

# Lecture 9

Graphs, BFS and DFS

# Announcements!

- No new HW for this week – study for the exam!
  - Review Session Thursday 3pm.
  - Study Party Thursday evening (instead of HW party)
- MIDTERM:
  - Pick-up exams after class on Monday.
  - Take them in any 3-hour window.
    - You may use one double-sided “cheat sheet” that you have prepared yourself.
    - During the exam you may not consult any other resources, talk to anyone, go online, use your phone, etc.
    - If you have a question during the exam, please make a note on your exam that you were confused and explicitly stating what you assume the question meant.
  - Return them Tuesday.
  - See Piazza post for more details, email staff list with any logistics questions.

# Stanford Tree Hacks Hackathon this weekend!



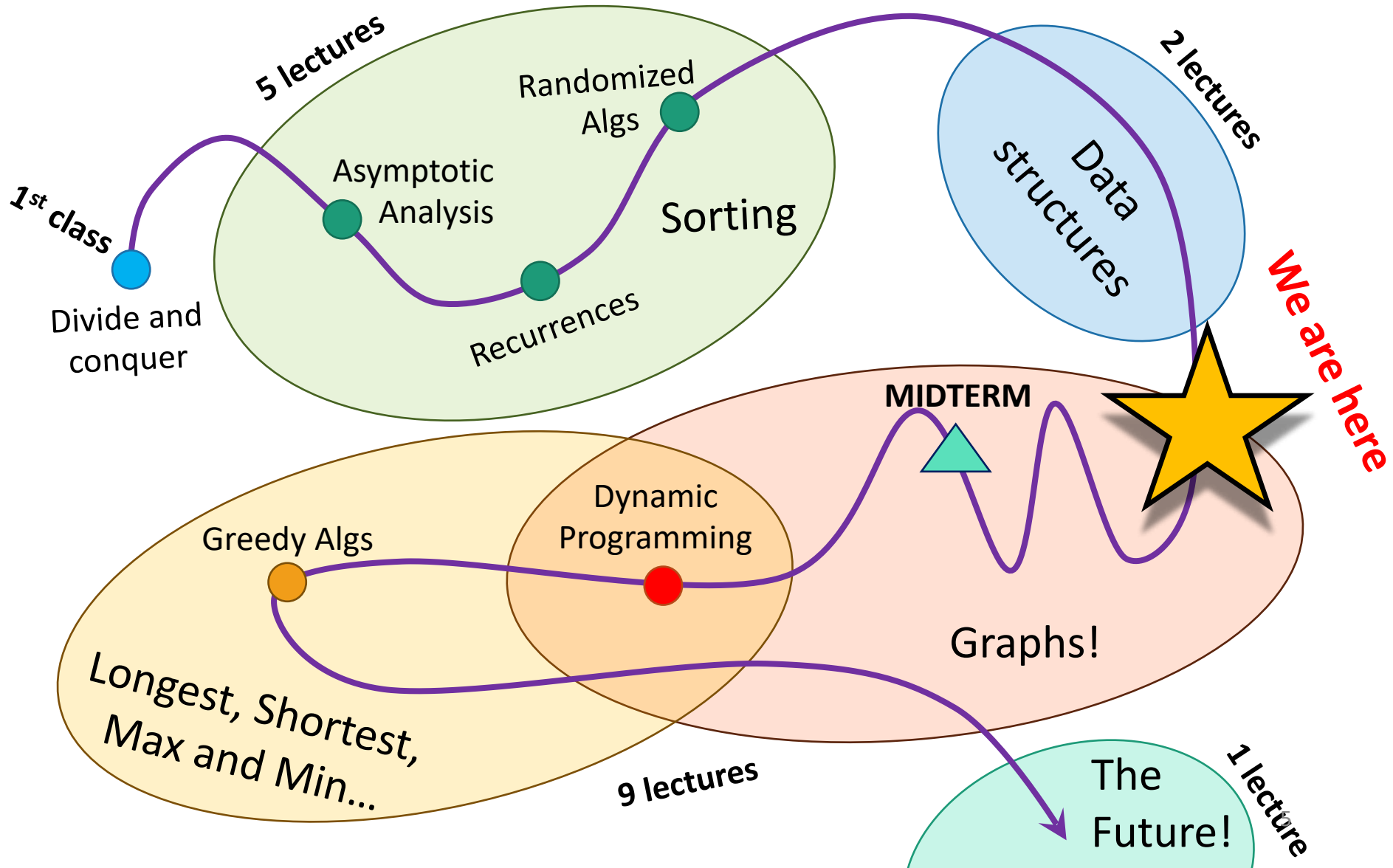
Build a cool project,  
meet amazing people,  
win incredible prizes!  
[root.treehacks.com](http://root.treehacks.com)



# One more thing about hash functions

- I said on Monday that an important thing about a hash family  $H$  is that it is small – this means that you can represent  $h \in H$  using  $\log(|H|)$  bits.
- That's true, and that's a necessary condition for a good hash function.
- BUT! We also need that we can evaluate  $h$  efficiently given this small representation:
  - Good: " $h_{a,b}(x) = ax + b \bmod p \bmod n$ "
  - Not helpful: " $h_{10331}(x)$  is the 10331'st function in  $H$ "
- Fortunately, that small universal hash family we saw is of the good type.

# Roadmap

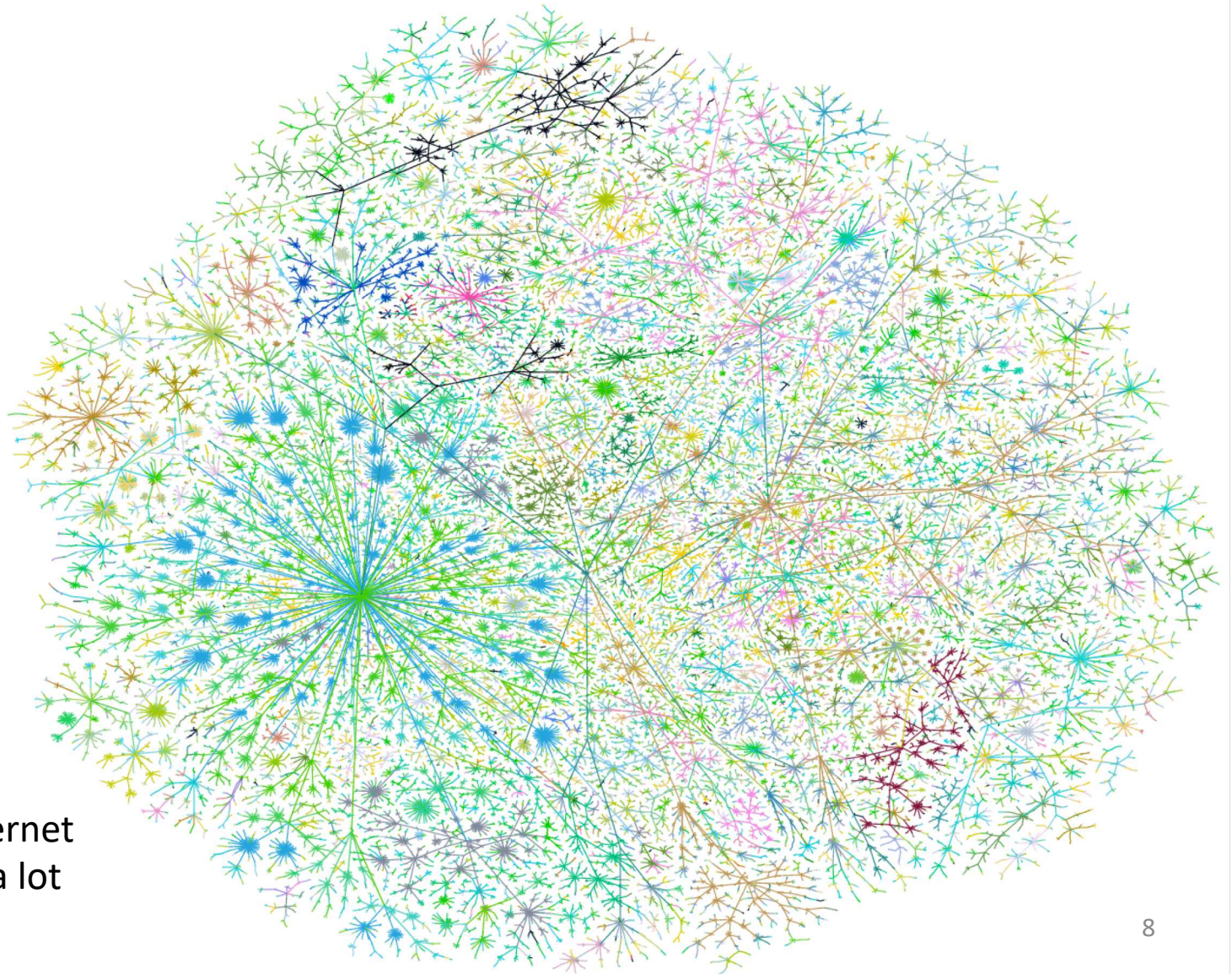


# Outline

- Part 0: Graphs and terminology
- Part 1: Depth-first search
  - Application: topological sorting
  - Application: in-order traversal of BSTs
- Part 2: Breadth-first search
  - Application: shortest paths
  - Application (if time): is a graph bipartite?

# Part 0: Graphs

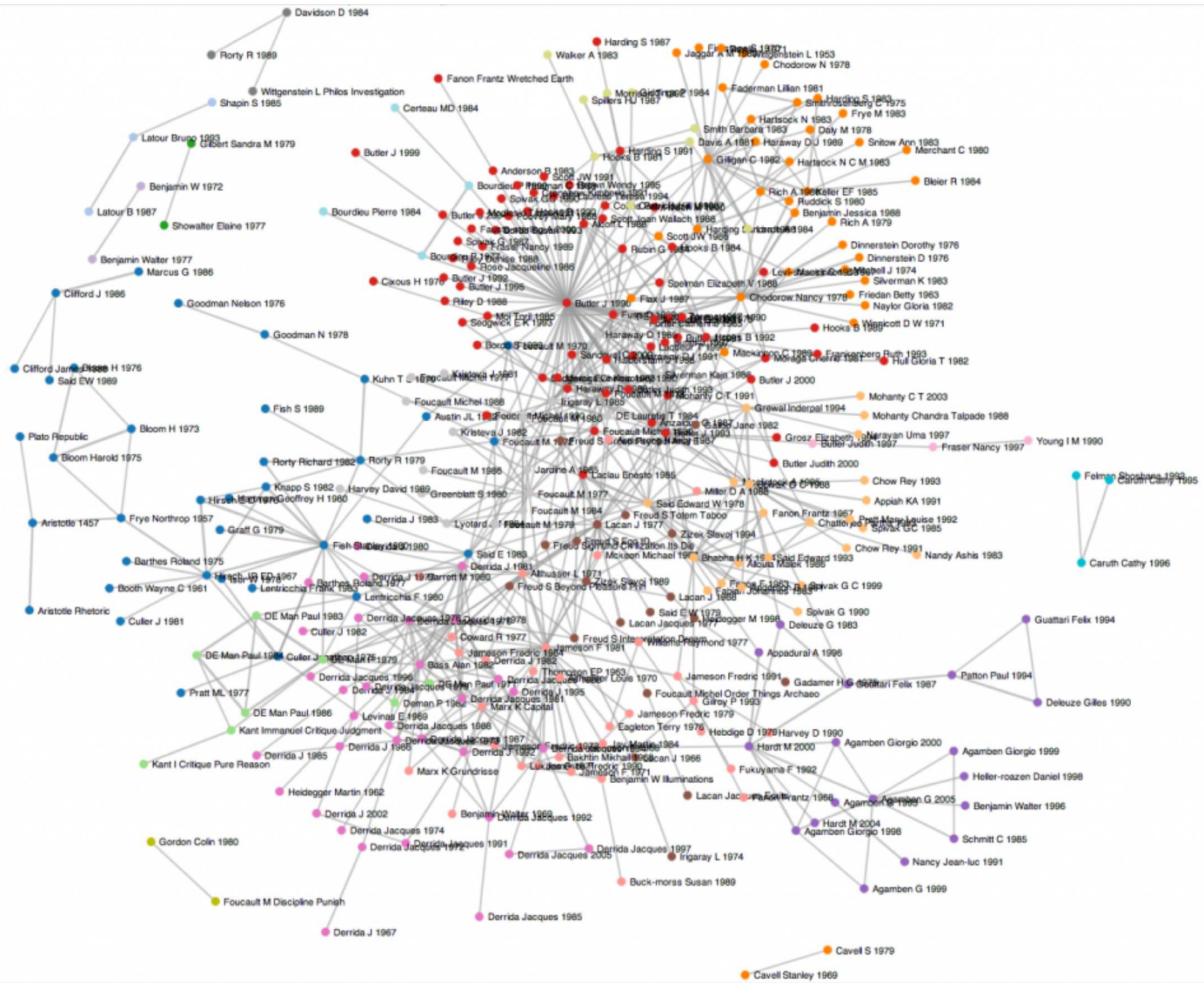
# Graphs



Graph of the internet  
(circa 1999...it's a lot  
bigger now...)

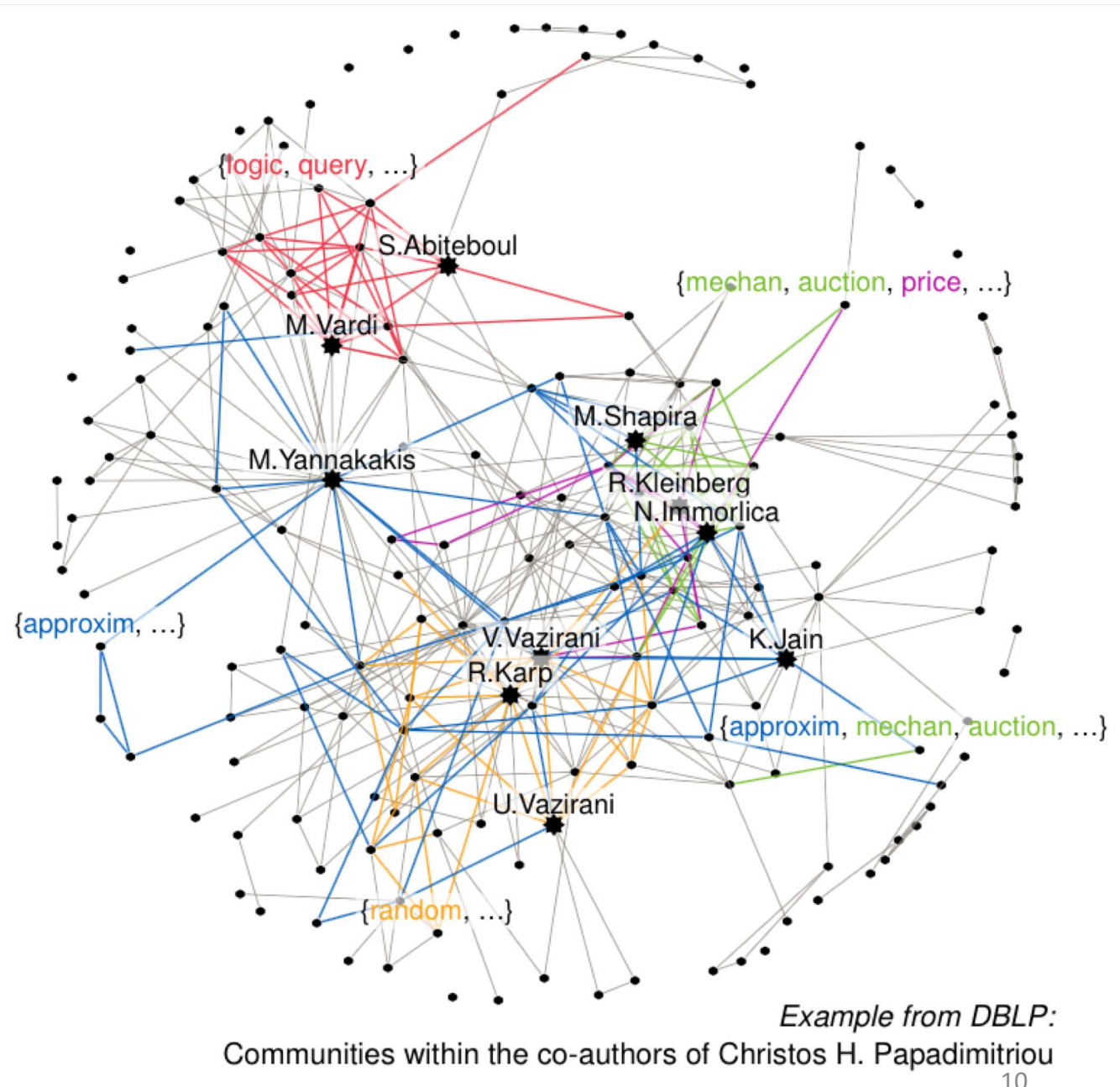
# Graphs

Citation graph of  
literary theory  
academic papers



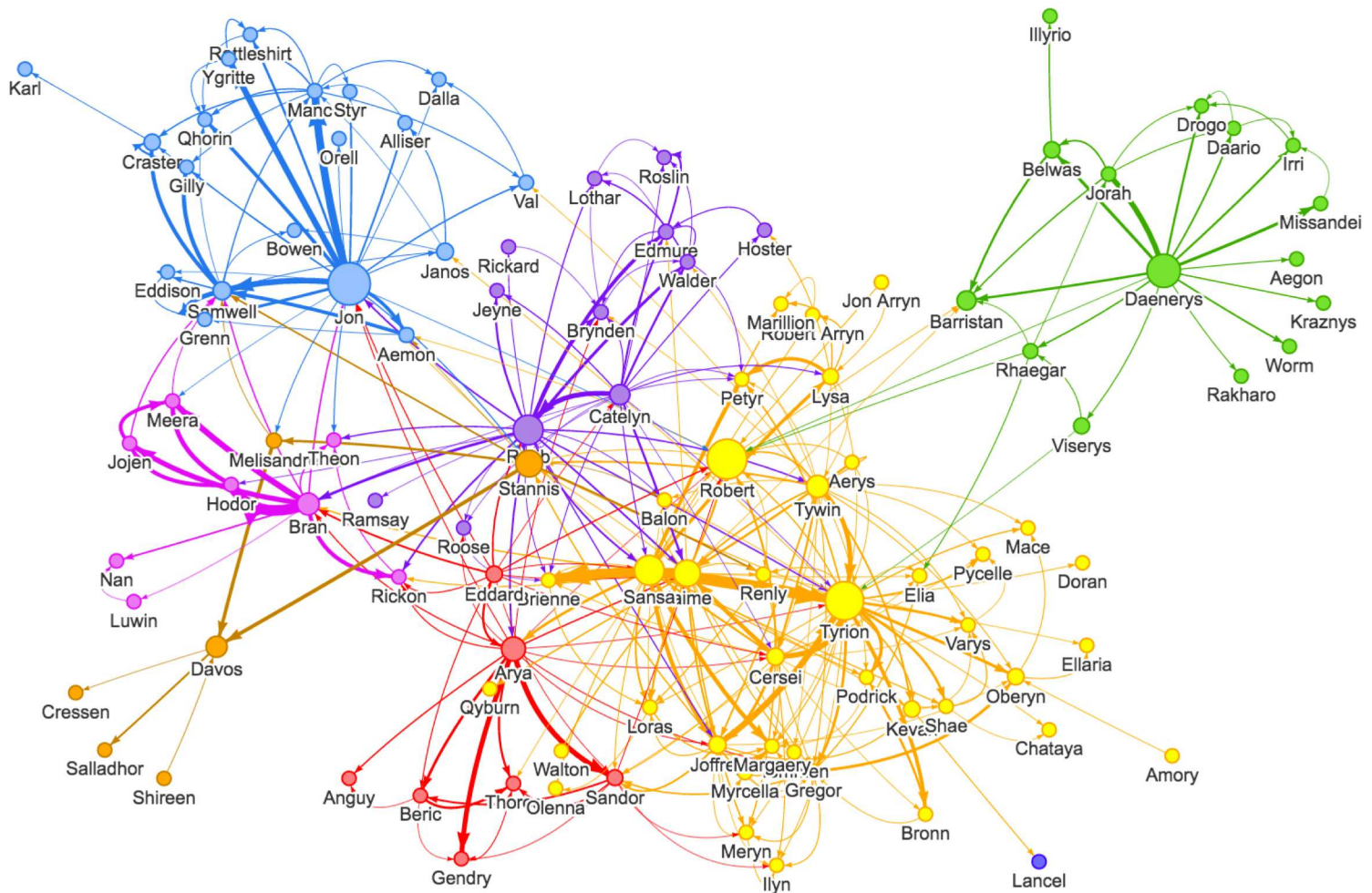
# Graphs

Theoretical Computer  
Science academic  
communities



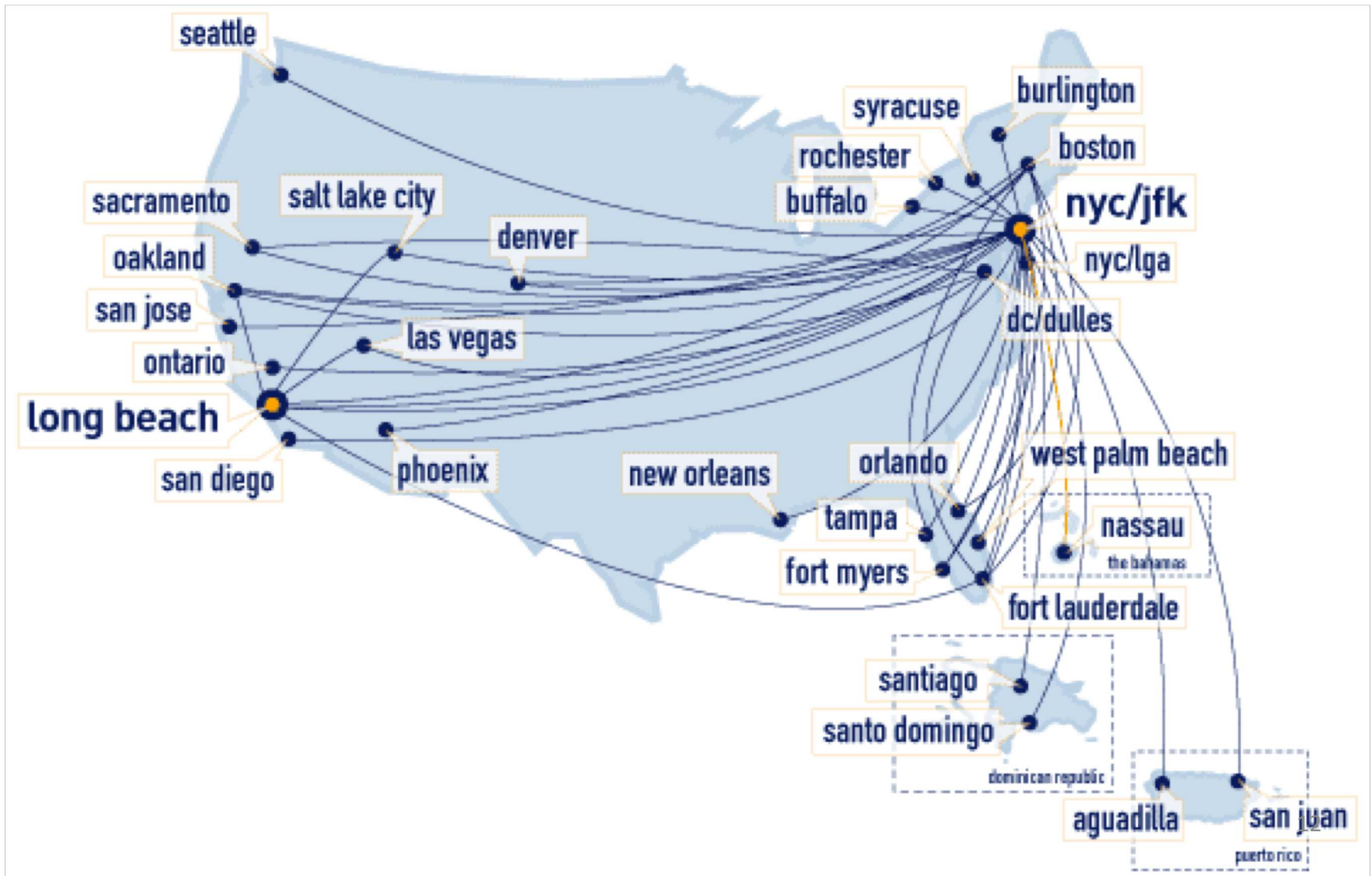
# Graphs

## Game of Thrones Character Interaction Network



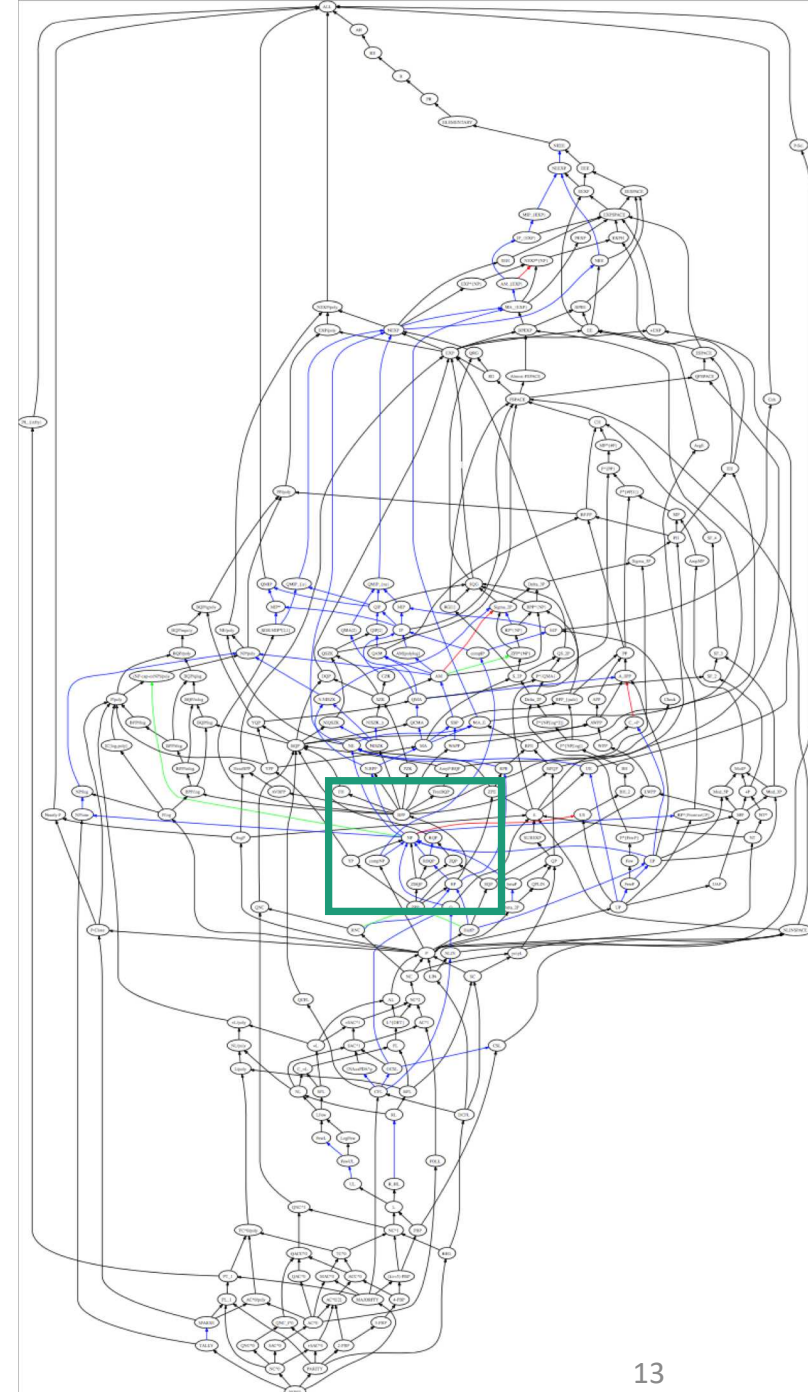
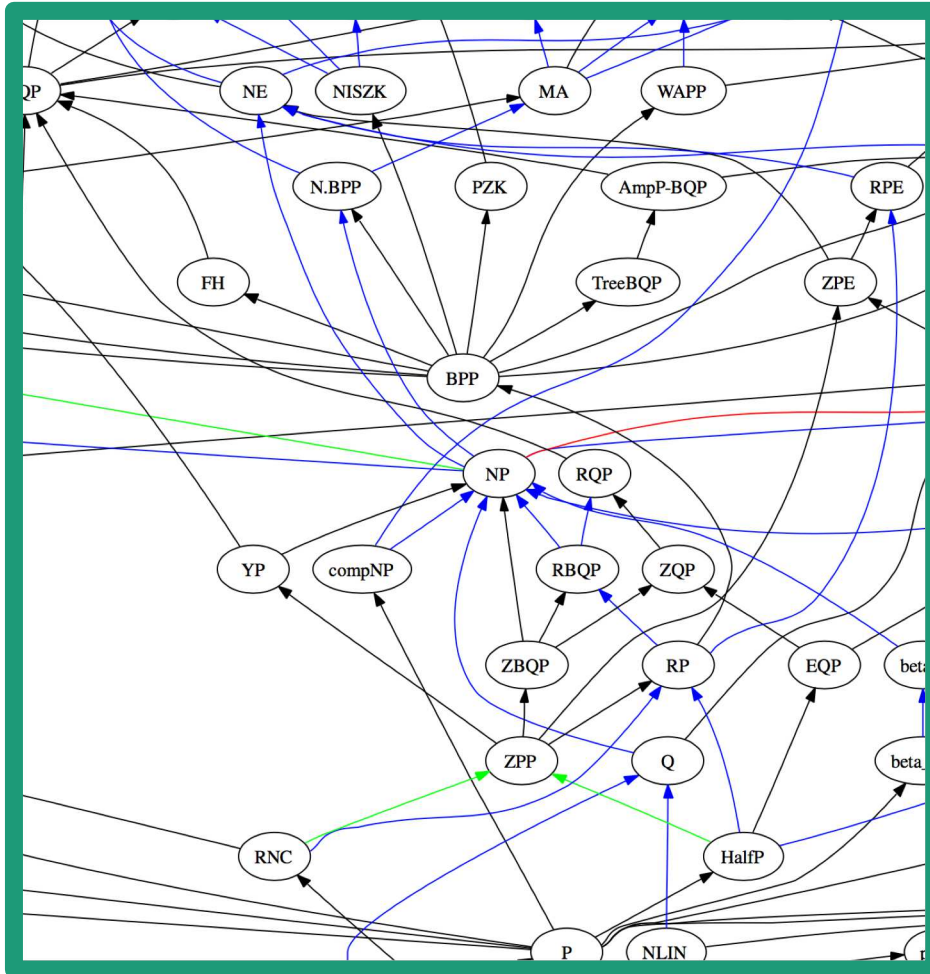
# Graphs

jetblue flights



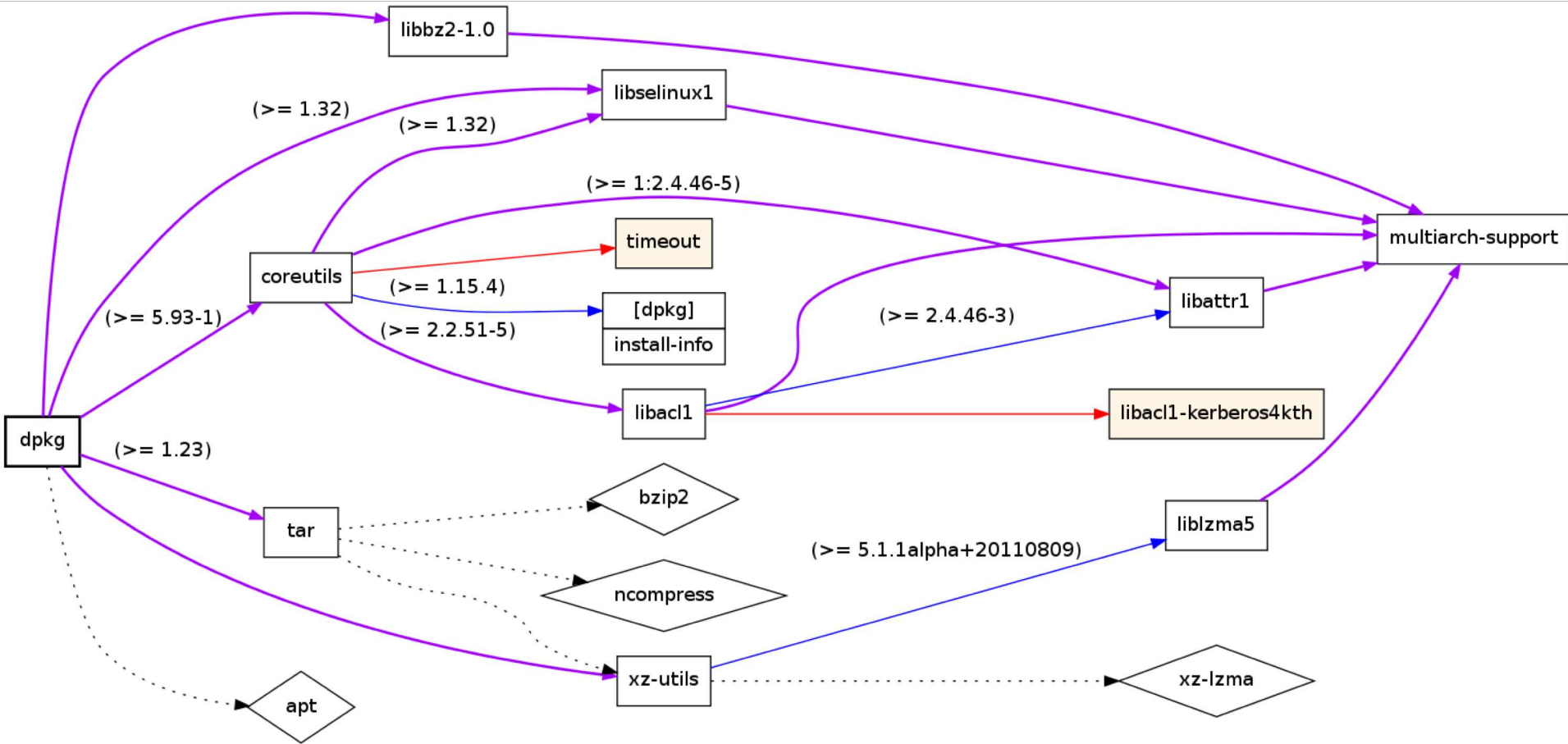
# Graphs

Complexity Zoo  
containment graph



# Graphs

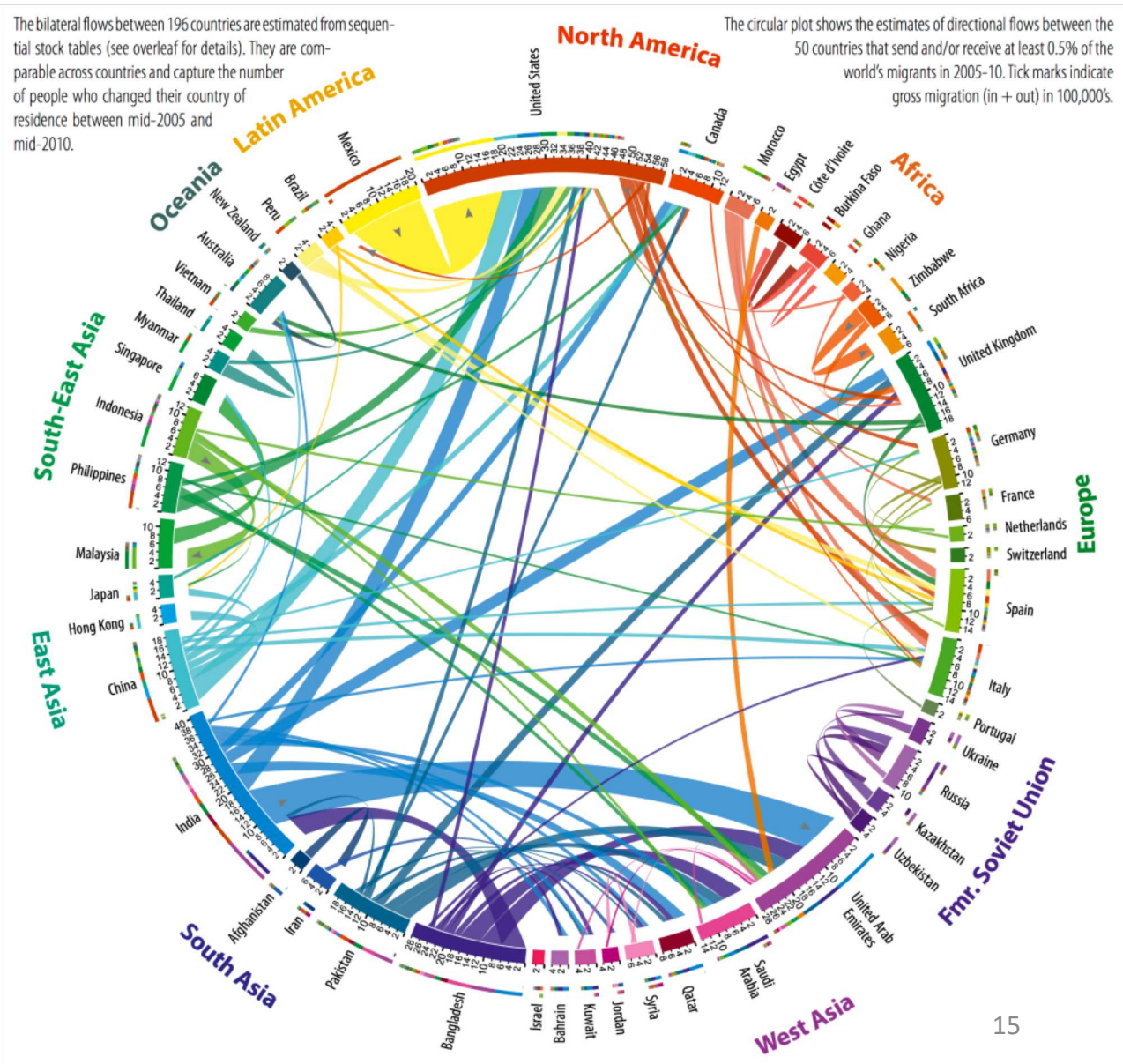
debian dependency (sub)graph



# Graphs

## Immigration flows

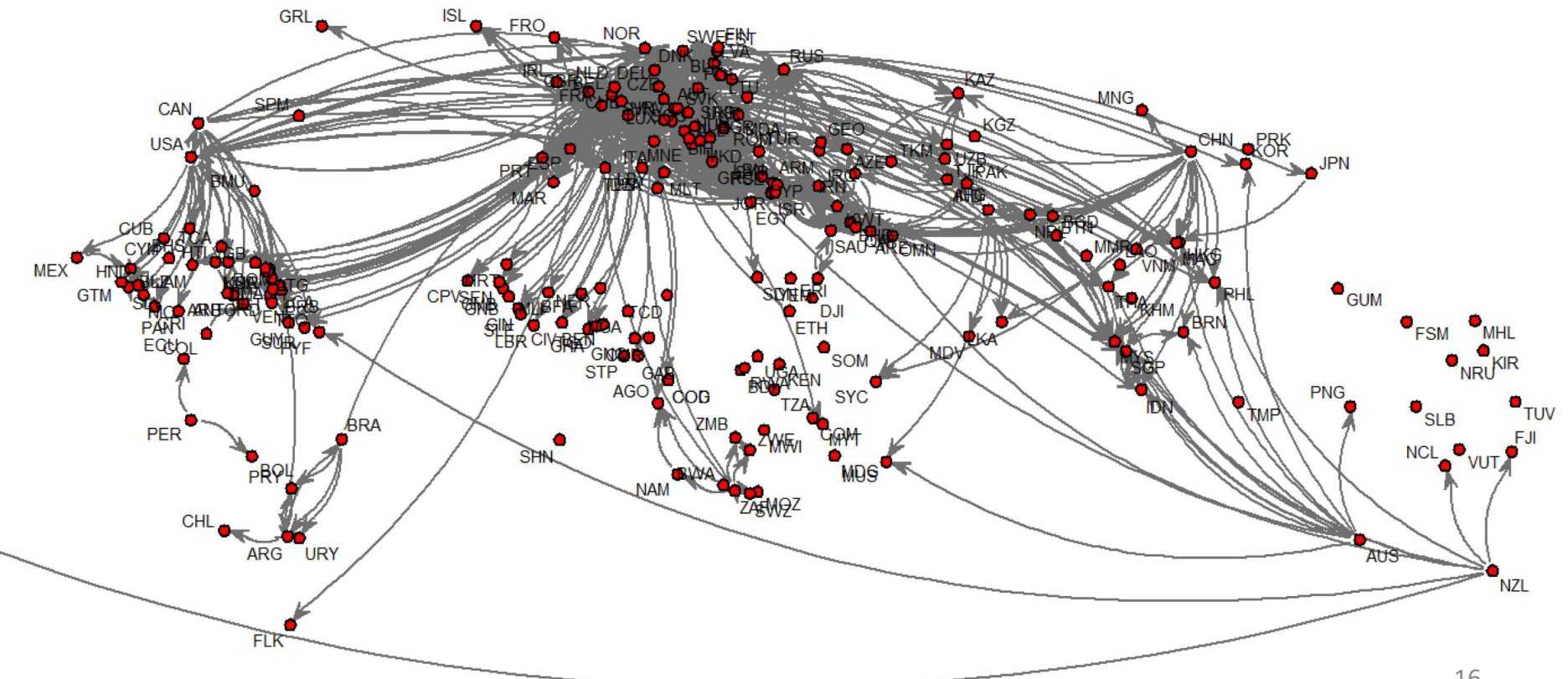
The bilateral flows between 196 countries are estimated from sequential stock tables (see overleaf for details). They are comparable across countries and capture the number of people who changed their country of residence between mid-2005 and mid-2010.



