BLG 336E

Analysis of Algorithms II

Lecture 4:

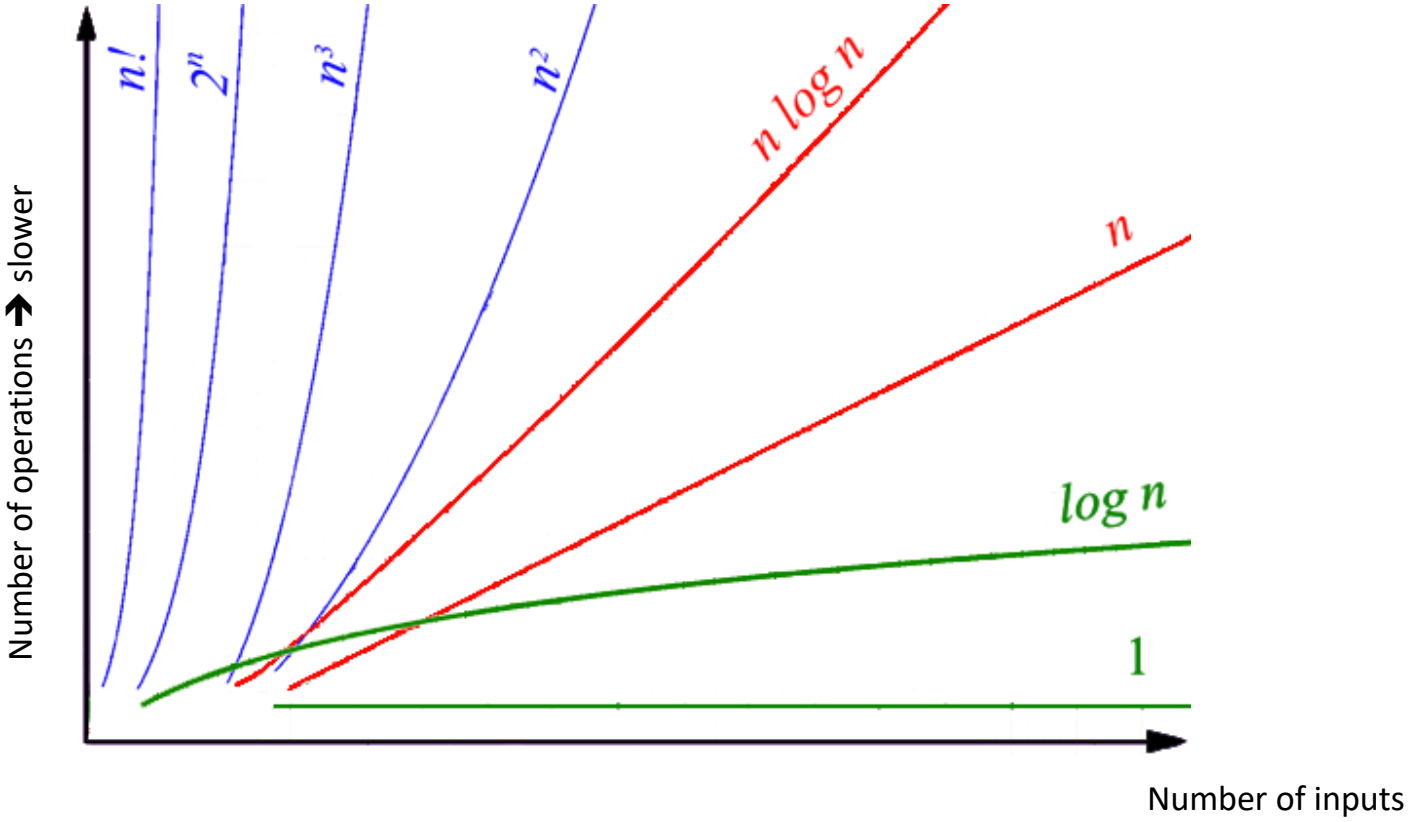
Graph Basics, Breadth First Search, Depth First Search

Time Versus **Space** Complexities

Two main characteristics for programs

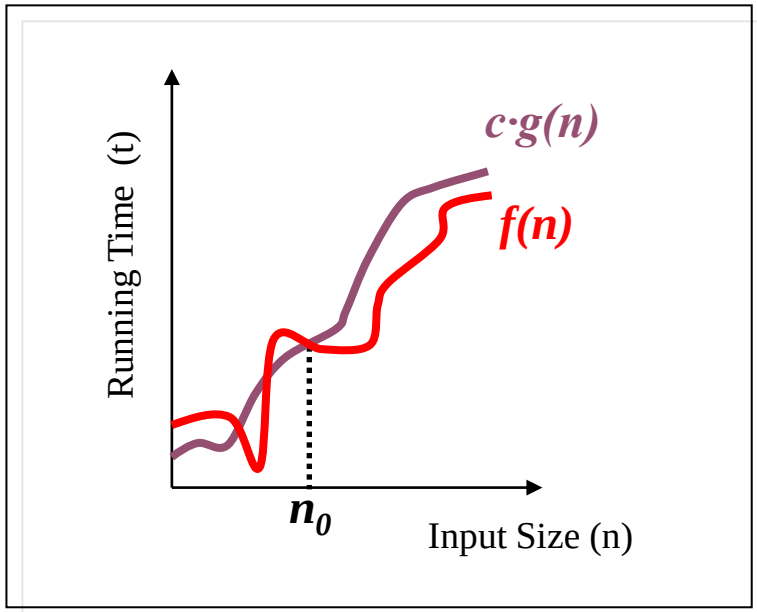
- **Time** complexity: ≈ **CPU** usage
- **Space** complexity: ≈ **RAM** usage

NB: if **time** complexity is "**high**" your algorithm may run for too long; if **space** complexity is **high**, your stack may be over flown, and you may not be able to run the algorithm at all!

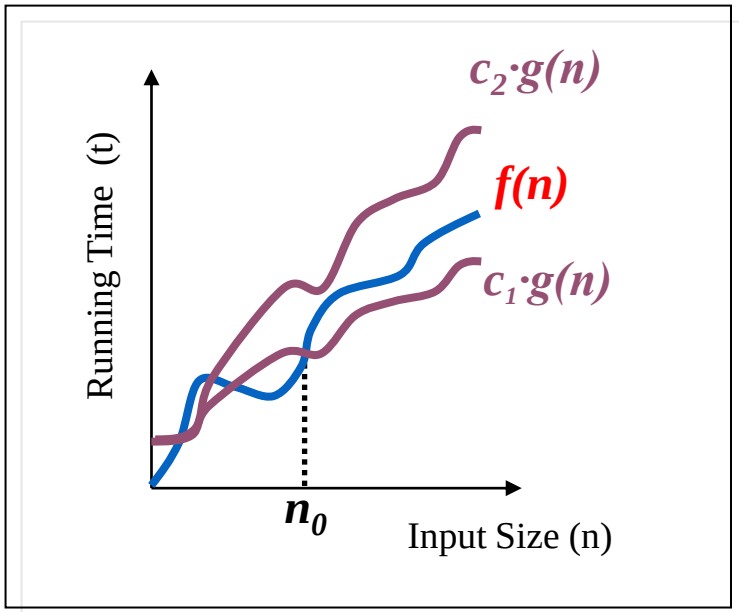


$$n! \gg 2^n \gg n^3 \gg n^2$$

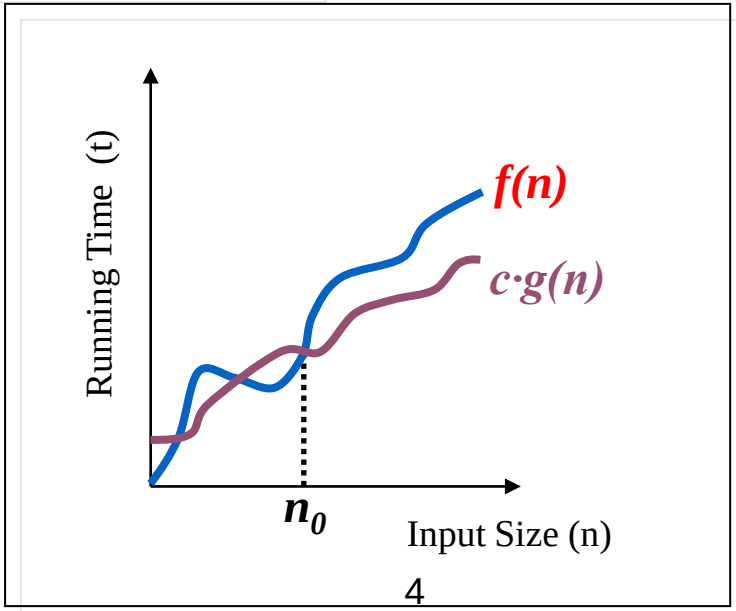
Asymptotic Notation



Big O



Big Theta



Big Omega

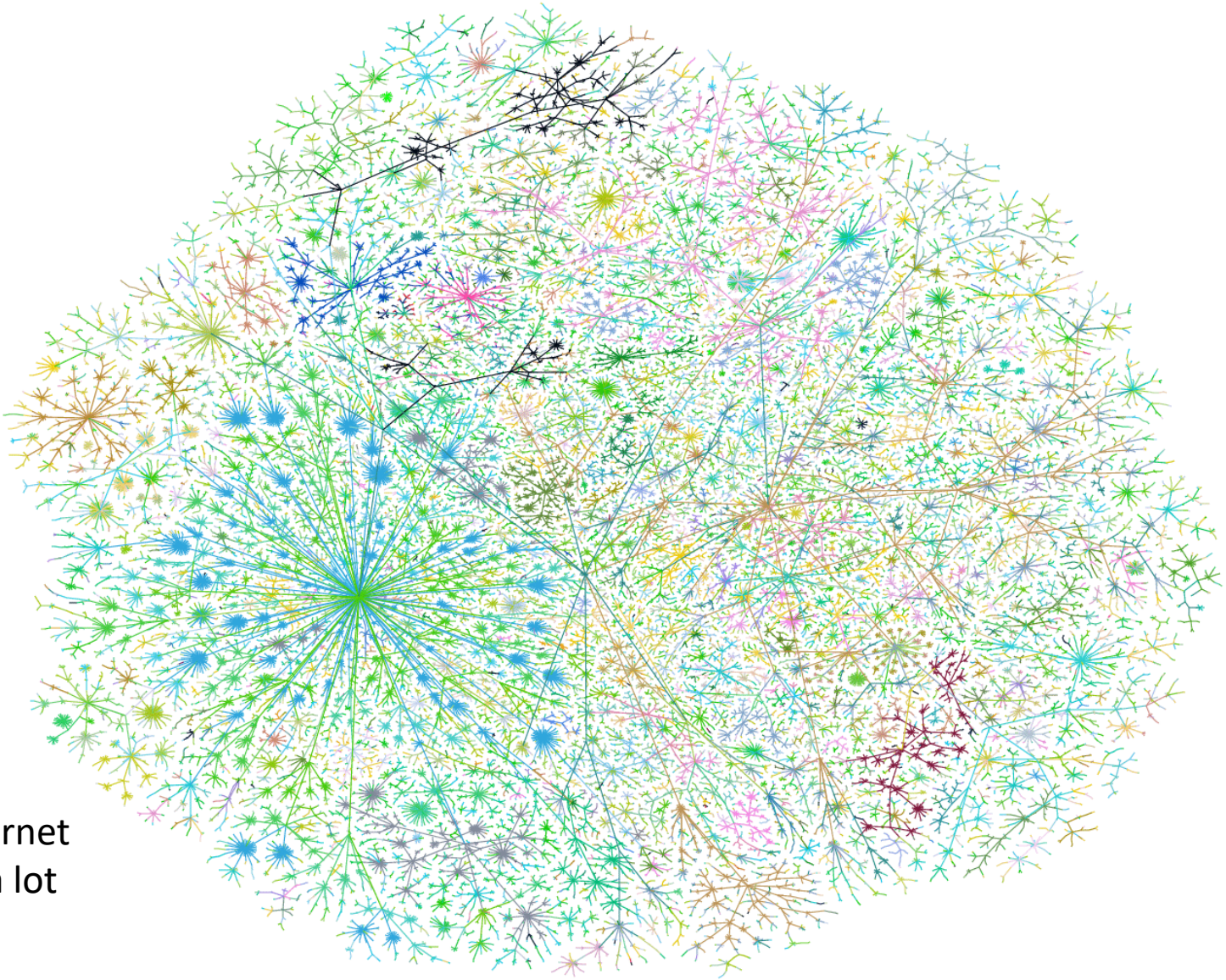
Outline

- 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?

