

Graphs Review and BFS

CS/MCS 401: Computer Algorithms I

Lecture 9

Professor: Jonathan Liu

Introduce yourself!

Icebreaker: Do you have a favorite graph?

Lesson Goals

This is the start of Unit 3: Graph Algorithms!

The main goal today is to review the graph terminology that you've seen before (probably in 251), and the two major graph traversals (DFS and BFS).

Then, we'll see a first example of the main takeaway from this unit: **the standard graph algorithms are really useful building blocks for new algorithms.**

Announcement

Office Hours have been extended:
Wednesdays 1-3pm, Thursdays
3:30-5:30pm.

Graph Vocabulary Review

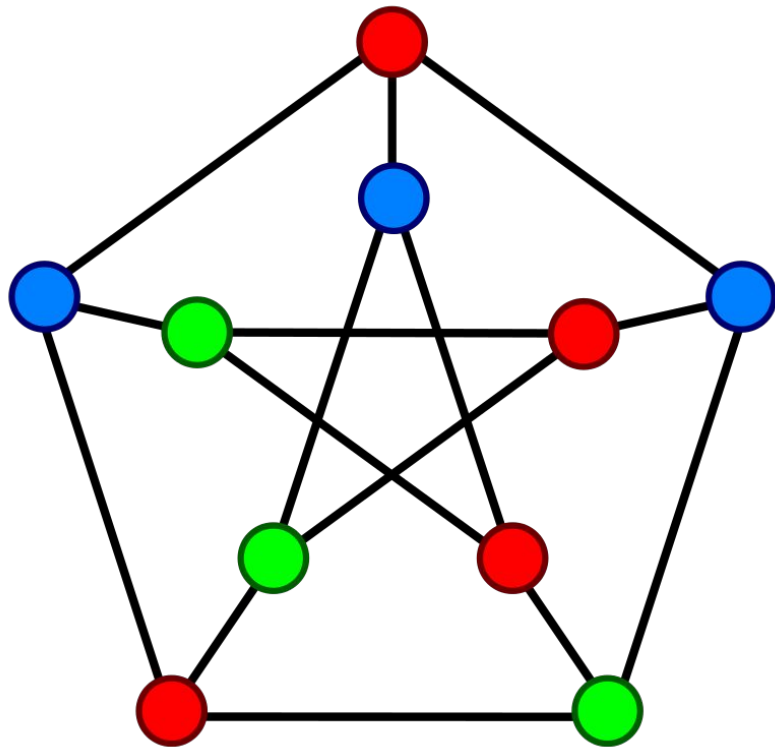


Graphs

A graph is a set of vertices and a set of edges that connect them. It is represented by $G=(V, E)$, where V is a set of *vertices*, and E is a set of *edges*.

Each edge is represented as a pair of vertices: (u, v) .

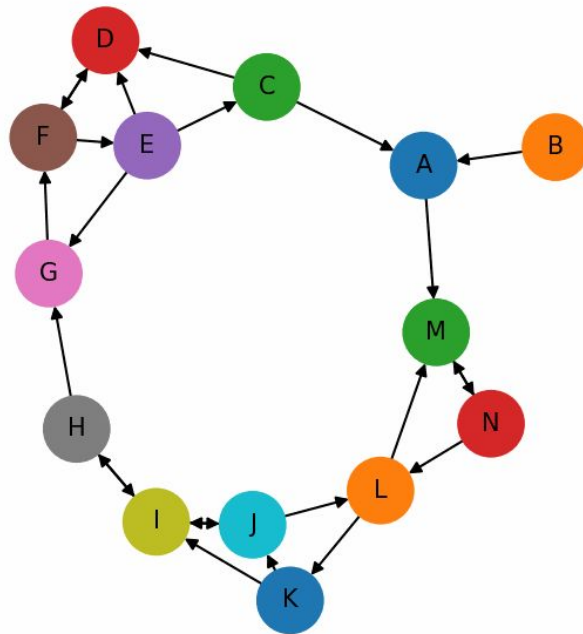
A vertex's **degree** is the number of edges the vertex has.



Graph Size

The graphs in this class might be *finite*, but you should remember that in computer science they can still be quite large.

PageRank is run on a graph of the internet. If each vertex is a webpage, that gives us something like 10 billion vertices.

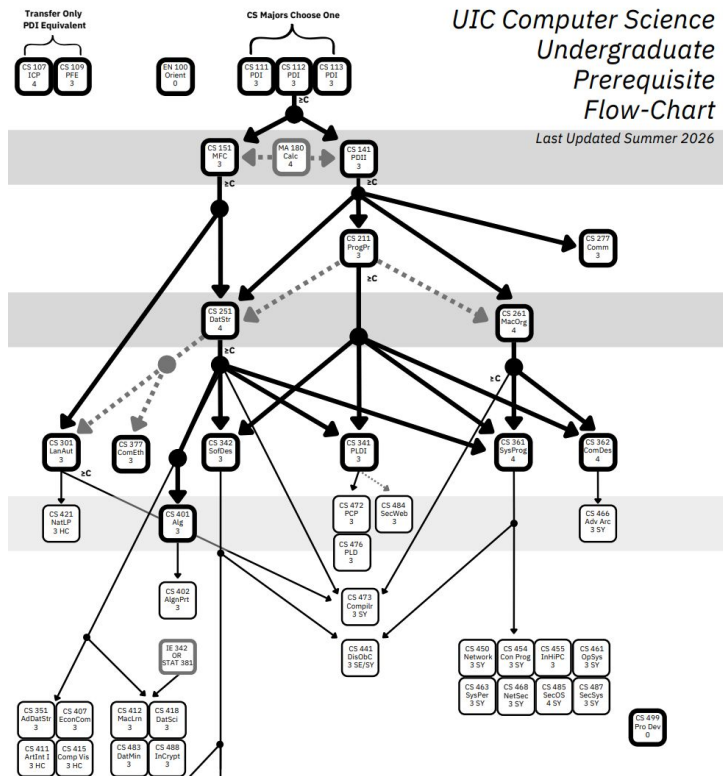


Directedness

Graphs can be **directed** or **undirected**.
In a **directed graph**, each edge (u, v) points from u to v .

This is used to represent one-way relationships, like prerequisites among a set of courses.

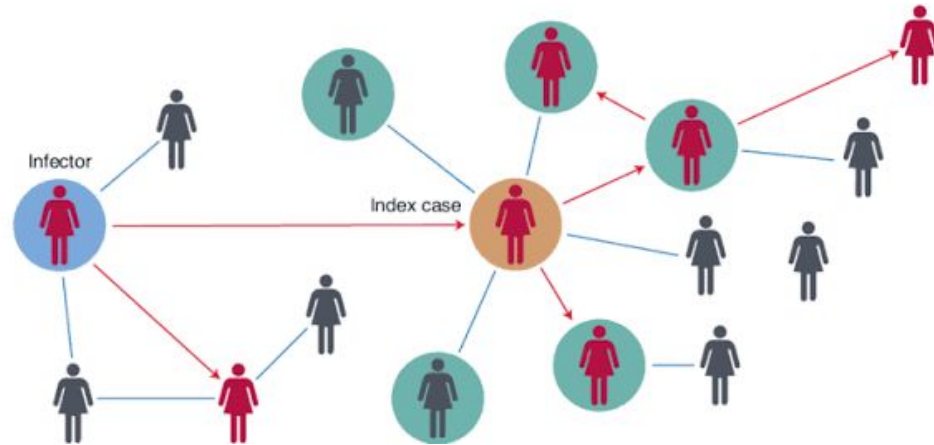
In this course, graphs are **undirected** unless otherwise specified.



Connectedness

An *undirected* graph is **connected** if there is a path from every node to every other node.

A *directed* graph is **strongly connected** if there is a path from every node to every other node, and **weakly connected** if the undirected version is connected.

