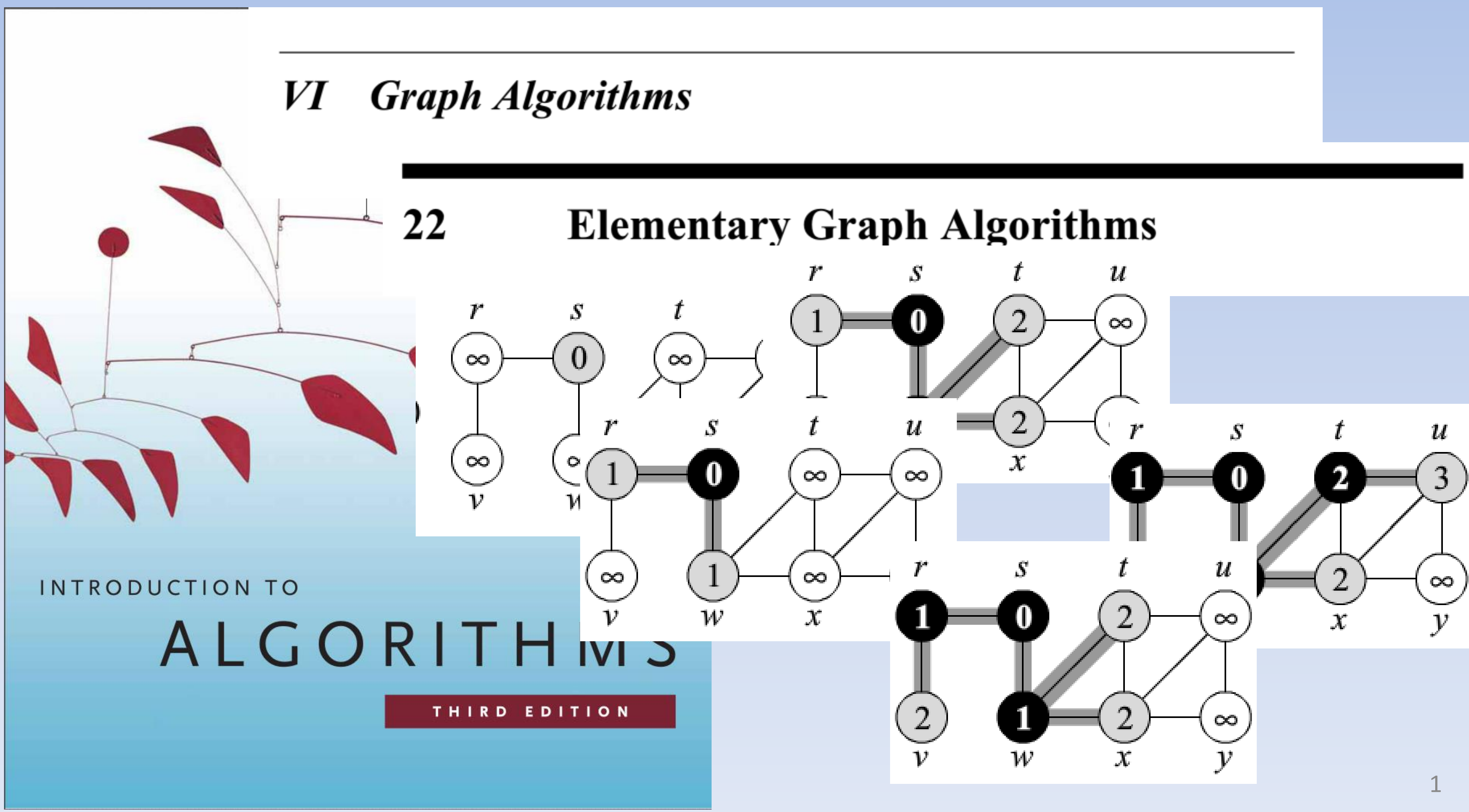
# Design and Analysis of Algorithms

## Section VI : Graph Algorithms

## Chapter 22: Elementary Graph Algorithms



# Graphs:
# Who Cares?

- Check if a network is connected?
  - Phone Network ->
    - Can Caller In California Call India?
- Movie Network
  - Graph of actors/actresses
  - Edge between actors/actresses if they appear together in a movie
  - Bacon Number
    - How many hops to get from actor 1 to Kevin Bacon??

# Graphs:
# Who Cares?


- I'm in Arad, Romania!
  - Sight Seeing
  - Photos
- WAIT:
  - I have a non-refundable
  - ticket leaving out of Bucharest tomorrow!!!
- I NEED TO GET TO BUCHAREST!


- Driving Directions:
- Cities connected w/ Roads
- Find a path from one city to another city?

# Graphs:
# Who Cares?

- Solving Sudoku:
  - Incomplete puzzles connect by directed edge w/ addition of one number.
  - Goal is to find a path from start configuration to a completed puzzle

# Graphs:
# Who Cares?

- Compute the “pieces” of a graph
  - Useful for internet analysis

# Representations

# Binary Search Trees

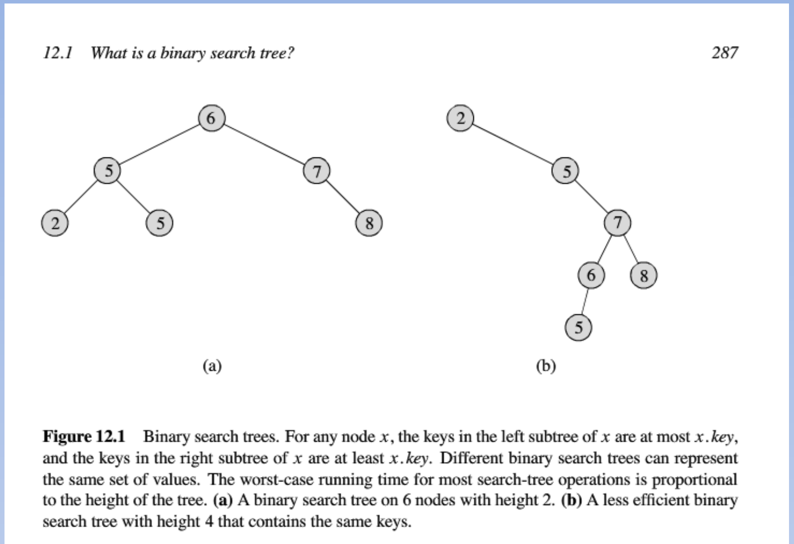
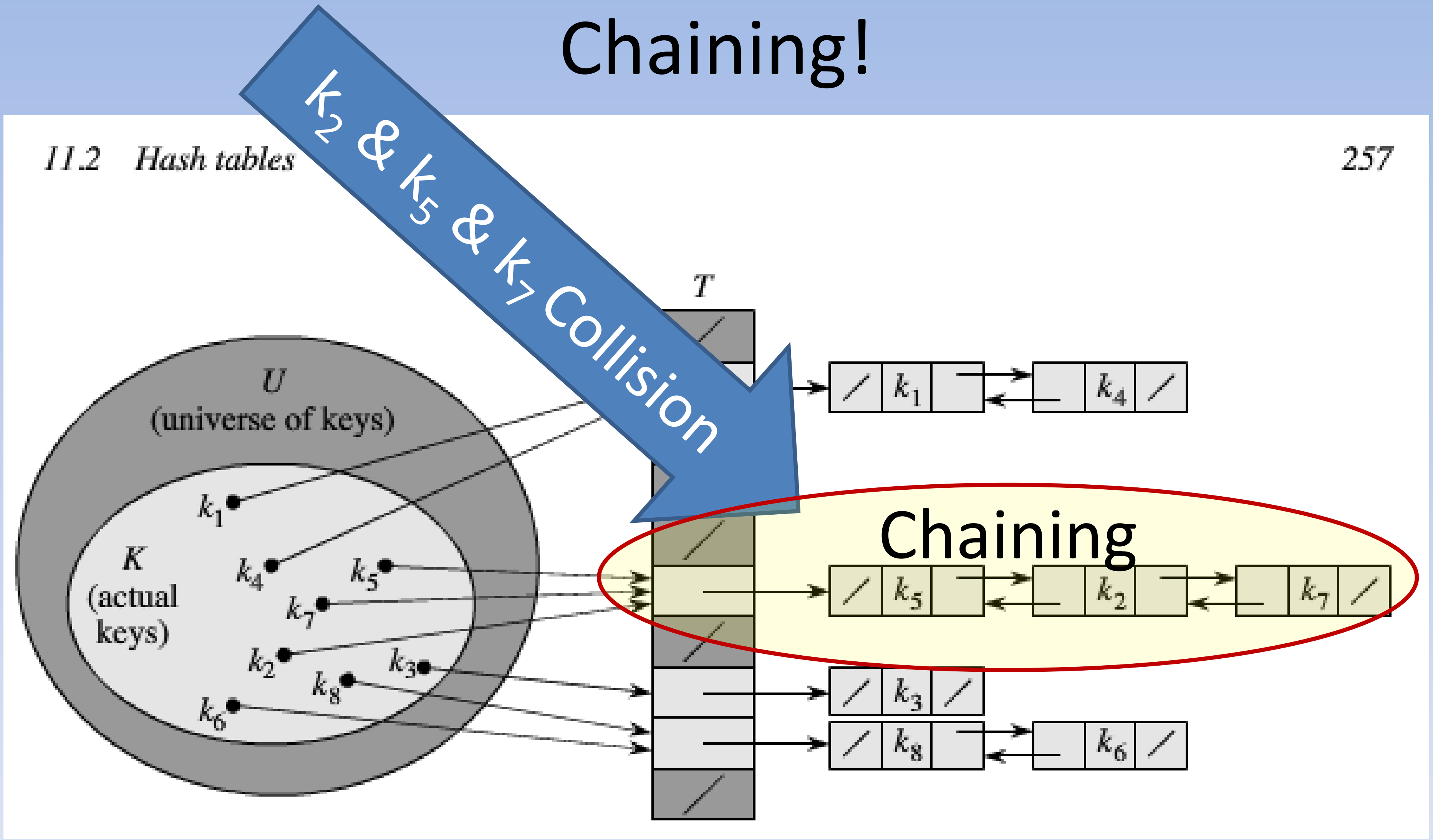

12.1 What is a binary search tree? 287



**Figure 12.1** Binary search trees. For any node $x$, the keys in the left subtree of $x$ are at most $x.key$, and the keys in the right subtree of $x$ are at least $x.key$. Different binary search trees can represent the same set of values. The worst-case running time for most search-tree operations is proportional to the height of the tree. **(a)** A binary search tree on 6 nodes with height 2. **(b)** A less efficient binary search tree with height 4 that contains the same keys.

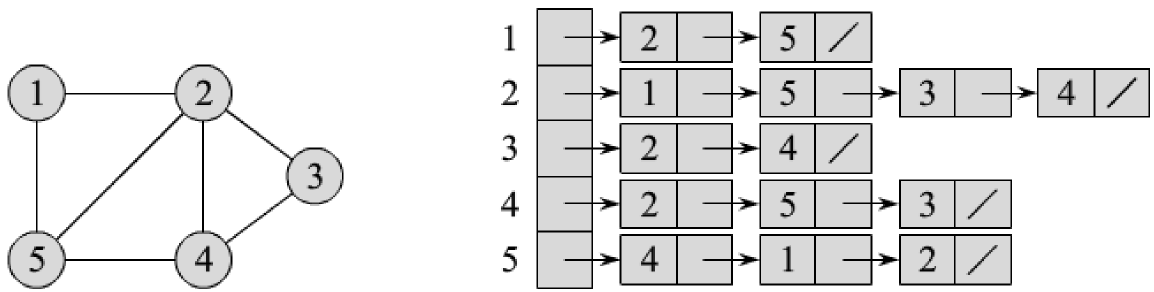
- Linked Data Structure
- Each node has pointers (along with key value and satellite data):
  - p: Parent
  - left: Left Subtree
  - right: Right Subtree

# Hash Tables w/ Chaining!



# Graph Representations
# Undirected



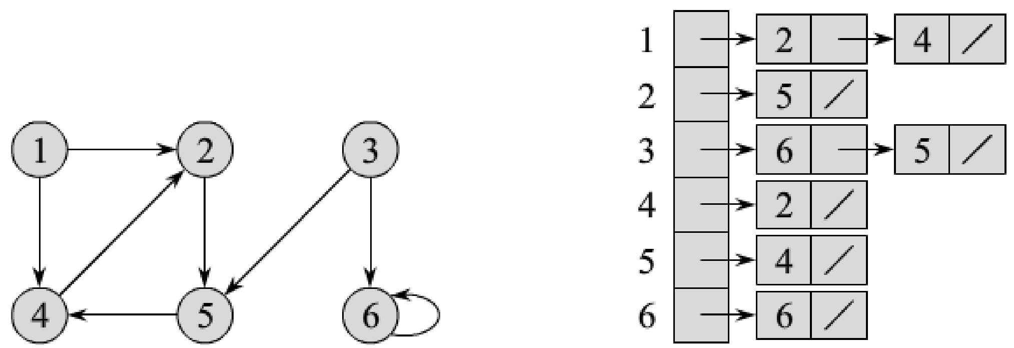



(a) (b)

| | 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 |

(c)

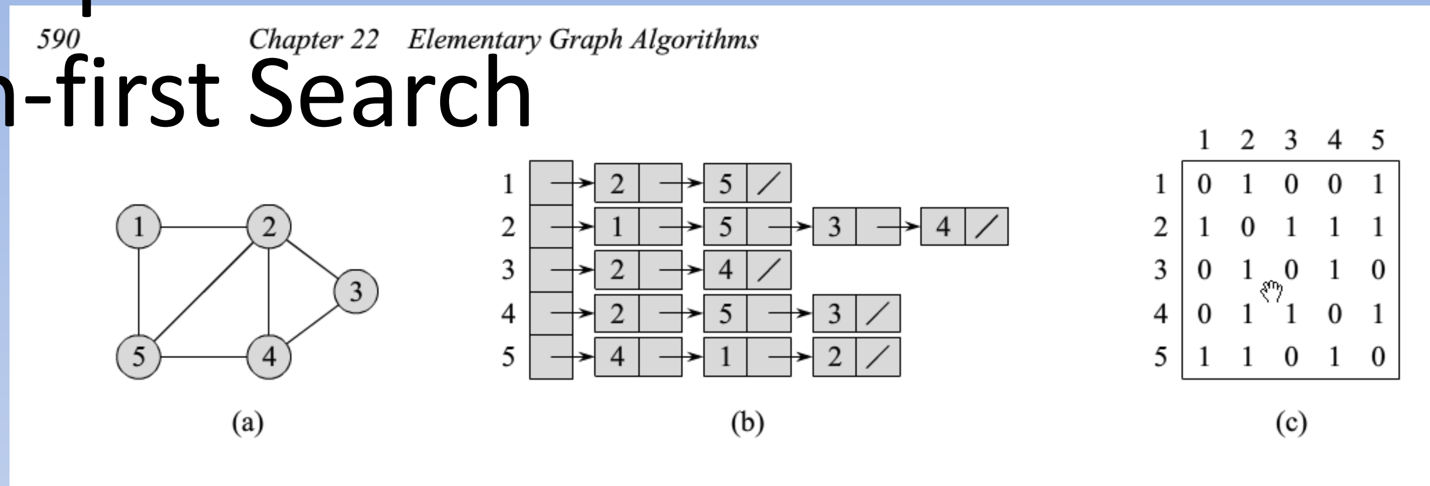
# Graph Representations
# Directed




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

(a) (b) (c)

# Graph Representations: Breadth-first Search




- G=(V,E)
  - G.V
    - All nodes in graph.
  - G.Adj[u]
    - All nodes adjacent to u in graph.
- Nodes $u \in V$
  - u.color
    - white, gray, black
  - u.d
    - Distance from start node
  - u.$\pi$
    - Predecessor graph

# BFS

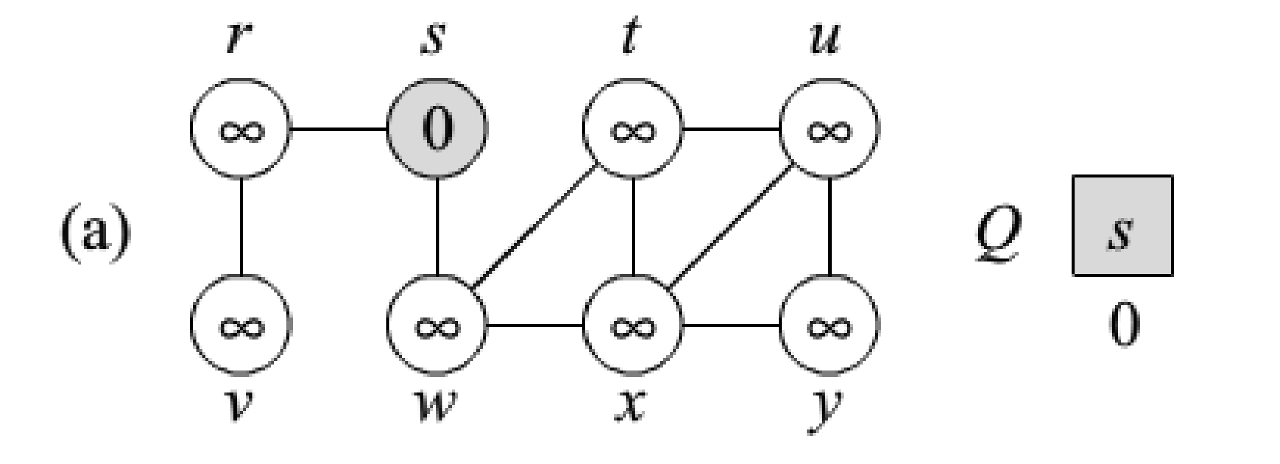
```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u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

# BFS

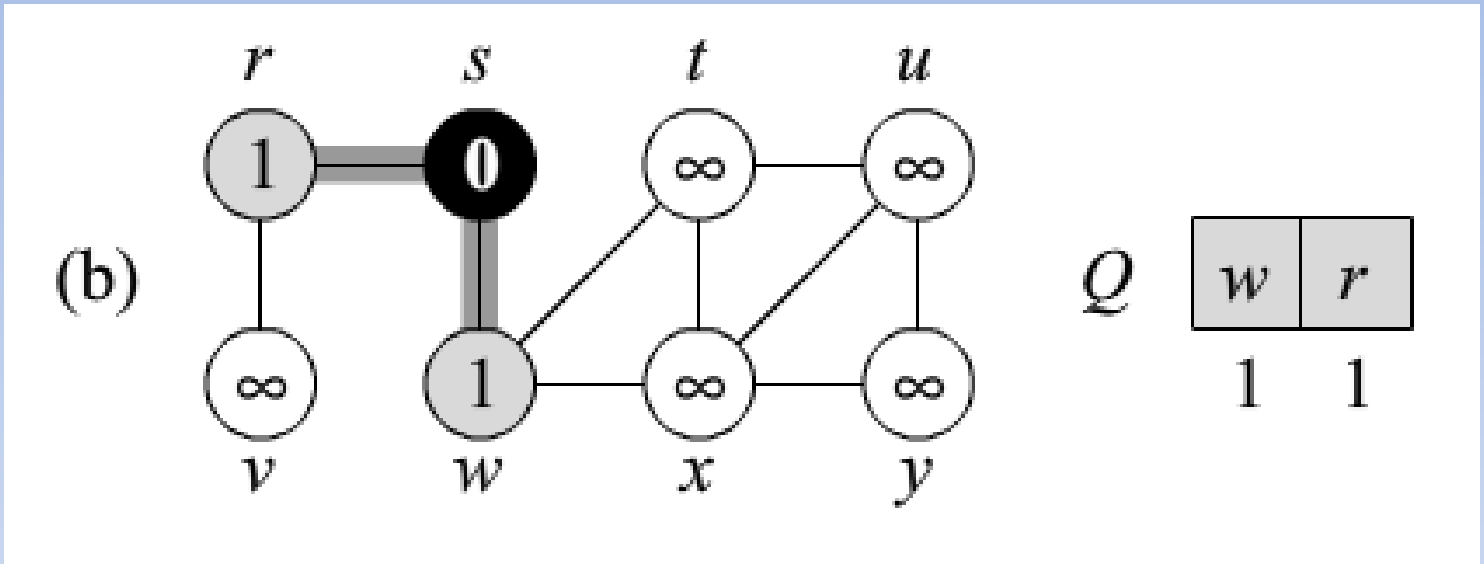
```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u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

