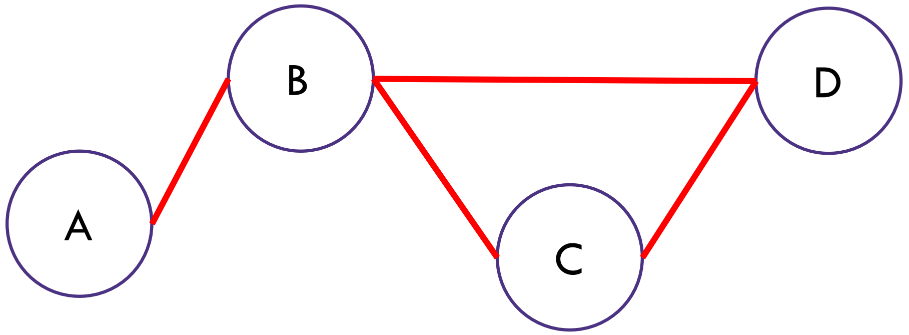
CSE 421
Algorithms

Autumn 2019
Lecture 5

Graphs

Represent objects and the relationships between pairs of them.



Formally:

A graph is a pair: G = (V,E)

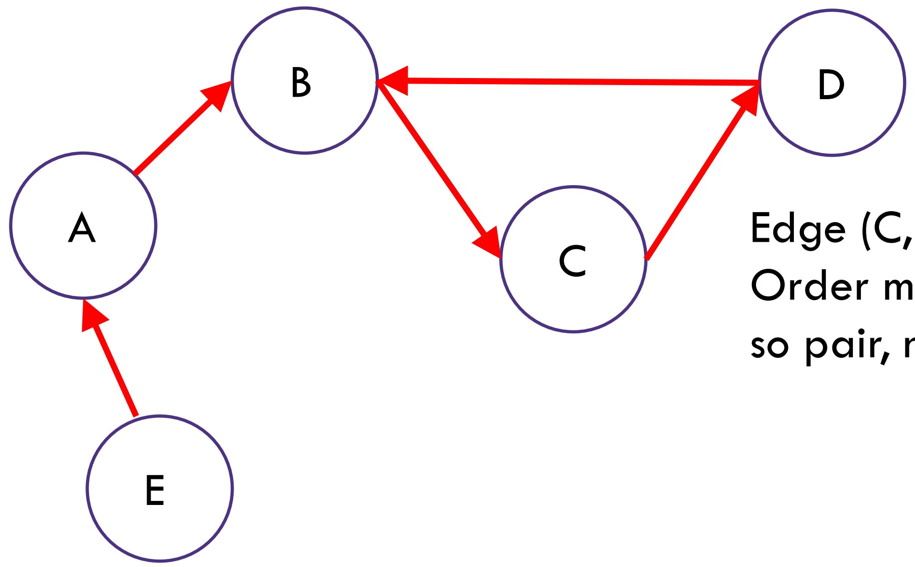
V: set of **vertices** (aka **nodes**) $\{A, B, C, D\}$

E: set of **edges** $\{\{A, B\}, \{B, C\}, \{B, D\}, \{C, D\}\}$

- Each edge is a pair of vertices.