# Graphs

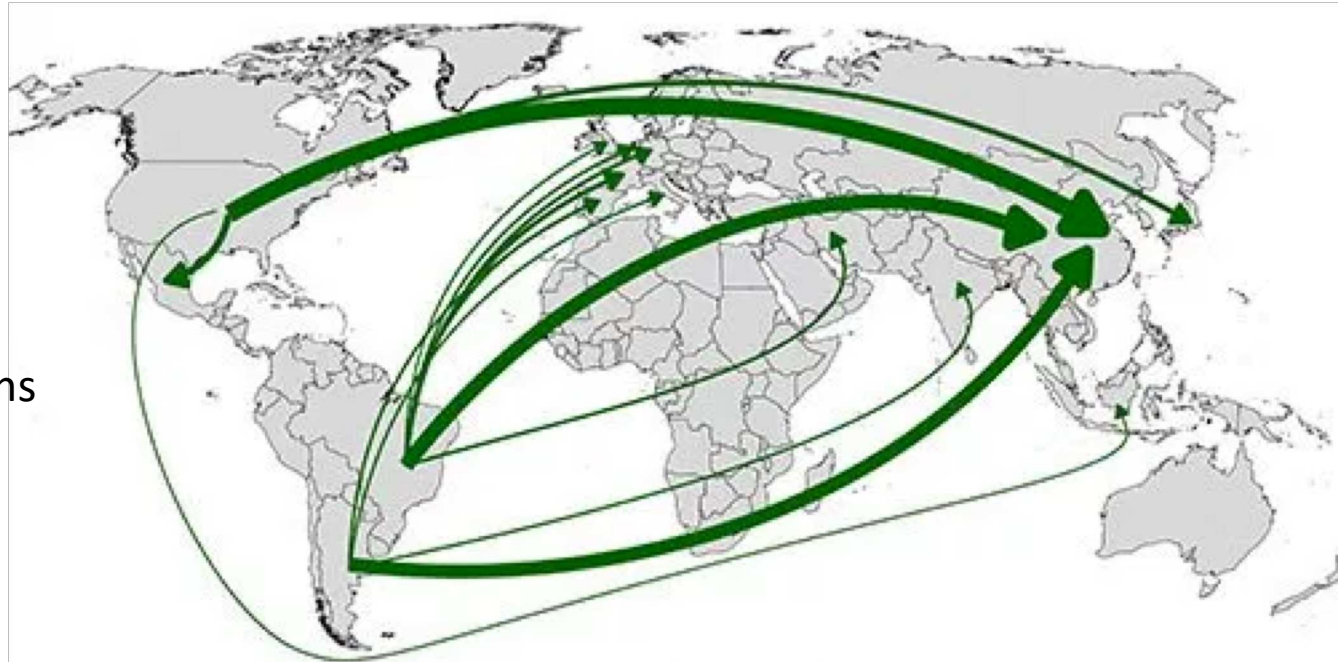
## Potato trade

World trade in fresh potatoes, flows over 0.1 m US\$ average 2005-2009

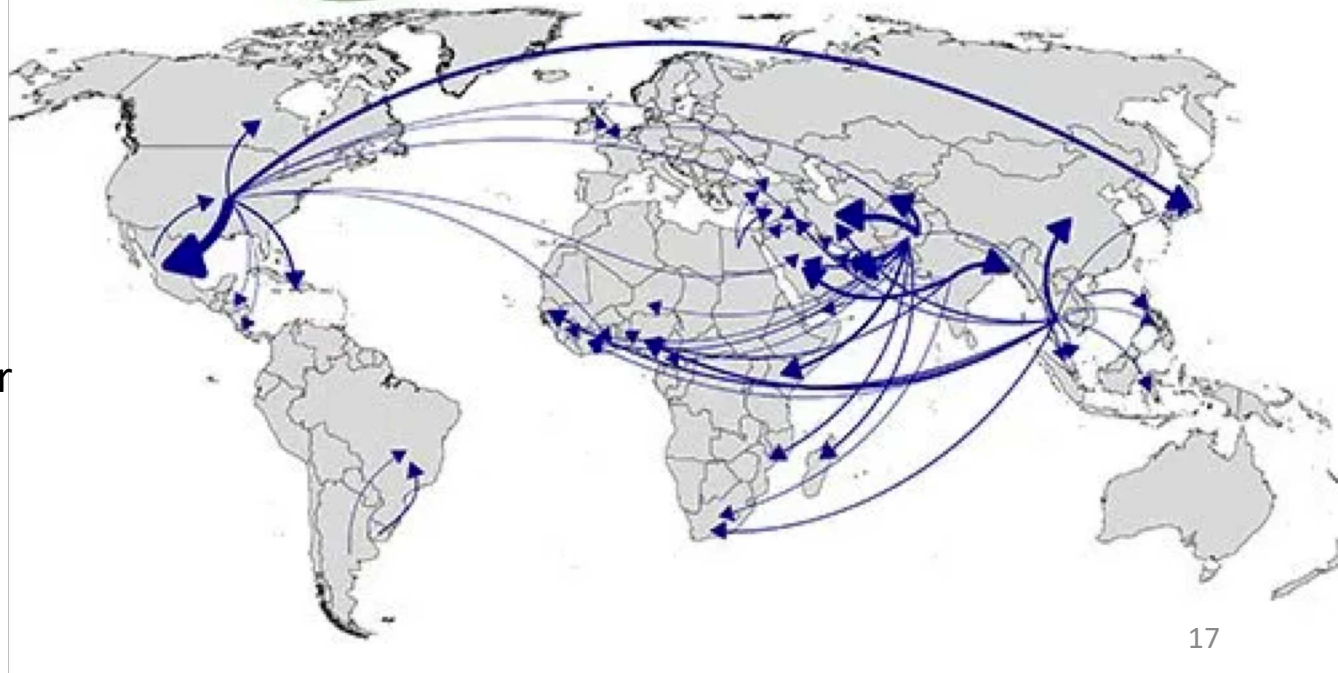


# Graphs

Soybeans

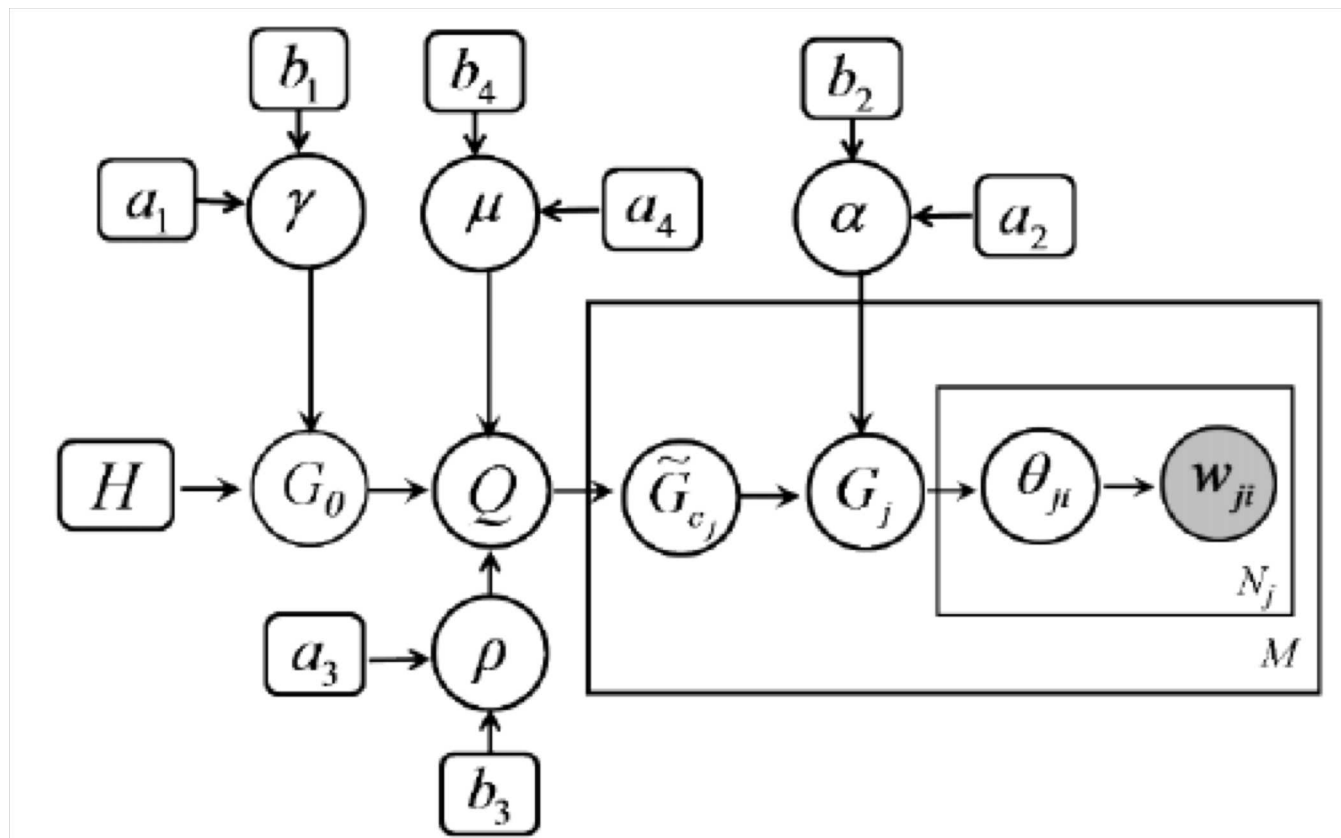


Water



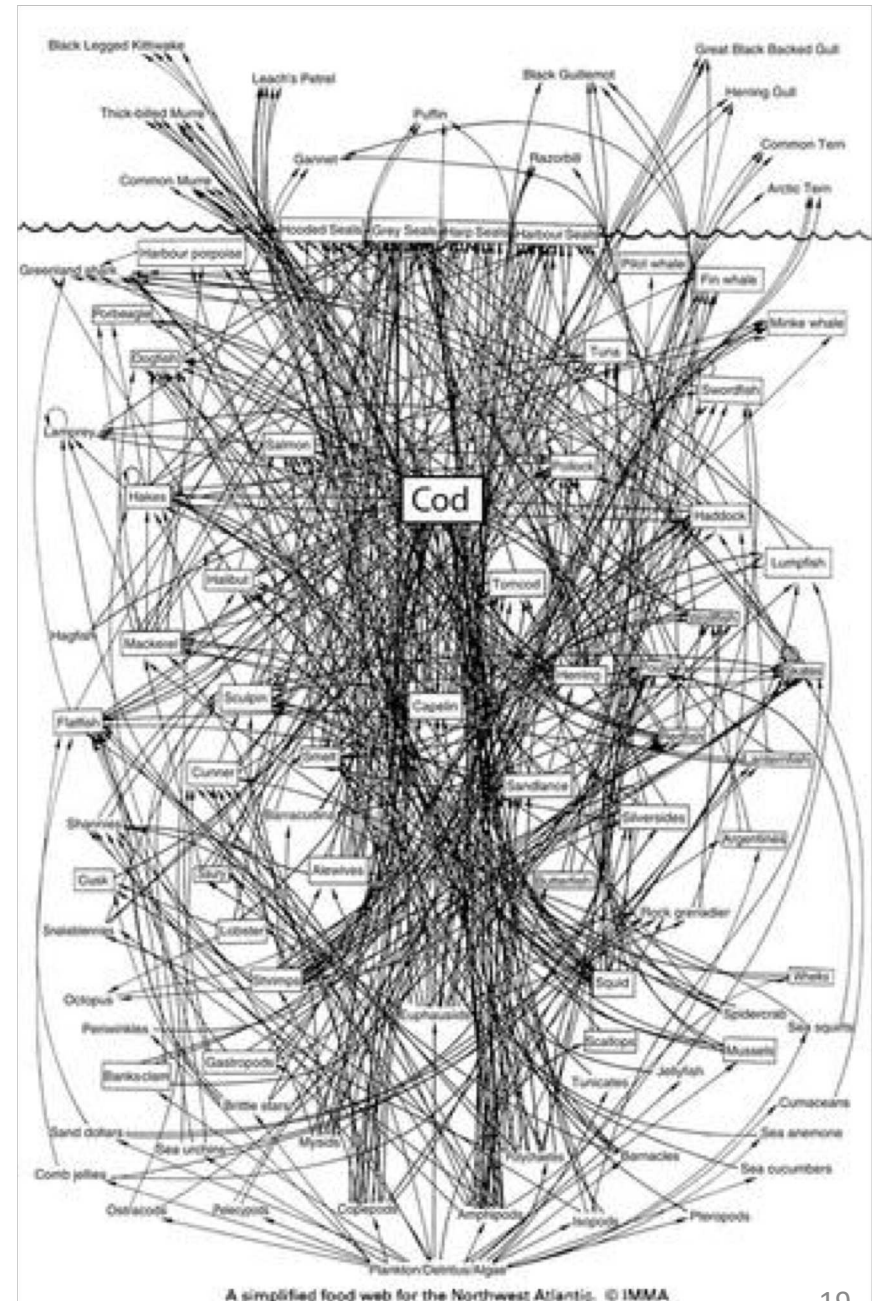
# Graphs

Graphical models



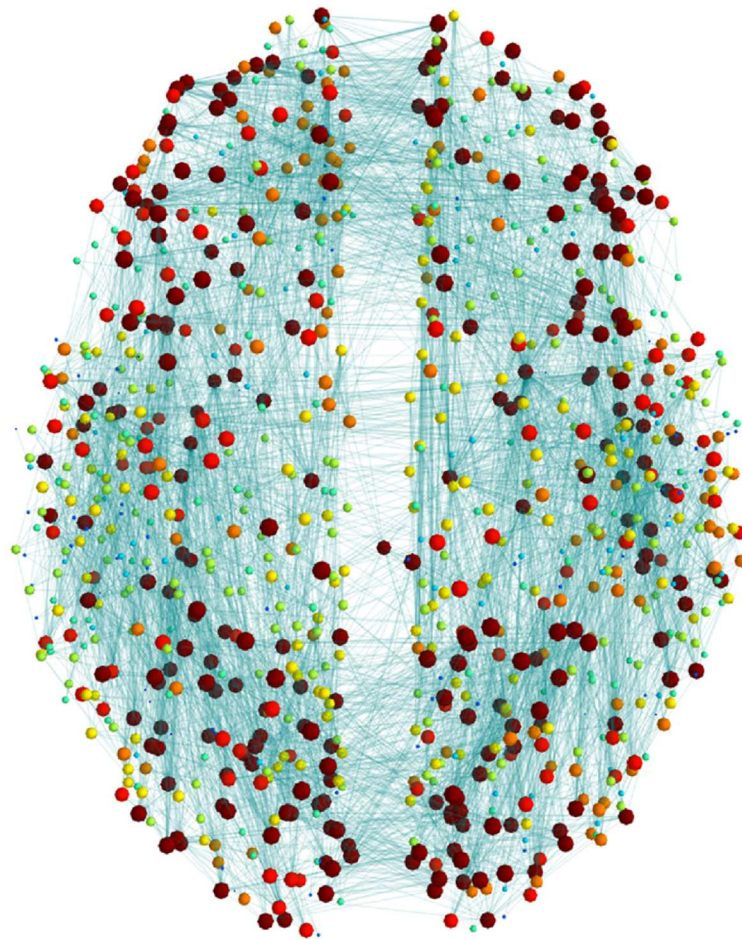
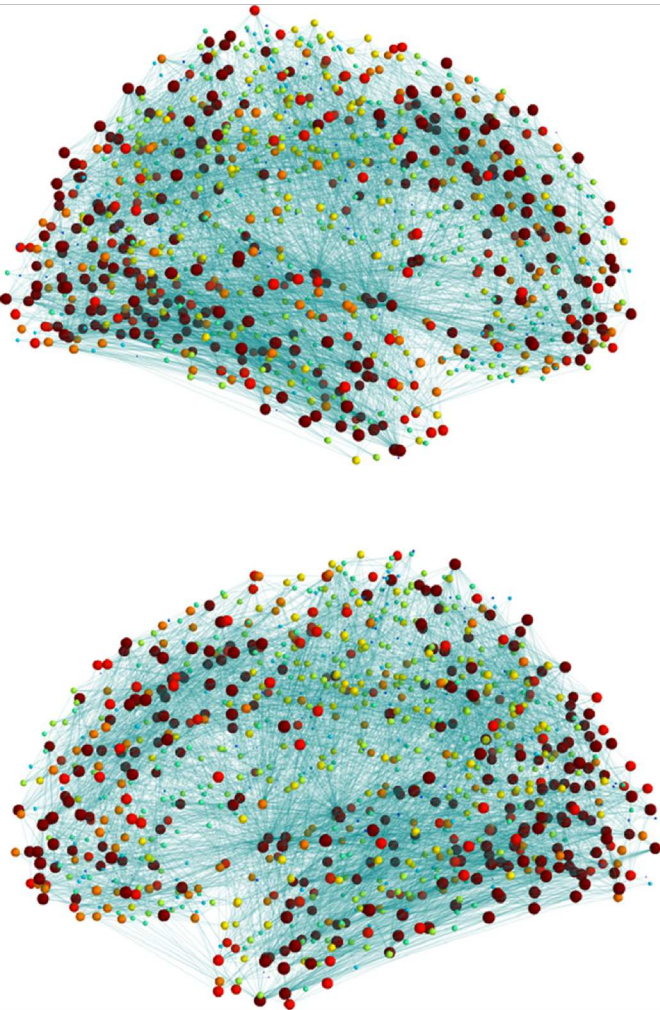
# Graphs

# What eats what in the Atlantic ocean?



# Graphs

Neural connections  
in the brain

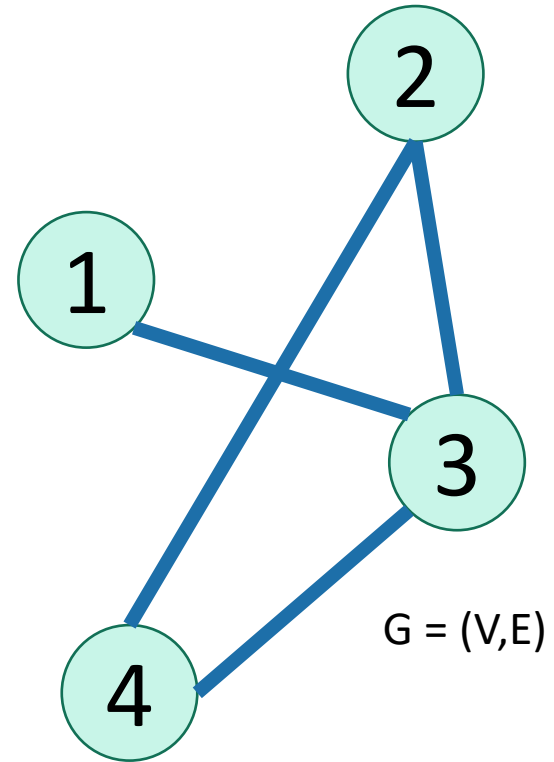


# Graphs

- **There are a lot of graphs.**
- We want to answer questions about them.
  - Efficient routing?
  - Community detection/clustering?
  - From pre-lecture exercise:
    - Computing Bacon numbers
    - Signing up for classes without violating pre-req constraints
    - How to distribute fish in tanks so that none of them will fight.
- This is what we'll do for the next several lectures.

# Undirected Graphs

- Has vertices and edges
  - $V$  is the set of vertices
  - $E$  is the set of edges
  - Formally, a graph is  $G = (V, E)$
- Example
  - $V = \{1, 2, 3, 4\}$
  - $E = \{ \{1, 3\}, \{2, 4\}, \{3, 4\}, \{2, 3\} \}$



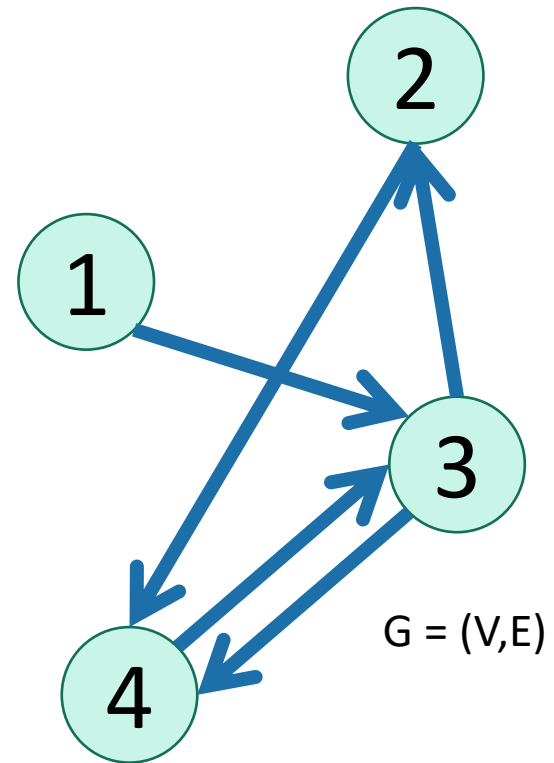
- The **degree** of vertex 4 is 2.
  - There are 2 edges coming out
- Vertex 4's **neighbors** are 2 and 3

# Directed Graphs

- Has vertices and edges
  - $V$  is the set of vertices
  - $E$  is the set of **DIRECTED** edges
  - Formally, a graph is  $G = (V, E)$

- Example

- $V = \{1, 2, 3, 4\}$
- $E = \{ (1, 3), (2, 4), (3, 4), (4, 3), (3, 2) \}$

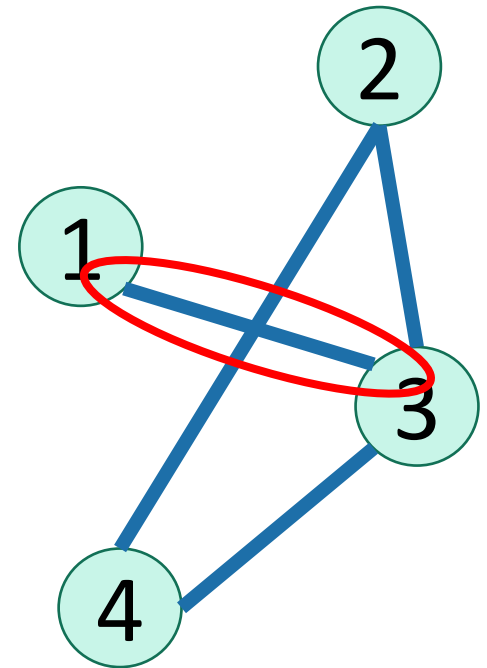


- The **in-degree** of vertex 4 is 2.
- The **out-degree** of vertex 4 is 1.
- Vertex 4's **incoming neighbors** are 2, 3
- Vertex 4's **outgoing neighbor** is 3.

# How do we represent graphs?

- Option 1: adjacency matrix

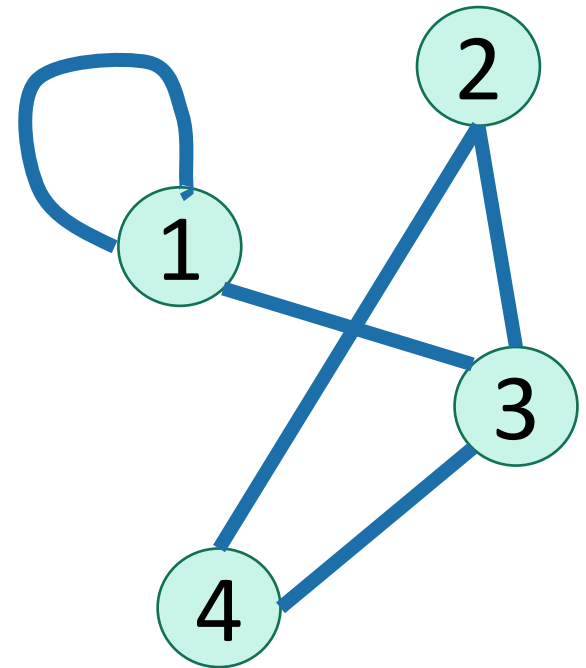
|   | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 0 |
| 2 | 0 | 0 | 1 | 1 |
| 3 | 1 | 1 | 0 | 1 |
| 4 | 0 | 1 | 1 | 0 |



# How do we represent graphs?

- Option 1: adjacency matrix

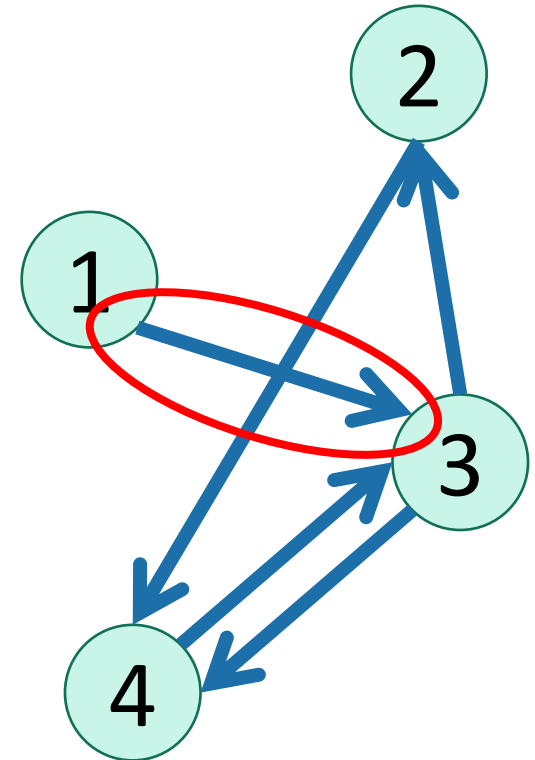
$$\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{bmatrix} \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} \end{bmatrix}$$



# How do we represent graphs?

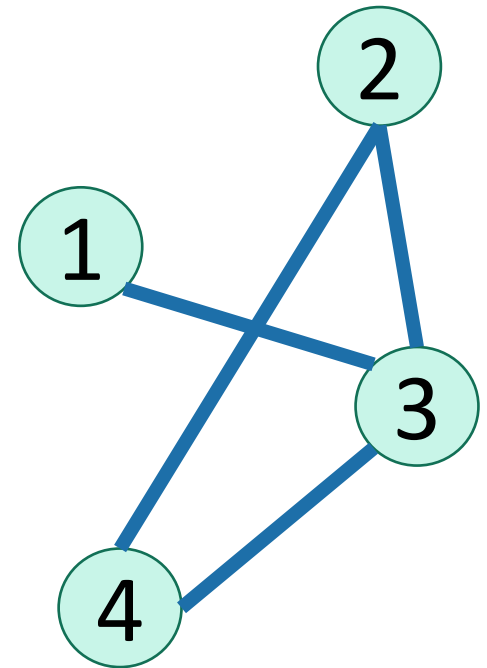
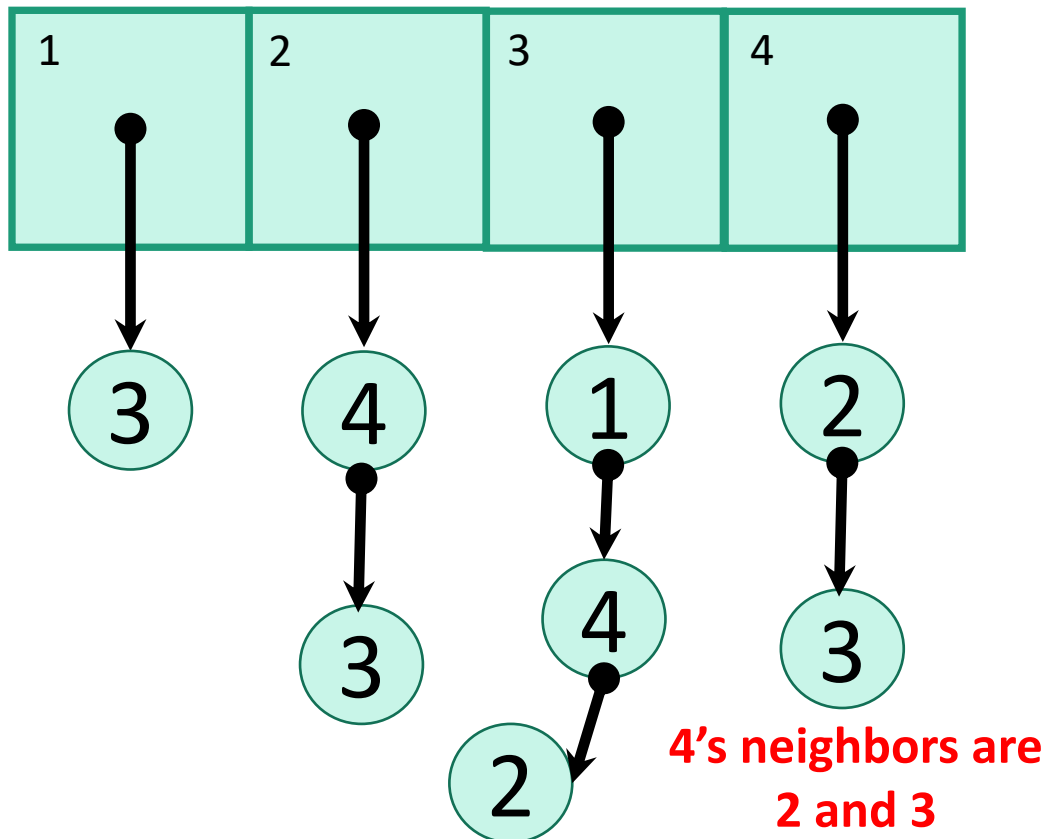
- Option 1: adjacency matrix

|        |   | Destination |   |   |   |
|--------|---|-------------|---|---|---|
|        |   | 1           | 2 | 3 | 4 |
| Source | 1 | 0           | 0 | 1 | 0 |
|        | 2 | 0           | 0 | 0 | 1 |
|        | 3 | 0           | 1 | 0 | 1 |
|        | 4 | 0           | 0 | 1 | 0 |



# How do we represent graphs?

- Option 2: adjacency lists.



How would you modify this for directed graphs?



# In either case

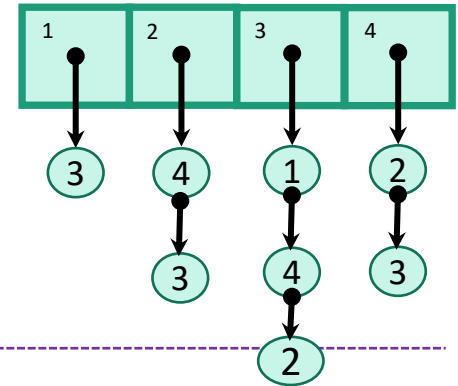
- Vertices can store other information
  - Attributes (name, IP address, ...)
  - helper info for algorithms that we will perform on the graph
- Want to be able to do the following operations:
  - **Edge Membership**: Is edge  $e$  in  $E$ ?
  - **Neighbor Query**: What are the neighbors of vertex  $v$ ?

# Trade-offs

Say there are  $n$  vertices  
and  $m$  edges.

$$\begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

Generally better for **sparse**  
graphs (where  $m \ll n^2$ )



Edge membership  
Is  $e = \{v, w\}$  in  $E$ ?

$O(1)$

$O(\deg(v))$  or  
 $O(\deg(w))$

Neighbor query  
Give me  $v$ 's neighbors.

$O(n)$

$O(\deg(v))$

Space requirements

$O(n^2)$

$O(n + m)$

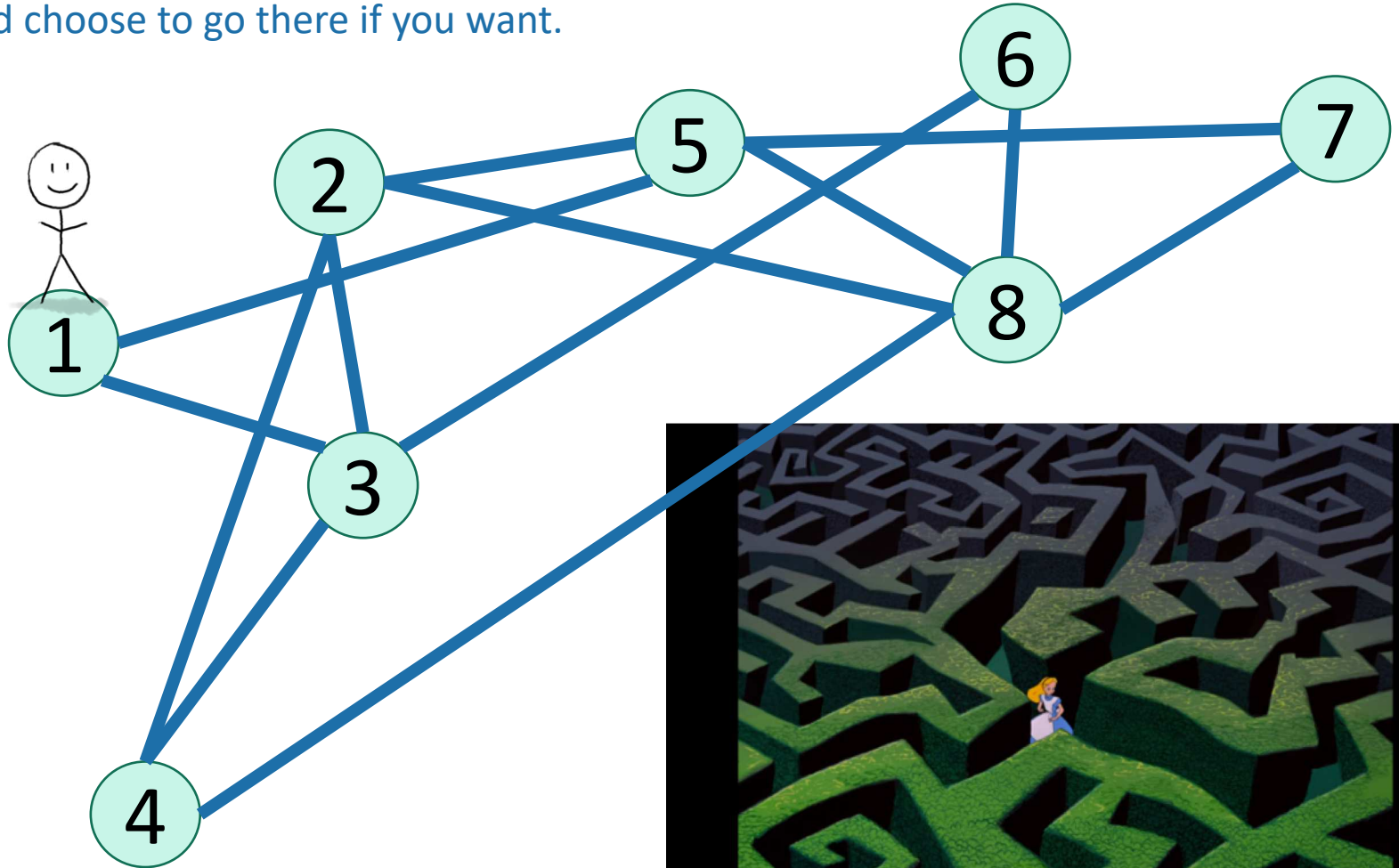
See Lecture 9 IPython notebook for an actual  
implementation!

We'll assume this  
representation for  
the rest of the class

# Part 1: Depth-first search

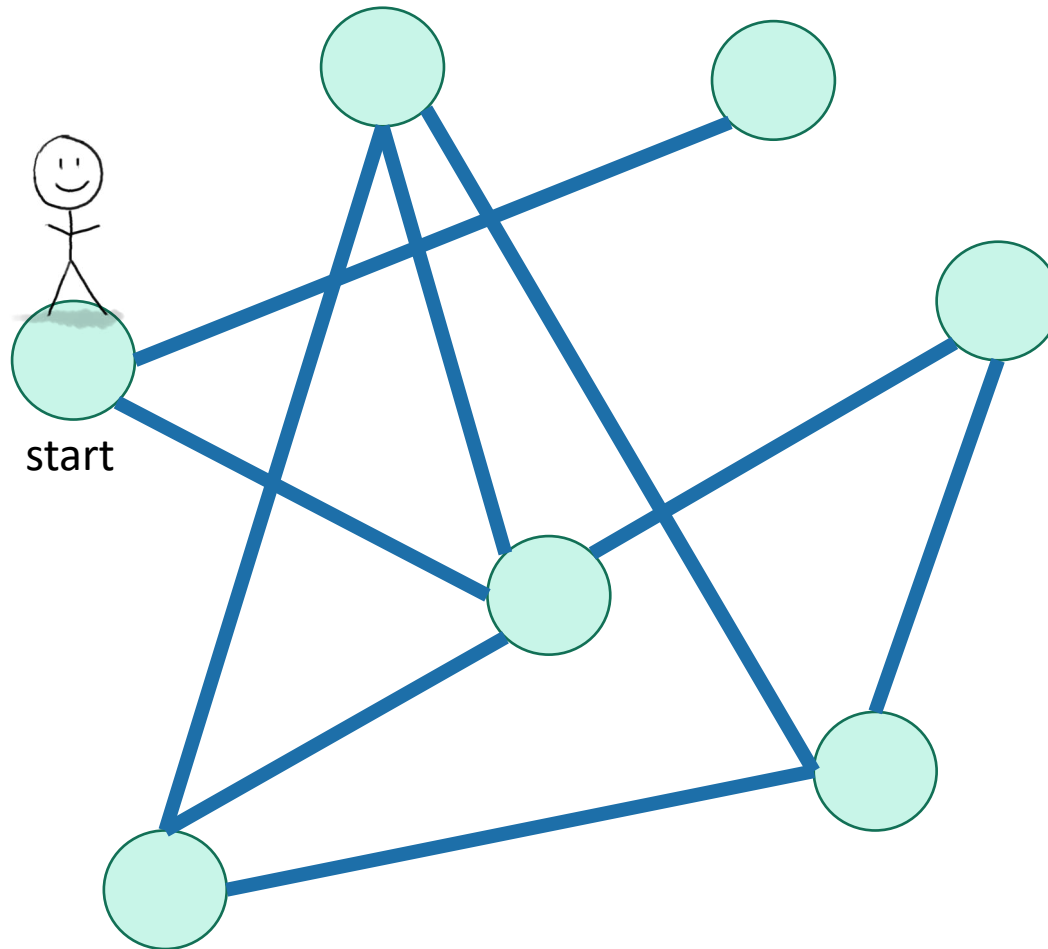
# How do we explore a graph?

At each node, you can get a list of neighbors,  
and choose to go there if you want.



# Depth First Search

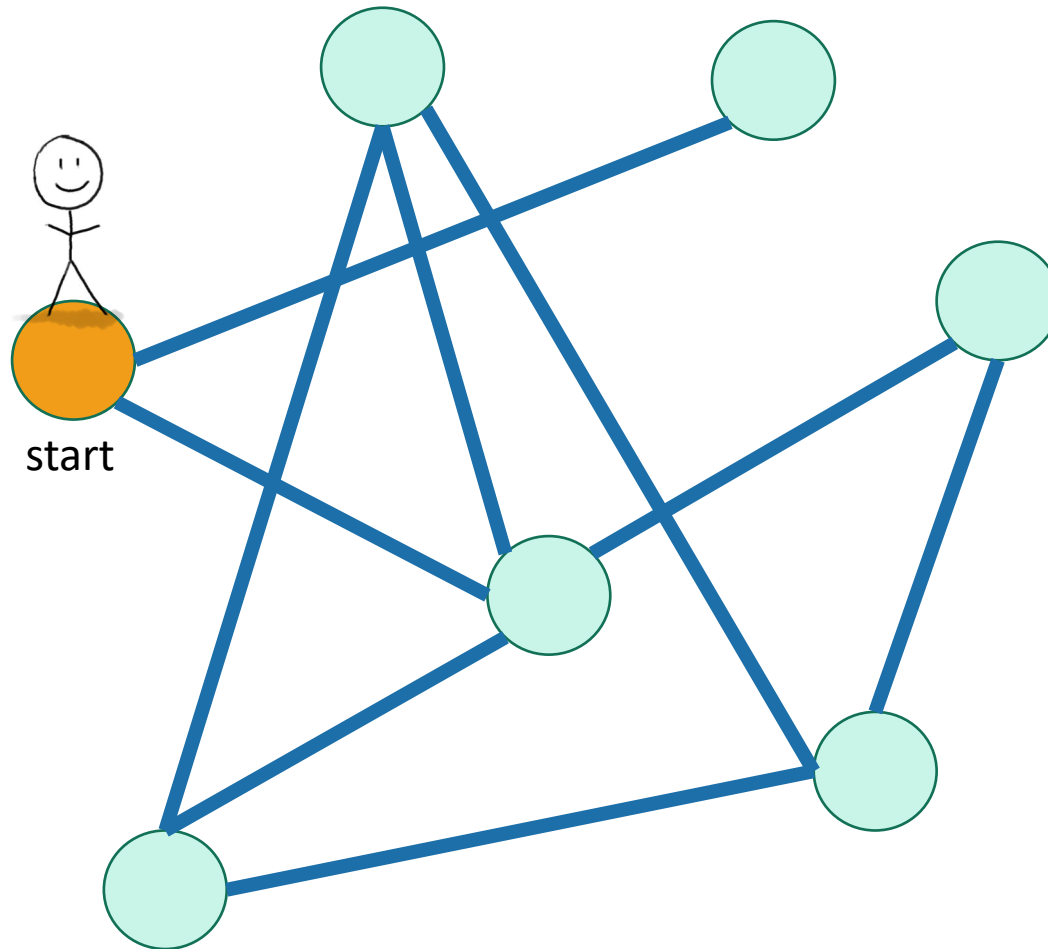
Exploring a labyrinth with chalk and a piece of string



-  Not been there yet
-  Been there, haven't explored all the paths out.
-  Been there, have explored all the paths out.

