



# 图及其遍历

# Graphs and Graph Traversal

钮鑫涛

Nanjing University

2024 Fall

*The slides are mainly adapted from the original ones shared by Chaodong Zheng and Kevin Wayne. Thanks for their supports!*



# Graphs are Everywhere!

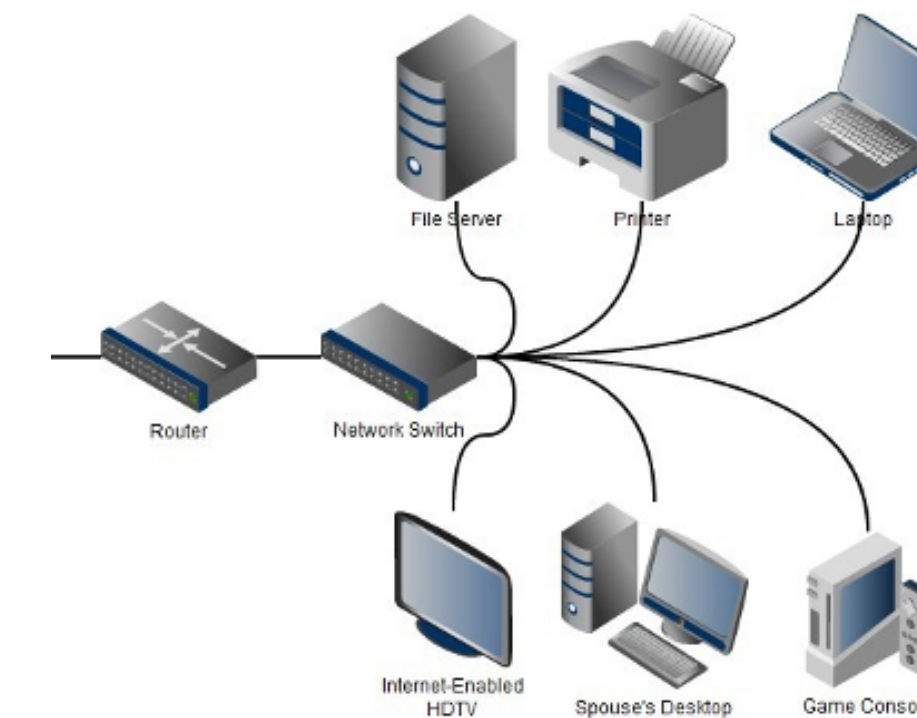
- **Transportation Networks.**

- ▶ **Nodes:** Airports; **Edges:** Nonstop flights.



- **Communication Networks.**

- ▶ **Nodes:** Computers; **Edges:** Physical links.



- **Social Networks.**

- ▶ **Nodes:** People; **Edges:** Friendship.



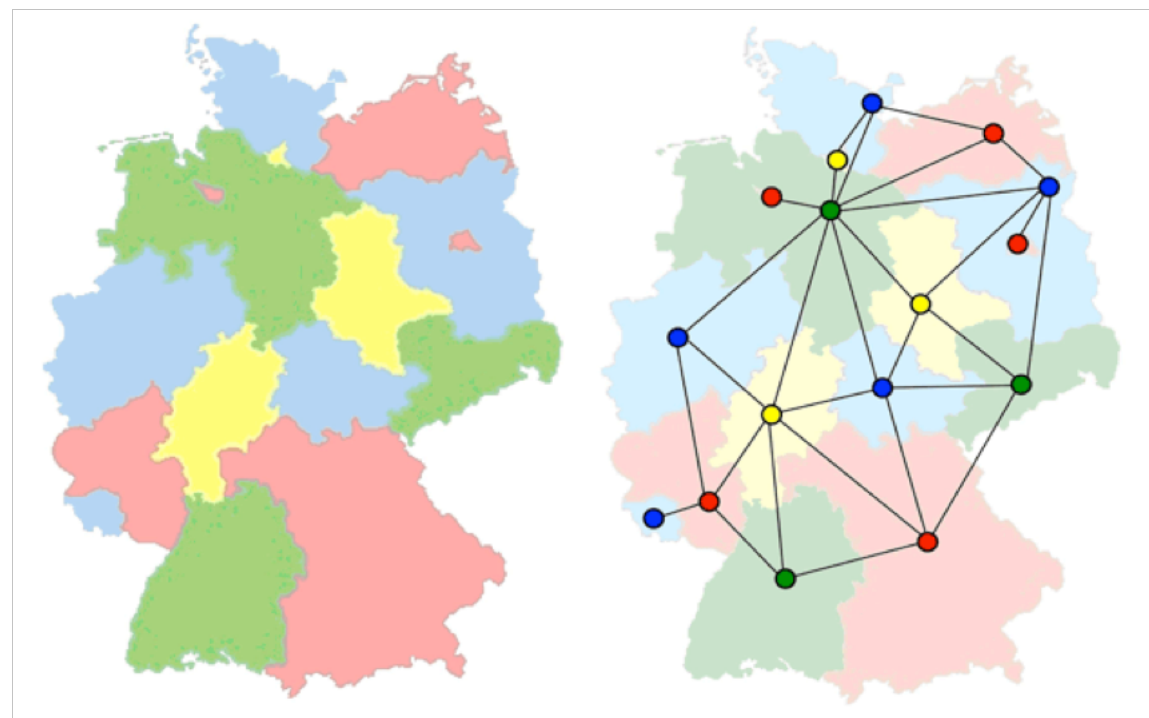
- ...



# Followed by many graph problems

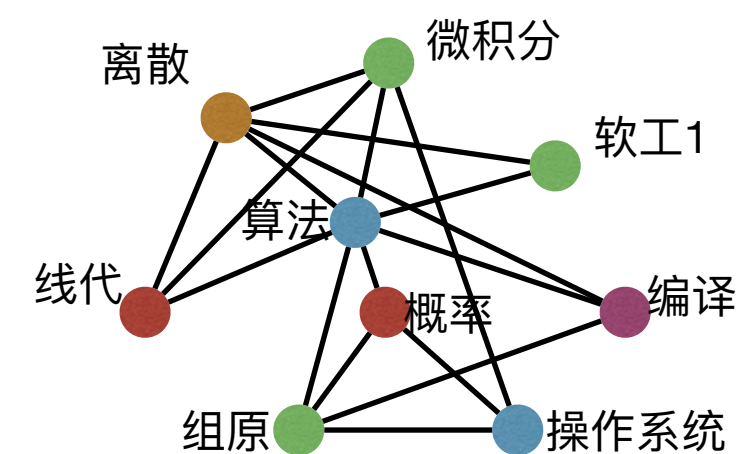
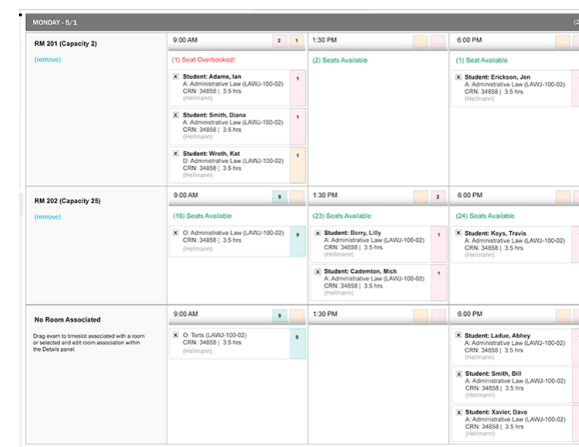
## Coloring Maps

- **Nodes:** Countries;  
**Edges:** Neighboring countries.
- Question of Interest: Chromatic number?



## Scheduling Exams

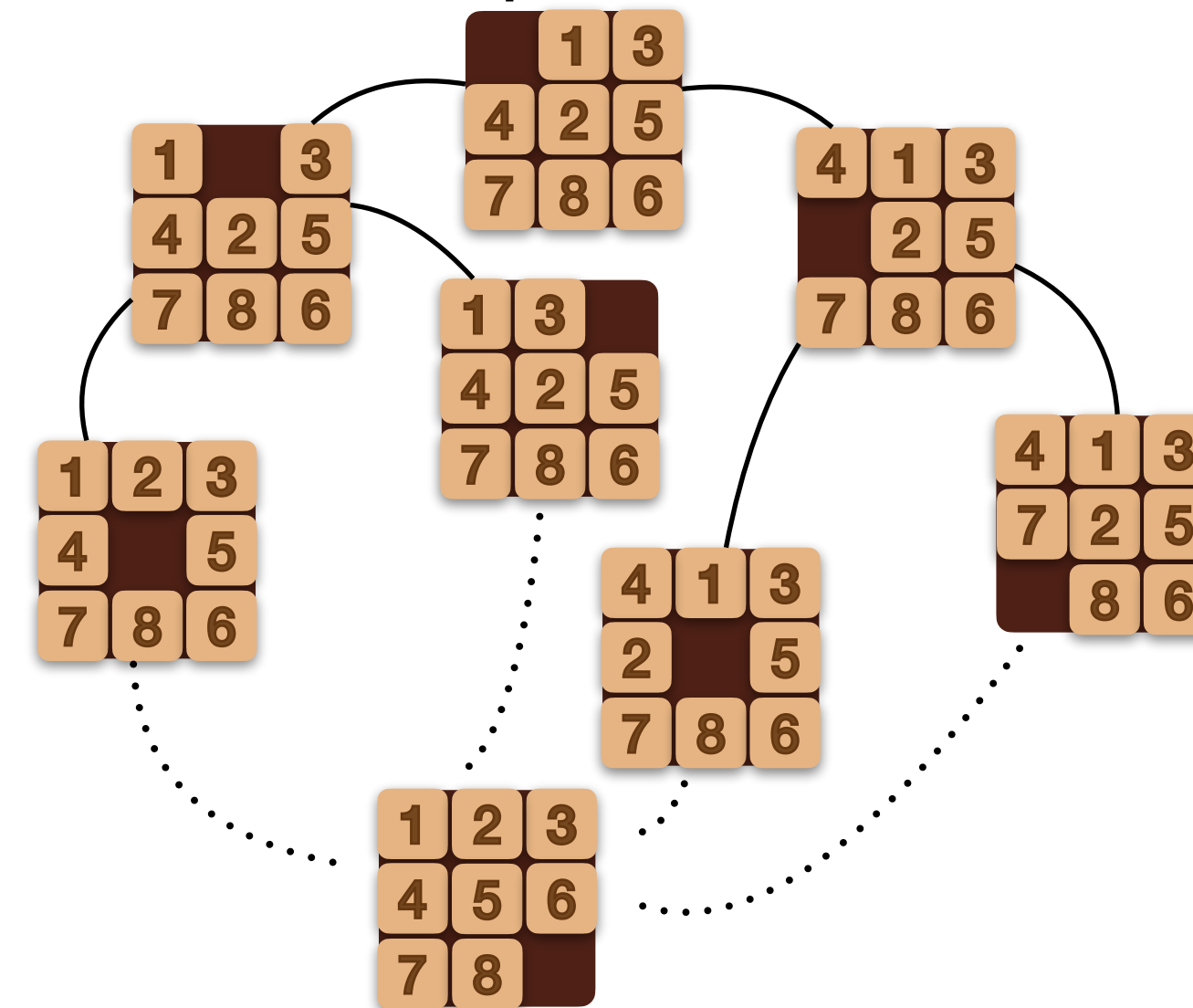
- **Nodes:** Exams;  
**Edges:** Conflicts.
- Question of Interest: Chromatic number?



both taken by at least one student ———

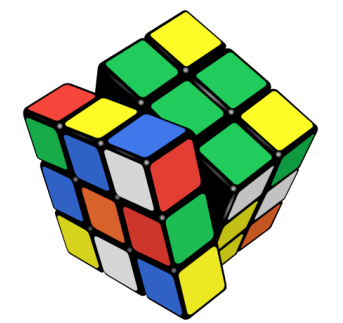
## Solving Sliding Puzzle

- **Nodes:** States;  
**Edges:** Legit moves
- Question of Interest: Shortest path?

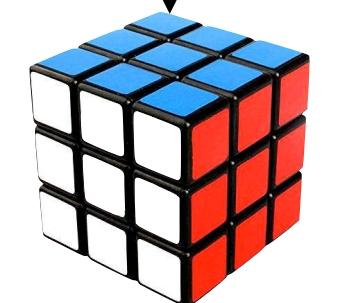


## Solving Rubik's Cube

- **Nodes:** States;  
**Edges:** Legit moves
- Question of Interest: God's Number?



minimal number of turns?





# Representing graphs in computers

- Adjacency Matrix (邻接矩阵)

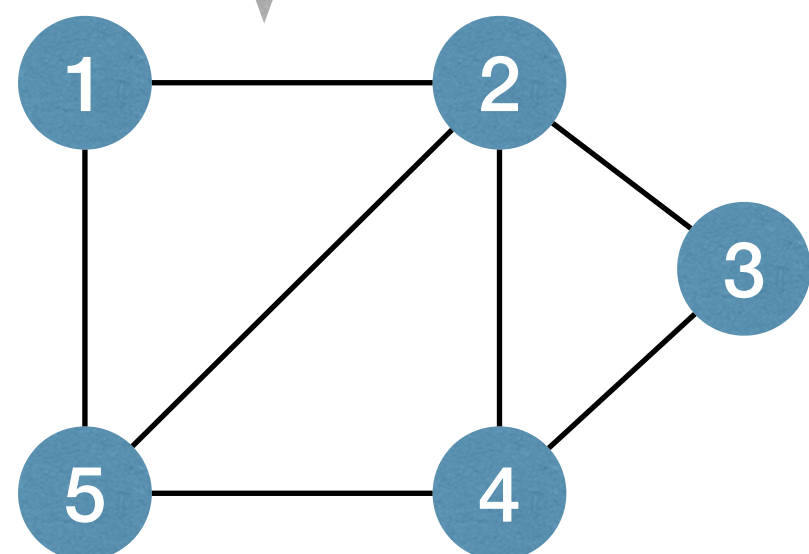
- Consider a graph  $G = (V, E)$ , where  $|V| = n$  and  $|E| = m$
- The Adjacency Matrix of  $G$  is an  $n \times n$  matrix  $A = (a_{ij})$  where

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

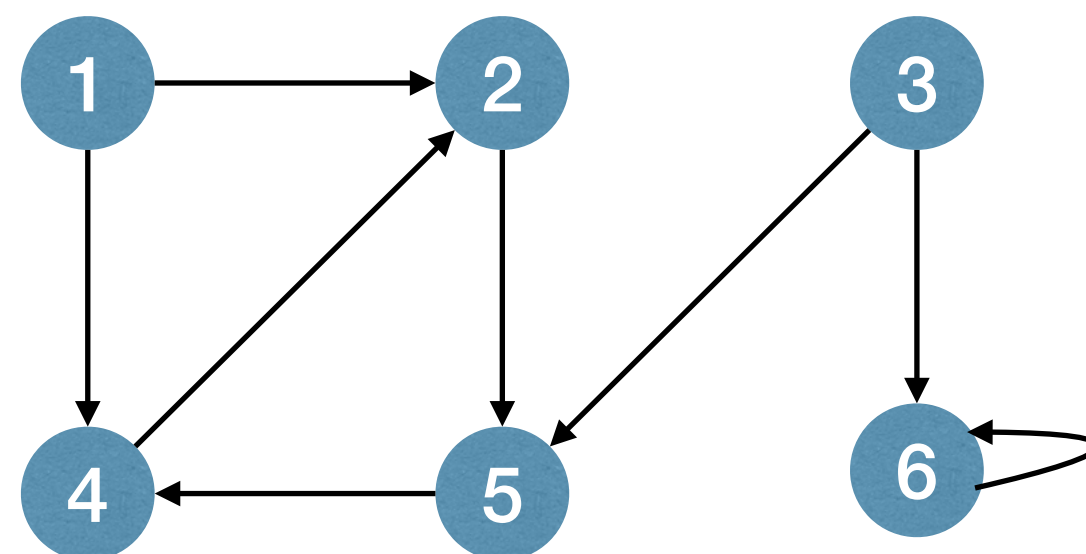
Quick Question: What does  $A^2$  mean, if anything?

- The matrix will be **symmetry** if  $G$  is undirected.

Simple Graph without self loop



	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



	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

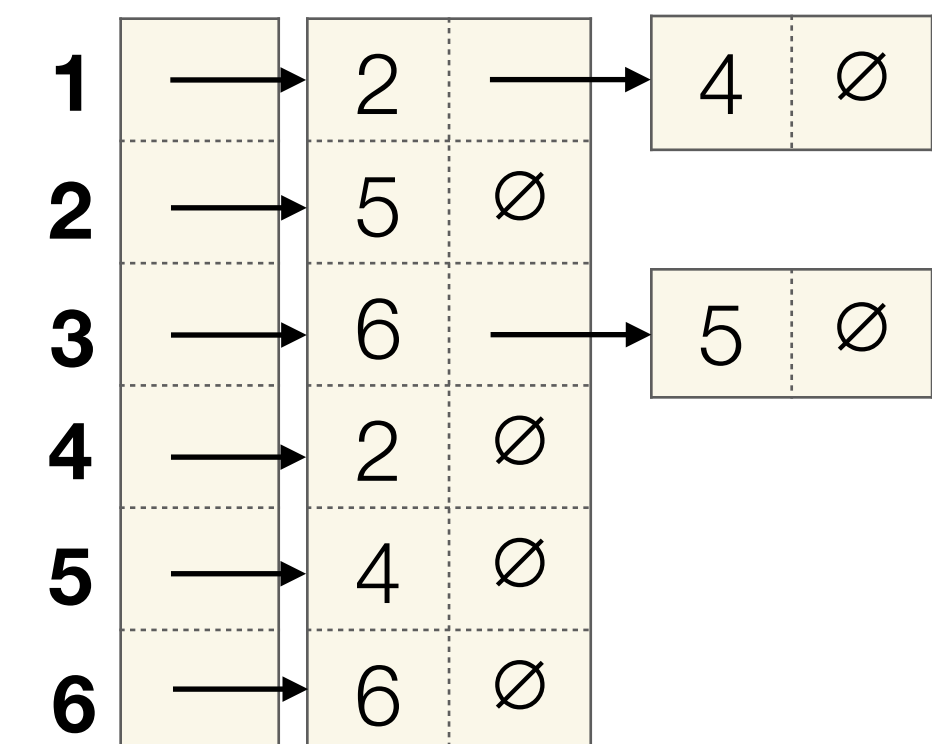
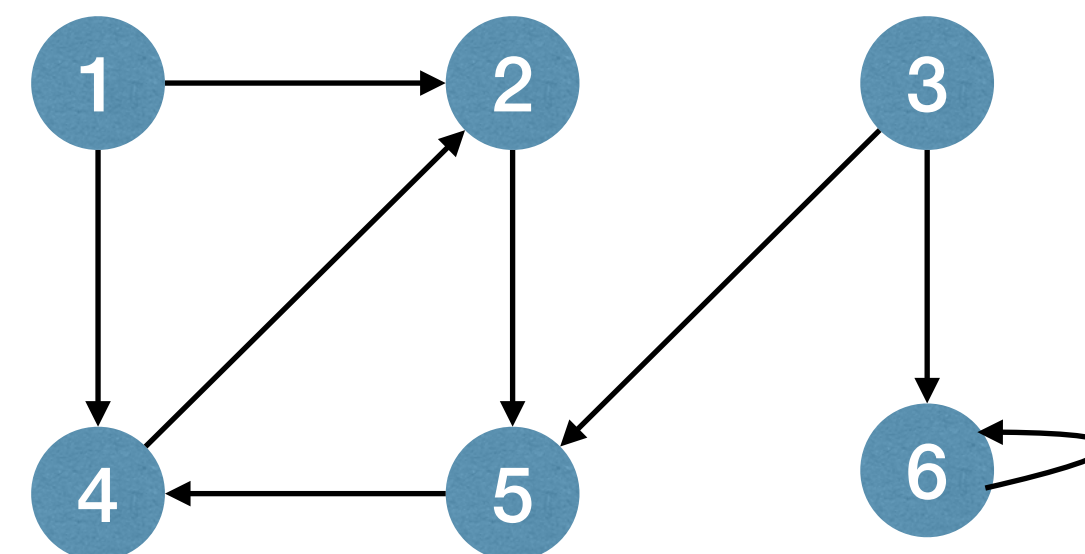
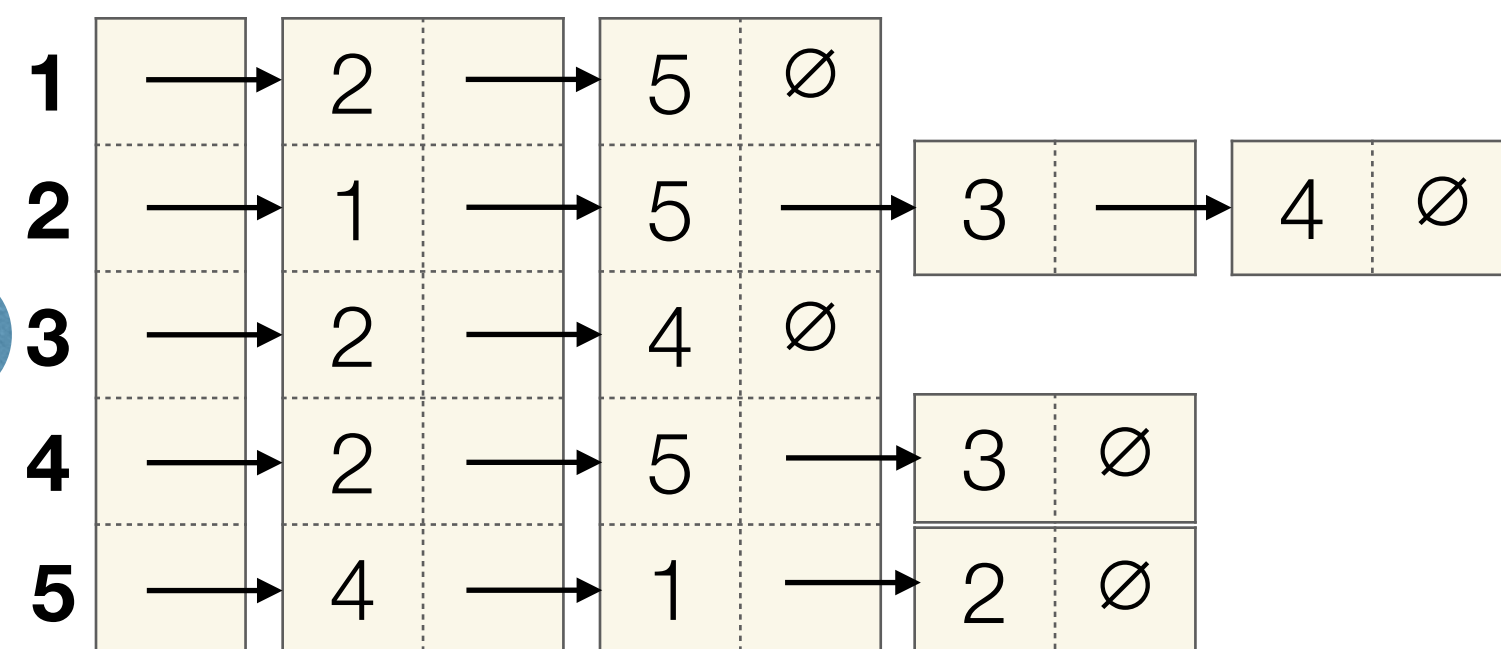
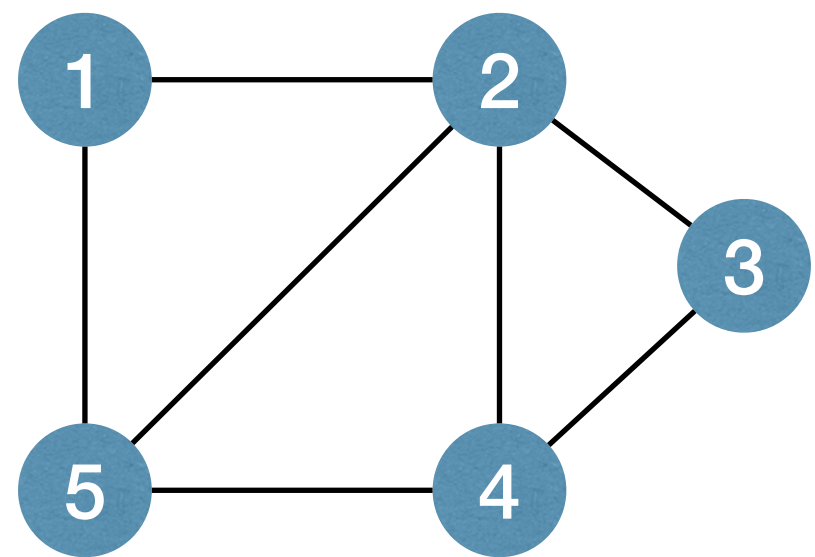
- The matrix will always cost  $\Theta(n^2)$  memory, regardless of  $m$ .



