

Algorithms & Data Structures I

CSC 225

Ali Mashreghi

Fall 2018



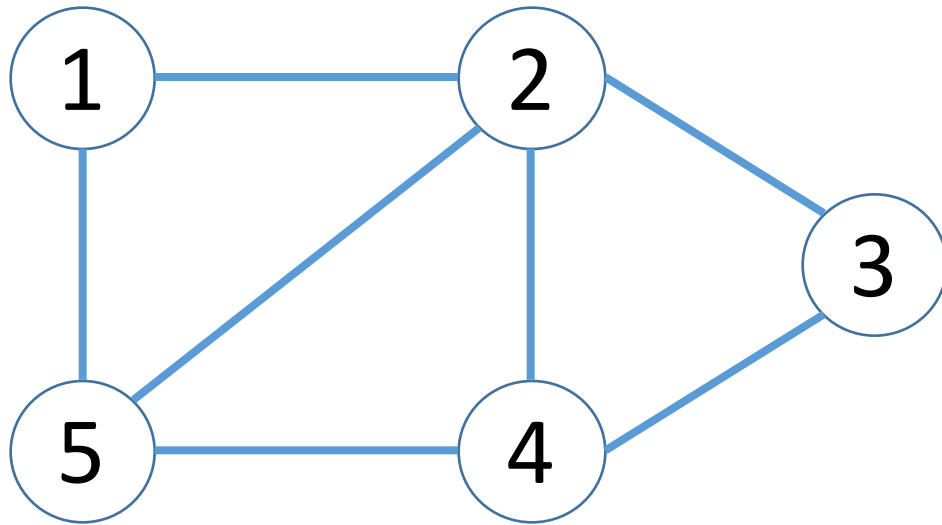
Department of Computer Science, University of Victoria

Graph representation

- We need a way to represent graphs in computers.
- There are two main ways for representing graphs
 1. Adjacency Matrices
 2. Adjacency Lists
- We can choose between these two based on the graphs we are working with.

Adjacency Matrix

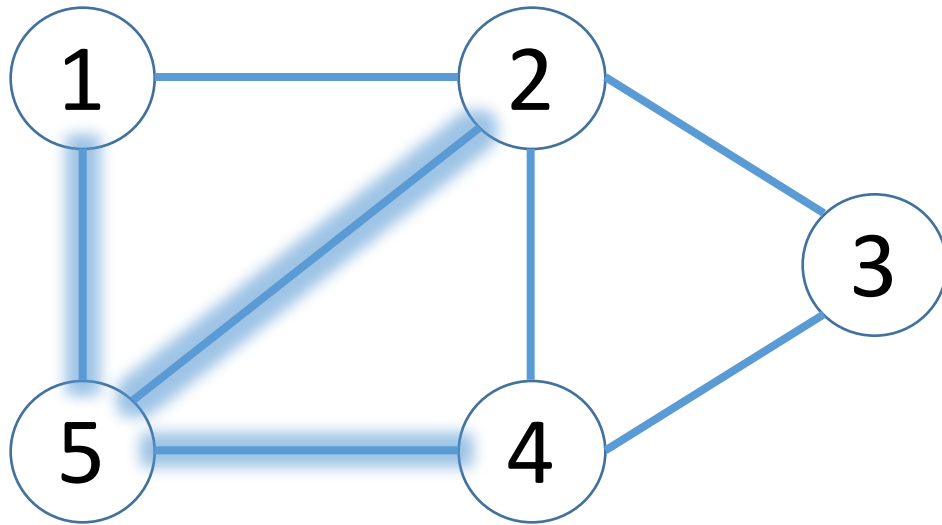
- A Boolean $n \times n$ matrix A represents the graph



	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

Adjacency Matrix

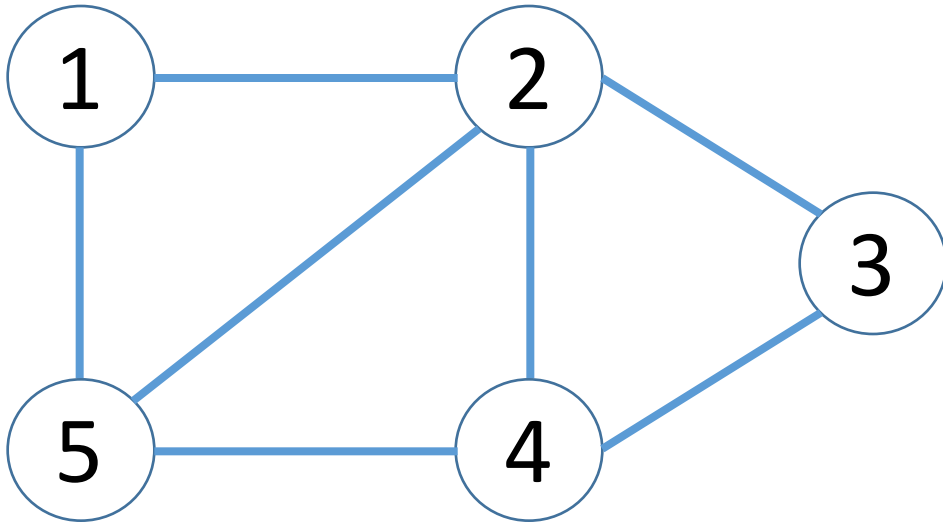
- $A[i][j]$ is 1 if only if node i and j are connected



	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

Properties for simple graphs

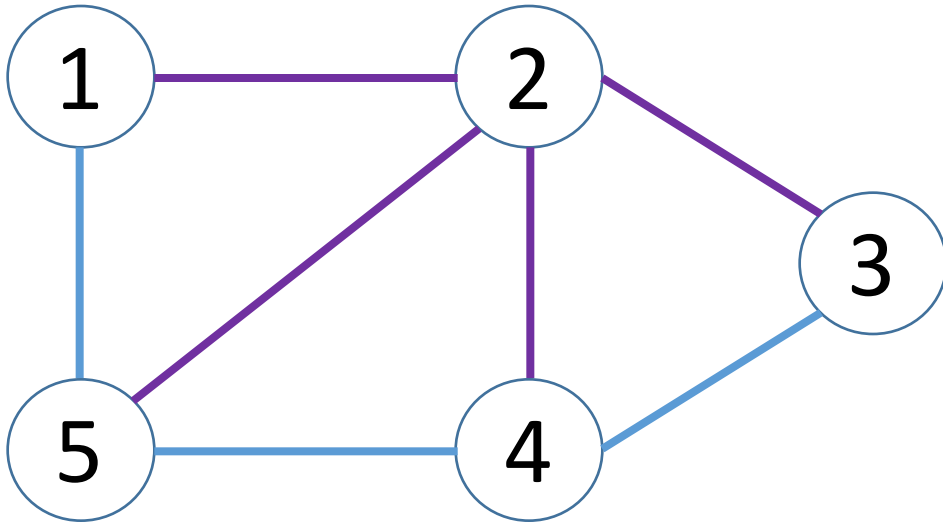
- All entries on the **diagonal** are 0 since there is no self-loop



	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

Properties for simple graphs

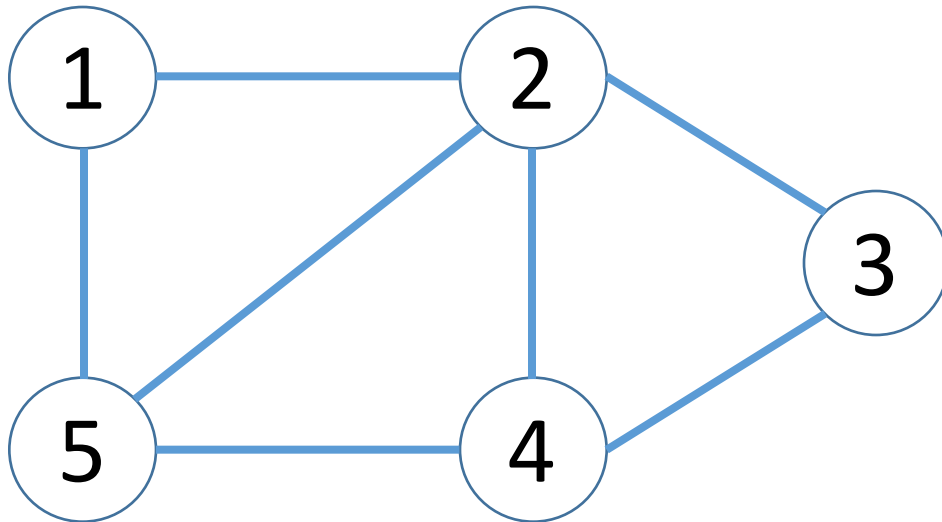
- Number of 1's in row/column i is equal to the degree of node i



	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

Properties for simple graphs

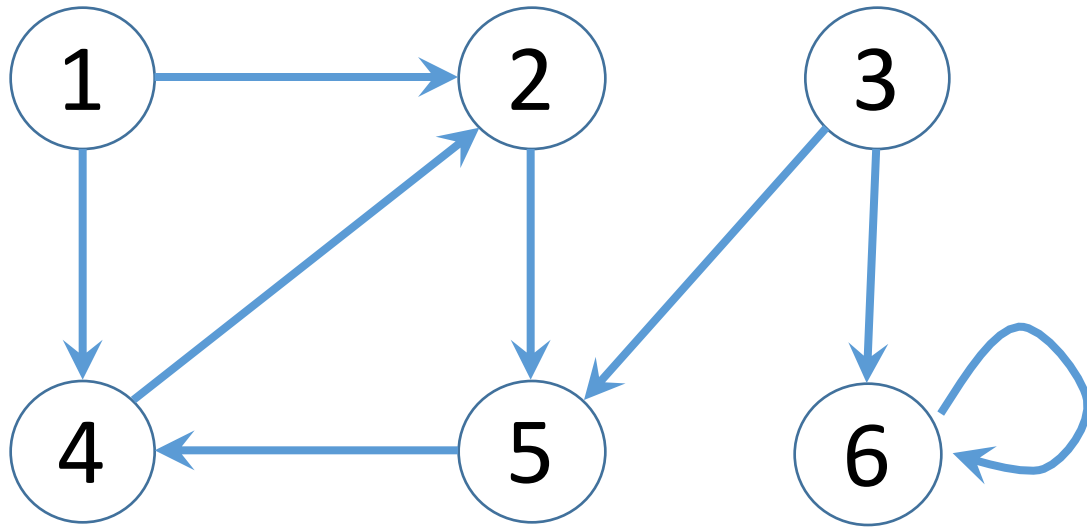
- A is symmetric, because $A[i][j]$ and $A[j][i]$ are either both 1 or both 0.



	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

Adjacency Matrix

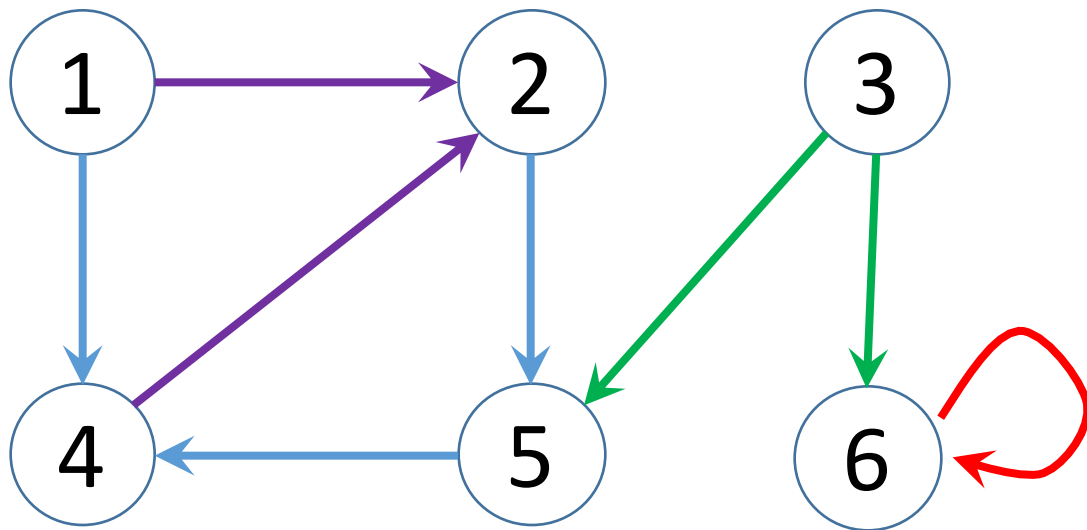
- For directed graphs $A[i][j] = 1$ iff **i has an edge to j**



	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

Adjacency Matrix

- Number of 1's in row x is the out-degree(x). In a column, it's in-degree(x)
- If there is a self-loop, diagonal entries could be 1
- The matrix is not necessarily symmetric



	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

Memory

- **Question:** How much memory do we need to represent a graph G , with n nodes and m edges?

	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

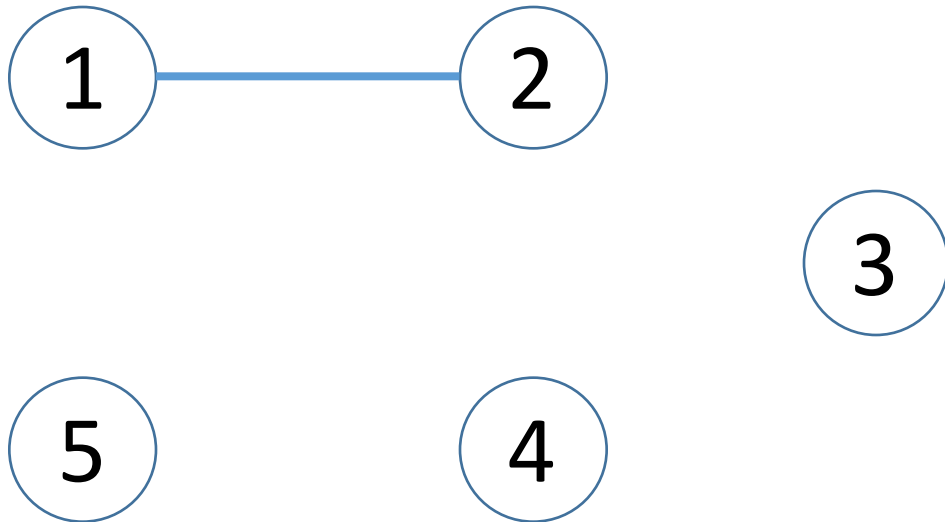
Memory

- **Question:** How much memory do we need to represent a graph G , with n nodes and m edges?
- **Answer:** $\Theta(n^2)$ since there are n^2 entries in the matrix. The amount of memory does not depend on the number of edges.

	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

Memory

- For a graph like this most of the entries are 0 which is a waste of memory.



	1	2	3	4	5
1	0	1	0	0	0
2	1	0	0	0	0
3	0	0	0	0	0
4	0	0	0	0	0
5	0	0	0	0	0

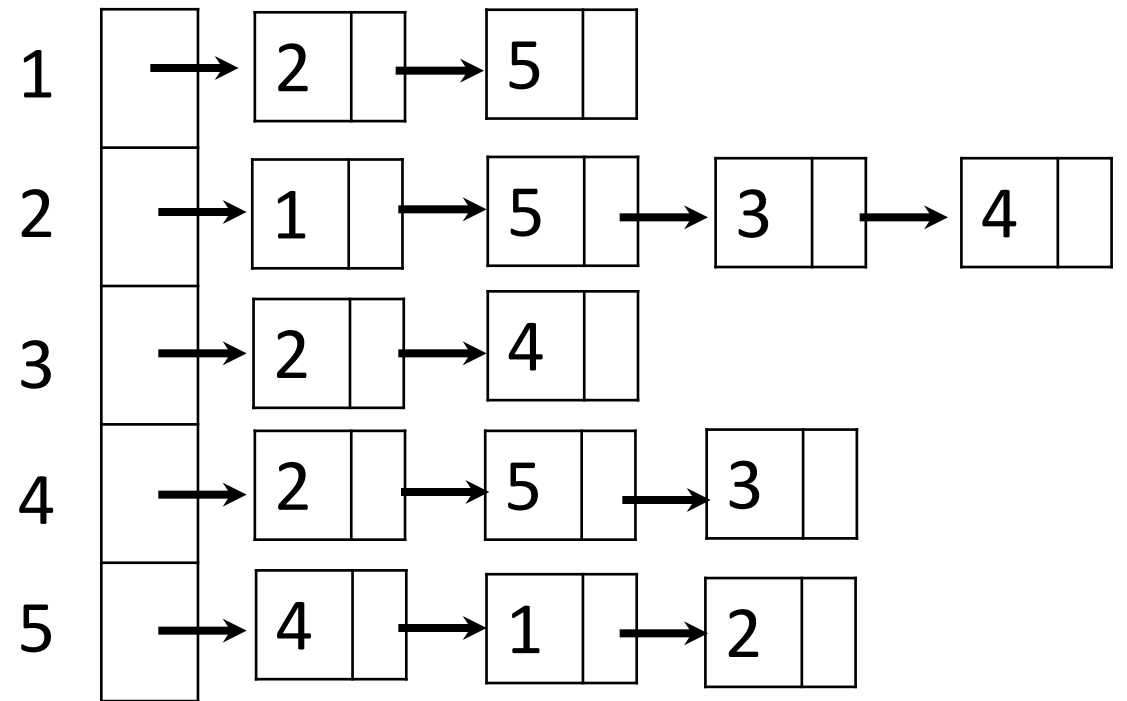
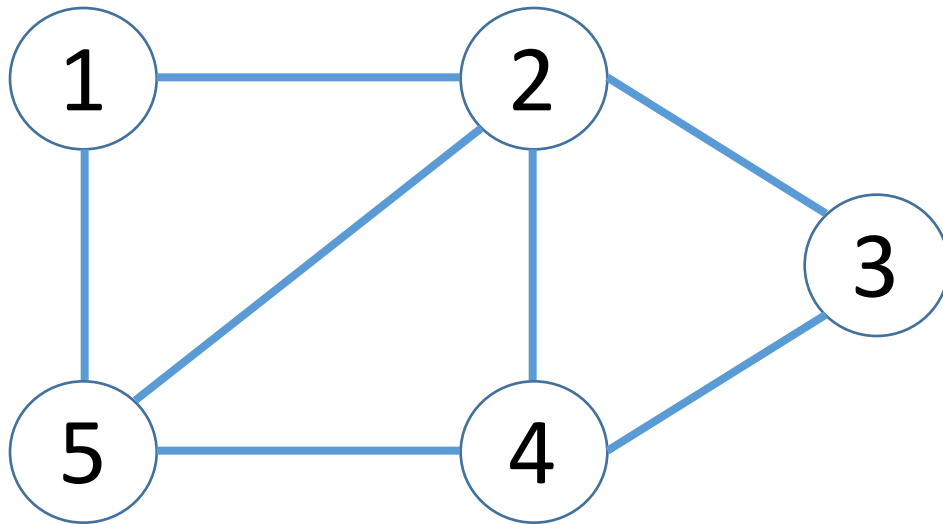
Memory

- If the graph does not have a lot of edges (much less than $\Theta(n^2)$), it's called a **sparse graph**.
- Similarly, if the number of edges is close to $\Theta(n^2)$, it's called **dense**.
- An adjacency matrix is **good** for representing **dense graphs**.