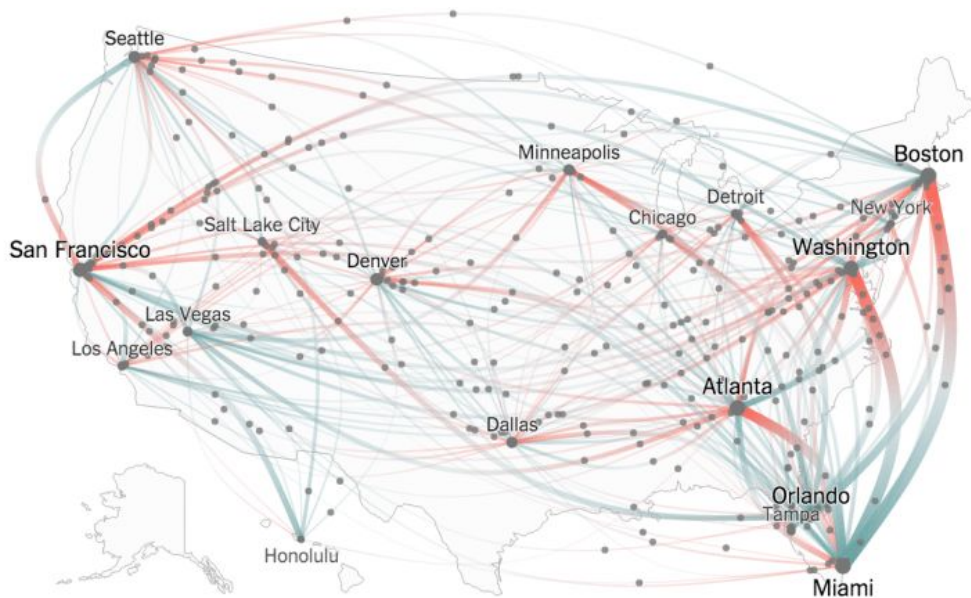


Walks, Paths, Cycles

A **walk** is any sequence of vertices $v_1, v_2, \dots, v_n$ where any two consecutive vertices must be adjacent on the graph.

A **path** is a walk that does not revisit any vertices.

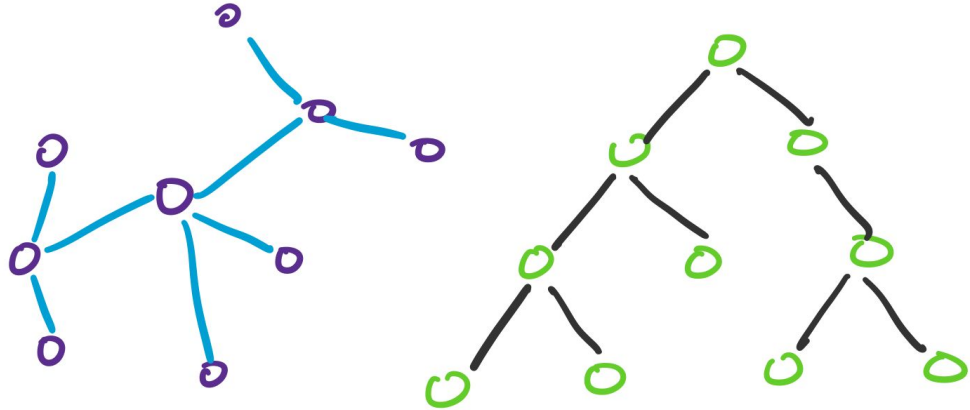
A **cycle** is a walk that starts and ends at the same vertex but does not revisit any other vertices or repeat any edges.



Trees

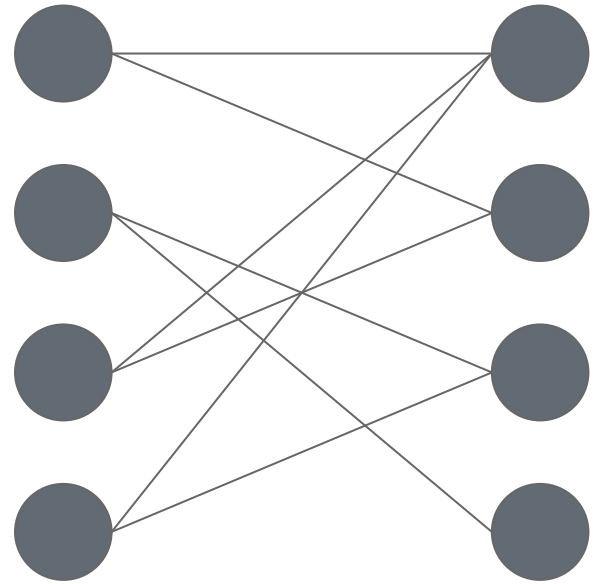
A **tree** is a **connected acyclic graph**.

Unlike in 251, trees **do not** have to contain roots. They can though – these are called **rooted trees**.



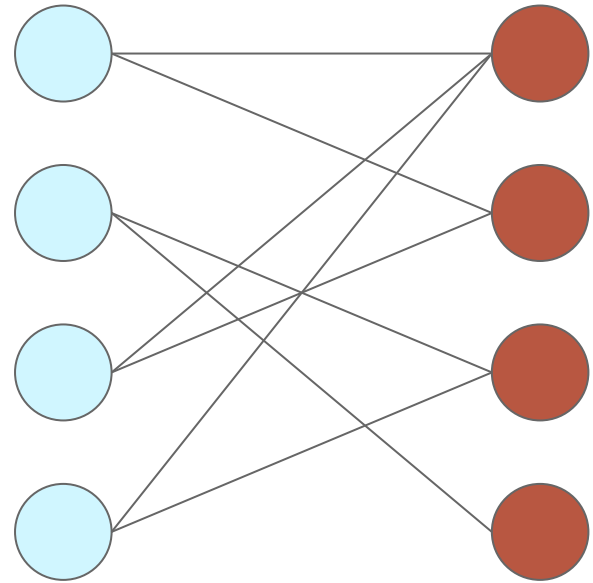
Bipartite

A **bipartite** graph is a graph whose vertices can be split into two groups, V_1 and V_2 , such that **all edges go between V_1 and V_2** .



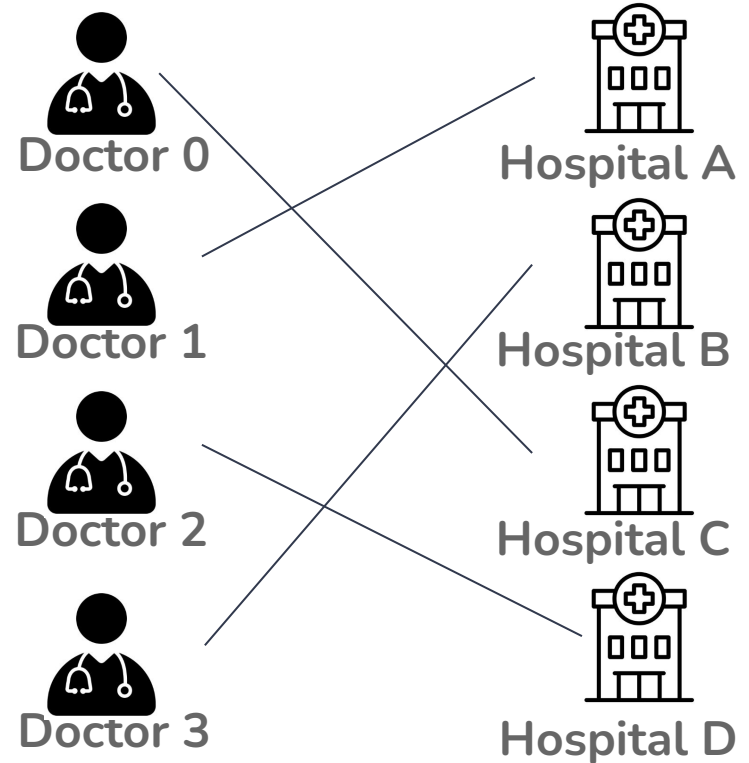
Bipartite

A **bipartite** graph is a graph whose vertices can be split into two groups, V_1 and V_2 , such that **all edges go between V_1 and V_2** .



Bipartite

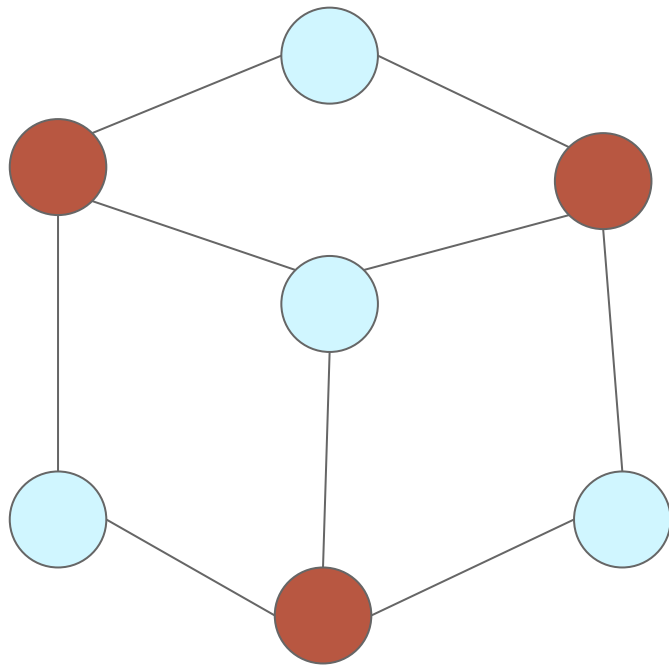
A **bipartite** graph is a graph whose vertices can be split into two groups, V_1 and V_2 , such that **all edges go between V_1 and V_2** .