# Representing graphs in computers

- Adjacency List (邻接表)

- Consider a graph  $G = (V, E)$ , where  $|V| = n$  and  $|E| = m$
- The Adjacency List of  $G$  is a collection of  $n$  lists:
  - One for each vertex  $u \in V$
  - In the list for  $u$ , vertex  $v$  exists iff edge  $(u, v) \in E$
- Each edge appears twice if  $G$  is undirected.
- The space cost is  $\Theta(n + m)$





# Adjacency Matrix and Adjacency List

## Adjacency Matrix

- **Fast Query:** Are  $u$  and  $v$  neighbors?
- **Slow Query:** Find me any neighbor of  $u$ .
- **Slow Query:** Enumerate all neighbors of  $u$ .

Trade-offs

## Adjacency List

- **Fast Query:** Find me any neighbor of  $u$ .
- **Fast Query:** Enumerate all neighbors of  $u$ .
- **Slow Query:** Are  $u$  and  $v$  neighbors?

Queries: What types of queries are needed and/or frequent?  
Space usage: Is the graph dense or sparse?

Important question to ask



# Searching in a Graph (or, Graph Traversal)

- **Goal:** Start at source node  $s$  and find some node  $t$ .
- **Or:** Visit all nodes reachable from  $s$ .
- Two Basic Strategies:
  - ▶ **Breadth-First Search (BFS)**
  - ▶ **Depth-First Search (DFS)**
- Many applications, beside searching and traversal!
- Usually use adjacency list when discussing BFS/DFS. (At least in this course...)



# Breadth-First Search





# Breadth-First Search (BFS)

- Basic Idea of BFS:

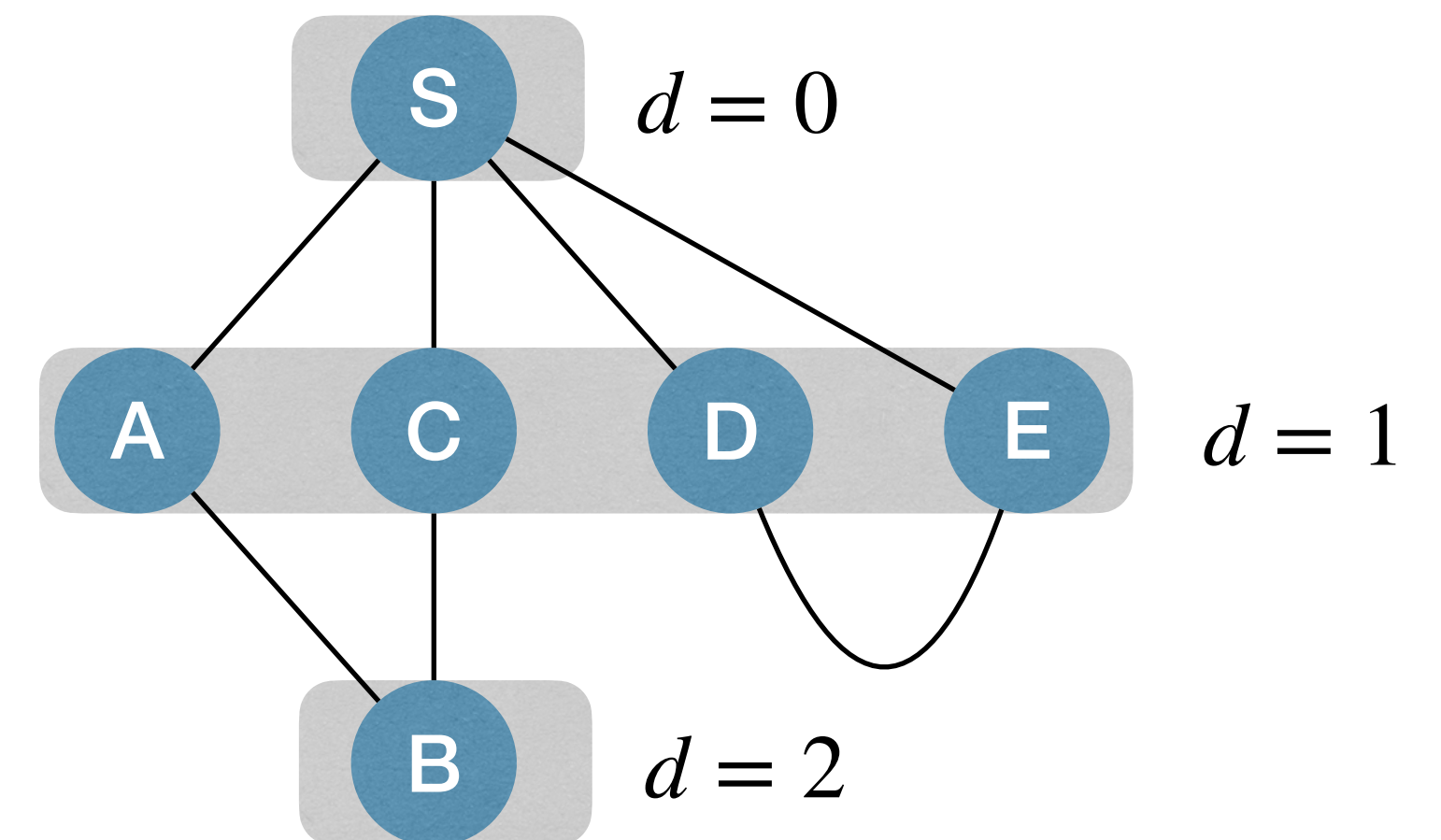
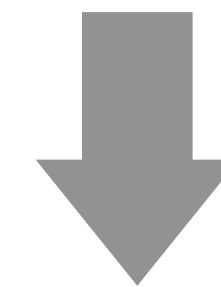
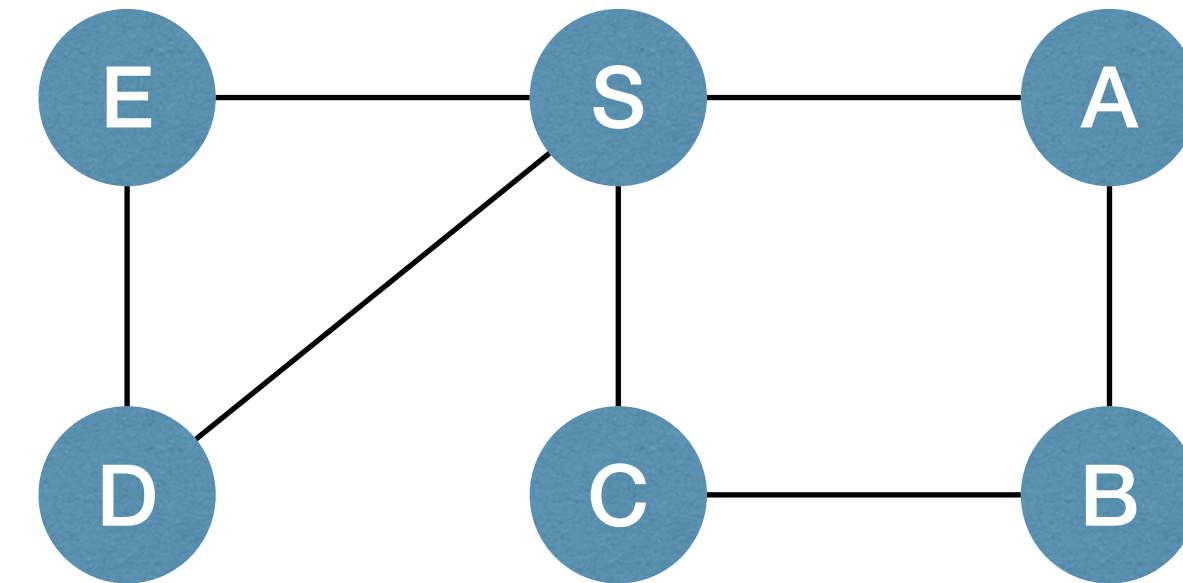
- ▶ Start at the source node  $s$ ;
- ▶ Visit other nodes (reachable from  $s$ ) “*layer by layer*”.

- More precise description:

- ▶ Start at the source node  $s$ ;
- ▶ Visit nodes at *distance* 1 from  $s$ ;
- ▶ Visit nodes at *distance* 2 from  $s$ ;
- ▶ ...

These nodes are neighbors of distance 1 nodes!

Visit all distance  $d$  nodes, before visit any distance  $d + 1$  node.





# BFS Implementation

- How to implement BFS? (Hint: recall traversal-by-layer in trees)
  - ▶ Use a FIFO Queue!
- Nodes have 3 status:
  - ▶ Undiscovered ( **WHITE** ): Not in queue yet.
  - ▶ Discovered but not visited ( **GRAY** ): In queue but not processed.
  - ▶ Visited ( **BLACK** ): Ejected from queue and processed.

**BFSSkeleton(G, s):**

**for each**  $u$  **in**  $V$

$u.dist := INF, u.discovered := False$

$s.dist := 0, s.discovered := True$

$Q.enqueue(s)$

**while**  $!Q.empty()$

$u := Q.dequeue()$

**for each** edge  $(u, v)$  **in**  $E$

**if**  $!v.discovered$

$v.dist := u.dist + 1$

$v.discovered := True$

$Q.enqueue(v)$



# BFS Implementation

BFS(G, s):

