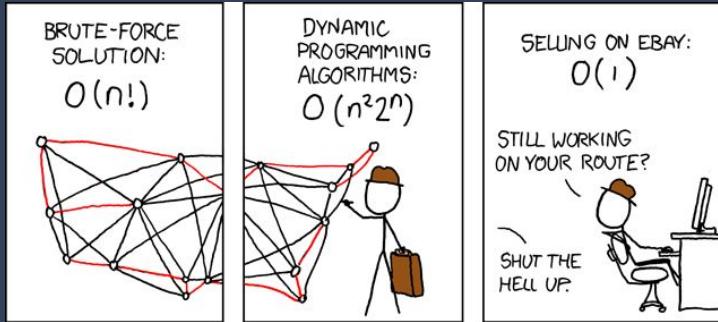


# Graphs Algorithms

{ik} INTERVIEW KICKSTART



(Some slides taken from CS161 @Stanford)

<https://xkcd.com/399/>

{ik} INTERVIEW KICKSTART

# Background

Ph.D in Computer Science from Stanford University (B.Tech in CS, IIT Delhi)

Co-taught CS374 (Algorithms in Biology) and CS262 (Computational Genomics)

## **Building, Maintaining, and Using Knowledge Bases: A Report from the Trenches**

(ACM SIGMOD)

Omkar Deshpande<sup>1</sup>, Digvijay S. Lamba<sup>1</sup>, Michel Tourn<sup>2</sup>,  
Sanjib Das<sup>3</sup>, Sri Subramaniam<sup>1</sup>, Anand Rajaraman, Venky Harinarayan, AnHai Doan<sup>1,3</sup>

<sup>1</sup>@WalmartLabs, <sup>2</sup>Google, <sup>3</sup>University of Wisconsin-Madison

## **Social Media Analytics: The Kosmix Story**

Xiaoyong Chai<sup>1</sup>, Omkar Deshpande<sup>1</sup>, Nikesh Garera<sup>1</sup>, Abhishek Gattani<sup>1</sup>, Wang Lam<sup>1</sup>,  
Digvijay S. Lamba<sup>1</sup>, Lu Liu<sup>1</sup>, Mitul Tiwari<sup>2</sup>, Michel Tourn<sup>3</sup>, Zoheb Vacheri<sup>1</sup>,  
STS Prasad<sup>1</sup>, Sri Subramaniam<sup>1</sup>, Venky Harinarayan<sup>4</sup>, Anand Rajaraman<sup>4</sup>,  
Adel Ardalan<sup>5</sup>, Sanjib Das<sup>5</sup>, Paul Suganthan G.C.<sup>5</sup>, AnHai Doan<sup>1,5</sup>

<sup>1</sup> @WalmartLabs, <sup>2</sup> LinkedIn, <sup>3</sup> Google, <sup>4</sup> Cambrian Ventures, <sup>5</sup> University of Wisconsin-Madison

**INTERVIEW  
KICKSTART**

(IEEE Data Eng Bulletin)

# Popular graph search algorithms for interviews

Breadth-first search (BFS)

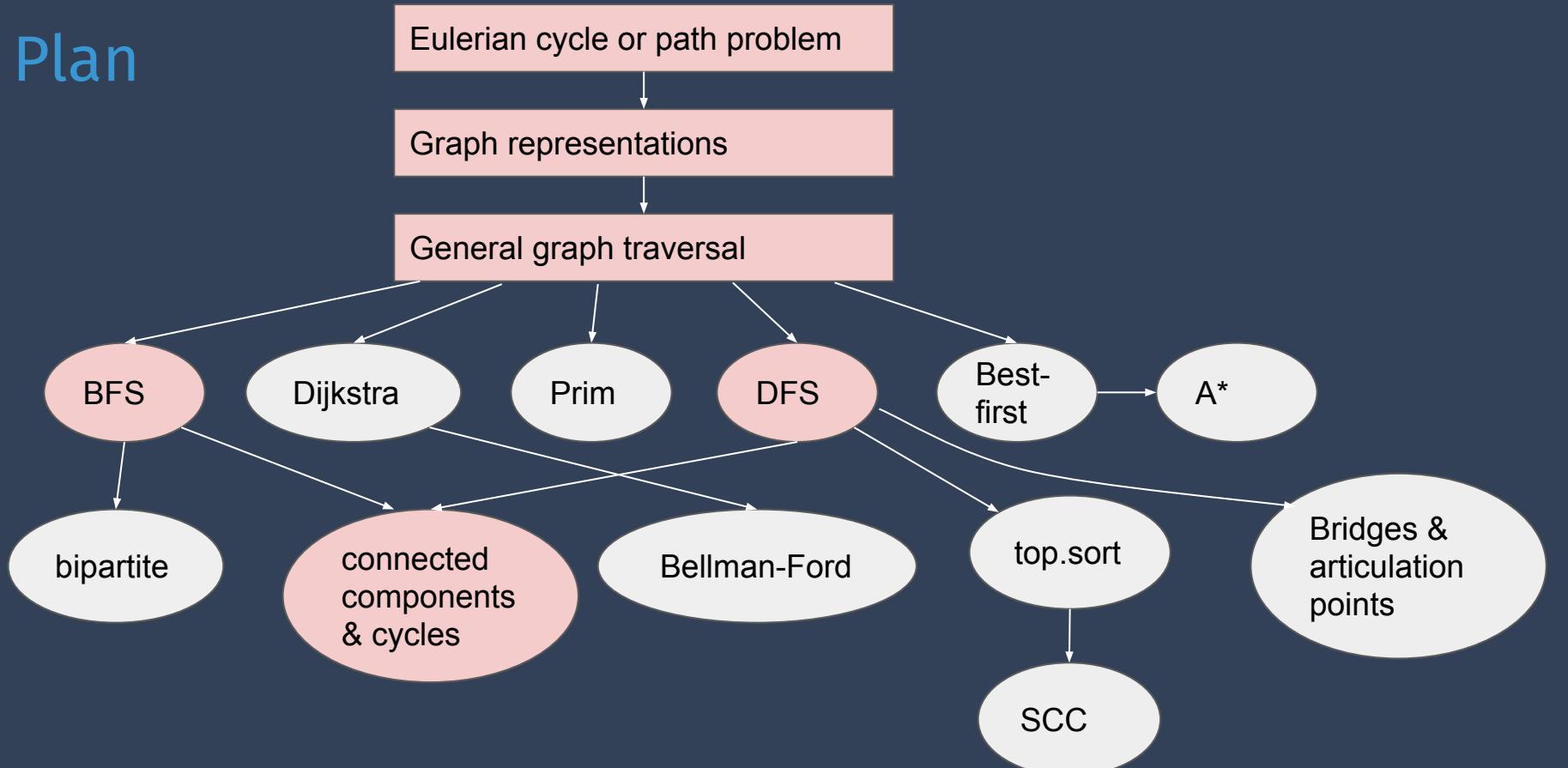
Depth-first search (DFS)

Prim's algorithm for Minimum Spanning Trees

Dijkstra's algorithm for shortest paths

Best-first search and A\*

# Plan



# Tackling interview questions on graphs

Step 1: Could the problem be modeled as a graph problem?

Step 2: Would a simple graph traversal help solve the problem?

Step 3: Would some popular standard extension of it help solve the problem?

Step 4: Code it up

MAN, YOU'RE BEING INCONSISTENT  
WITH YOUR ARRAY INDICES. SOME  
ARE FROM ONE, SOME FROM ZERO.

DIFFERENT TASKS CALL FOR  
DIFFERENT CONVENTIONS. TO  
QUOTE STANFORD ALGORITHMS  
EXPERT DONALD KNUTH,  
"WHO ARE YOU? HOW DID  
YOU GET IN MY HOUSE?"



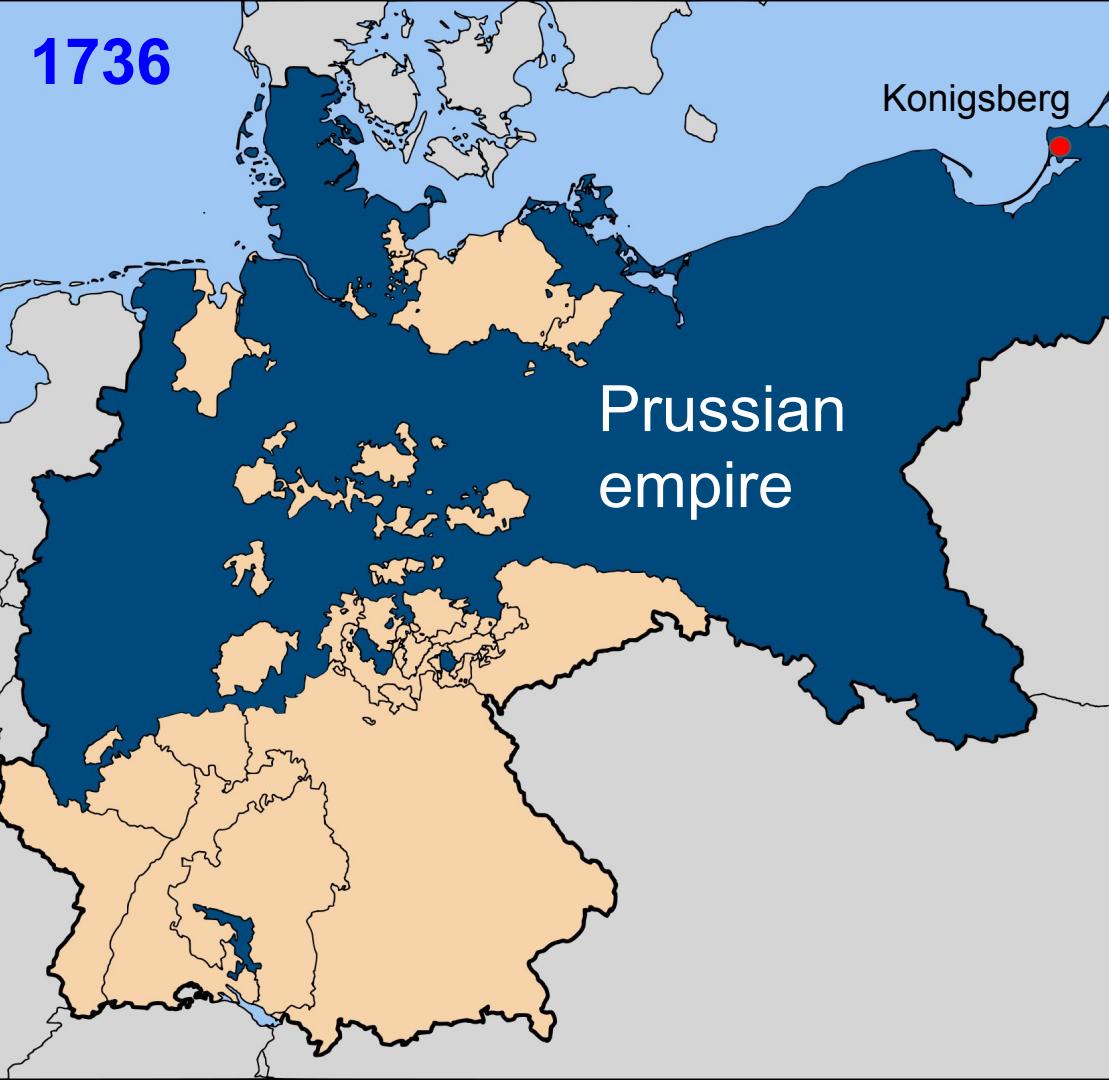
WAIT, WHAT?

WELL, THAT'S WHAT HE  
SAID WHEN I ASKED  
HIM ABOUT IT.



How it began: A person,  
a place, a problem and  
a paper

1736



{ik}

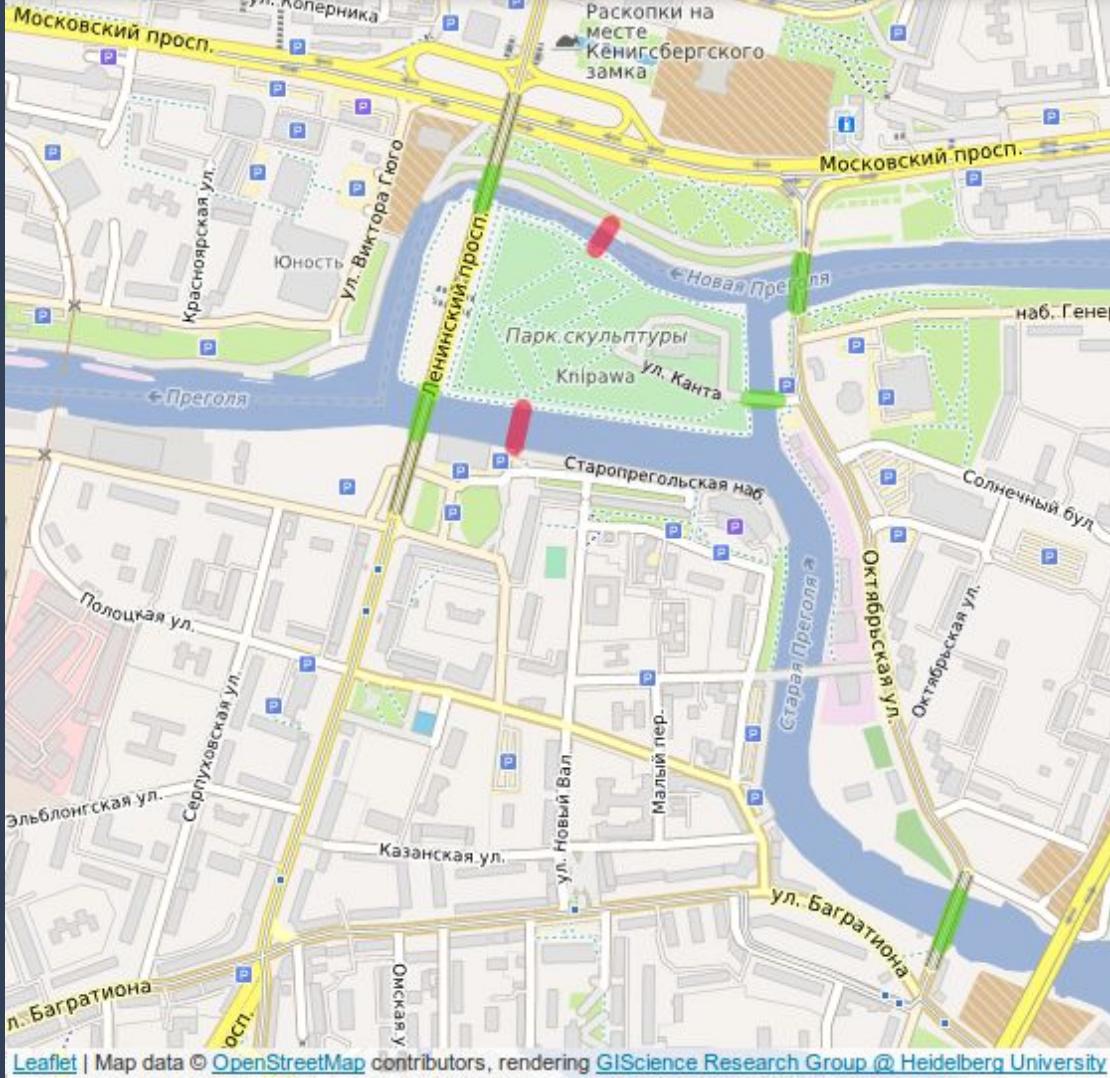
INTERVIEW  
KICKSTART

# Pregel river



{ik}

INTERVIEW  
KICKSTART



Leaflet | Map data © OpenStreetMap contributors, rendering GIScience Research Group @ Heidelberg University

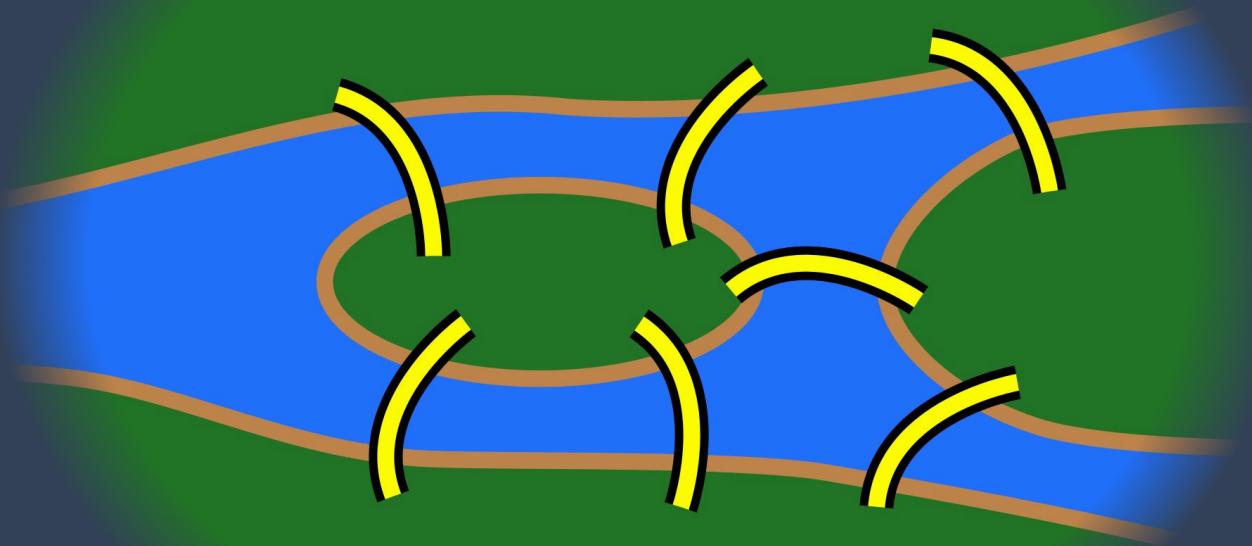
INTERVIEW  
KICKSTART

Can one walk across the  
seven bridges without  
crossing the same  
bridge twice?

“You would render to me and our friend Kuhn a most valuable service, putting us greatly in your debt, most learned sir, if you would send us the solution, which you know well, to the problem of the seven Konigsberg bridges together with a proof. It would prove to an outstanding example of the calculus of position worthy of your great genius. I have added a sketch of the said bridges”.

(Letter from the Mayor to Euler)



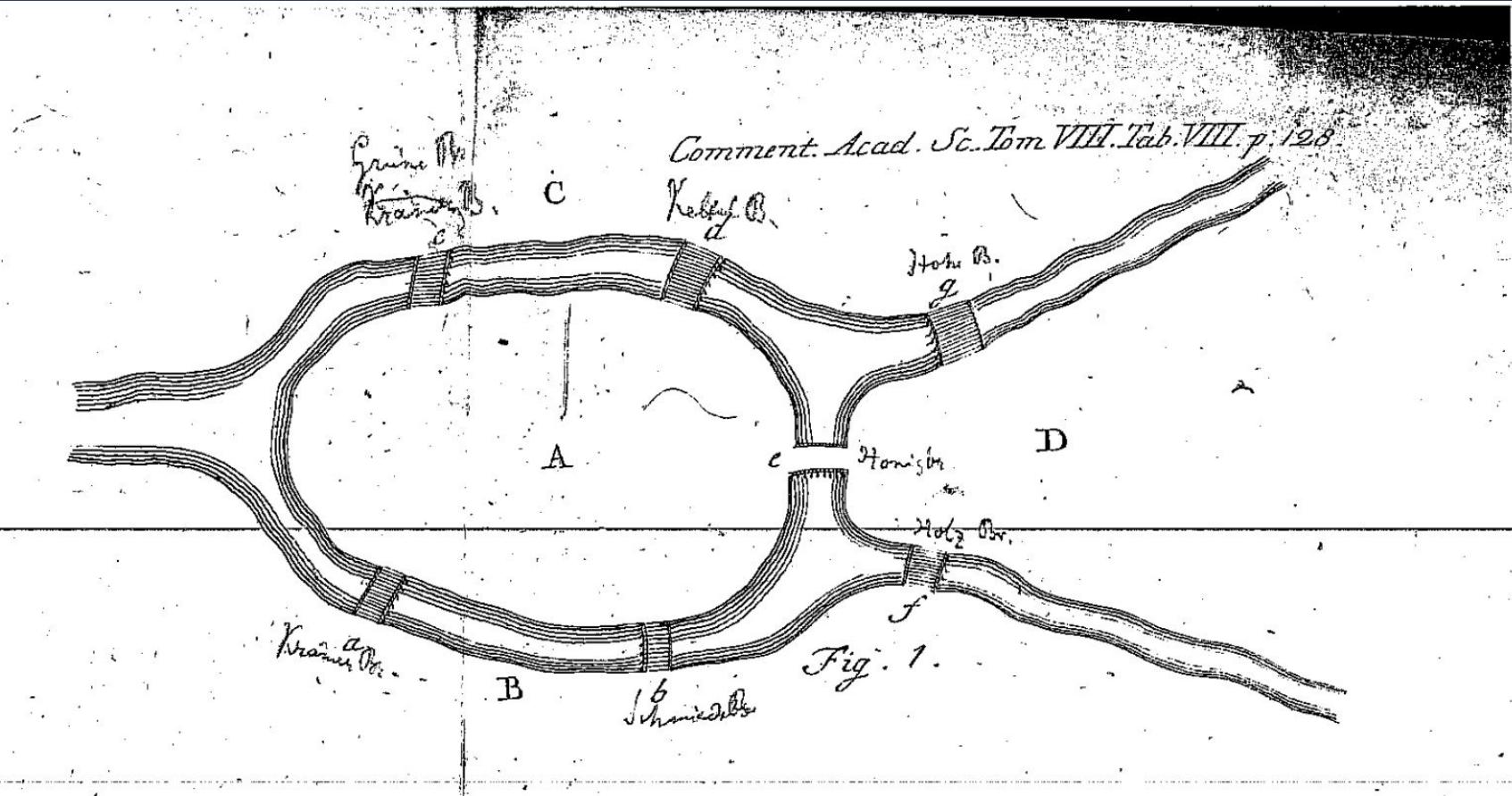


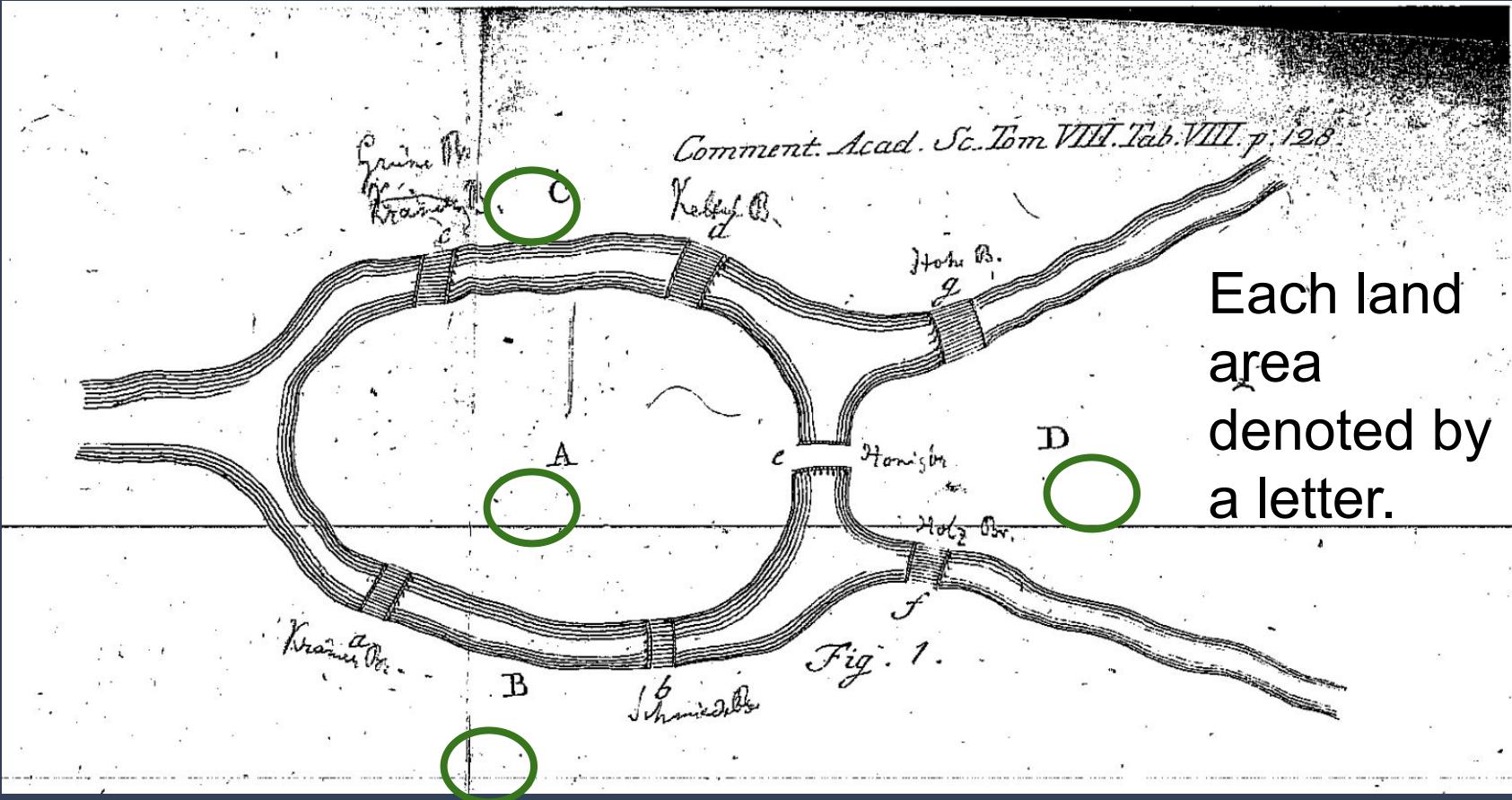
{ik} INTERVIEW  
KICKSTART

“This question is so banal, but seemed to me worthy of attention, in that neither geometry, nor algebra, nor the art of counting was sufficient to solve it. In view of this, it occurred to me to wonder whether it belonged to the geometry of position....”.

(Euler's letter to an Italian mathematician)







Each land  
area  
denoted by  
a letter.

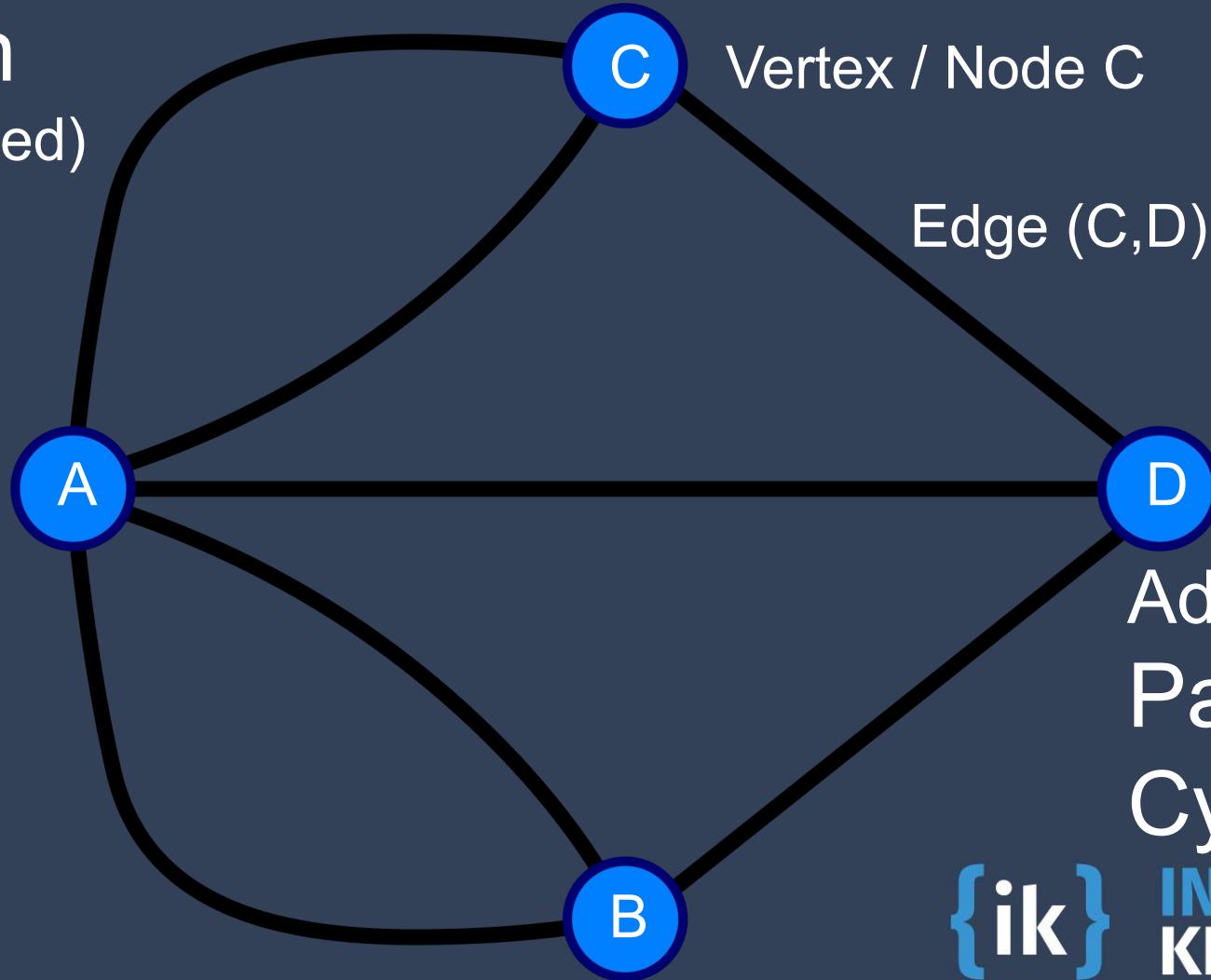
“For this I use the capital letters A, B, C, D, for each of the land areas separated by the river. If a traveller goes from A to B over bridge a or b, I write this as AB — where the first letter refers to the area the traveller is leaving, and the second refers to the area he arrives at after crossing the bridge.”

(Euler's 1736 paper)

Tip: Check if the given problem can be modeled as a graph.



# Graph (undirected)



“After some deliberation, I obtained a simple rule with whose help one can immediately decide for all examples of this kind, with any number of bridges in any arrangement, whether such a round-trip is possible”.

(Letter from Euler to an Italian mathematician)

Tip: Make sure your solution works for a general instance, not just the example given.

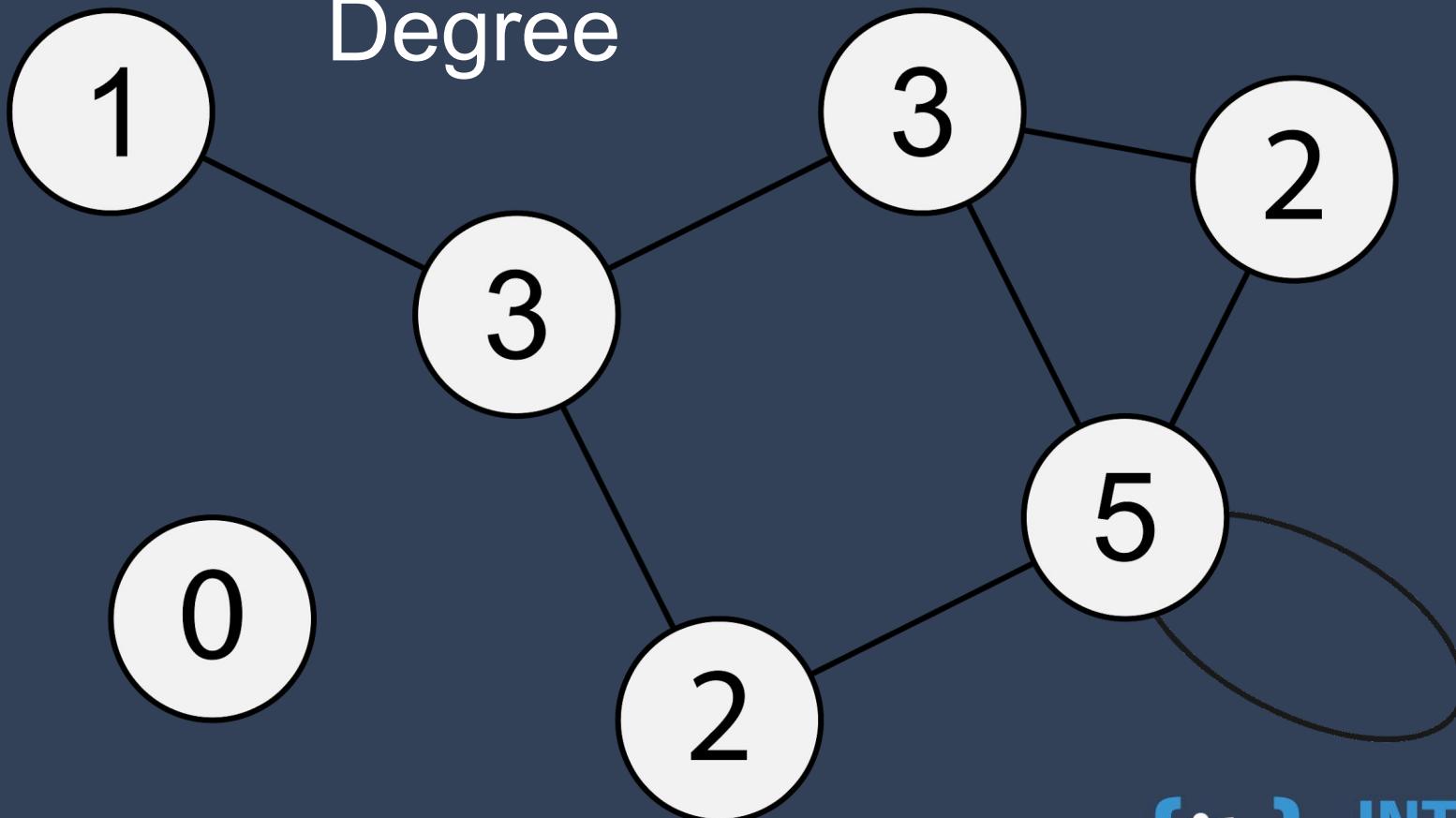


“As far as the problem of the seven bridges of Konigsberg is concerned, it can be solved by making an exhaustive list of all possible routes, and then finding whether or not any route satisfies the conditions of the problem. Because of the number of possibilities, this method of solution would be too difficult and laborious, and in other problems with more bridges it would be impossible”.

Tip: You may start with a brute force solution but need to go beyond it.



Degree



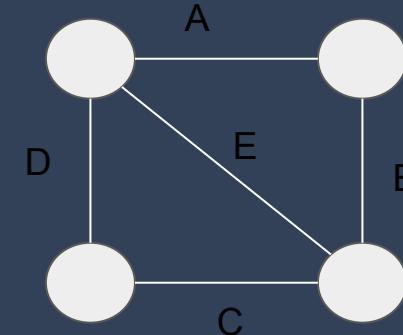
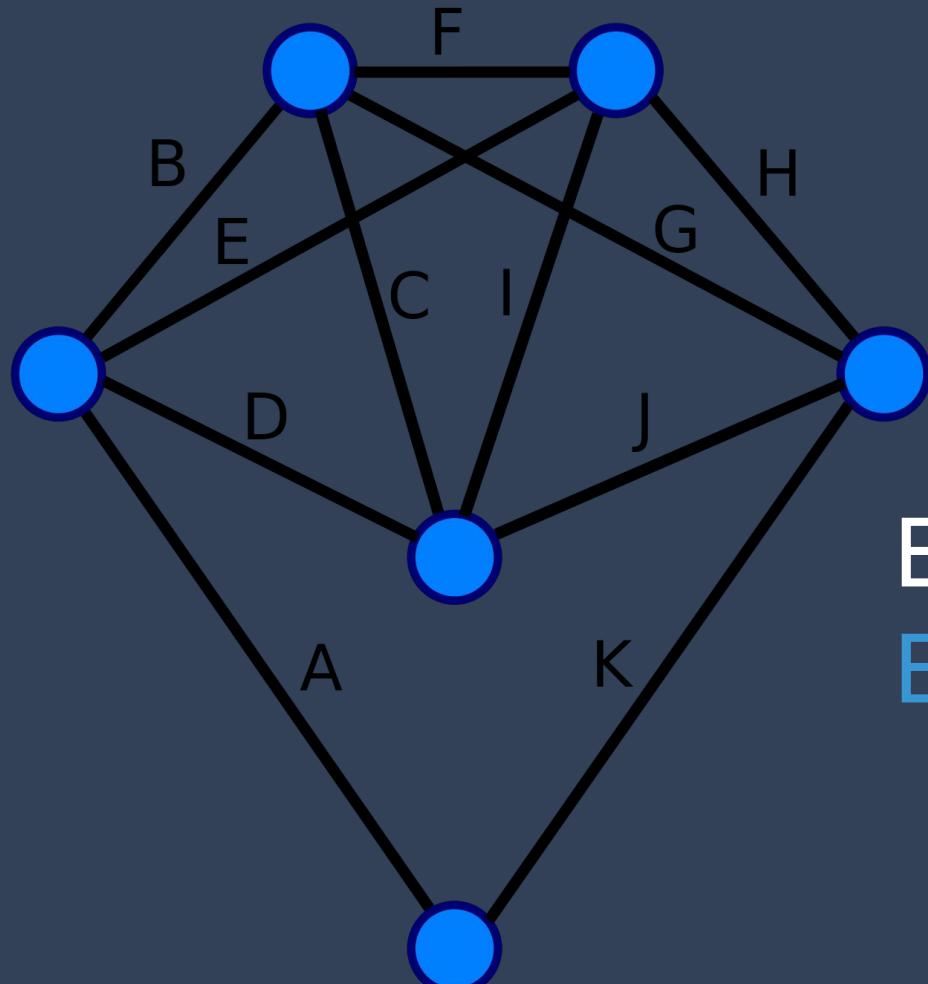
The sum of the degrees of all the vertices in an undirected graph  $G = (V, E)$  is

1.  $|E|$
2.  $|V|$
3.  $2|E|$
4.  $2|V|$

What about directed graphs?

“I observe that the numbers of bridges written next to the letters A, B, C, etc. together add up to twice the total number of bridges. The reason for this is that, in the calculation where every bridge leading to a given area is counted, each bridge is counted twice, once for each of the two areas which it joins.”





Eulerian path  
Eulerian cycle

# Optimal Museum Traversal Using Graph Theory

Mitchell Eithun  
Michigan State University  
[eithunmi@msu.edu](mailto:eithunmi@msu.edu)

August 8, 2018



The Egyptian Room at the British Museum, 1844 [1].

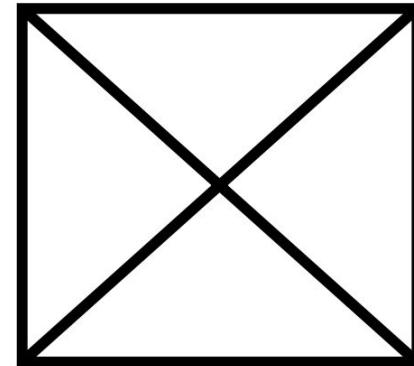
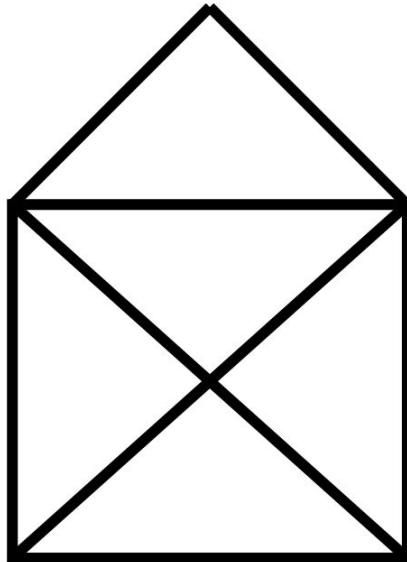
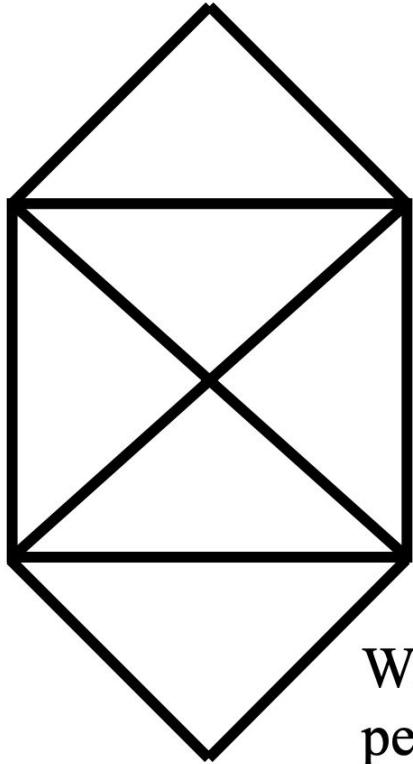
# The snow ploughing problem solved by a graph theory algorithm

Article · December 1984 with 143 Reads

DOI: [10.1080/02630258408970368](https://doi.org/10.1080/02630258408970368)

"An algorithm to trace the itinerary of a snow ploughing truck that respects street priorities is developed. It is based on the notion of an Eulerian circuit... "

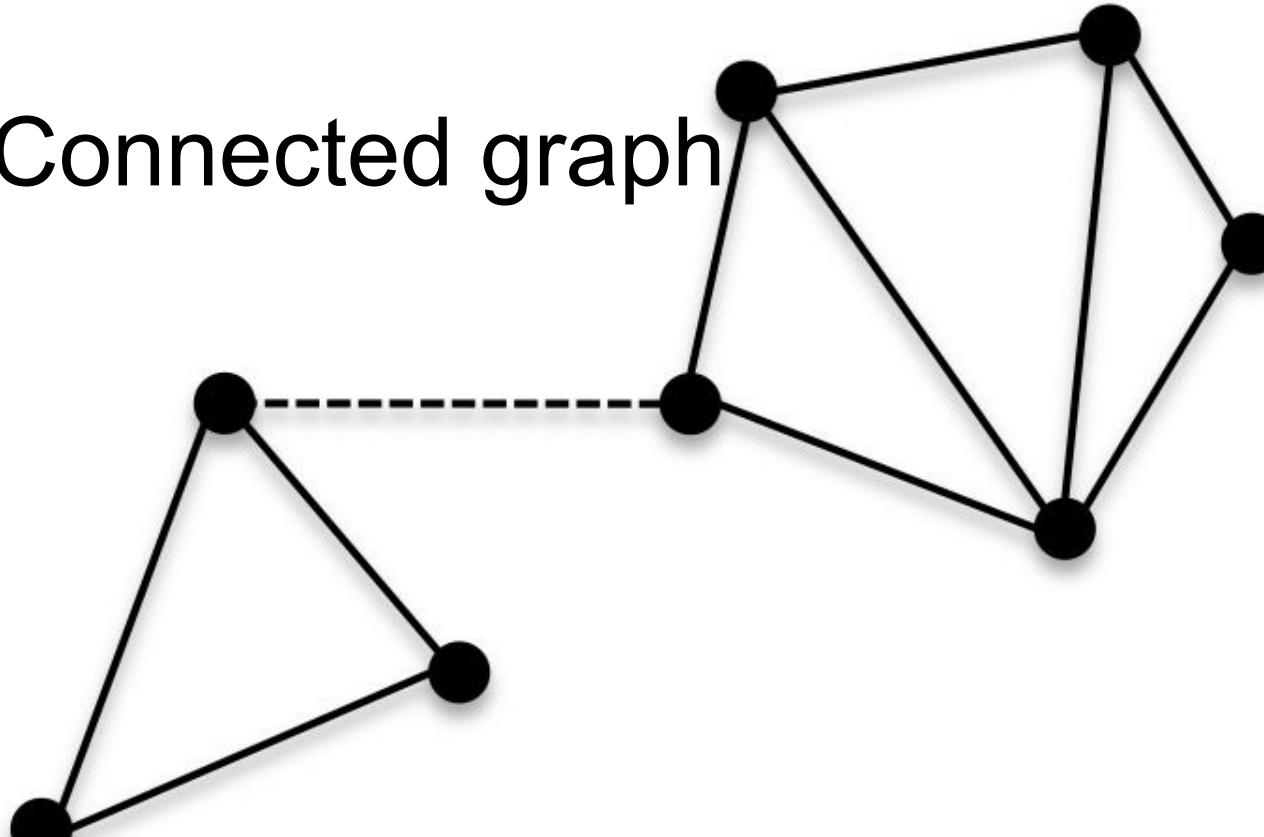




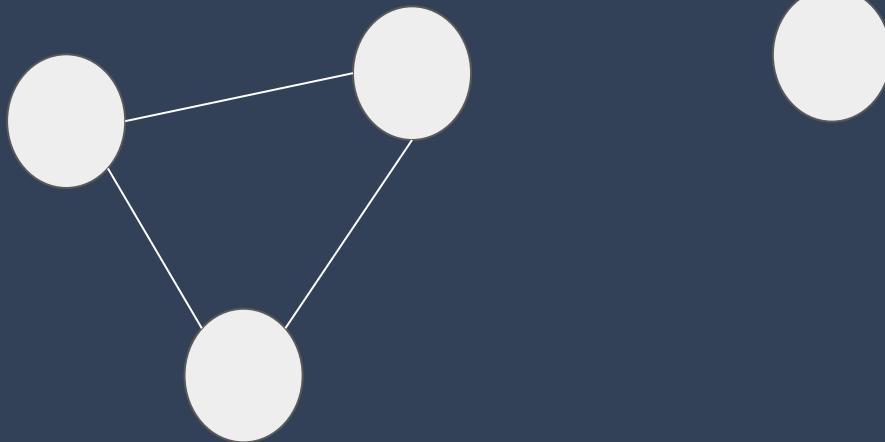
Which of these can you draw without lifting your pencil, drawing each line only once?  
Can you start and end at the same point?

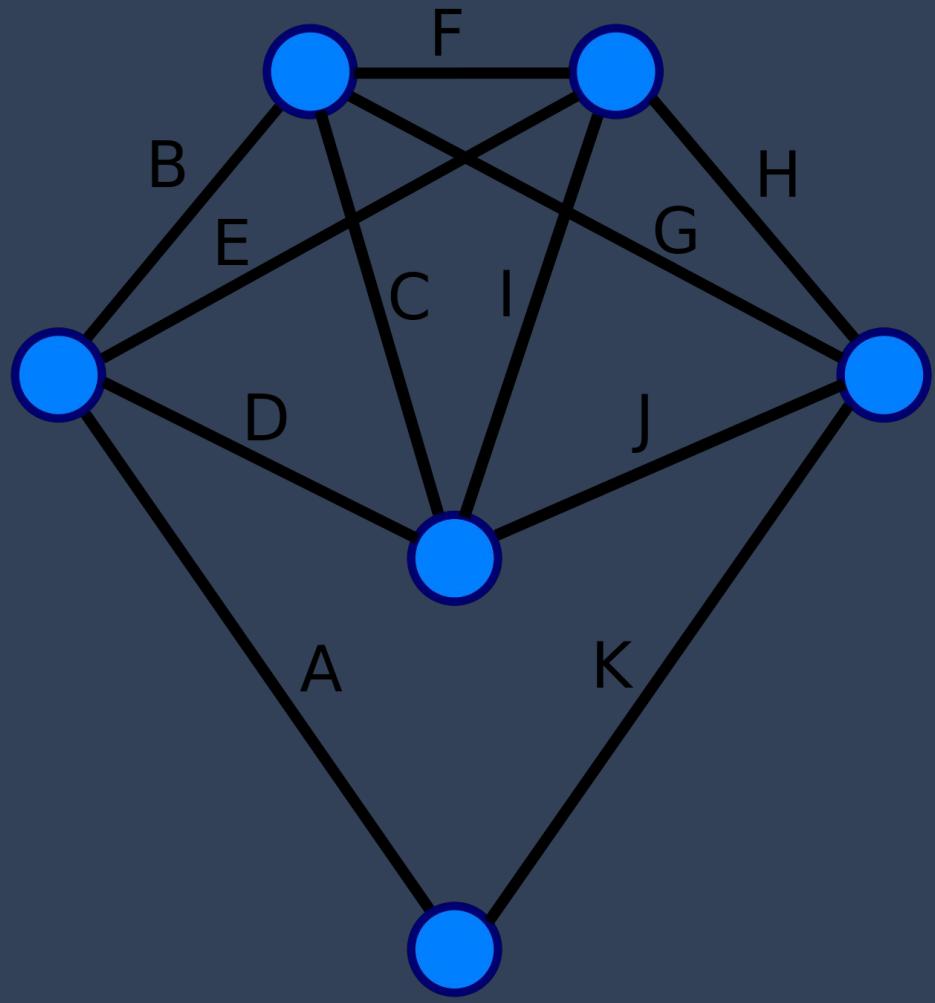
Different word  
problems can map to  
the same graph  
problem.

# Connected graph



If a graph has an  
Eulerian cycle,  
it must be connected?





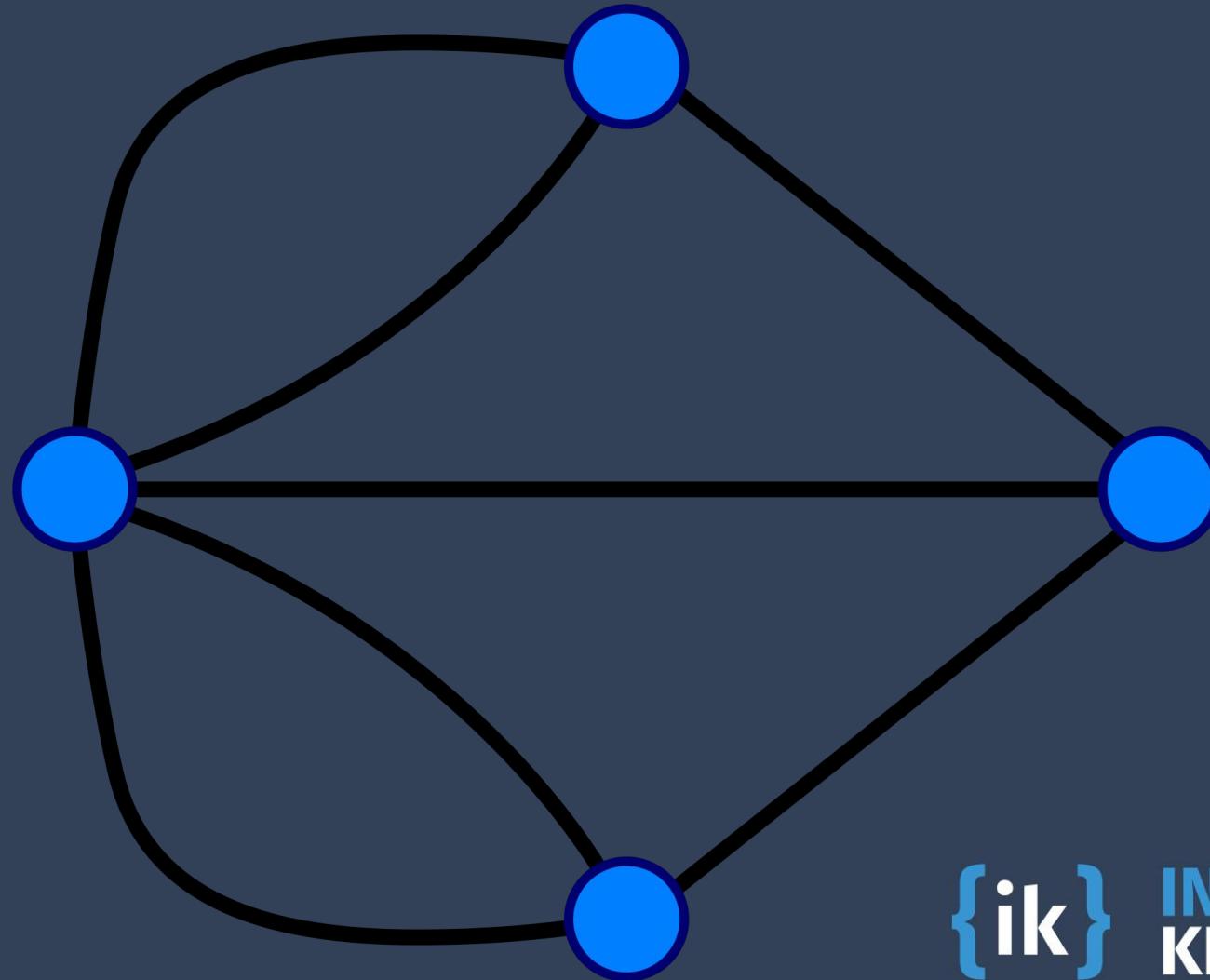
**Observation: Every time  
we enter a vertex, we  
exit it along some  
unused edge.**

**Observation: Every time we enter a vertex, we can exit it along some unused edge.**

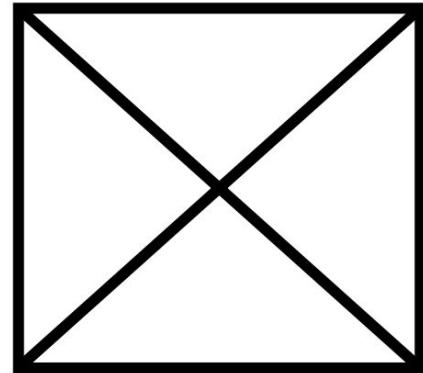
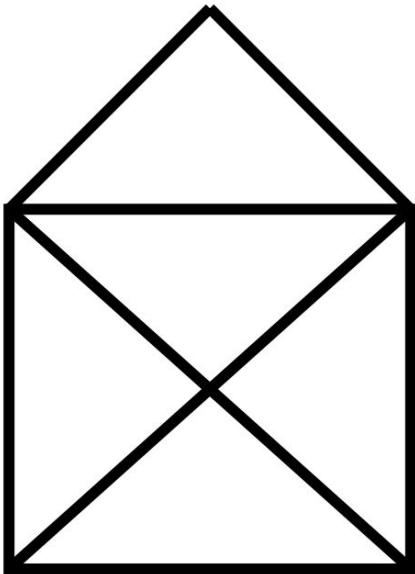
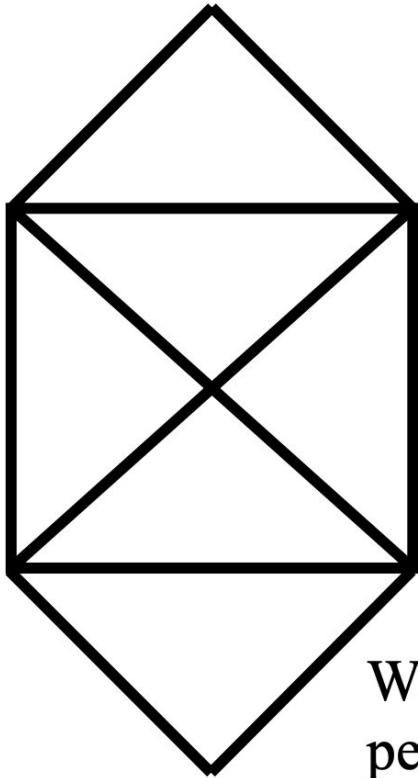
**So the degree of every vertex must be**

**even.**

If the degree of any vertex is odd, the graph cannot have an Eulerian cycle.



{ik} INTERVIEW  
KICKSTART



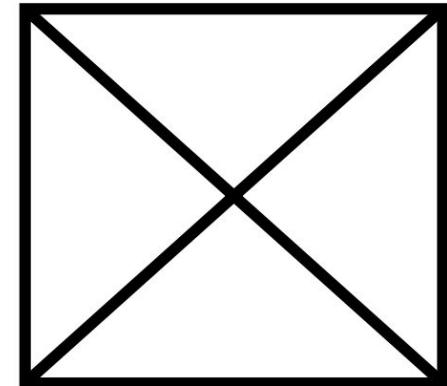
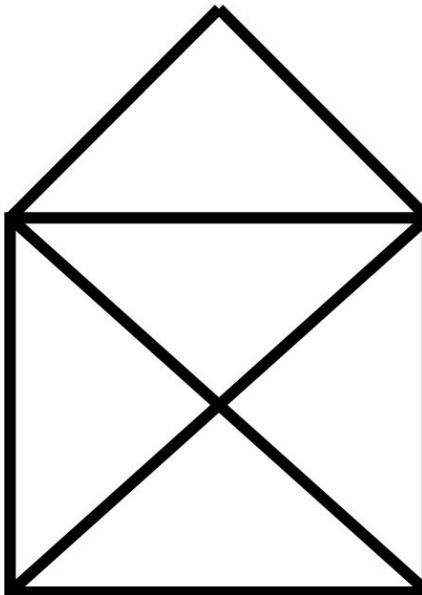
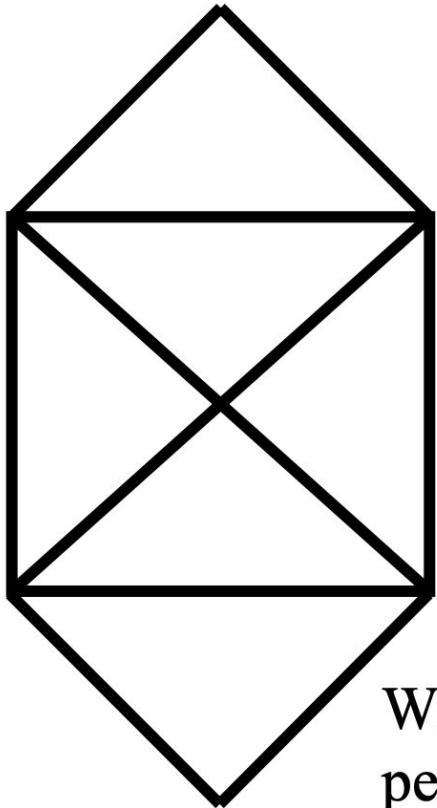
Which of these can you draw without lifting your pencil, drawing each line only once?  
Can you start and end at the same point?

If a graph is connected,  
and the degree of every  
vertex is even, does it  
have an Eulerian cycle?

If a graph is connected,  
and the degree of every  
vertex is even, does it  
have an Eulerian cycle?

Yes!

A connected graph has  
an Eulerian cycle if and  
only if the degree of  
every vertex is even.



Which of these can you draw without lifting your pencil, drawing each line only once?  
Can you start and end at the same point?

# What about Eulerian paths?

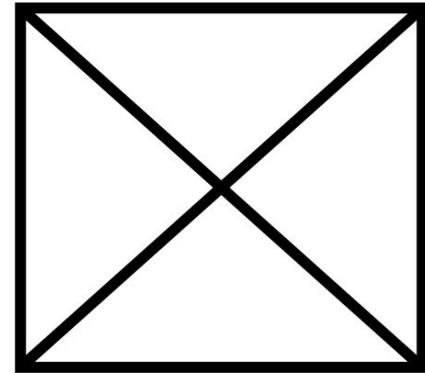
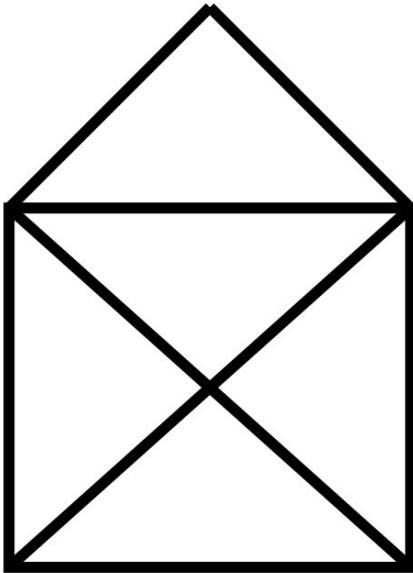
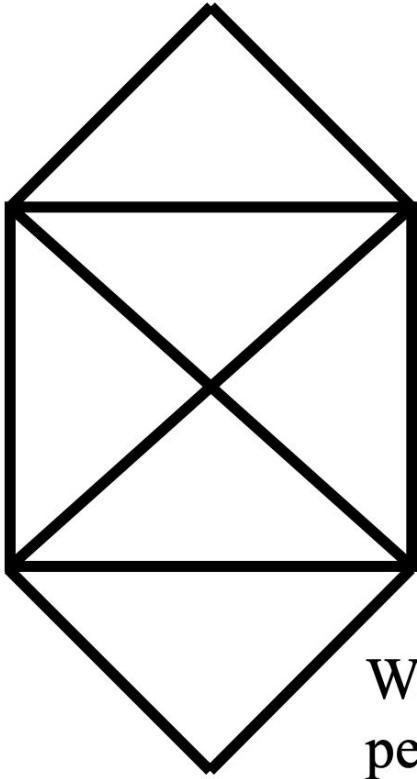
If there exists an Eulerian path in a connected graph, either that path is a cycle, or it starts and ends at two different vertices.

If there exists an Eulerian path in a connected graph, either that path is a cycle, or it starts and ends at two different vertices. These are the only two vertices that would have an odd degree.

If a connected graph has exactly two vertices with an odd degree, does there necessarily exist an Eulerian path in it?

If a connected graph has exactly two vertices with an odd degree, does there necessarily exist an Eulerian path in it?

Yes! Suppose you added a new edge to link those two vertices together. The resulting graph would have an Eulerian cycle. If we then remove the edge we had added, we would get an Eulerian path.



Which of these can you draw without lifting your pencil, drawing each line only once?  
Can you start and end at the same point?

“If there are more than two areas to which an odd number of bridges lead, then such a journey is impossible.

If, however, the number of bridges is odd for exactly two areas, then the journey is possible if it starts from either of these areas.

If, finally, there are no areas to which an odd number of bridges lead, then the required journey can be accomplished from any starting point.

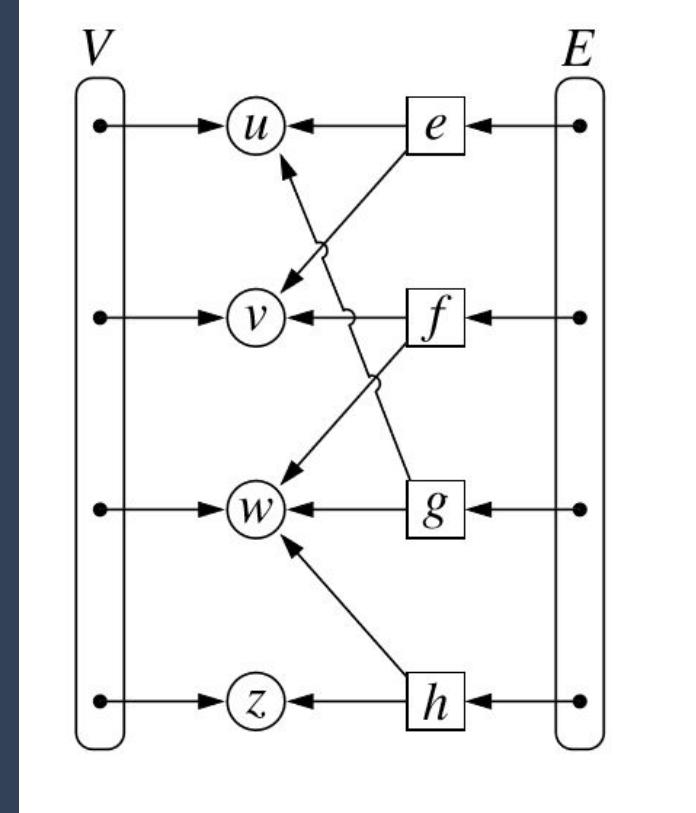
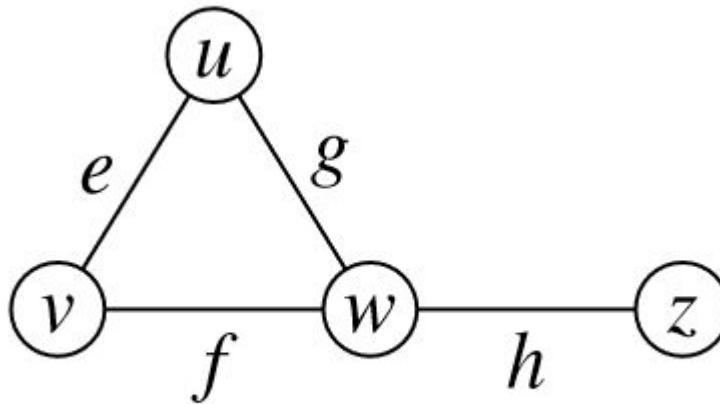
With these rules, the given problem can also be solved.”



