Algorithms and Data Structures (II)

Gabriel Istrate

May 6, 2020

Where are we:

Why did we want dynamic sets to start with ?

- Wrap up Computational geometry
- Graph algorithms.
- Goal: To see some algorithms that use stacks, queues, red-black trees.

Jarvis's March

- uses a technique known as gift wrapping.
- Simulates wrapping a piece of paper around set Q.
- Start at the same point p_0 as in Graham's scan.
- Pull the paper to the right, then higher until it touches a point. This point is a vertex in the convex hull. Continue this way until we come back to p_0.
- Formally: start at p_0. Choose p_1 as the point with the smallest polar angle from p_0. Choose p_2 as the point with the smallest polar angle from $p_1 \ldots$
- $\ldots$ until we reached the highest point p_k.
- We have constructed the right chain.
- Construct the left chain by starting from p_k and measuring polar angles with respect to the negative x-axis.

Jarvis's March

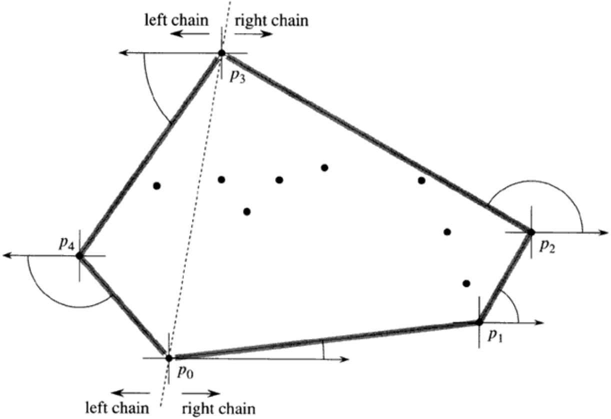


Figure 33.9 The operation of Jarvis's march. The first vertex chosen is the lowest point p_0. The next vertex, p_1, has the smallest polar angle of any point with respect to p_0. Then, p_2 has the smallest polar angle with respect to p_1. The right chain goes as high as the highest point p_3. Then, the left chain is constructed by finding smallest polar angles with respect to the negative x-axis.

Finding closest points

- W.r.t. euclidean distance.
- Brute force: $\theta(n^2)$.
- Divide and conquer algorithm with $O(n \log n)$ complexity.

Finding closest points: Idea

- Each iteration: subset $P \subseteq Q$, arrays X and Y.
- Points in X are sorted in increasing order of their x coordinates.
- Points in Y are sorted in increasing order of their y coordinates.
- To maintain upper bound cannot afford to sort in each iteration.
- $|P| \leq 3$: brute force. Otherwise recursive divide-and-conquer.
- **Divide:** Find a vertical line l that bisects set P into two sets P_L and P_R such that $|P_L| = \lceil |P|/2 \rceil$, $|P_R| = \lfloor |P|/2 \rfloor$, all points of P_L to the left, all points of P_R to the right.
- X_L: subarray that contains point of P_L, X_R: subarray that contains point of P_R.
- Similarly for Y.

Finding closest points (III)

- **Conquer**. Recursive calls: P_L, X_L, Y_L and P_R, X_R, Y_R. Returns smallest distances δ_L and δ_R.
- **Combine.** $\delta = \min\{\delta_L, \delta_R\}$.
- Have to test whether some point in P_L is at distance $< \delta$ from some point in P_R.
- Both such points, if they exist, are within the 2δ-wide strip around l.
- Create an array Y' which is Y with all points not in the 2δ-wide strip around l removed, sorted by y-coordinate.
- **For each point** p **in** Y' try to find points in Y' at distance less than δ.
- Only the 7 points that follow p need to be considered.
- Compute smallest such distance δ'. If $\delta' < \delta$ we found a better pair. Otherwise δ is the smallest distance.
- Correctness, implementation nontrivial.

Finding closest points (IV)

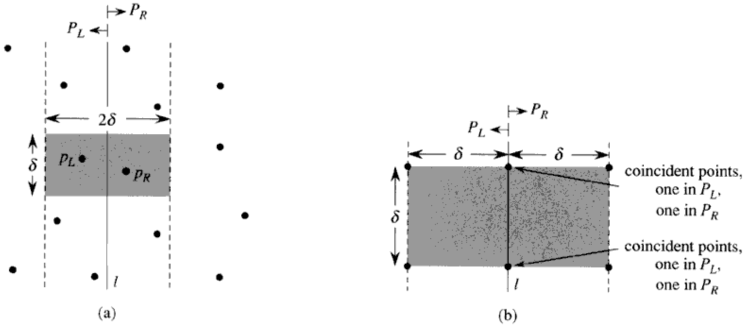


Figure 33.11 Key concepts in the proof that the closest-pair algorithm needs to check only 7 points following each point in the array Y'. **(a)** If $p_L \in P_L$ and $p_R \in P_R$ are less than δ units apart, they must reside within a $\delta \times 2\delta$ rectangle centered at line l. **(b)** How 4 points that are pairwise at least δ units apart can all reside within a $\delta \times \delta$ square. On the left are 4 points in P_L, and on the right are 4 points in P_R. There can be 8 points in the $\delta \times 2\delta$ rectangle if the points shown on line l are actually pairs of coincident points with one point in P_L and one in P_R.