**for** each  $u$  in  $V$

$u.c := WHITE, u.d := INF, u.p := NIL$

$s.c := GRAY, s.d := 0, s.p := NIL$

$Q.enqueue(s)$

**while**  $!Q.empty()$

$u := Q.dequeue()$

$u.c := BLACK$

**for** each edge  $(u, v)$  in  $E$

**if**  $v.c = WHITE$

$v.c := GRAY$

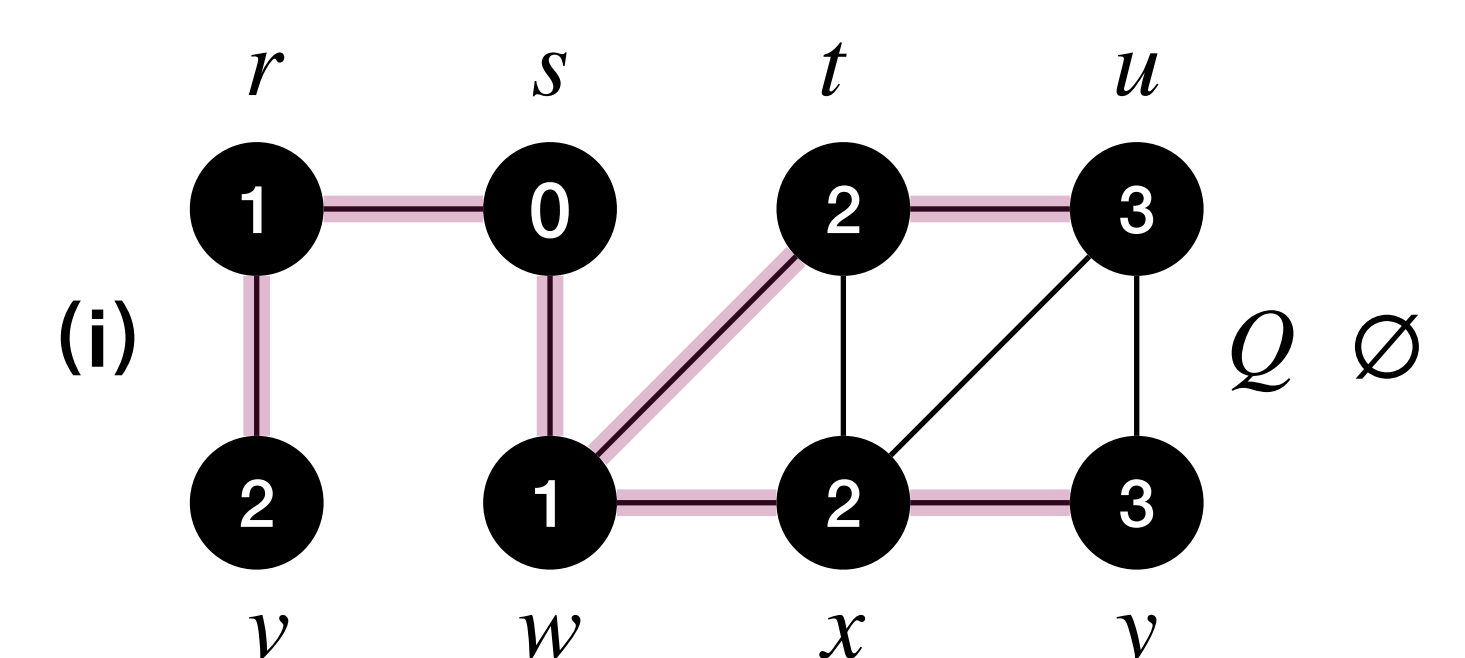
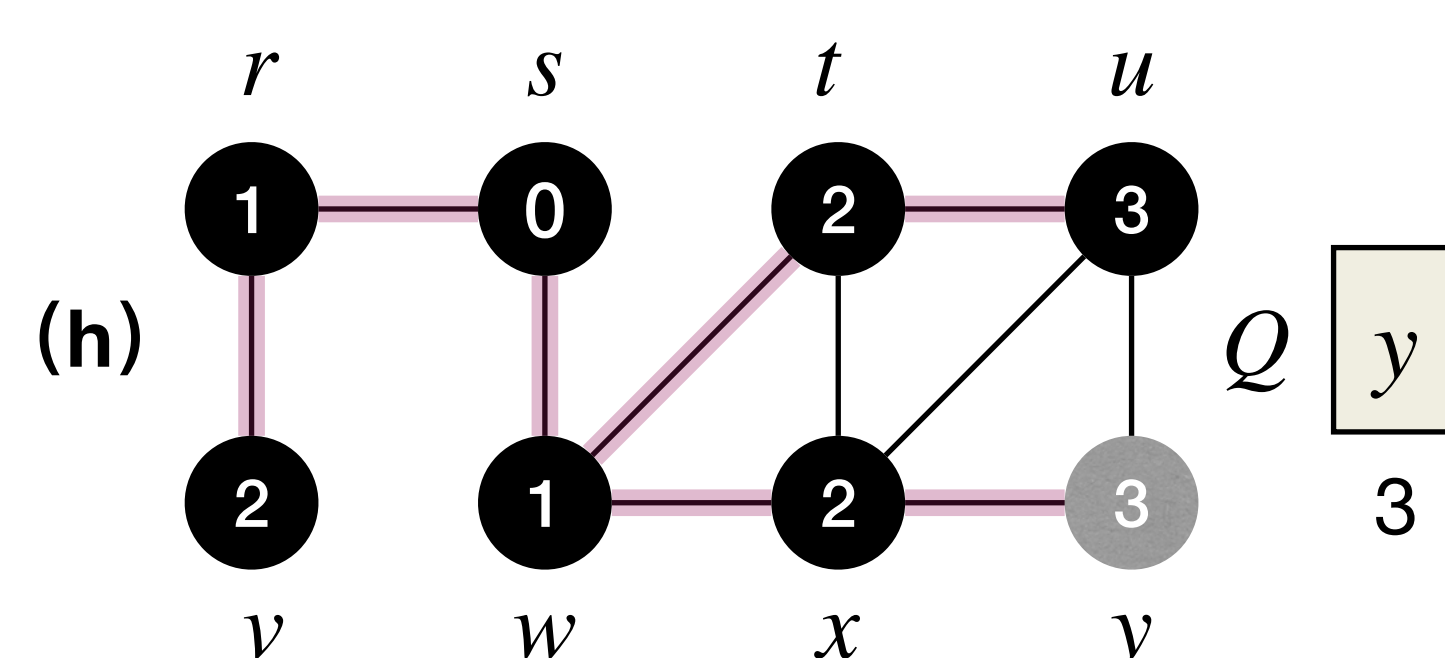
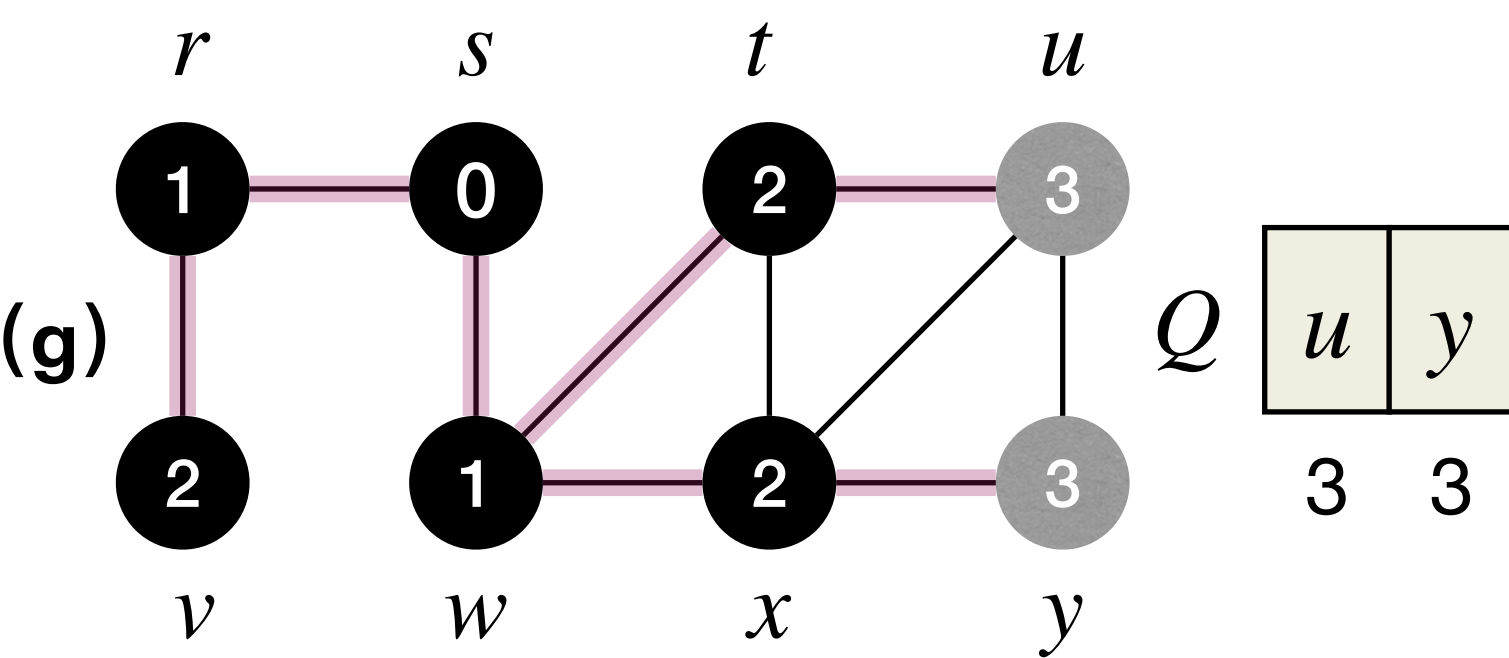
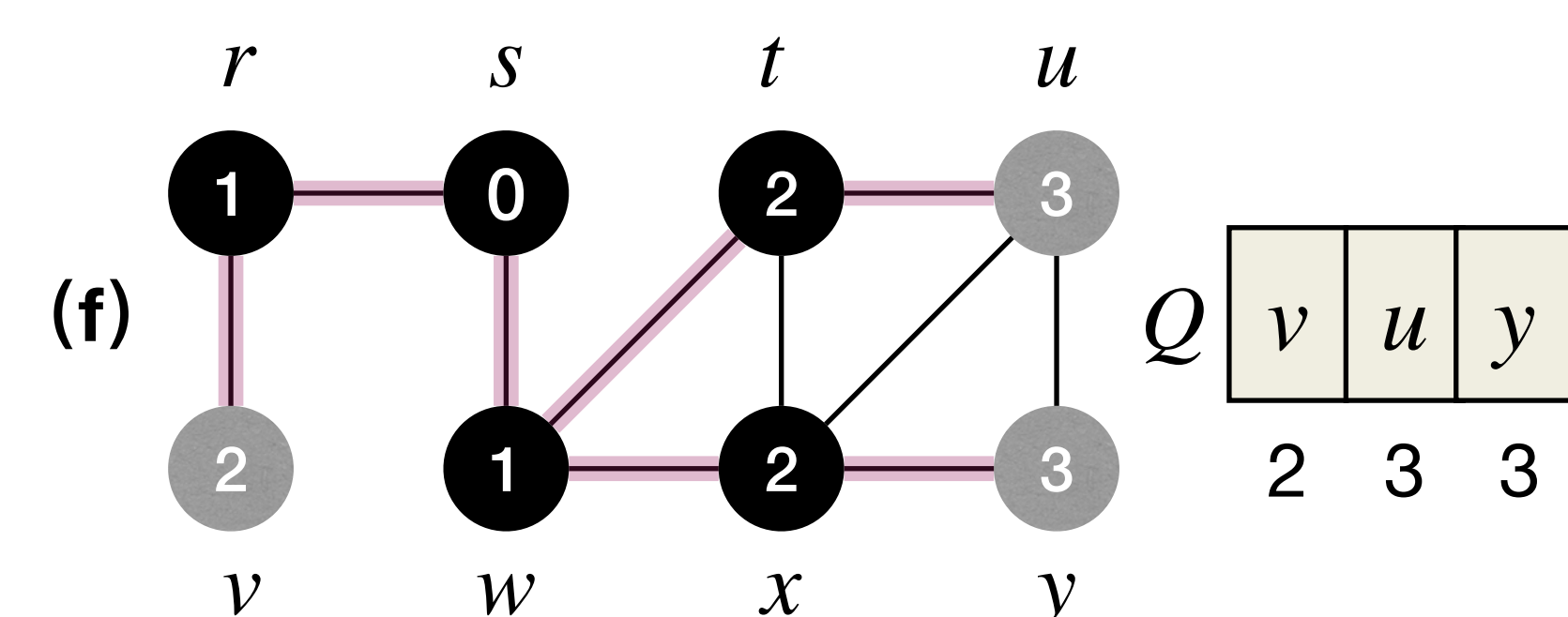
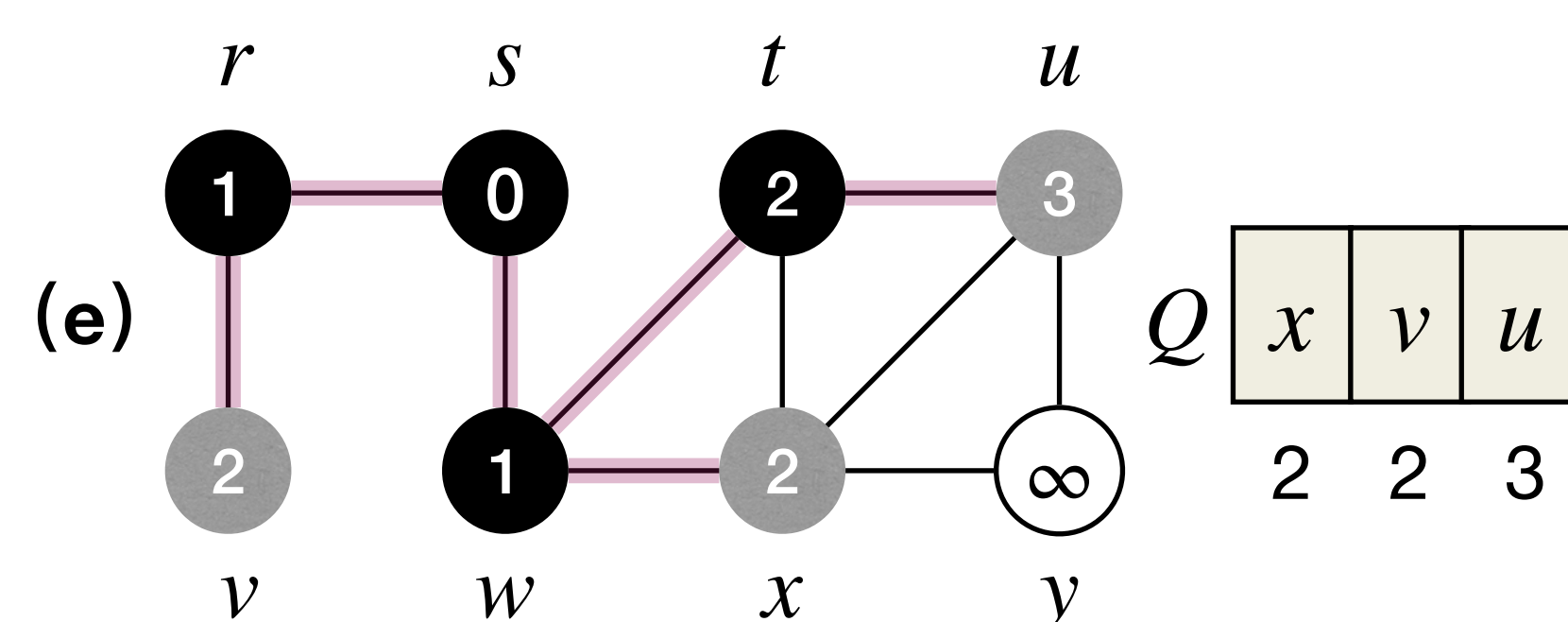
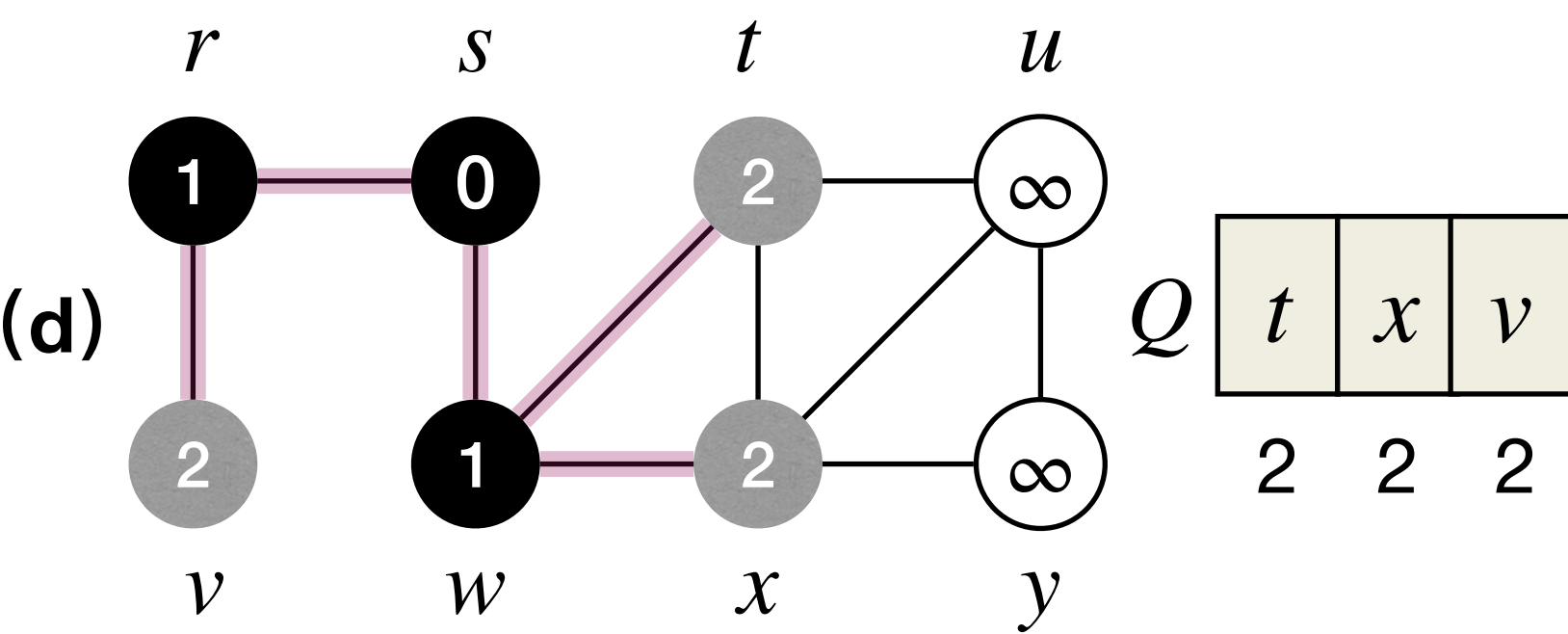
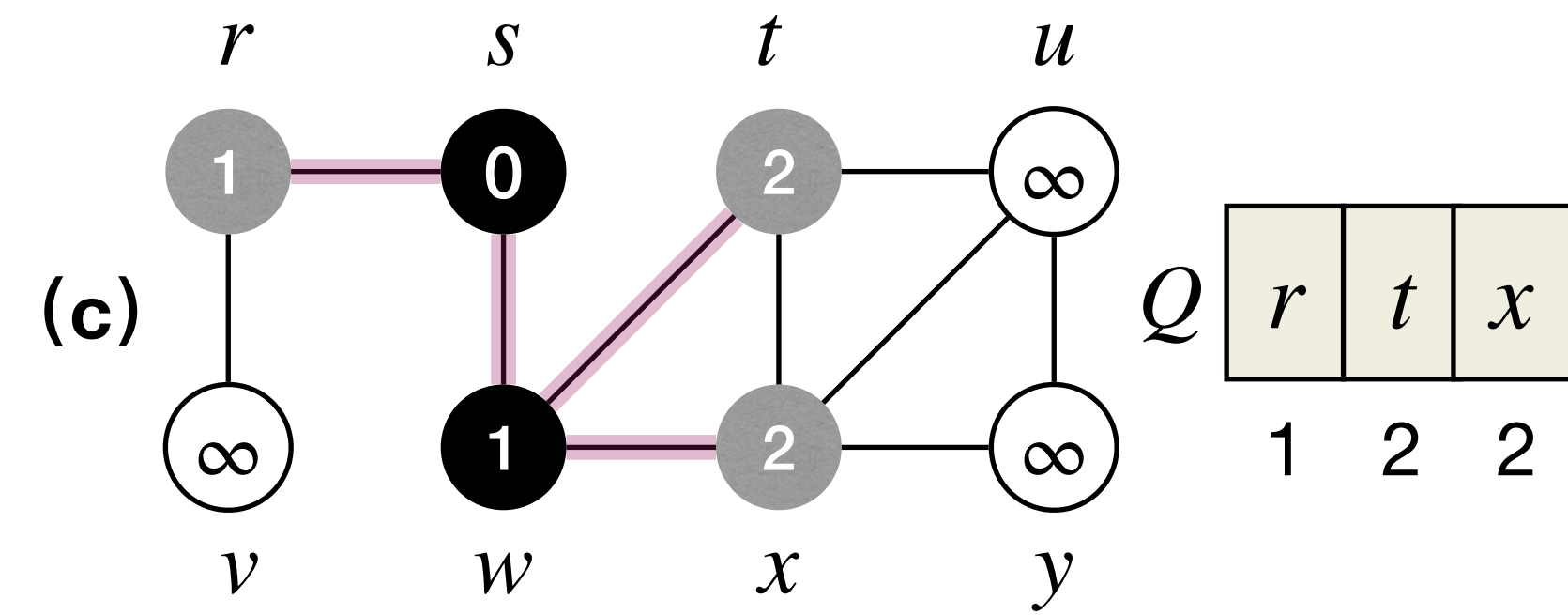
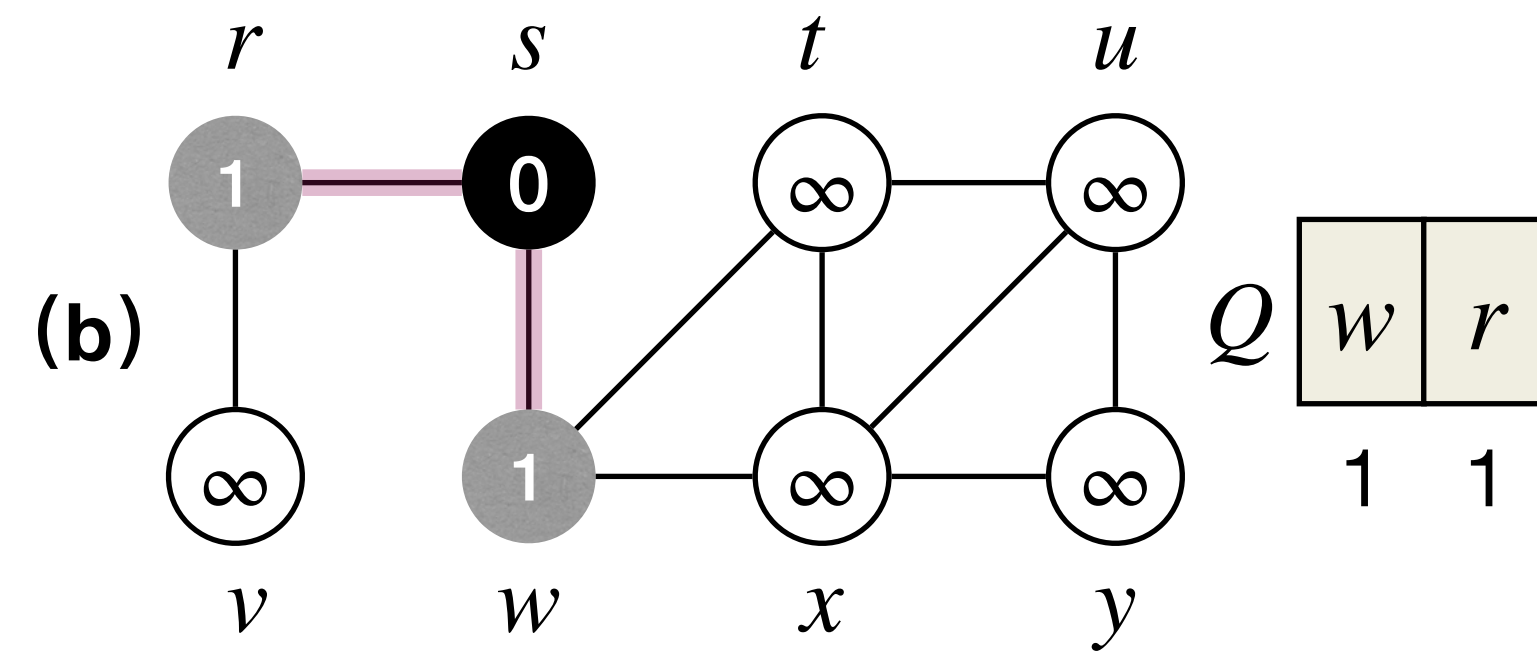
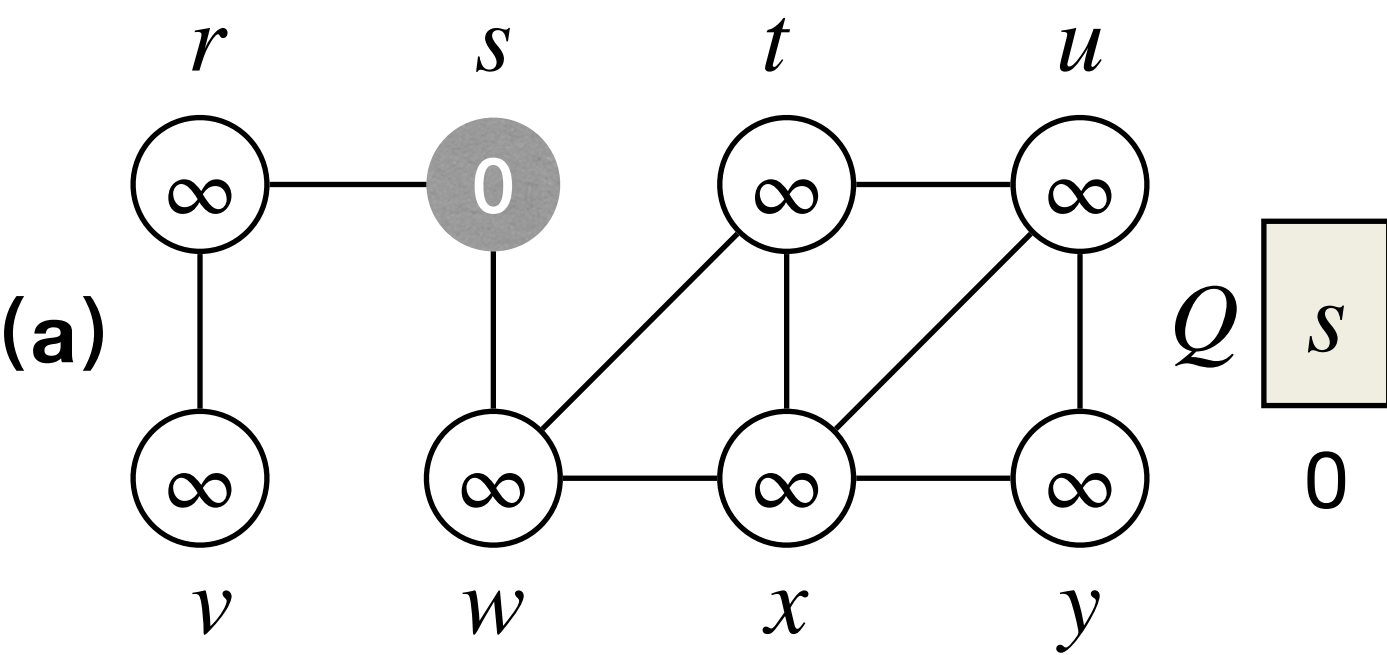
$v.d := u.d + 1$

$v.p := u$

$Q.enqueue(v)$



# Sample Execution





# Performance of BFS

- Runtime of BFS? (Assuming  $G$  is connected.)
- “While” loop  $\Theta(n)$  times.
  - ▶ Each node in  $Q$  at most once.
- “For” loop  $\Theta(m)$  times.
  - ▶ Each edge visited at most once or twice.
- Runtime of BFS is  $\Theta(n + m)$ .

BFS( $G, s$ ):

**for** each  $u$  in  $V$

$u.c := WHITE, u.d := INF, u.p := NIL$

$s.c := GRAY, s.d := 0, s.p := NIL$

$Q.enqueue(s)$

**while**  $!Q.empty()$

$u := Q.dequeue()$

$u.c := BLACK$

**for** each edge  $(u, v)$  in  $E$

**if**  $v.c = WHITE$

$v.c := GRAY$

$v.d := u.d + 1$

$v.p := u$

$Q.enqueue(v)$

What if we use adjacency matrix instead of adjacency list?



# Correctness and Properties of BFS

**Theorem** BFS visits a node iff it is reachable from  $s$ .

## Proof:

- [*only if*] If a node is not reachable from  $s$ , then BFS does not visit it, since BFS only moves along edges.
- [*if*] If a node is reachable from  $s$ , then BFS visits it.
  - **Claim:** For all  $k \geq 0$ , all nodes within  $k$  hops of  $s$  are visited.
    - [*Basis*]: Clearly  $s$  is visited.
    - [*Hypothesis*]: All nodes within  $k - 1$  hops of  $s$  are visited.



# Correctness and Properties of BFS

**Theorem** BFS visits a node iff it is reachable from  $s$ .

- [*Inductive Step*]: Consider a node  $v$  that is  $k$  hops away from  $s$ . Let  $u$  be  $v$ 's neighbor on (one of)  $v$ 's shortest path back to  $s$

By induction hypothesis,  $u$  gets visited.

When BFS visits  $u$ , node  $v$  is already **GRAY** or **BLACK**, or will be put in  $Q$ .

Either way,  $v$  eventually gets visited.

Will this really happen?!



# Correctness and Properties of BFS

**Theorem** BFS correctly computes  $u.dist$ , for every node  $u$  that is reachable from  $s$

- [**Proof Idea**] Use induction to show:
  - ▶ For all  $d \geq 0$ , there is a moment at which:
    - **(a)** every node  $u$  with  $dist(s, u) \leq d$  correctly computes  $u.dist$ ;
    - **(b)** every other node  $v$  has  $v.dist = \infty$ ;
    - **(c)**  $Q$  contains exactly the nodes  $d$  hops away from  $s$ .