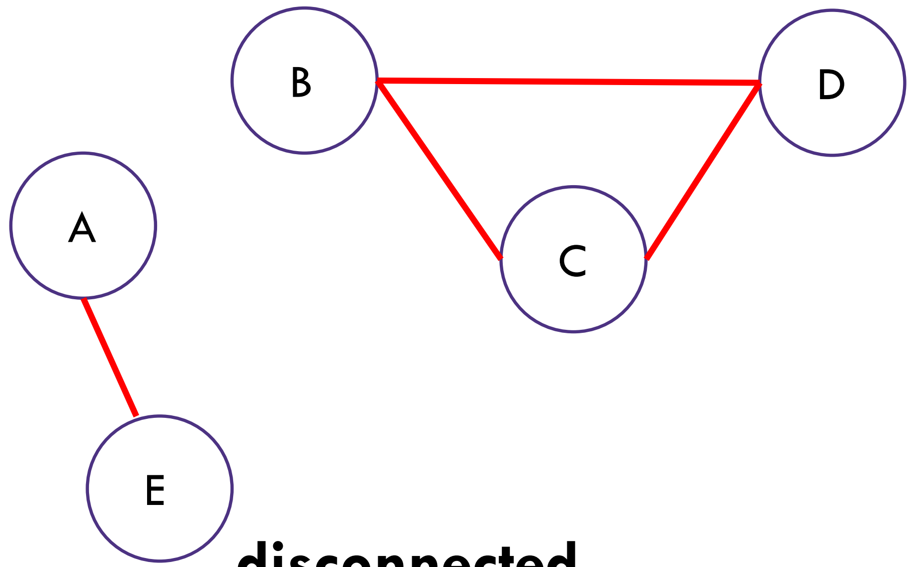
Graph Terms

Graphs can be directed or undirected.



Edge (C,D)
Order matters,
so pair, not a set.

{B,C,D} is a component

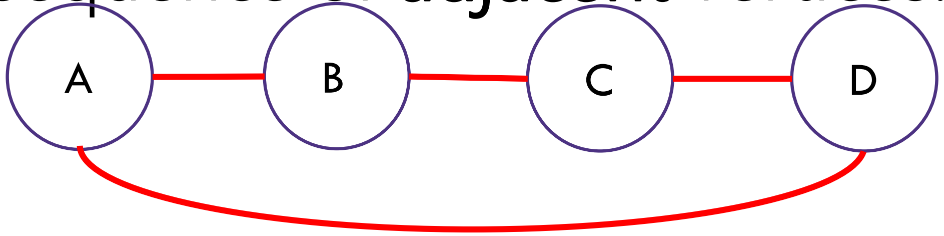


disconnected

Weakly connected:
if you ignore the direction of edges, it's connected
Strongly connected:
Can get from u to v AND from v back to u for all u,v

Graph Terms

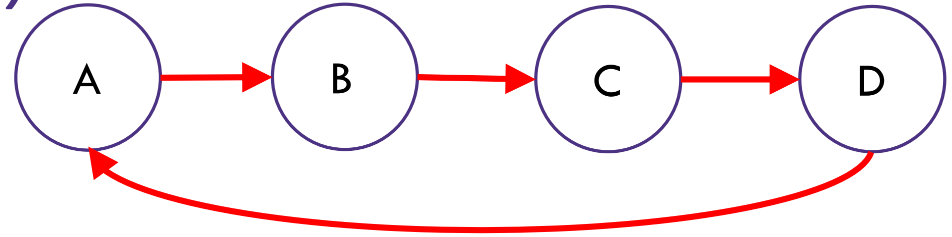
Path – A sequence of **adjacent** vertices. Each connected to next by an edge.



A,B,C,D is a path.

So is A,B,A

(Directed) Path–must follow the direction of the edges



A,B,C,D,B is a directed path.

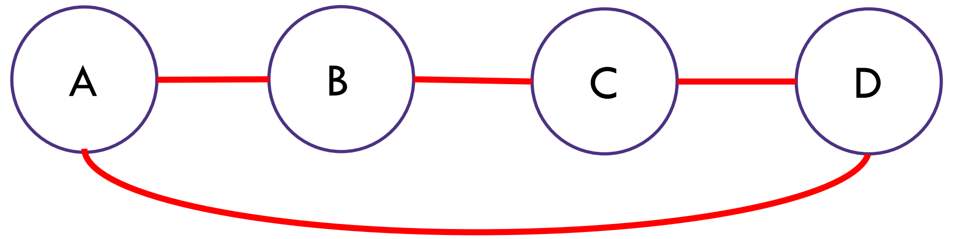
A,B,A is not.

Length – The number of edges in a path

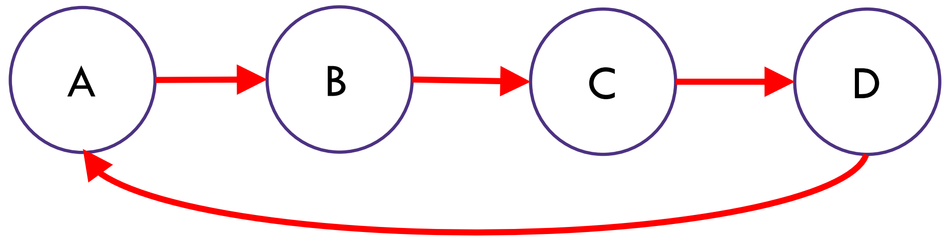
- (A,B,C,D) has length 3.