Correctness & complexity

- For each point: Consider the $\delta \times 2\delta$ rectangle centered at line l.
- At most 8 points within this rectangle.
- Assuming δ_L lower than δ_R, it follows that δ_R among the next 7 points following δ_L.
- $O(n \log n)$ bound from recurrence $T(n) = 2T(n/2) + O(n)$.
- Main difficulty: making sure that X_L, X_R, Y_L, Y_R, Y' sorted by appropriate coordinate.
- Key observation: in each call we wish to form a sorted subset of a sorted array.
- Splitting the array into two halves.
- Can be viewed as the inverse of the operation *MERGE* in *MERGESORT*.
- How to get sorted arrays in the first place ? presort. $\theta(n \log n)$.

Splitting: Pseudocode

```
length[Y_L] = length[Y_R] = 0;
for i = 1 to length[Y]
  if (Y[i] ∈ P_L)
  {
   length[Y_L]++;
   Y_L[length[Y_L]] = Y[i];
  }
  else
  {
   length[Y_R] + +;
   Y_R[length[Y_R]] = Y[i];
  }
}
```

And now for something totally different ...

Graph algorithms.

We live in a highly connected world ...

... and that’s important.

A certain disease from Wuhan, China has dramatic effects all over the planet ...

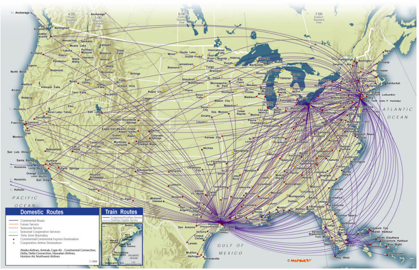
A software bug in the alarm system at the control room of FirstEnergy, in Akron, Ohio knocks out the power grid in the whole Northeast United States (2003).

To understand, for my (and your) generation

How do real networks look like ?

How do network properties impact **the processes** that take place on them ?

Some real networks



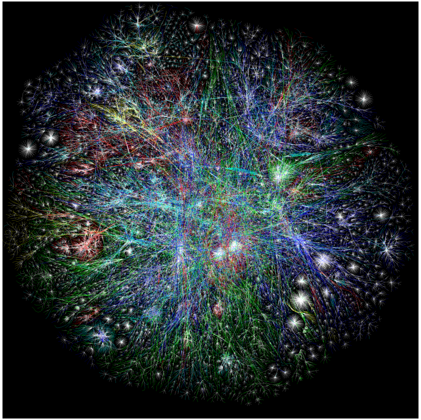
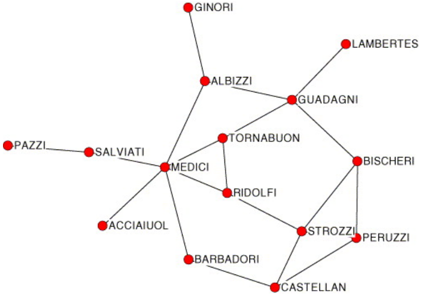
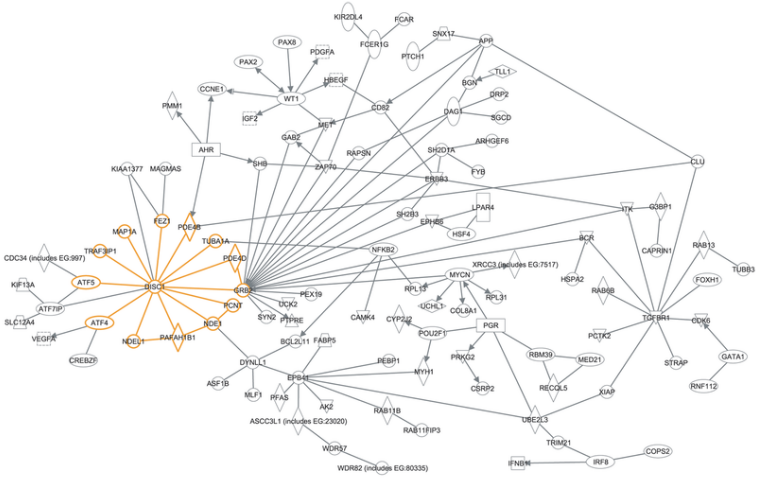


Figure: (a). Air traffic map of the U.S. (b). Physical Internet

Marriage Networks of important families in Medieval Florence.