Bipartite

A **bipartite** graph is a graph whose vertices can be split into two groups, V_1 and V_2 , such that **all edges go between V_1 and V_2** .

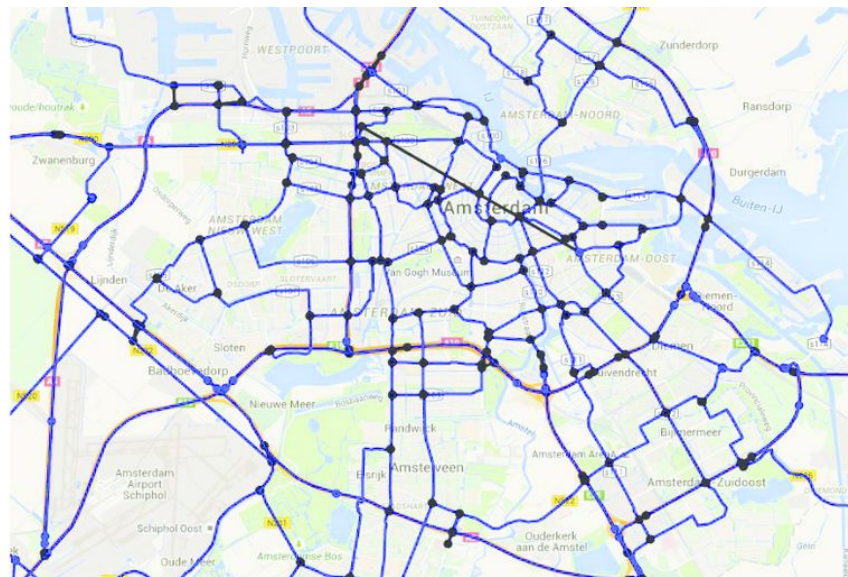
This is often also referred to as **2-colorable**, which means you can color the vertices using two colors in a way that no edge has the same color on both sides.



Default Terminology

In this course, unless otherwise specified, you may assume that graphs are:

- **Simple:** No edge goes from a vertex to itself, and no pair of vertices has more than one edge.
- **Finite:** There are a finite number of vertices and edges.
- **Undirected**



Worksheet: Terminology Review

1. In any graph, the number of vertices of odd degree must be even.

TRUE!

Each edge adds two degrees, so the sum of all degrees must be even. That's not possible if the number of vertices of odd degree is odd.

2. If there is a walk from u to v , there is a path from u to v .

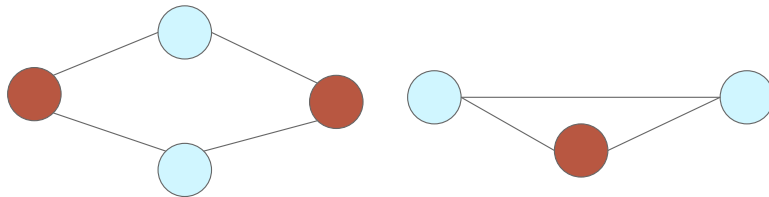
TRUE!

Just skip any cycles.

Worksheet: Terminology Review

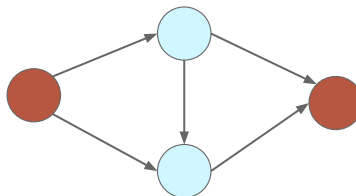
3. If every vertex in a graph has degree 2, it is a cycle.

FALSE!



4. If a directed graph has no cycles, it can not contain more edges than vertices.

FALSE!



Worksheet: Terminology Review

5. In a directed graph, the smallest cycle has length 3.

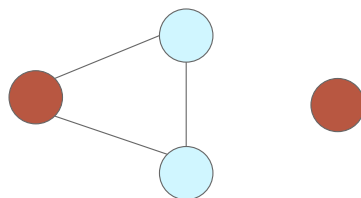
FALSE!

You can make a cycle with $u \rightarrow v$ and $v \rightarrow u$.

6. A graph with n vertices and $n - 1$ edges is connected.

FALSE!

Those edges could go anywhere.



Worksheet: Terminology Review

7. Adding a new edge to a tree creates exactly one cycle.

TRUE!

Say we added the edge (u, v) and it created two cycles. Then, merging those cycles at (u, v) makes another larger cycle that doesn't include (u, v) .

8. There can only be one path between any two vertices on a tree.

TRUE!

Otherwise, take the last point where the two paths meet. Afterwards, you have two non-overlapping paths from there to the end, which makes a cycle!

Worksheet: Terminology Review

9. A graph with no edges is bipartite.

TRUE!

No matter how you split the vertices, there are no edges between vertices of the same color.

10. All trees are bipartite.

TRUE!

We'll see why later today :)

How should we store the set of edges?



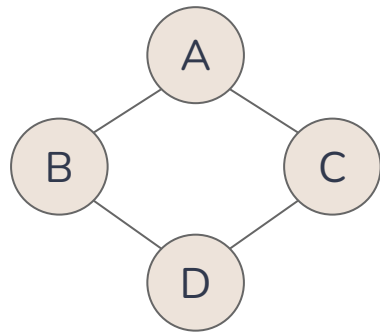
Adjacency Matrix

Create a matrix A , where $A[u][v] = 1$ if (u, v) is an edge, and 0 if it's not.

Runtimes:

- Adding/Removing an edge: $O(1)$
- Check if (u, v) exists: $O(1)$
- Get neighbors of v : $O(n)$

Space: $O(n^2)$



	A	B	C	D
A	0	1	1	0
B	1	0	0	1
C	1	0	0	1
D	0	1	1	0

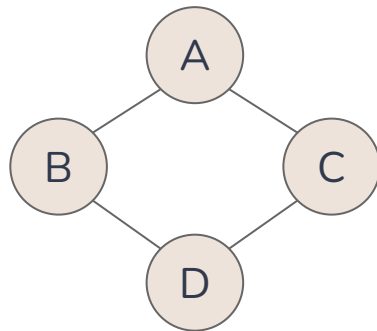
Adjacency List

For each vertex, store a list of all vertices that it is adjacent to.

If $n=|V|$ and $m=|E|$:

- Adding/Removing an edge: $O(1)$
- Check if (u, v) exists: $O(\deg(u))$
- Get neighbors of v : $O(\deg(v))$

Space: $O(n + m)$



A	→	[B, C]
B	→	[A, C]
C	→	[A, D]
D	→	[B, C]

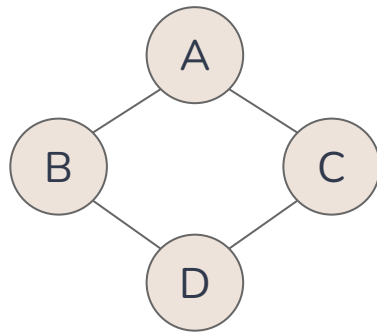
Adjacency List

For each vertex, store a list of all vertices that it is adjacent to.

If $n=|V|$ and $m=|E|$:

- Adding/Removing an edge: $O(1)$
- Check if (u, v) exists: $O(1)$
- Get neighbors of v : $O(\deg(v))$

Space: $O(n + m)$



A	→	[B, C]
B	→	[A, D]
C	→	[A, D]
D	→	[B, C]

We can also use hash tables for these, so edge lookup becomes $O(1)$.

Adjacency List

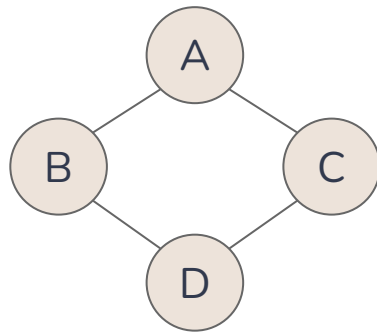
For each vertex, store a list of all vertices that it is adjacent to.

If $n=|V|$ and $m=|E|$:

- Adding/Removing an edge: $O(1)$
- Check if (u, v) exists: $O(1)$
- Get neighbors of v : $O(\deg(v))$

Space: $O(n + m)$

In this class, you may assume every graph is implemented this way.



A	→	[B, C]
B	→	[A, D]
C	→	[A, D]
D	→	[B, C]

We can also use hash tables for these, so edge lookup becomes $O(1)$.

