CS217 - Data Structures & Algorithm Analysis (DSAA)

Lecture #13

▶Elementary Graph Algorithms

Prof. Pietro S. Oliveto
Department of Computer Science and Engineering
Southern University of Science and Technology (SUSTech)

olivetop@sustech.edu.cn
https://faculty.sustech.edu.cn/olivetop

Reading: Chapter 20

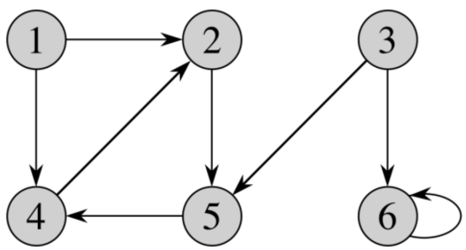
Aims for this lecture

- To discuss **breadth-first search (BFS)** and breadth-first trees.
- To discuss **depth-first search (DFS)** and depth-first trees.
- To analyse the **runtime** of BFS and DFS.
- To show how DFS can **classify edges** for additional information about the graph.
- To show how to use DFS to
  - Check whether a graph contains cycles
  - Put tasks in the right order (topological sorting)
  - Compute strongly connected components in graphs
- To show the **correctness** of some remarkable algorithms.

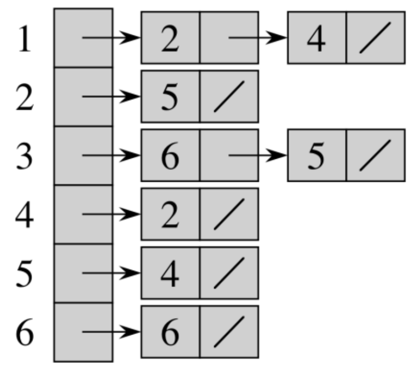
Representations of graphs

- Using terminology for graphs G = (V, E) from Appendix B
- **Adjacency-list representation**:
  - Array Adj of |V| lists, one for each vertex.
  - The list Adj[u] contains all vertices v adjacent to u in G, i.e. there is an edge $(u, v) \in E$.
  - The sum of all adjacency list lengths equals |E|.
- **Adjacency-matrix representation**:
  - Assume that vertices are numbered $1, 2, \ldots, n$.
  - Adjacency matrix is a $|V| \times |V|$ matrix with entries $a_{ij} = 1$ if $(i, j) \in E$ and $a_{ij} = 0$ otherwise.

▶Example for a directed graph



(a)

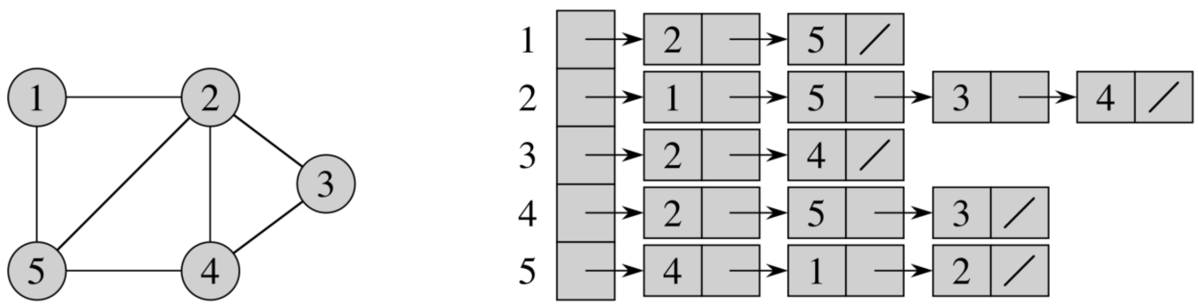


(b)

$$
\begin{array}{c|cccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
1 & 0 & 1 & 0 & 1 & 0 & 0 \\
2 & 0 & 0 & 0 & 0 & 1 & 0 \\
3 & 0 & 0 & 0 & 0 & 1 & 1 \\
4 & 0 & 1 & 0 & 0 & 0 & 0 \\
5 & 0 & 0 & 0 & 1 & 0 & 0 \\
6 & 0 & 0 & 0 & 0 & 0 & 1 \\
\hline
\end{array}
$$

(c)

Example for an undirected graph



	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

(a) (b) (c)

无向边

- For every undirected edge $\{u, v\}$, *v* is in *u*'s adjacency list and *u* is in *v*'s adjacency list.
- Note the symmetry in the adjacency matrix along the main diagonal. It's sufficient to store the entries on and above the diagonal.