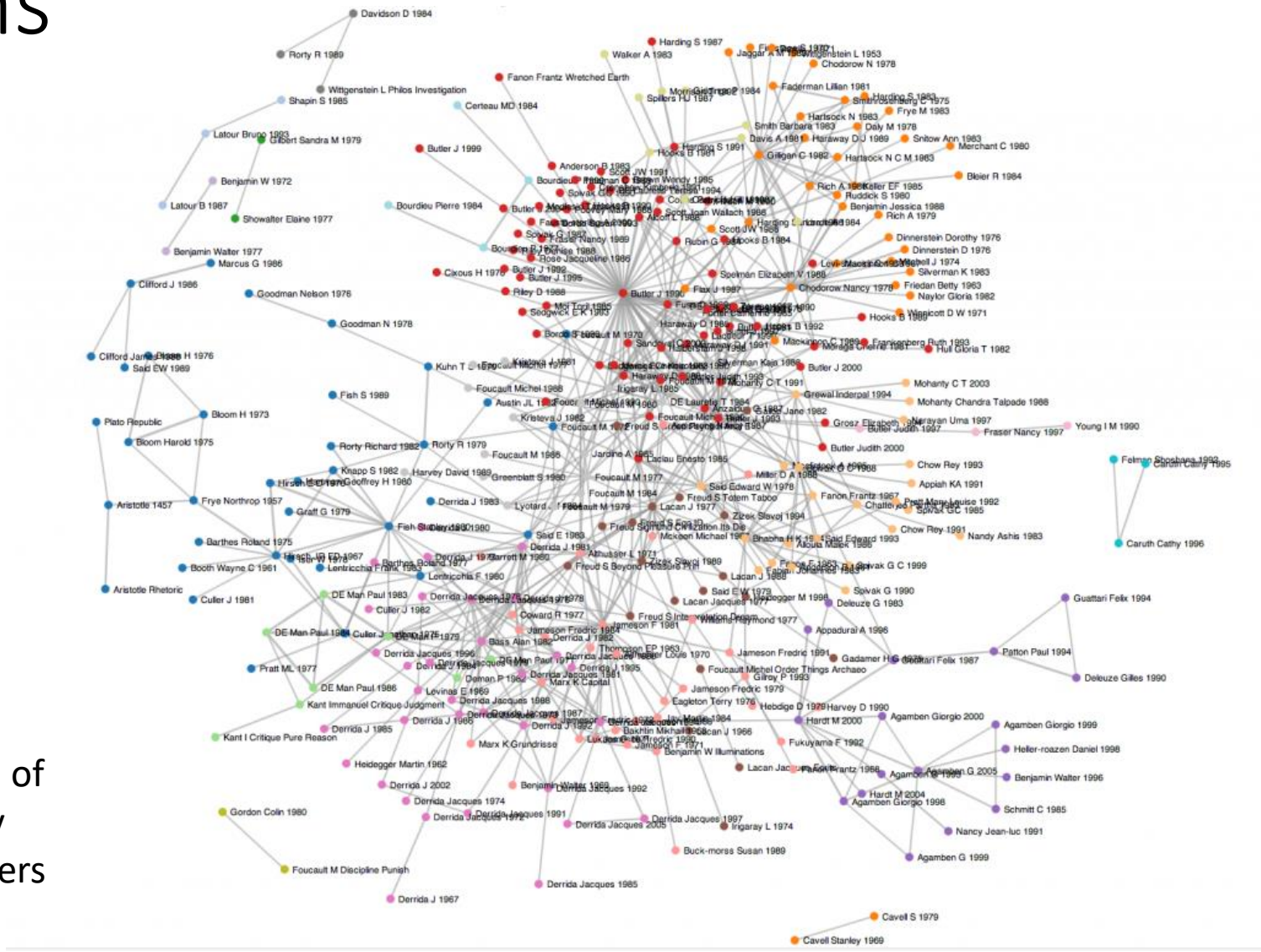
Part 0: Graphs

Graphs



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

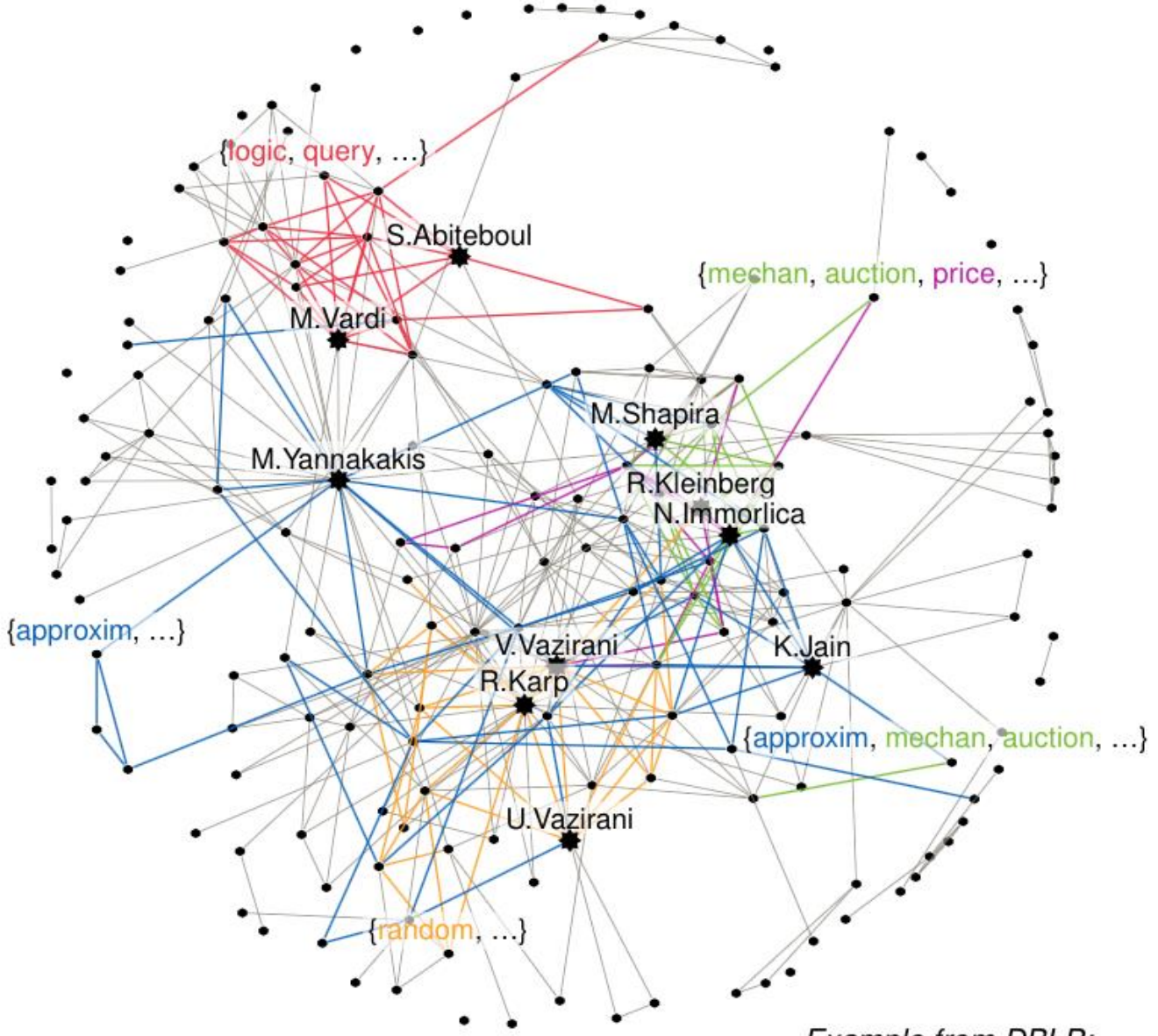
Graphs



Citation graph of literary theory academic papers

Graphs

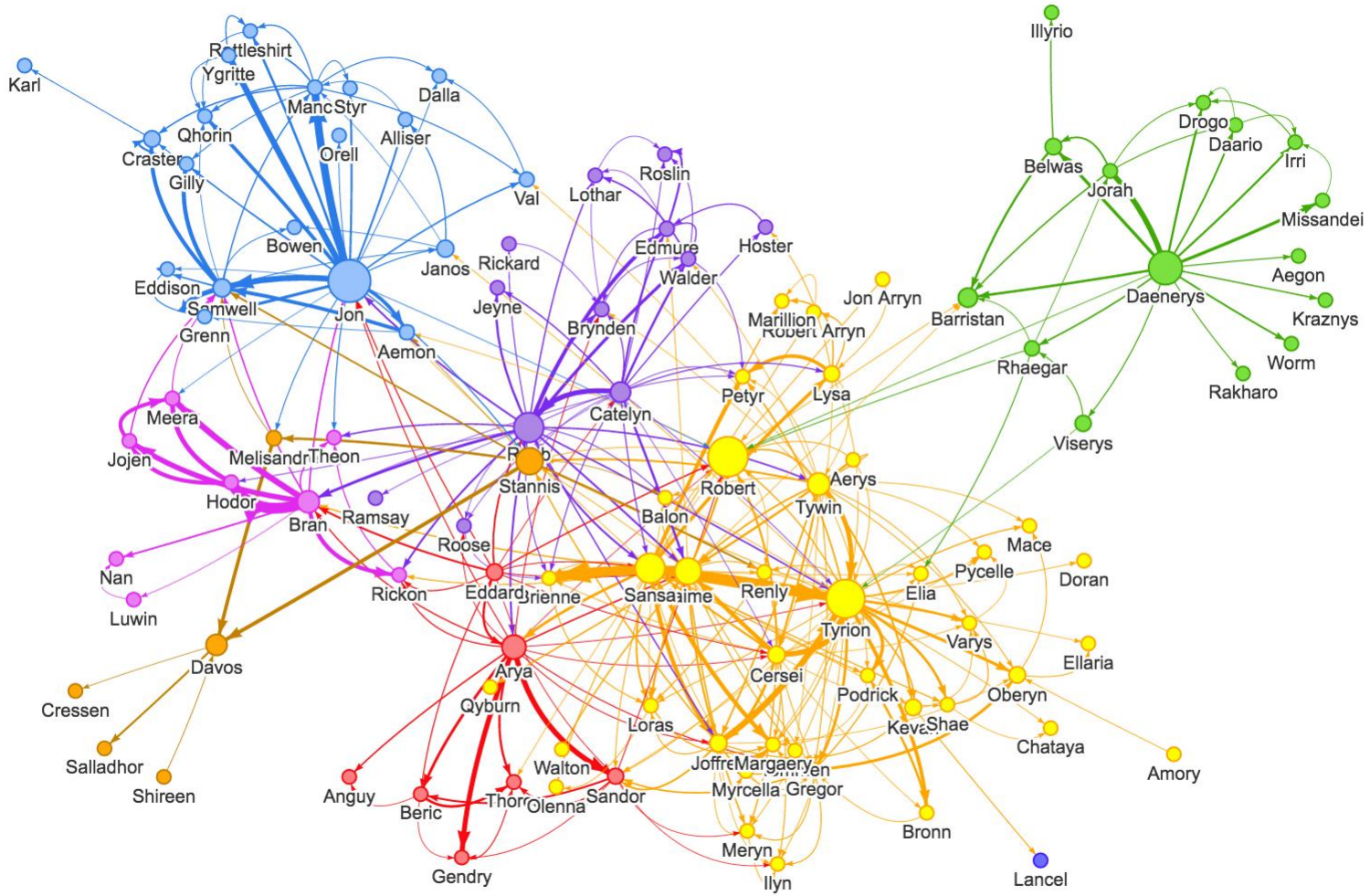
Theoretical Computer Science academic communities



Example from DBLP:
Communities within the co-authors of Christos H. Papadimitriou