# Depth First Search

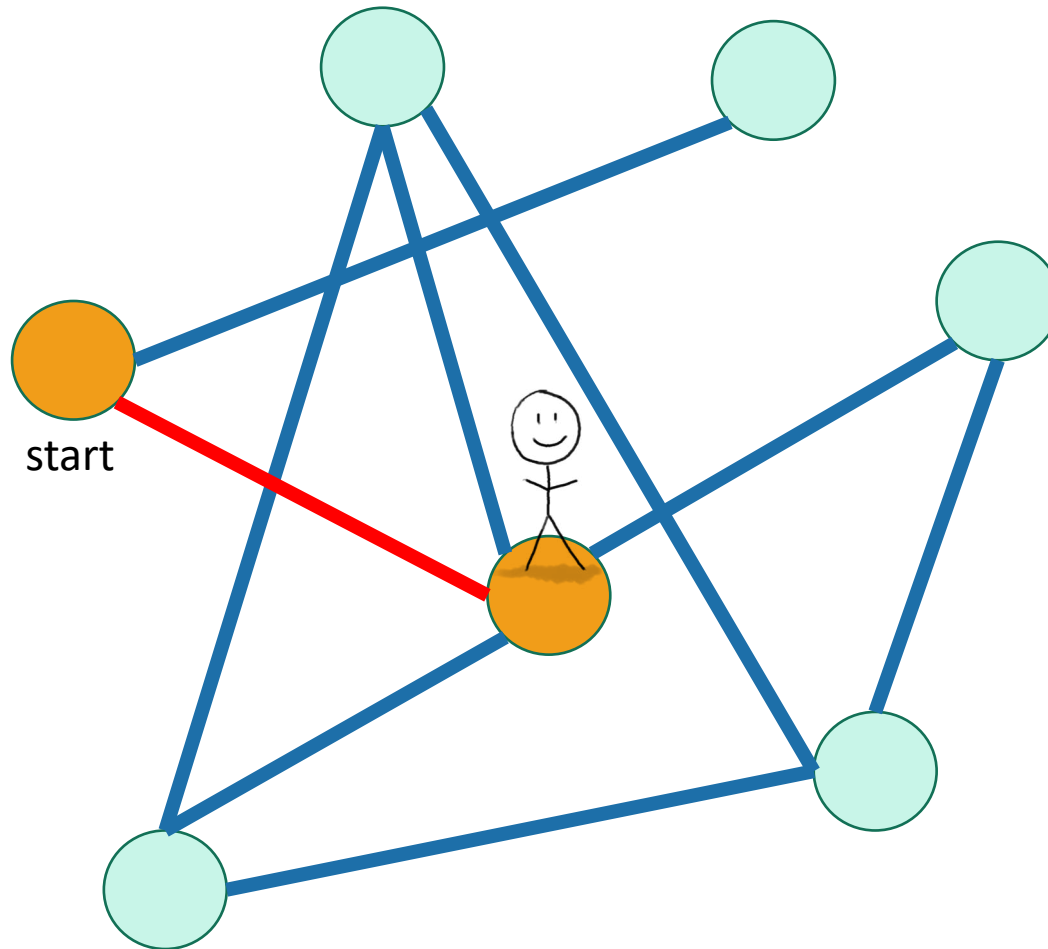
Exploring a labyrinth with chalk and a piece of string



-  Not been there yet
-  Been there, haven't explored all the paths out.
-  Been there, have explored all the paths out.

# Depth First Search

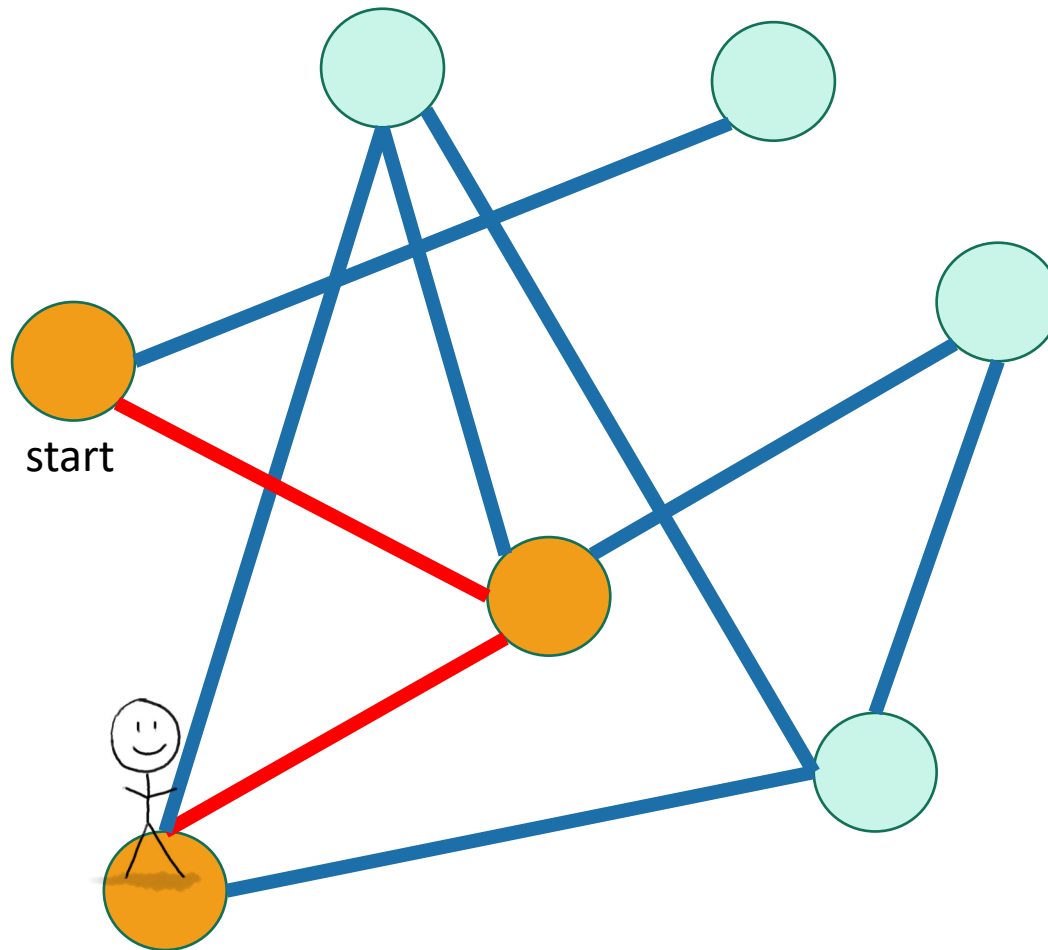
Exploring a labyrinth with chalk and a piece of string



-  Not been there yet
-  Been there, haven't explored all the paths out.
-  Been there, have explored all the paths out.

# Depth First Search

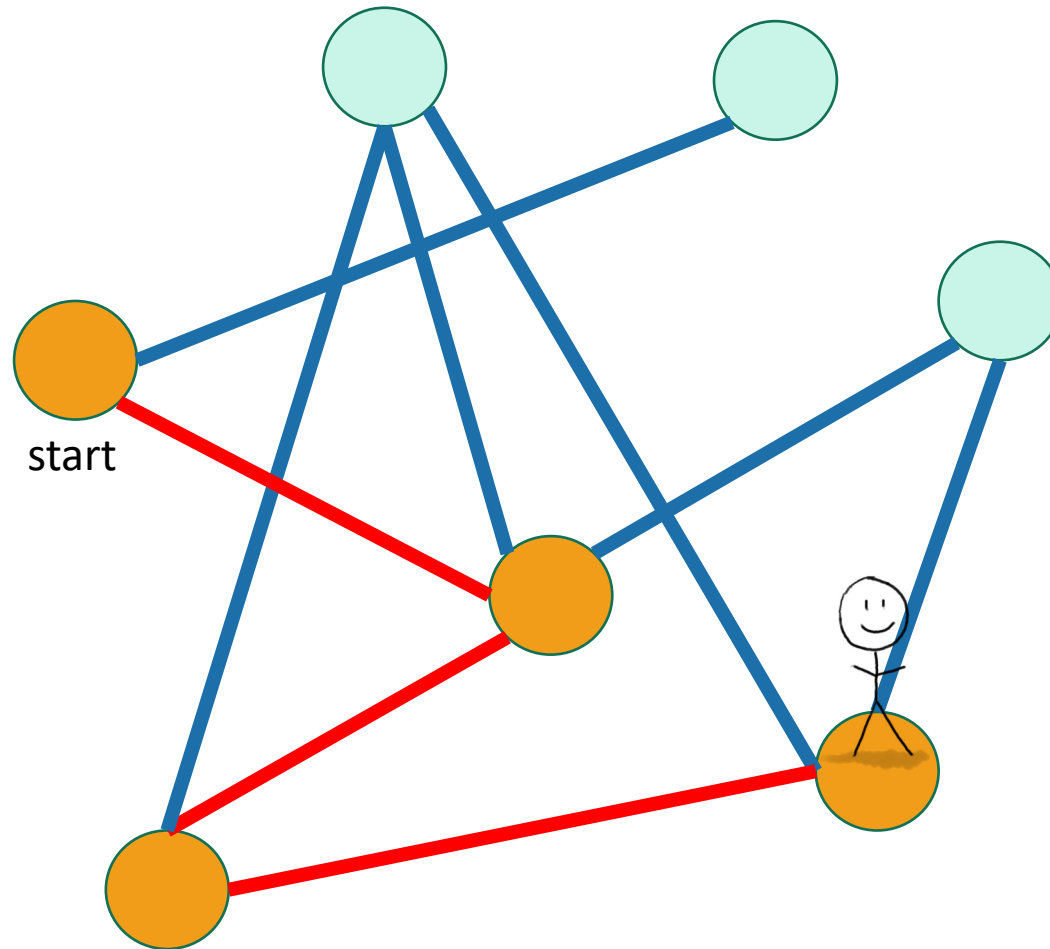
Exploring a labyrinth with chalk and a piece of string



-  Not been there yet
-  Been there, haven't explored all the paths out.
-  Been there, have explored all the paths out.

# Depth First Search

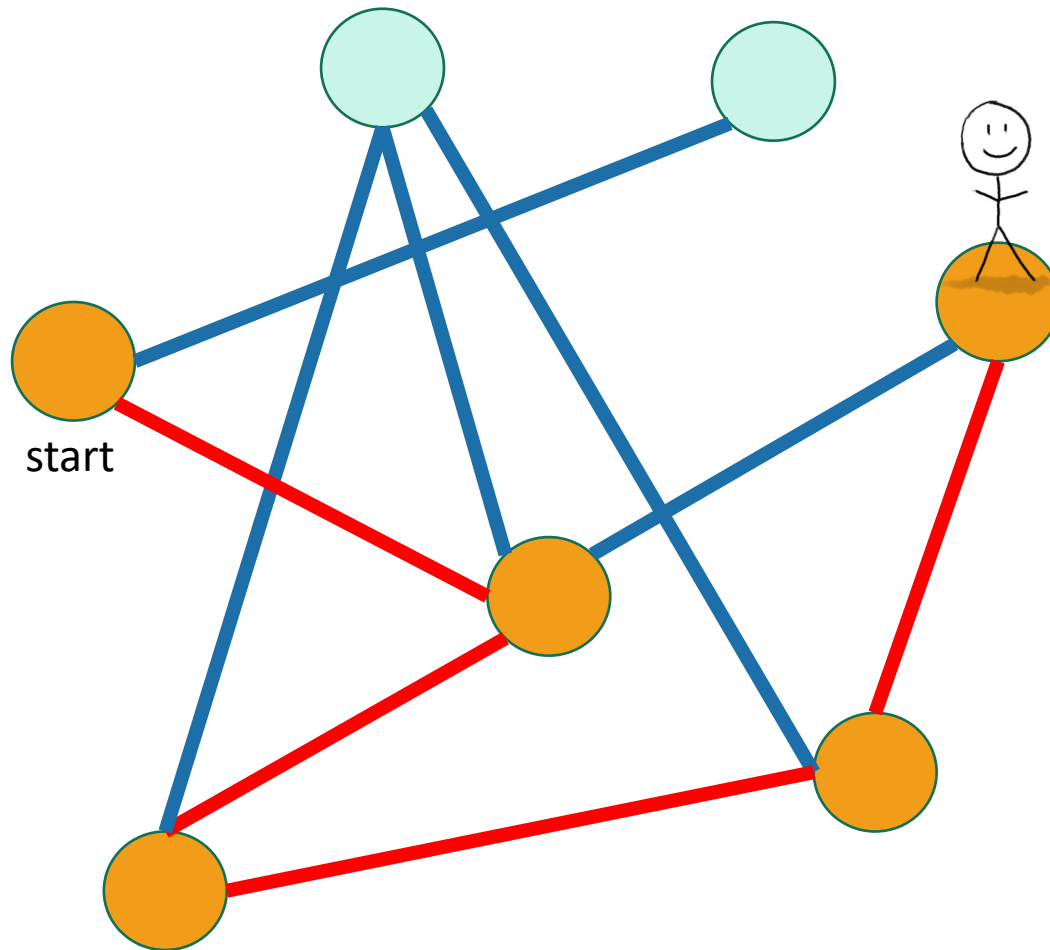
Exploring a labyrinth with chalk and a piece of string



-  Not been there yet
-  Been there, haven't explored all the paths out.
-  Been there, have explored all the paths out.

# Depth First Search

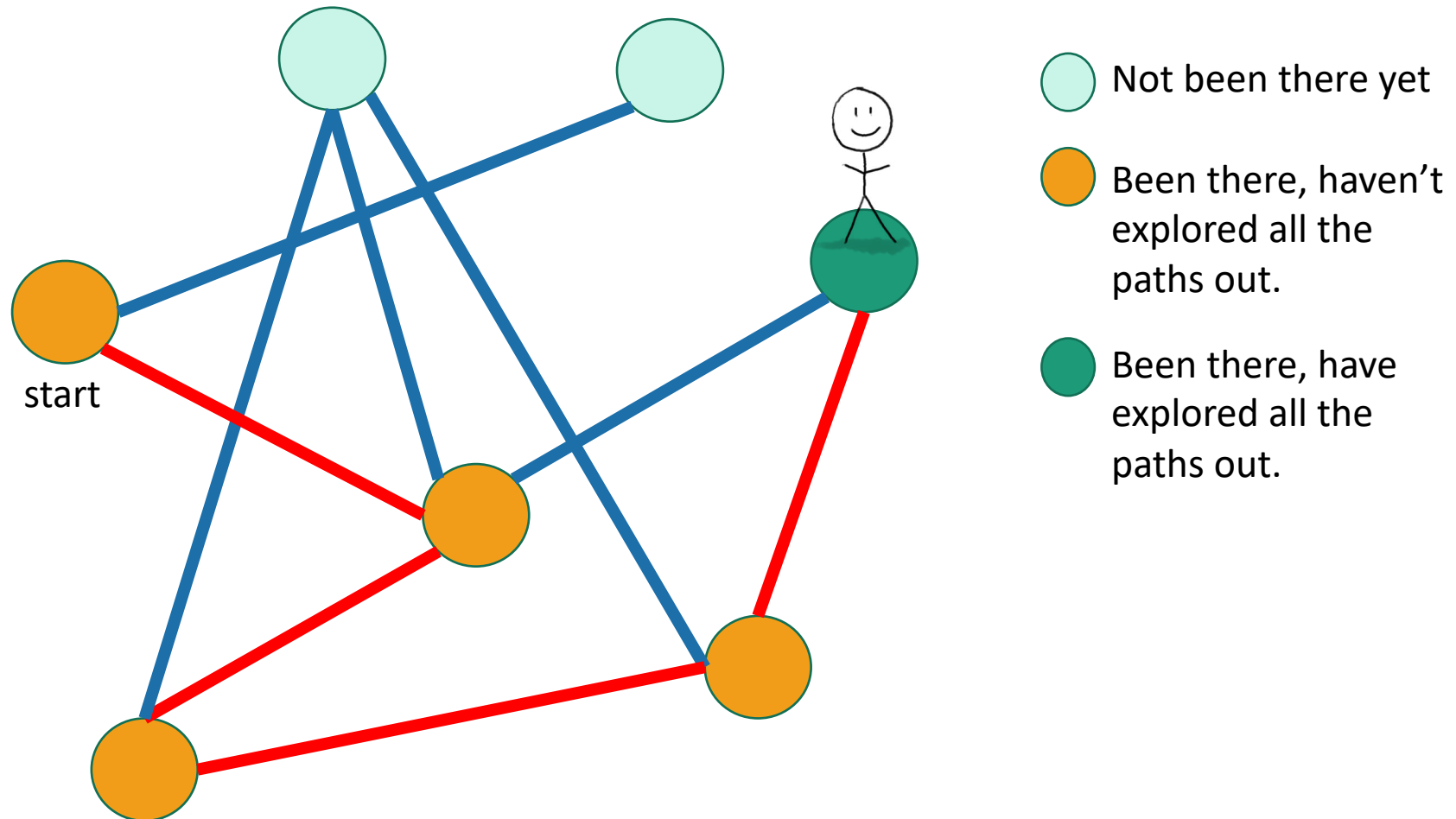
Exploring a labyrinth with chalk and a piece of string



-  Not been there yet
-  Been there, haven't explored all the paths out.
-  Been there, have explored all the paths out.

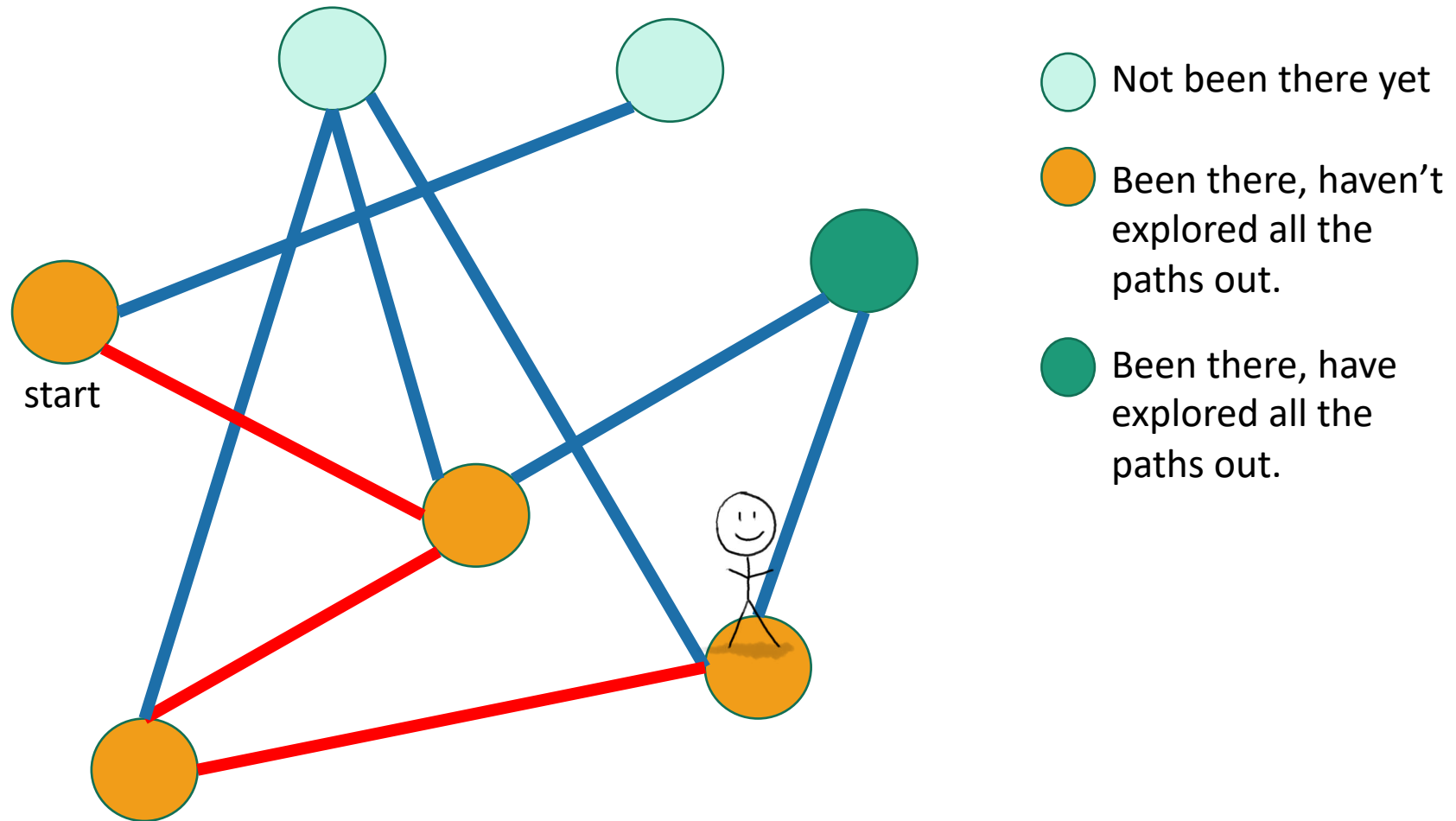
# Depth First Search

Exploring a labyrinth with chalk and a piece of string



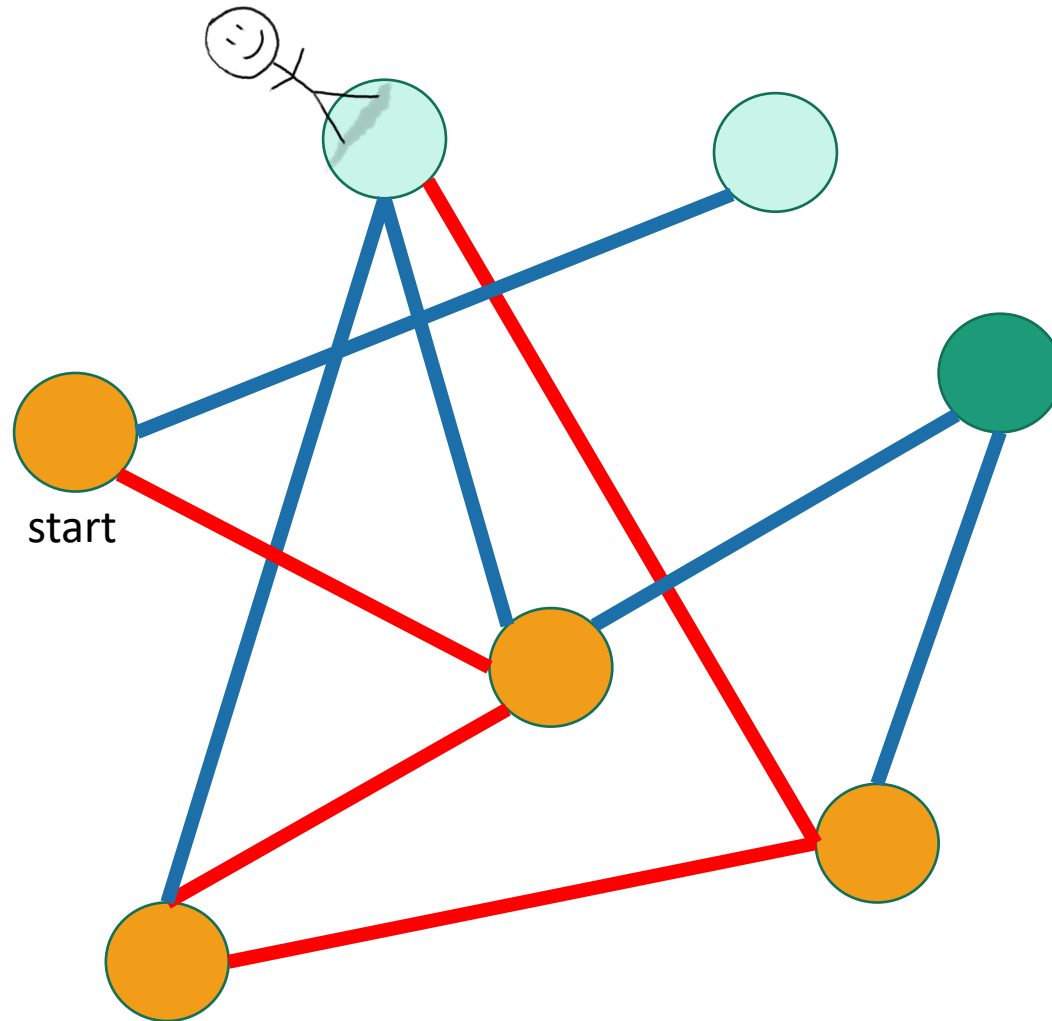
# Depth First Search

Exploring a labyrinth with chalk and a piece of string



# Depth First Search

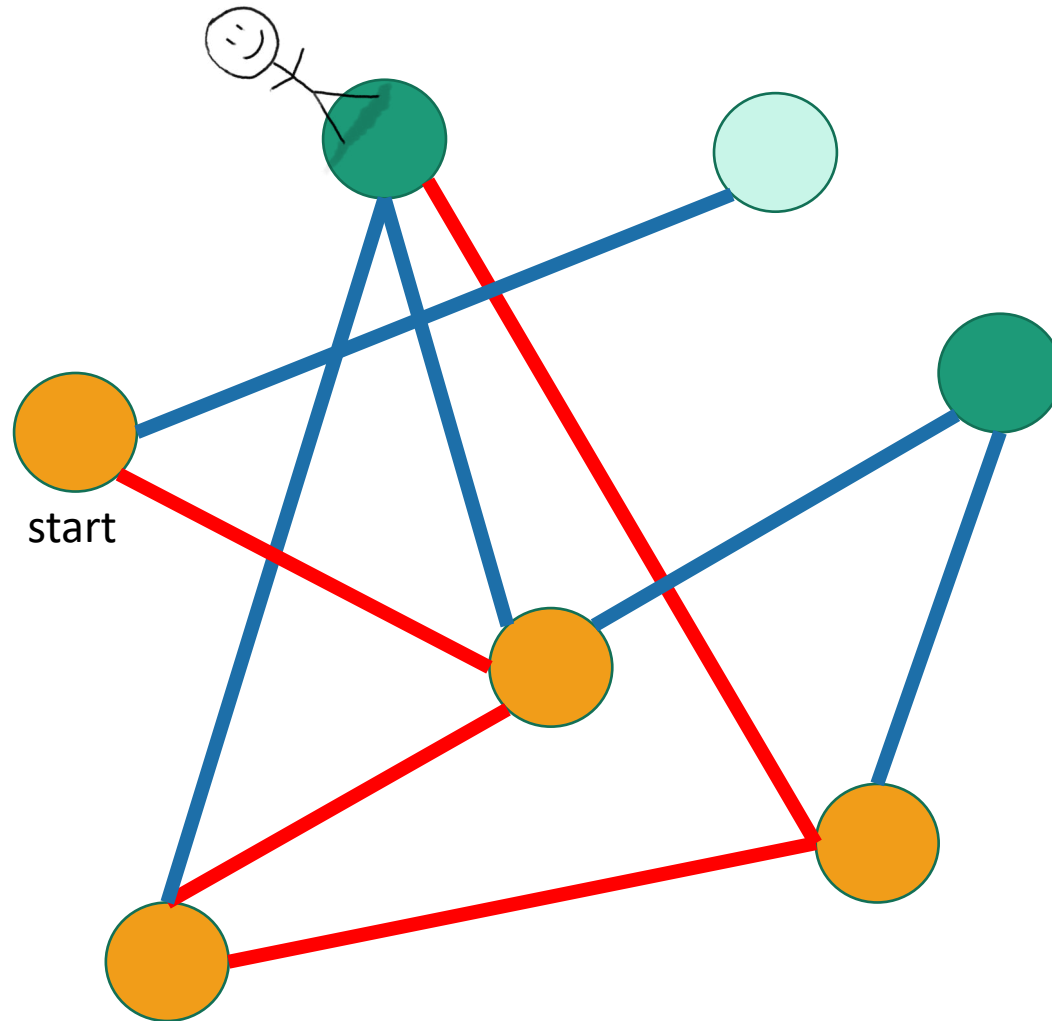
Exploring a labyrinth with chalk and a piece of string



-  Not been there yet
-  Been there, haven't explored all the paths out.
-  Been there, have explored all the paths out.

# Depth First Search

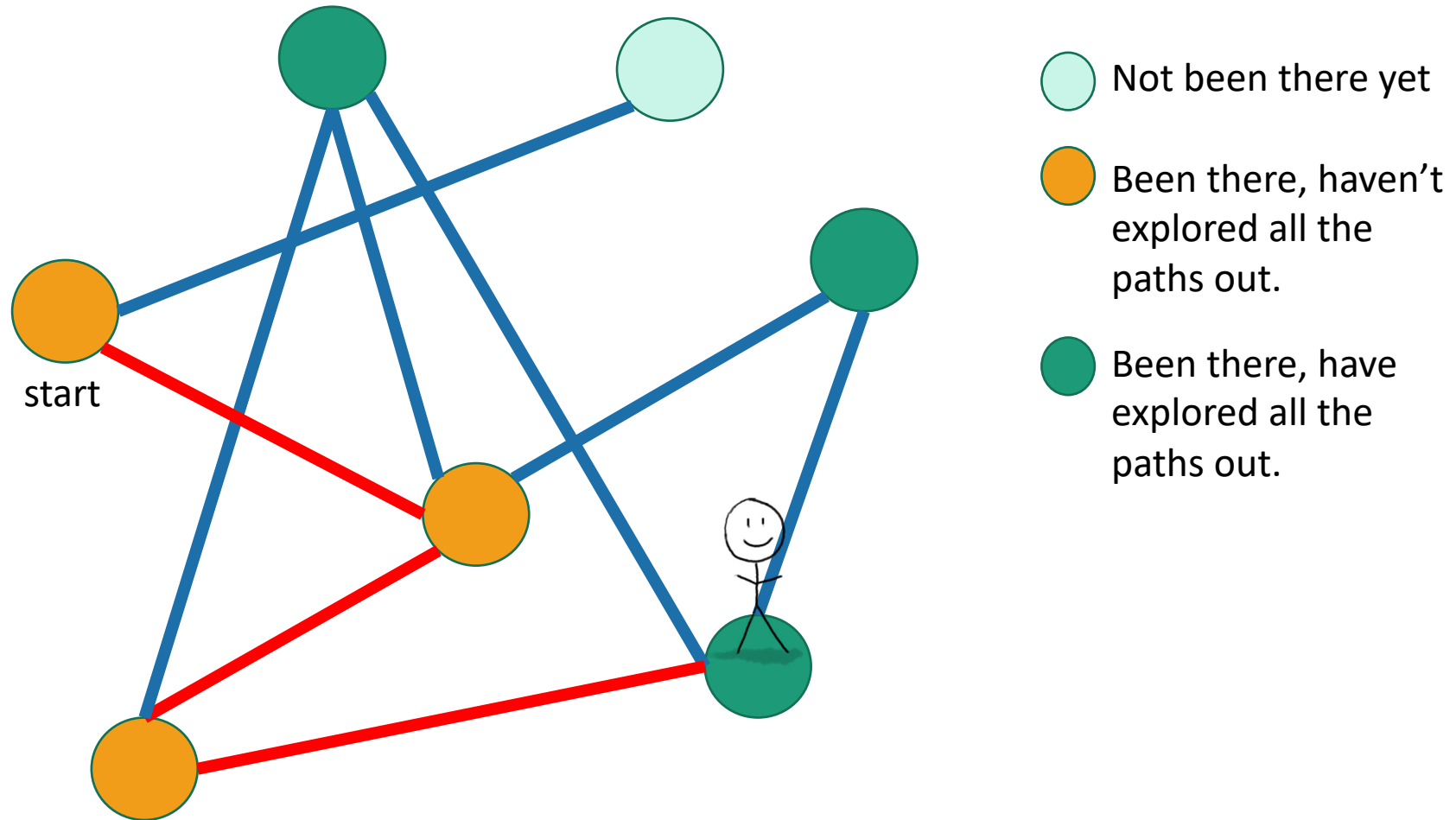
Exploring a labyrinth with chalk and a piece of string



-  Not been there yet
-  Been there, haven't explored all the paths out.
-  Been there, have explored all the paths out.

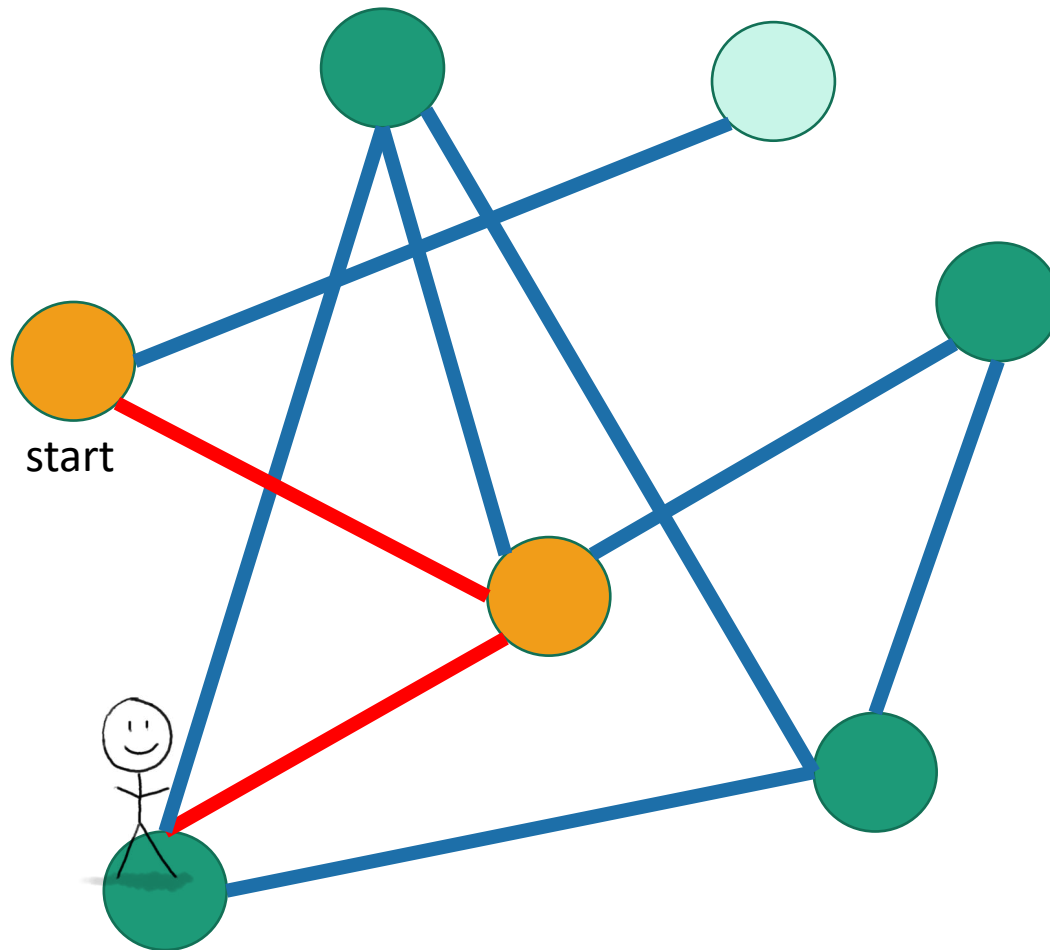
# Depth First Search

Exploring a labyrinth with chalk and a piece of string



# Depth First Search

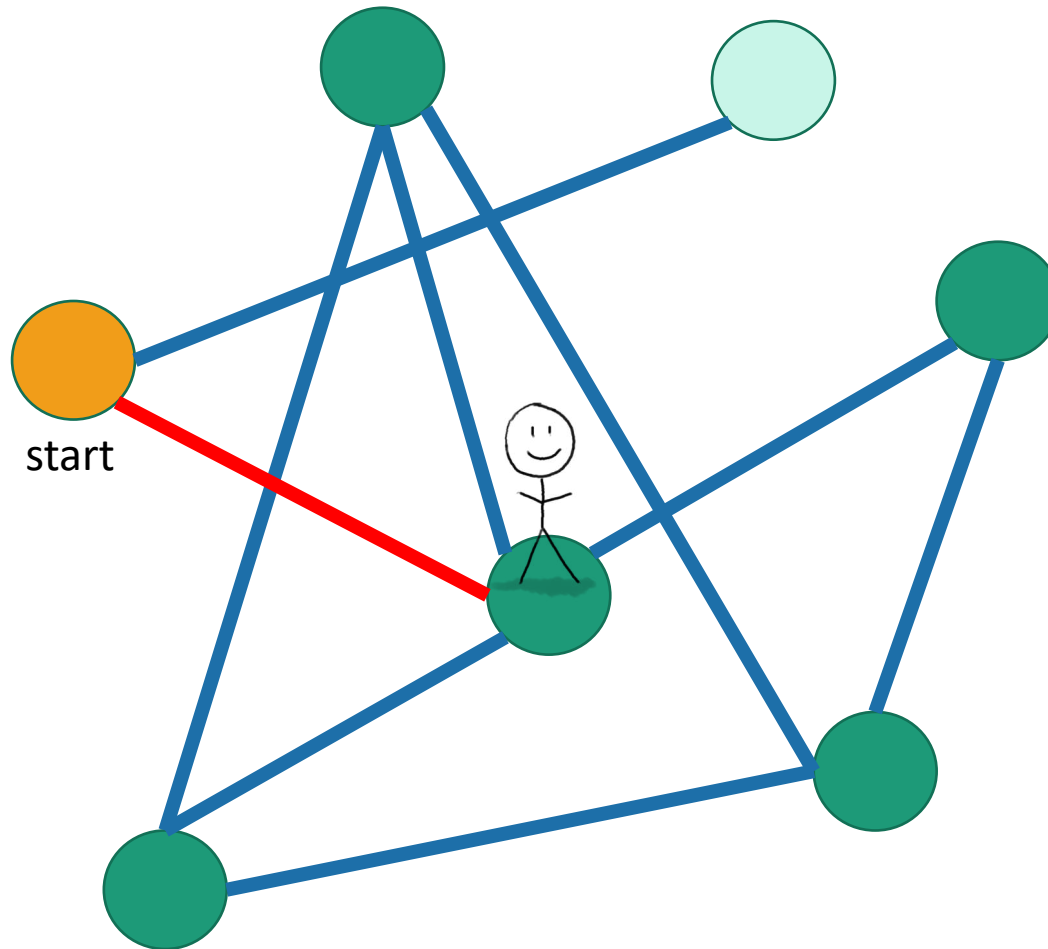
Exploring a labyrinth with chalk and a piece of string



-  Not been there yet
-  Been there, haven't explored all the paths out.
-  Been there, have explored all the paths out.

# Depth First Search

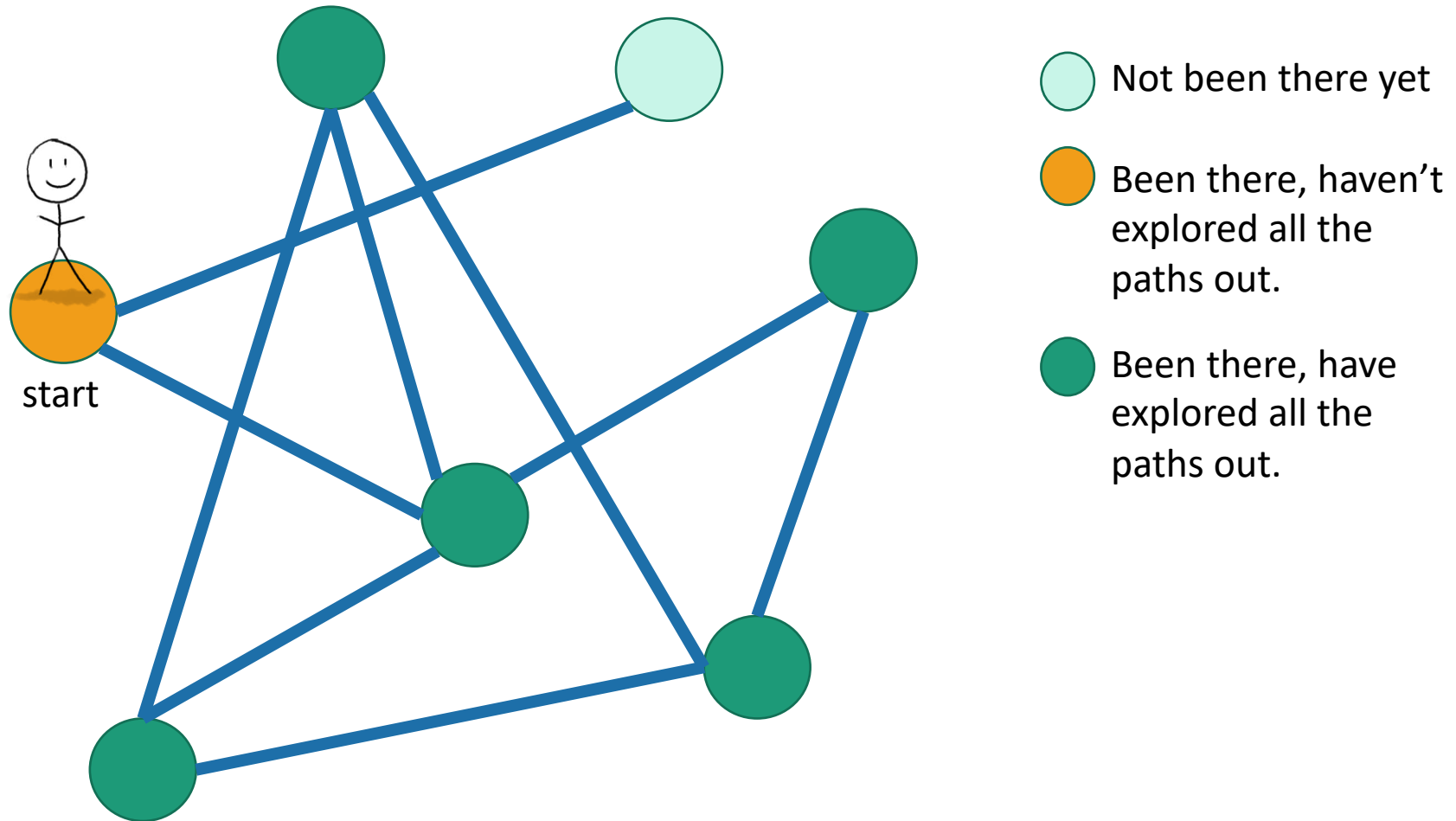
Exploring a labyrinth with chalk and a piece of string



-  Not been there yet
-  Been there, haven't explored all the paths out.
-  Been there, have explored all the paths out.