Adjacency lists vs. adjacency matrix

- Input sizes are:
  - $\Theta(|V| + |E|)$ for adjacency lists as

$$\sum_{u \in V} |\mathrm{Adj}(u)| = \begin{cases} |E| & \text{for directed graphs} \\ 2|E| & \text{for undirected graphs} \end{cases}$$

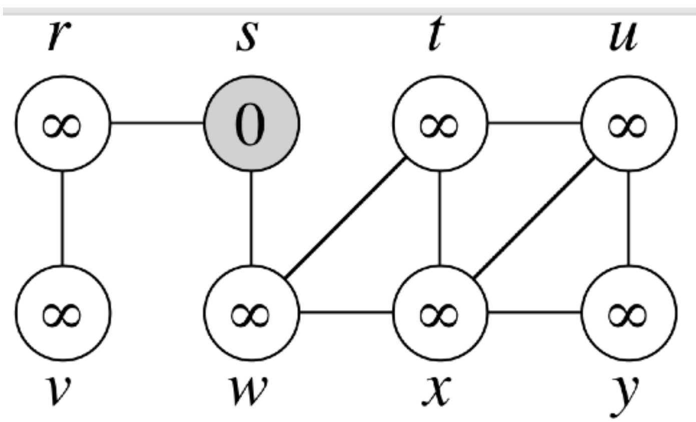
  - $\Theta(|V|^2)$ for adjacency matrices
- Adjacency lists are more compact and preferable for **sparse** graphs. A graph is **sparse** if $|E| = o(|V|^2)$ and **dense** if $|E| = \Theta(|V|^2)$.
- Testing whether *u* and *v* are adjacent takes time O(1) in an adjacency matrix and can take time $\Omega(|V|)$ with adjacency lists.

Breadth-first search (BFS)

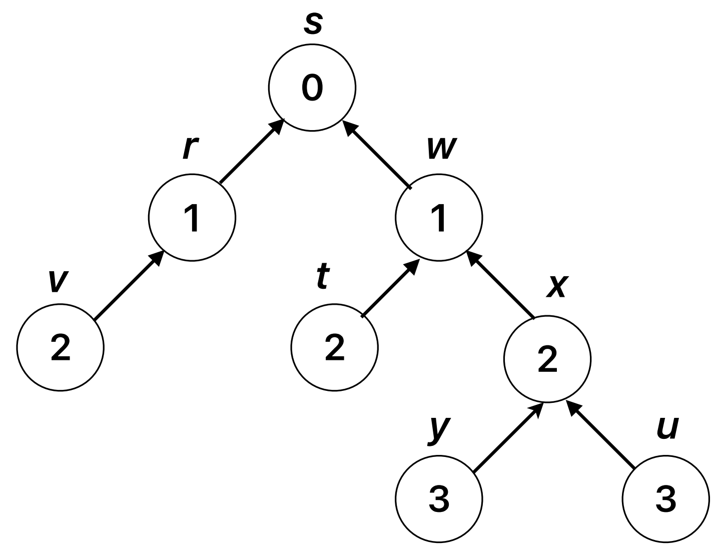
- One of the simplest algorithms for searching graphs.
- Given a graph G = (V, E) and a distinguished **source s**, BFS computes the distance from s to each reachable vertex.
- It also produces a **breadth-first tree** with root s that contains all reachable vertices: the simple path in the breadth-first tree from s to v corresponds to a **shortest path** from s to v (shortest = smallest number of edges).
- We'll see algorithms for other problems (minimum spanning trees and shortest paths) that use similar ideas.

Breadth-first search: Result

Input graph



Output attributes (BFS tree)