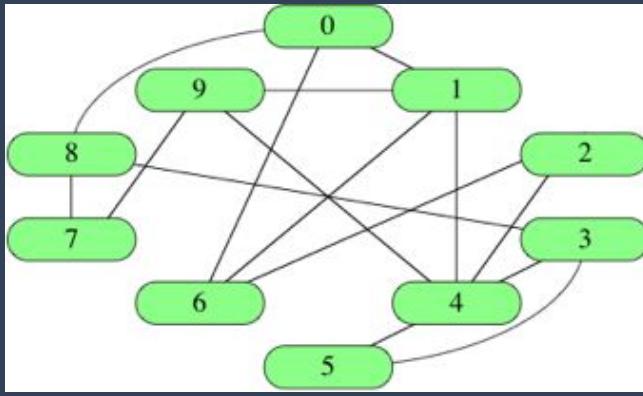
# Graph representation

# Edge lists

Since  $G = (V, E)$ , why not maintain  $V$  and  $E$  as lists?



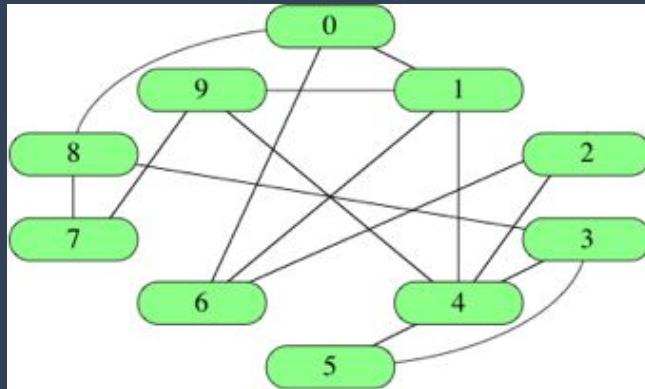
# Adjacency list



0	1	6	8
1	0	4	6
2	4 6		
3	4 5		8
4	1	2	3
5	5 9		
6	3	4	
7	0	1	2
8	8 9		
9	0	3	7
	1	4	7

Applies to both undirected and directed graphs.

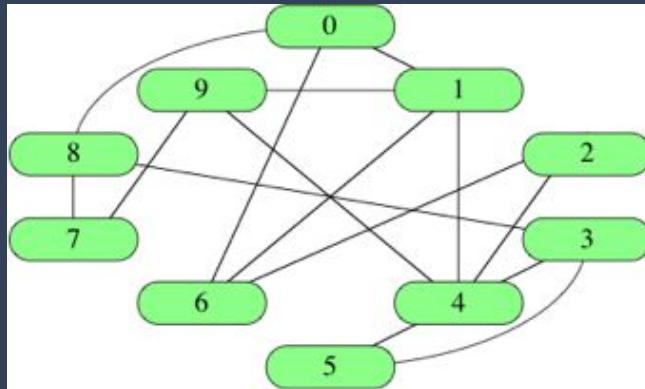
# Adjacency matrix



	0	1	2	3	4	5	6	7	8	9
0	0	1	0	0	0	0	1	0	1	0
1	1	0	0	0	1	0	1	0	0	1
2	0	0	0	0	1	0	1	0	0	0
3	0	0	0	0	1	1	0	0	1	0
4	0	1	1	1	0	1	0	0	0	1
5	0	0	0	1	1	0	0	0	0	0
6	1	1	1	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	1
8	1	0	0	1	0	0	0	1	0	0
9	0	1	0	0	1	0	0	1	0	0

Applies to both undirected and directed graphs.

# Adjacency matrix



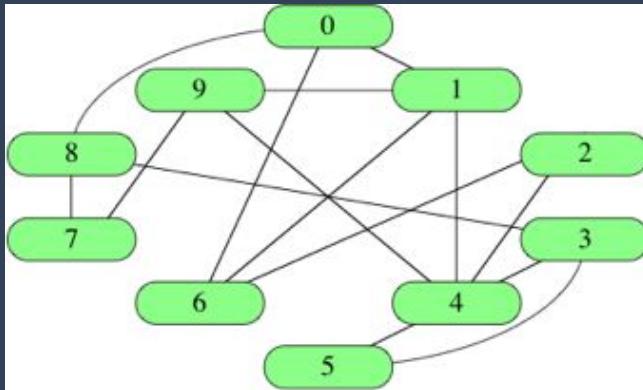
	0	1	2	3	4	5	6	7	8	9
0	0	1	0	0	0	0	1	0	1	0
1	1	0	0	0	1	0	1	0	0	1
2	0	0	0	0	1	0	1	0	0	0
3	0	0	0	0	1	1	0	0	1	0
4	0	1	1	1	0	1	0	0	0	1
5	0	0	0	1	1	0	0	0	0	0
6	1	1	1	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	1
8	1	0	0	1	0	0	0	1	0	0
9	0	1	0	0	1	0	0	1	0	0

Needs  $O(|V|^2)$  space.

Good for **dense** graphs (in which  $|E| \sim |V|^2$ )

Only one bit per entry sufficient

# Adjacency list



0	1	6	8
1	0	4	6
2	4 6		
3	4 5		8
4	1	2	3
5	4	5	9
6	3	4	
7	0	1	2
8	8	9	
9	0	3	7
	1	4	7

Needs  $O(|V| + |E|)$  space

Good for **sparse** graphs (in which  $|E| \ll |V|^2$ )

Majority of algorithms use this representation

What if we need to quickly tell if there is an edge  $(u,v)$  in the graph?

What if each edge has an associated weight? How would we incorporate that?

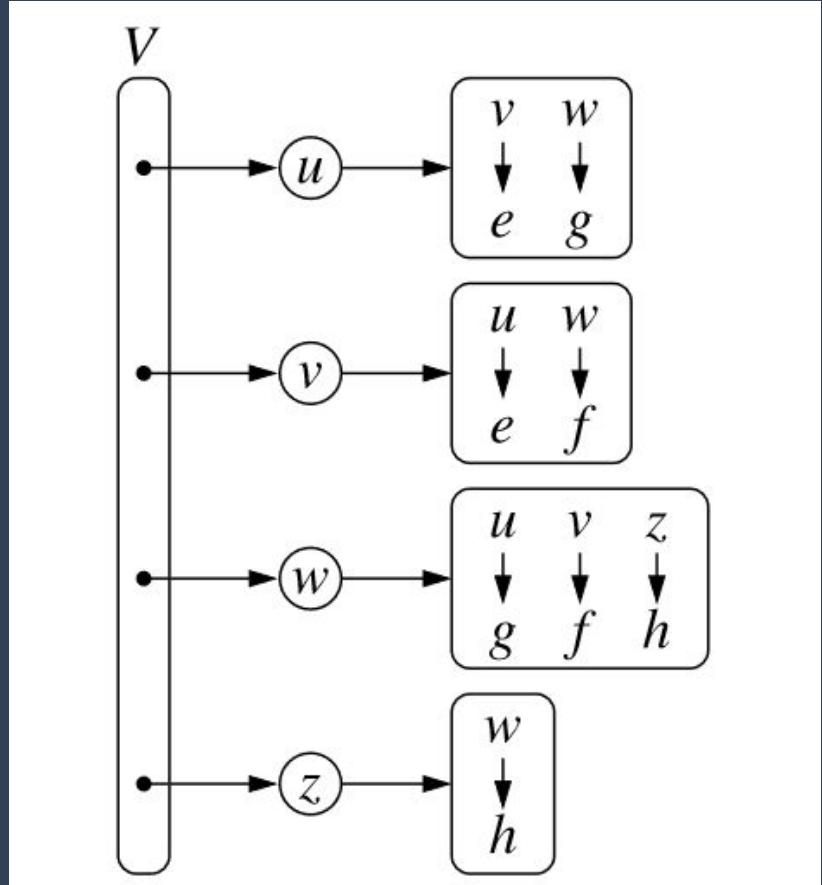
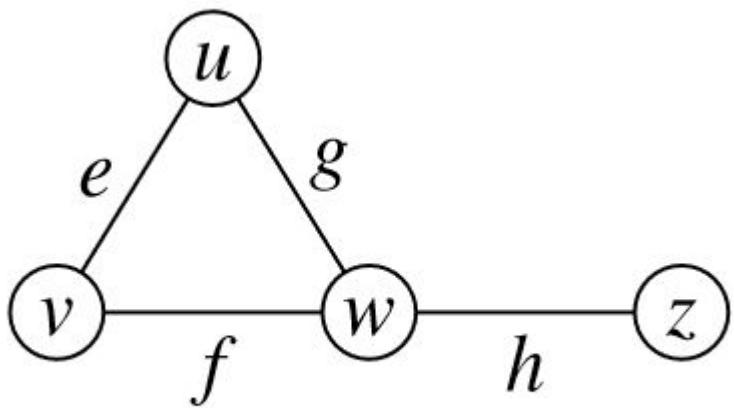
# Adjacency maps

Adjacency map uses a dictionary (hash table) to store the neighbors of vertex  $v$ , instead of a list of neighbors (as in adjacency list).

This allows you to not only iterate over the neighbors if you wish (like in adjacency list), but also find out in  $O(1)$  expected time if  $w$  is a neighbor of  $v$  and to get the edge weight of  $(v,w)$  if applicable.

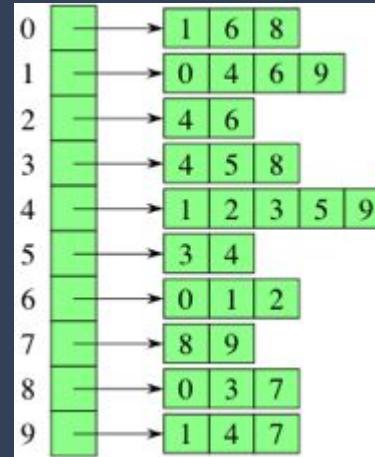
Adjacency matrices provide that lookup in  $O(1)$  time, so in some sense, Adjacency map combines the advantages of Adjacency lists (in space) and Adjacency matrices (in time).

# Adjacency map



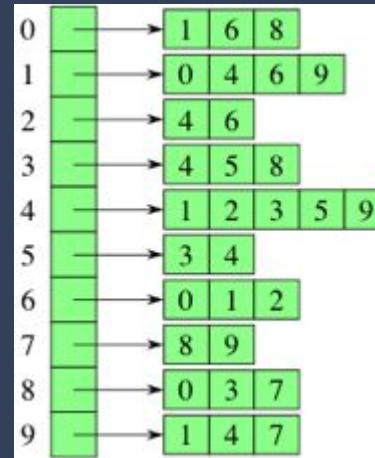
# Adjacency list

```
class Graph {  
}  
}
```



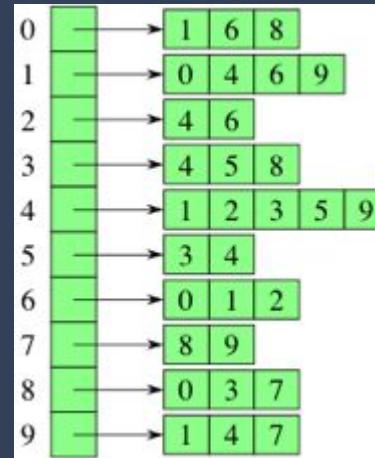
# Adjacency list

```
class Graph {  
    adjList;  
}
```



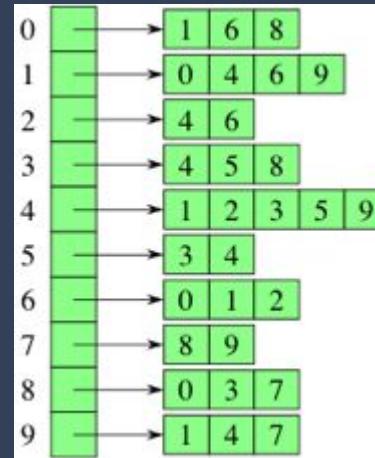
# Adjacency list

```
class Graph {  
    list [ ] adjList;  
}
```



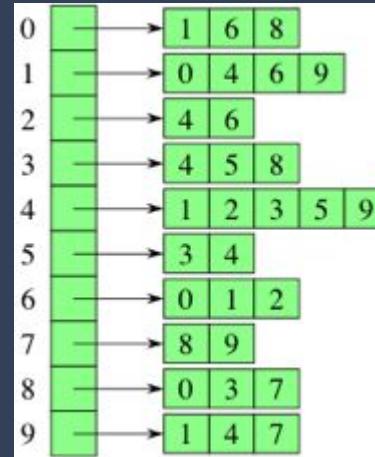
# Adjacency list

```
class Graph {  
    list<int> [ ] adjList;  
}
```



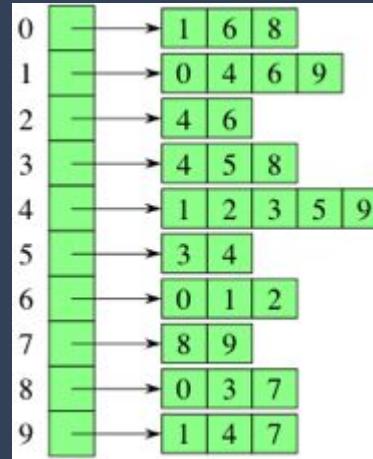
# Adjacency list

```
class Graph {  
    list<int> [ ] adjList;  
  
    int V;  
  
}
```



# Adjacency list

```
class Graph {  
  
    list<int> [ ] adjList;  
  
    int V;  
  
    Graph(int size) {  
  
        V = size;  
  
        adjList = new list<int>[V];  
  
    }  
  
}
```



g = new Graph(10);

```
class Graph {
```

```
    list<int> [ ] adjList;
```

```
    int V;
```

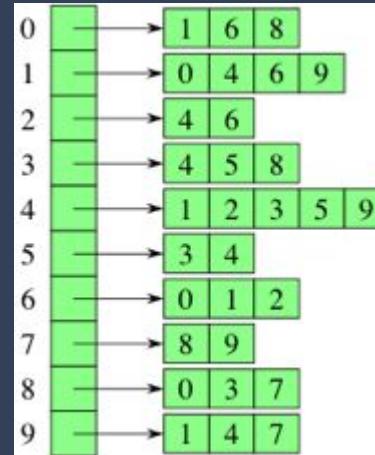
```
    void addEdge(int start, int end, bidir=true) {
```

```
        adjList[start].add(end);
```

```
        if (bidir==true)
```

```
            adjList[end].add(start);
```

# Adjacency list



```
}
```

{ik}

INTERVIEW  
KICKSTART

```
class Graph {
```

```
list<int> [ ] adjList;
```

```
int V;
```

```
void addEdge(int start, int end, bidir=true) {
```

```
adjList[start].add(end);
```

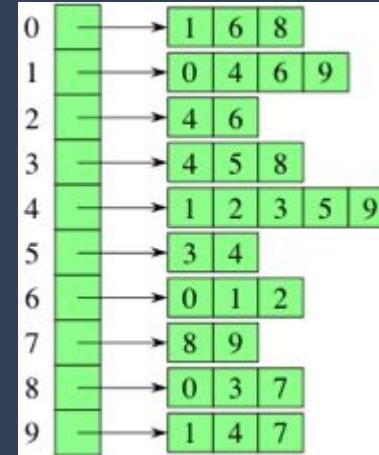
```
if (bidir==true)
```

```
adjList[end].add(start);
```

```
}
```

```
...
```

# Adjacency list



```
g.addEdge(0,1);
```

```
g.addEdge(0,6);
```

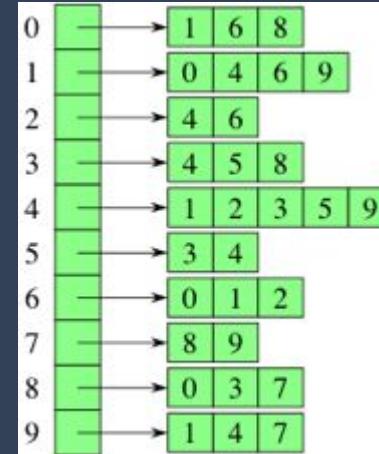
```
g.addEdge(0,8);
```

```
g.addEdge(1,4);
```

```
.....
```

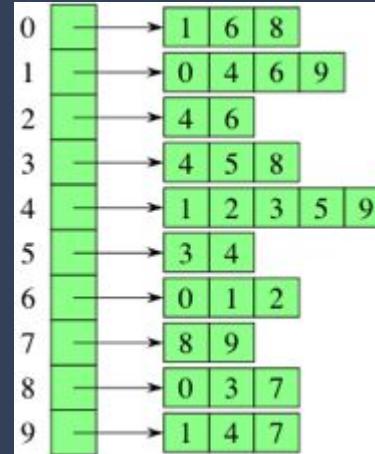
```
class Graph {  
    list<int> [ ] adjList;  
  
    int V;  
    ...  
    bool hasEulerianCycle() {  
  
        odd = 0; //no. of vertices with odd degree  
        for vertex in adjList:  
            if (adjList[vertex].size() mod 2) == 1:  
                odd++;  
        if (odd == 0) return True;  
        else return False;  
  
    }  
}
```

# Adjacency list



```
class Graph {  
    list<int> [ ] adjList;  
  
    int V;  
  
    bool hasEulerianPath() {  
  
        odd = 0; //no. of vertices with odd degree  
        for vertex in adjList:  
            if (adjList[vertex].size() mod 2) == 1:  
                odd++;  
            if (odd == 0) or (odd == 2) return True;  
            else return False;  
  
    }  
}
```

# Adjacency list



# What if we had used an adjacency matrix?

Most graph algorithms, including the previous one, take time at least  $O(|V|^2)$  with adjacency matrices.

But think about the following problem -

Determine whether a directed graph  $G = (V, E)$  contains a **universal sink**, i.e, a vertex with in-degree  $= |V| - 1$  and out-degree 0.

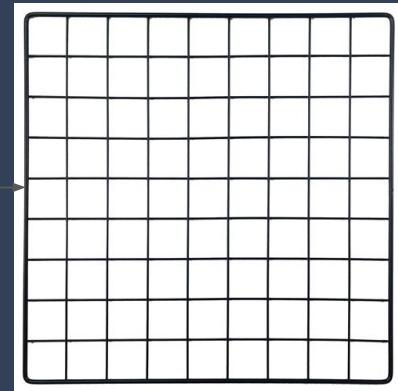
# Implicit representations of edges

In some cases, we don't need to use adjacency lists/maps to store the neighbors of a vertex. The neighbors can be dynamically calculated (by a *getNeighbors* function) when the need arises. This typically happens when the given graph is structured like a *grid*, with rows and columns of vertices.

This approach saves us space without sacrificing time.

E.g, City Map

Every vertex has 4 neighbors unless it is close to the grid boundary.

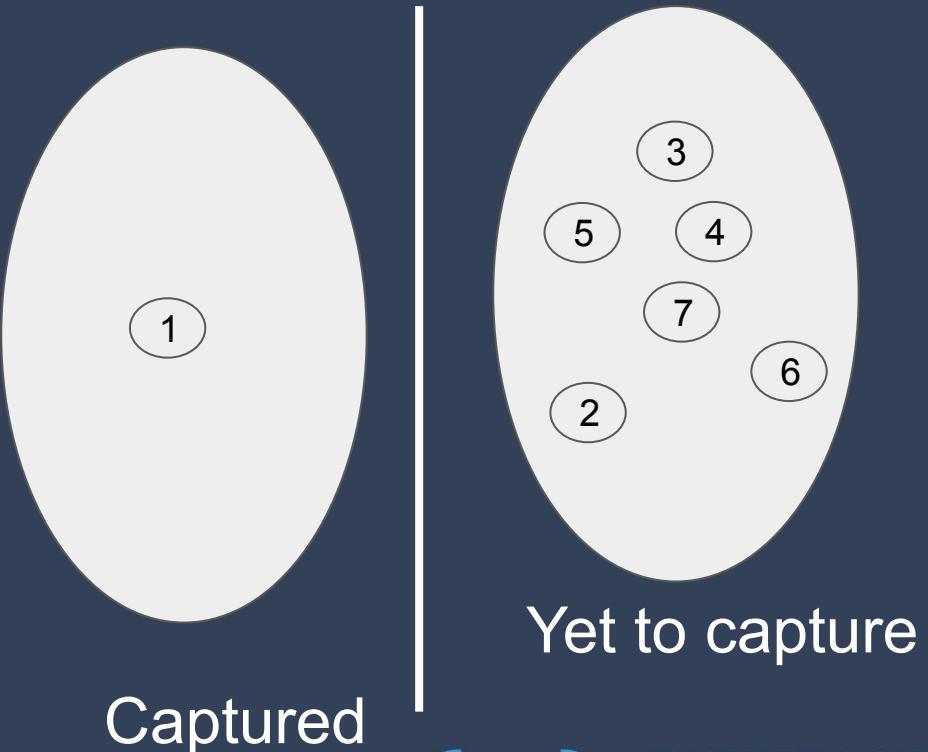
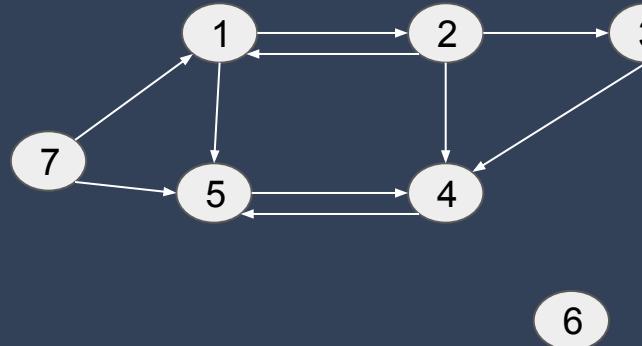


**Problem:** We assumed the graph is connected. If it is not connected, we need to check that all the edges belong to the same connected component.

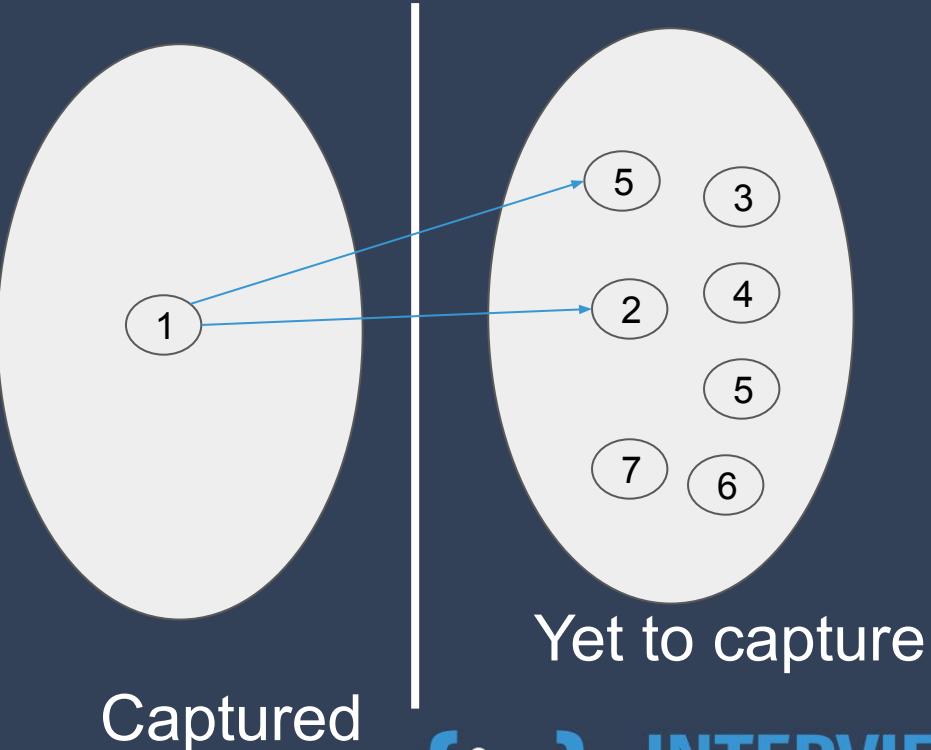
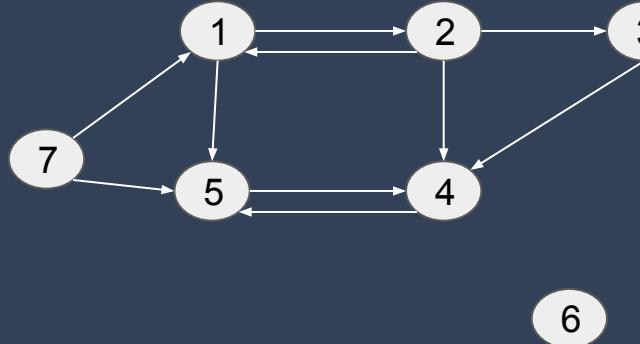
For that, we need to find a way to explore (traverse) a graph *starting from a source vertex s.*

Find all reachable vertices from s.

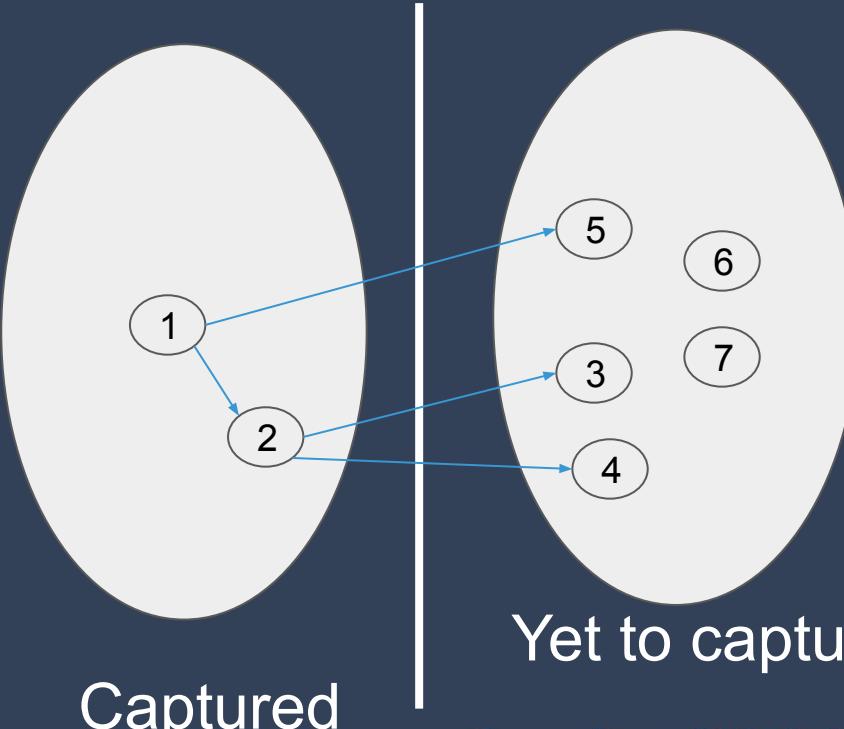
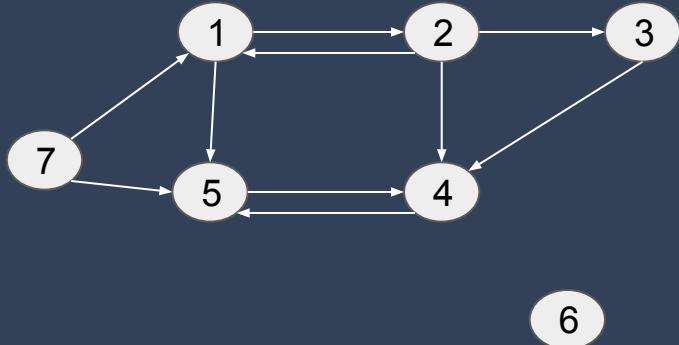
# Skeleton search



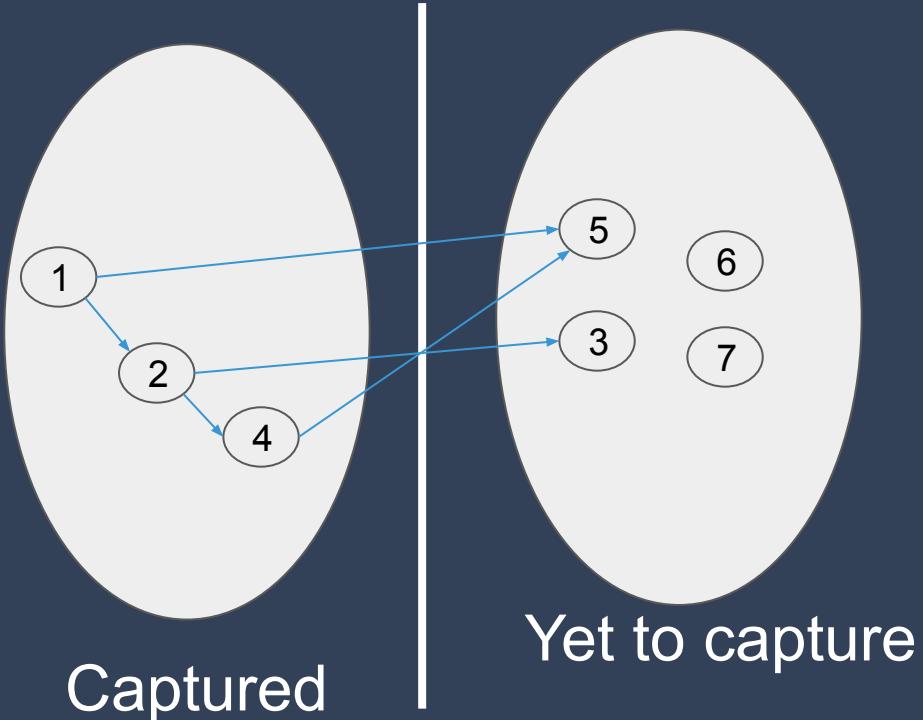
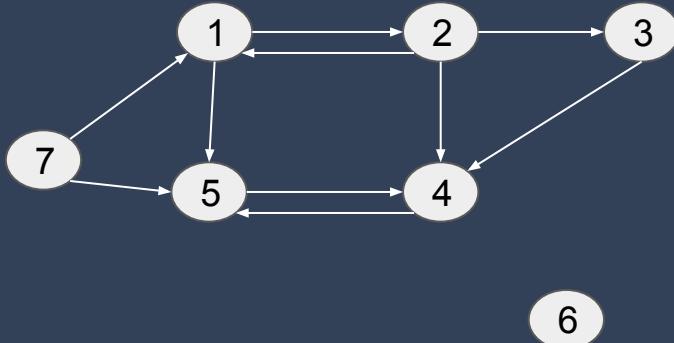
# Skeleton search



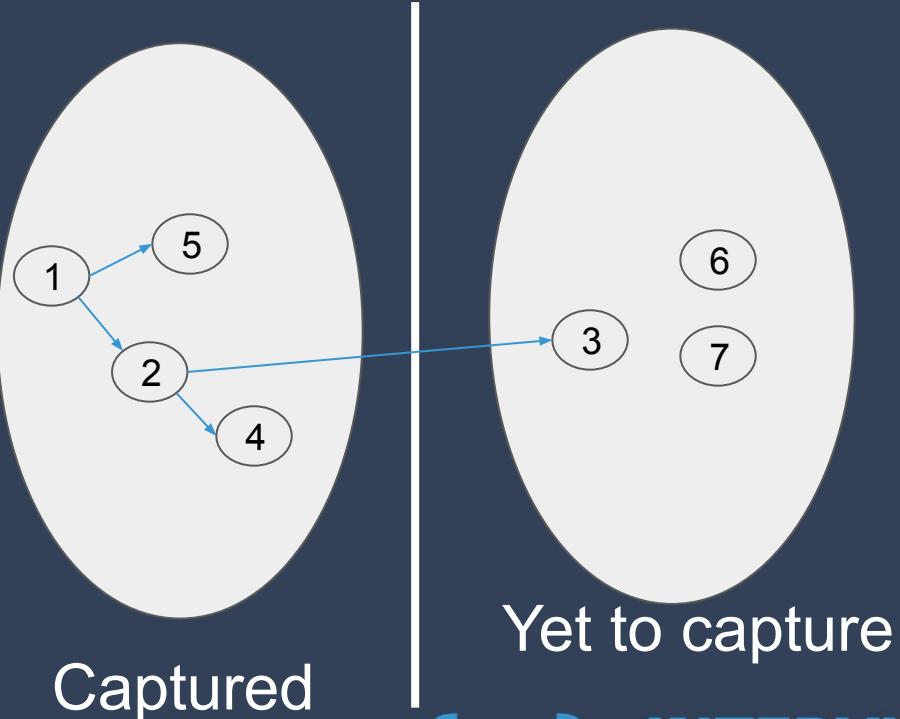
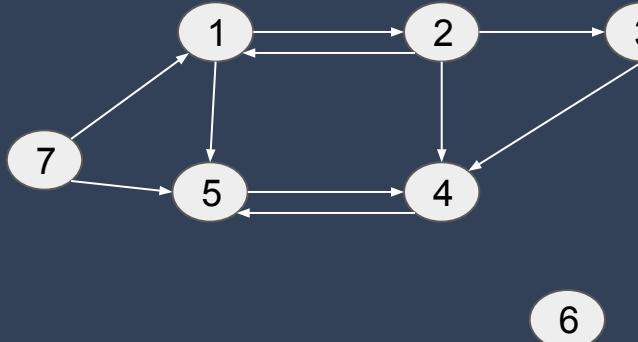
# Skeleton search



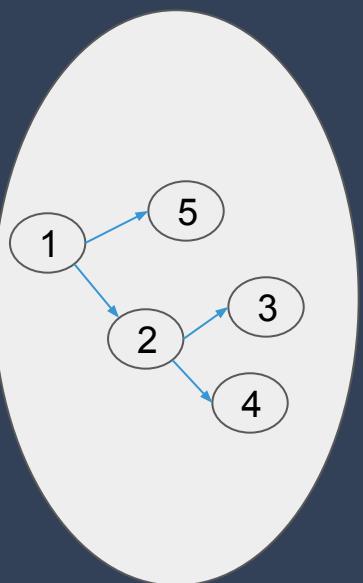
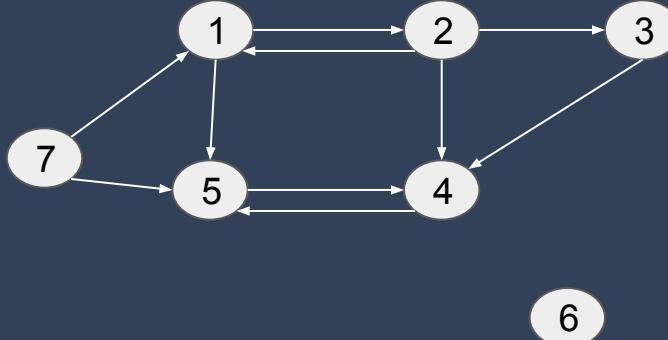
# Skeleton search



# Skeleton search

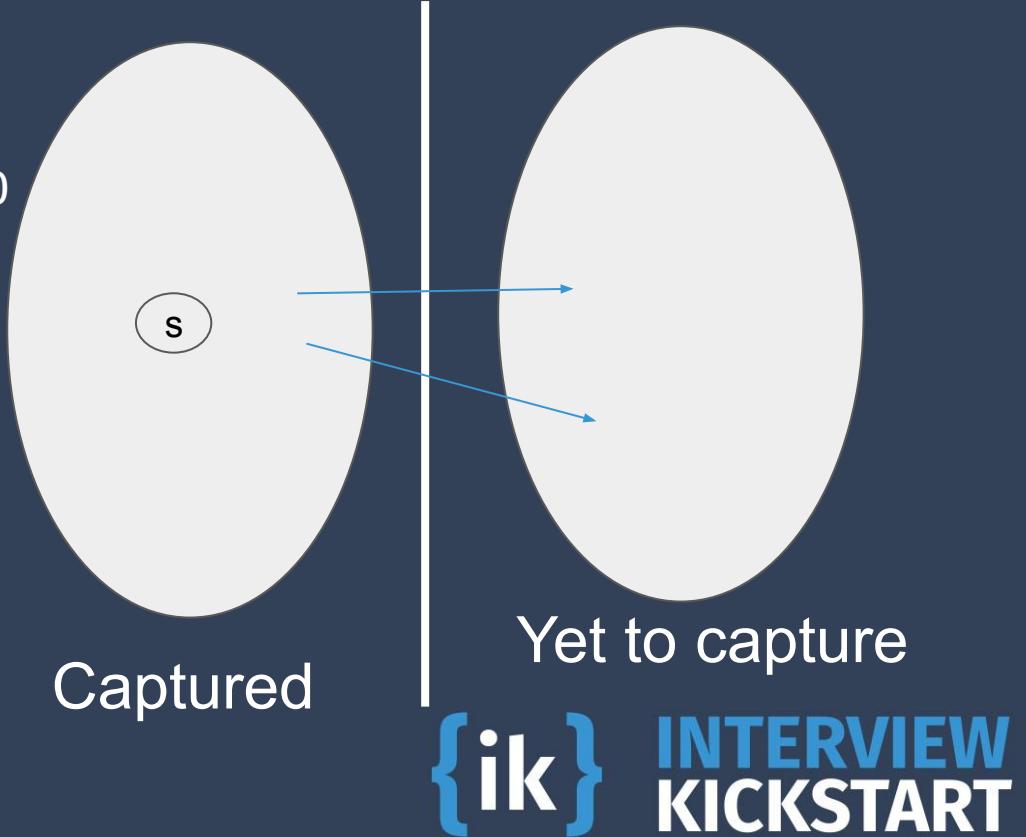


# Skeleton search



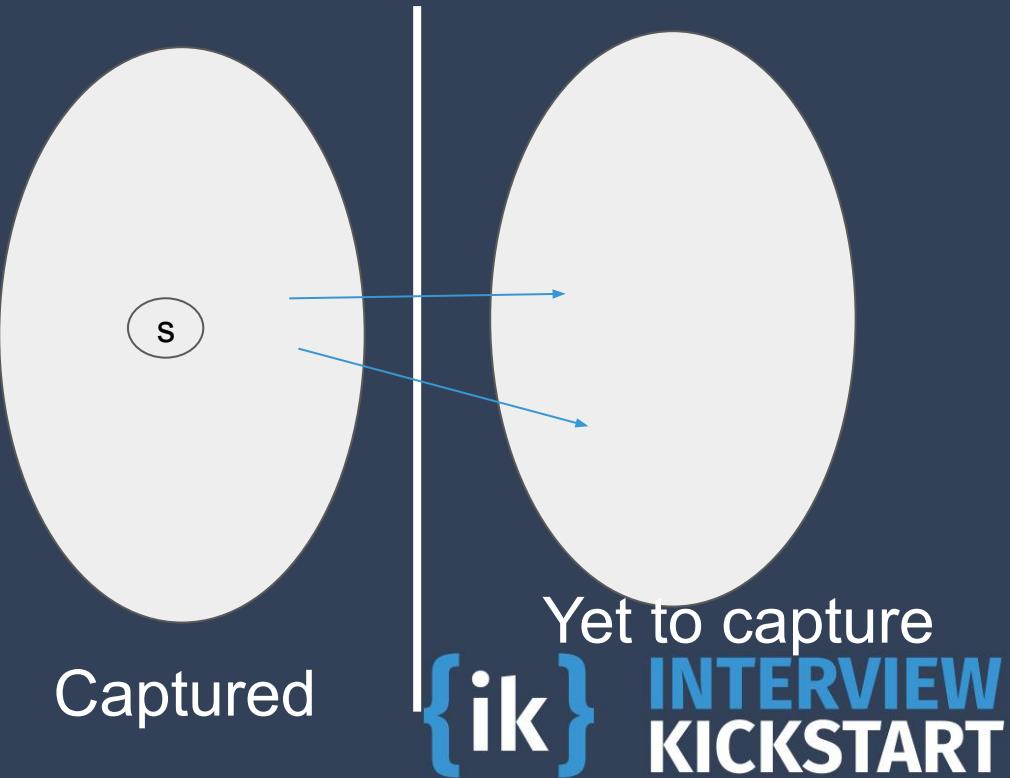
# Skeleton search

```
class Graph {  
  
    void search(int s) {  
        //captured is an array initialized to 0  
        //for all vertices  
        captured[s] = 1  
  
    }  
    ....  
}
```



# Skeleton search

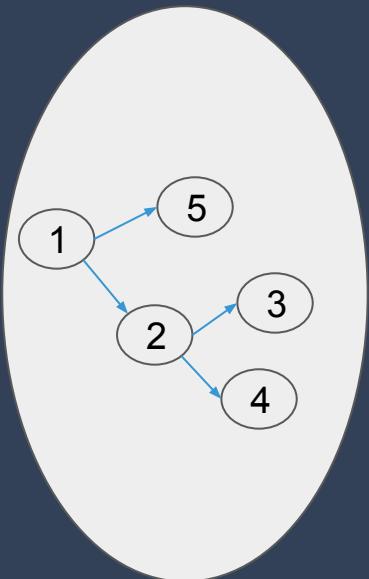
```
class Graph {  
    void search(int s) {  
        captured[s] = 1  
  
        while there exists a fringe edge:  
            pick one of them => (u,v)  
            captured[v] = 1  
    }  
}
```



# Result

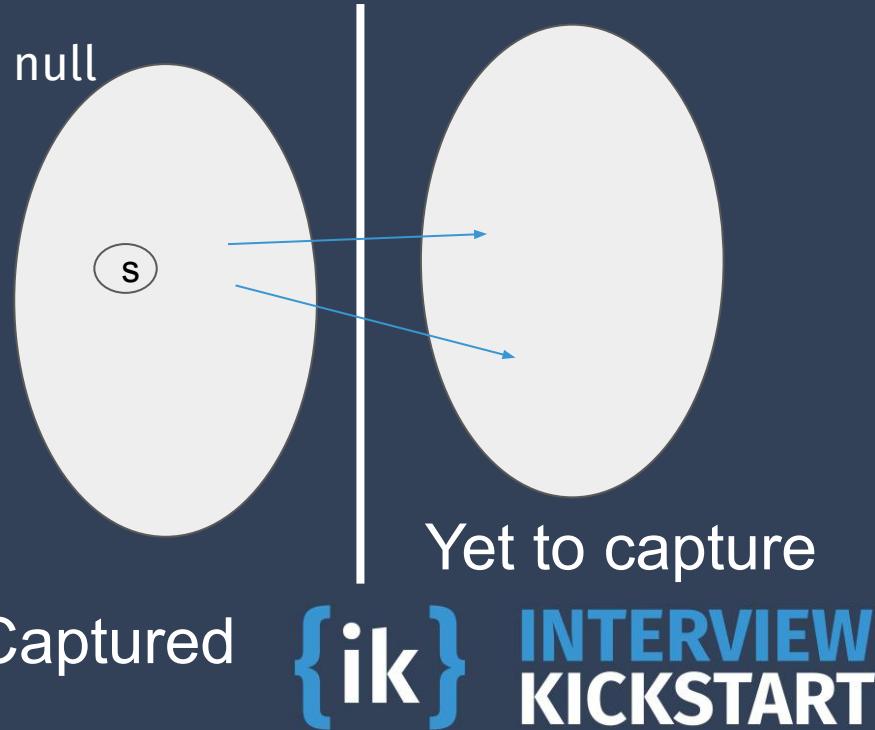
1. We get a search tree with root = source vertex s
2. The set of captured vertices = the set of reachable nodes from s

Captured



# Keeping track of parents in search tree

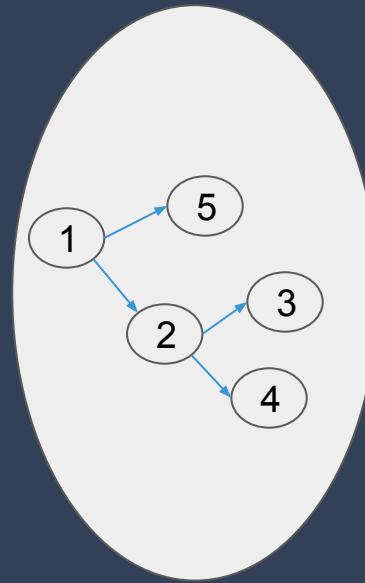
```
class Graph {  
  
void search(int s) {  
    //captured and parent initialized to 0 and null  
  
    captured[s] = 1  
    while there exists a fringe edge:  
        pick one of them => (u,v)  
        captured[v] = 1  
        parent[v] = u  
  
    }  
}
```



# captured

1	1	1	1	1	0	0
1	2	3	4	5	6	7
null	1	2	2	1	null	null

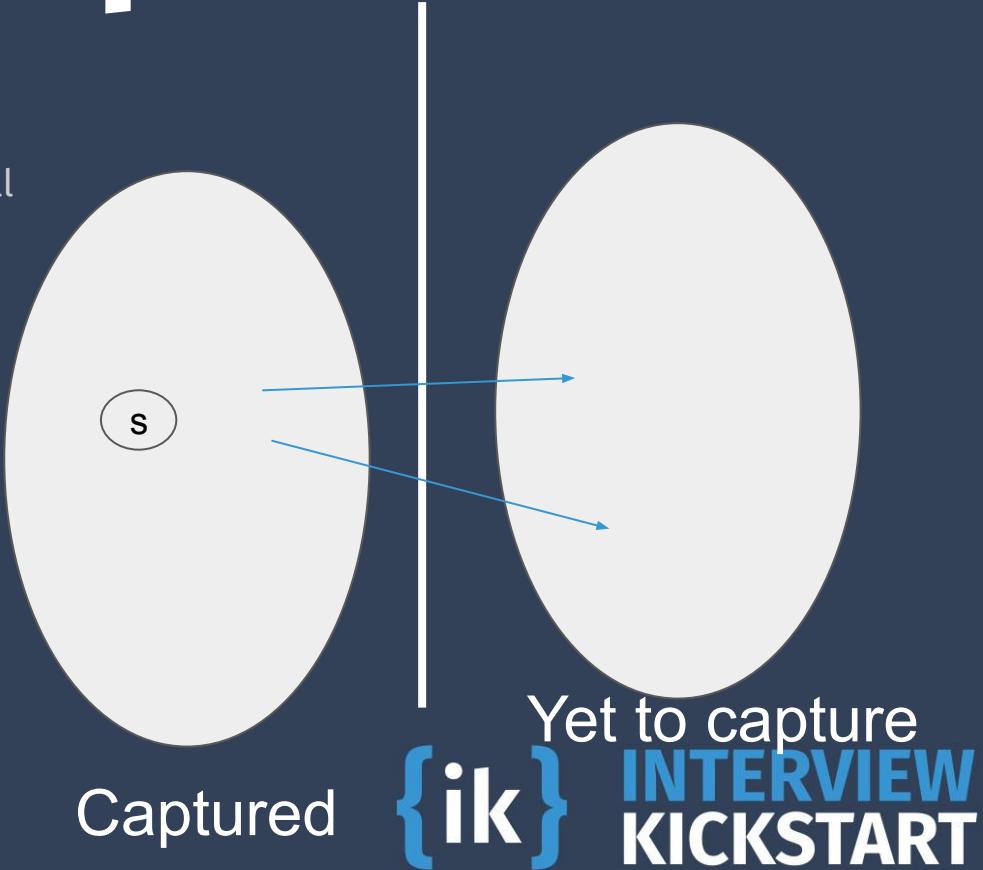
# parent



Captured

# Different policies

```
class Graph {  
  
    void search(int s) {  
        //captured and parent initialized to 0 and null  
  
        captured[s] = 1  
        while there exists a fringe edge:  
            pick one of them => (u,v)  
            captured[v] = 1  
            parent[v] = u  
  
    }  
}
```

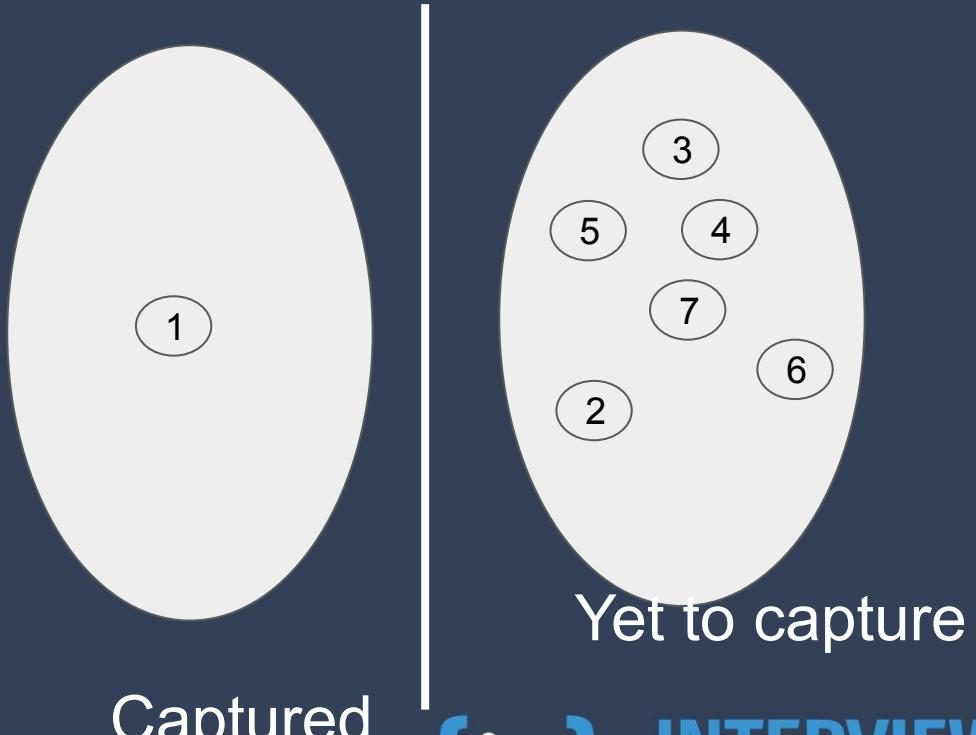
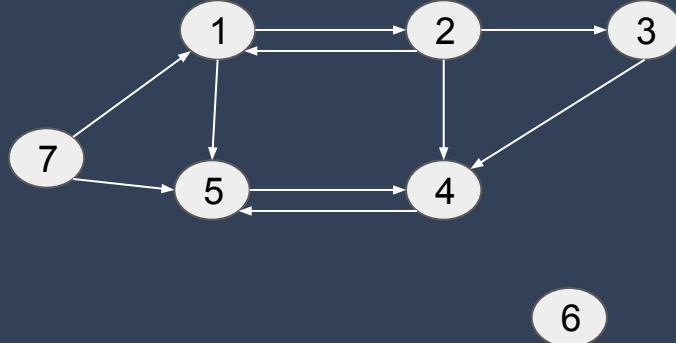


## BFS & DFS

BFS policy: Among all fringe edges  $(u,v)$ , pull the one which was examined **first**.

DFS policy: Among all fringe edges  $(u,v)$ , pull the one which was examined **last (i.e, most recently)**.

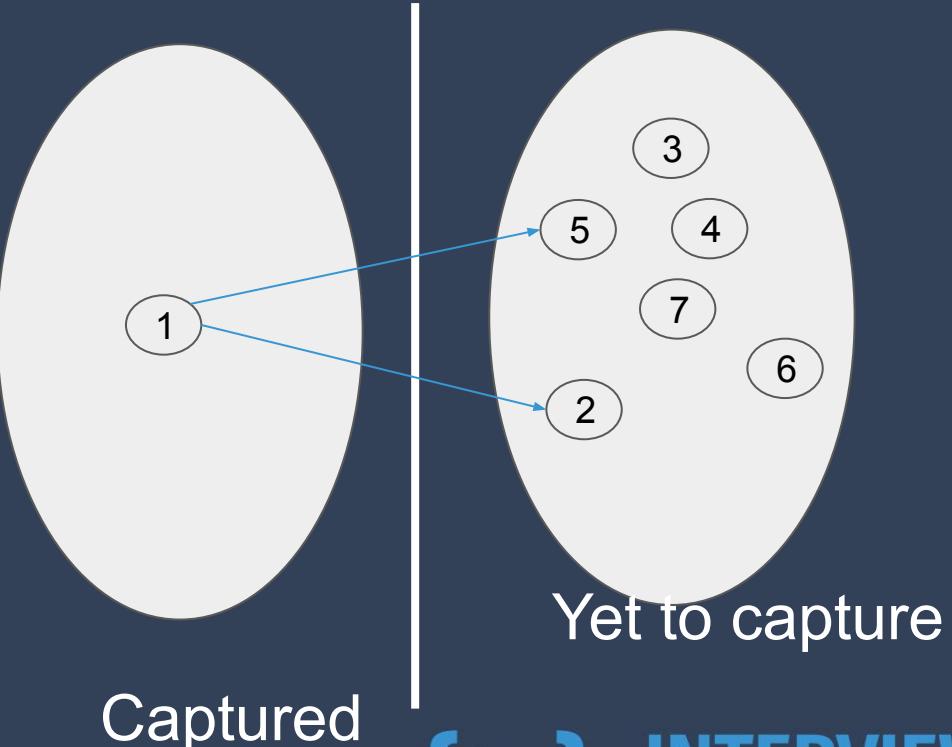
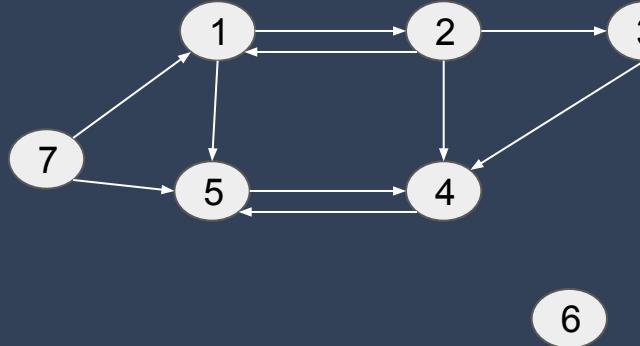
# BFS



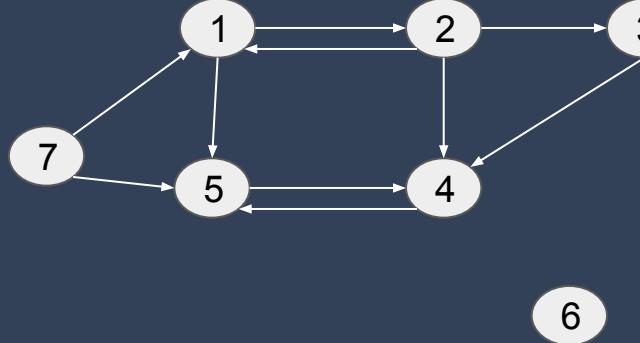
Captured

Yet to capture

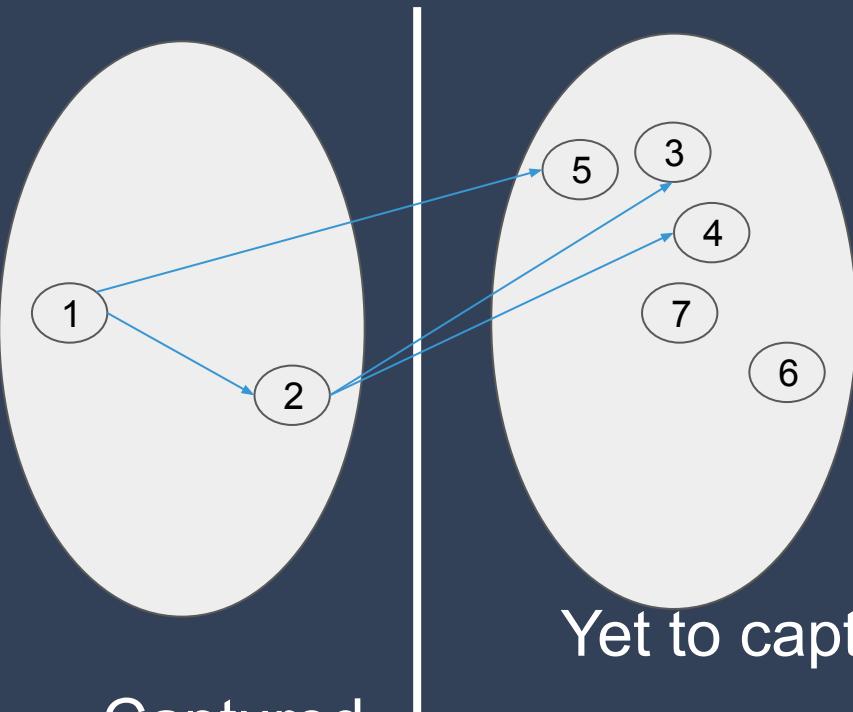
# BFS



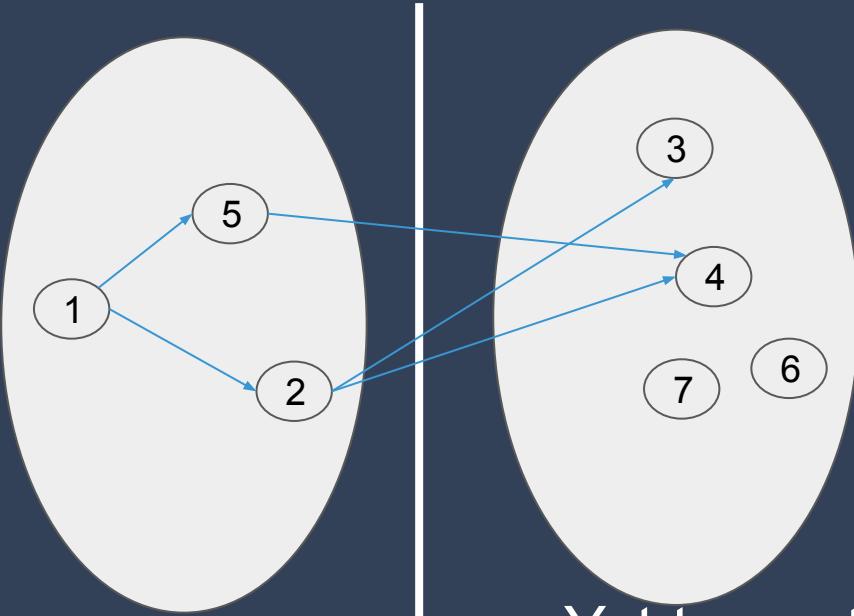
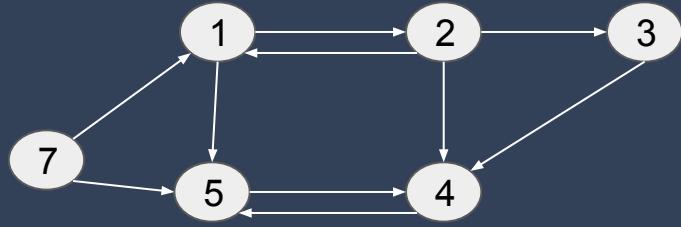
# BFS



6



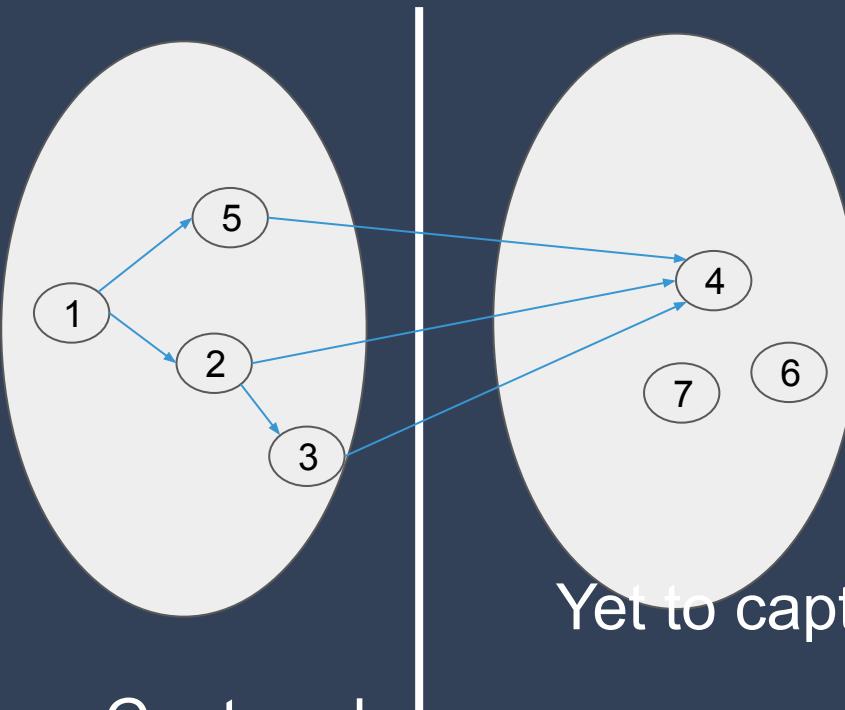
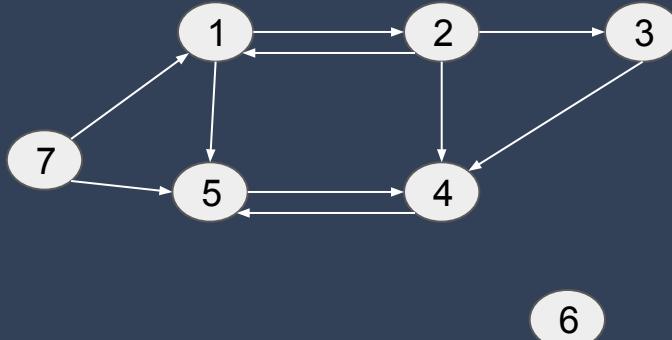
# BFS



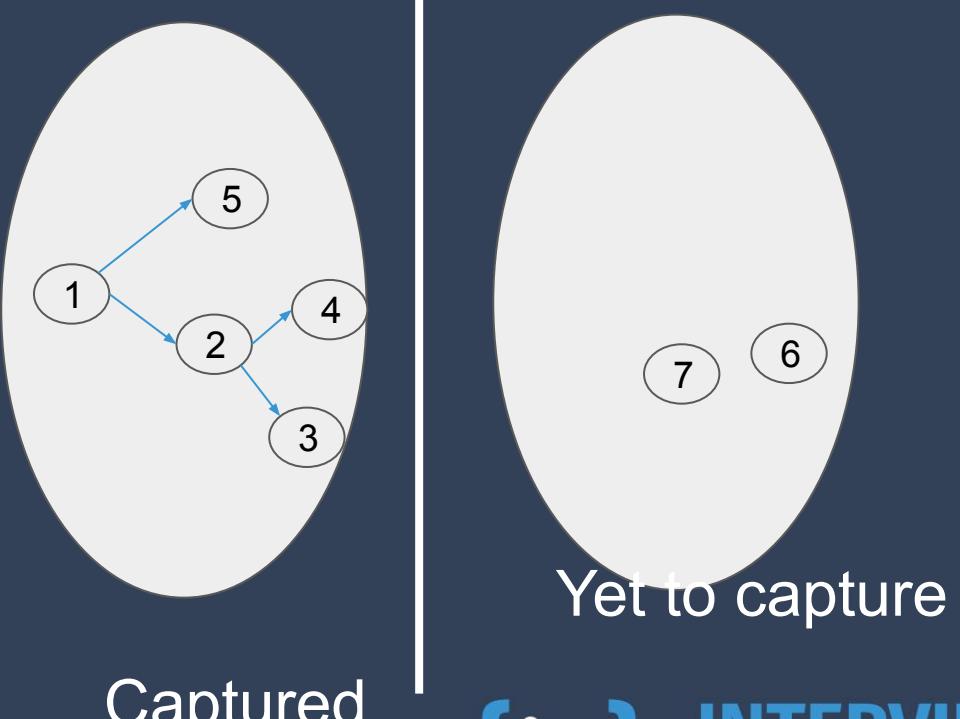
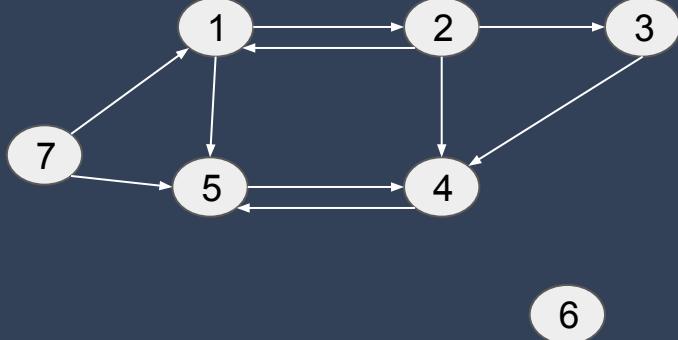
Captured

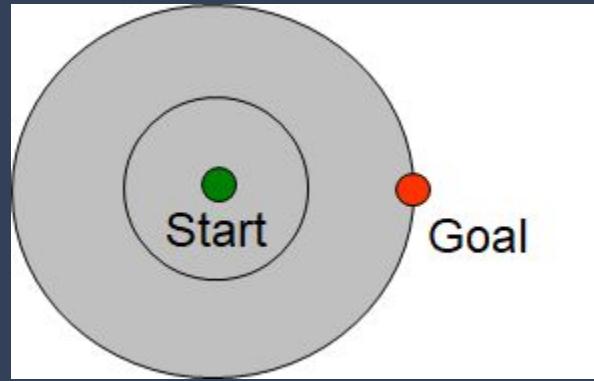
Yet to capture

# BFS



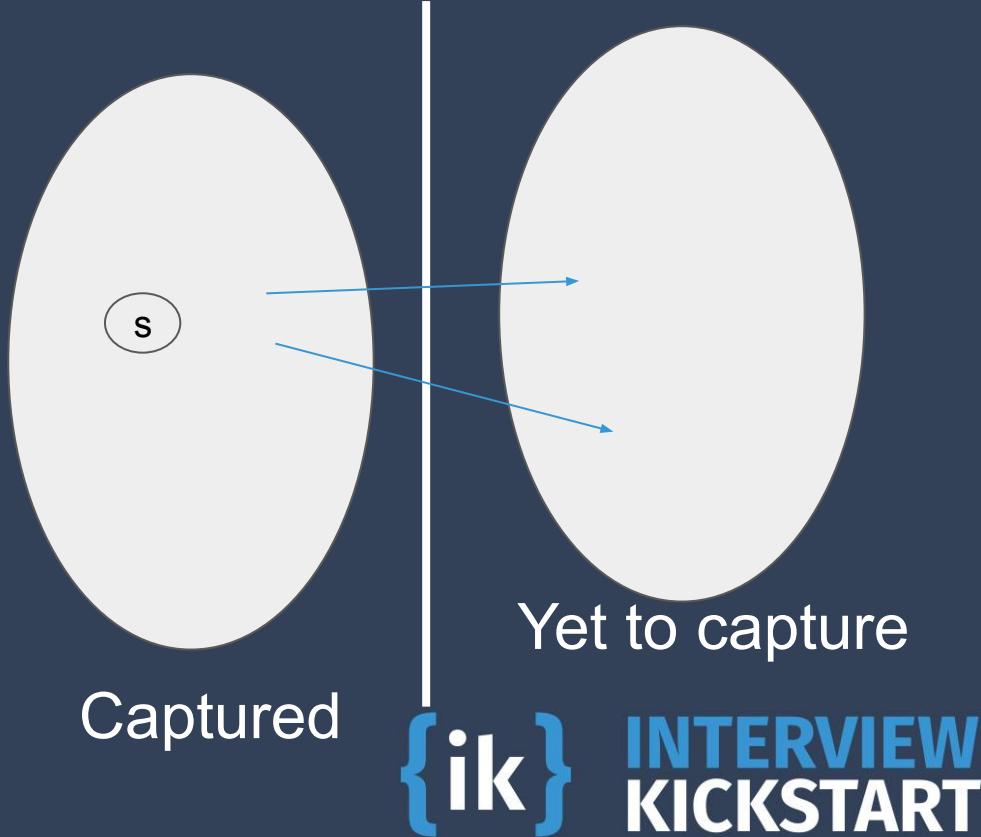
# BFS





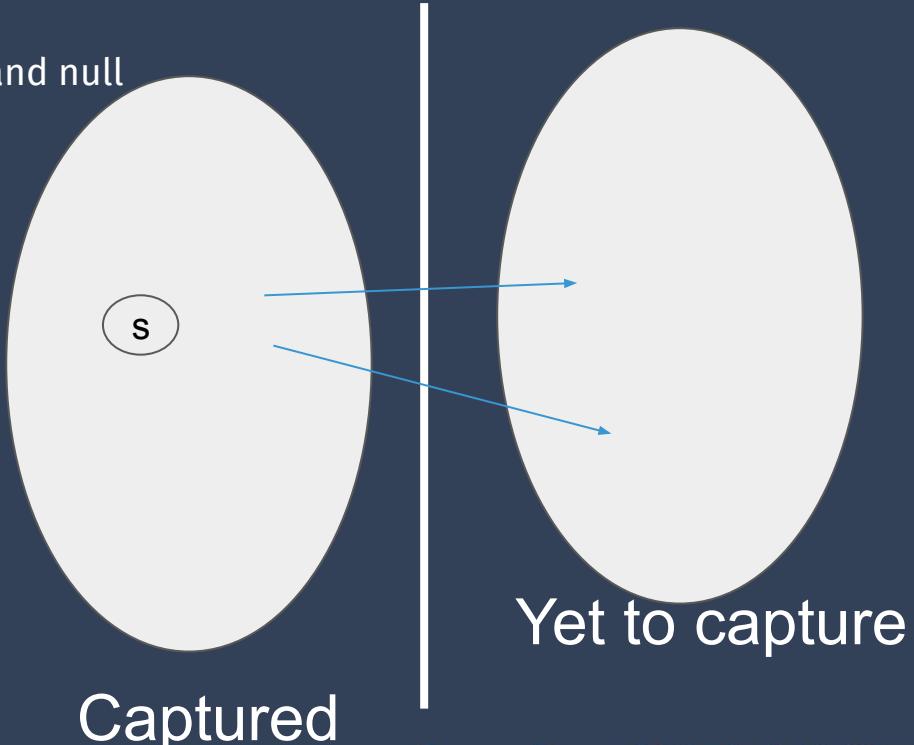
# Modify the code for BFS

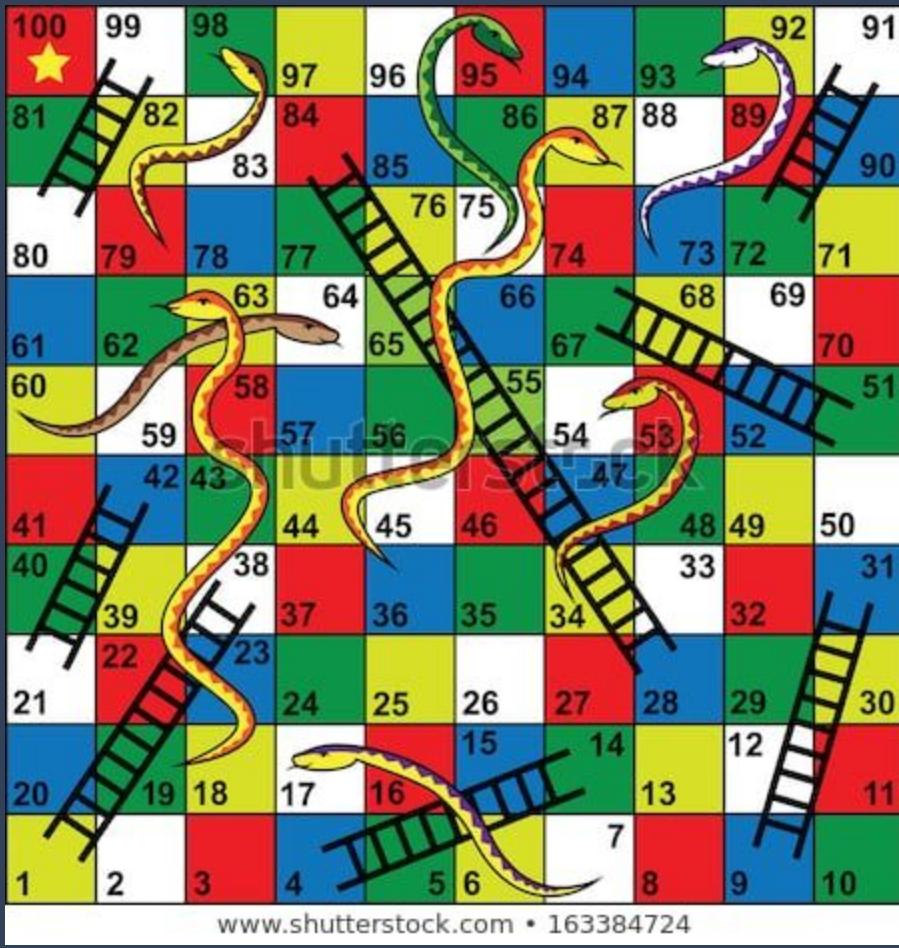
```
class Graph {  
  
void search(int s) {  
    //captured and parent initialized to 0 and null  
  
    captured[s] = 1  
while there exists a fringe edge:  
    pick one of them => (u,v)  
    captured[v] = 1  
    parent[v] = u  
  
}  
}  
}
```



# BFS

```
class Graph {  
    void BFS(int s) {  
        //visited, captured and parent initialized to 0, 0 and null  
        captured[s] = 1; visited[s] = 1  
  
        q = new Queue(); q.push(s);  
        while not isEmpty(q): //capture the next vertex  
            v = q.pop()  
            captured[v] = 1  
            for w in Adjlist[v]:  
                if visited[w] == 0 then  
                    visited[w] = 1; parent[w] = v;  
                    q.push(w);  
    }  
}
```





Given a snake and ladder game, find the minimum number of throws required to win the game.