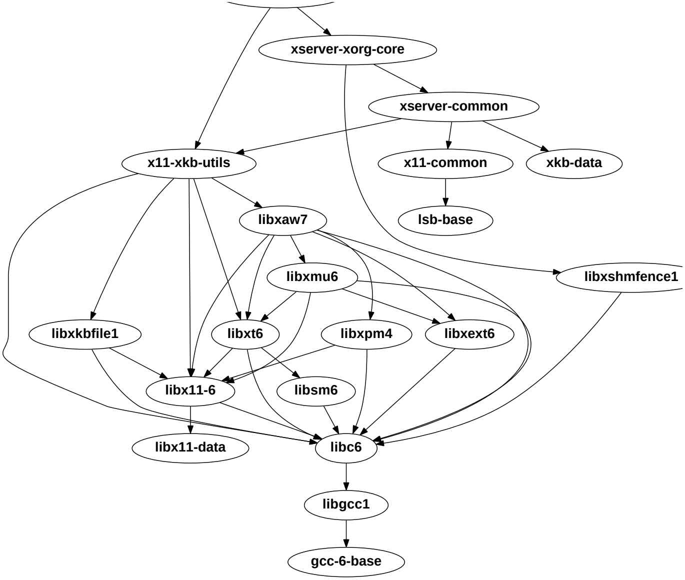




© 2000-2009 Ingenuity Systems, Inc. All rights reserved.

Part of the DISC1 interactome, with genes represented by text in boxes and interactions noted by lines between the genes. From Hennah and Porteous (2009).

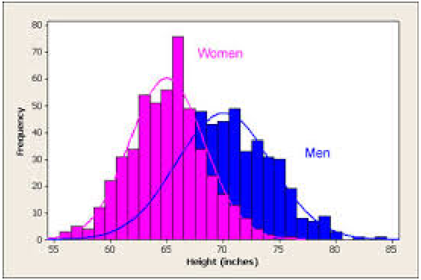
... or even ...

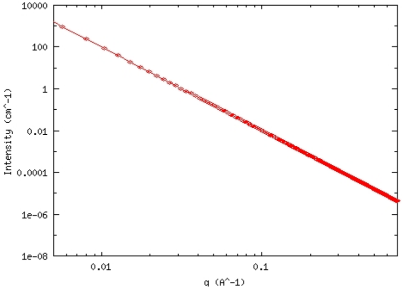


xserver-xorg-core
xserver-common
x11-xkb-utils
x11-common
xkb-data
libxaw7
lsb-base
libxmu6
libxshmfence1
libxkbfile1
libxt6
libxpm4
libxext6
libx11-6
libsm6
libx11-data
libc6
libgcc1
gcc-6-base

What's so interesting about networks ?

Small worlds: everyone is "not very far from everyone".





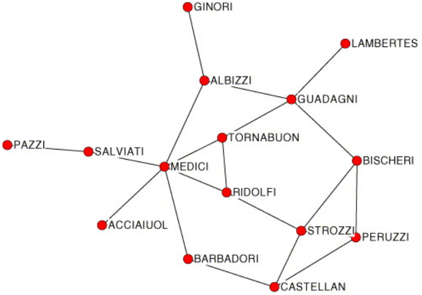
(a). Distribution of heights in the U.S. population.

(b). Degree distribution (aproximately) power law. Few "tall" people, "many" well connected people

What's so interesting about networks ? (II)

Centrality

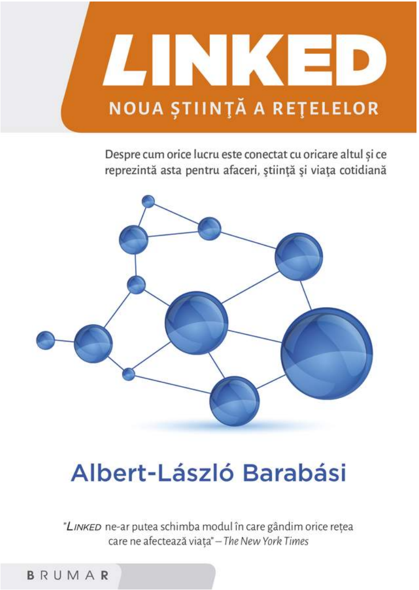
Some nodes are "more important/central than others".



How do we measure centrality ?

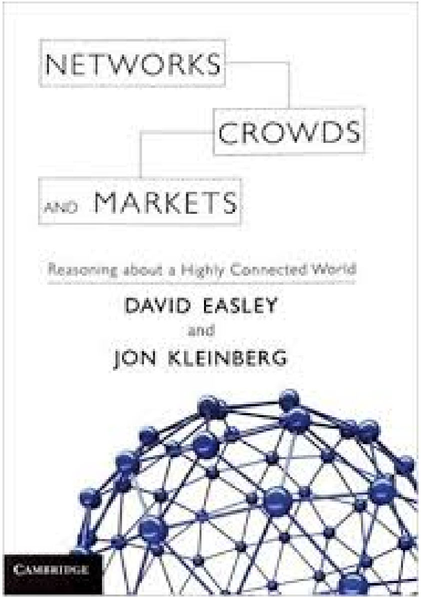
- degree centrality: $c(y) = \frac{1}{n-1} deg(y)$.
- betweenness centrality: $c(y) = \sum_{x \neq z} \frac{\sigma_{x,z}[y]}{\sigma_{x,z}}$
- eigenvector centrality: $c(y) = w(y)$, w the eigenvector of the adjacency matrix A of G corresponding to the largest eigenvalue.