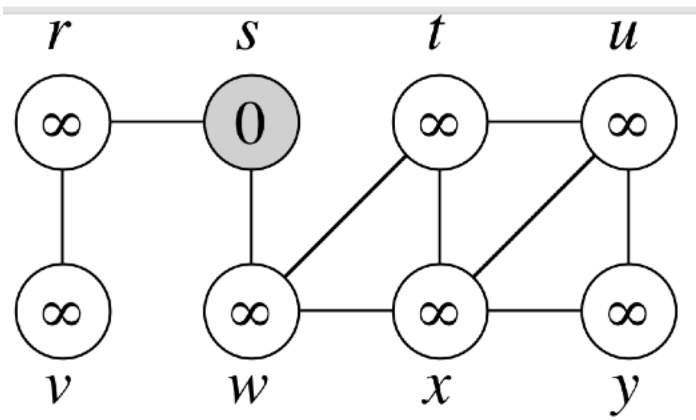


Breadth-first search: Ideas

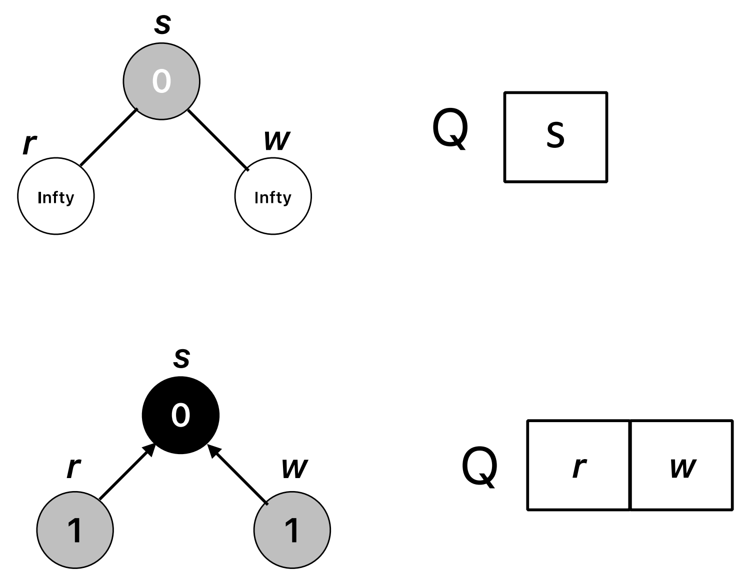
- Start from the source and then explore the **frontier** between discovered and undiscovered vertices. BFS explores the whole breadth of this frontier.
- A **queue** is used to store the next vertices to be processed: BFS extracts the vertex at the front of the queue and adds its neighbours to the end of the queue.
- We assign colours to vertices to indicate their status:
  - **White**: vertex has not been discovered yet
  - **Gray**: vertex has been discovered, but needs to be processed.
  - **Black**: vertex has been discovered and processed.
- Vertices have **attributes**: .color, .d (distance) and $.\pi$ (predecessor/ parent in BF tree). Following $.\pi$ pointers gives shortest path to s.

Breadth-first search: Idea (2)

Input graph



First Step
(BFS tree)



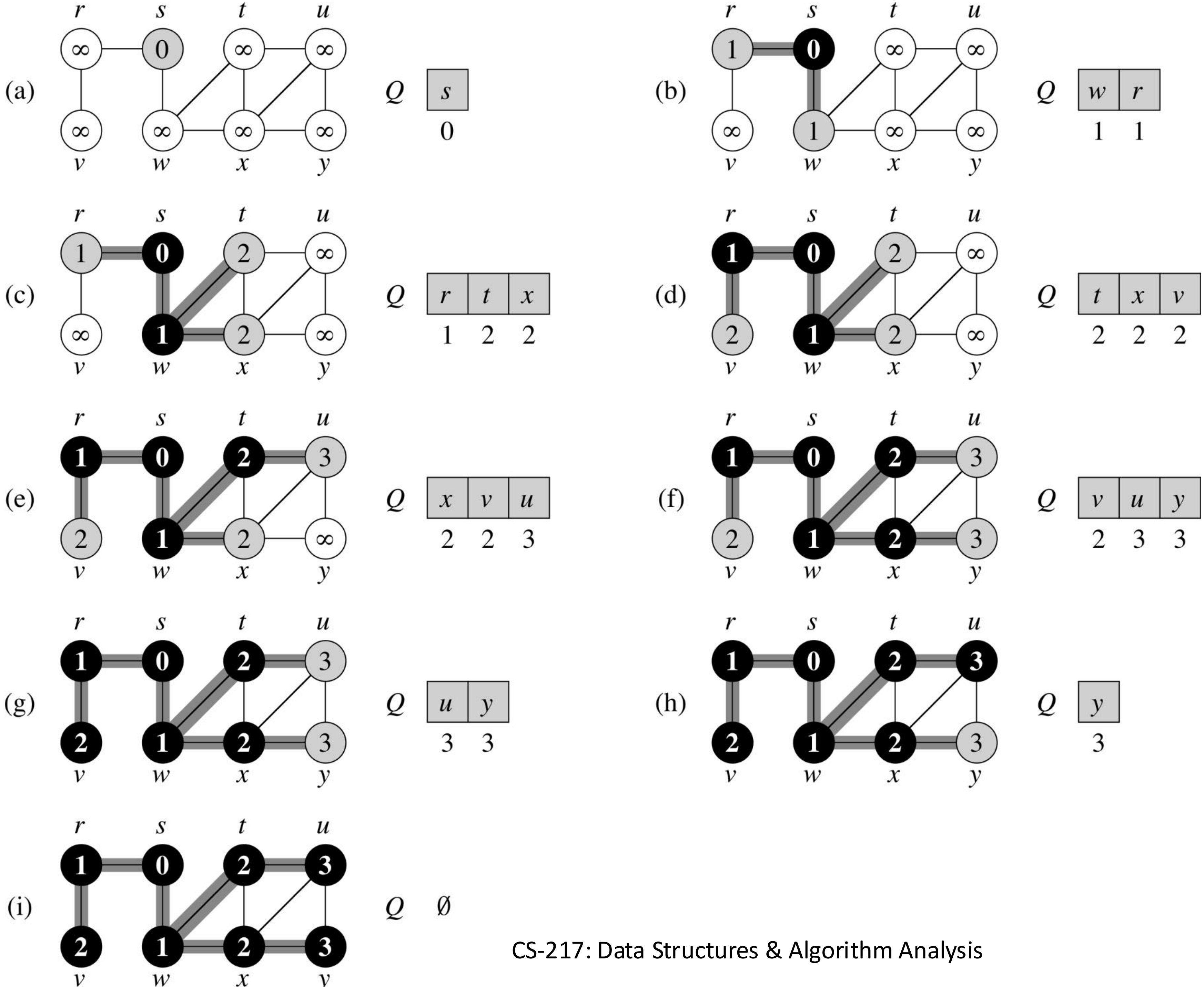
- We dequeue current gray node (*s*)
- Enqueue the adjacent nodes to s (r,w): set their distance to current distance +1, set their predecessor to current node (s), make them gray (current frontier)
- Set current node colour to black (s.color = BLACK)
- Repeat -> Dequeue