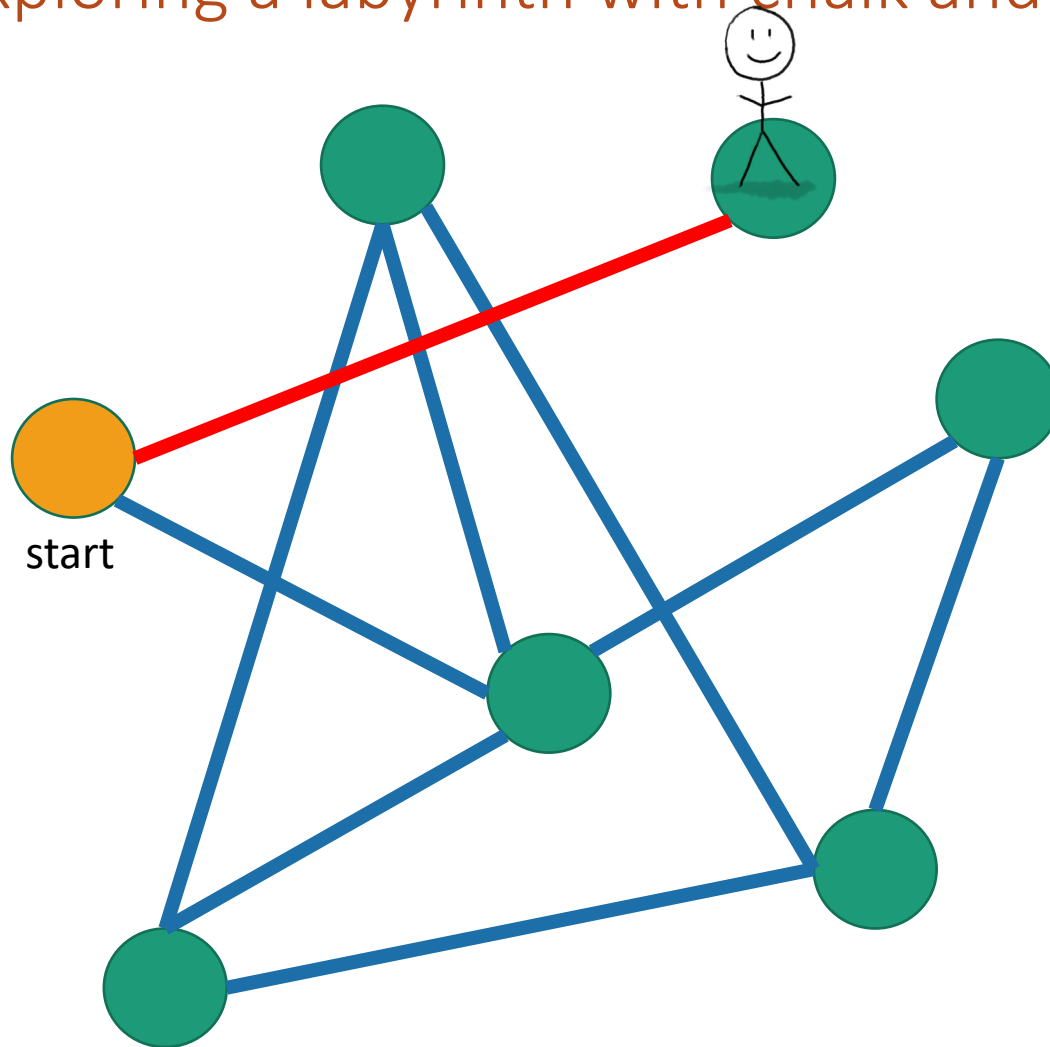
# Depth First Search

Exploring a labyrinth with chalk and a piece of string



# Depth First Search

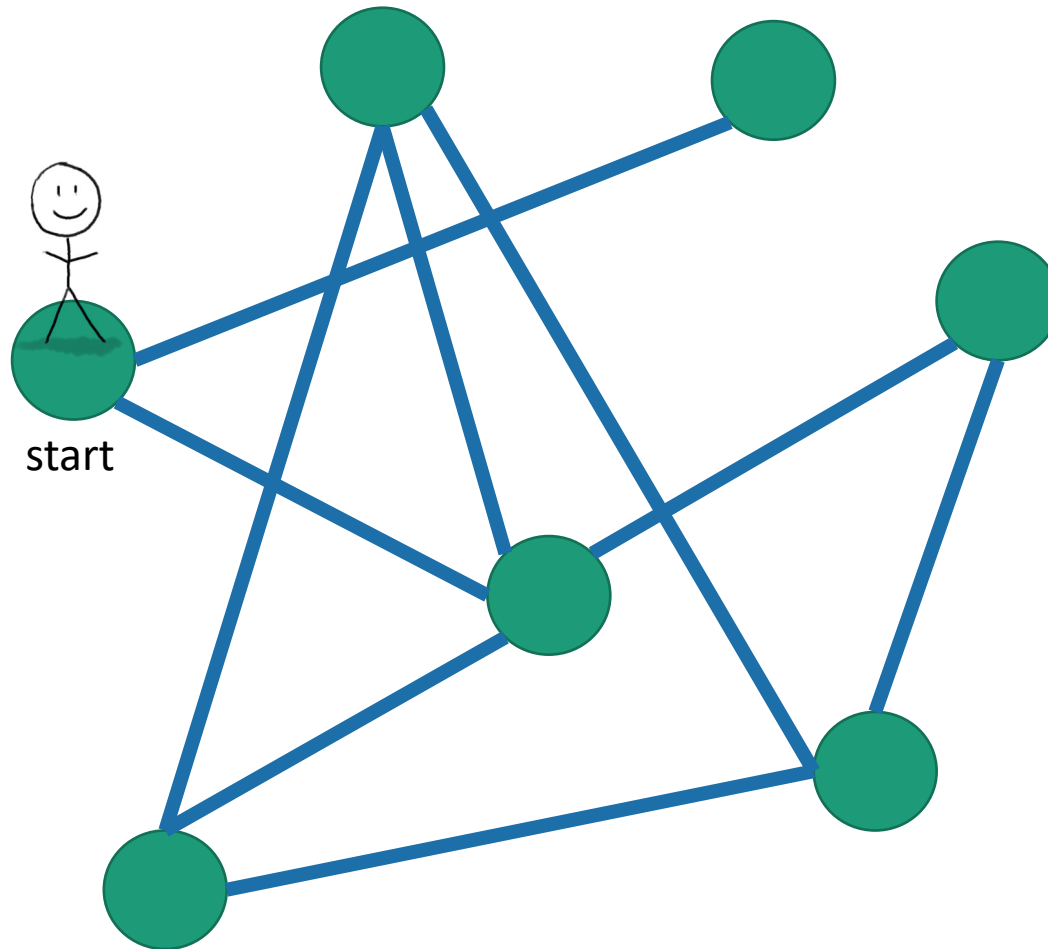
Exploring a labyrinth with chalk and a piece of string



-  Not been there yet
-  Been there, haven't explored all the paths out.
-  Been there, have explored all the paths out.

# Depth First Search

Exploring a labyrinth with chalk and a piece of string



-  Not been there yet
-  Been there, haven't explored all the paths out.
-  Been there, have explored all the paths out.

Labyrinth:  
**EXPLORED!**

# Depth First Search

## Exploring a labyrinth with pseudocode

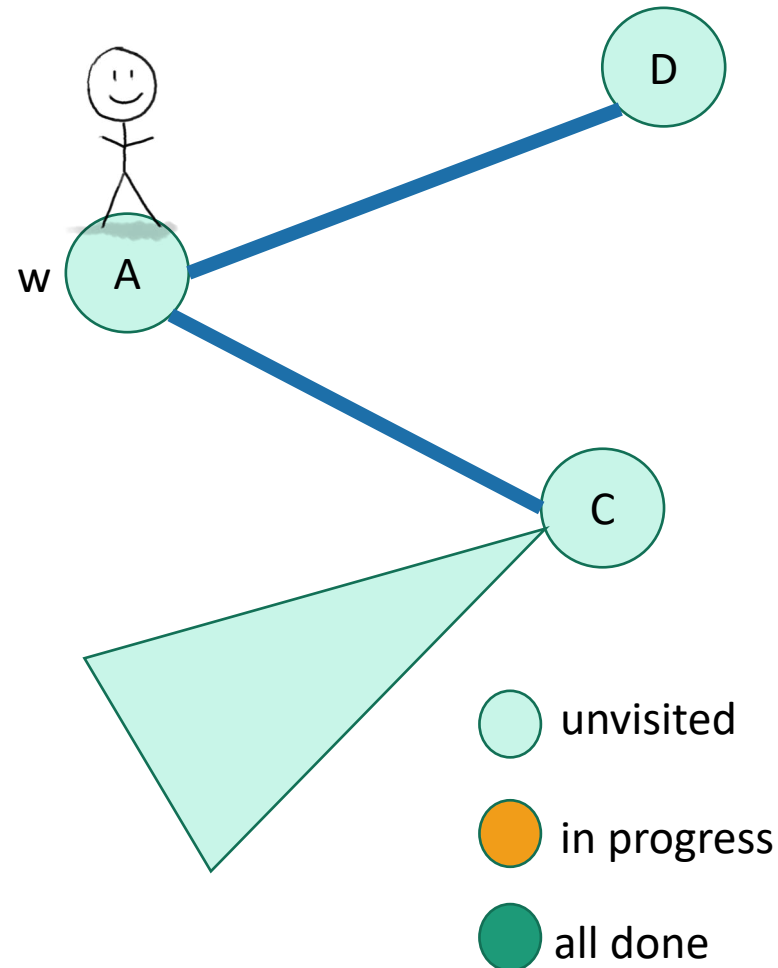
- Each vertex keeps track of whether it is:
  - Unvisited 
  - In progress 
  - All done 
- Each vertex will also keep track of:
  - The time we **first enter it**.
  - The time we finish with it and mark it **all done**.



You might have seen other ways to implement DFS than what we are about to go through. This way has more bookkeeping – the bookkeeping will be useful later!

# Depth First Search

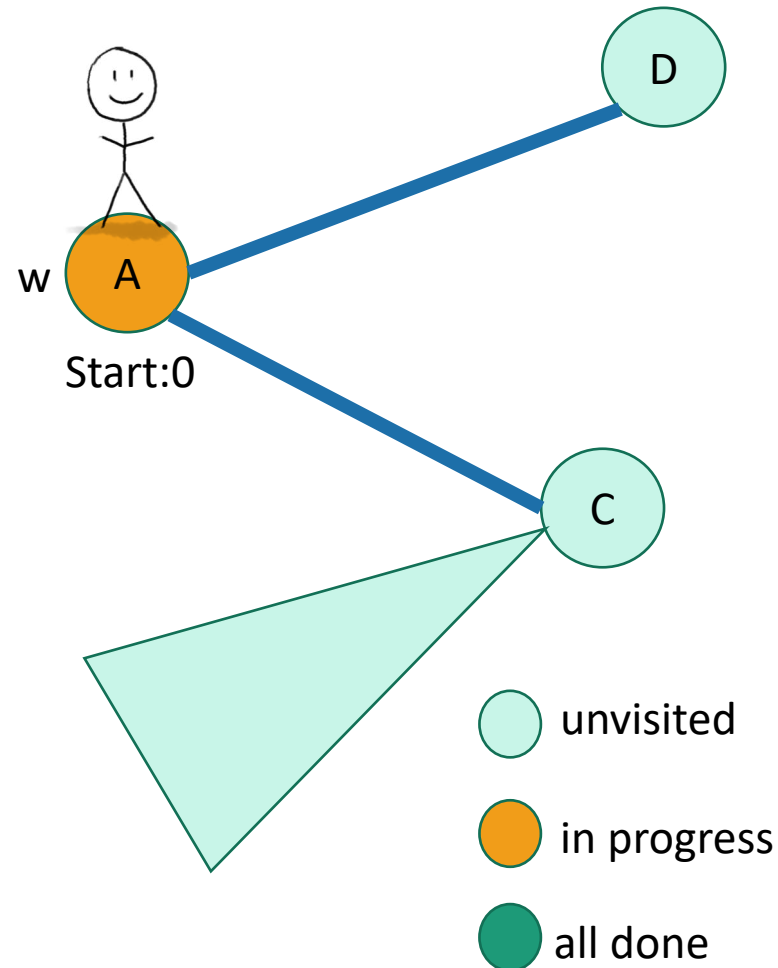
currentTime = 0



- **DFS**(w, currentTime):
  - w.startTime = currentTime
  - currentTime ++
  - Mark w as **in progress**.
  - **for** v in w.neighbors:
    - **if** v is **unvisited**:
      - currentTime = **DFS**(v, currentTime)
      - currentTime ++
  - w.finishTime = currentTime
  - Mark w as **all done**
  - **return** currentTime

# Depth First Search

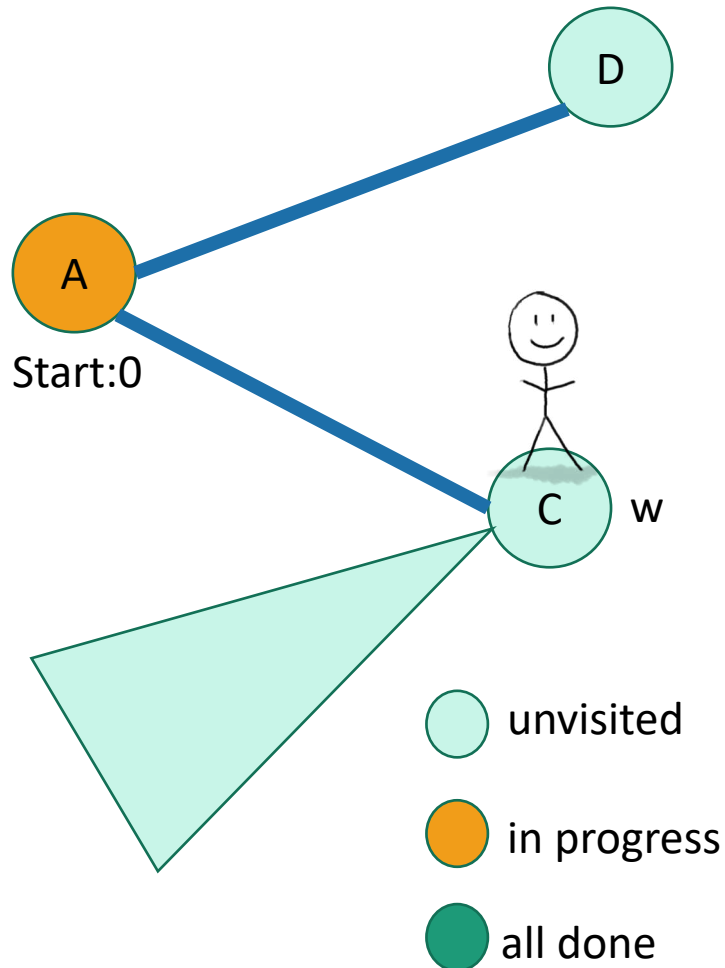
currentTime = 1



- **DFS(w, currentTime):**
  - w.startTime = currentTime
  - currentTime ++
  - Mark w as **in progress**.
  - **for** v in w.neighbors:
    - **if** v is **unvisited**:
      - currentTime = **DFS(v, currentTime)**
      - currentTime ++
  - w.finishTime = currentTime
  - Mark w as **all done**
  - **return** currentTime

# Depth First Search

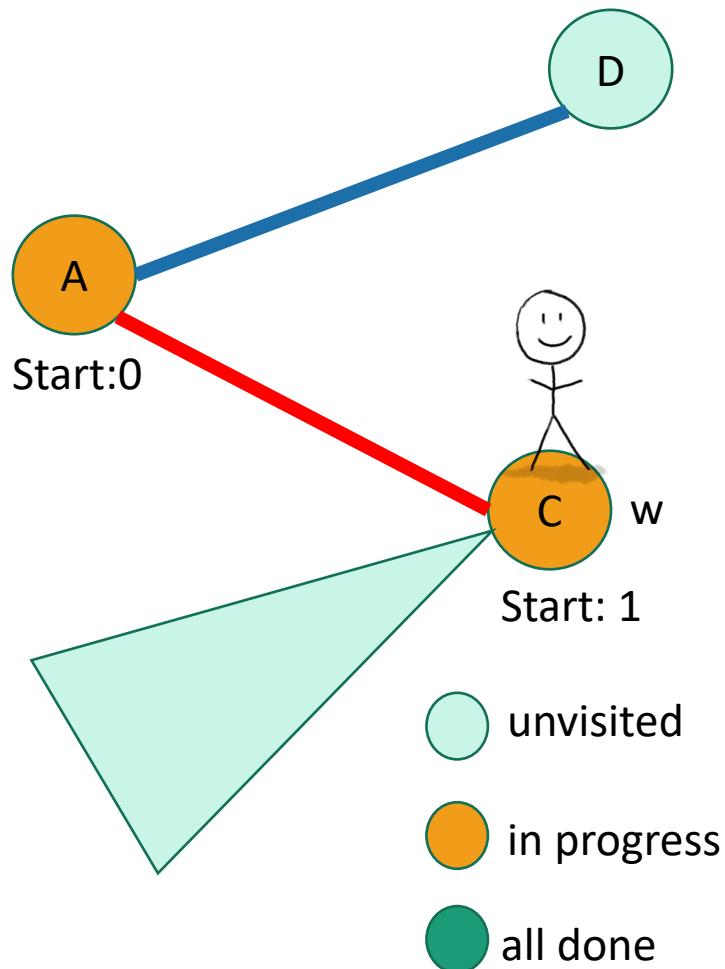
currentTime = 1



- **DFS(w, currentTime):**
  - w.startTime = currentTime
  - currentTime ++
  - Mark w as **in progress**.
  - **for** v in w.neighbors:
    - **if** v is **unvisited**:
      - currentTime = **DFS(v, currentTime)**
      - currentTime ++
  - w.finishTime = currentTime
  - Mark w as **all done**
  - **return** currentTime

# Depth First Search

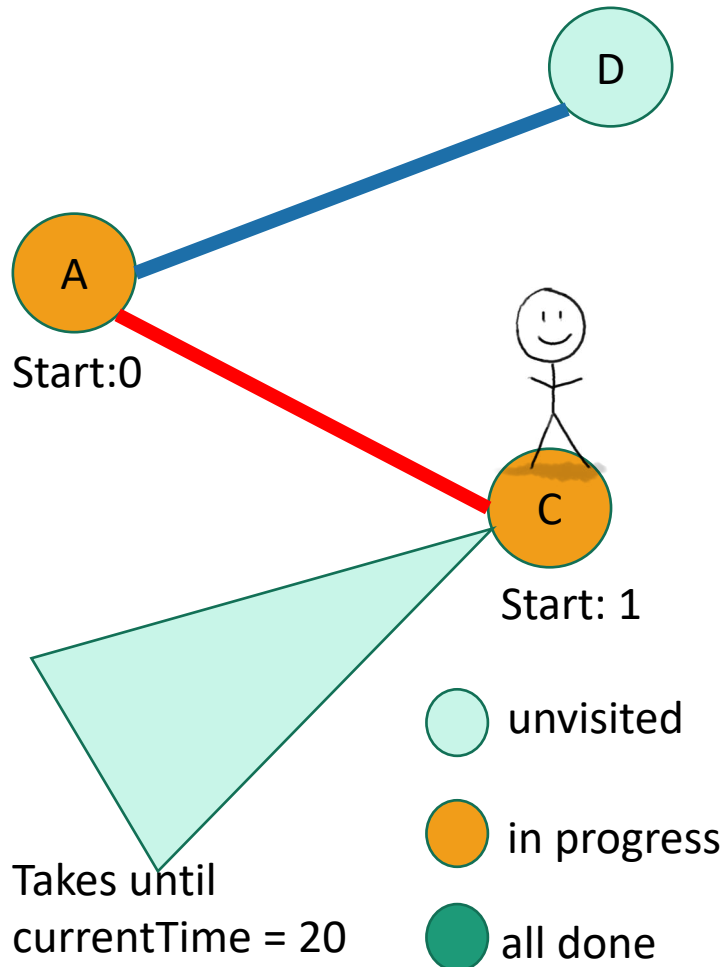
currentTime = 2



- **DFS(w, currentTime):**
  - w.startTime = currentTime
  - currentTime ++
  - Mark w as **in progress**.
  - **for** v in w.neighbors:
    - **if** v is **unvisited**:
      - currentTime = **DFS(v, currentTime)**
      - currentTime ++
  - w.finishTime = currentTime
  - Mark w as **all done**
  - **return** currentTime

# Depth First Search

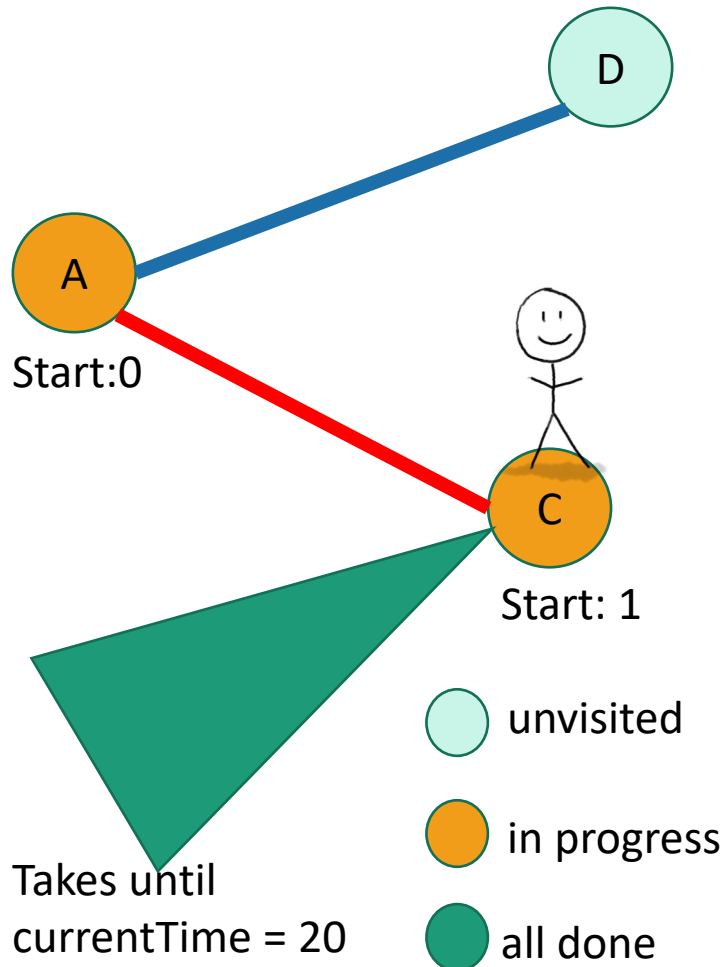
currentTime = 20



- **DFS**(w, currentTime):
  - w.startTime = currentTime
  - currentTime ++
  - Mark w as **in progress**.
  - **for** v in w.neighbors:
    - **if** v is **unvisited**:
      - currentTime  
= **DFS**(v, currentTime)
      - currentTime ++
  - w.finishTime = currentTime
  - Mark w as **all done**
  - **return** currentTime

# Depth First Search

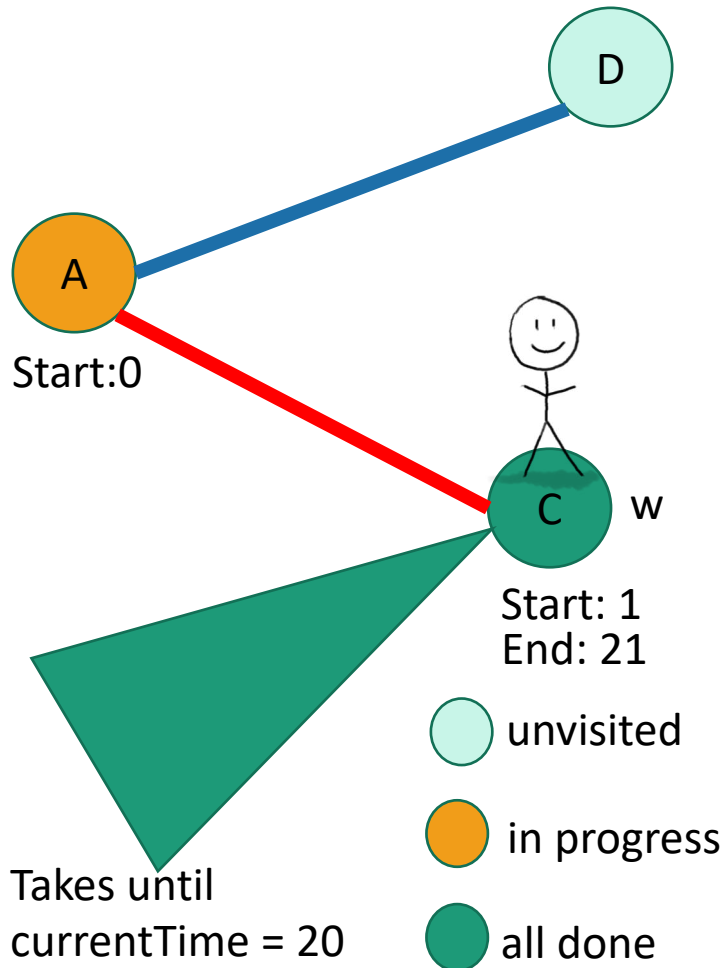
currentTime = 21



- **DFS**(w, currentTime):
  - w.startTime = currentTime
  - currentTime ++
  - Mark w as **in progress**.
  - **for** v in w.neighbors:
    - **if** v is **unvisited**:
      - currentTime  
= **DFS**(v, currentTime)
      - currentTime ++
  - w.finishTime = currentTime
  - Mark w as **all done**
  - **return** currentTime

# Depth First Search

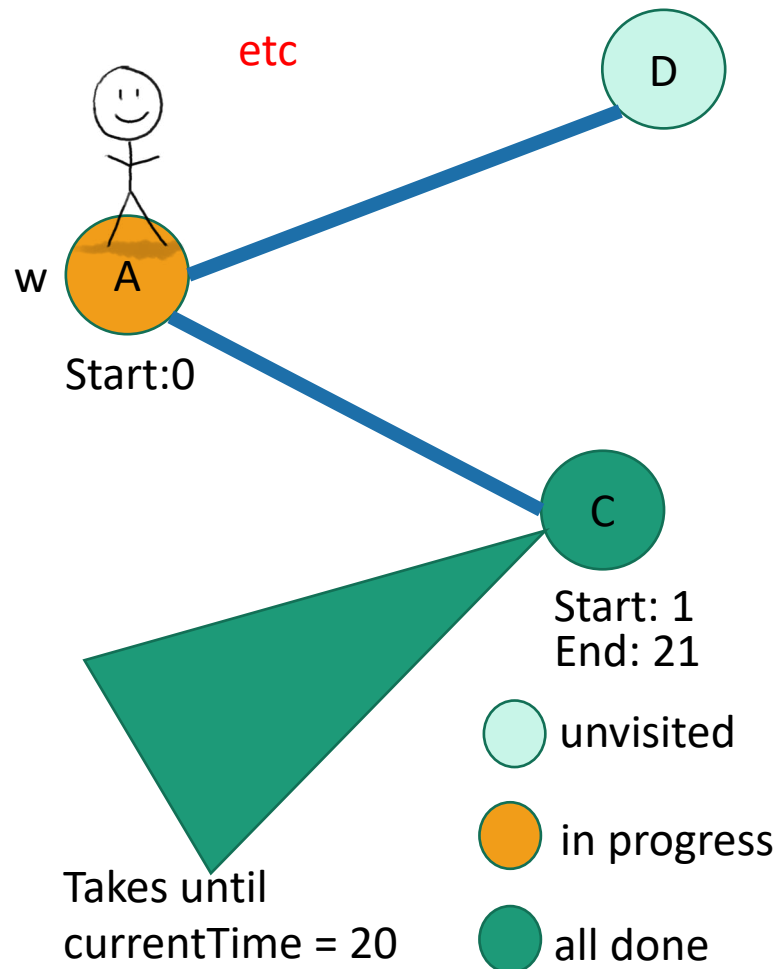
currentTime = 21



- **DFS**(w, currentTime):
  - w.startTime = currentTime
  - currentTime ++
  - Mark w as **in progress**.
  - **for** v in w.neighbors:
    - **if** v is **unvisited**:
      - currentTime  
= **DFS**(v, currentTime)
      - currentTime ++
  - w.finishTime = currentTime
  - Mark w as **all done**
  - **return** currentTime

# Depth First Search

currentTime = 22



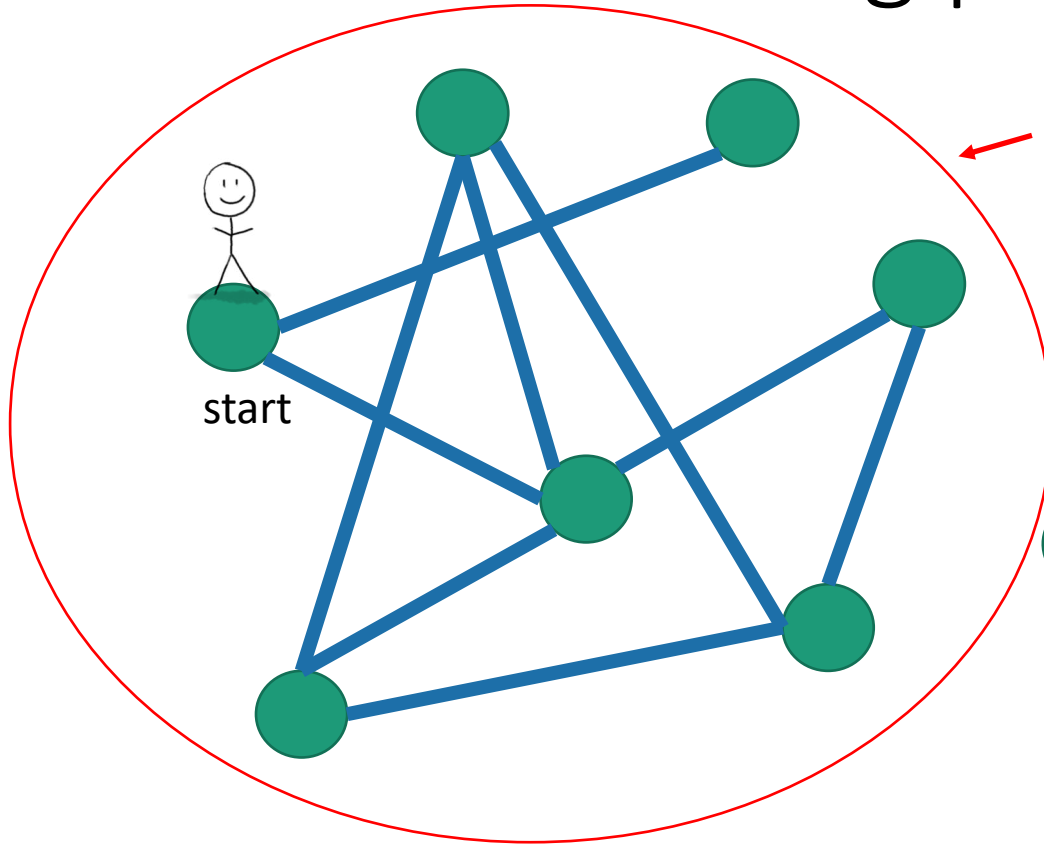
- **DFS(w, currentTime):**
  - w.startTime = currentTime
  - currentTime ++
  - Mark w as **in progress**.
  - **for** v in w.neighbors:
    - **if** v is **unvisited**:
      - currentTime  
= **DFS(v, currentTime)**
      - currentTime ++
  - w.finishTime = currentTime
  - Mark w as **all done**
  - **return** currentTime

# This is not the only way to write DFS!

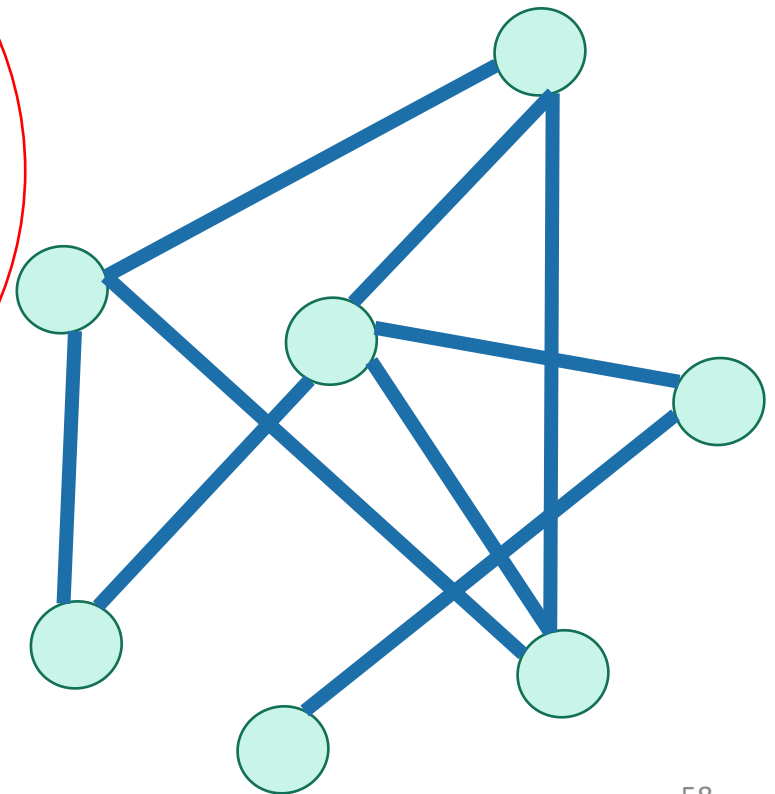
- See the textbook for an iterative version.
- (And/or figure out how to do it yourself!)



# DFS finds all the nodes reachable from the starting point



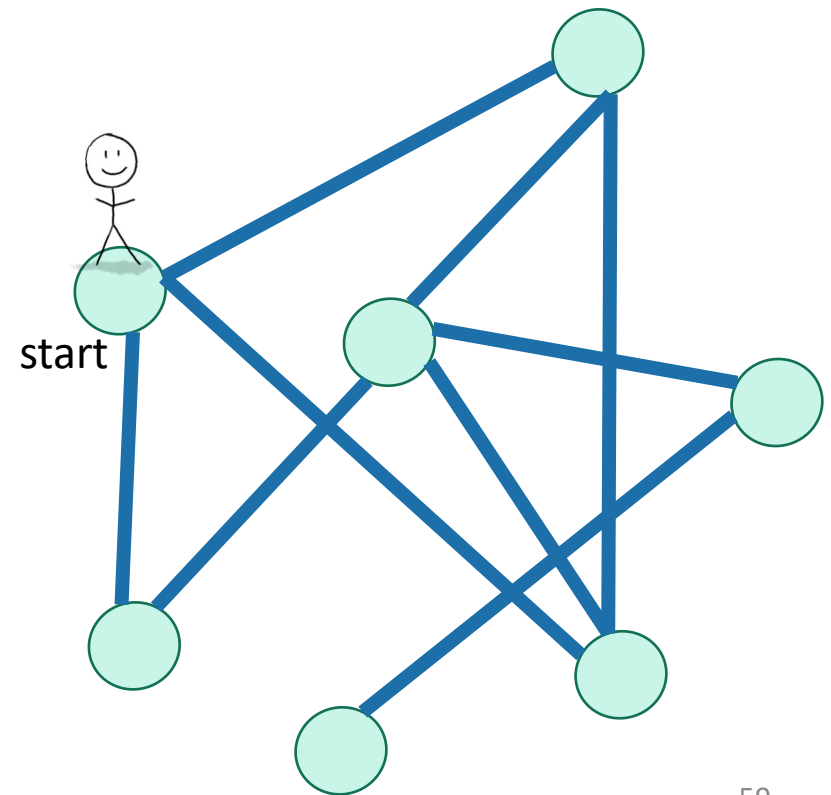
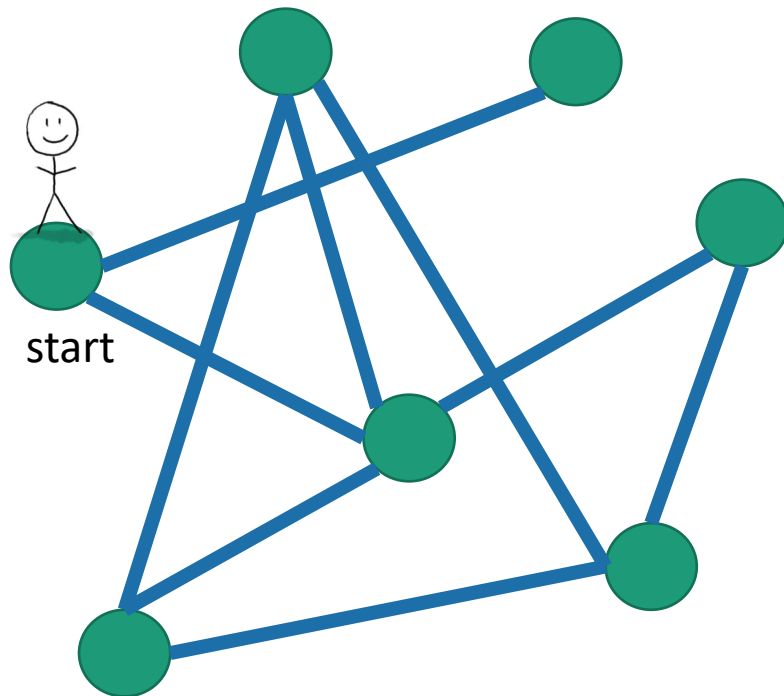
In an undirected graph, this is called a **connected component**.



**One application of DFS:** finding connected components.

# To explore the whole graph

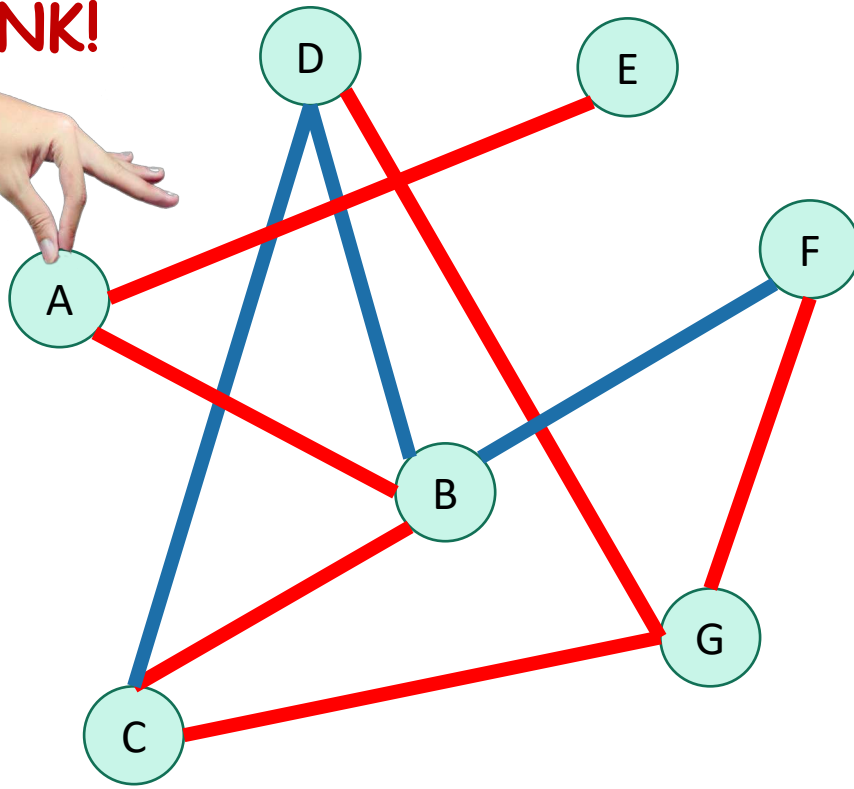
- Do it repeatedly!



# Why is it called depth-first?

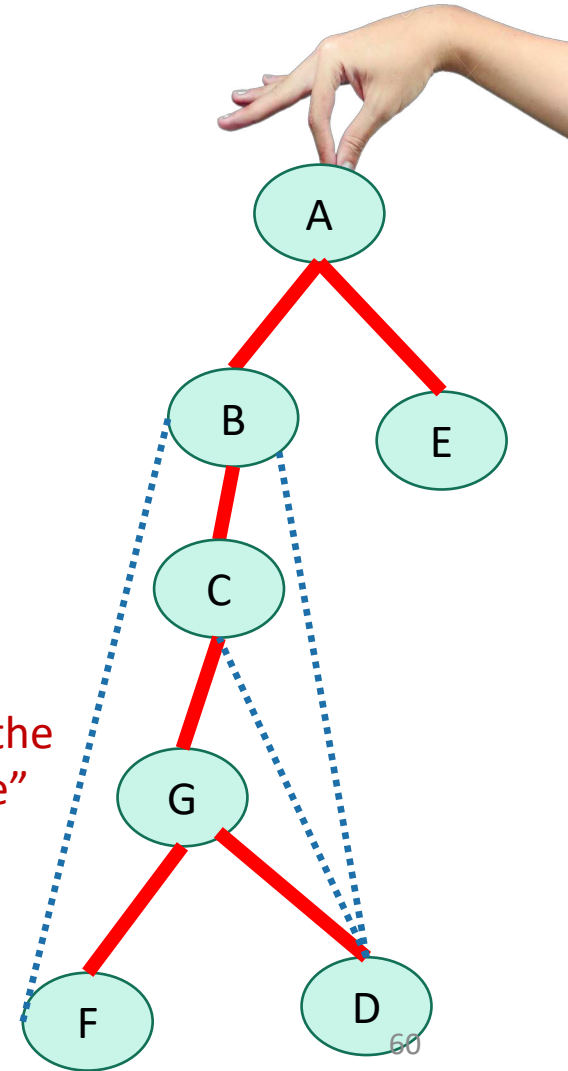
- We are implicitly building a tree:

YOINK!



- First, we go as deep as we can.

Call this the  
“DFS tree”



# Running time

To explore just the connected component we started in

- We look at each edge at most twice.
  - Once from each of its endpoints
- And basically we don't do anything else.
- So...



$O(m)$

# Running time

To explore just the connected component we started in

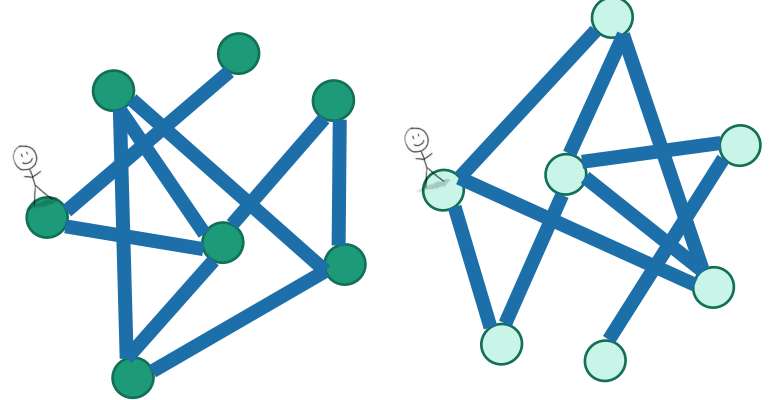
- Assume we are using the linked-list format for  $G$ .
- Say  $C = (V', E')$  is a connected component.
- We visit each vertex in  $C$  exactly once.
  - Here, “visit” means “call DFS on”
- At each vertex  $w$ , we:
  - Do some book-keeping:  $O(1)$
  - Loop over  $w$ 's neighbors and check if they are visited (and then potentially make a recursive call):  $O(1)$  per neighbor or  $O(\deg(w))$  total.
- Total time:
  - $\sum_{w \in V'} (O(\deg(w)) + O(1))$
  - $= O(|E'| + |V'|)$
  - $= O(|E'|)$



In a connected graph,  
 $|V'| \leq |E'| + 1$ .