**Given n nodes labeled from 0 to n - 1 and a list of undirected edges (each edge is a pair of nodes), write a function to find the number of connected components in an undirected graph.**

# BFS

```
class Graph {  
  
    void BFS(int s, int c) {  
  
        captured[s] = 1; visited[s] = c  
  
        q = new Queue(); q.push(s);  
        while not isEmpty(q): //capture the next  
        vertex  
            v = q.pop()  
            captured[v] = 1  
            for w in Adjlist[v]:  
                if visited[w] == 0 then  
                    visited[w] = c; parent[w] = v;  
                    q.push(w);  
    }  
}
```

```
void findComponents() {  
    //visited, captured and parent are initialized  
    to 0, 0 and null  
    component = 0  
    for i in 1 to V:  
        if visited[i] = 0:  
            component++ //start new component  
            BFS(i, component)  
  
    return component  
}
```

Eulerian cycle or path problem

Graph representations

General graph traversal

BFS

Dijkstra

Prim

DFS

Best-first

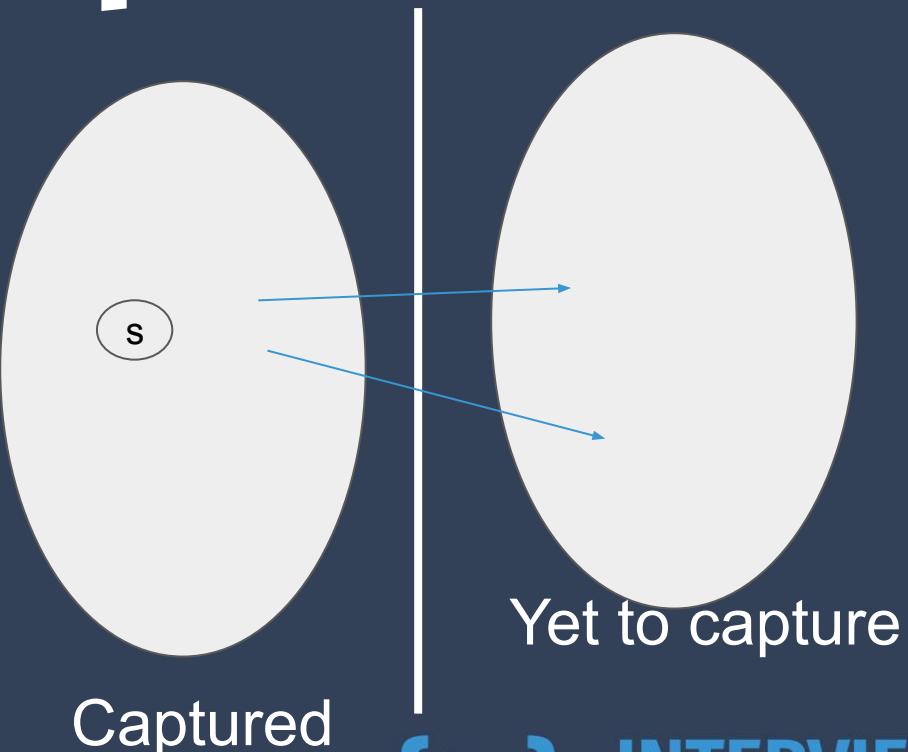
A\*

bipartite

connected  
components

# Different policies

```
class Graph {  
  
    void search(int s) {  
        //captured and parent initialized to 0 and null  
  
        captured[s] = 1  
        while there exists a fringe edge:  
            pick one of them => (u,v)  
            captured[v] = 1  
            parent[v] = u  
  
    }  
}
```



# Graphs

{ik} INTERVIEW  
KICKSTART

Live Class



- **Graphs:** Consider if a problem can be applied with graph algorithms like distance, search, connectivity, cycle-detection, etc. There are three basic ways to represent a graph in memory (objects and pointers, matrix, and adjacency list) – familiarize yourself with each representation and its pros and cons. You should know the basic graph traversal algorithms, breadth-first search and depth-first search. Know their computational complexity, their tradeoffs and how to implement them in real code.

<https://careers.google.com/how-we-hire/interview/#interviews-for-software-engineering-and-technical-roles>

# facebook

- ✓ Go over data structures, algorithms and complexity: Be able to discuss the big-O complexity of your approaches. Don't forget to brush up on your data structures like lists, arrays, hash tables, hash maps, stacks, queues, graphs, trees, heaps. Also sorts, searches, and traversals (BFS, DFS). Also review recursion and iterative approaches.

<https://www.facebook.com/careers/life/preparing-for-your-software-engineering-interview-at-facebook/>

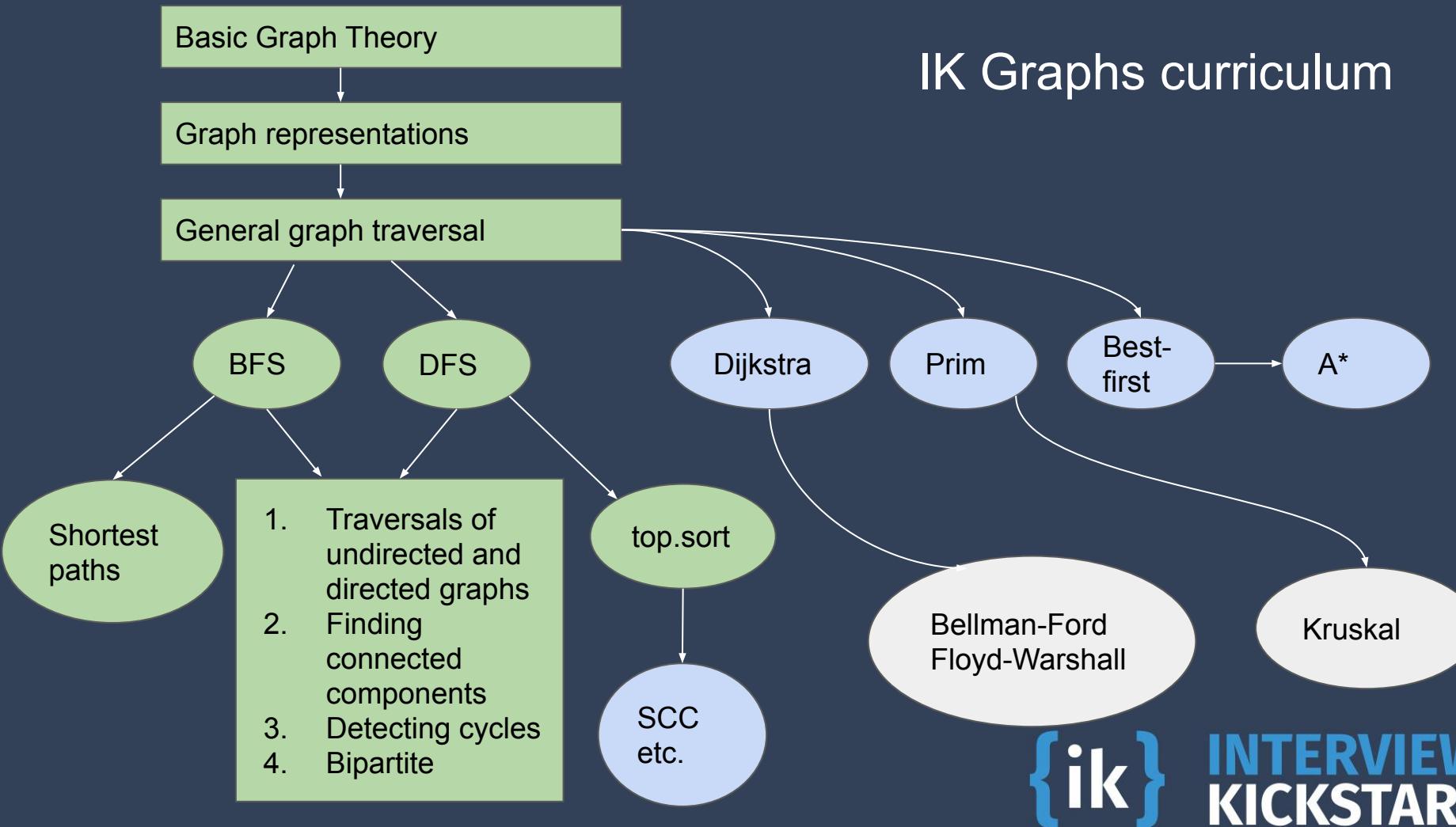


## Algorithms

Your interview will not be focused on rote memorization of algorithms. However, having a good understanding of the most common algorithms will likely make solving some of the questions a lot easier. Consider reviewing common algorithms such as traversals, divide and conquer, breadth-first search vs. depth-first search and understand the tradeoffs for each. Knowing the runtimes, theoretical limitations, and basic implementation strategies of different classes of algorithms is more important than memorizing the specific details of any given algorithm.

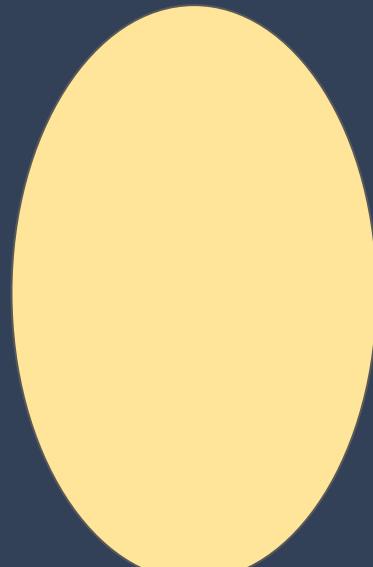
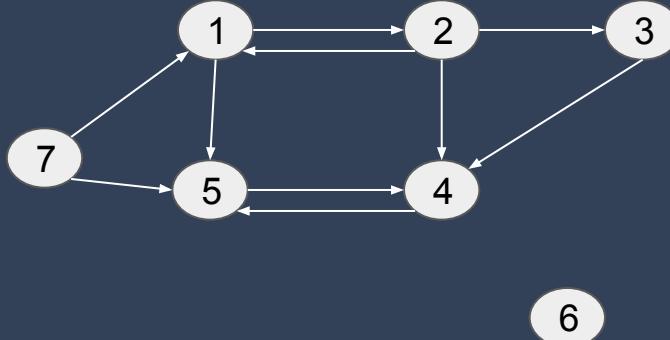
[https://www.amazon.jobs/en/landing\\_pages/in-software-development-topics](https://www.amazon.jobs/en/landing_pages/in-software-development-topics)

# IK Graphs curriculum

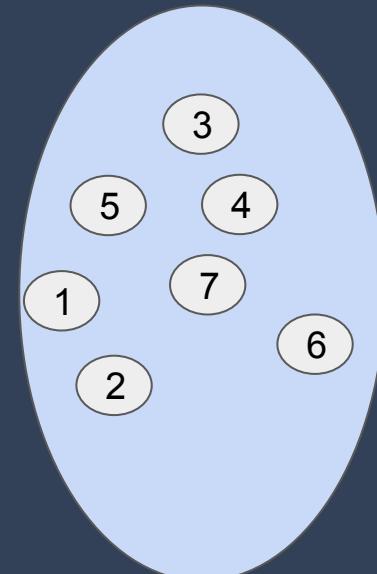


# General Graph Traversal

Given this directed graph, which vertices are reachable from vertex 1?



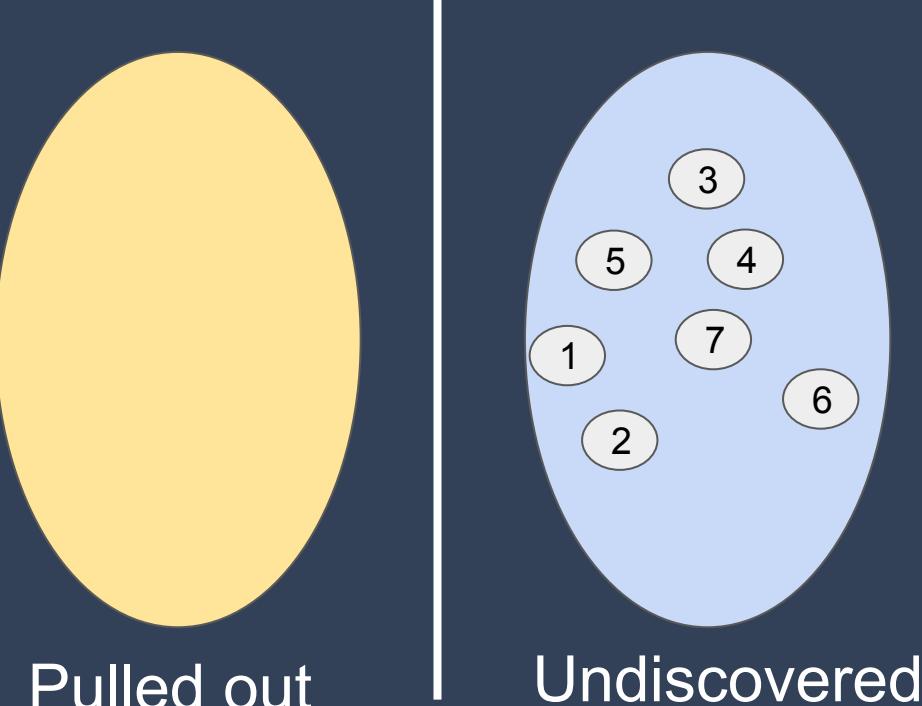
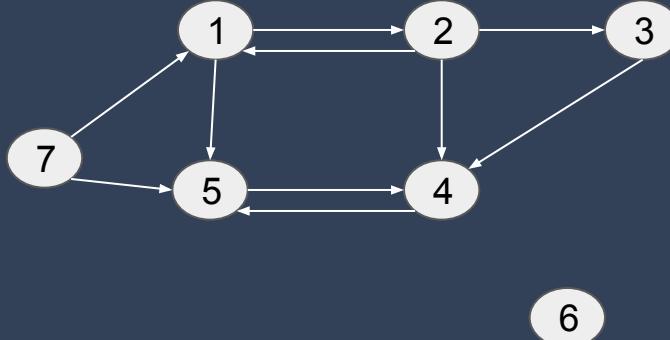
Pulled out



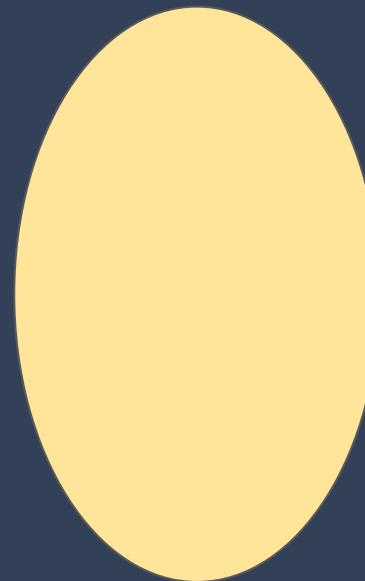
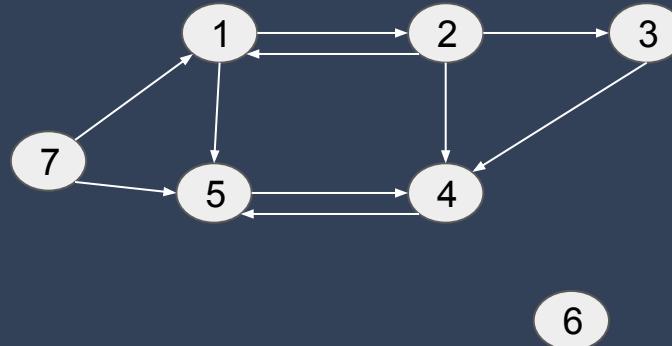
Undiscovered

# General Graph Traversal

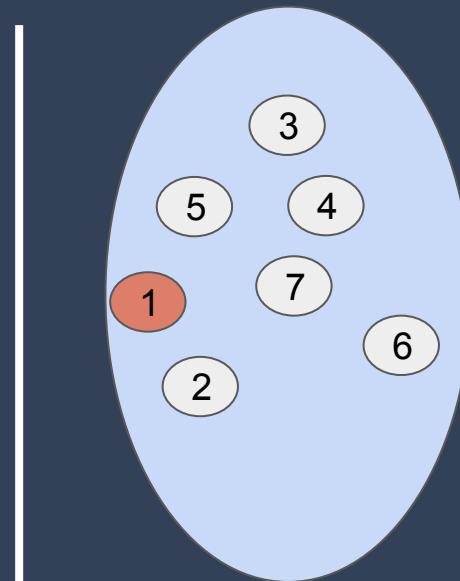
Initially, all the vertices are unreachable. They lie undiscovered, like fish swimming deep inside an ocean.



We do have one vertex that is reachable though, and that is the source vertex.

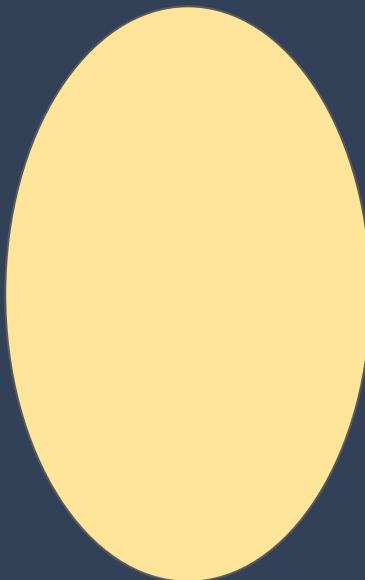
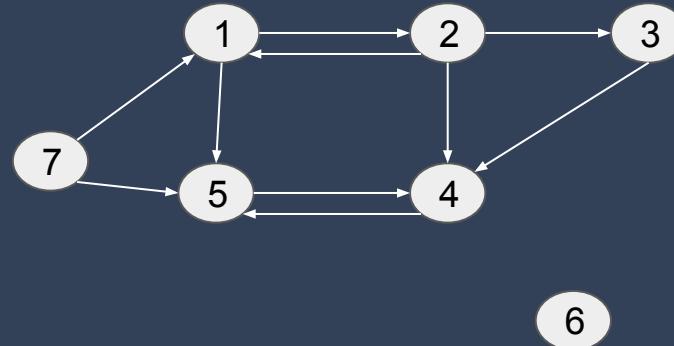


Pulled out

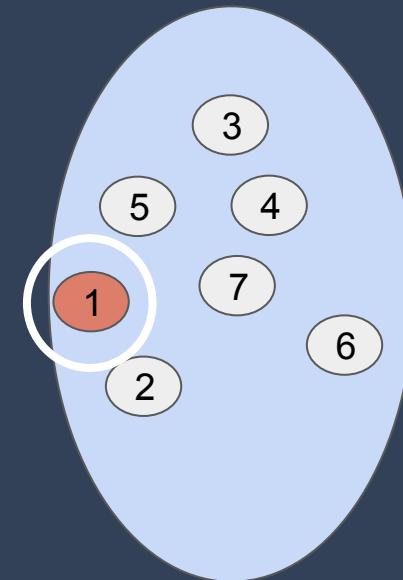


Undiscovered

Like a fish caught inside a fishing net/bag.

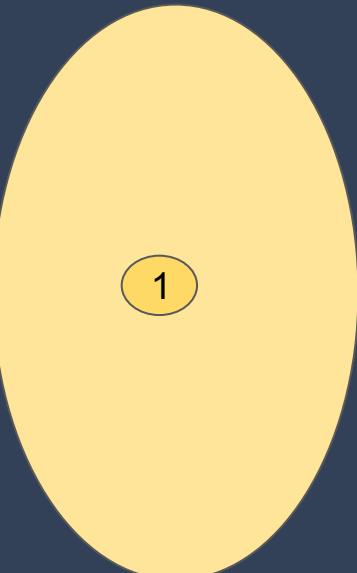
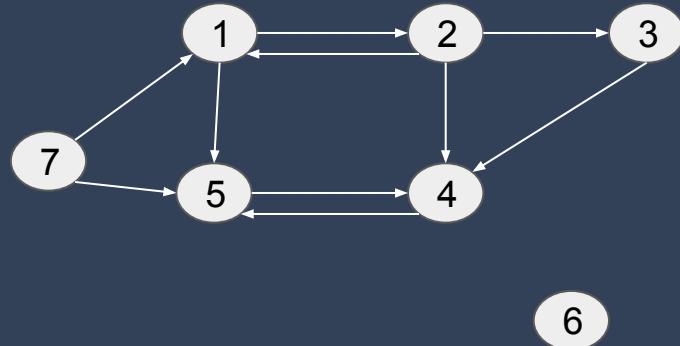


Pulled out

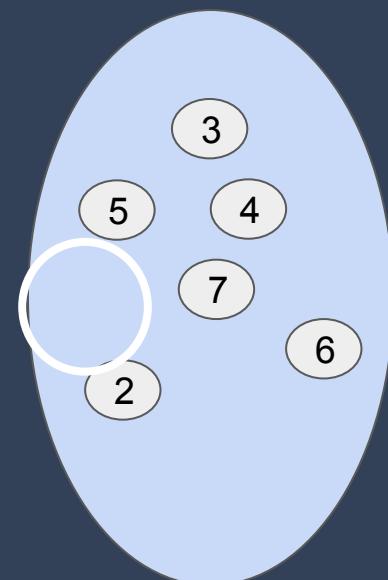


Undiscovered

Now it has been pulled out of the bag, and brought ashore.

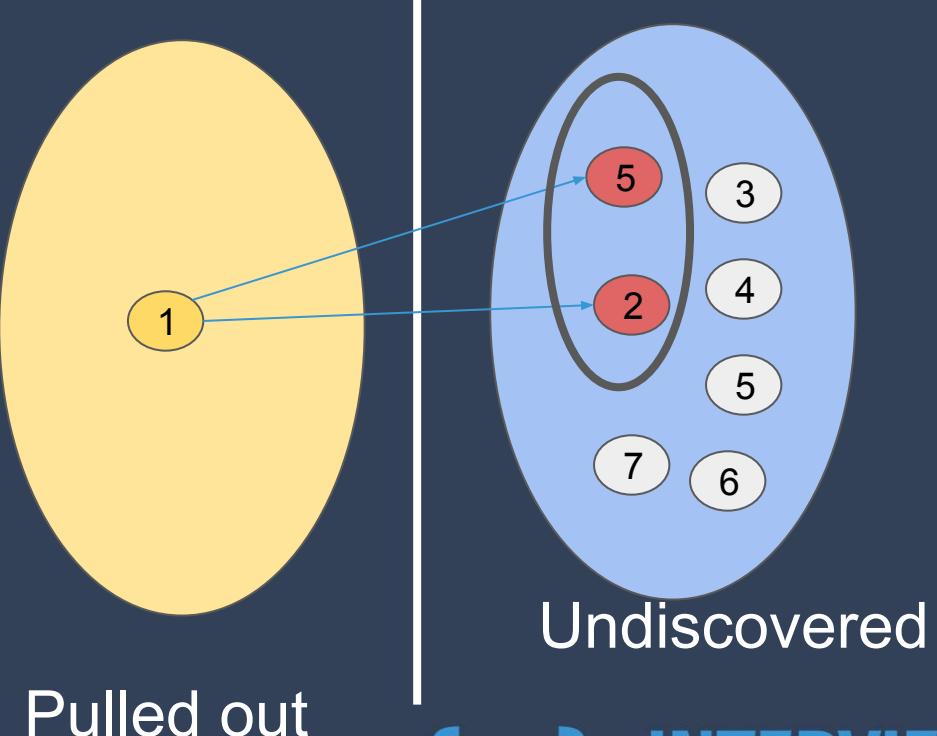
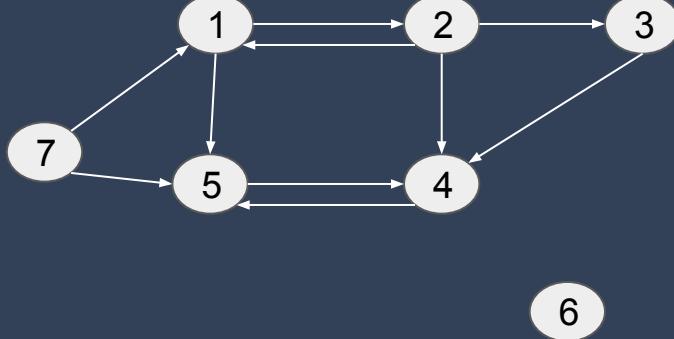


Pulled out

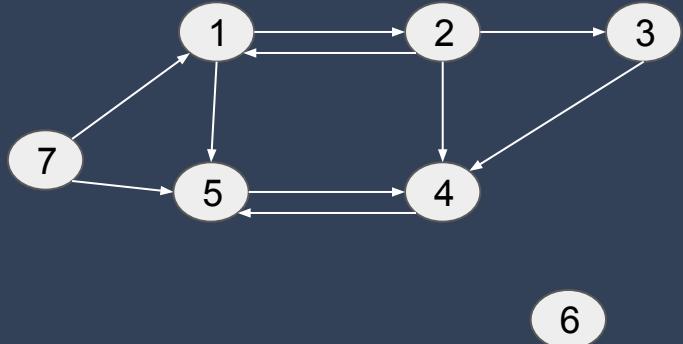


Undiscovered

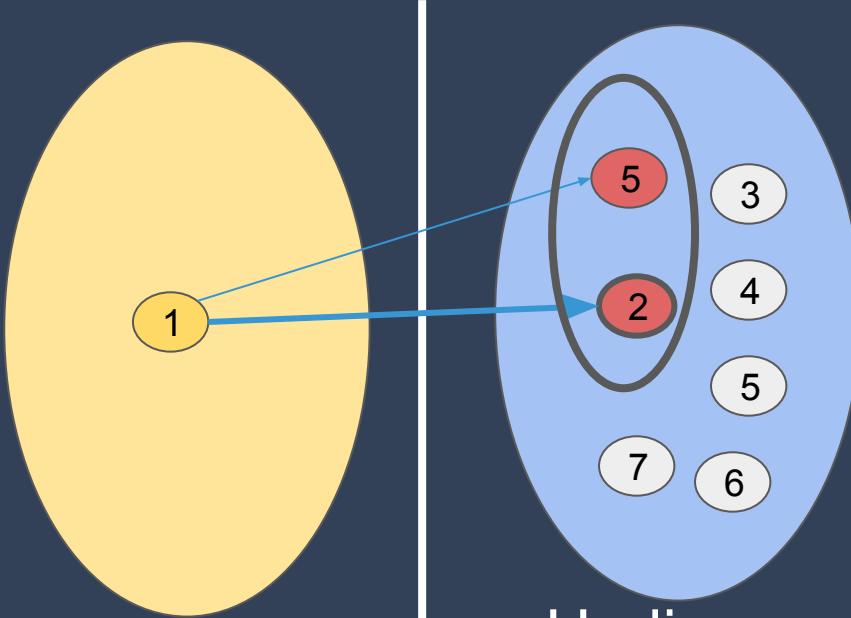
But it has these “hooks” (relationships) connecting it to neighbors that were so far undiscovered, but which now become reachable. We put them into the bag.

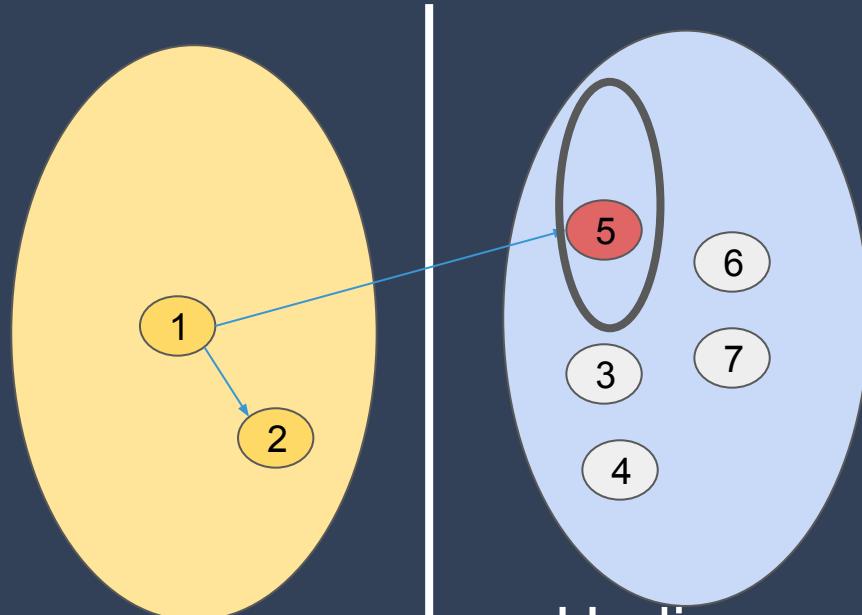
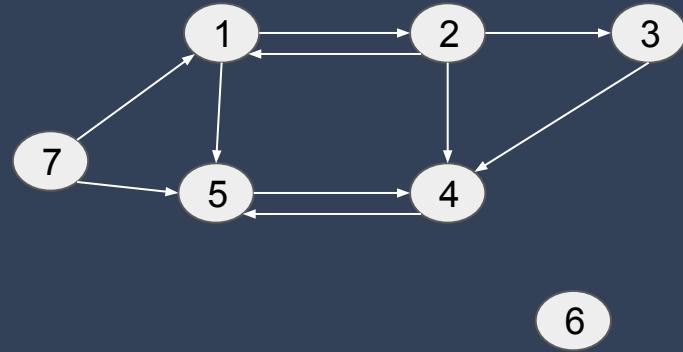


We now pull another vertex out of the bag.

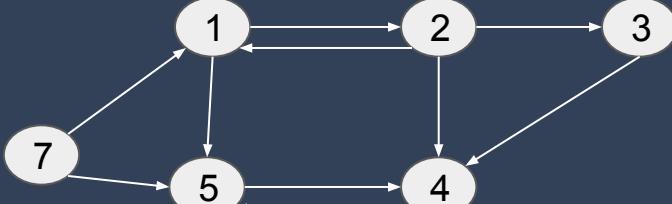


Pulled out



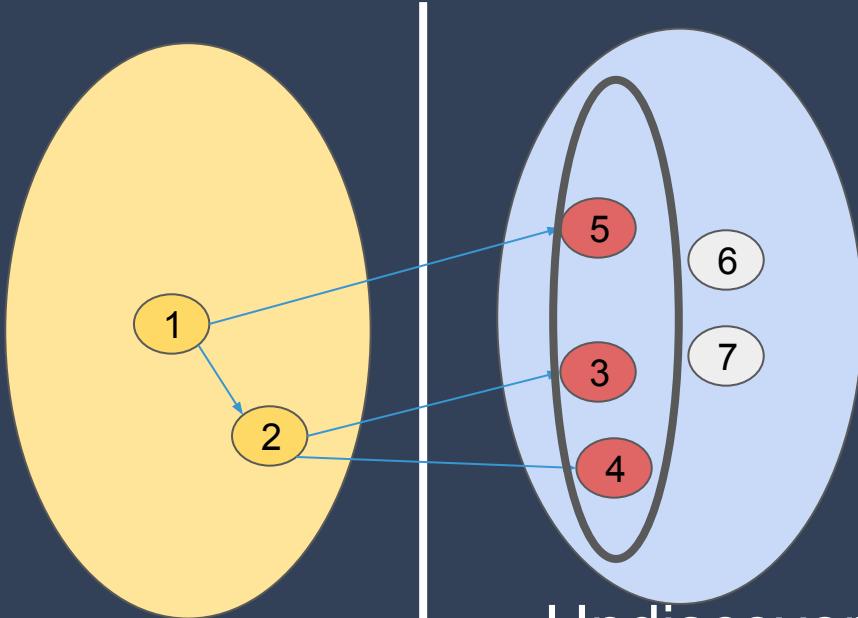


Look at the neighbors of 2. If any of them was undiscovered, they are now reachable, so put them into the bag.



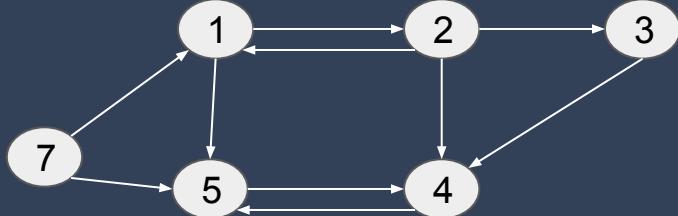
6

Pulled out



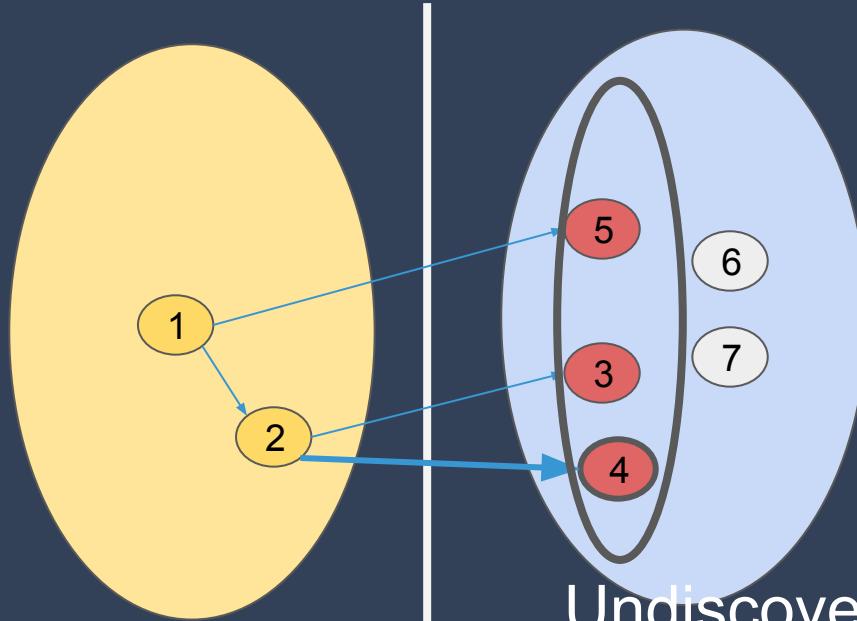
Undiscovered

Choose the next vertex that will  
be pulled out of the bag.

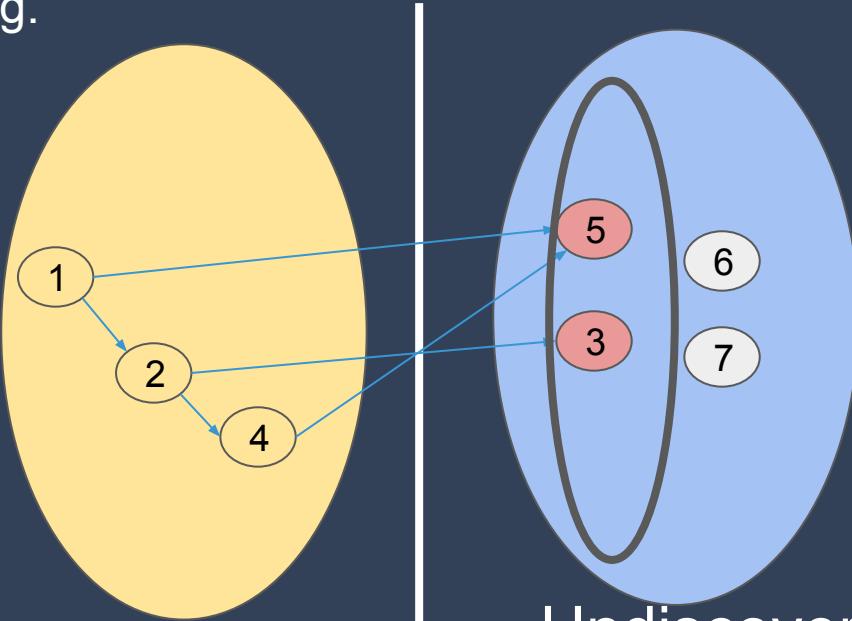
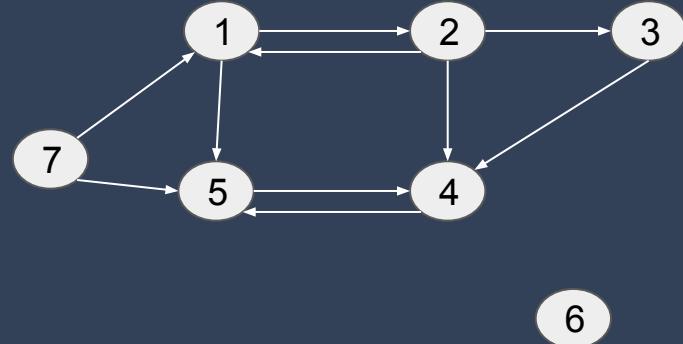


6

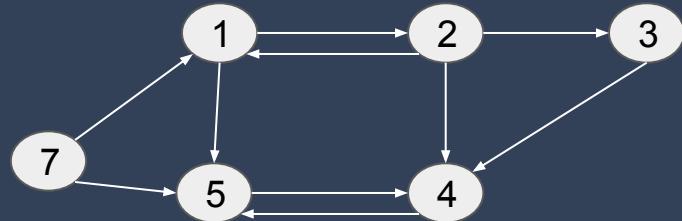
Pulled out



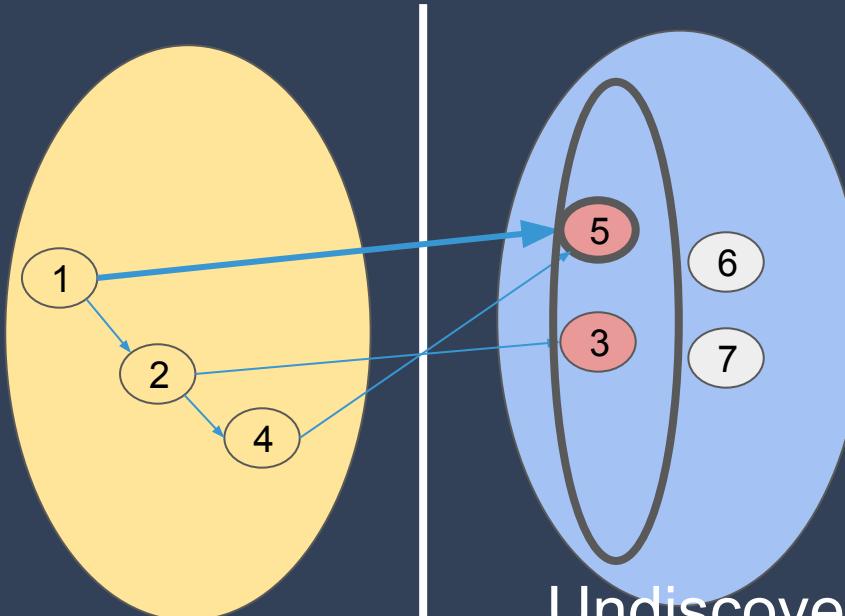
Vertex 4 has no undiscovered neighbors, so no new additions to the bag.



Choose the next vertex that will  
be pulled out of the bag.

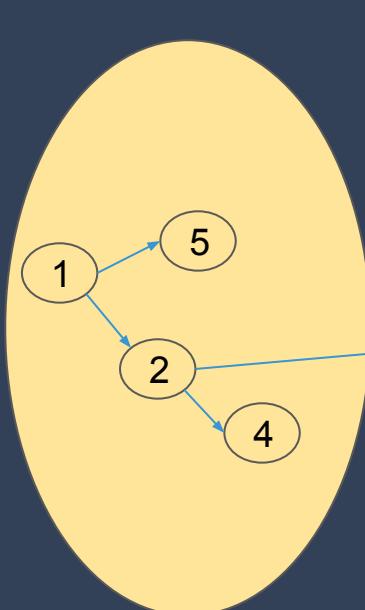
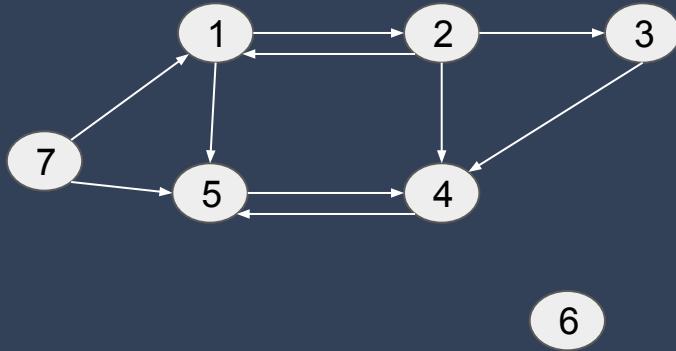


6

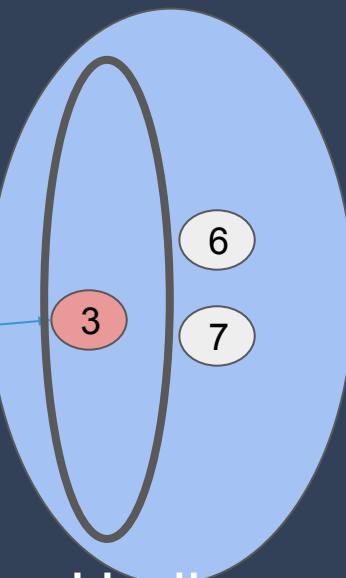


Pulled out

Undiscovered

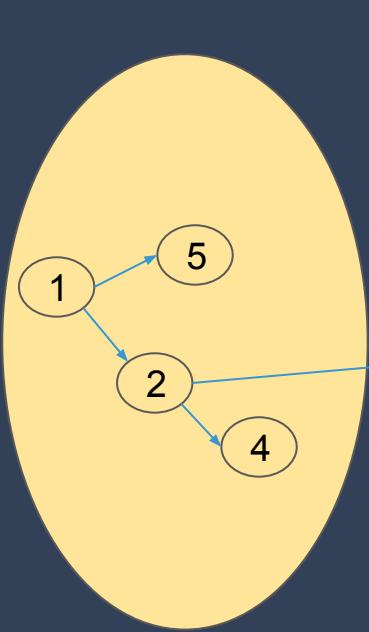
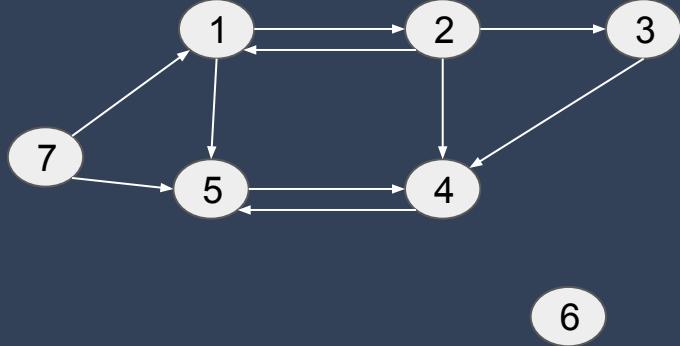


Pulled out

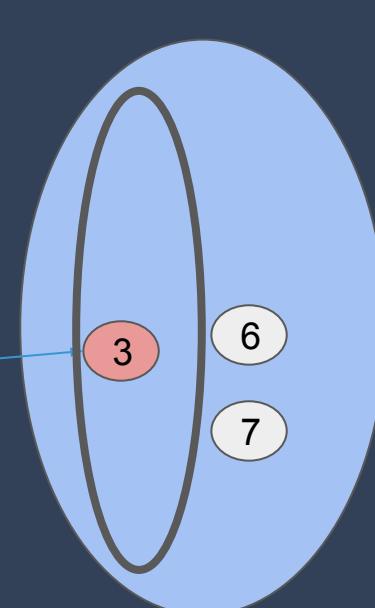


Undiscovered

No additions to the bag

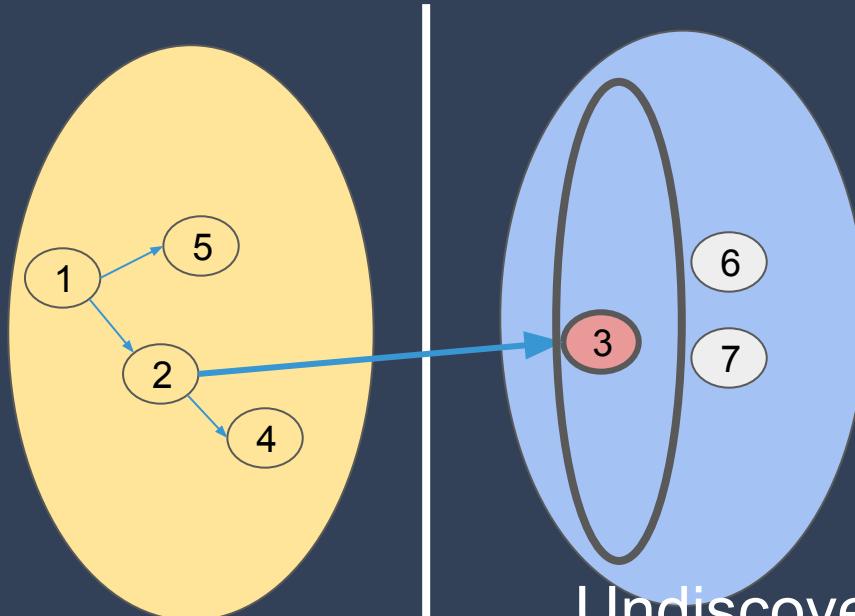
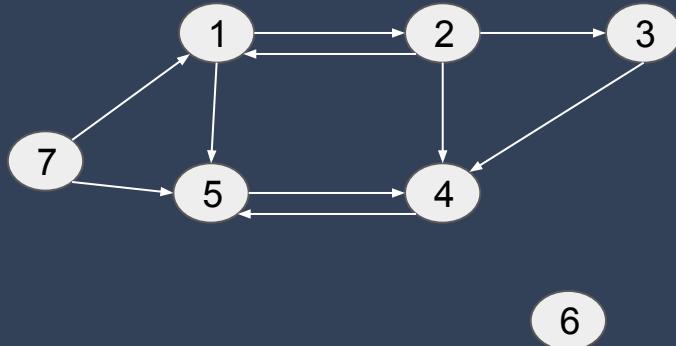


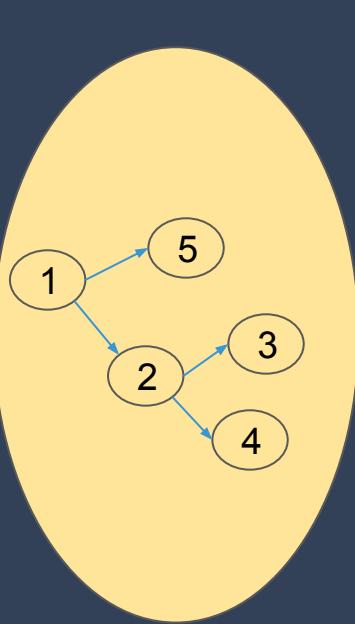
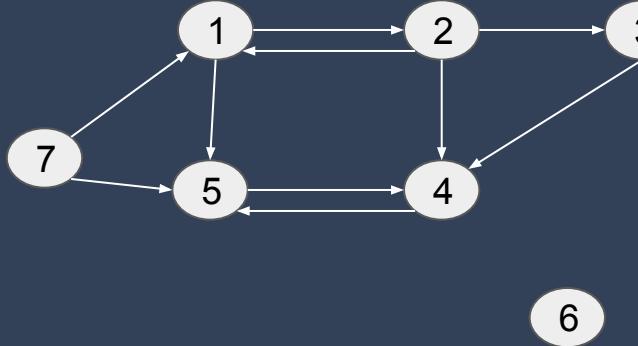
Pulled out



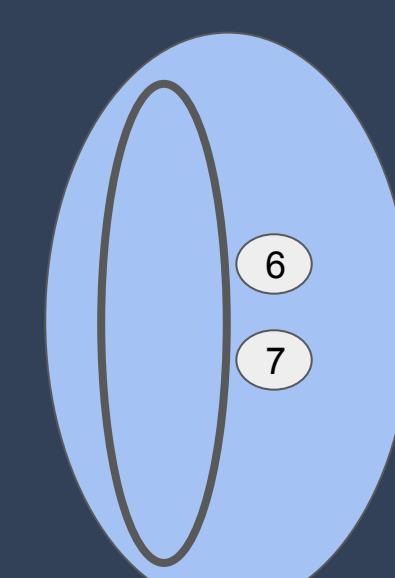
Undiscovered

Choose the next vertex that will  
be pulled out of the bag.



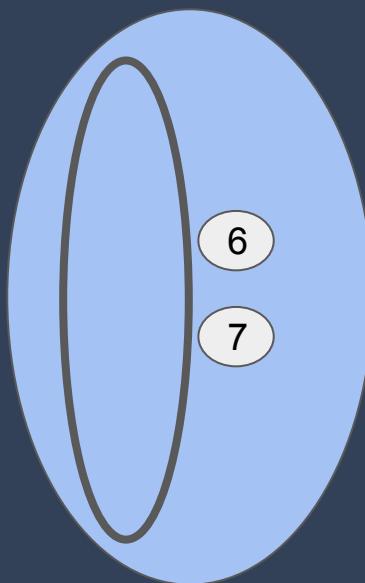
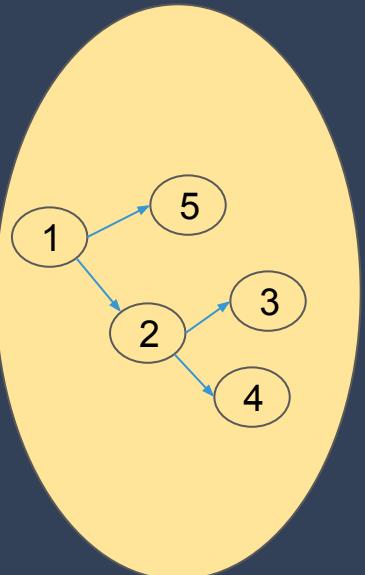
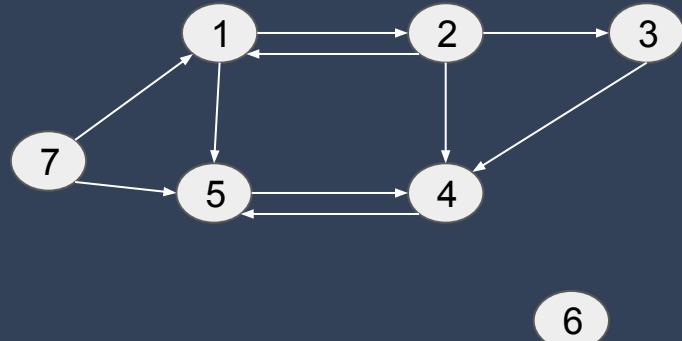


Captured



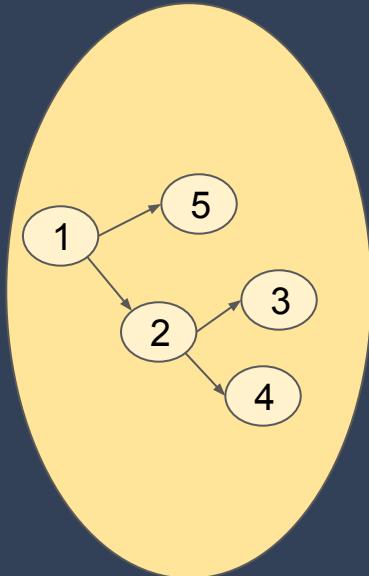
Undiscovered

Now the bag is empty!



# Result

1. We get a search tree with root = source vertex s
2. The vertices in it are the set of reachable nodes from s
3. If a vertex c was pulled out of the bag by vertex p, then p is the parent of c in the search tree

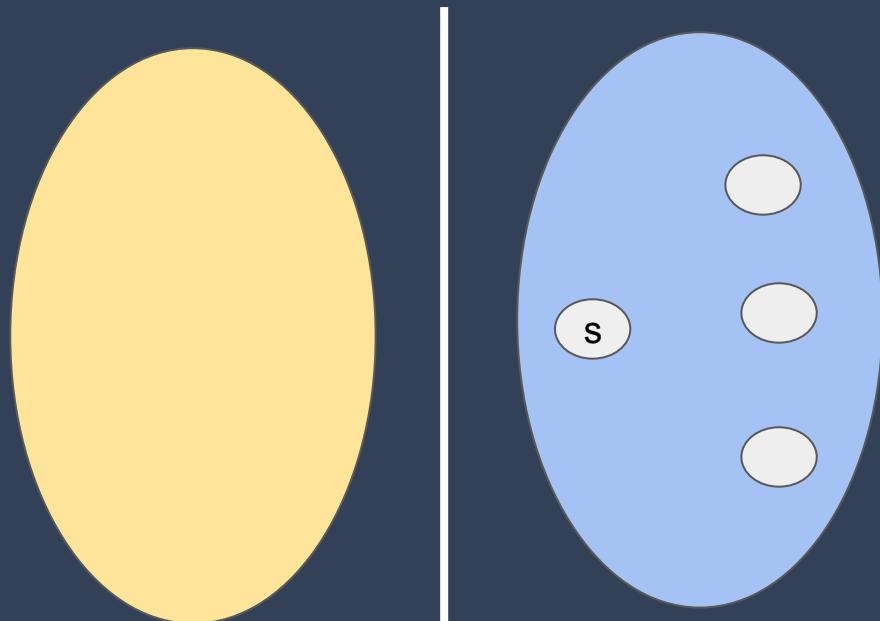


# Pseudocode for General Graph Traversal

# General Graph Traversal algorithm

```
void GeneralSearch(Vertex s):
```

Initialization: All vertices are undiscovered



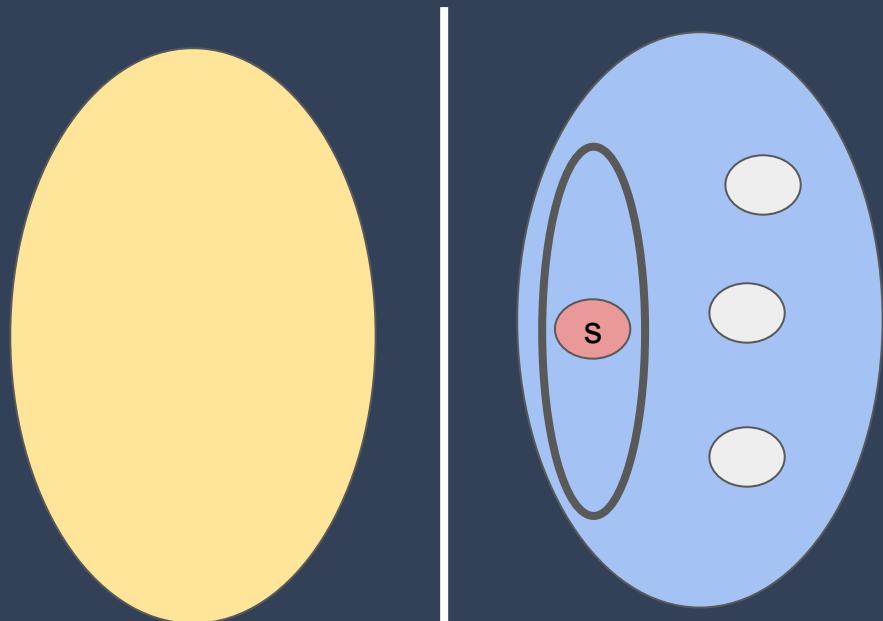
# General Graph Traversal algorithm

Initialization: All vertices are undiscovered

```
void GeneralSearch(Vertex s):
```

  Initialize an empty bag and put s in it

  Mark s as discovered



# General Graph Traversal algorithm

Initialization: All vertices are undiscovered

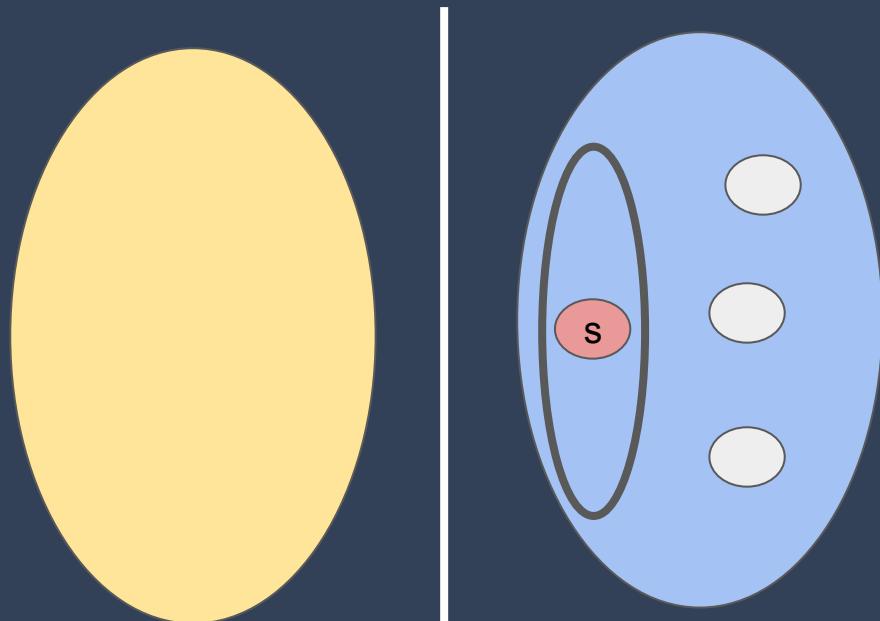
```
void GeneralSearch(Vertex s):
```

  Initialize an empty bag and put s in it

  Mark s as discovered

  while the bag is not empty:

    ???