BFS

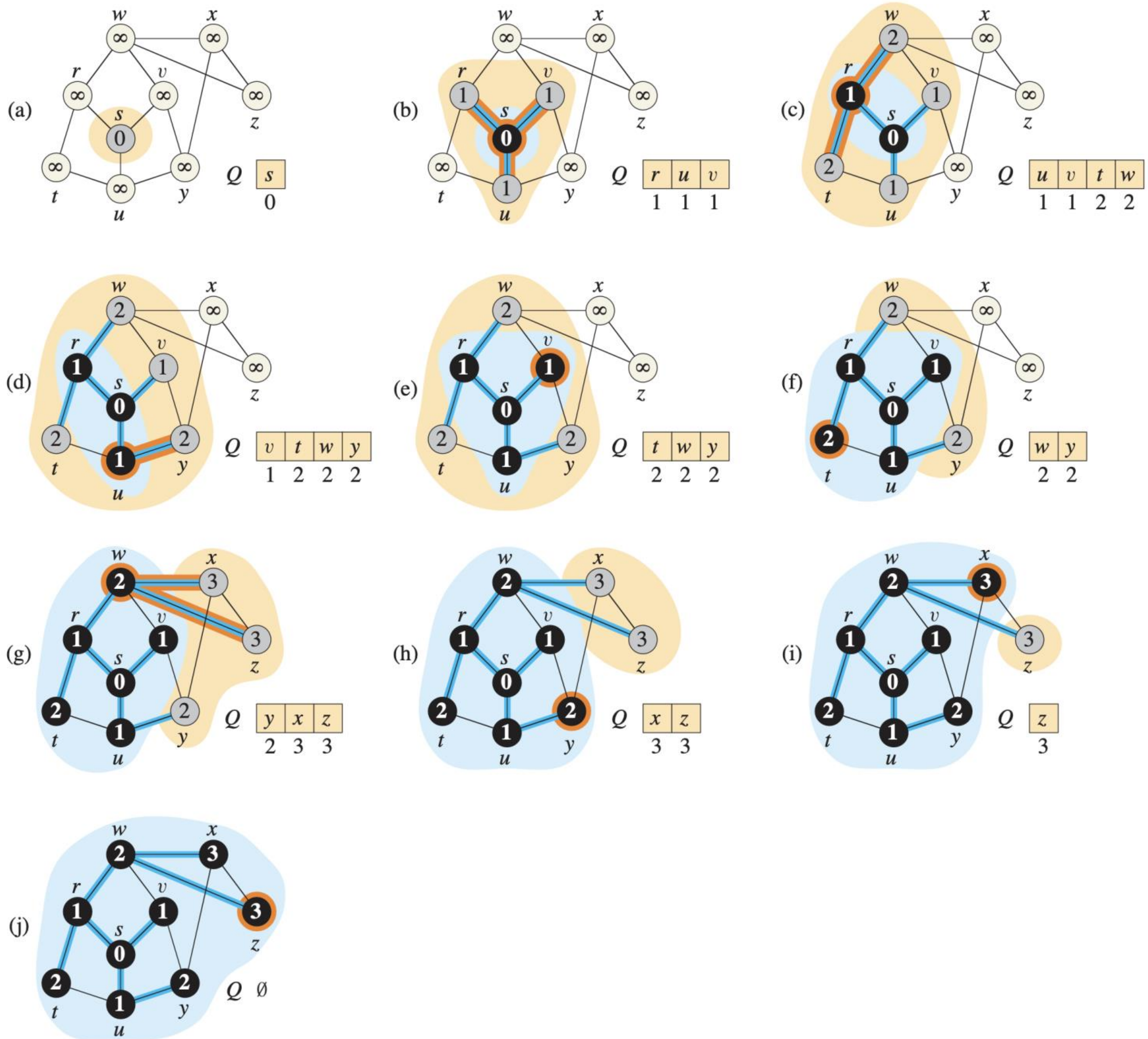
- Adj list representation is assumed
- Lines 1-8: Initially all vertices but *s* are white.
- Enqueue *s*
- While loop: extract front vertex *u* and add all its unseen (white) adjacent vertices *v* to the end of the queue.
- *v*'s distance is one larger than *u*'s, *u* becomes *v*'s predecessor.
- Enqueued vertices become gray, dequeued ones are turned black.