# Running time

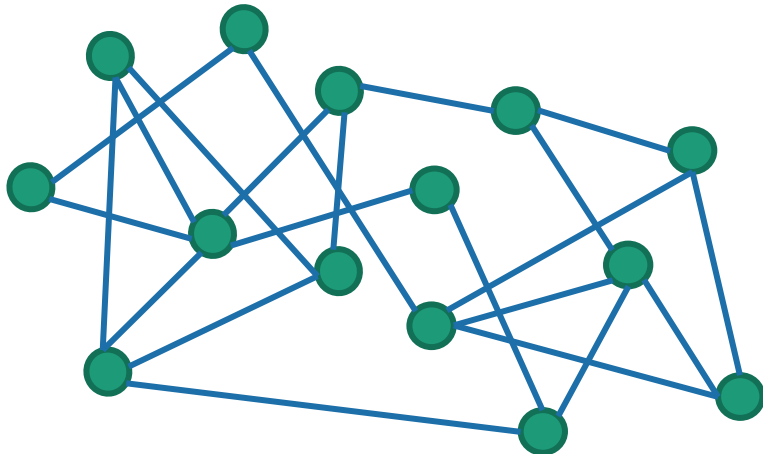
To explore **the whole graph**



- Explore the connected components one-by-one.
- This takes time  $O(n + m)$

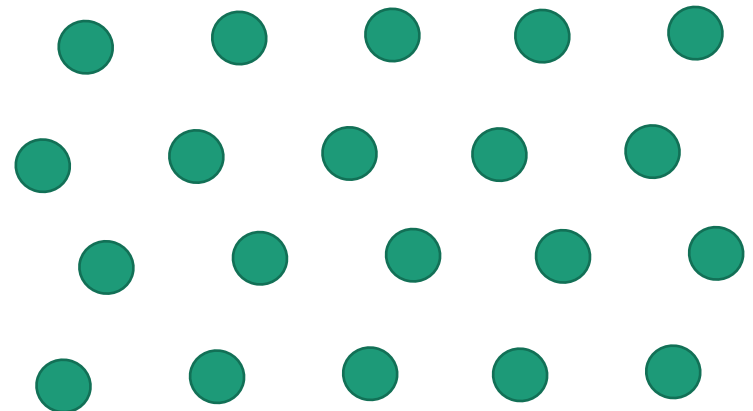
- Same computation as before:

$$\sum_{w \in V} (O(\deg(w)) + O(1)) = O(|E| + |V|) = O(n + m)$$



Here the running time is  $O(m)$  like before

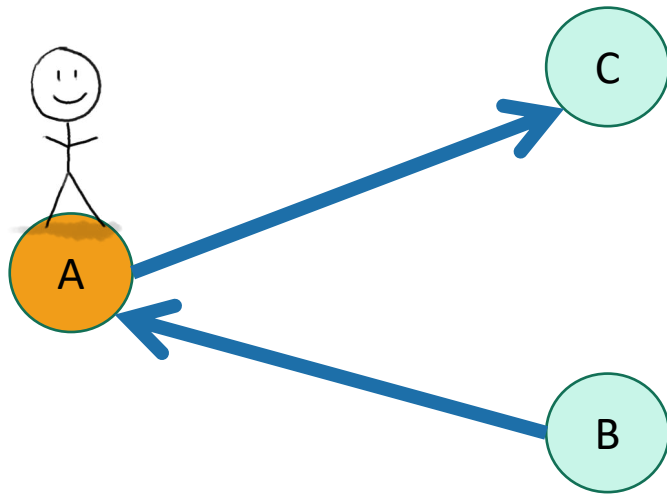
or



Here  $m=0$  but it still takes time  $O(n)$  to explore the graph.

# You check:

DFS works fine on directed graphs too!



Only walk to C, not to B.



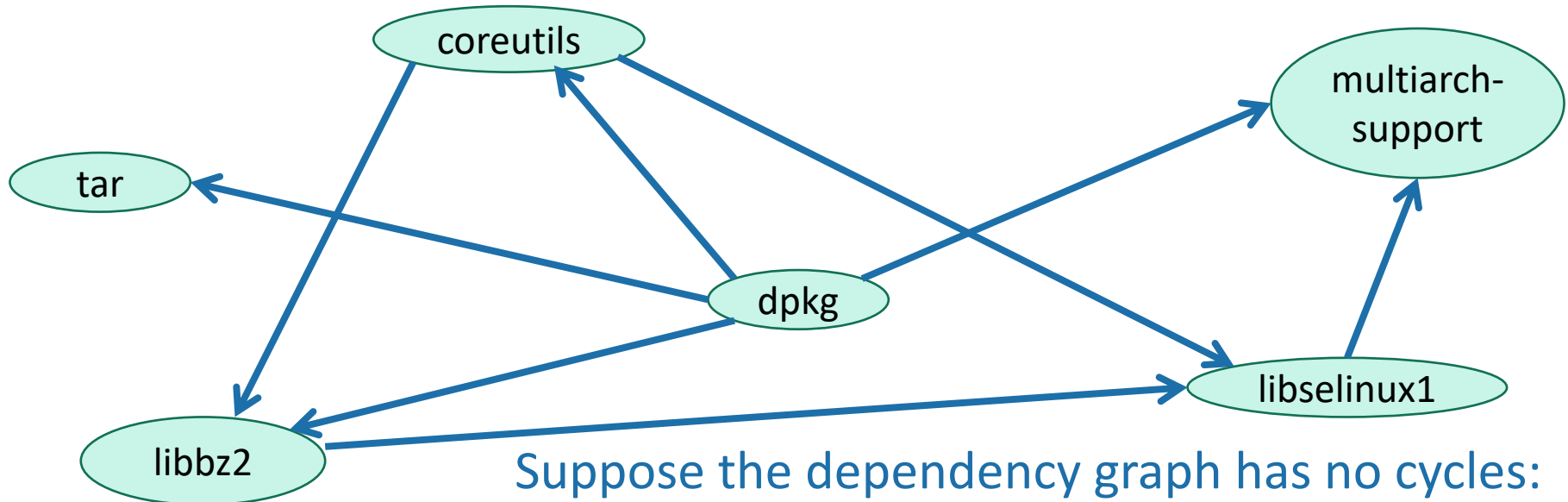
Siggi the studious stork

# Pre-lecture exercise

- How can you sign up for classes so that you never violate the pre-req requirements?
- More practically, how can you install packages without violating dependency requirements?

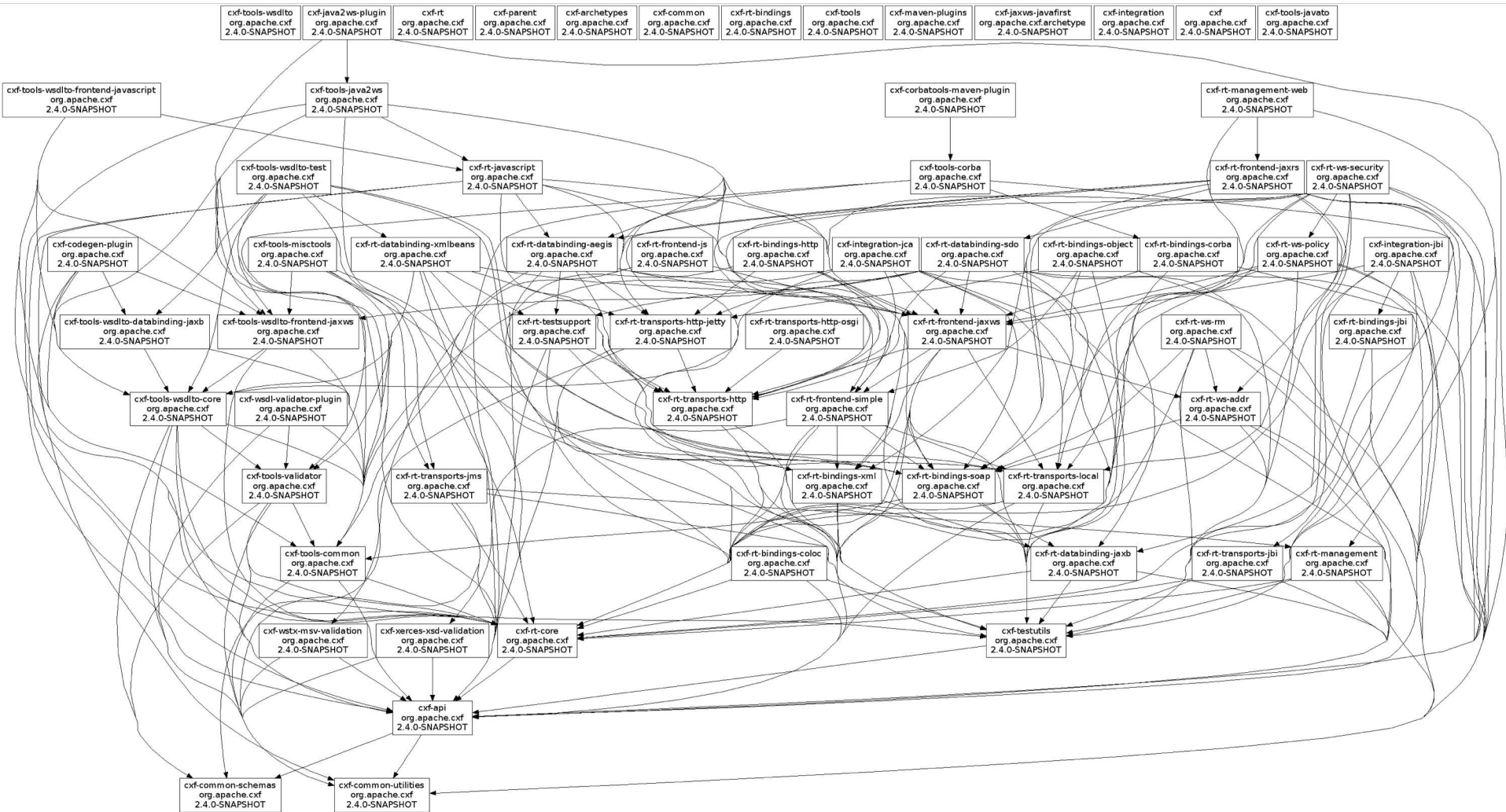
# Application of DFS: topological sorting

- Find an ordering of vertices so that all of the dependency requirements are met.
  - Aka, if  $v$  comes before  $w$  in the ordering, there is not an edge from  $w$  to  $v$ .



Suppose the dependency graph has no cycles:  
it is a **Directed Acyclic Graph (DAG)**

# Can't always eyeball it.

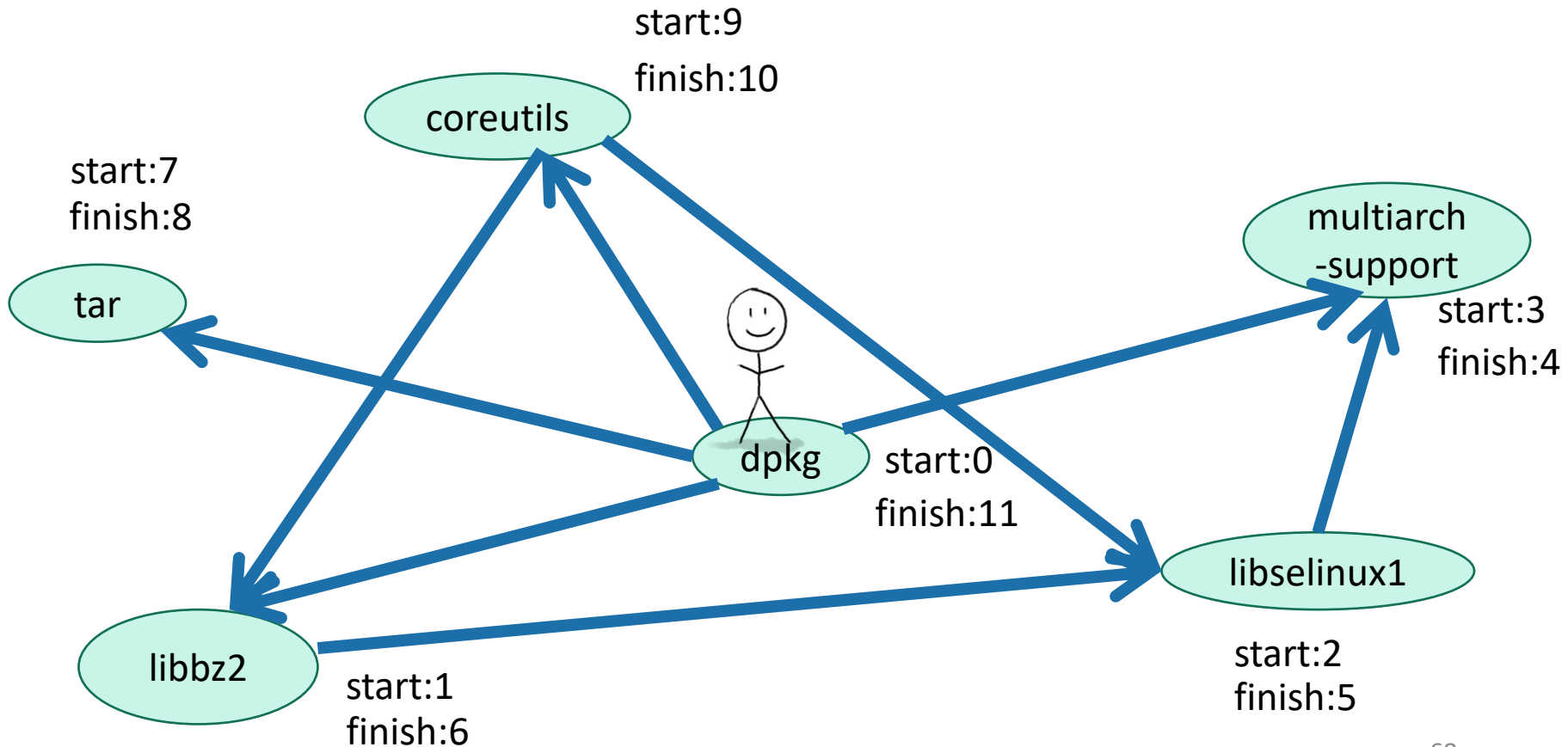
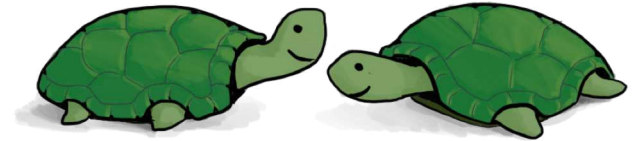


# Let's do DFS

What do you notice about the finish times? Any ideas for how we should do topological sort?

1 minute think

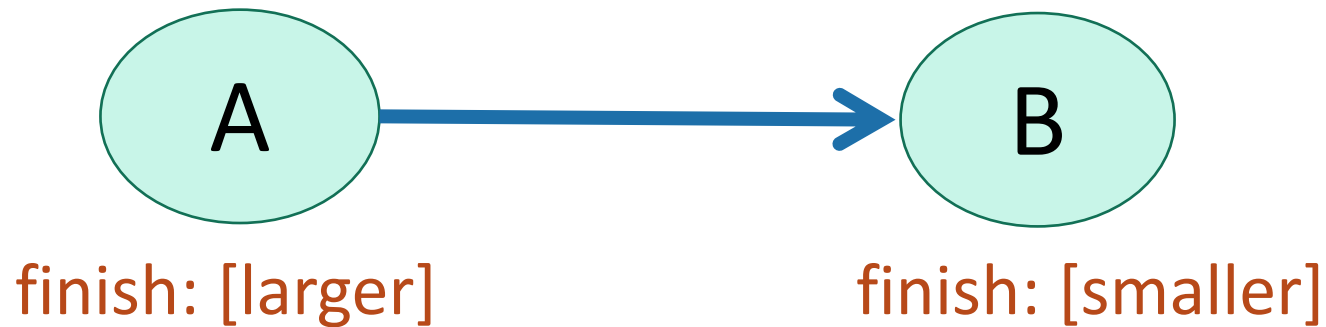
1 minute pair+share



Suppose the underlying  
graph has no cycles

# Finish times seem useful

**Claim:** In general, we'll always have:



To understand why, let's go back to that DFS tree.

# A more general statement (this holds even if there are cycles)

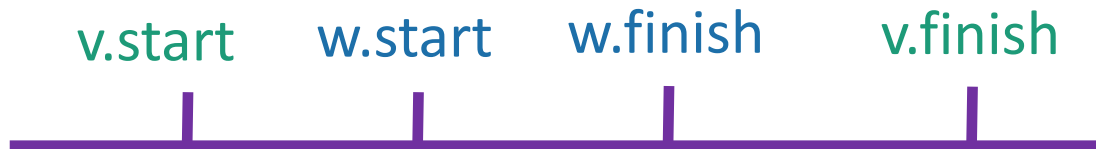
(check this  
statement  
carefully!)



- If  $v$  is a descendant of  $w$  in this tree:



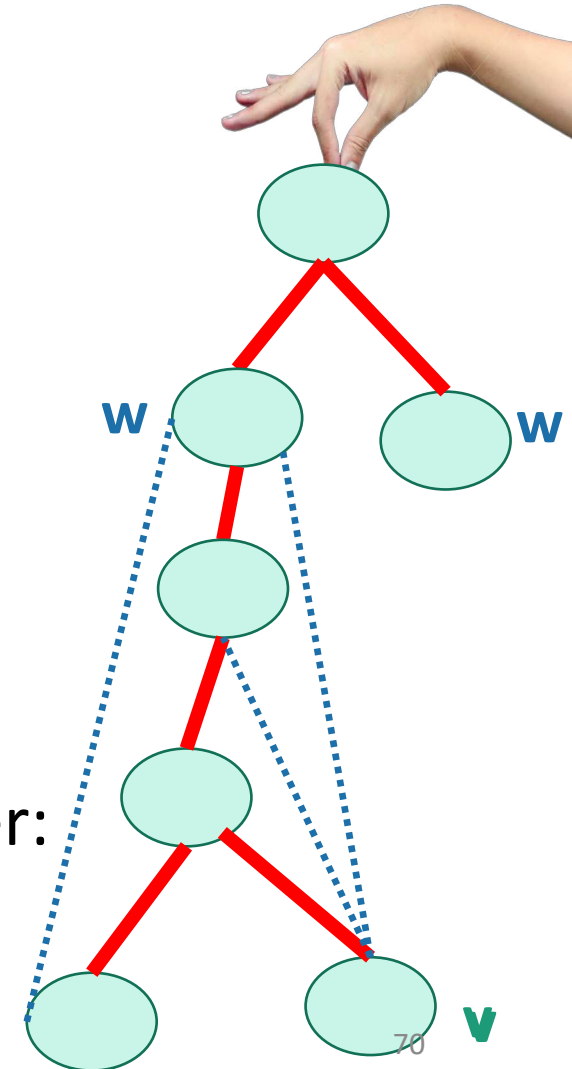
- If  $w$  is a descendant of  $v$  in this tree:



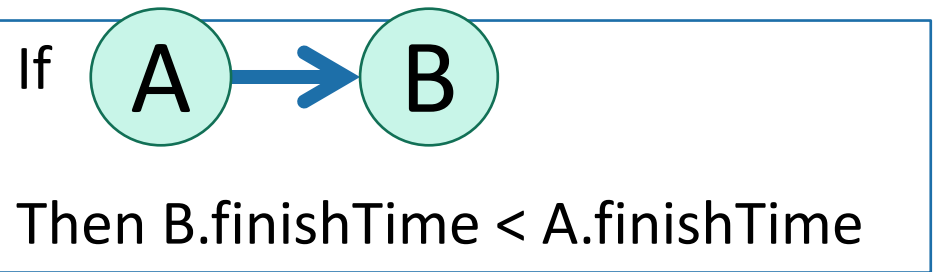
- If neither are descendants of each other:



(or the other way around)



So to prove this →



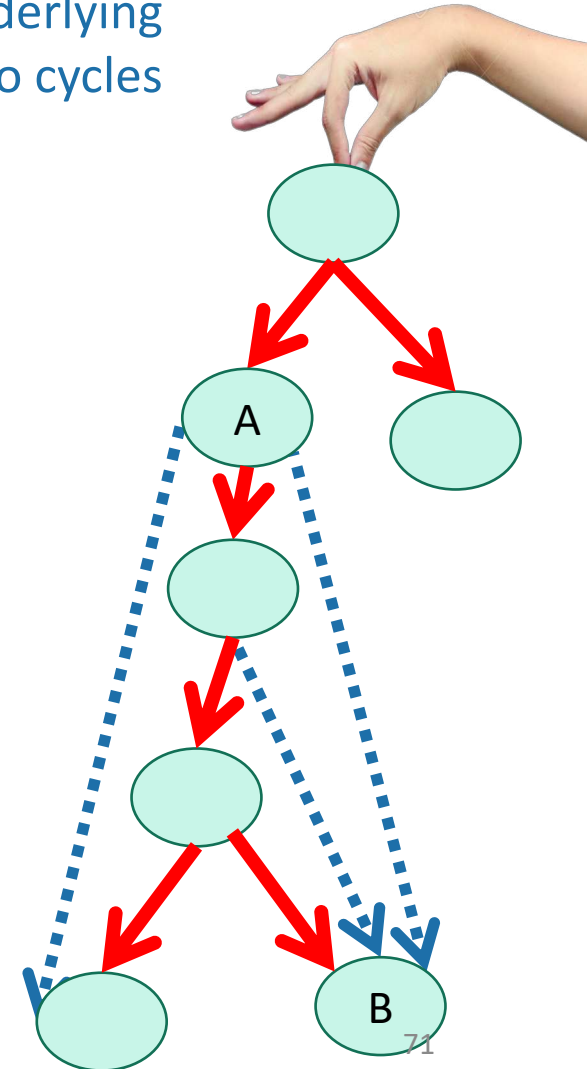
Suppose the underlying graph has no cycles

- **Case 1:** B is a descendant of A in the DFS tree.

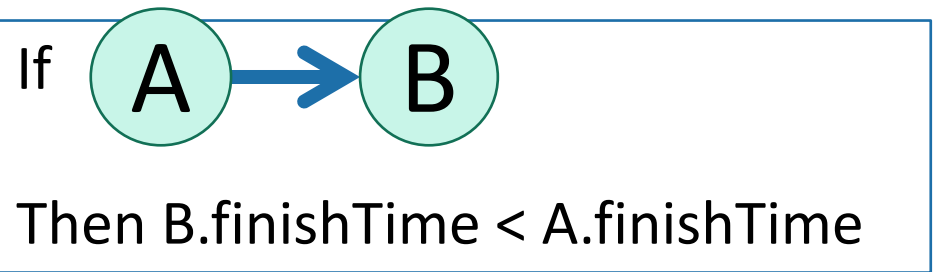
- Then



- aka,  $B.\text{finishTime} < A.\text{finishTime}$ .



So to prove this →



Suppose the underlying graph has no cycles

- **Case 2:** B is a **NOT** descendant of A in the DFS tree.

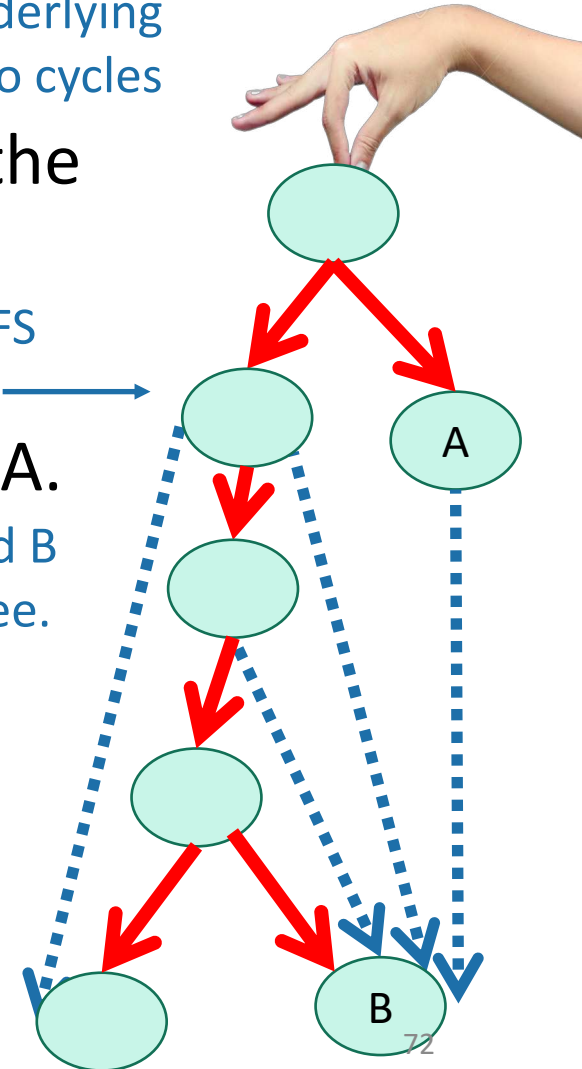
- Notice that A can't be a descendant of B in the DFS tree or else there'd be a cycle; so it looks like this →

- Then we must have explored B before A.
  - Otherwise we would have gotten to B from A, and B would have been a descendant of A in the DFS tree.

- Then

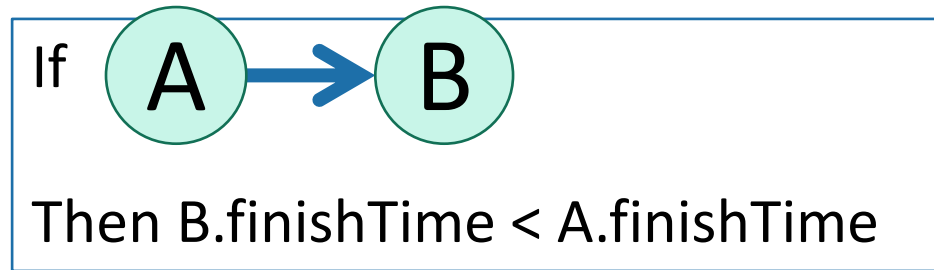


- aka,  $B.\text{finishTime} < A.\text{finishTime}$ .

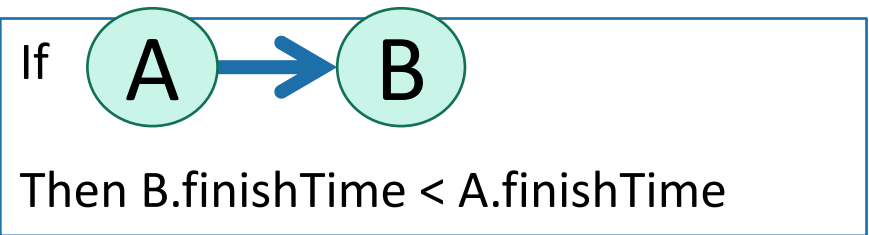


# Theorem

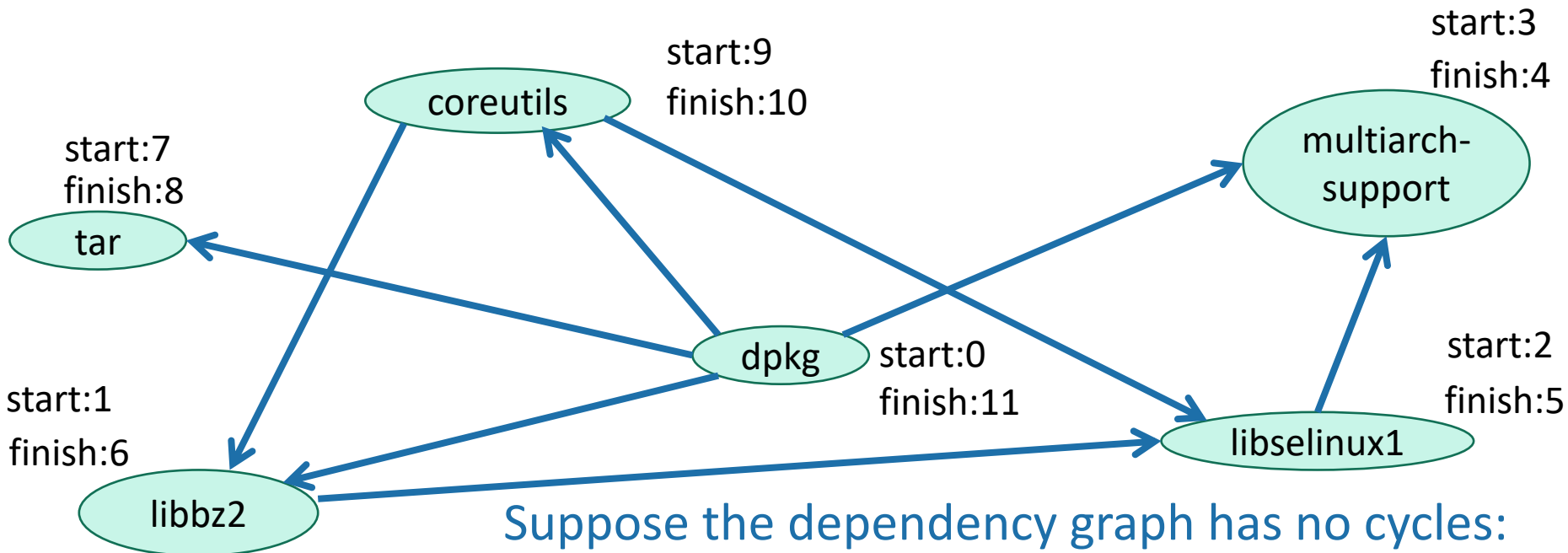
- If we run DFS on a directed acyclic graph,



# Back to topological sorting



- In what order should I install packages?
- In reverse order of finishing time in DFS!

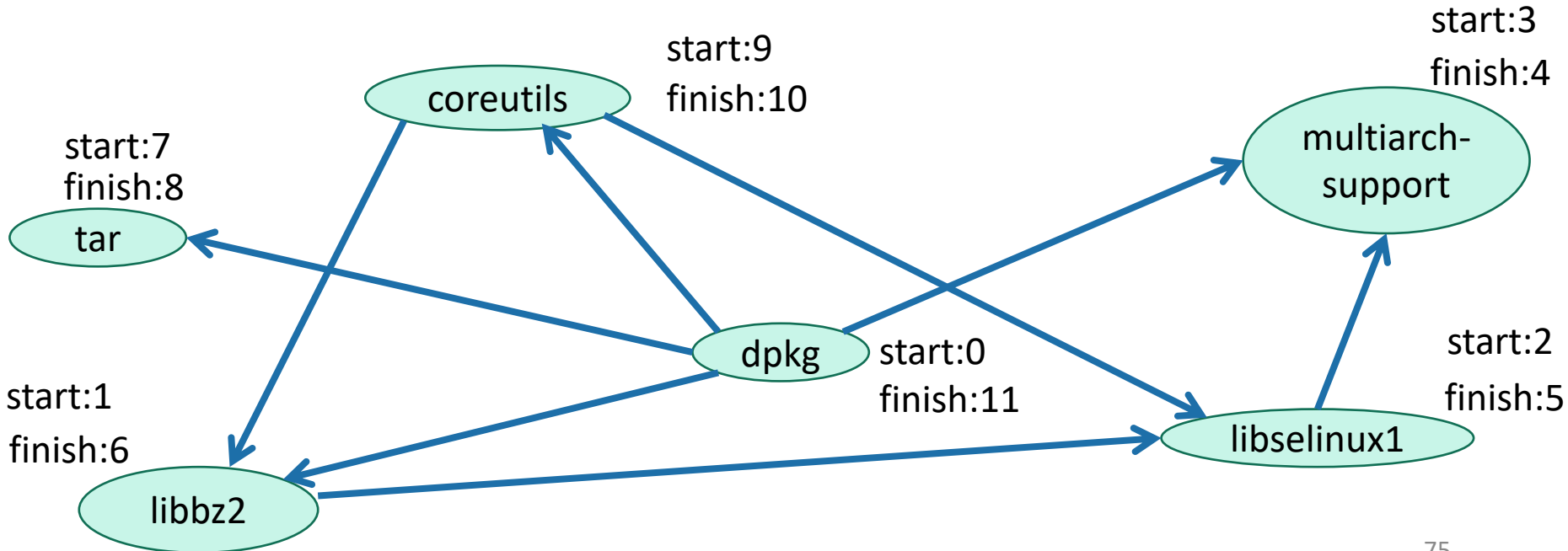


Suppose the dependency graph has no cycles:  
it is a **Directed Acyclic Graph (DAG)**

# Topological Sorting (on a DAG)

- Do DFS
- When you mark a vertex as **all done**, put it at the **beginning** of the list.

- dpkg
- coreutils
- tar
- libbz2
- libselinux1
- multiarch\_support



# For implementation, see IPython notebook

```
In [69]: print(G)
```

```
CS161Graph with:
  Vertices:
    dkpg,coreutils,multiarch_support,libselinux1,libbz2,tar,
  Edges:
    (dkpg,multiarch_support) (dkpg,coreutils) (dkpg,tar) (dkpg,libbz2
) (coreutils,libbz2) (coreutils,libselinux1) (libselinux1,multiarch_suppo
rt) (libbz2,libselinux1)
```

```
In [71]: V = topoSORT(G)
for v in V:
    print(v)
```

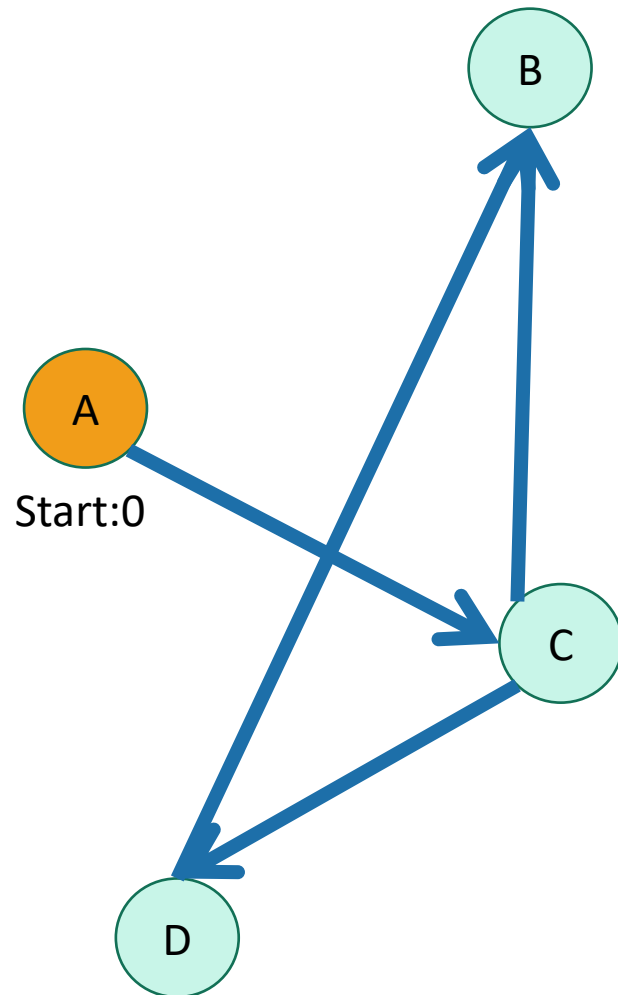
```
dkpg
tar
coreutils
libbz2
libselinux1
multiarch_support
```

# What did we just learn?

- DFS can help you solve the **topological sorting problem**
  - That's the fancy name for the problem of finding an ordering that respects all the dependencies
- Thinking about the DFS tree is helpful.

# Example:

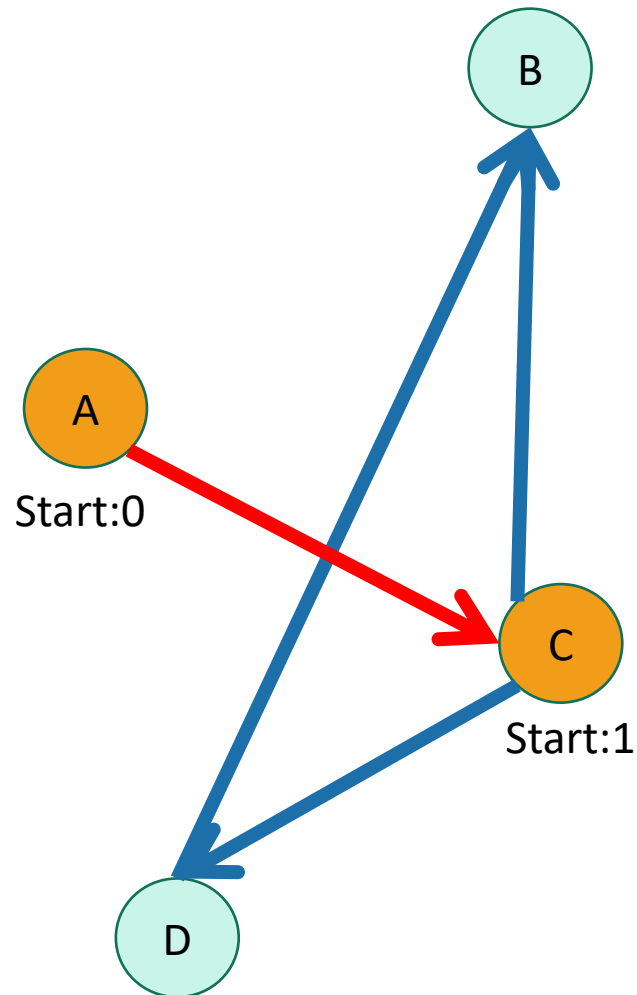
This example skipped in class – here for reference.



- Unvisited
- In progress
- All done

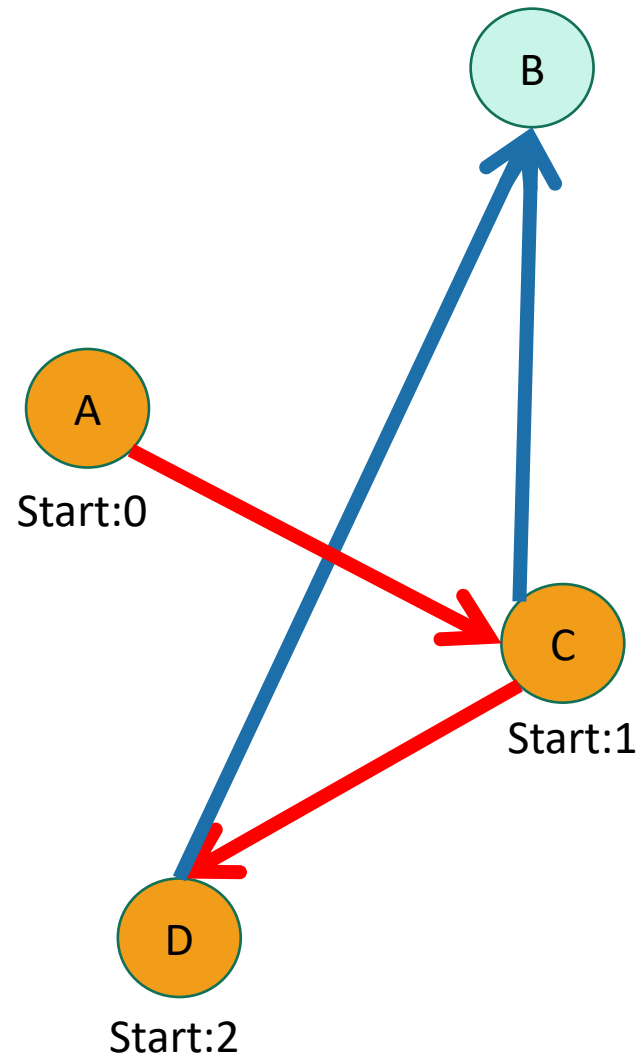
# Example

This example skipped in class – here for reference.



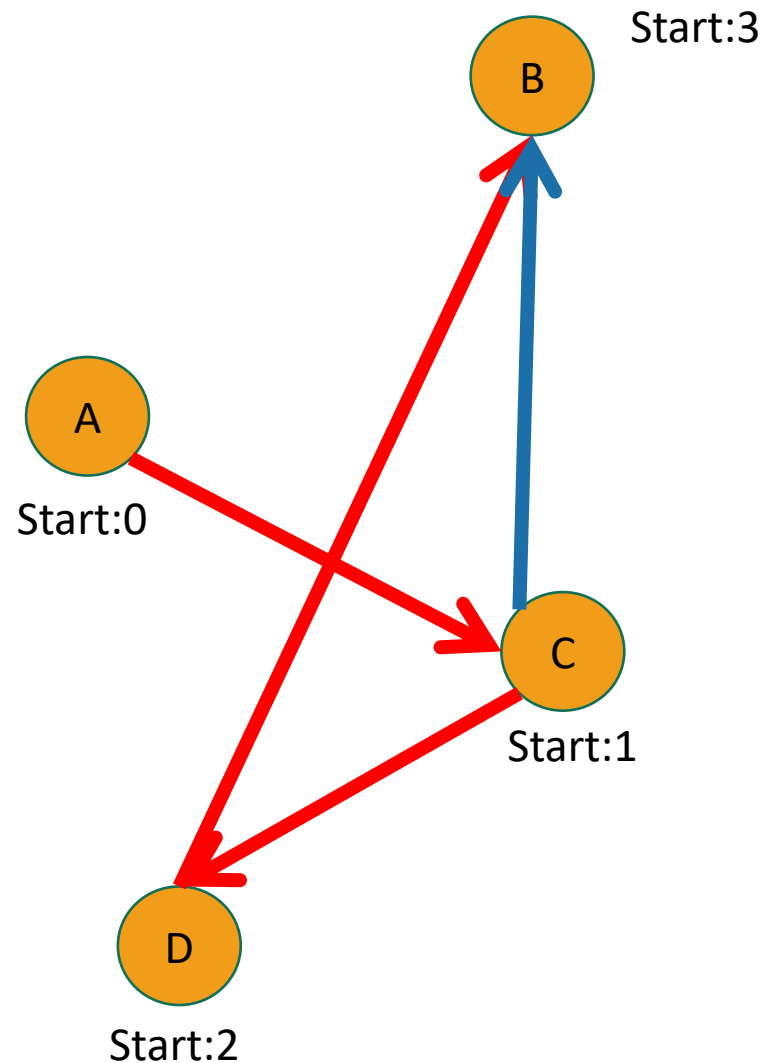
# Example

This example skipped in class – here for reference.