	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

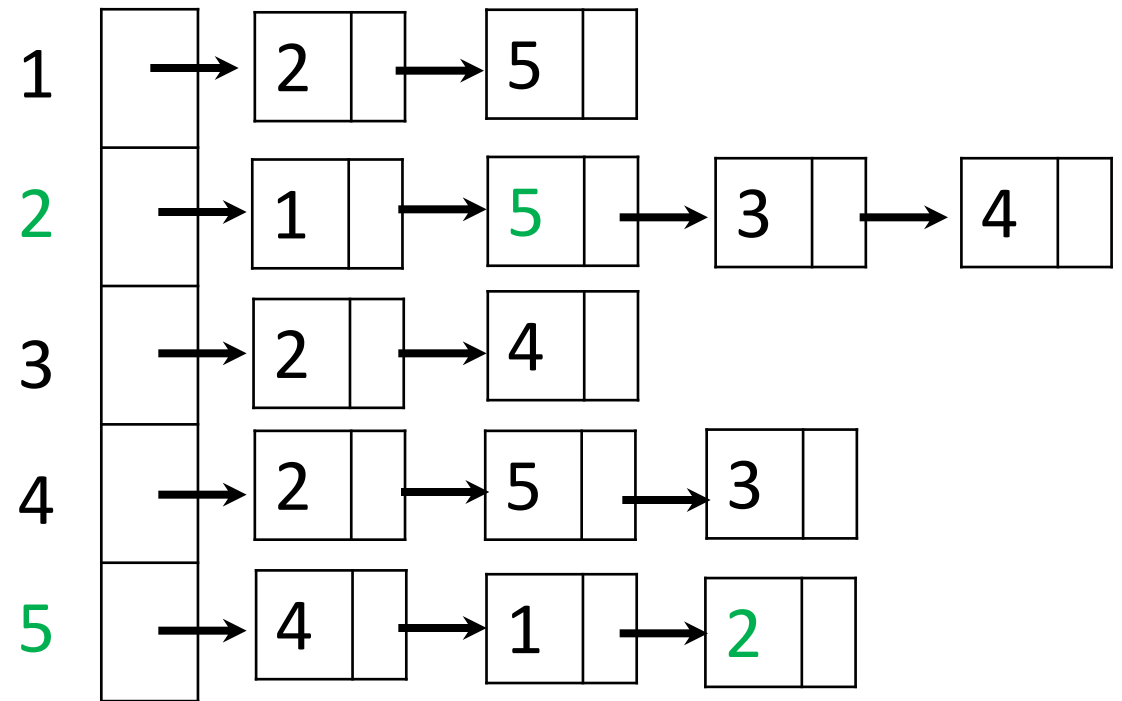
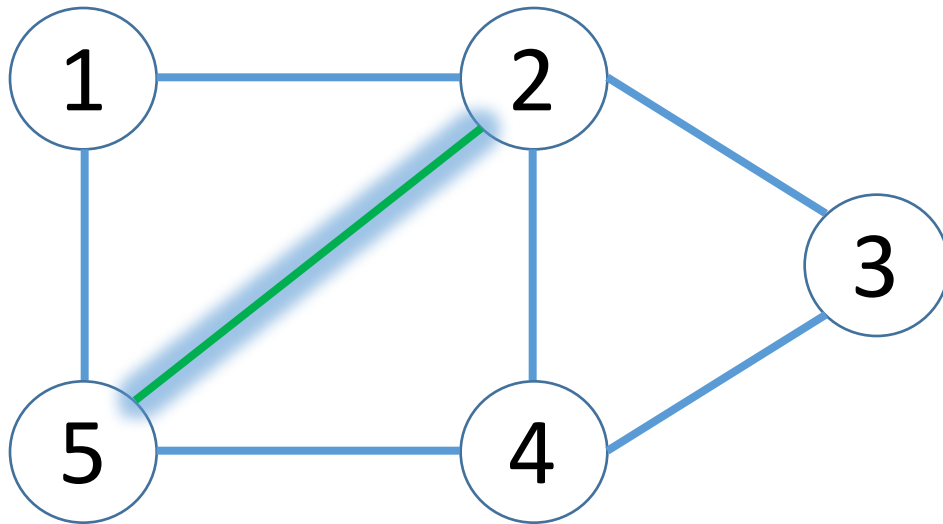
Adjacency list

- Another way is to have the list of neighbors for each node, as an array of lists. Usually, we name this array of lists *Adj*.



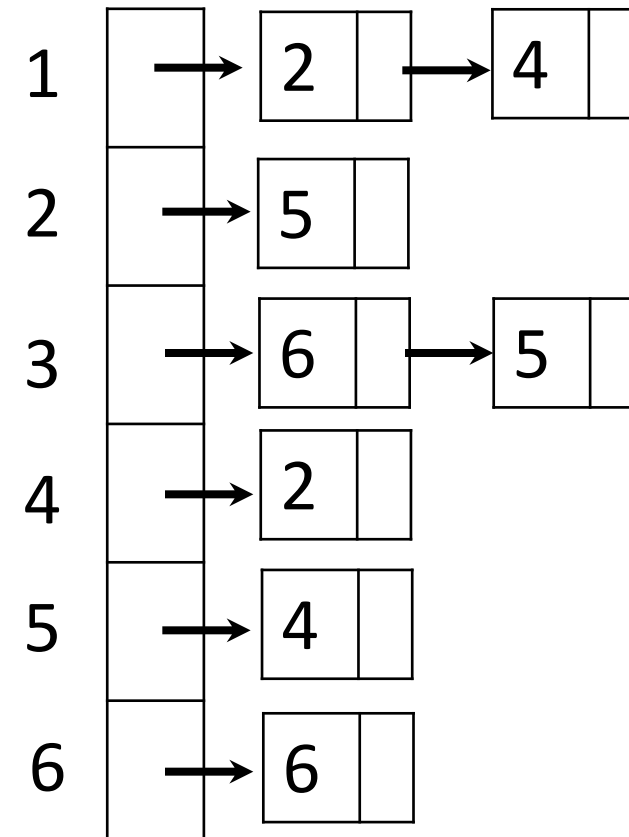
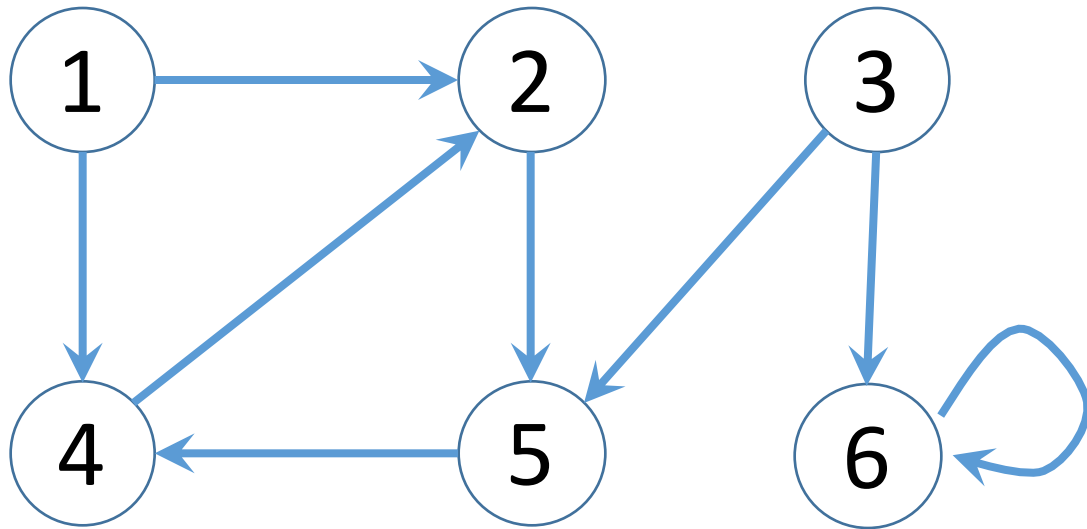
Adjacency list

- For an undirected graph, both endpoints of an edge should appear in each other's lists



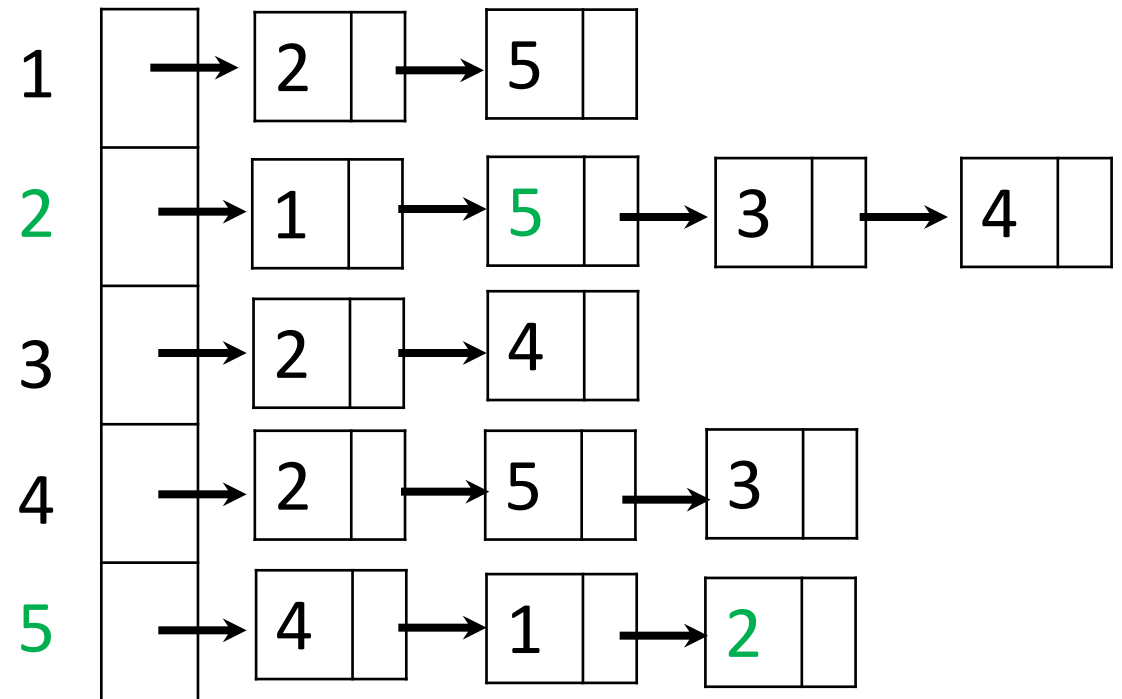
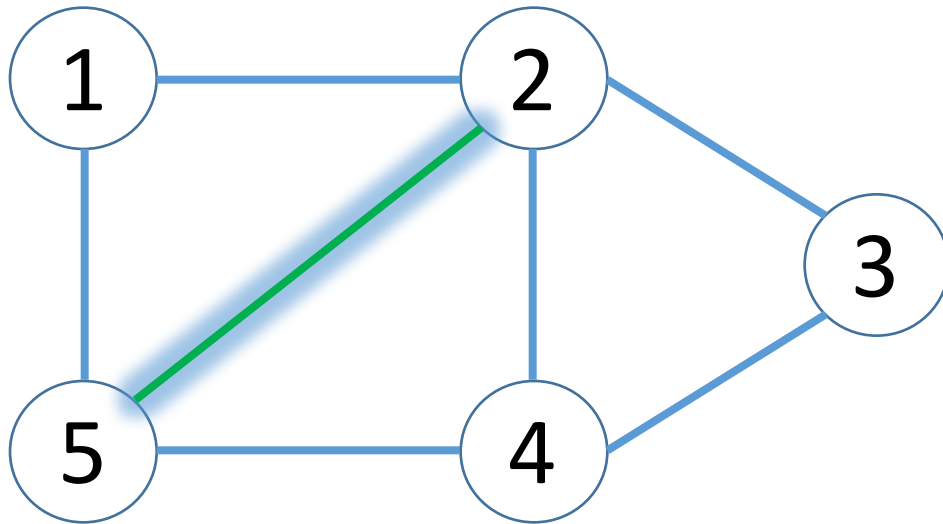
Adjacency list

- For a directed graph, v appears in u 's list if the edge (u, v) is present.



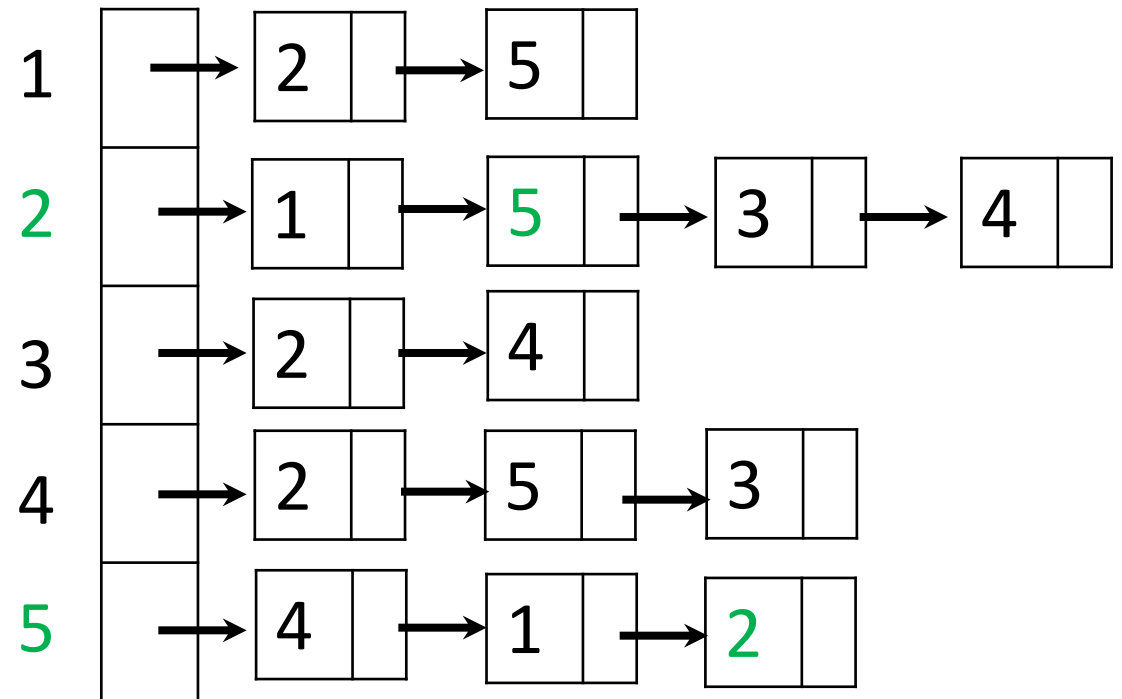
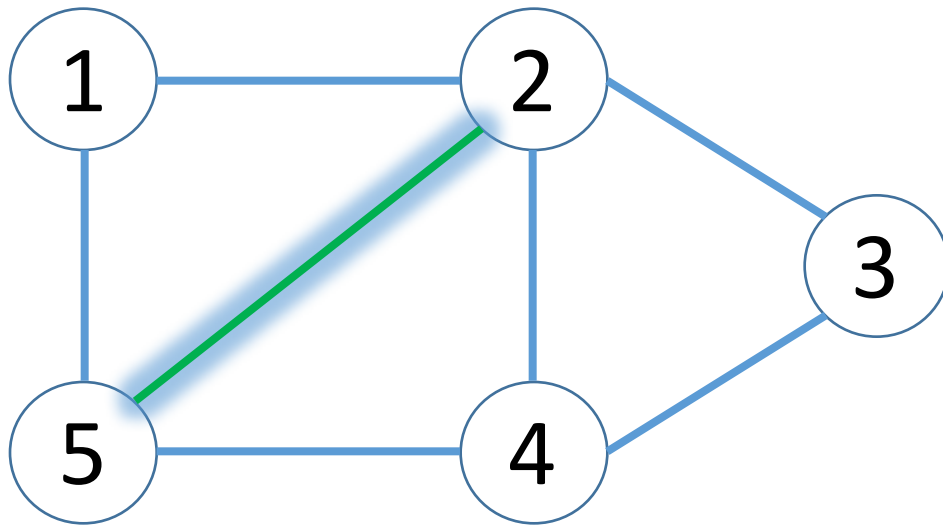
Memory

- **Question:** How much memory do we need to represent a graph G , with n nodes and m edges?



Memory

- **Answer:** $\Theta(n + m)$. We need at least n empty lists. For each edge in an undirected graph we need 2 nodes in the lists, i.e. $2m$ overall. For directed graphs we need m nodes.

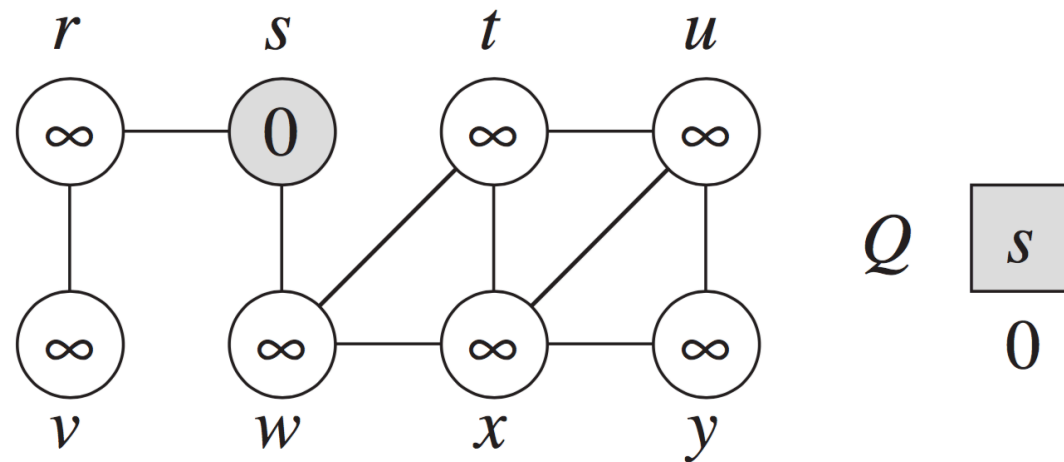


Exploring a graph

- We usually want to start from a given node in the graph and **explore** all nodes. (e.g. graph of a social network)
- We assume that the graph is given to us as an adjacency list.
- There are two systematic ways to explore a graph:
 1. Breadth First Search (BFS)
 2. Depth First Search (DFS)

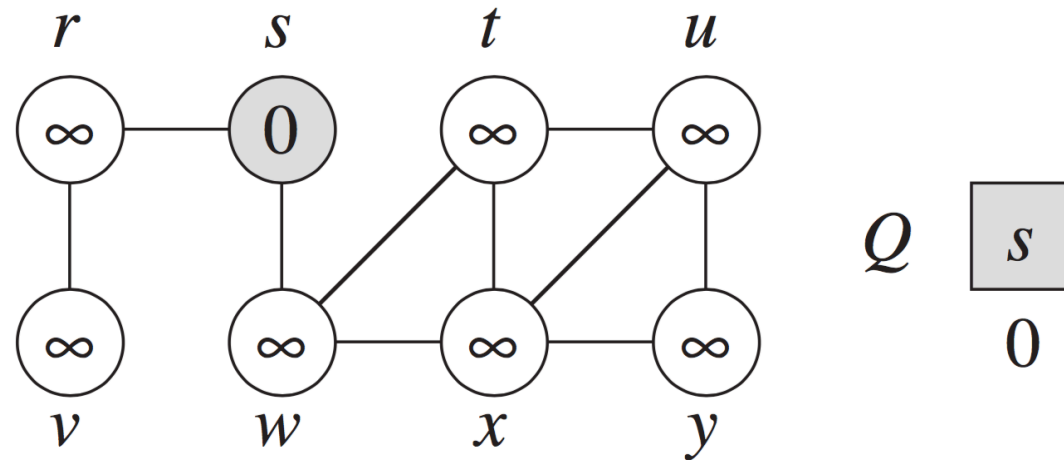
Breadth First Search

- The BFS algorithm explores all nodes reachable from a given **source node s** .
- In each step, the algorithm discovers a **new layer** of nodes.



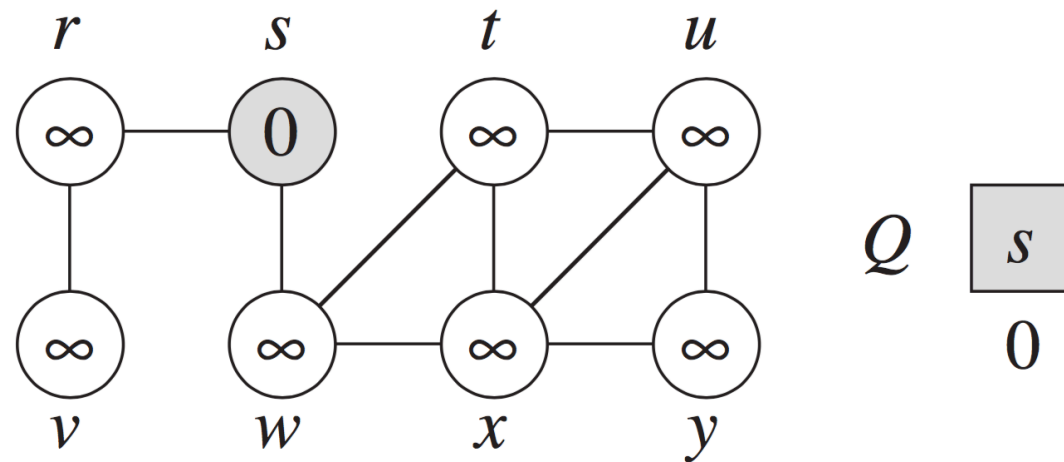
Breadth First Search

- BFS also computes the shortest distance between s , and all other nodes.
- **Initially, only s** is discovered and its distance from s is 0. All other nodes, are undiscovered and at distance ∞ from s .