BFS(G, s)

```
1:  for each vertex u ∈ V \ {s} do
2:      u.colour = WHITE
3:      u.d = ∞
4:      u.π = NIL
5:  s.colour = GRAY
6:  s.d = 0
7:  s.π = NIL
8:  Q = ∅
9:  ENQUEUE(Q, s)
10: while Q ≠ ∅ do
11:     u = DEQUEUE(Q)
12:     for each v ∈ Adj[u] do
13:         if v.colour = WHITE then
14:             v.colour = GRAY
15:             v.d = u.d + 1
16:             v.π = u
17:             ENQUEUE(Q, v)
18:     u.colour = BLACK
```



(a)
r s t u
∞ 0 ∞ ∞
∞ ∞ ∞ ∞
v w x y
Q s
0
(b)
r s t u
1 0 ∞ ∞
∞ 1 ∞ ∞
v w x y
Q w r
1 1
(c)
r s t u
1 0 2 ∞
∞ 1 2 ∞
v w x y
Q r t x
1 2 2
(d)
r s t u
1 0 2 ∞
2 1 2 ∞
v w x y
Q t x v
2 2 2
(e)
r s t u
1 0 2 3
2 1 2 ∞
v w x y
Q x v u
2 2 3
(f)
r s t u
1 0 2 3
2 1 2 3
v w x y
Q v u y
2 3 3
(g)
r s t u
1 0 2 3
2 1 2 3
v w x y
Q u y
3 3
(h)
r s t u
1 0 2 3
2 1 2 3
v w x y
Q y
3
(i)
r s t u
1 0 2 3
2 1 2 3
v w x y
Q ∅



(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
(i)
(j)
Q
s
0
r u v
1 1 1
u v t w
1 1 2 2
v t w y
1 2 2 2
t w y
2 2 2
w y
2 2
y x z
2 3 3
x z
3 3
z
3
Q ∅

BFS: Runtime (for scanning whole graph)

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

$\boldsymbol{O(V)}$

?

BFS: Runtime (for scanning whole graph)

- No vertex becomes white.
- Test for whiteness is positive only once, as vertices are made gray immediately.
- Hence each vertex is **enqueued** and **dequeued** at most once. Time O(V) for queue operations.
- Adjacency list of each vertex is scanned at most once, hence total time for scanning all adjacency lists is O(V+E).

BFS(G, s)

```
...
while Q ≠ ∅ do
    u = DEQUEUE(Q)
    for each v ∈ Adj[u] do
        if v.colour = WHITE then
            v.colour = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.colour = BLACK
```

- Overhead before while loop is O(V), hence total time is **O(V + E)**, **linear in the input size.**

BFS: Correctness (1)

- **Lemma 20.2** (Helper Lemma)
  Let $G(V, E)$ be a graph, and BFS is run on source $s \in V$. Let $\delta(s, v)$ be the shortest path from s to v for all $v \in V$. Then for each vertex $v \in V$, the value $v.d$ computed by BFS satisfies $\boldsymbol{v.d \geq \delta(s, v)}$ **at all times** including at termination.