Predecessor Subgraph

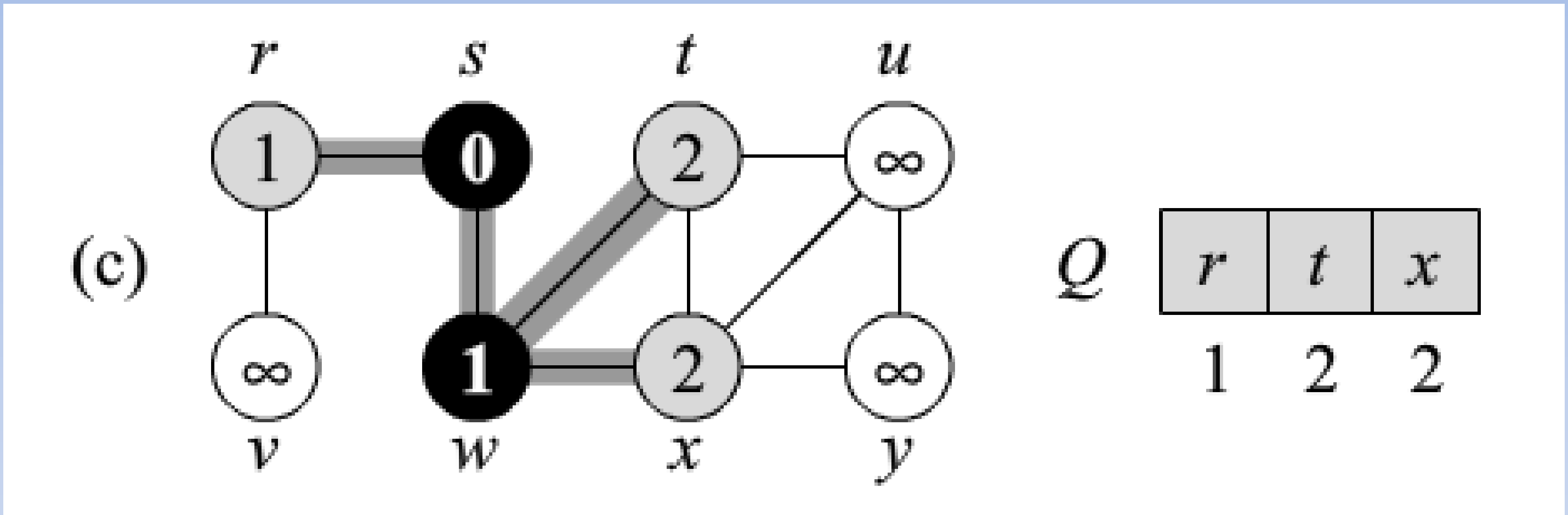
# BFS
# Example



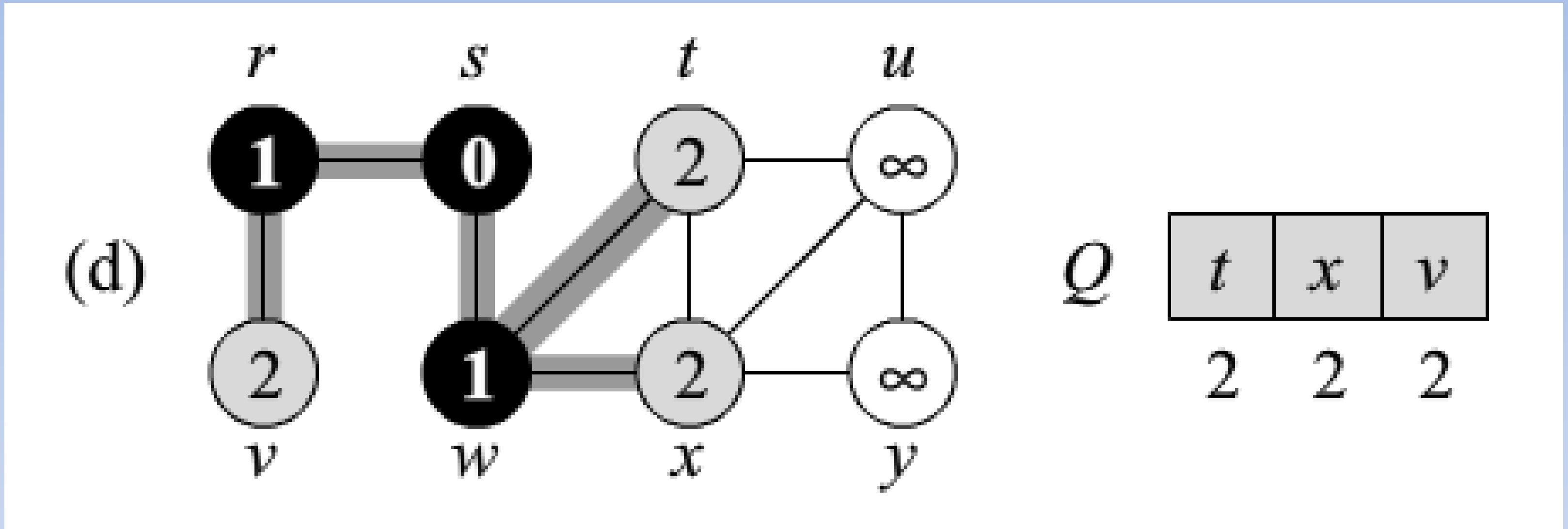
# BFS
# Example



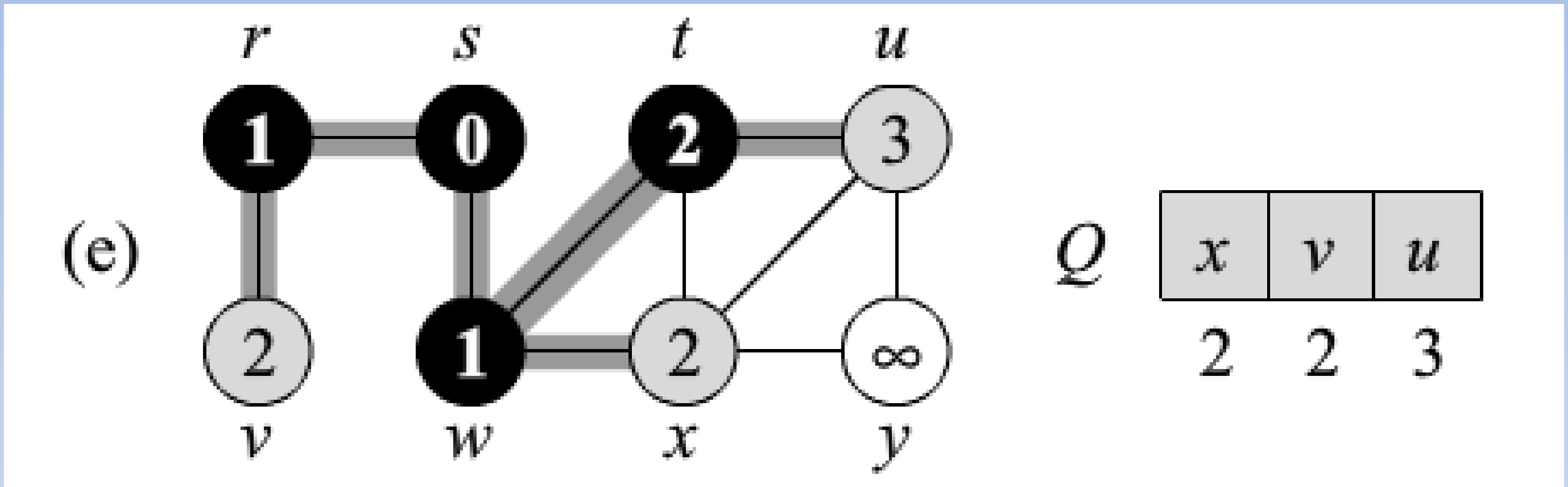
# BFS
# Example



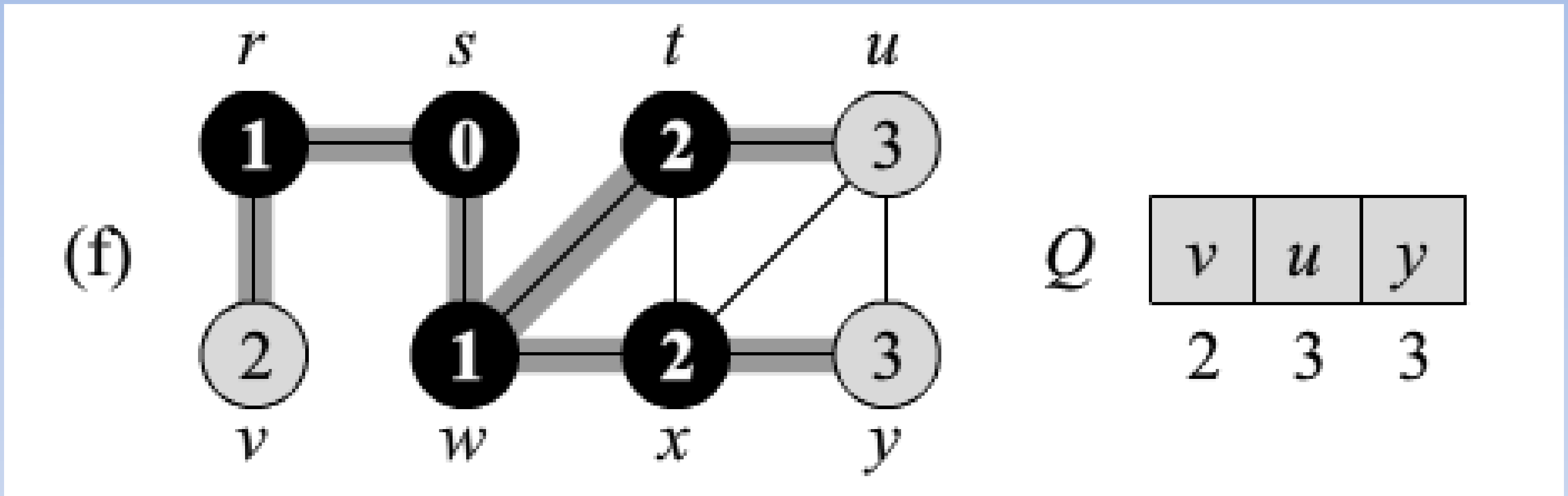
# BFS
# Example



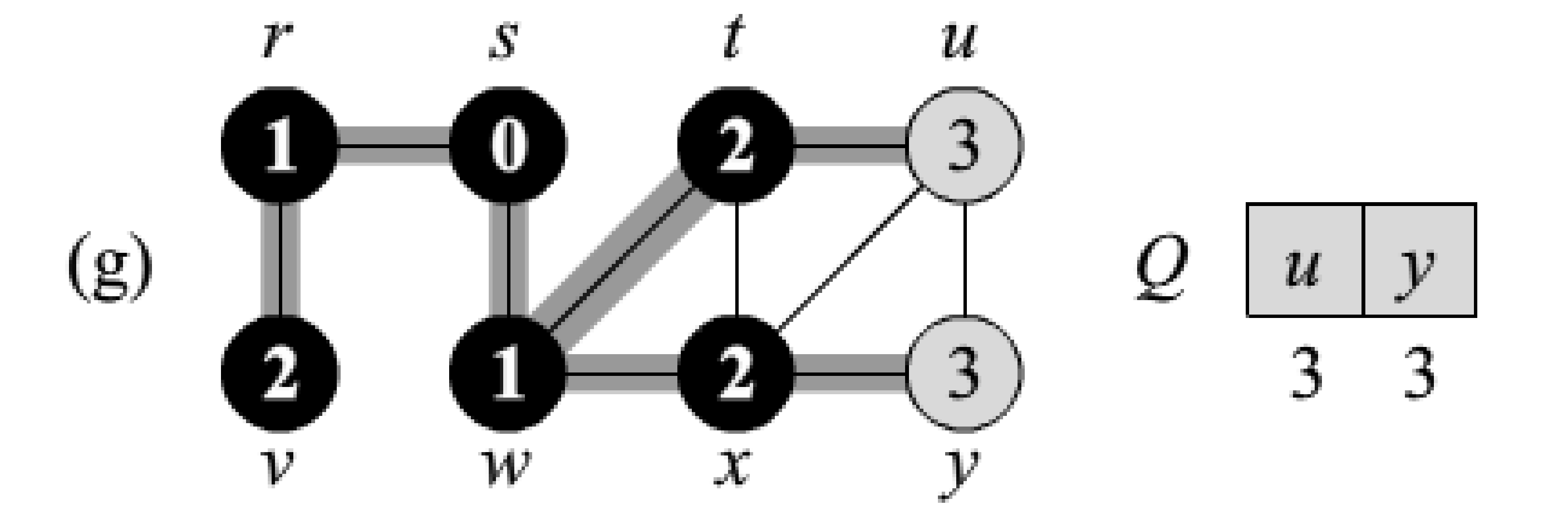
# BFS
# Example



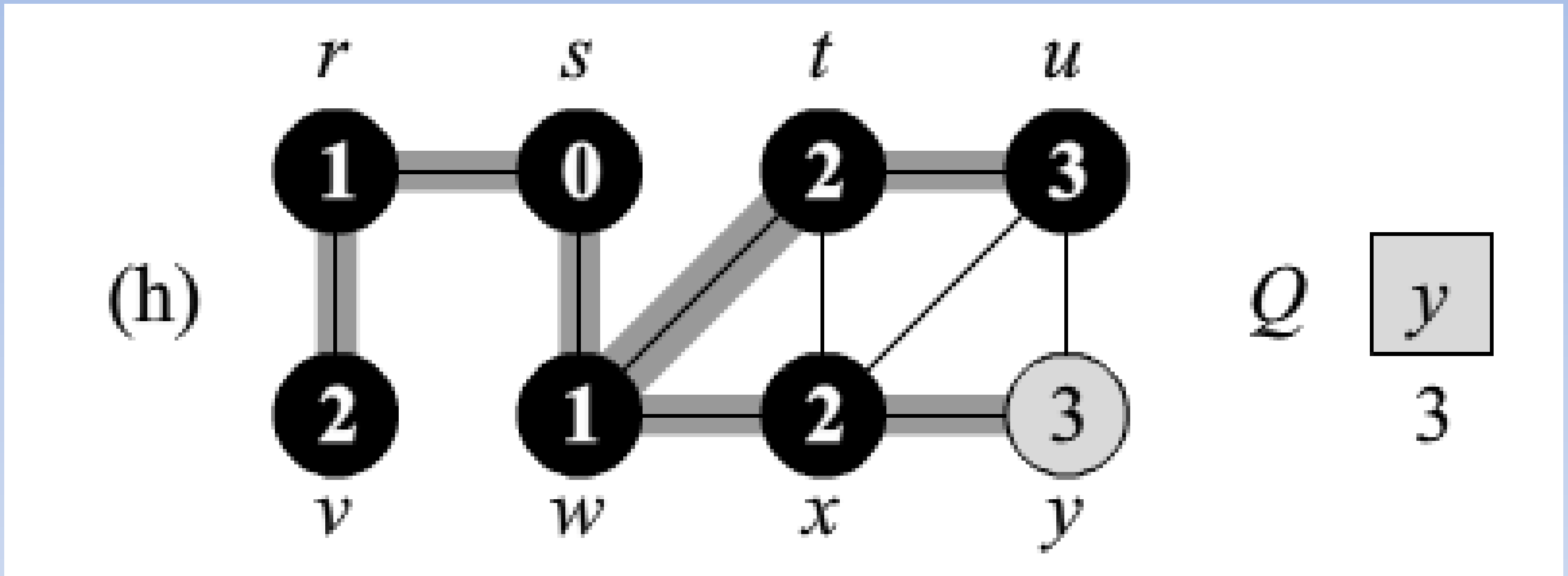
# BFS
# Example



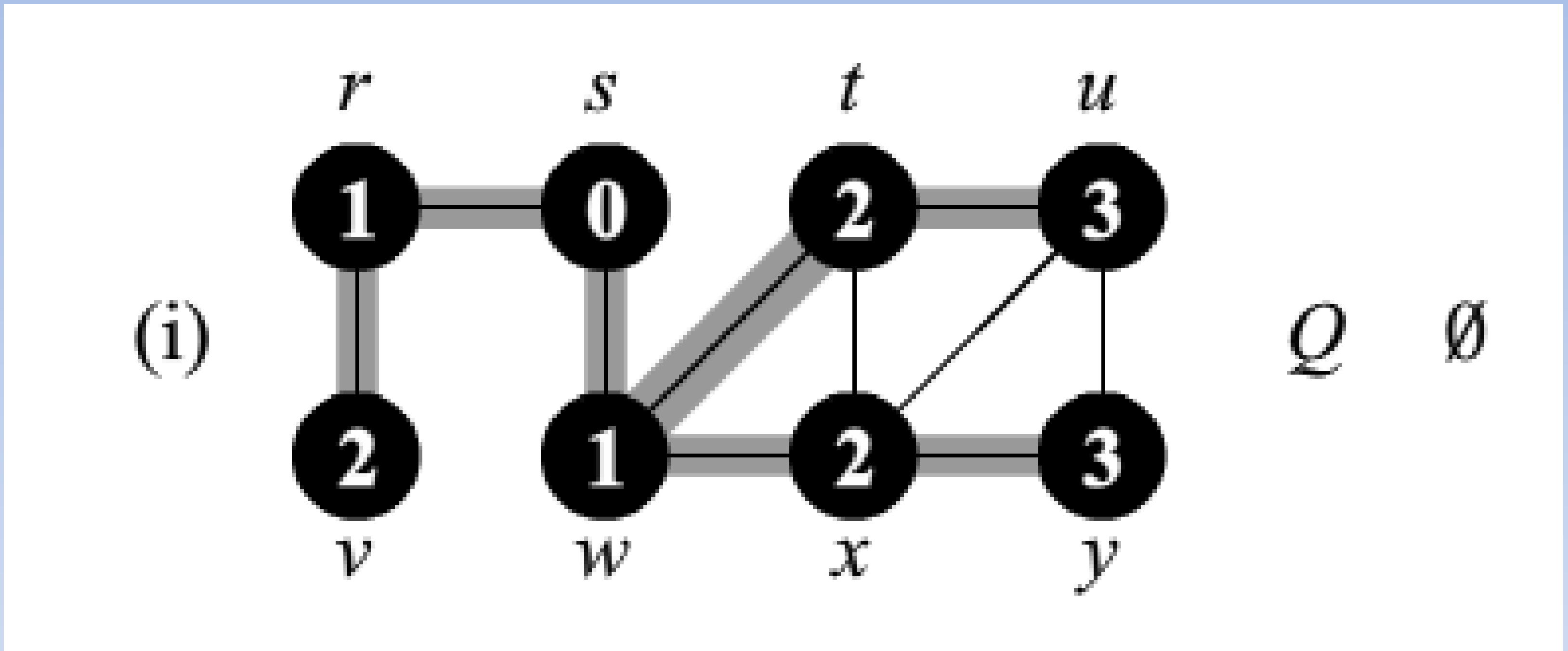
# BFS
# Example



# BFS
# Example



# BFS
# Example

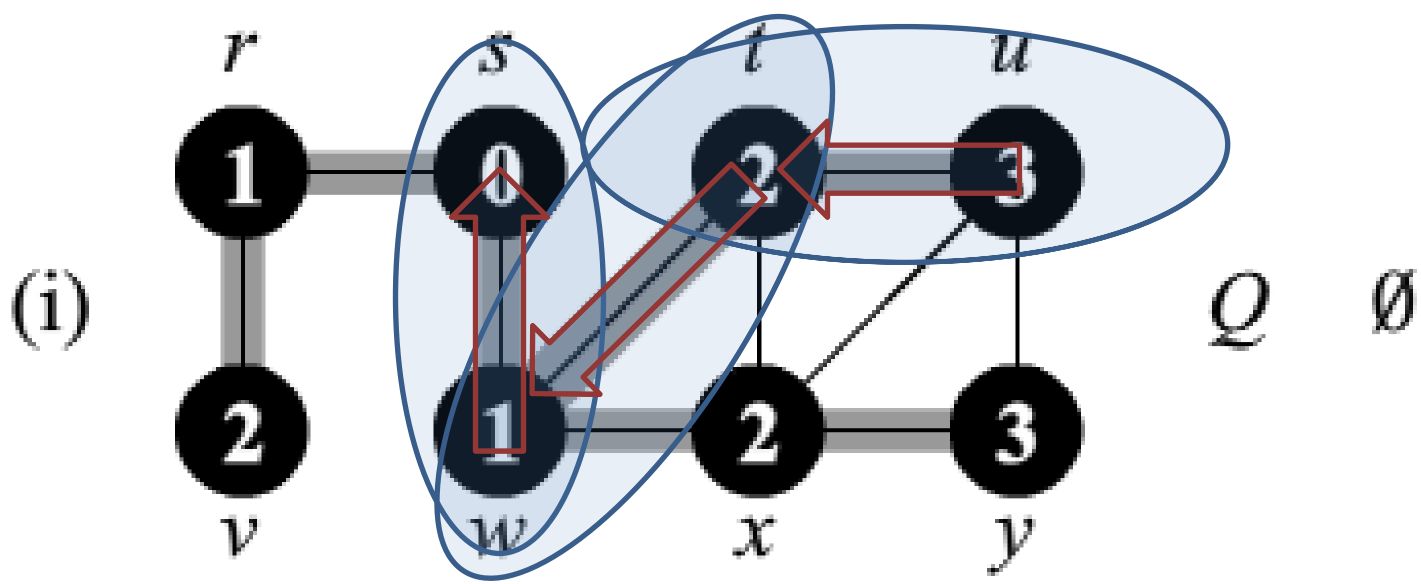


# Shortest Path

```
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
```

# Shortest Path: s->u

```
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
```



# Look @ BFS Proofs

- Given our graph $G=(V,E)$ w/ $s \in V$
- $\delta(s, v)$ **shortest-path distance** from s to v
  - minimum number of edges in any path from vertex s to vertex v.
  - any path of length $\delta(s, v)$ is a **shortest path**

# Lemma 22.1

- Let G=(V,E) be a directed or undirected graph, and let s∈V be an arbitrary vertex. Then, for any edge (u,v)∈E

$$\delta(u,v) \leq \delta(s,u) + 1$$


- There is definitely a path from S-->U->V
  - This path has distance $\delta(s,u) + 1$
- $\delta(s,v)$ the shortest-path distance cannot be greater than the path we know about.

# Lemma 22.2

- Let G=(V,E) be a directed or undirected graph, and suppose that BFS is run on G from a given source vertex s∈V.
- THEN: Upon termination, for each vertex v∈V, the value of v.d computed by BFS satisfies $v.d \geq \delta(s,v)$.