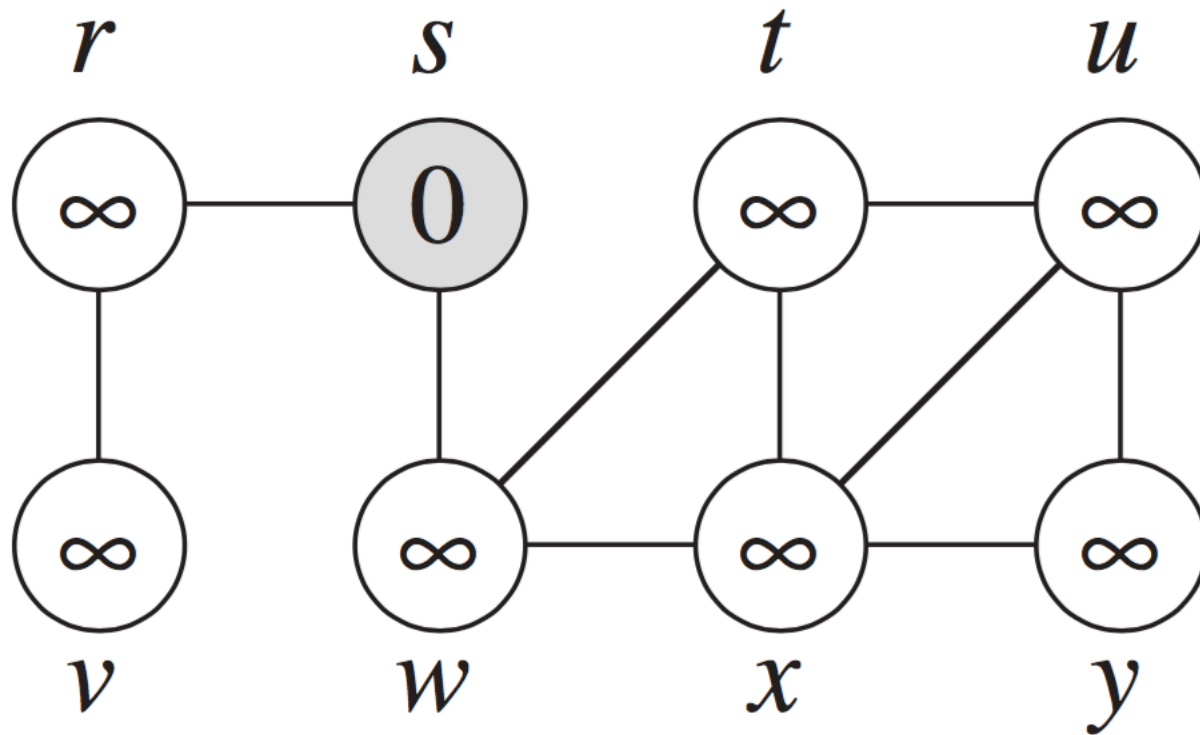
Breadth First Search

- We use a queue Q , to see which nodes may still lead to discovering newer nodes.



Breadth First Search

- **Undiscovered** nodes are **white**.
- Nodes that are **being discovered** are **gray**.



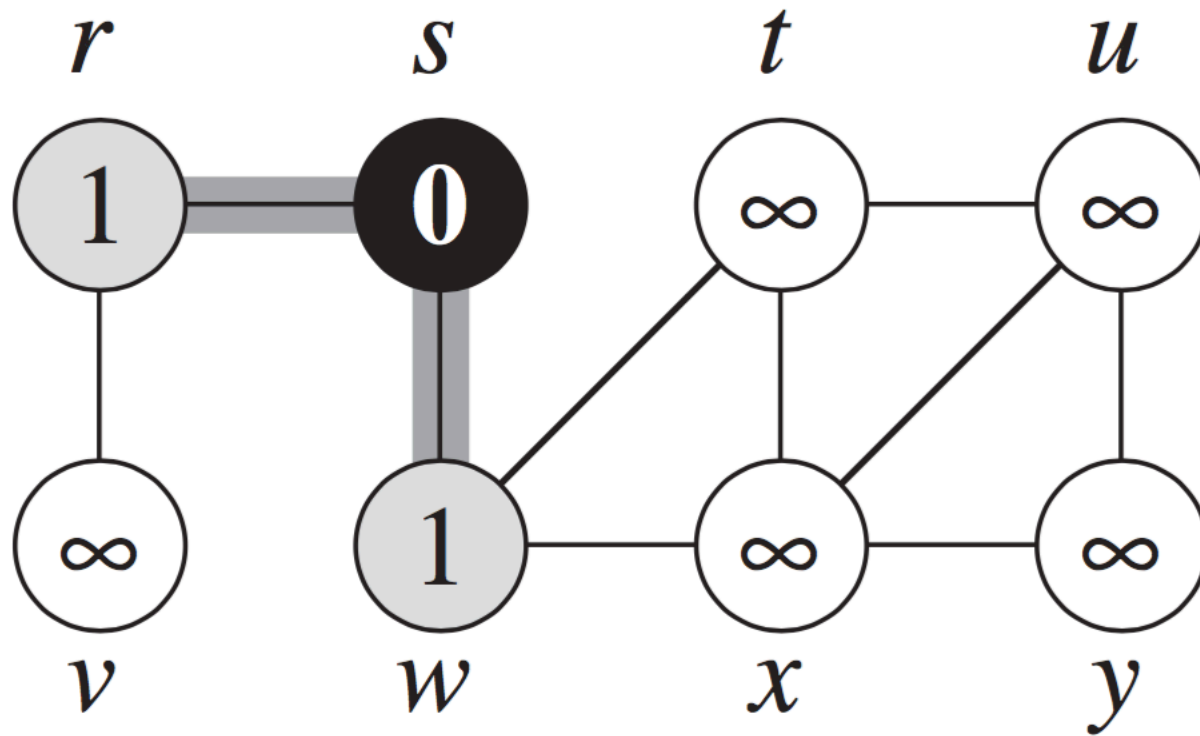
Q

s

 d 0
These are distances from s .

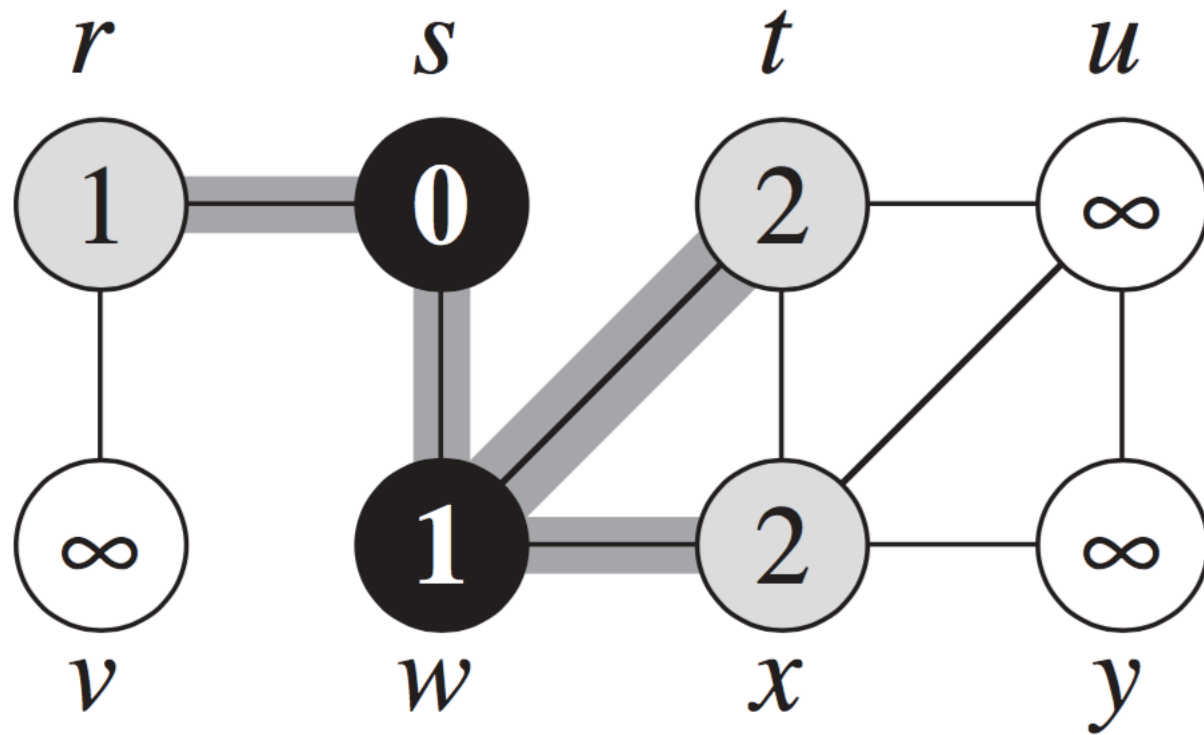
Breadth First Search

- A **fully discovered** node becomes **black**. This is when we have added all of its **undiscovered neighbors** to Q