Graph Terms

Simple Path – A path that doesn't repeat a vertex.
A,B,C,D is a simple path. A,B,A is not.



Cycle – simple path with an extra edge from last vertex back to first.



Extra edge must be a new edge, not reusing an old one.

Be careful when looking at other sources!

- Other sources use "path" to mean what we call a "simple path"

More Graph Terms

Neighborhood

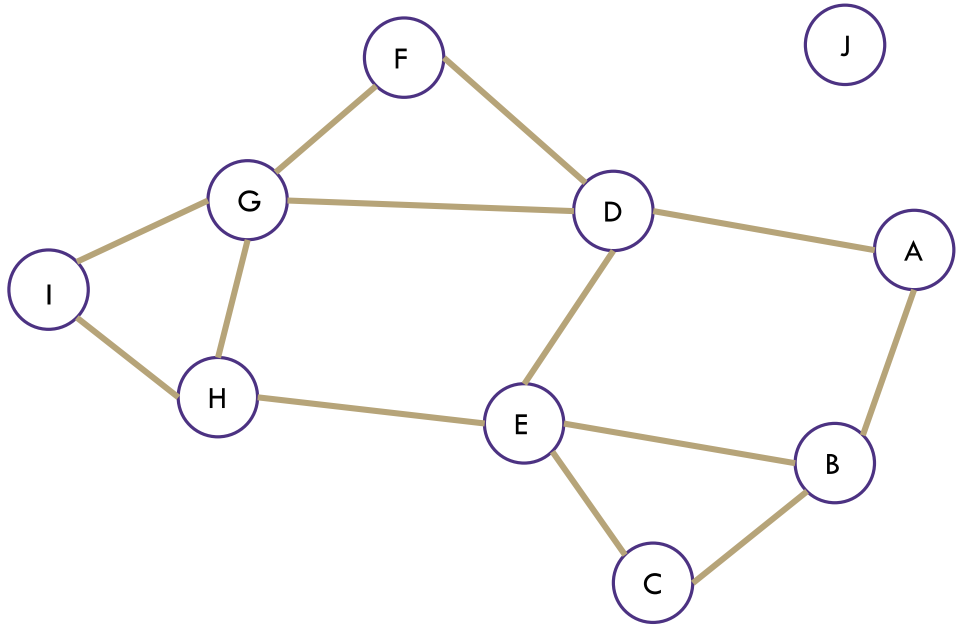
- All the edges you have are connected to directly by an edge.
- $N(D) = \{A, E, F, G\}$
- In directed graphs, have "out-neighborhood" and "in-neighborhood"

Distance – length of the shortest path between two vertices.

- In directed graphs, $d(u, v)$ might not be $d(v, u)$.

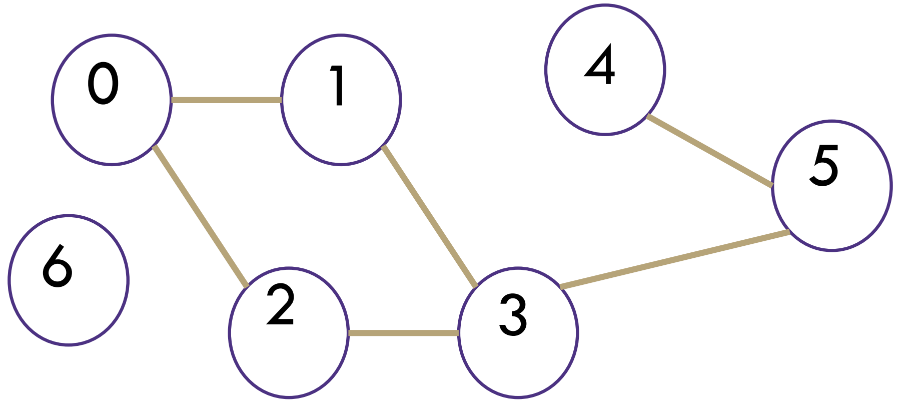
Trees – are connected graphs without any cycles.

We can (optionally) "root" a tree, and think of edges going from "top" to "bottom"



Representing and Using Graphs

Incidence Matrix



In an incidence matrix a[u][v] is 1 if there is an edge (u,v), and 0 otherwise.