# Lemma 22.2 w/ Proof

- Let G=(V,E) be a directed or undirected graph, and suppose that BFS is run on G from a given source vertex s∈V, Then upon termination, for each vertex v∈V, the value of v.d computed by BFS satisfies $v.d \geq \delta(s, v)$.
- Proof by INDUCTION w/ Enqueue Ops
- Base case w/ s.d=0, and other vertices v.d=∞

# Lemma 22.2 w/ Proof

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

- Consider a white vertex w/ Line 13
- It is adjacent to gray vertex u.
- By Inductive Hypothesis u.$d \;\geq\; \delta(s,u)$
- NOW w/
  - Lemma 22.1
  - Line 15 assignment

$$\begin{aligned} v.d &= u.d + 1 \\ &\geq\; \delta(s,u) + 1 \\ &\geq\; \delta(s,v) \end{aligned}$$

- v is Enqueued  only once, since it turn Gray the first time.
  - v.d is never changes again!

# Lemma 22.3 w/ Corollary 22.4

- Suppose that during the execution of BFS on a graph G=(V,E), the queue Q contains vertices <$v_1$, $v_2$, …, $v_r$>, where $v_1$ is the head of Q and $v_r$ is the tail. THEN

$$v_r.d \leq v_1.d + 1$$

$$v_i.d \leq v_{i+1}.d + 1, i = 1,2, \dots, r-1$$

- Corollary 22.4: Suppose that vertices $v_i$ and $v_j$ are enqueued during the execution of BFS, and that $v_i$ is enqueued befor $v_j$. THEN $v_i.d \leq v_j.d$ at the time $v_j$ is enqueued.

# FINALLY Correctness of BFS w/ Theorem 22.5

- Let G=(V,E) be a directed or undirected graph, and suppose that BFS is run on G from a given source vertex s∈V.
- THEN: BFS discovers every vertex v∈V that is reachable from the source s, and upon termination, v.d=$\delta(s,u)$, ∀v∈V
- MOREOVER: any v≠s, reachable from s, one of the shortest paths from s to v is a shortest path from s->>v.π->v.
- PROOF BY CONTRADICTION!

# Proof By Contradiction

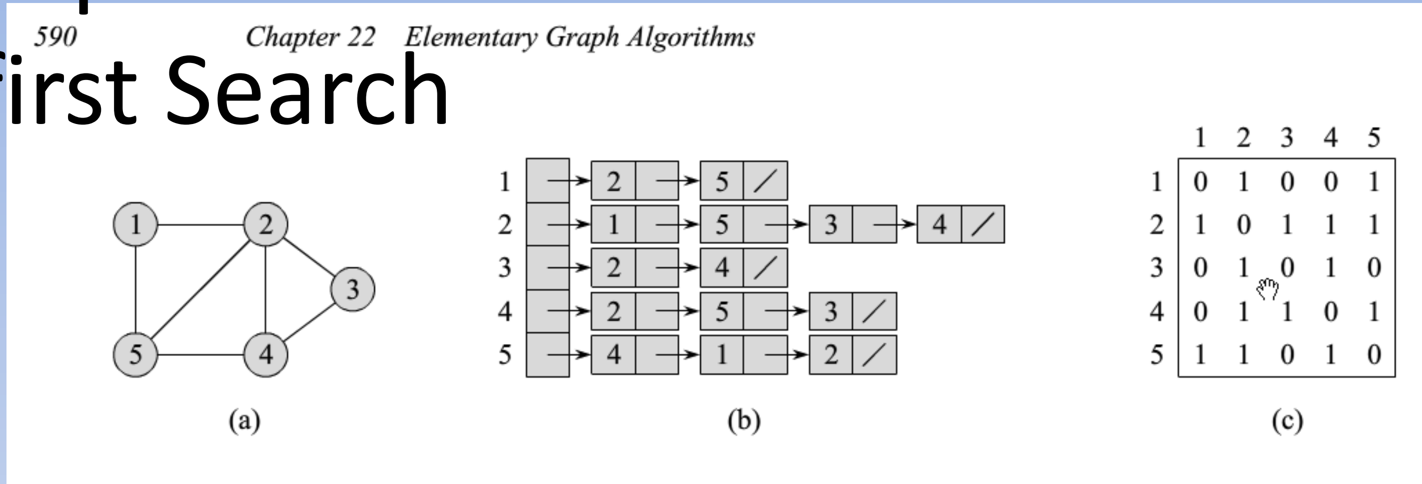
- Assume some vertex v is gets the incorrect v.d
- By Lemma 22.2 $v.d \geq \delta(s,u) + 1 = u.d + 1$
- Let u be the vertex right before v on the shortest path, so:
  - $\delta(s,v) = \delta(s,u) + 1$
- Because of how we chose v, u.d must equal $\delta(s,u)$
- So v.d must be greater than u.d+1

# Proof By Contradiction

```
10 while Q ≠ Ø
11     u = DEQUEUE(Q)
12     for each v ∈ G.Adj[u]
13         if v.color == WHITE
14             v.color = GRAY
15             v.d = u.d + 1
16             v.π = u
17             ENQUEUE(Q, v)
18     u.color = BLACK
```

- v.d > u.d + 1
- Now when u was dequeued
  - If v was WHITE:
    - line 15 sets v.d=u.d+1
    - CONTRADICTION
  - If v was BLACK it was already dequeued so by Cor 22.4: $v.d \leq u.d$
  - If v was GRAY: it must have been grayed when dequeuing some node w which was removed earlier than u
    - so by Cor 22.4 w. $d \leq u.d$ so $v.d = w.d + 1 \ \leq u.d + 1$
      - CONTRADICTION!

# Graph Representations: Depth-first Search



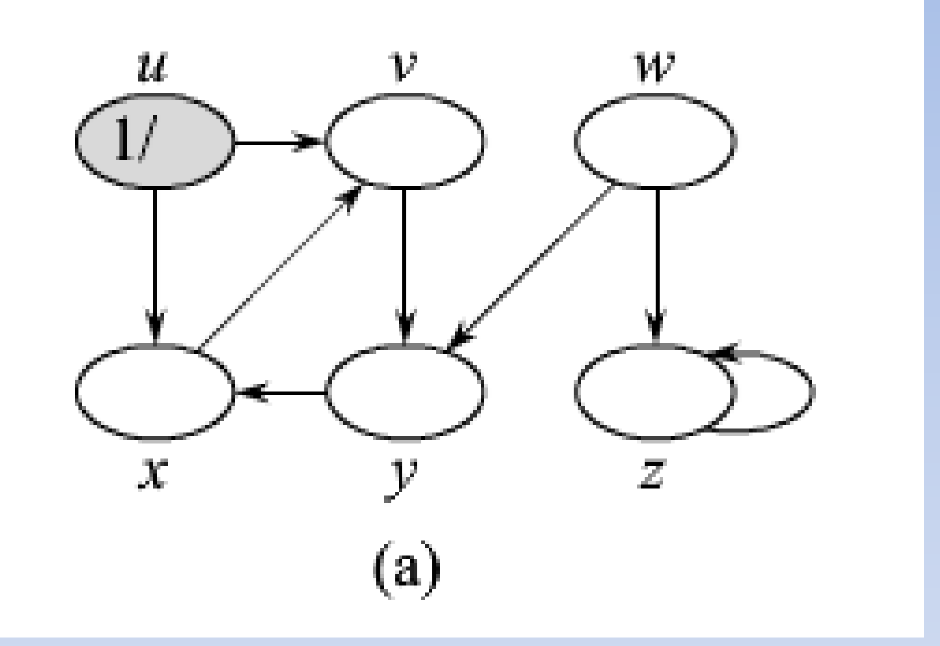

- Time
- G=(V,E)
  - G.V
    - All nodes in graph.
  - G.Adj[u]
    - All nodes adjacent to u in graph.
- Nodes $u \in V$
  - u.color
    - white, gray, black
  - u.d
    - Discovery time
  - u.f
    - Finish time
  - u.$\pi$
    - Predecessor graph

# DFS

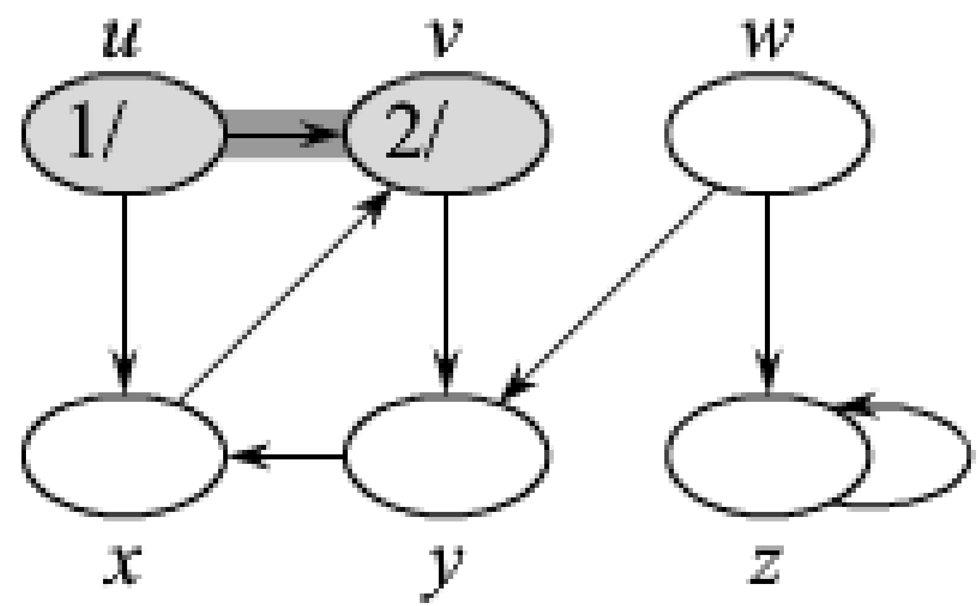
```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-VISIT(G, u)

DFS-VISIT(G, u)
time = time + 1                 // white vertex u has just been discovered
u.d = time
u.color = GRAY
for each v ∈ G.Adj[u]           // explore edge (u, v)
    if v.color == WHITE
        v.π = u
        DFS-VISIT(G, v)
u.color = BLACK                 // blacken u; it is finished
time = time + 1
u.f = time
```

# DFS
# Example

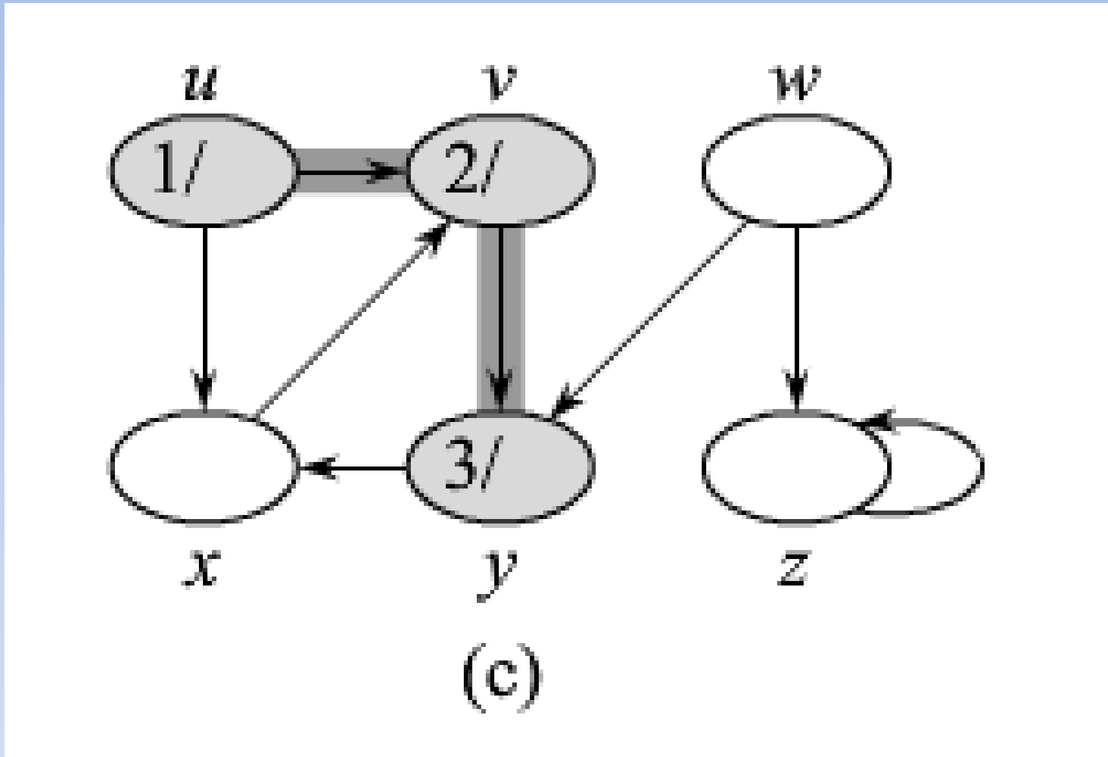


# DFS
# Example

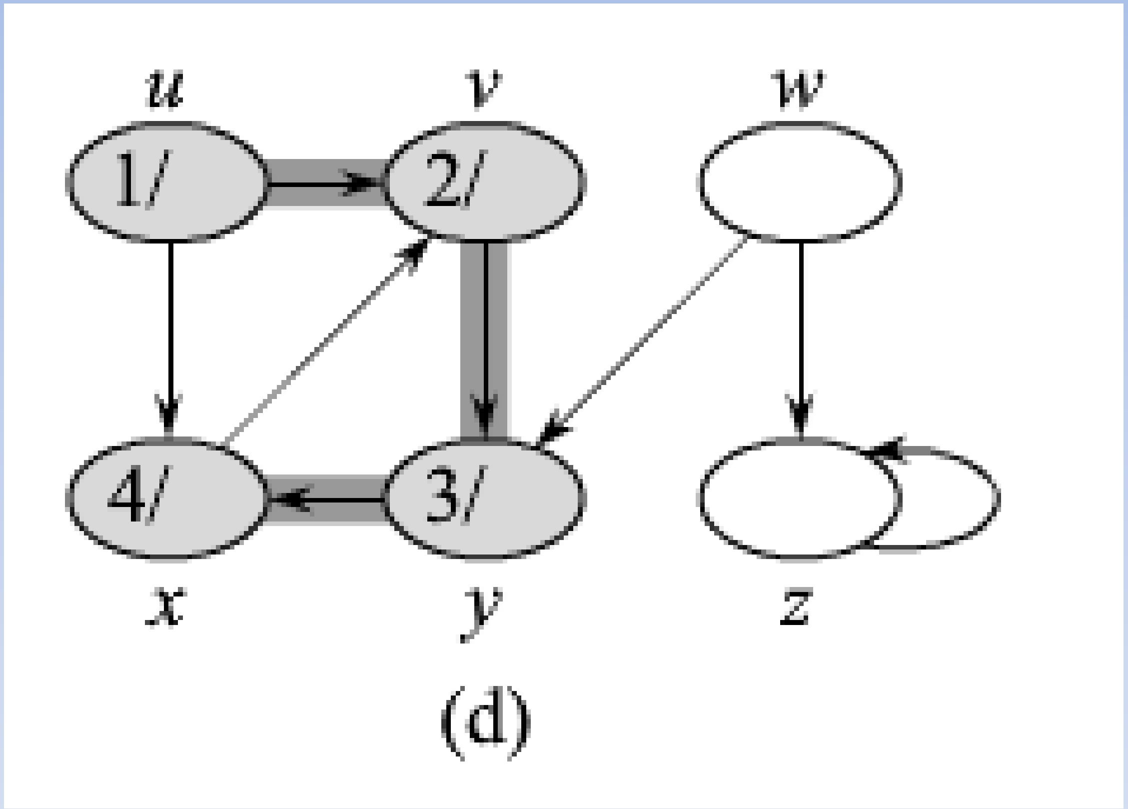



(b)

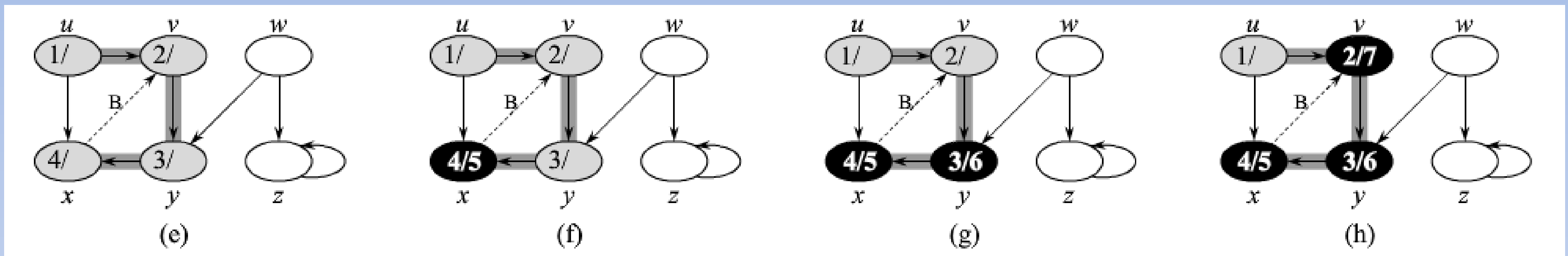
# DFS
# Example



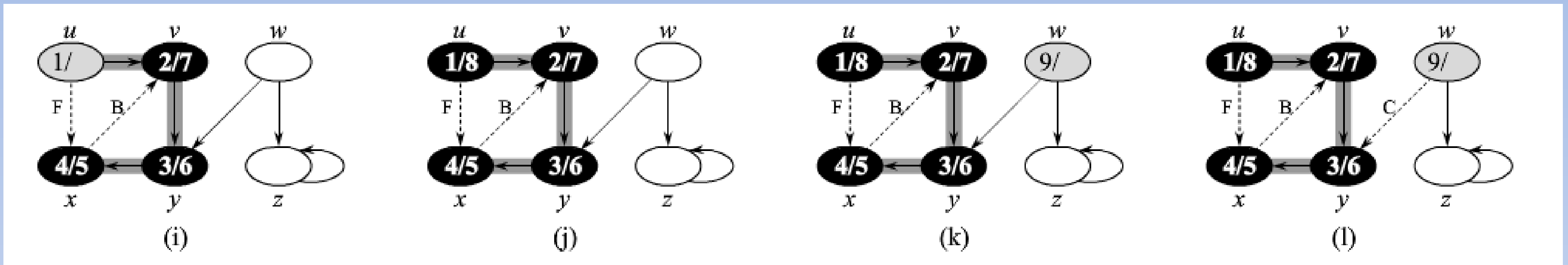
# DFS
# Example



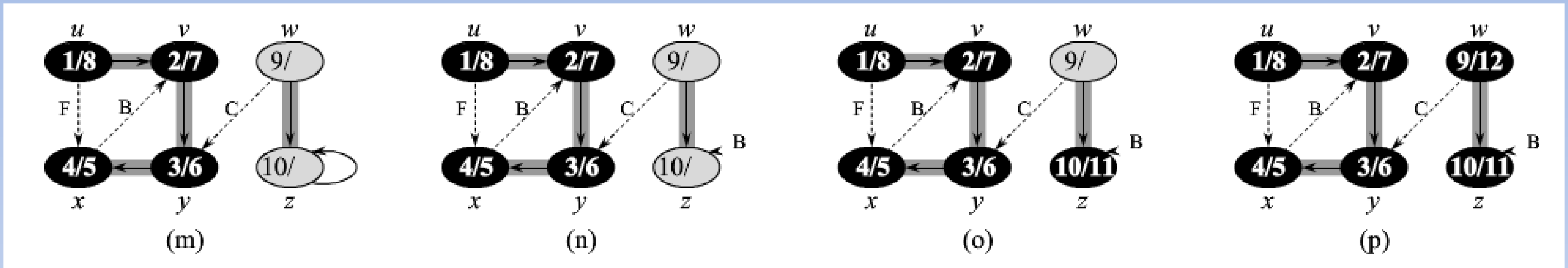
# DFS
# Example


u
v
w
1/
2/
B
4/
3/
x
y
z
(e)
u
v
w
1/
2/
B
4/5
3/
x
y
z
(f)
u
v
w
1/
2/
B
4/5
3/6
x
y
z
(g)
u
v
w
1/
2/7
B
4/5
3/6
x
y
z
(h)

# DFS
# Example



# DFS
# Example



u v w
1/8 2/7 9/
F B C
4/5 3/6 10/
x y z
(m)
u v w
1/8 2/7 9/
F B C B
4/5 3/6 10/
x y z
(n)
u v w
1/8 2/7 9/
F B C B
4/5 3/6 10/11
x y z
(o)
u v w
1/8 2/7 9/12
F B C B
4/5 3/6 10/11
x y z
(p)

# DFS

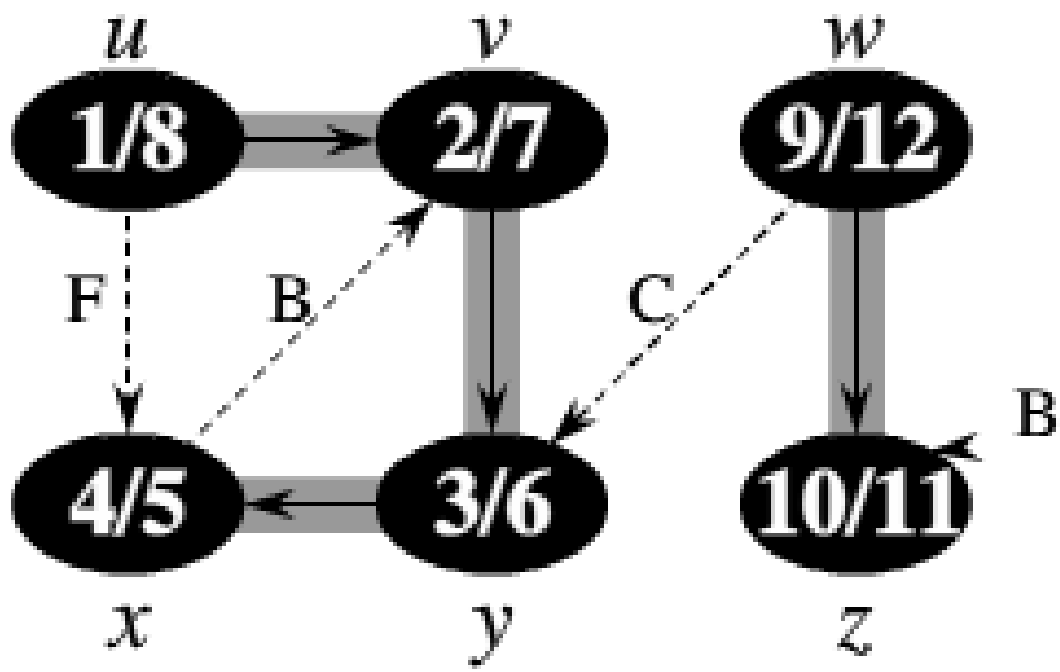
```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-VISIT(G, u)

DFS-VISIT(G, u)
time = time + 1                // white vertex u has just been discovered
u.d = time
u.color = GRAY
for each v ∈ G.Adj[u]          // explore edge (u, v)
    if v.color == WHITE
        v.π = u
        DFS-VISIT(G, v)
u.color = BLACK                // blacken u; it is finished
time = time + 1
u.f = time
```

# Edge Types

- Tree Edges: Edges in the depth-first forest
- Back Edges: Nontree edges connecting a vertex to a ancestor
- Forward Edges: Nontree edges connecting a vertex to a descendant
- Cross Edges: All others (generalized cousins).