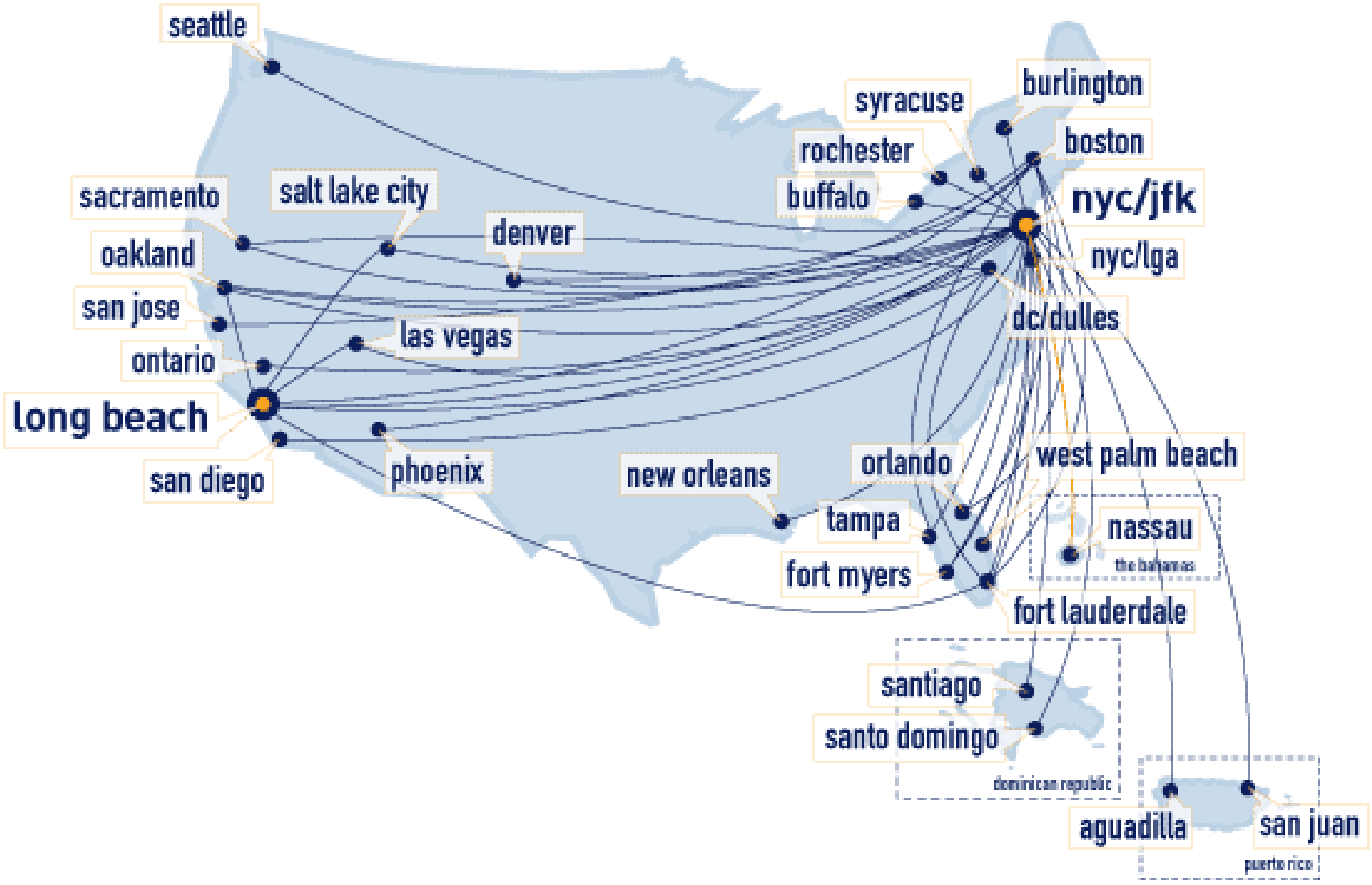
Graphs

Game of Thrones Character Interaction Network



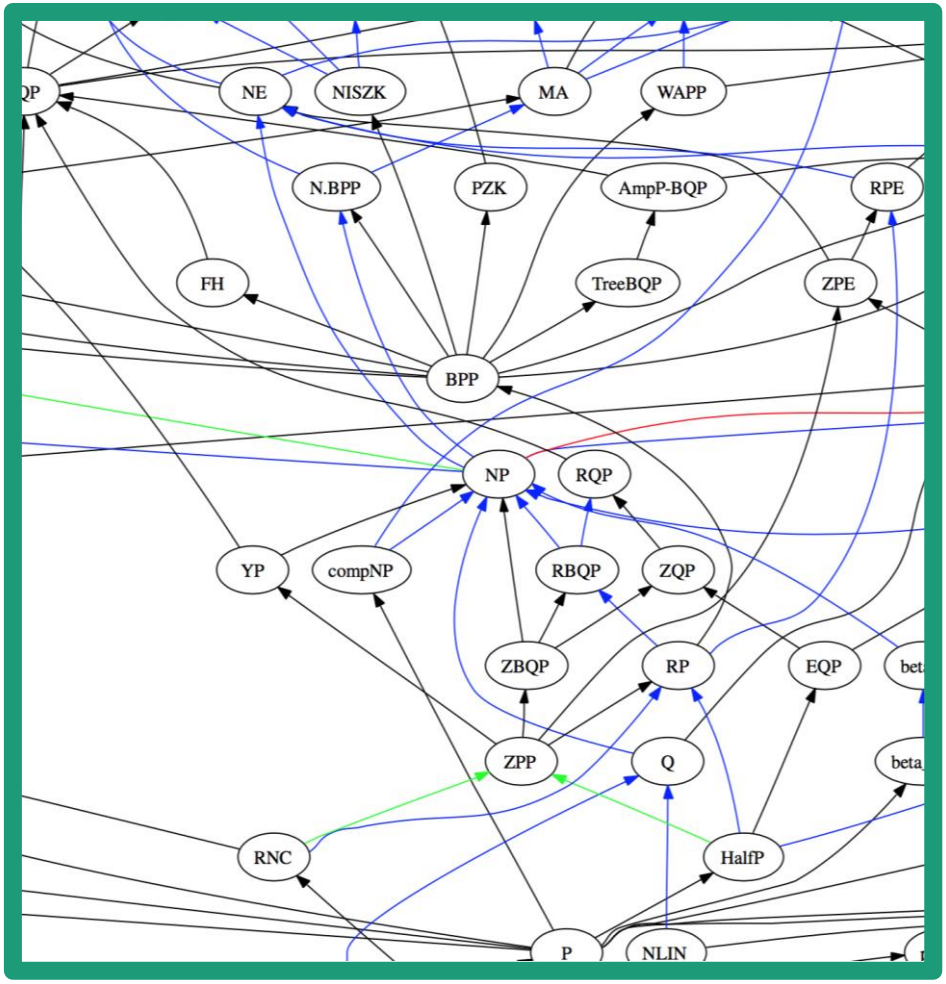
Graphs

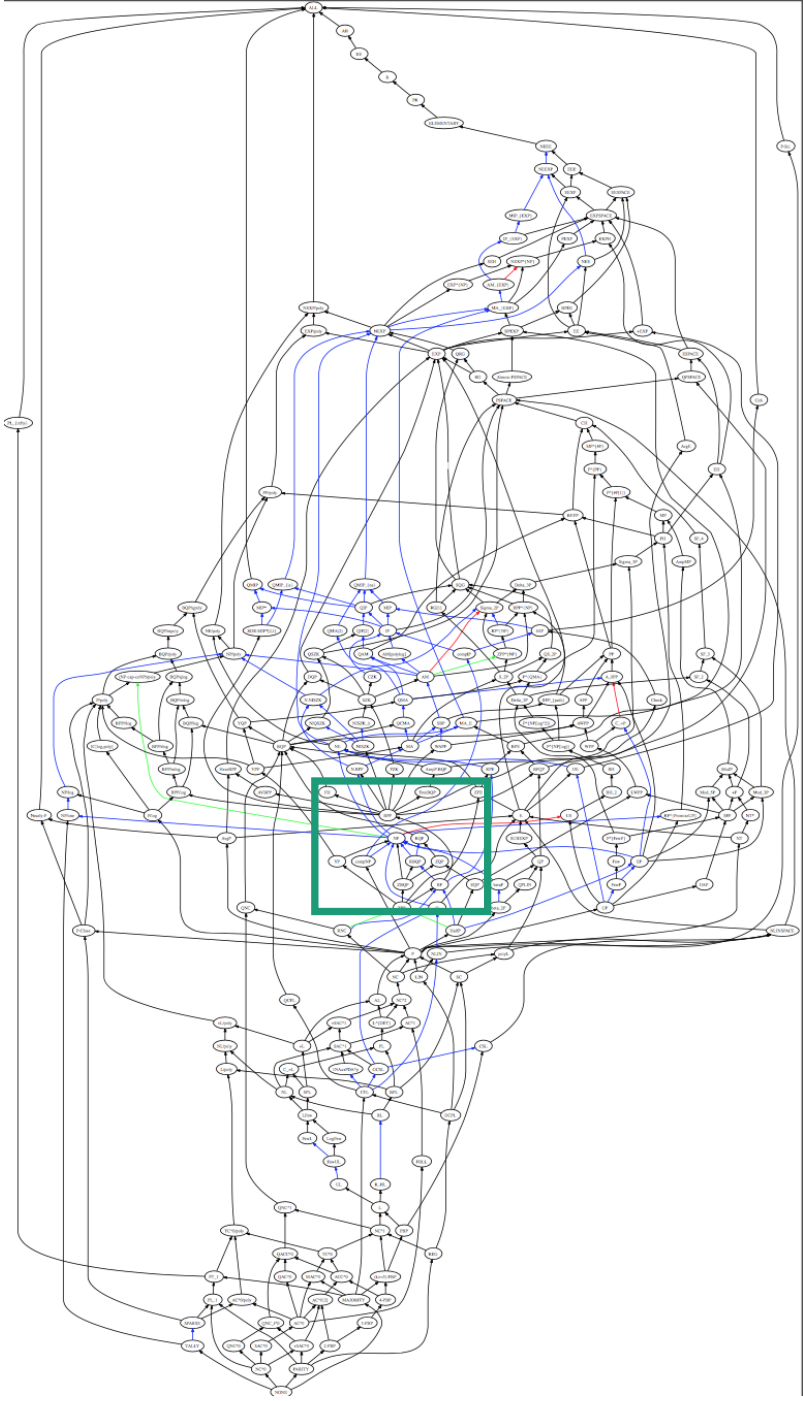
jetblue flights



Graphs

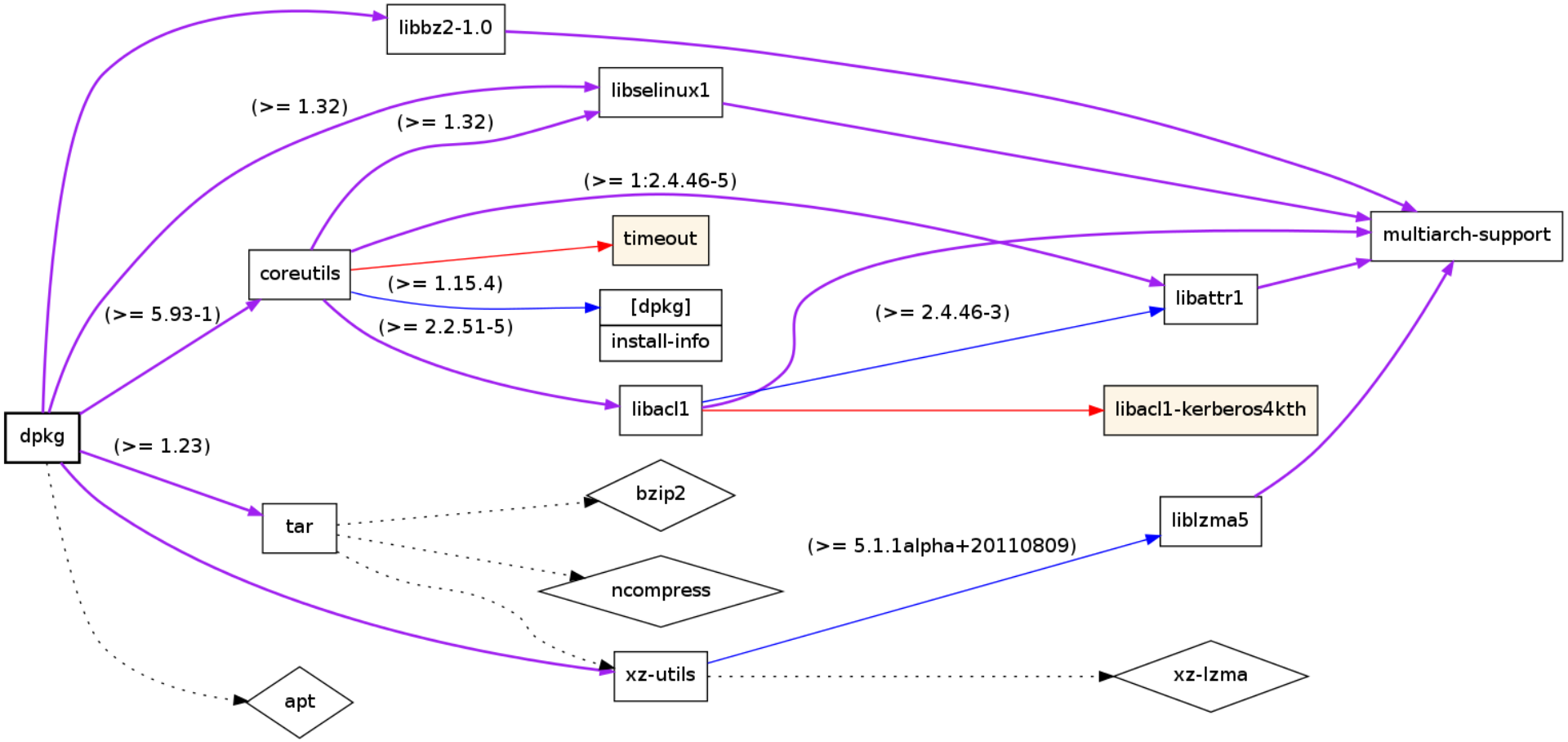
Complexity Zoo
containment graph





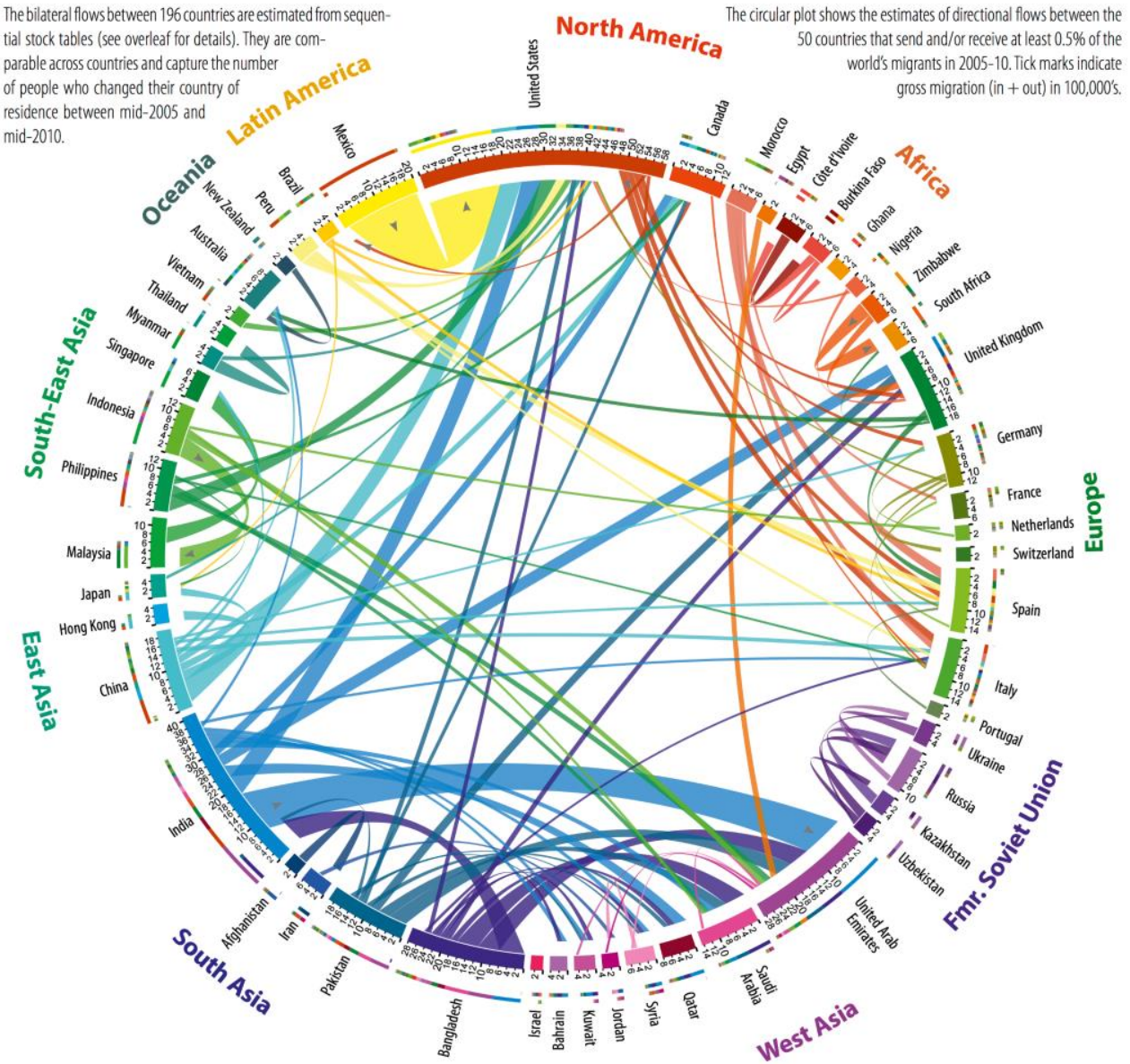
Graphs

debian dependency (sub)graph



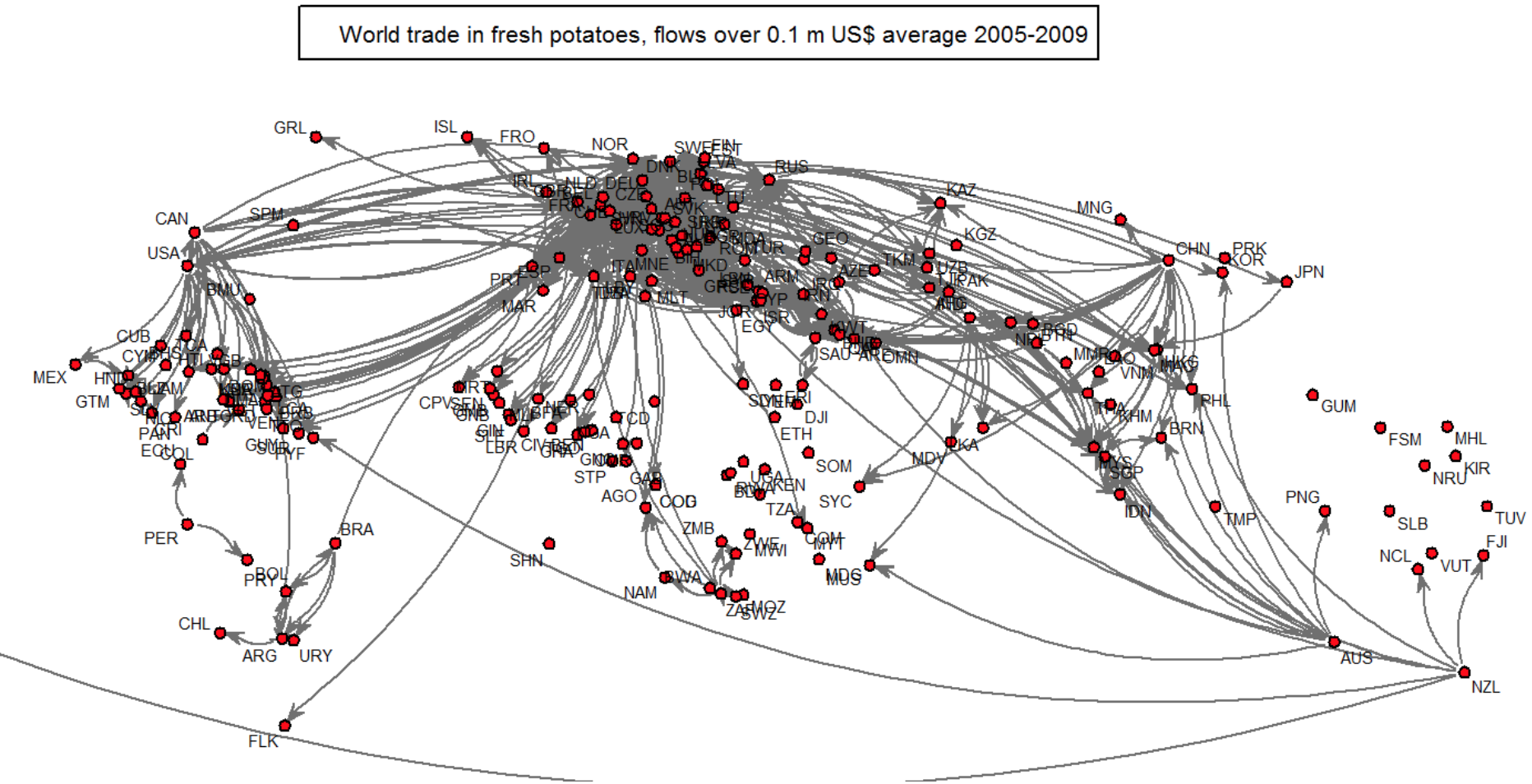
Graphs

Immigration flows



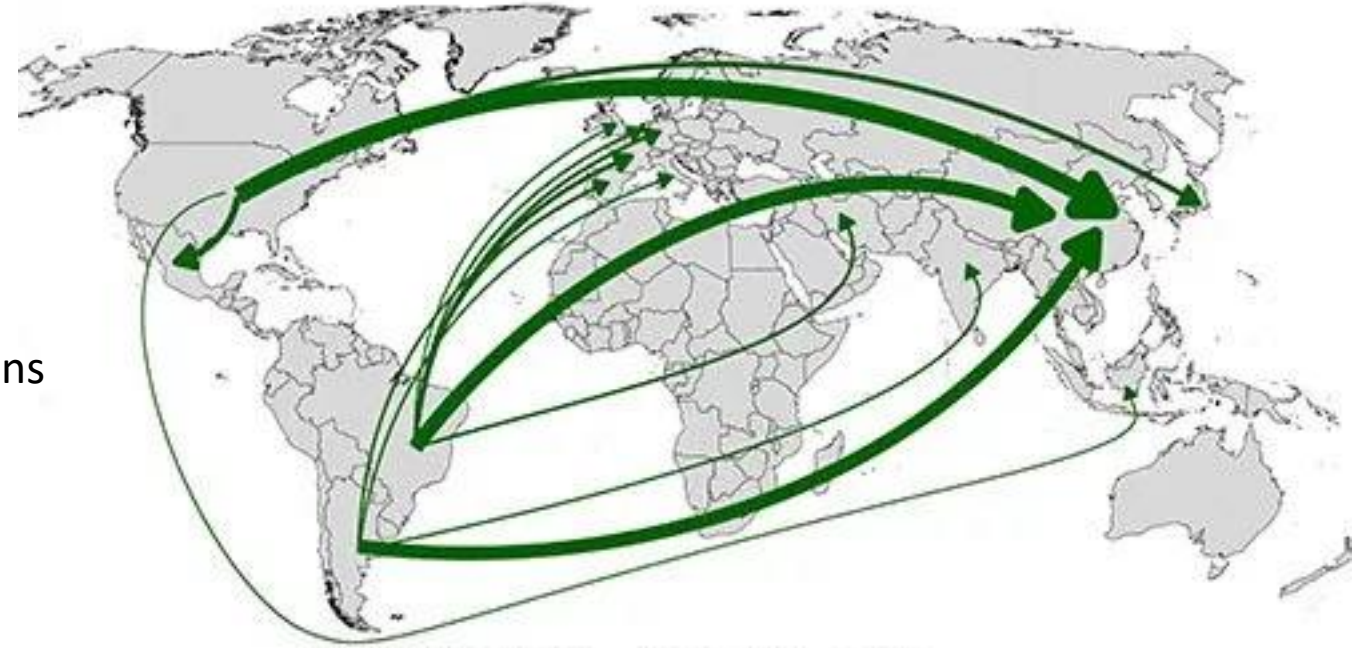
Graphs

Potato trade

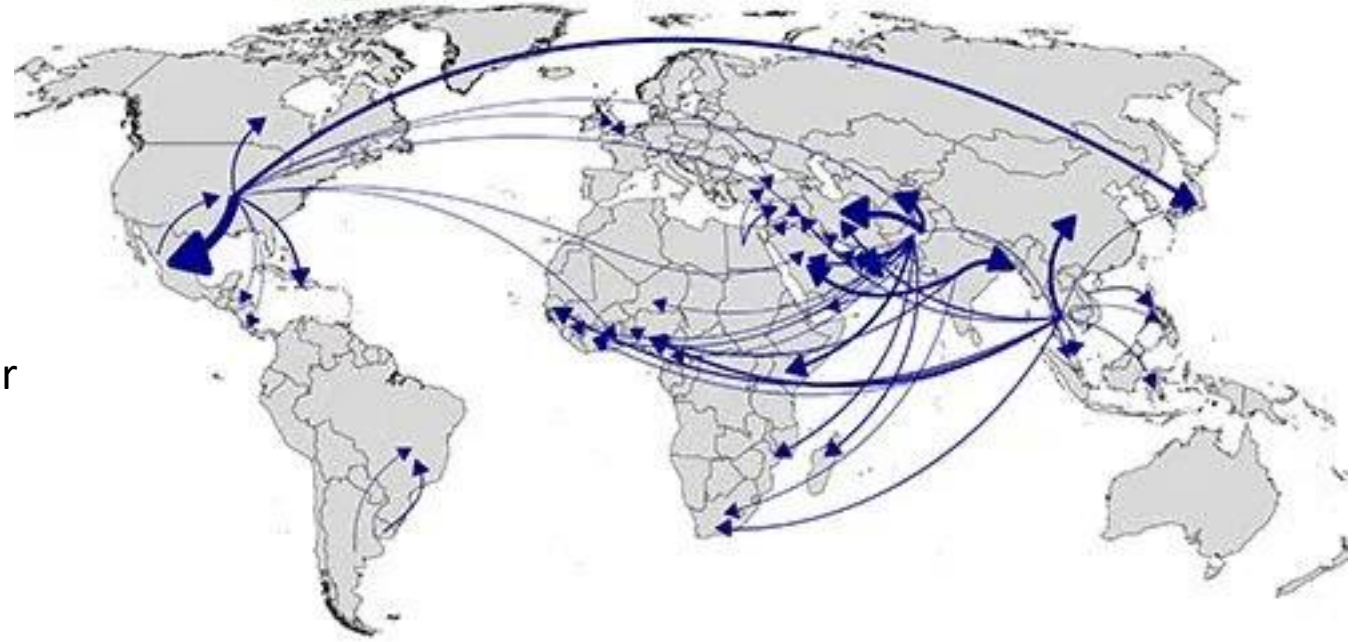


Graphs

Soybeans

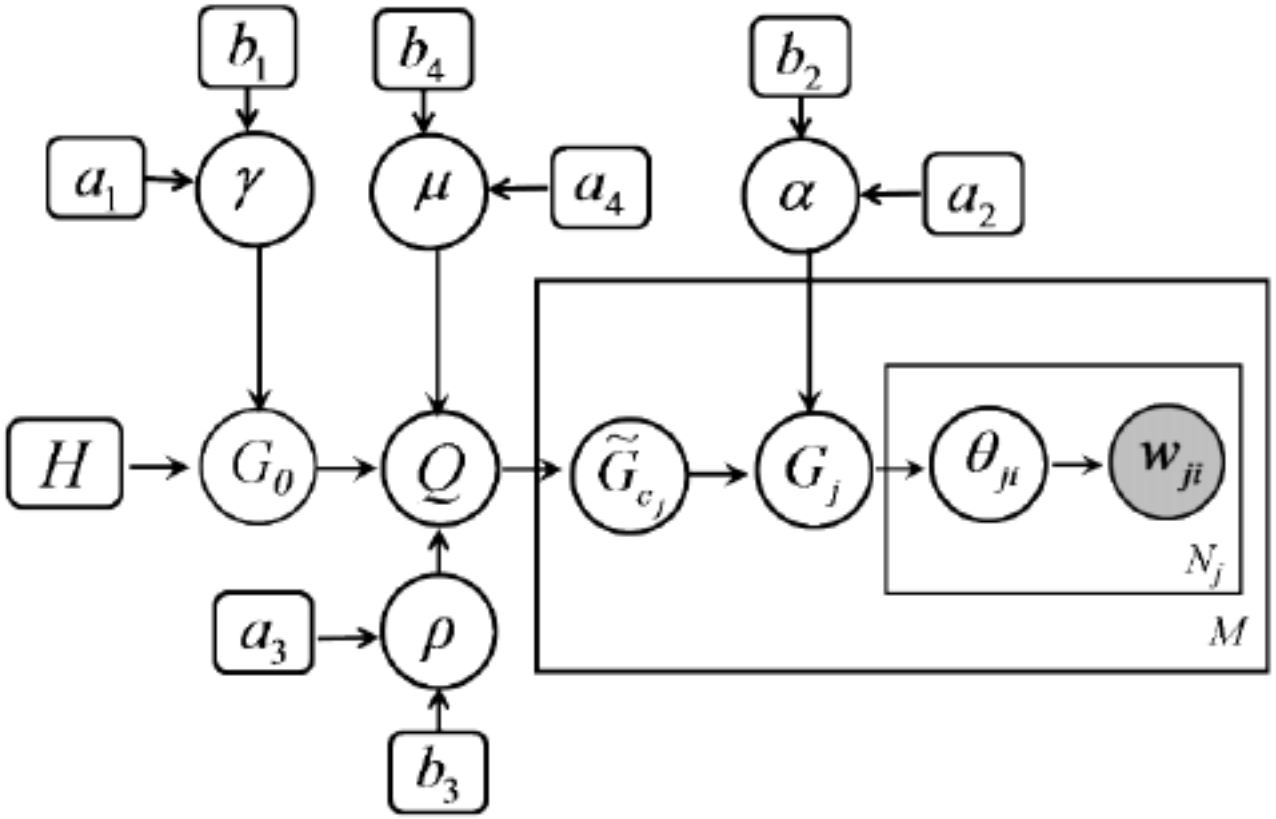


Water



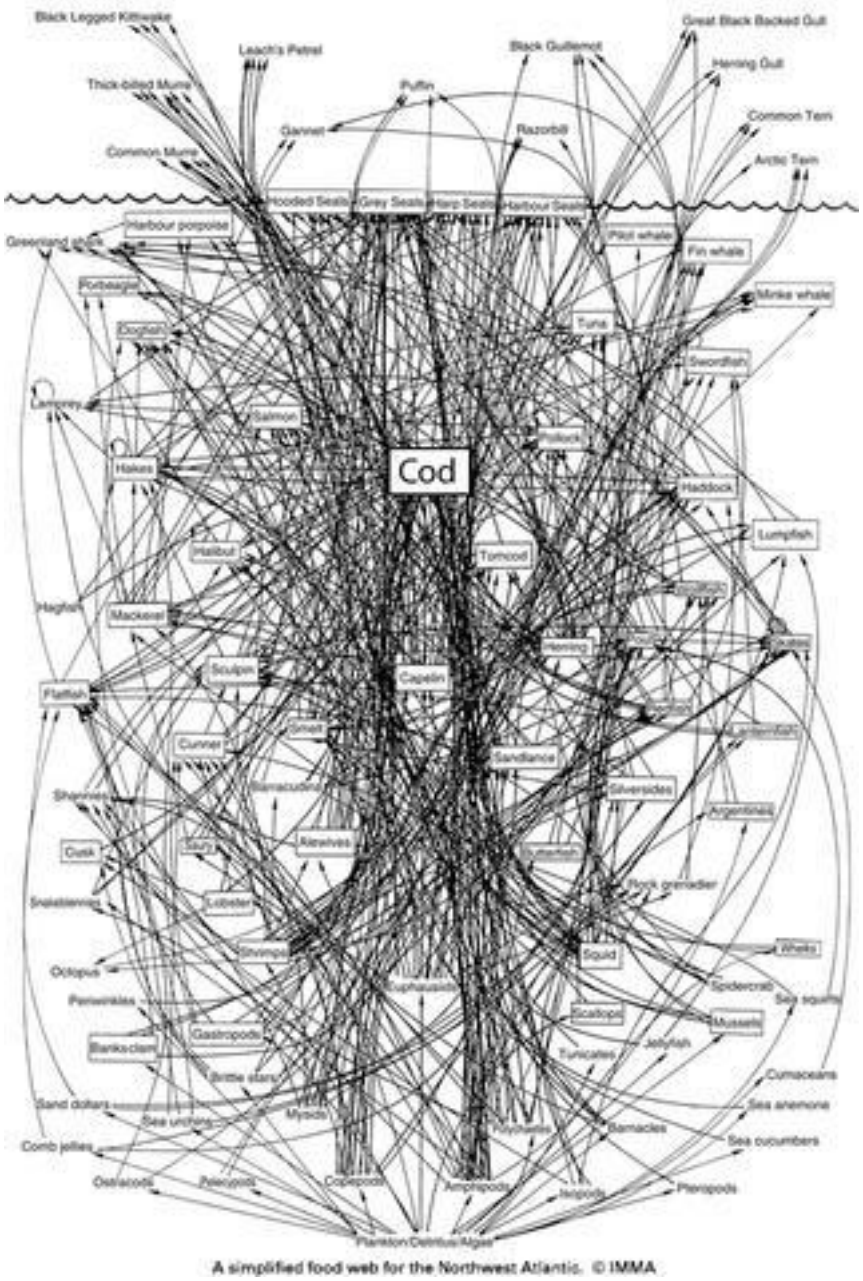
Graphs

Graphical models



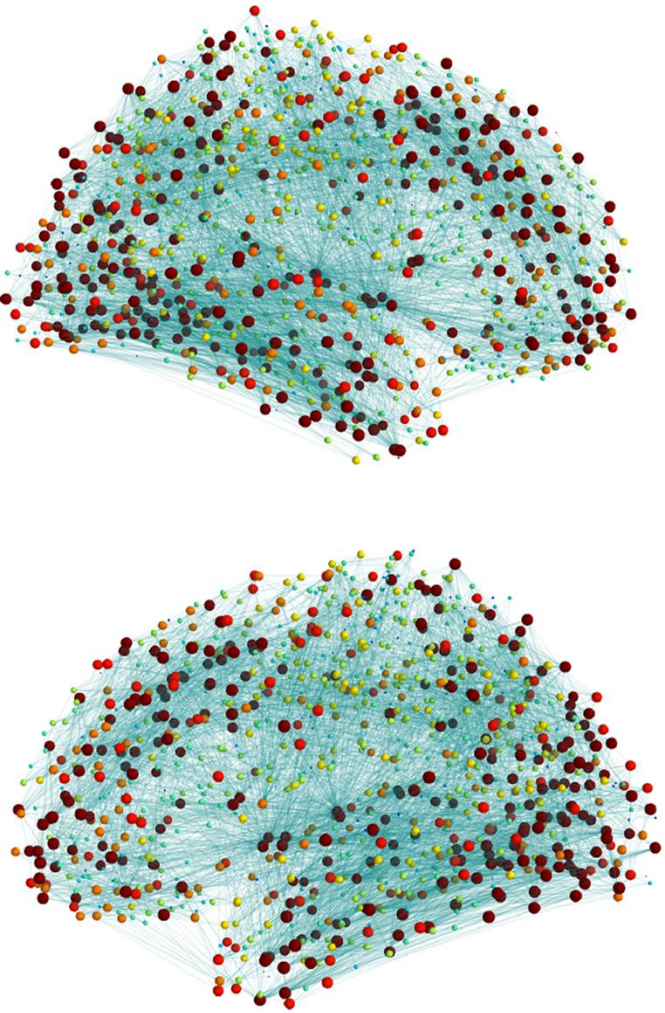
Graphs

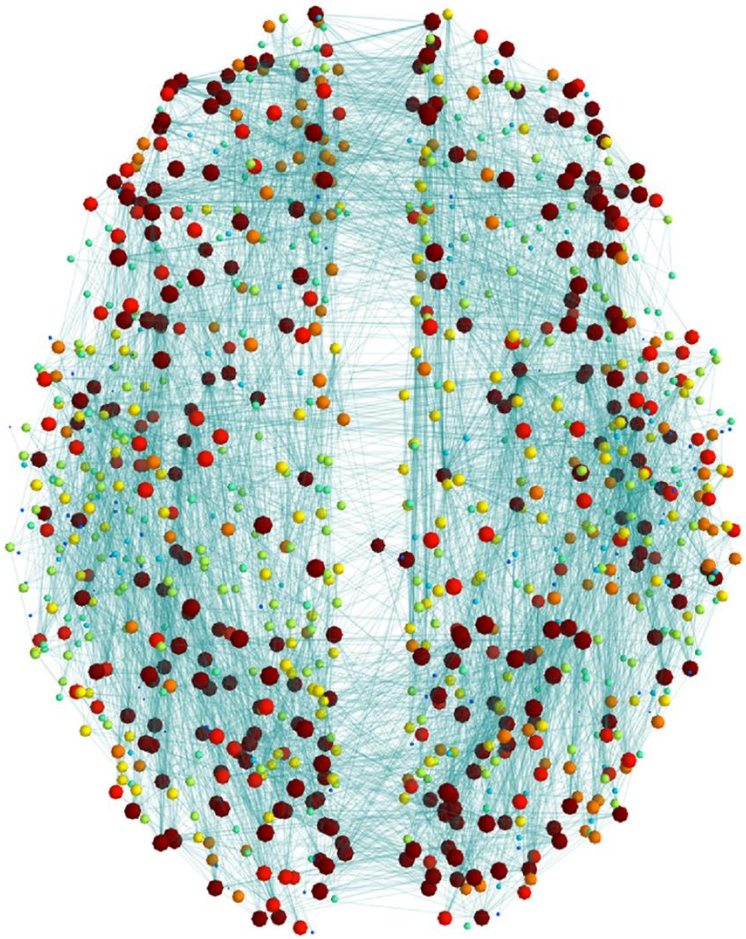
What eats what in the Atlantic ocean?

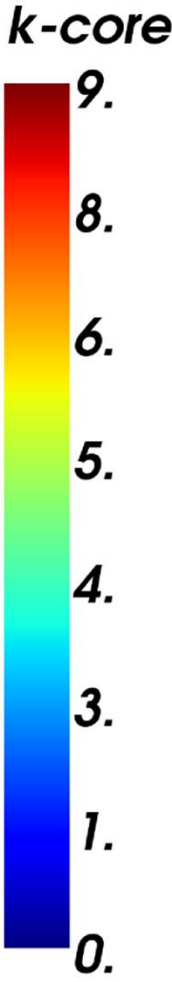


Graphs

Neural connections
in the brain







Graphs

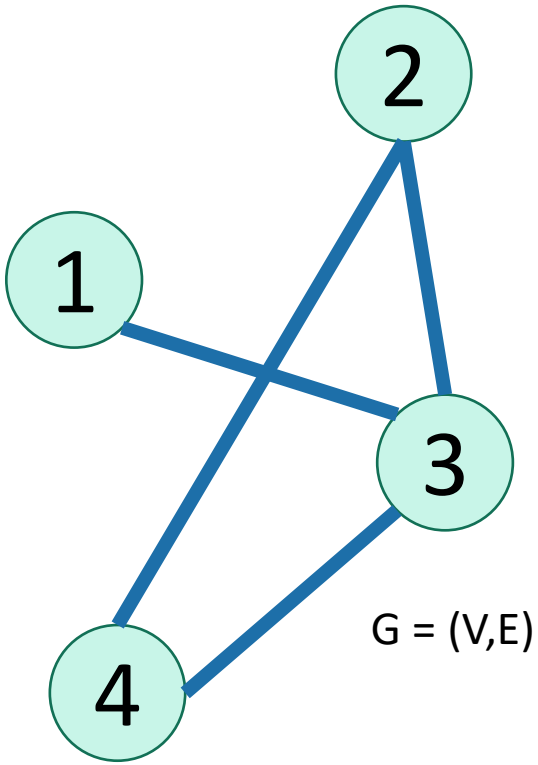
- **There are a lot of graphs.**

- We want to answer questions about them.
  - Efficient routing?
  - Community detection/clustering?