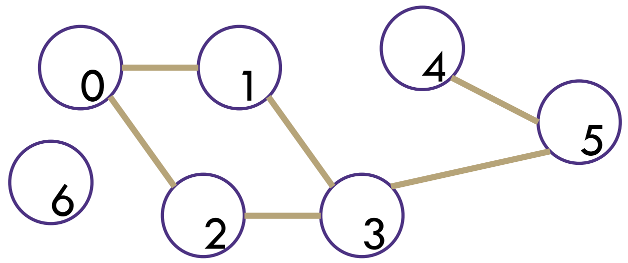
Worst-case Time Complexity (|V| = n, |E| = m):

- Add Edge: $O(1)$
- Remove Edge: $O(1)$
- Check edge exists from (u,v): $O(1)$
- Get outneighbors of u: $O(n)$
- Get inneighbors of u: $O(n)$

Space Complexity: $O(n^2)$

	0	1	2	3	4	5	6
0	0	1	1	0	0	0	0
1	1	0	0	1	0	0	0
2	1	0	0	1	0	0	0
3	0	1	1	0	0	1	0
4	0	0	0	0	0	1	0
5	0	0	0	1	1	0	0
6	0	0	0	0	0	0	0

Adjacency List



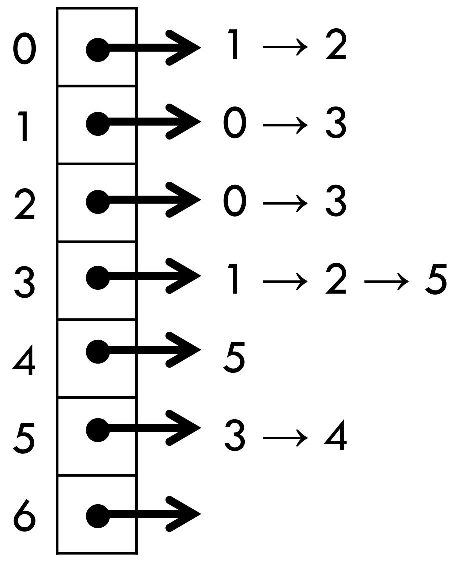
An array where the u^{th} element contains a list of neighbors of u.

Directed graphs: list of out-neighbors (a[u] has v for all (u,v) in E)

Time Complexity (|V| = n, |E| = m):

- Add Edge: $O(1)$
- Remove Edge (u,v): $O(\deg(u))$
- Check edge exists from (u,v): $O(\deg(u))$
- Get all outneighbors of u: $O(\deg(u))$
- Get all inneighbors of u: $O(n+m)$

Space Complexity: $O(n+m)$



Suppose we use a linked list for each node.

Graph Search

Graph Search

Ways to answer the question “Can I get from u to v?”

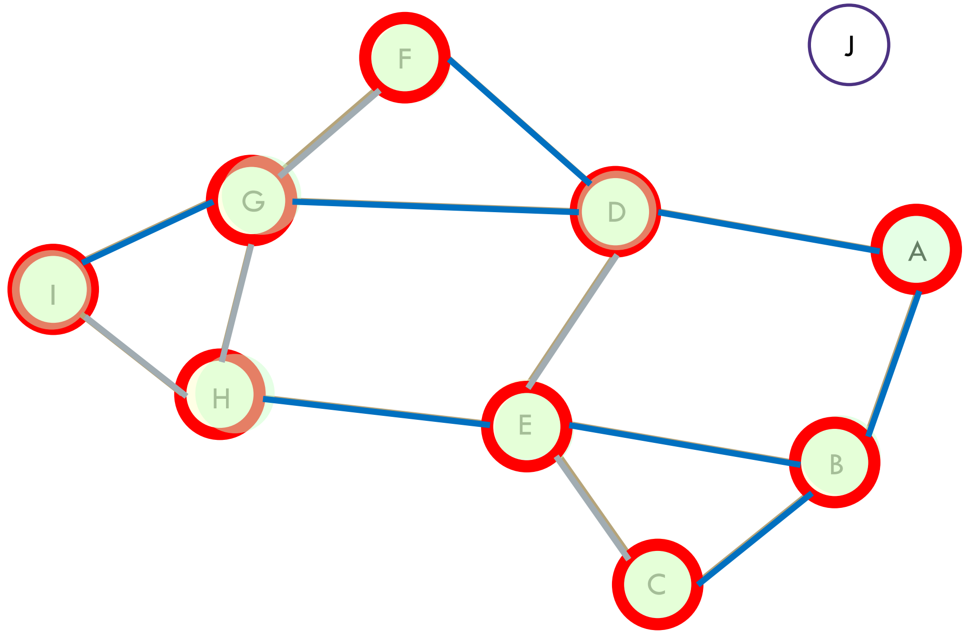
Slight modifications will let us answer lots of other questions.