# Edge Types



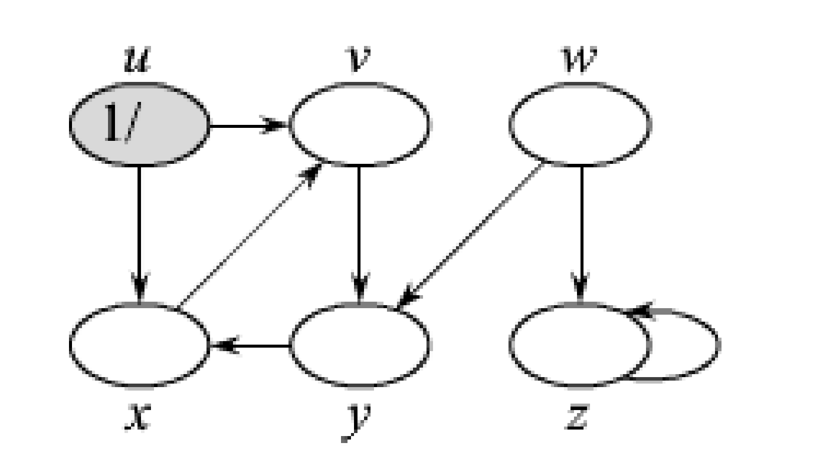

(p)

# Determining Edge Types

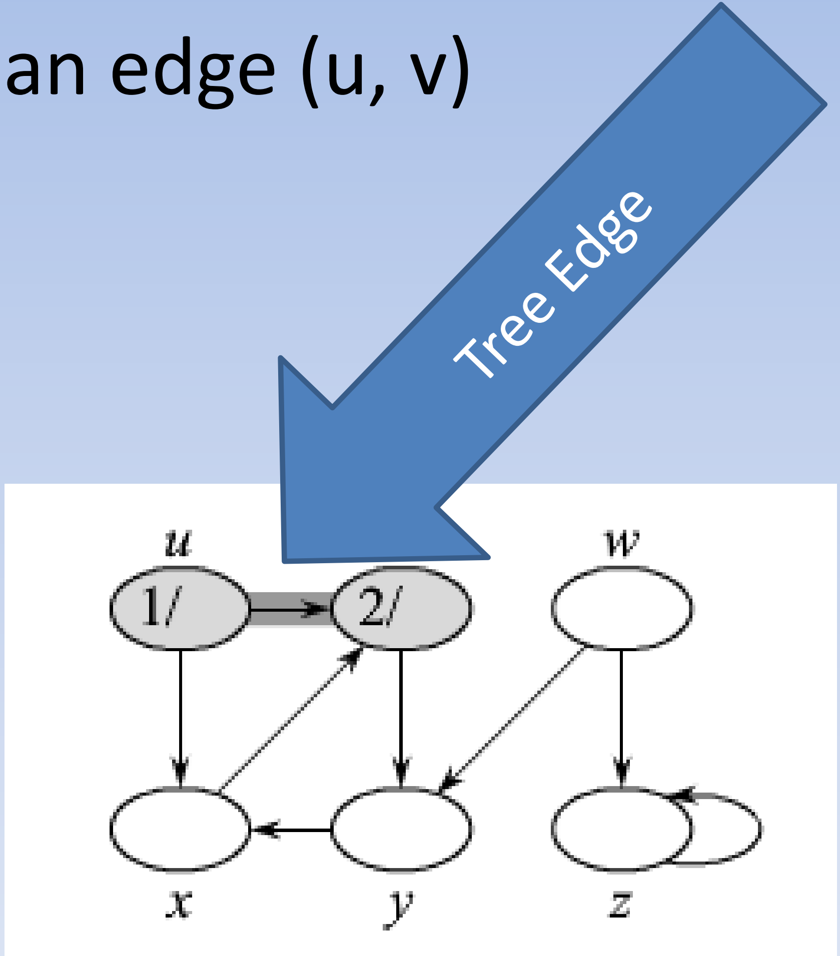
- When first exploring an edge (u, v)
- If color V:
  - WHITE: tree edge
  - GRAY: back edge
  - BLACK: forward edge or cross edge
    - u.d < v.d: forward edge
    - u.d > v.d: cross edge

# Determining Edge Types: Tree Edge

- When first exploring an edge (u, v)
- If color V:
  - WHITE: tree edge

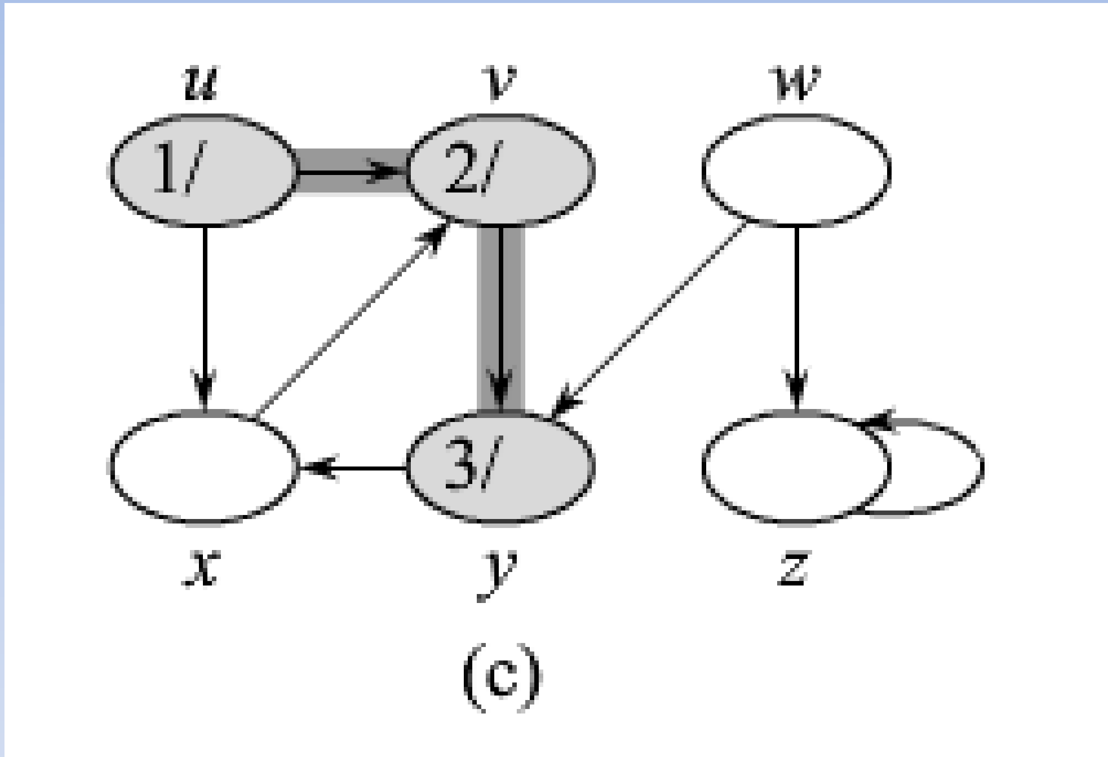



(a)

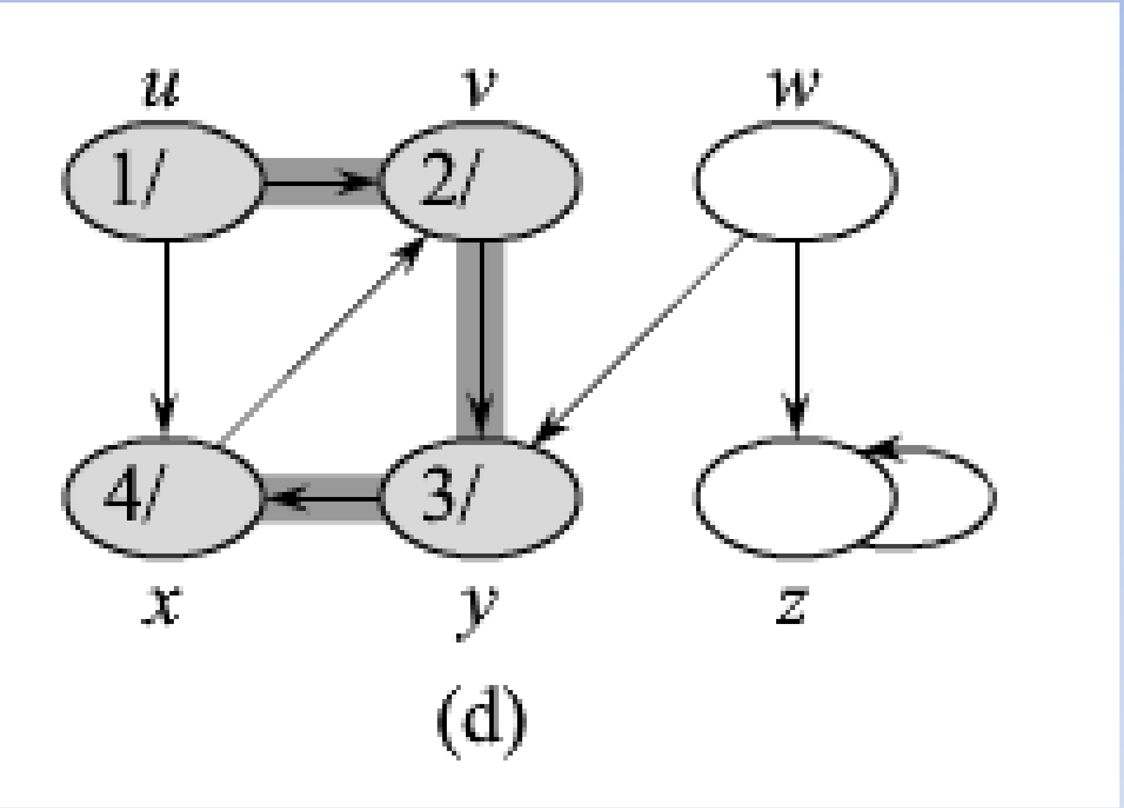



(b)

# Determining Edge Types: Tree Edge

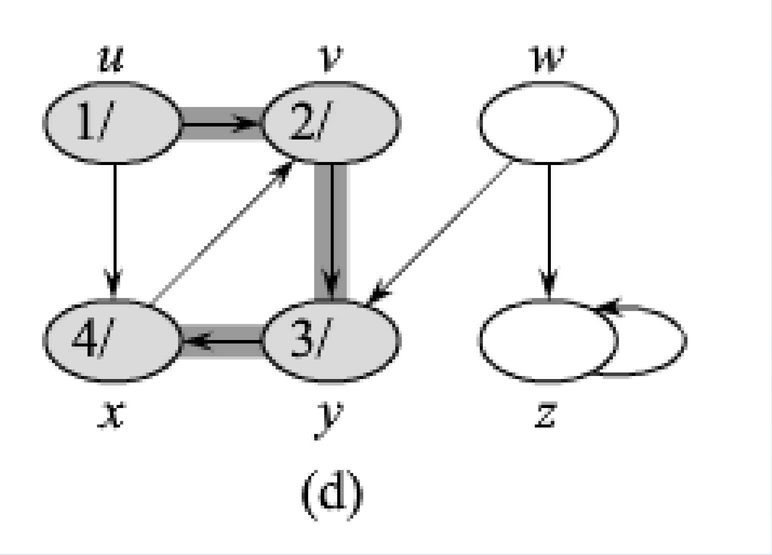


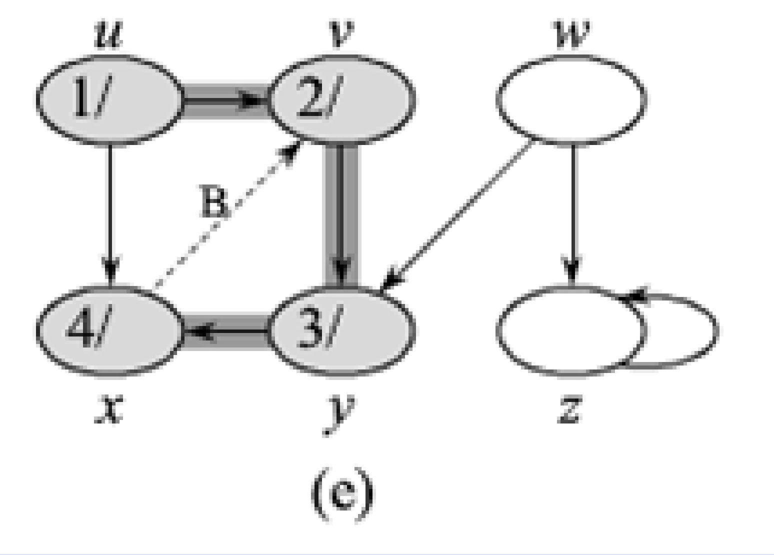
# Determining Edge Types: Tree Edge



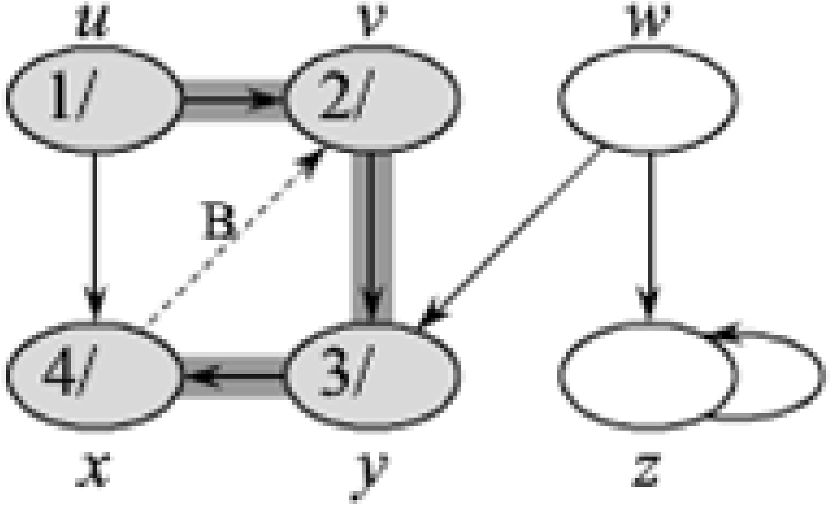
# Determining Edge Types: Back Edge

- When first exploring an edge (u, v)
- If color V:
  - WHITE: tree edge
  - **GRAY: back edge**

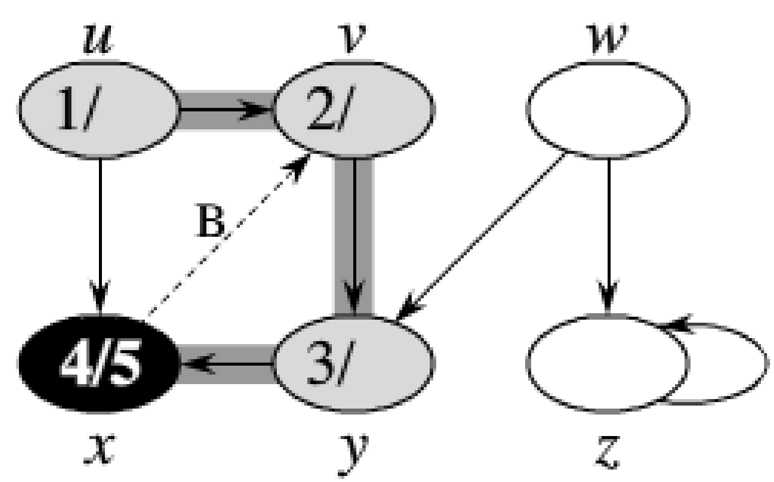


# DFS Example:
# X Finishes

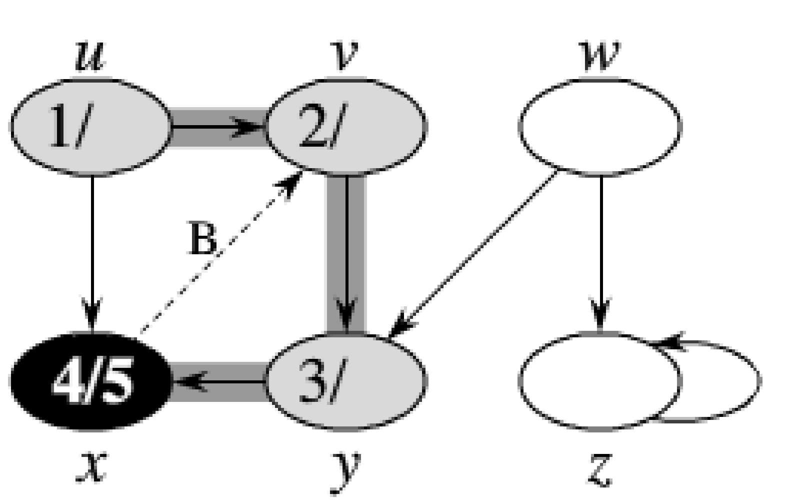



(e)

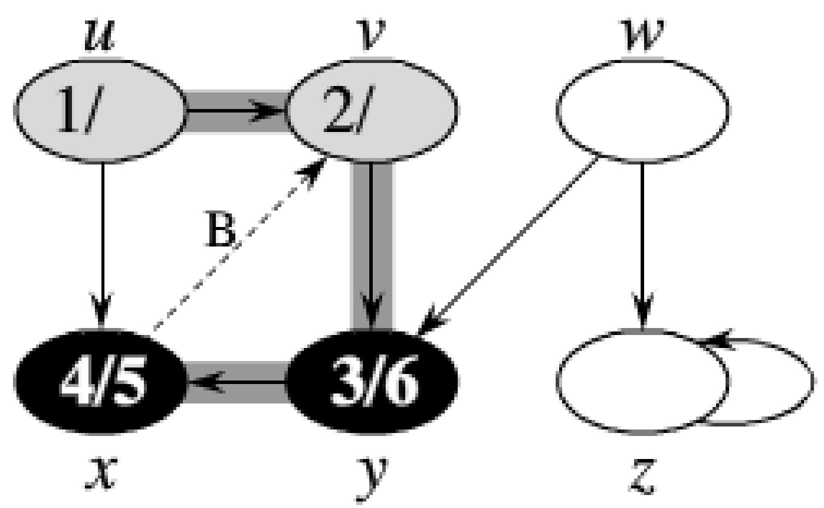



(f)

# DFS Example:
# y Finishes

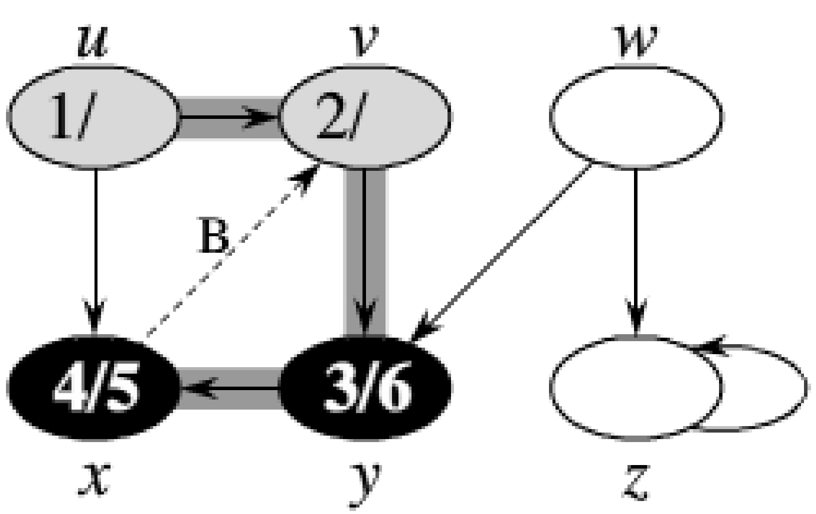


(f)