**Fast forward to a  
general point...**

# General Graph Traversal algorithm

Initialization: All vertices are undiscovered

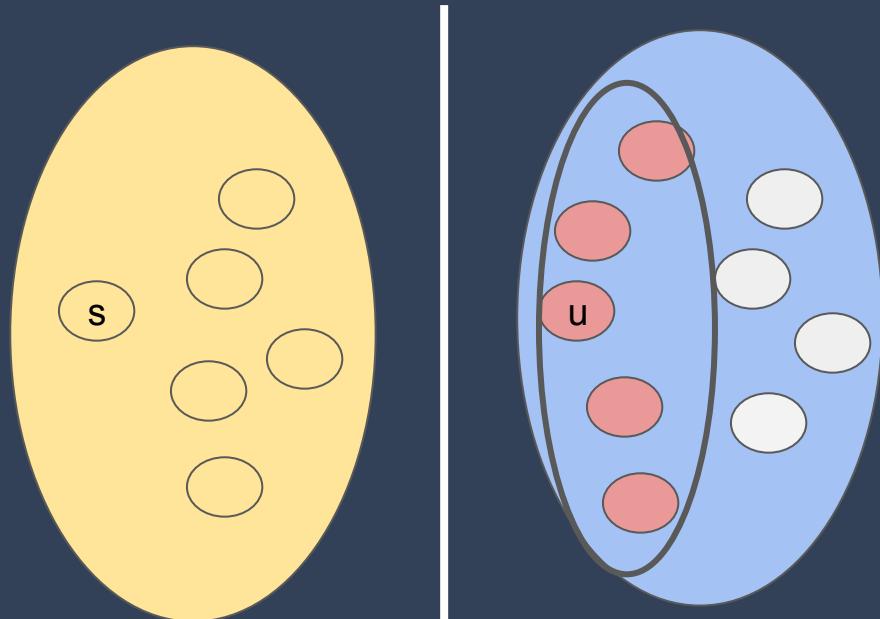
```
void GeneralSearch(Vertex s):
```

    Initialize an empty bag and put s in it

    Mark s as discovered

    while the bag is not empty:

        ???



# General Graph Traversal algorithm

Initialization: All vertices are undiscovered

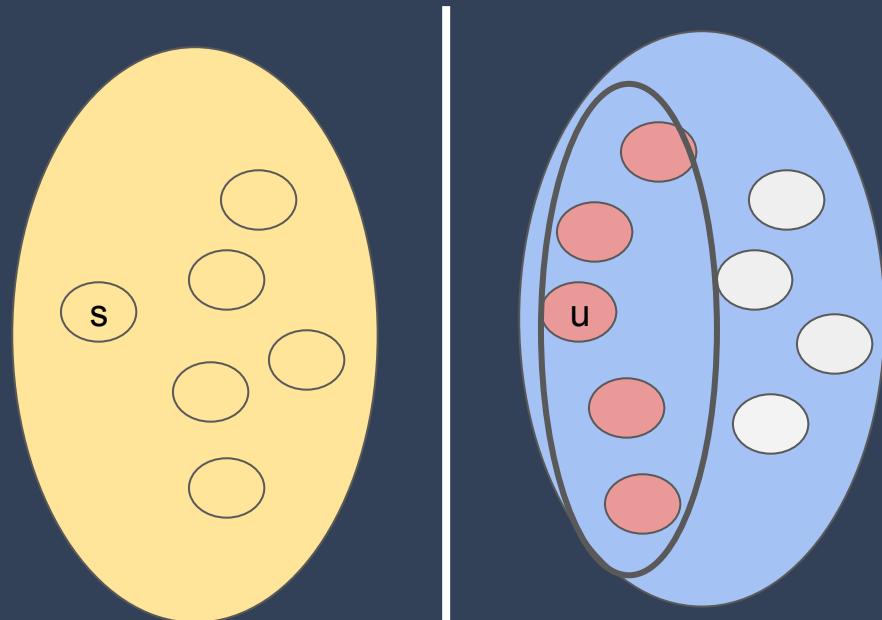
void GeneralSearch(Vertex s):

    Initialize an empty bag and put s in it

    Mark s as discovered

    while the bag is not empty:

**Pull out one of the vertices, say u**



# General Graph Traversal algorithm

Initialization: All vertices are undiscovered

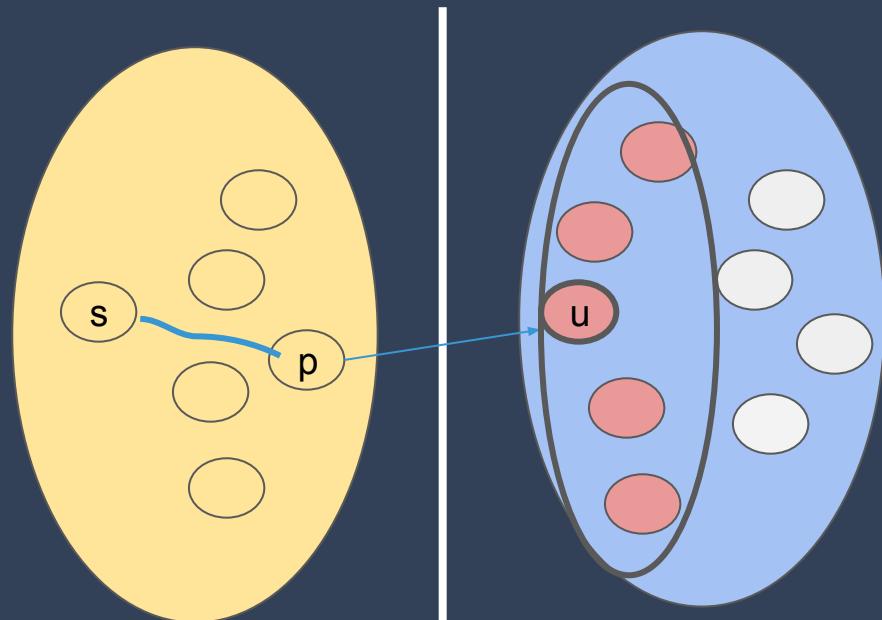
```
void GeneralSearch(Vertex s):
```

  Initialize an empty bag and put s in it

  Mark s as discovered

  while the bag is not empty:

**Pull out one of the vertices, say u**



# General Graph Traversal algorithm

Initialization: All vertices are undiscovered

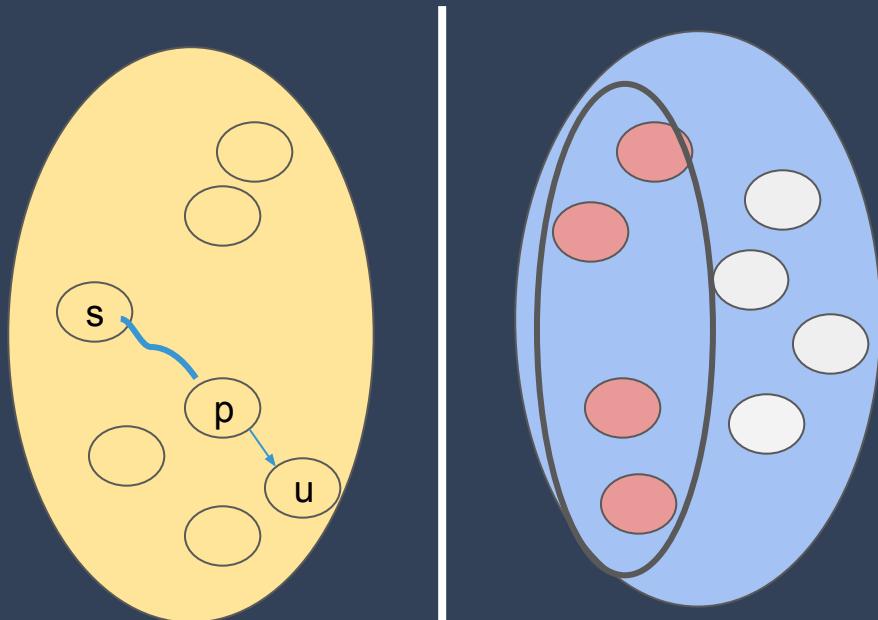
void GeneralSearch(Vertex s):

    Initialize an empty bag and put s in it

    Mark s as discovered

    while the bag is not empty:

**Pull out one of the vertices, say u**



# General Graph Traversal algorithm

```
void GeneralSearch(Vertex s):
```

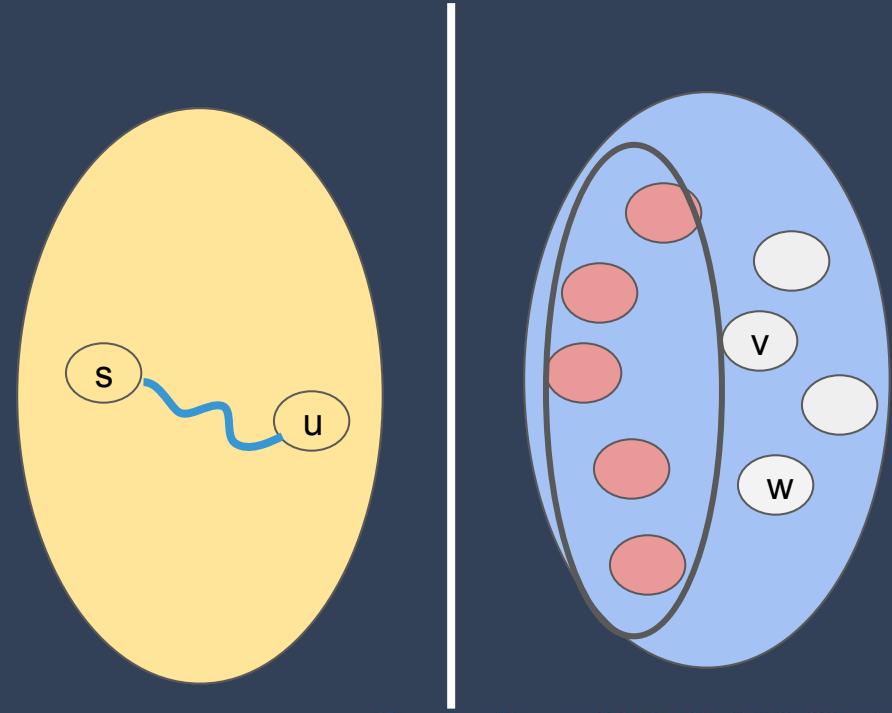
- Initialize an empty bag and put s in it

- Mark s as discovered

- while the bag is not empty:

- Pull out one of the vertices, say u**

Initialization: All vertices are undiscovered



# General Graph Traversal algorithm

```
void GeneralSearch(Vertex s):
```

    Initialize an empty bag and put s in it

    Mark s as discovered

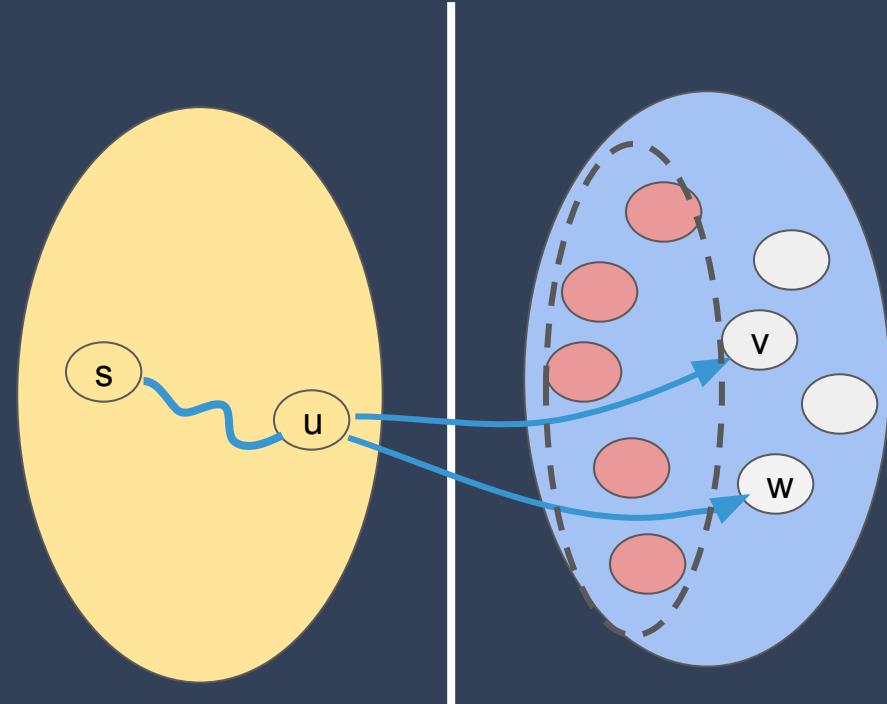
    while the bag is not empty:

**Pull out one of the vertices, say u**

        For each neighbor of u:

            if the neighbor is undiscovered:

Initialization: All vertices are undiscovered



# General Graph Traversal algorithm

```
void GeneralSearch(Vertex s):
```

    Initialize an empty bag and put s in it

    Mark s as discovered

    while the bag is not empty:

**Pull out one of the vertices, say u**

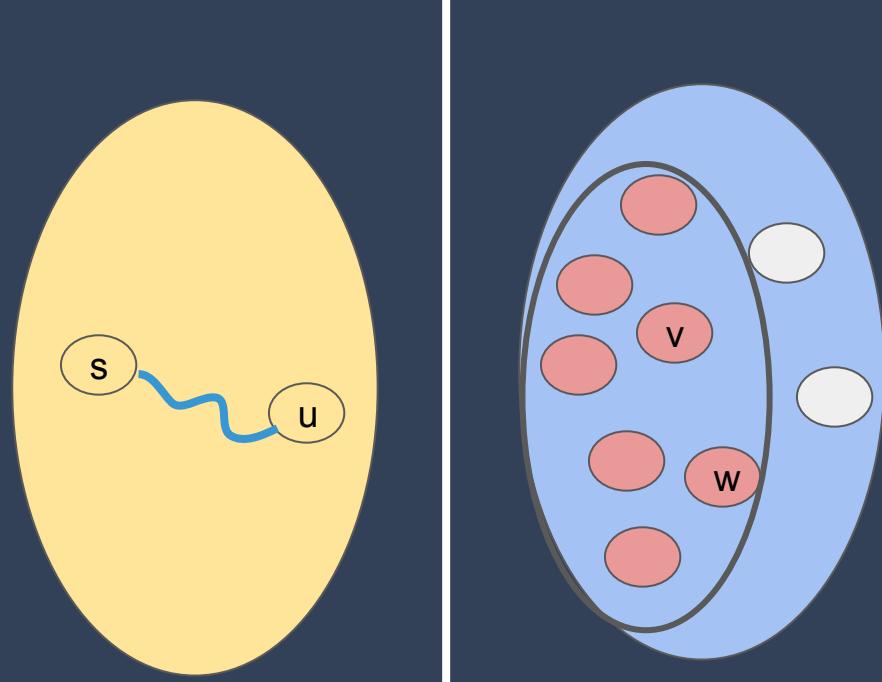
        For each neighbor of u:

            if the neighbor is undiscovered:

                put the neighbor in the bag

                mark it as discovered

Initialization: All vertices are undiscovered



# Life cycle of a vertex

```
void GeneralSearch(Vertex s):
```

  Initialize an empty bag and put s in it

  Mark s as discovered

  while the bag is not empty:

**Pull out one of the vertices, say u**

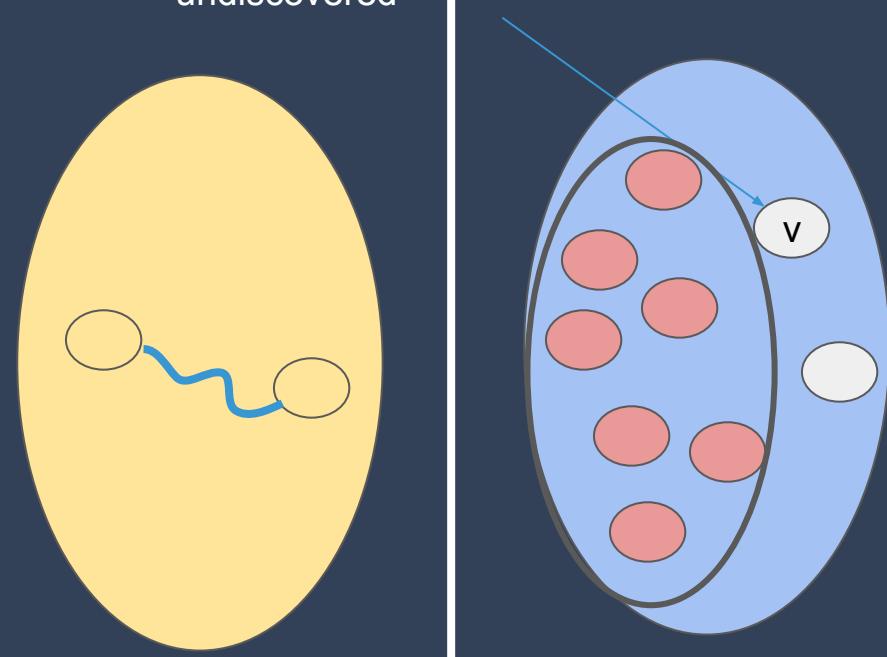
    For each neighbor of u:

      if the neighbor is undiscovered:

        put the neighbor in the bag

        mark it as discovered

Every vertex (except s) starts out undiscovered



# Life cycle for a vertex

```
void GeneralSearch(Vertex s):
```

    Initialize an empty bag and put s in it

    Mark s as discovered

    while the bag is not empty:

**Pull out one of the vertices, say u**

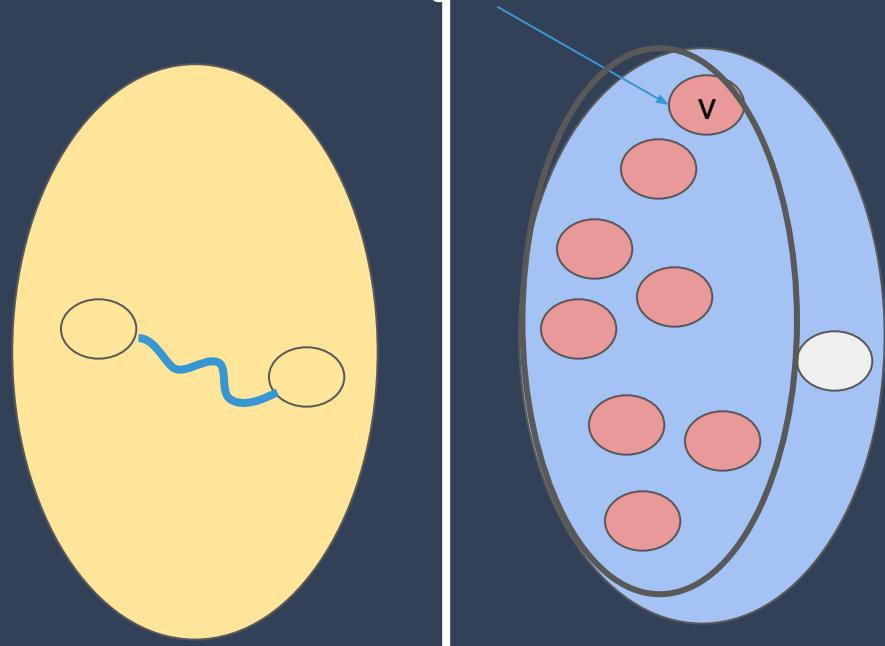
        For each neighbor of u:

            if the neighbor is undiscovered:

                put the neighbor in the bag

                mark it as discovered

At some point, it is discovered and added to the bag



# Life cycle of a vertex

```
void GeneralSearch(Vertex s):
```

  Initialize an empty bag and put s in it

  Mark s as discovered

  while the bag is not empty:

**Pull out one of the vertices, say u**

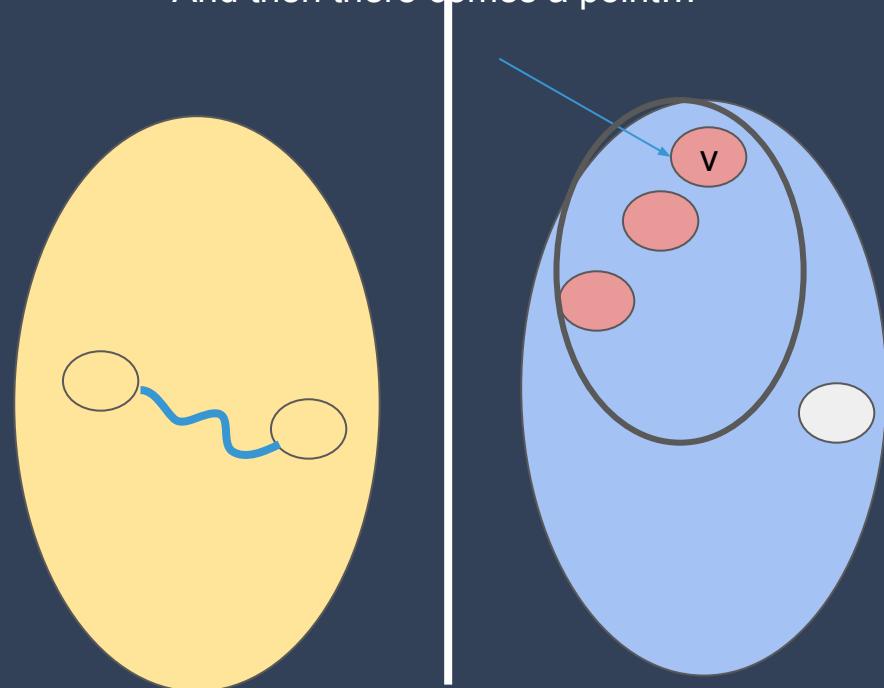
    For each neighbor of u:

      if the neighbor is undiscovered:

        put the neighbor in the bag

        mark it as discovered

And then there comes a point...



# Life cycle of a vertex

void GeneralSearch(Vertex s):

    Initialize an empty bag and put s in it

    Mark s as discovered

    while the bag is not empty:

**Pull out one of the vertices, say u**

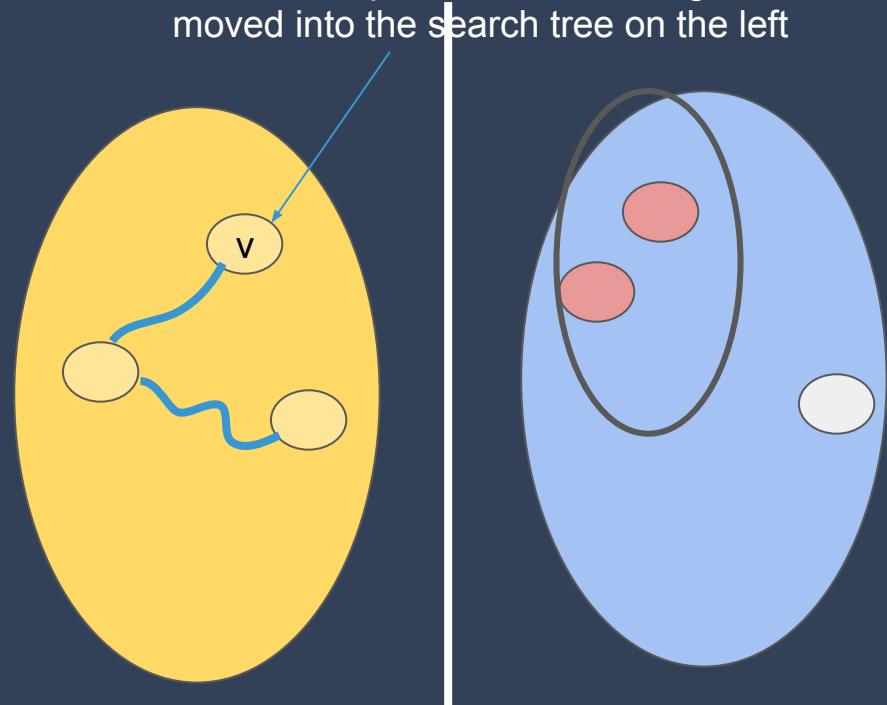
        For each neighbor of u:

            if the neighbor is undiscovered:

                put the neighbor in the bag

                mark it as discovered

...when it is pulled out of the bag and moved into the search tree on the left



# How do we maintain the collection of vertices in the bag?

If we use a **FIFO Queue** to store the vertices, we get **BFS**.

If we use a **LIFO Stack** to store the vertices, we get **DFS**.

If we use a **Priority Queue** to store the vertices, then each vertex is inserted into the bag with some priority. The vertex that is next pulled out of the bag is the one with the highest priority. Depending on how the priority is defined, we get **Dijkstra's** algorithm, **Prim's** algorithm or **Best-first/A\***.

## Basic Graph Theory

## Graph representations

## General graph traversal

All of these are variations of the General Graph traversal algorithm.

BFS

DFS

Dijkstra

Prim

Best-first

A\*

Shortest paths

1. Traversals of undirected and directed graphs
2. Finding connected components
3. Detecting cycles
4. Bipartite

top.sort

SCC etc.

Bellman-Ford  
Floyd-Warshall

Kruskal

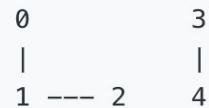
### 323. Number of Connected Components in an Undirected Graph

Medium    465    14    Favorite    Share

Given  $n$  nodes labeled from  $0$  to  $n - 1$  and a list of undirected edges (each edge is a pair of nodes), write a function to find the number of connected components in an undirected graph.

#### Example 1:

**Input:**  $n = 5$  and  $\text{edges} = [[0, 1], [1, 2], [3, 4]]$



**Output:** 2

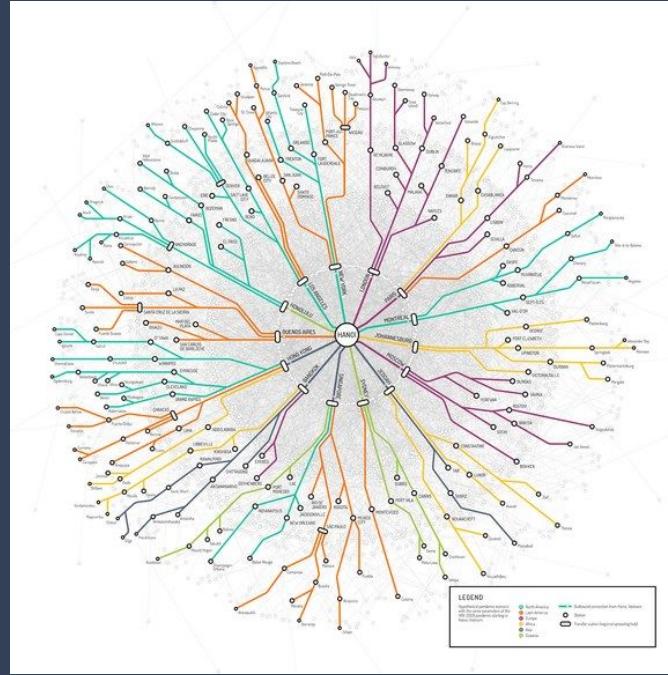
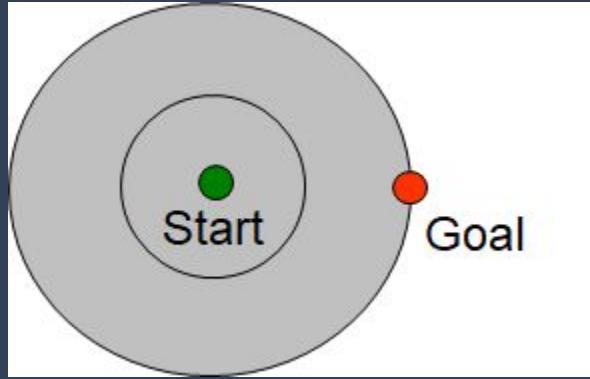
#### Example 2:

**Input:**  $n = 5$  and  $\text{edges} = [[0, 1], [1, 2], [2, 3], [3, 4]]$



**Output:** 1

# BFS

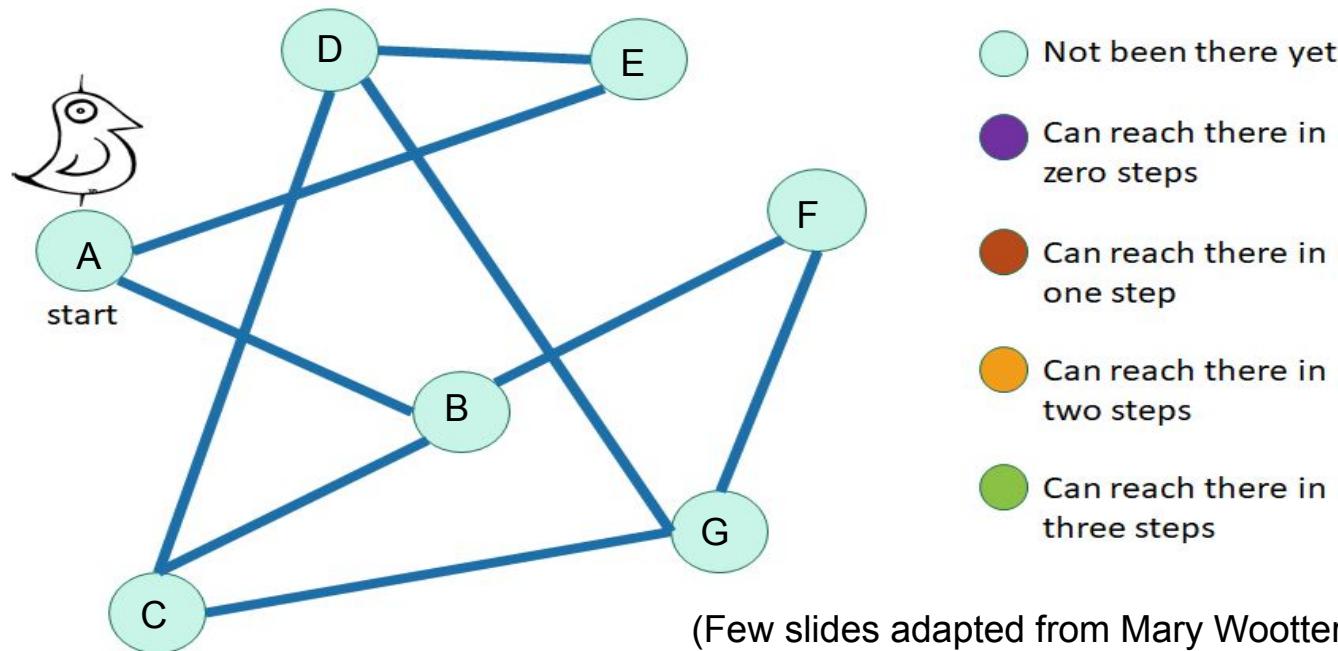


{ik}

INTERVIEW  
KICKSTART

# Breadth-First Search (Undirected graph example)

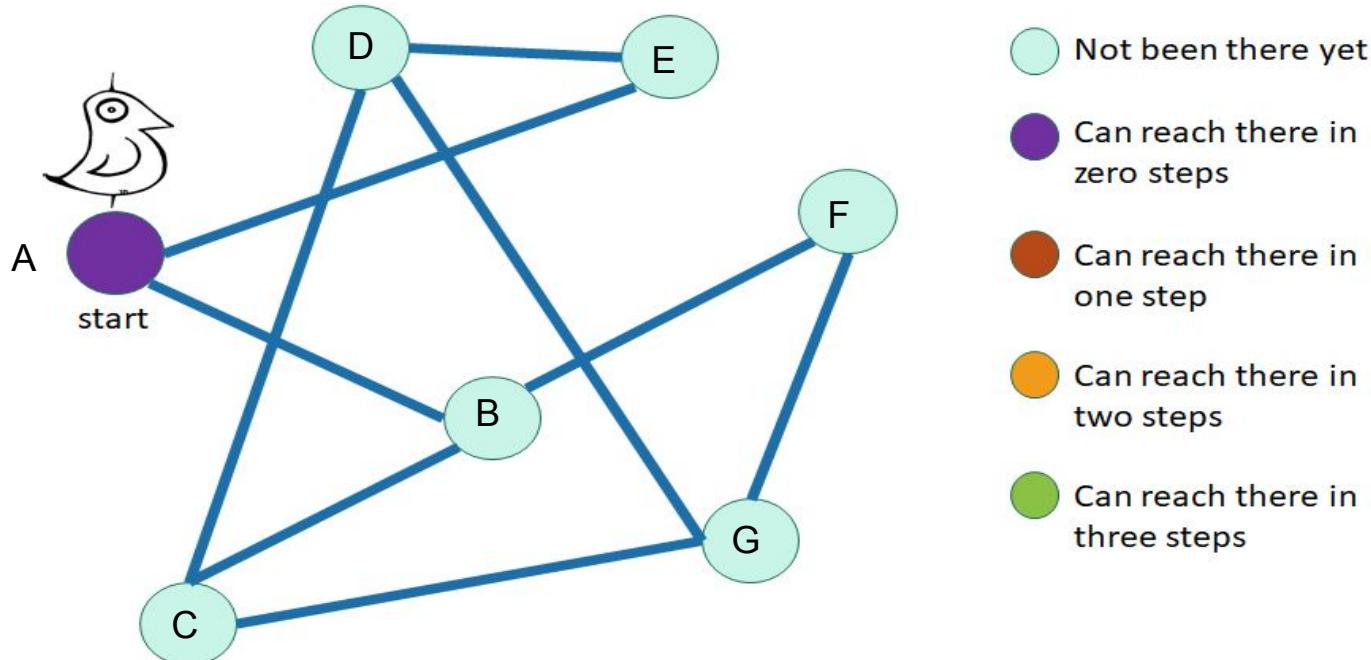
Exploring the world with a bird's-eye view



# Breadth-First Search

Exploring the world with a bird's-eye view

$q = [A]$   
visited = {A}

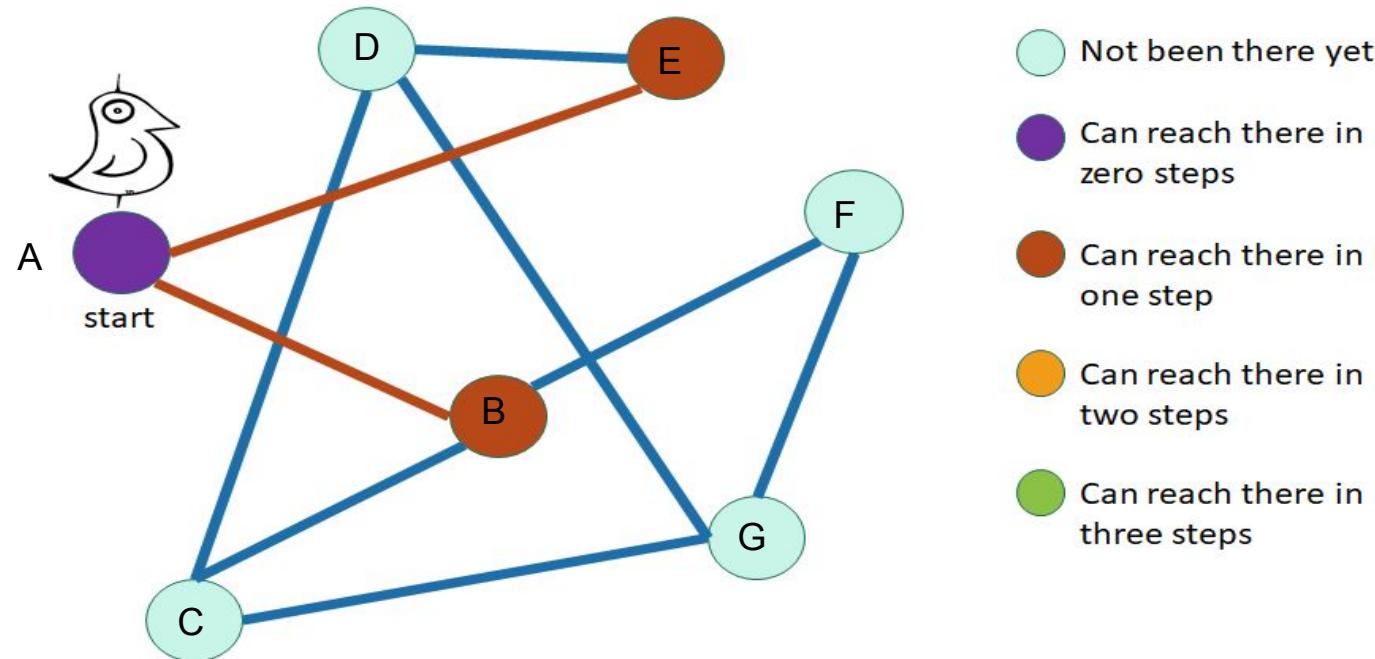


- Not been there yet
- Can reach there in zero steps
- Can reach there in one step
- Can reach there in two steps
- Can reach there in three steps

Adjacency list:  
A -> B, E  
B -> C, F, A  
C -> D, B, G  
D -> C, E, G  
E -> A, D  
F -> B, G  
G -> D, F, C

# Breadth-First Search

Exploring the world with a bird's-eye view



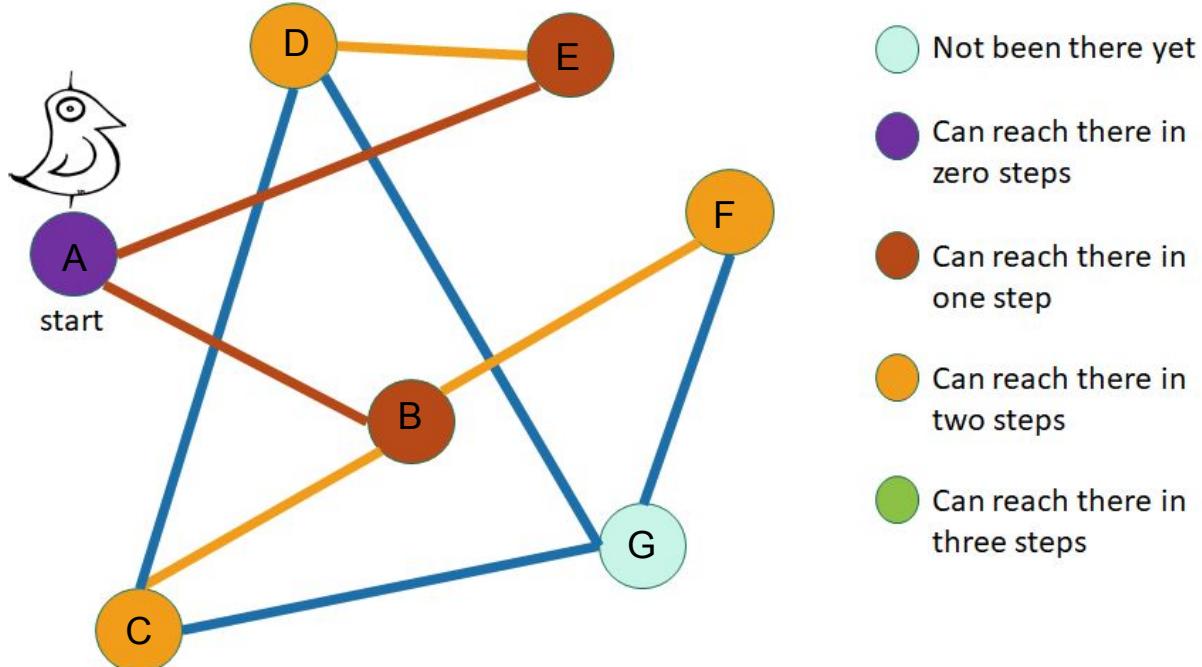
$q = [B, E]$   
visited = {A, B, E}

Adjacency list:

A -> B, E  
B -> C, F, A  
C -> D, B, G  
D -> C, E, G  
E -> A, D  
F -> B, G  
G -> D, F, C

# Breadth-First Search

Exploring the world with a bird's-eye view

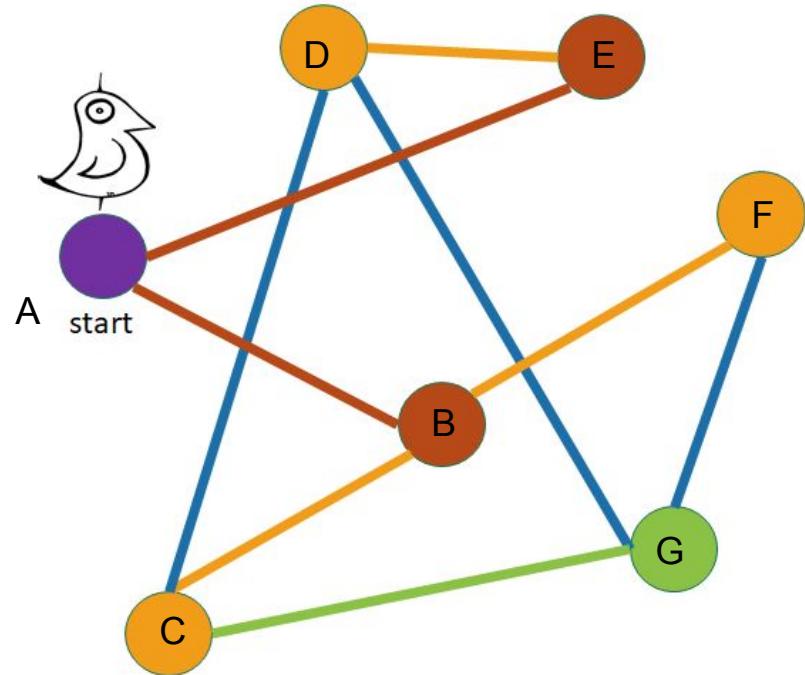


$$q = [C, F, D]$$

$$\text{visited} = \{A, B, C, D, E, F\}$$

# Breadth-First Search

Exploring the world with a bird's-eye view



Not been there yet

Can reach there in zero steps

Can reach there in one step

Can reach there in two steps

Can reach there in three steps

World:  
explored!

$$q = [G]$$

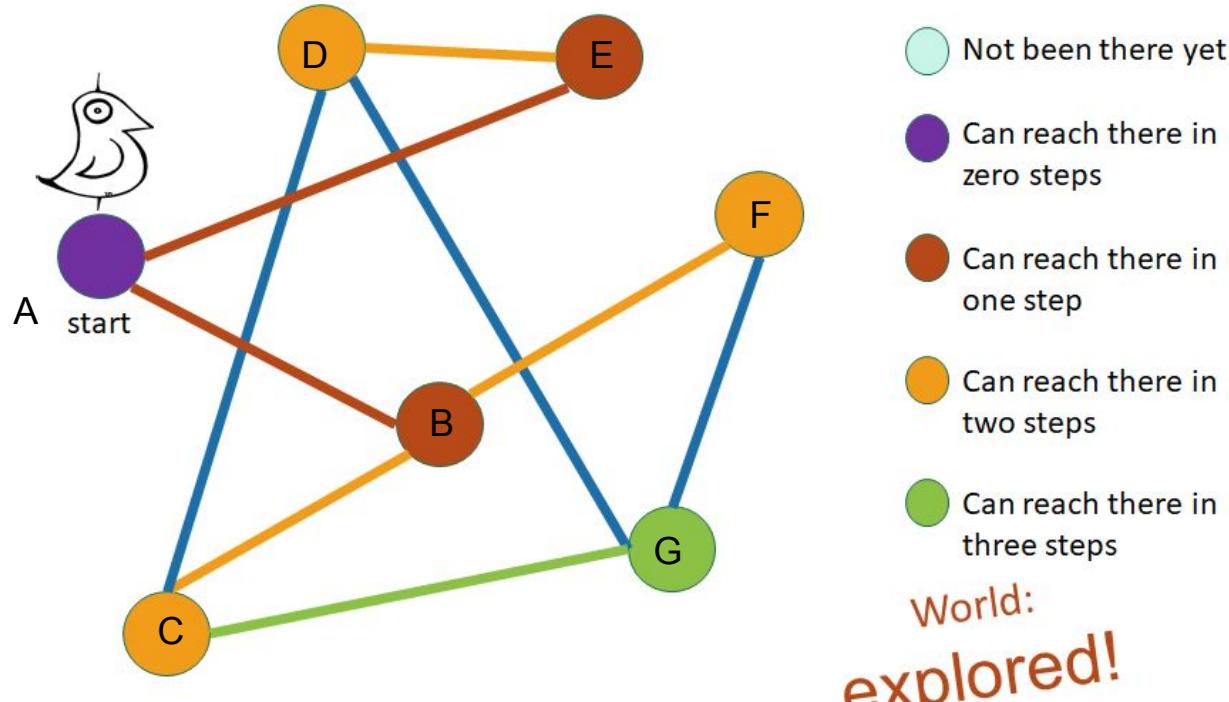
visited = {A, B, C, D, E, F, G}

Adjacency list:

A -> B, E  
B -> C, F, A  
C -> D, B, G  
D -> C, E, G  
E -> A, D  
F -> B, G  
G -> D, F, C

# Breadth-First Search

Exploring the world with a bird's-eye view

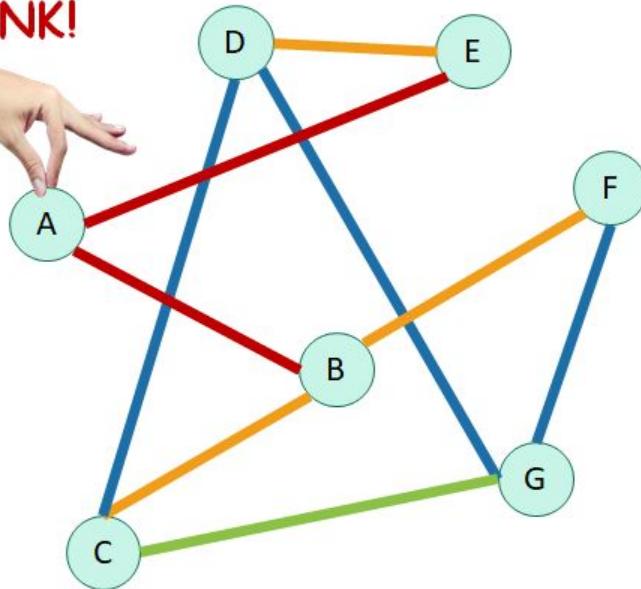


$q = []$   
 $visited = \{A, B, C, D, E, F, G\}$

Adjacency list:  
A -> B, E  
B -> C, F, A  
C -> D, B, G  
D -> C, E, G  
E -> A, D  
F -> B, G  
G -> D, F, C

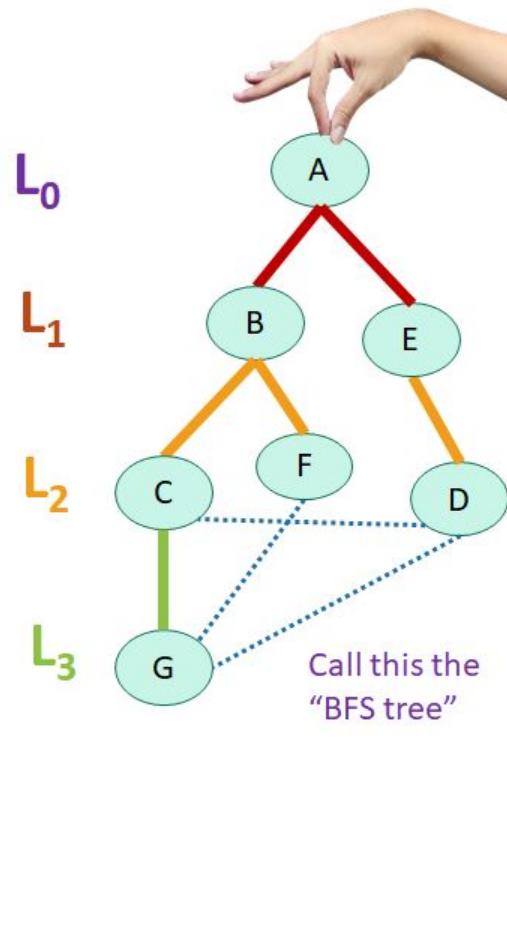
- We are implicitly building a tree:

**YOINK!**



Notice layer partitioning.

Can we have cross edges skipping a layer?



Tree edge

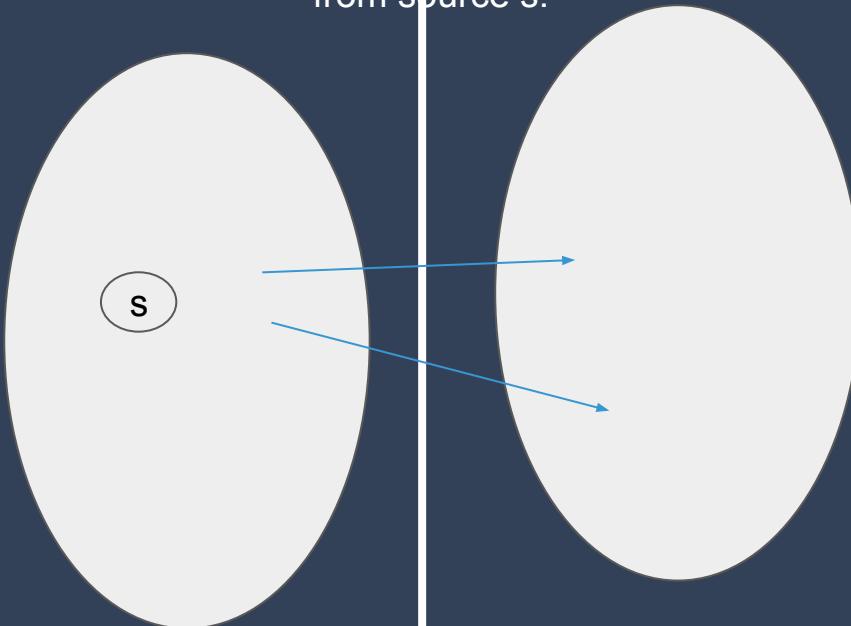
Cross edge

Adjacency list:  
 A -> B, E  
 B -> C, F, A  
 C -> D, B, G  
 D -> C, E, G  
 E -> A, D  
 F -> B, G  
 G -> D, F, C

```
class Graph {  
  
void BFS(int s) {  
    //visited and parent initialized to 0 and null  
    visited[s] = 1  
  
    q = new Queue(); q.push(s);  
    while not isEmpty(q):  
        v = q.pop()  
        for w in Adjlist[v]:  
            if visited[w] == 0 then  
                visited[w] = 1; parent[w] = v;  
                q.push(w);  
  
    }  
}
```

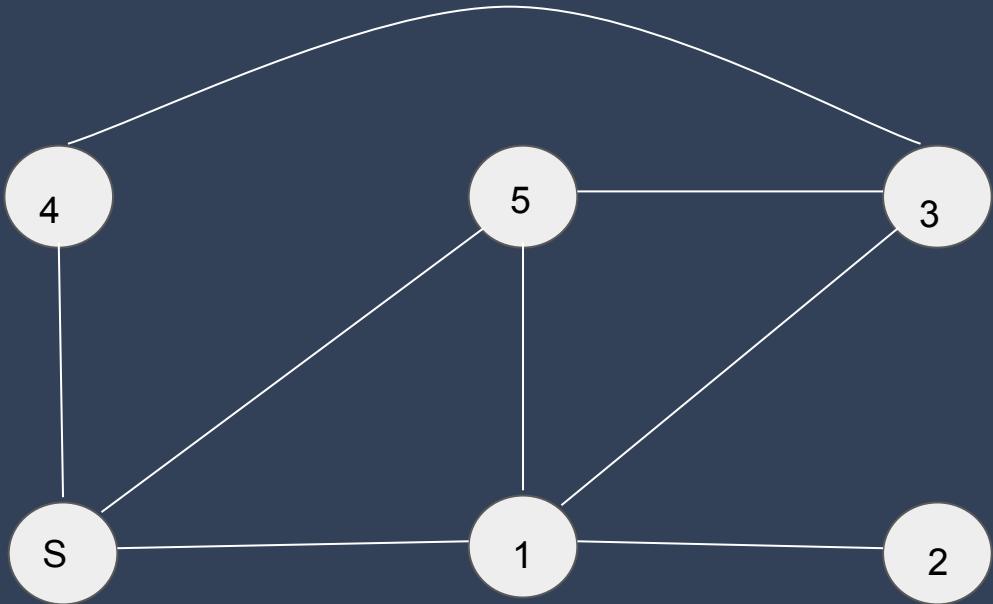
# BFS

Vertices are visited in increasing order of distance from source s.

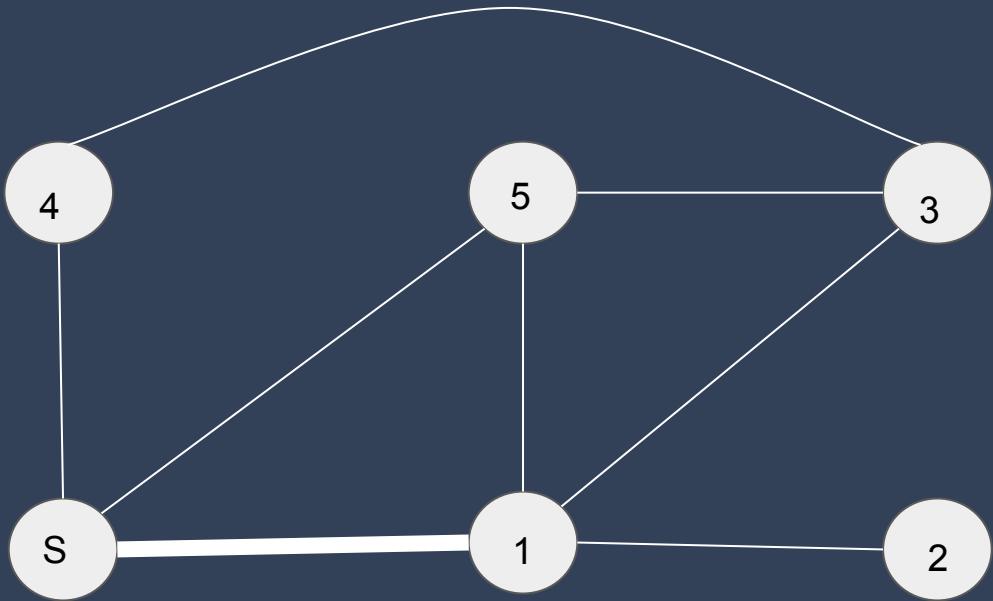


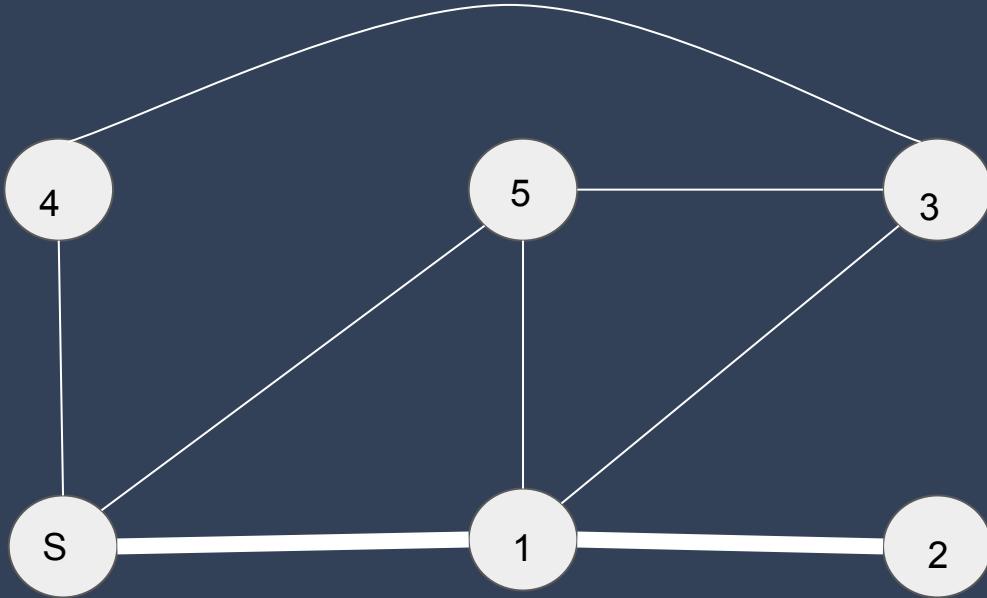
Captured

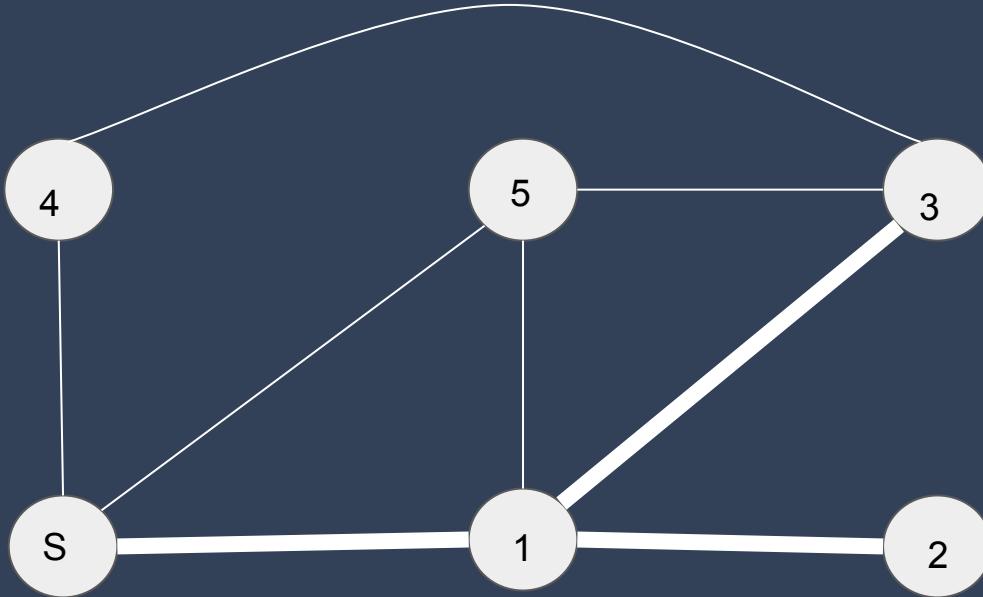
Yet to capture  
**{ik}** **INTERVIEW**  
**KICKSTART**

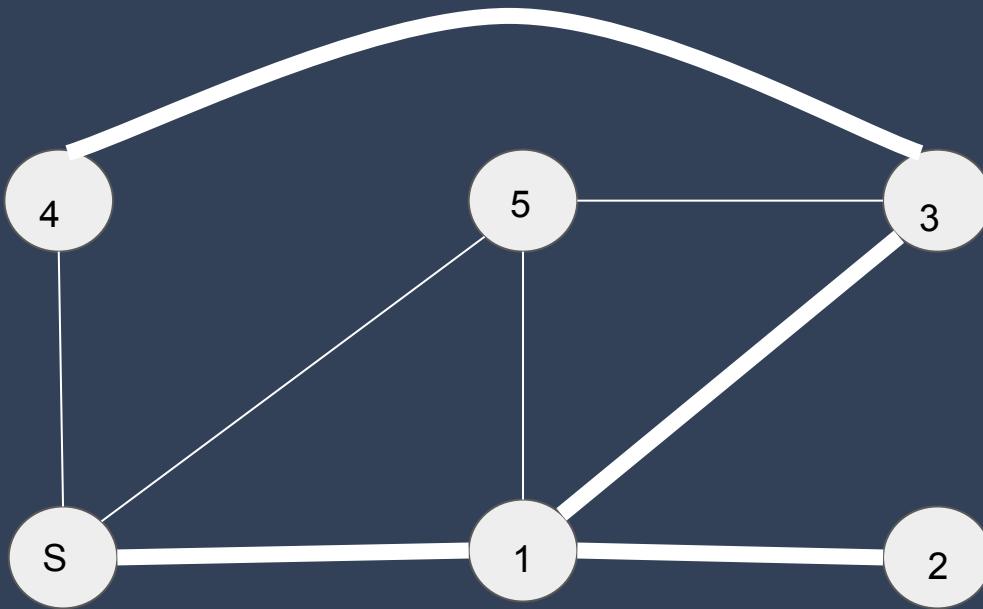


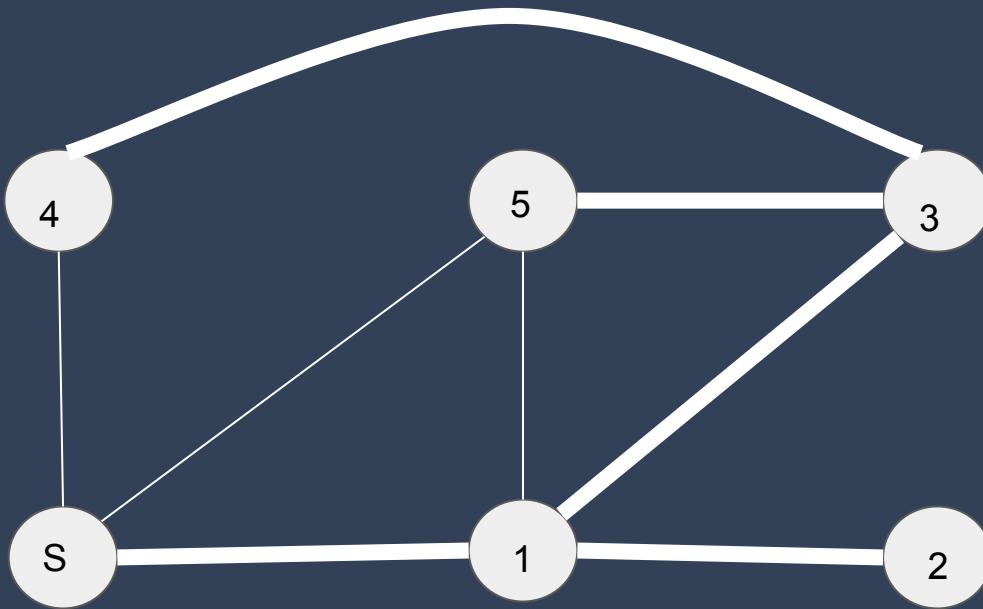
DFS

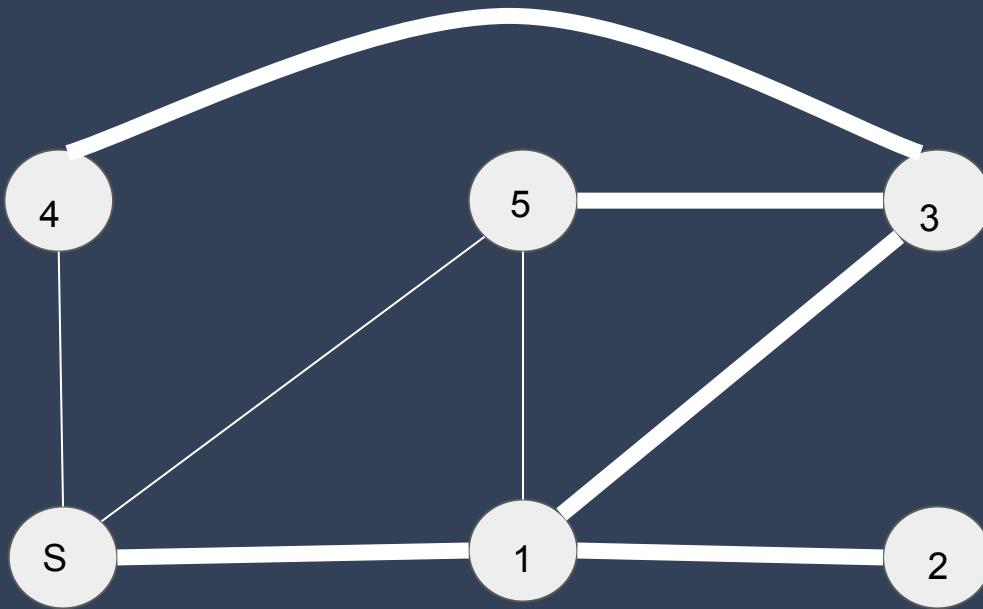


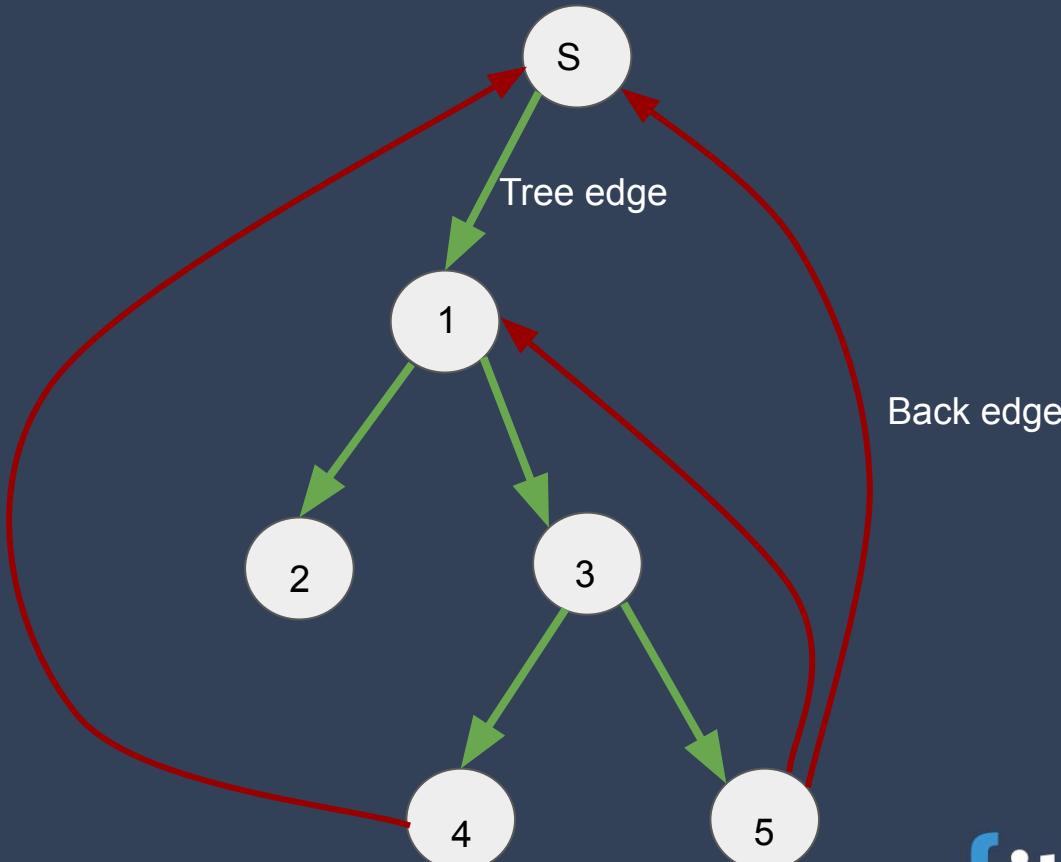












```
class Graph {  
    ....  
    //adjList is an array of lists
```

```
    void DFS(int source) {  
        visited[source] = 1  
        for neighbor in adjList[source]:  
            if visited[neighbor] == -1:  
                DFS(neighbor)  
    }  
}
```

### 323. Number of Connected Components in an Undirected Graph

Medium

465

14

Favorite

Share

Given  $n$  nodes labeled from  $0$  to  $n - 1$  and a list of undirected edges (each edge is a pair of nodes), write a function to find the number of connected components in an undirected graph.