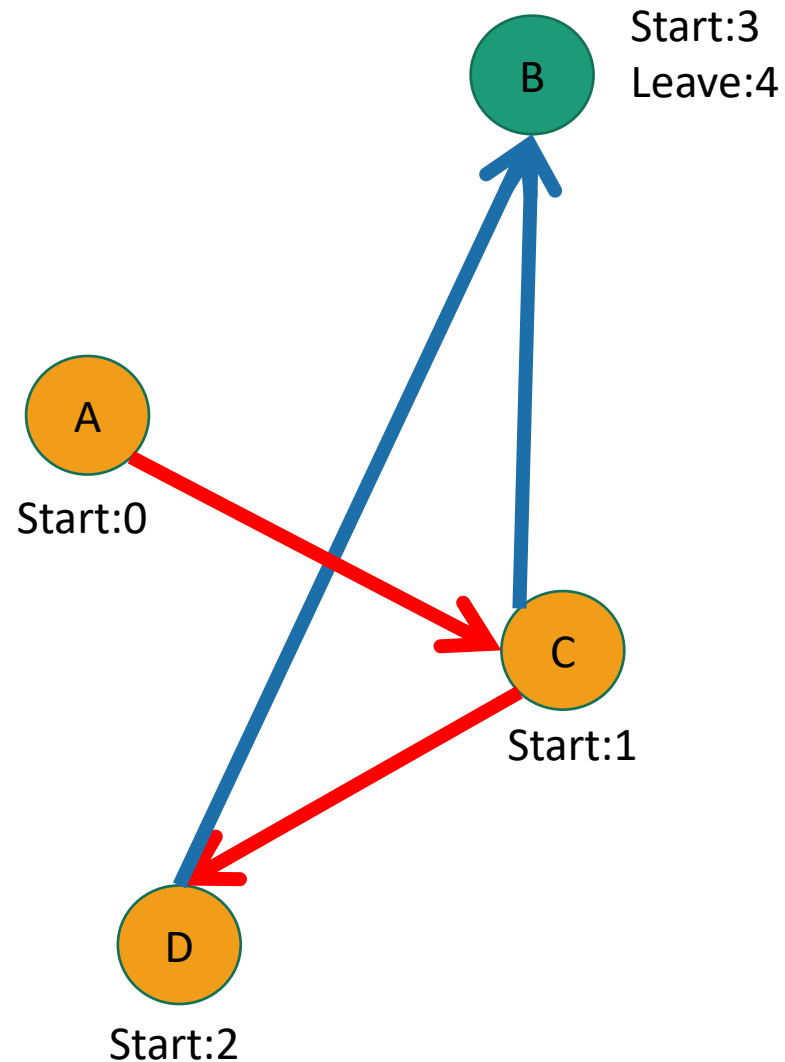
# Example

This example skipped in class – here for reference.



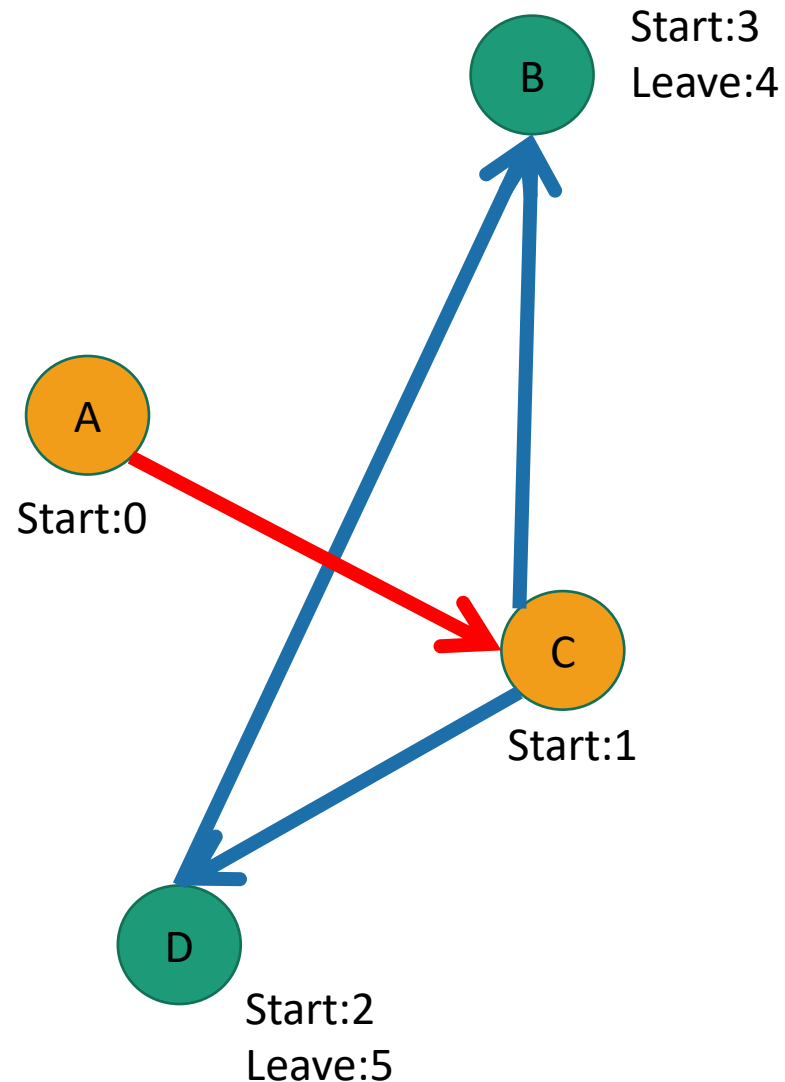
# Example

This example skipped in class – here for reference.



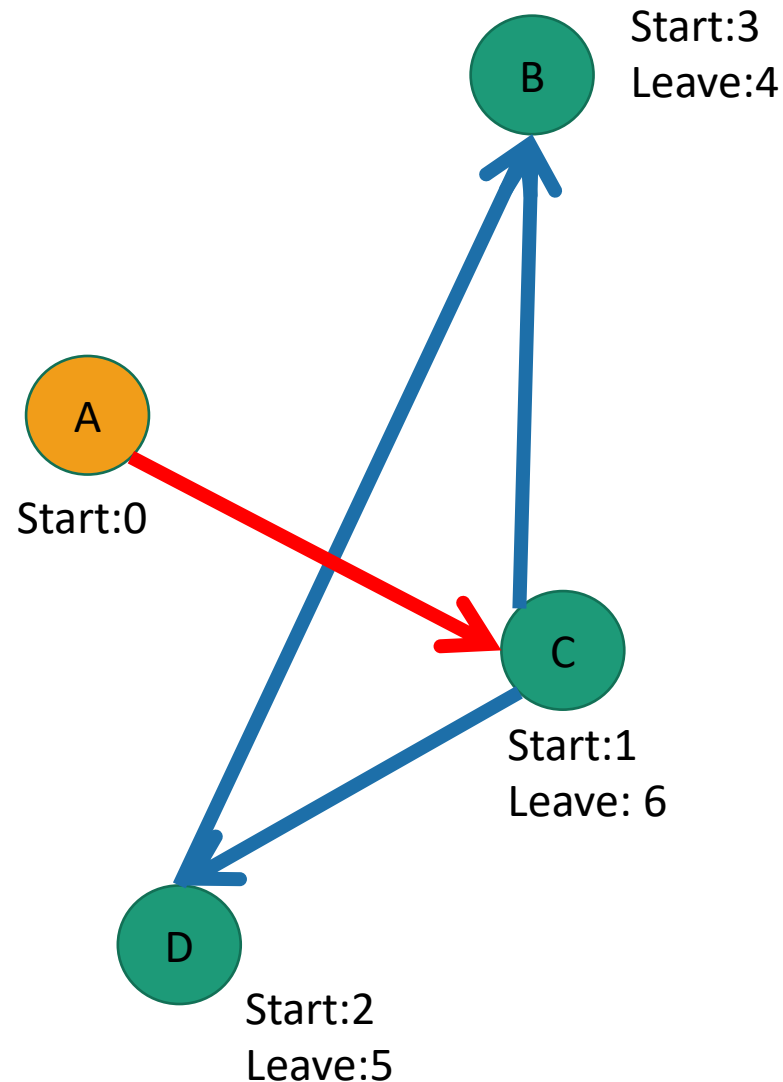
# Example

This example skipped in class – here for reference.



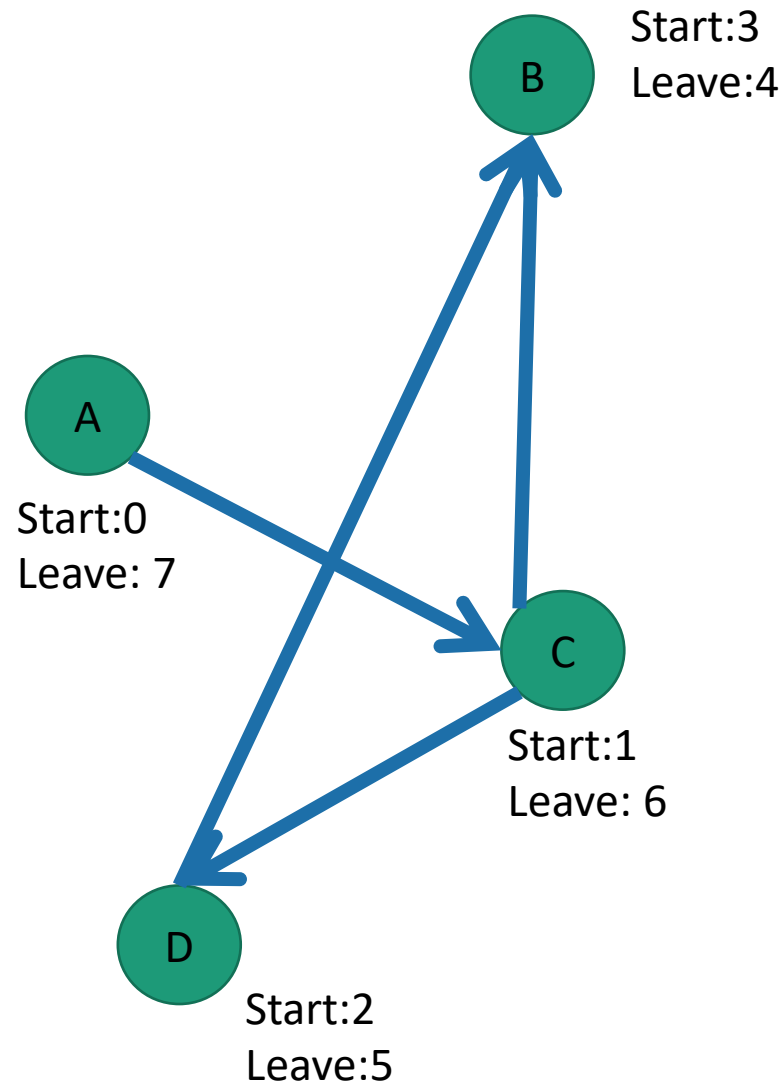
# Example

This example skipped in class – here for reference.



# Example

This example skipped in class – here for reference.

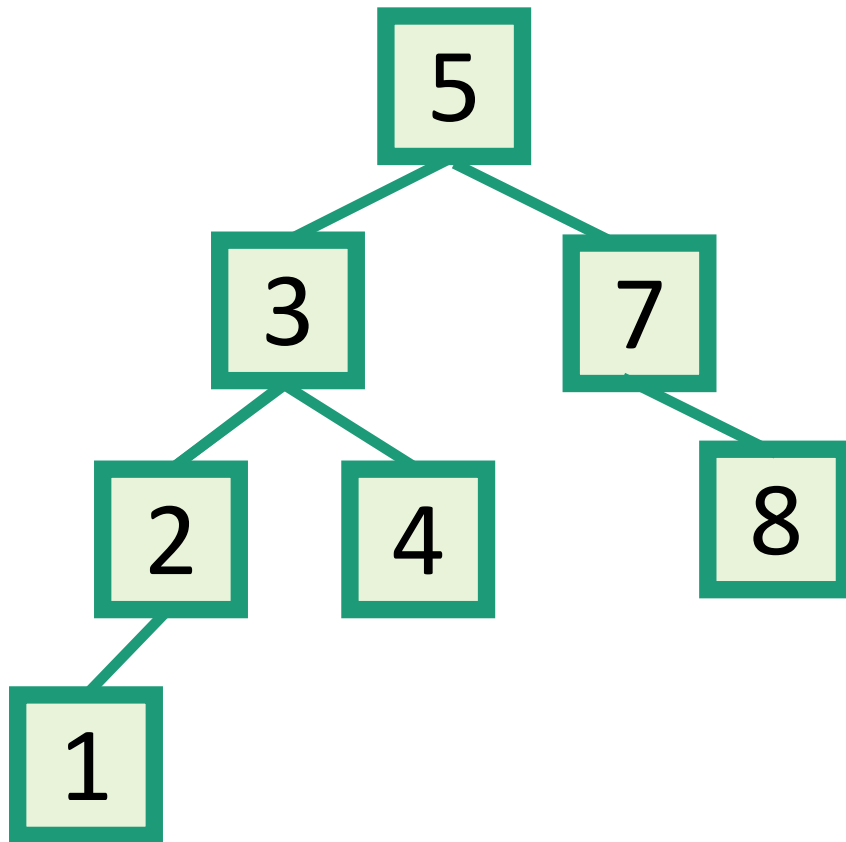


Do them in this order:



# Another use of DFS that we've already seen

- In-order enumeration of binary search trees

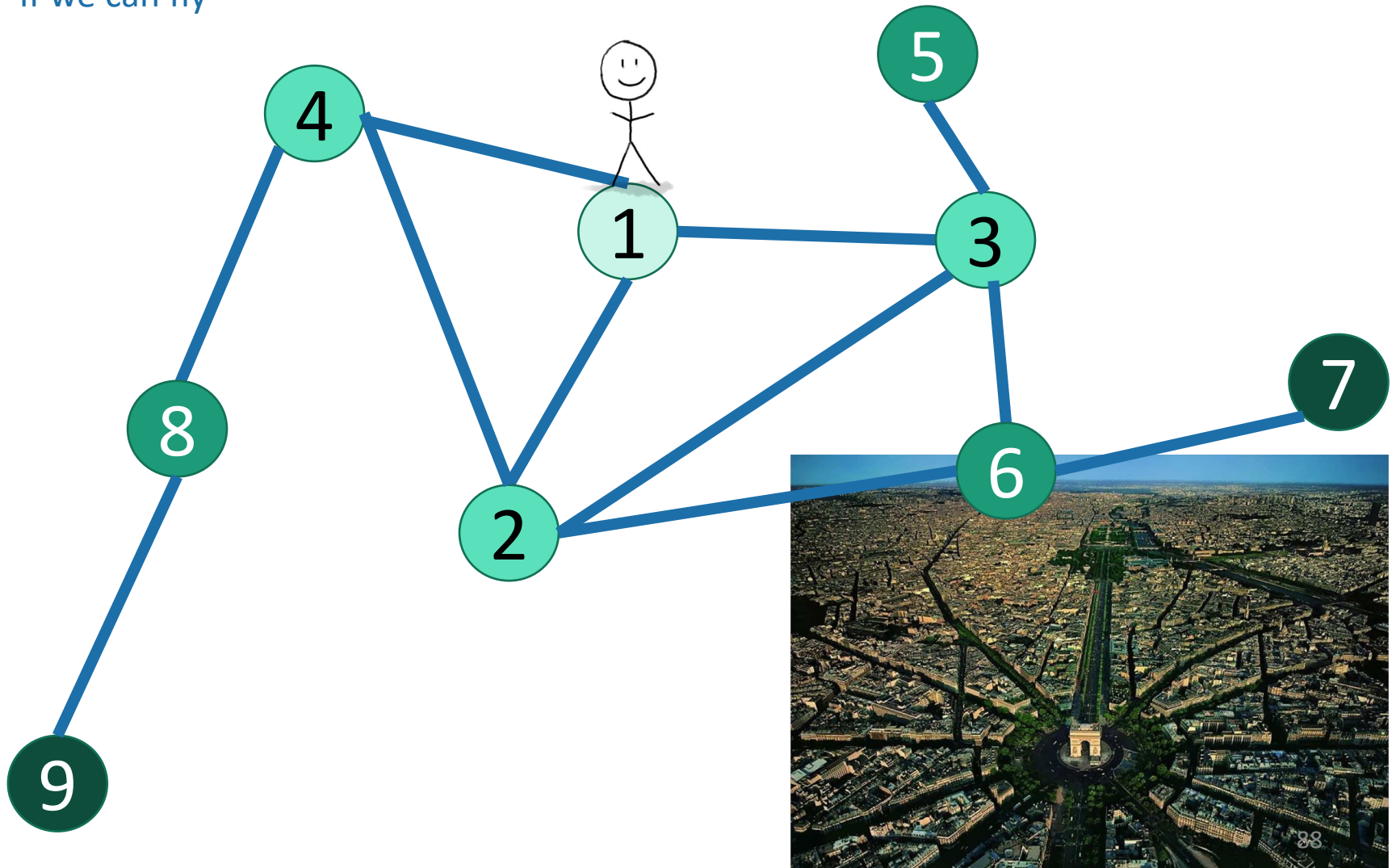


Do DFS and print a node's label when you are done with the left child and before you begin the right child.

# Part 2: breadth-first search

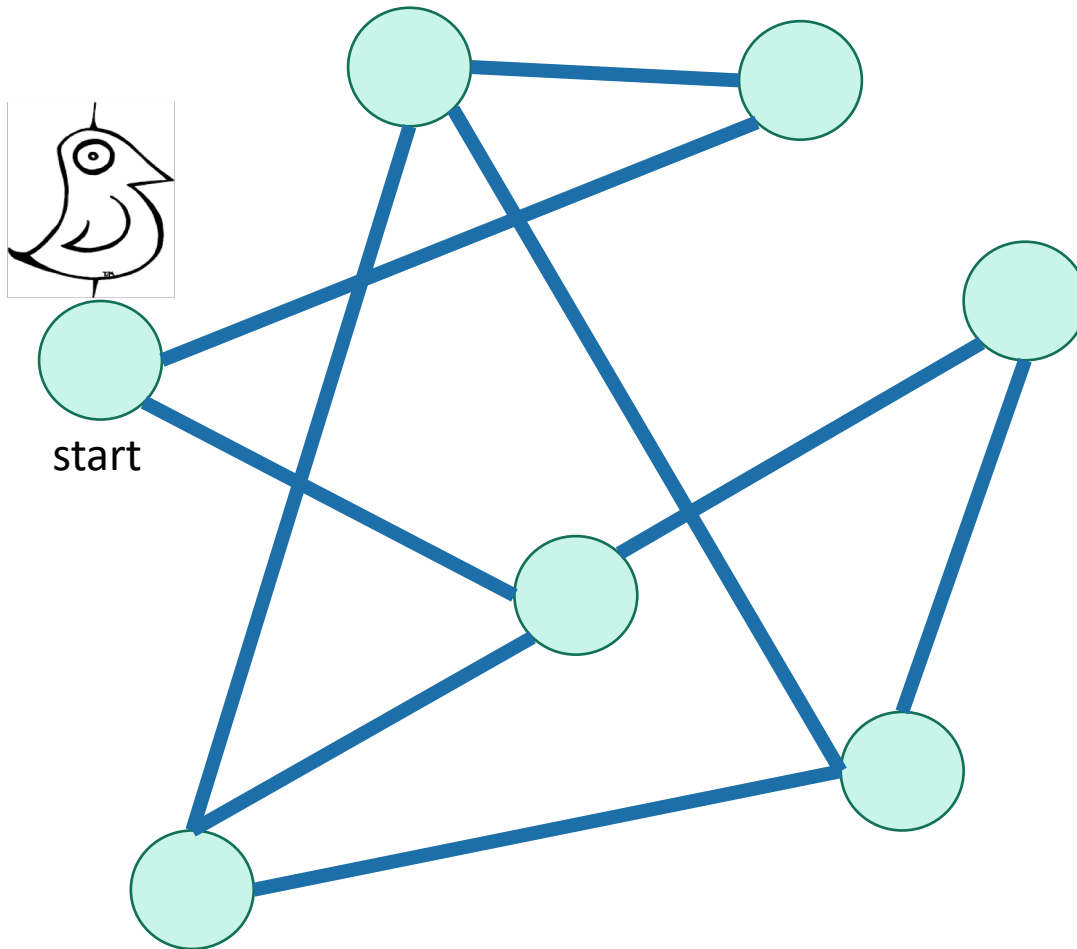
# How do we explore a graph?

If we can fly



# Breadth-First Search

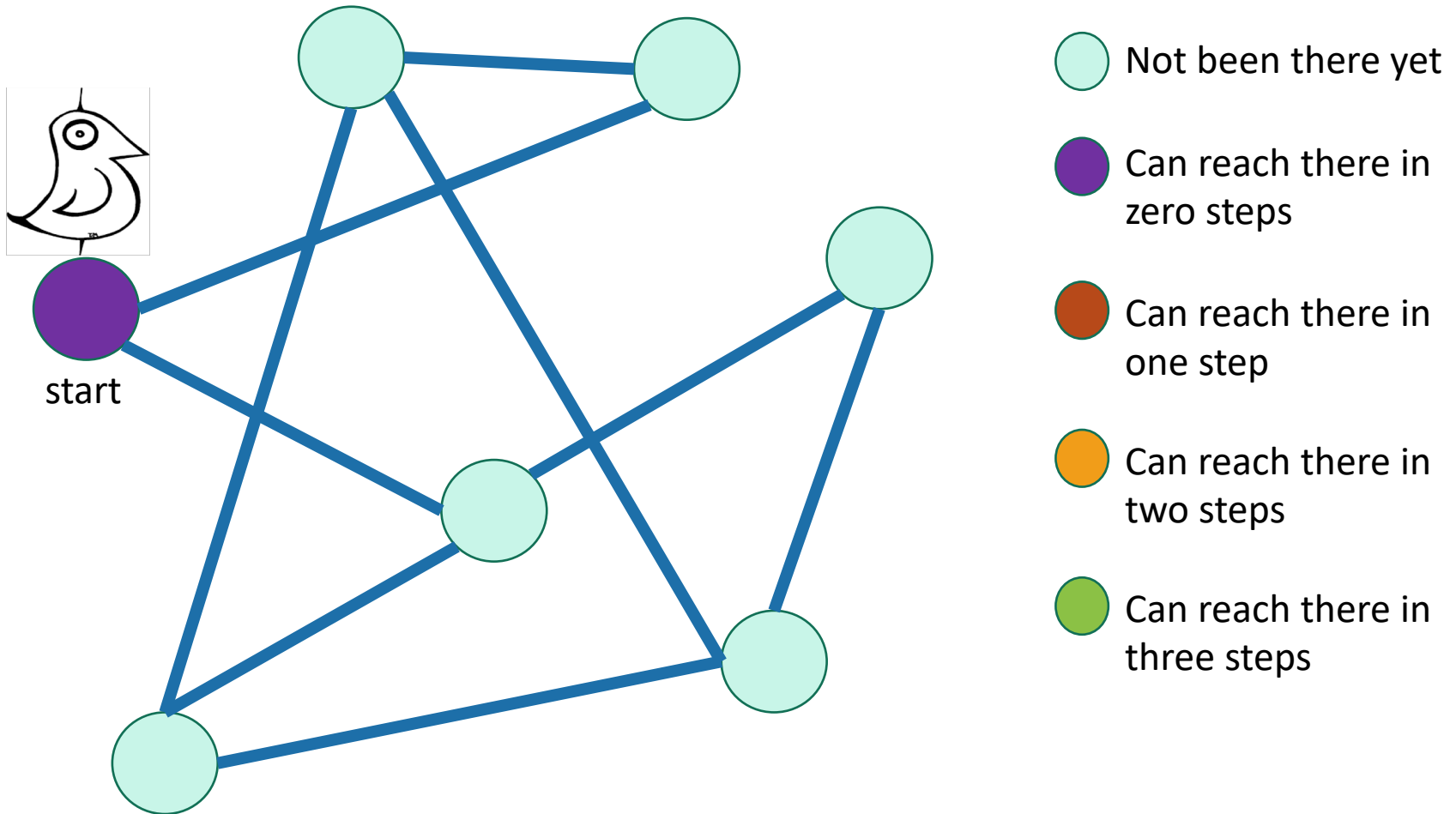
Exploring the world with a bird's-eye view



-  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

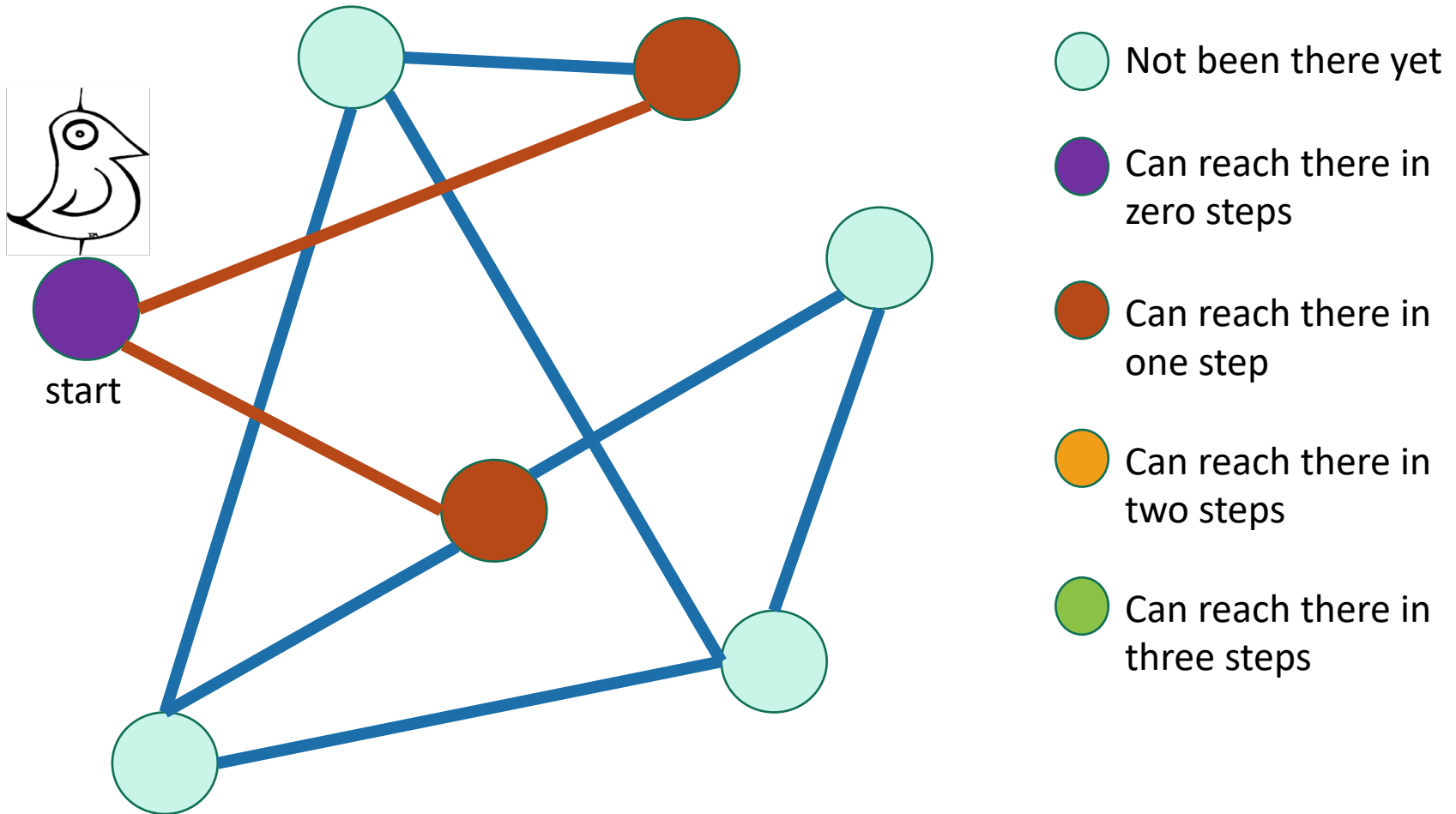
# Breadth-First Search

Exploring the world with a bird's-eye view



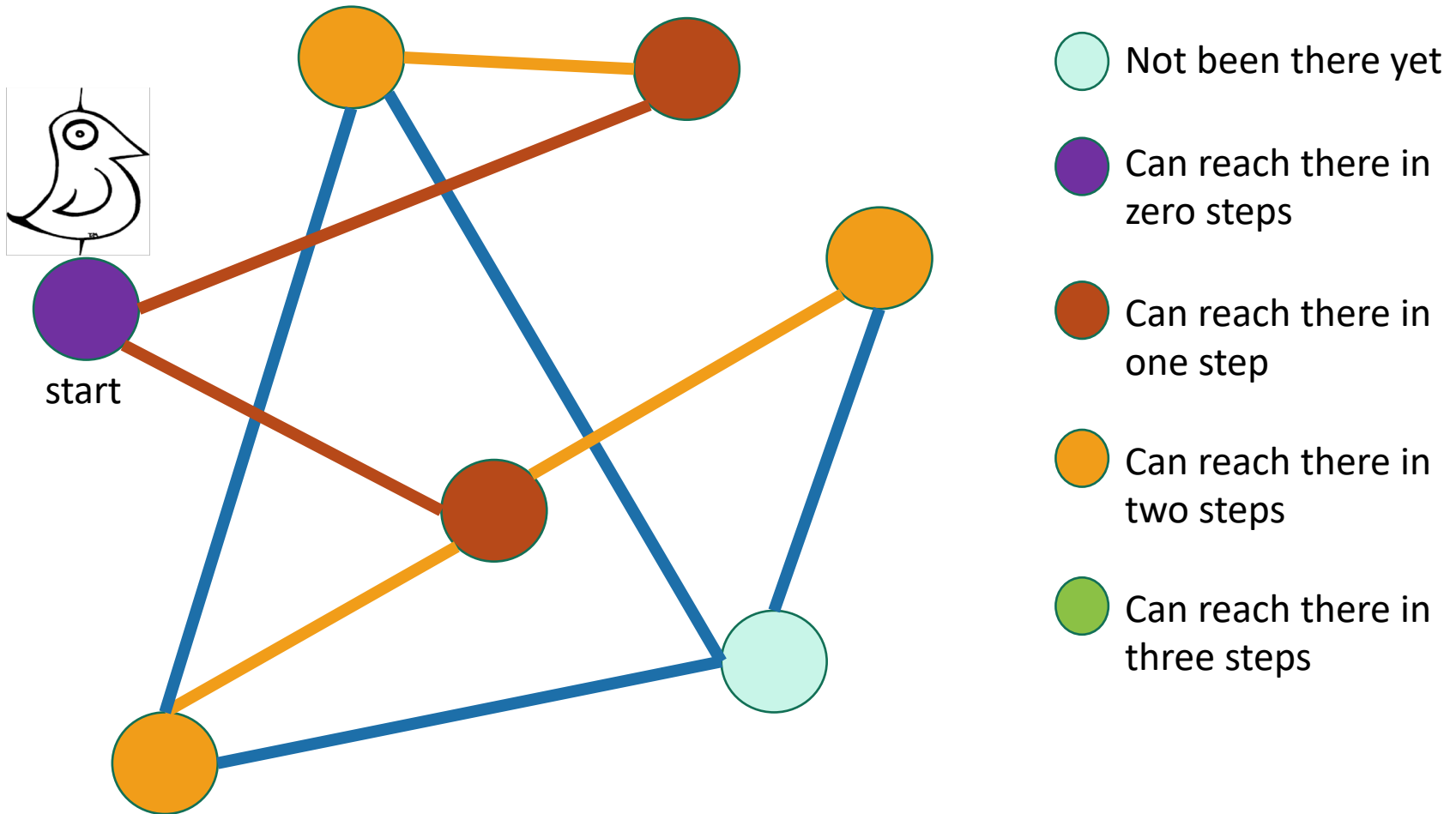
# Breadth-First Search

Exploring the world with a bird's-eye view



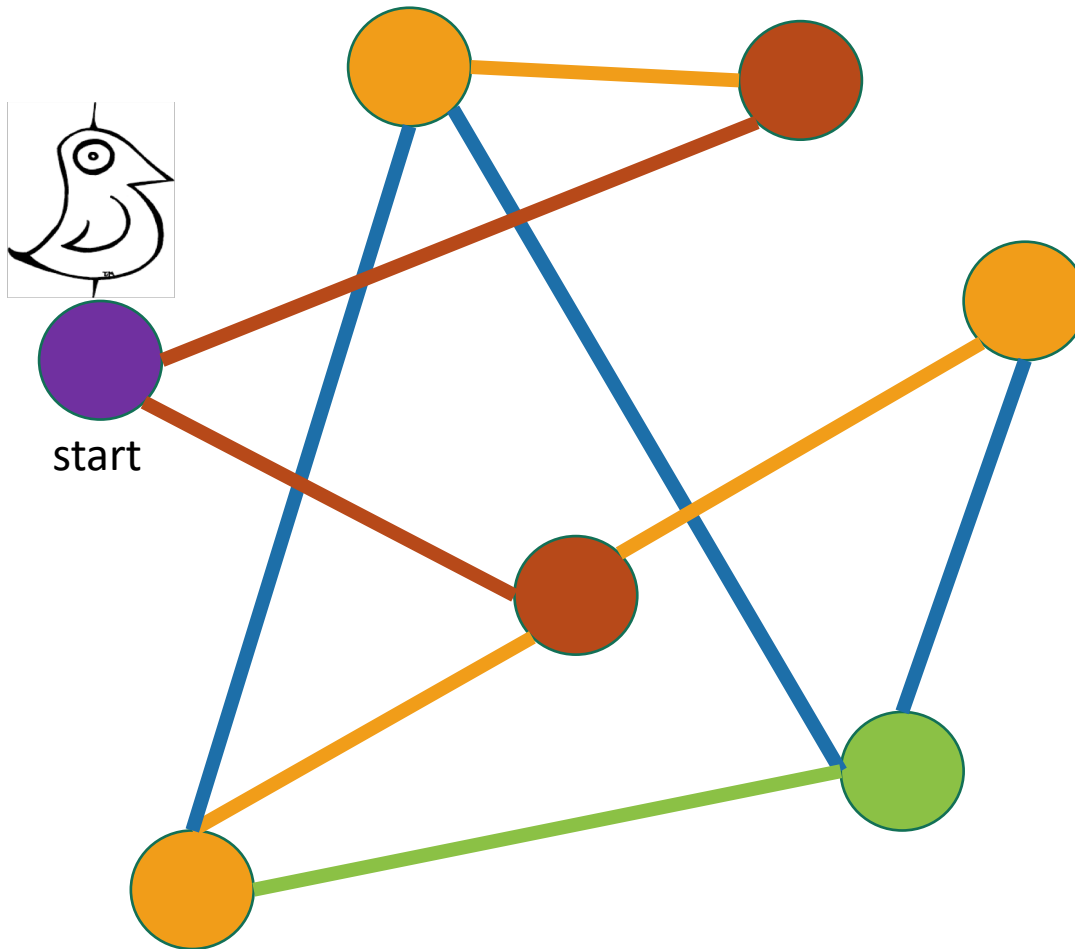
# Breadth-First Search

Exploring the world with a bird's-eye view



# 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!**

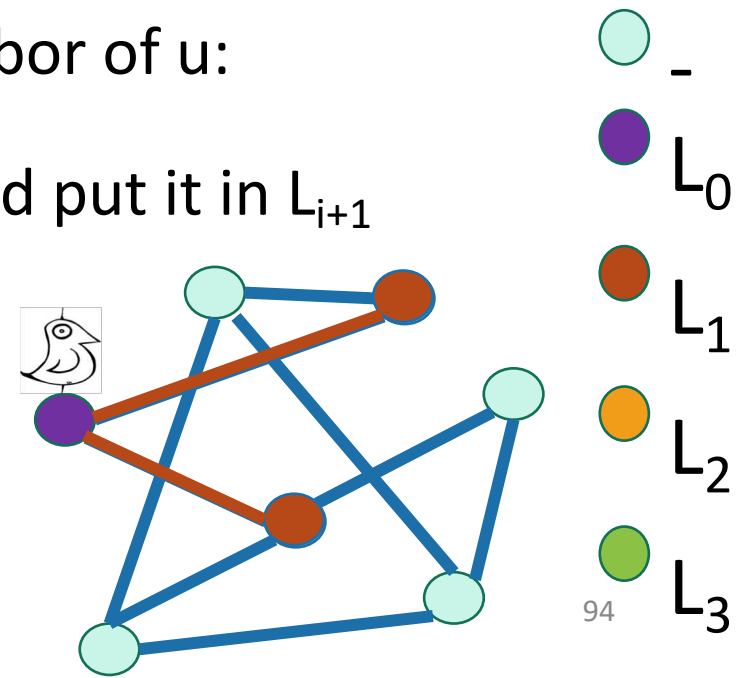
# Breadth-First Search

## Exploring the world with pseudocode

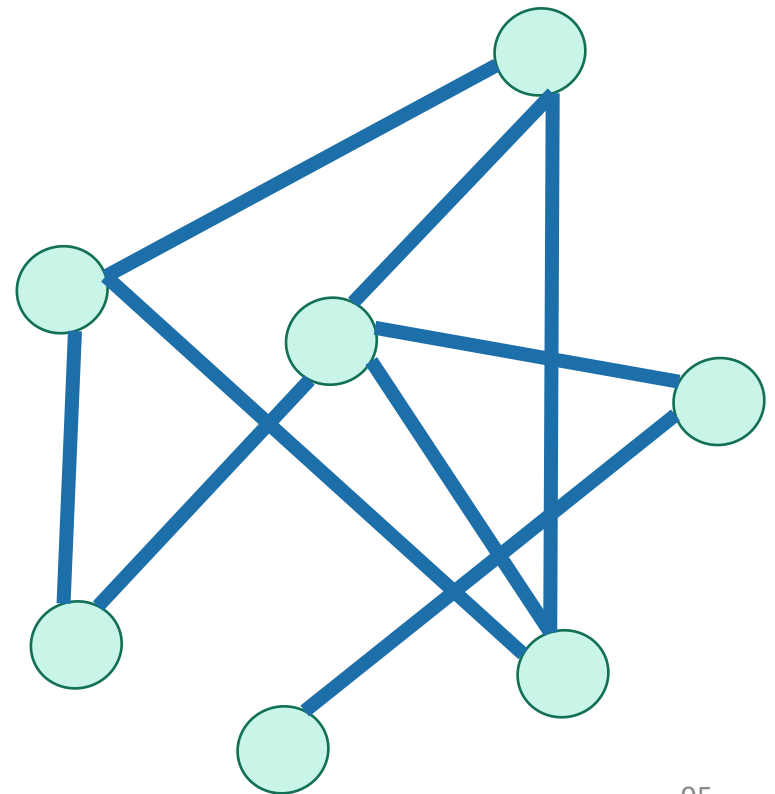
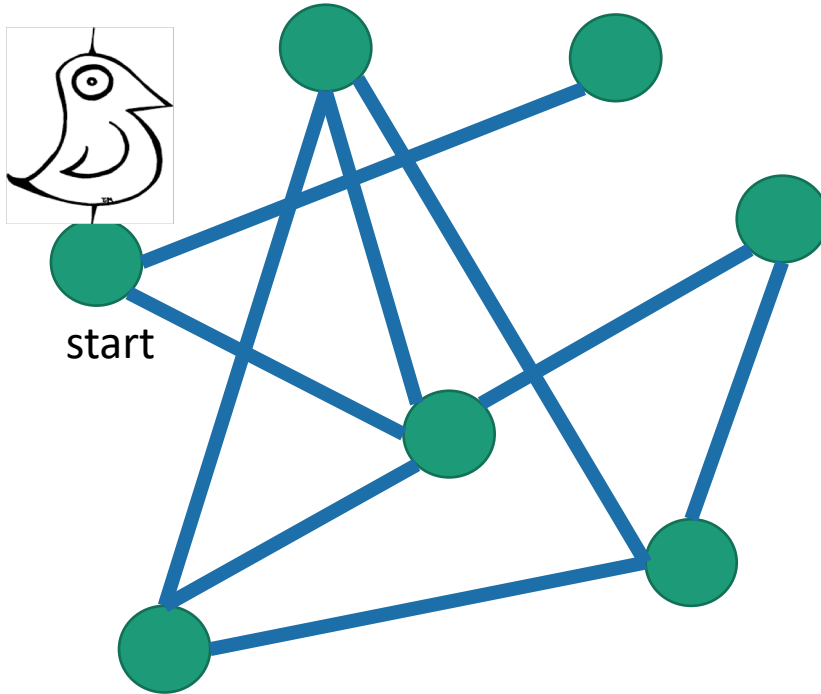
- Set  $L_i = []$  for  $i=1, \dots, n$
- $L_0 = [w]$ , where  $w$  is the start node
- Mark  $w$  as visited
- **For**  $i = 0, \dots, n-1$ :
  - **For**  $u$  in  $L_i$ :
    - **For** each  $v$  which is a neighbor of  $u$ :
      - **If**  $v$  isn't yet visited:
        - mark  $v$  as visited, and put it in  $L_{i+1}$

$L_i$  is the set of nodes  
we can reach in  $i$   
steps from  $w$

Go through all the nodes  
in  $L_i$  and add their  
unvisited neighbors to  $L_{i+1}$



# BFS also finds all the nodes reachable from the starting point



It is also a good way to find all the **connected components**.

# Running time and extension to directed graphs

- To explore the whole graph, explore the connected components one-by-one.
  - Same argument as DFS: BFS running time is  $O(n + m)$
- Like DFS, BFS also works fine on directed graphs.