# Correctness and Properties of BFS

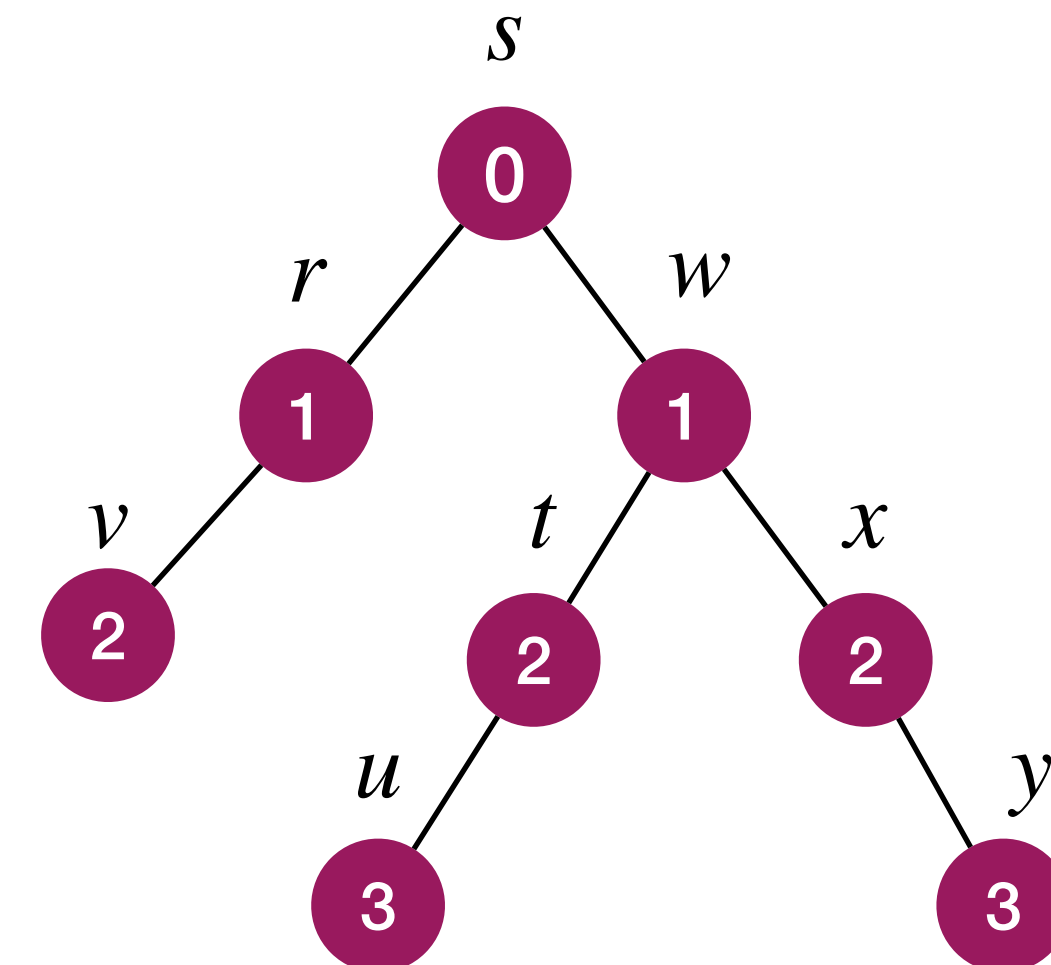
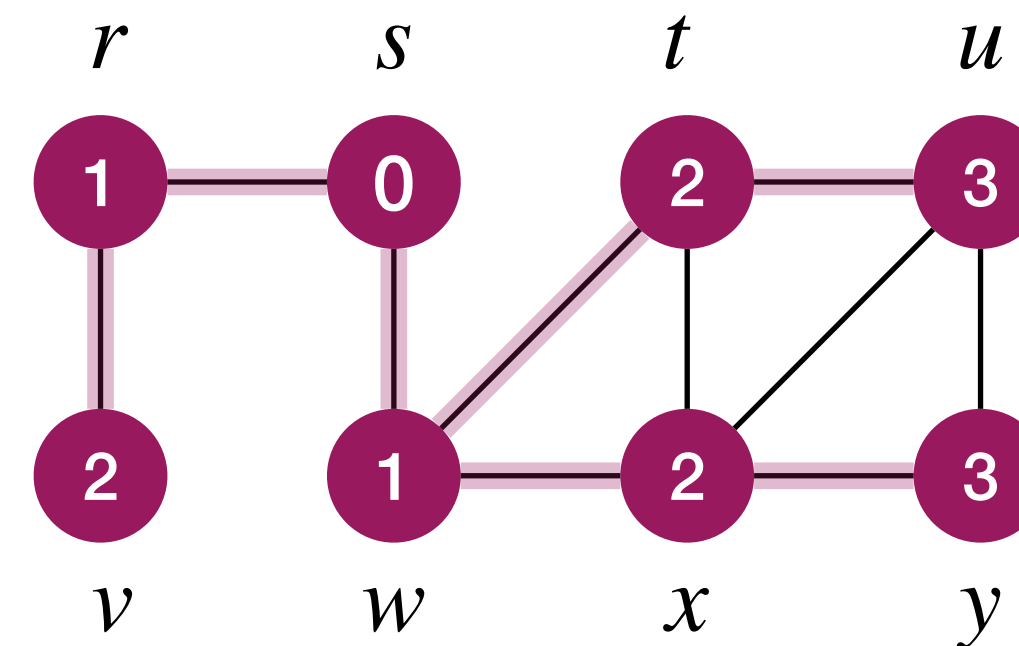
**Theorem** BFS correctly computes  $u.dist$ , for every node  $u$  that is reachable from  $s$

**Corollary** For any  $u \neq s$  that is reachable from  $s$ , one of the shortest path from  $s$  to  $u$  is a shortest path from  $s$  to  $u.p$  followed by the edge  $(u.p, u)$

- $G_p = (V_p, E_p)$  is a **breadth-first tree**, which can print on a shortest path from any node  $v$  to the source node  $s$ . Here:

$$\triangleright V_p = \{u \in V : u.p \neq \text{NIL}\} \cup \{s\},$$

$$\triangleright E_p = \left\{ (u.p, u) : u \in V_p - \{s\} \right\}.$$





# One last note on BFS

- What if the graph is not connected?
  - ▶ Easy, do a BFS for each connected component!

Runtime of this procedure?

BFS(G):

**for** each  $u$  in  $V$

$u.c := WHITE, u.d := INF, u.p := NIL$

**for** each  $u$  in  $V$

**if**  $u.c = WHITE$

$u.c := GRAY, u.d := 0, u.p := NIL$

$Q.enqueue(u)$

**while**  $!Q.empty()$

$v := Q.dequeue()$

$v.c := BLACK$

**for** each edge  $(v, w)$  in  $E$

**if**  $w.c = WHITE$

$w.c := GRAY$

$w.d := v.d + 1$

$w.p := v$

$Q.enqueue(w)$



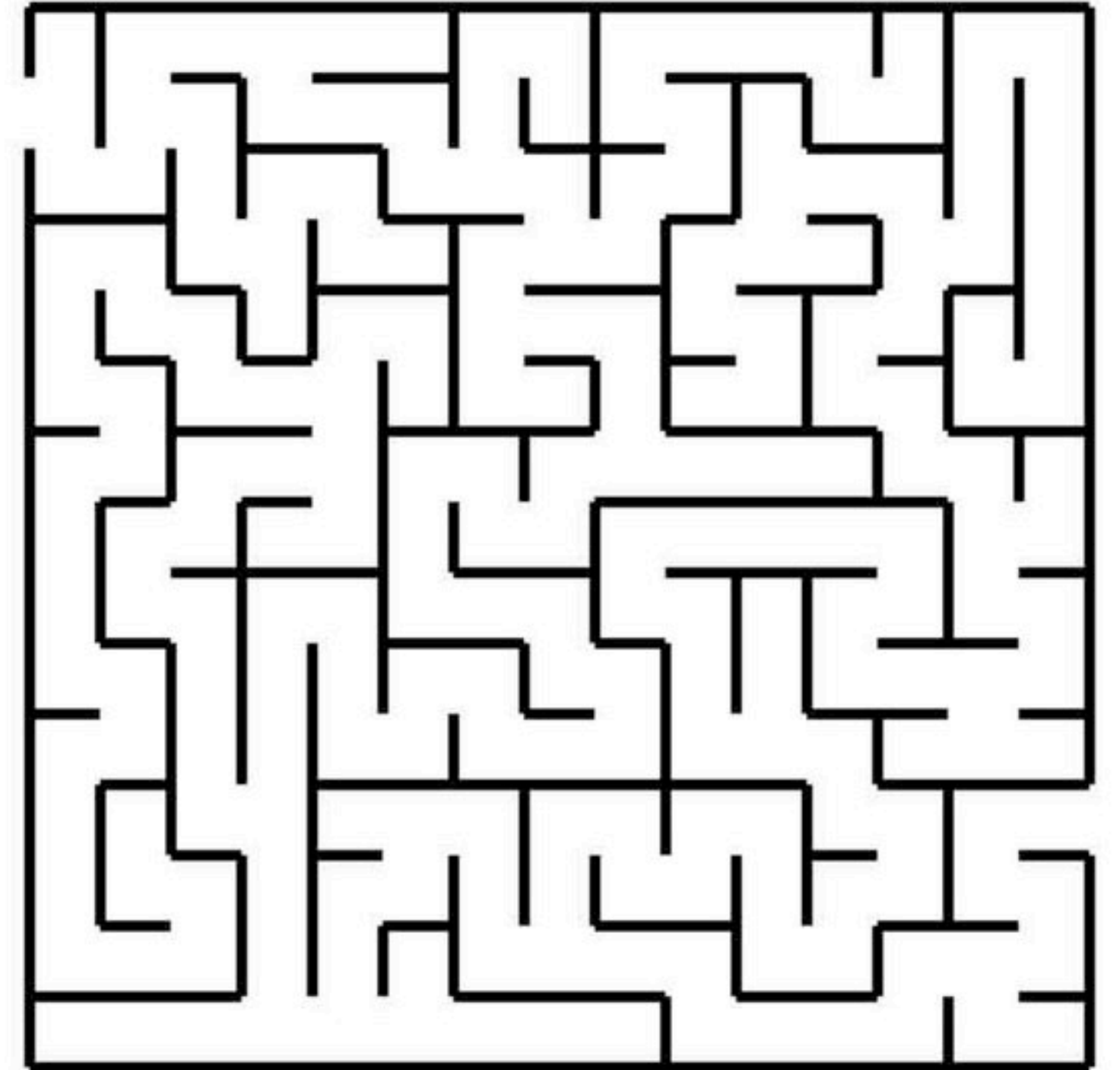
# Depth-First Search





# Depth-First Search (DFS)

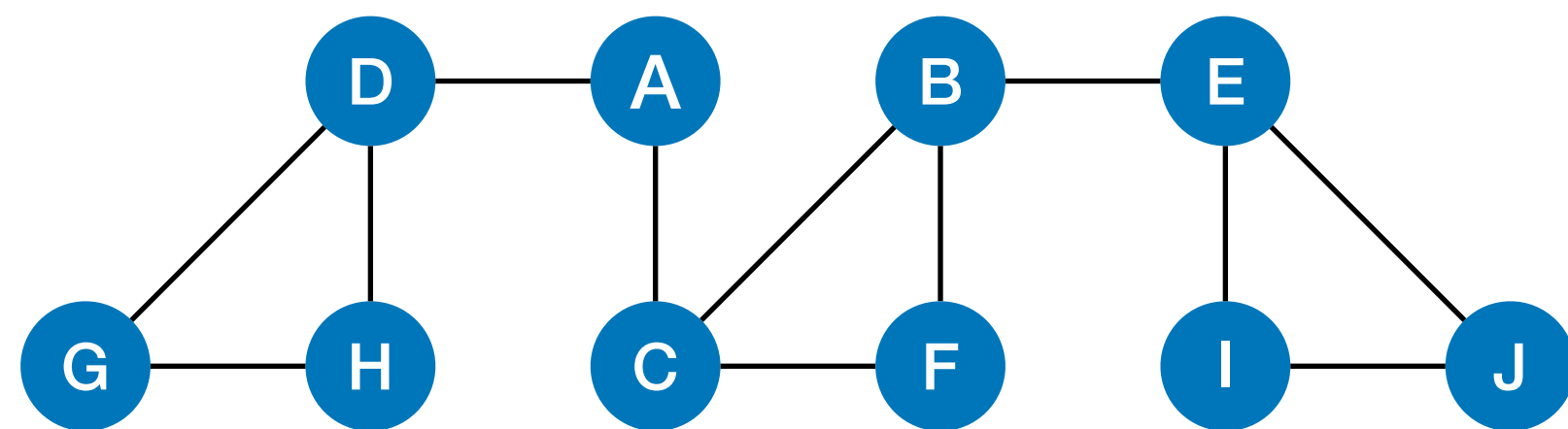
- Much like exploring a maze:
  - ▶ Use **a ball of string** and **a piece of chalk**.
  - ▶ Follow path (unwind string and mark at intersections), until stuck (reach dead-end or already-visited place).
  - ▶ Backtrack (rewind string), until find unexplored neighbor (intersection with unexplored direction).
  - ▶ Repeat above two steps.





# Depth-First Search (DFS)

- How to do this for a graph, in computer?
  - ▶ **Chalk**: boolean variables.
  - ▶ **String**: a stack.



## DFS Skeleton(G, s):

$s.visited := True$

**for each** edge  $(s, v)$  **in**  $E$

**if**  $!v.visited$

$DFS\text{Skeleton}(G, v)$

## DFSIterSkeleton(G, s):

*Stack*  $Q$

$Q.push(s)$

**while**  $!Q.empty()$

$u := Q.pop()$

**if**  $!u.visited$

$u.visited := True$

**for each** edge  $(u, v)$  **in**  $E$

$Q.push(v)$



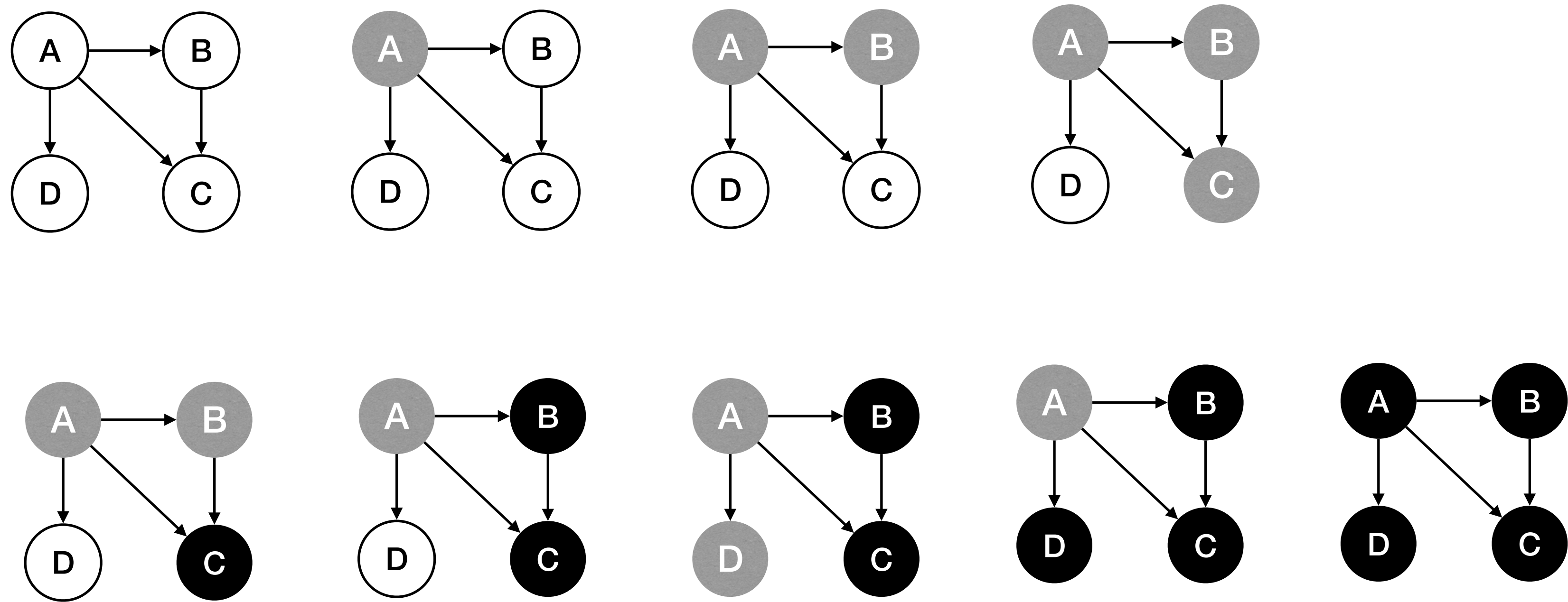
### DFSSkeleton(G, s):

$s.visited := True$

**for each** edge  $(s, v)$  **in**  $E$

**if**  $!v.visited$

$DFSSkeleton(G, v)$



### DFSIterSkeleton(G, s):

*Stack*  $Q$

$Q.push(s)$

**while**  $!Q.empty()$

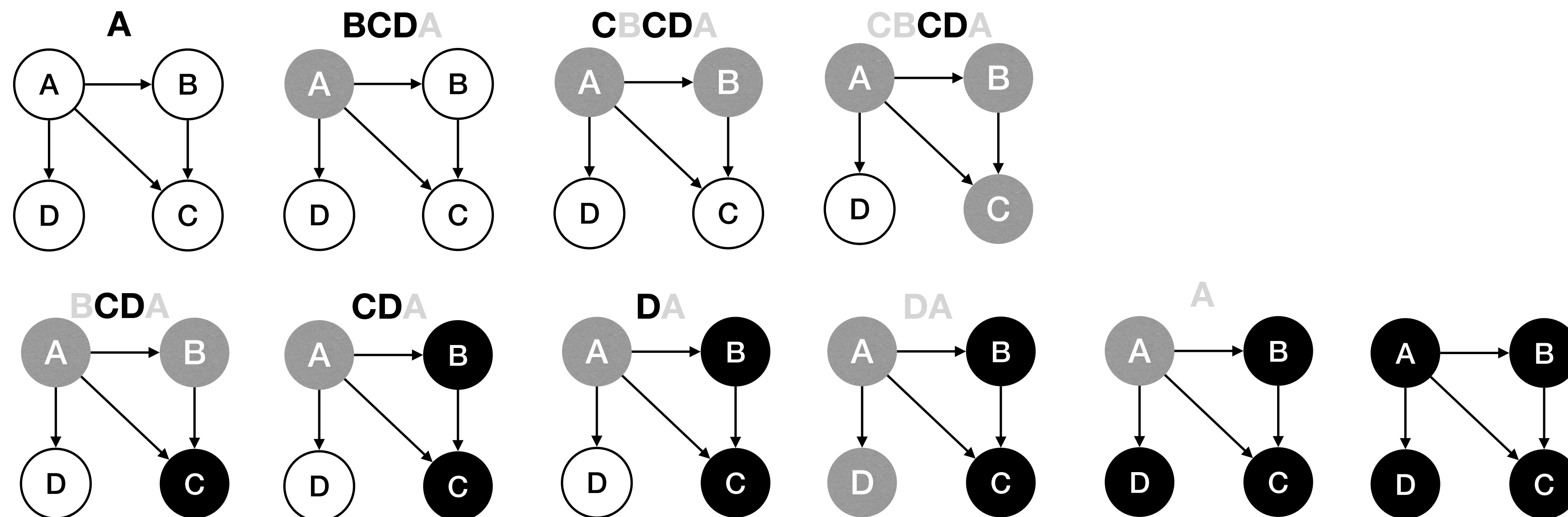
$u := Q.pop()$

**if**  $!u.visited$

$u.visited := True$

**for each** edge  $(u, v)$  **in**  $E$

$Q.push(v)$





# Depth-First Search (DFS)

- What if the graph is not (strongly) connected?
  - Do DFS from multiple sources.