#### Example 1:

**Input:**  $n = 5$  and  $\text{edges} = [[0, 1], [1, 2], [3, 4]]$



**Output:** 2

#### Example 2:

**Input:**  $n = 5$  and  $\text{edges} = [[0, 1], [1, 2], [2, 3], [3, 4]]$



**Output:** 1

```
1 ▼ class Solution(object):
2 ▼     def countComponents(self, n, edges):
3
4         :type n: int
5         :type edges: List[List[int]]
6         :rtype: int
7
8         adjlist = [ [] for _ in range(n)]
9 ▼     for (src,dst) in edges:
10            adjlist[src].append(dst)
11            adjlist[dst].append(src)
12
13    visited = [-1] * n
14
15 ▼     def dfs(source):
16        visited[source] = 1
17        for neighbor in adjlist[source]:
18            if visited[neighbor] == -1:
19                dfs(neighbor)
```

```
21 def bfs(source):
22     visited[source] = 1
23     q = collections.deque([source])
24     while len(q) != 0:
25         node = q.popleft()
26         for neighbor in adjlist[node]:
27             if visited[neighbor] == -1:
28                 visited[neighbor] = 1
29                 q.append(neighbor)
30
31     components = 0
32     for v in range(n):
33         if visited[v] == -1:
34             components += 1
35             dfs(v)
36
37     return components
```

## 261. Graph Valid Tree

Medium    720    22    Favorite    Share

Given `n` nodes labeled from `0` to `n-1` and a list of undirected edges (each edge is a pair of nodes), write a function to check whether these edges make up a valid tree.

### Example 1:

**Input:** `n = 5`, and `edges = [[0,1], [0,2], [0,3], [1,4]]`

**Output:** `true`

### Example 2:

**Input:** `n = 5`, and `edges = [[0,1], [1,2], [2,3], [1,3], [1,4]]`

**Output:** `false`

**Note:** you can assume that no duplicate edges will appear in `edges`. Since all edges are undirected, `[0,1]` is the same as `[1,0]` and thus will not appear together in `edges`.

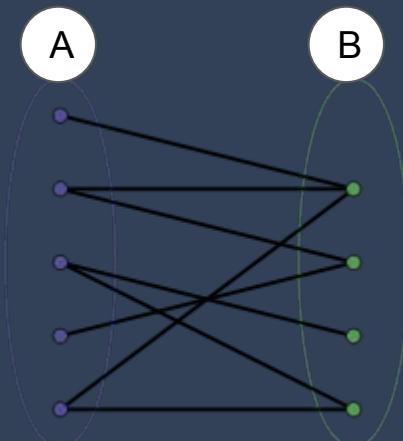


```
29 ▼             def bfs(source):
30                  visited[source] = 1
31                  q = collections.deque([source])
32 ▼                  while len(q) != 0:
33                      node = q.popleft()
34 ▼                      for neighbor in adjlist[node]:
35                          if visited[neighbor] == -1:
36                              visited[neighbor] = 1
37                              parent[neighbor] = node
38                              q.append(neighbor)
39 ▼                      else:
40                          if parent[node] != neighbor:
41                              #Cycle
42                              return True
43
44                  return False
```

```
46             components = 0
47     ▼         for v in range(n):
48     ▼             if visited[v] == -1:
49                 components += 1
50     ▼             if components > 1:
51                     return False
52     ▼             if bfs(v):
53                 #If cycle found, it is not a tree
54                     return False
55
      return True
```

**Given an undirected graph, return true if and only if it is bipartite.**

**Recall that a graph is bipartite if we can split its set of nodes into two independent subsets A and B such that every edge in the graph has one node in A and another node in B.**



## 785. Is Graph Bipartite?

Medium    698    89    Favorite    Share

Given an undirected graph, return true if and only if it is bipartite.

Recall that a graph is *bipartite* if we can split its set of nodes into two independent subsets A and B such that every edge in the graph has one node in A and another node in B.

The graph is given in the following form: `graph[i]` is a list of indexes `j` for which the edge between nodes `i` and `j` exists. Each node is an integer between `0` and `graph.length - 1`. There are no self edges or parallel edges: `graph[i]` does not contain `i`, and it doesn't contain any element twice.

**Example 1:**

**Input:** [[1,3], [0,2], [1,3], [0,2]]

**Output:** true

**Explanation:**

The graph looks like this:

0----1

| |

| |

3----2

We can divide the vertices into two groups: {0, 2} and {1, 3}.

Can the vertices of a given graph  $G$  be colored using only two colors so that adjacent vertices have different colors?

```
7  
8     adjlist = graph  
9     n = len(graph)  
10  
11    visited = [-1] * n  
12    parent = [-1] * n  
13    distance = [-1] * n  
14
```

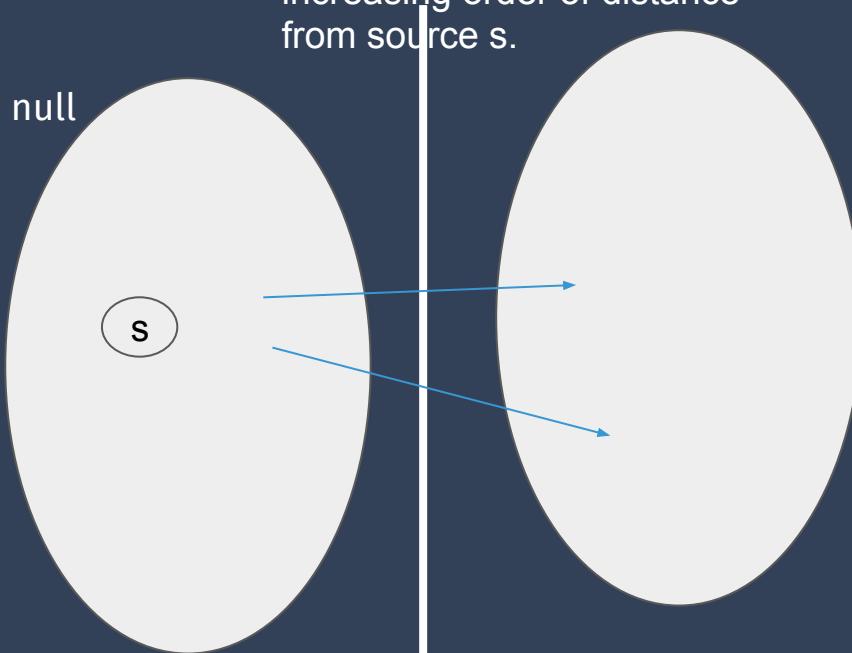
```
28 def bfs(source):
29     visited[source] = 1
30     distance[source] = 0
31     q = collections.deque([source])
32     while len(q) != 0:
33         node = q.popleft()
34         for neighbor in adjlist[node]:
35             if visited[neighbor] == -1:
36                 visited[neighbor] = 1
37                 parent[neighbor] = node
38                 distance[neighbor] = distance[node] + 1
39                 q.append(neighbor)
40             else:
41                 if parent[node] != neighbor:
42                     #Cycle
43                     if distance[node] == distance[neighbor]:
44                         #Not bipartite (odd length cycle found)
45                         return False
46     return True #Bipartite
47
```

```
48
49         components = 0
50     for v in range(n):
51         if visited[v] == -1:
52             components += 1
53             #if components > 1:
54                 #return False
55     if bfs(v) == False:
56         #Even if a single bfs call returns False,
57         #i.e, even if one connected component isn't bipartite,
58         #...then the whole graph isn't bipartite
59         return False
60     return True  #ALL connected components were bipartite, so the graph is bipartite
```

# BFS

```
class Graph {  
    void BFS(int s) {  
        //visited, captured and parent initialized to 0, 0 and null  
        captured[s] = 1; visited[s] = 1  
  
        q = new Queue(); q.push(s);  
        while not isEmpty(q): //capture the next vertex  
            v = q.pop()  
            captured[v] = 1  
            for w in Adjlist[v]:  
                if visited[w] == 0 then  
                    visited[w] = 1; parent[w] = v;  
                    q.push(w);  
    }  
}
```

Vertices are captured in increasing order of distance from source s.

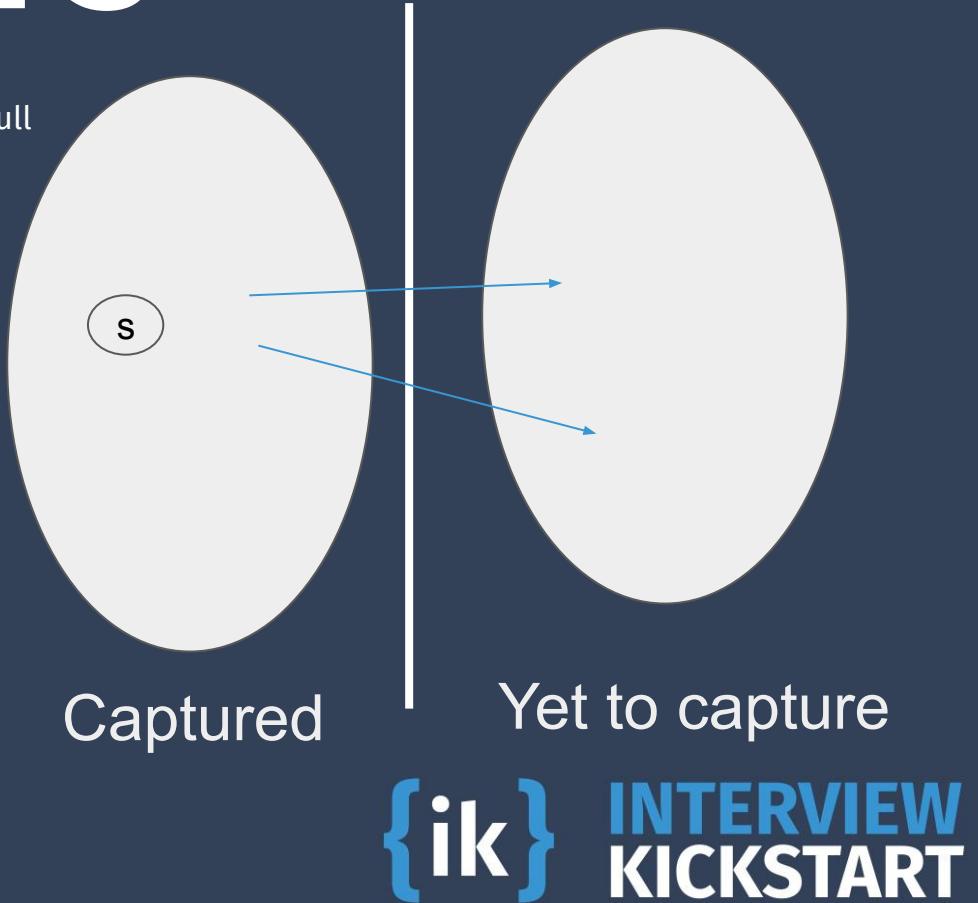


Captured

Yet to capture

# BFS

```
class Graph {  
  
void BFS(int s) {  
    //d, visited, captured and parent initialized to INF, 0, 0 and null  
    d[s] = 0; captured[s] = 1; visited[s] = 1  
  
    q = new Queue(); q.push(s);  
    while not isEmpty(q): //capture the next vertex  
        v = q.pop()  
        captured[v] = 1  
        for w in Adjlist[v]:  
            if visited[w] == 0 then  
                visited[w] = 1; parent[w] = v; d[w] = d[v] + 1  
                q.push(w);  
            else  
                if d[w] == d[v] then graph is not bipartite  
    //Conclusion: Graph is bipartite  
}  
}
```



```
8      adjlist = graph
9      n = len(graph)
10
11     visited = [-1] * n
12     parent = [-1] * n
13     distance = [-1] * n
14     color = [-1] * n
15
```

```
33 ▼         def dfs(source):
34             visited[source] = 1
35 ▼             if parent[source] == -1:
36                 color[source] = 0
37 ▼             else:
38                 color[source] = 1 - color[parent[source]]
39 ▼             for neighbor in adjlist[source]:
40                 if visited[neighbor] == -1:
41                     parent[neighbor] = source
42 ▼                     if dfs(neighbor) == False:
43                         return False
44 ▼                     else:
45 ▼                         if color[source] == color[neighbor]:
46                             return False
```

```
52     components = 0
53 ▼     for v in range(n):
54 ▼         if visited[v] == -1:
55             components += 1
56             #if components > 1:
57             #    return False
58 ▼         if dfs(v, 0) == False: #0 is default color
59             #Even if a single bfs call returns False,
60             #i.e, even if one connected component isn't bipartite,
61             #...then the whole graph isn't bipartite
62             return False
63     return True #ALL connected components were bipartite, so the graph is bipartite
```

## 886. Possible Bipartition

Medium    313    14    Favorite    Share

Given a set of `N` people (numbered `1, 2, ..., N`), we would like to split everyone into two groups of **any** size.

Each person may dislike some other people, and they should not go into the same group.

Formally, if `dislikes[i] = [a, b]`, it means it is not allowed to put the people numbered `a` and `b` into the same group.

Return `true` if and only if it is possible to split everyone into two groups in this way.

### Example 1:

**Input:** `N = 4, dislikes = [[1,2],[1,3],[2,4]]`

**Output:** `true`

**Explanation:** `group1 [1,4], group2 [2,3]`

### Example 2:

**Input:** `N = 3, dislikes = [[1,2],[1,3],[2,3]]`

**Output:** `false`

### Example 3:

**Input:** `N = 5, dislikes = [[1,2],[2,3],[3,4],[4,5],[1,5]]`

**Output:** `false`

```
8          n = N+1
9          adjlist = [ [] for _ in range(n)]
10         for (src,dst) in dislikes:
11             adjlist[src].append(dst)
12             adjlist[dst].append(src)
13
```

OR

```
8          n = N
9          adjlist = [ [] for _ in range(n)]
10         for (src,dst) in dislikes:
11             adjlist[src-1].append(dst-1)
12             adjlist[dst-1].append(src-1)
13
```

## 200. Number of Islands

Medium    2809    100    Favorite    Share

Given a 2d grid map of '1' s (land) and '0' s (water), count the number of islands. An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

### Example 1:

**Input:**

11110

11010

11000

00000

**Output:** 1

### Example 2:

**Input:**

11000

11000

00100

00011

**Output:** 3

```
1  class Solution(object):
2      def numIslands(self, grid):
3          """
4              :type grid: List[List[str]]
5              :rtype: int
6          """
7
8      def getneighbors(x,y):
9          result = []
10     if x+1 < len(grid):
11         result.append((x+1,y))
12     if y+1 < len(grid[0]):
13         result.append((x,y+1))
14     if x-1 >= 0:
15         result.append((x-1,y))
16     if y-1 >= 0:
17         result.append((x,y-1))
18     return result
19
20     def bfs(i,j):
21         q = collections.deque()
22         q.append((i,j))
23         grid[i][j] = '0'
24     while len(q) != 0:
25         (row,col) = q.popleft()
26         neighbors = getneighbors(row,col)
27         for (nr,nc) in neighbors:
28             if grid[nr][nc] != '0':
29                 q.append((nr,nc))
30                 grid[nr][nc] = '0'
31
32     islands = 0
33     for x in range(len(grid)):
34         for y in range(len(grid[0])):
35             if grid[x][y] != '0':
36                 bfs(x,y)
37                 islands += 1
38
39     return islands
```

```
1 v     class Solution(object):
2 v         def numIslands(self, grid):
3 v             """
4 v                 :type grid: List[List[str]]
5 v                 :rtype: int
6 v             """
7
8 v         def getneighbors(x,y):
9 v             result = []
10 v            if x+1 < len(grid):
11 v                result.append((x+1,y))
12 v                if y+1 < len(grid[0]):
13 v                    result.append((x,y+1))
14 v                    if x-1 >= 0:
15 v                        result.append((x-1,y))
16 v                        if y-1 >= 0:
17 v                            result.append((x,y-1))
18 v                            return result
19
20 v        def dfs(i,j):
21 v            grid[i][j] = '0'
22 v            neighbors = getneighbors(i,j)
23 v            for (nr,nc) in neighbors:
24 v                if grid[nr][nc] != '0':
25 v                    dfs(nr,nc)
26
27         islands = 0
28 v        for x in range(len(grid)):
29 v            for y in range(len(grid[0])):
30 v                if grid[x][y] != '0':
31 v                    dfs(x,y)
32 v                    islands += 1
33
34         return islands
```

Note: Set  $\text{grid}[x][y] = 0$  when  $(x,y)$  is visited, not when it is captured. Otherwise, the queue size can become very large, with duplicate entries packed inside, and you would get a “time limit exceeded” issue on online coding platforms.

## 695. Max Area of Island

Medium    1189    63    Favorite    Share

Given a non-empty 2D array `grid` of 0's and 1's, an **island** is a group of 1's (representing land) connected 4-directionally (horizontal or vertical.) You may assume all four edges of the grid are surrounded by water.

Find the maximum area of an island in the given 2D array. (If there is no island, the maximum area is 0.)

**Example 1:**

```
[[0,0,1,0,0,0,0,1,0,0,0,0,0],  
 [0,0,0,0,0,0,0,1,1,0,0,0],  
 [0,1,1,0,1,0,0,0,0,0,0,0],  
 [0,1,0,0,1,1,0,0,1,0,1,0,0],  
 [0,1,0,0,1,1,0,0,1,1,1,0,0],  
 [0,0,0,0,0,0,0,0,0,1,0,0],  
 [0,0,0,0,0,0,0,1,1,1,0,0,0],  
 [0,0,0,0,0,0,1,1,0,0,0,0]]
```

Given the above grid, return 6. Note the answer is not 11, because the island must be connected 4-directionally.

**Example 2:**

```
[[0,0,0,0,0,0,0,0]]
```

Given the above grid, return 0.

**Note:** The length of each dimension in the given `grid` does not exceed 50.

## 733. Flood Fill

Easy    493    114    Favorite    Share

An `image` is represented by a 2-D array of integers, each integer representing the pixel value of the image (from 0 to 65535).

Given a coordinate `(sr, sc)` representing the starting pixel (row and column) of the flood fill, and a pixel value `newColor`, "flood fill" the image.

To perform a "flood fill", consider the starting pixel, plus any pixels connected 4-directionally to the starting pixel of the same color as the starting pixel, plus any pixels connected 4-directionally to those pixels (also with the same color as the starting pixel), and so on. Replace the color of all of the aforementioned pixels with the `newColor`.

At the end, return the modified image.

### Example 1:

#### Input:

```
image = [[1,1,1],[1,1,0],[1,0,1]]  
sr = 1, sc = 1, newColor = 2
```

**Output:** [[2,2,2],[2,2,0],[2,0,1]]

#### Explanation:

From the center of the image (with position `(sr, sc) = (1, 1)`), all pixels connected by a path of the same color as the starting pixel are colored with the new color.

Note the bottom corner is not colored 2, because it is not 4-directionally connected to the starting pixel.

### Note:

- The length of `image` and `image[0]` will be in the range [1, 50].
- The given starting pixel will satisfy  $0 \leq sr < \text{image.length}$  and  $0 \leq sc < \text{image[0].length}$ .
- The value of each color in `image[i][j]` and `newColor` will be an integer in [0, 65535].

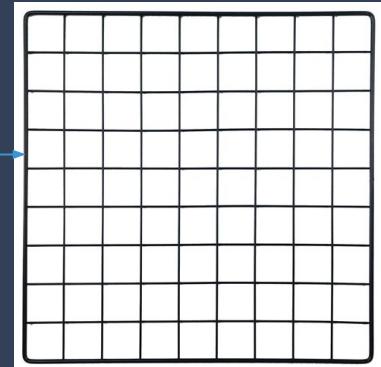
# Implicit representations of edges

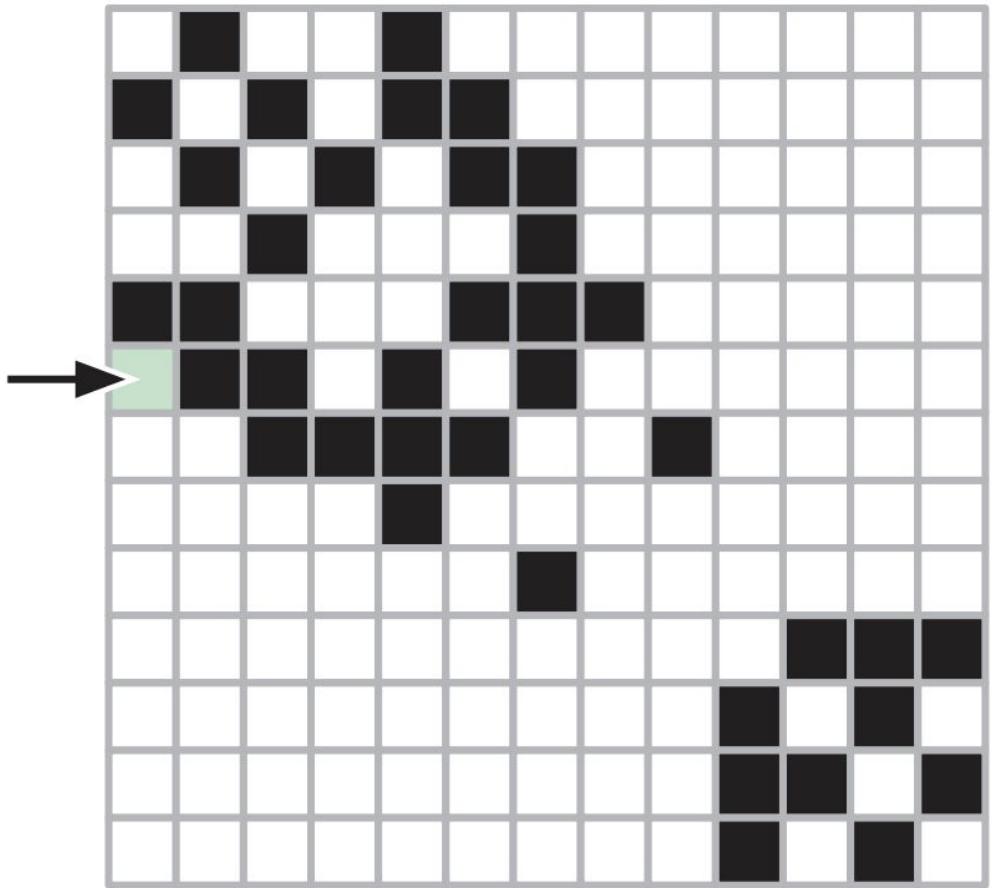
In some cases, we don't need to use adjacency lists/maps to store the neighbors of a vertex. The neighbors can be dynamically calculated (by a *getNeighbors* function) when the need arises. This typically happens when the given graph is structured like a *grid*, with rows and columns of vertices.

This approach saves us space without sacrificing time.

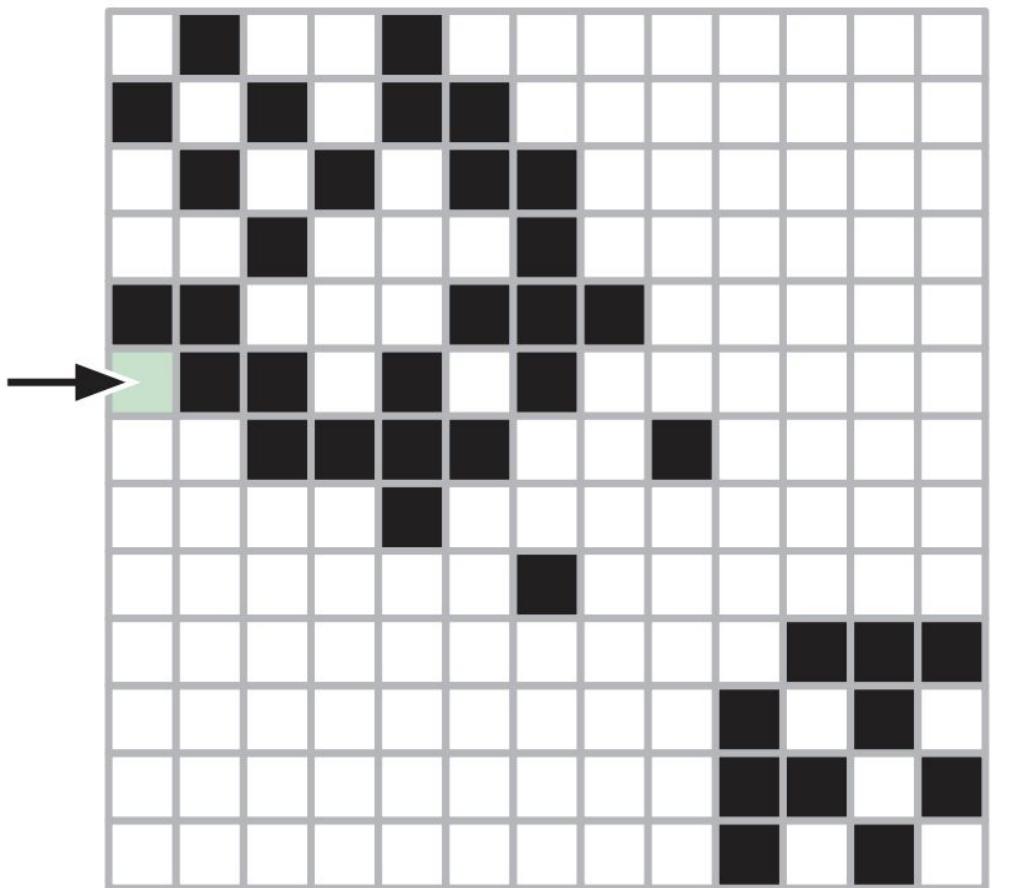
E.g, City Map

Every vertex has 4 neighbors unless it is close to the grid boundary.





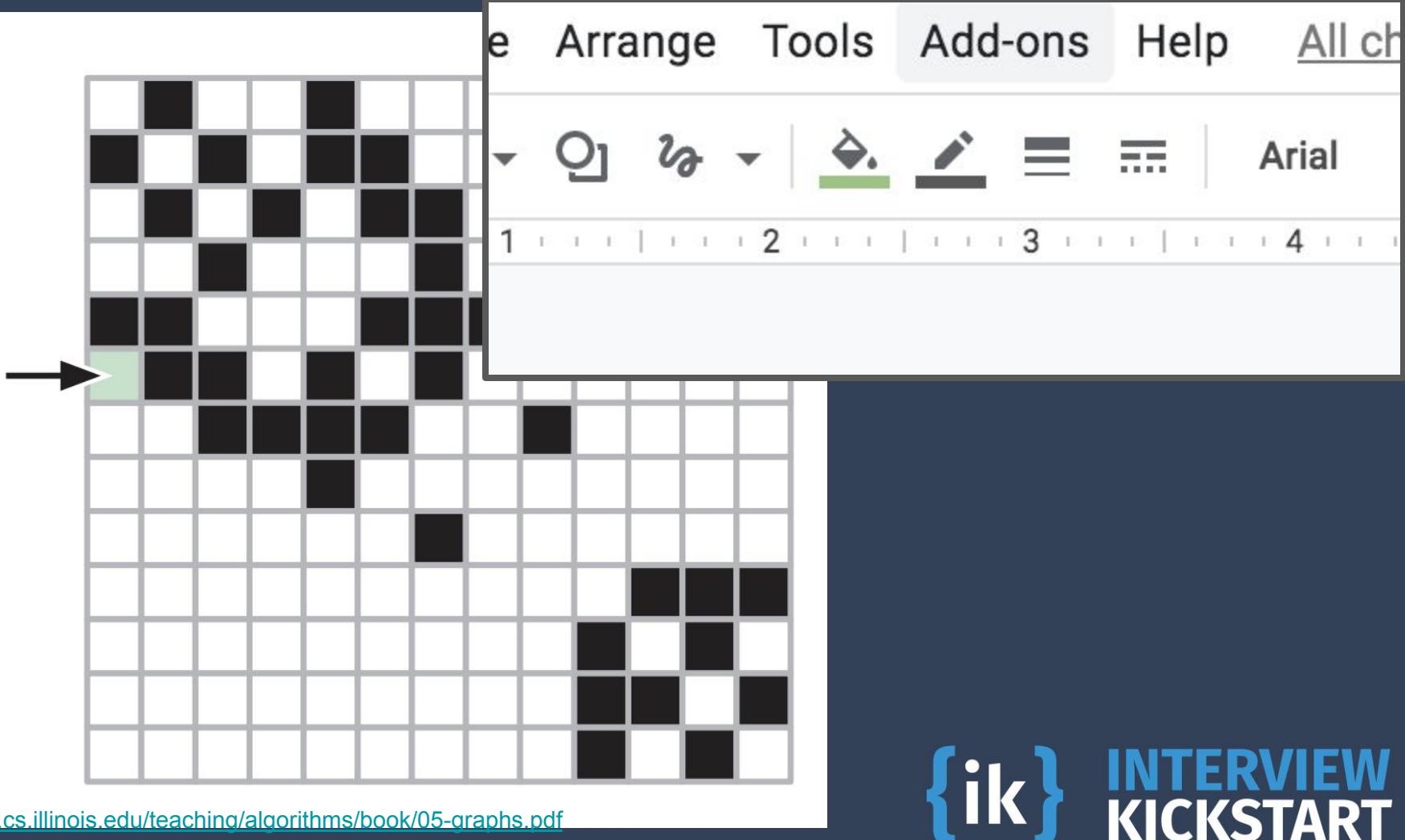
# The Flood Fill problem

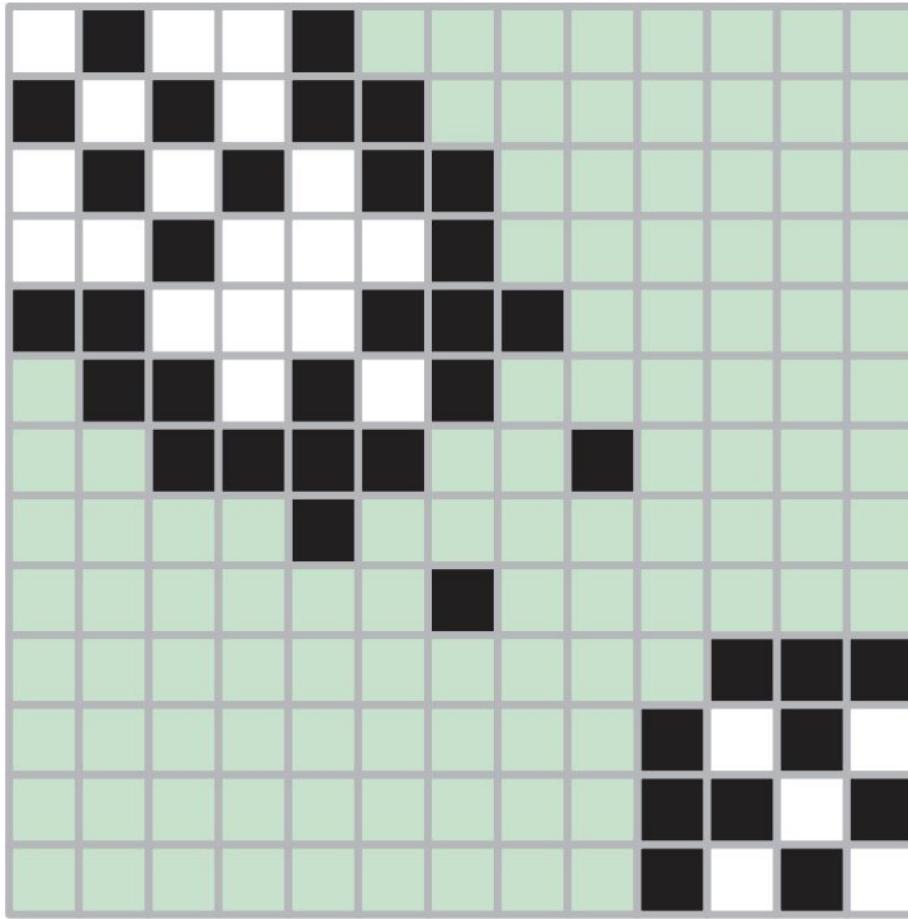


Given:

- 1) A 2D array of integer values, where the values represent colors,
- 2) the coordinates of the starting pixel, and
- 3) the value of the new color:

**Implement the flood fill operation.**





<http://jeffe.cs.illinois.edu/teaching/algorithms/book/05-graphs.pdf>

# General Graph Traversal algorithm

void GeneralSearch(Vertex s):

Initialization: All vertices are undiscovered

    Initialize an empty bag and put s in it

    Mark s as discovered

    while the bag is not empty:

**Pull out one of the vertices, say u**

        For each neighbor of u:

            if the neighbor is undiscovered:

                put the neighbor in the bag

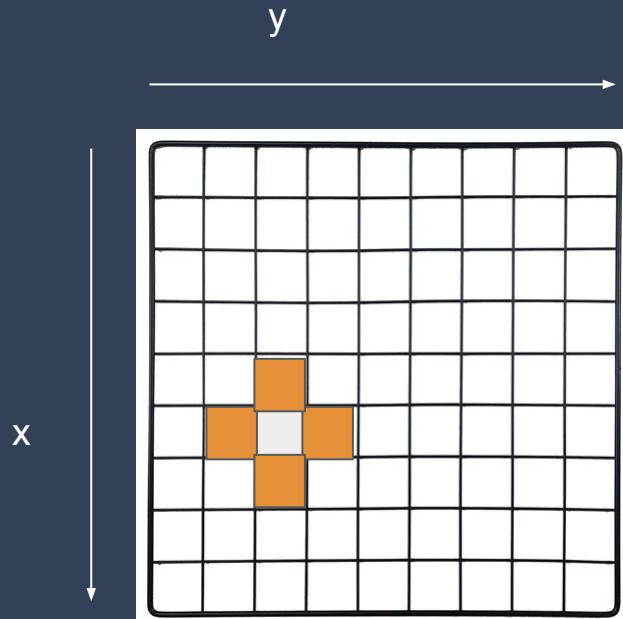
                mark it as discovered

# General Graph Traversal algorithm

```
void BFS(Vertex s):
    q = new Queue([s])
    grid[s] = newColor
    while q is not empty:
        u = q.pop()
        for neighbor in getNeighbors(u):
            if grid[neighbor] == oldColor:
                q.push(neighbor)
                grid[neighbor] = newColor
```

bfs(sx,sy)  
return grid

```
def getNeighbors(x,y):  
    result = empty array  
    if x+1 <= length(grid)-1:  
        result.append((x+1,y))  
    if y+1 <= length(grid[0])-1:  
        result.append((x,y+1))  
    if x-1 >= 0:  
        result.append((x-1,y))  
    if y-1 >= 0:  
        result.append((x,y-1))  
    return result
```



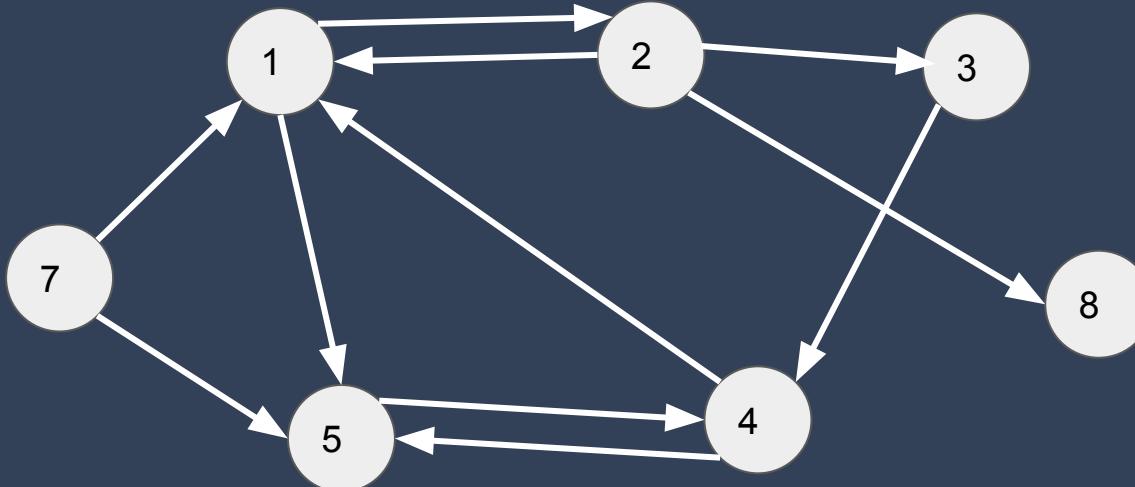
Edge case:

```
if oldColor == newColor:  
    return grid
```

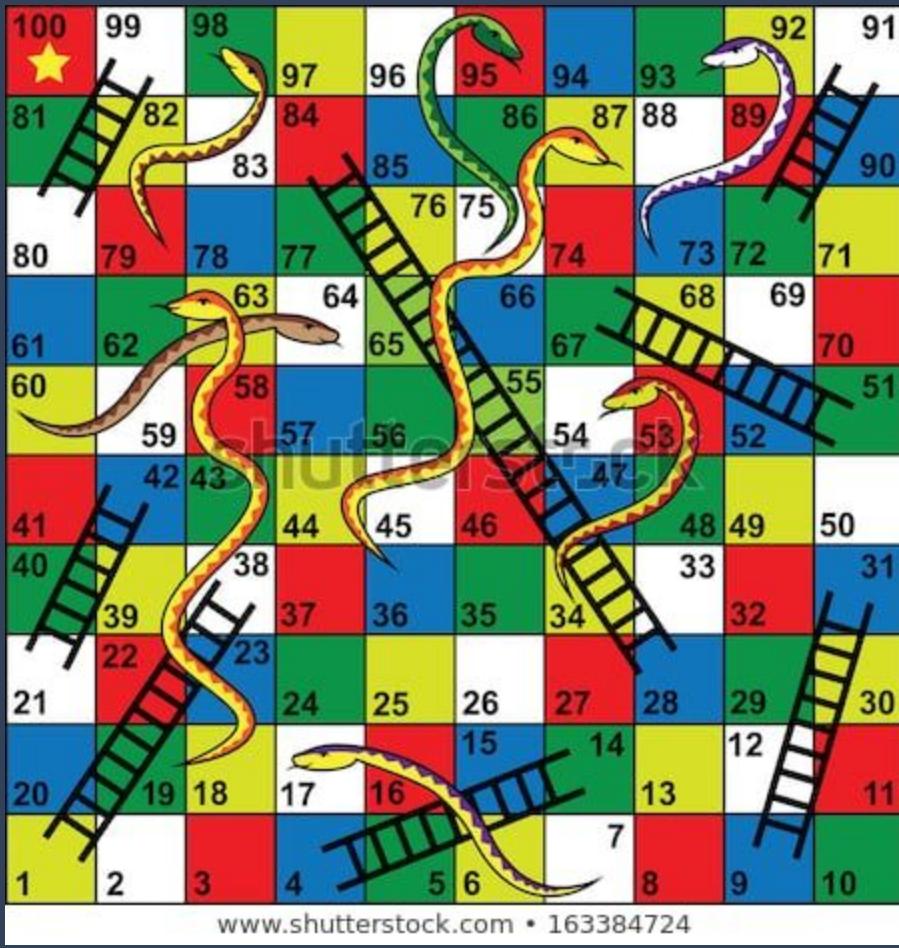
What if the Queue was replaced by a Stack?

# From undirected to directed graphs...

# BFS (Directed graph example)



Is it straightforward to detect if a directed graph has a cycle by looking at the BFS tree?  
When would you do BFS on a directed graph?



Given a snake and ladder game, find the minimum number of throws required to win the game.

## 909. Snakes and Ladders

Medium   137   404   Favorite   Share

On an  $N \times N$  board, the numbers from 1 to  $N^2$  are written *boustrophedonically* starting from the bottom left of the board, and alternating direction each row. For example, for a  $6 \times 6$  board, the numbers are written as follows:



You start on square 1 of the board (which is always in the last row and first column). Each move, starting from square  $x$ , consists of the following:

- You choose a destination square  $s$  with number  $x+1, x+2, x+3, x+4, x+5$ , or  $x+6$ , provided this number is  $\leq N^2$ .
  - (This choice simulates the result of a standard 6-sided die roll: ie, there are always **at most 6 destinations, regardless of the size of the board.**)
- If  $s$  has a snake or ladder, you move to the destination of that snake or ladder. Otherwise, you move to  $s$ .

A board square on row  $r$  and column  $c$  has a "snake or ladder" if `board[r][c] != -1`. The destination of that snake or ladder is `board[r][c]`.

Note that you only take a snake or ladder at most once per move: if the destination to a snake or ladder is the start of another snake or ladder, you do **not** continue moving. (For example, if the board is `[[4,-1],[-1,3]]`, and on the first move your destination square is '2', then you finish your first move at '3', because you do **not** continue moving to '4'.)

Return the least number of moves required to reach square  $N^2$ . If it is not possible, return `-1`.

```

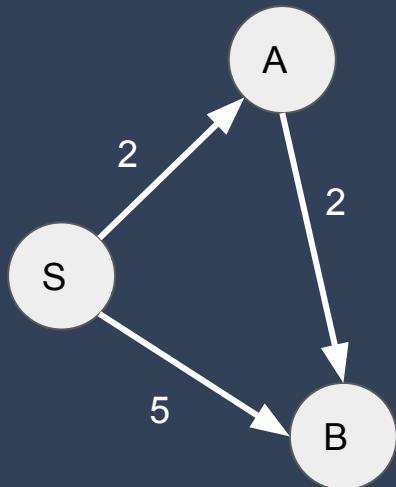
1 class Solution(object):
2     def snakesAndLadders(self, board):
3         """
4             :type board: List[List[int]]
5             :rtype: int
6         """
7         size = len(board)
8         maxsquare = size*size
9
10    def numtorowcol(n):
11        row = (maxsquare-n)/size
12        c = (n-1) % (2*size) #c is in the range of 0 to 2*size - 1
13        if c < size:
14            col = c
15        else:
16            col = 2*size-1 - c
17        return (row,col)
18
19    def minmoves(n):
20        q = collections.deque([n])
21        visited = {}
22        visited[n] = 0 #records shortest path distance from source to this vertex
23        while len(q) != 0:
24            curr = q.popleft()
25            for i in range(1,7):
26                nxt = curr + i
27                if nxt > maxsquare:
28                    continue
29                (r,c) = numtorowcol(nxt)
30                if board[r][c] != -1:
31                    nxt = board[r][c]
32                if nxt not in visited:
33                    q.append(nxt)
34                    visited[nxt] = visited[curr] + 1
35                if nxt == maxsquare:
36                    return visited[nxt]
37
38        return -1
39
40    return minmoves(1)

```

**How do we find the shortest path from source  $s$  to some destination when there are lengths (weights) on edges?**

**BFS is a special case of this problem where all weights are 1.**

Analogy: Think of how a disease spreads in a population.



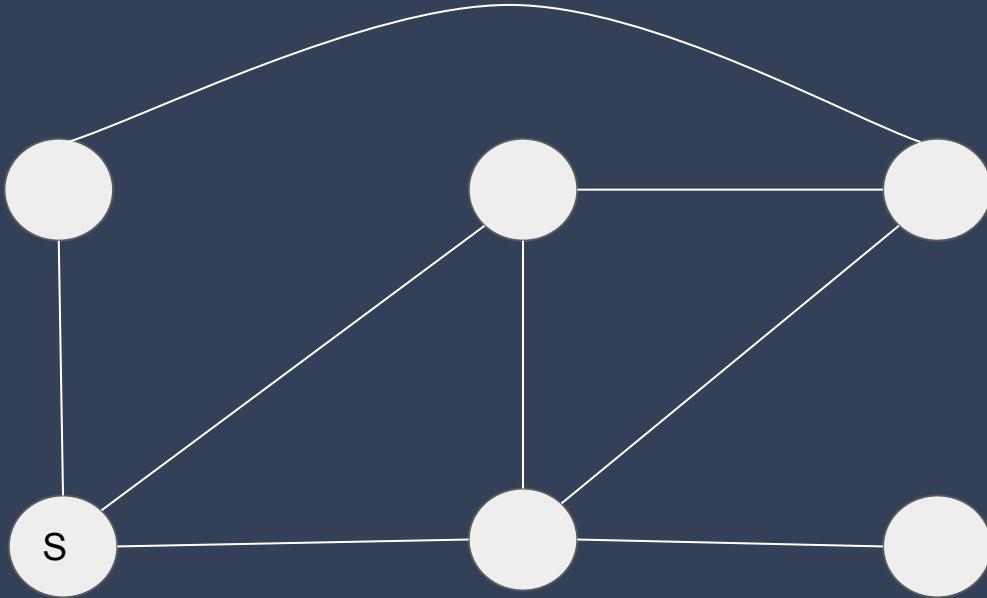
- What if the graphs are **weighted**?
  - All nonnegative weights: Dijkstra
  - If there are negative weights: Bellman-Ford

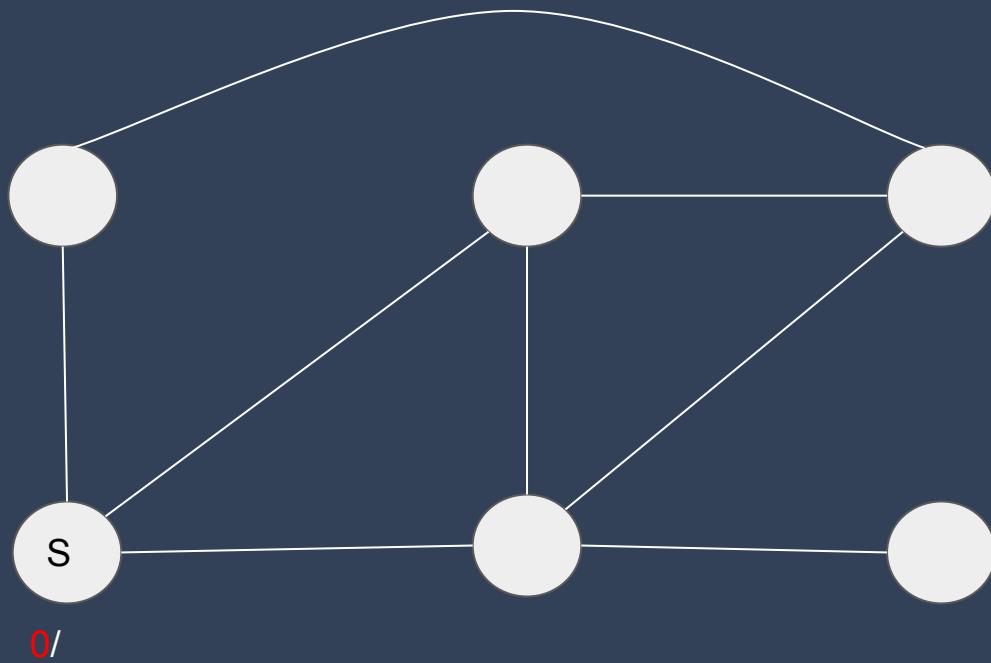
**DFS with a little  
bookkeeping...**

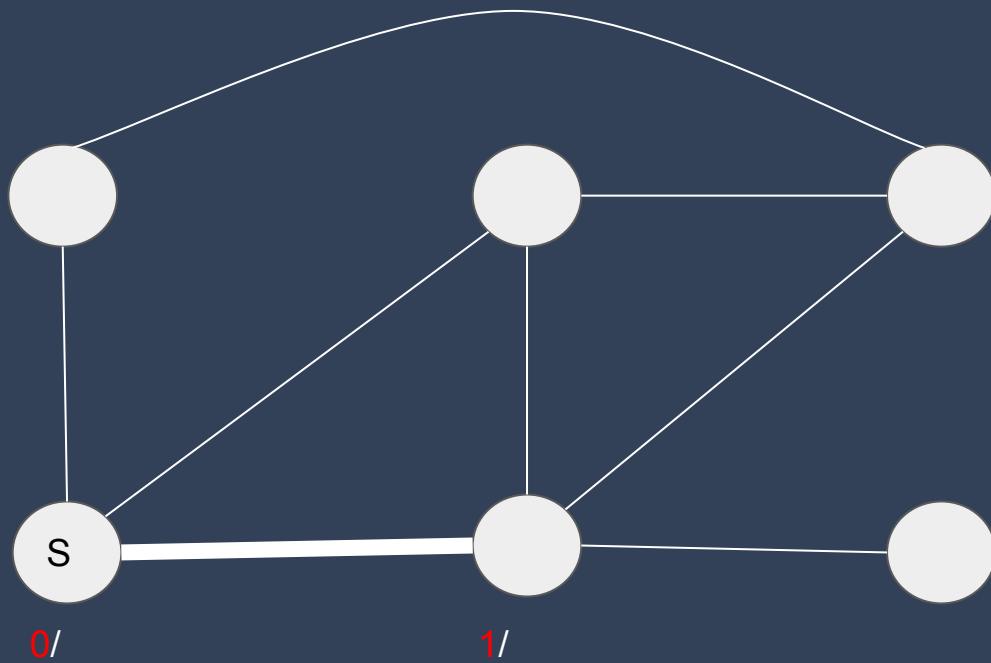
```
class Graph {  
    ....  
    //adjList is an array of lists
```

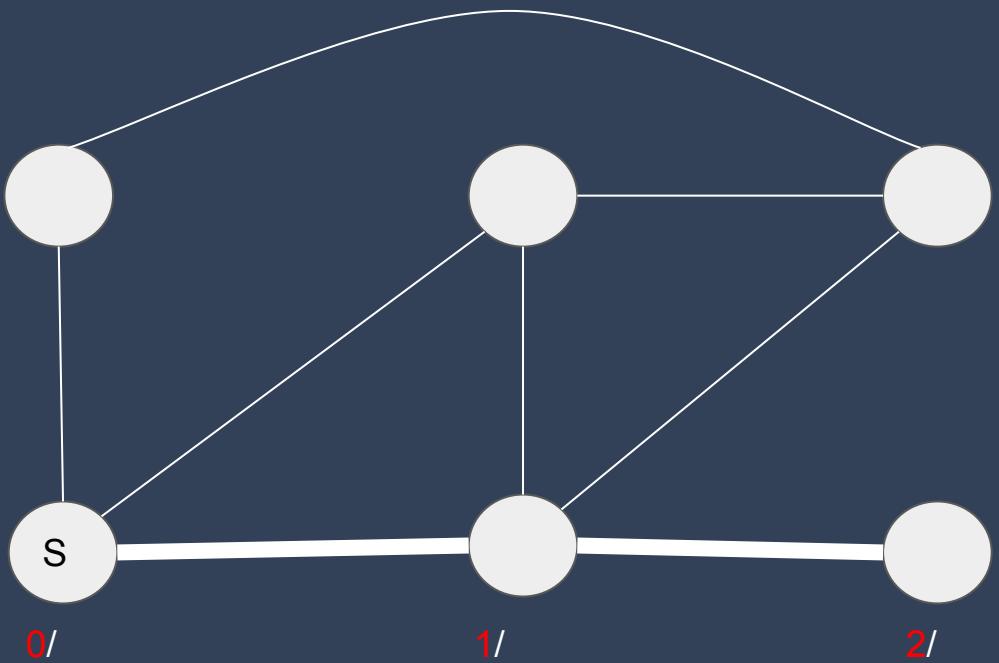
```
    void DFS(int source) {  
        visited[source] = 1  
        for neighbor in adjList[source]:  
            if visited[neighbor] == -1:  
                DFS(neighbor)  
    }  
}
```

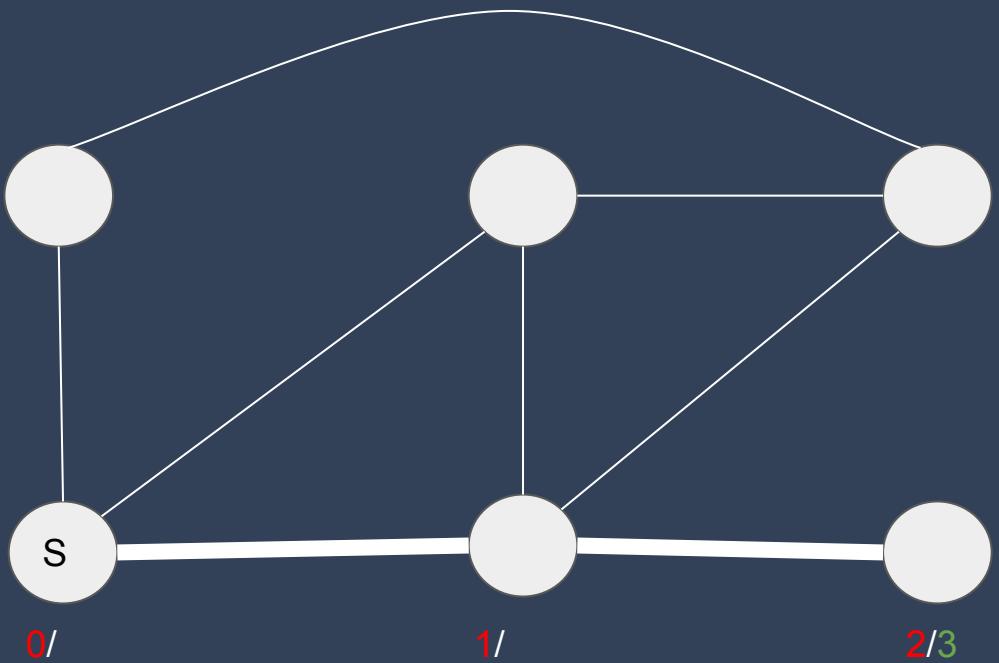
```
class Graph {  
    time = 0  
    //arr, dep are two arrays recording arrival and departure times of each vertex; initialized to -1  
  
    void DFS(int source) {  
        visited[source] = 1  
        arr[source] = time++  
        for neighbor in adjList[source]:  
            if visited[neighbor] == -1:  
                DFS(neighbor)  
                dep[source] = time++  
    }  
}
```

