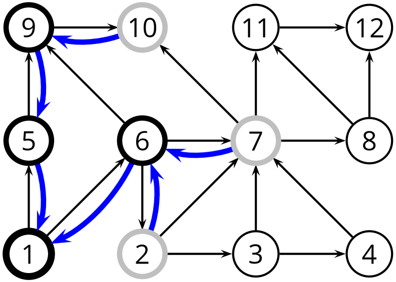


$u = 9$

$Q = \{2, 7, 10\}$

BFS Algorithm

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
        color[u] = WHITE
        d[u] = ∞
        π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
        u = DEQUEUE(Q)
        for each v ∈ Adj[u]
                if color[v] == WHITE
                        color[v] = GRAY
                        d[v] = d[u] + 1
                        π[v] = u
                        ENQUEUE(Q, v)
        color[u] = BLACK
```

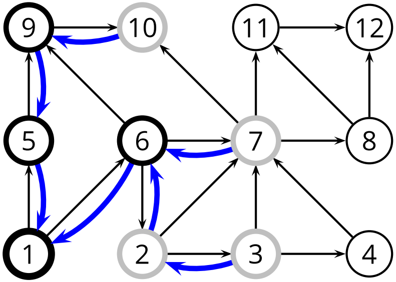


$u = 2$

$Q = \{7, 10\}$

BFS Algorithm

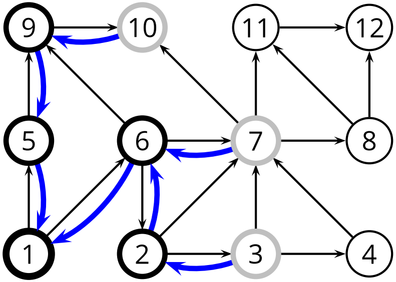
```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```



$u = 2$

$Q = \{7, 10, 3\}$

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```

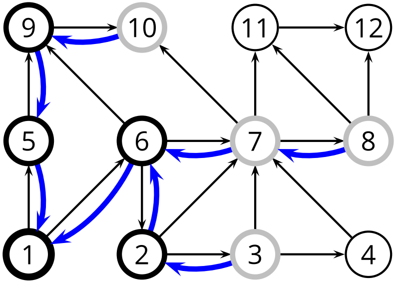


$u = 7$

$Q = \{10, 3\}$

BFS Algorithm

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```

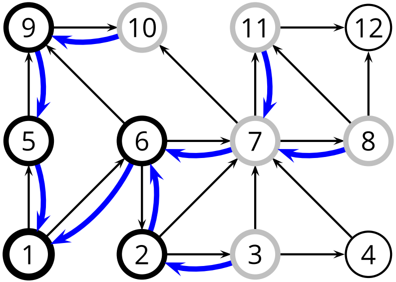


$u = 7$

$Q = \{10, 3, 8\}$

BFS Algorithm

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
        color[u] = WHITE
        d[u] = ∞
        π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
        u = DEQUEUE(Q)
        for each v ∈ Adj[u]
                if color[v] == WHITE
                        color[v] = GRAY
                        d[v] = d[u] + 1
                        π[v] = u
                        ENQUEUE(Q, v)
        color[u] = BLACK
```

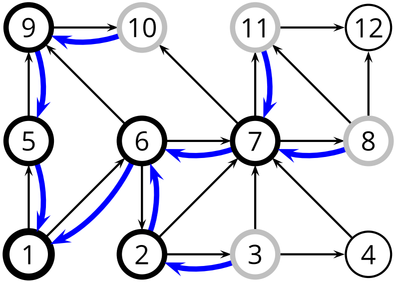


$u = 7$

$Q = \{10, 3, 8, 11\}$

BFS Algorithm

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```

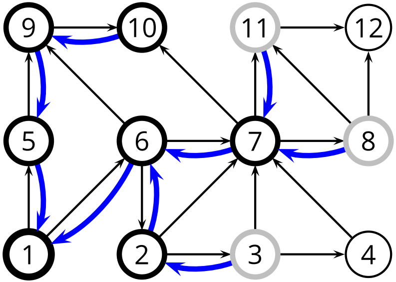


$u = 10$

$Q = \{3, 8, 11\}$

BFS Algorithm

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```

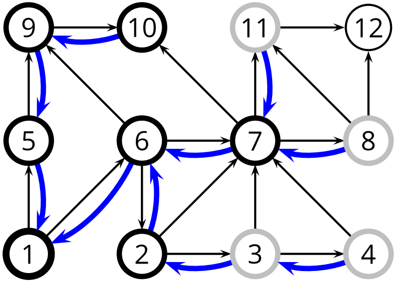


$u = 3$

$Q = \{8, 11\}$

BFS Algorithm

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
        color[u] = WHITE
        d[u] = ∞
        π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
        u = DEQUEUE(Q)
        for each v ∈ Adj[u]
                if color[v] == WHITE
                        color[v] = GRAY
                        d[v] = d[u] + 1
                        π[v] = u
                        ENQUEUE(Q, v)
        color[u] = BLACK
```