    - 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)

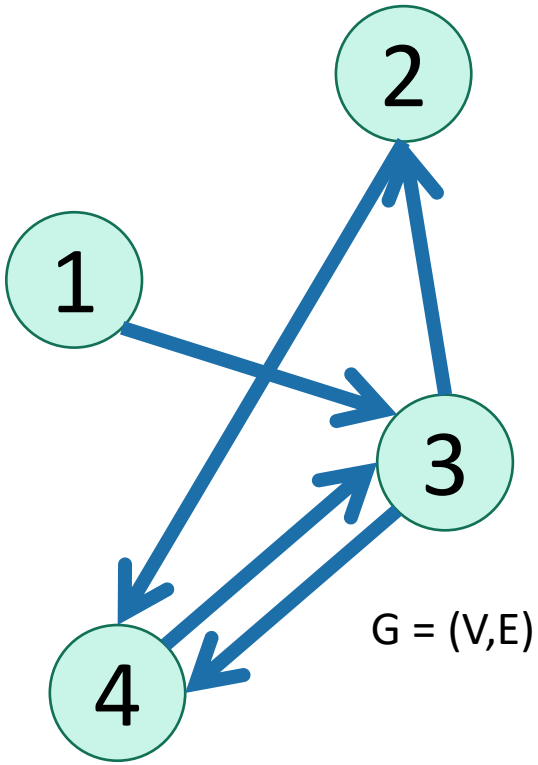


- Example
  - V = {1,2,3,4}
  - E = { {1,3}, {2,4}, {3,4}, {2,3} }

- 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)



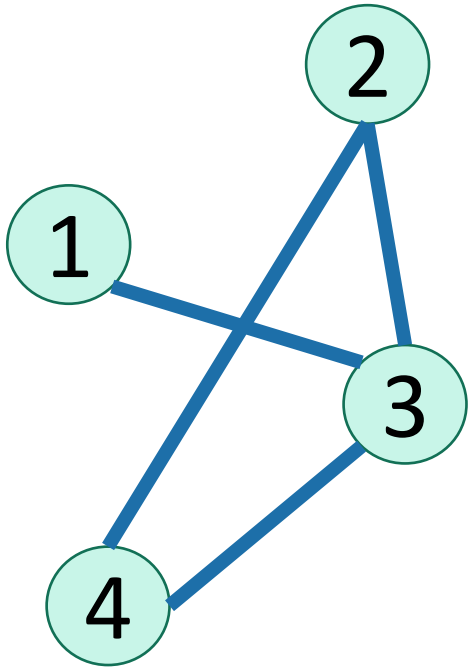
- Example
  - V = {1,2,3,4}
  - E = { (1,3), (2,4), (3,4), (4,3), (3,2) }

- 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

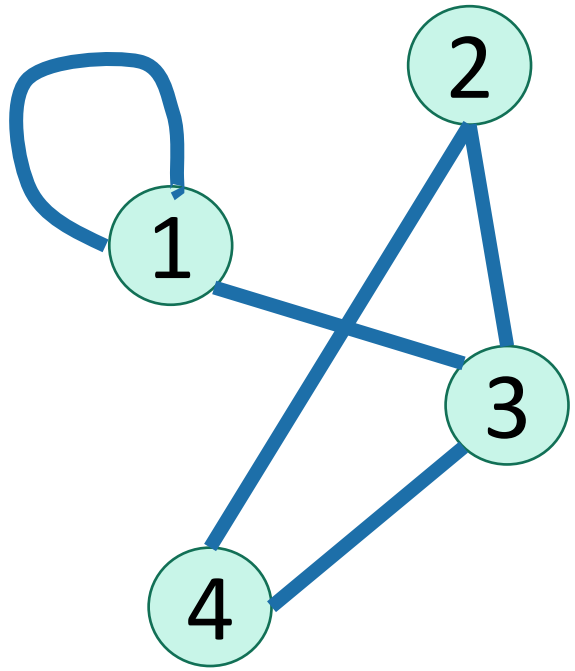
	1	2	3	4
1	0	0	1	0
2	0	0	1	1
3	1	1	0	1
4	0	1	1	0



How do we represent graphs?

- Option 1: adjacency matrix

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

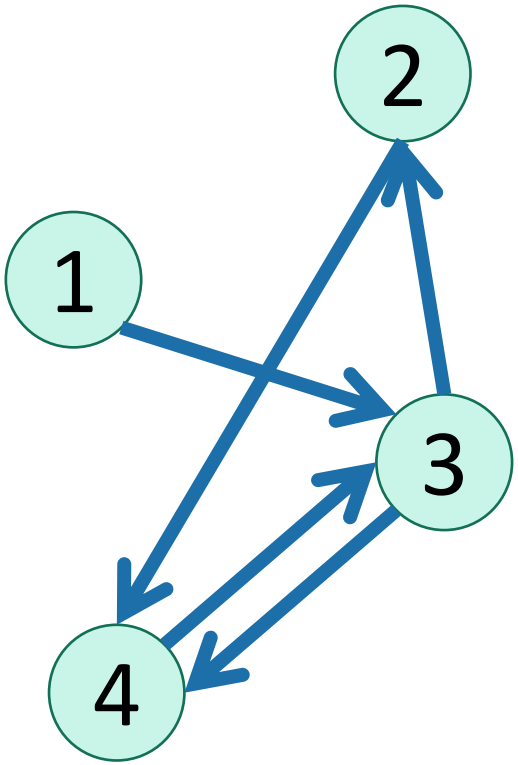


How do we represent graphs?

- Option 1: adjacency matrix

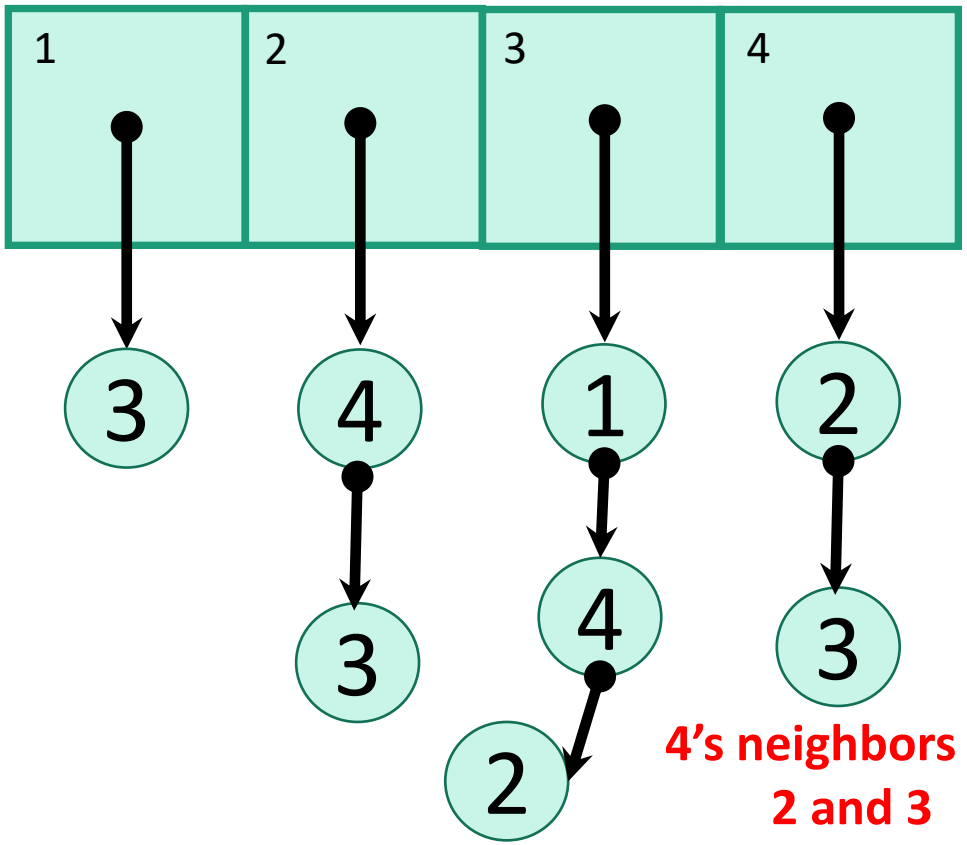
Destination

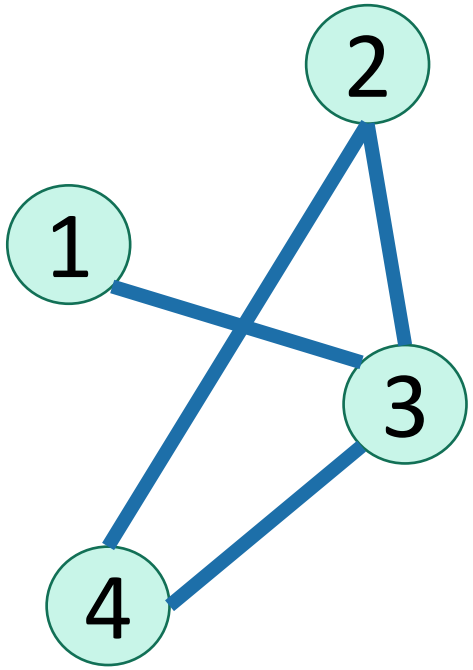
Source	1	2	3	4
1	0	0	1	0
2	0	0	0	1
3	0	1	0	1
4	0	0	1	0



How do we represent graphs?

- Option 2: linked lists.





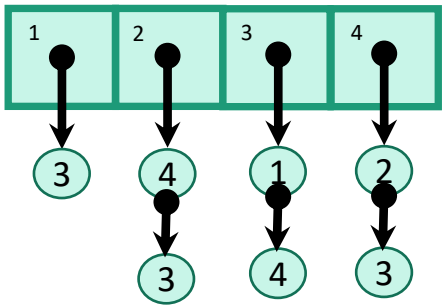
In either case

- 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

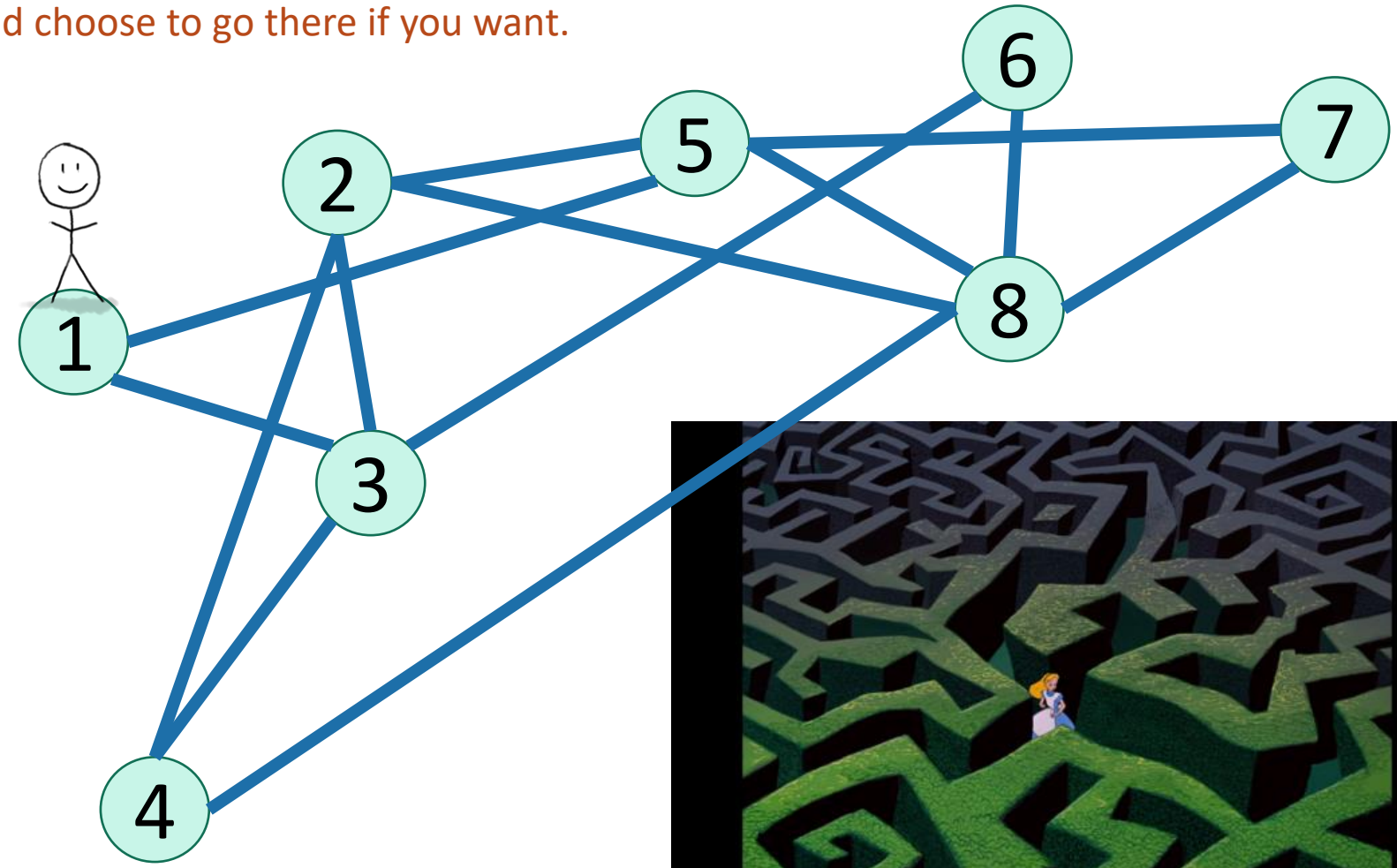
Say there are n vertices and m edges.