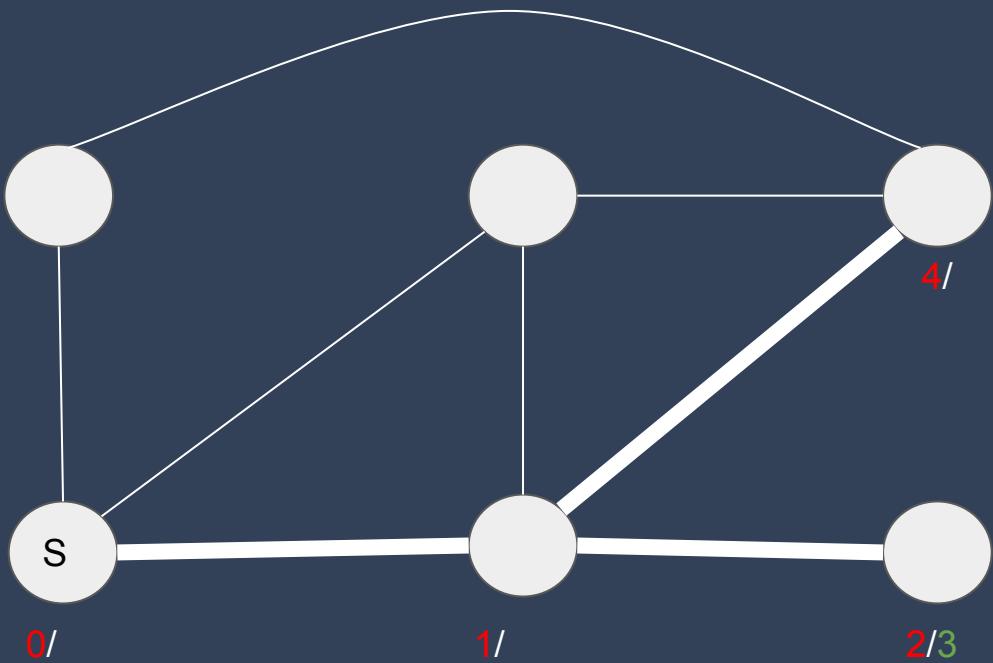


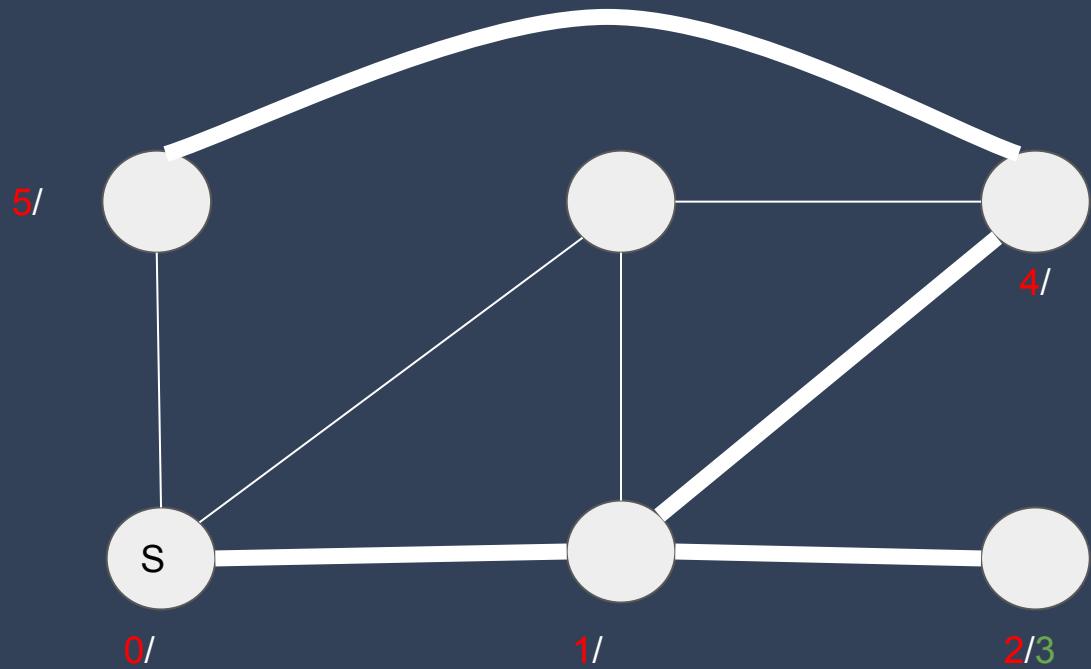


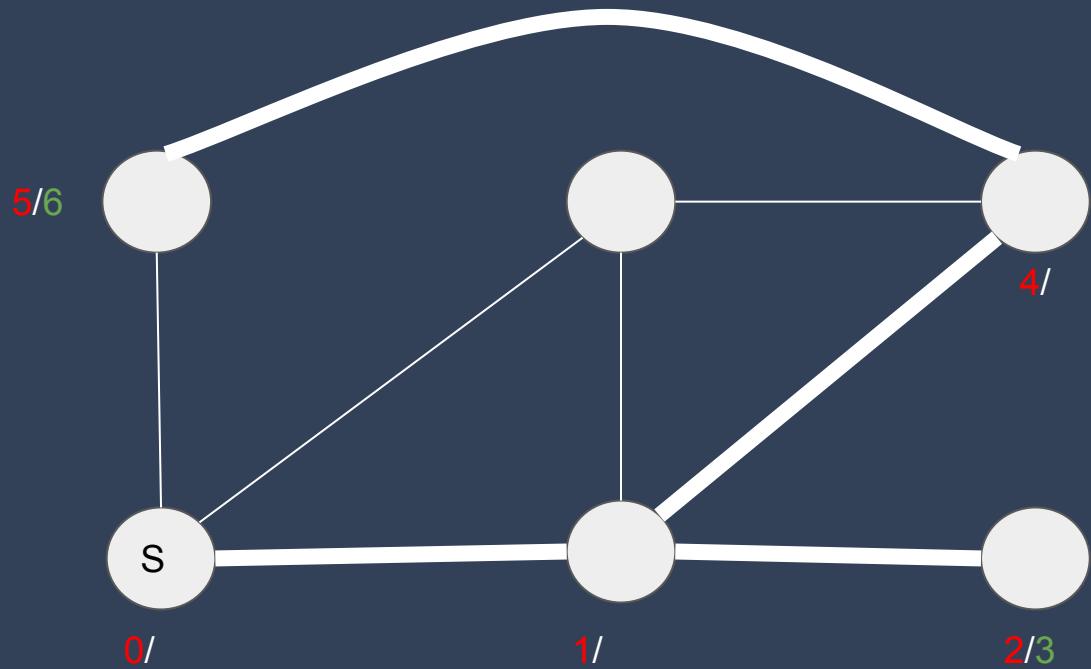


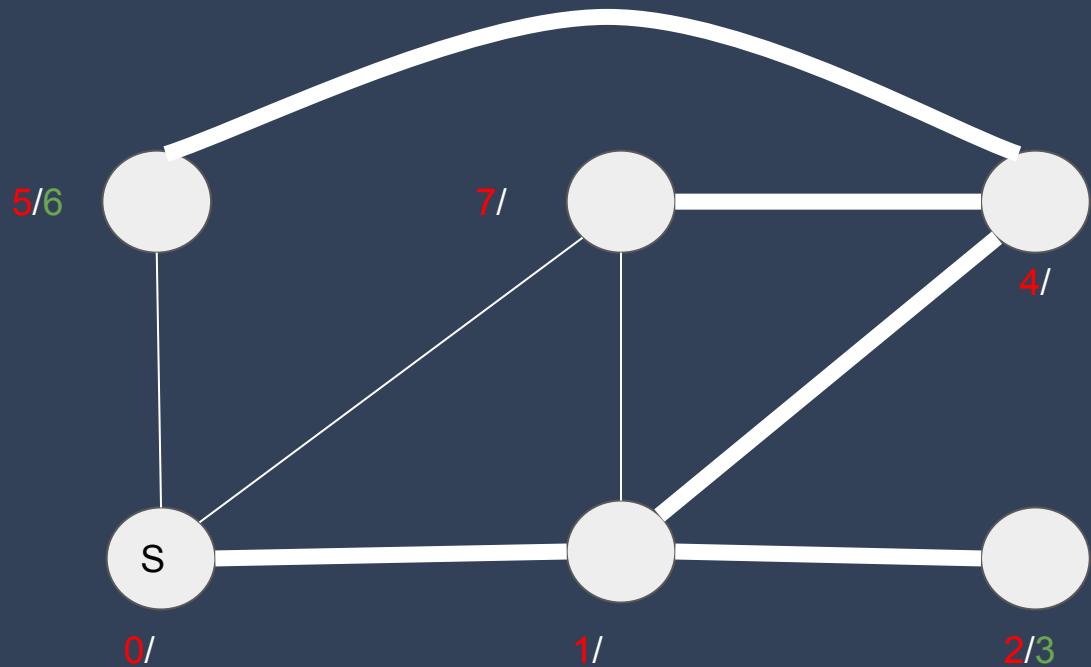


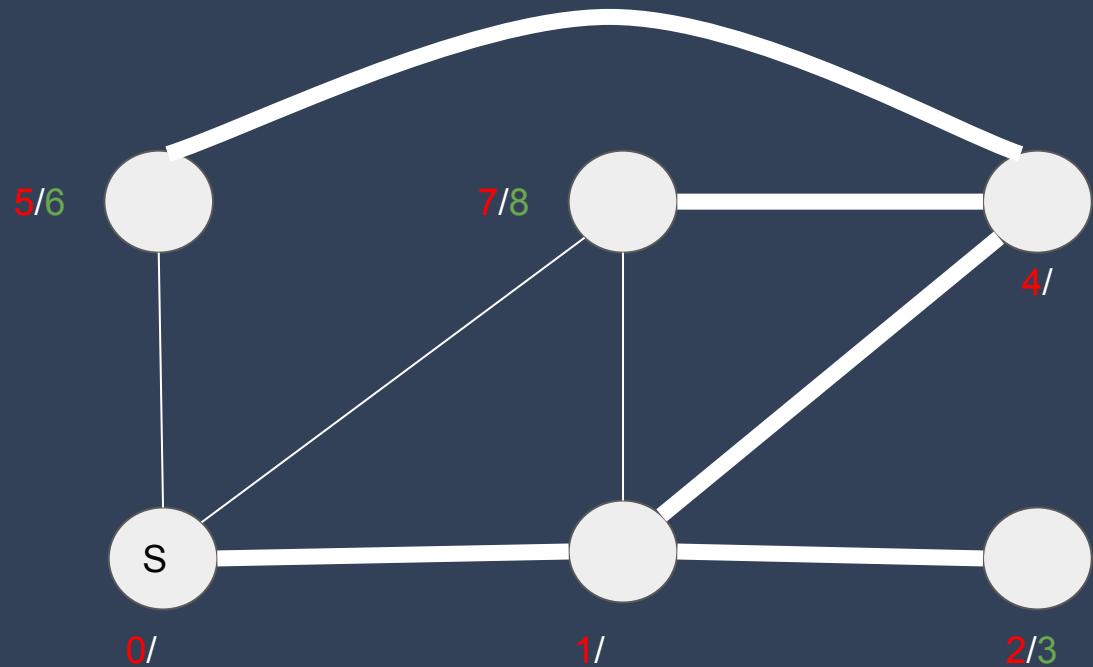


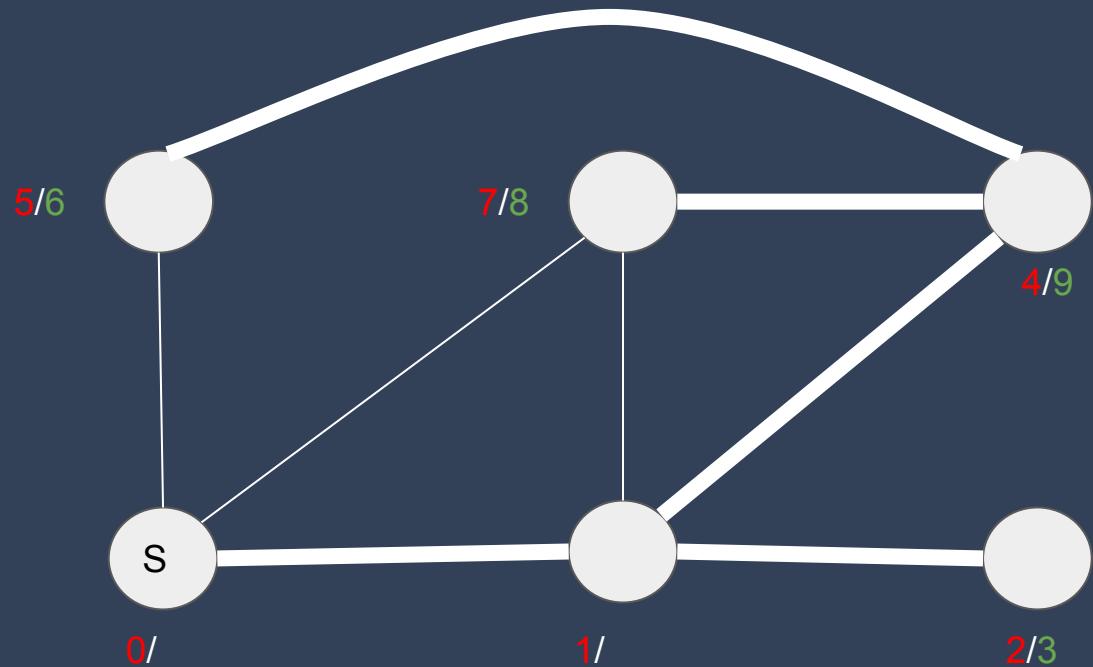


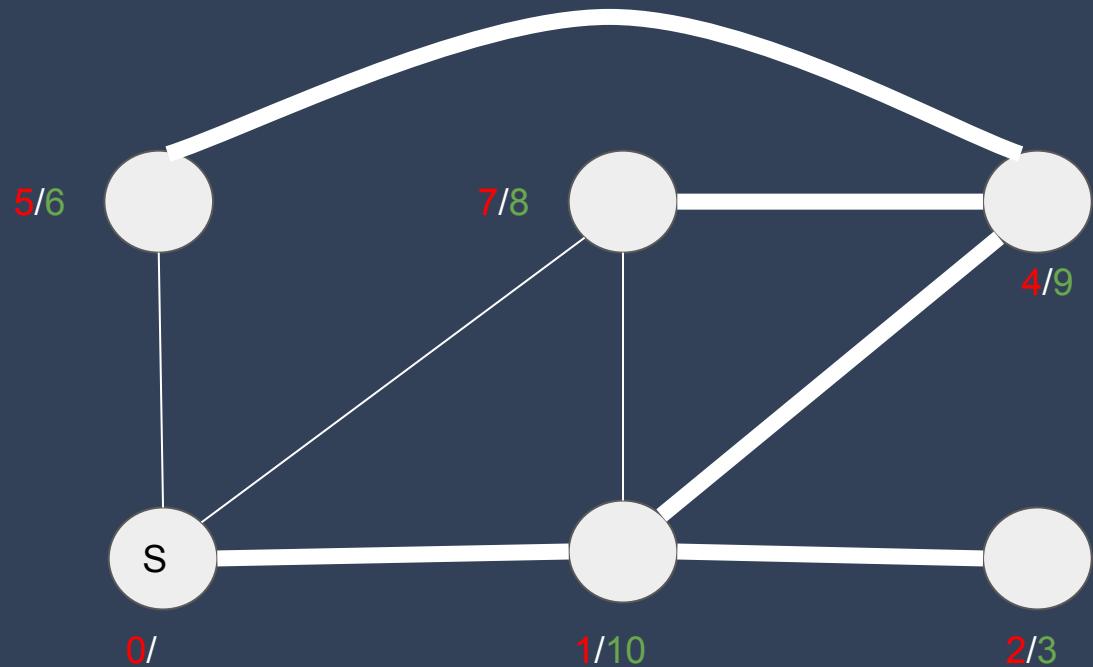


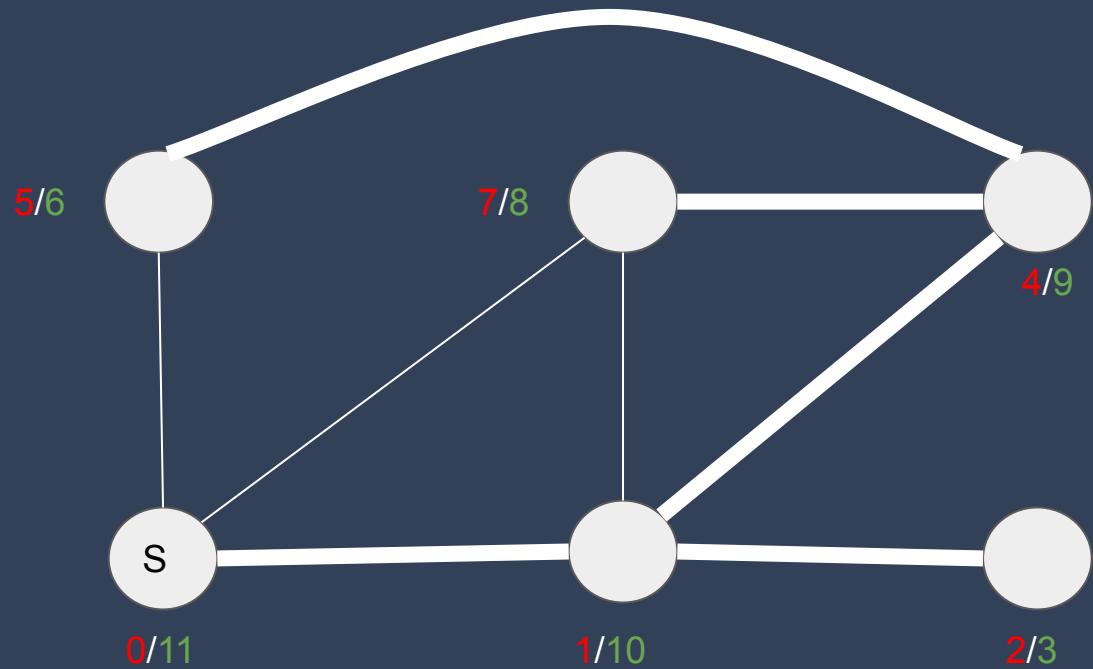


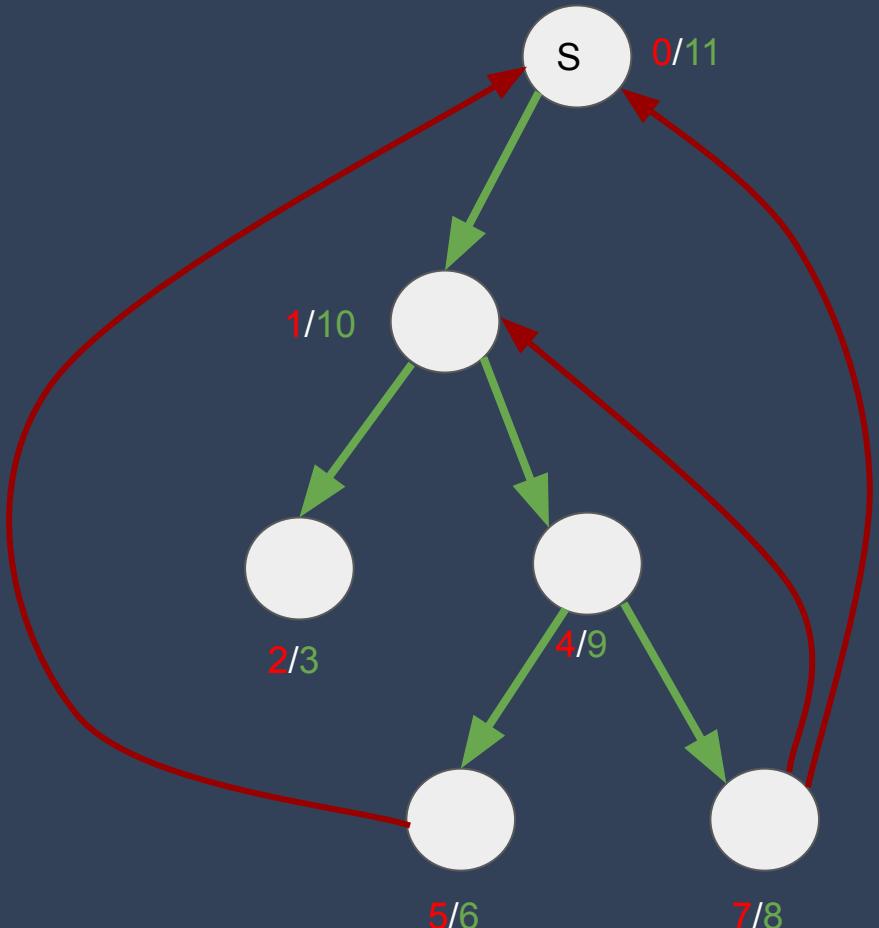


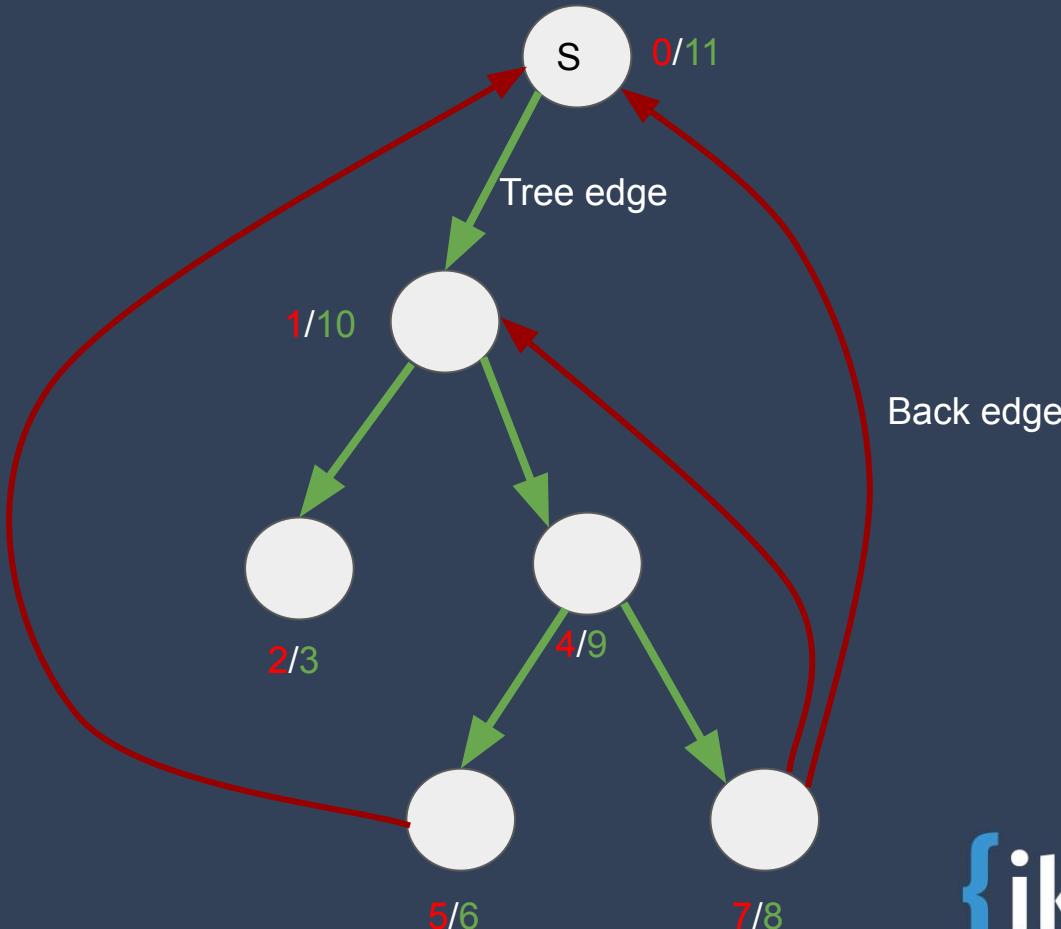












**Notice the relationship  
between arrival and  
departure times of ancestors  
and descendants?**

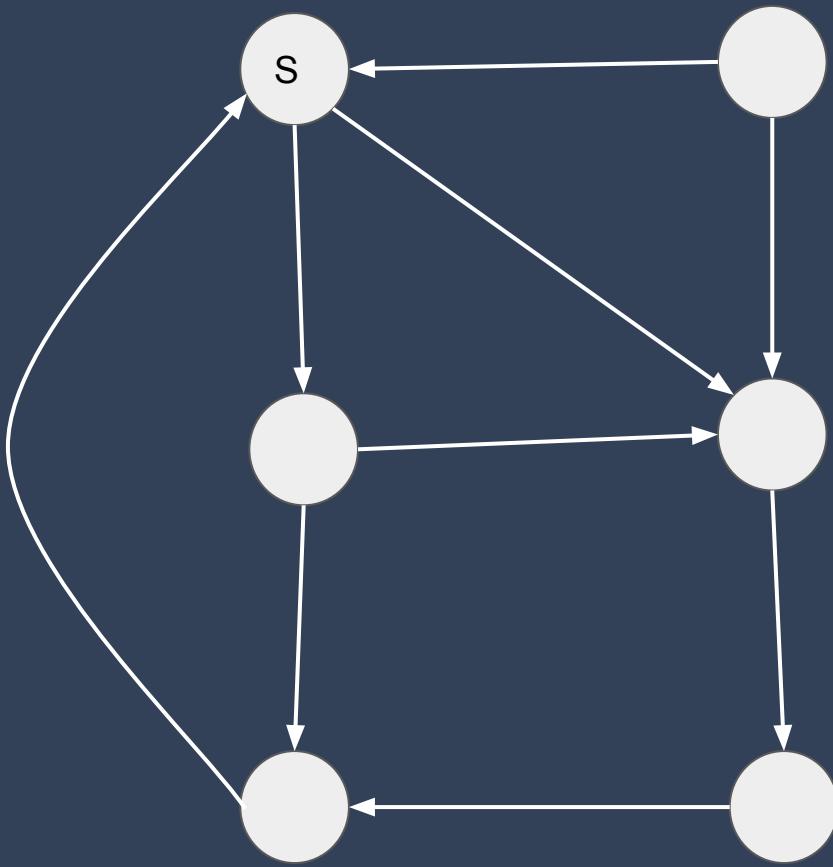
We have already seen how to find cycles in an undirected graph (using BFS/DFS). How do we find cycles in a directed graph?

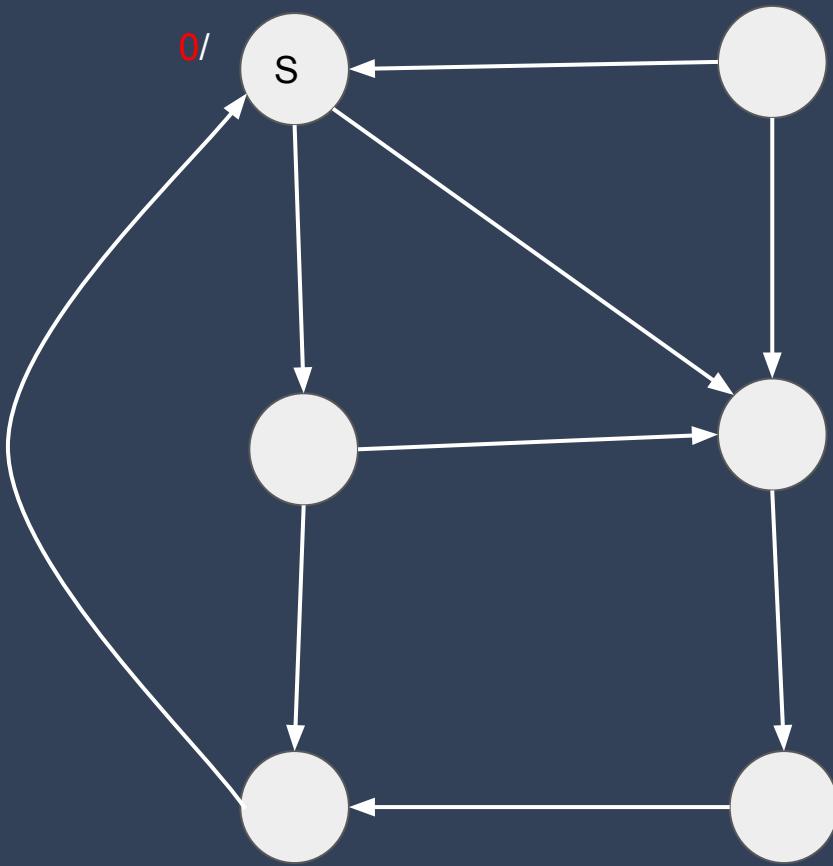
Don't use BFS! Why?

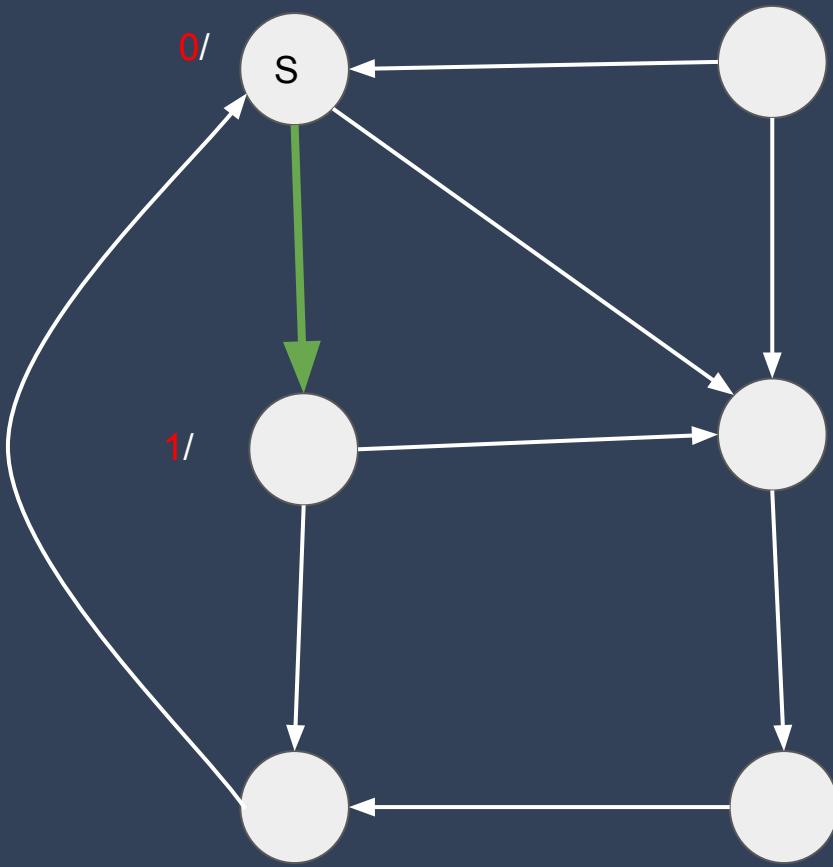
{ik}

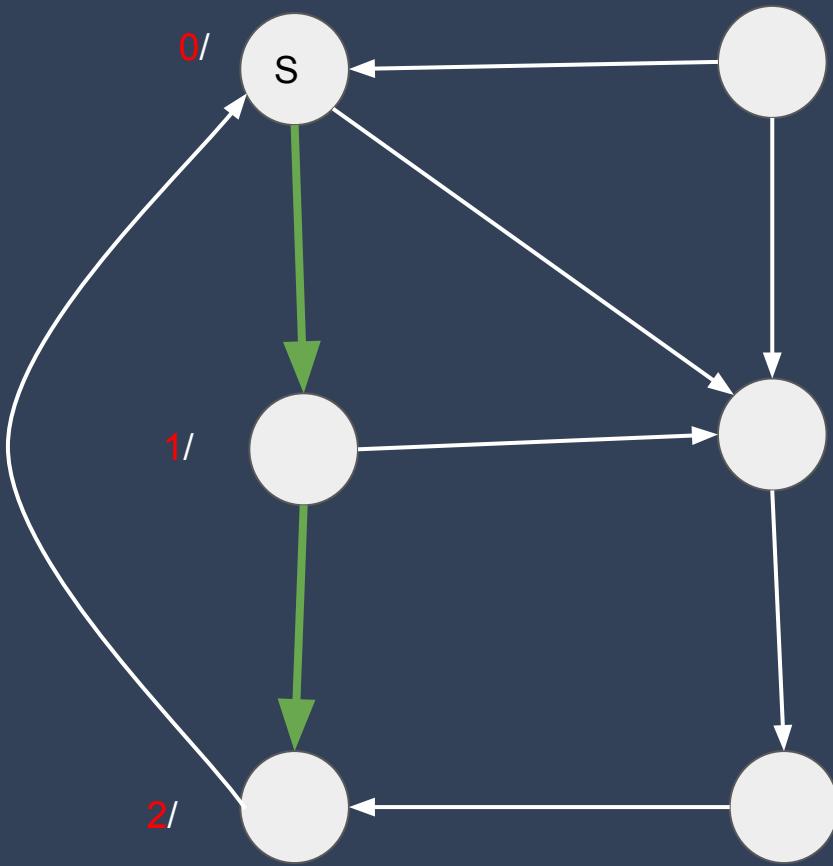
INTERVIEW  
KICKSTART

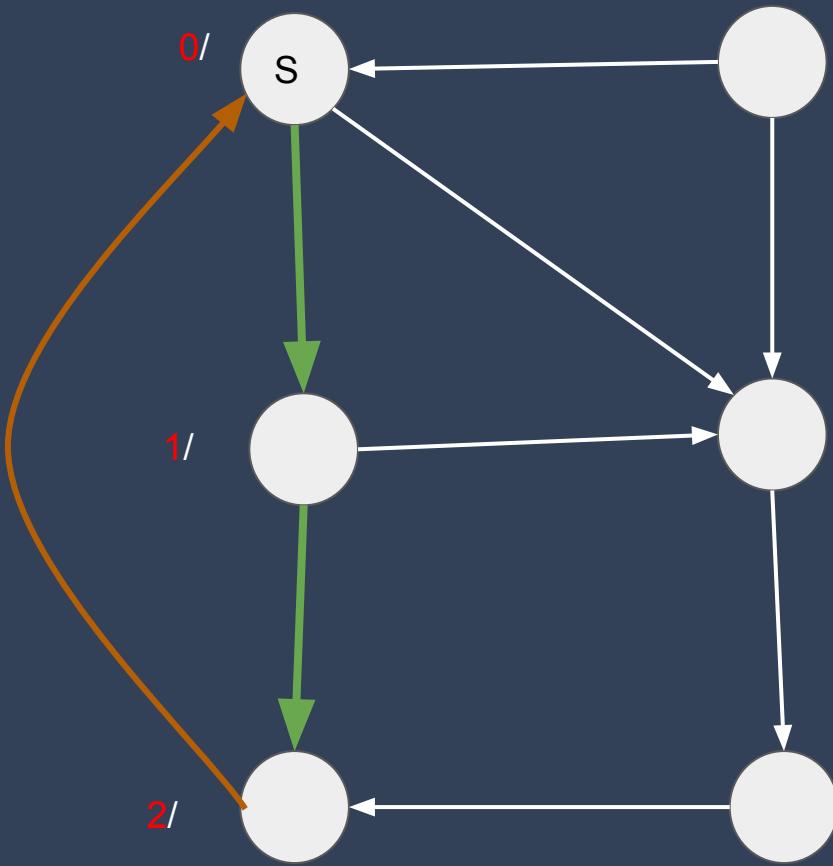
# DFS on Directed graphs

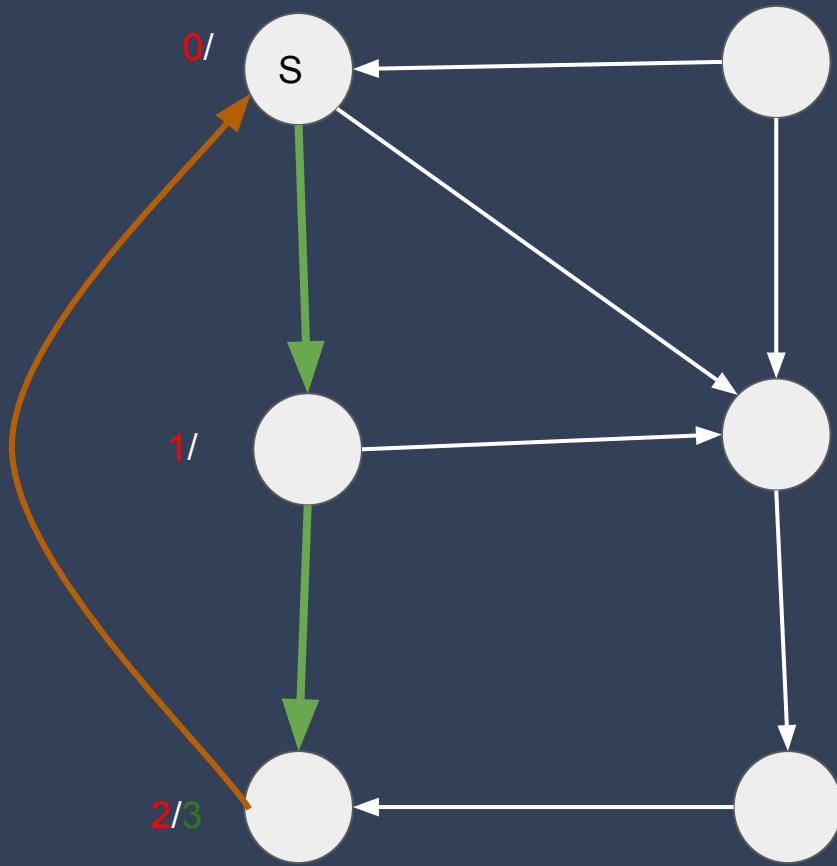


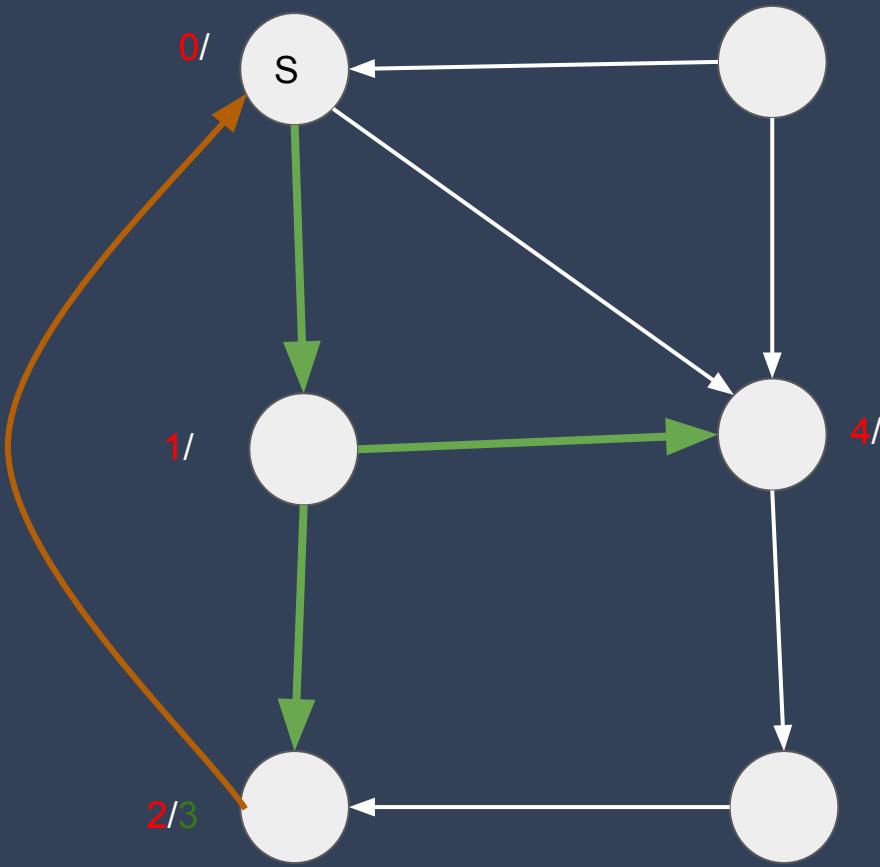


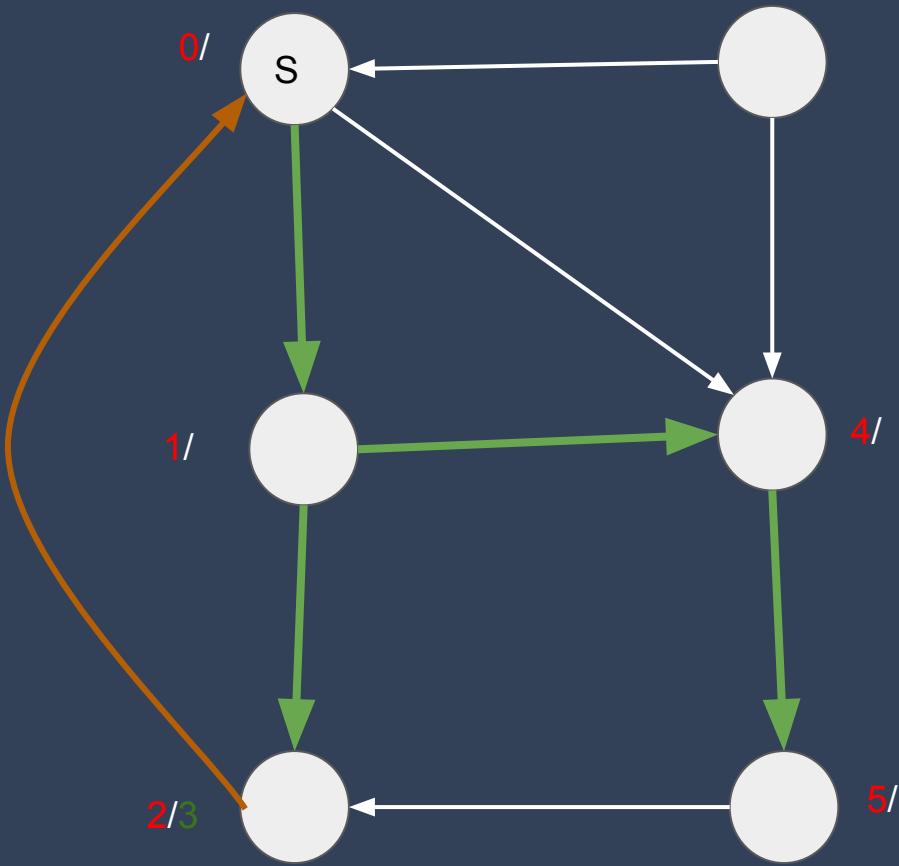


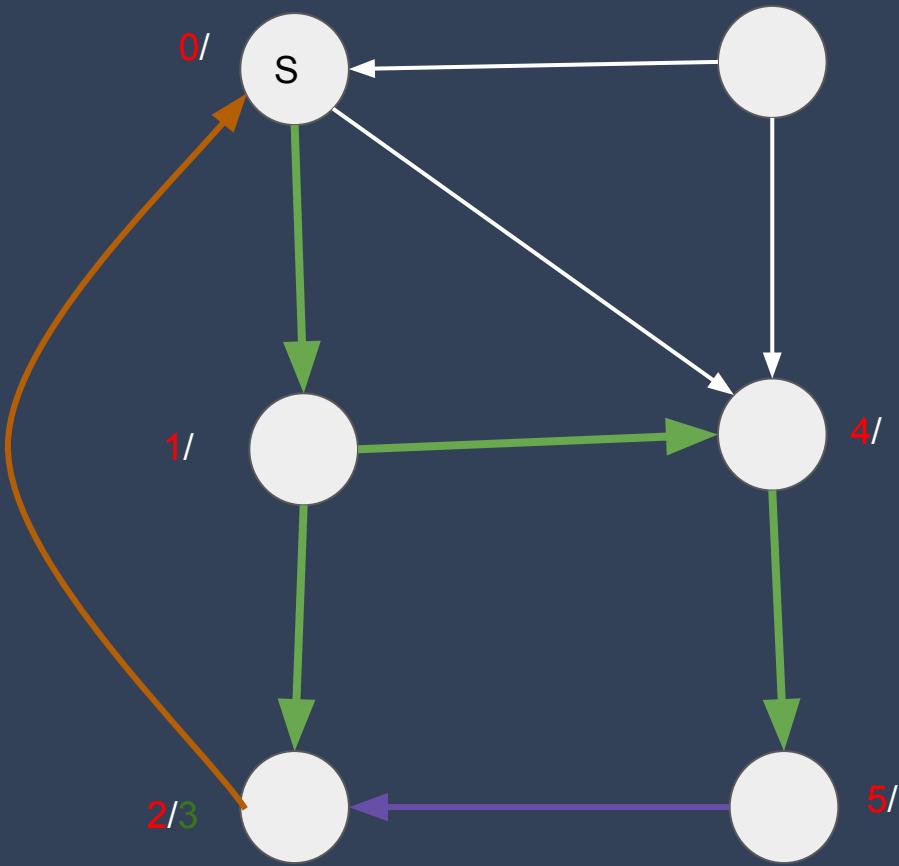


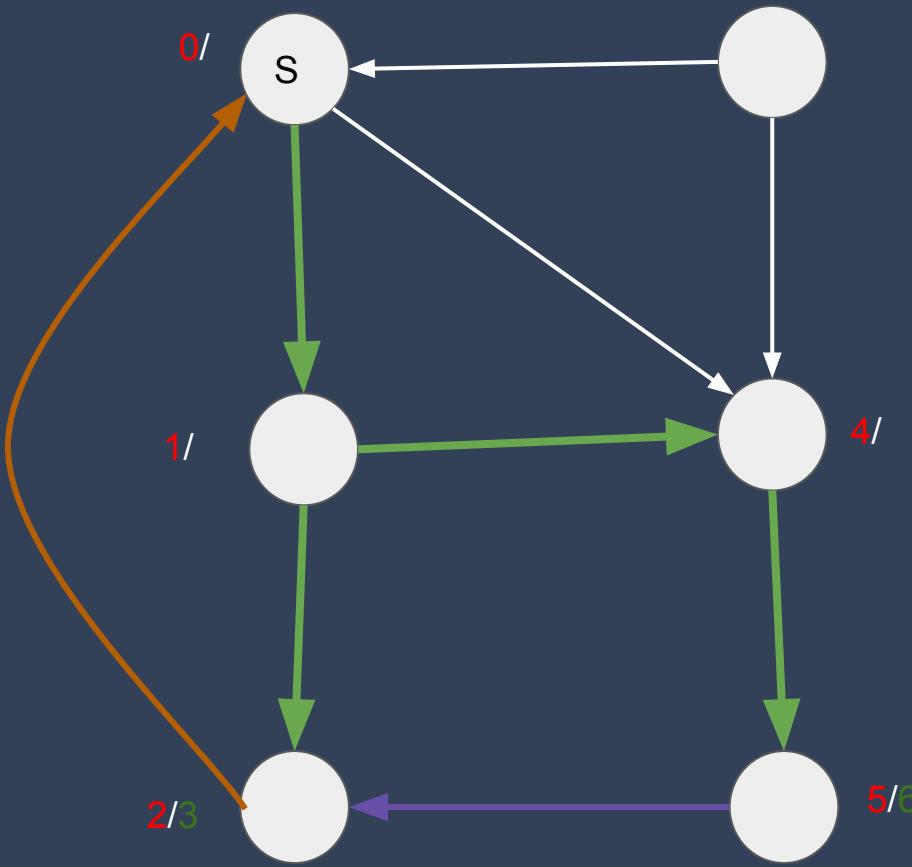


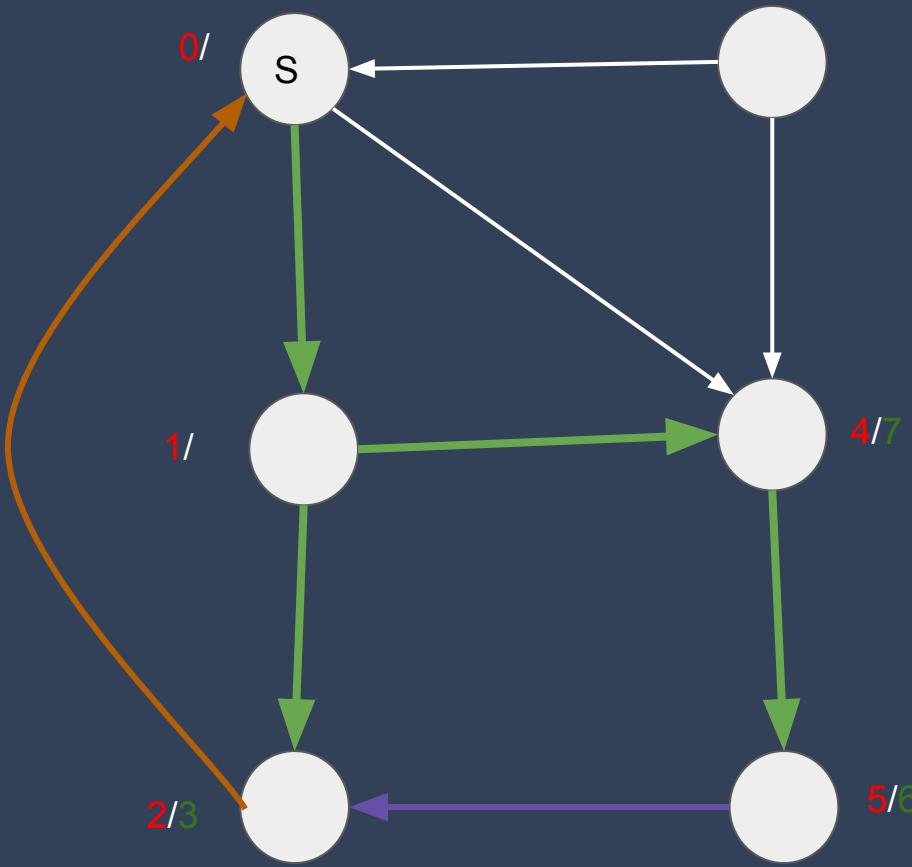


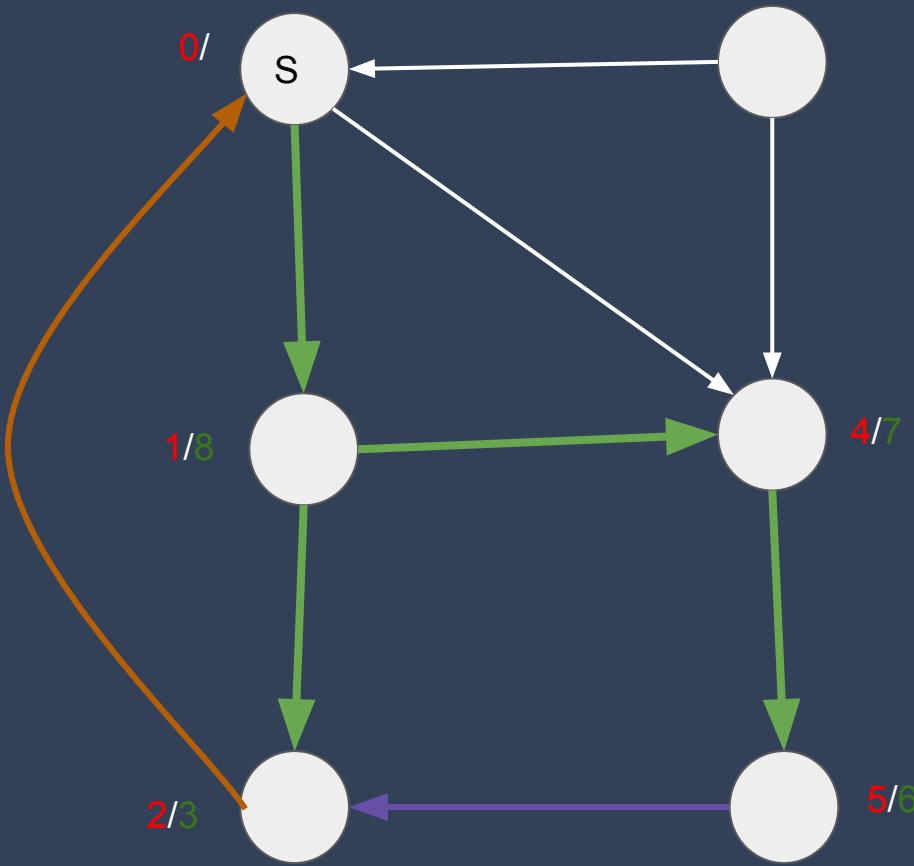


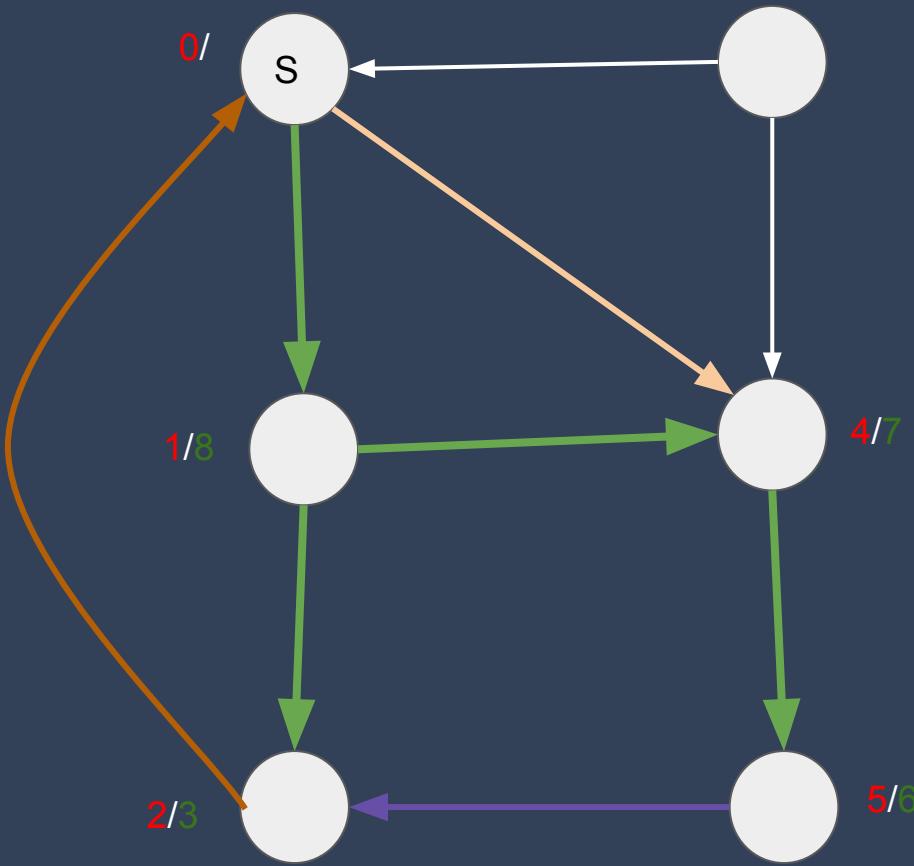


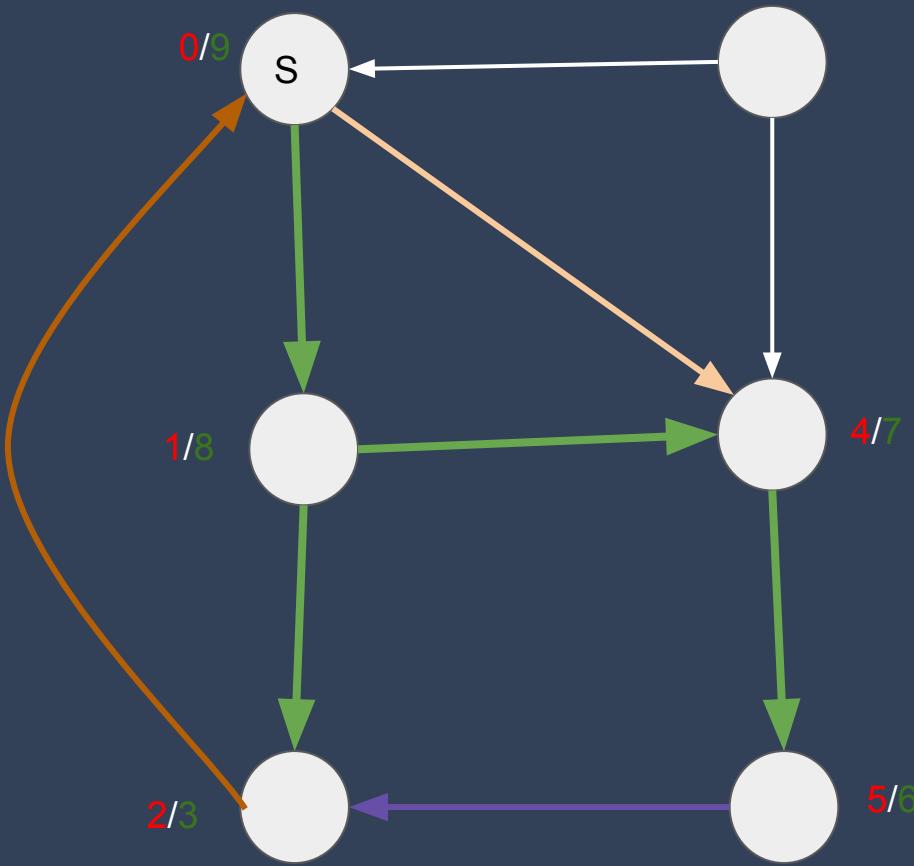


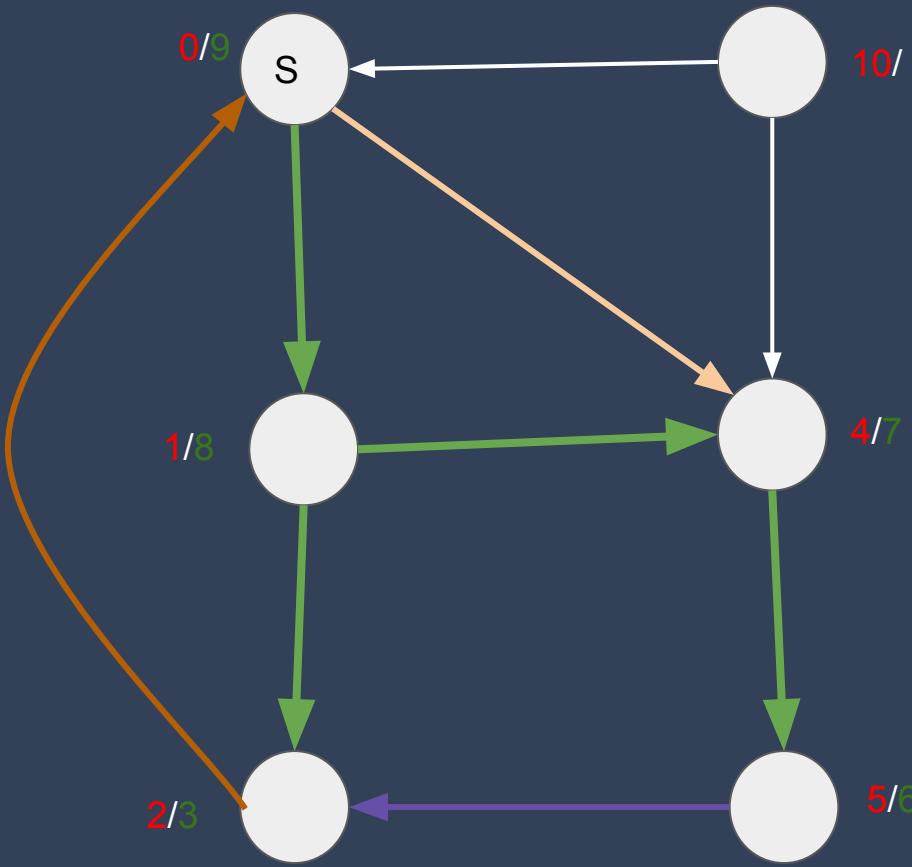


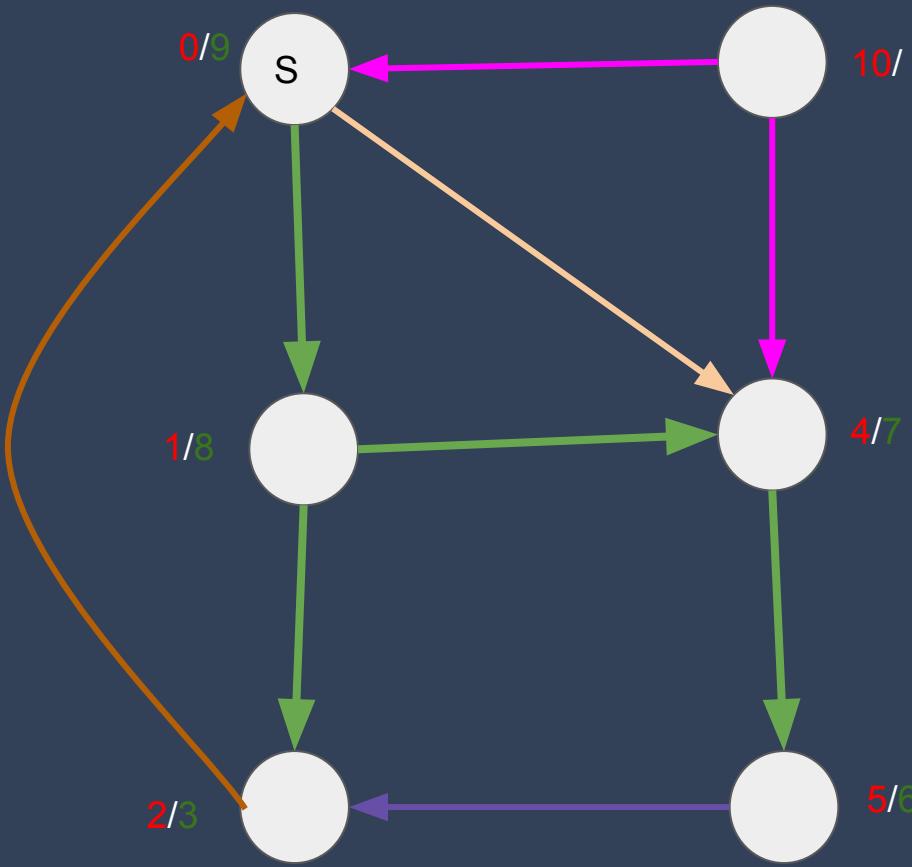


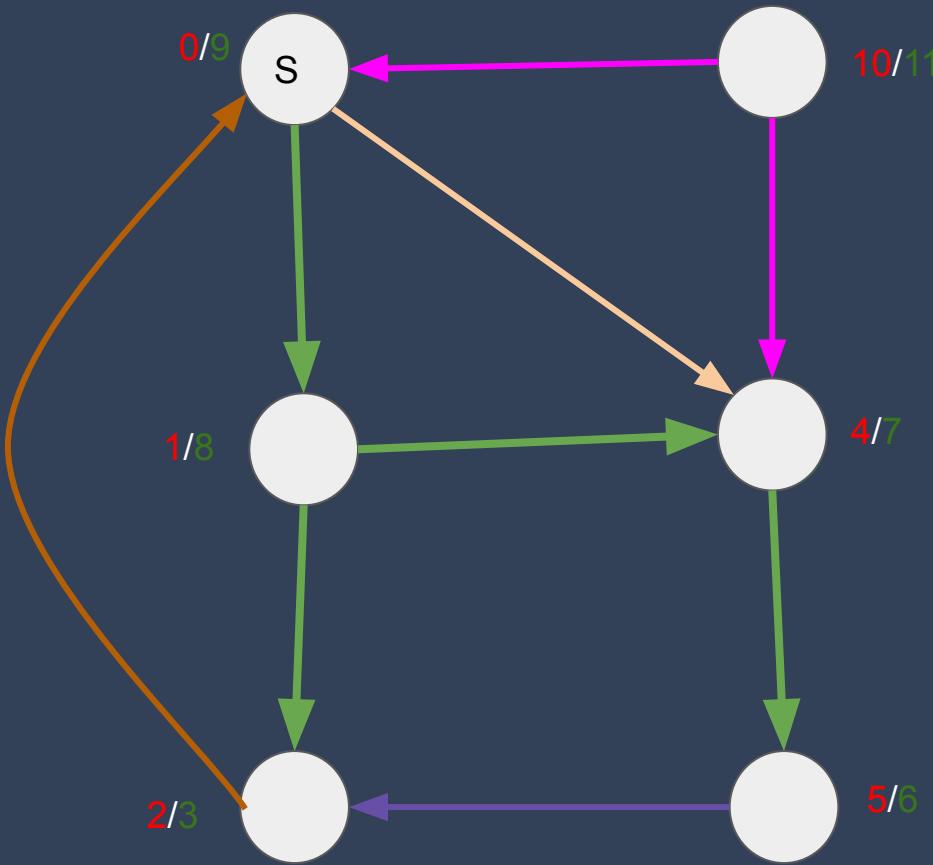


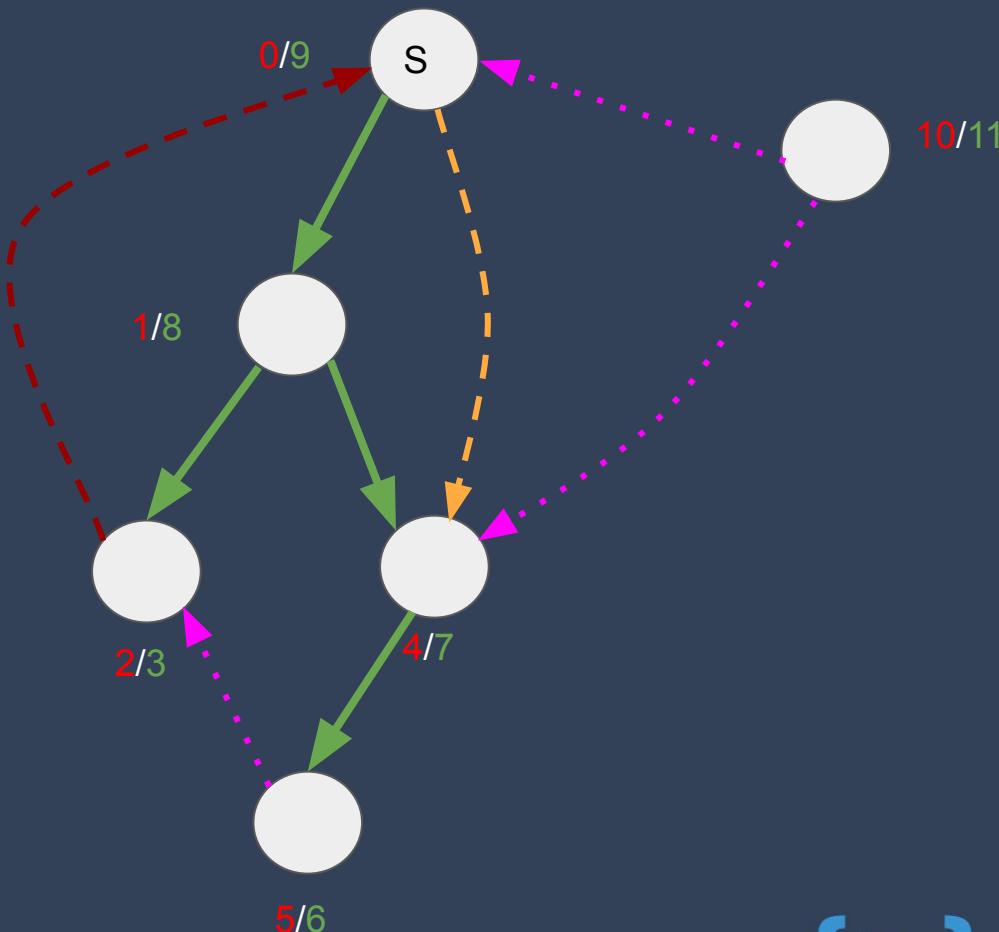


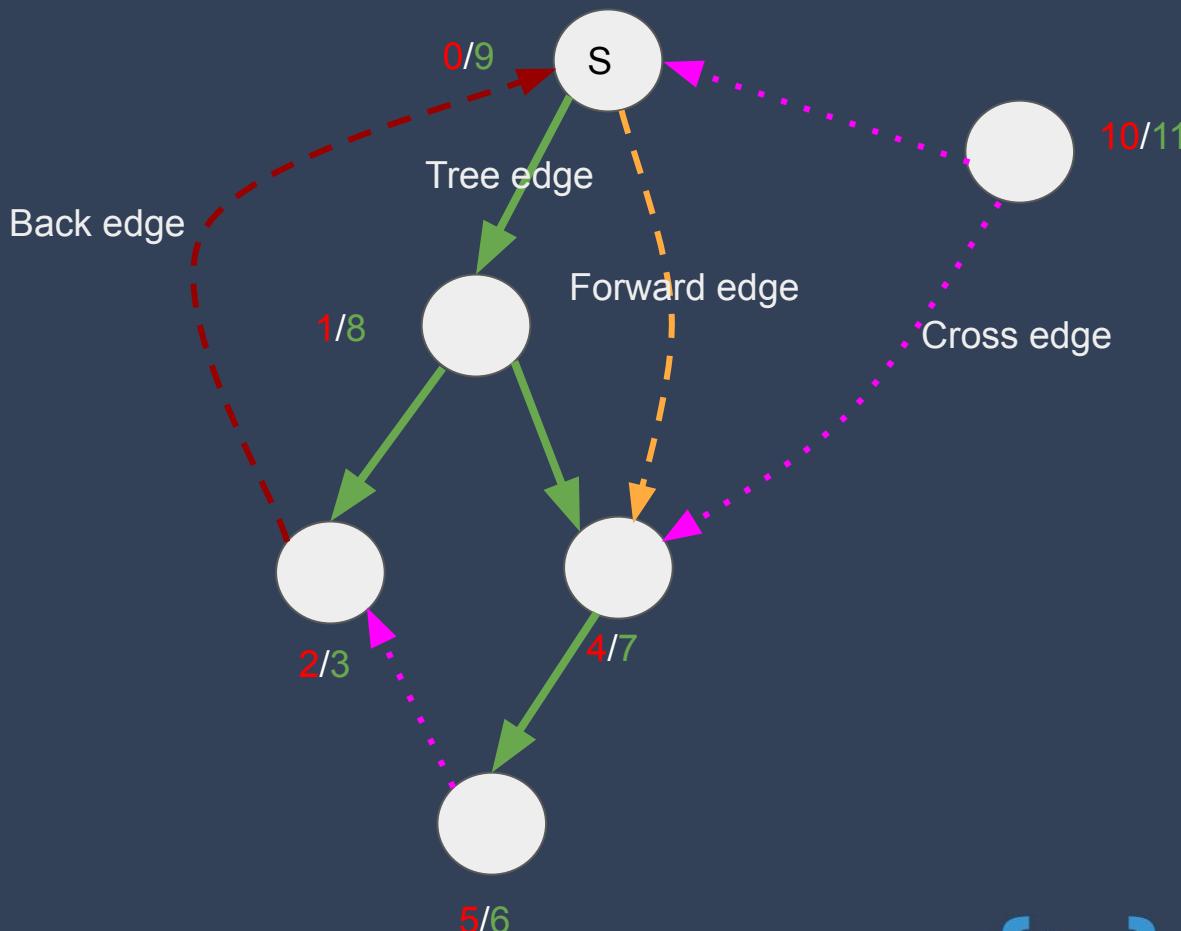












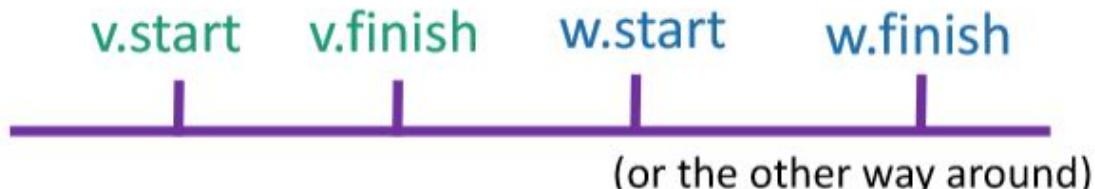
- If v is a descendent of w in this tree:



- If w is a descendent of v in this tree:



- If neither are descendants of each other:



**Imagine starting from the root and tumbling down tree edges, forward edges and going across cross edges. What is happening to the departure times?**

# Only back edges create a cycle.

Tree edges, forward edges and cross edges always go from bigger to smaller departure times. To go from a smaller to bigger departure time, need a back edge.