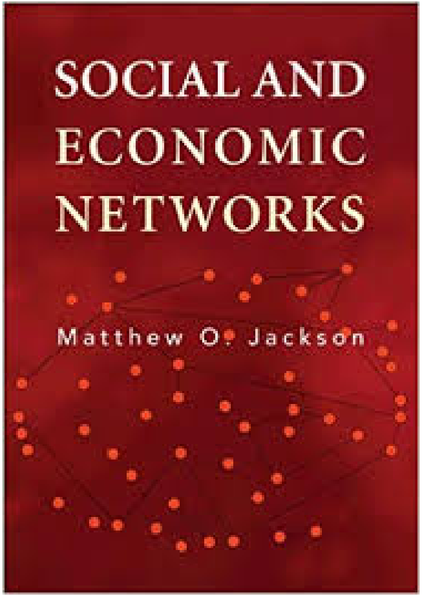
Want to read something interesting ?



By the way, not only in America ...

Want to read something even more interesting ?





Can you study large networks without good algorithms ?

To conclude: 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*

- A ***graph***

$$G = (V, E)$$

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

- 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

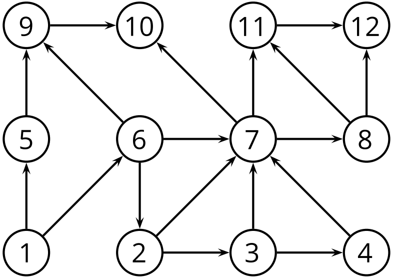
- How do we represent a graph $G = (E, V)$ in a computer?

- 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*

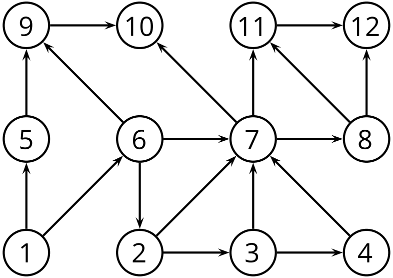
Graph Representation

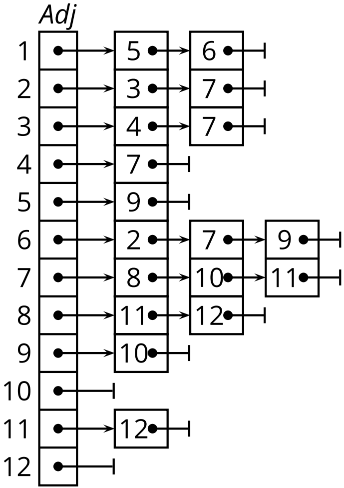
- How do we represent a graph $G = (E, V)$ in a computer?
- *Adjacency-list representation*
- $V = \{1, 2, \dots |V|\}$
- G consists of an array Adj
- A vertex $u \in V$ is represented by an element in the array Adj
- $Adj[u]$ is the ***adjacency list*** of vertex u
  - the list of the vertices that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

Example

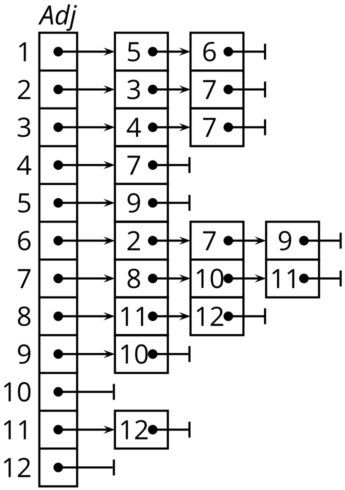


Example

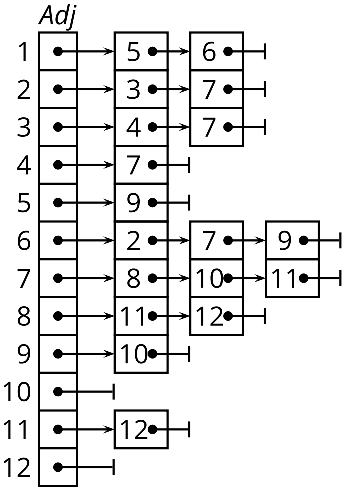

9
10
11
12
5
6
7
8
1
2
3
4


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

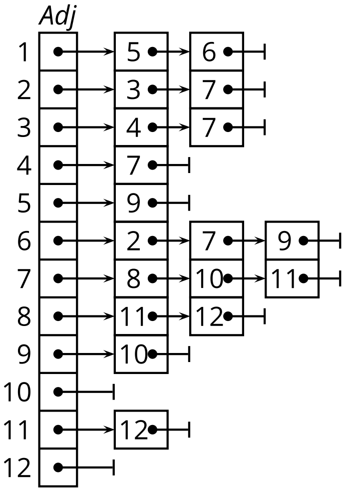


- Accessing a vertex u? $\checkmark$.