Breadth First Search Review



Breadth First Search (BFS)

BFS($G=(V,E)$):

 toVisit.enqueue(an unvisited vertex)

while toVisit is not empty:

 current $\leftarrow$ toVisit.dequeue()

 mark current as visited

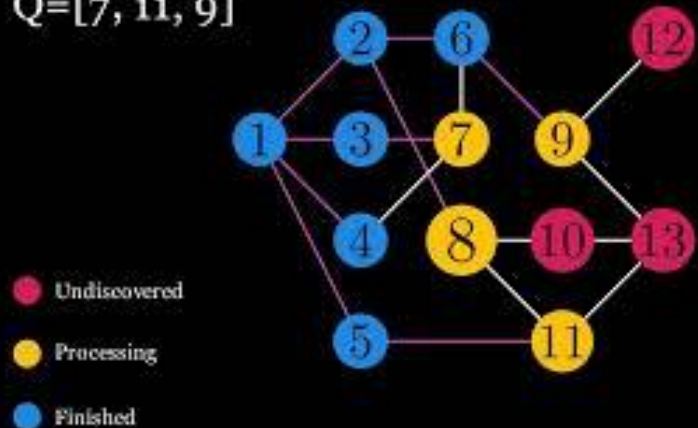
for each edge (current, v):

if v is not visited:

 toVisit.enqueue(v)

Caution: we're assuming here that the graph is connected.

$Q=[7, 11, 9]$



BFS Runtime

```
BFS(G=(V,E)):
  toVisit.enqueue(an unvisited vertex)
  while toVisit is not empty:
    current ← toVisit.dequeue()
    mark current as visited
    for each edge (current, v):
      if v is not visited:
        toVisit.enqueue(v)
```

Remember, by definition there are n vertices and m edges.

Goes through all n vertices.

Can go through up to m edges.

$O(nm)$??

BFS Runtime

```
BFS(G=(V,E)):
  toVisit.enqueue(an unvisited vertex)
  while toVisit is not empty:
    current ← toVisit.dequeue()
    mark current as visited
    for each edge (current, v):
      if v is not visited:
        toVisit.enqueue(v)
```

Remember, by definition there are n vertices and m edges.

Goes through all n vertices.

Can go through up to m edges.

$O(nm)$??

BFS Runtime

```
BFS(G=(V,E)):
  toVisit.enqueue(an unvisited vertex)
  while toVisit is not empty:
    current ← toVisit.dequeue()
    mark current as visited
    for each edge (current, v):
      if v is not visited:
        toVisit.enqueue(v)
```

Remember, by definition there are n vertices and m edges.

Goes through all n vertices.

Across all loops, sees each edge only twice. $O(m)$.

$O(n+m)$

A First Application: Shortest Paths



BFS can be used to find distances from a vertex!

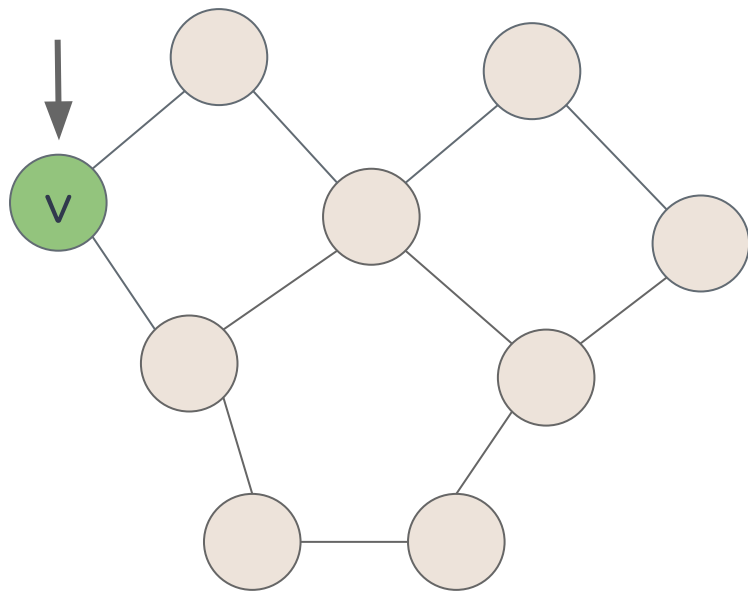
Original BFS

```
BFS(G=(V,E)):
  toVisit.enqueue(an unvisited vertex)
  while toVisit is not empty:
    current ← toVisit.dequeue()
    mark current as visited
    for each edge (current, v):
      if v is not visited:
        toVisit.enqueue(v)
```

Keeping Track of Distance

```
BFS(G=(V,E), starting vertex v):
  toVisit.enqueue(v)
  set d(v) = 0
  set all other distances to -1
  while toVisit is not empty:
    current ← toVisit.dequeue()
    for each edge (current, w):
      if d(w) ≠ -1:
        d(w) = d(current) + 1
        toVisit.enqueue(w)
```

BFS can be used to find distances from a vertex!



Keeping Track of Distance

BFS($G=(V,E)$, **starting vertex v**):

toVisit.enqueue(**v**)

set $d(v) = 0$

set all other distances to -1

while toVisit is not empty:

 current $\leftarrow$ toVisit.dequeue()

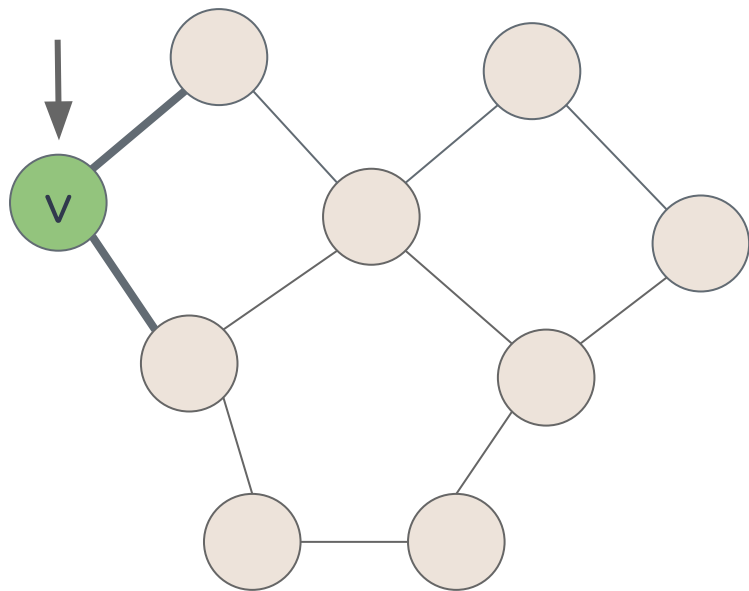
 for each edge (current, w):

 if **$d(w) \neq -1$** :

$d(w) = d(\text{current}) + 1$

 toVisit.enqueue(w)

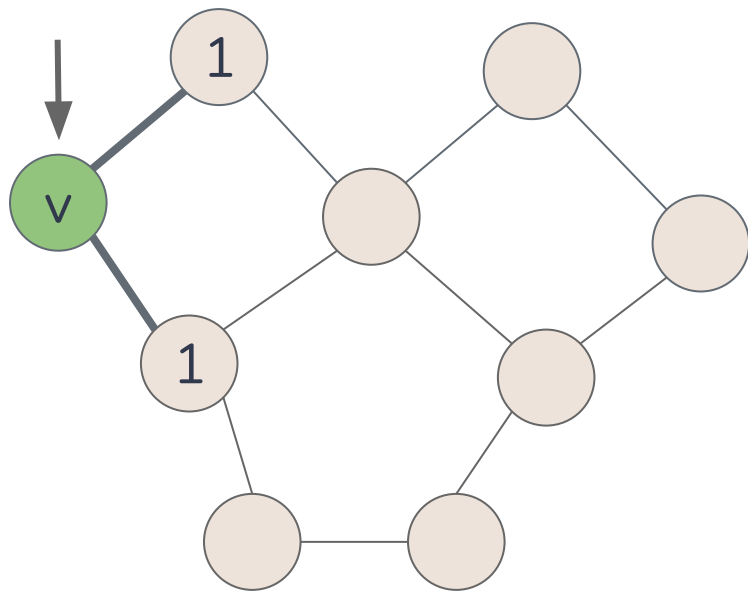
BFS can be used to find distances from a vertex!