	$\begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$	Generally better for **sparse** graphs 
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) **We'll assume this representation for the rest of the class**

Part 1: Depth-first search

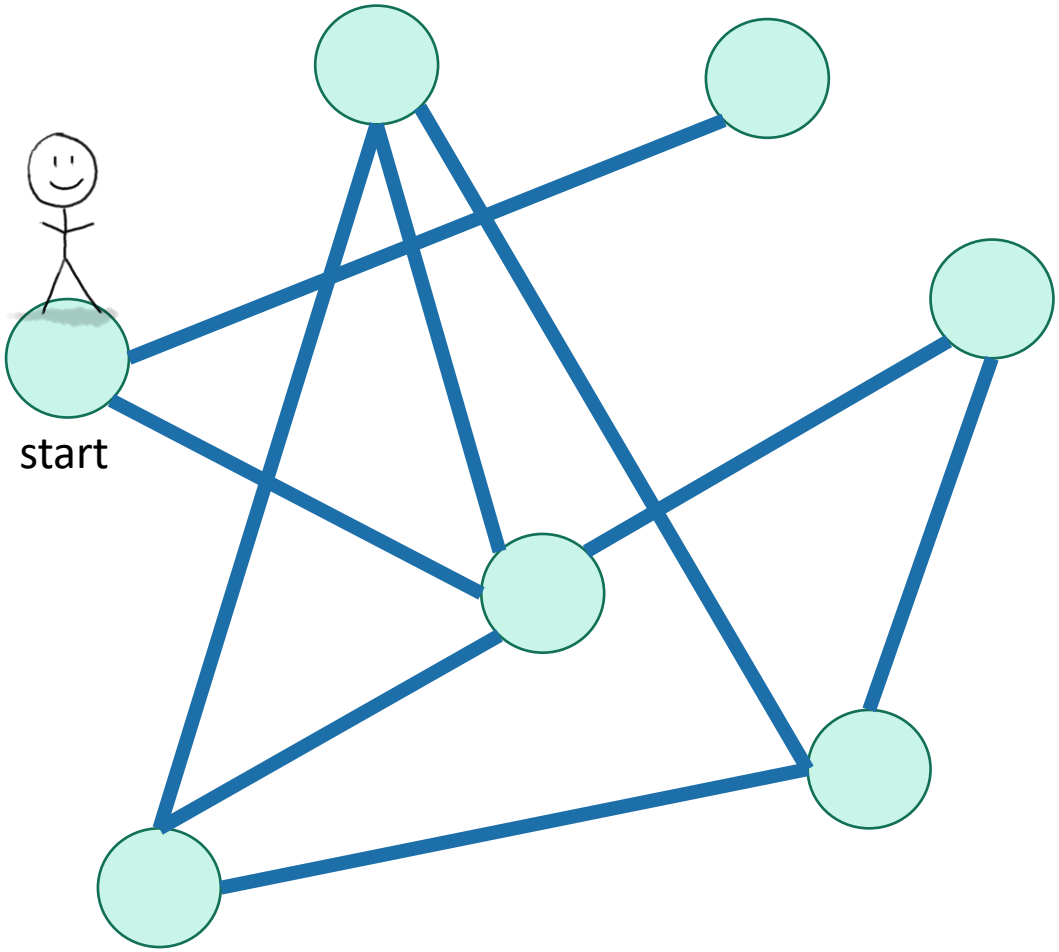
How do we explore a graph?