- **Proof Idea** (formal proof by induction in the book)

- For all vertices $v.d = \infty$ until it becomes gray (if ever)
  $[v.d \geq \delta(s, v)]$ ✓

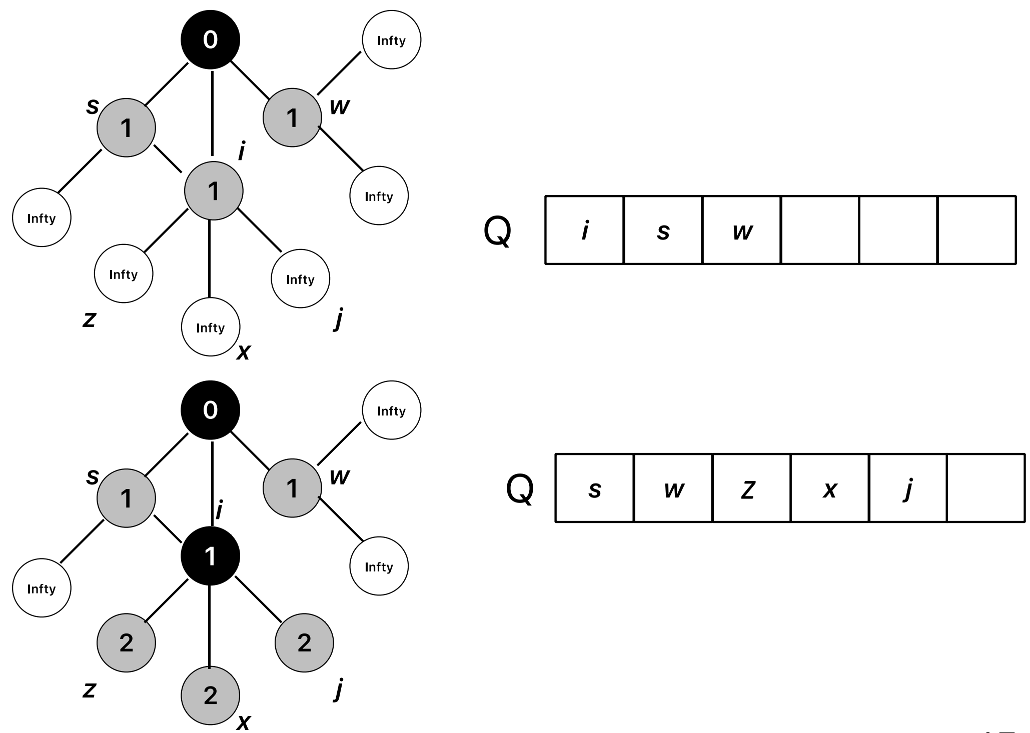
- When it becomes gray it will equal the length of some path from s to v (or it would have not been reached): at each step on the path we increase the distance counter by 1. Each vertex is assigned a distance only once, so it will never change.
  $[v.d \geq \delta(s, v)]$ ✓

- If it never becomes gray, then it stays $v.d = \infty$ $[v.d \geq \delta(s, v)]$ ✓

BFS: Correctness (2)

- **Corollary 20.4 (of Lemma 20.3)** (Helper Lemma)
  Suppose that vertices v_i and v_j are enqueued during the execution of BFS, and that v_i is enqueued before v_j. Then $\boldsymbol{v_i.d \leq v_j.d}$ at the time that v_j is enqueued.

- **Proof Idea** (formal proof by induction in the book)



BFS: Correctness (3)

- **Theorem 20.5**
  Let $G(V,E)$ be a graph, and BFS is run on source $s \in V$. Then BFS discovers every vertex $v \in V$ that is reachable from s, and upon termination $\boldsymbol{v.d = \delta(s,v)}$ **for all** $\boldsymbol{v \in V}$.
- **Proof (By contradiction)**
- Assume that $\exists\ v \mid v.d\ \neq\ \delta(s.v)$
- Let v be the one that has **minimum** $\delta(s,v)$
- Then:
  - $v.d >\ \delta(s,v)$ (By Lemma 20.2 $v.d\ \geq\ \delta(s,v)$)
  - $v\ \neq s$ ($s.d = 0$ & $\delta(s,s) = 0$)
  - v is reachable from s (otherwise $\delta(s,v) = \infty$)
  - => **There exists a path** of length at least 1 from s to v

BFS: Correctness (4)

- **Theorem 20.5**
  Let $G(V, E)$ be a graph, and BFS is run on source $s \in V$. Then BFS discovers every vertex $v \in V$ that is reachable from s, and upon termination $v.d = \delta(s, v)$ for all $v \in V$.

- **Proof (By contradiction) (2)**

- Let u be the vertex preceding v on some **shortest path** from s to v (u exists because $v \neq s$)

- Then
  - $\delta(s, v) = \delta(s, u) + 1$
  - $u.d = \delta(s, u)$ (because $\delta(s, u) < \delta(s, v)$ & v has minimum $\delta(s, v)$ amongst nodes where $v.d \neq \delta(s.v)$)

- Thus, $\boldsymbol{v.d > \delta(s.v) = \delta(s, u) + 1 = u.d + 1}$

- Now we can show the contradiction!