Keeping Track of Distance

```
BFS( $G=(V,E)$ , starting vertex  $v$ ):  
  toVisit.enqueue( $v$ )  
  set  $d(v) = 0$   
  set all other distances to -1  
  while toVisit is not empty:  
    current  $\leftarrow$  toVisit.dequeue()  
    for each edge (current,  $w$ ):  
      if  $d(w) \neq -1$ :  
         $d(w) = d(\text{current}) + 1$   
        toVisit.enqueue( $w$ )
```

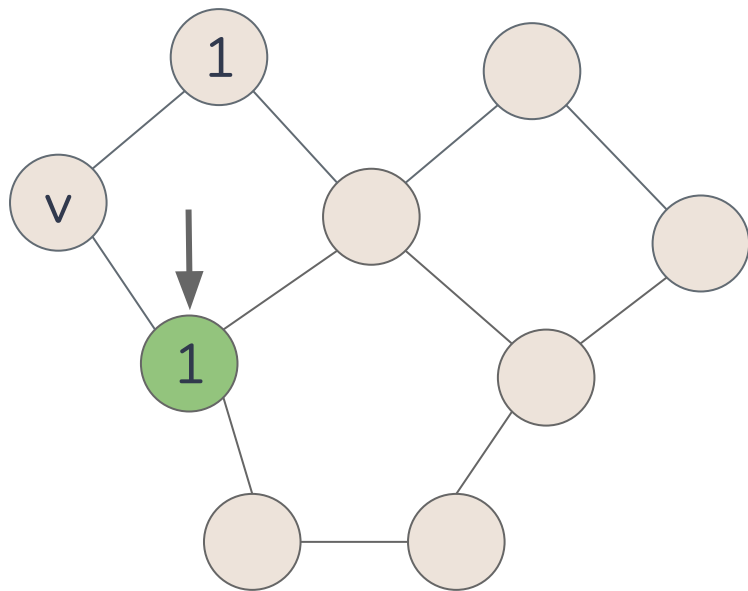
BFS can be used to find distances from a vertex!



Keeping Track of Distance

```
BFS( $G=(V,E)$ , starting vertex  $v$ ):  
  toVisit.enqueue( $v$ )  
  set  $d(v) = 0$   
  set all other distances to -1  
  while toVisit is not empty:  
    current  $\leftarrow$  toVisit.dequeue()  
    for each edge (current,  $w$ ):  
      if  $d(w) \neq -1$ :  
         $d(w) = d(\text{current}) + 1$   
        toVisit.enqueue( $w$ )
```

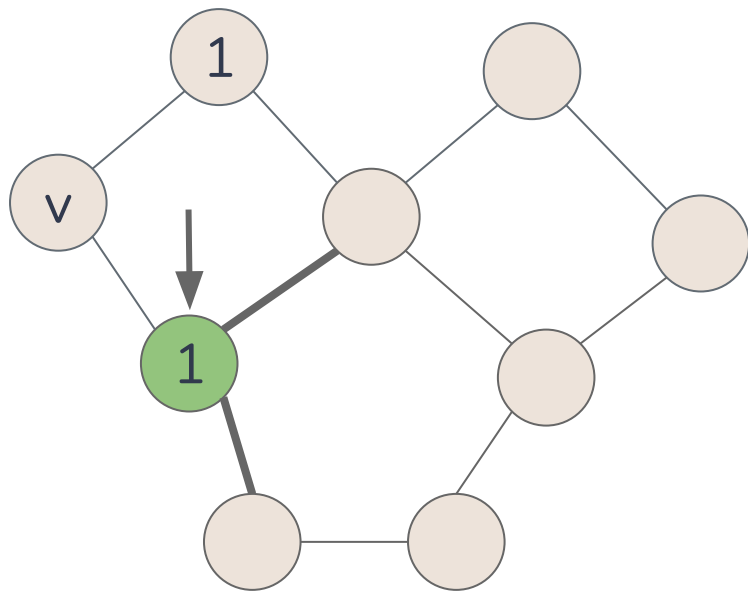
BFS can be used to find distances from a vertex!



Keeping Track of Distance

```
BFS( $G=(V,E)$ , starting vertex  $v$ ):  
  toVisit.enqueue( $v$ )  
  set  $d(v) = 0$   
  set all other distances to -1  
  while toVisit is not empty:  
    current  $\leftarrow$  toVisit.dequeue()  
    for each edge (current,  $w$ ):  
      if  $d(w) \neq -1$ :  
         $d(w) = d(\text{current}) + 1$   
        toVisit.enqueue( $w$ )
```

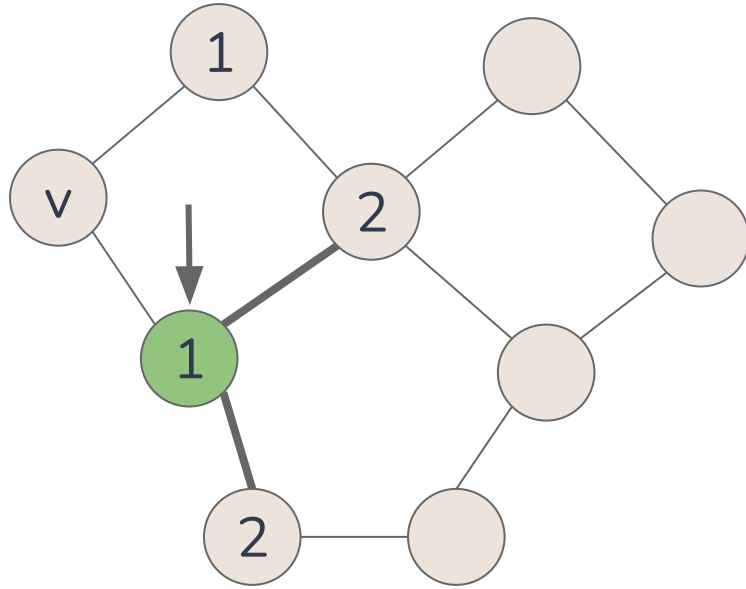
BFS can be used to find distances from a vertex!



Keeping Track of Distance

```
BFS( $G=(V,E)$ , starting vertex  $v$ ):  
  toVisit.enqueue( $v$ )  
  set  $d(v) = 0$   
  set all other distances to -1  
  while toVisit is not empty:  
    current  $\leftarrow$  toVisit.dequeue()  
    for each edge (current,  $w$ ):  
      if  $d(w) \neq -1$ :  
         $d(w) = d(\text{current}) + 1$   
        toVisit.enqueue( $w$ )
```

BFS can be used to find distances from a vertex!



Keeping Track of Distance

BFS($G=(V,E)$, **starting vertex v**):

toVisit.enqueue(**v**)

set $d(v) = 0$

set all other distances to -1

while toVisit is not empty:

 current $\leftarrow$ toVisit.dequeue()

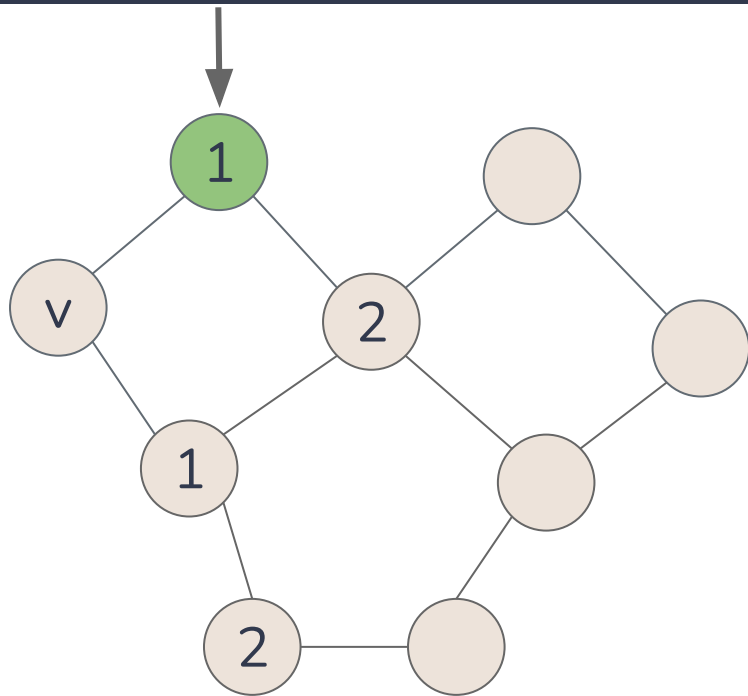
 for each edge (current, w):

 if $d(w) \neq -1$:

$d(w) = d(\text{current}) + 1$

 toVisit.enqueue(w)

BFS can be used to find distances from a vertex!