## DFSAll(G):

```
for each node  $u$  in  $V$ 
     $v.visited := False$ 
for each node  $u$  in  $V$ 
    if  $!u.visited$ 
        DFSSkeleton( $G, u$ )
```

## DFS Skeleton( $G, s$ ):

```
 $s.visited := True$ 
for each edge  $(s, v)$  in  $E$ 
    if  $!v.visited$ 
        DFSSkeleton( $G, v$ )
```

## DFSAll(G):

```
for each node  $u$  in  $V$ 
     $v.visited := False$ 
for each node  $u$  in  $V$ 
    if  $!u.visited$ 
        DFSIterSkeleton( $G, u$ )
```

## DFSIterSkeleton( $G, s$ ):

```
Stack  $Q$ 
 $Q.push(s)$ 
while  $!Q.empty()$ 
     $u := Q.pop()$ 
    if  $!u.visited$ 
         $u.visited := True$ 
        for each edge  $(u, v)$  in  $E$ 
             $Q.push(v)$ 
```



# Depth-First Search (DFS)

## DFSAll(G):

```
for each node  $u$  in  $V$   
     $u.color := WHITE$   
     $u.parent := NIL$ 
```

```
for each node  $u$  in  $V$   
    if  $u.color = WHITE$   
         $DFS(G, u)$ 
```

## DFS(G, s):

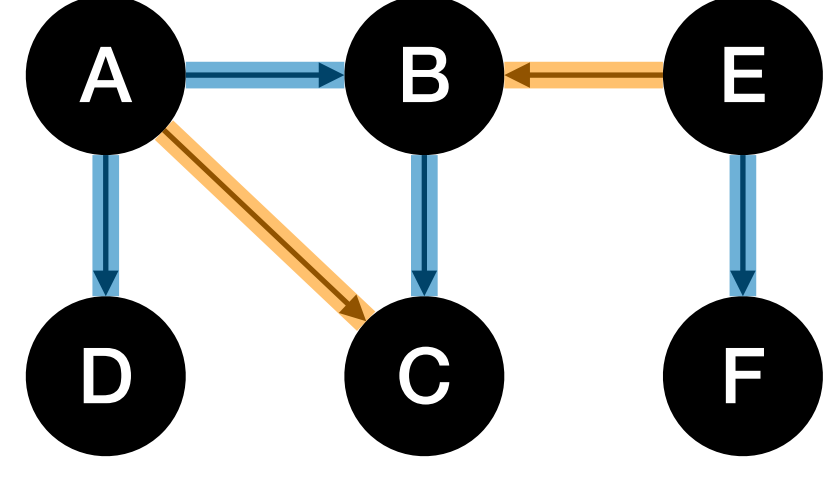
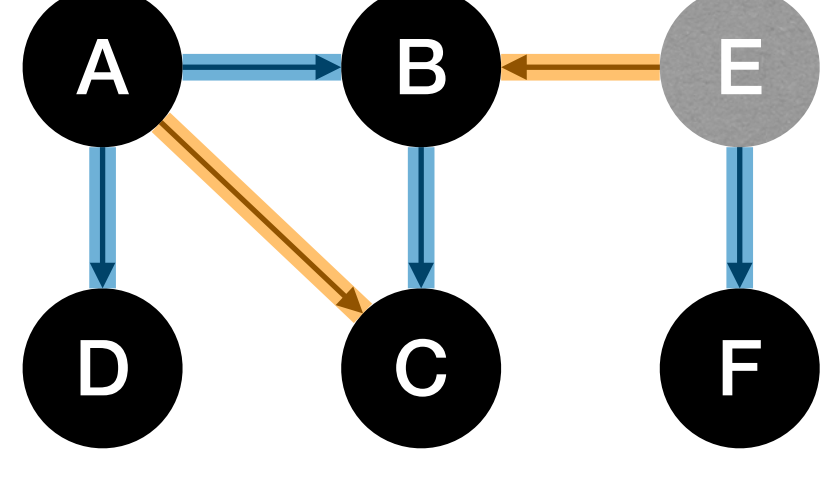
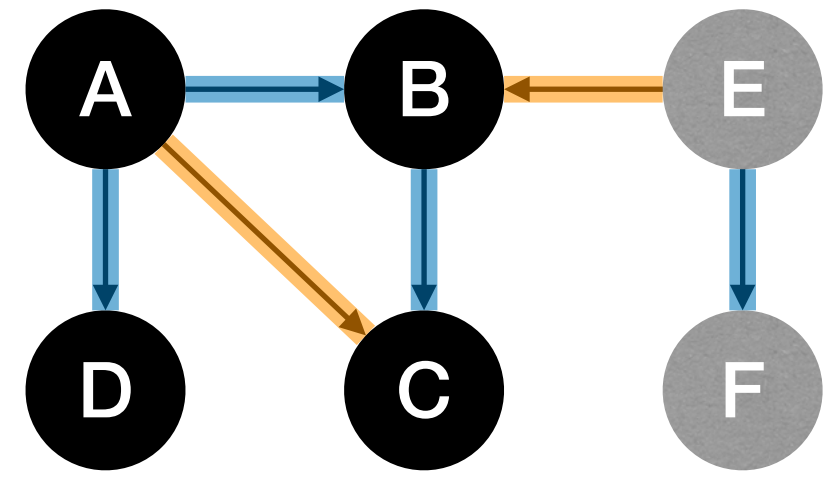
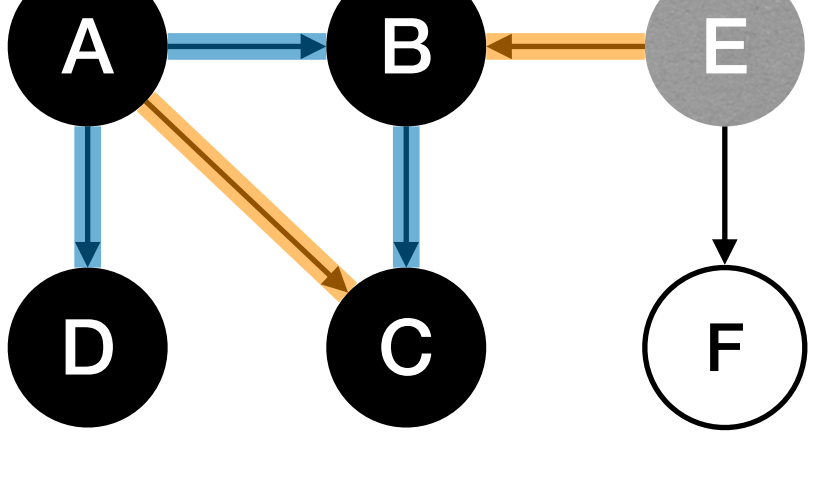
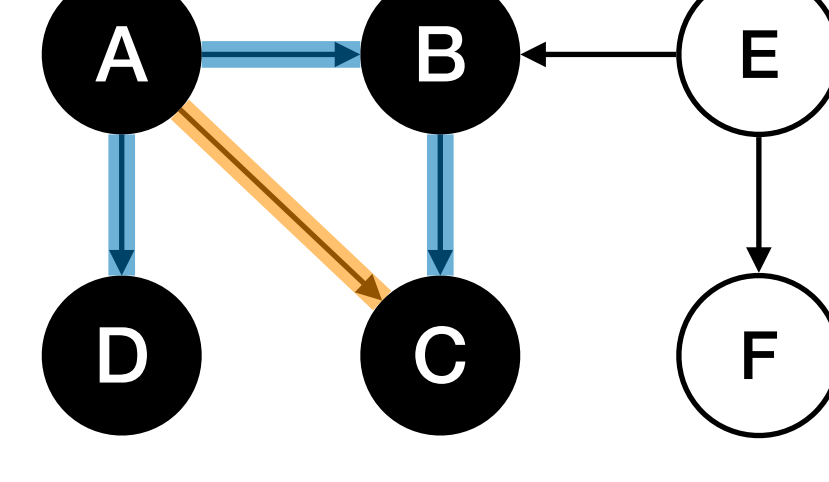
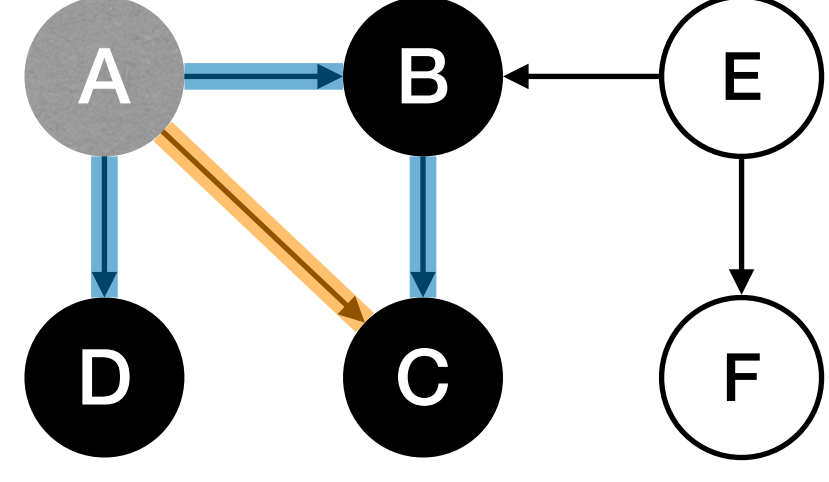
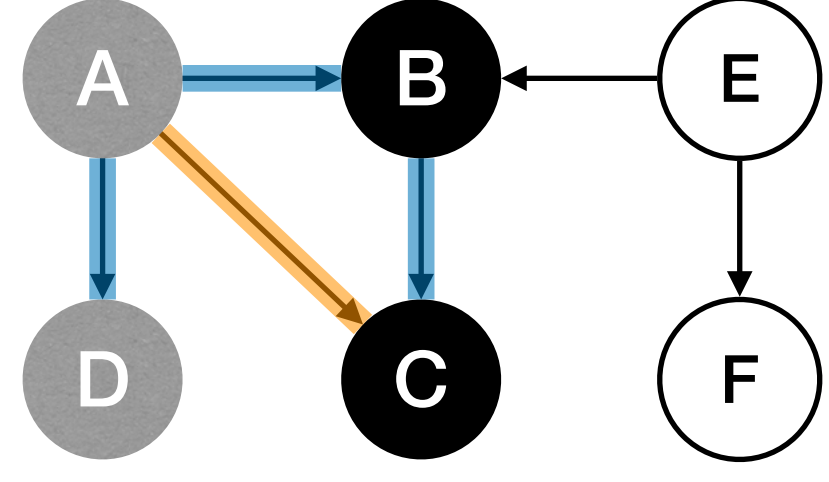
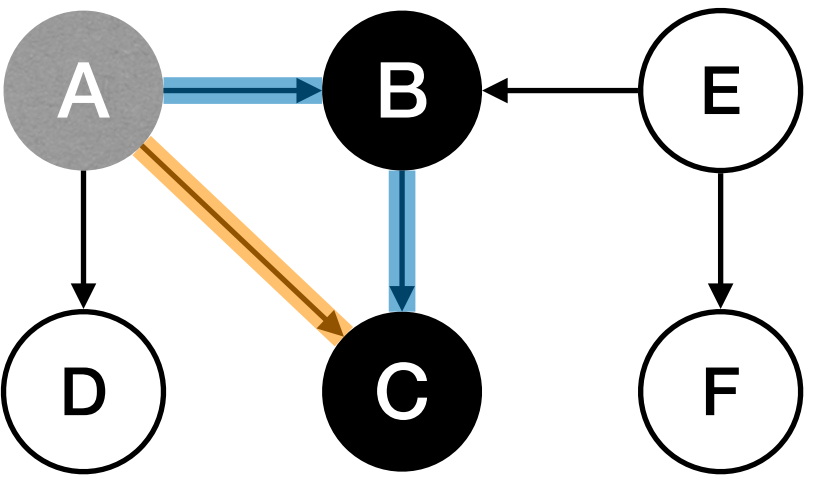
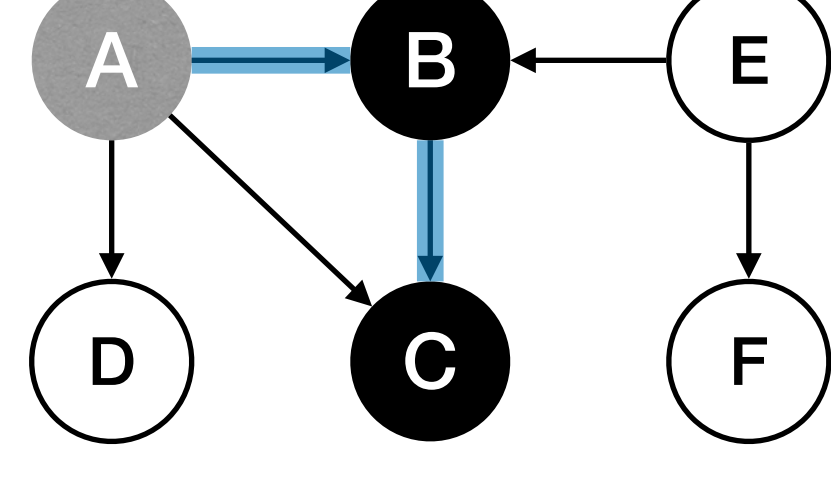
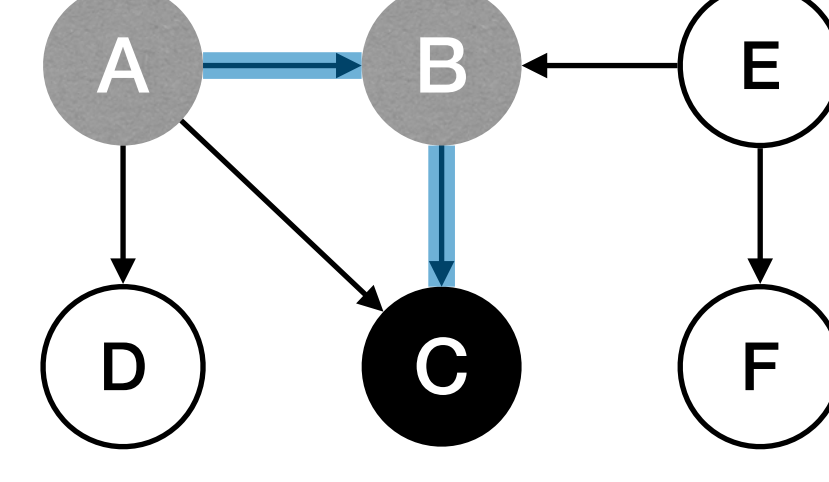
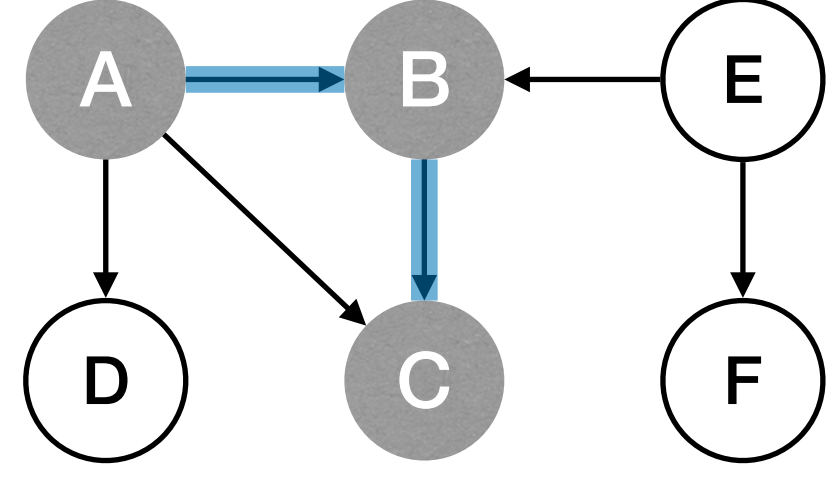
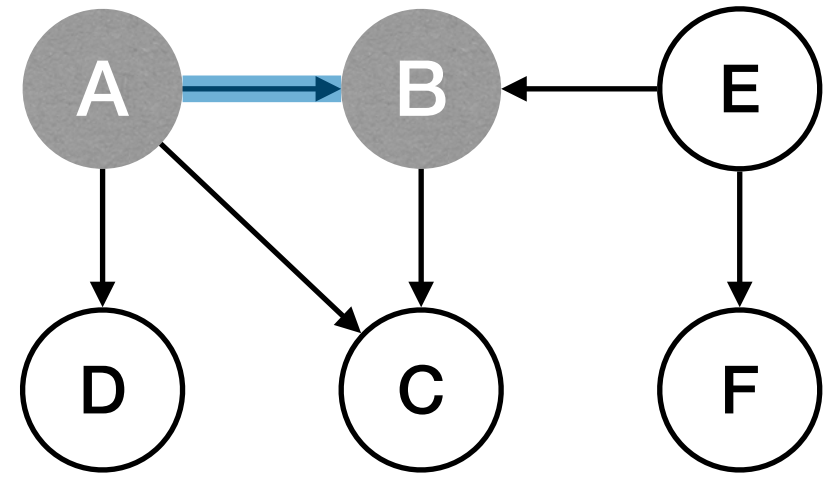
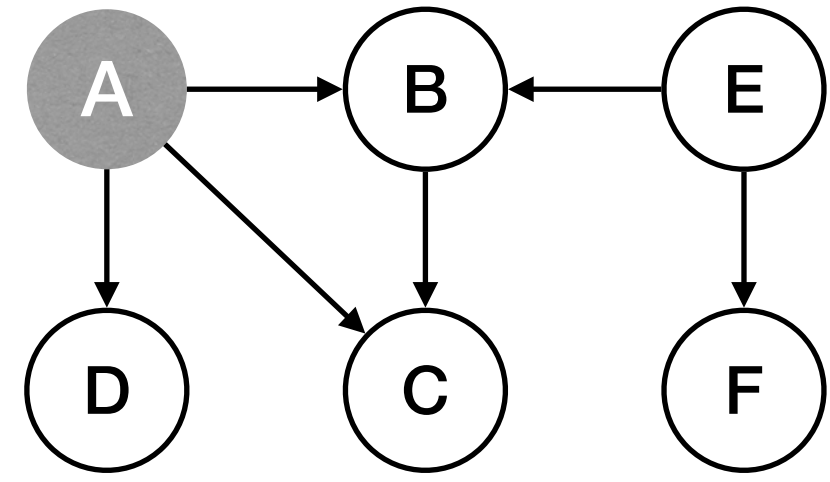
```
 $s.color := GRAY$   
for each edge  $(s, v)$  in  $E$   
    if  $v.color = WHITE$ 
```

```
         $v.parent := s$   
         $DFS(G, v)$ 
```

```
 $s.color := BLACK$ 
```

- Each node  $u$  have 3 status during DFS:
  - ▶ **Undiscovered** [ **WHITE** ]: before calling `DFSkeleton(G, u)`
  - ▶ **Discovered** [ **GRAY** ]: during execution of `DFSkeleton(G, u)`
  - ▶ **Finished** [ **BLACK** ]: `DFSkeleton(G, u)` returned
- `DFS(G, u)` builds a **tree** among nodes reachable from  $u$ :
  - ▶ Root of this tree is  $u$ .
  - ▶ For each non-root, its parent is the node that makes it turn GRAY.
- DFS on entire graph builds a **forest**.







# Depth-First Search (DFS)

- DFS provides (at least) two chances to process each node:
  - ▶ **Pre-Visit:** WHITE  $\rightarrow$  GRAY
  - ▶ **Post-Visit:** GRAY  $\rightarrow$  BLACK
- Sample application: Track **active intervals** of nodes
  - ▶ Clock ticks whenever some node's color changes.
  - ▶ **Discovery time:** when the node turns to GRAY.
  - ▶ **Finish time:** when the node turns to BLACK.

## DFSAll(G):

### *PreProcess(G)*

```

for each node  $u$  in  $V$ 
   $u.color := WHITE$ 
   $u.parent := NIL$ 
for each node  $u$  in  $V$ 
  if  $\neg u.visited$ 
     $DFS(G, u)$ 
  
```

## DFS(G, s):

### *PreVisit(s)*

```

 $s.color := GRAY$ 
for each edge  $(s, v)$  in  $E$ 
  if  $v.color = WHITE$ 
     $v.parent := s$ 
     $DFS(G, v)$ 
 $s.color := BLACK$ 
PostVisit(s)
  