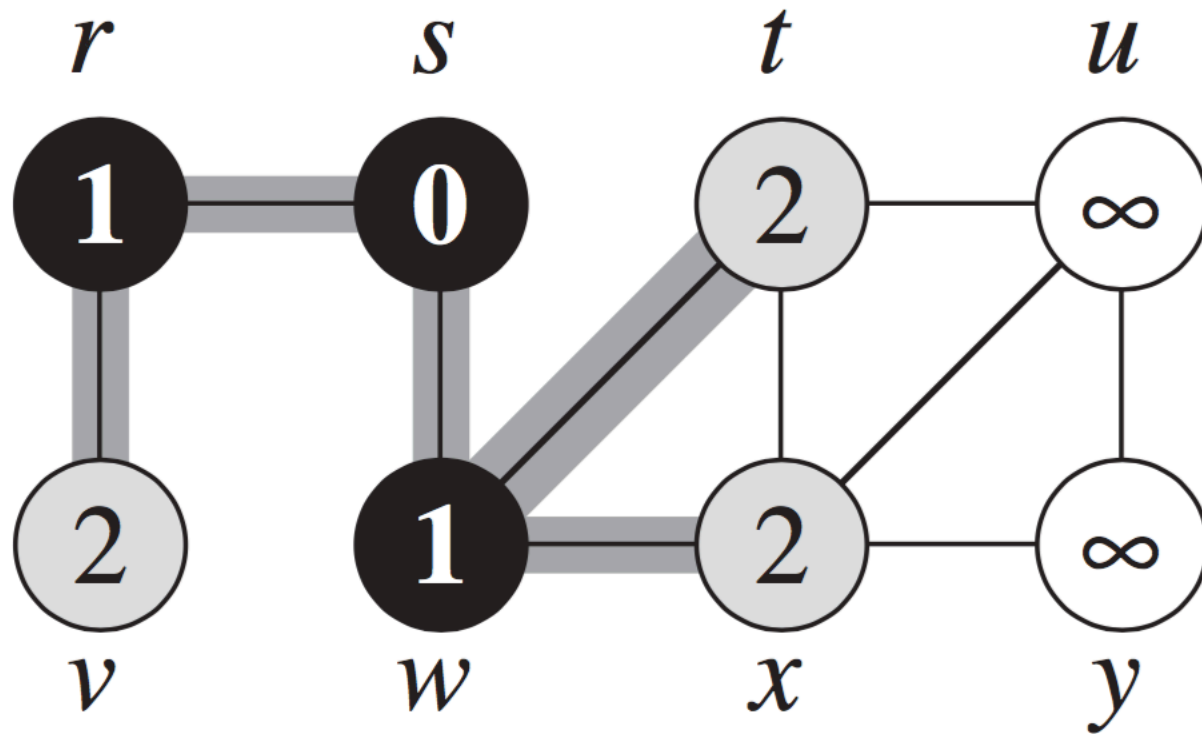
Q	w	r
d	1	1

Breadth First Search



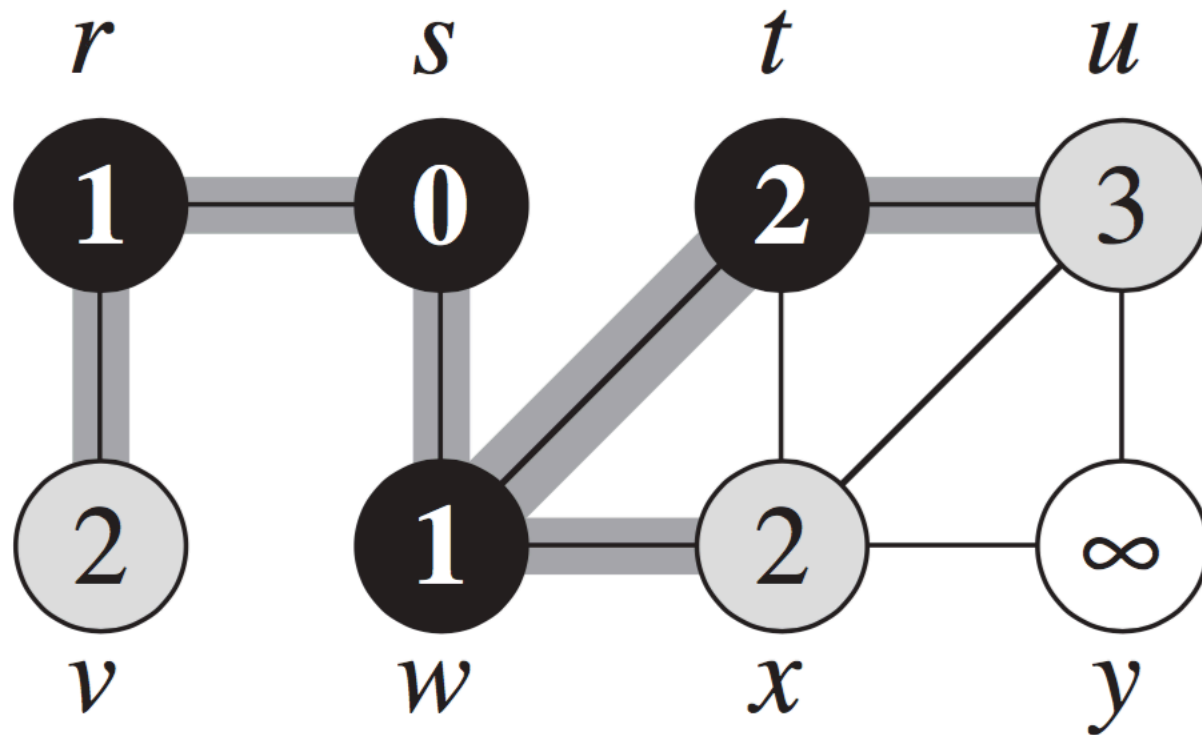
Q	r	t	x
d	1	2	2

Breadth First Search



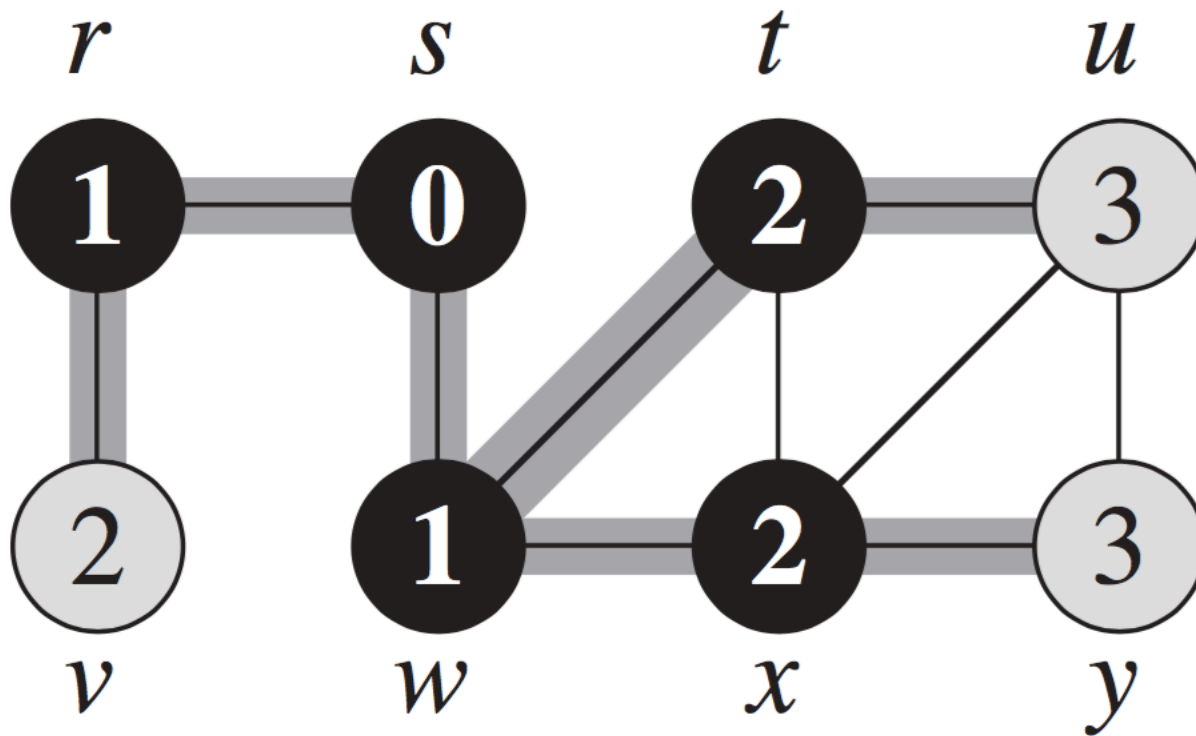
Q	t	x	v
d	2	2	2

Breadth First Search



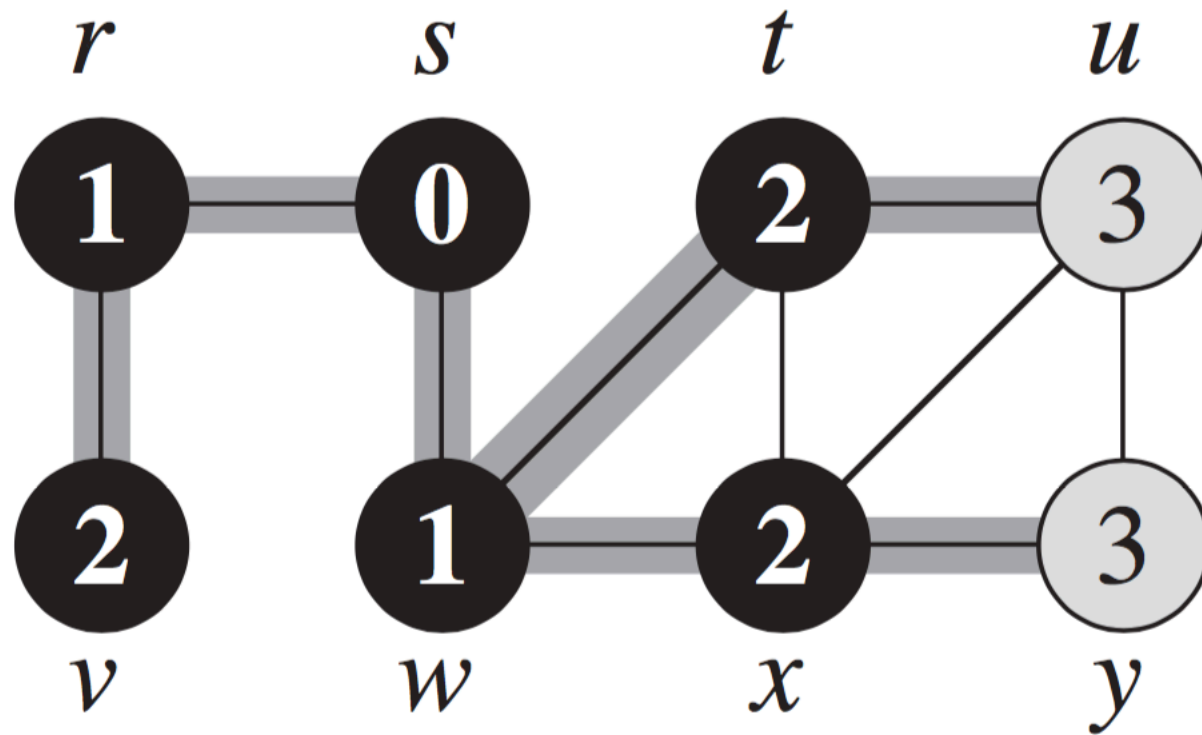
Q	x	v	u
d	2	2	3

Breadth First Search



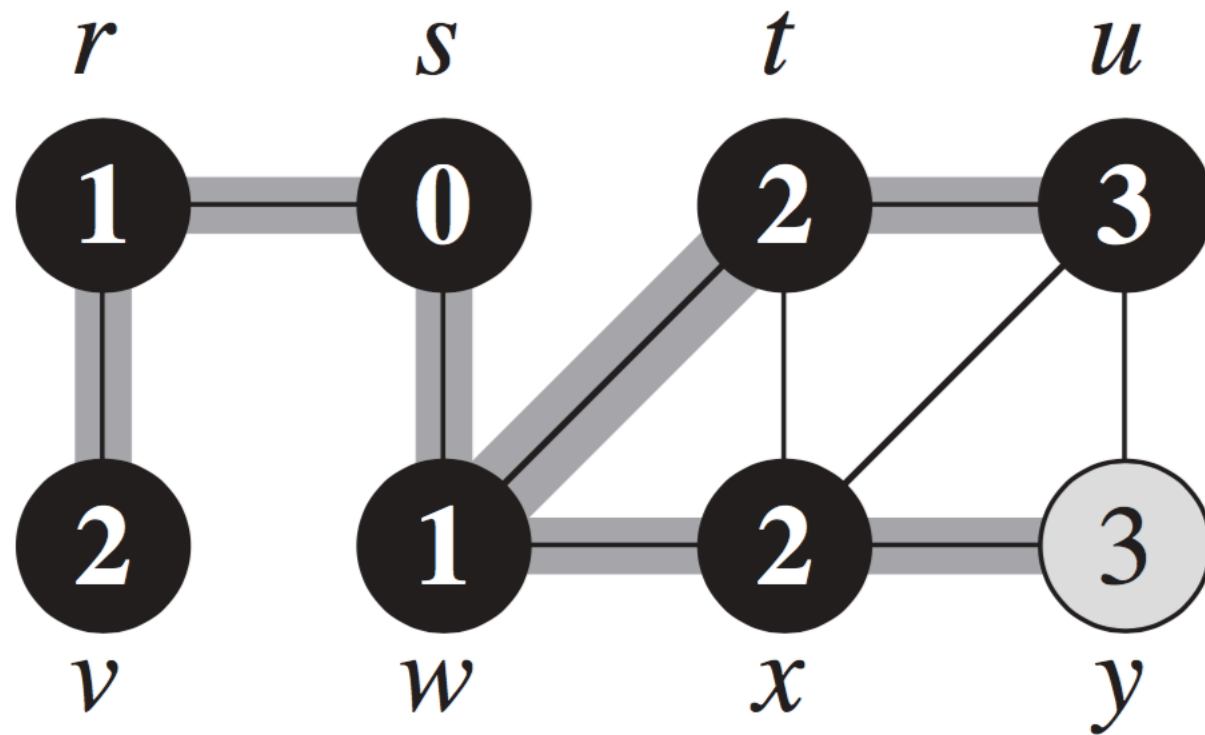
Q	v	u	y
d	2	3	3

Breadth First Search



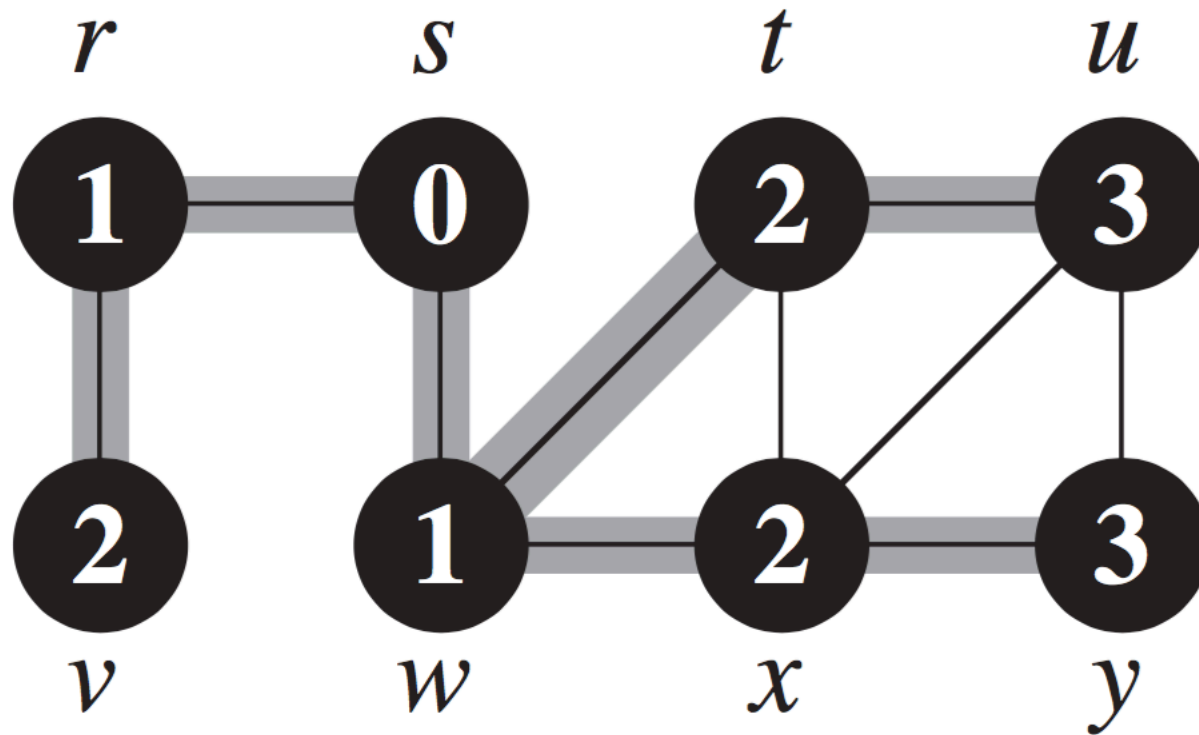
Q	u	y
d	3	3

Breadth First Search



Q y
 d 3

Breadth First Search

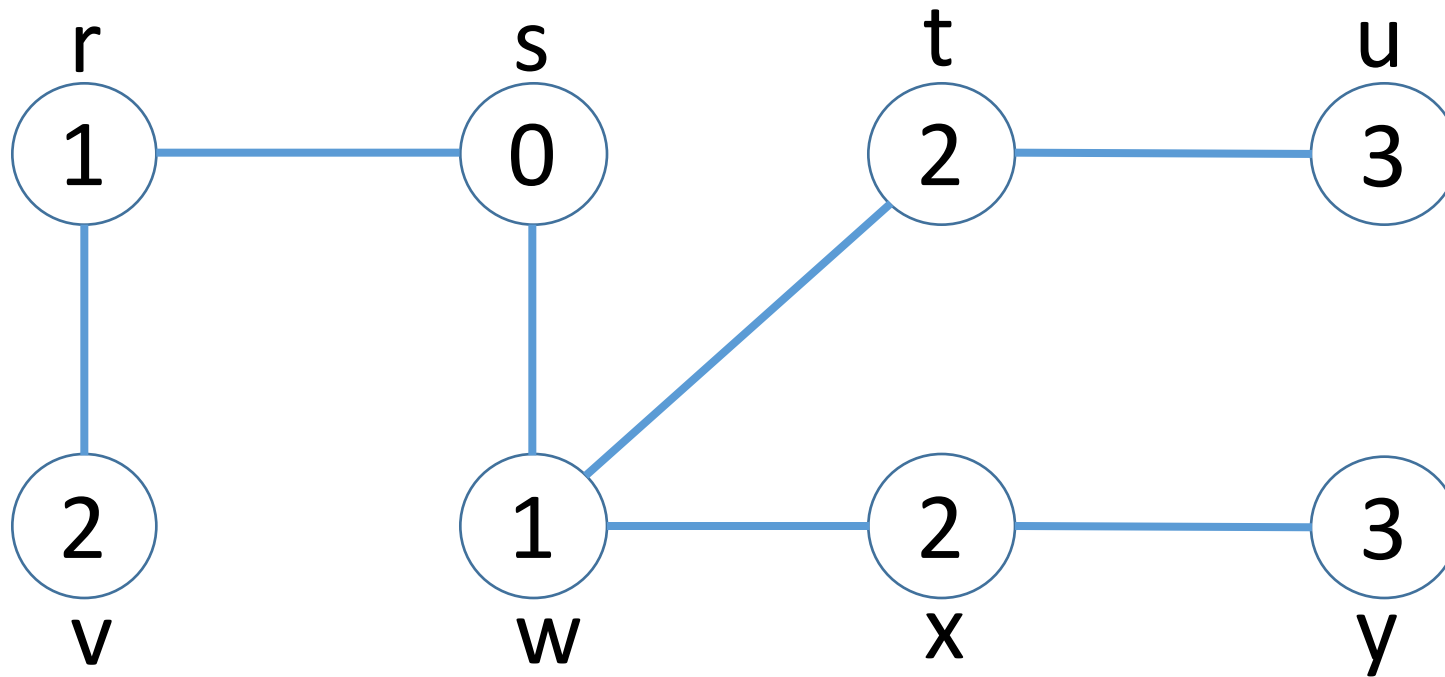


Q
 d

\emptyset

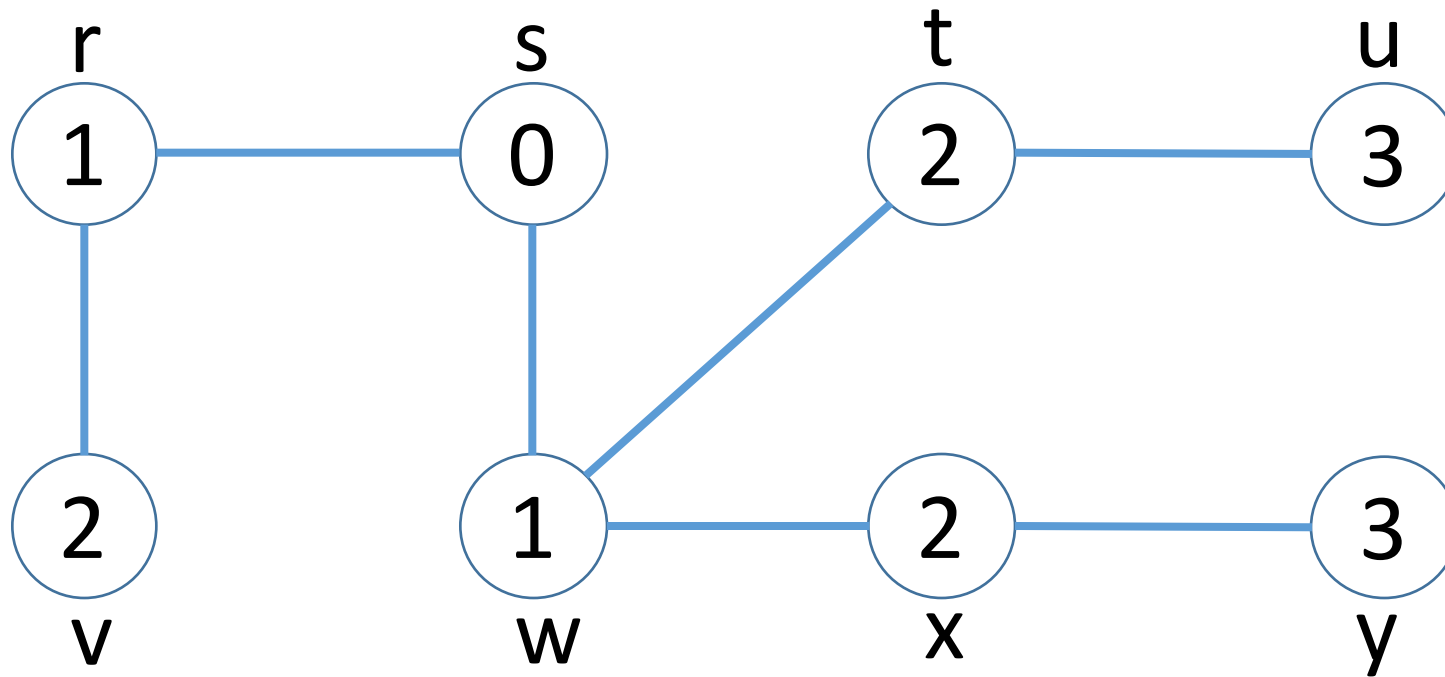
Breadth First Search

- When we explored the graph we highlighted the edges that were going to white vertices.
- These edges always result in a **tree**.



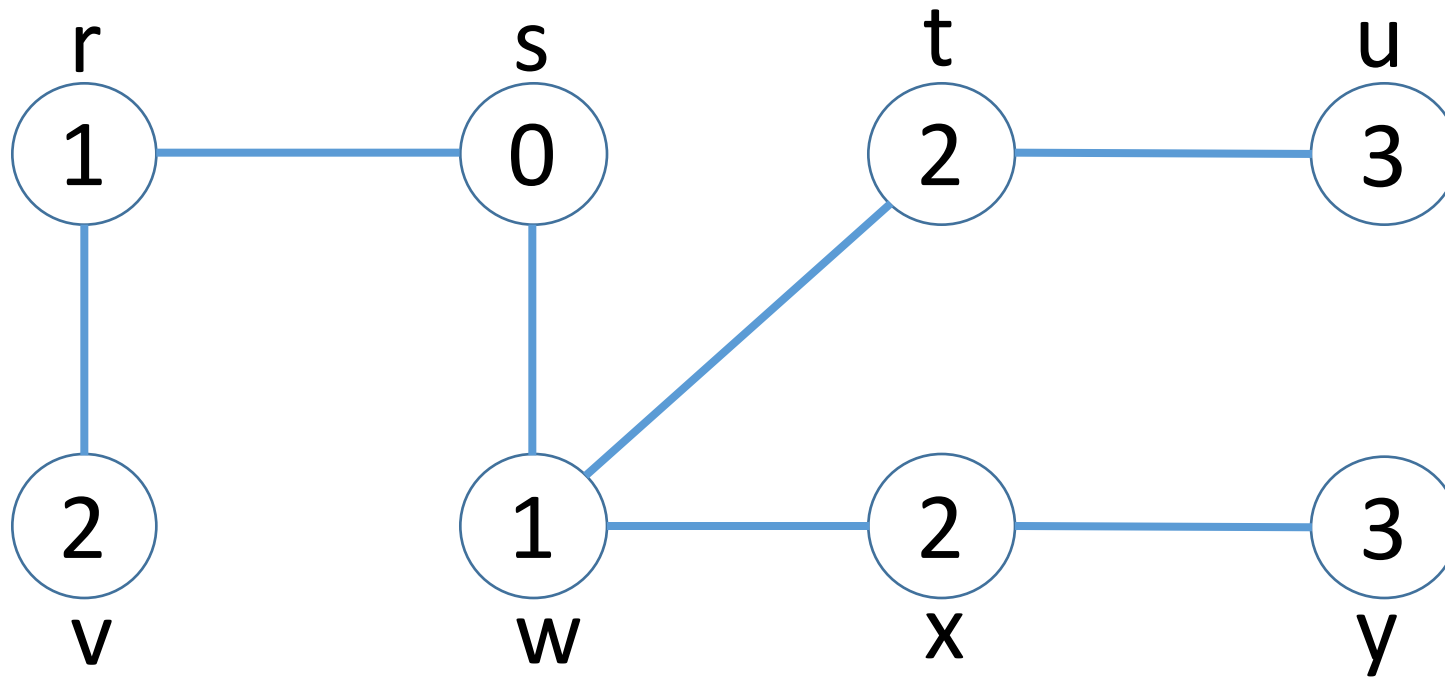
Breadth First Search

- This is the **tree of shortest paths** from s to all other reachable nodes.
- Each path from s to some node n is a shortest path.



Breadth First Search

- The length of these shortest paths are the **distances** that we computed and wrote on the nodes.
- E.g.: the shortest path from s to u has length 3.



Breadth First Search

- We usually denote the length of the shortest path between two nodes u and v with $\delta(u, v)$.
- After BFS is finishes, for any node v , $v.d = \delta(s, v)$.

BFS(G, s)

```
1  for each vertex  $u \in G.V - \{s\}$ 
2       $u.color = \text{WHITE}$ 
3       $u.d = \infty$ 
4       $u.\pi = \text{NIL}$ 
5   $s.color = \text{GRAY}$ 
6   $s.d = 0$ 
7   $s.\pi = \text{NIL}$ 
8   $Q = \emptyset$ 
9  ENQUEUE( $Q, s$ )
10 while  $Q \neq \emptyset$ 
11      $u = \text{DEQUEUE}(Q)$ 
12     for each  $v \in G.Adj[u]$ 
13         if  $v.color == \text{WHITE}$ 
14              $v.color = \text{GRAY}$ 
15              $v.d = u.d + 1$ 
16              $v.\pi = u$ 
17             ENQUEUE( $Q, v$ )
18      $u.color = \text{BLACK}$ 
```

BFS(G, s)

```
1  for each vertex  $u \in G.V - \{s\}$ 
2       $u.color = \text{WHITE}$ 
3       $u.d = \infty$ 
4       $u.\pi = \text{NIL}$ 
5   $s.color = \text{GRAY}$ 
6   $s.d = 0$ 
7   $s.\pi = \text{NIL}$ 
8   $Q = \emptyset$ 
9  ENQUEUE( $Q, s$ )
10 while  $Q \neq \emptyset$ 
11      $u = \text{DEQUEUE}(Q)$ 
12     for each  $v \in G.Adj[u]$ 
13         if  $v.color == \text{WHITE}$ 
14              $v.color = \text{GRAY}$ 
15              $v.d = u.d + 1$ 
16              $v.\pi = u$ 
17             ENQUEUE( $Q, v$ )
18      $u.color = \text{BLACK}$ 
```

This takes $\Theta(n)$ since we just have to do constant work for each node

BFS(G, s)

```
1  for each vertex  $u \in G.V - \{s\}$ 
2       $u.color = \text{WHITE}$ 
3       $u.d = \infty$ 
4       $u.\pi = \text{NIL}$ 
5   $s.color = \text{GRAY}$ 
6   $s.d = 0$ 
7   $s.\pi = \text{NIL}$ 
8   $Q = \emptyset$ 
9  ENQUEUE( $Q, s$ )
10 while  $Q \neq \emptyset$ 
11      $u = \text{DEQUEUE}(Q)$ 
12     for each  $v \in G.Adj[u]$ 
13         if  $v.color == \text{WHITE}$ 
14              $v.color = \text{GRAY}$ 
15              $v.d = u.d + 1$ 
16              $v.\pi = u$ 
17             ENQUEUE( $Q, v$ )
18      $u.color = \text{BLACK}$ 
```

This takes $\Theta(n)$ since we just have to do constant work for each node

Each node is enqueued and dequeued exactly once

BFS(G, s)

```
1  for each vertex  $u \in G.V - \{s\}$ 
2       $u.color = \text{WHITE}$ 
3       $u.d = \infty$ 
4       $u.\pi = \text{NIL}$ 
5   $s.color = \text{GRAY}$ 
6   $s.d = 0$ 
7   $s.\pi = \text{NIL}$ 
8   $Q = \emptyset$ 
9  ENQUEUE( $Q, s$ )
10 while  $Q \neq \emptyset$ 
11      $u = \text{DEQUEUE}(Q)$ 
12     for each  $v \in G.Adj[u]$ 
13         if  $v.color == \text{WHITE}$ 
14              $v.color = \text{GRAY}$ 
15              $v.d = u.d + 1$ 
16              $v.\pi = u$ 
17             ENQUEUE( $Q, v$ )
18      $u.color = \text{BLACK}$ 
```

This takes $\Theta(n)$ since we just have to do constant work for each node

Each node is enqueued and dequeued exactly once

For a node u , we do $\Theta(\deg(u))$ work; so, over all nodes we have: $\sum_u \deg(u) = ???$

BFS(G, s)

```
1  for each vertex  $u \in G.V - \{s\}$ 
2       $u.color = \text{WHITE}$ 
3       $u.d = \infty$ 
4       $u.\pi = \text{NIL}$ 
5   $s.color = \text{GRAY}$ 
6   $s.d = 0$ 
7   $s.\pi = \text{NIL}$ 
8   $Q = \emptyset$ 
9  ENQUEUE( $Q, s$ )
10 while  $Q \neq \emptyset$ 
11      $u = \text{DEQUEUE}(Q)$ 
12     for each  $v \in G.Adj[u]$ 
13         if  $v.color == \text{WHITE}$ 
14              $v.color = \text{GRAY}$ 
15              $v.d = u.d + 1$ 
16              $v.\pi = u$ 
17             ENQUEUE( $Q, v$ )
18      $u.color = \text{BLACK}$ 
```

This takes $\Theta(n)$ since we just have to do constant work for each node

Each node is enqueued and dequeued exactly once

For a node u , we do $\Theta(\deg(u))$ work; so, over all nodes we have: $\sum_u \deg(u) = 2m$

BFS(G, s)

```
1  for each vertex  $u \in G.V - \{s\}$ 
2       $u.color = \text{WHITE}$ 
3       $u.d = \infty$ 
4       $u.\pi = \text{NIL}$ 
5   $s.color = \text{GRAY}$ 
6   $s.d = 0$ 
7   $s.\pi = \text{NIL}$ 
8   $Q = \emptyset$ 
9  ENQUEUE( $Q, s$ )
10 while  $Q \neq \emptyset$ 
11      $u = \text{DEQUEUE}(Q)$ 
12     for each  $v \in G.Adj[u]$ 
13         if  $v.color == \text{WHITE}$ 
14              $v.color = \text{GRAY}$ 
15              $v.d = u.d + 1$ 
16              $v.\pi = u$ 
17             ENQUEUE( $Q, v$ )
18      $u.color = \text{BLACK}$ 
```

So, overall BFS takes $\Theta(n + m)$ time.