Keeping Track of Distance

BFS($G=(V,E)$, **starting vertex v**):

toVisit.enqueue(**v**)

set $d(v) = 0$

set all other distances to -1

while toVisit is not empty:

 current $\leftarrow$ toVisit.dequeue()

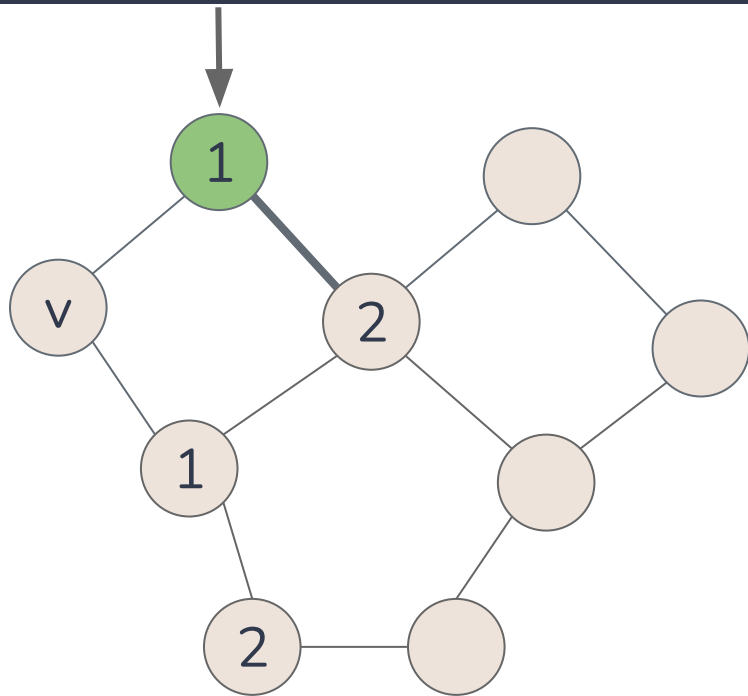
 for each edge (current, w):

 if **$d(w) \neq -1$** :

$d(w) = d(\text{current}) + 1$

 toVisit.enqueue(w)

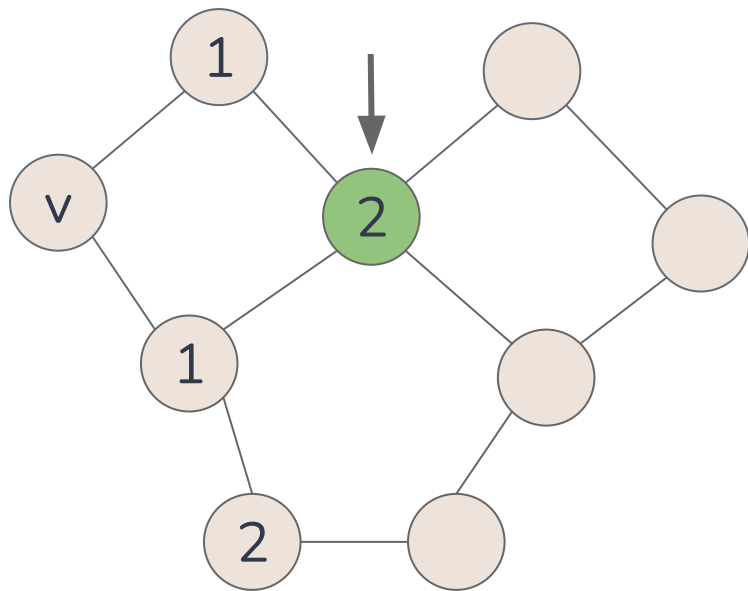
BFS can be used to find distances from a vertex!



Keeping Track of Distance

```
BFS( $G=(V,E)$ , starting vertex  $v$ ):  
  toVisit.enqueue( $v$ )  
  set  $d(v) = 0$   
  set all other distances to -1  
  while toVisit is not empty:  
    current  $\leftarrow$  toVisit.dequeue()  
    for each edge (current,  $w$ ):  
      if  $d(w) \neq -1$ :  
         $d(w) = d(\text{current}) + 1$   
        toVisit.enqueue( $w$ )
```

BFS can be used to find distances from a vertex!



Keeping Track of Distance

BFS($G=(V,E)$, **starting vertex v**):

toVisit.enqueue(**v**)

set $d(v) = 0$

set all other distances to -1

while toVisit is not empty:

 current $\leftarrow$ toVisit.dequeue()

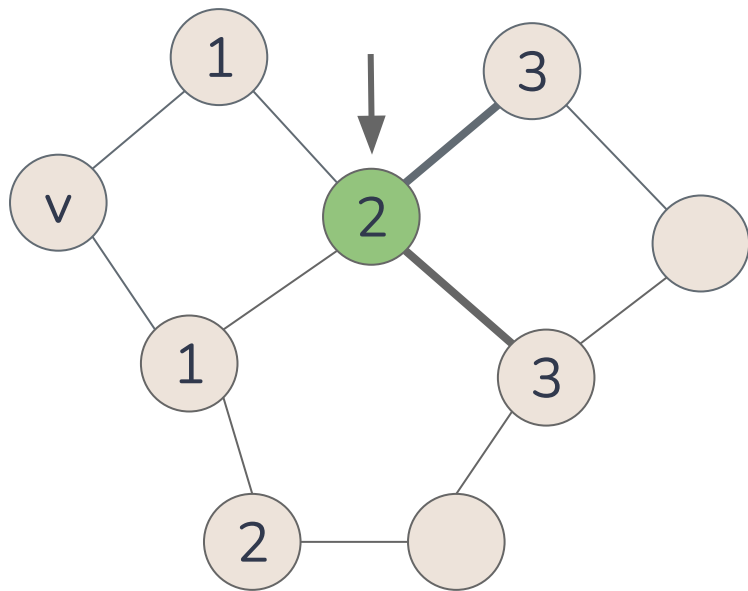
 for each edge (current, w):

 if $d(w) \neq -1$:

$d(w) = d(\text{current}) + 1$

 toVisit.enqueue(w)

BFS can be used to find distances from a vertex!



Keeping Track of Distance

BFS($G=(V,E)$, **starting vertex v**):

toVisit.enqueue(**v**)

set $d(v) = 0$

set all other distances to -1

while toVisit is not empty:

current $\leftarrow$ toVisit.dequeue()

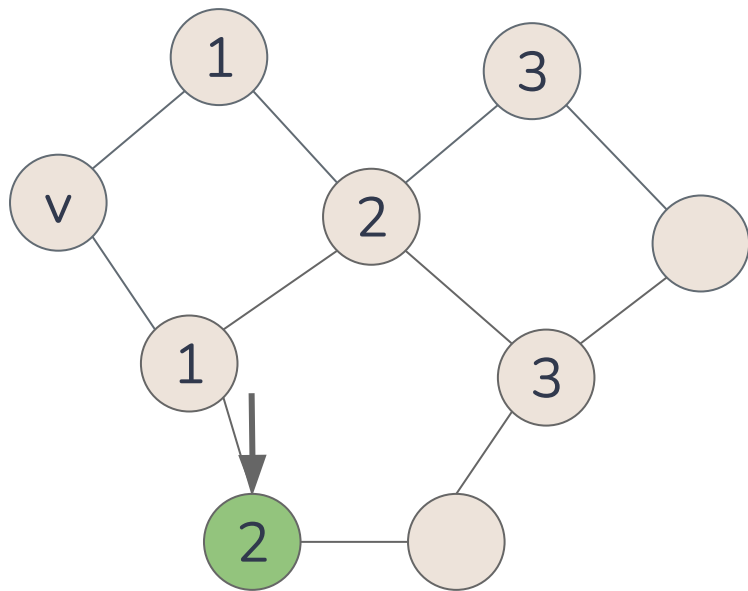
for each edge (current, w):

if $d(w) \neq -1$:

$d(w) = d(\text{current}) + 1$

toVisit.enqueue(w)

BFS can be used to find distances from a vertex!



Keeping Track of Distance

BFS($G=(V,E)$, **starting vertex v**):

toVisit.enqueue(**v**)

set $d(v) = 0$

set all other distances to -1

while toVisit is not empty:

current $\leftarrow$ toVisit.dequeue()

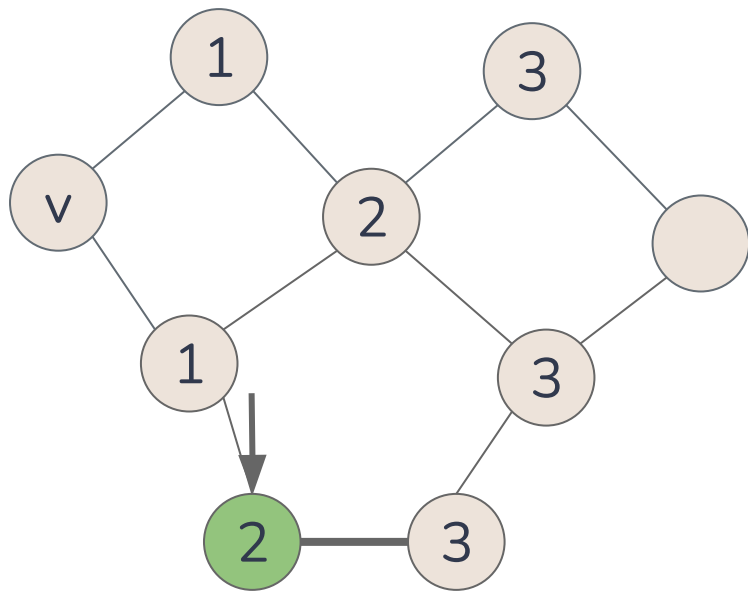
for each edge (current, w):

if $d(w) \neq -1$:

$d(w) = d(\text{current}) + 1$

toVisit.enqueue(w)

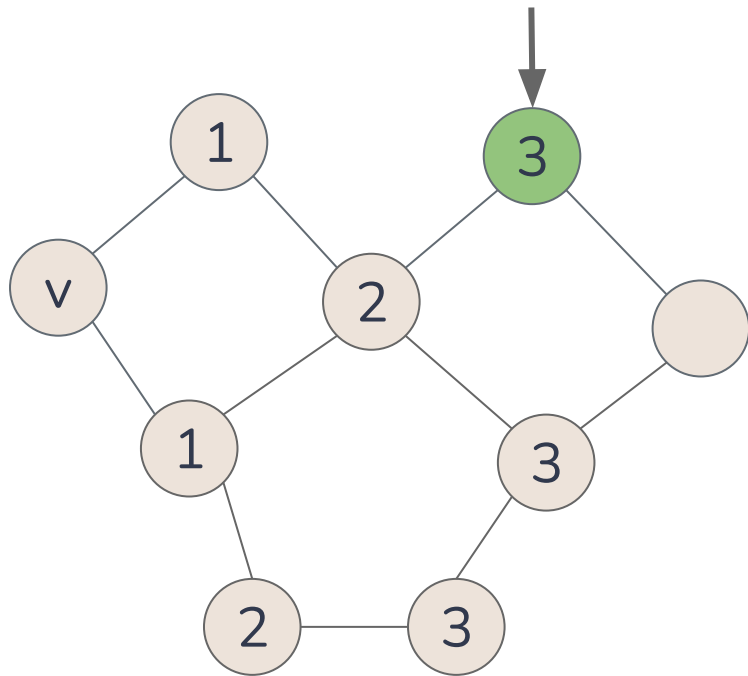
BFS can be used to find distances from a vertex!



Keeping Track of Distance

```
BFS( $G=(V,E)$ , starting vertex  $v$ ):  
  toVisit.enqueue( $v$ )  
  set  $d(v) = 0$   
  set all other distances to -1  
  while toVisit is not empty:  
    current  $\leftarrow$  toVisit.dequeue()  
    for each edge (current,  $w$ ):  
      if  $d(w) \neq -1$ :  
         $d(w) = d(\text{current}) + 1$   
        toVisit.enqueue( $w$ )
```

BFS can be used to find distances from a vertex!



Keeping Track of Distance

BFS($G=(V,E)$, starting vertex v):

```
toVisit.enqueue(v)
```

set $d(v) = 0$

set all other distances to -1

while toVisit is not empty:

```
current ← toVisit.dequeue()
```

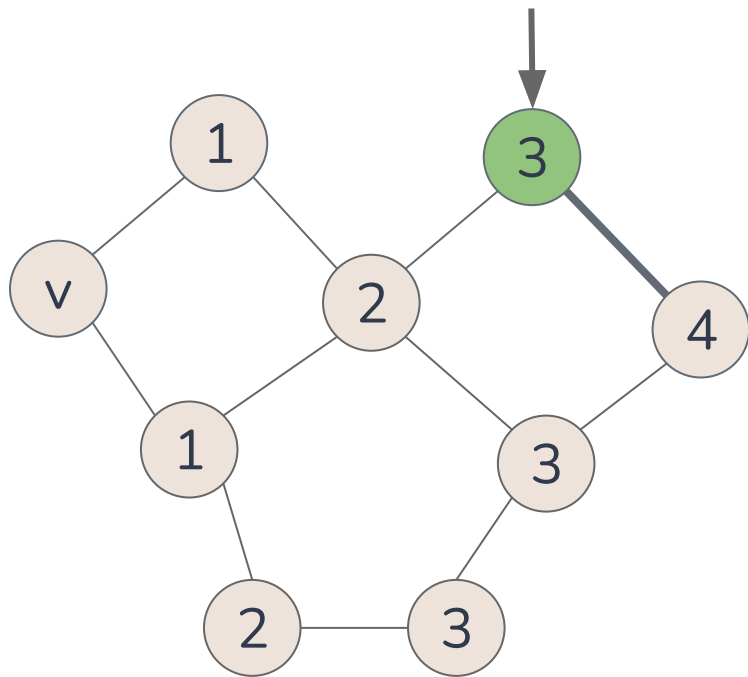
for each edge (current, w):

if $d(w) \neq -1$:

$$d(w) = d(\text{current}) + 1$$

```
toVisit.enqueue(w)
```

BFS can be used to find distances from a vertex!



Keeping Track of Distance

BFS($G=(V,E)$, **starting vertex v**):

toVisit.enqueue(**v**)

set $d(v) = 0$

set all other distances to -1

while toVisit is not empty:

 current $\leftarrow$ toVisit.dequeue()

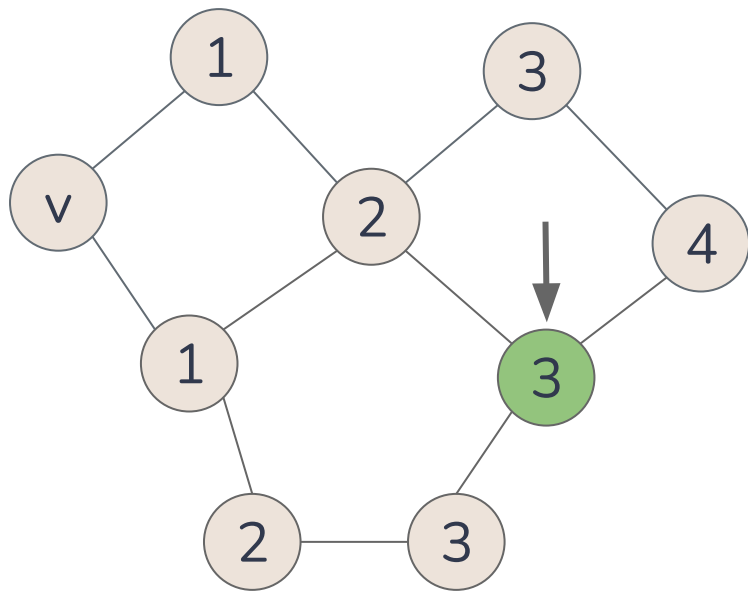
 for each edge (current, w):

 if $d(w) \neq -1$:

$d(w) = d(\text{current}) + 1$

 toVisit.enqueue(w)

BFS can be used to find distances from a vertex!



Keeping Track of Distance

BFS($G=(V,E)$, **starting vertex v**):

toVisit.enqueue(**v**)

set $d(v) = 0$

set all other distances to -1

while toVisit is not empty:

 current $\leftarrow$ toVisit.dequeue()

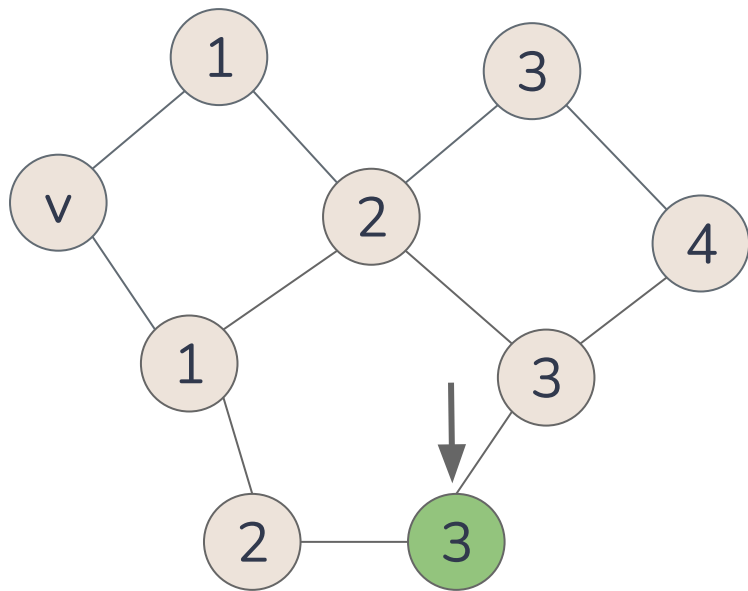
 for each edge (current, w):

 if $d(w) \neq -1$:

$d(w) = d(\text{current}) + 1$

 toVisit.enqueue(w)

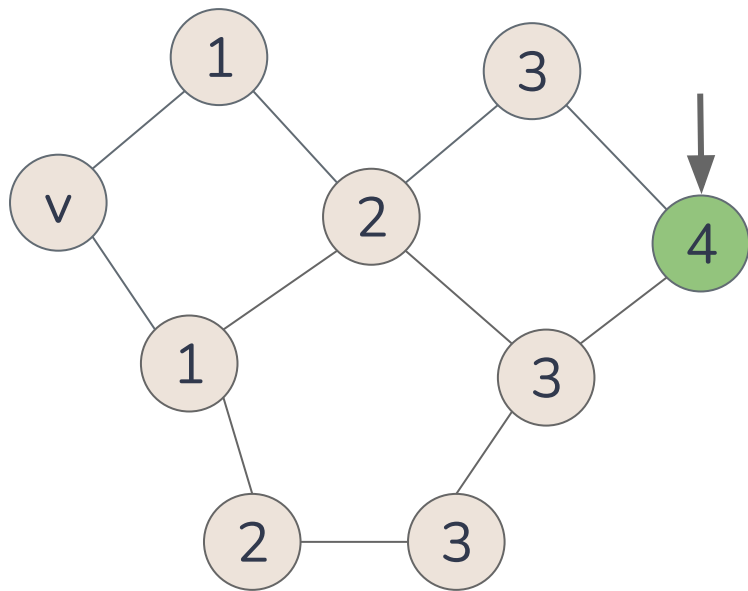
BFS can be used to find distances from a vertex!