```

## PreProcess(G):

$time := 0$

## PreVisit(s):

$time := time + 1$

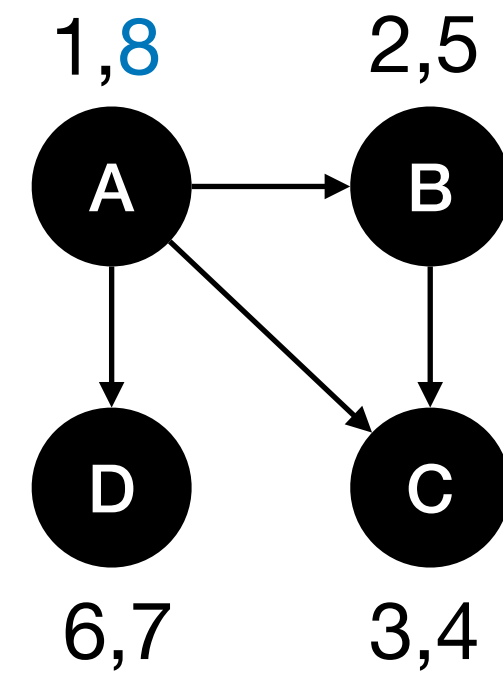
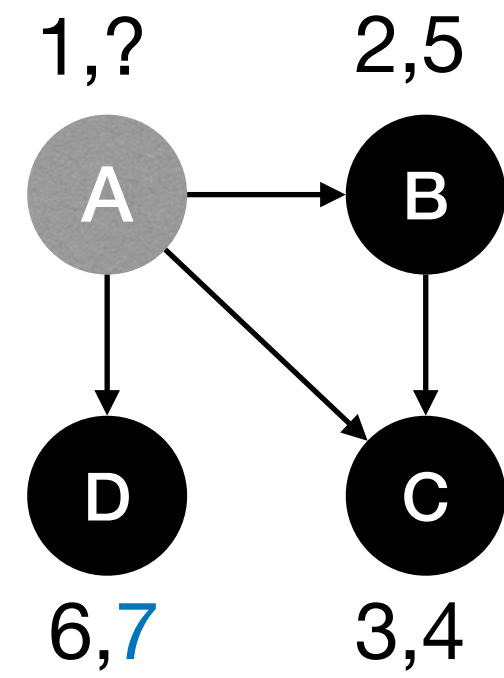
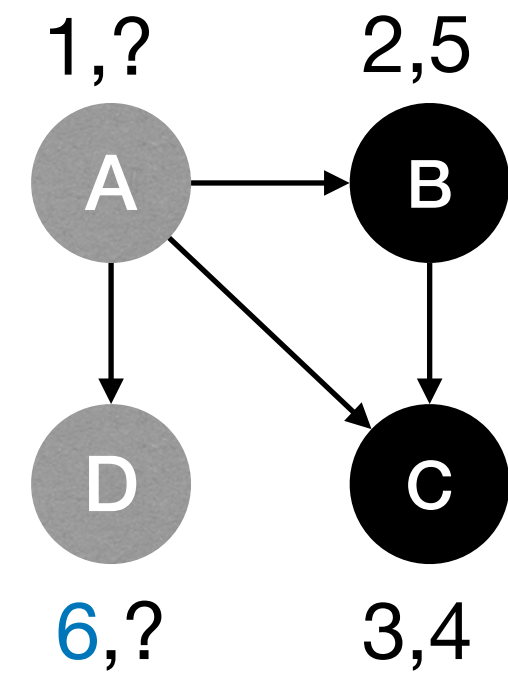
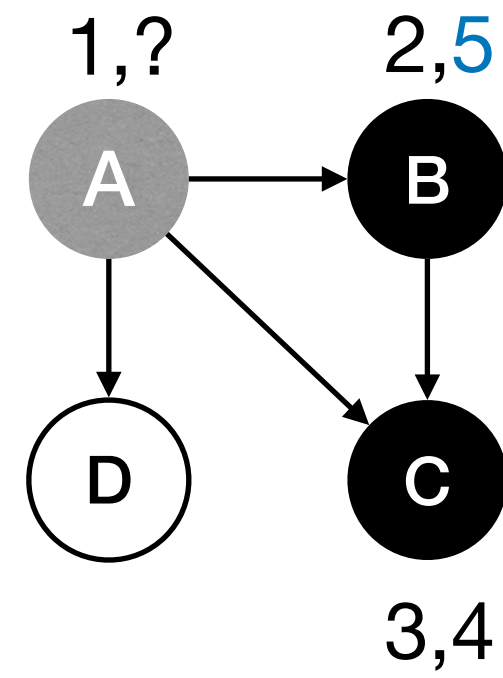
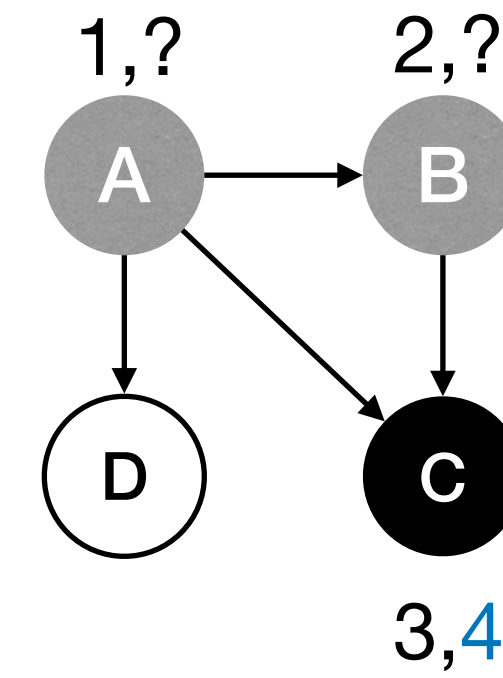
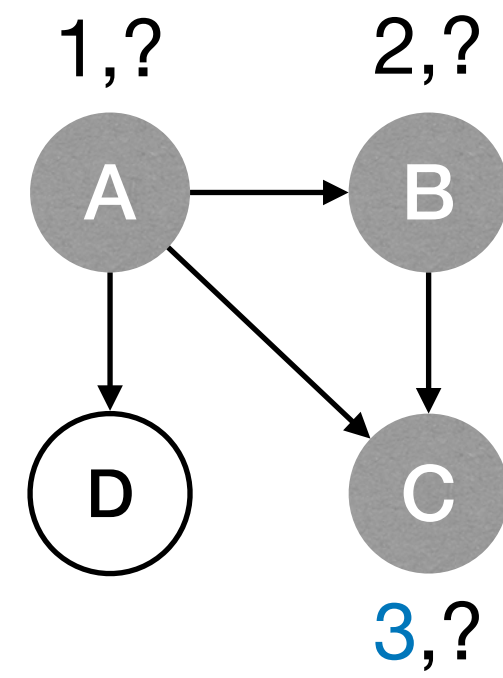
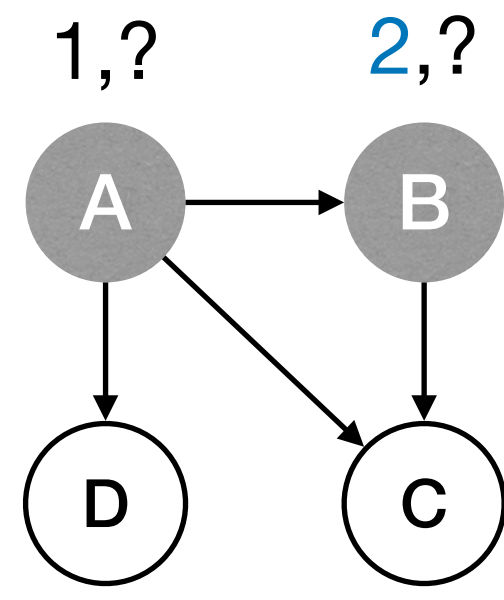
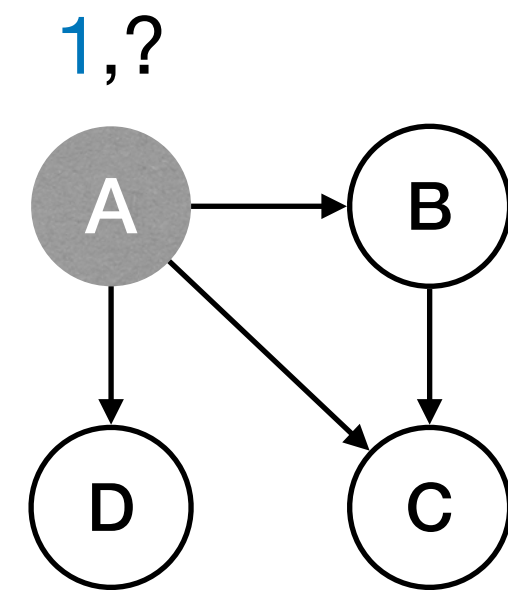
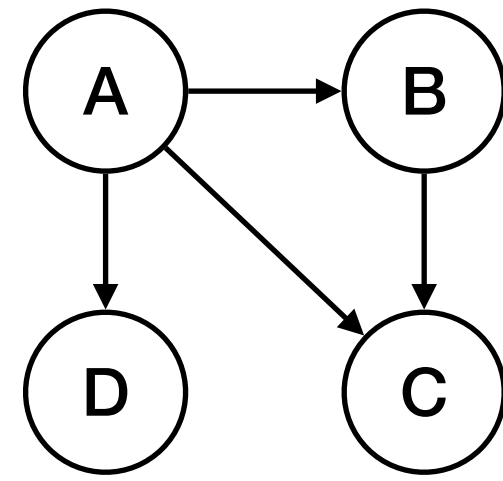
$s.d := time$

**Note:** here it indicates the discovery time

## PostProcess(G):

$time := time + 1$

$s.f := time$



### DFSAll(G):

*PreProcess(G)*

**for each** node  $u$  **in**  $V$

$u.color := WHITE$

$u.parent := NIL$

**for each** node  $u$  **in**  $V$

**if**  $\neg u.visited$

$DFS(G, u)$

### DFS(G, s):

*PreVisit(s)*

$s.color := GRAY$

**for each** edge  $(s, v)$  **in**  $E$

**if**  $v.color = WHITE$

$v.parent := s$

$DFS(G, v)$

$s.color := BLACK$

*PostVisit(s)*

### PreProcess(G):

$time := 0$

### PreVisit(s):

$time := time + 1$

$s.d := time$

### PostProcess(G):

$time := time + 1$

$s.f := time$



# Runtime of DFS

- Time spent on each node:  $O(1)$ 
  - DFS ( $G, u$ ) is called once for each node  $u$ .
- Time spent on each edge:  $O(1)$ 
  - Each edge is examined  $O(1)$  times.

Total runtime:  $O(n + m)$

## DFSAll(G):

### *PreProcess(G)*

```
for each node  $u$  in  $V$ 
   $u.color := WHITE$ 
   $u.parent := NIL$ 
for each node  $u$  in  $V$ 
  if  $\neg u.visited$ 
    DFS( $G, u$ )
```

## DFS(G, s):

### *PreVisit(s)*

```
 $s.color := GRAY$ 
for each edge  $(s, v)$  in  $E$ 
  if  $v.color = WHITE$ 
     $v.parent := s$ 
    DFS( $G, v$ )
 $s.color := BLACK$ 
PostVisit(s)
```

## PreProcess(G):

$time := 0$

## PreVisit(s):

$time := time + 1$   
 $s.d := time$

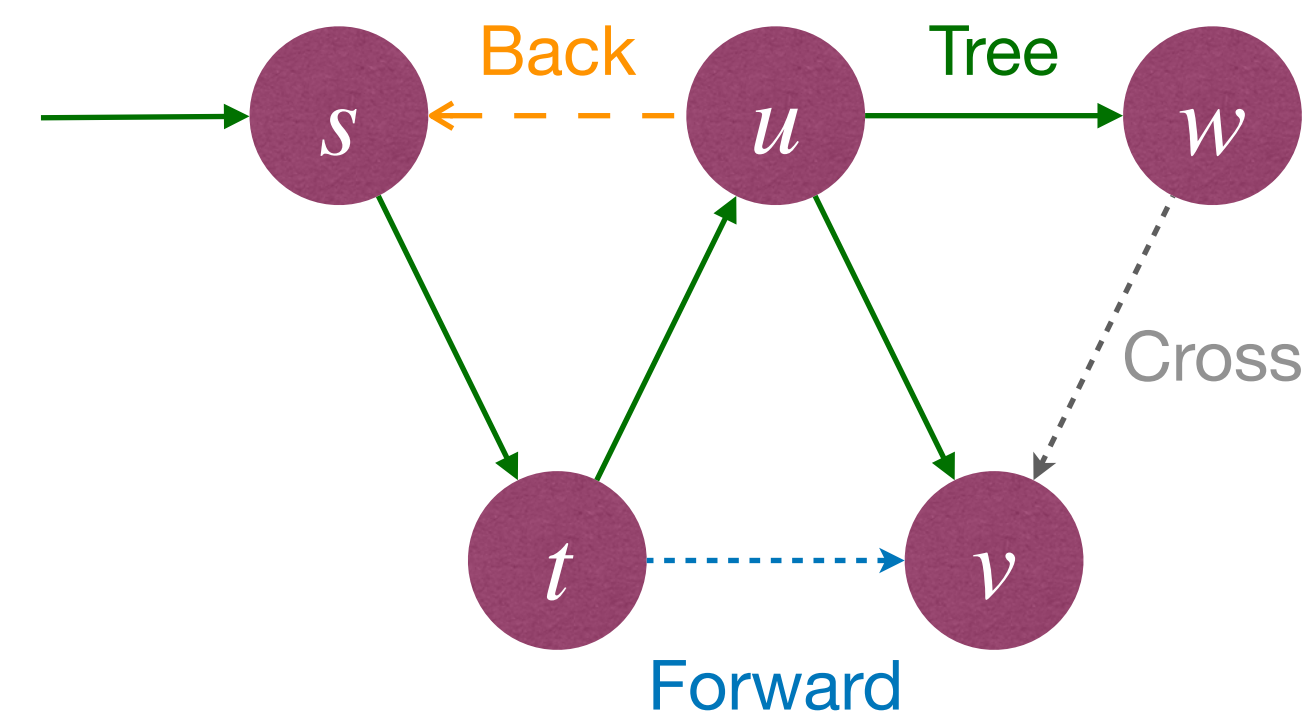
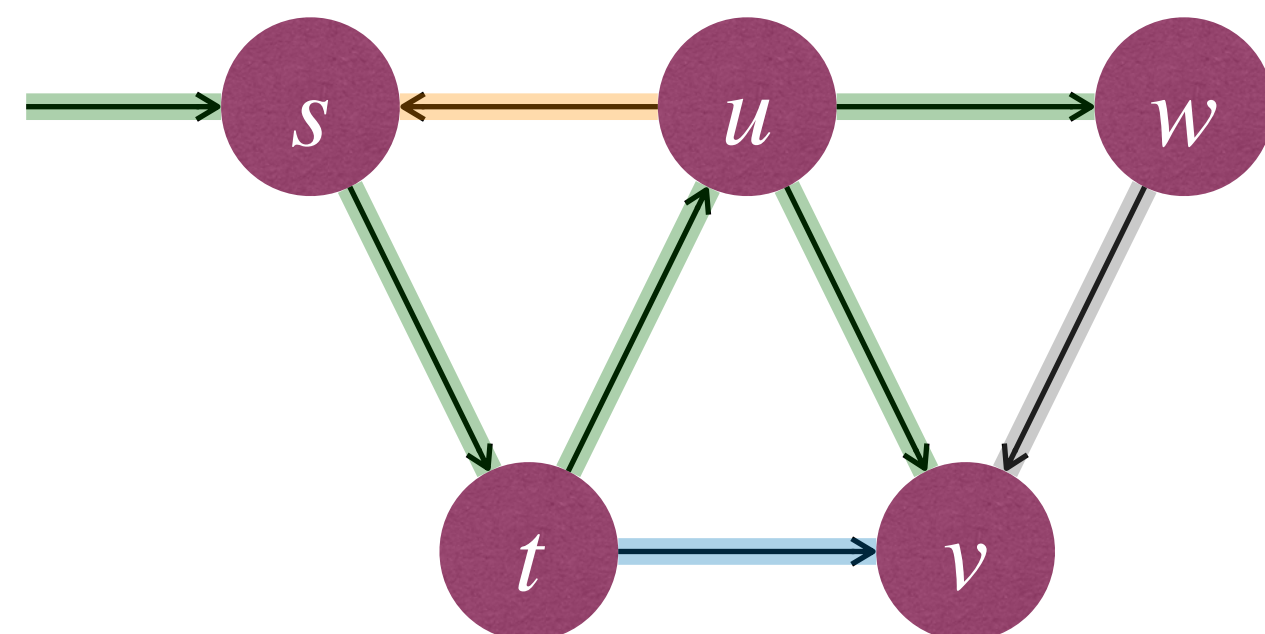
## PostProcess(G):