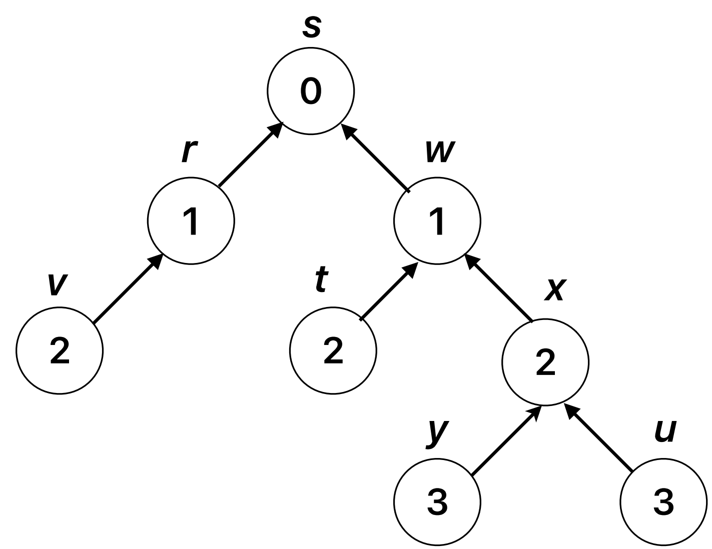
BFS: Correctness (5)

- **Theorem 20.5**
  Let $G(V, E)$ be a graph, and BFS is run on source $s \in V$. Then BFS discovers every vertex $v \in V$ that is reachable from s, and upon termination $v.d = \delta(s, v)$ for all $v \in V$.
- **Proof (By contradiction) (3)**
- $\boldsymbol{v.d > \delta(s.v) = \delta(s, u) + 1 = u.d + 1}$
- Consider when vertex u is dequeued. Then v is either
  - White: then $v.d = u.d + 1$ ✗
  - Black: then $v.d \leq u.d$ (Cor. 20.4) ✗
  - Gray: then it was painted gray by some w such that:
    - $w.d \ \leq u.d$ (Cor 20.4) and $v.d = w.d + 1$. So,
    - $v.d = w.d + 1 \ \leq u.d + 1$ ✗
- Thus we conclude that $v.d = \ \delta(s, v)$ for all $v \in V$.

BFS: Printing shortest path

- The following algorithm prints the shortest path between the source and any reachable node $v \in V$

```
PRINT-PATH(G, s, v)
if v == s
        print s
elseif v.π == NIL
        print "no path from" s "to" v "exists"
else PRINT-PATH(G, s, v.π)
        print v
```



- **Runtime?**

Summary for Breadth-First Search

- Breadth-first search searches the breadth of the frontier between discovered and undiscovered vertices.
- It creates a **breadth-first tree** that encodes shortest paths for all vertices. Following predecessors/parents in the tree reconstructs a shortest path from a vertex v to s.
- The running time of BFS is **O(V + E)**, **linear in the input size.**

Depth-first search (DFS)

- Works for undirected and directed graphs.
- Ideas:
  - Go into depth by exploring edges out of the most recently discovered vertex and backtrack when stuck.
  - Continue until all vertices reachable from the start vertex are discovered.
  - If any undiscovered vertices remain, continue with one of them as new source.
- As for BFS, define predecessors $v.\pi$ that represent several **depth-first trees**.
- These trees form a **depth-first forest.**

DFS: Colours and timestamps

- DFS uses colours white, gray, black as for BFS:
  - **White**: vertex has not been discovered yet
  - **Gray**: vertex has been discovered, but is not finished yet.
  - **Black**: vertex has been finished (finished scan of adjacency list).
- Also uses **timestamps**:
  - **v.d** is the time v is first **discovered** (and grayed)
  - **v.f** is the time v is **finished** (and blackened)
  - Global variable time is incremented with each event
  - Hence for all vertices v.d < v.f