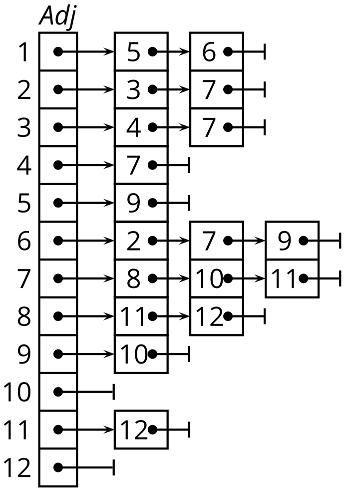
Using the Adjacency List

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



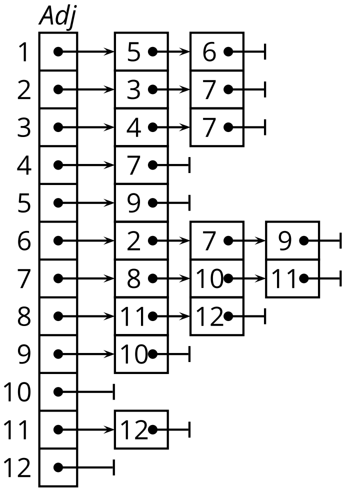
Using the Adjacency List

- Accessing a vertex u? $O(1)$ ✓.
  - optimal ✓
- Iteration through V?

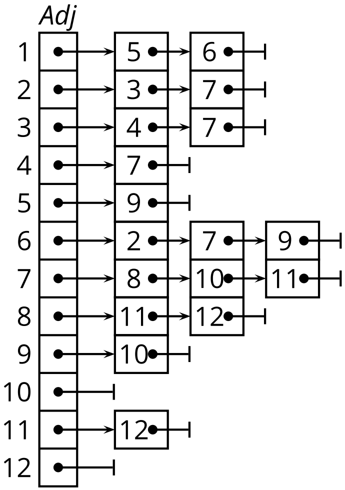


Using the Adjacency List

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

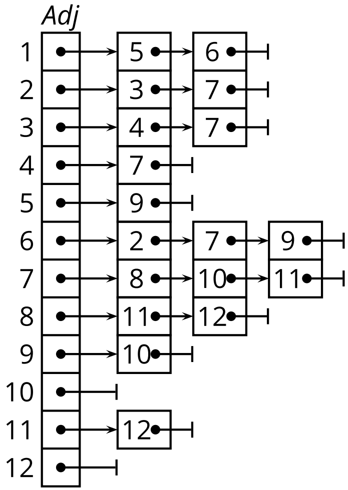


- Accessing a vertex u? $O(1)$ ✓.
  - optimal ✓
- Iteration through V? $\Theta(|V|)$
  - optimal ✓
- Iteration through E?



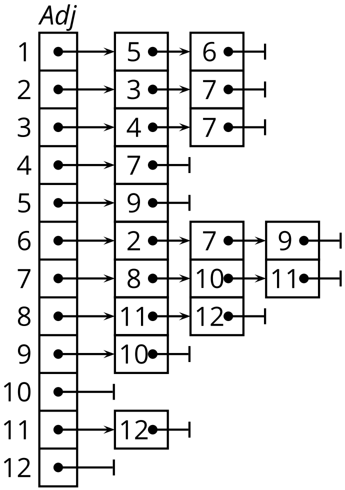
Using the Adjacency List

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



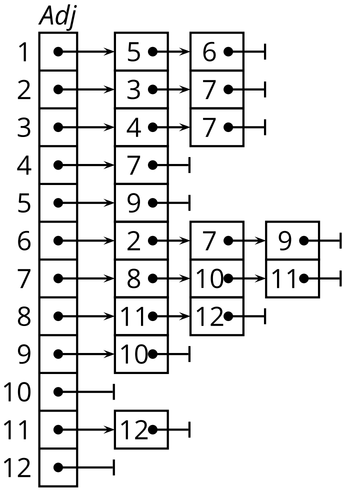
Using the Adjacency List

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



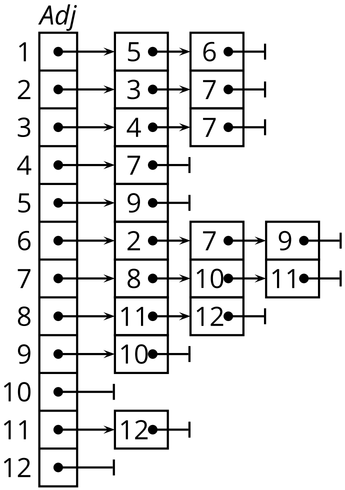
Using the Adjacency List

- 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 ✗



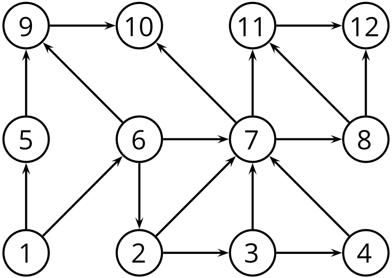
Graph Representation (2)

Graph Representation (2)

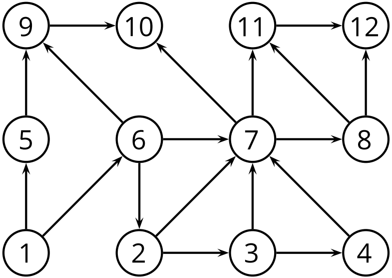
- *Adjacency-matrix representation*
- $V = \{1, 2, \ldots |V|\}$
- G consists of a $|V| \times |V|$ matrix A
- $A = (a_{ij})$ such that

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

Example

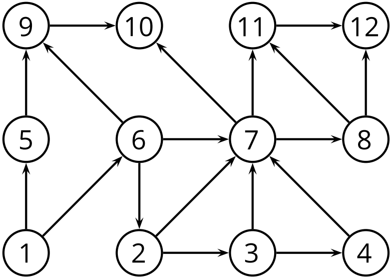


Example



	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												

- 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

Space Complexity

- Adjacency-list representation

- Adjacency-list representation

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

- Adjacency-list representation

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

optimal .