$time := time + 1$   
 $s.f := time$



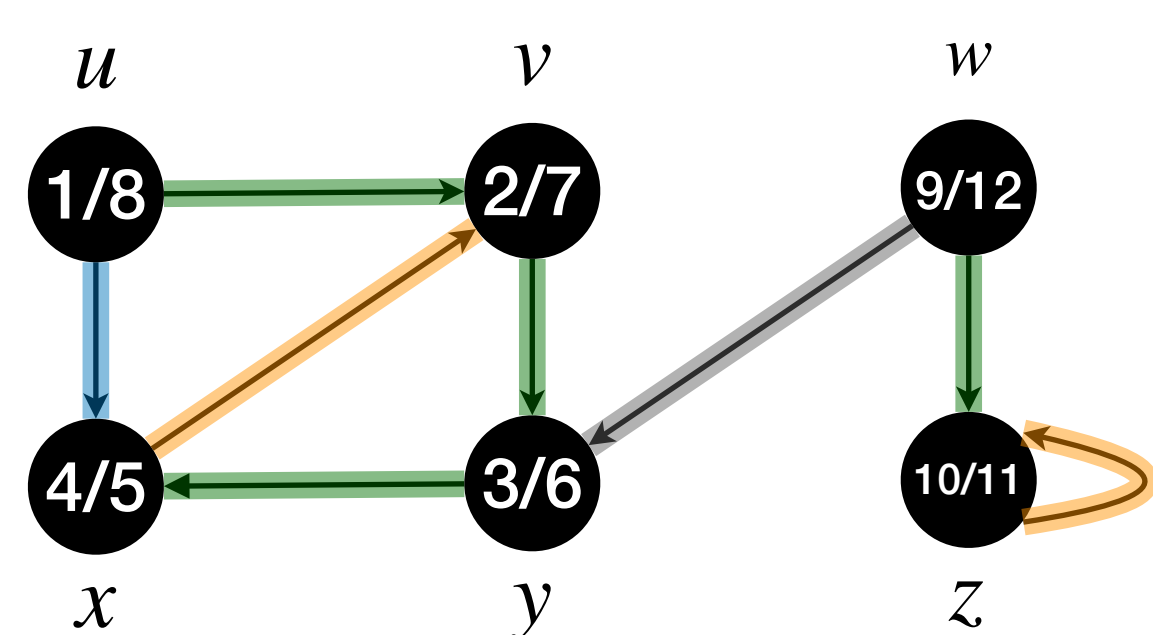
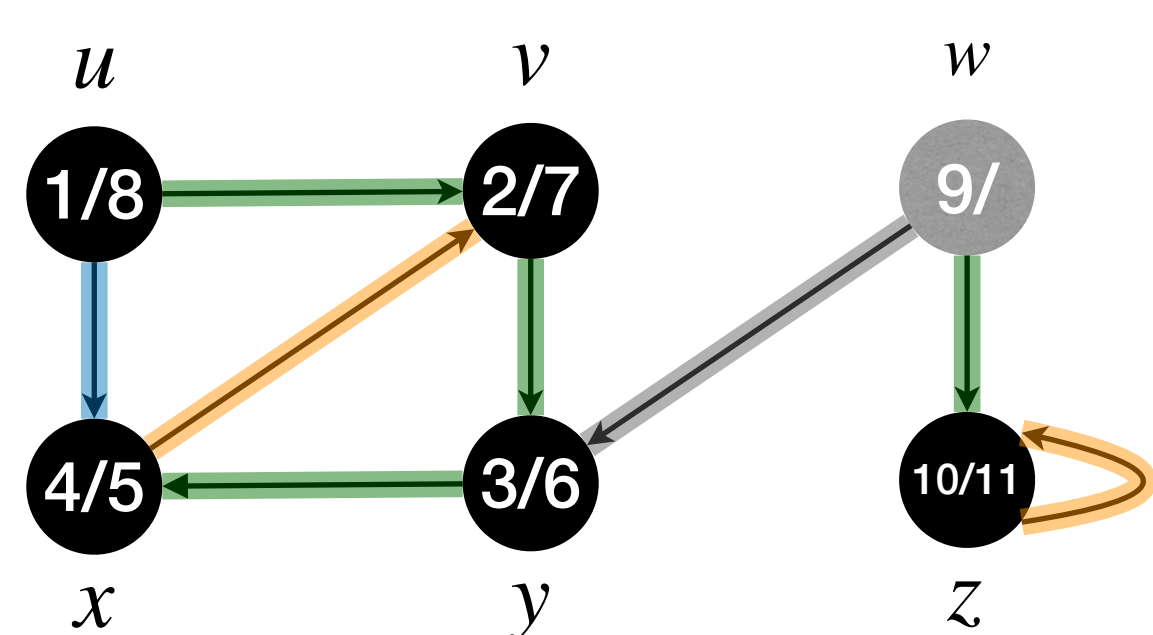
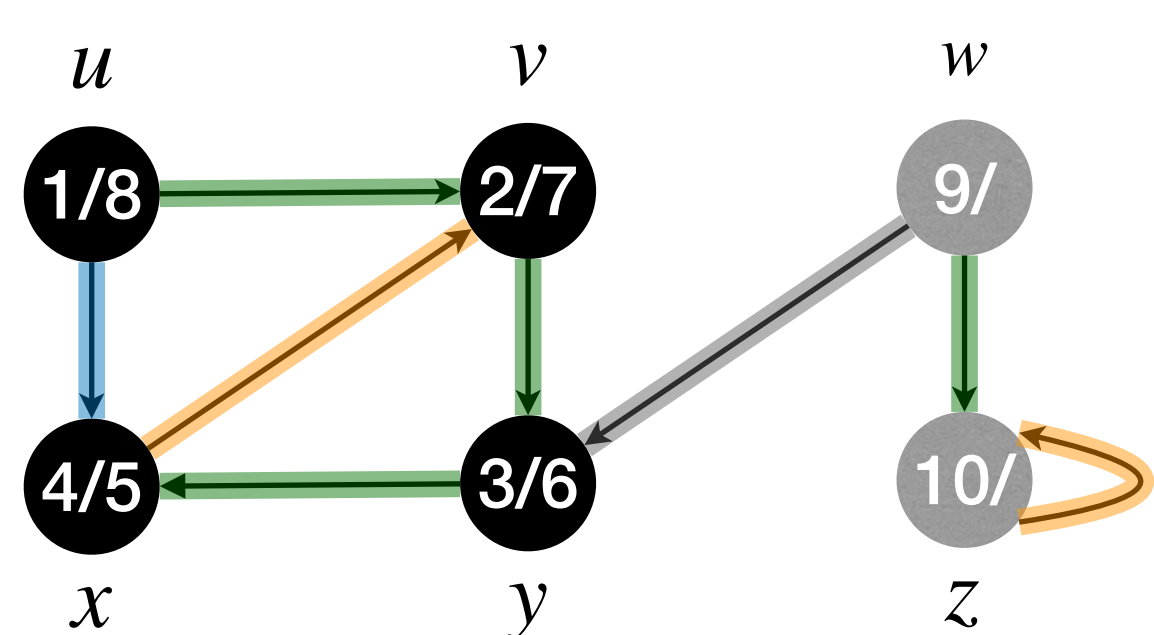
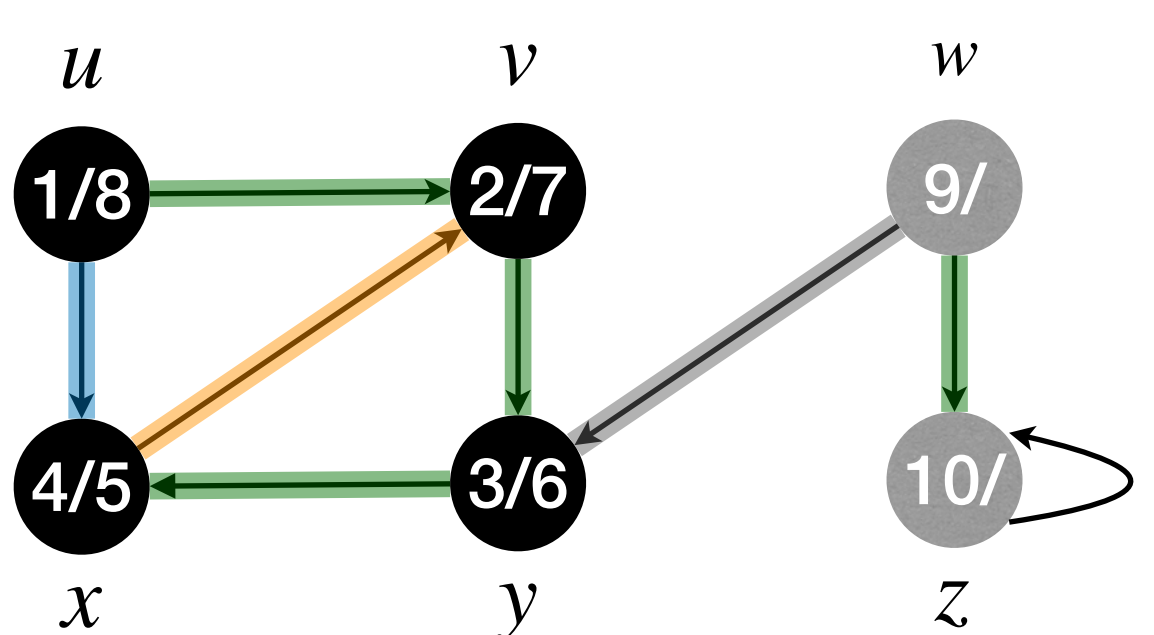
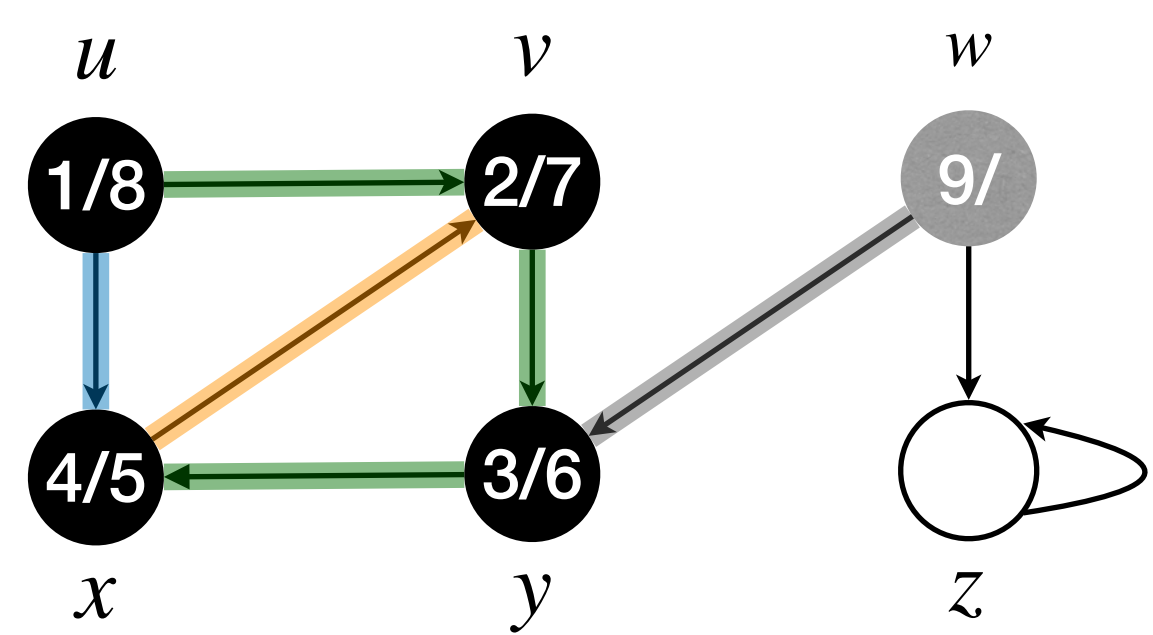
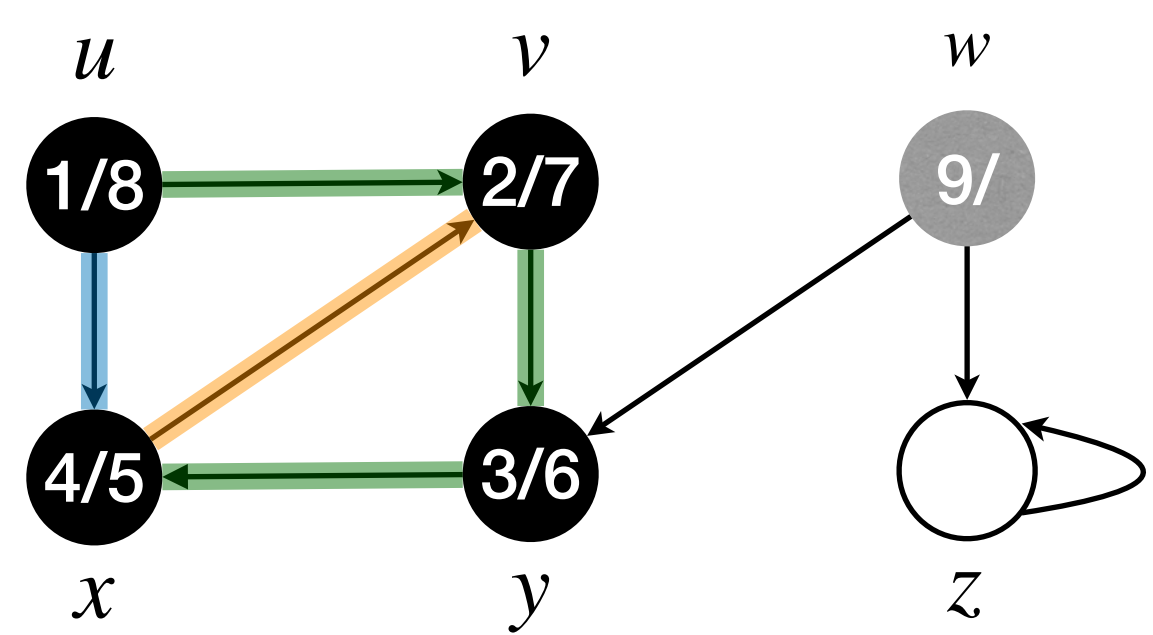
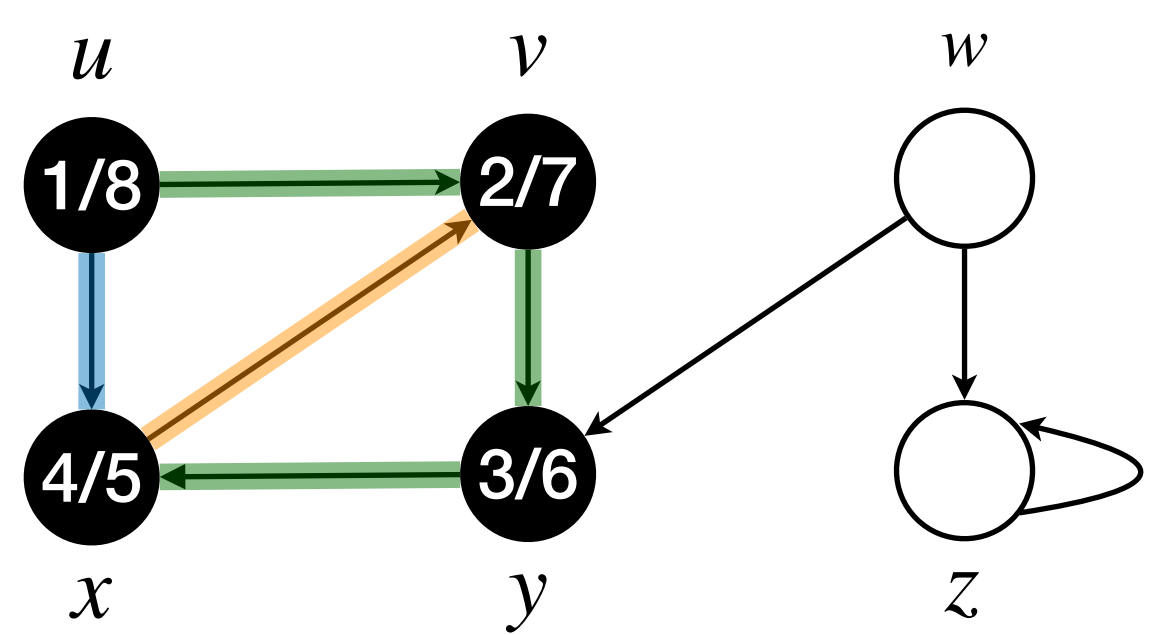
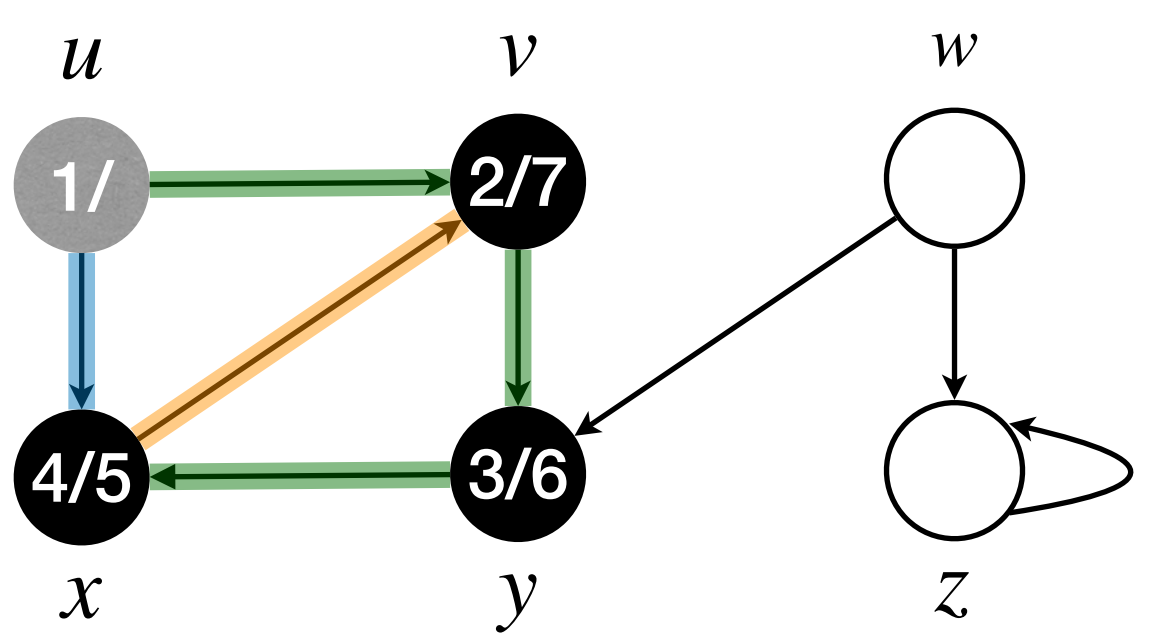
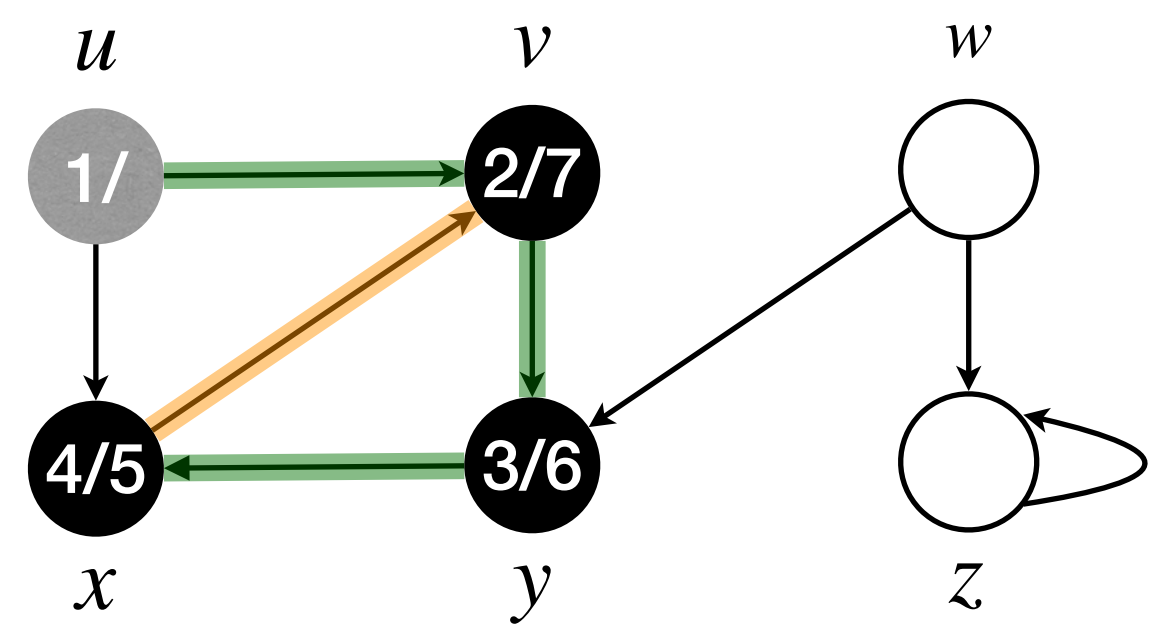
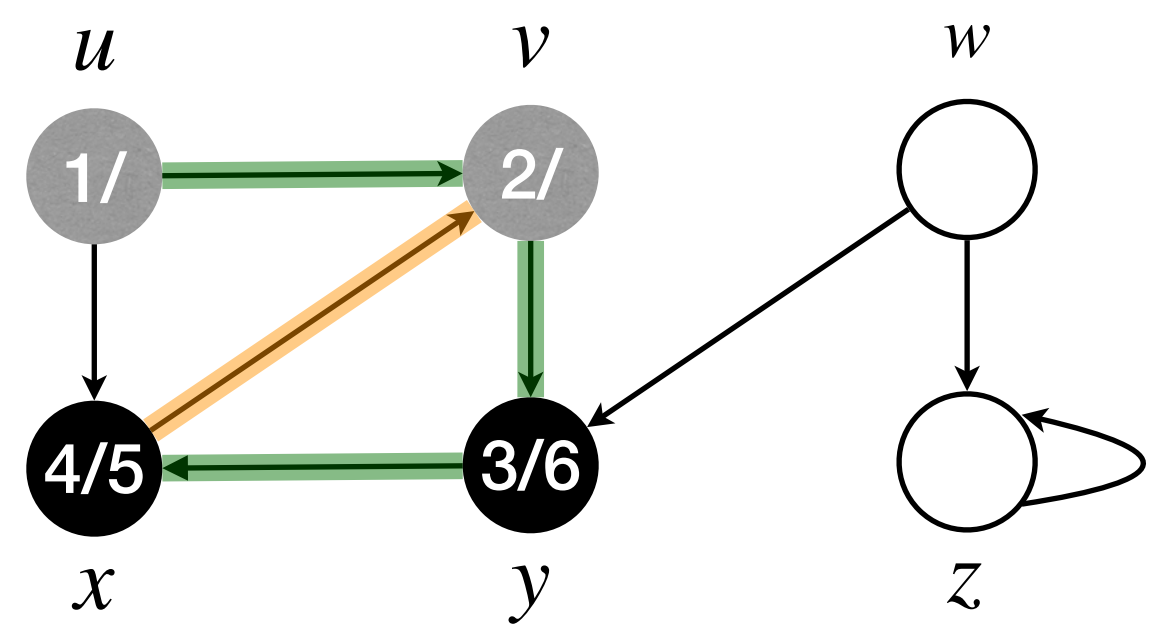
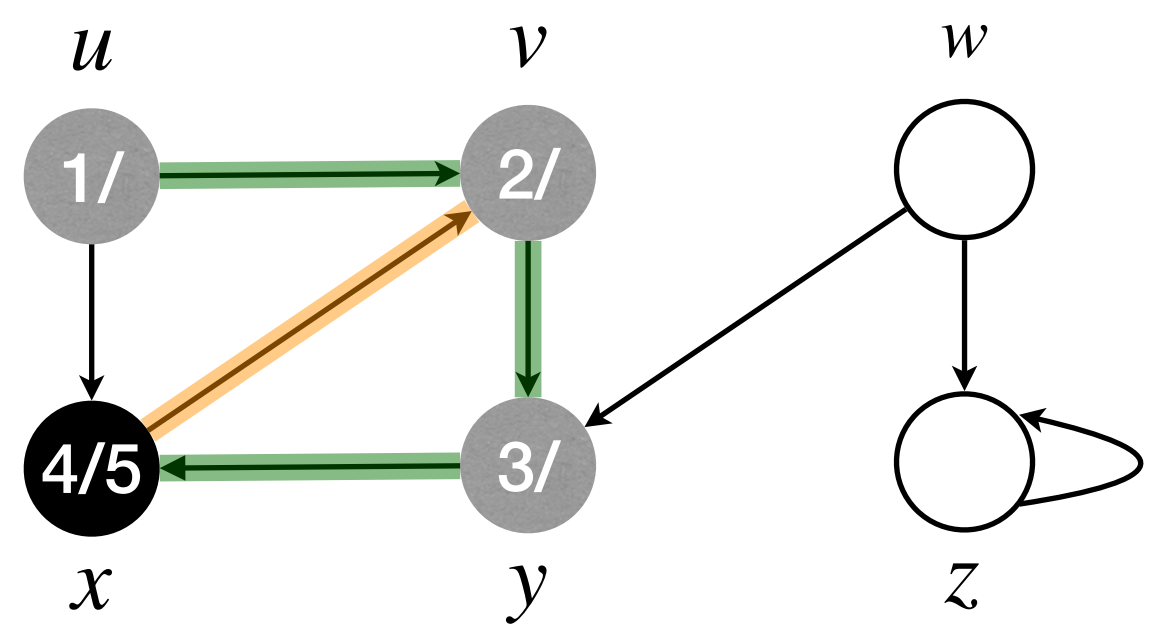
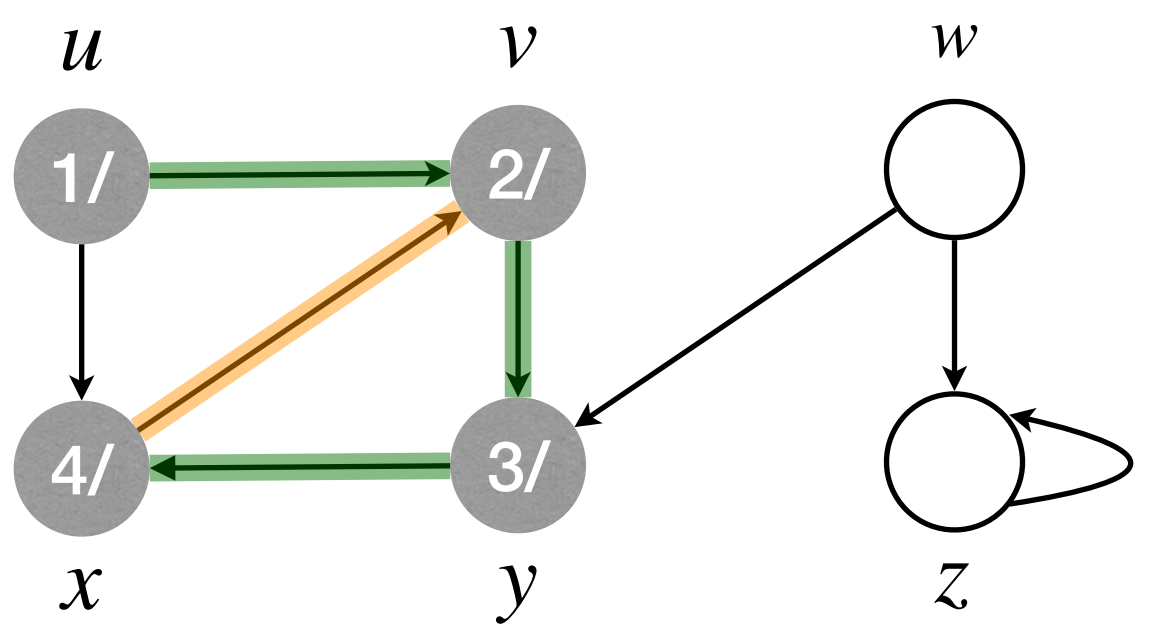
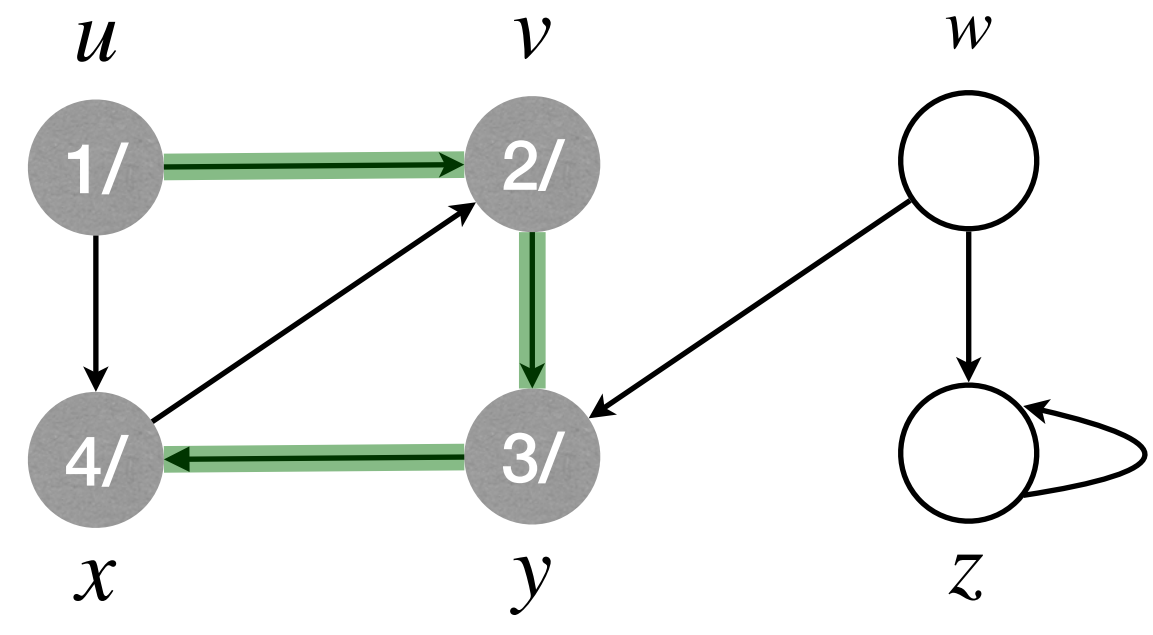
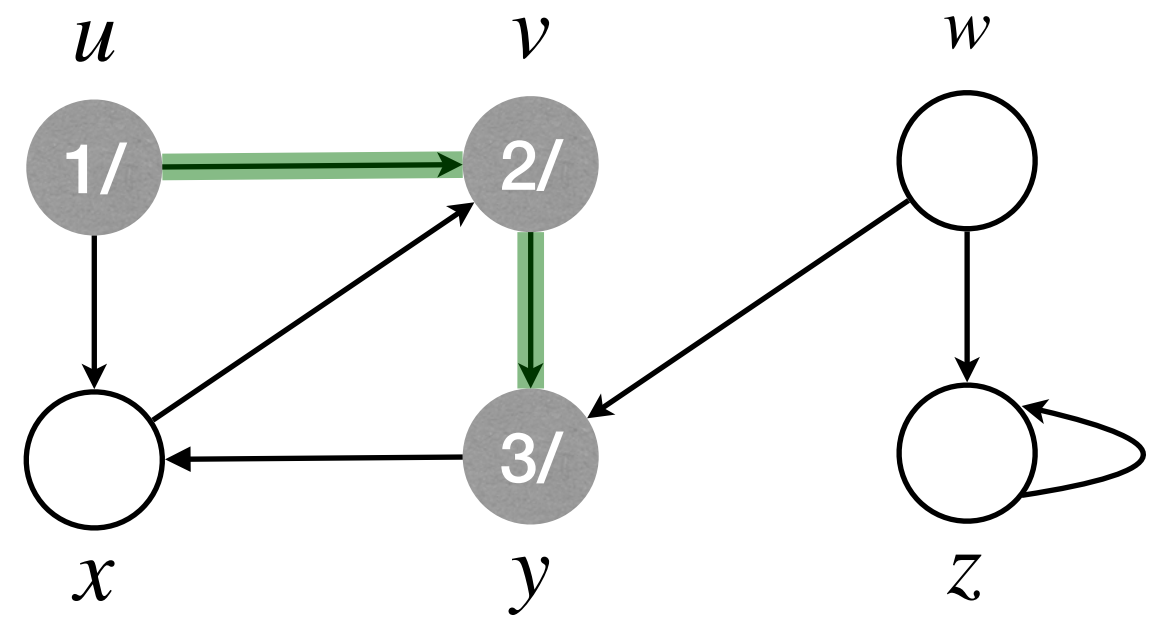
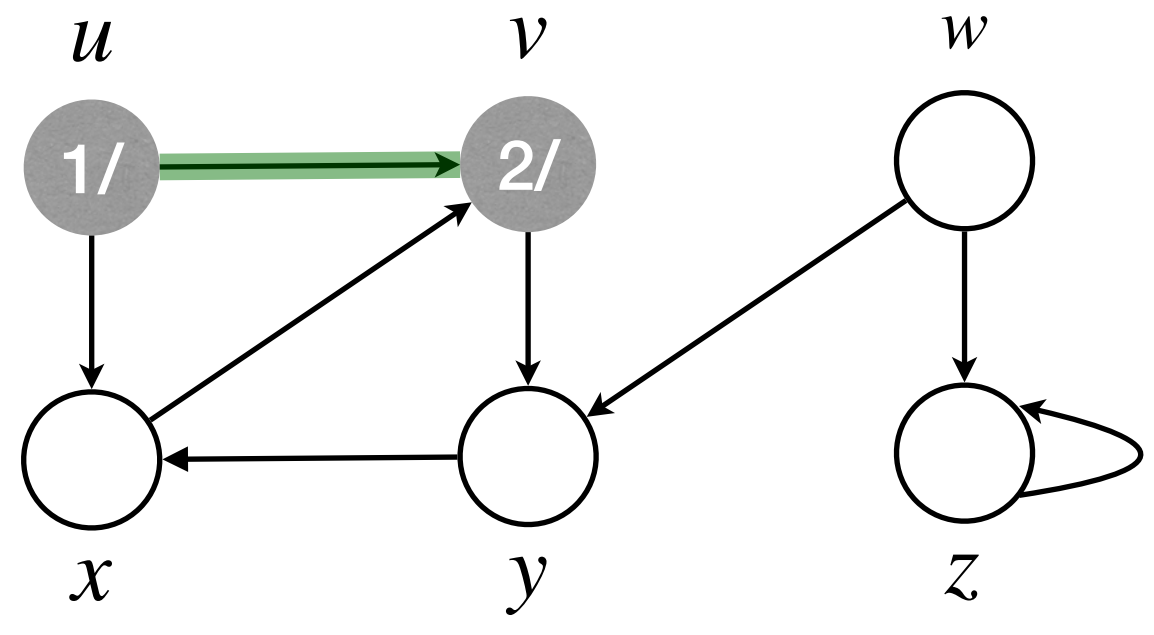
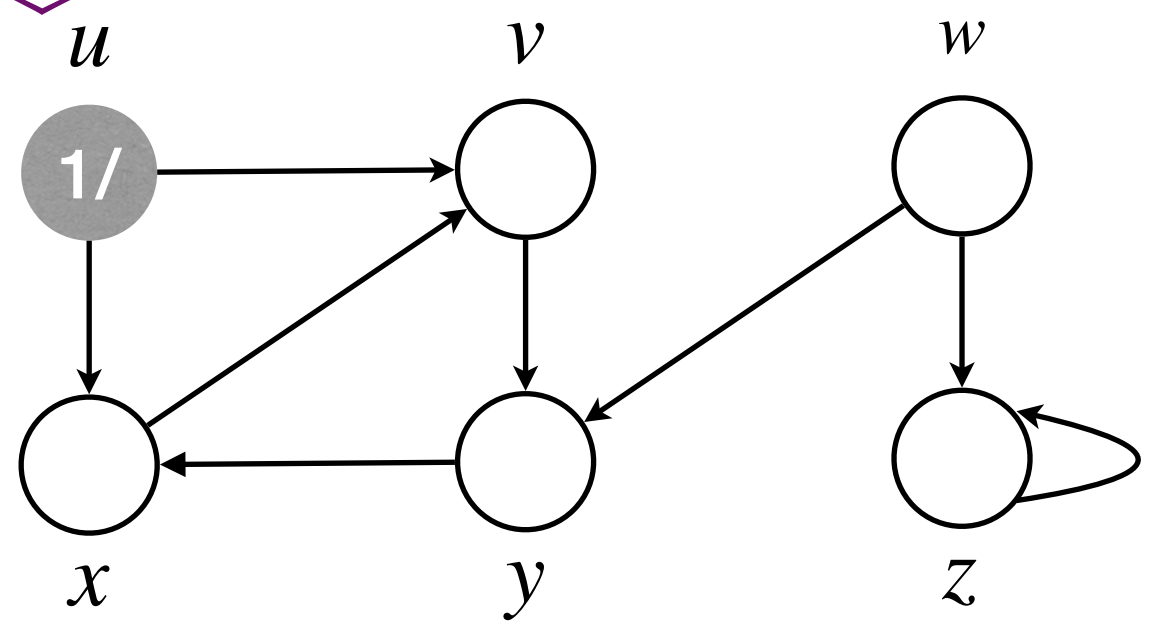
# Classification of edges