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



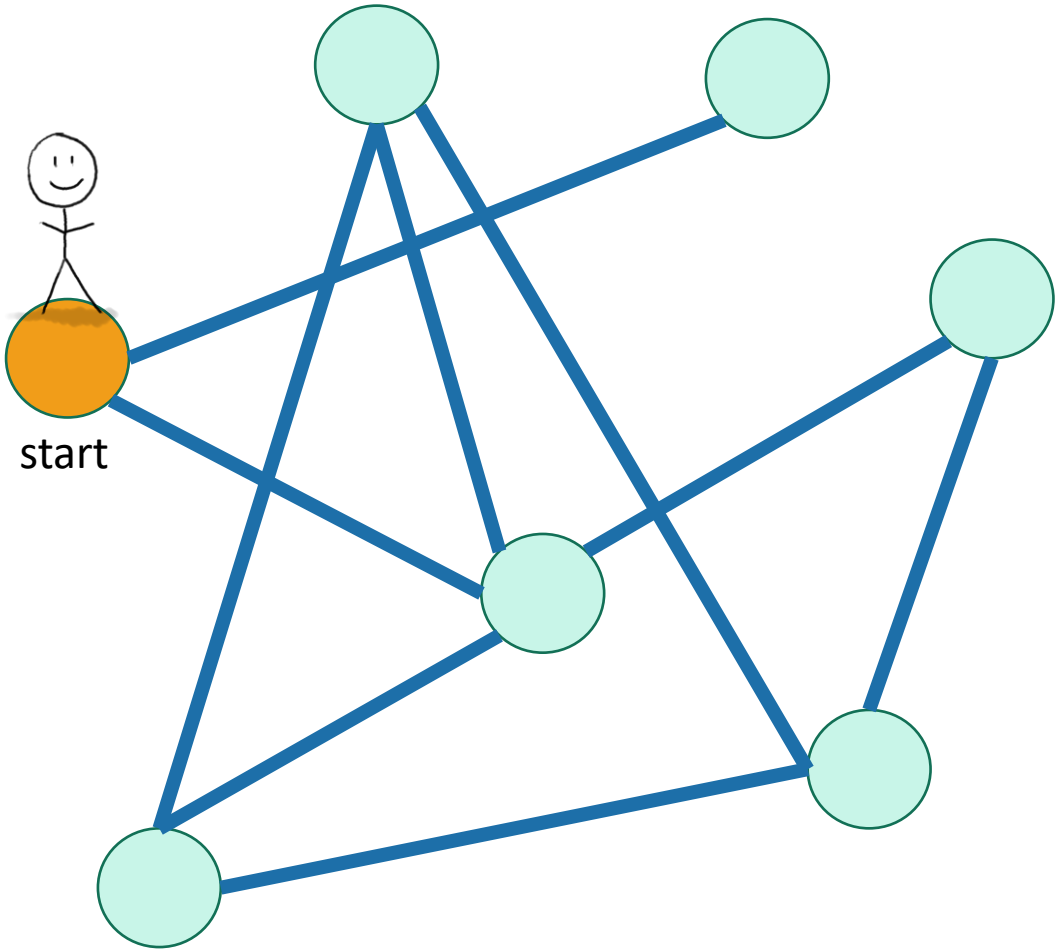
Not been there yet

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



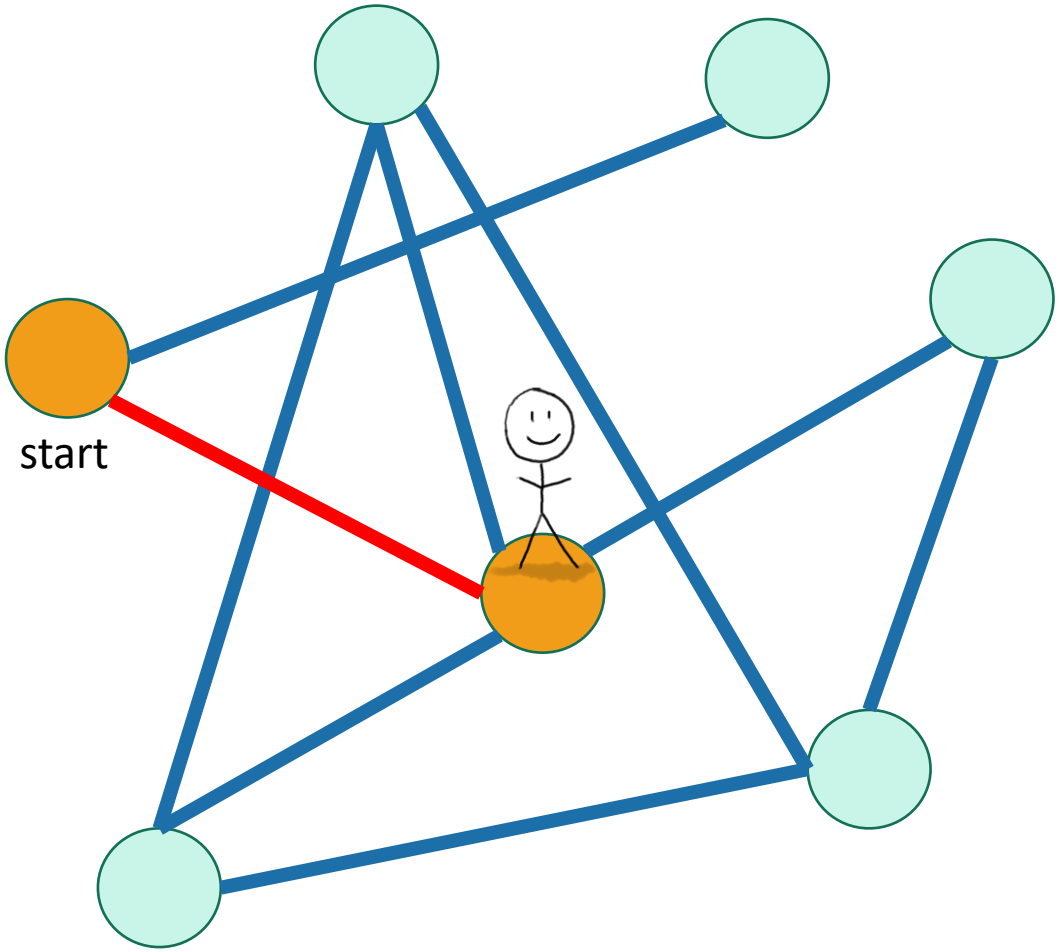
Not been there yet

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



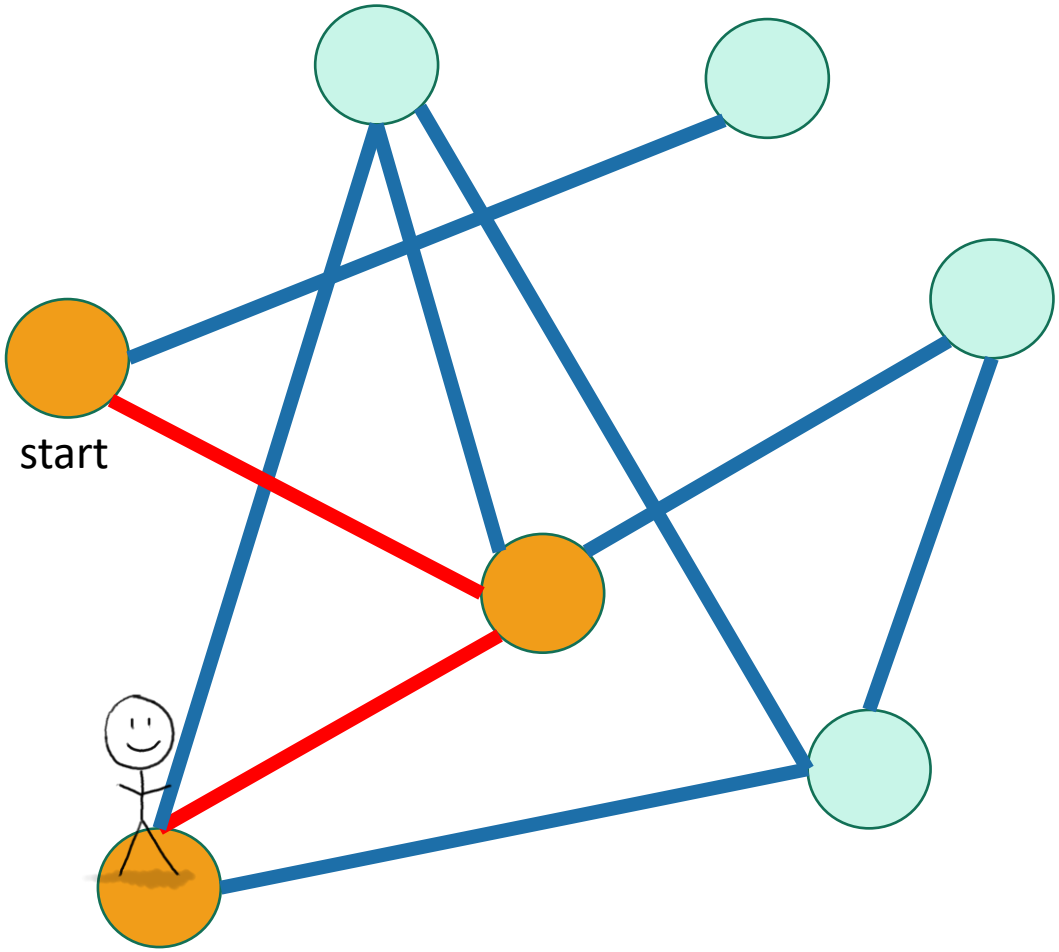
Not been there yet

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



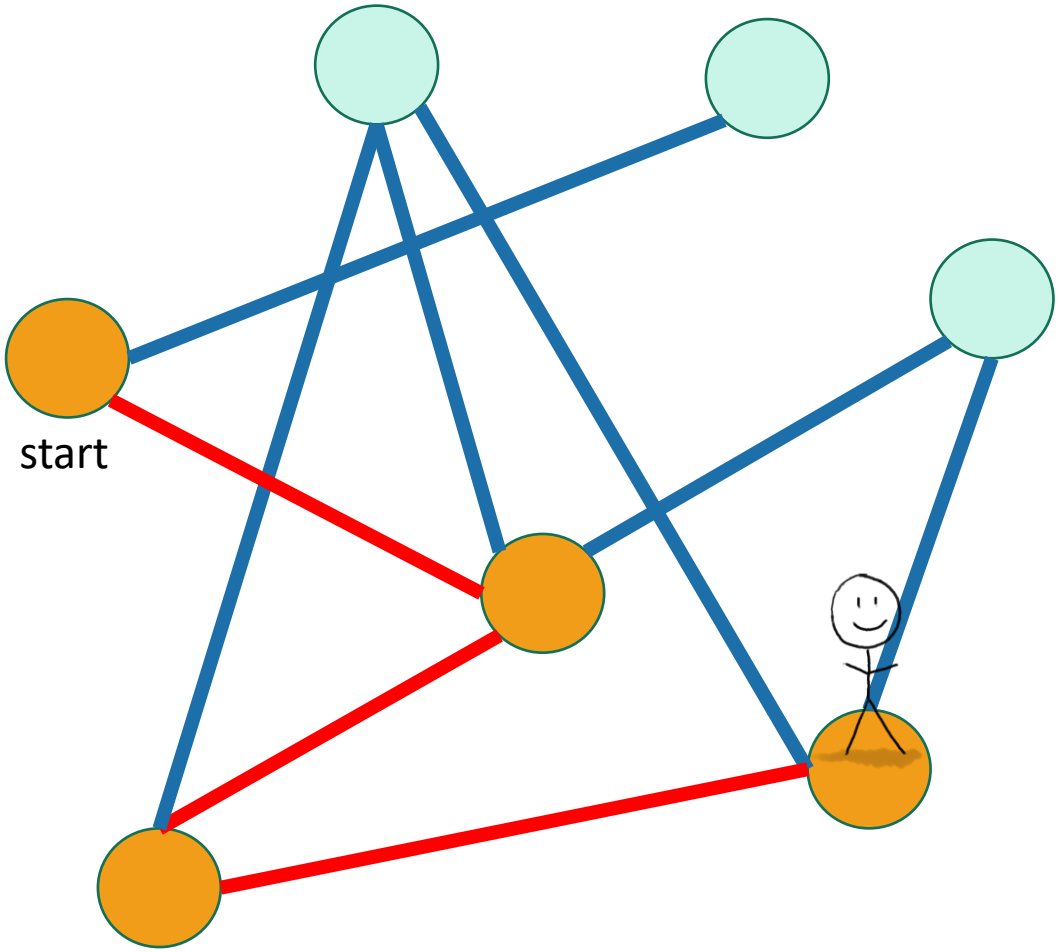
Not been there yet

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



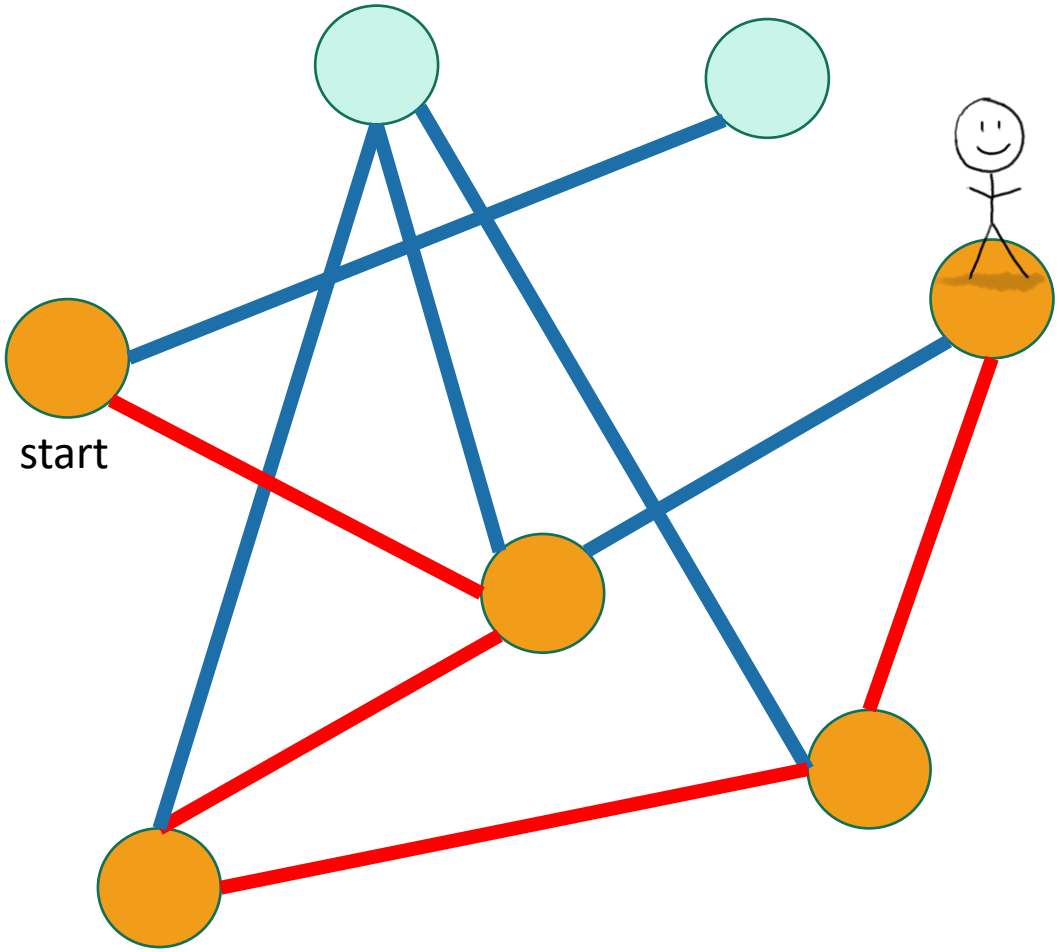
Not been there yet

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



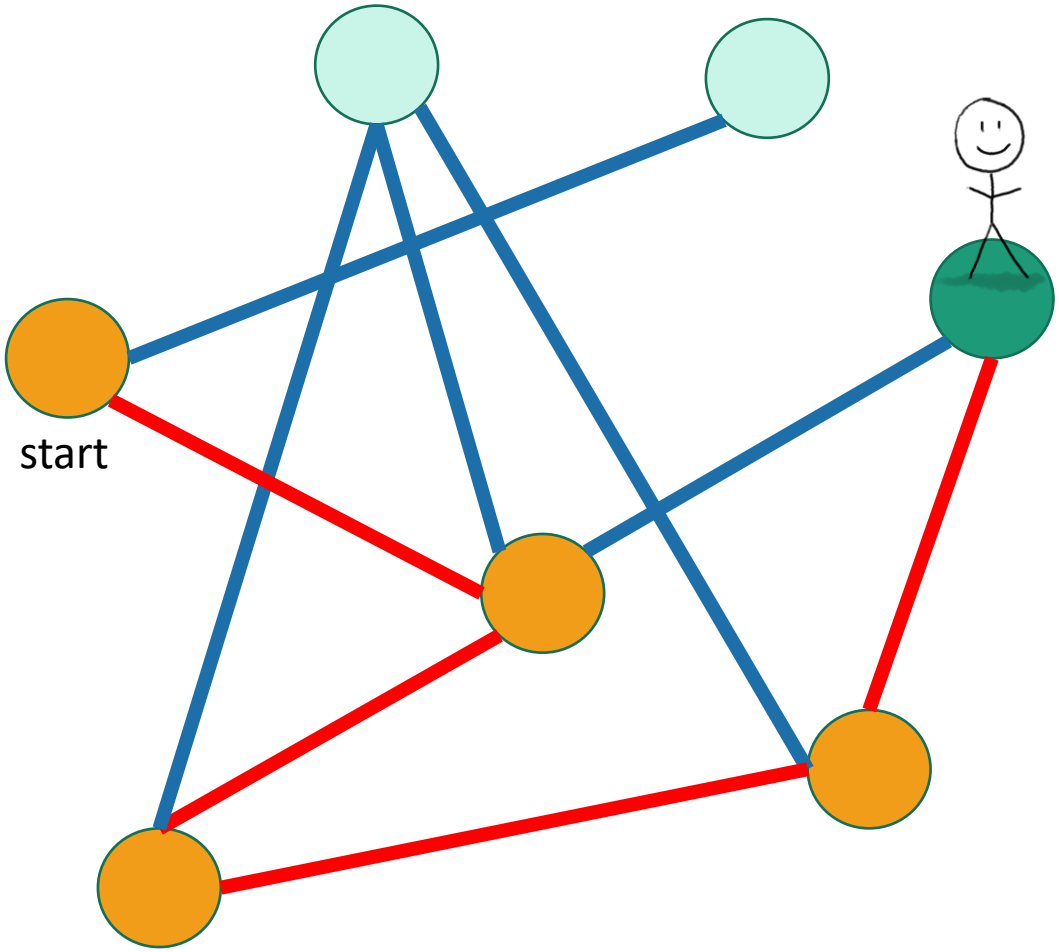
Not been there yet

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



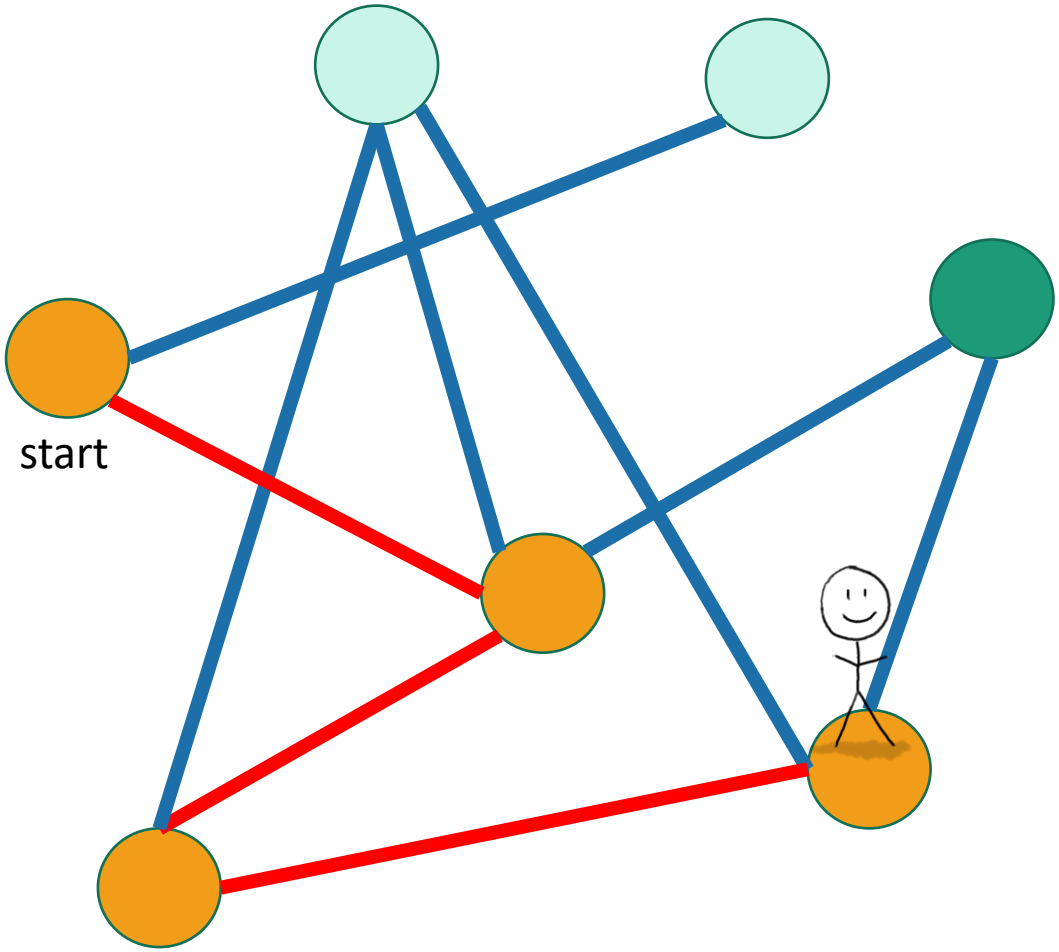
Not been there yet

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



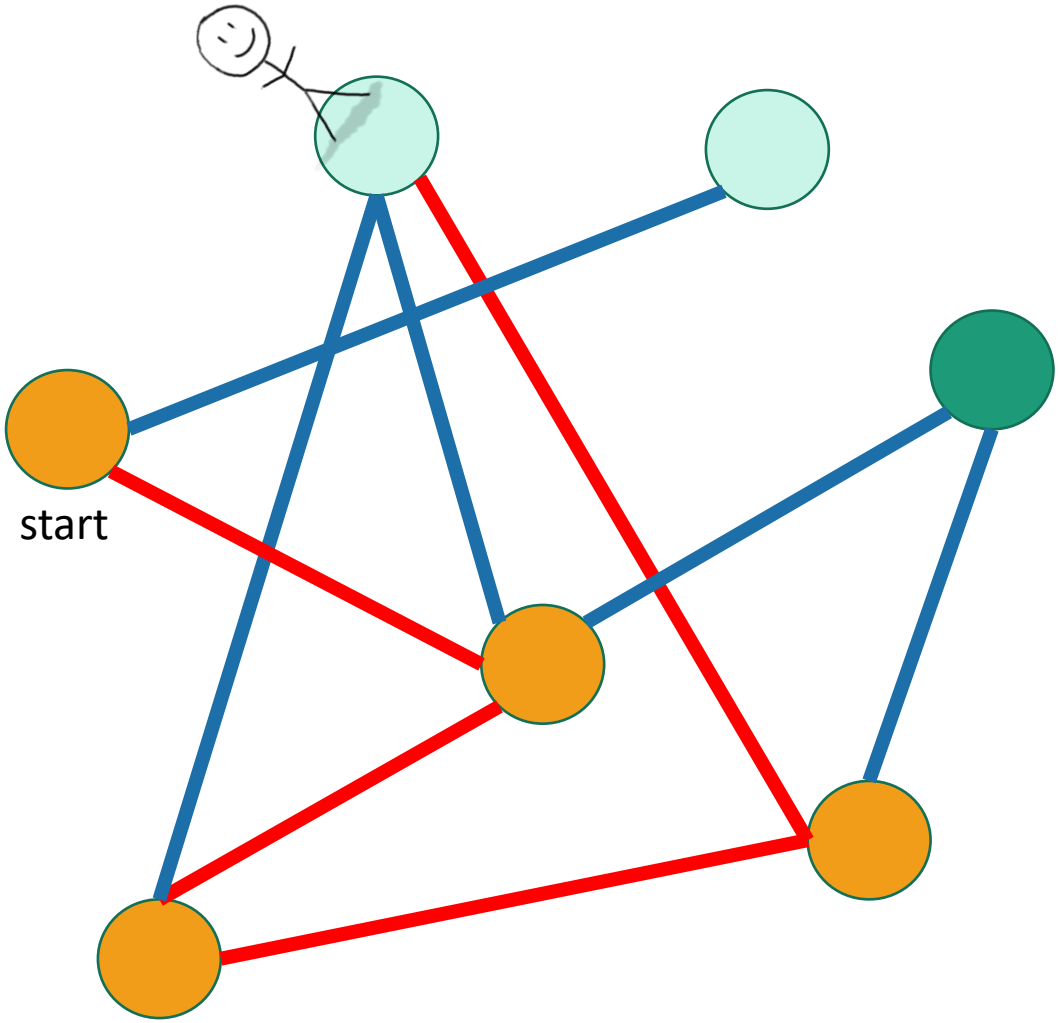
Not been there yet

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



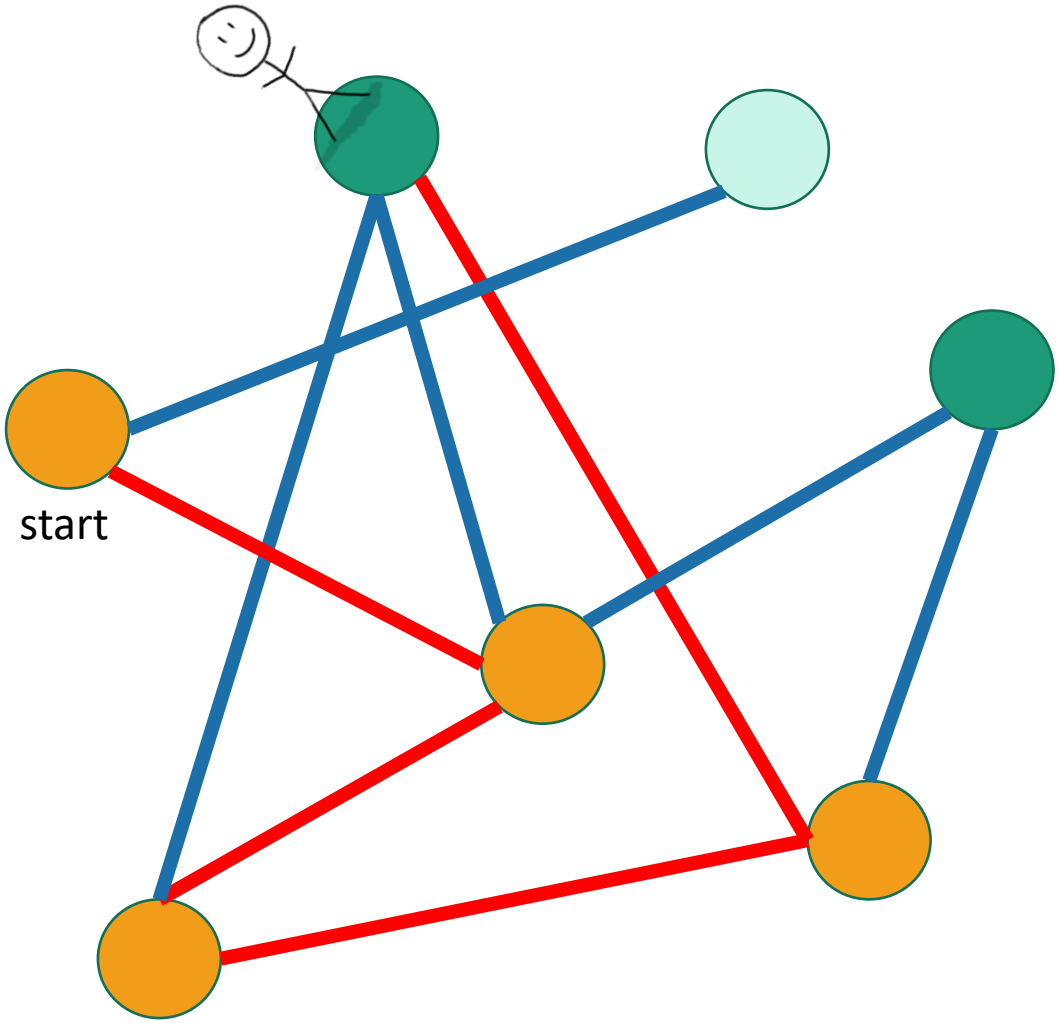
Not been there yet

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



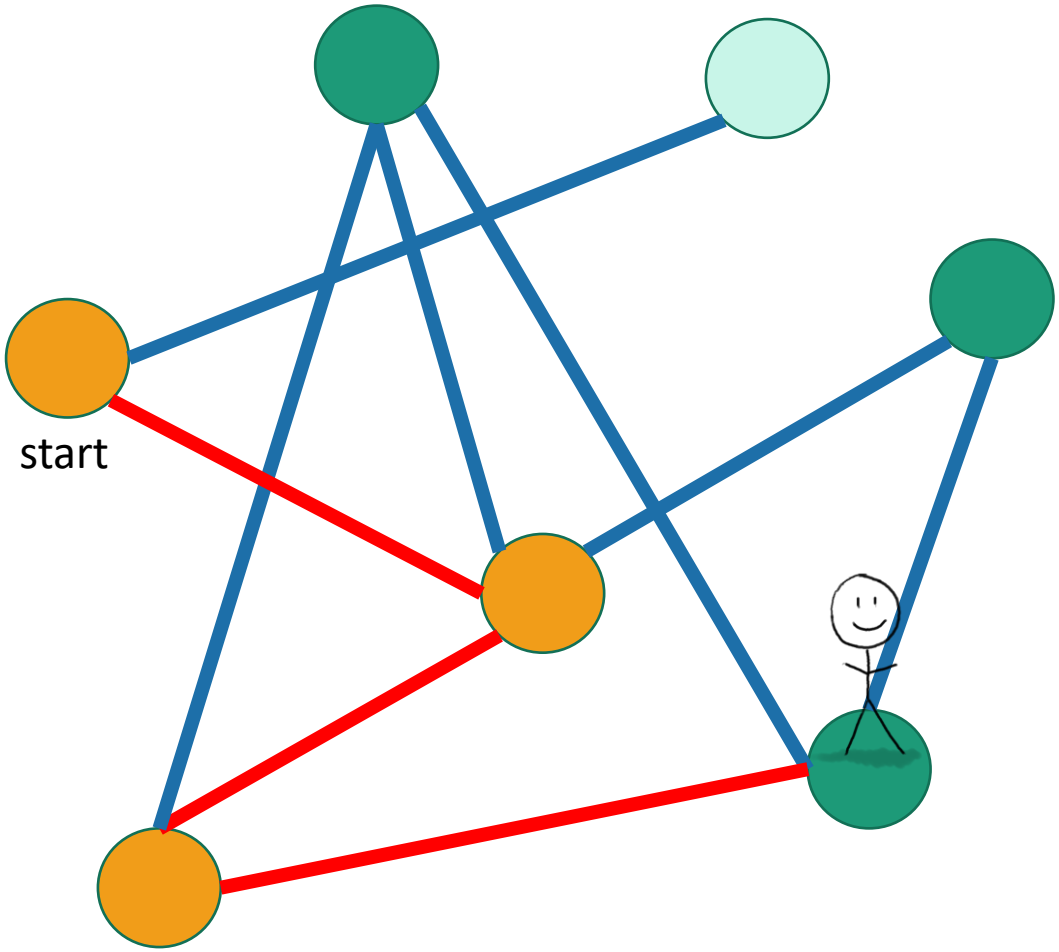
Not been there yet

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



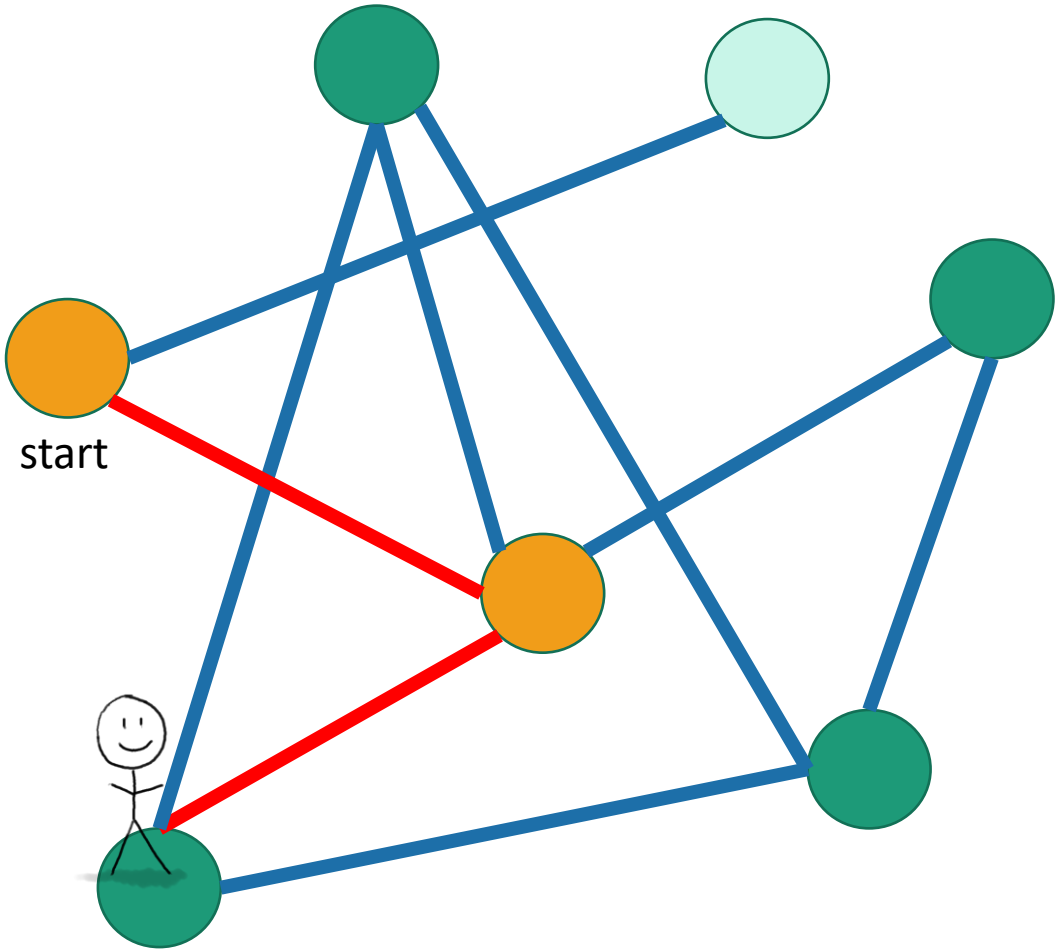
Not been there yet