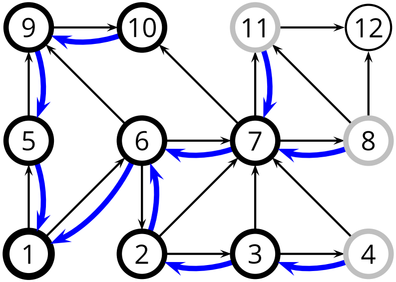


$u = 3$

$Q = \{8, 11, 4\}$

BFS Algorithm

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```

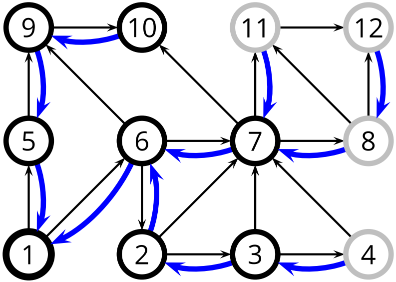


$u = 8$

$Q = \{11, 4\}$

BFS Algorithm

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```

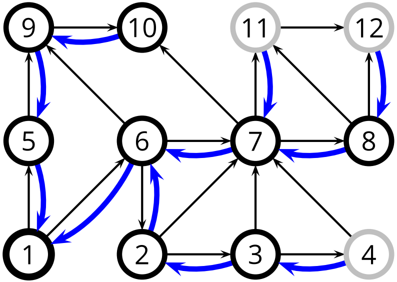


$u = 8$

$Q = \{11, 4, 12\}$

BFS Algorithm

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
        color[u] = WHITE
        d[u] = ∞
        π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
        u = DEQUEUE(Q)
        for each v ∈ Adj[u]
                if color[v] == WHITE
                        color[v] = GRAY
                        d[v] = d[u] + 1
                        π[v] = u
                        ENQUEUE(Q, v)
        color[u] = BLACK
```

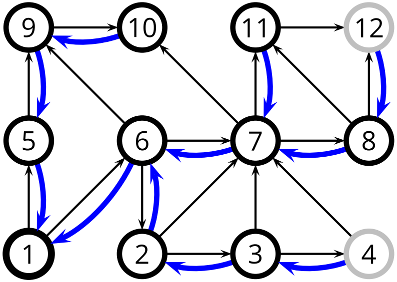


$u = 11$

$Q = \{4, 12\}$

BFS Algorithm

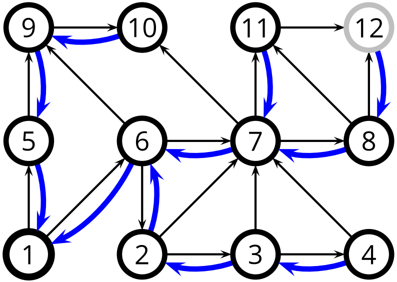
```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```



$u = 4$

$Q = \{12\}$

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
        color[u] = WHITE
        d[u] = ∞
        π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
        u = DEQUEUE(Q)
        for each v ∈ Adj[u]
                if color[v] == WHITE
                        color[v] = GRAY
                        d[v] = d[u] + 1
                        π[v] = u
                        ENQUEUE(Q, v)
        color[u] = BLACK
```

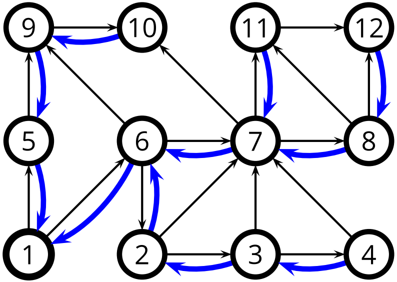


$u = 12$

$Q = \emptyset$

BFS Algorithm

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
        color[u] = WHITE
        d[u] = ∞
        π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
        u = DEQUEUE(Q)
        for each v ∈ Adj[u]
                if color[v] == WHITE
                        color[v] = GRAY
                        d[v] = d[u] + 1
                        π[v] = u
                        ENQUEUE(Q, v)
        color[u] = BLACK
```



Complexity of BFS

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```

Complexity of BFS

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```

- We enqueue a vertex only if it is white, and we immediately color it gray; thus, we enqueue every vertex *at most once*

Complexity of BFS

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```

- We enqueue a vertex only if it is white, and we immediately color it gray; thus, we enqueue every vertex *at most once*
- So, the (dequeue) while loop executes $O(|V|)$ times