- DFS process classify edges of input graph into four types.
  - ▶ **Tree Edges**: Edges in the DFS forest.
  - ▶ **Back Edges**: Edges  $(u, v)$  connecting  $u$  to an ancestor  $v$  in a DFS tree.
  - ▶ **Forward Edges**: Non-tree edges  $(u, v)$  connecting  $u$  to a descendant  $v$  in a DFS tree.
  - ▶ **Cross Edges**: Other edges. (Connecting nodes in same DFS tree with no ancestor-descendant relation, or connecting nodes in different DFS trees.)





Tree Back Forward Cross

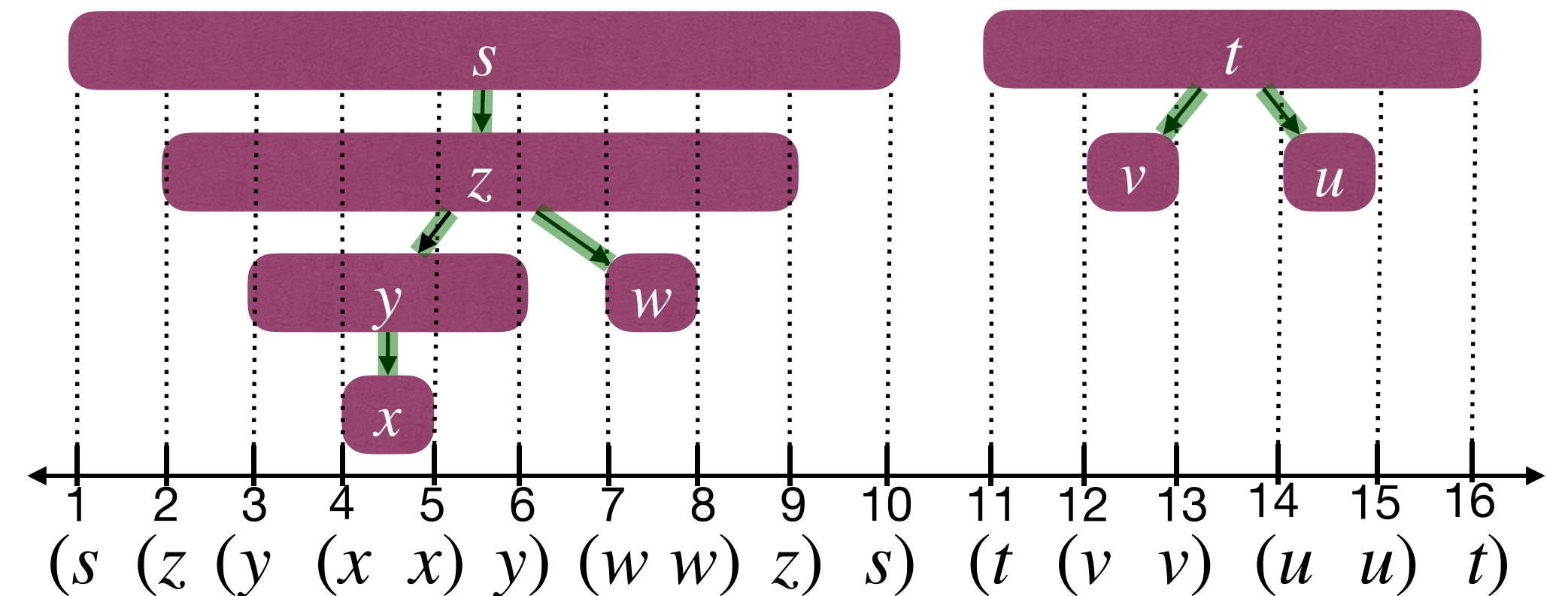
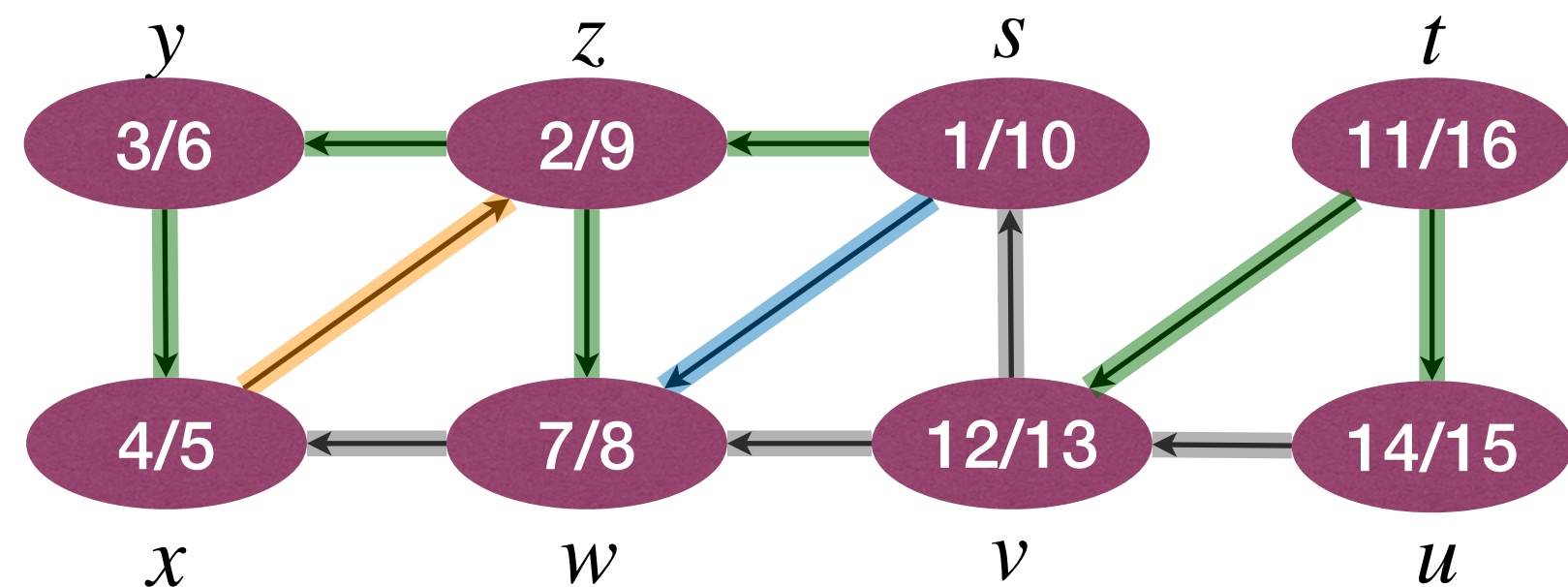




# Properties of DFS: Parenthesis Theorem

**Theorem:** Active intervals of two nodes are either: **(a)** entirely disjoint; or **(b)** one is entirely contained within another.

- For any two nodes  $u$  and  $v$ , exactly one of following holds:
  - (a)**  $[u.d, u.f]$  and  $[v.d, v.f]$  are disjoint, and  $u, v$  have no ancestor-descendant relation in the DFS forest;
  - (b)**  $[u.d, u.f] \subset [v.d, v.f]$ , and  $u$  is a descendant of  $v$  in a DFS tree;
  - (c)**  $[v.d, v.f] \subset [u.d, u.f]$ , and  $u$  is an ancestor of  $v$  in a DFS tree.





# Properties of DFS: Parenthesis Theorem

- **Proof:** Consider two nodes  $u$  and  $v$ . W.l.o.g., assume  $u.d < v.d$ .
- If  $v.d < u.f$ , then  $v$  is discovered (WHITE  $\rightarrow$  GRAY) while  $u$  is being processed (GRAY); and DFS will finish  $v$  first, before returning to  $u$ .
  - In this case,  $[v.d, v.f] \subset [u.d, u.f]$ , and  $u$  is an ancestor of  $v$ .
- If  $v.d > u.f$ , then obviously  $u.d < u.f < v.d < v.f$ ; and DFS has finished exploring  $u$  (BLACK), before  $v$  is discovered (WHITE  $\rightarrow$  GRAY).
  - In this case,  $[u.d, u.f]$  and  $[v.d, v.f]$  are disjoint, and  $u, v$  have no ancestor-descendant relation.

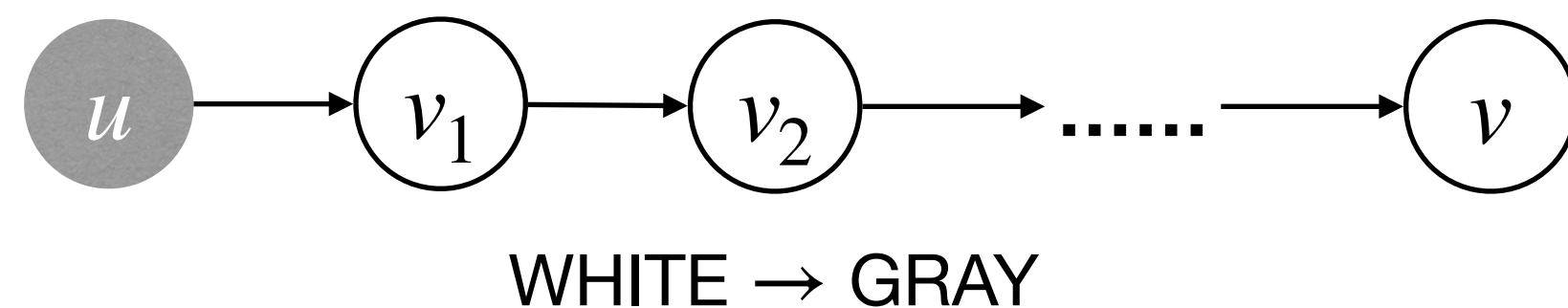




# Properties of DFS: White-path Theorem

**Theorem** In the DFS forest,  $v$  is a descendant of  $u$  iff when  $u$  is discovered, there is a path in the graph from  $u$  to  $v$  containing only WHITE nodes.

- Proof of  $[ \implies ]$ 
  - ▶ **Claim:** If  $v$  is a proper descendant of  $u$ , then  $v$  is WHITE when  $u$  is discovered.
    - Since if  $v$  is a proper descendant of  $u$ , then  $u.d < v.d$ .
  - ▶ For any node along the path from  $u$  to  $v$  in the DFS forest, above claim holds.
  - ▶ Therefore,  $[ \implies ]$  direction of the theorem holds.





# Properties of DFS: White-path Theorem

- Proof of  $[\Leftarrow]$ :

- ▶ W.l.o.g., assume  $v$  is the *first* node along the path that does *not* become a descendant of  $u$ .

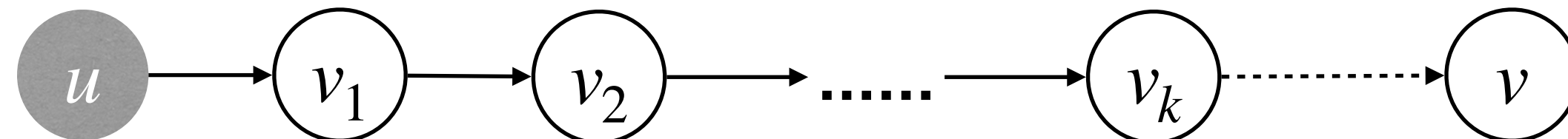
- ▶ So we have  $[v_k.d, v_k.f] \subset [u.d, u.f]$ .

Depth-first search until all the edges of  $v_k$  is explored!

- ▶ But  $v$  is discovered after  $u$  is discovered, and must **before  $v_k$  is finished.**

- ▶ So we have  $u.d < v.d < v_k.f \leq u.f$ .

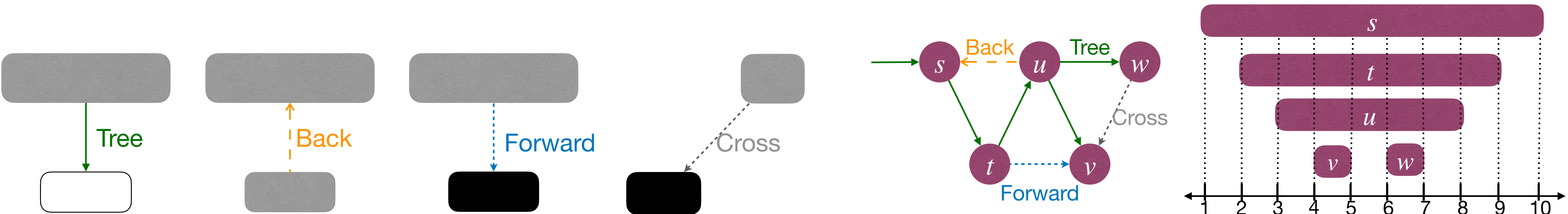
- ▶ Then it must be  $[v.d, v.f] \subset [u.d, u.f]$ , implying  $v$  is a descendant of  $u$ .





# Properties of DFS: Classification of edges

- Determine  $(u, v)$  type by color of  $v$  during DFS execution.
  - ▶ **Tree Edges:** Edges in the DFS forest. Node  $v$  is WHITE
  - ▶ **Back Edges:** Edges  $(u, v)$  connecting  $u$  to an ancestor  $v$  in a DFS tree. Node  $v$  is GRAY
  - ▶ **Forward Edges:** Non-tree edges  $(u, v)$  connecting  $u$  to a descendant  $v$  in a DFS tree. Node  $v$  is BLACK
  - ▶ **Cross Edges:** Other edges. (Connecting nodes in same DFS tree with no ancestor-descendant relation, or connecting nodes in different DFS trees.) Node  $v$  is BLACK






# Properties of DFS: Types of edges in undirected graphs

- Will all four types of edges appear in DFS of undirected graphs?

**Theorem** In DFS of an *undirected* graph  $G$ , every edge of  $G$  is either a ***tree edge*** or a ***back edge***.

- **Proof:**
  - ▶ Consider an arbitrary edge  $(u, v)$ . W.l.o.g., assume  $u.d < v.d$ .
  - ▶ Edge  $(u, v)$  must be explored while  $u$  is GRAY. 
  - ▶ Consider the first time the edge  $(u, v)$  is explored.



# Properties of DFS: Types of edges in undirected graphs

- Proof (continued):
  - ▶ If the direction is  $u \rightarrow v$ . Then,  $v$  must be WHITE by then, for otherwise the edge would have been explored from direction  $v \rightarrow u$  earlier.
    - In such case, the edge  $(u, v)$  becomes a **tree edge**.
  - ▶ If the direction is  $v \rightarrow u$ . Then, the edge is “GRAY  $\rightarrow$  GRAY”.
    - In such case, the edge  $(u, v)$  becomes a **back edge**.



# DFS, BFS, and others...

## DFSIterSkeleton(G, s):

*Stack*  $Q$

$Q.push(s)$

**while**  $!Q.empty()$

$u := Q.pop()$

**if**  $!u.visited$

$u.visited := True$

**for each** edge  $(u, v)$  **in**  $E$

$Q.push(v)$

## BFSSkeletonAlt(G, s):

*FIFOQueue*  $Q$

$Q.enqueue(s)$

**while**  $!Q.empty()$

$u := Q.dequeue()$

**if**  $!u.visited$

$u.visited := True$

**for each** edge  $(u, v)$  **in**  $E$

$Q.enqueue(v)$

## GraphExploreSkeleton(G, s):

*GenericQueue*  $Q$

$Q.add(s)$

**while**  $!Q.empty()$

$u := Q.remove()$

**if**  $!u.visited$

$u.visited := True$

**for each** edge  $(u, v)$  **in**  $E$

$Q.add(v)$

Other queuing disciplines lead to more interesting algorithms!



# Further reading

- [CLRS] Ch.22 (22.1-22.3)