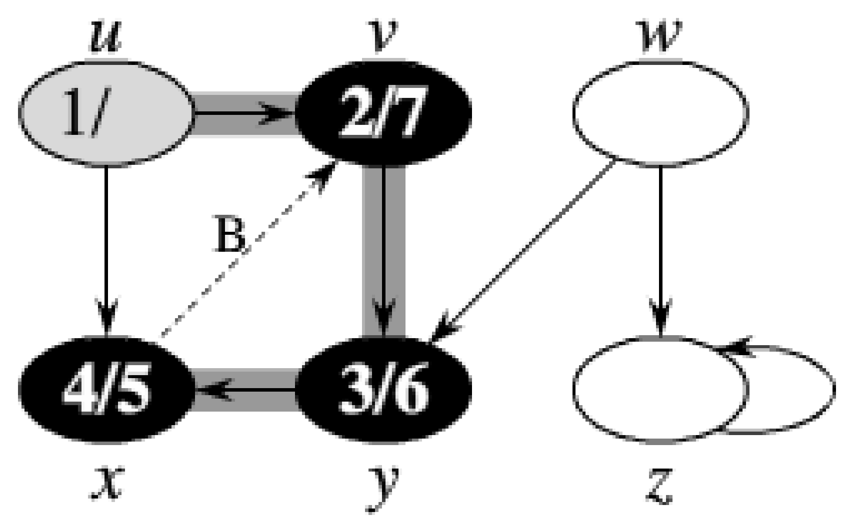


(g)

# DFS Example: v Finishes



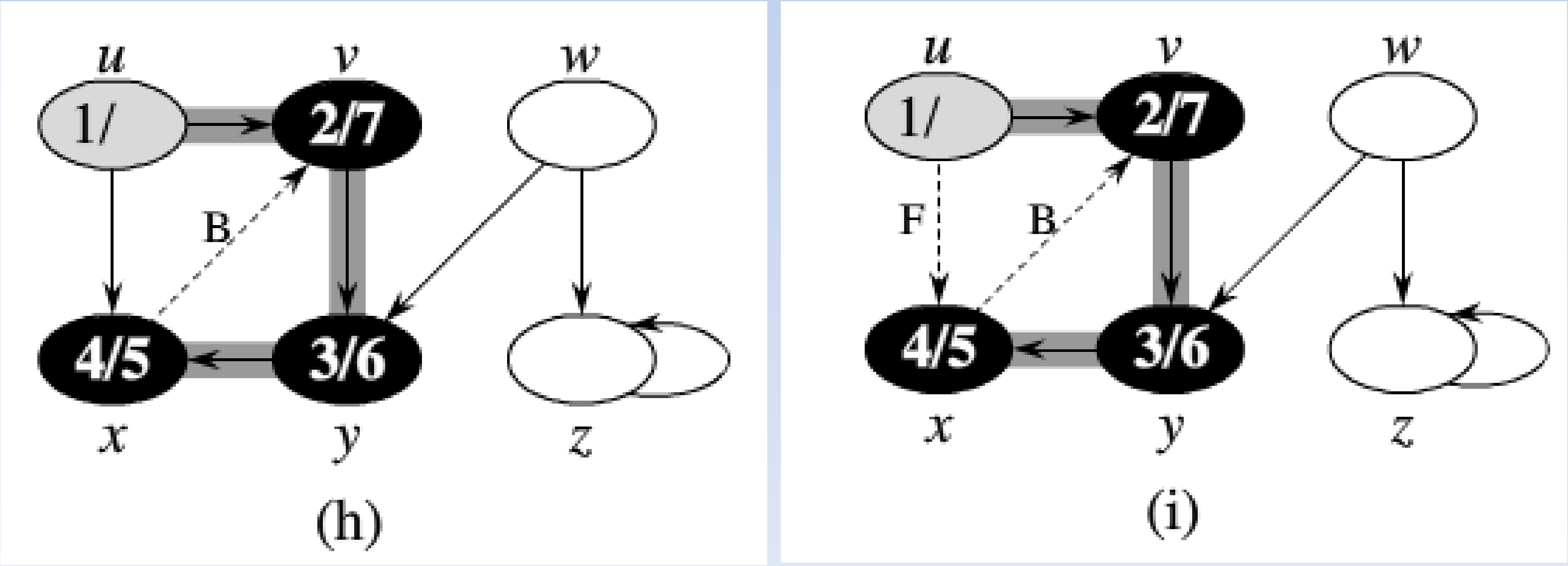

(g)




(h)

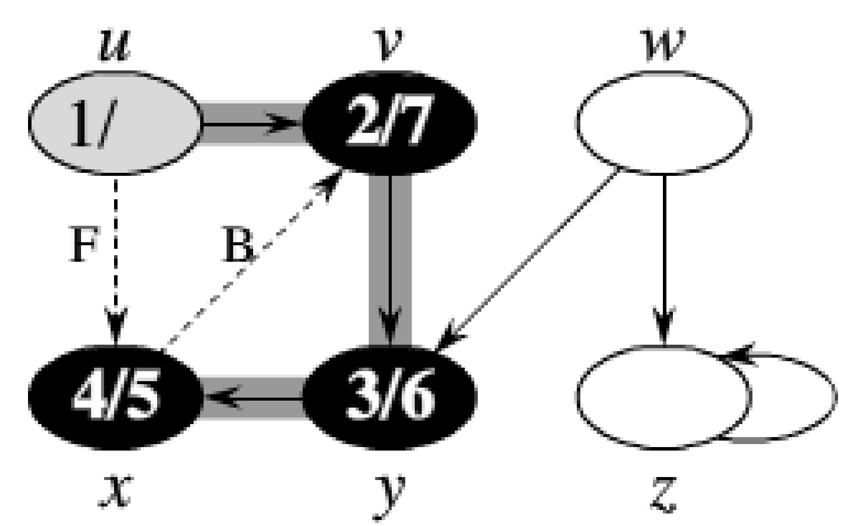
# DFS Example:
# u Discovers Forward Edge

- When first exploring an edge (u, v)
- If color V:
  - WHITE: tree edge
  - GRAY: back edge
  - BLACK: forward edge or cross edge
    - u.d < v.d: forward edge // **u.d=1 & x.d=4**
    - u.d > v.d: cross edge

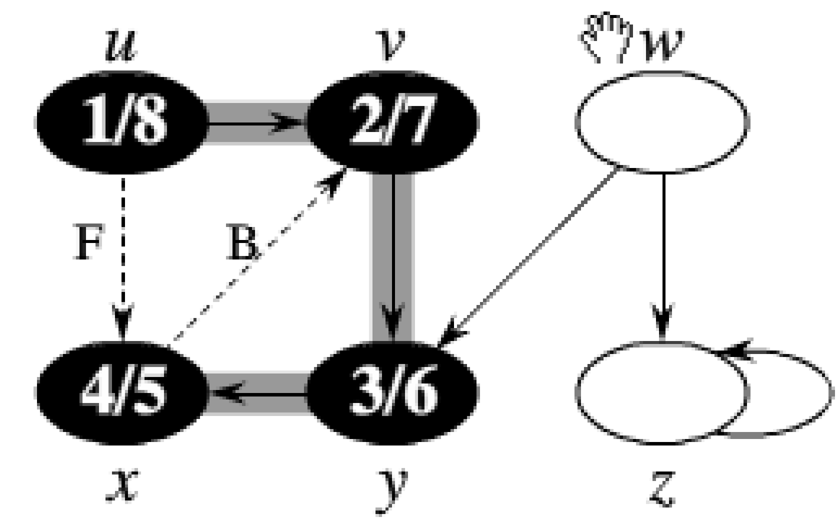


# DFS Example: u Finishes

- When first exploring an edge (u, v)
- If color V:
  - WHITE: tree edge
  - GRAY: back edge
  - BLACK: forward edge or cross edge
    - u.d < v.d: forward edge // **u.d=1 & x.d=4**
    - u.d > v.d: cross edge



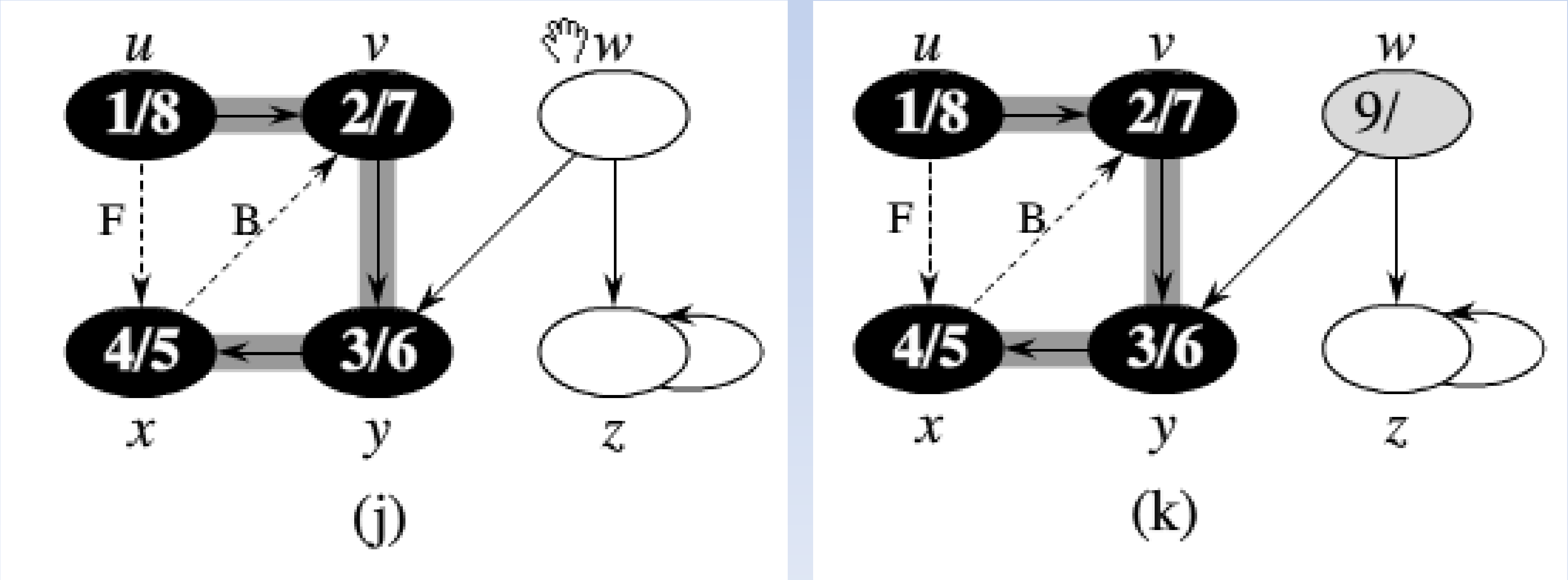

(i)




(j)

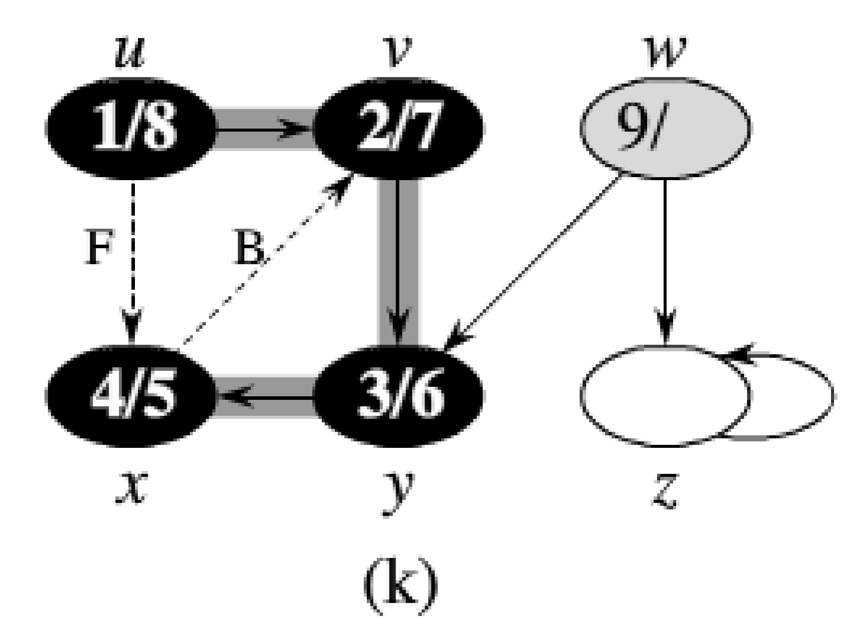
# DFS Example:
# w discovered @ Time = 9

- When first exploring an edge (u, v)
- If color V:
  - WHITE: tree edge
  - GRAY: back edge
  - BLACK: forward edge or cross edge
    - u.d < v.d: forward edge   // **u.d=1 & x.d=4**
    - u.d > v.d: cross edge

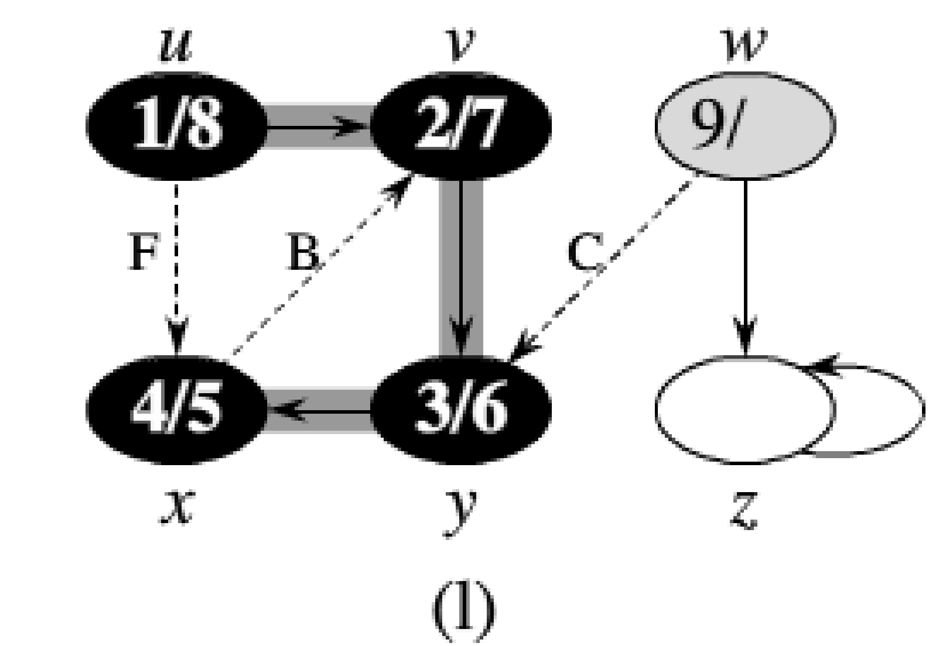


# DFS Example: Cross Edge

- When first exploring an edge (u, v)
- If color V:
  - WHITE: tree edge
  - GRAY: back edge
  - BLACK: forward edge or cross edge
    - u.d < v.d: forward edge
    - u.d > v.d: cross edge **// w.d=9 & y.d=3**



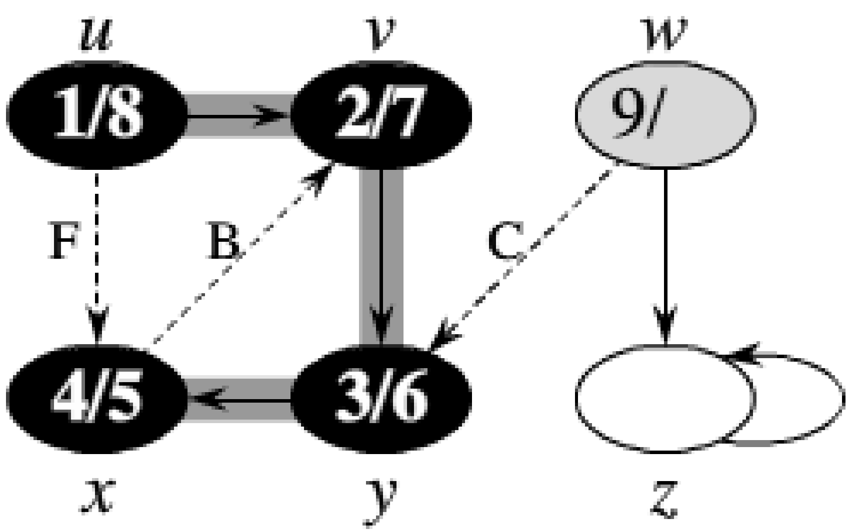
(k)



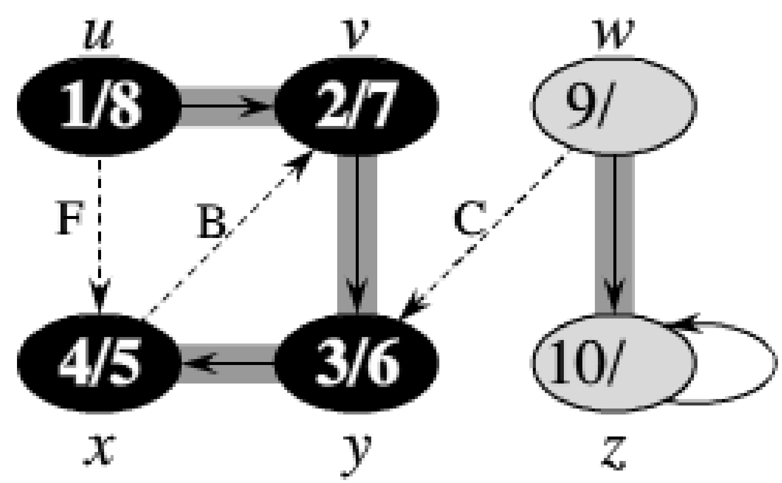
(l)

# DFS Example:
# z discovered @ time=10

- When first exploring an edge (u, v)
- If color V:
  - WHITE: tree edge
  - GRAY: back edge
  - BLACK: forward edge or cross edge
    - u.d < v.d: forward edge
    - u.d > v.d: cross edge



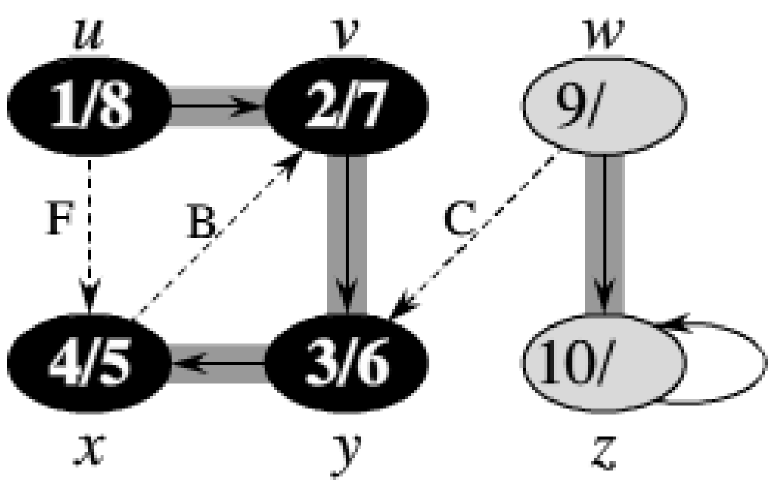

(l)



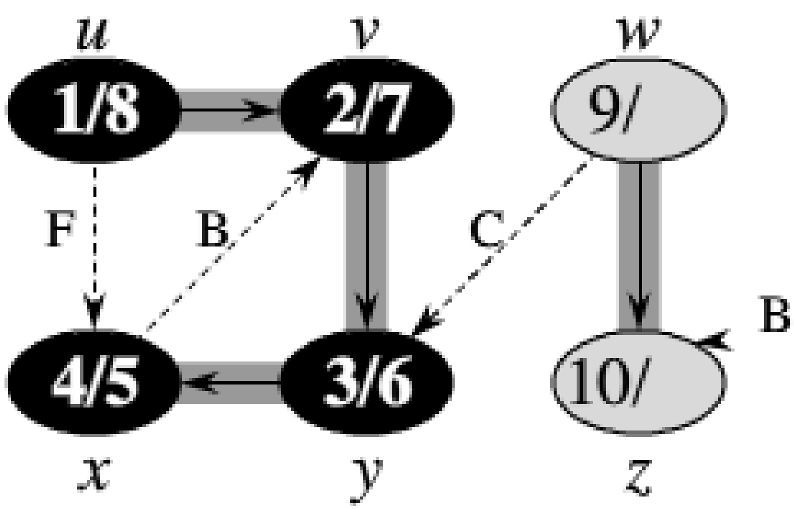

(m)

# DFS Example:
# Back Edge

- When first exploring an edge (u, v)
- If color V:
  - WHITE: tree edge
  - GRAY: back edge
  - BLACK: forward edge or cross edge
    - u.d < v.d: forward edge
    - u.d > v.d: cross edge



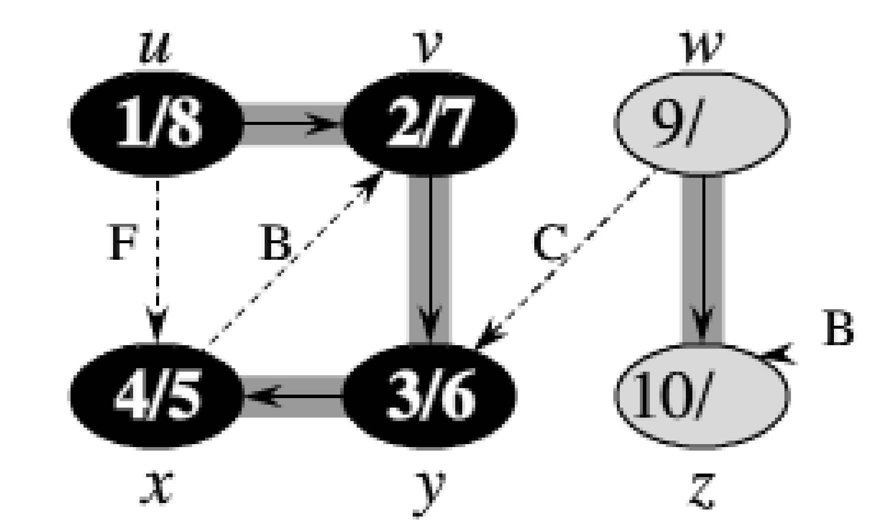
(m)