Keeping Track of Distance

```
BFS( $G=(V,E)$ , starting vertex  $v$ ):  
  toVisit.enqueue( $v$ )  
  set  $d(v) = 0$   
  set all other distances to -1  
  while toVisit is not empty:  
    current  $\leftarrow$  toVisit.dequeue()  
    for each edge (current,  $w$ ):  
      if  $d(w) \neq -1$ :  
         $d(w) = d(\text{current}) + 1$   
        toVisit.enqueue( $w$ )
```

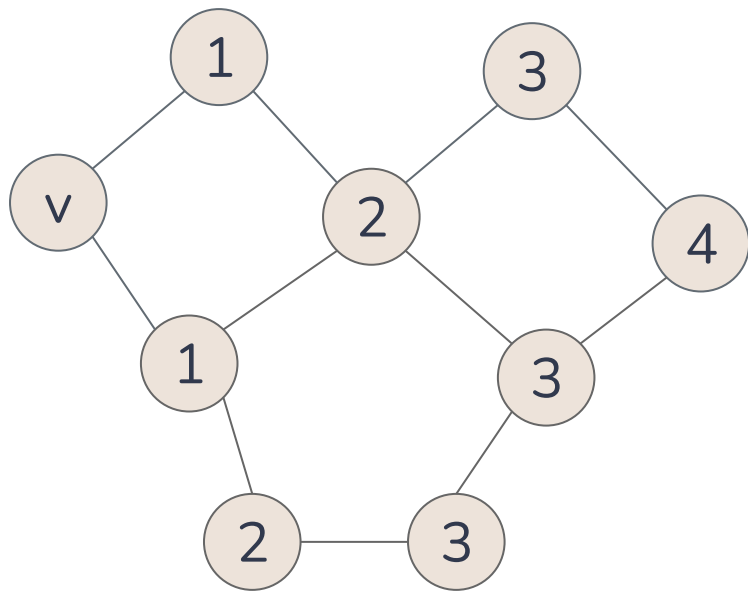
BFS can be used to find distances from a vertex!



Keeping Track of Distance

```
BFS( $G=(V,E)$ , starting vertex  $v$ ):  
  toVisit.enqueue( $v$ )  
  set  $d(v) = 0$   
  set all other distances to -1  
  while toVisit is not empty:  
    current  $\leftarrow$  toVisit.dequeue()  
    for each edge (current,  $w$ ):  
      if  $d(w) \neq -1$ :  
         $d(w) = d(\text{current}) + 1$   
        toVisit.enqueue( $w$ )
```

BFS can be used to find distances from a vertex!



Keeping Track of Distance

```
BFS( $G=(V,E)$ , starting vertex  $v$ ):  
  toVisit.enqueue( $v$ )  
  set  $d(v) = 0$   
  set all other distances to -1  
  while toVisit is not empty:  
    current  $\leftarrow$  toVisit.dequeue()  
    for each edge (current,  $w$ ):  
      if  $d(w) \neq -1$ :  
         $d(w) = d(\text{current}) + 1$   
        toVisit.enqueue( $w$ )
```

A Second Application: Bipartiteness

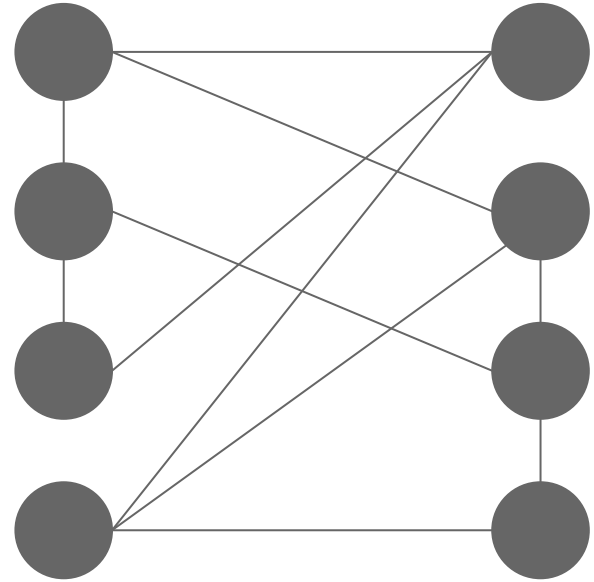


How can we tell if a graph is bipartite?

Brute Force:

2^n different colorings of the n vertices.

Try the worksheet problems!

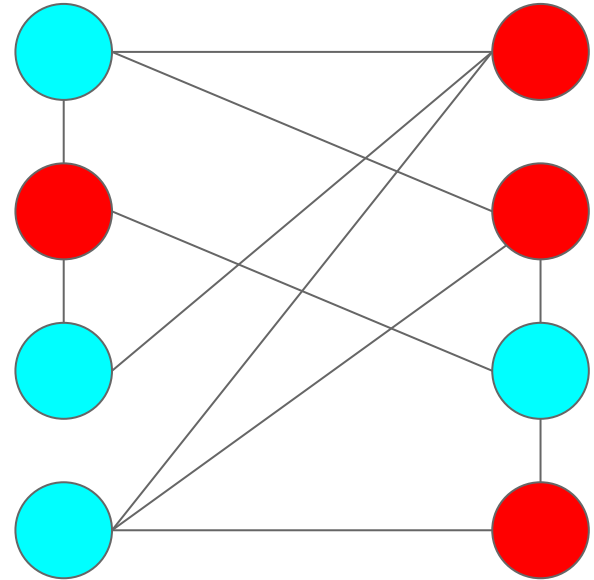


How can we tell if a graph is bipartite?

Brute Force:

2^n different colorings of the n vertices.

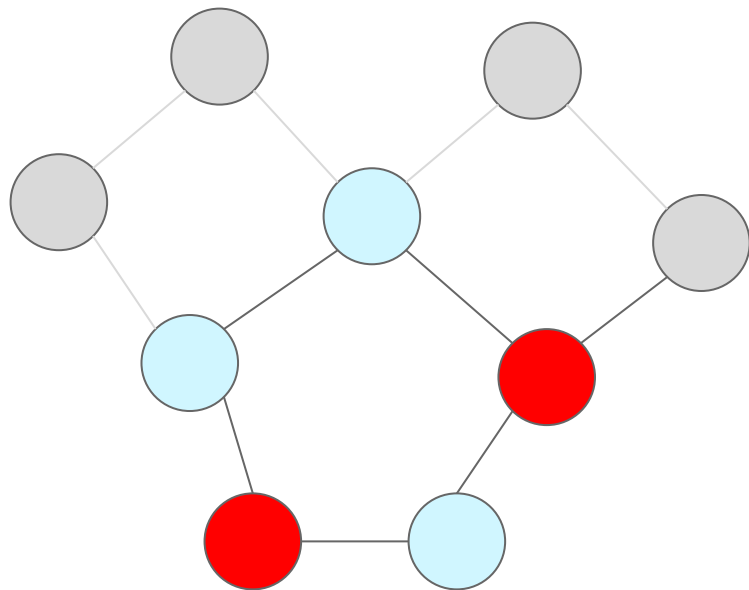
Try the worksheet problems!



How can we tell if a graph is bipartite?

Claim: If a graph has an odd cycle, it can not be bipartite.

Proof: Start at any vertex and color it blue, then alternate along the cycle. At the end of the cycle, the last vertex must be the same color as the starting vertex.



Parting Question: Is that enough?

If a graph does not have an odd cycle, does it have to be bipartite?