- Adjacency-list representation

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

  optimal .

- Adjacency-matrix representation

- Adjacency-list representation

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

  optimal .

- Adjacency-matrix representation

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

- Adjacency-list representation

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

optimal .

- Adjacency-matrix representation

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

possibly very bad $\times$.

- Adjacency-list representation

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

  optimal .

- Adjacency-matrix representation

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

  possibly very bad $\times$.

- 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

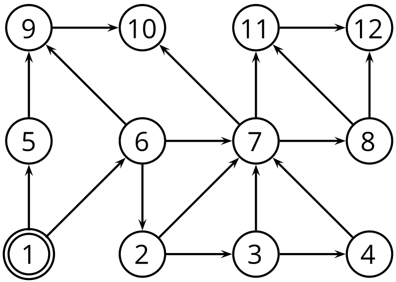
- Adjacency-list representation
  - generally good, especially for its optimal space complexity
  - bad for ***dense*** graphs and algorithms that require random access to edges
  - preferable for ***sparse*** graphs or graphs with ***low degree***
- Adjacency-matrix representation
  - suffers from a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

- One of the simplest but fundamental algorithms

- One of the simplest but fundamental algorithms
- *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

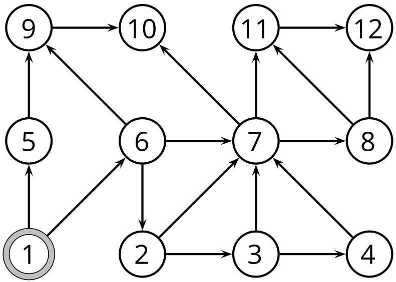
BFS Algorithm

```
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

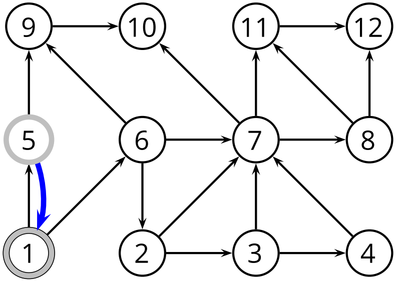
```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
          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



$u = 1$

$Q = \varnothing$

```
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
```

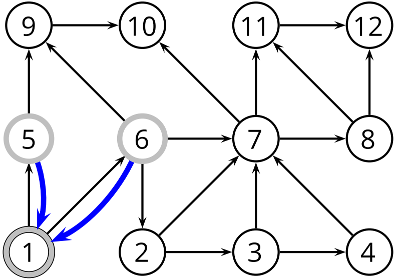


$u = 1$

$Q = \{5\}$

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
```

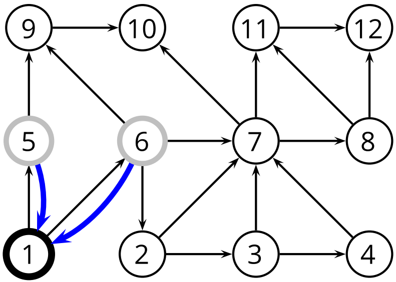


$u = 1$

$Q = \{5, 6\}$

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
```

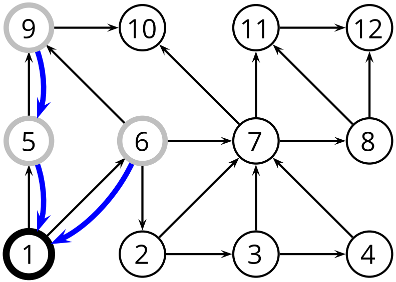


$u = 5$

$Q = \{6\}$

BFS 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
```

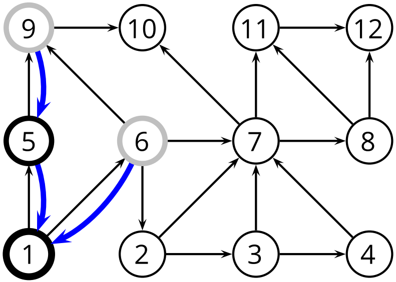


$u = 5$

$Q = \{6, 9\}$

BFS Algorithm

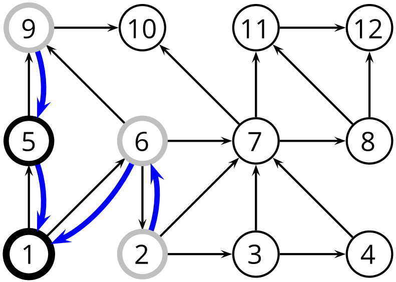
```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
          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