Complexity of BFS

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```

- We enqueue a vertex only if it is white, and we immediately color it gray; thus, we enqueue every vertex *at most once*
- So, the (dequeue) while loop executes $O(|V|)$ times
- For each vertex u, the inner loop executes $\Theta(|E_u|)$, for a total of $O(|E|)$ steps

Complexity of BFS

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
        color[u] = WHITE
        d[u] = ∞
        π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
        u = DEQUEUE(Q)
        for each v ∈ Adj[u]
                if color[v] == WHITE
                        color[v] = GRAY
                        d[v] = d[u] + 1
                        π[v] = u
                        ENQUEUE(Q, v)
        color[u] = BLACK
```

- We enqueue a vertex only if it is white, and we immediately color it gray; thus, we enqueue every vertex *at most once*
- So, the (dequeue) while loop executes $O(|V|)$ times
- For each vertex u, the inner loop executes $\Theta(|E_u|)$, for a total of $O(|E|)$ steps
- So, $O(|V| + |E|)$

Depth-First Search

- Immediately follow the links of the most recently-visited vertex, then backtrack when you reach a dead-end
  - i.e., backtrack when the current vertex has no more adjacent vertices that have not yet been visited

Depth-First Search

- Immediately follow the links of the most recently-visited vertex, then backtrack when you reach a dead-end
  - i.e., backtrack when the current vertex has no more adjacent vertices that have not yet been visited

- *Input:* $G = (V, E)$
  - explores the graph, touching *all vertices*

- Immediately follow the links of the most recently-visited vertex, then backtrack when you reach a dead-end
  - i.e., backtrack when the current vertex has no more adjacent vertices that have not yet been visited
- *Input:* $G = (V, E)$
  - explores the graph, touching *all vertices*
  - produces a ***depth-first forest***, consisting of all the ***depth-first trees*** defined by the DFS exploration

- Immediately follow the links of the most recently-visited vertex, then backtrack when you reach a dead-end
  - i.e., backtrack when the current vertex has no more adjacent vertices that have not yet been visited
- *Input:* $G = (V, E)$
  - explores the graph, touching *all vertices*
  - produces a ***depth-first forest***, consisting of all the ***depth-first trees*** defined by the DFS exploration
  - associates ***two time-stamps*** to each vertex
    - $d[u]$ records when u is first discovered
    - $f[u]$ records when DFS finishes examining u's edges, and therefore backtracks from u

DFS Algorithm

```
DFS(G)
for each vertex u ∈ V(G)
        color[u] = WHITE
        π[u] = NIL
time = 0 // "global" variable
for each vertex u ∈ V(G)
        if color[u] == WHITE
                DFS-VISIT(u)
```

```
DFS-VISIT(u)
color[u] = GREY
time = time + 1
d[u] = time
for each v ∈ Adj[u]
        if color[v] == WHITE
                π[v] = u
                DFS-VISIT(v)
color[u] = BLACK
time = time + 1
f[u] = time
```

Complexity of DFS

Complexity of DFS

- The loop in **DFS-VISIT**(u) (lines 4–7) accounts for $\Theta(|E_u|)$

- The loop in **DFS-VISIT**(u) (lines 4–7) accounts for $\Theta(|E_u|)$

- We call **DFS-VISIT**(u) *once* for each vertex u
  - either in **DFS**, or recursively in **DFS-VISIT**
  - because we call it only if $color[u] = \text{WHITE}$, but then we immediately set $color[u] = \text{GREY}$

Complexity of DFS

- The loop in **DFS-VISIT**(u) (lines 4–7) accounts for $\Theta(|E_u|)$

- We call **DFS-VISIT**(u) *once* for each vertex u

  - either in **DFS**, or recursively in **DFS-VISIT**
  - because we call it only if $color[u] = $ WHITE, but then we immediately set $color[u] = $ GREY

- So, the overall complexity is $\Theta(|V| + |E|)$

Applications of DFS: Topological Sort

Applications of DFS: Topological Sort

- Given a *directed acyclic graph* (DAG)
  - find an ordering of vertices such that you only end up with *forward links*

Applications of DFS: Topological Sort

- Given a *directed acyclic graph* (DAG)
  - find an ordering of vertices such that you only end up with *forward links*

- Example: dependencies in software packages
  - find an installation order for a set of software packages
  - such that every package is installed only after all the packages it depends on

Topological Sort Algorithm

Topological Sort Algorithm

```
TOPOLOGICAL-SORT(G)
1   DFS(G)
2   output V sorted in reverse order of f[·]
```

TOPOLOGICAL-SORT(G)

1 **DFS**(G)
2 output V sorted in reverse order of $f[\cdot]$