BFS(G, s)

```
1  for each vertex  $u \in G.V - \{s\}$ 
2       $u.color = \text{WHITE}$ 
3       $u.d = \infty$ 
4       $u.\pi = \text{NIL}$ 
5   $s.color = \text{GRAY}$ 
6   $s.d = 0$ 
7   $s.\pi = \text{NIL}$ 
8   $Q = \emptyset$ 
9  ENQUEUE( $Q, s$ )
10 while  $Q \neq \emptyset$ 
11      $u = \text{DEQUEUE}(Q)$ 
12     for each  $v \in G.Adj[u]$ 
13         if  $v.color == \text{WHITE}$ 
14              $v.color = \text{GRAY}$ 
15              $v.d = u.d + 1$ 
16              $v.\pi = u$ 
17             ENQUEUE( $Q, v$ )
18      $u.color = \text{BLACK}$ 
```

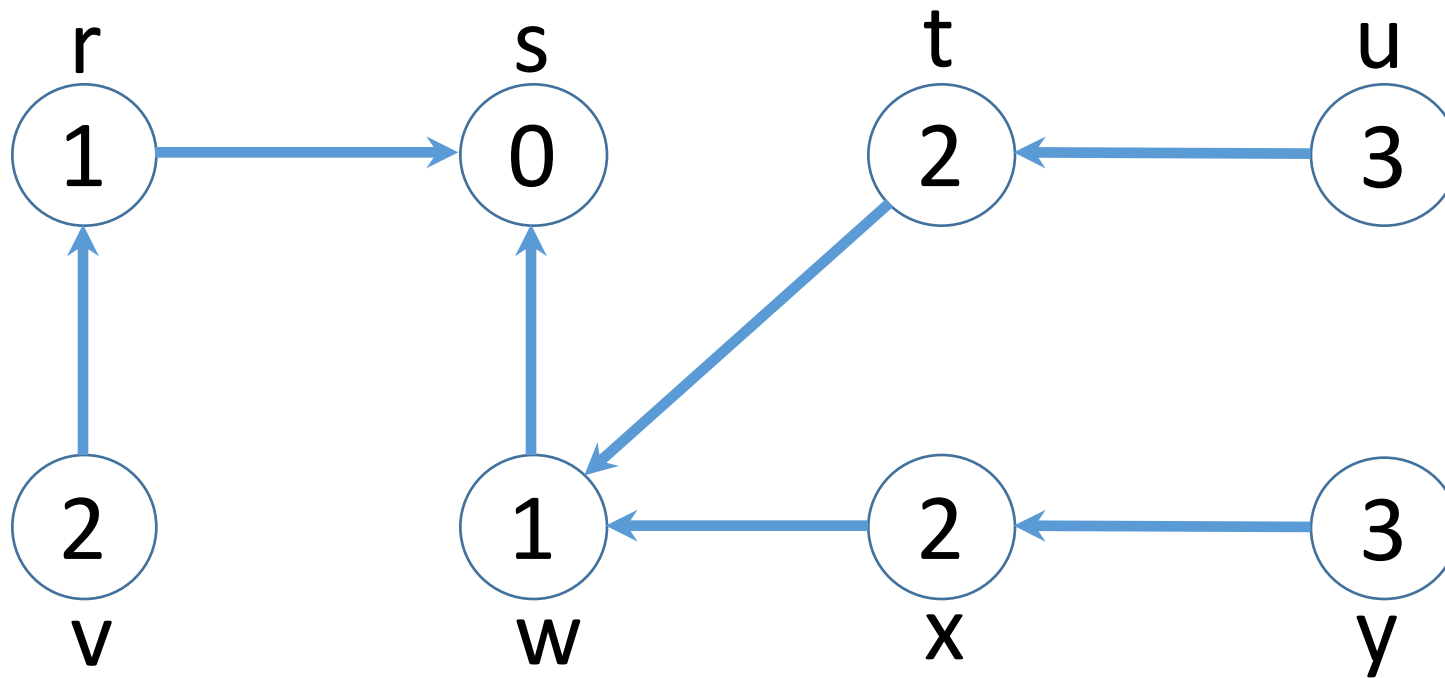
There is one more thing!

For a node v , $v.\pi$ can be considered the **parent of node v in the shortest path tree**.

In fact, when the BFS search is finished, every node points to its parent, and s (the source node) does not have a parent.

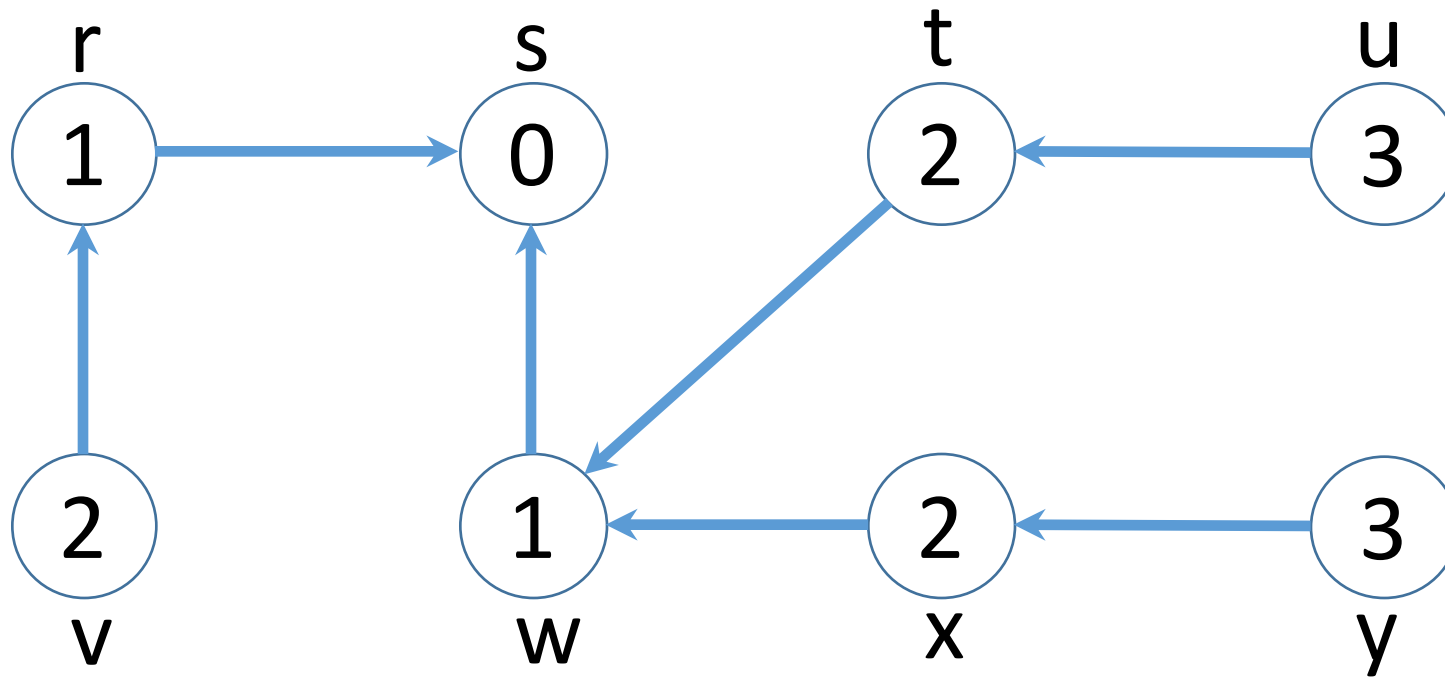
Breadth First Search

- The parent pointers look like this:



Breadth First Search

- **Exercise:** write a subroutine `PRINTPATH(z)` that prints the nodes shortest path from `s` to `z` using the parent pointers. (compare it with the one in CLRS)



Tips on simplifying BFS

1. If the objective is to merely find the length of the shortest path, then, there is **no need to store parent pointers**.
2. Instead of graying a node **we can just blacken it** since we just want to know if a node was visited or not.
3. Assuming that the nodes are numbered from 1 to n, instead of defining a node class, we could have **visited (boolean array), distance (int array), and parent (int array)** to store these info. (much faster than pointers)

Applications of BFS

- Say we are modeling a city by considering each **intersection** as a **node** and each **street** as an **edge**.
- Then, we might be interested to know the shortest path from all intersections to a special intersection (source node).
- That special intersection could be near **a hospital** or **the police department**.

Applications of BFS

- Let's say in the city (modeled as a 2D matrix) we want to go from **intersection A** to **intersection H** with the least number of moves.
- We can go left, right, up, or down in each move.
- Also, some intersections are blocked (#) and we can't use them.

	A				
			#	#	
	#	#			
			#	#	
	#	#	H		
					#

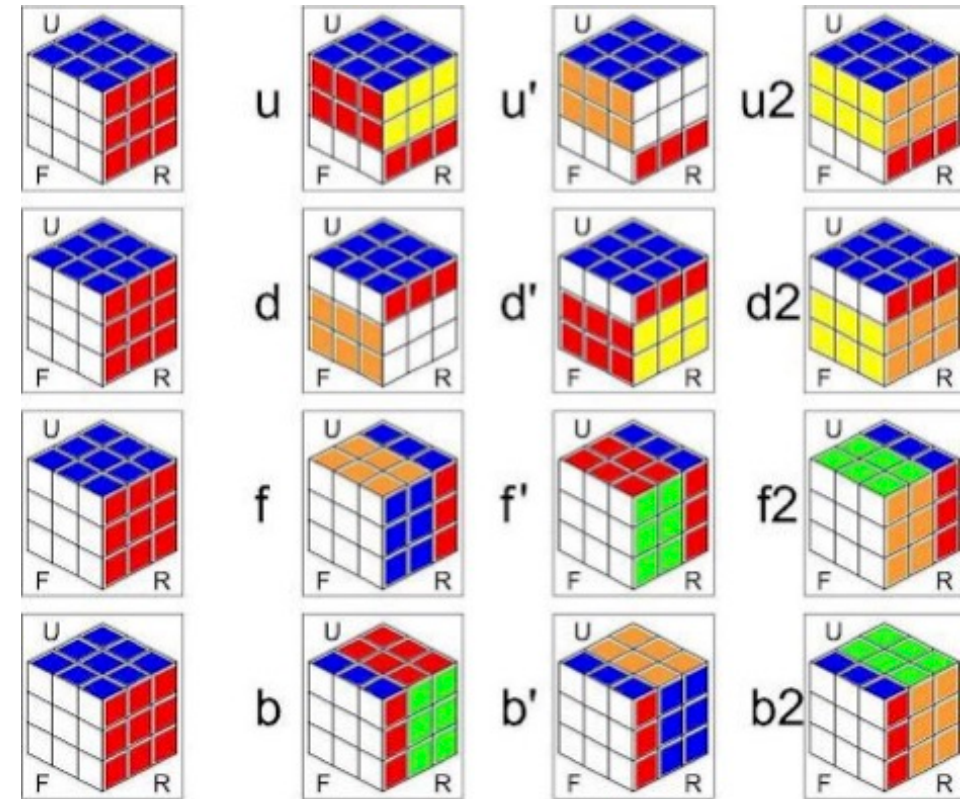
Applications of BFS

- **Important note:**
- We don't have to always explicitly make the whole graph and start the search.
- As long as we know how to find the neighbors of a node we can still perform a BFS, and discover new nodes.

	A				
			#	#	
	#	#			
			#	#	
	#	#	H		
					#

Applications of BFS

- Another example could be **solving a Rubix cube** with the **minimum** number of moves.
- Each state is a node, and each rotation gives you a neighbor.
- For many types of games the **optimal strategy for winning** could be found using ideas similar to a BFS.



Depth First Search

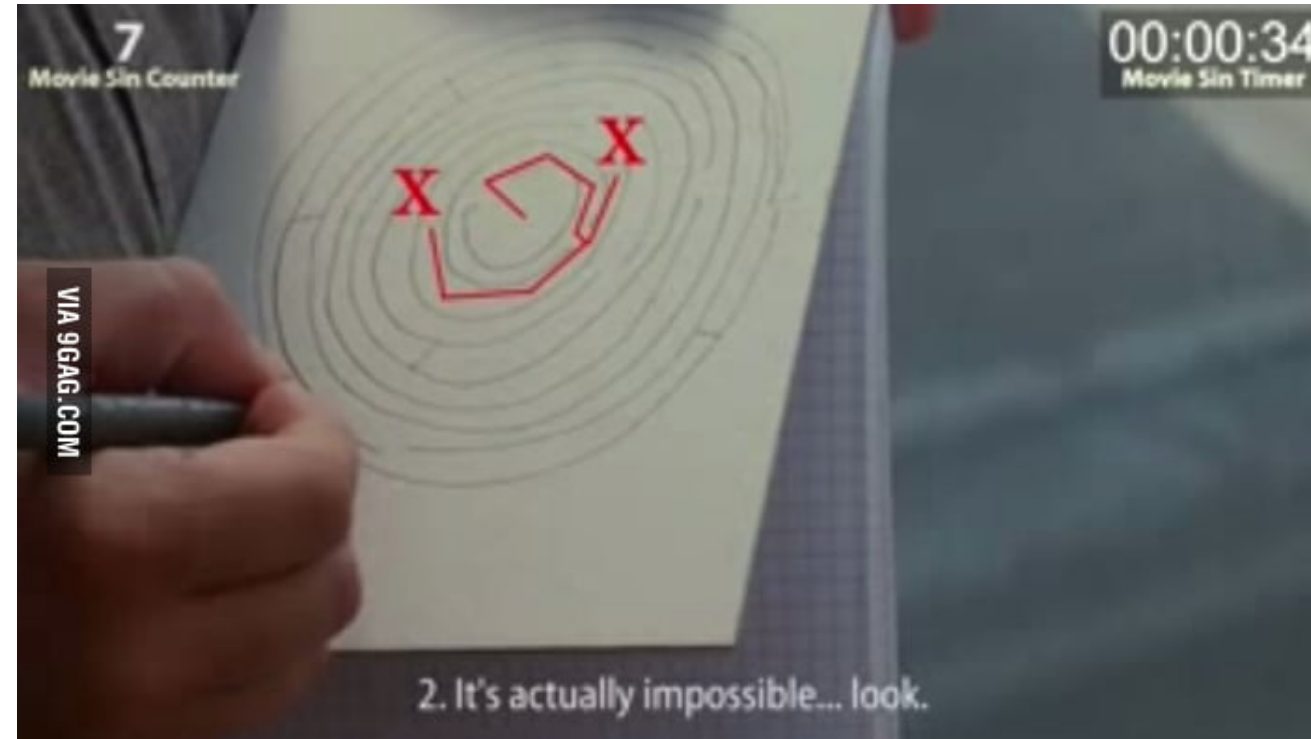
- Instead of exploring the graph layer by layer, **DFS goes in depth as much as possible.**
- When it reaches a dead-end it **backtracks** to a node where it still has more options to try out.

Depth First Search

- It's exactly like when you are trying to solve a maze with bread crumbs!



Depth First Search



Movie mistake fact! (inception 2010)

Depth First Search

- We use DFS when we want to **explore all nodes** in a graph.
- BFS and DFS that we implement here work on **directed graphs**, as well.

Depth First Search

- DFS algorithm can be implemented recursively.
- Our version of DFS records **two timestamps**:
 1. When it **starts discovering** (grays the node)
 2. When it **finishes discovering** (blackens the node)
- These timestamps provide valuable information that we'll use later.


DFS(G)

```
1  for each vertex  $u \in G.V$ 
2       $u.color = \text{WHITE}$ 
3       $u.\pi = \text{NIL}$ 
4   $time = 0$ 
5  for each vertex  $u \in G.V$ 
6      if  $u.color == \text{WHITE}$ 
7          DFS-VISIT( $G, u$ )
```

time is a global variable



Question: why do we have
to have a for loop?



DFS(G)

```
1  for each vertex  $u \in G.V$ 
2       $u.color = \text{WHITE}$ 
3       $u.\pi = \text{NIL}$ 
4   $time = 0$ 
5  for each vertex  $u \in G.V$ 
6      if  $u.color == \text{WHITE}$ 
7          DFS-VISIT( $G, u$ )
```

time is a global variable

Answer: This allows us to explore the whole graph even when some nodes are not reachable from each other.