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



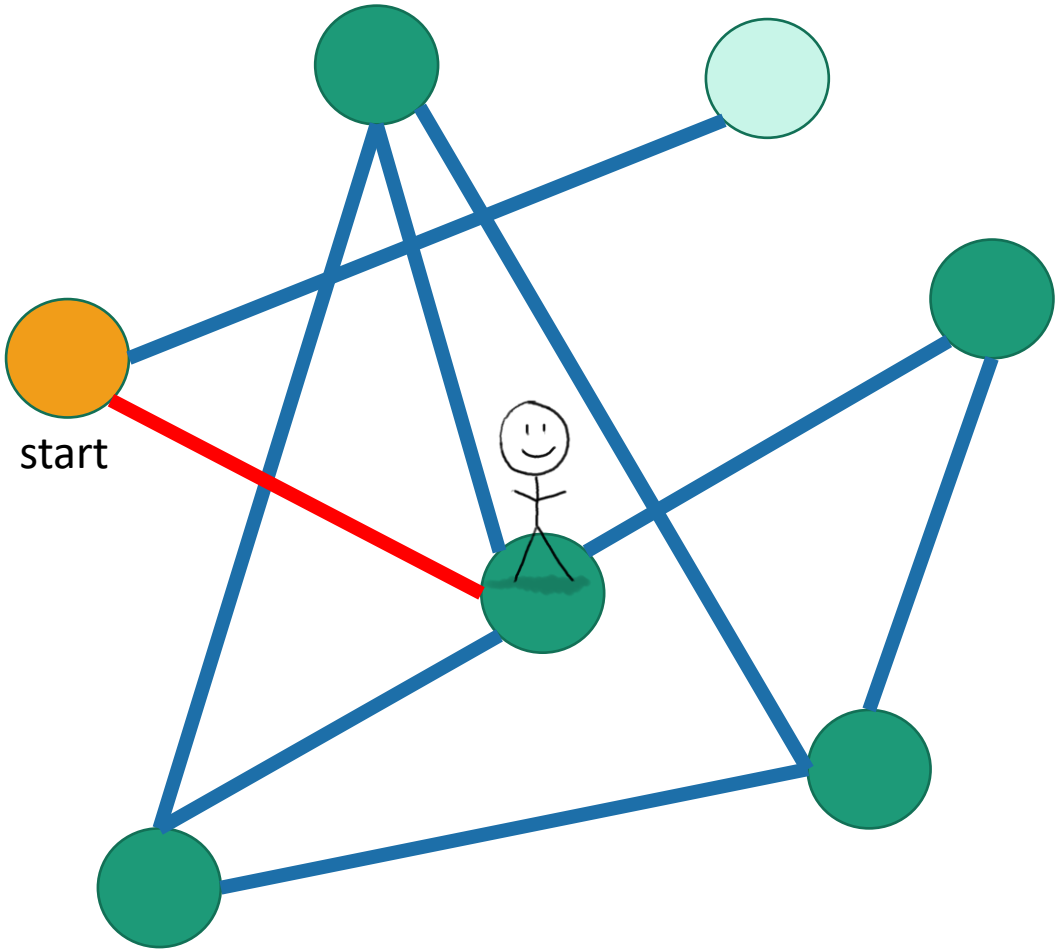
Not been there yet

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



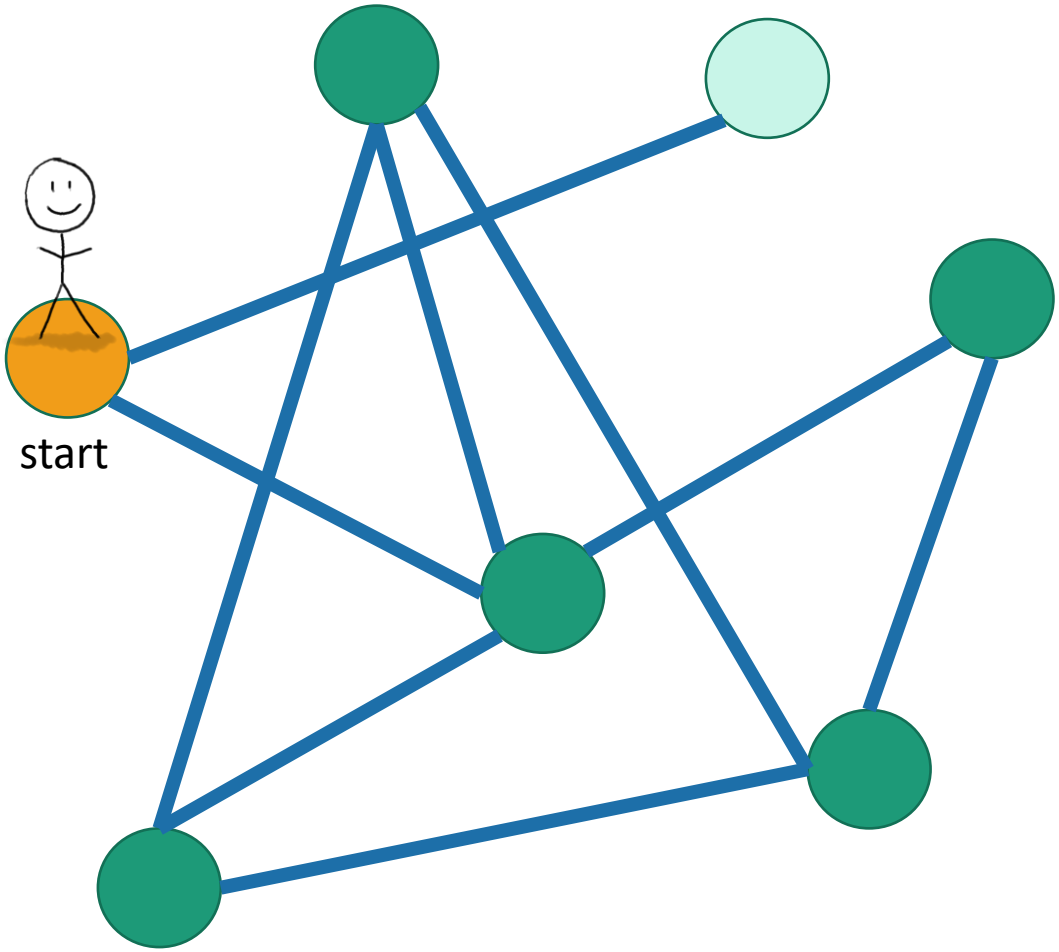
Not been there yet

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



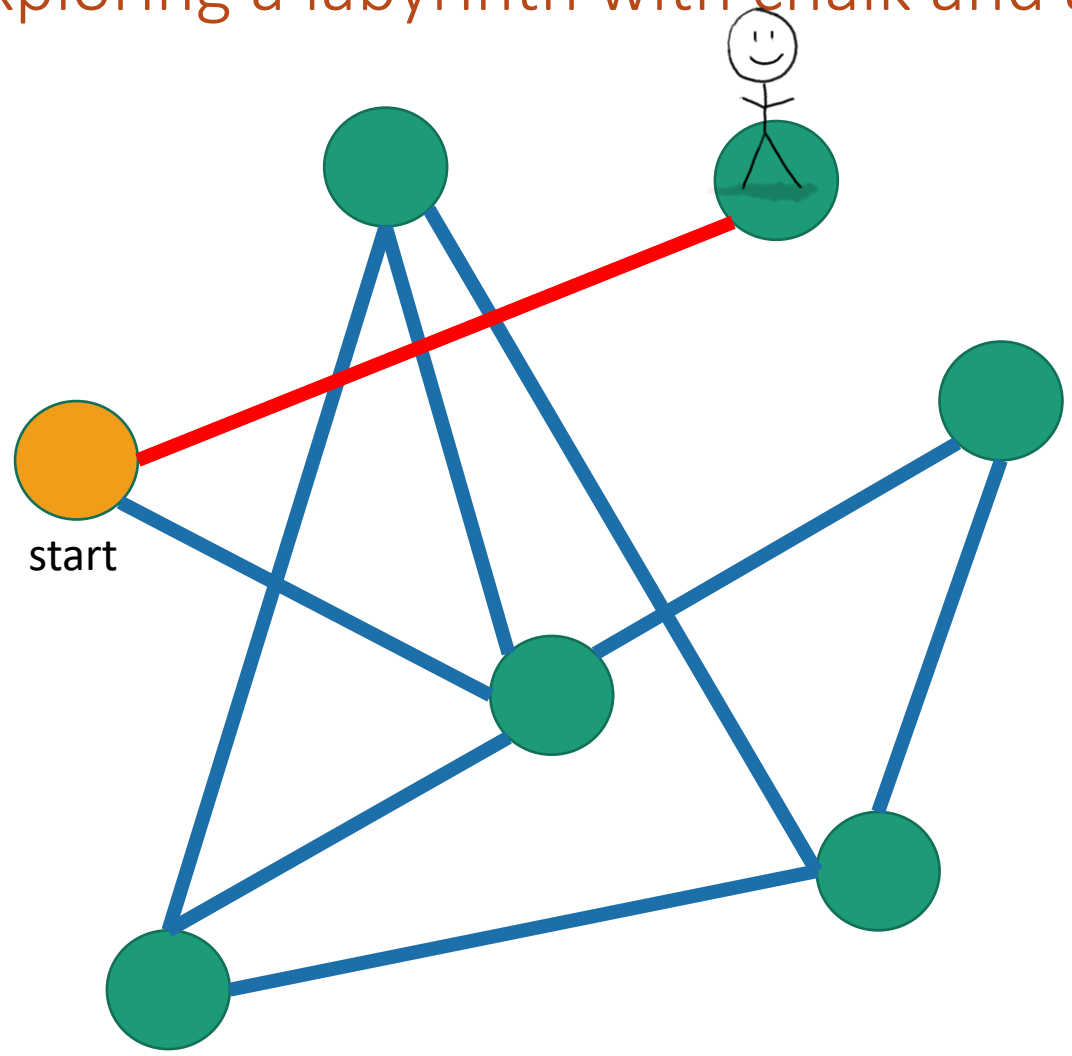
Not been there yet

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



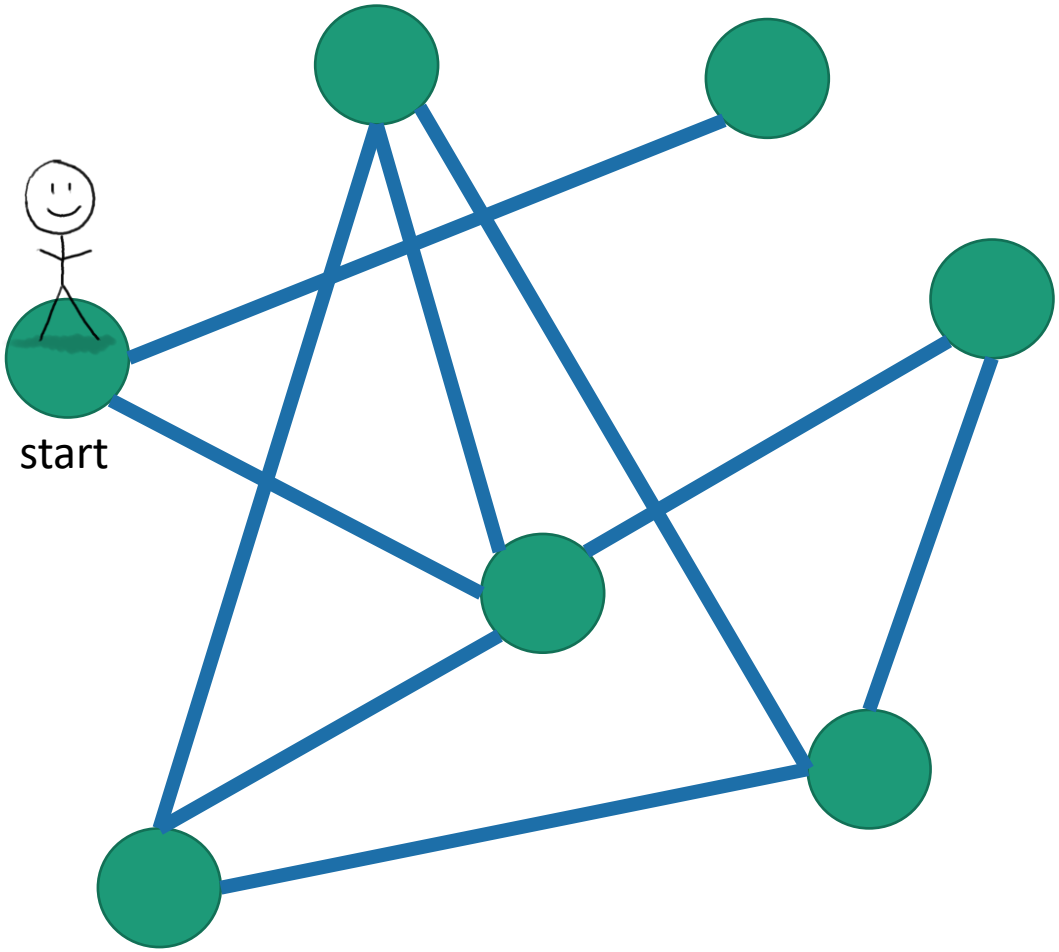
Not been there yet

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.

Labyrinth:
explored!

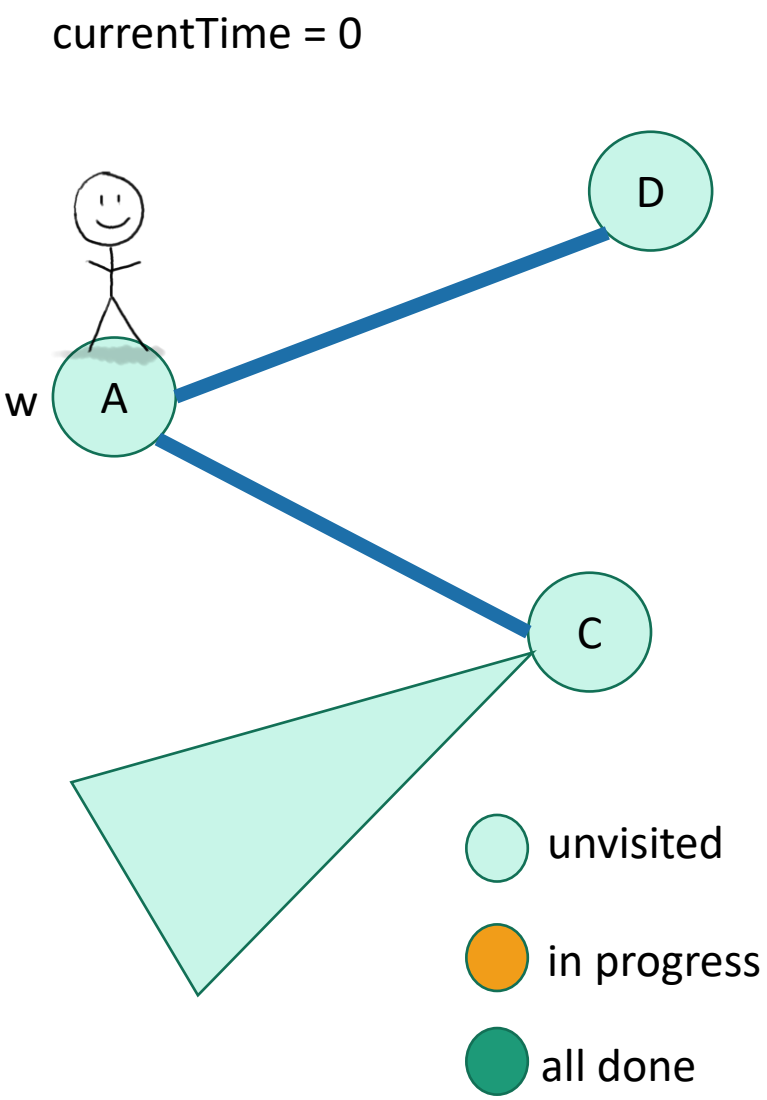
Depth First Search

Exploring a labyrinth with pseudocode

- Each vertex keeps track of whether it is:
  - Unvisited 
  - In progress 
  - All done 

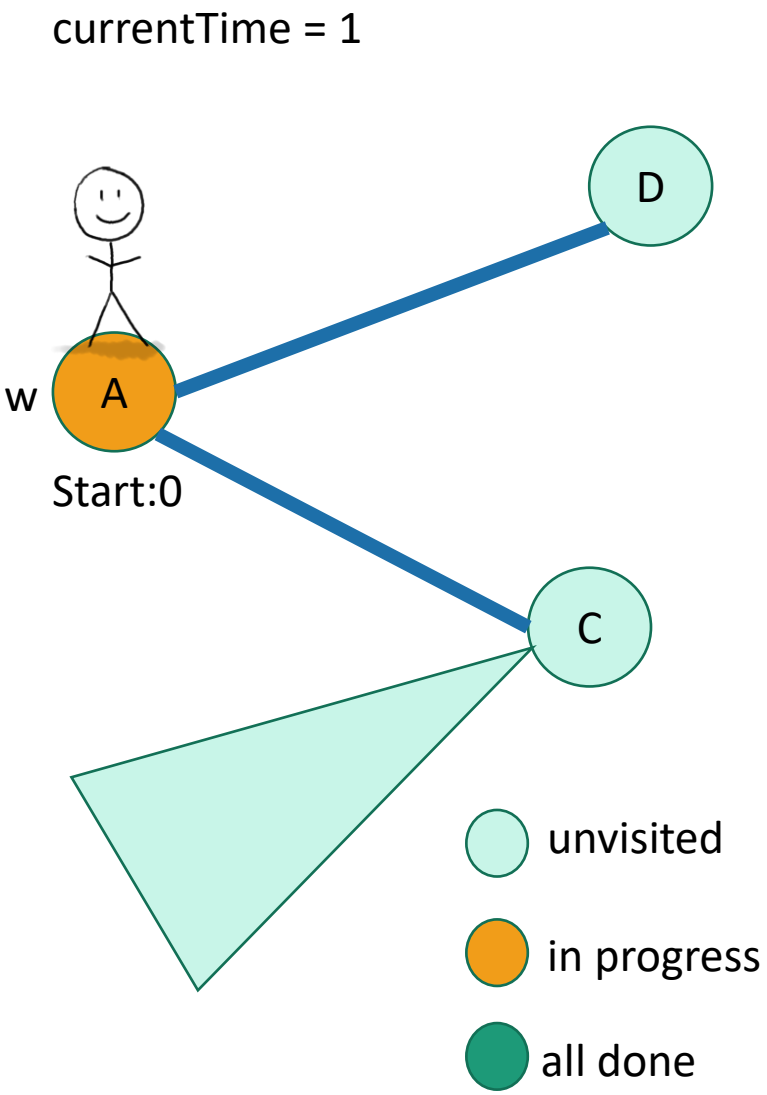
- Each vertex will also keep track of:
  - The time we **first enter it**.
  - The time we finish with it and mark it **all done**.

Depth First Search



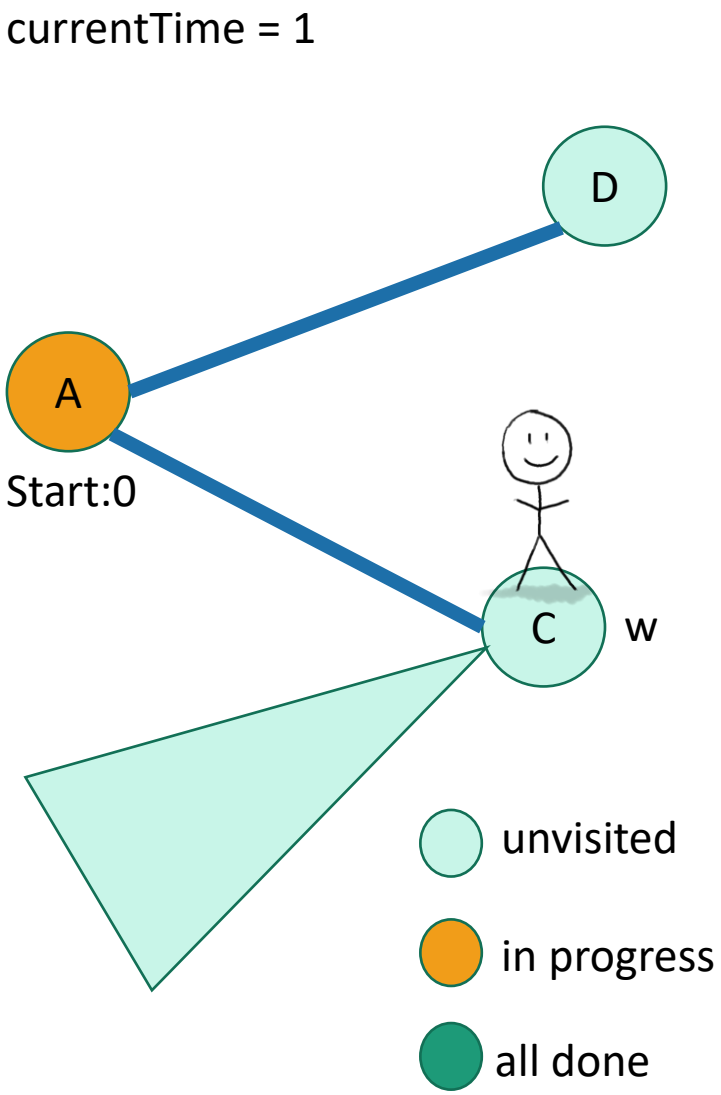
- **DFS**(w, currentTime):
  - w.startTime = currentTime
  - currentTime ++
  - Mark w as **in progress**.
  - **for** v in w.neighbors:
    - **if** v is **unvisited**:
      - currentTime = **DFS**(v, currentTime)
      - currentTime ++
  - w.finishTime = currentTime
  - Mark w as **all done**
  -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