$u = 6$

$Q = \{9\}$

```
BFS(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
```

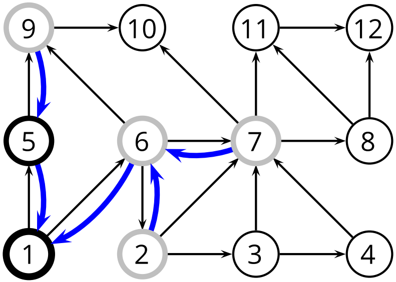


$u = 6$

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

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
```

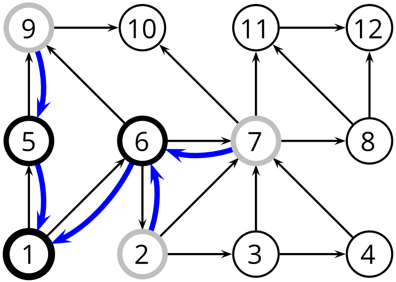


$u = 6$

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

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
```

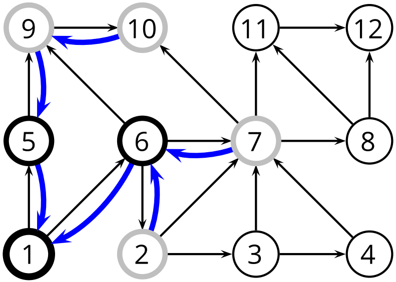


$u = 9$

$Q = \{2, 7\}$

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
```

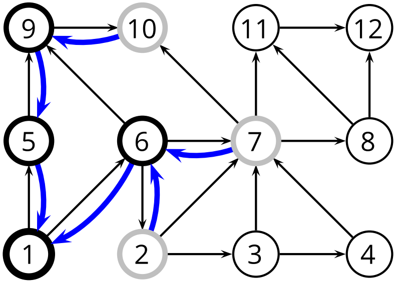


$u = 9$

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

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
```

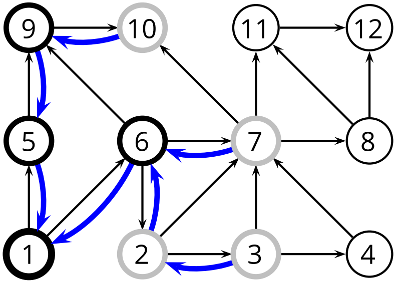


$u = 2$

$Q = \{7, 10\}$

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
```

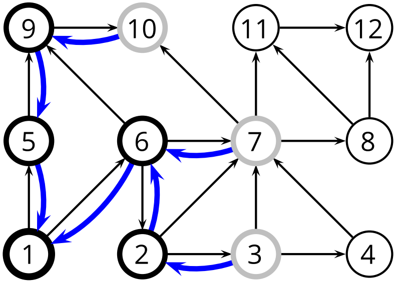


$u = 2$

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

BFS 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
```

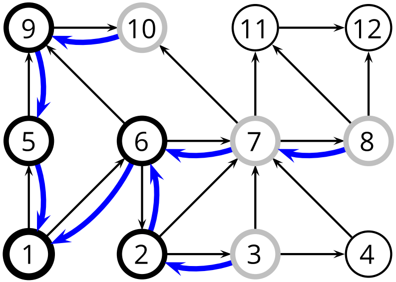


$u = 7$

$Q = \{10, 3\}$

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
```

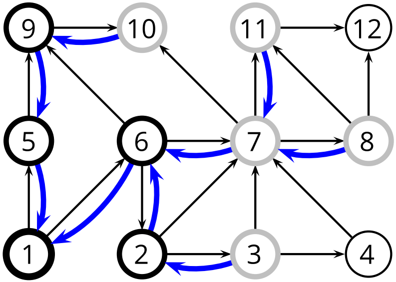


$u = 7$

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

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
```

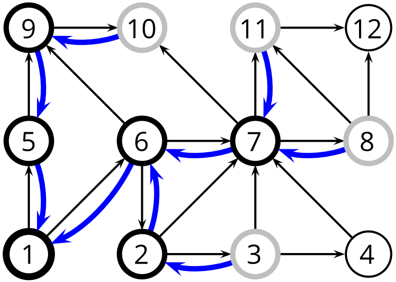


$u = 7$

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

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
```

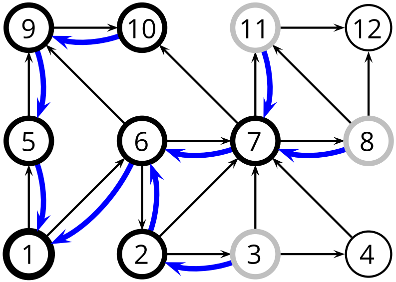


$u = 10$

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

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
```