In 332, you used BFS to find shortest paths (in unweighted graphs)

We’ll have more applications of both BFS and DFS in this class.

Breadth First Search

```
search(graph)
    toVisit.enqueue(first vertex)
    mark first vertex as seen
    while(toVisit is not empty)
        current = toVisit.dequeue()
        for (v : current.neighbors())
            if (v is not seen)
                mark v as seen
                toVisit.enqueue(v)
```



Current node: I

Queue: B D E C F G H I

Finished: A B D E C F G H I

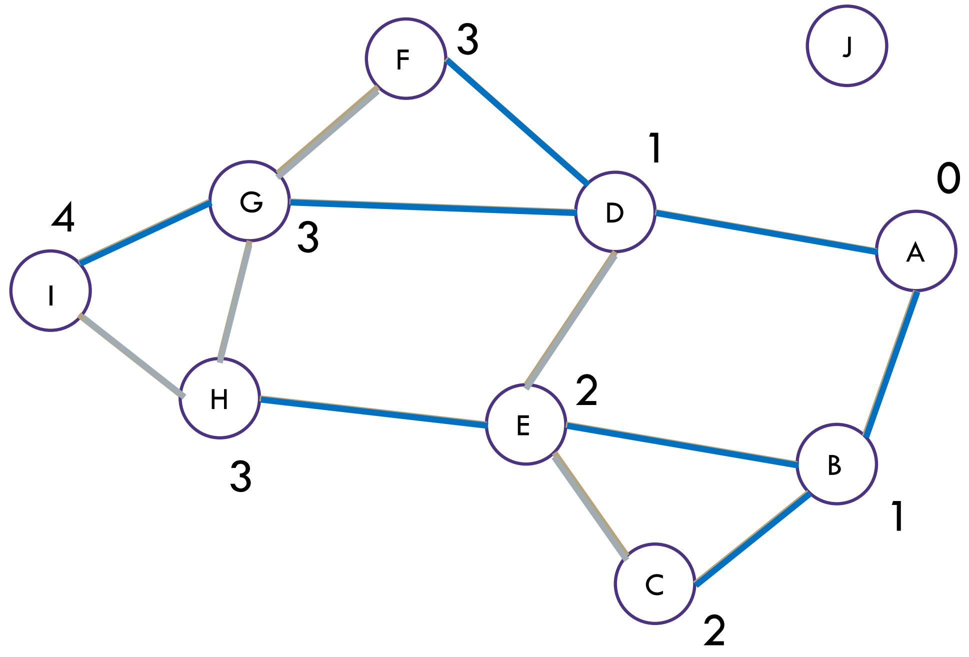
Layers

Call the starting node (A) layer 0.
$N(A)$ is layer 1.
All nodes we discovered while processing layer 1 is layer 2.
…
All nodes discovered while processing layer i is layer $i + 1$.

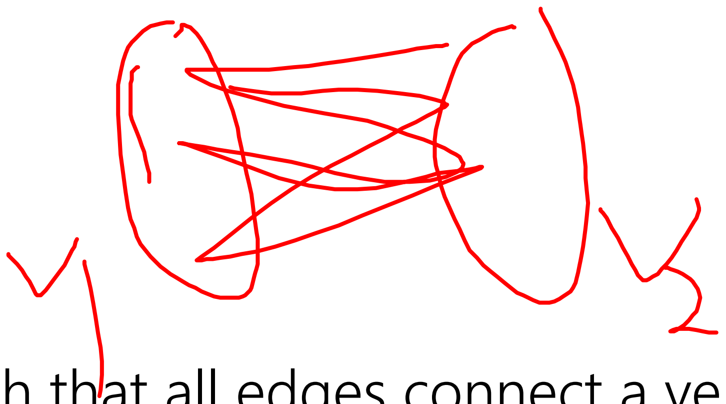
These are the distances you found in 332 (we're just using them differently today).