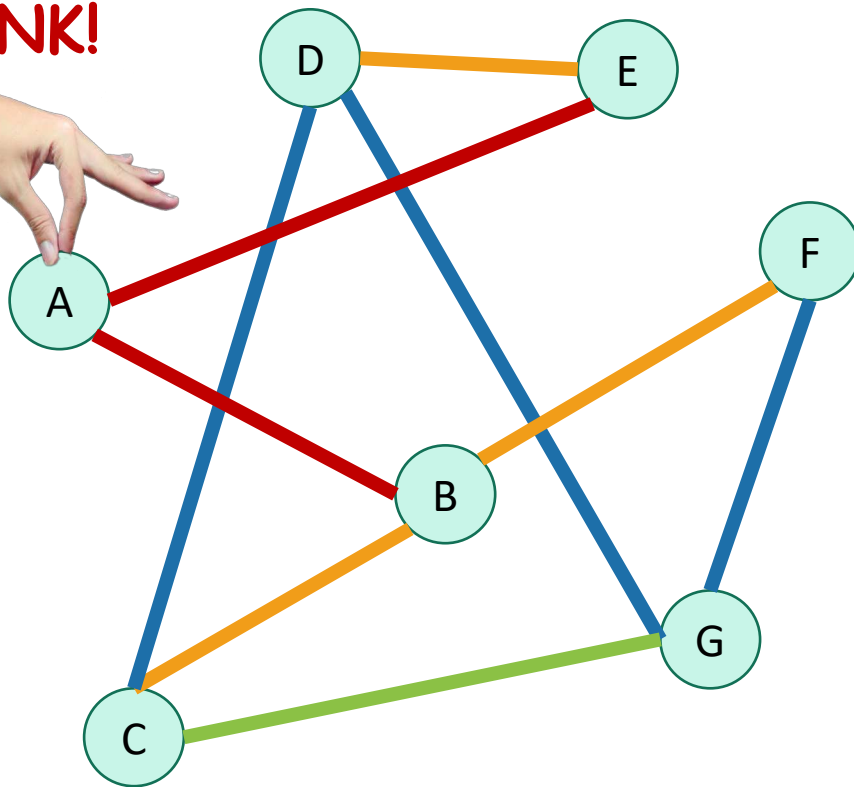
Verify these!



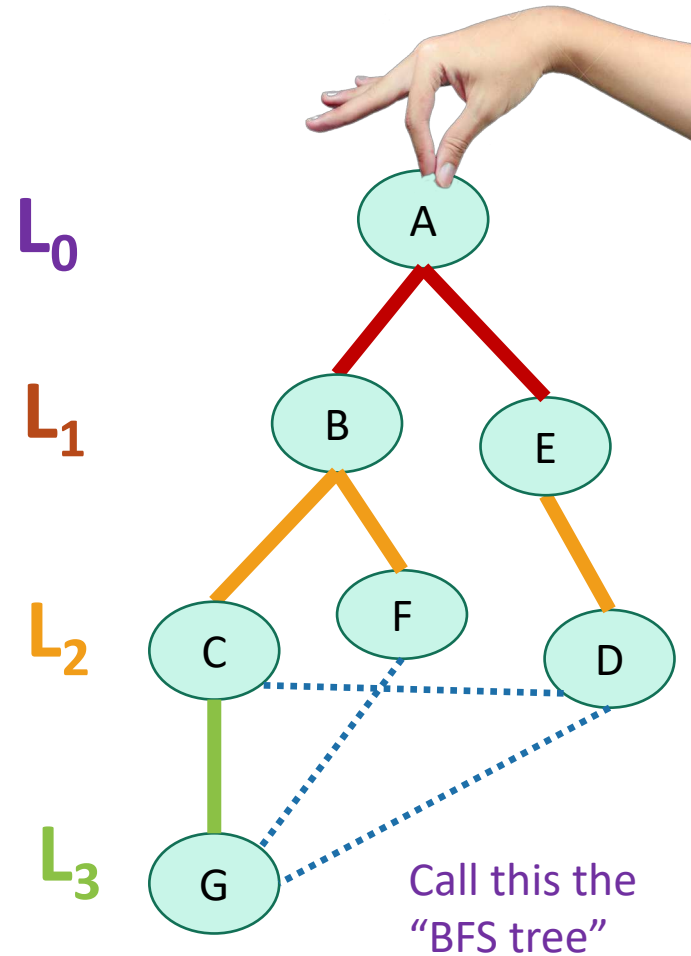
# Why is it called breadth-first?

- We are implicitly building a tree:

YOINK!

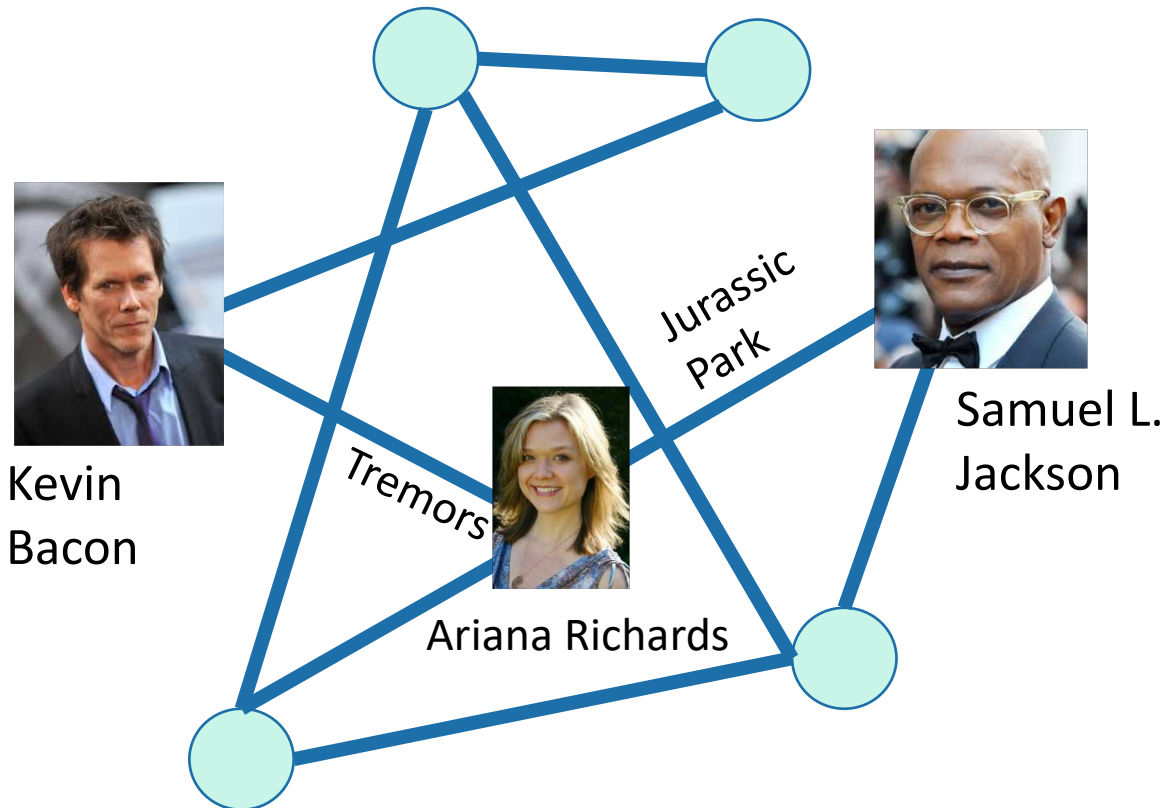


- First we go as broadly as we can.



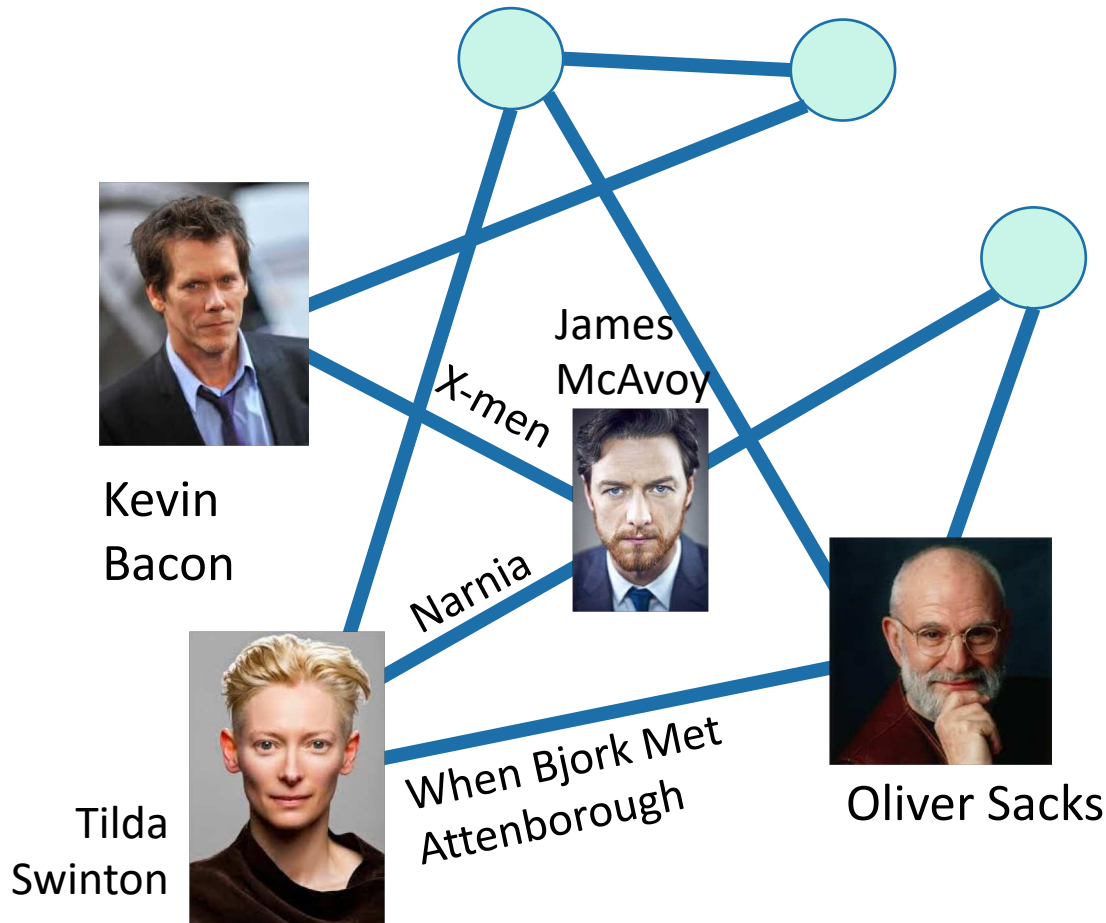
# Pre-lecture exercise

- What Samuel L. Jackson's Bacon number?



(Answer: 2)

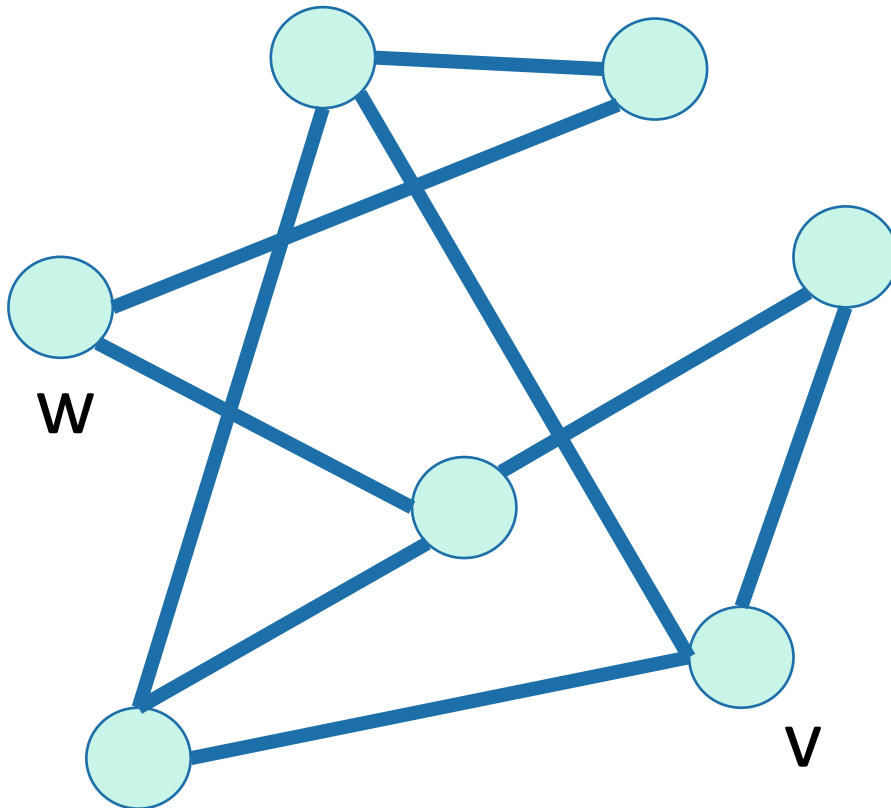
I wrote the pre-lecture exercise  
before I realized that I really wanted  
an example with distance 3



It is really hard to find  
people with Bacon  
number 3!

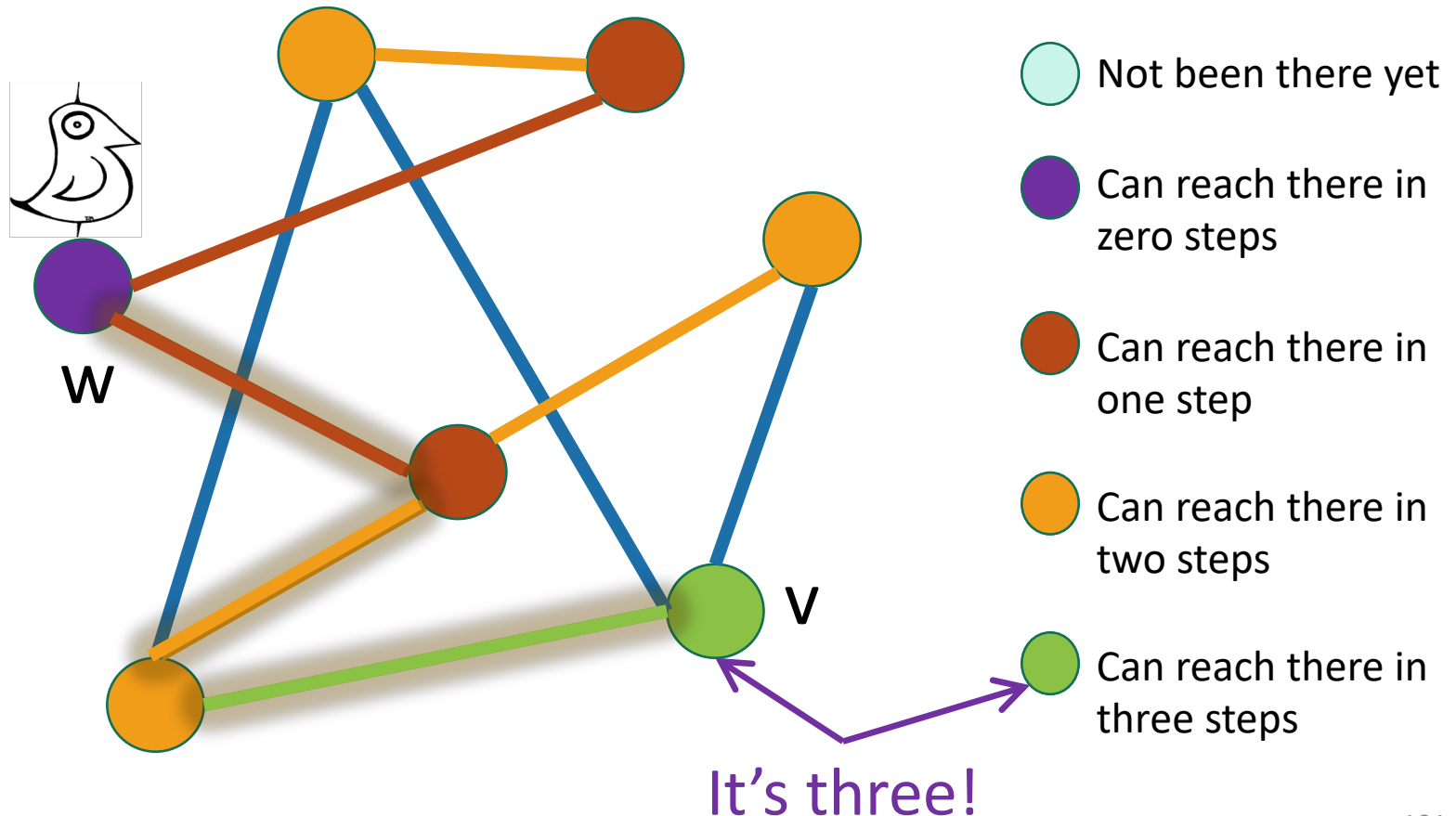
# Application of BFS: shortest path

- How long is the shortest path between  $w$  and  $v$ ?



# Application of BFS: shortest path

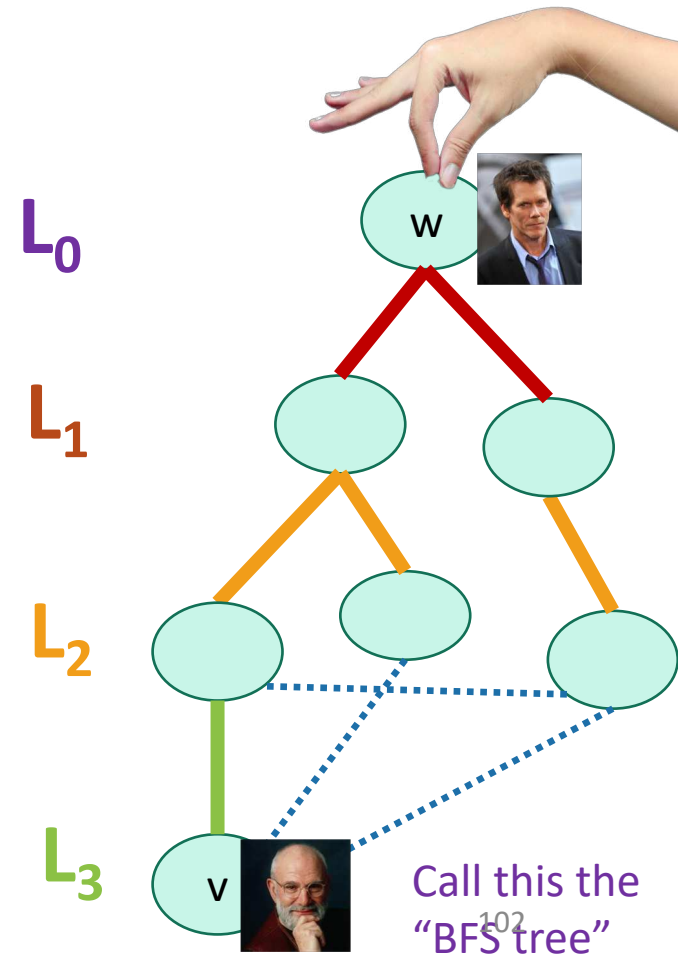
- How long is the shortest path between w and v?



# To find the **distance** between $w$ and all other vertices $v$

- Do a BFS starting at  $w$
- For all  $v$  in  $L_i$ 
  - The shortest path between  $w$  and  $v$  has length  $i$
  - A shortest path between  $w$  and  $v$  is given by the path in the BFS tree.
- If we never found  $v$ , the distance is infinite.

The **distance** between two vertices is the number of edges in the shortest path between them.



Modify the BFS pseudocode  
to return shortest paths!  
Prove that this indeed  
returns shortest paths!



Gauss has no  
Bacon number



# What have we learned?

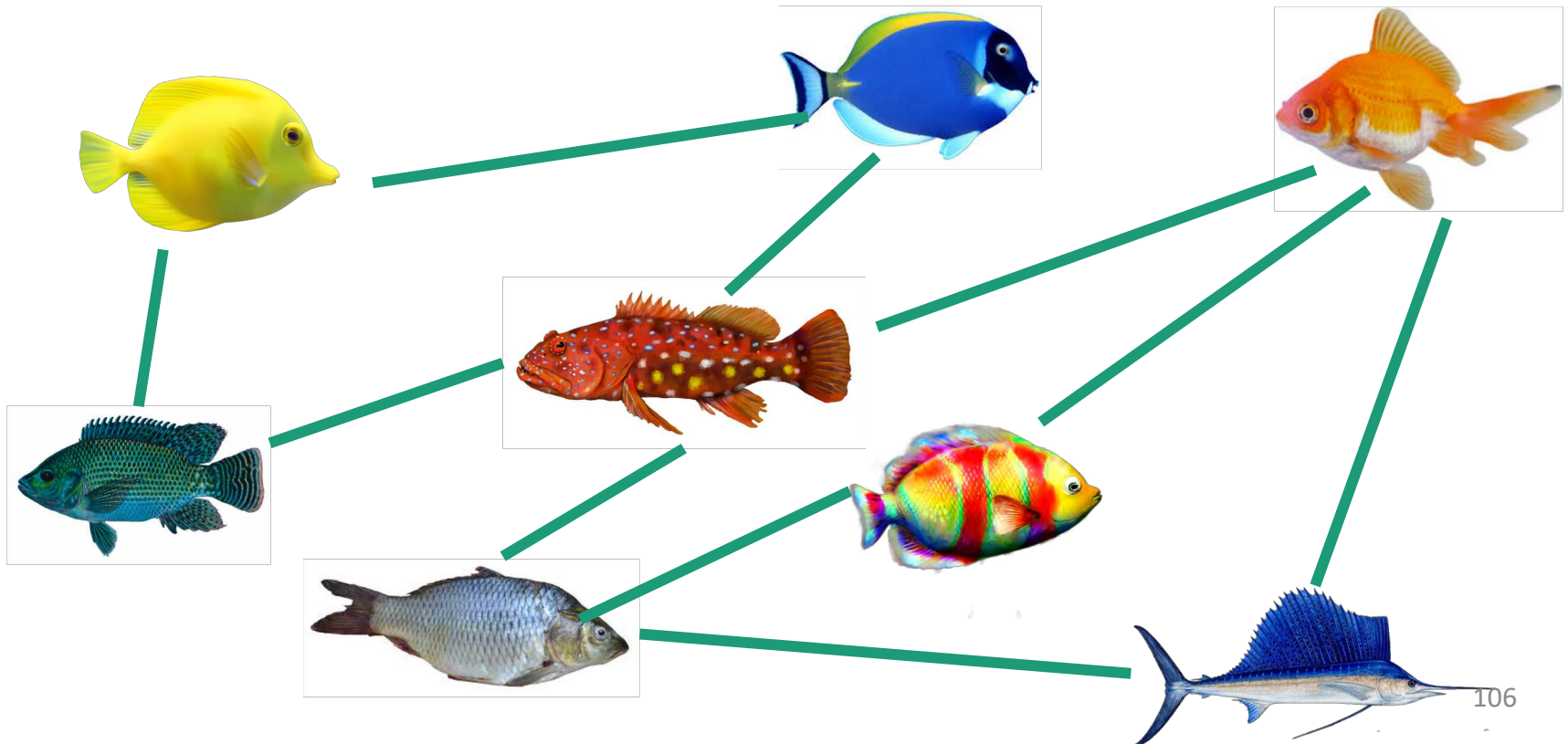
- The BFS tree is useful for computing distances between pairs of vertices.
- We can find the shortest path between  $u$  and  $v$  in time  $O(m)$ .

# Another application of BFS

- Testing bipartite-ness

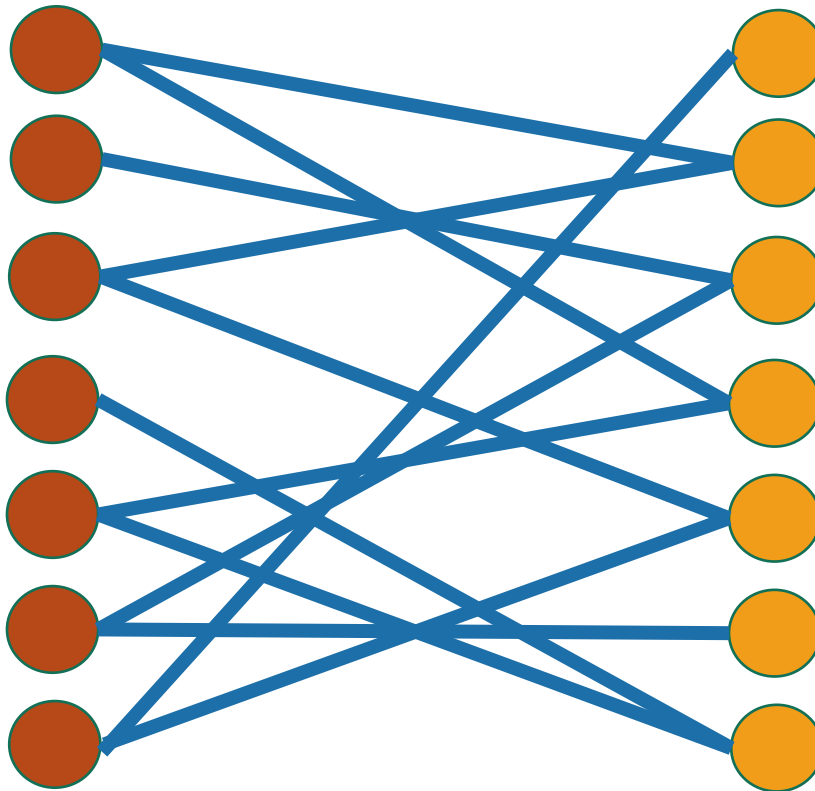
# Pre-lecture exercise: fish

- You have a bunch of fish and two fish tanks.
- Some pairs of fish will fight if put in the same tank.
  - Model this as a graph: connected fish will fight.
- Can you put the fish in the two tanks so that there is no fighting?



# Bipartite graphs

- A bipartite graph looks like this:



Can color the vertices red and orange so that there are no edges between any same-colored vertices

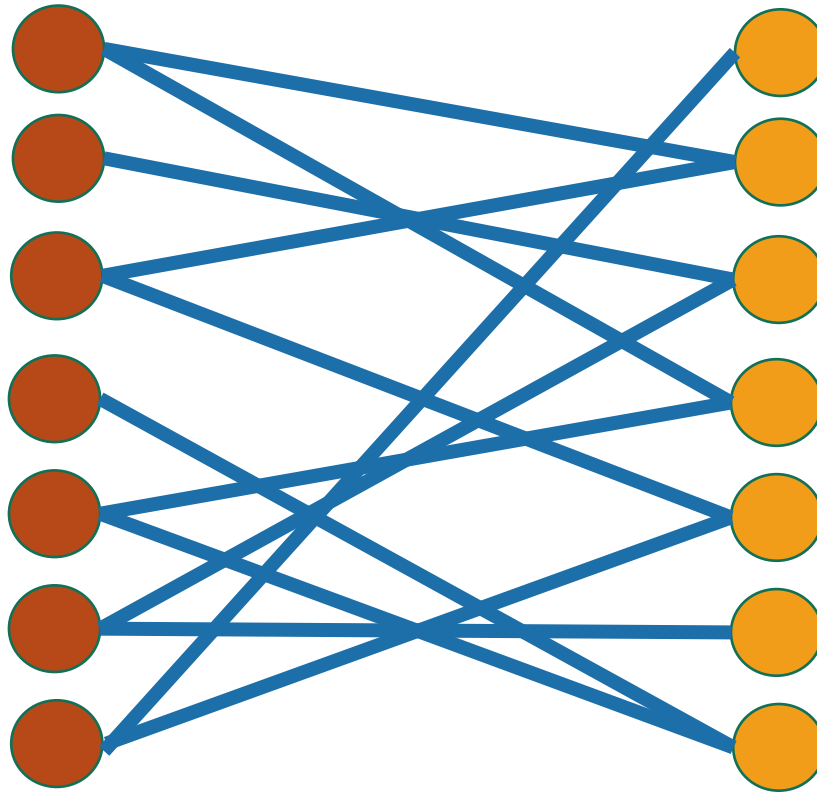
## Example:

- are in tank A
- are in tank B
- — ● if the fish fight

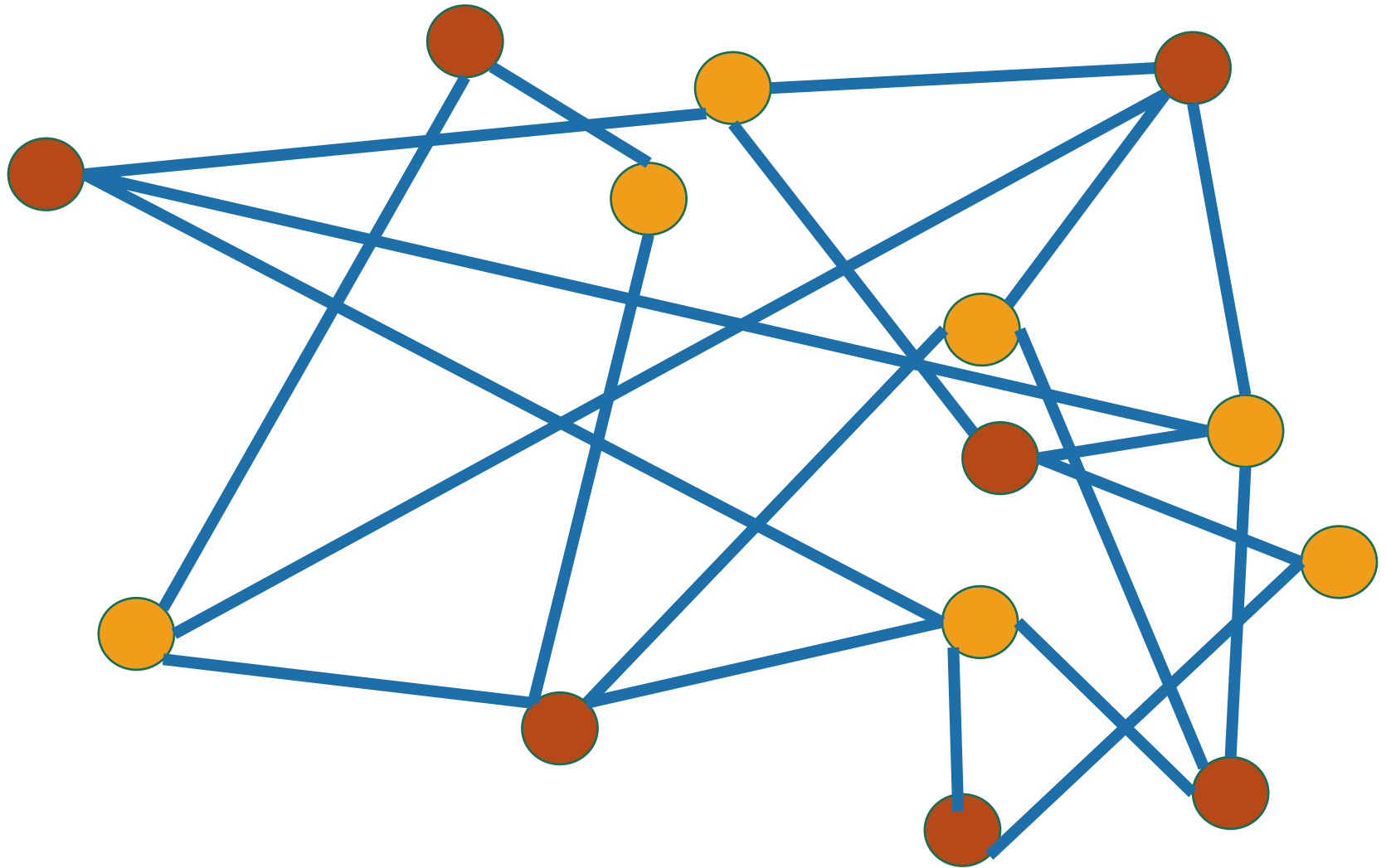
## Example:

- are students
- are classes
- — ● if the student is enrolled in the class

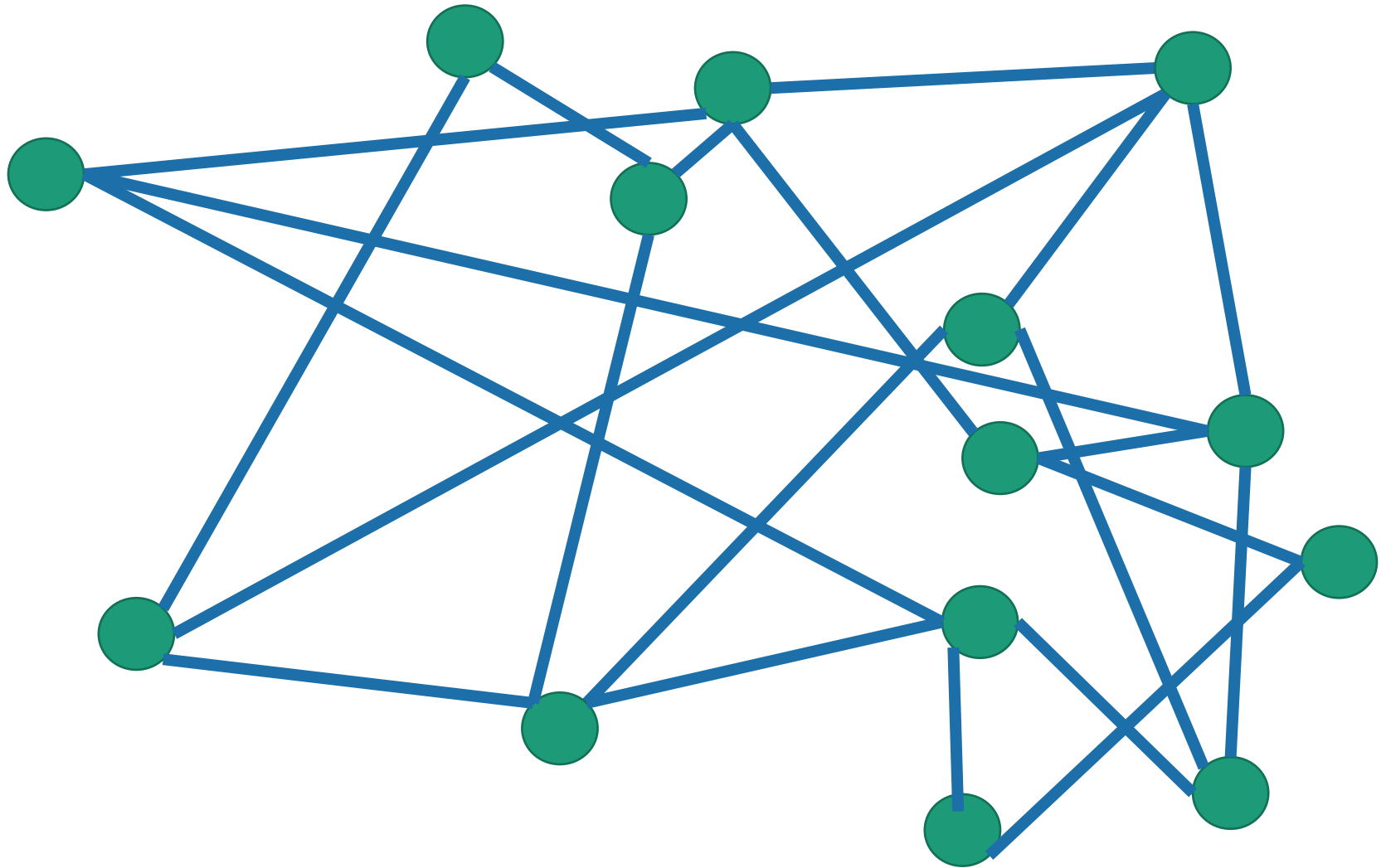
Is this graph bipartite?



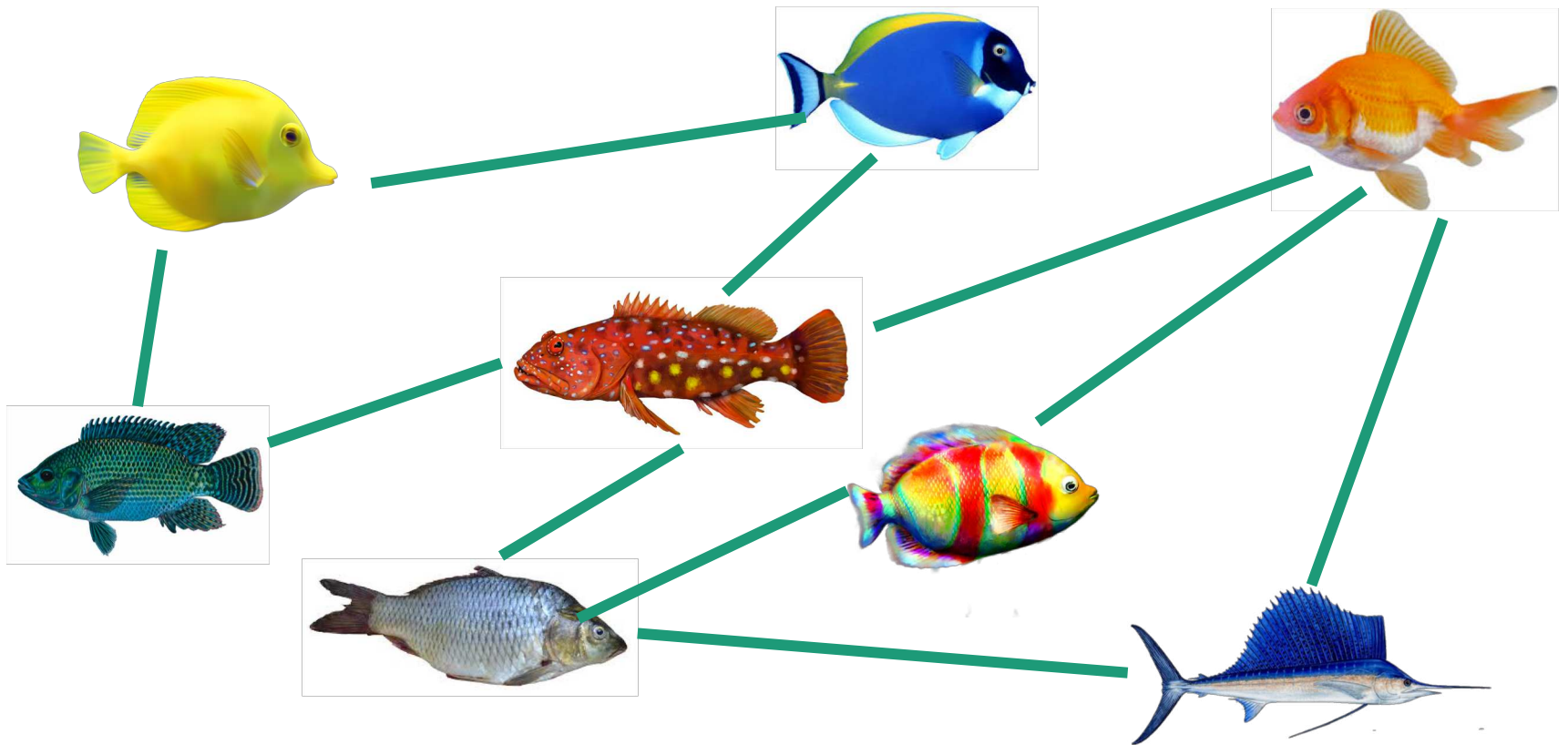
How about this one?



How about this one?

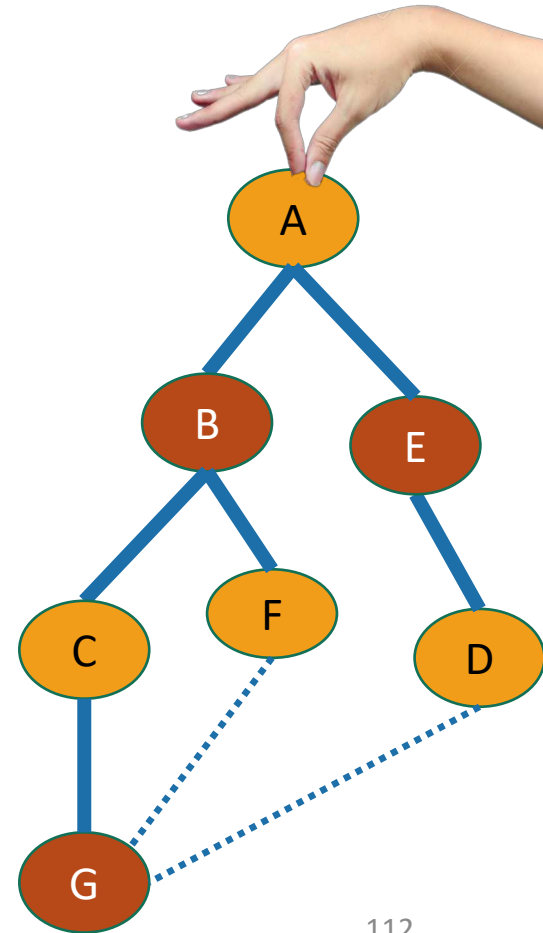


# This one?



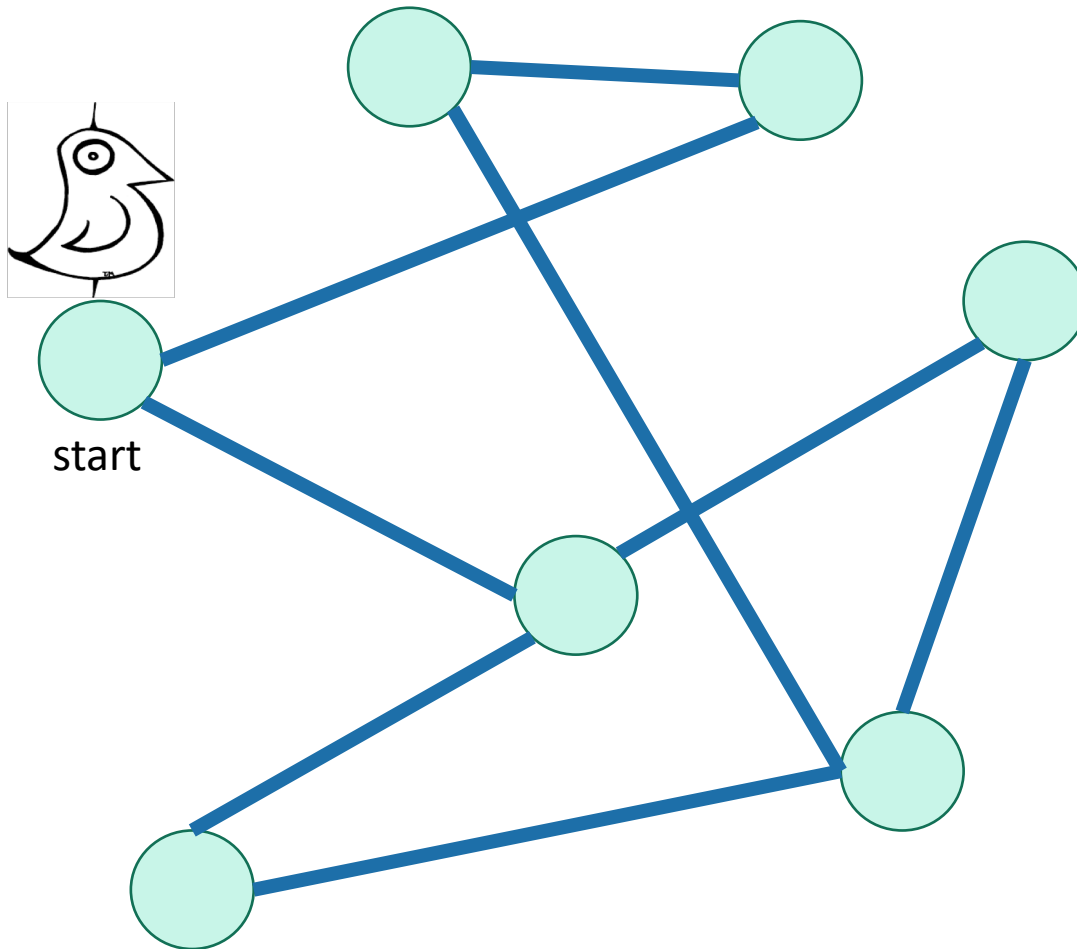
# Application of BFS: Testing Bipartiteness

- Color the levels of the BFS tree in alternating colors.
- If you never color two connected nodes the same color, then it is bipartite.
- Otherwise, it's not.