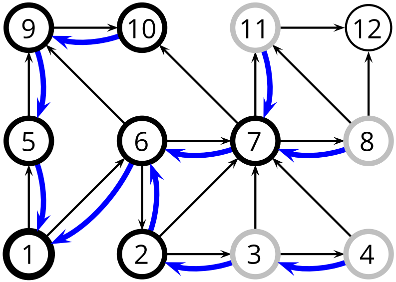


$u = 3$

$Q = \{8, 11\}$

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
```

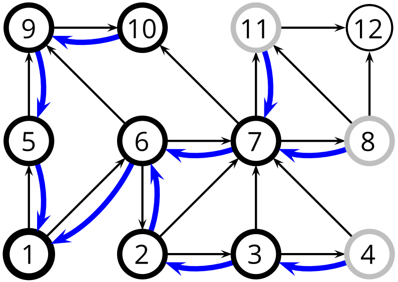


$u = 3$

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

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
```

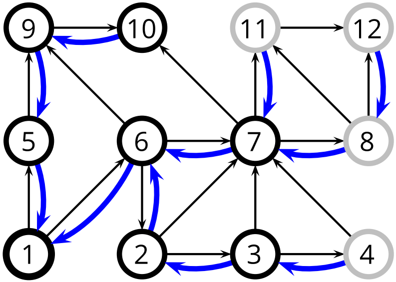


$u = 8$

$Q = \{11, 4\}$

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
```

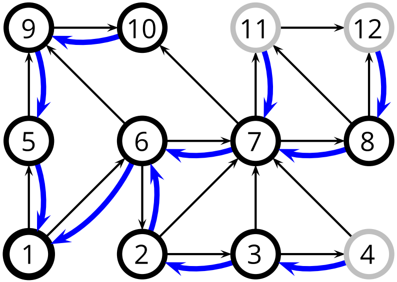


$u = 8$

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

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
```

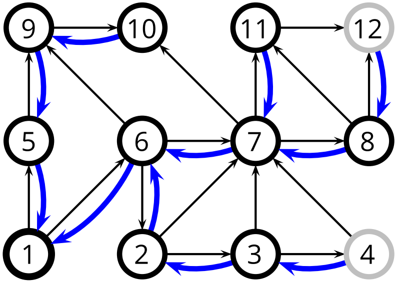


$u = 11$

$Q = \{4, 12\}$

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
```

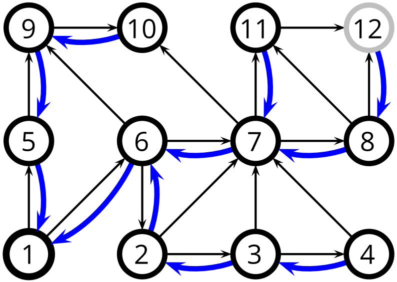


$u = 4$

$Q = \{12\}$

BFS 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
```

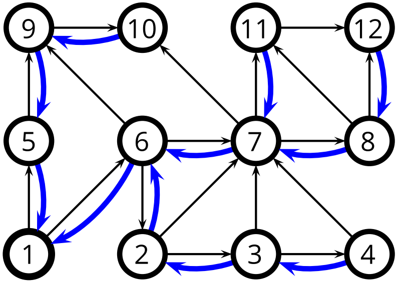


$u = 12$

$Q = \emptyset$

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

- 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

- 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] = $ WHITE, but then we immediately set $color[u] = $ GREY

- 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 $color[u] = \text{GREY}$
- So, the overall complexity is $\Theta(|V| + |E|)$

Applications of DFS: Topological Sort

- **Problem:** (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

- **Problem:** (topological sort)

  Given a *directed acyclic graph* (DAG)

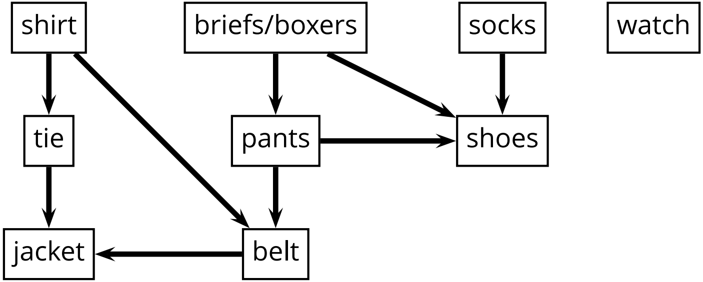
  - find an ordering of vertices such that you only end up with *forward links*

- **Example:** dependencies in software packages

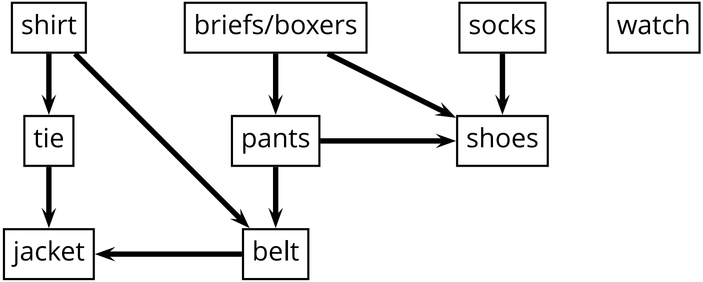
  - find an installation order for a set of software packages
  - such that every package is installed only after all the packages it depends on

Topological Sort Algorithm

Topological Sort Algorithm



Topological Sort Algorithm



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