Every time an edge discovered a vertex we had never seen before, we colored it blue. The others we made gray.

The blue edges form a tree! (why?)
It's called the "BFS tree"



Bipartite Graphs



Application for today: checking if a graph is bipartite.

A graph V is bipartite if V can be partitioned into V_1, V_2 such that all edges connect a vertex in V_1 to a vertex in V_2.

- For convenience, we'll also talk about "coloring" vertices – everything in V_1 is red, everything in V_2 is blue.

Bipartite graphs show up a lot.

- You've already used them to represent stable matchings.
- You'll see more when you get to network flows.

Let's try to identify them.

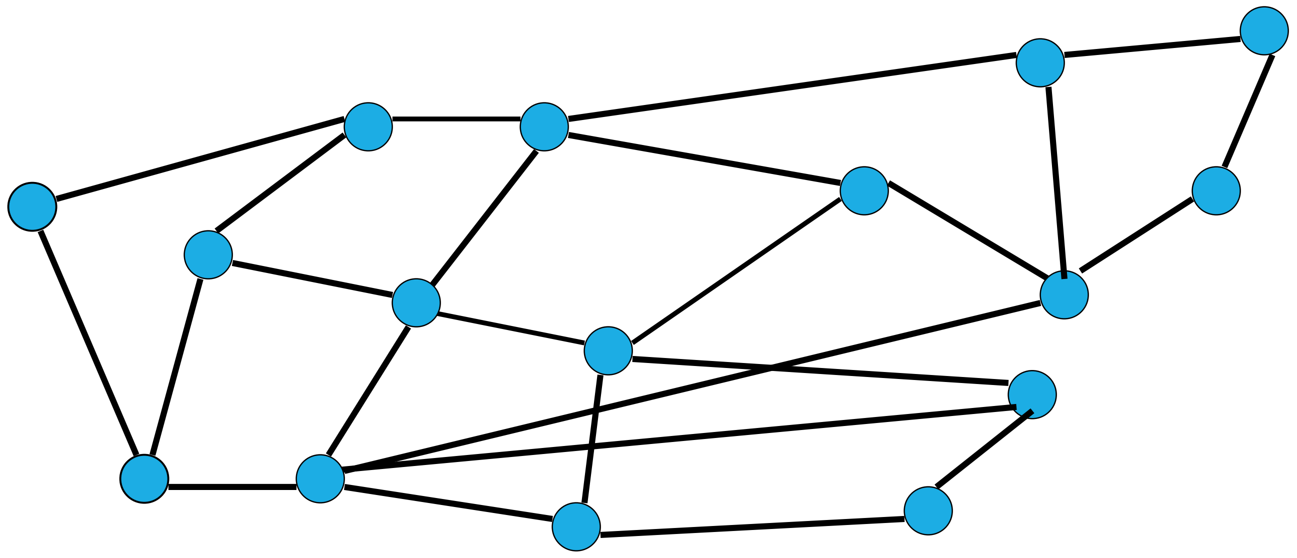
Bipartite Graph identification

Given a graph, return TRUE it is bipartite, and FALSE otherwise.

By hand

Before we design an algorithm, let’s see if doing it by hand will give us any ideas.