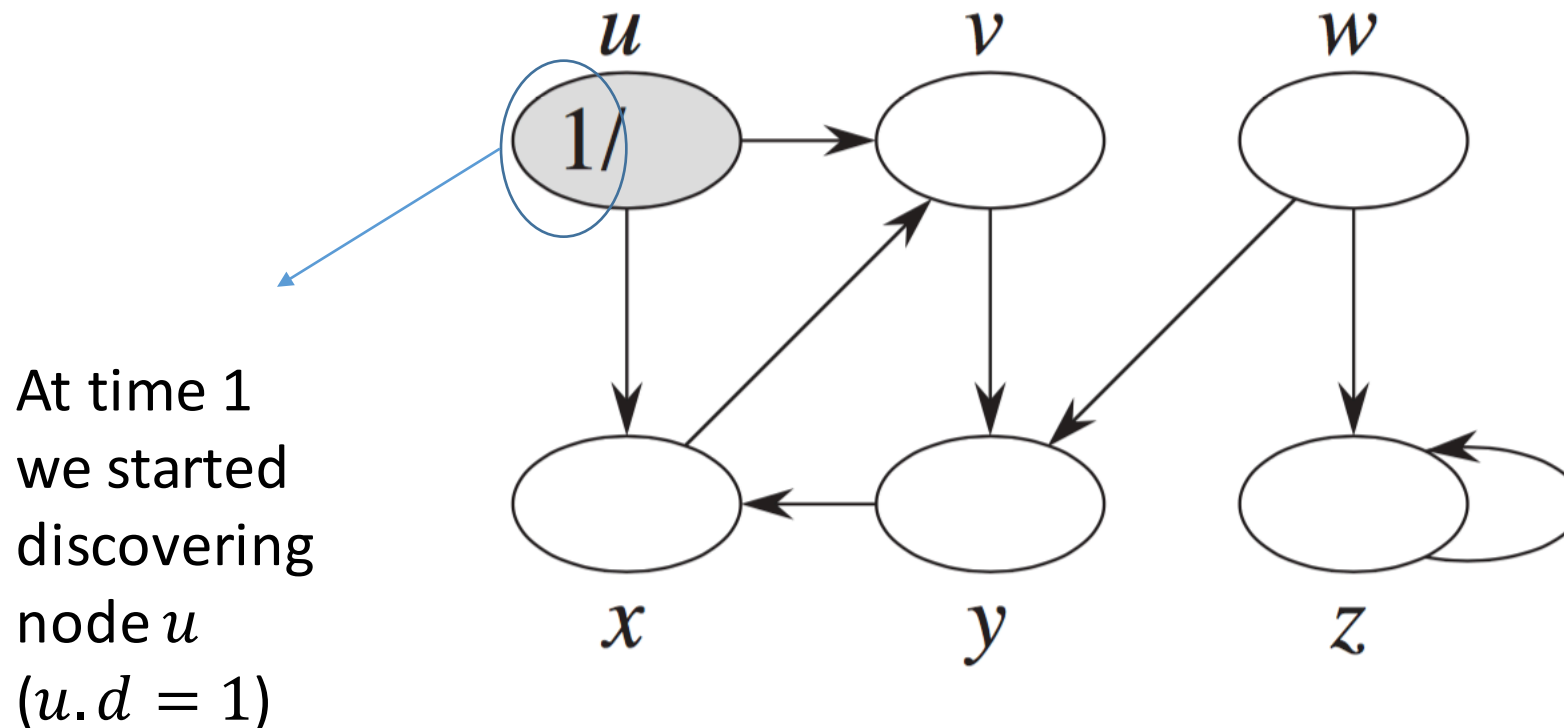
DFS-VISIT(G, u)

```
1  time = time + 1           // white vertex  $u$  has just been discovered
2   $u.d = time$ 
3   $u.color = \text{GRAY}$ 
4  for each  $v \in G.Adj[u]$       // explore edge  $(u, v)$ 
5      if  $v.color == \text{WHITE}$ 
6           $v.\pi = u$ 
7          DFS-VISIT( $G, v$ )
8   $u.color = \text{BLACK}$         // blacken  $u$ ; it is finished
9  time = time + 1
10  $u.f = time$ 
```

Depth First Search

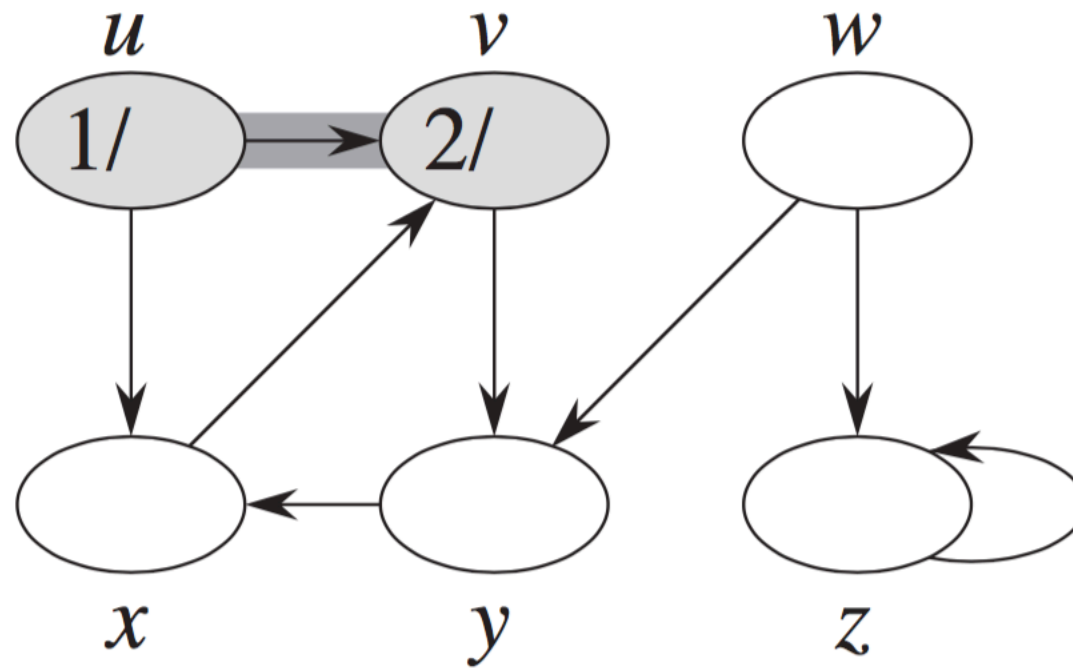
- We don't keep distances in DFS search since we don't care about them.