# Breadth-First Search

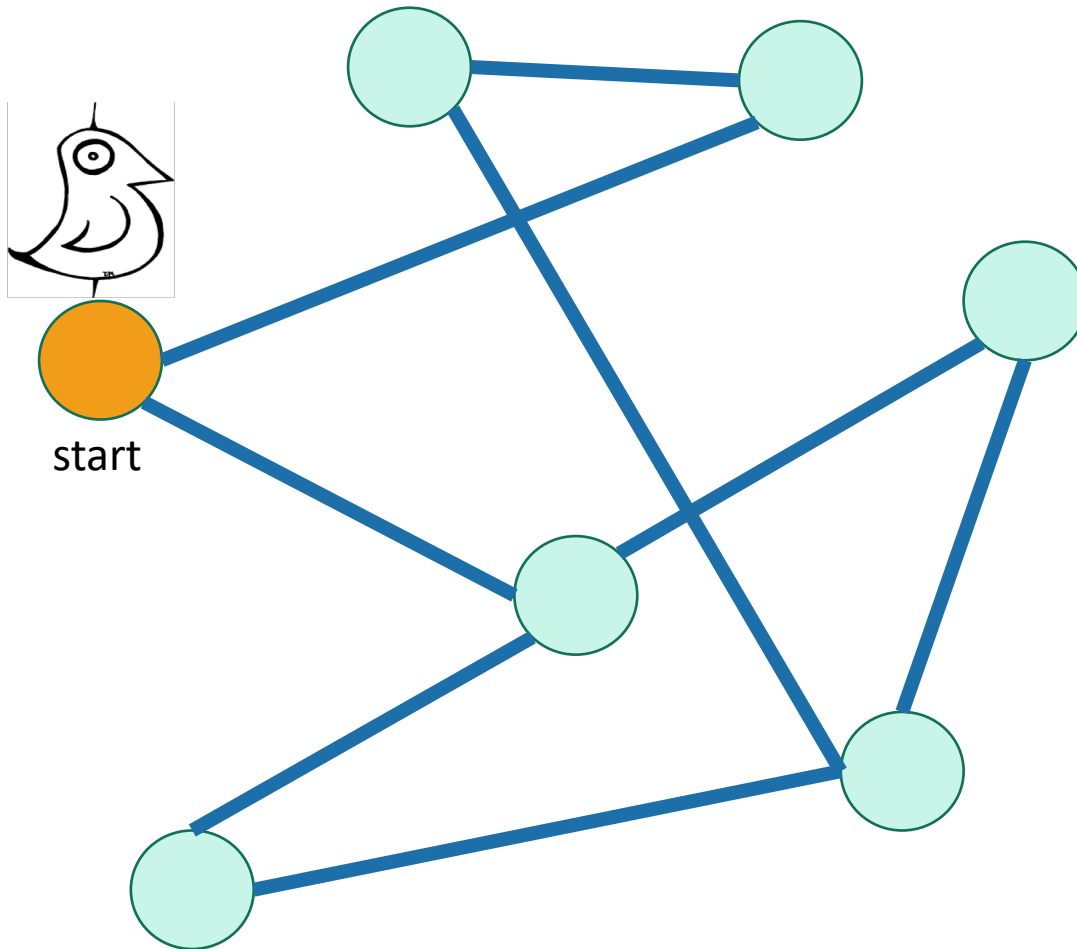
## For testing bipartite-ness



-  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

# Breadth-First Search

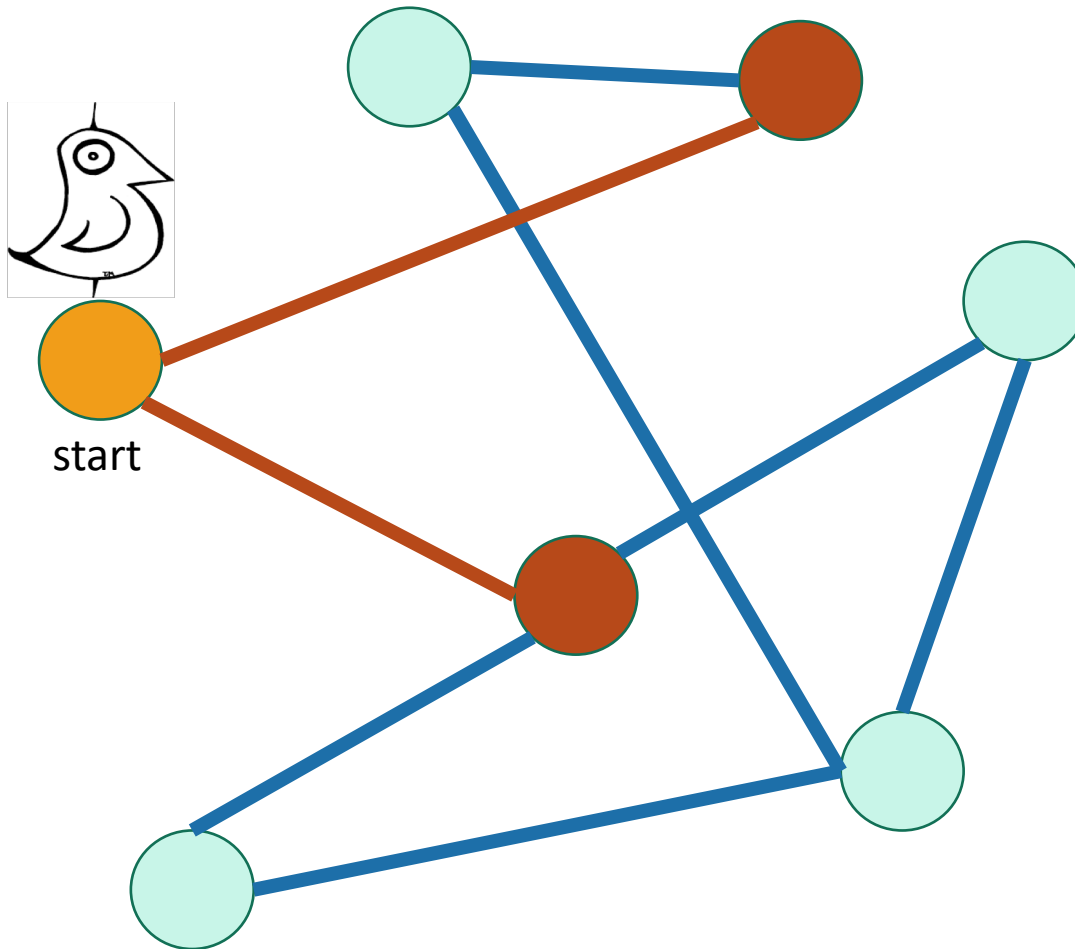
## For testing bipartite-ness



-  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

# Breadth-First Search

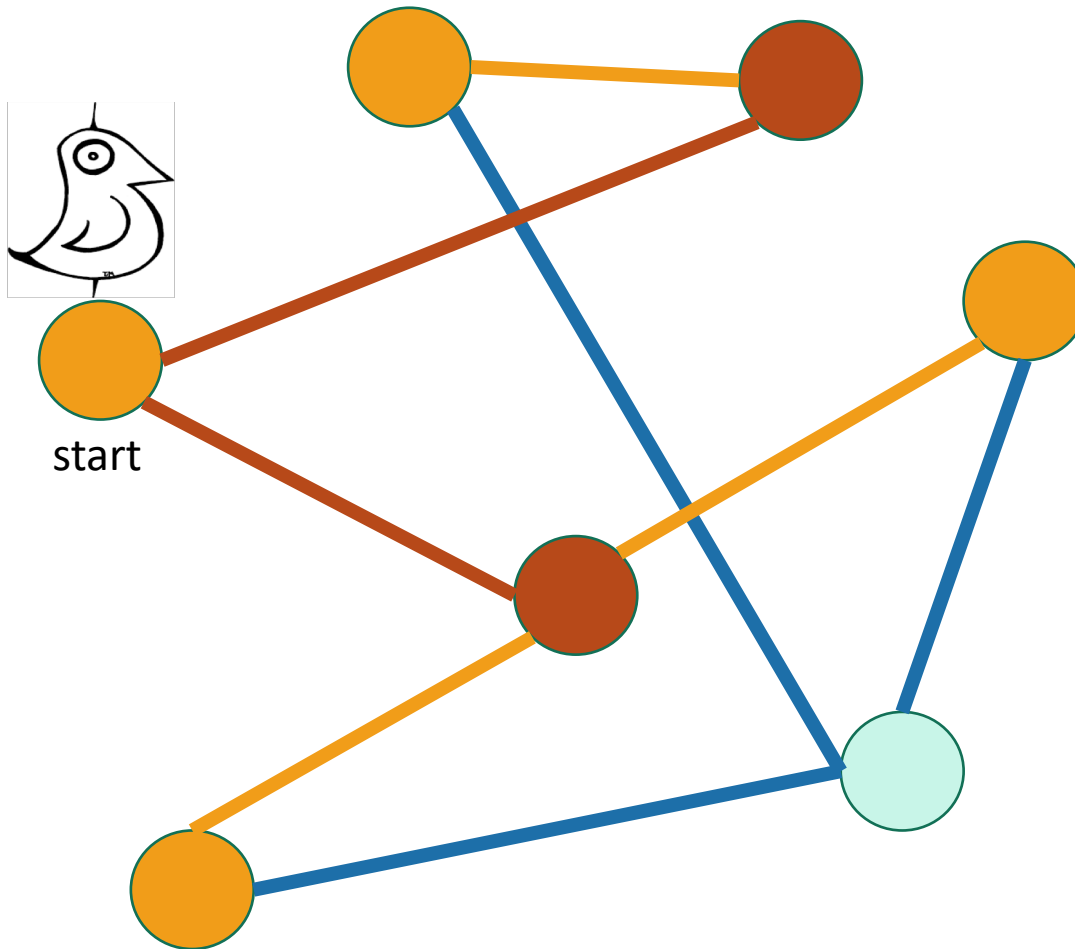
## For testing bipartite-ness



-  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

# Breadth-First Search

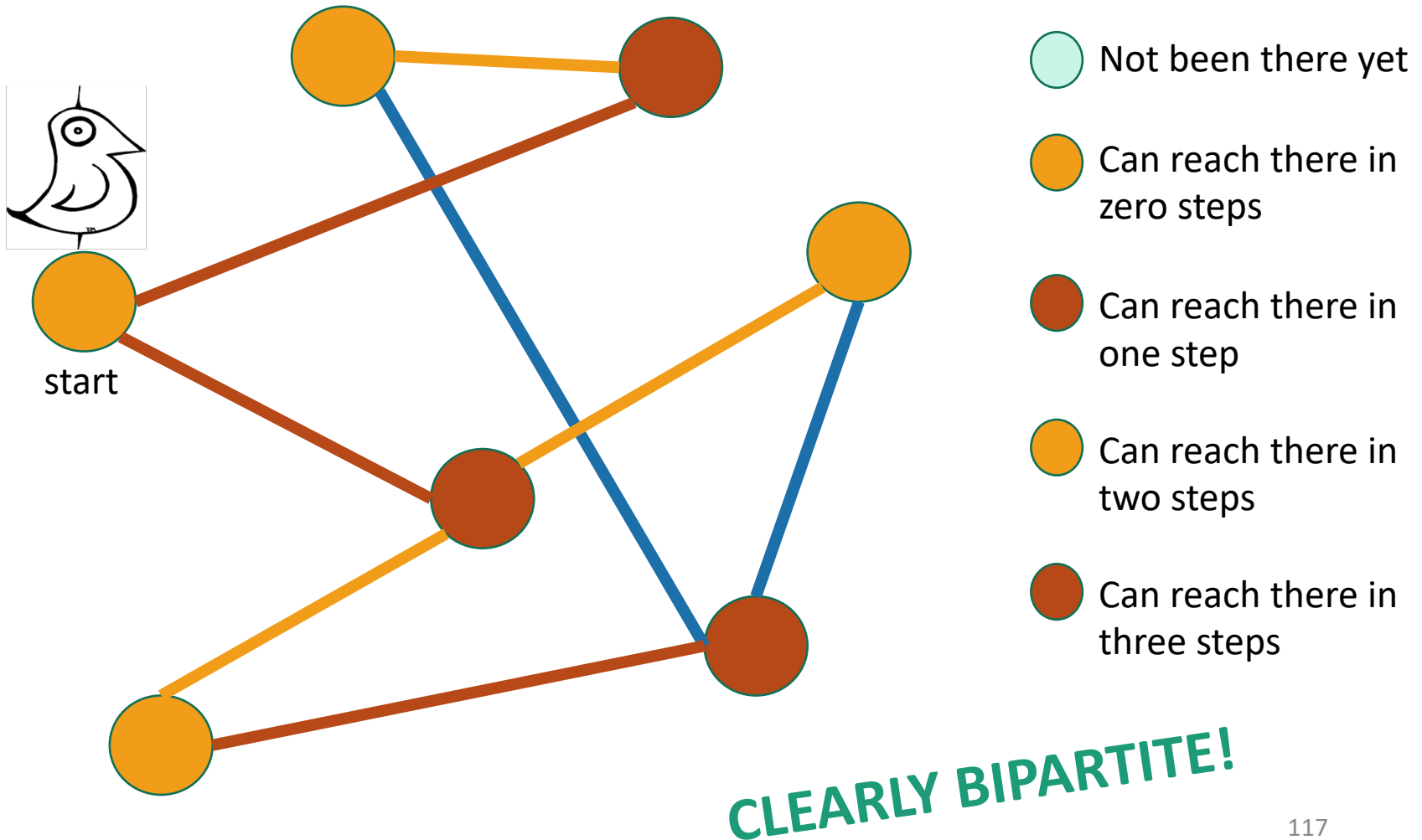
## For testing bipartite-ness



- 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

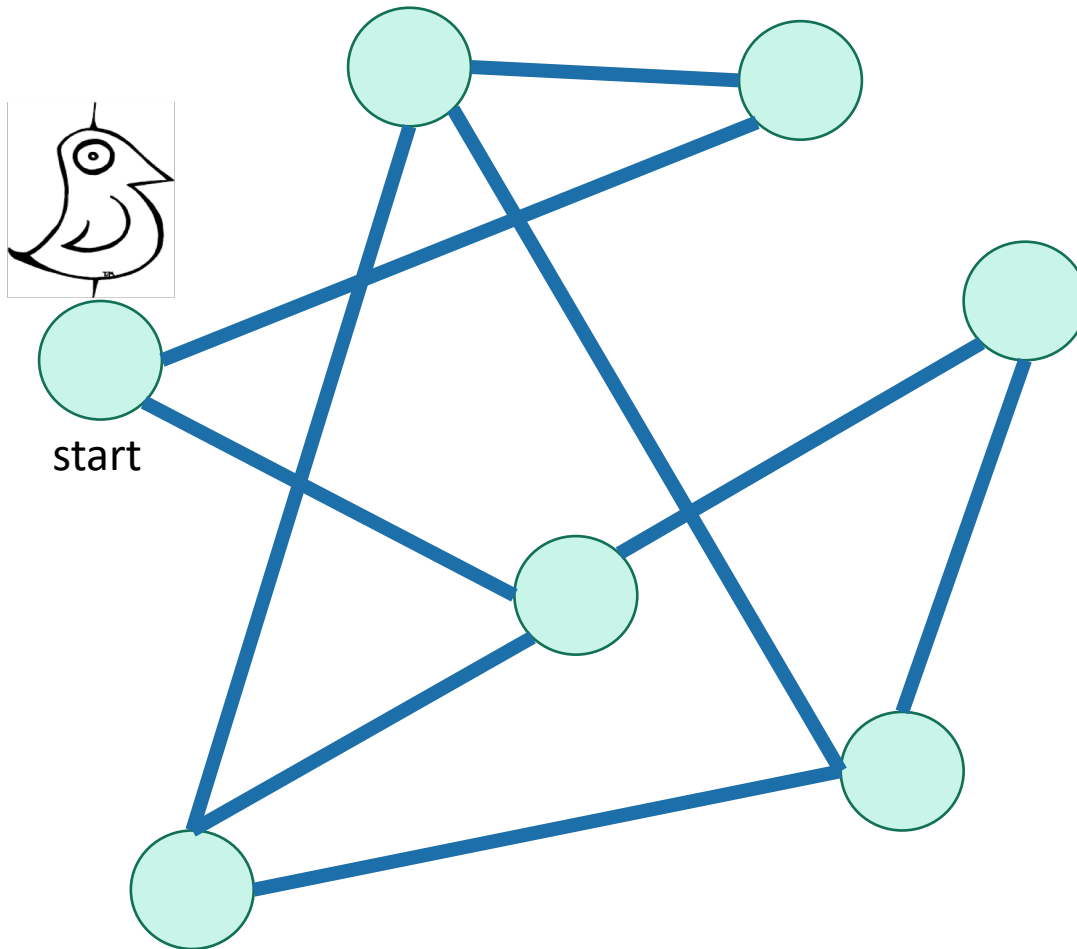
# Breadth-First Search

## For testing bipartite-ness



# Breadth-First Search

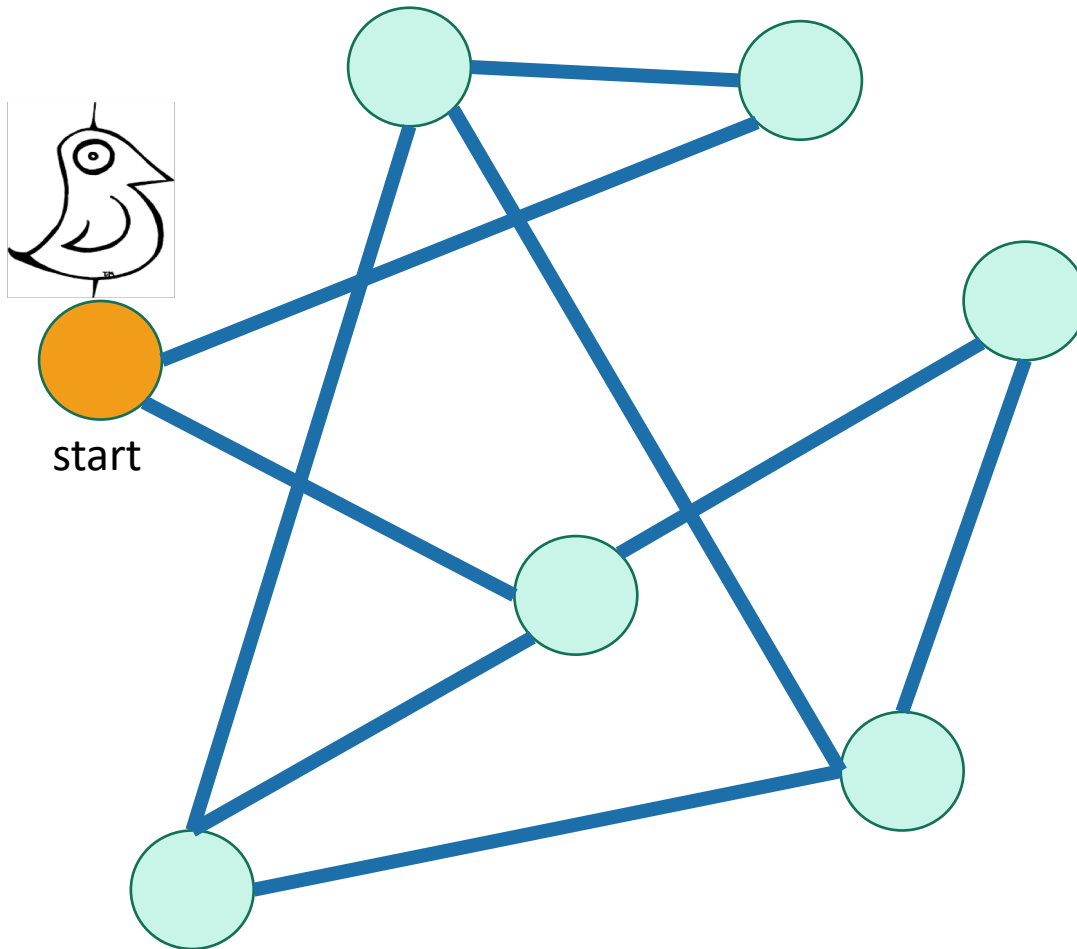
## For testing bipartite-ness



-  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

# Breadth-First Search

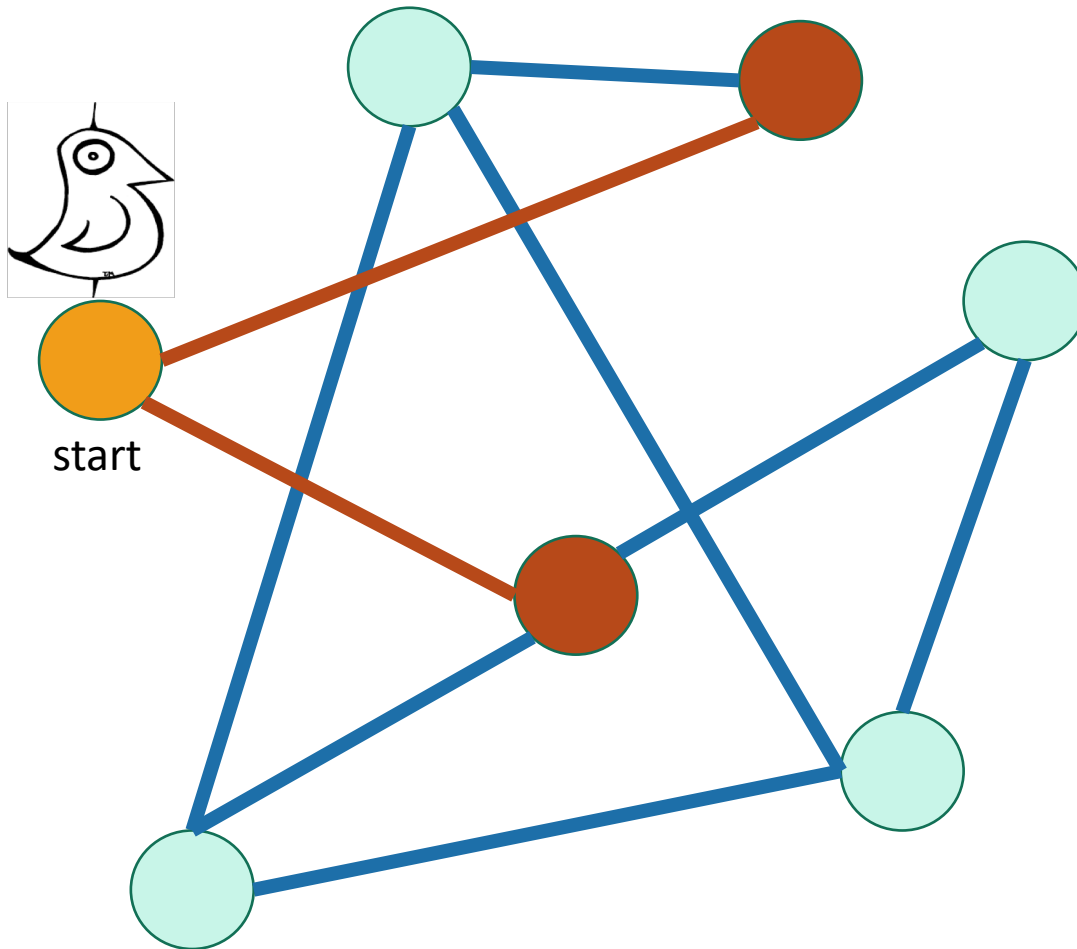
## For testing bipartite-ness



-  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

# Breadth-First Search

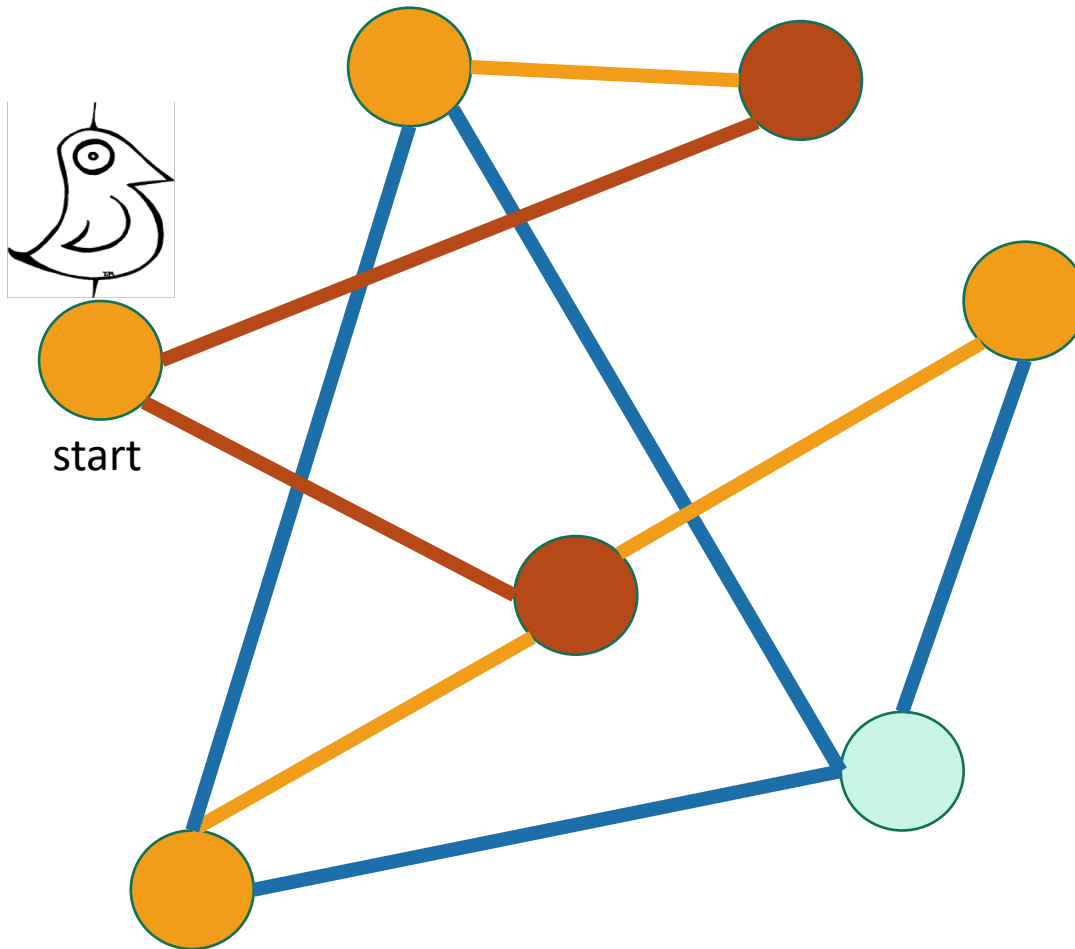
## For testing bipartite-ness



-  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

# Breadth-First Search

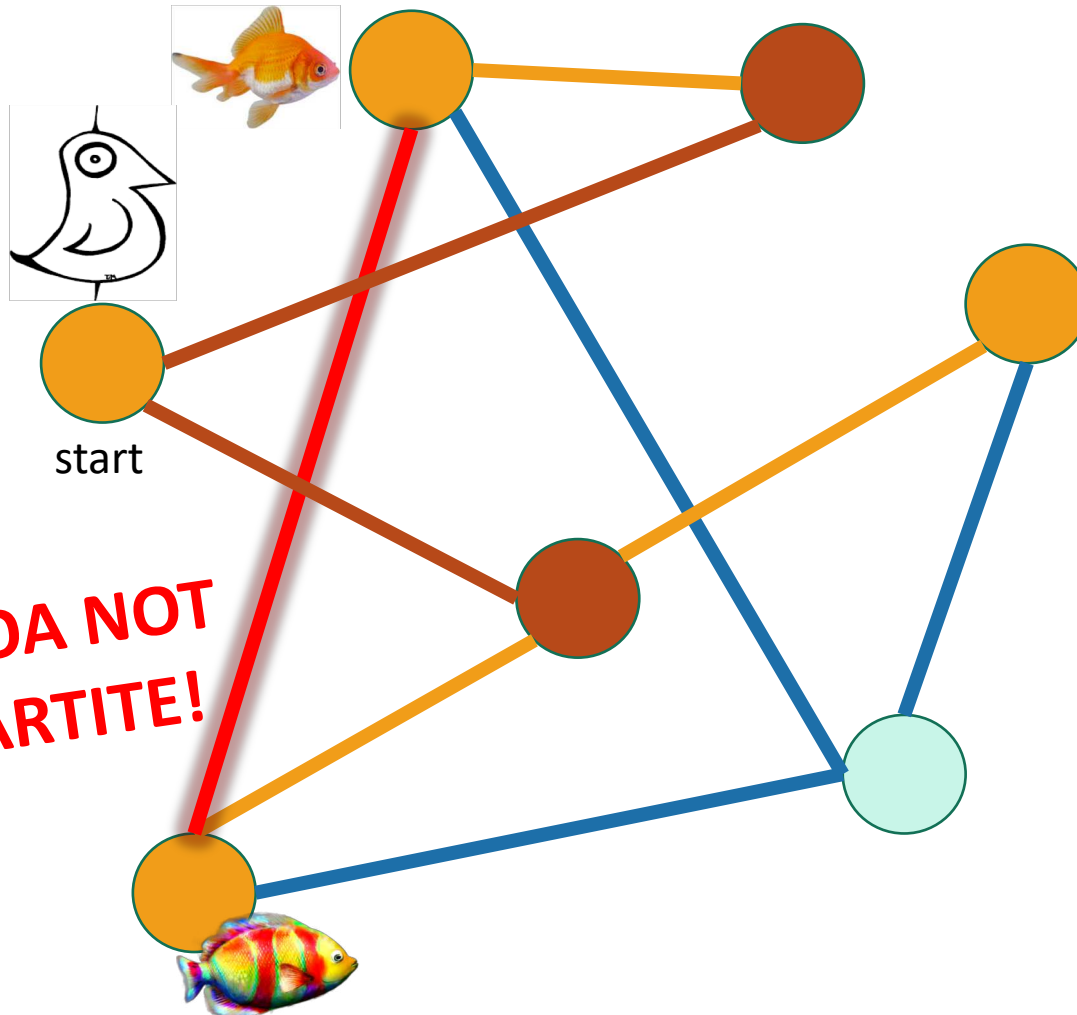
## For testing bipartite-ness



- 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

# Breadth-First Search

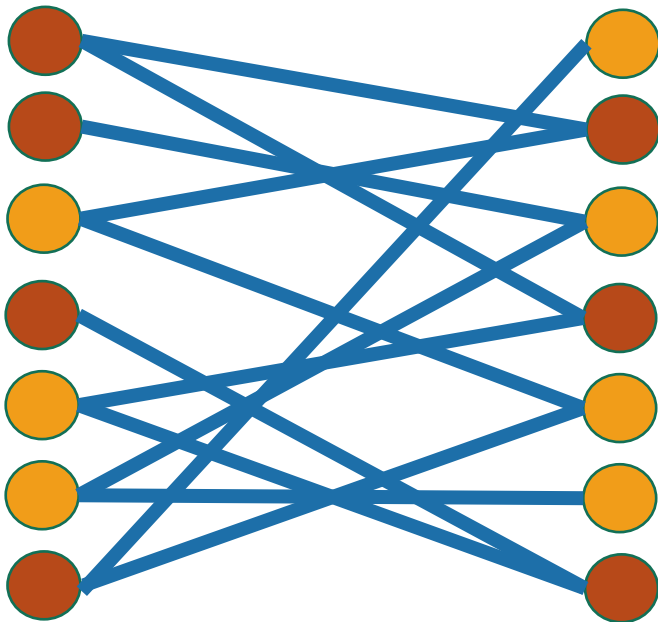
## For testing bipartite-ness



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

# Hang on now.

- Just because **this** coloring doesn't work, why does that mean that there is **no** coloring that works?

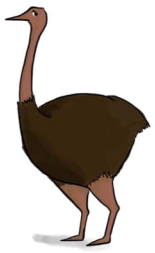


I can come up  
with plenty of bad  
colorings on this  
legitimately  
bipartite graph...



Plucky the  
pedantic penguin

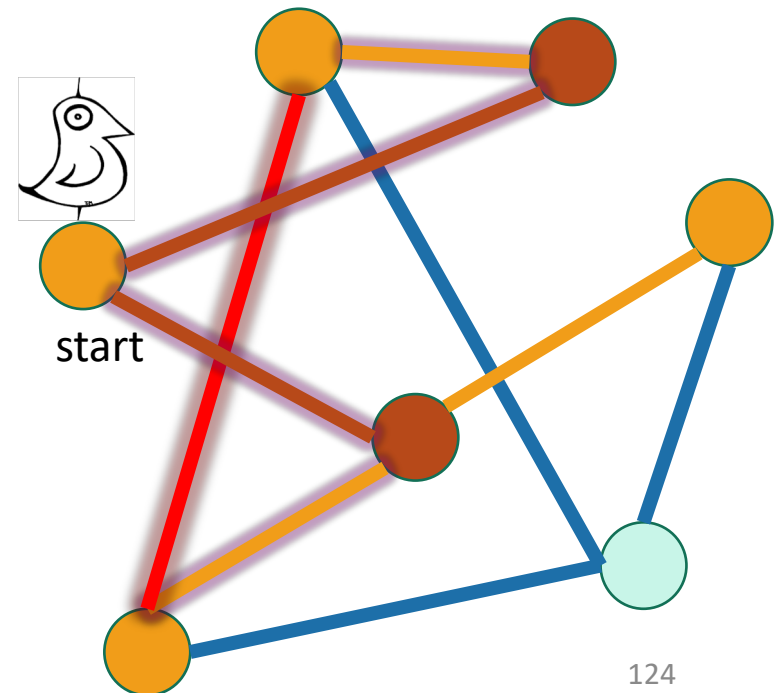
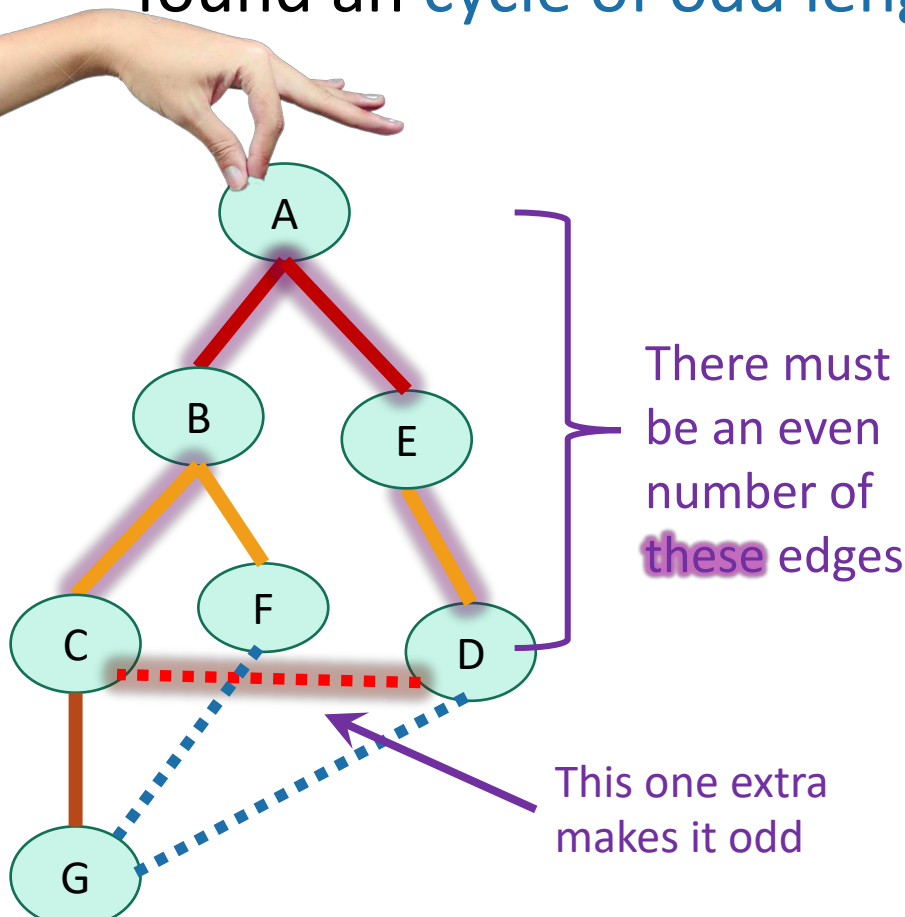
Make this proof  
sketch formal!



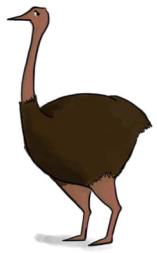
Ollie the over-achieving ostrich

# Some proof required

- If BFS colors two neighbors the same color, then it's found an **cycle of odd length** in the graph.



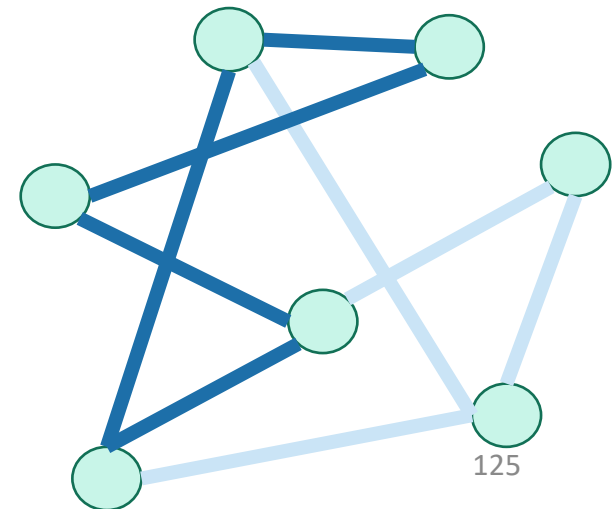
Make this proof  
sketch formal!



Ollie the over-achieving ostrich

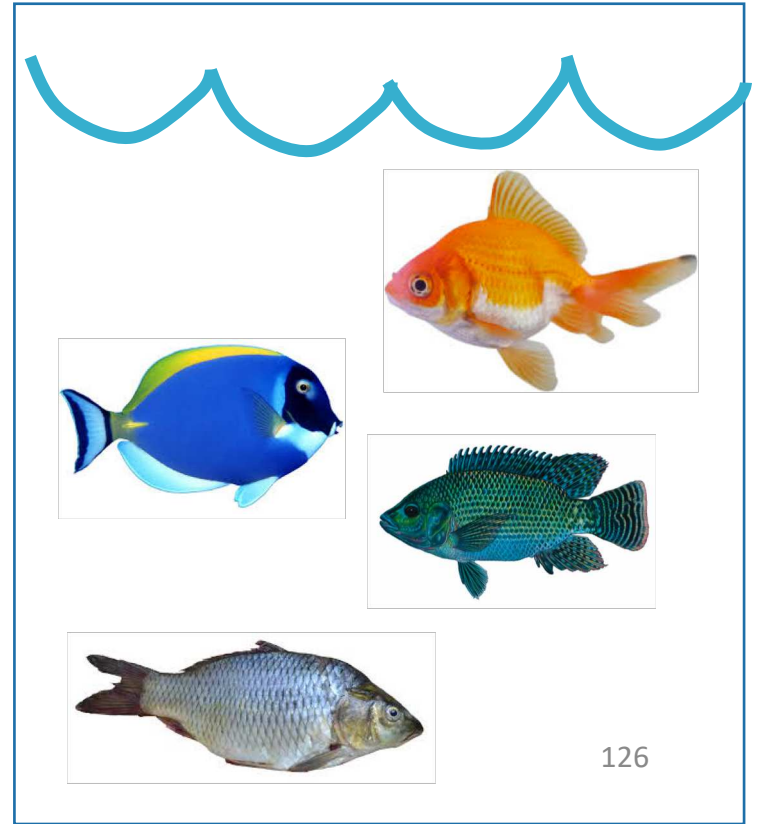
# Some proof required

- If BFS colors two neighbors the same color, then it's found an **cycle of odd length** in the graph.
- But you can **never** color an odd cycle with two colors so that no two neighbors have the same color.
  - [Fun exercise!]
- So you can't legitimately color the whole graph either.
- **Thus it's not bipartite.**



# What have we learned?

BFS can be used to detect bipartite-ness in time  $O(n + m)$ .



# Outline

- Part 0: Graphs and terminology
- Part 1: Depth-first search
  - Application: topological sorting
  - Application: in-order traversal of BSTs
- Part 2: Breadth-first search
  - Application: shortest paths
  - Application (if time): is a graph bipartite?



Recap

# Recap

- Depth-first search
  - Useful for topological sorting
  - Also in-order traversals of BSTs
- Breadth-first search
  - Useful for finding shortest paths
  - Also for testing bipartiteness
- Both DFS, BFS:
  - Useful for exploring graphs, finding connected components, etc

# Still open (next few classes)

- We can now find components in undirected graphs...
  - What if we want to find strongly connected components in directed graphs?
- How can we find shortest paths in weighted graphs?
- What is Samuel L. Jackson's Erdos number?
  - (Or, what if I want everyone's everyone-else number?)

# Next Time

- Strongly Connected Components

## Before Next Time

- Pre-lecture exercise: Strongly Connected What-Now?
- Study for exam!