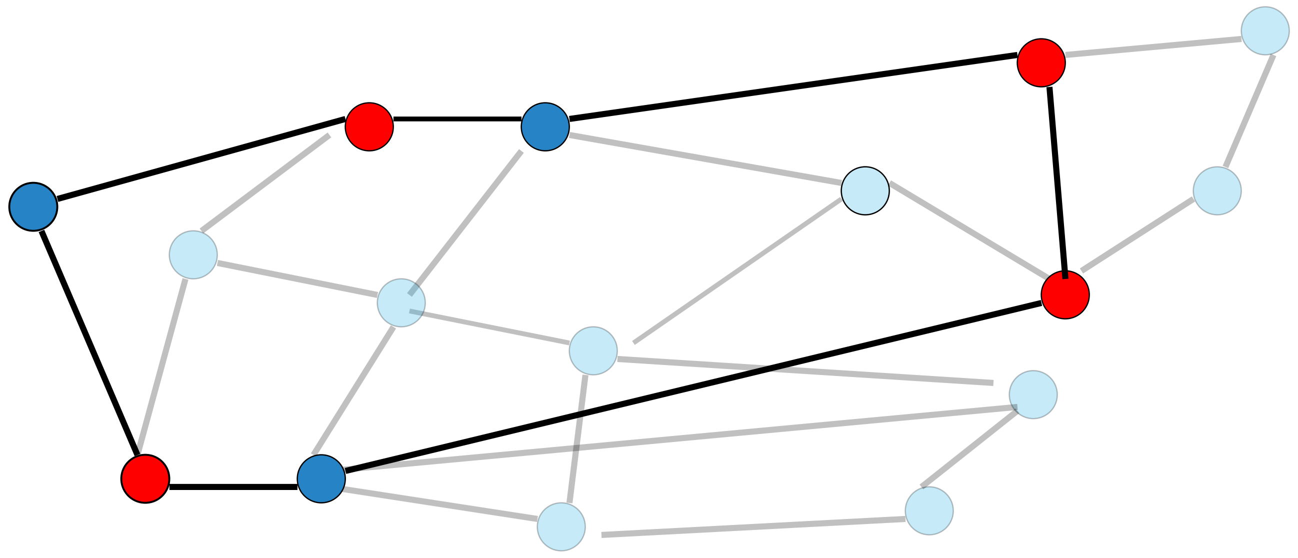
Is this graph bipartite?



By hand

Before we design an algorithm, let's see if doing it by hand will give us any ideas.

Is this graph bipartite?



How did we know that graph wasn't bipartite?

It had an odd cycle!

How can we test whether a graph is bipartite?

Pick some arbitrary vertex, and color it red

All of its neighbors must be blue, color them blue (making sure you don't create a blue-blue edge).

All of their neighbors must be red, color them red (making sure you don't create a red-red edge).

…

What algorithm works like this? BFS!

```
search(graph)
   toVisit.enqueue(first vertex)

     mark first vertex as seen
   while(toVisit is not empty)
      current = toVisit.dequeue()

      for (v : current.neighbors())
         if (v is not seen)
            mark v as seen
            toVisit.enqueue(v)
```

How do we modify this pseudocode to 2-color?

Color a vertex when you discover it.

When you reach the end of a layer, change colors (how do you know?).

When an edge goes to an already discovered vertex, make sure it’s ok.

 i.e. connecting opposite colors.

If you finish (for every component) return true, if an edge isn’t ok return false.

Proof of correctness:

If the algorithm outputs TRUE, it’s correct.

- We’ve found the bipartition we’re looking for.

If the algorithm outputs FALSE

- Need to show there isn’t a bipartition.
- The fact that we found a red-red or blue-blue edge isn’t a proof. That edge means *that particular* assignments of red and blue isn’t a bipartition, but maybe there’s another assignment we never checked.

You can argue that the one you were checking is the “only possibility” (and therefore since it doesn’t work, the graph really is bipartite)

But an easier argument is to show directly the graph isn’t bipartite.

- By finding an odd cycle in the graph!

Where’s the odd cycle?

Well…why did our algorithm return false?

It found an edge between two vertices of the same color.

Claim: They were actually in the same layer

Claim: Which means we can find an odd (simple) cycle.

Where's the odd cycle?

Well…why did our algorithm return false?

It found an edge between two vertices of the same color (call them u and v).

Claim: They were actually in the same layer

- **Edges can only go from layer $\boldsymbol{i}$ to layer $\boldsymbol{i}$ or from $\boldsymbol{i}$ to $\boldsymbol{i+1}$**
- otherwise would have been explored by the algorithm.

Claim: Which means we can find an odd cycle.

- Let w be the lowest common ancestor of u and v. Consider the path: from w to u in the the BFS tree, along (u, v), then back up from v to w in the BFS tree.
- It's a cycle (why?) and it's odd length! (why?)

Takeaways

BFS travels through a graph layer-by-layer.

- You can use those layers in algorithms.

New BFS application: Checking if a graph is bipartite.

New Proof technique:

- Sometimes the output of an algorithm can help us prove a theorem
- The output of BFS let us prove a theorem about graphs
  - A graph is bipartite if and only if it has no odd cycles.
- You'll can also do the opposite: prove a theorem about graphs (or whatever you're working on) and use that to design an algorithm or prove it works.

Monday: Paul Beame will lecture. He'll talk about Depth First Search.