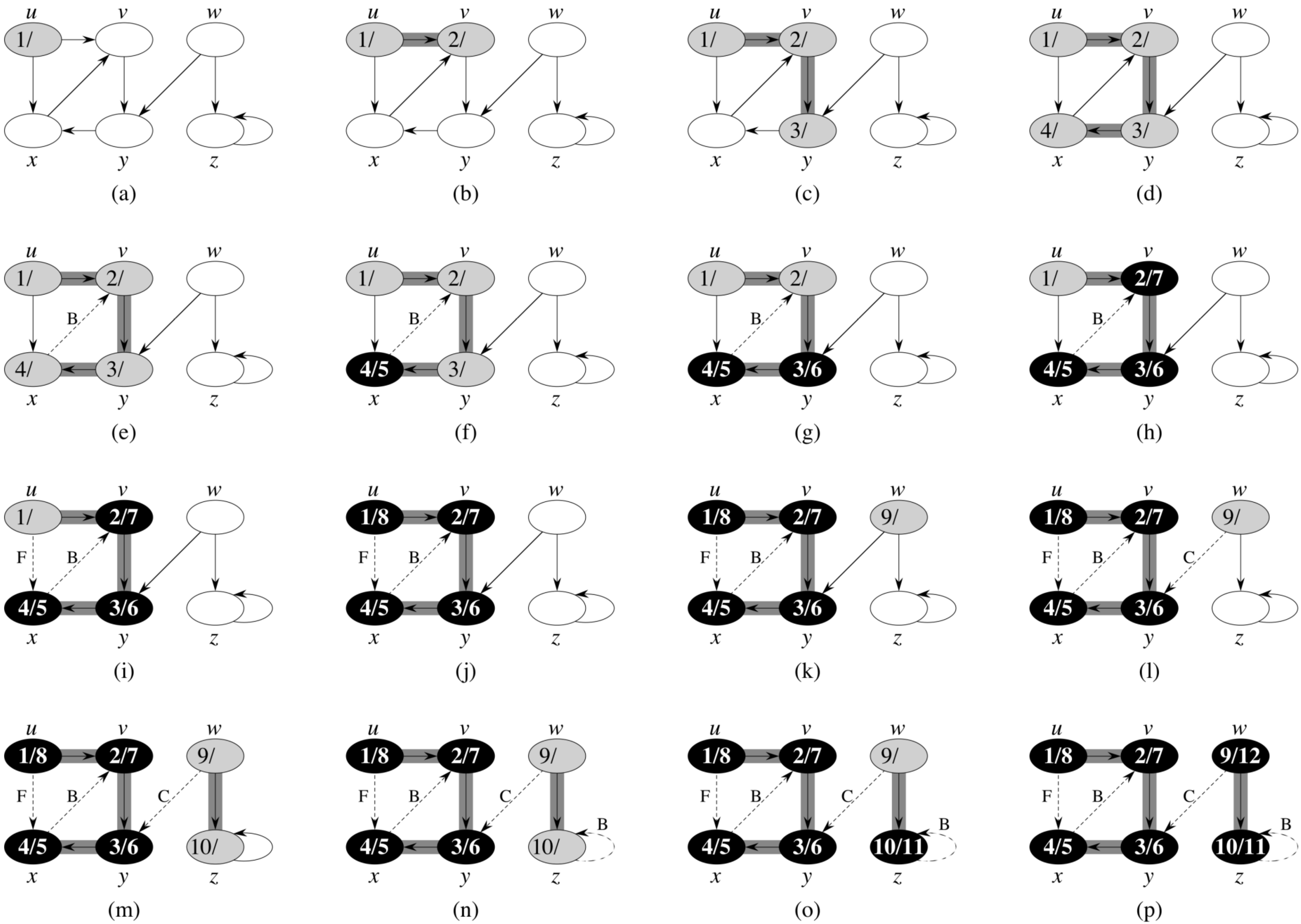
DFS: Pseudocode and runtime

DFS(G)

```
for each vertex u ∈ V do
    u.colour = white
    u.π = NIL
time = 0
for each vertex u ∈ V do
    if u.colour == white then
        DFS-VISIT(G, u)
```

DFS-VISIT(G, u)

```
time = time+1
u.d = time
u.colour = gray
for each v ∈ Adj[u] do
    if v.colour == white then
        v.π = u
        DFS-VISIT(G, v)
u.colour = black
time = time+1
u.f = time
```



u v w
x y z
1/ 2/ 3/ 4/ 4/5 3/6 2/7 1/8 9/ 10/ 10/11 9/12
B F C
(a) (b) (c) (d)
(e) (f) (g) (h)
(i) (j) (k) (l)
(m) (n) (o) (p)

DFS: Pseudocode and runtime

DFS(G)

```
for each vertex u ∈ V do
    u.colour = white
    u.π = NIL
time = 0
for each vertex u ∈ V do
    if u.colour == white then
        DFS-Visit(G, u)
```

DFS-Visit(G, u)

```
time = time+1
u.d = time
u.colour = gray
for each v ∈ Adj[u] do
    if v.colour == white then
        v.π = u
        DFS-Visit(G, v)
u.colour = black
time = time+1
u.f = time
```

Runtime?

- Runtime is $\Theta(|V| + |E|)$:
  - DFS runs in time $\Theta(|V|)$ exclusive of the time for DFS-Visit.
  - DFS-Visit is only called once for each vertex v as *v* must be white and is grayed immediately. The loop executes |Adj[u]| times.
  - Since $\sum_{v \in V} |\mathrm{Adj}[v]| = \Theta(|E|)$, the total cost for loop is $\Theta(|E|)$.