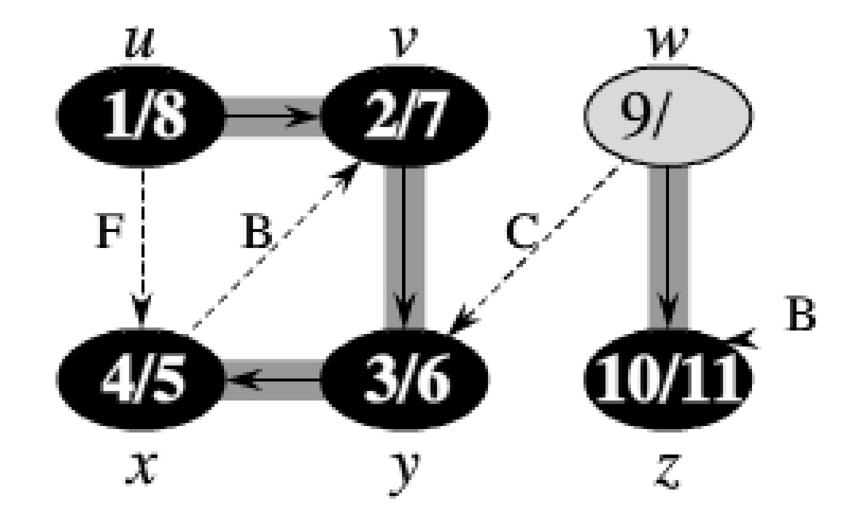
(n)

# DFS Example: z finishes

- When first exploring an edge (u, v)
- If color V:
  - WHITE: tree edge
  - GRAY: back edge
  - BLACK: forward edge or cross edge
    - u.d < v.d: forward edge
    - u.d > v.d: cross edge



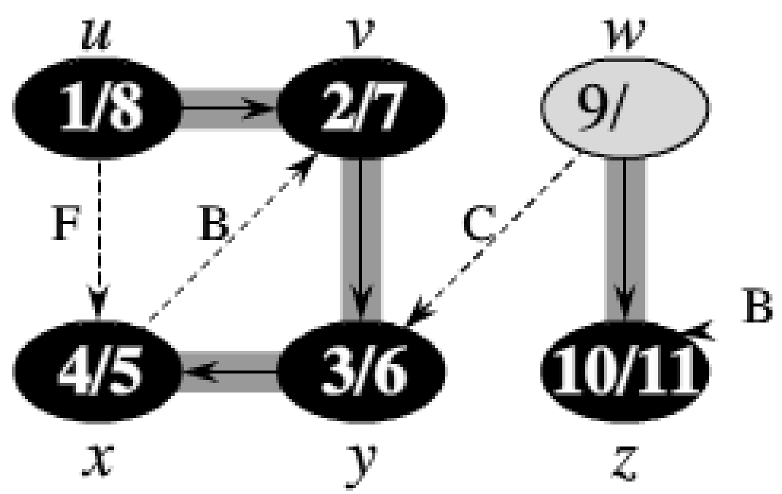
(n)



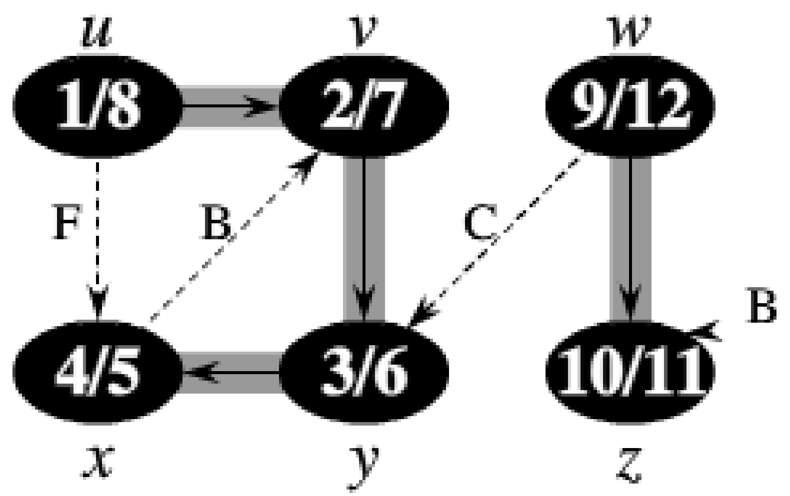
(o)

# DFS Example: z finishes

- When first exploring an edge (u, v)
- If color V:
  - WHITE: tree edge
  - GRAY: back edge
  - BLACK: forward edge or cross edge
    - u.d < v.d: forward edge
    - u.d > v.d: cross edge

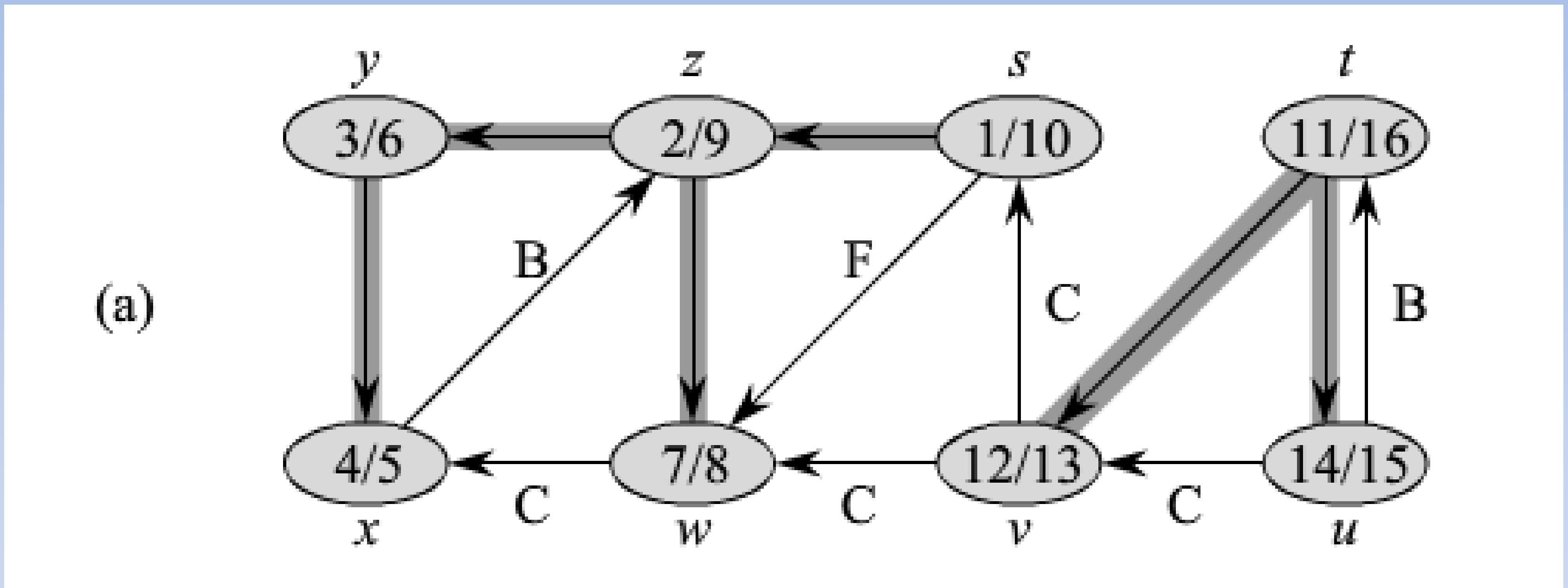


(o)

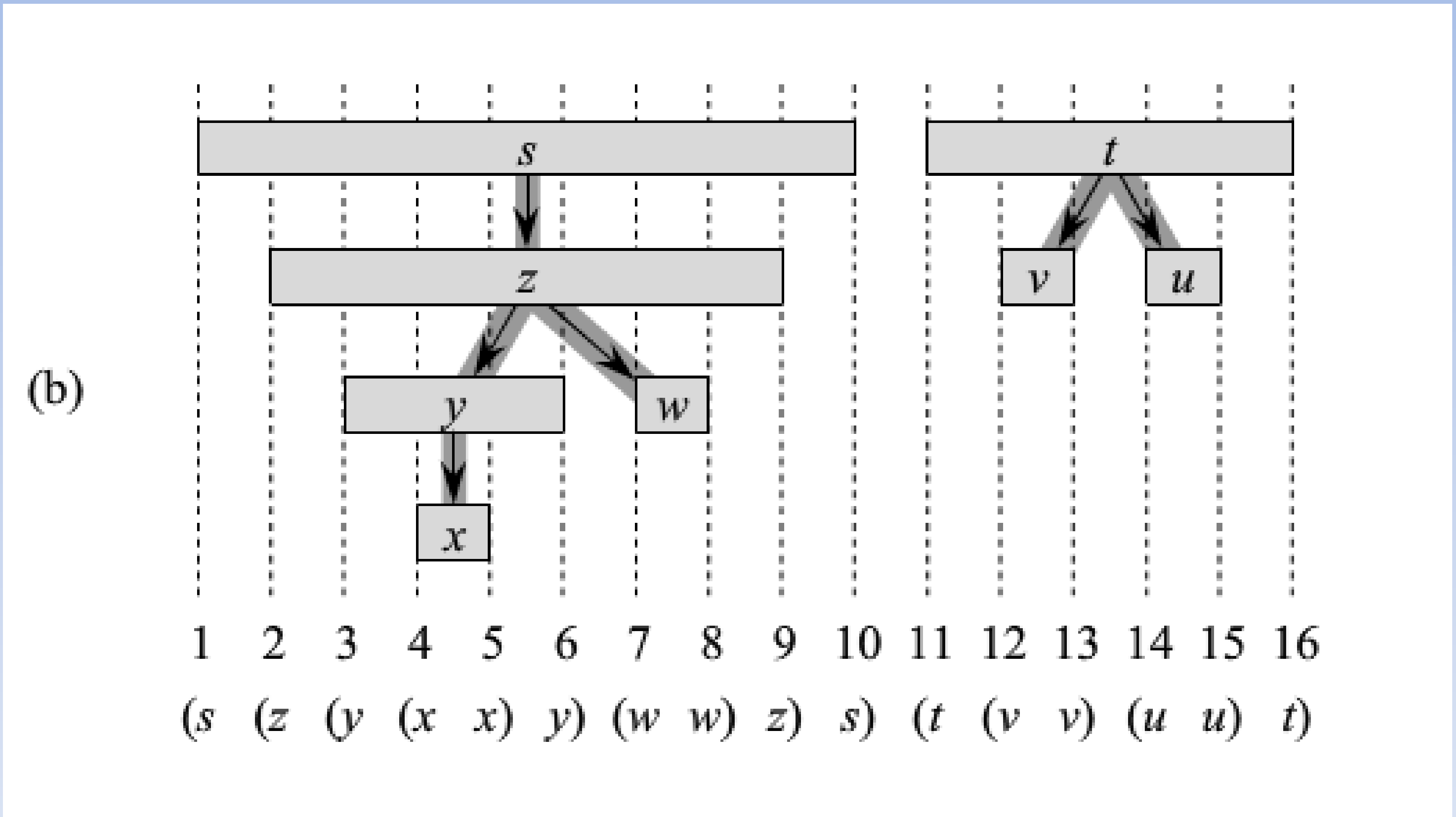


(p)

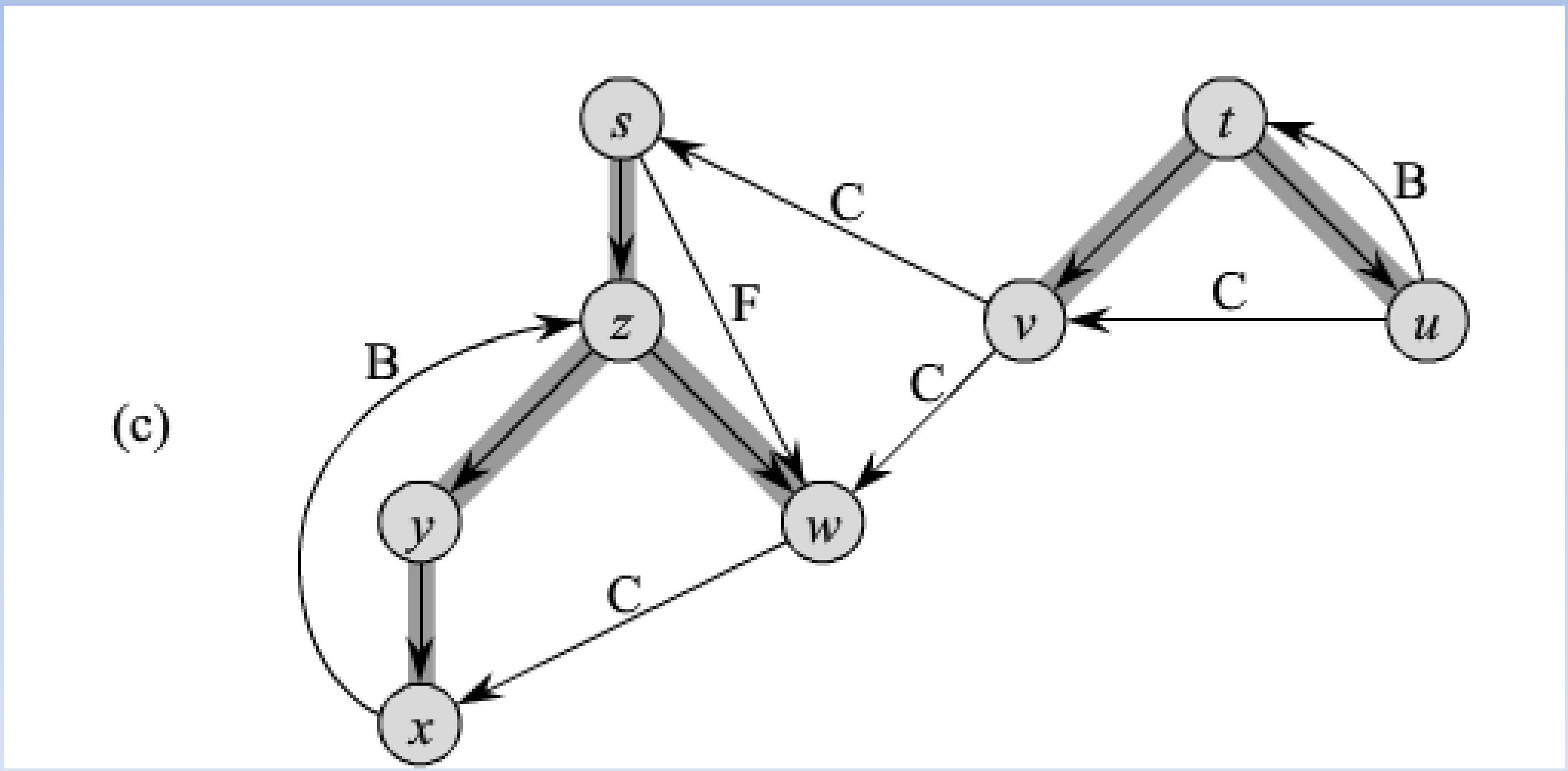
# Another DFS Example



# Discovery/Finish Times



# Redrawn Graph



# Topological Sort

- Given Directed Acyclic Graph (DAG) G=(V, E)
- Linearly order vertices $v \in V$ such that if G contains and edge (u, v) then u appears before v in ordering.

# Henry A. Bumstead

## Henry A. Bumstead

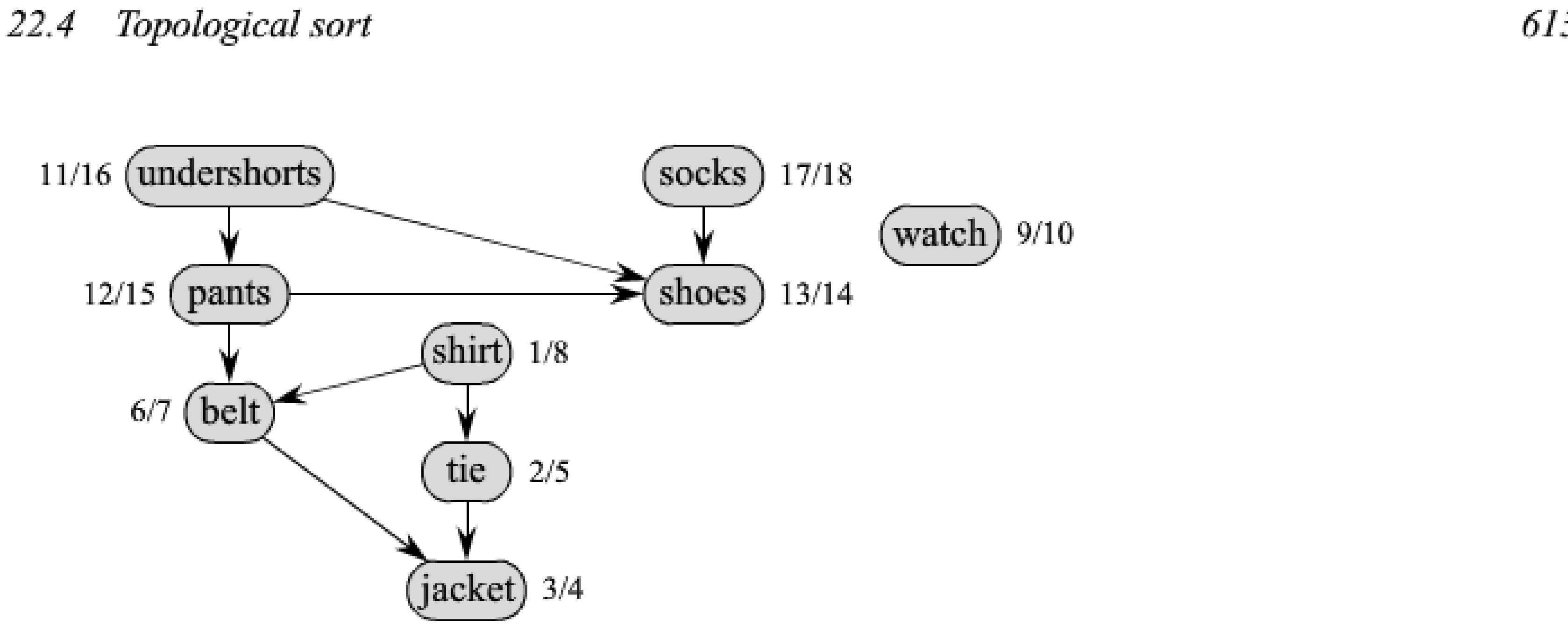
From Wikipedia, the free encyclopedia

**Henry Andrews Bumstead** (1870–1920) was an American physicist who taught at Yale from 1897 to 1920.[1] In 1918 he was scientific attache to the U.S. embassy in London. In 1920 he was Chairman of the National Research Council.

**Contents** [hide]



- In World War I Bumstead was selected to serve as the head of the Scientific Section in London under Admiral William Sims, Commander of the American Forces countering the U-boat campaign in the North Atlantic:[3]
- In 1920 Bumstead was elected Chairman of the National Research Council.[4] He was a member of the Connecticut Academy of Arts and Sciences.



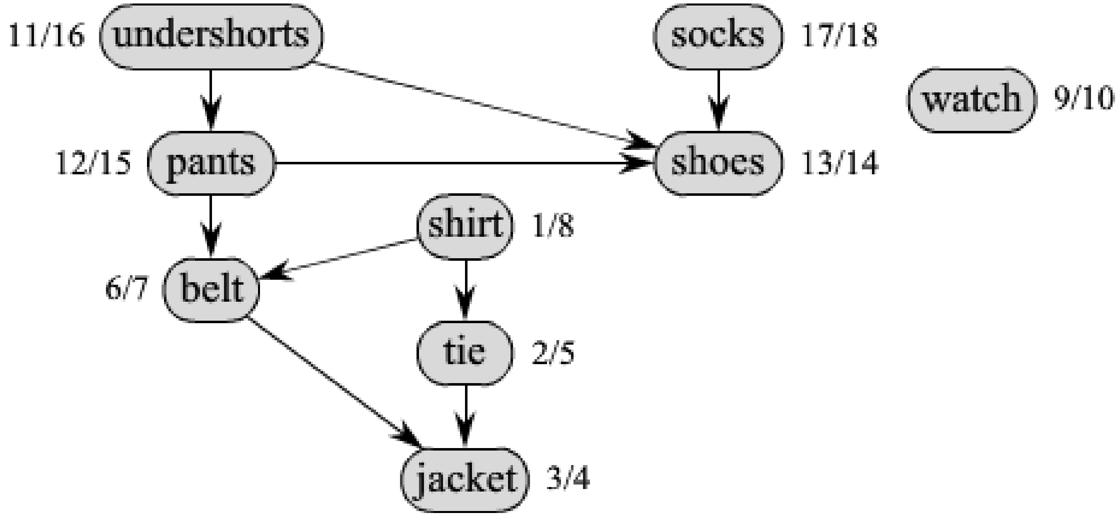

(a)

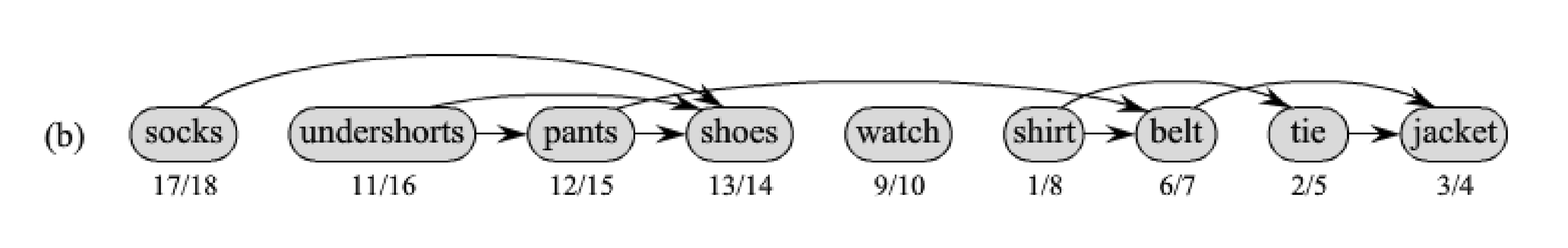
- Professor Bumstead topologically sorts his clothing when getting dressed.
- Each directed edge (u,v) means that garment u must be put on before garment v.
- The discovery and finishing times from a depth-first search are shown next to each vertex.