time = 1



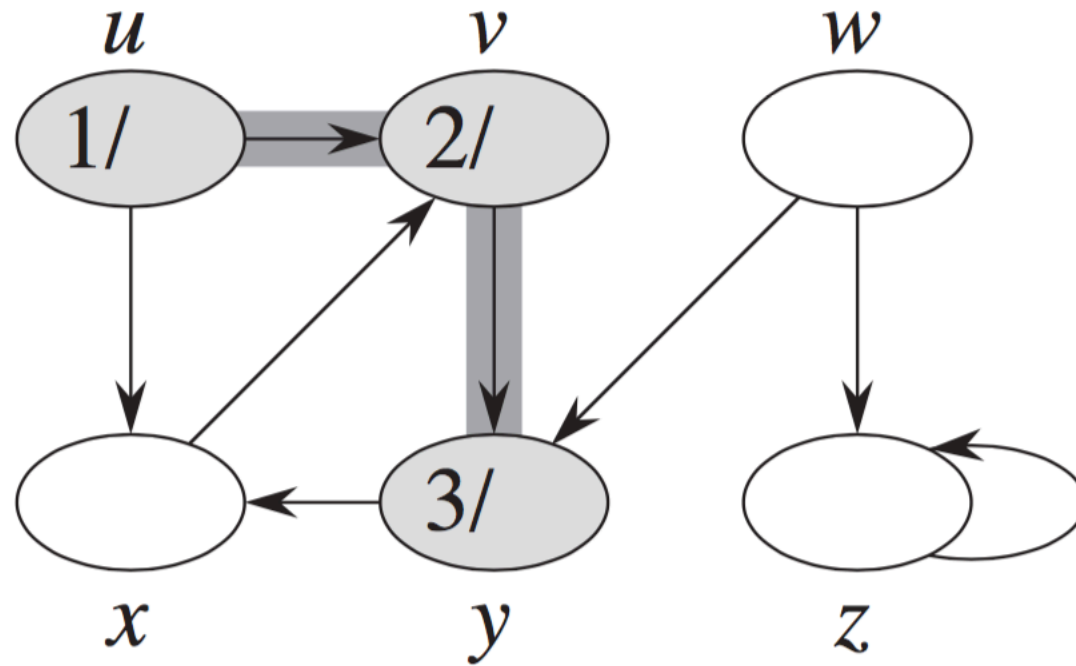
Depth First Search

time = 2



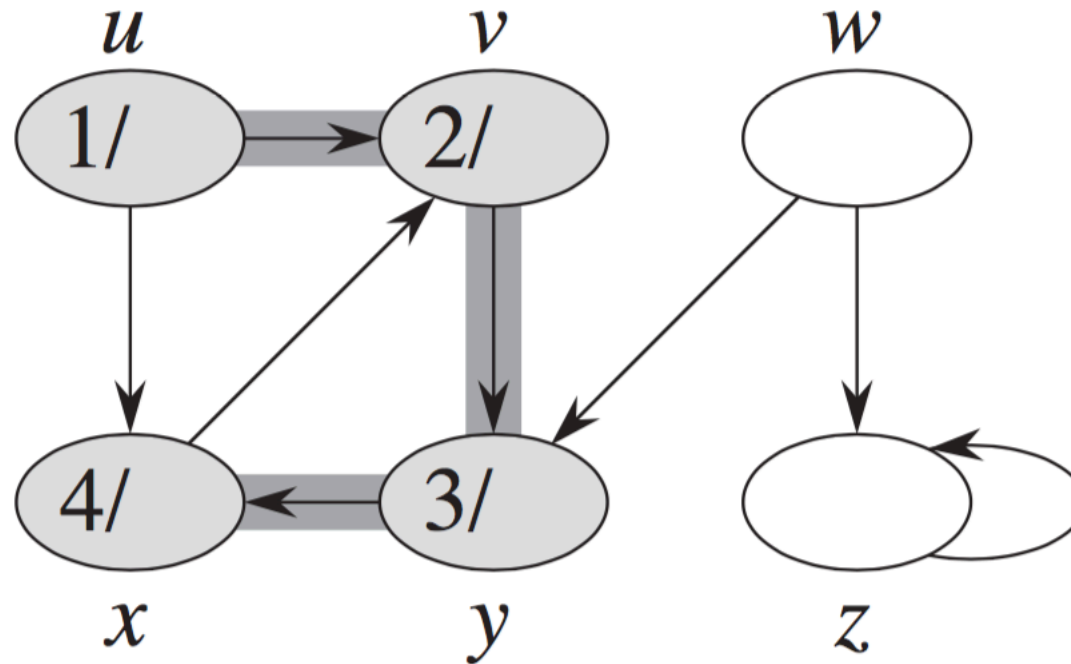
Depth First Search

time = 3



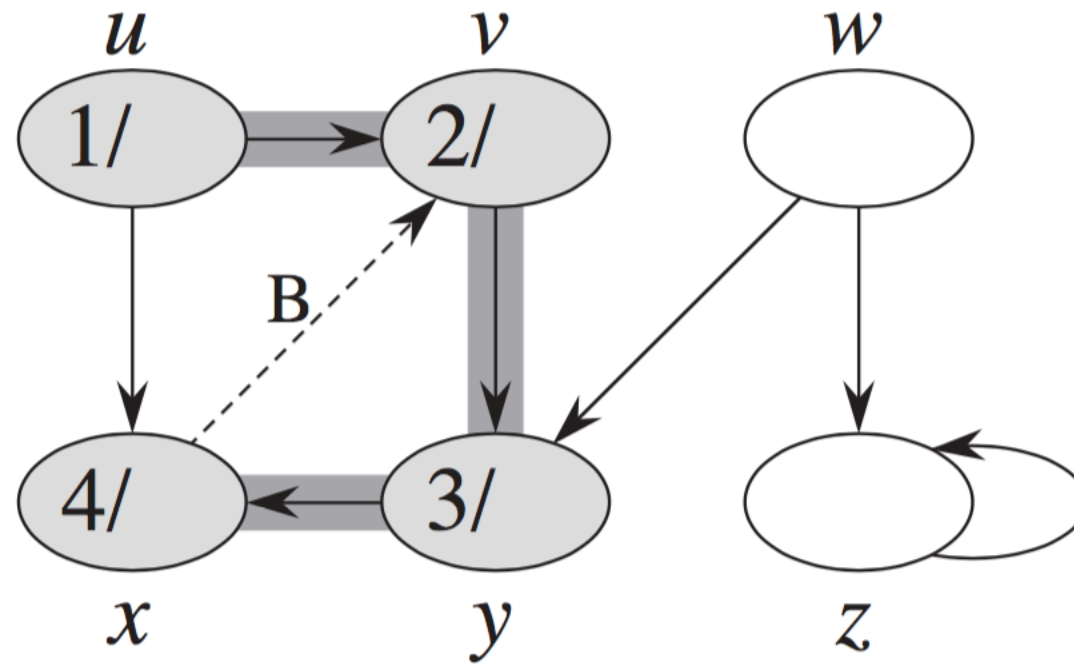
Depth First Search

time = 4



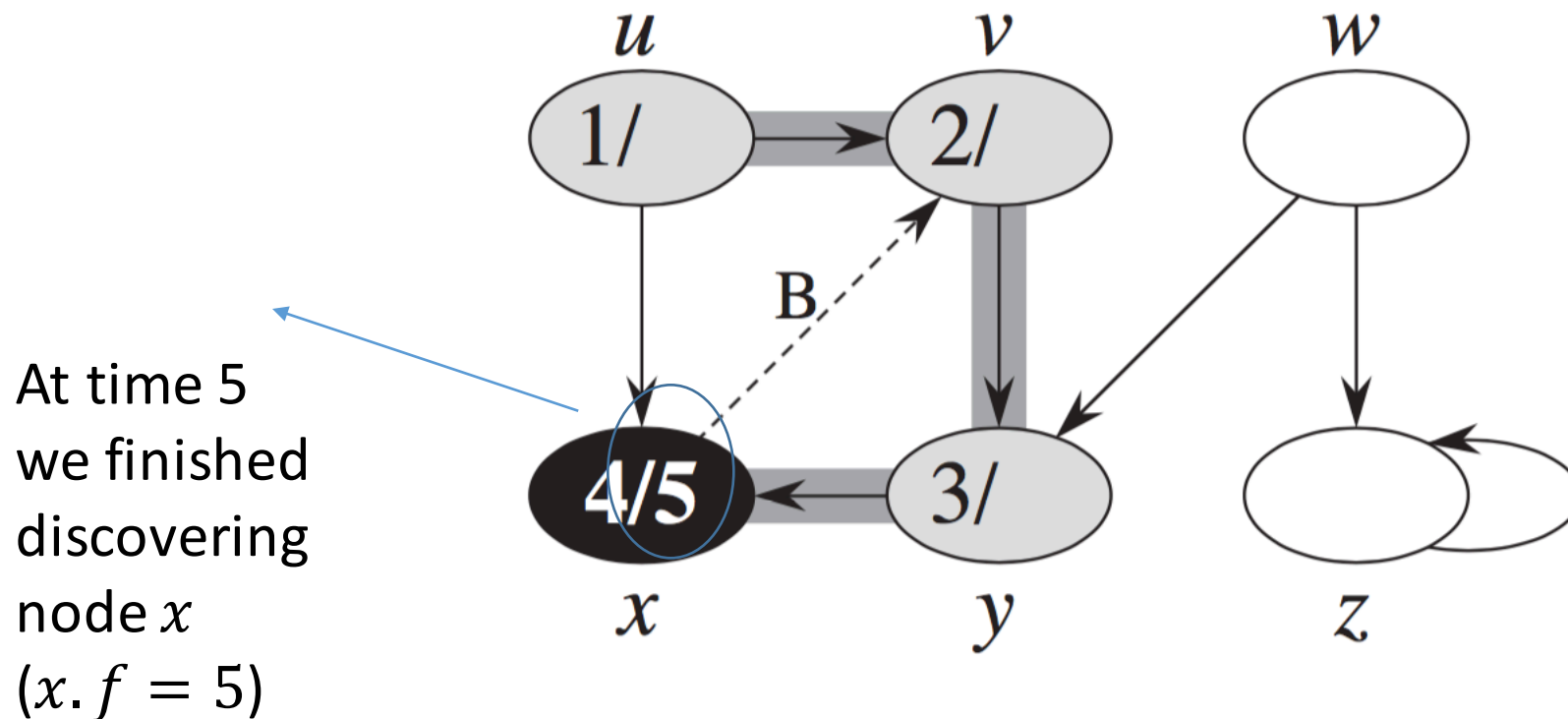
Depth First Search

time = 4



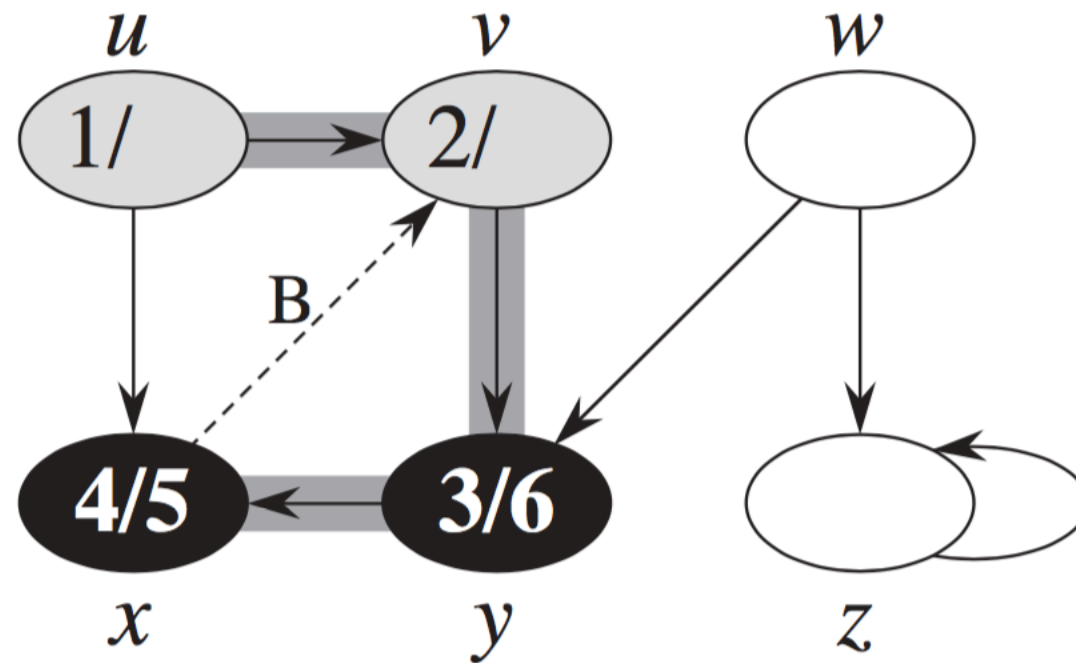
Depth First Search

time = 5



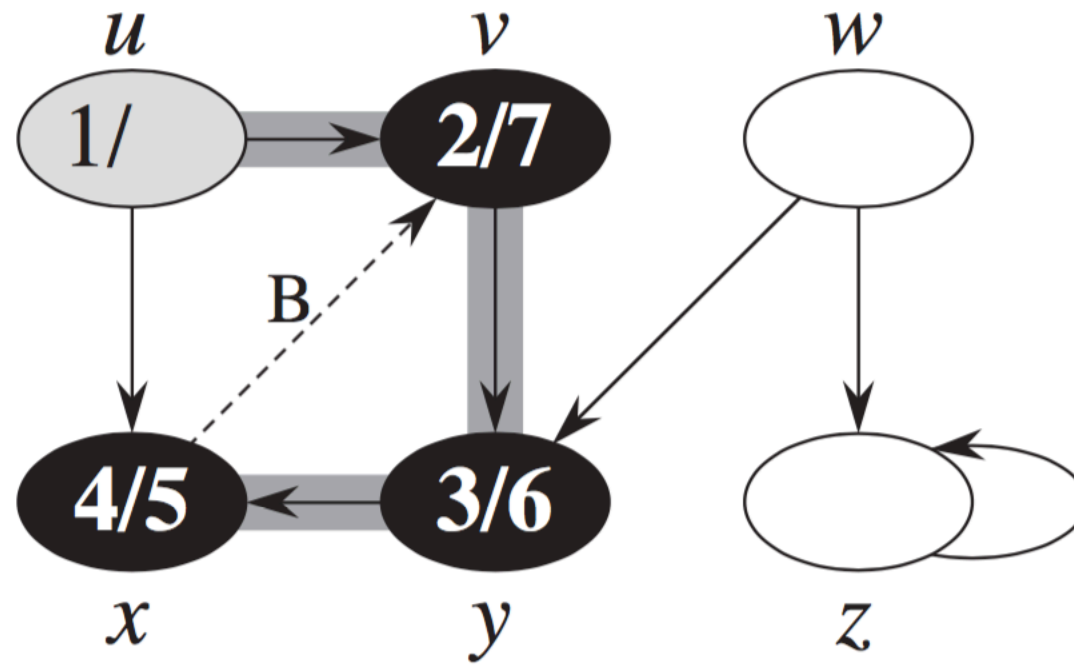
Depth First Search

time = 6



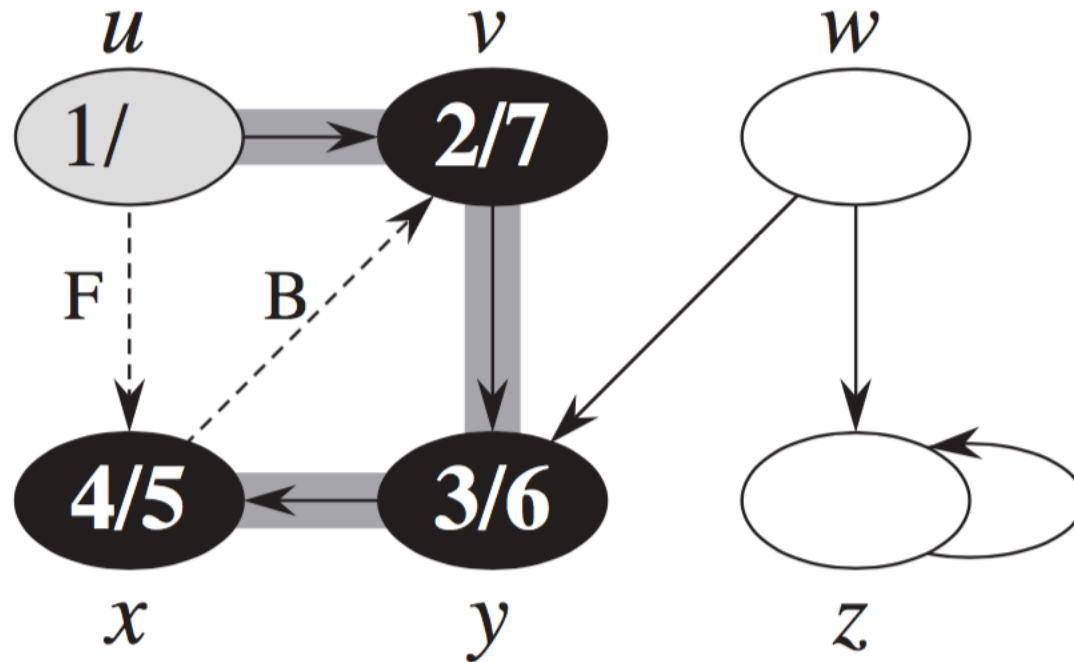
Depth First Search

time = 7



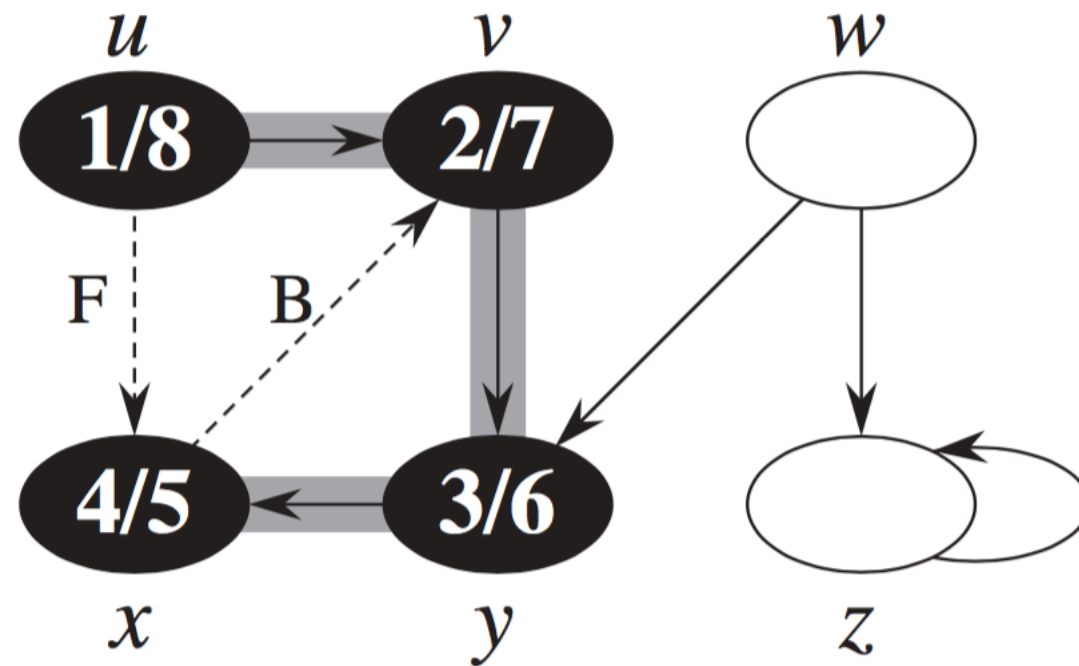
Depth First Search

time = 7



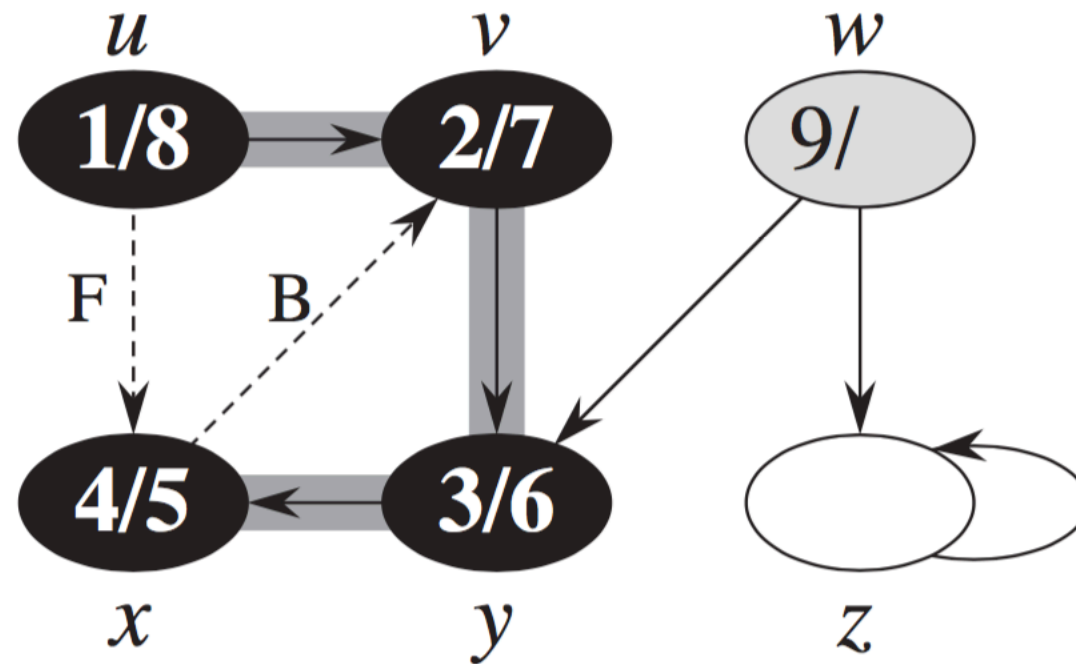
Depth First Search

time = 8



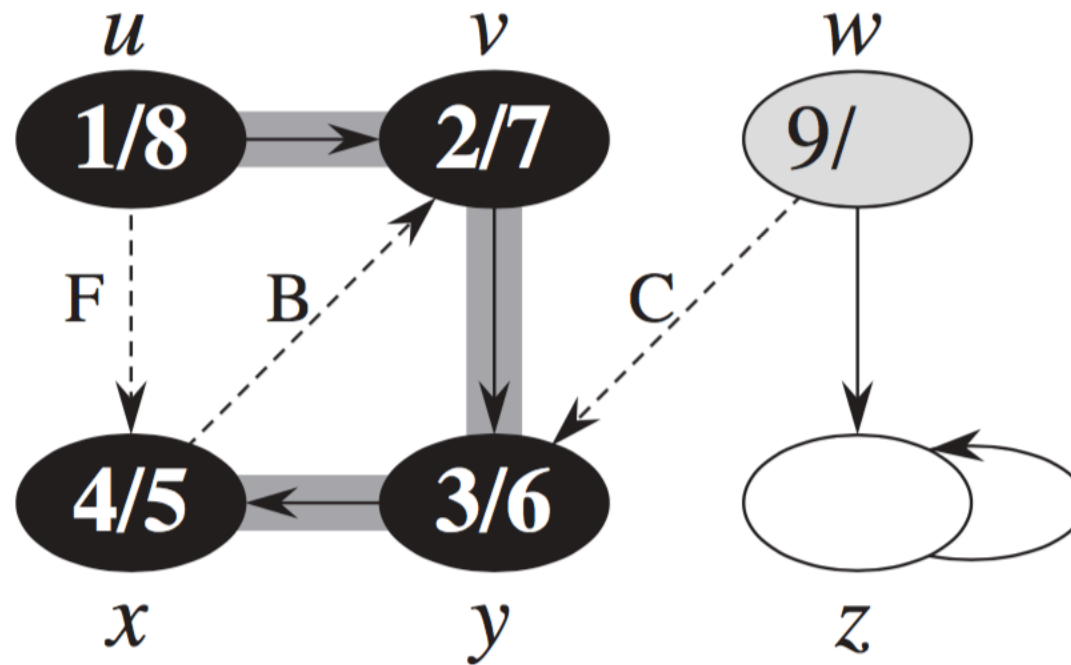
Depth First Search

time = 9



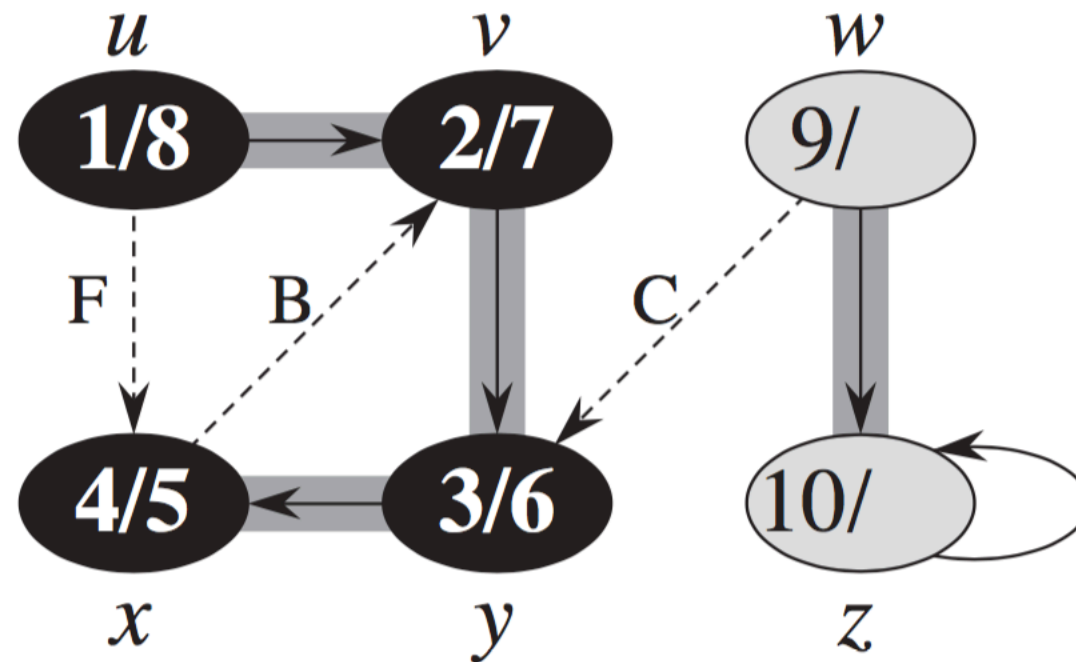
Depth First Search

time = 9



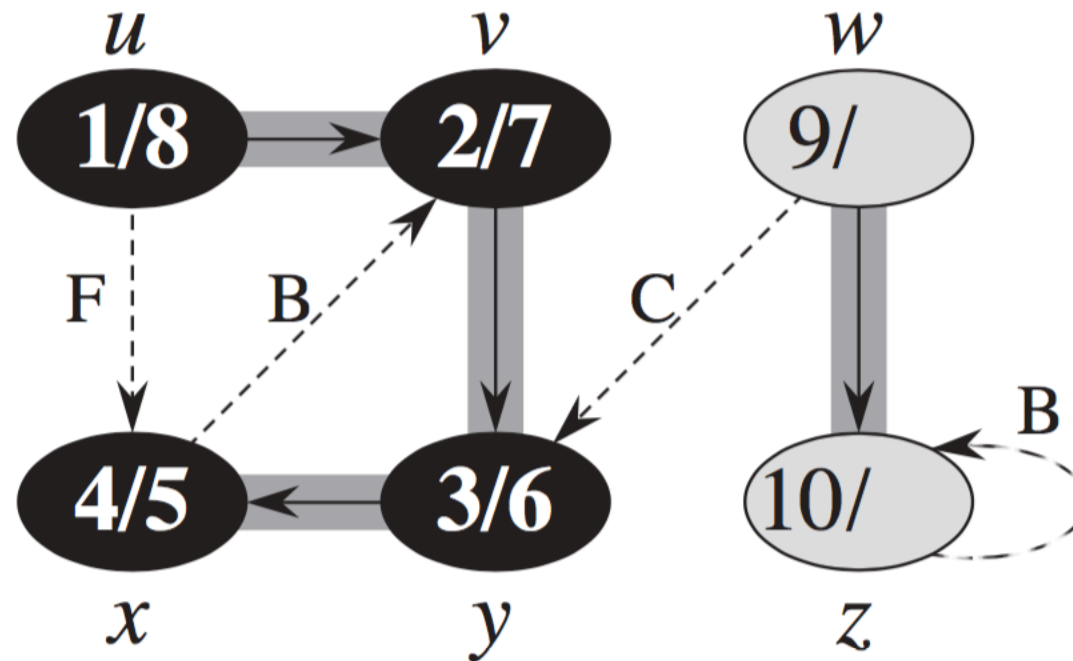
Depth First Search

time = 10



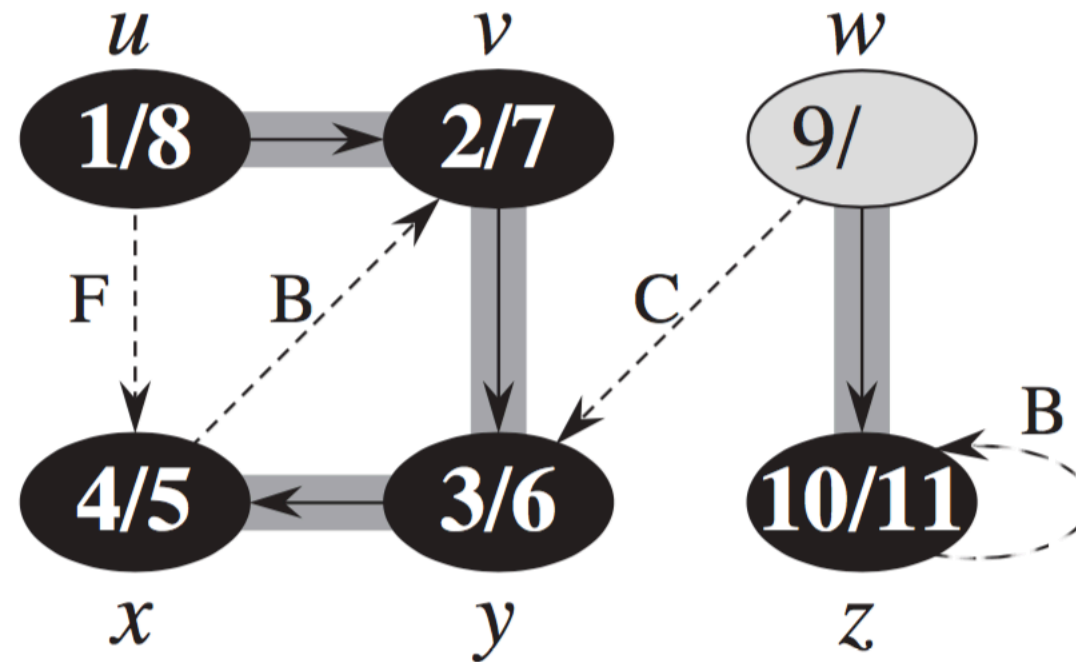
Depth First Search

time = 10



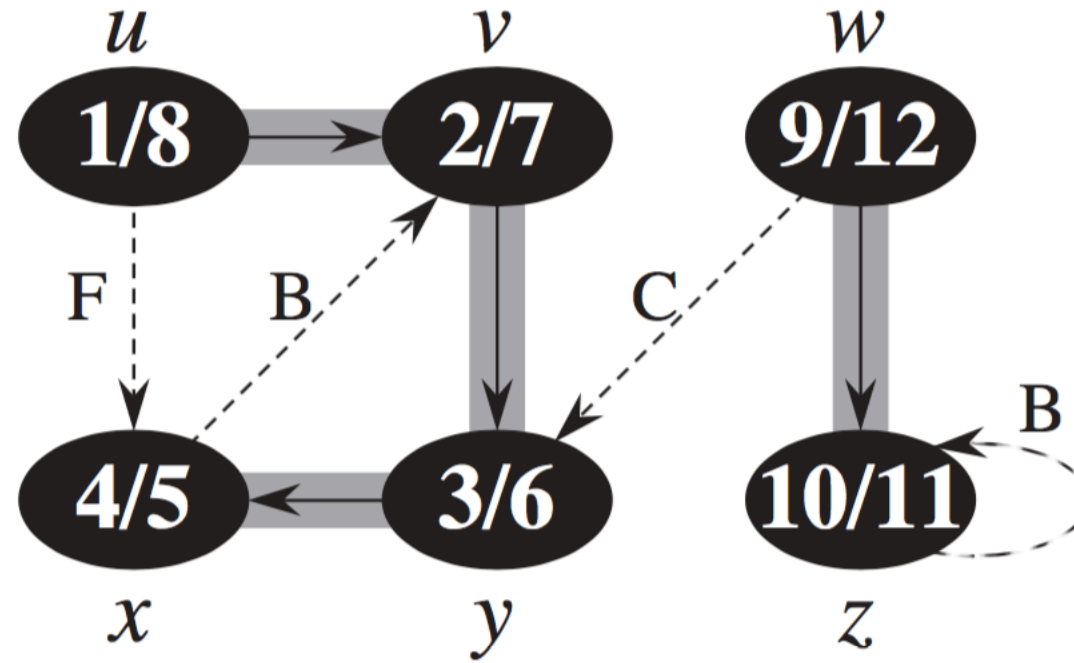
Depth First Search

time = 11



Depth First Search

time = 12



Depth First Search

- **Question:** What is the running time of DFS?

DFS(G)

```
1  for each vertex  $u \in G.V$ 
2       $u.color = \text{WHITE}$ 
3       $u.\pi = \text{NIL}$ 
4   $time = 0$ 
5  for each vertex  $u \in G.V$ 
6      if  $u.color == \text{WHITE}$ 
7          DFS-VISIT( $G, u$ )
```

DFS-VISIT(G, u)

```
1   $time = time + 1$ 
2   $u.d = time$ 
3   $u.color = \text{GRAY}$ 
4  for each  $v \in G.Adj[u]$ 
5      if  $v.color == \text{WHITE}$ 
6           $v.\pi = u$ 
7          DFS-VISIT( $G, v$ )
8   $u.color = \text{BLACK}$ 
9   $time = time + 1$ 
10  $u.f = time$ 
```

Depth First Search

- **Answer:** The time complexity is still $\Theta(n + m)$. The two for loops in DFS, each take $\Theta(n)$ time. Moreover, for any node v will eventually look at all neighbors of v , which takes $\deg(v)$. Since the sum of all degrees is $2m$ in an undirected graph and m in a directed graph, overall it takes $\Theta(n + m)$.