So a directed graph has no cycle iff there is no back edge in its DFS tree.

But how does one  
detect a back edge?  
Can it be done in a  
single DFS pass?

```

class Graph {
    time = 0
    //arr, dep are two arrays recording arrival and departure times of each vertex; initialized to -1

    void DFS(int source) {
        visited[source] = 1
        arr[source] = time++
        for neighbor in adjList[source]:
            if visited[neighbor] == 0:
                // (source,neighbor) is a tree edge
                DFS(neighbor)
            else: //visited[neighbor] == 1
                if departure time of neighbor is not yet set:
                    //back edge
                elif arr[source] < arr[neighbor]: //and departure time of neighbor is set
                    //forward edge
                else: // arr[source] > arr[neighbor] and departure time of neighbor is set
                    //cross edge
                dep[source] = time++
    }
}

```

## 207. Course Schedule

Medium    1941    92    Favorite    Share

There are a total of  $n$  courses you have to take, labeled from  $0$  to  $n-1$ .

Some courses may have prerequisites, for example to take course  $0$  you have to first take course  $1$ , which is expressed as a pair:  $[0,1]$

Given the total number of courses and a list of prerequisite **pairs**, is it possible for you to finish all courses?

### Example 1:

**Input:** 2, [[1,0]]

**Output:** true

**Explanation:** There are a total of 2 courses to take.

To take course 1 you should have finished course 0. So it is possible.

### Example 2:

**Input:** 2, [[1,0],[0,1]]

**Output:** false

**Explanation:** There are a total of 2 courses to take.

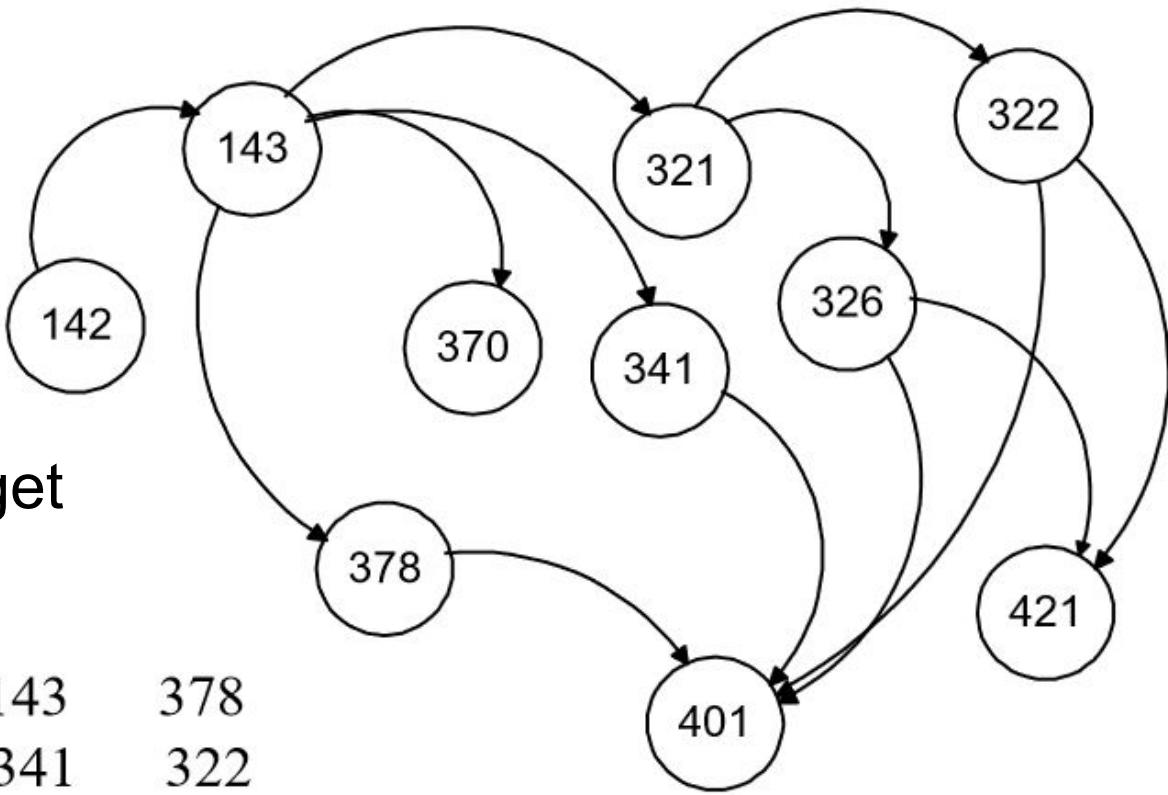
To take course 1 you should have finished course 0, and to take course 0 you should also have finished course 1. So it is impossible.

### Note:

1. The input prerequisites is a graph represented by **a list of edges**, not adjacency matrices. Read more about how a graph is represented.
2. You may assume that there are no duplicate edges in the input prerequisites.

Is it possible to get  
your degree?

Example: 142      143      378  
370      321      341      322  
326      421      401

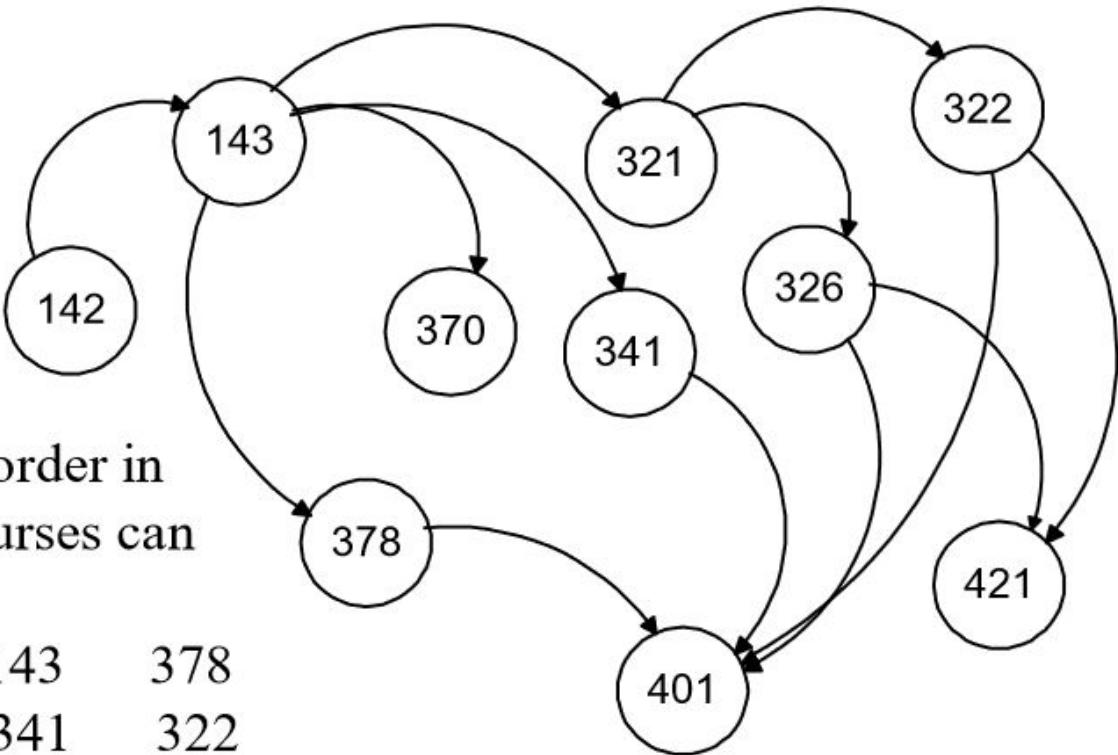


```
8             n = numCourses
9             adjlist = [ [] for _ in range(n)]
10            for (src,dst) in prerequisites:
11                #adjlist[src].append(dst)
12                adjlist[dst].append(src)
13
14            visited = [-1] * n
15            #parent = [-1] * n
16            timestamp = [0]
17            arrival = [-1] * n
18            departure = [-1] * n
```

```
20 def dfs(source):
21     arrival[source] = timestamp[0]
22     timestamp[0] += 1
23     visited[source] = 1
24     for neighbor in adjlist[source]:
25         if visited[neighbor] == -1:
26             #parent[neighbor] = source
27             if dfs(neighbor):
28                 return True
29         else:
30             if departure[neighbor] == -1:
31                 #This is a back edge, hence a Cycle!
32                 return True
33     departure[source] = timestamp[0]
34     timestamp[0] += 1
35     return False
```

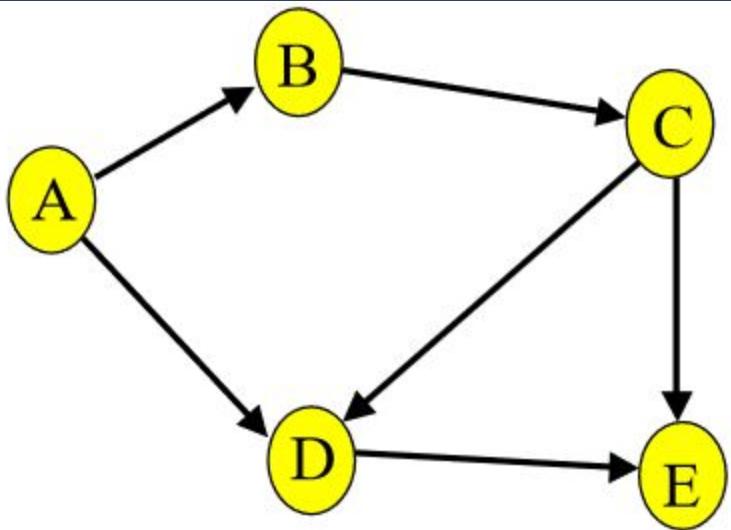
```
37     #components = 0 -- This makes no sense in a directed graph
38     for v in range(n):
39         if visited[v] == -1:
40             #components += 1
41             #if components > 1:
42                 #    return False
43         if dfs(v):
44             #If cycle found, you cannot complete the courses
45             return False
46     return True #No cycle found anywhere
```

Given a directed acyclic graph G, we can order the vertices of G so that every edge goes from left to right. This ordering is called a topological sort.

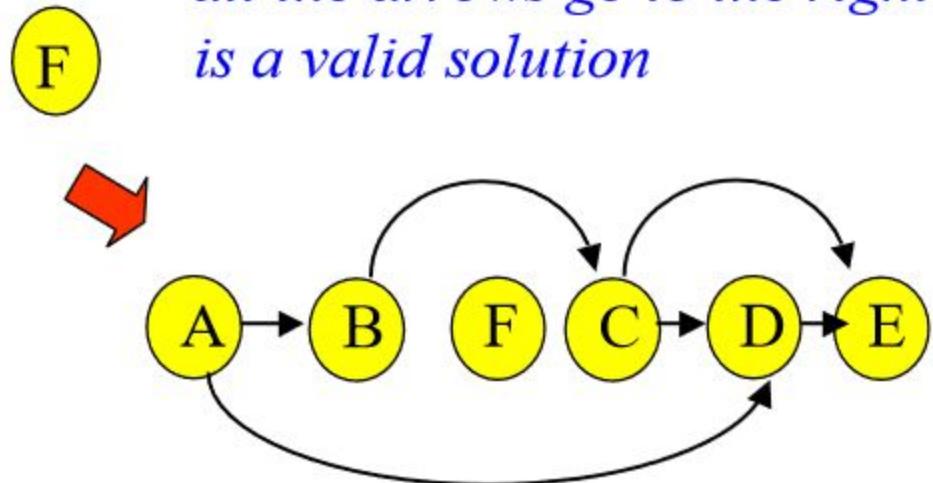


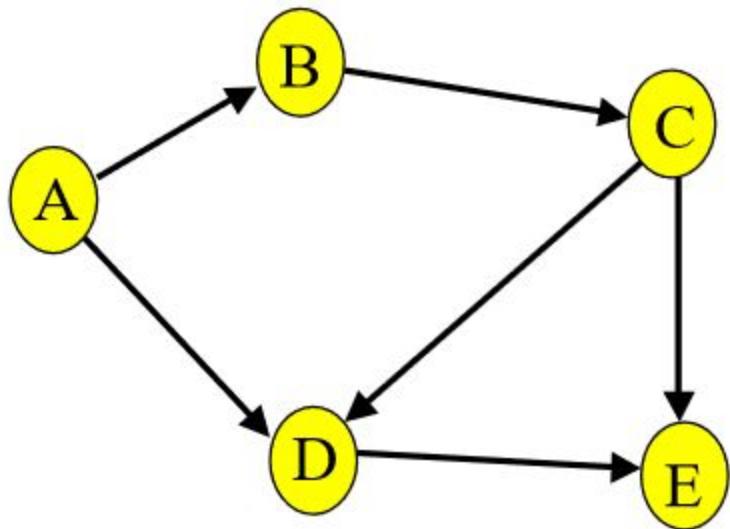
Problem: Find an order in which all these courses can be taken.

Example: 142    143    378  
          370    321    341    322  
          326    421    401

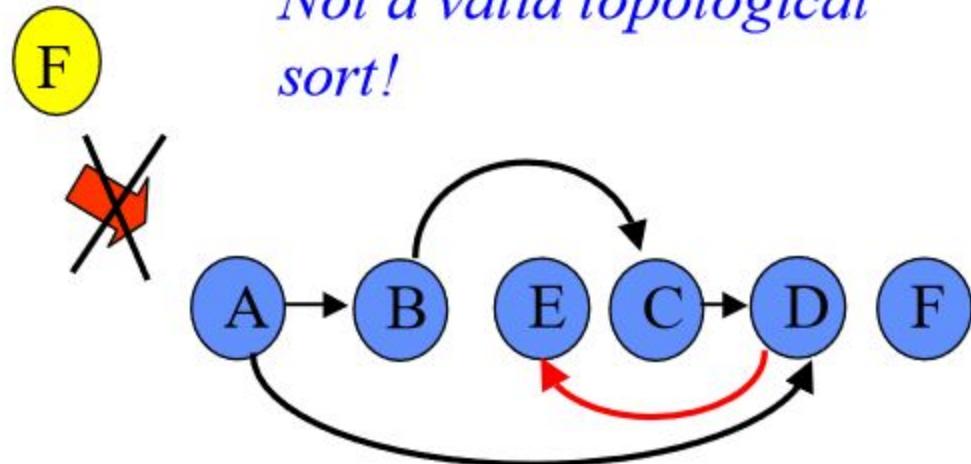


*Any linear ordering in which all the arrows go to the right is a valid solution*





*Not a valid topological sort!*



## 210. Course Schedule II

Medium    1033    75    Favorite    Share

There are a total of  $n$  courses you have to take, labeled from  $0$  to  $n-1$ .

Some courses may have prerequisites, for example to take course  $0$  you have to first take course  $1$ , which is expressed as a pair:  $[0,1]$

Given the total number of courses and a list of prerequisite **pairs**, return the ordering of courses you should take to finish all courses.

There may be multiple correct orders, you just need to return one of them. If it is impossible to finish all courses, return an empty array.

### Example 1:

**Input:** 2, [[1,0]]

**Output:** [0,1]

**Explanation:** There are a total of 2 courses to take. To take course 1 you should have finished course 0. So the correct course order is [0,1] .

### Example 2:

**Input:** 4, [[1,0],[2,0],[3,1],[3,2]]

**Output:** [0,1,2,3] or [0,2,1,3]

**Explanation:** There are a total of 4 courses to take. To take course 3 you should have finished both courses 1 and 2. Both courses 1 and 2 should be taken after you finished course 0. So one correct course order is [0,1,2,3]. Another correct ordering is [0,2,1,3] .

```
8             n = numCourses
9
10            adjlist = [ [] for _ in range(n)]
10 ▼          for (src,dst) in prerequisites:
11                  #adjlist[src].append(dst)
12                  adjlist[dst].append(src)
13
14            visited = [-1] * n
15            #parent = [-1] * n
16            timestamp = [0]
17            arrival = [-1] * n
18            departure = [-1] * n
19            topsort = []
20
```

```
21 def dfs(source):
22     arrival[source] = timestamp[0]
23     timestamp[0] += 1
24     visited[source] = 1
25     for neighbor in adjlist[source]:
26         if visited[neighbor] == -1:
27             #parent[neighbor] = source
28             if dfs(neighbor):
29                 return True
30         else:
31             if departure[neighbor] == -1:
32                 #This is a back edge, hence a Cycle!
33                 return True
34     departure[source] = timestamp[0]
35     timestamp[0] += 1
36     topsort.append(source)
37     return False
38
```

```
39         #components = 0 -- This makes no sense in a directed graph
40     for v in range(n):
41         if visited[v] == -1:
42             #components += 1
43             #if components > 1:
44             #    return False
45         if dfs(v):
46             #If cycle found, you cannot complete the courses
47             return []
48     topsort.reverse()
49     return topsort #No cycle found anywhere
```

## Alternate method (Kahn)

Calculate and maintain an in-degree array for each vertex. Keep all vertices with zero in-degree in a queue/stack. If graph is a DAG, there must exist a vertex with zero indegree. Make it the first vertex in the topological sort, and delete all its outgoing edges (which means update in-degree array by decrementing the in-degree counts). If any new vertex with in-degree count 0 is found, add it to the queue/stack. Repeat until all vertices in the graph are sorted.  
Doable in  $O(V+E)$ ?

## 1192. Critical Connections in a Network

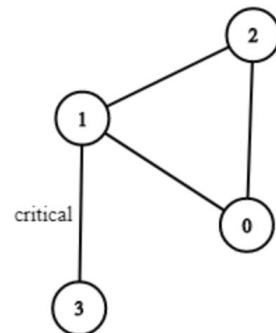
Hard    239    25    Favorite    Share

There are `n` servers numbered from `0` to `n-1` connected by undirected server-to-server `connections` forming a network where `connections[i] = [a, b]` represents a connection between servers `a` and `b`. Any server can reach any other server directly or indirectly through the network.

A *critical connection* is a connection that, if removed, will make some server unable to reach some other server.

Return all critical connections in the network in any order.

### Example 1:



**Input:** `n = 4, connections = [[0,1],[1,2],[2,0],[1,3]]`

**Output:** `[[1,3]]`

**Explanation:** `[[3,1]]` is also accepted.

```
2 def criticalConnections(self, n, connections):
3     """
4         :type n: int
5         :type connections: List[List[int]]
6         :rtype: List[List[int]]
7     """
8
9     adjlist = [ [] for _ in range(n)]
10    for (src, dst) in connections:
11        adjlist[src].append(dst)
12        adjlist[dst].append(src)
```

```
14     visited = [-1]*n
15     arrival = [-1]*n
16     oldestarrival = [-1]*n
17     parent = [-1]*n
18     departure = [-1]*n
19     timestamp = [0]
20
21     result = []
22
23     def dfs(source):
24         arrival[source] = timestamp[0]
25         timestamp[0] += 1
26         oldestarrival[source] = arrival[source]
27         visited[source] = 1
28         for neighbor in adjlist[source]:
29             if visited[neighbor] == -1:
30                 parent[neighbor] = source
31                 oldestarrival[source] = min(oldestarrival[source], dfs(neighbor))
32             else:
33                 if neighbor != parent[source]:
34                     oldestarrival[source] = min(oldestarrival[source], arrival[neighbor])
35         if oldestarrival[source] == arrival[source] and source != 0:
36             result.append((source, parent[source]))
37             departure[source] = timestamp[0]
38             timestamp[0] += 1
39         return oldestarrival[source]
40
41     dfs(0) #The graph is connected, so outer loop not needed
42     return result
43
```

{ik} INTERVIEW  
KICKSTART

# Defunct slides

```
//initialize a “departure_times” array of size 2V to 0.  
bool hasCycle(int v) {  
    flag = false  
    visited[v] = 1  
    arr[v] = time++  
    for w in adjList[v]:  
        if visited[w] == 0:  
            // v,w is a tree edge  
            flag = flag or DFS(w)  
        else: //visited[w] == 1  
            If a back edge is detected:  
                flag = true  
    dep[v] = time++  
    departure_times[dep[v]] = v  
    return flag  
}
```

Just sort the vertices in decreasing order of departure times. Can do it by maintaining a stack, and pushing every vertex into it at the time of departure from that vertex.

Alternate method: See pseudocode to the left (and next slide)

Method 3: Calculate and maintain an in-degree array for each vertex. Keep all vertices with zero in-degree in a queue/stack. If graph is a DAG, there must exist a vertex with zero indegree. Make it the first vertex in the topological sort, and delete all its outgoing edges (which means update in-degree array by decrementing the in-degree counts). If any new vertex with in-degree count 0 is found, add it to the queue/stack. Repeat until all vertices in the graph are sorted.

```

//initialize a "departure_times" array of size 2V to 0. This will record which vertex had that departure time
bool hasCycle(int v) {
    flag = false
    visited[v] = 1
    arr[v] = time++
    for w in adjList[v]:
        if visited[w] == 0:
            // v,w is a tree edge
            flag = flag or DFS(w)
        else: //visited[w] == 1
            if arr[v] < arr[w] and dep[v] > dep[w]:
                //forward edge
            if arr[v] > arr[w] and dep[v] < dep[w]:
                flag = true //back edge
            if arr[w] < dep[w] < arr[v] < dep[v]:
                //cross edge
            dep[v] = time++
    departure_times[dep[v]] = v
    return flag
}

```

```

topsort = new List();
for t in 2V down to 1:
    if departure_times[t] != 0:
        topsort.append(departure_times[t])

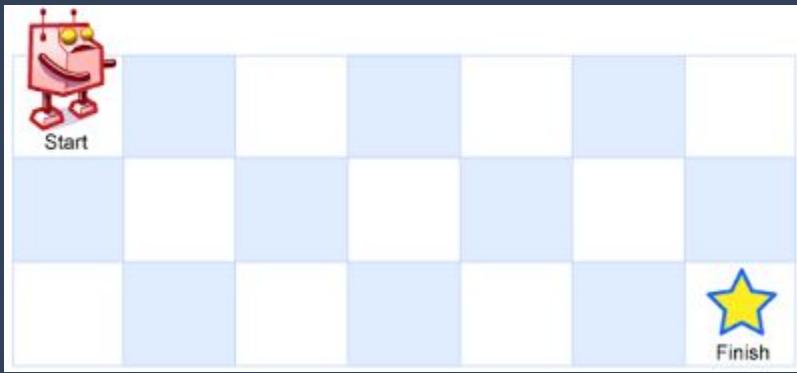
```

//topsort now contains the topologically sorted list of vertices

**A robot is located at the top-left corner of a  $m \times n$  grid (marked 'Start' in the diagram below).**

**The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid (marked 'Finish' in the diagram below).**

**How many possible unique paths are there?**



```

//initialize a “departure_times” array of size 2V to 0. This will record which vertex had that departure time
bool hasCycle(int v) {
    flag = false
    visited[v] = 1
    arr[v] = time++
    for w in adjList[v]:
        if visited[w] == 0:
            // v,w is a tree edge
            flag = flag or DFS(w)
        else: //visited[w] == 1
            if arr[v] < arr[w] and dep[v] > dep[w]:
                //forward edge
            if arr[v] > arr[w] and dep[v] < dep[w]:
                flag = true //back edge
            if arr[w] < dep[w] < arr[v] < dep[v]:
                //cross edge
            dep[v] = time++
            departure_times[dep[v]] = v
    return flag
}

//Initialize array “numpaths” to 0 to record
//number of paths to goal vertex out of v
//numpaths[goal] = 1
topsort = new List();
for t in 2V down to 1:
    if departure_times[t] != 0:
        topsort.append(departure_times[t])
        //Let v = departure_times[t]
        for w in adjList[v]:
            numpaths[v] += numpaths[w]

//numpaths[1] will tell you number of paths
from start vertex to goal vertex

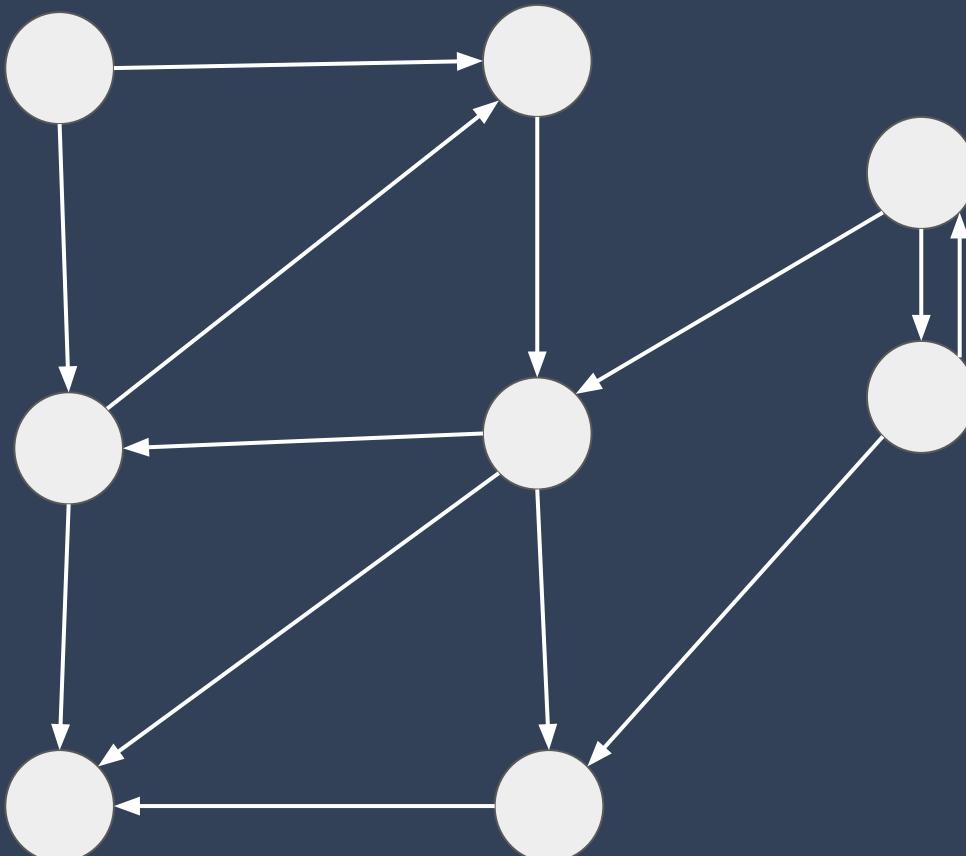
```

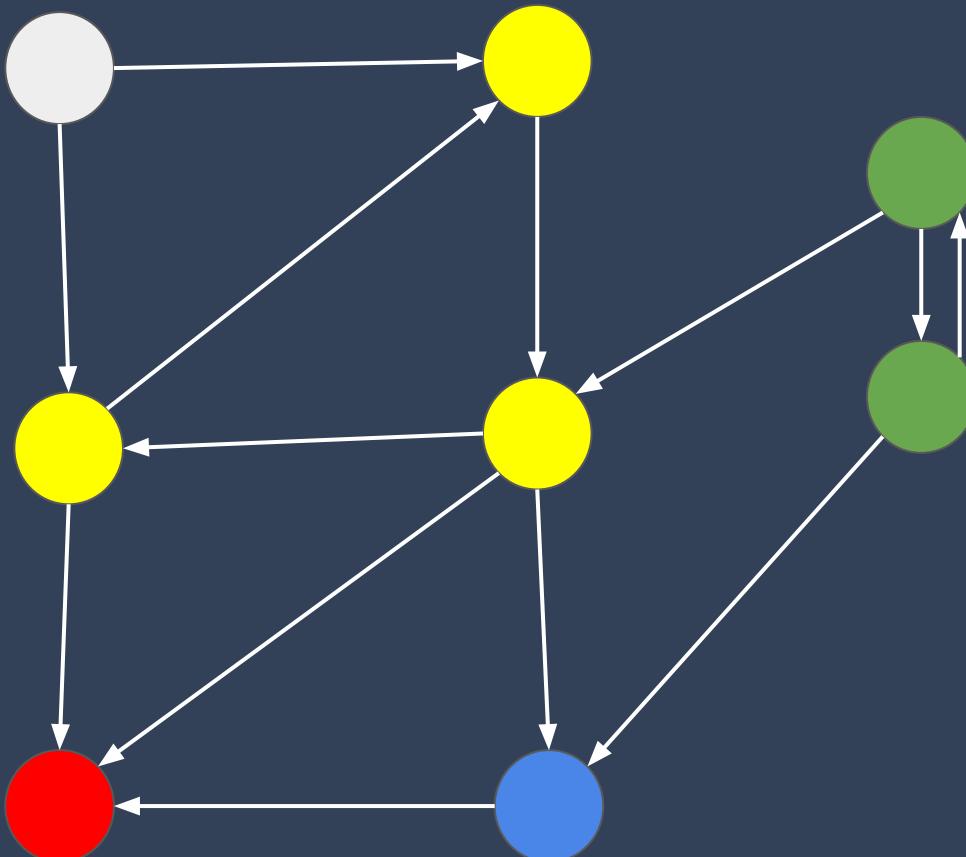
}

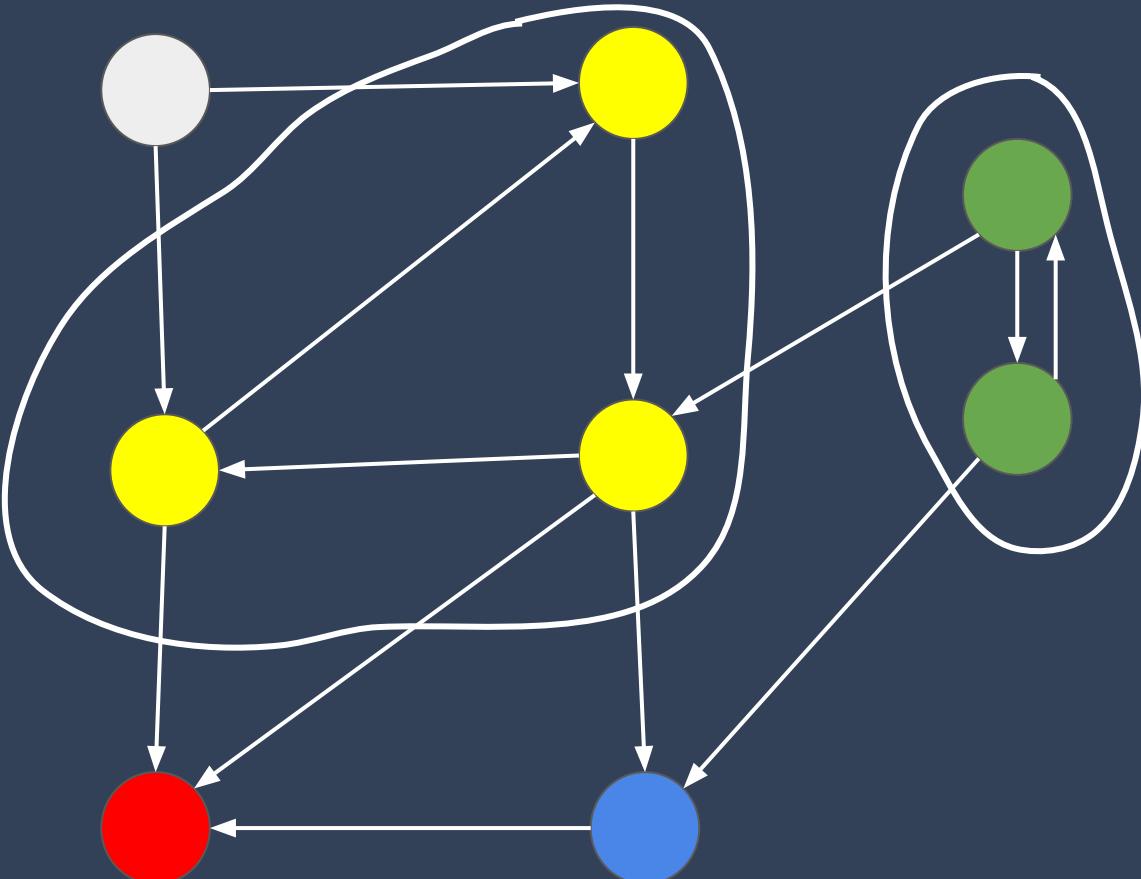
**Find the longest path in a DAG  
with weighted edges.**

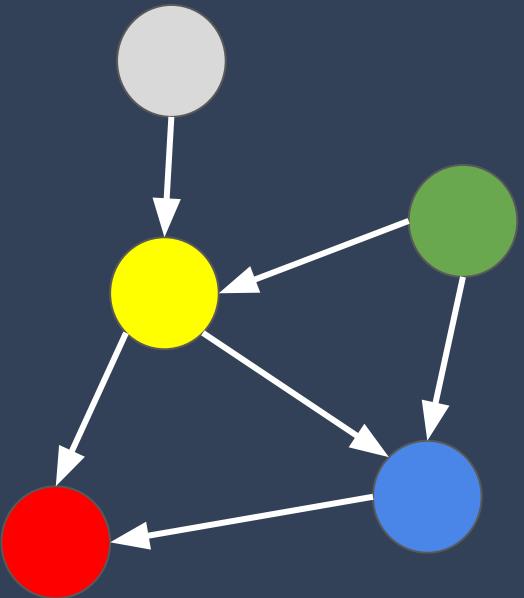
**How about shortest path  
from a source node s?**

If  $G$  is not a DAG, then it has some cycles. The vertices within a cycle are mutually reachable. They are in the same **strongly connected component.**



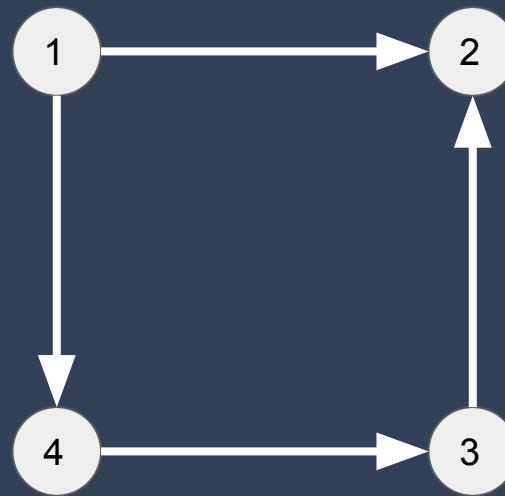
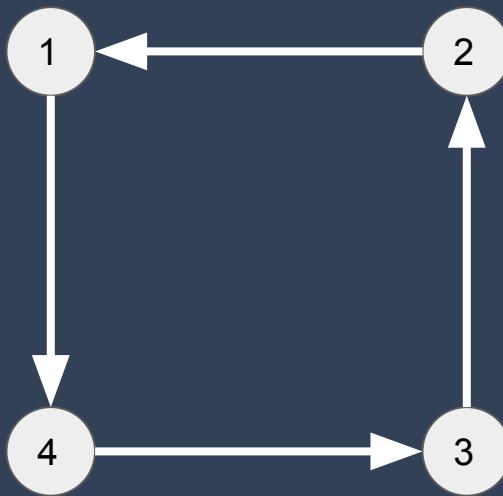


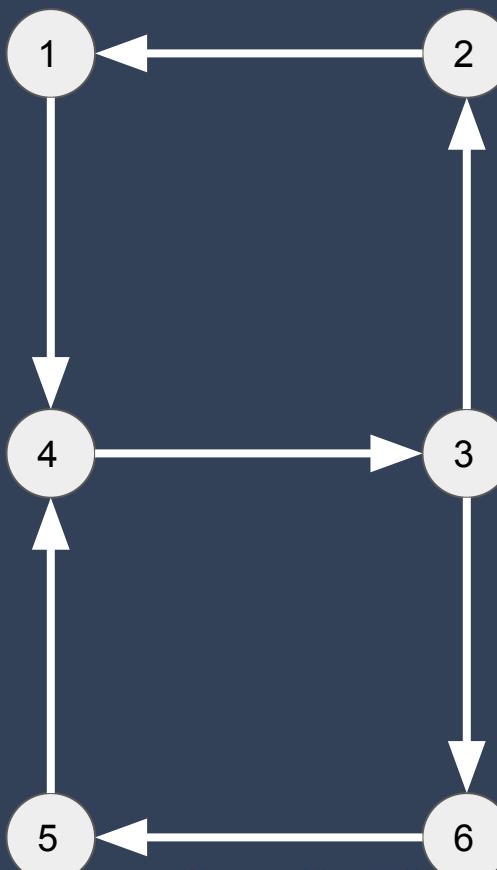
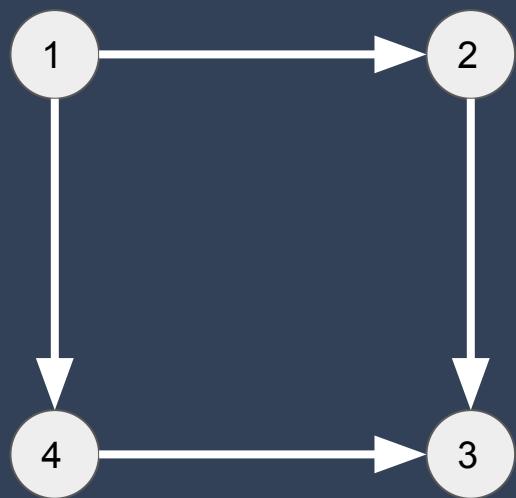




DAG of  
strongly  
connected  
components

Given a directed graph  
G, is it strongly  
connected?





If  $\text{DFS}(v)$  visits all vertices in  $G$ , there exists a path from  $v$  to every vertex in  $G$ . Suppose there also exists a path from every vertex in  $G$  to  $v$ . Is  $G$  strongly connected?

How do we determine  
that there exists a path  
from every vertex in G  
to v?

**Method 1: In  $\text{reverse}(G)$ ,  
there should exist a  
path from  $v$  to every  
vertex in  $G$ .**

# Method 2: Use arrival times.

There should exist an edge (back or cross) going out of every subtree.

**Make  $\text{DFS}(v)$  return the node  
with the smallest arrival time  
to which there is an edge  
from the subtree rooted at  $v$ .**

```

class Graph {
    time = 0
    //arr, dep are two arrays recording arrival and departure times of each vertex; initialized to -1

    void DFS(int v) {
        visited[v] = 1
        arr[v] = time++
        for w in adjList[v]:
            if visited[w] == 0:
                // v,w is a tree edge
                DFS(w)
            else: //visited[w] == 1
                if arr[v] < arr[w] and dep[v] > dep[w]:
                    //forward edge
                if arr[v] > arr[w] and dep[v] < dep[w]:
                    //back edge (but exclude edge to parent!)
                if arr[w] < dep[w] < arr[v] < dep[v]:
                    //cross edge
        dep[v] = time++
    }
}

```

```

class Graph {
    time = 0; flag = true
    //arr, dep are two arrays recording arrival and departure times of each vertex; initialized to -1

    int DFS(int v) {
        visited[v] = 1
        arr[v] = time++
        sat = arr[v] //smallest arrival time to any node from this subtree.
        for w in adjList[v]:
            if visited[w] == 0:
                // v,w is a tree edge
                sat = min(sat, DFS(w))
            else: //visited[w] == 1
                if arr[v] < arr[w] and dep[v] > dep[w]:
                    //forward edge
                if arr[v] > arr[w] and dep[v] < dep[w]:
                    sat = min(sat, DFS(w)) //back edge
                if arr[w] < dep[w] < arr[v] < dep[v]:
                    sat = min(sat, DFS(w)) //cross edge
        dep[v] = time++; if sat == arr[v] then flag = false; //but ok for root to have sat == arr[v]
    }
}

```

Given an undirected  
graph G, does it have a  
**bridge?**

Does it have an  
**articulation point?**

Find all strongly  
connected components  
of a directed graph G.

# Kosaraju's algorithm

1. Do a DFS on  $G$  and mark the departure times (or finish times) of all the vertices. (This may involve multiple DFS calls from unvisited vertices)
2. Reverse all edges to create a new graph  $G_r$
3. Do a DFS on  $G_r$  with new DFS calls initiated on vertices in decreasing order of departure times in step 1.
4. Every DFS tree will be a strongly connected component in the original graph  $G$ .

## **Uninformed search:**

**Explores options in every “direction”**

**No information about goal location**

## **Informed search:**

**Uses a heuristic**

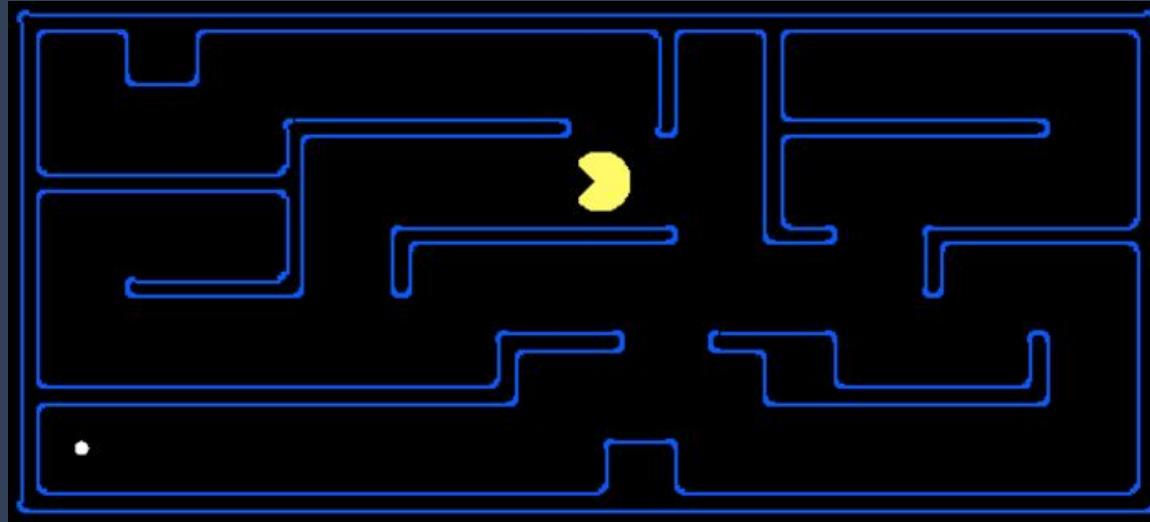
**For any search problem, you know the goal. Now, you have an idea of how far away you are from the goal.**

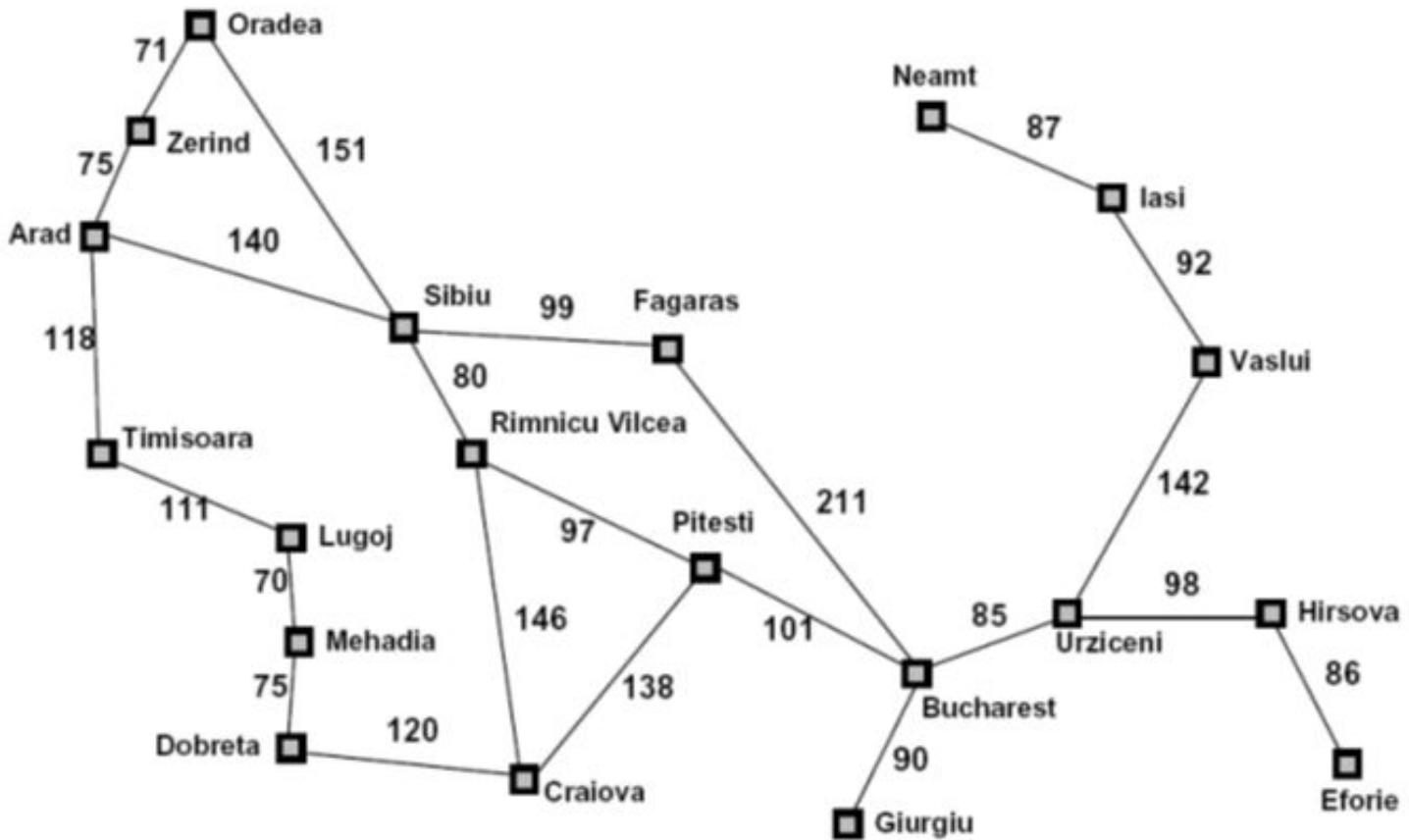
# A heuristic is:

A function that *estimates* how close a state is to a goal

Designed for a particular search problem

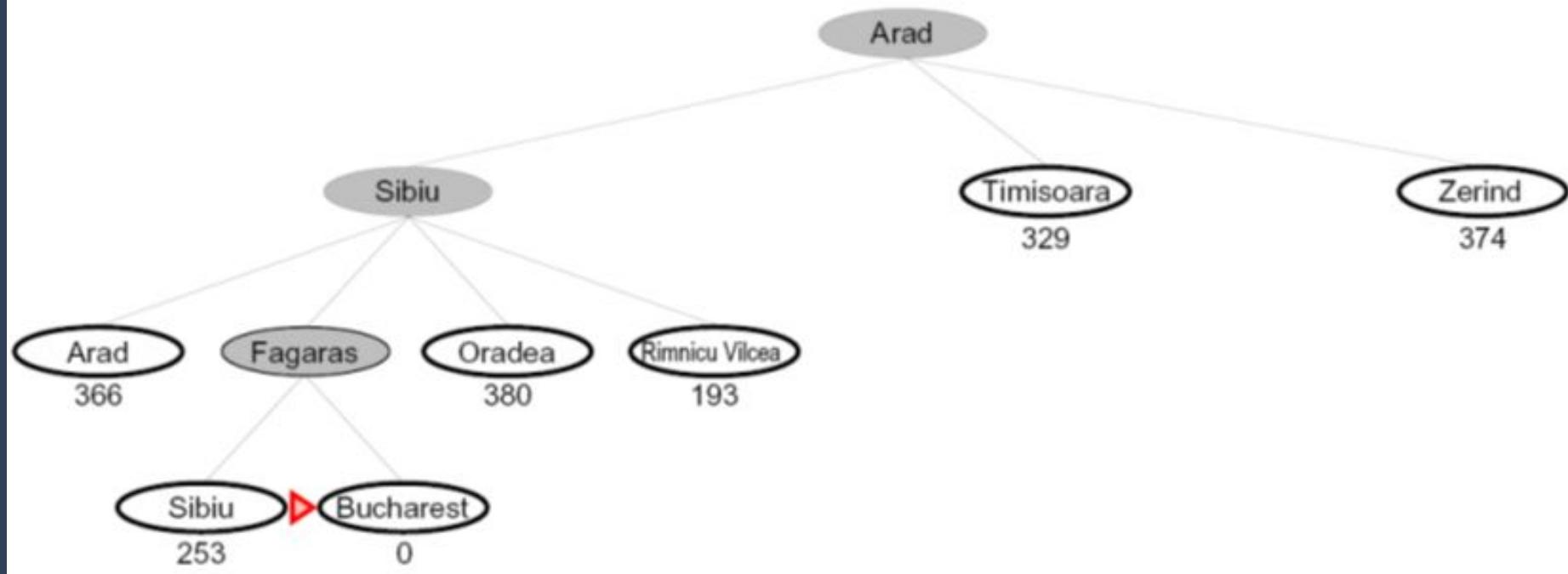
Examples: Manhattan distance, Euclidean distance for pathing





Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

# Best-first search

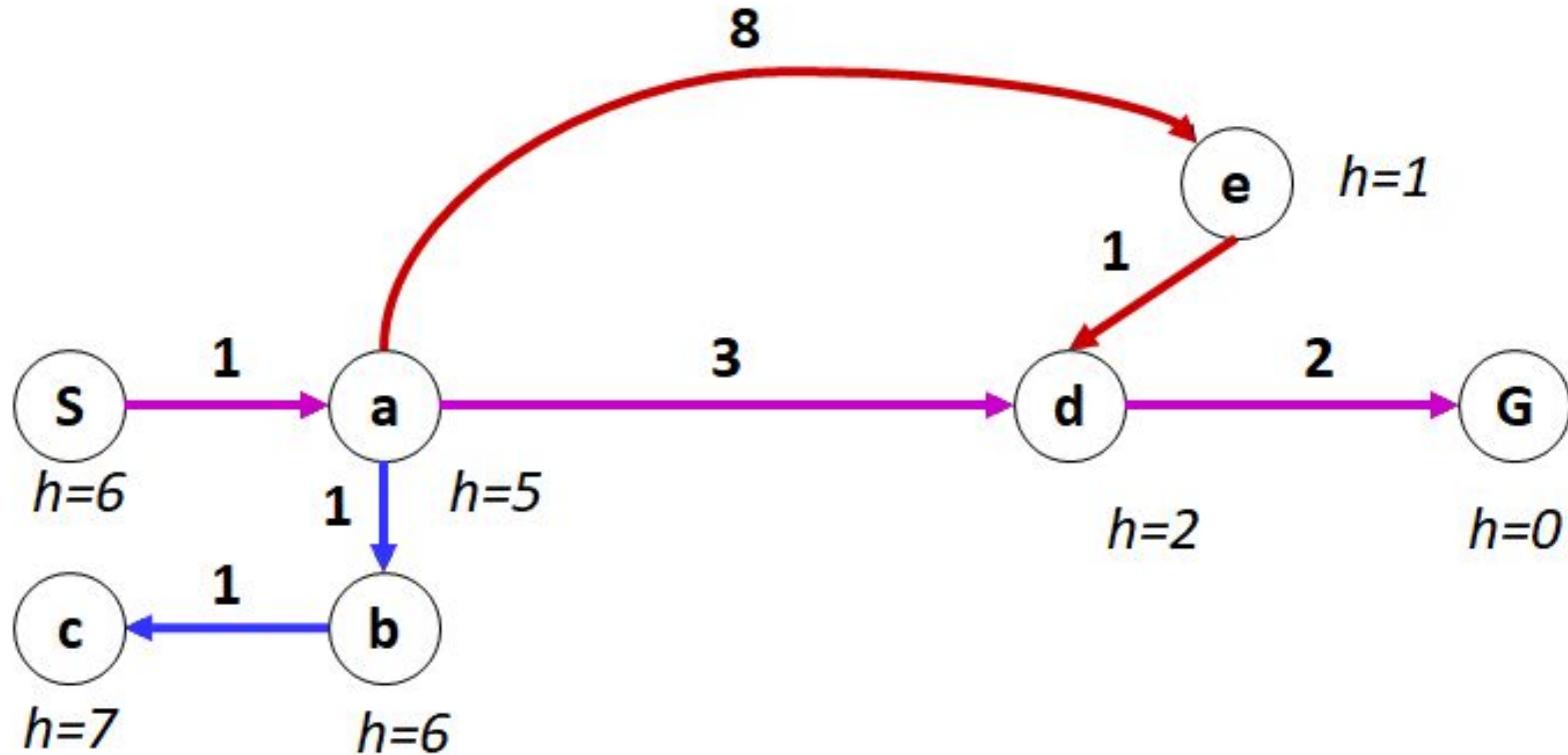


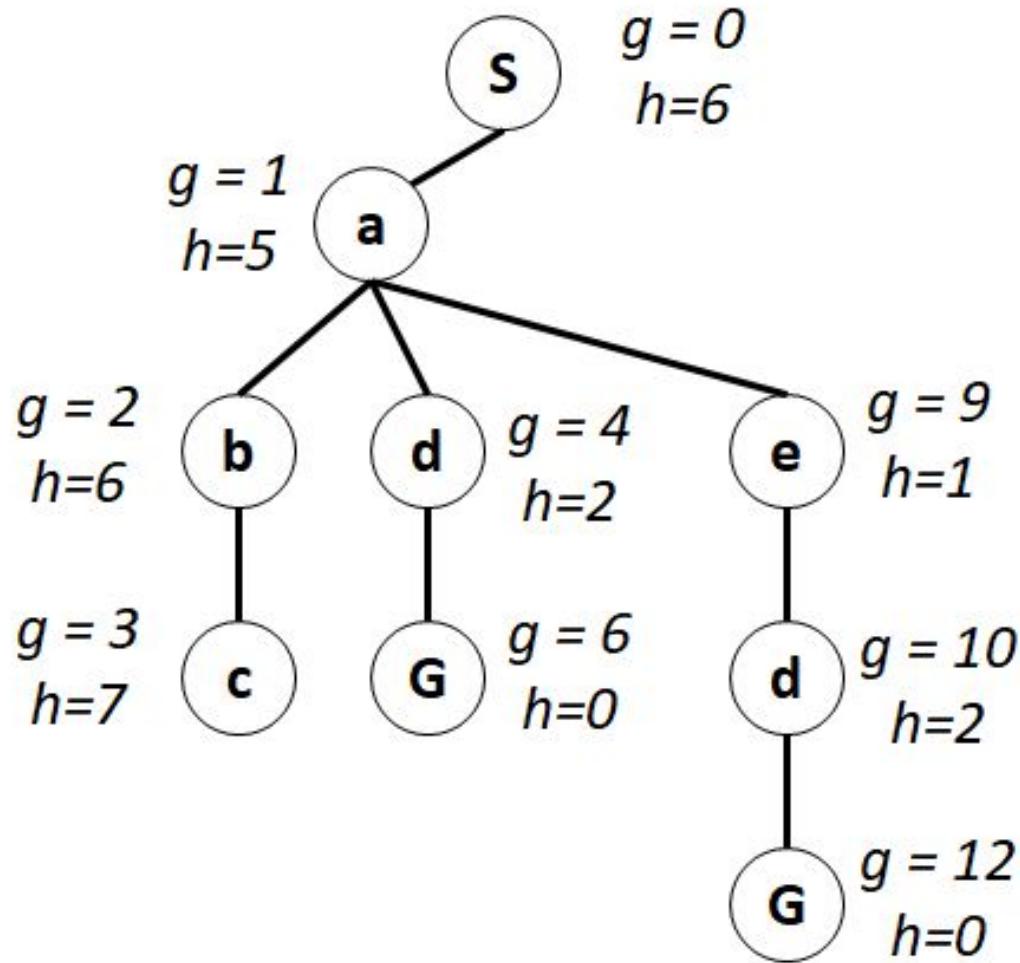
Strategy: expand a node that you think is closest to a goal state

Dijkstra orders by path cost, or *backward cost*  $g(n)$

Best-first orders by goal proximity, or *forward cost*  $h(n)$

A\* Search orders by the sum:  $f(n) = g(n) + h(n)$





Some citizens of Königsberg  
Were walking on the strand  
Beside the river Pregel  
With its seven bridges spanned.

“O Euler, come and walk with us”,  
Those burghers did beseech.  
“We’ll roam the seven bridges o’er,  
And pass but once by each.”

“It can’t be done”, thus Euler cried.  
“Here comes the Q. E. D.  
Your islands are but vertices,  
And four have odd degree.”

From Königsberg to König’s book,  
So runs the graphic tale,  
And still it grows more colorful,  
In Michigan and Yale.

(This poem was published in *Proof Techniques in Graph Theory*, ed. Frank Harary, p. 25, Copyright Academic Press (1969).)

# Objects and Pointers

When you implement a Graph, one popular way to implement it is by having a Vertex class with several member variables, and *one of them is a pointer to a list of neighboring Vertices.*

This is different from what we saw for pure adjacency lists. In the pure Adjacency list implementation, *no vertex attributes are being stored in the structure.*

In the "**Objects and Pointers**" approach, the vertex objects can have a bunch of attributes, and a pointer to the list of neighbors is just one of the attributes. For example, if you create a transportation graph with vertices as cities and edges as roads connecting cities, then for a city like San Francisco, you may want to store population size, area, elevation above sea level, annual rainfall and other attributes. You're not just storing the neighboring cities. Think about how/where you will store these attributes.

# Standard Graph Representations focus on edges

Whether you look at pure Adjacency lists, adjacency maps or adjacency matrices, the focus in all of them is only on storing the edge information (presence/absence of edges, and associated weights on edges). No information about vertex attributes is being stored in them. You need to maintain vertices separately.

“Objects and pointers” is the approach where you store vertex attributes in the same structure with adjacency lists/maps.

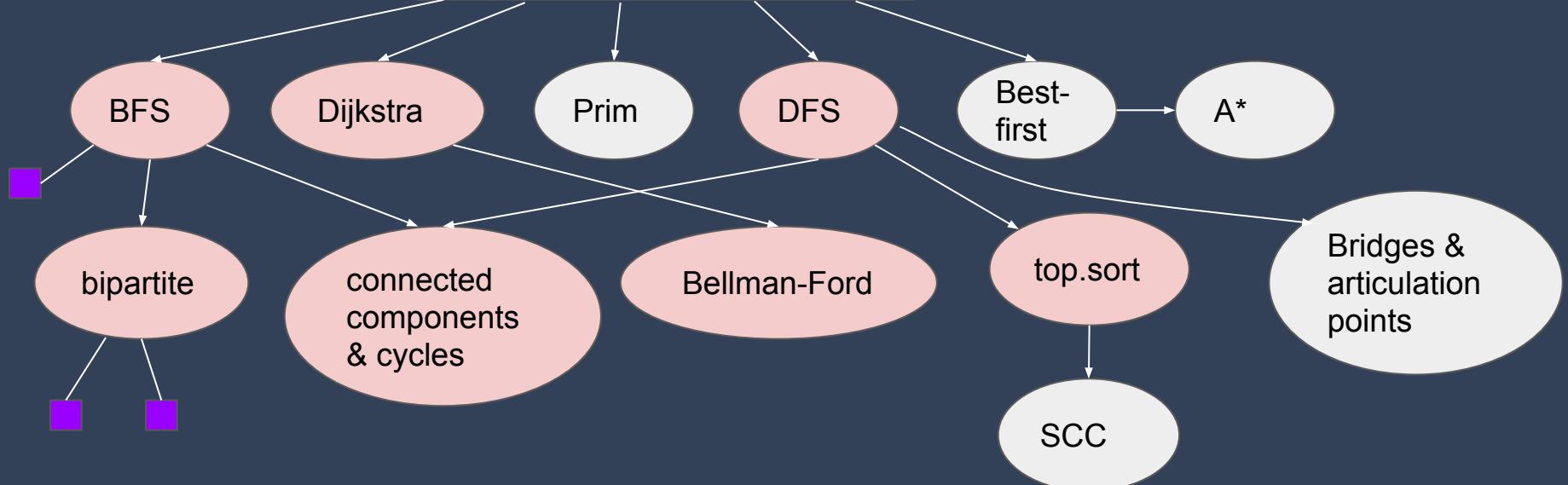
In some cases, even the edge attributes may be stored outside the Adjacency list/map/matrix structure, and the Adjacency structure may only store pointers to the Edge objects that have all the detailed edge attribute information.

These Edge objects can even be linked together in an unordered “**Edge List**”, with each Edge object having (aside from other edge attributes) two pointers to the two end-point vertex objects.