TOPOLOGICAL-SORT($G$)

call DFS($G$) to compute finishing times $v.f$ for each vertex $v$
as each vertex is finished, insert it onto the front of a linked list
**return** the linked list of vertices

(a)

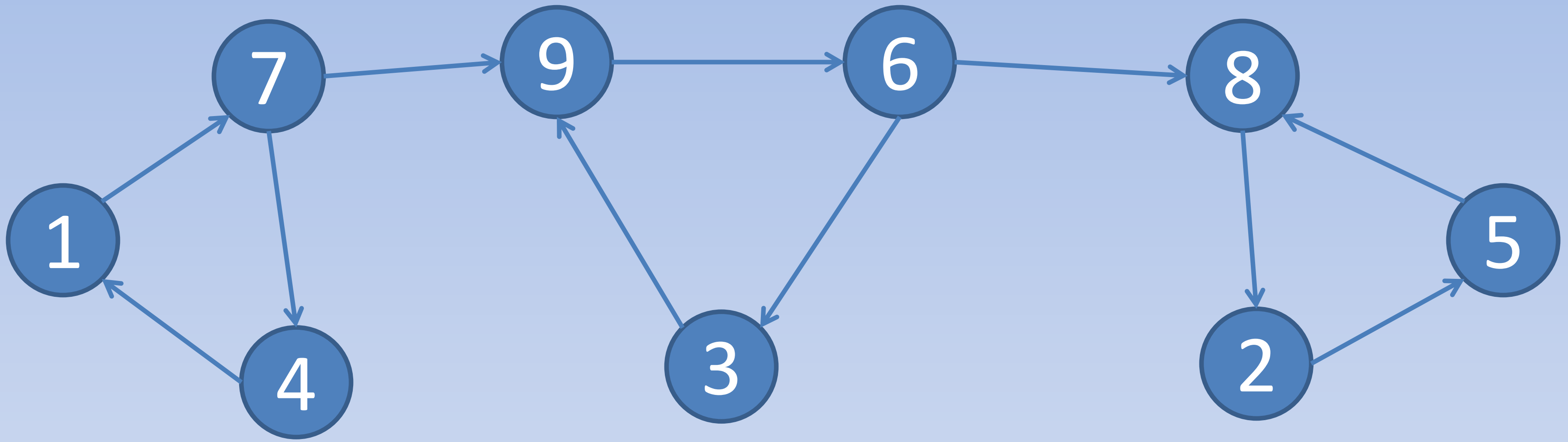

11/16
undershorts
socks
17/18
watch
9/10
12/15
pants
shoes
13/14
shirt
1/8
6/7
belt
tie
2/5
jacket
3/4


(b)
socks
17/18
undershorts
11/16
pants
12/15
shoes
13/14
watch
9/10
shirt
1/8
belt
6/7
tie
2/5
jacket
3/4

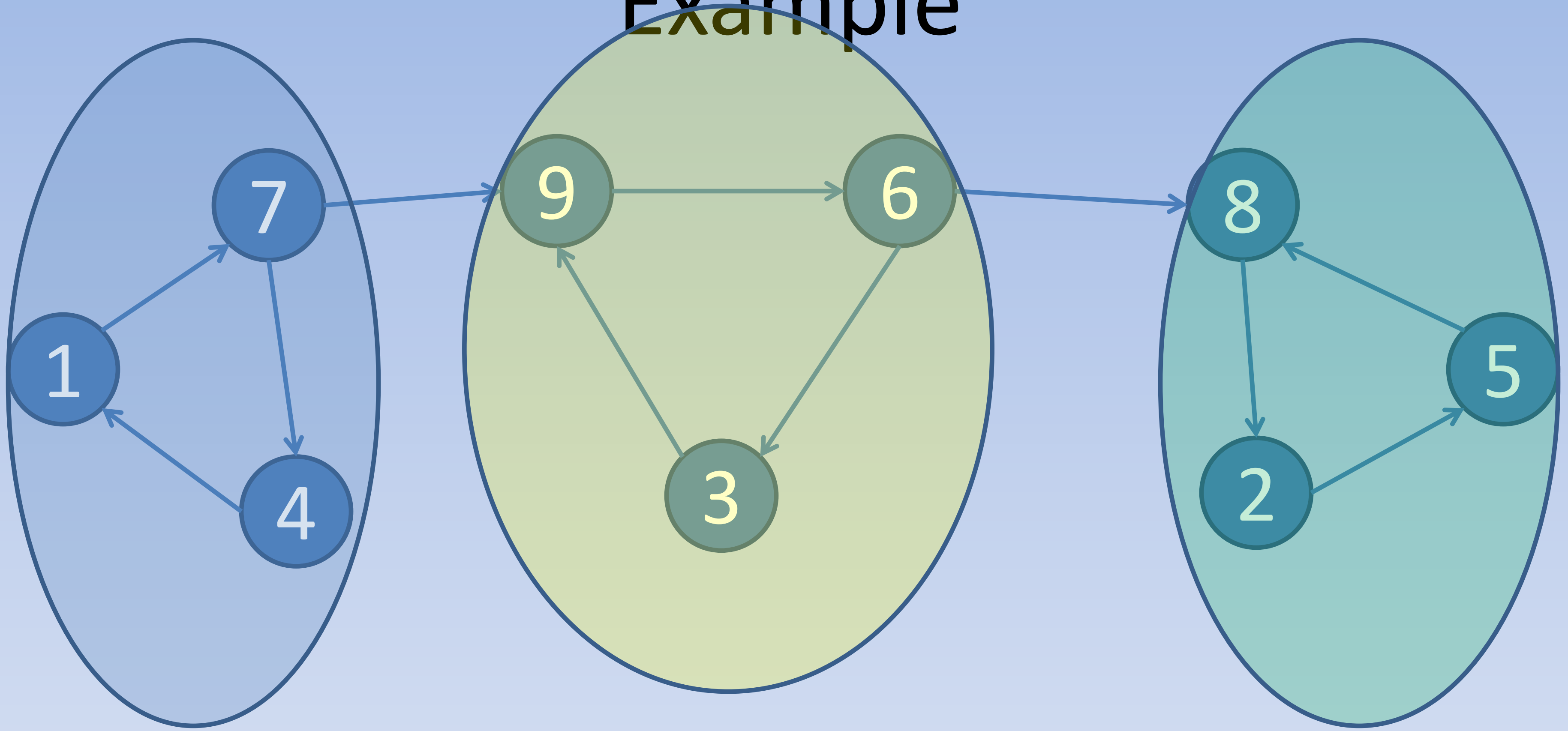
# Finding Strongly Connected Components

- Classic Use for DFS!
- We want to decompose a graph into its Strongly Connected Components:

# Example

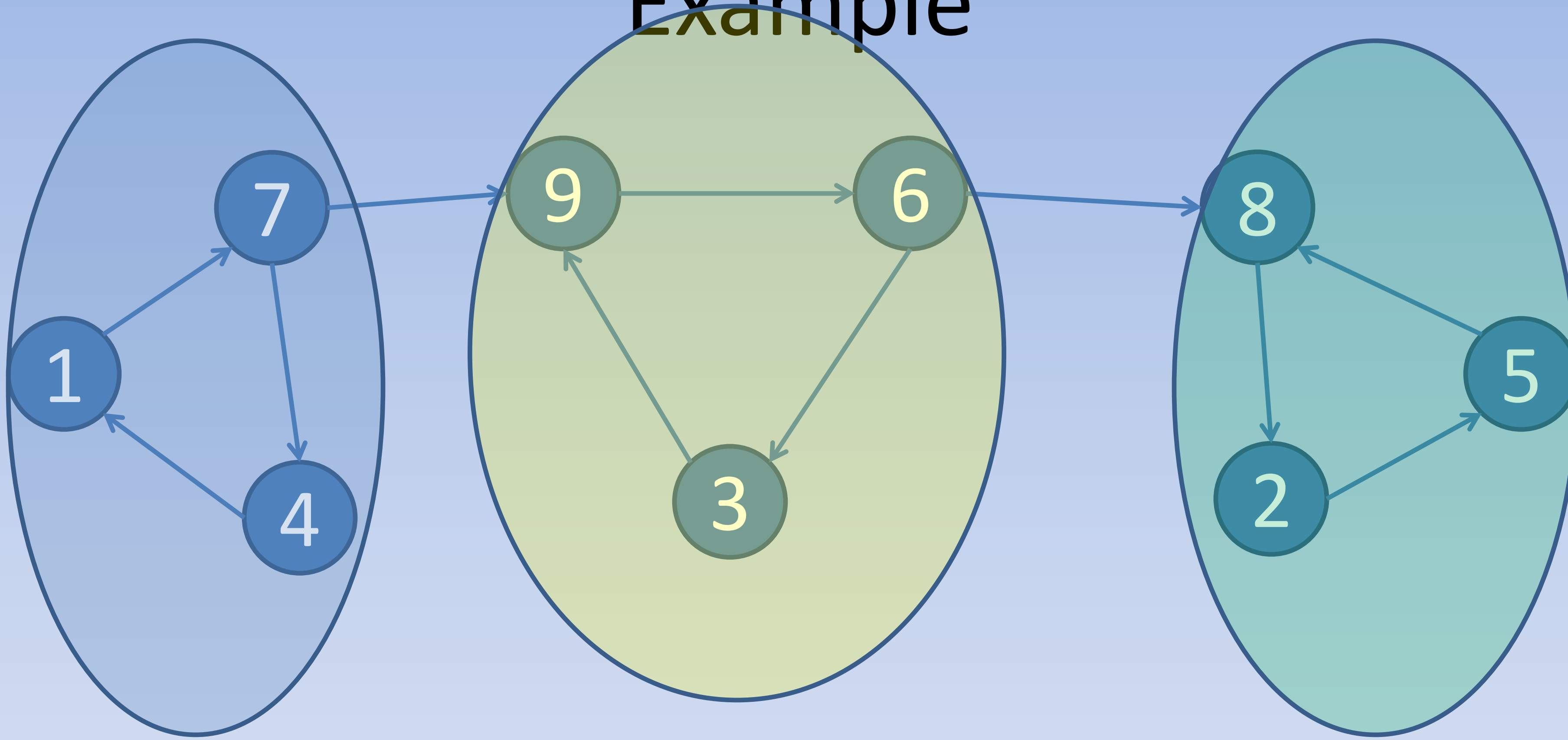


# Example




- Every Node is Reachable from Every Other Node w/ Strongly Connected Components

# Example

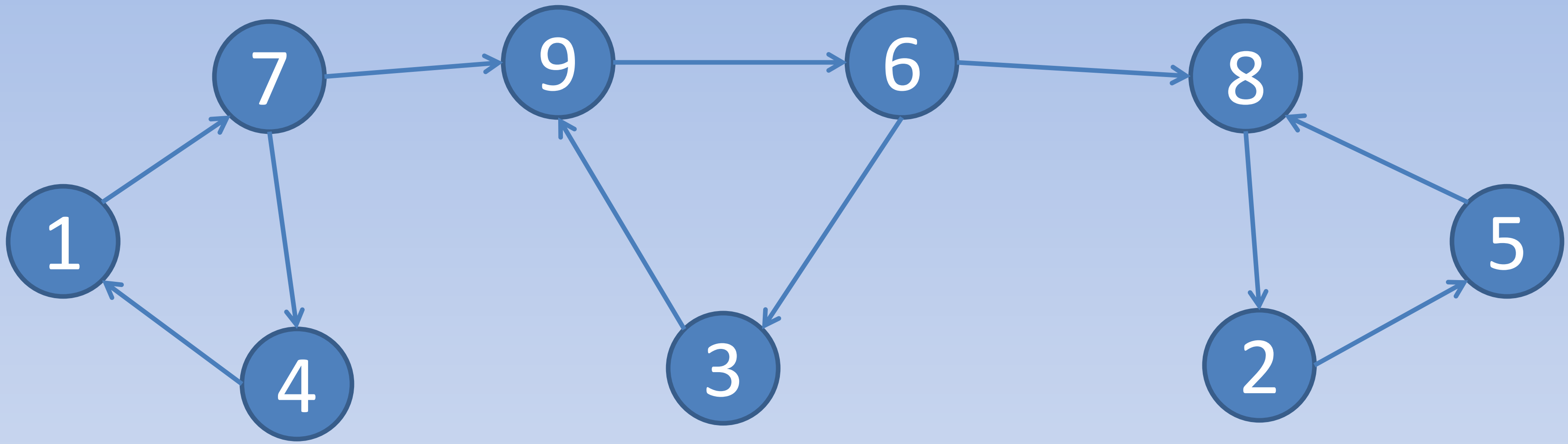



- The maximal set of vertices $C \subseteq V$ such that for every pair of vertices u and v in C, we have both u -->-> v and v -->-> u.

# $G^T$: The Transpose of a Graph

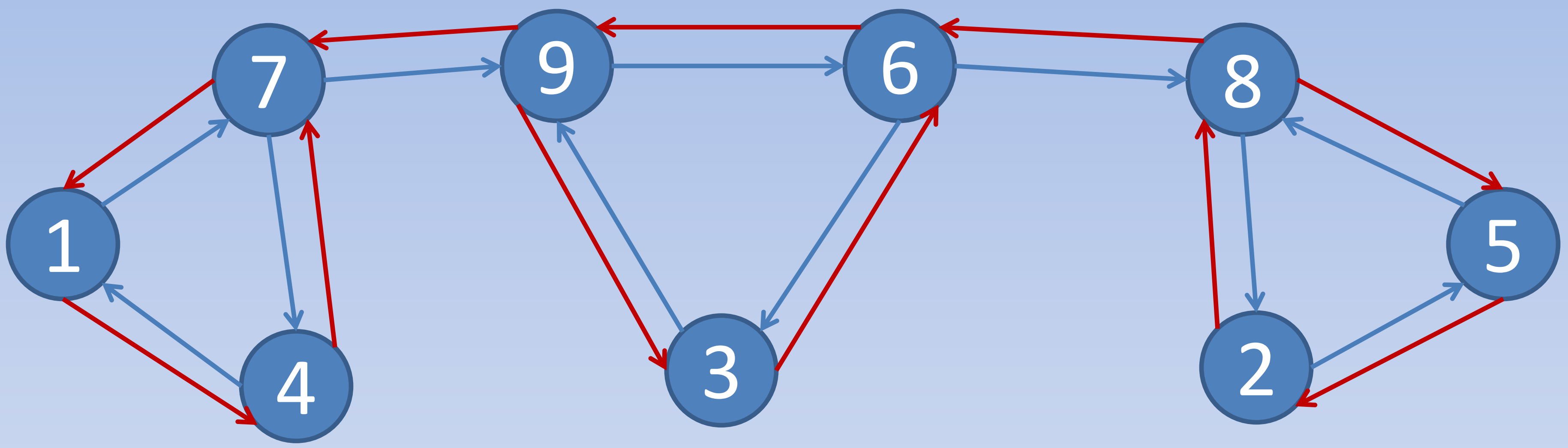
- Our algorithm for finding strongly connected components will utilize the Transpose of a graph.

# Example Graph G



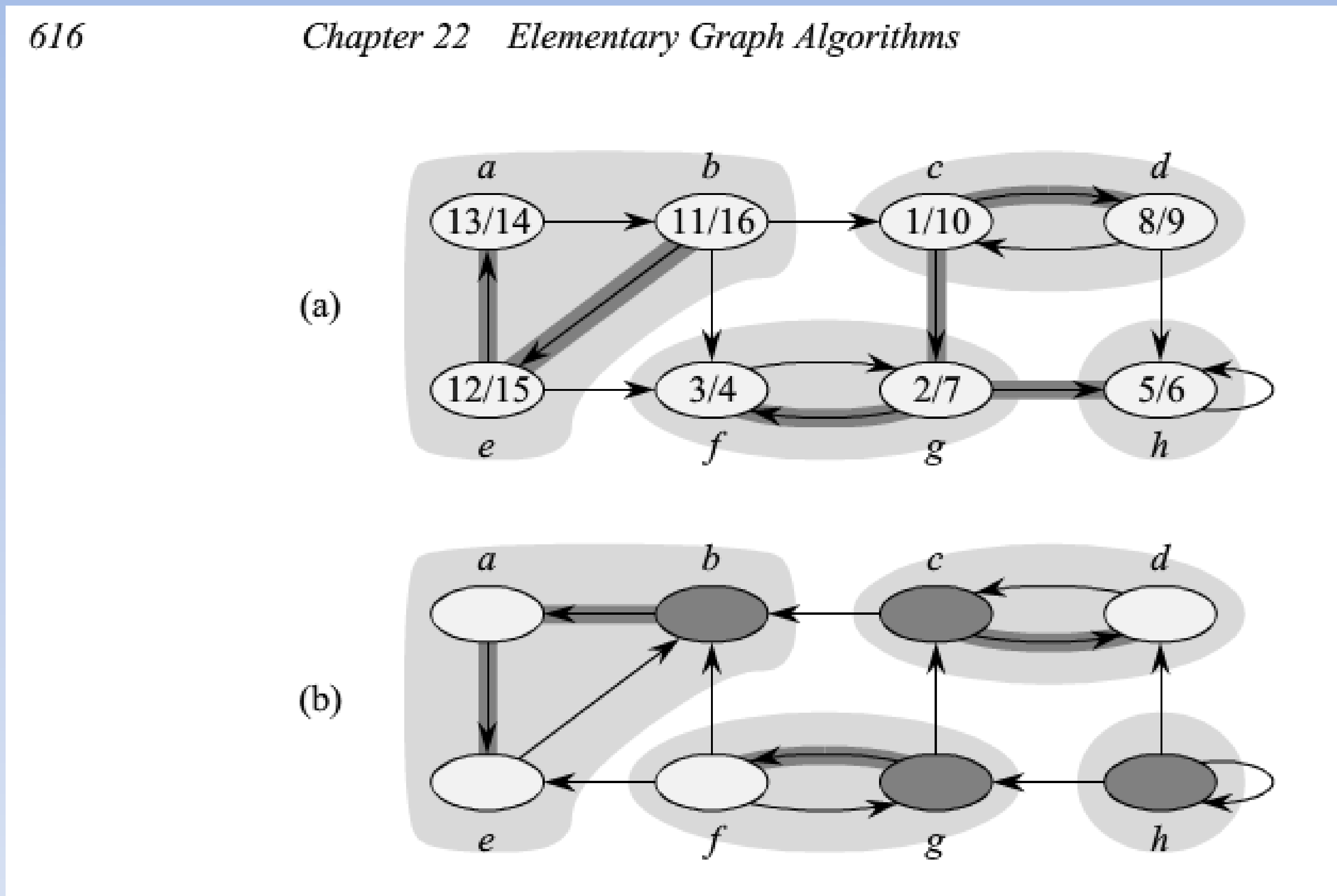

- $G^T = (V, ET)$
  - $E^T = \{u,v): (v,u) \in E\}$

# Example Graph G



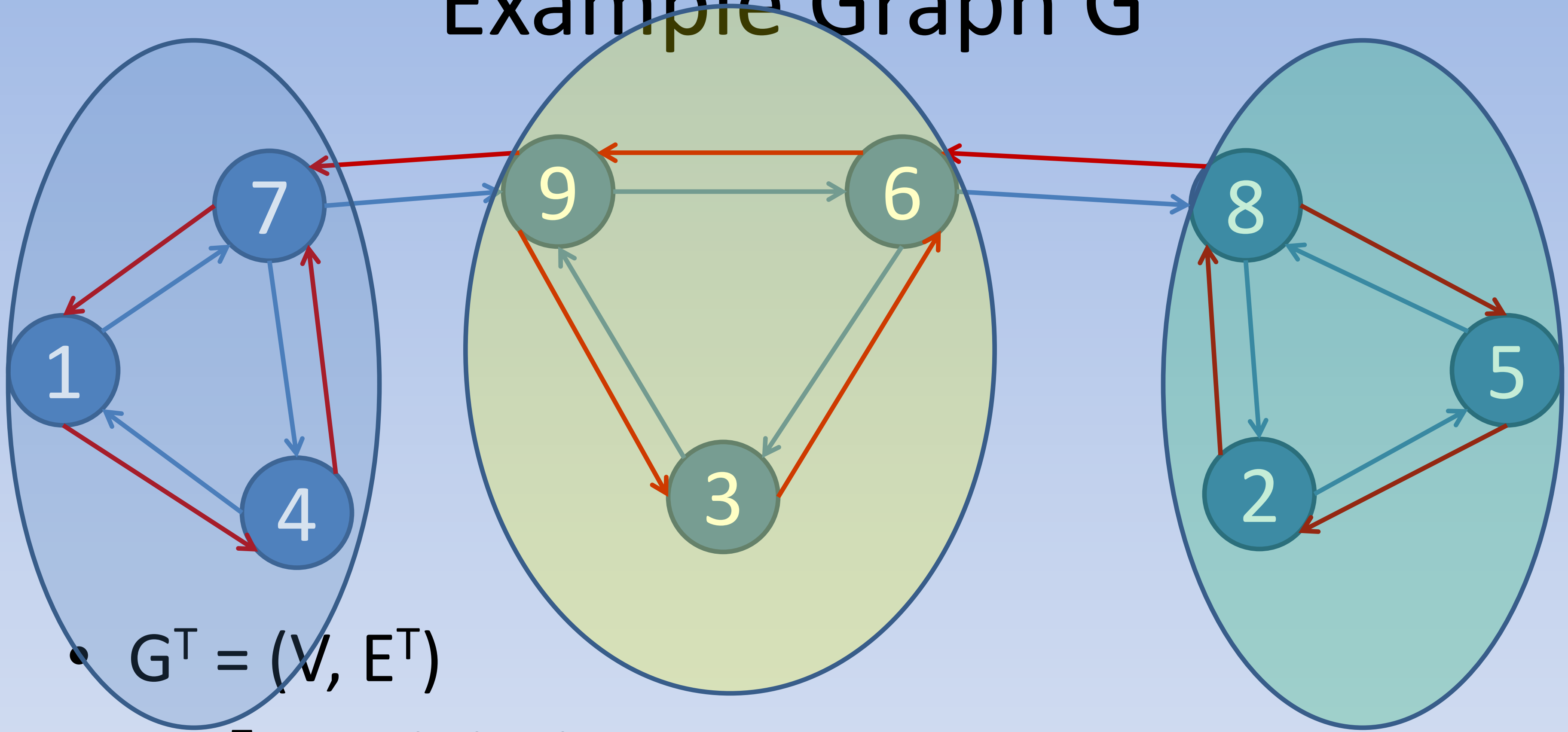

- $G^T = (V, E^T)$
  - $E^T = \{u,v\}: (v,u) \in E\}$
  - RED Edges!