Simplifying DFS

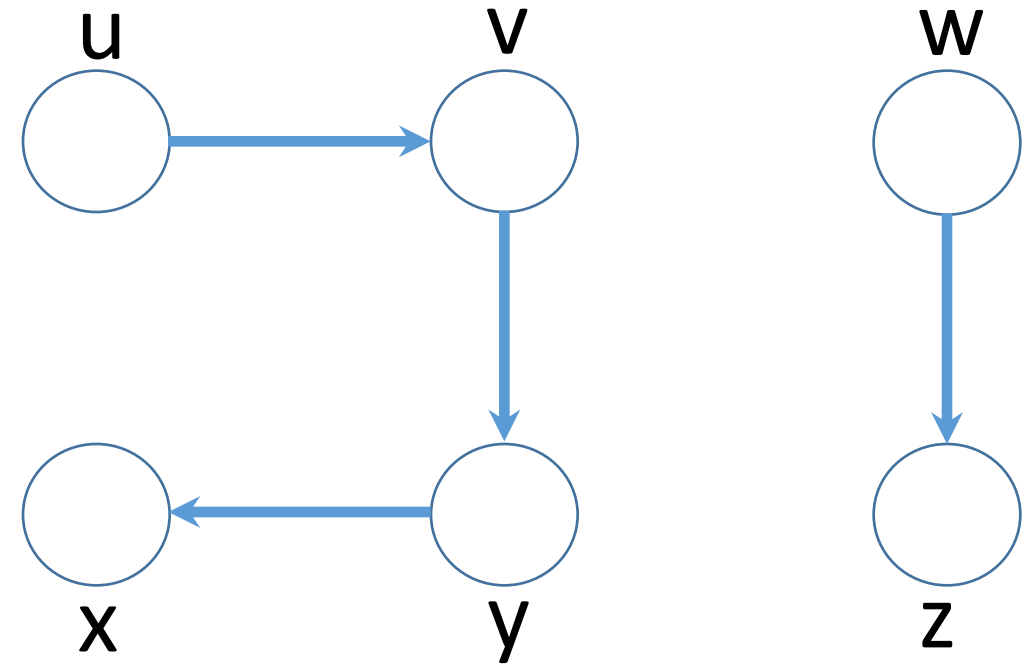
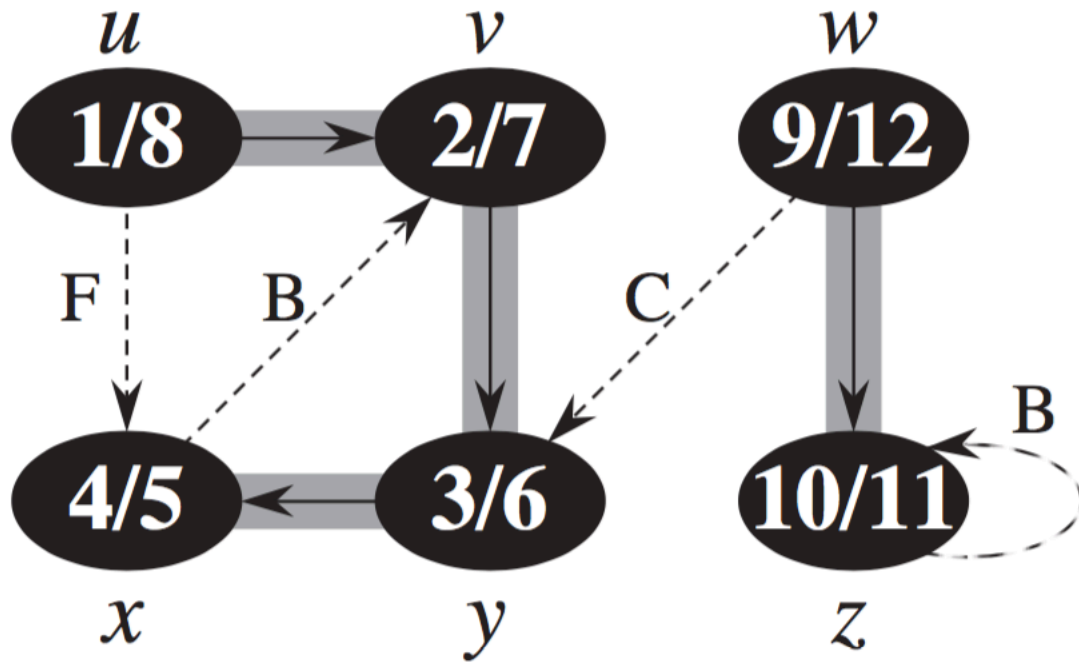
1. If the objective is to merely visit all nodes and find a forest, there is no need to record discovery time, finish times, and also parent pointers.
2. We can use arrays instead of node classes to speed up the algorithm.
3. More importantly, there are **two drawbacks with recursive implementations**:
 - a. Recursive calls **make algorithms slow** as calling another function and storing necessary variables in the call stack has some overhead.

Simplifying DFS

- b. The memory for calling functions is allocated from the system's stack which has a limited fixed size, whereas the memory for arrays, stacks, queues, and other data structures defined inside a program are allocated from the heap memory and can grow upon request. So, having a lot of nested calls (~ 10000) **could result in a stack overflow error.**
- ✓ So, if you think that the maximum depth of the recursion is too much, it's preferred to **re-implement DFS-VISIT with a stack that you define inside your code.**

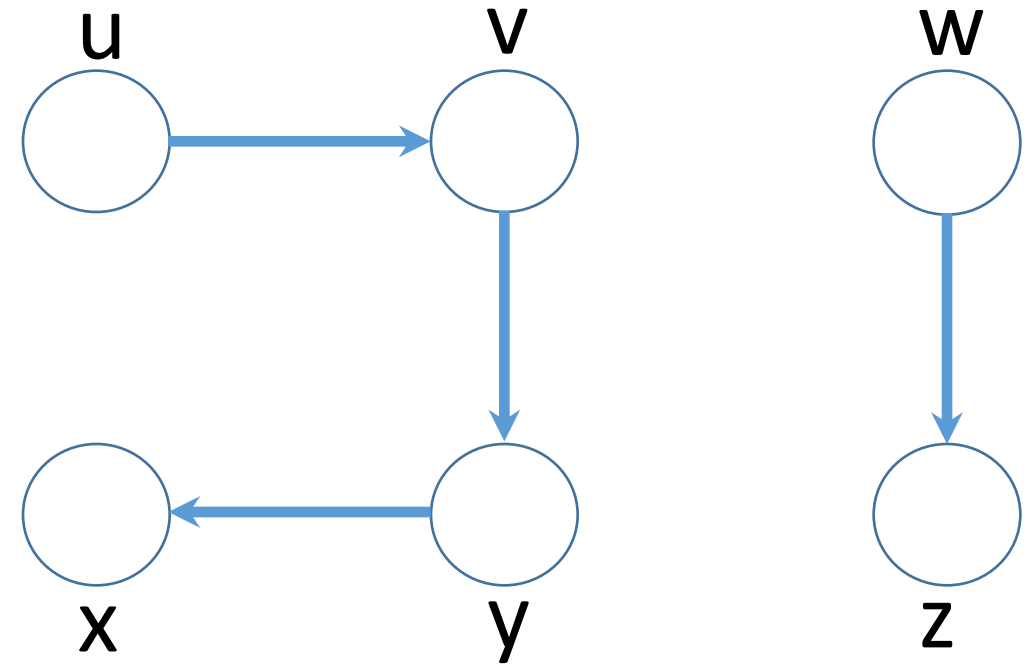
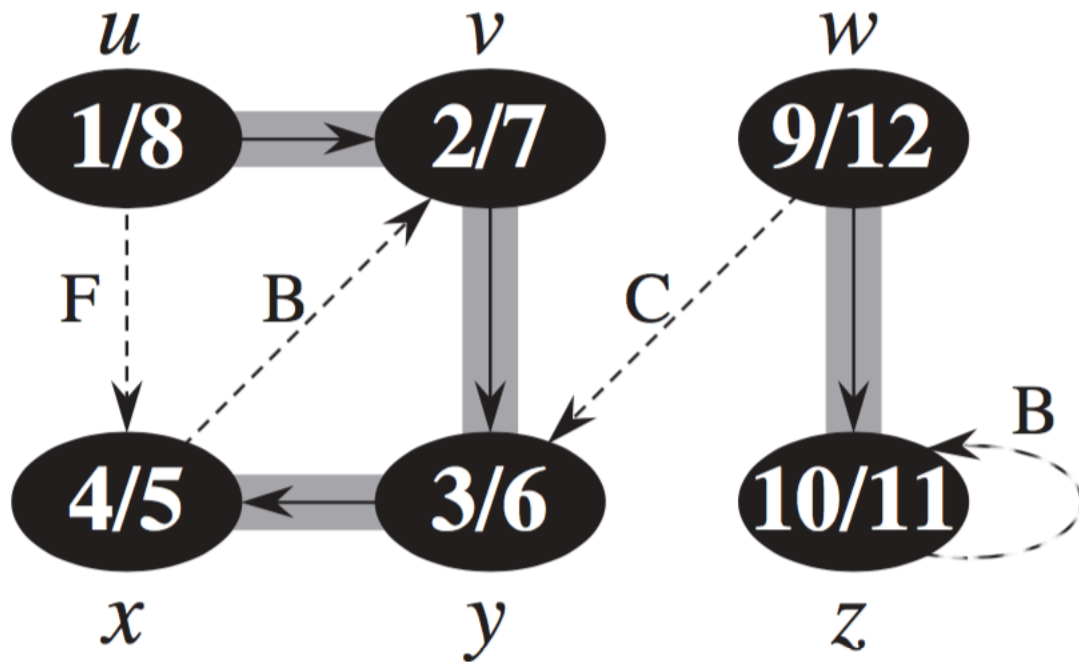
Depth First Search

- DFS results in a forest.



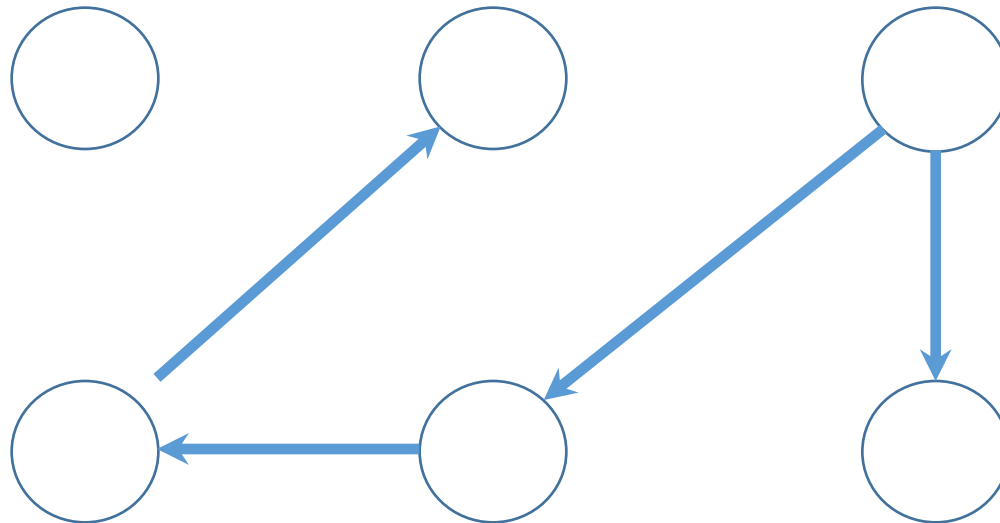
Depth First Search

- **Question:** Does DFS always result in the same forest?



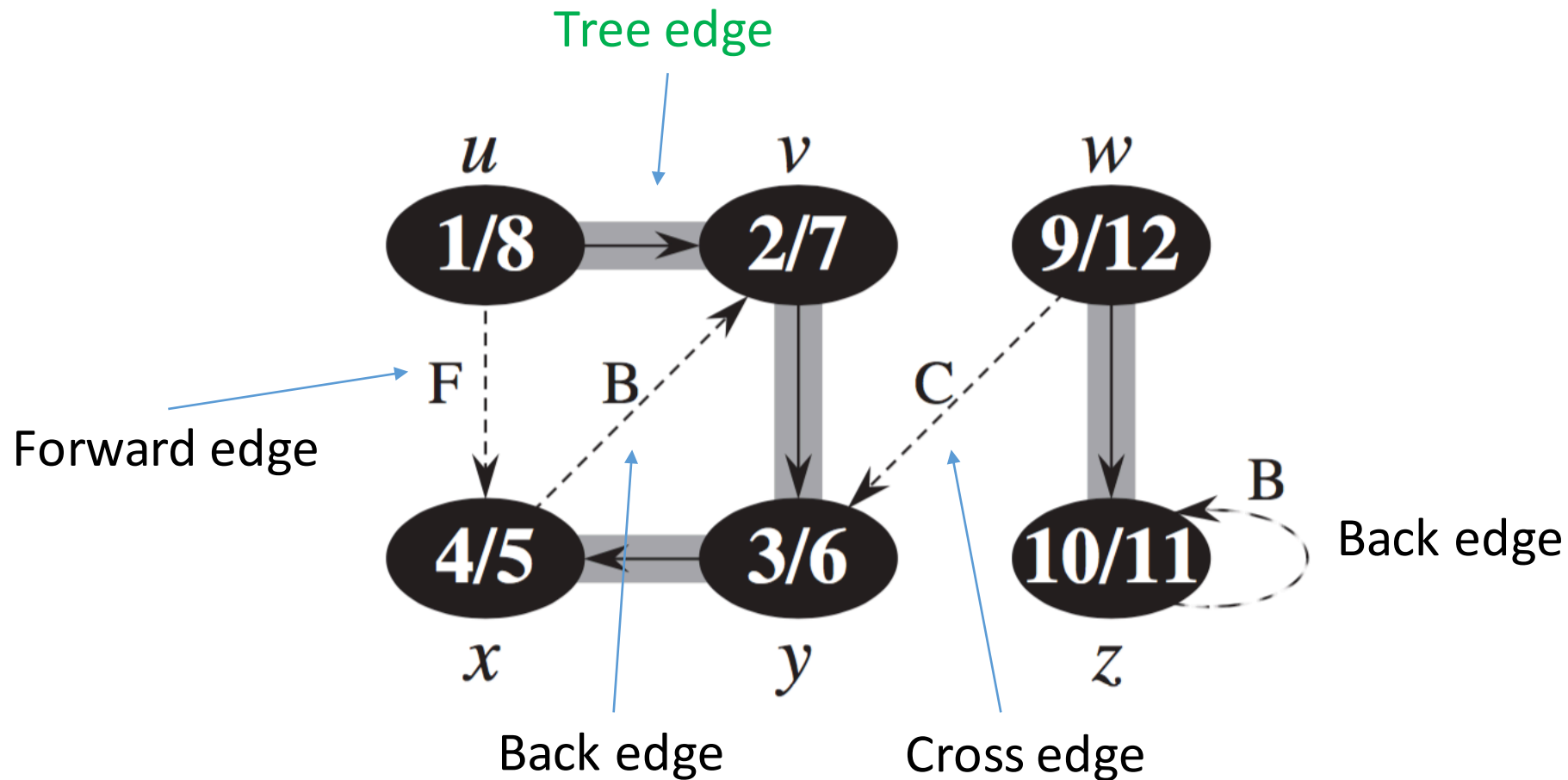
Depth First Search

- **Answer:** Not necessarily, if we start from w we could actually have a the one on the left. It depends on where we start and also on the ordering of the neighbors inside the adjacency lists.



Edge classification

- There are 4 types of edges that we will have after a DFS:



Edge classification

- There are 4 types of edges that we have in a DFS:

1. **Tree edge:** an edge in the forest obtained by DFS

2. **Back edge:** an edge from a node to its ancestor

✓ By convention a self-loop is only considered a back edge

3. **Forward edge:** an edge from a node to a descendent

4. **Cross edge:** all other edges which are (1) an edge between two nodes that are neither ancestor or descendent of one another, and (2) those that are between two different trees in DFS forest.

Edge classification

- **Theorem:** In an undirected graph we can only have tree edges or back edges.

Edge classification

- **Theorem:** In an undirected graph we can only have tree edges or back edges.

Proof: We can argue that if the edges are undirected then a **forward edge would become a back edge.**

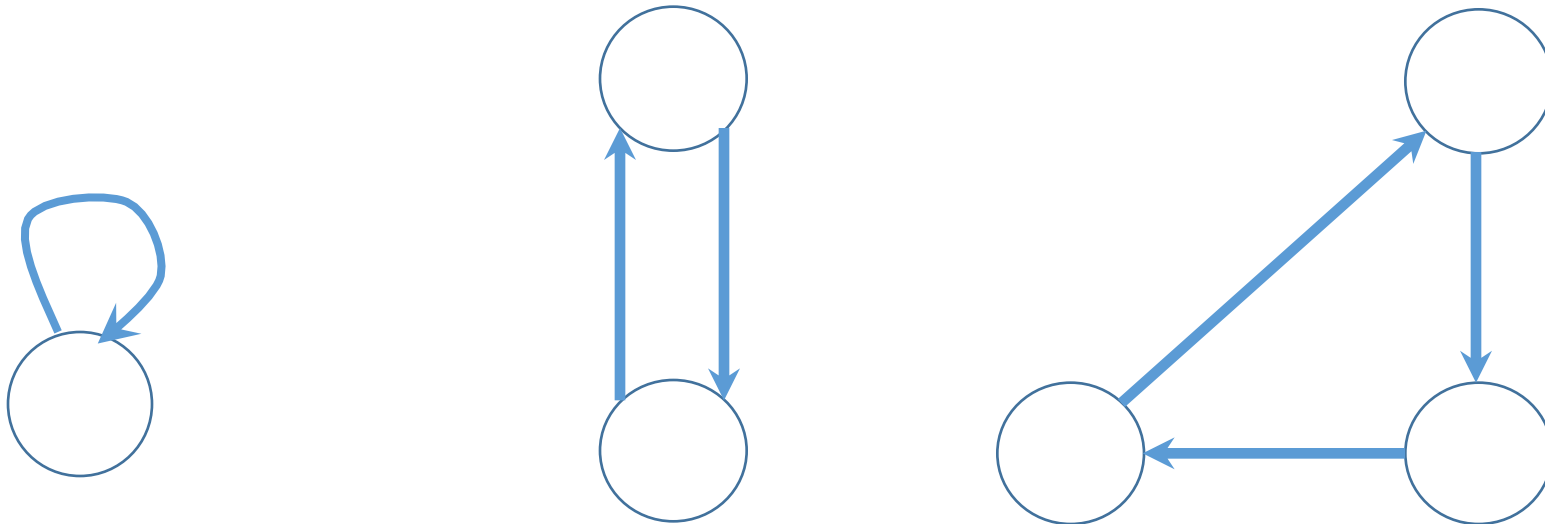
Also, having a **cross edge** is impossible since if the edge was undirected it could have been traversed in the other direction to result in a **tree edge.**

How to classify?

- We mainly care about finding **tree edges** and **back edges**.
- Assume that node **u is looking at node v** through the edge $e = (u, v)$ during the search:
 1. If **v is white** then e becomes a tree edge
 2. If **v is gray** then e is a back edge since v is an ancestor of u .

Finding cycles

- **Theorem:** A DFS on a directed or an undirected graph finds a **back edge** **if and only if** the graph **has a cycle**.
- **Note:** By cycle in a directed graph we mean directed cycle. Examples:



Finding cycles

- **Theorem:** A DFS on a directed or an undirected graph finds a **back edge** **if and only if** the graph **has a cycle**.

Proof: We argue for a directed graph and the same reasoning applies to an undirected graph as well.

1. If there is a back edge from u to v : Then, v must be an ancestor of u , or v is equal to u . In either case, there is a path from v to u , and an edge from u to v which forms a cycle.

Finding cycles

2. Let's say there is a cycle: assume that x is the first node in the cycle that is discovered by the DFS, and y_1, y_2, \dots, y_k are consecutive edges on the cycle such that y_k has an edge to x . Since DFS explores all nodes reachable from a certain node before it finishes the discovery of that node, then all nodes in the cycle are visited and finished before x can be finished (be marked black). As a result, also node y_k is visited and finished before x . When that happens the edge (y_k, x) will be a back edge since x is still gray (being explored) at that time.