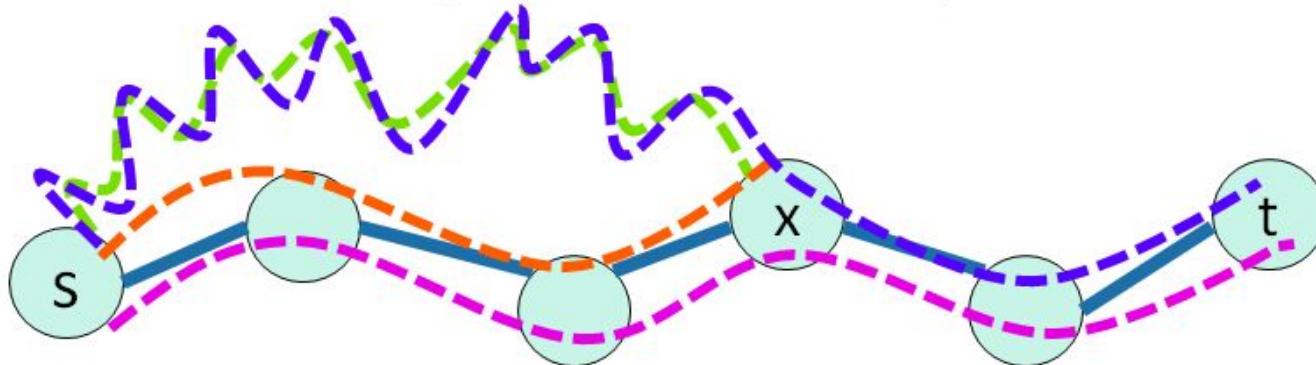
Eulerian cycle or path problem

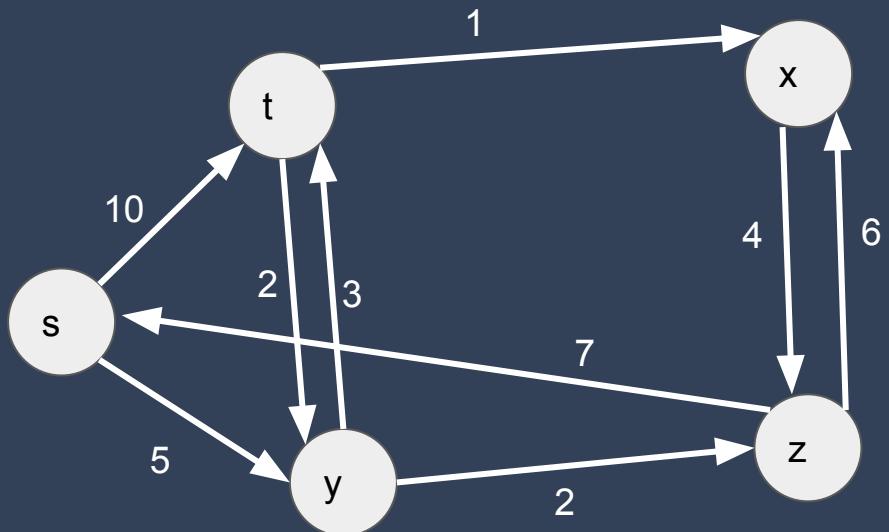
Graph representations

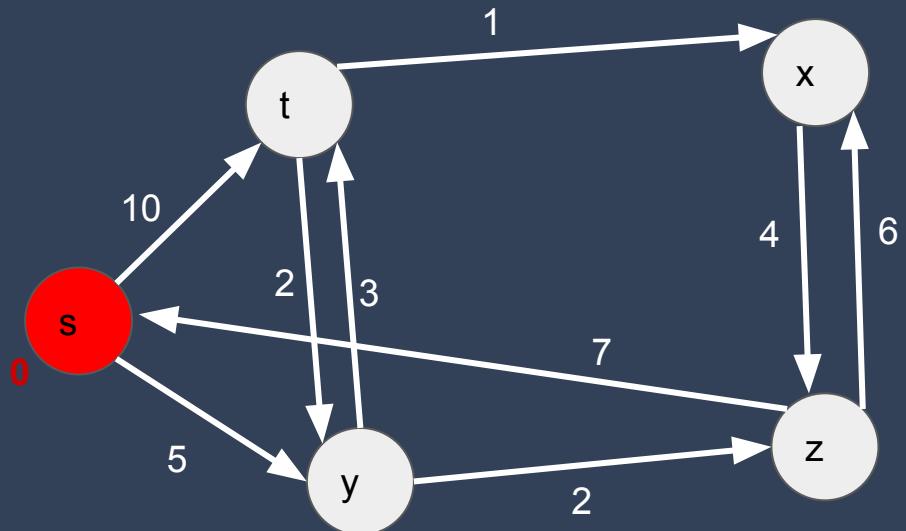
General graph traversal

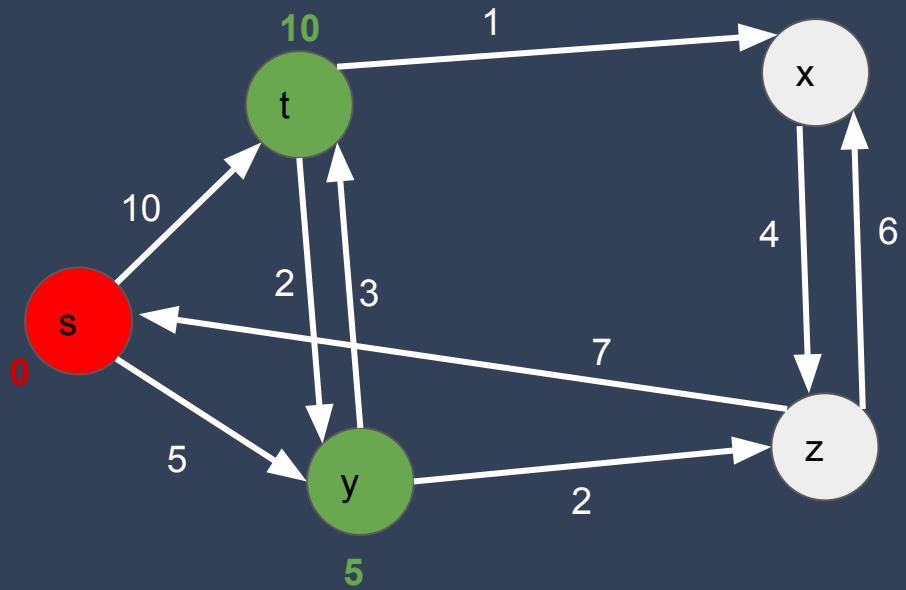


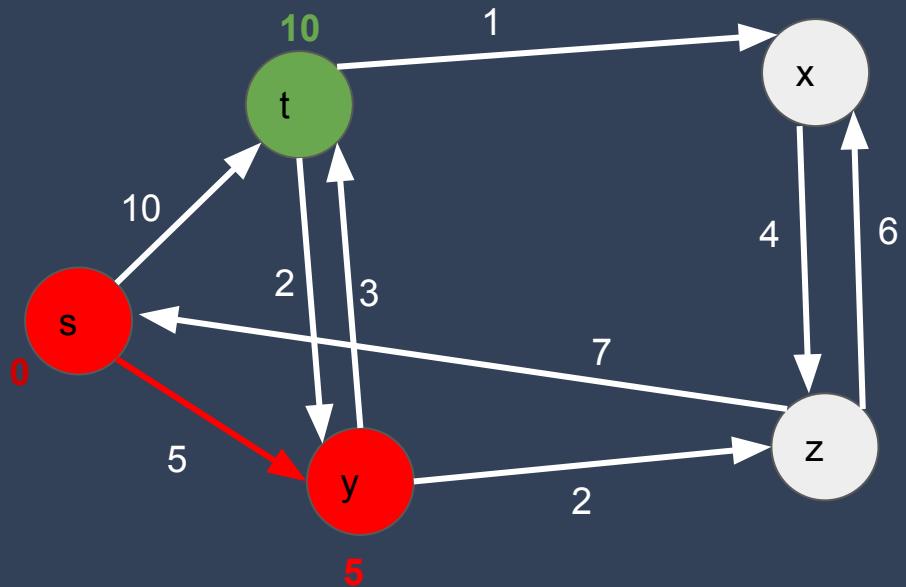
- A sub-path of a shortest path is also a shortest path.
- Say **this** is a shortest path from  $s$  to  $t$ .
- Claim: **this** is a shortest path from  $s$  to  $x$ .
  - Suppose not, **this** one is shorter.
  - But then that gives an **even shorter path** from  $s$  to  $t$ !

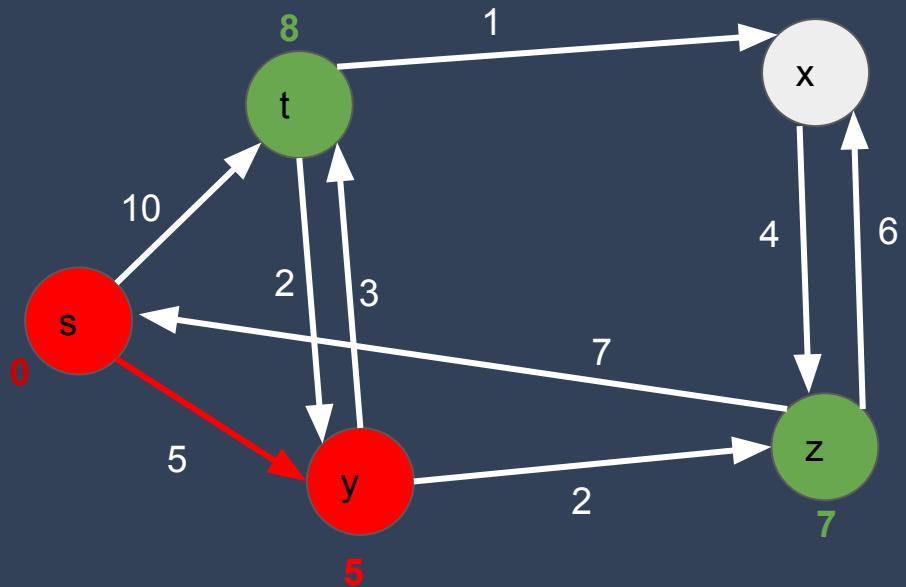


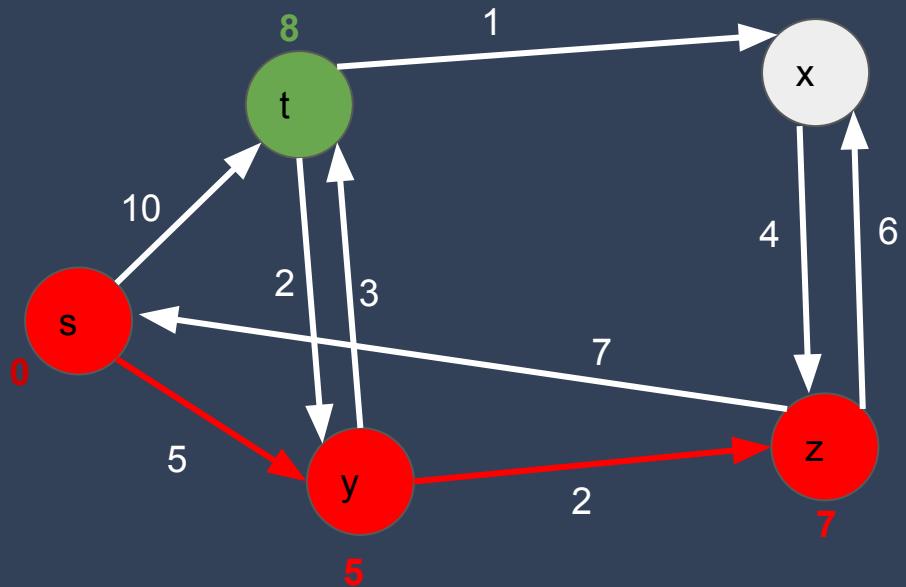


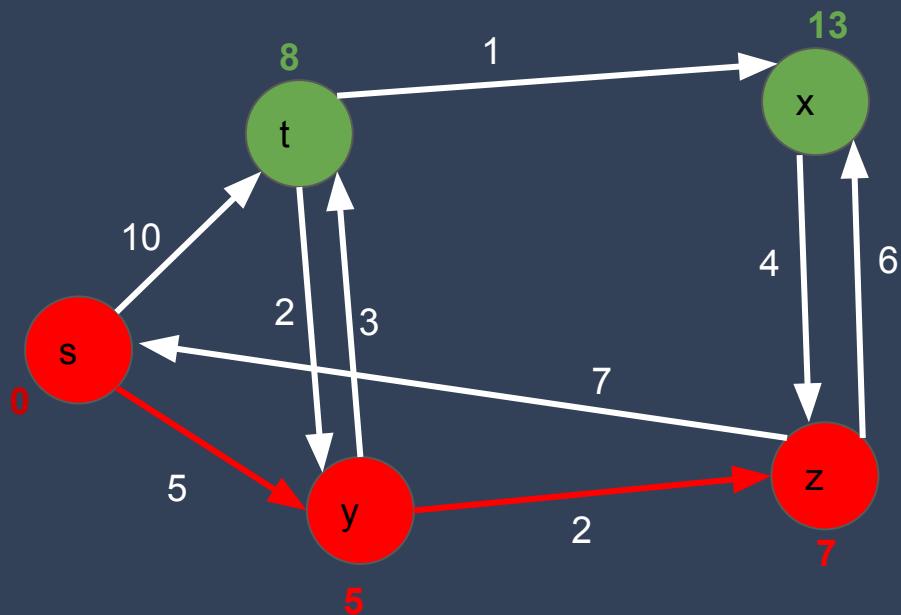


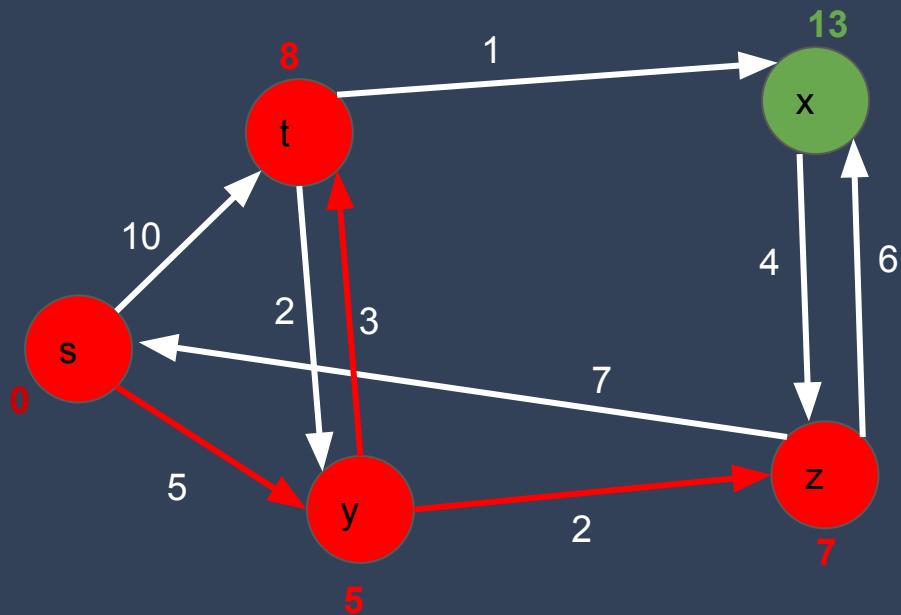


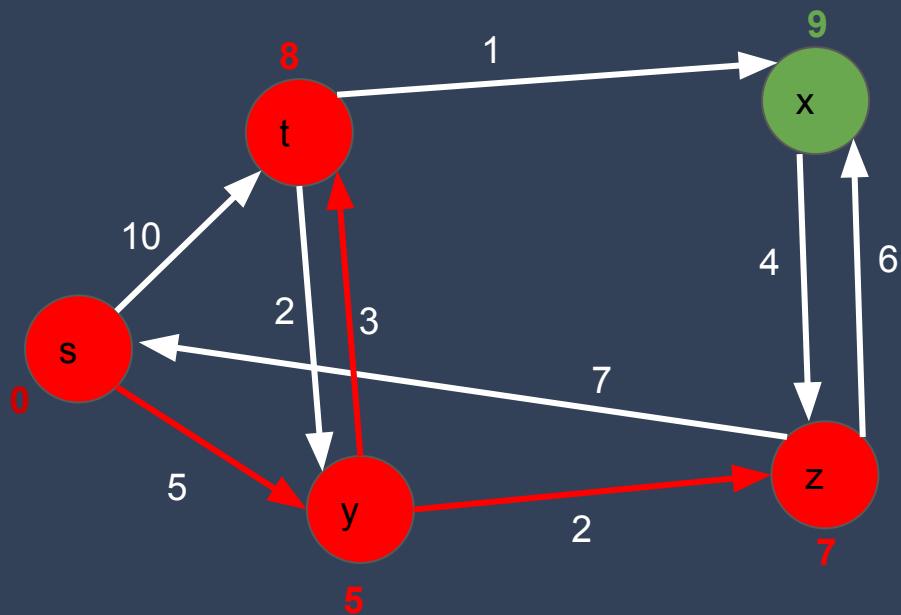


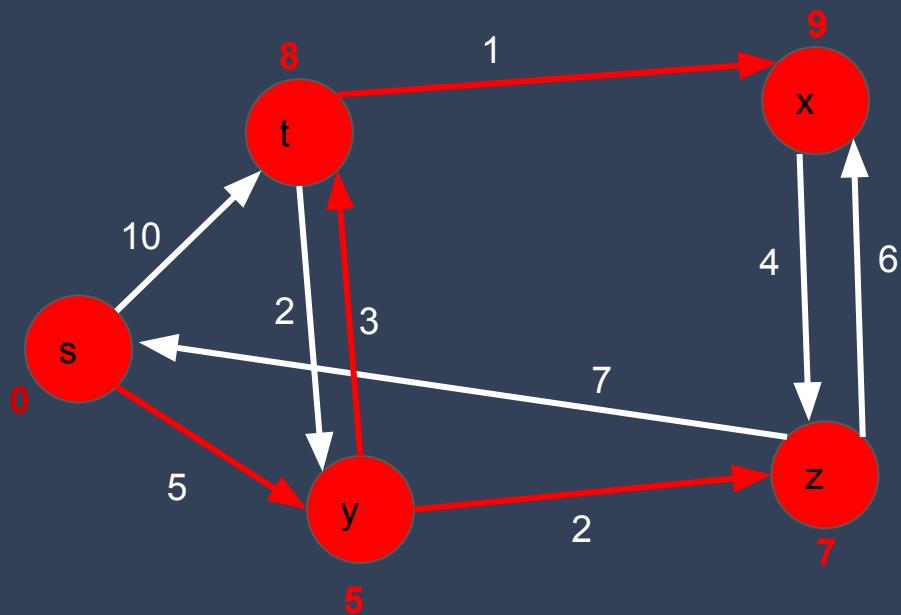












```

class Graph {

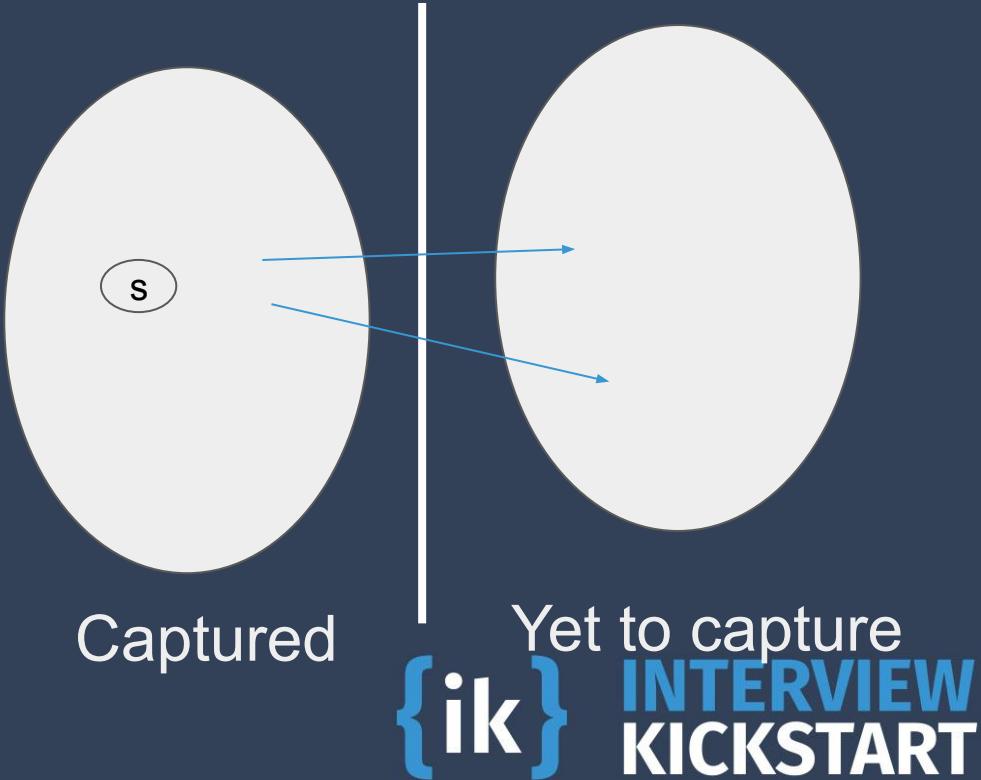
void BFS(int s) {
    //d, visited, captured and parent initialized to INF, 0, 0 and null
    d[s] = 0; captured[s] = 1; visited[s] = 1

    q = new Queue(); q.push(s);
    while not isEmpty(q): //capture the next vertex
        v = q.pop()
        captured[v] = 1
        for w in Adjlist[v]:
            if visited[w] == 0 then
                visited[w] = 1; parent[w] = v; d[w] = d[v] + 1
                q.push(w);
            else
                if d[w] == d[v] then graph is not bipartite
    //Conclusion: Graph is bipartite
}

}

```

# BFS



```
class Graph {
```

```
void Dijkstra(int s) {
```

```
    //distance, visited, captured and parent initialized to INF, 0, 0 and null
```

```
    distance[s] = 0; captured[s] = 1; visited[s] = 1
```

```
    q = new PQueue(); q.push(s, priority=0);
```

```
    while not isEmpty(q): //capture the next vertex
```

```
        v = q.pop()
```

```
        captured[v] = 1
```

```
        for w in Adjlist[v]:
```

```
            if visited[w] == 0: //undiscovered
```

```
                visited[w] = 1; parent[w] = v;
```

```
                distance[w] = distance[v] + weight(v,w)
```

```
                q.push(w, priority=distance[w]);
```

```
            else if !captured[w]: //discovered & in the fringe
```

```
                if distance[v] + weight(v,w) < distance(w):
```

```
                    distance[w] = distance[v] + weight(v,w)
```

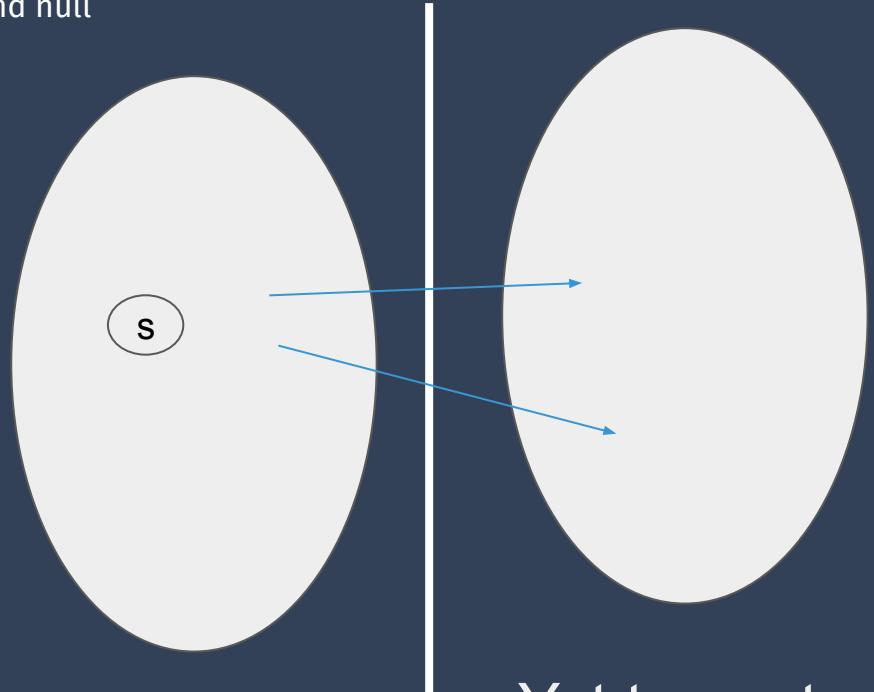
```
                    q.decreaseKey(w,priority=distance[w]);
```

```
                    parent[w] = v;
```

```
}
```

```
}
```

# Dijkstra's algorithm



{ik}

INTERVIEW  
KICKSTART

```
class Graph {
```

```
void Dijkstra(int s) {
```

```
    //distance, visited, captured and parent initialized to INF, 0, 0 and null
```

```
    distance[s] = 0; captured[s] = 1; visited[s] = 1
```

```
    q = new PQueue(); q.push(s, priority=0);
```

```
    while not isEmpty(q): //capture the next vertex
```

```
        v = q.pop()
```

```
        captured[v] = 1
```

```
        for w in Adjlist[v]:
```

```
            if visited[w] == 0: //undiscovered
```

```
                visited[w] = 1; parent[w] = v;
```

```
                distance[w] = distance[v] + weight(v,w)
```

```
                q.push(w, priority=distance[w]);
```

```
            else if !captured[w]: //discovered & in the fringe
```

```
                if distance[v] + weight(v,w) < distance(w):
```

```
                    distance[w] = distance[v] + weight(v,w)
```

```
                    q.decreaseKey(w,priority=distance[w]);
```

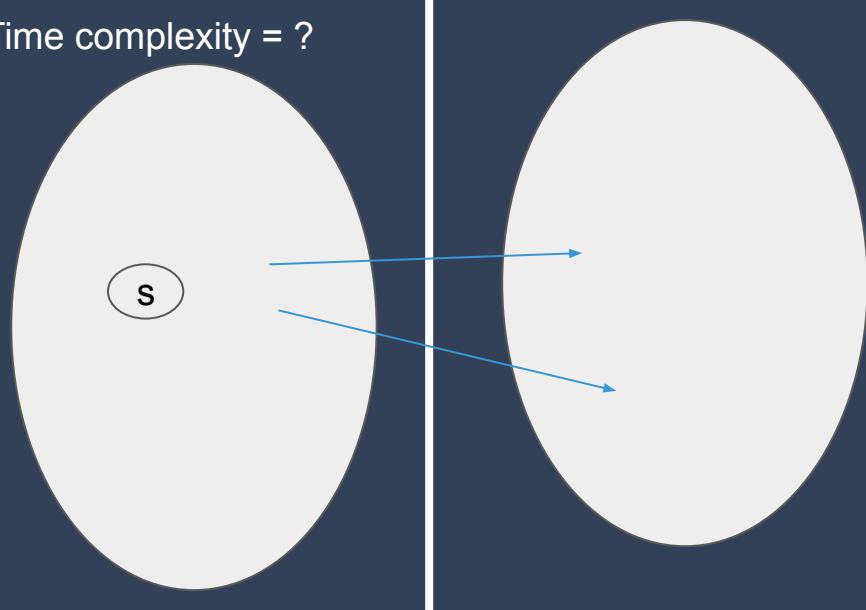
```
                    parent[w] = v;
```

```
}
```

```
}
```

# Dijkstra's algorithm

Time complexity = ?



Captured

{ik}

Yet to capture

INTERVIEW  
KICKSTART

```
class Graph {
```

# Dijkstra's algorithm

```
void Dijkstra(int s) {
```

```
    //distance, visited, captured and parent initialized to INF, 0, 0 and null
```

```
    distance[s] = 0; captured[s] = 1; visited[s] = 1
```

```
    q = new PQQueue(); q.push(s, priority=0);
```

```
    //Also push other vertices with priority=INF
```

```
    while not isEmpty(q): //capture the next vertex
```

```
        v = q.pop()
```

```
        captured[v] = 1
```

```
        for w in Adjlist[v]:
```

```
            if distance[v] + weight(v,w) < distance(w):
```

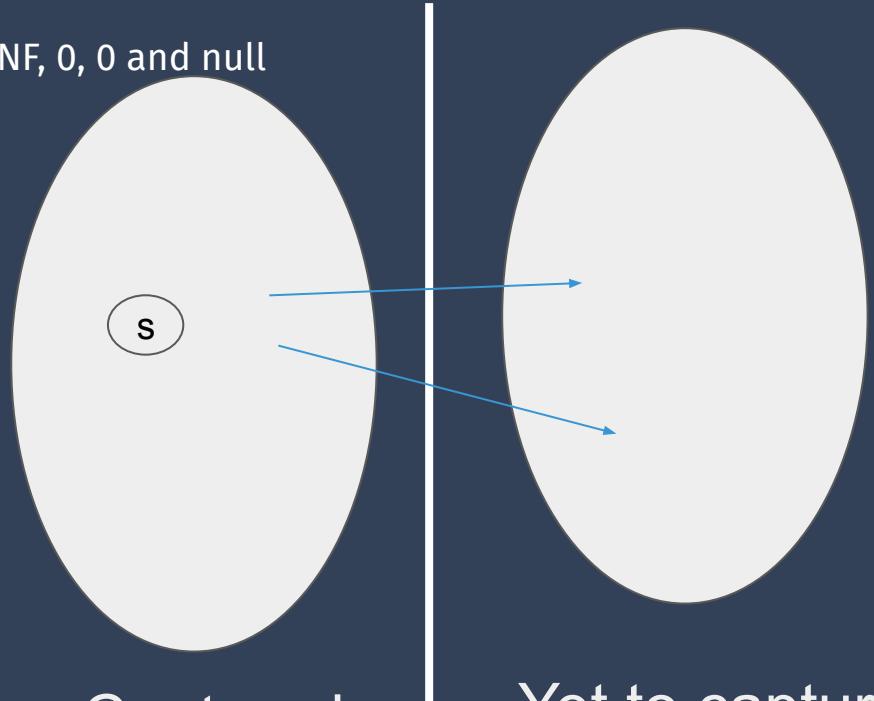
```
                distance[w] = distance[v] + weight(v,w)
```

```
                q.decreaseKey(w,priority=distance[w]);
```

```
                parent[w] = v;
```

```
}
```

```
}
```



Captured

{ik}

Yet to capture  
**INTERVIEW  
KICKSTART**

```
class Graph {
```

```
    void Dijkstra(int s) {
```

```
        //distance, visited, captured and parent initialized to
```

```
        INF, 0, 0 and null
```

```
        distance[s] = 0; captured[s] = 1; visited[s] = 1
```

```
        q = new PQueue(); q.push(s, priority=0);
```

```
        //Also push other vertices with priority=INF
```

```
        while not isEmpty(q): //capture the next vertex
```

```
            v = q.pop()
```

```
            captured[v] = 1
```

```
            for w in Adjlist[v]:
```

```
                if distance[v] + weight(v,w) < distance(w):
```

```
                    distance[w] = distance[v] + weight(v,w)
```

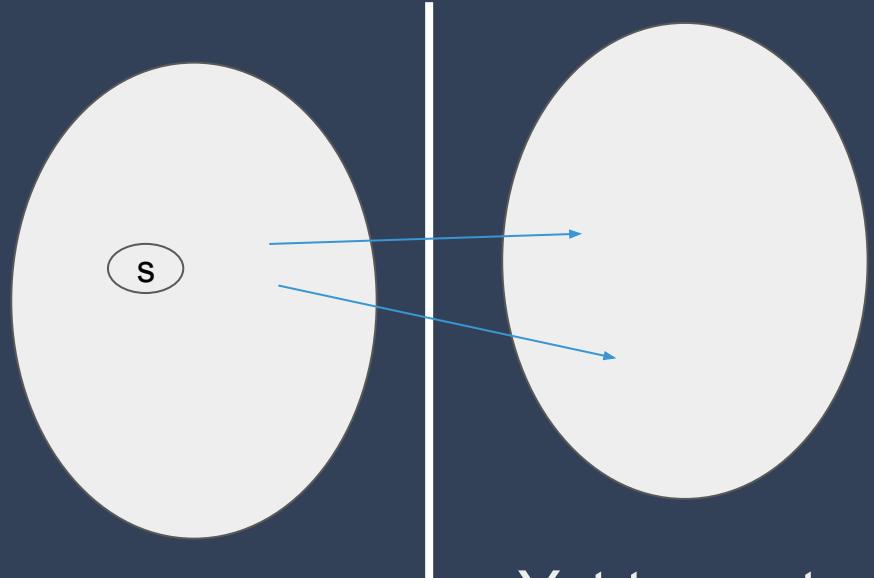
```
                    q.decreaseKey(w,priority=distance[w]);
```

```
                    parent[w] = v;
```

```
}
```

```
}
```

# Dijkstra's algorithm



Captured

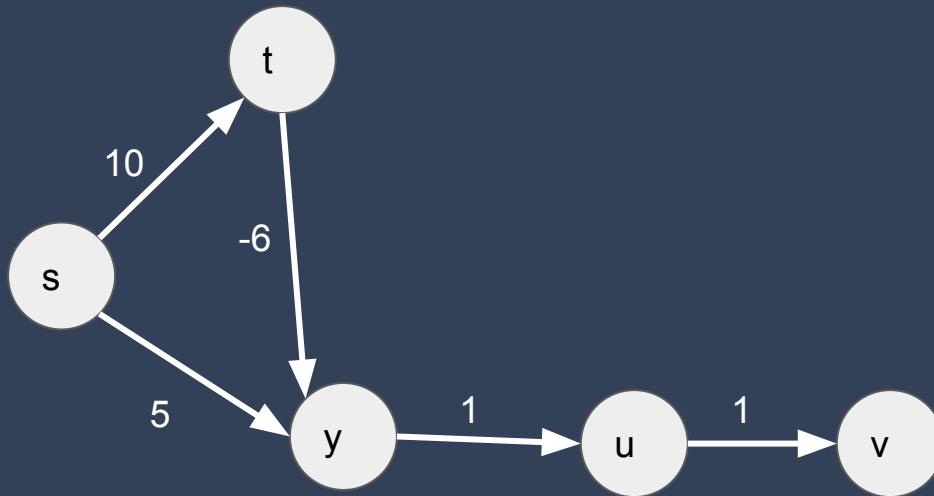
Yet to capture

Edge relaxation

{ik}

INTERVIEW  
KICKSTART

Time complexity = ?  
n extract-min + m  
decrease-key  
operations



**Can apply same algorithm to directed  
and undirected graphs.**

**If weights are allowed to be negative,  
then we may later discover a path back  
to a sure vertex following negative  
weights: problem!**

**Shortest paths are not defined if there are negative cycles.**

**Can have negative weights without negative cycles though.**

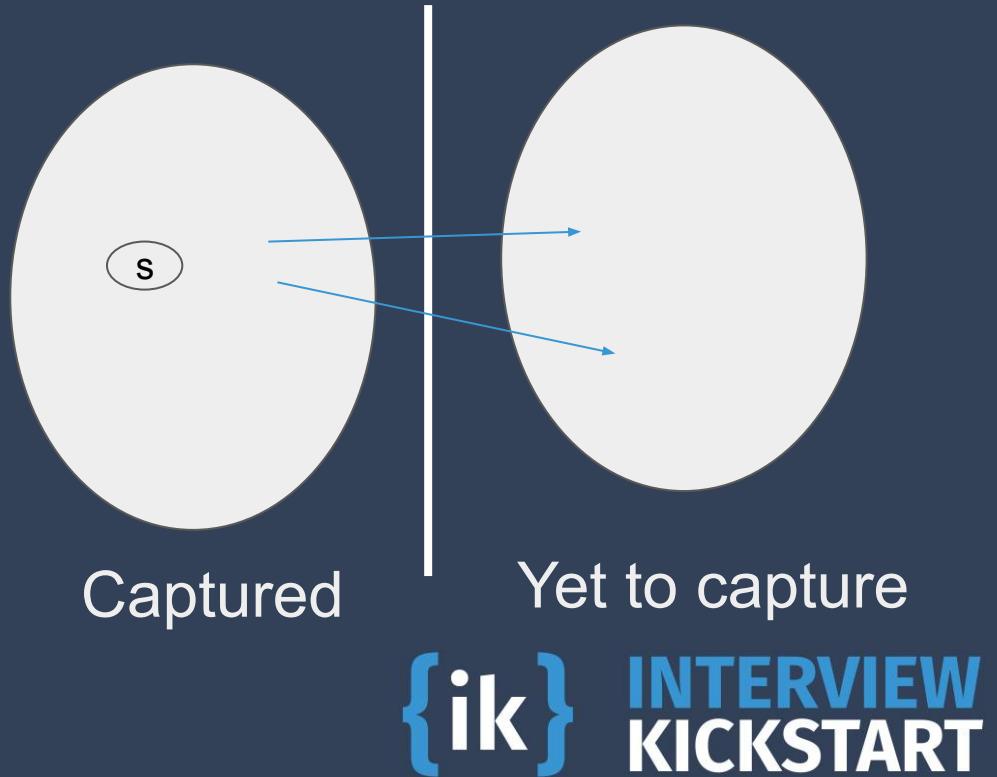
# Bellman-Ford

- For  $v \in V$ :
  - $d[v] = \infty$
- $d[s] = 0$
- For  $i = 1, \dots, n-1$ :
  - For each edge  $e = (u, v) \in E$ :
    - $d[v] \leftarrow \min(d[v], d[u] + \text{edgeWeight}(u, v))$

Time complexity = ?

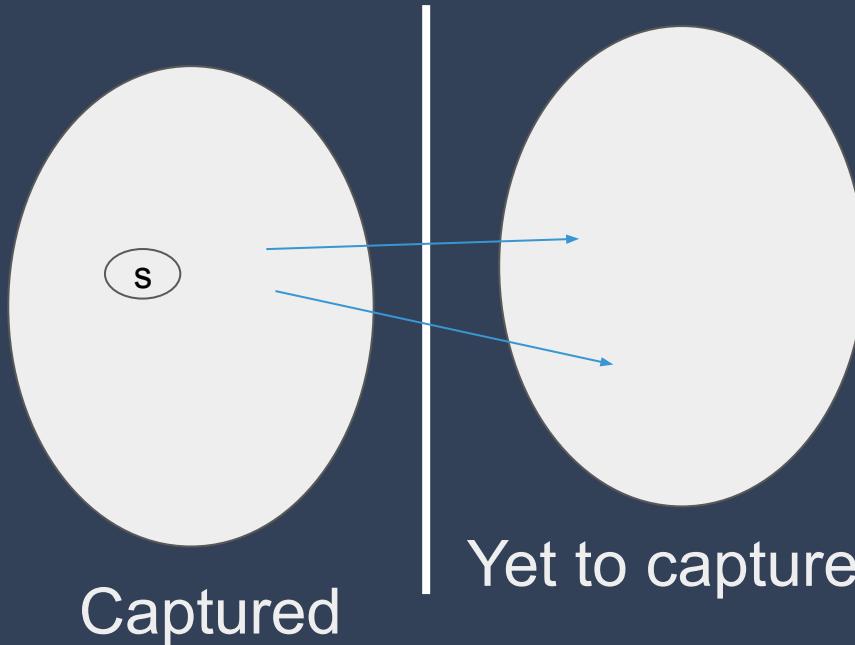
# Mystery Algorithm

```
class Graph {  
  
    void MysterySearch(int s) {  
        captured = new int[V]; parent = new int[V];  
        for i in 1 to V:  
            captured[i] = 0; parent[i] = null  
        captured[s] = 1  
  
        q = new Queue(); q.push(s);  
        while not isEmpty(q):  
            v = q.pop()  
            captured[v] = 1  
            for w in Adjlist[v]:  
                if visited[w] == 0 then  
                    visited[w] = 1; parent[w] = v;  
                    q.push(w);  
    }  
}
```



# Mystery Algorithm

```
class Graph {  
  
    void MysterySearch(int s) {  
        captured = new int[V]; parent = new int[V];  
        for i in 1 to V:  
            captured[i] = 0; parent[i] = null  
        captured[s] = 1  
  
        q = new Stack(); q.push(s);  
        while not isEmpty(q):  
            v = q.pop()  
            captured[v] = 1  
            for w in Adjlist[v]:  
                if visited[w] == 0 then  
                    visited[w] = 1; parent[w] = v;  
                    q.push(w);  
    }  
}
```

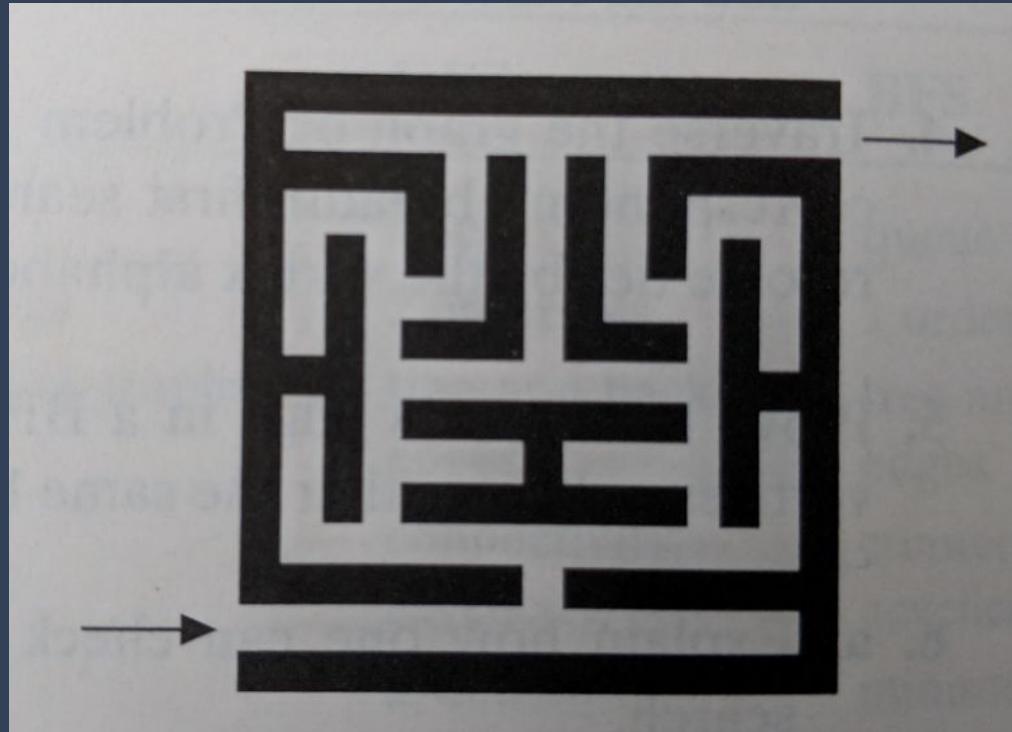


## BFS & DFS

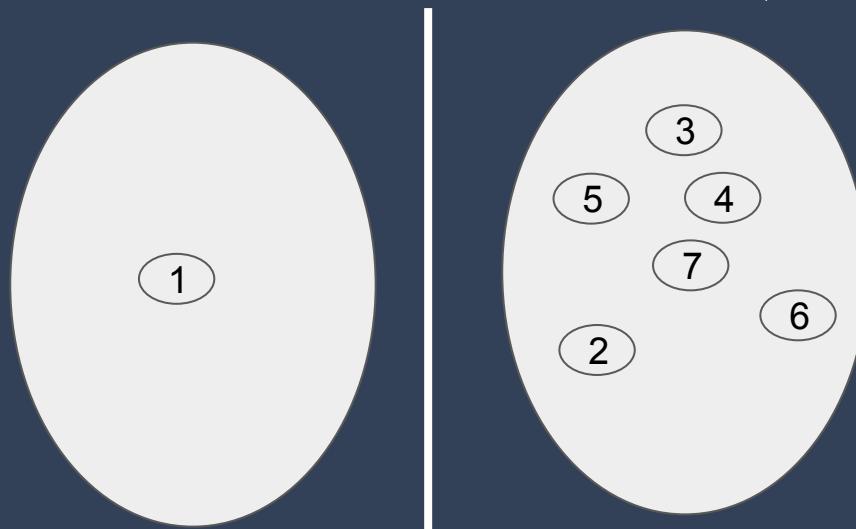
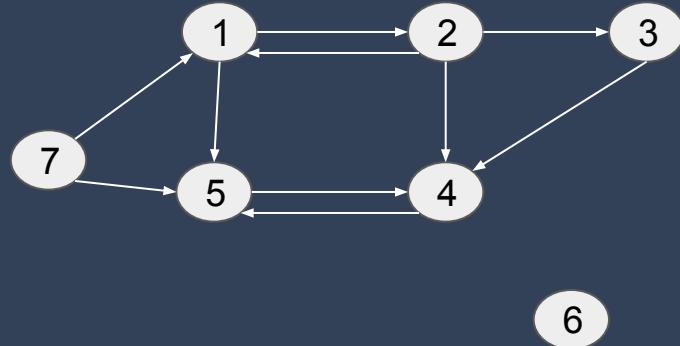
BFS policy: Among all fringe edges  $(u,v)$ , pull the one which was examined **first**.

DFS policy: Among all fringe edges  $(u,v)$ , pull the one which was examined **last (i.e, most recently)**.

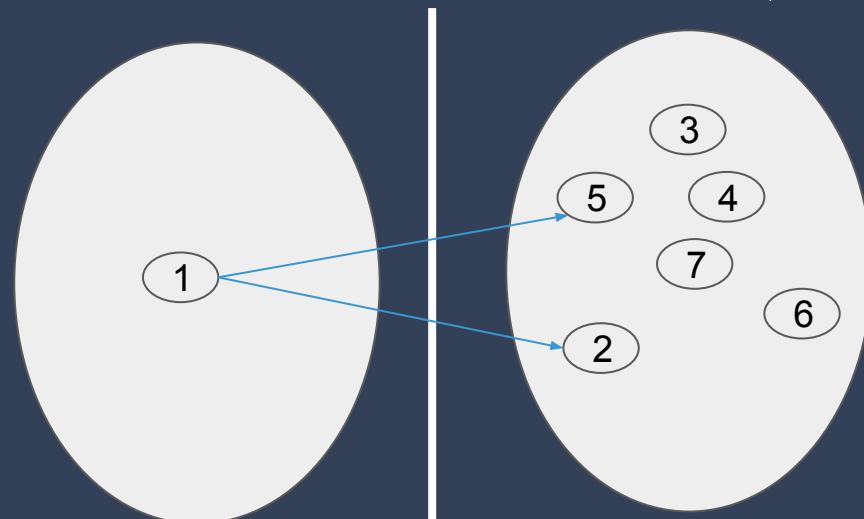
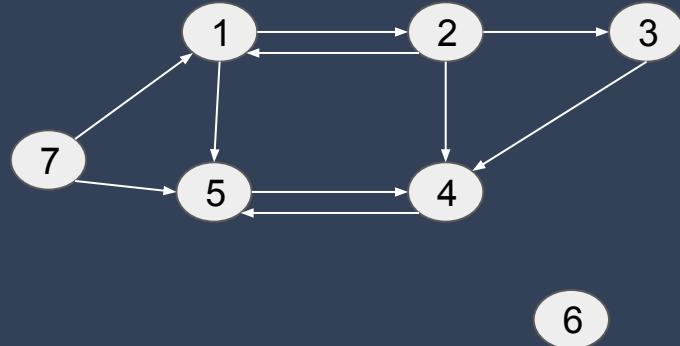
Which traversal would you use to exit this maze?



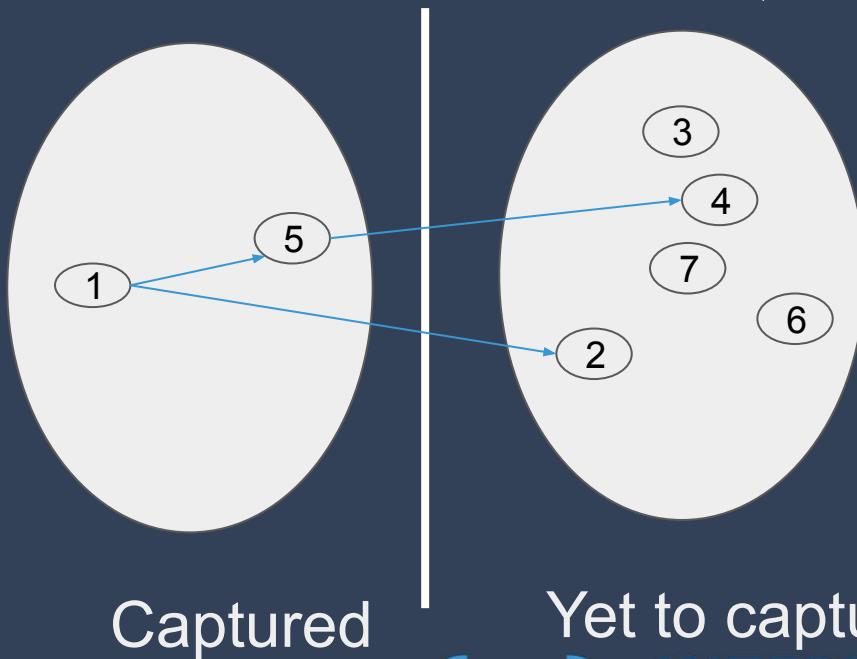
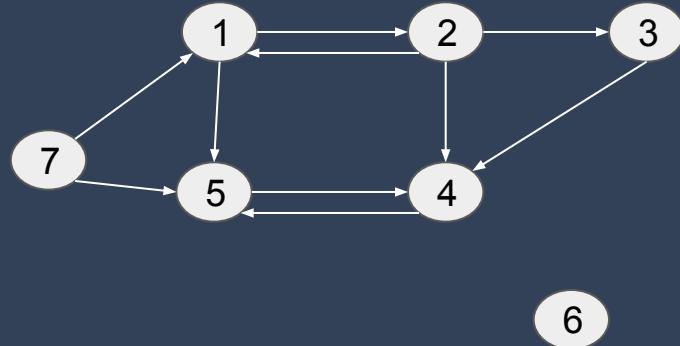
# DFS (iterative stack)



# DFS (iterative stack)



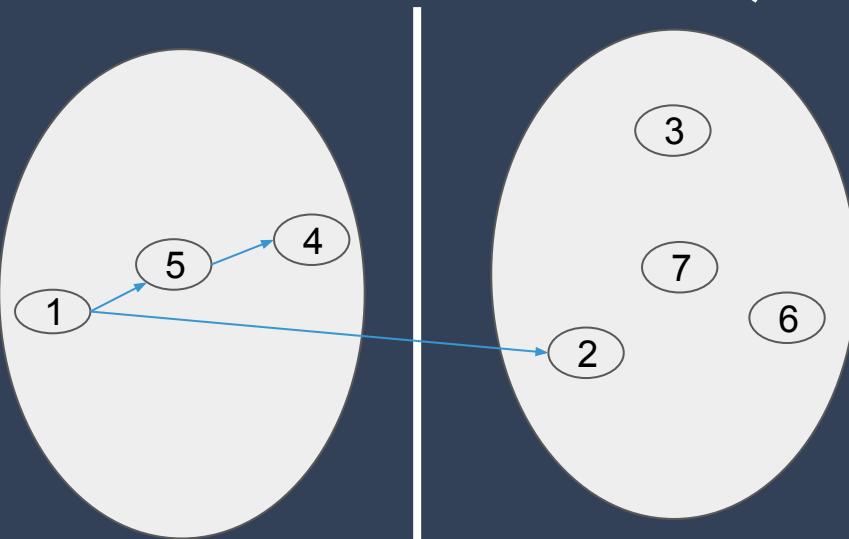
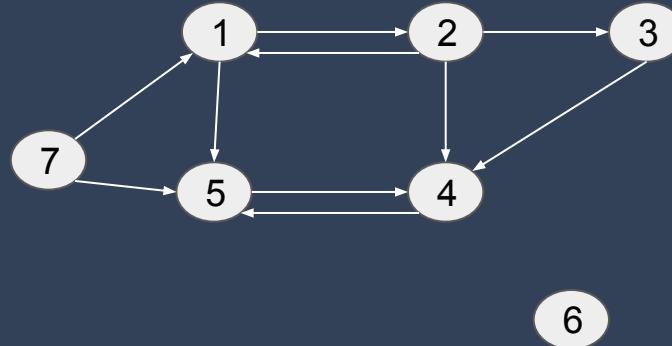
# DFS (iterative stack)



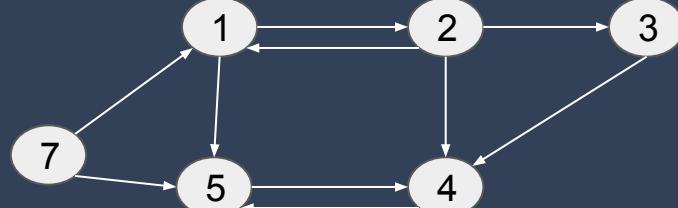
Captured

Yet to capture

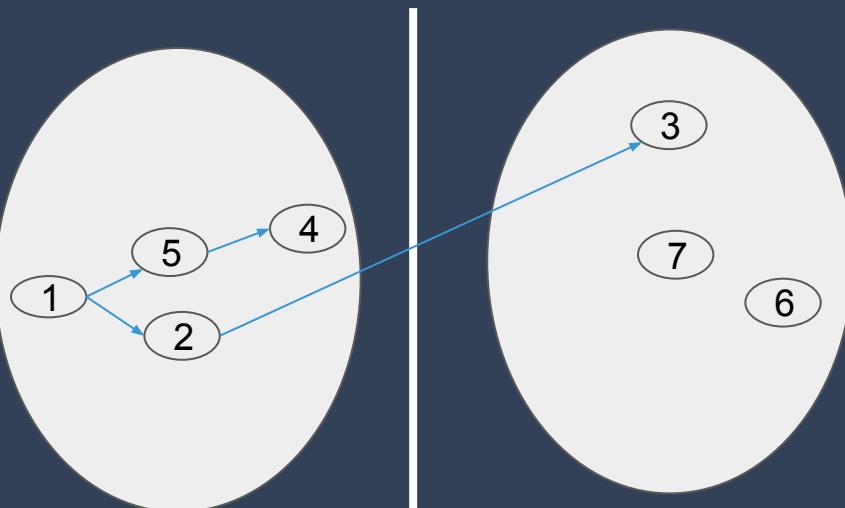
# DFS (iterative stack)



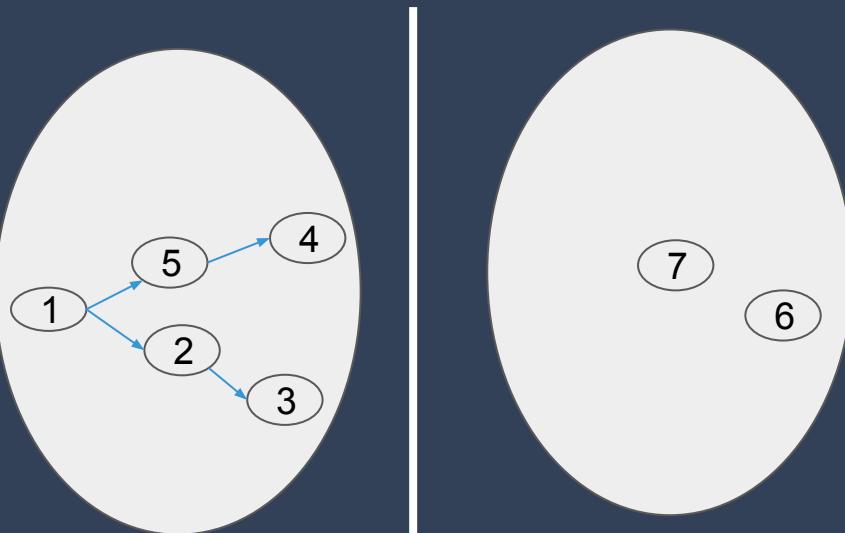
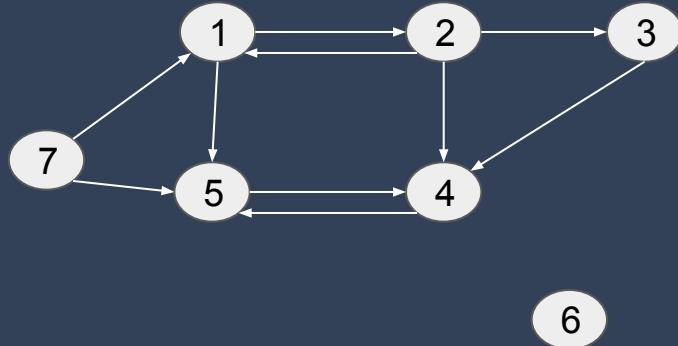
# DFS (iterative stack)



6

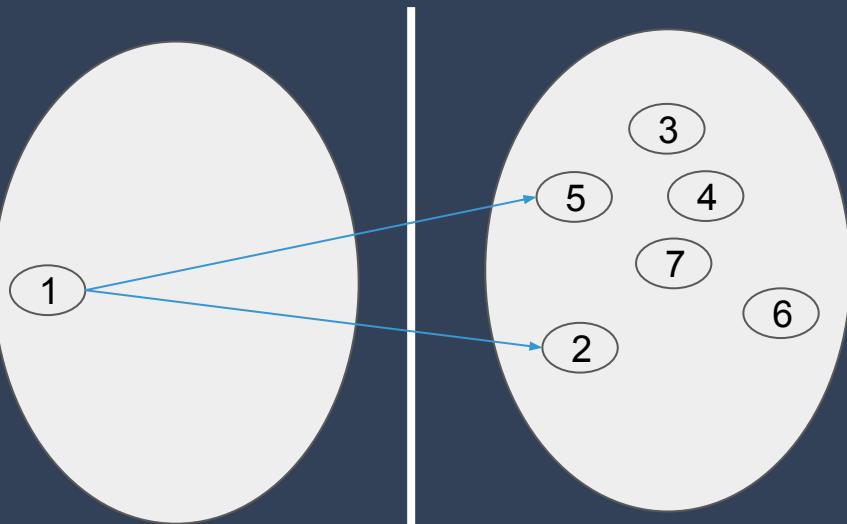
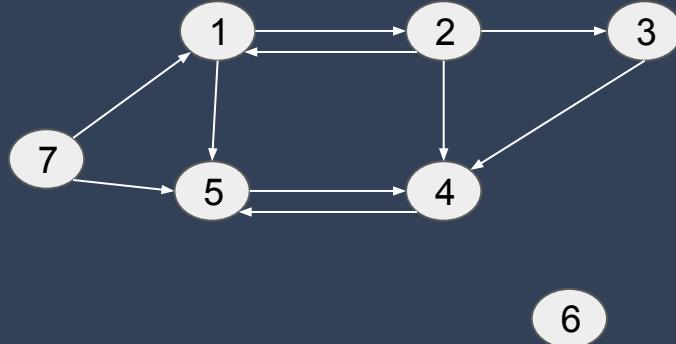


# DFS (iterative stack)



```
class Graph {  
  
    void DFS(int source) {  
        visited[source] = 1 //visited and captured mean the same thing in recursive DFS  
        for w in adjList[source]:  
            if visited[w] == 0:  
                DFS(w)  
    }  
}
```

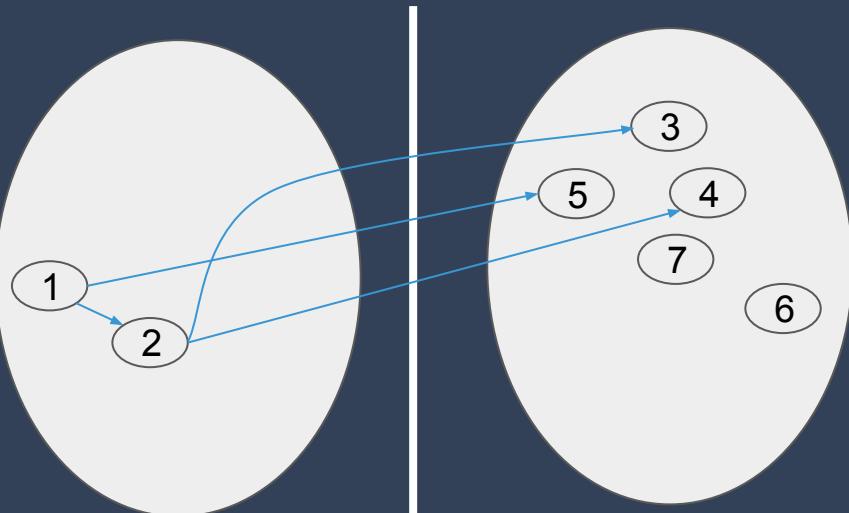
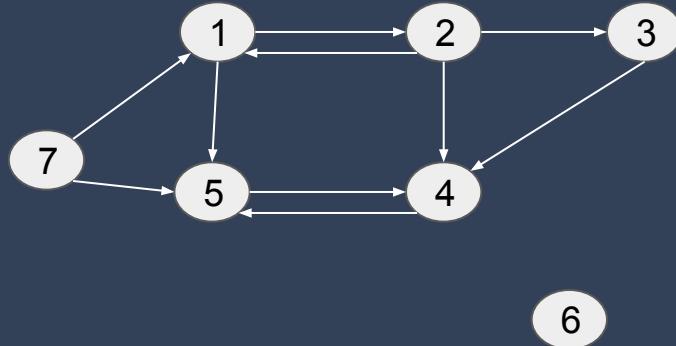
# DFS (recursive)



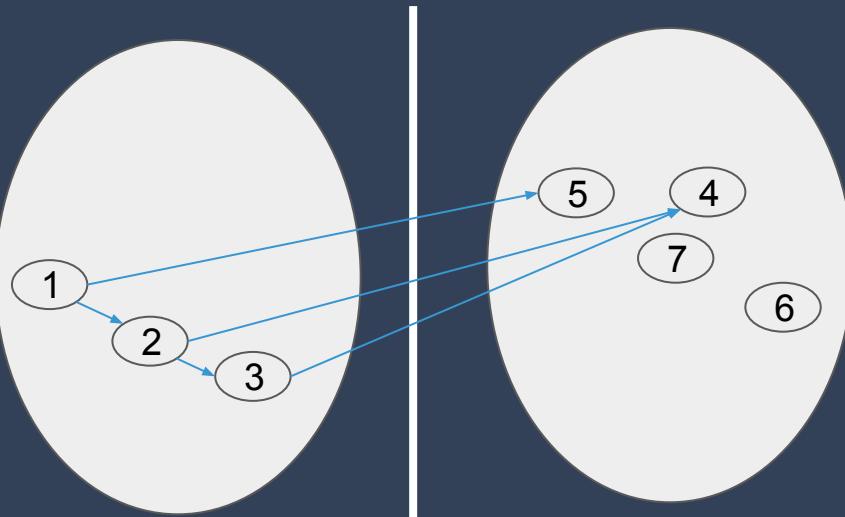
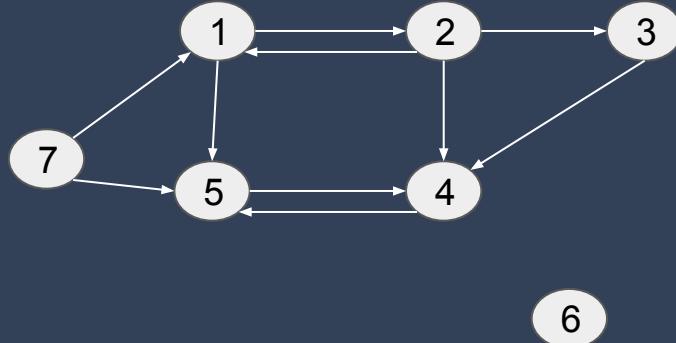
Captured

Yet to capture

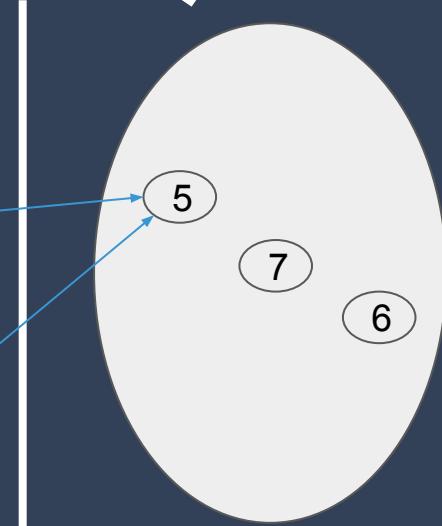
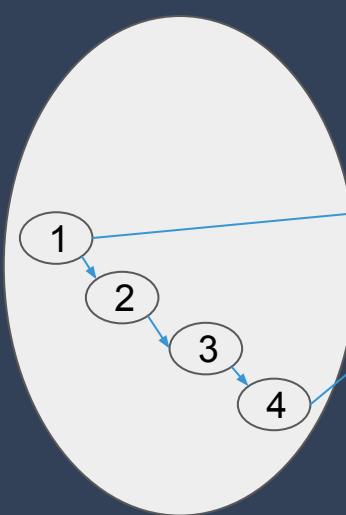
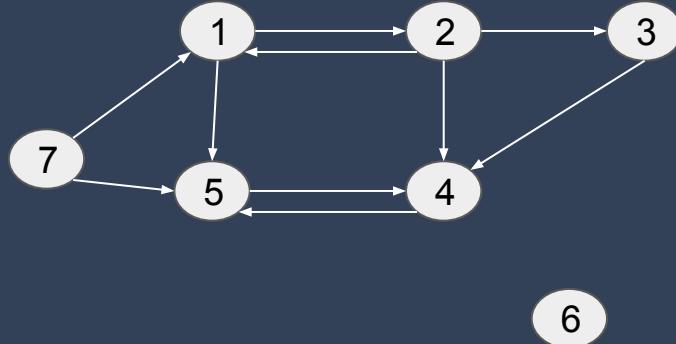
# DFS (recursive)



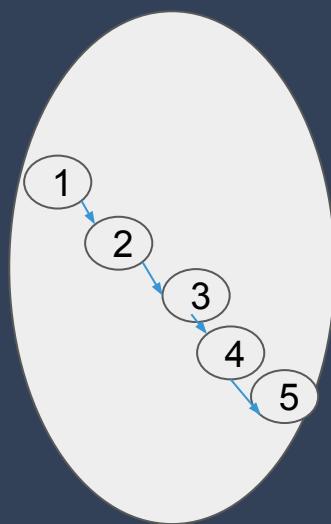
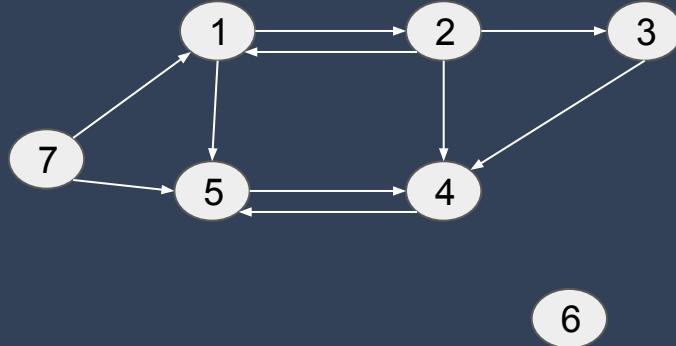
# DFS (recursive)



# DFS (recursive)



# DFS (recursive)



**Given n nodes labeled from 0 to n - 1 and a list of undirected edges (each edge is a pair of nodes), write a function to find the number of connected components in an undirected graph.**

```
class Graph {  
  
    void DFS(int source, int c) {  
        visited[source] = c  
        for w in adjList[source]:  
            if visited[w] == 0:  
                DFS(w, c)  
    }  
  
}
```

```
void findComponents() {  
    component = 0  
    for i in 1 to V:  
        if visited[i] == 0:  
            component++  
            DFS(i, component)  
  
    return component  
}  
}
```

# Curriculum Designed by:

Omkar Deshpande

<http://ai.stanford.edu/~omkard>