# G and $G^t$




- G & $G^T$ have SAME connected components!

# Example Graph G




- $G^T = (V, E^T)$
  - $E^T = \{u,v): (v,u) \in E\}$
  - RED Edges!
- G & $G^T$ have SAME connected components!

# Strongly Connected Components

STRONGLY-CONNECTED-COMPONENTS($G$)

1 call DFS($G$) to compute finishing times $u.f$ for each vertex $u$
2 compute $G^{\mathrm{T}}$
3 call DFS($G^{\mathrm{T}}$), but in the main loop of DFS, consider the vertices
    in order of decreasing $u.f$ (as computed in line 1)
4 output the vertices of each tree in the depth-first forest formed in line 3 as a
    separate strongly connected component

# Strongly Connected Components

STRONGLY-CONNECTED-COMPONENTS($G$)
```
1  call DFS(G) to compute finishing times u.f for each vertex u
2  compute G^T
3  call DFS(G^T), but in the main loop of DFS, consider the vertices
       in order of decreasing u.f (as computed in line 1)
4  output the vertices of each tree in the depth-first forest formed in line 3 as a
       separate strongly connected component
```

1. RUN DFS to compute finishing times!

# Strongly Connected Components

STRONGLY-CONNECTED-COMPONENTS$(G)$

1 call DFS$(G)$ to compute finishing times $u.f$ for each vertex $u$
2 compute $G^T$
3 call DFS$(G^T)$, but in the main loop of DFS, consider the vertices
  in order of decreasing $u.f$ (as computed in line 1)
4 output the vertices of each tree in the depth-first forest formed in line 3 as a
  separate strongly connected component

1. RUN DFS to compute finishing times!
2. Compute $G^T$

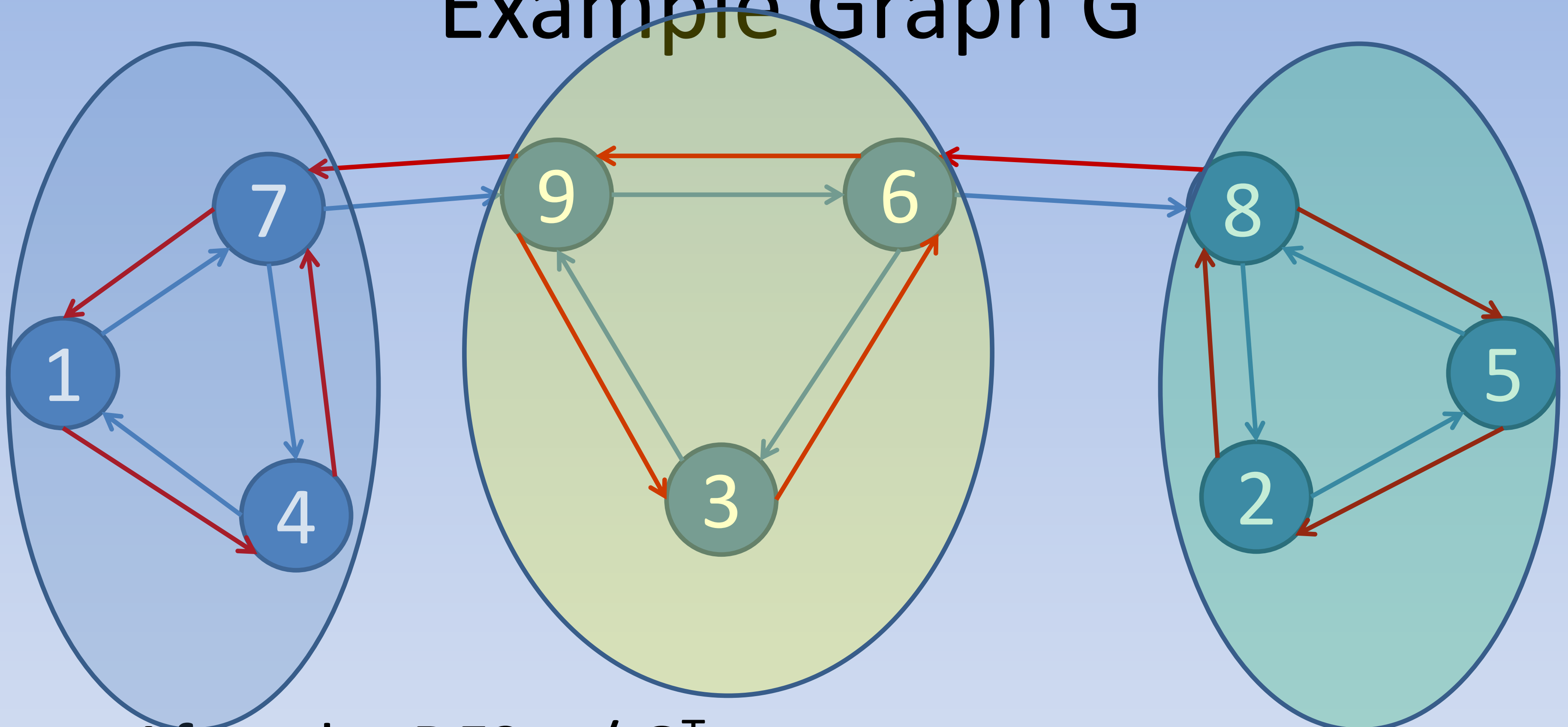
# Strongly Connected Components

STRONGLY-CONNECTED-COMPONENTS$(G)$

1 call DFS$(G)$ to compute finishing times $u.f$ for each vertex $u$
2 compute $G^T$
3 call DFS$(G^T)$, but in the main loop of DFS, consider the vertices
in order of decreasing $u.f$ (as computed in line 1)
4 output the vertices of each tree in the depth-first forest formed in line 3 as a
separate strongly connected component

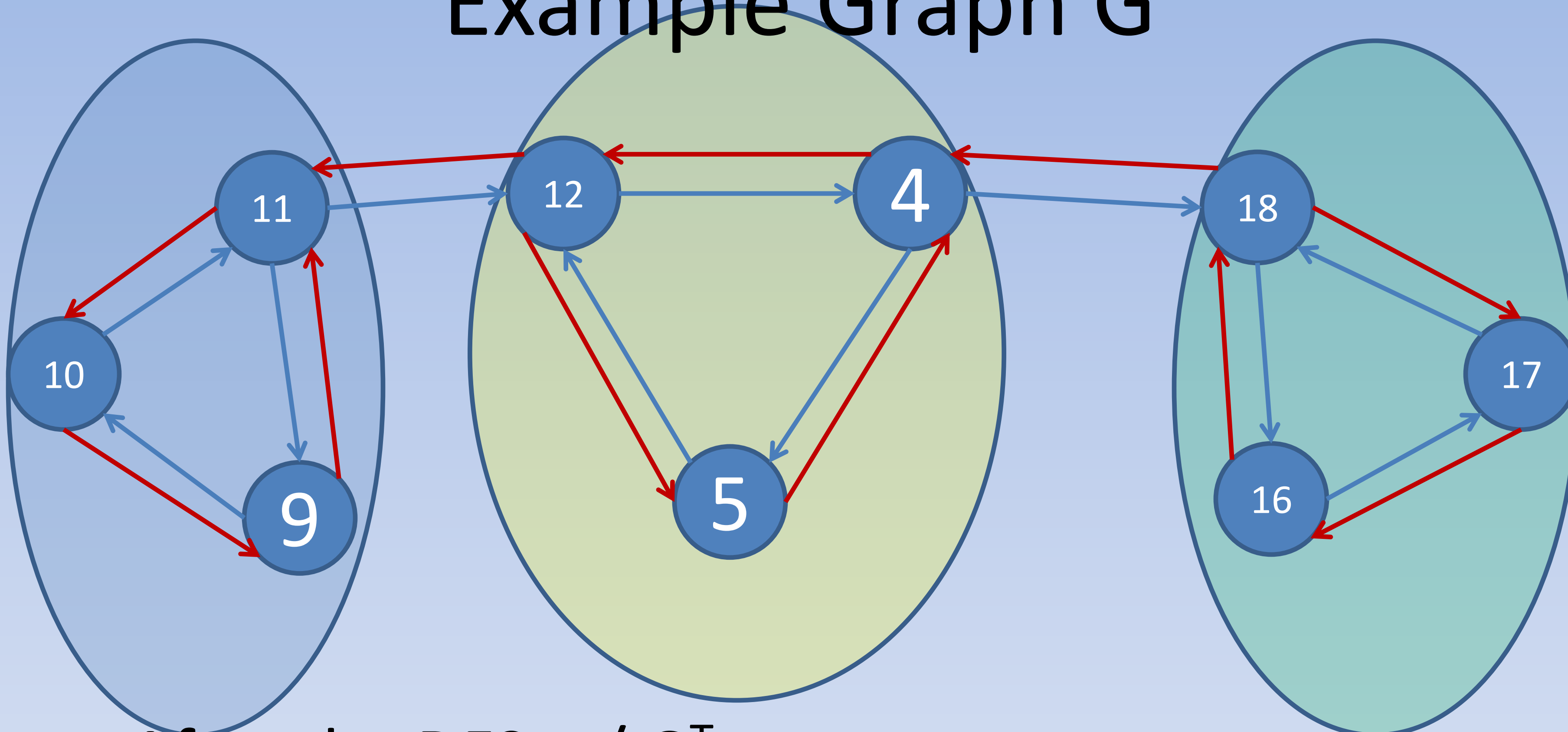
1. RUN DFS to compute finishing times!
2. Compute $G^T$
3. Run DFS on $G^T$ using finishing time from 1.
4. Out the depth-first tree forest.
   - These will be the strongly connected components.

# Example Graph G



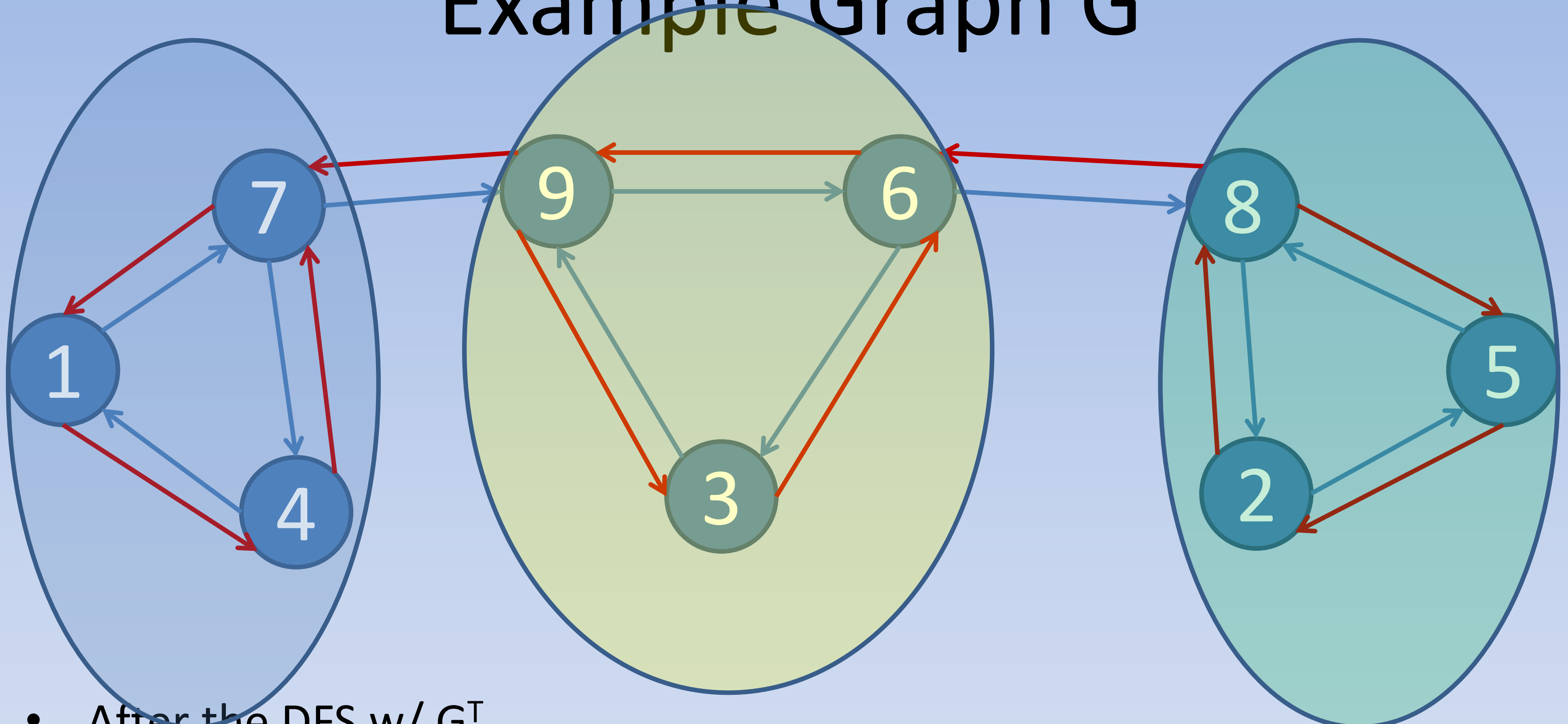

- After the DFS w/ $G^T$
  - Finish Times = [(4, 6), (5, 3), (9, 4), (10, 1), (11, 7), (12, 9), (16, 2), (17, 5), (18, 8)]

# Example Graph G



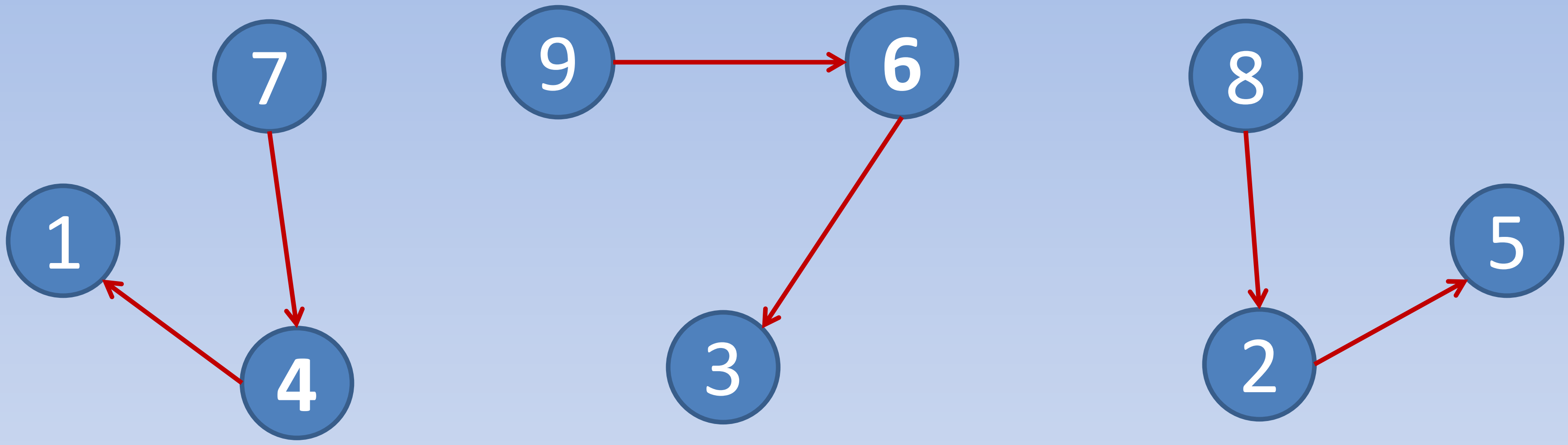

- After the DFS w/ $G^T$
  - Finish Times = [(4, 6), (5, 3), (9, 4), (10, 1), (11, 7), (12, 9), (16, 2), (17, 5), (18, 8)]

# Example Graph G



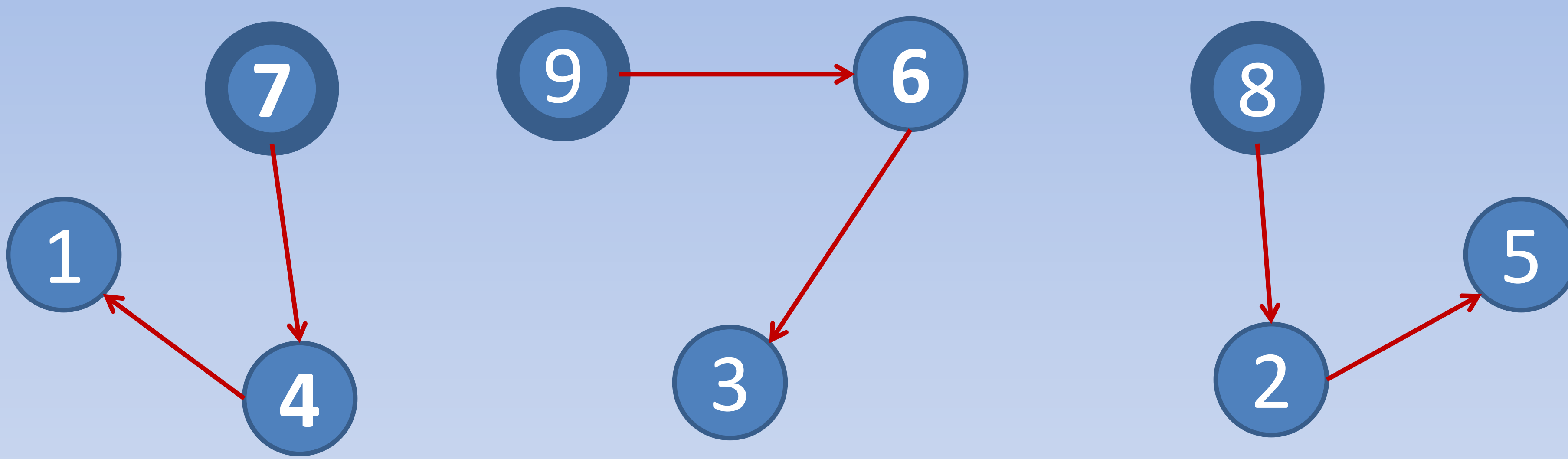

- After the DFS w/ $G^T$
  - Finish Times = [(4, 6), (5, 3), (9, 4), (10, 1), (11, 7), (12, 9), (16, 2), (17, 5), (18, 8)]
- Second DFS w/ Node Ordering
- Second DFS Vertex Ordering:
  - [8, 5, 2, 9, 7, 1, 4, 3, 6]

# Graph G w/ Final Tree Edges




- Final Tree Edges produced by Second DFS

# Graph G w/
# Final Tree Edges



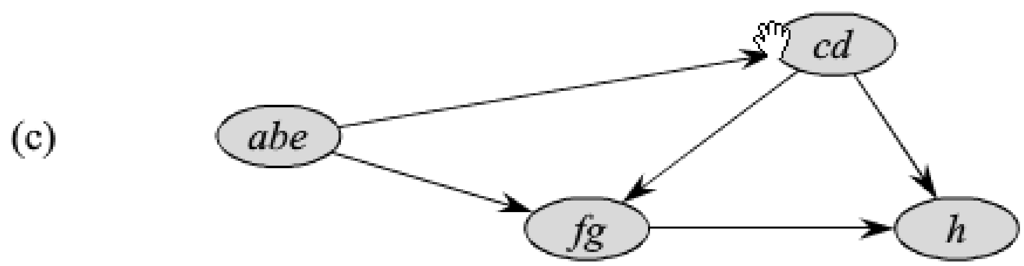

- Final Tree Edges produced by Second DFS

# Follow the Leader

- 3 nodes have 7 as leader
- 3 nodes have 8 as leader
- 3 nodes have 9 as leader

- Number of nodes with leader is Size of Clique

# Acyclic Component G ($G^{SCC}$)




- Key Idea: Component Graph
  - Constructed contracting edges within strongly connected components.