Depth First Search



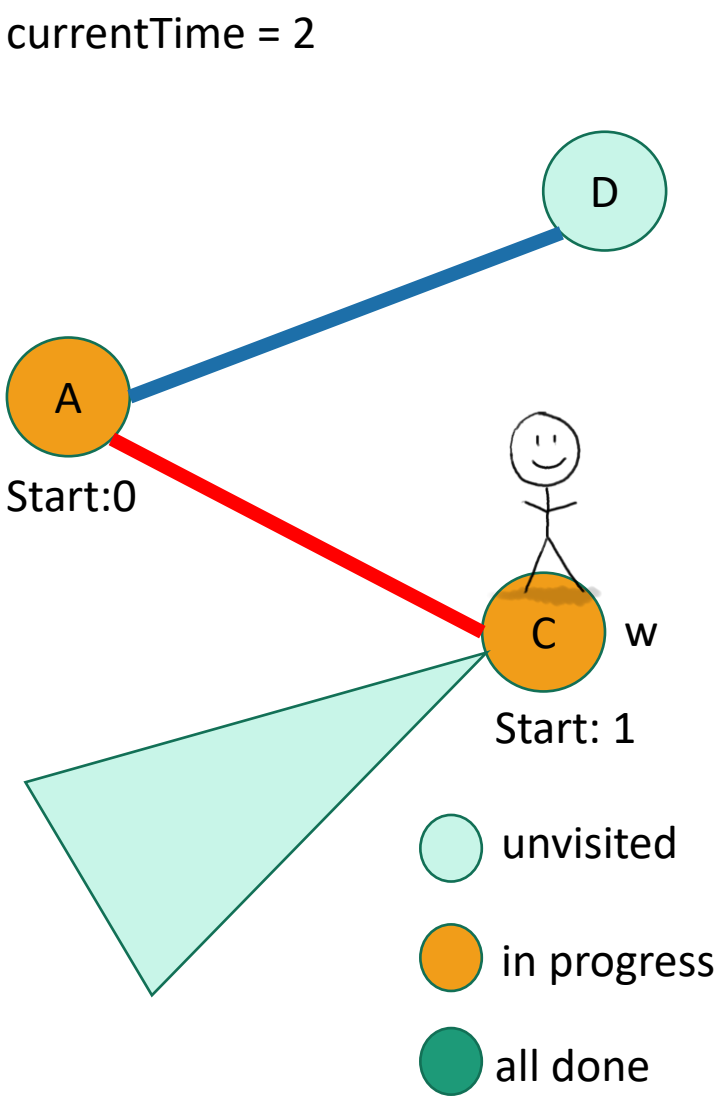
- **DFS**(w, currentTime):
  - w.startTime = currentTime
  - currentTime ++
  - Mark w as **in progress**.
  - **for** v in w.neighbors:
    - **if** v is **unvisited**:
      - currentTime = **DFS**(v, currentTime)
      - currentTime ++
  - w.finishTime = currentTime
  - Mark w as **all done**
  -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

Depth First Search



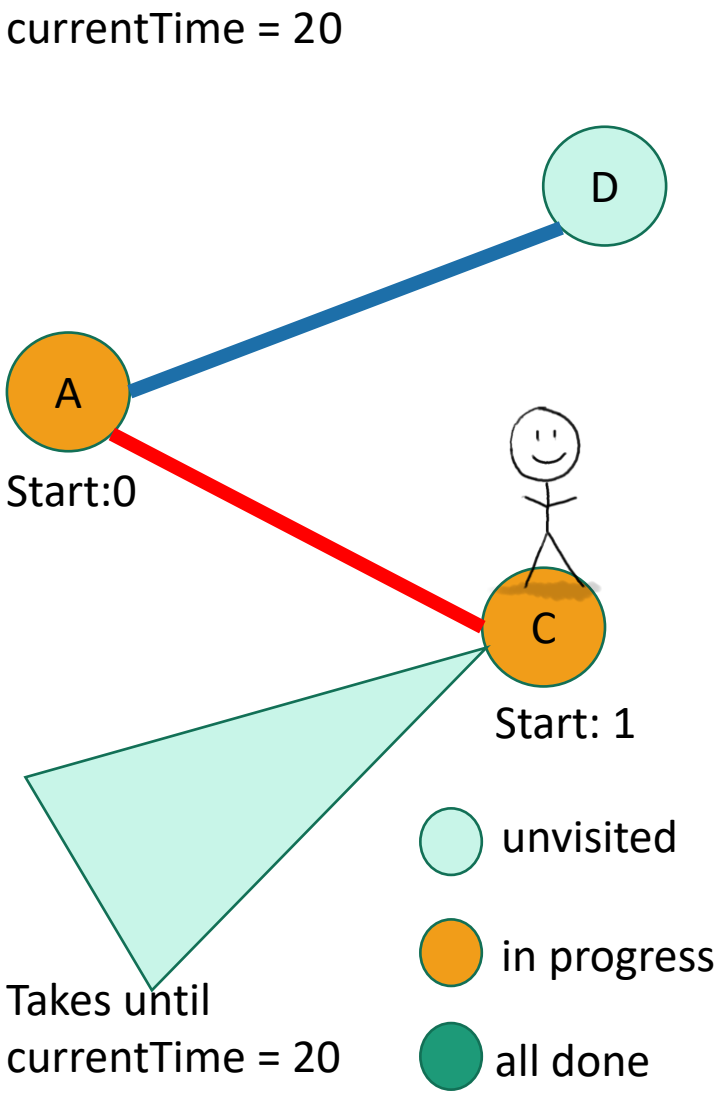
- **DFS**(w, currentTime):
  - w.startTime = currentTime
  - currentTime ++
  - Mark w as **in progress**.
  - **for** v in w.neighbors:
    - **if** v is **unvisited**:
      - currentTime = **DFS**(v, currentTime)
      - currentTime ++
  - w.finishTime = currentTime
  - Mark w as **all done**
  -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

Depth First Search



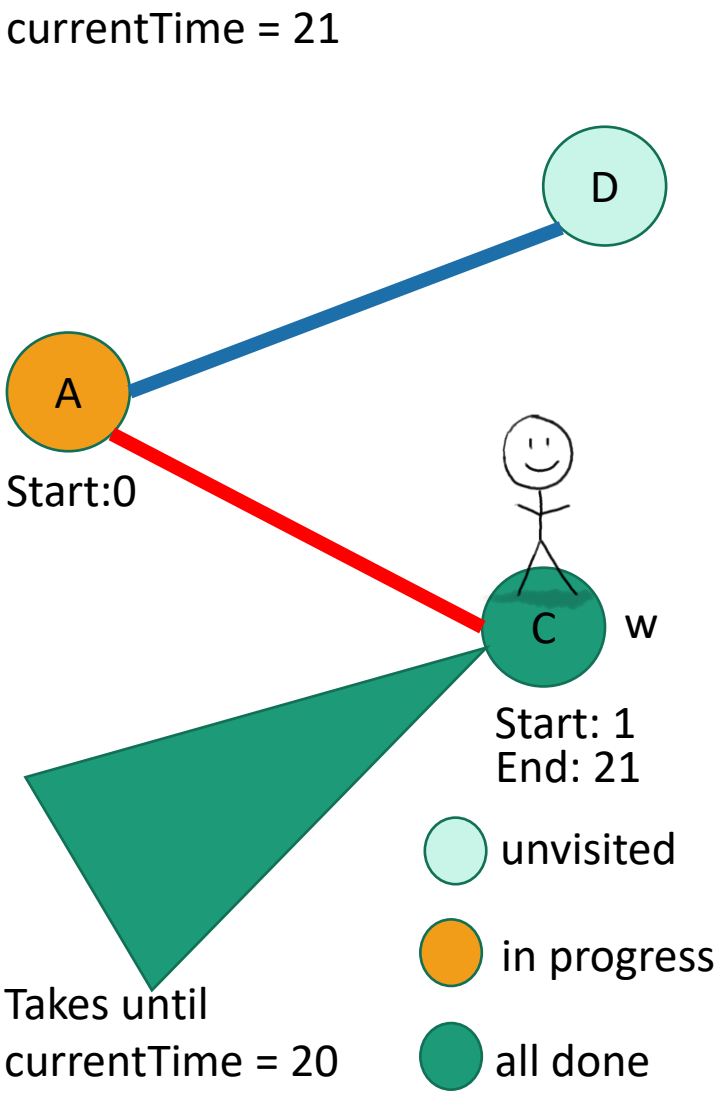
- **DFS**(w, currentTime):
  - w.startTime = currentTime
  - currentTime ++
  - Mark w as **in progress**.
  - **for** v in w.neighbors:
    - **if** v is **unvisited**:
      - currentTime = **DFS**(v, currentTime)
      - currentTime ++
  - w.finishTime = currentTime
  - Mark w as **all done**
  -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

Depth First Search



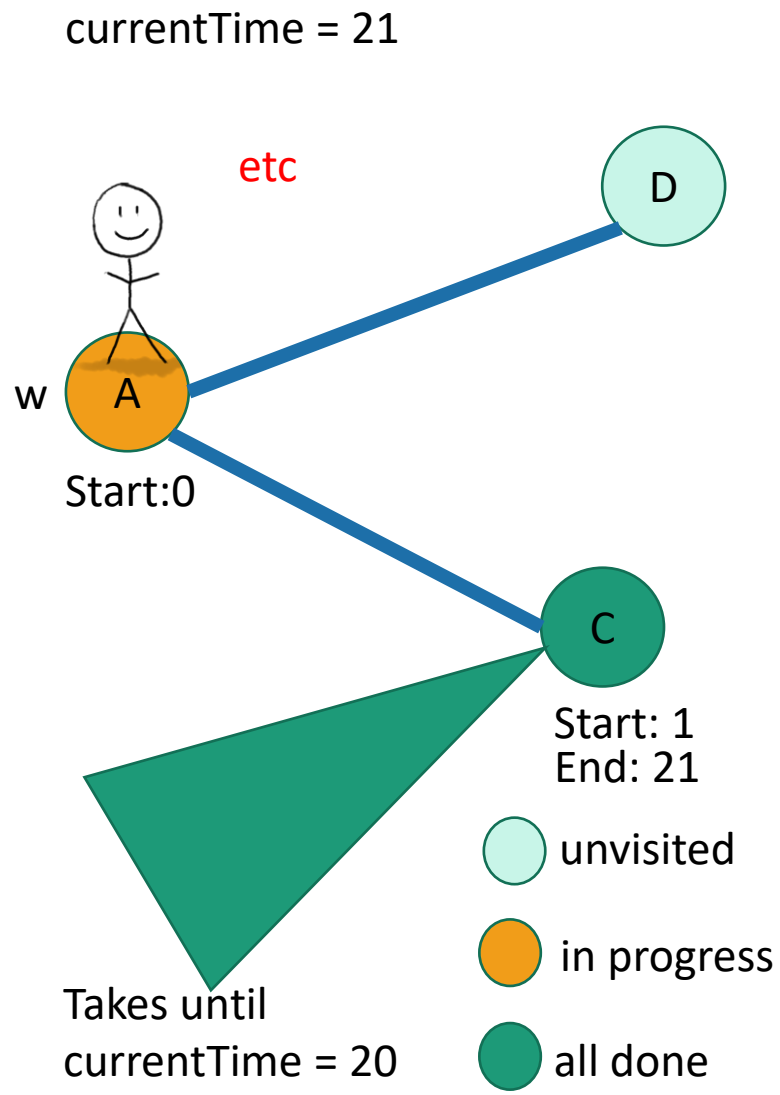
- **DFS**(w, currentTime):
  - w.startTime = currentTime
  - currentTime ++
  - Mark w as in progress.
  - **for** v in w.neighbors:
    - **if** v is unvisited:
      - currentTime = **DFS**(v, currentTime)
      - currentTime ++
  - w.finishTime = currentTime
  - Mark w as **all done**
  - **return**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

Depth First Search



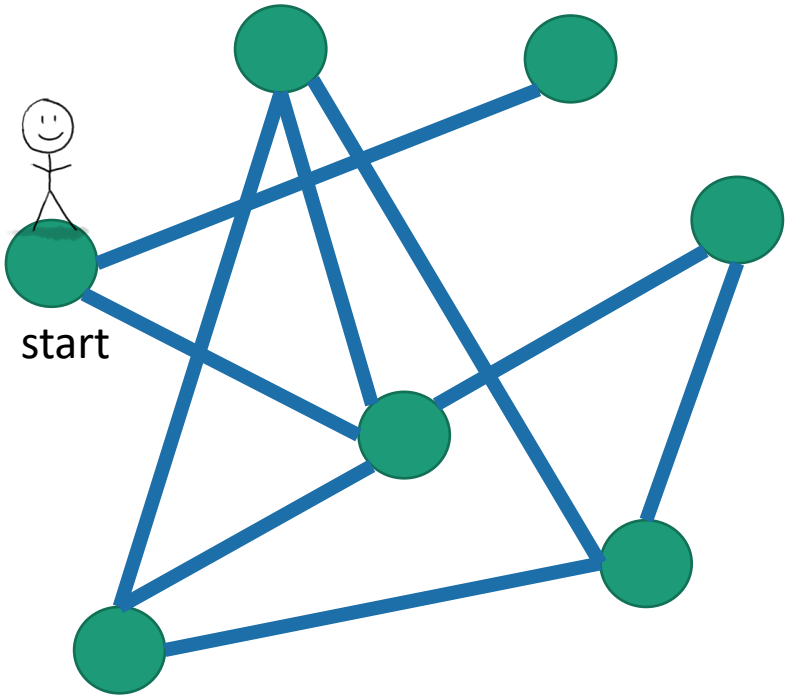
- **DFS**(w, currentTime):
  - w.startTime = currentTime
  - currentTime ++
  - Mark w as in progress.
  - **for** v in w.neighbors:
    - **if** v is unvisited:
      - currentTime = **DFS**(v, currentTime)
      - currentTime ++
  - w.finishTime = currentTime
  - Mark w as **all done**
  - **return**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

Depth First Search

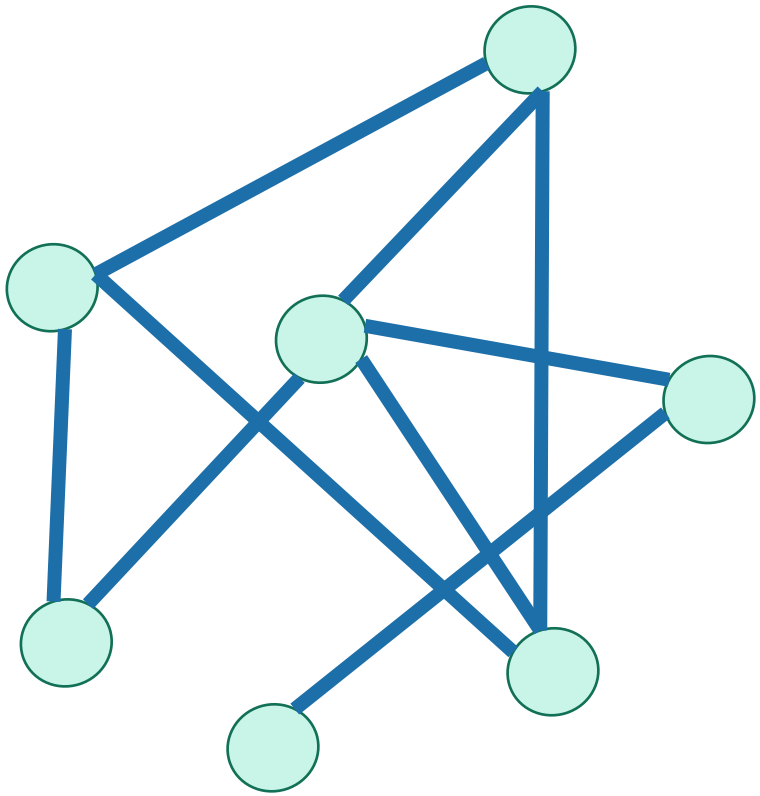


- **DFS**(w, currentTime):
  - w.startTime = currentTime
  - currentTime ++
  - Mark w as in progress.
  - **for** v in w.neighbors:
    - **if** v is unvisited:
      - currentTime = **DFS**(v, currentTime)
      - currentTime ++
  - w.finishTime = currentTime
  - Mark w as **all done**
  - **return**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

DFS finds all the nodes reachable from the starting point



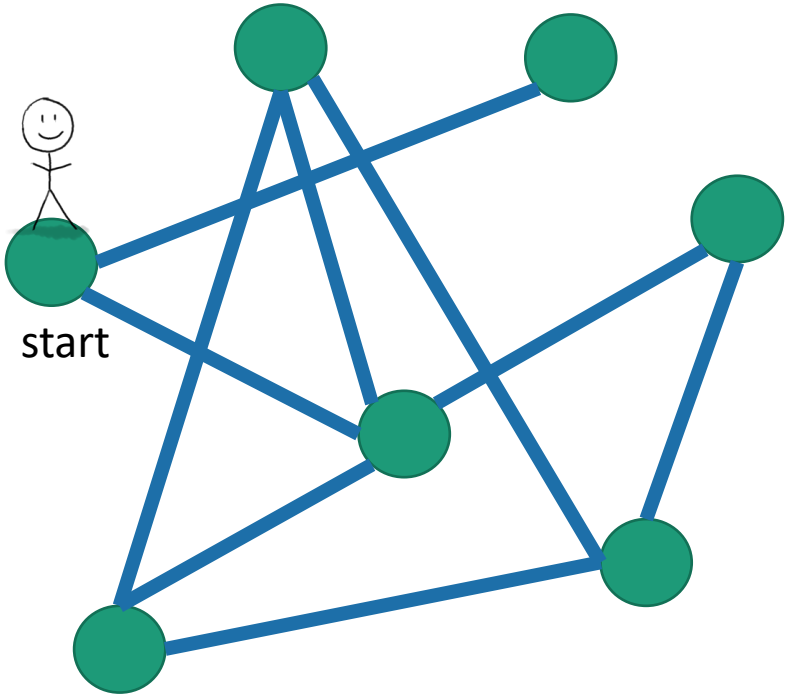
In an undirected graph, this is called a **connected component.**

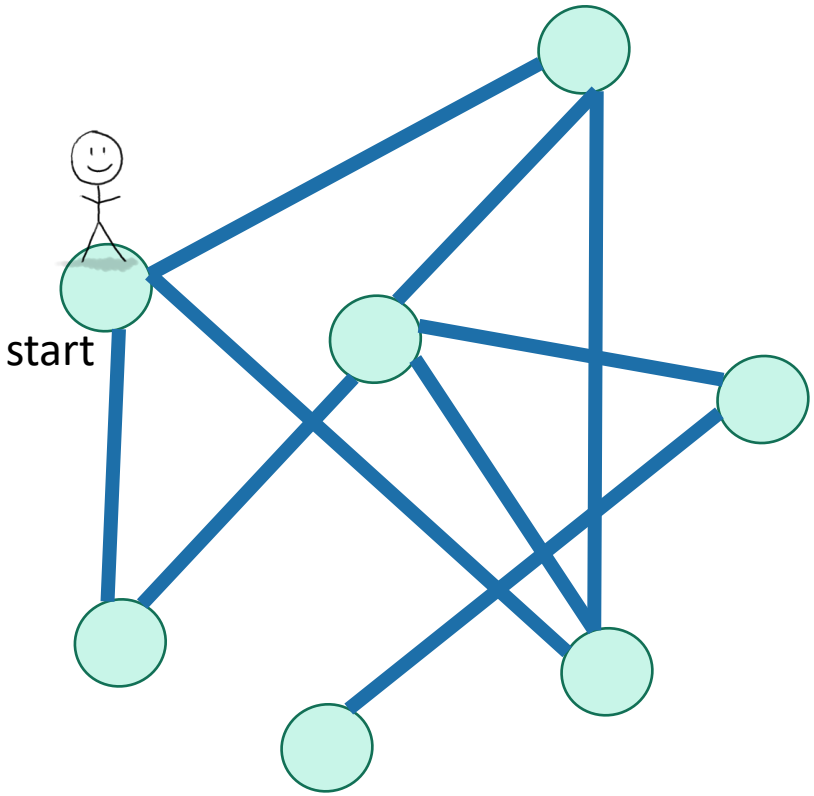


One application: finding connected components.

To explore the whole graph

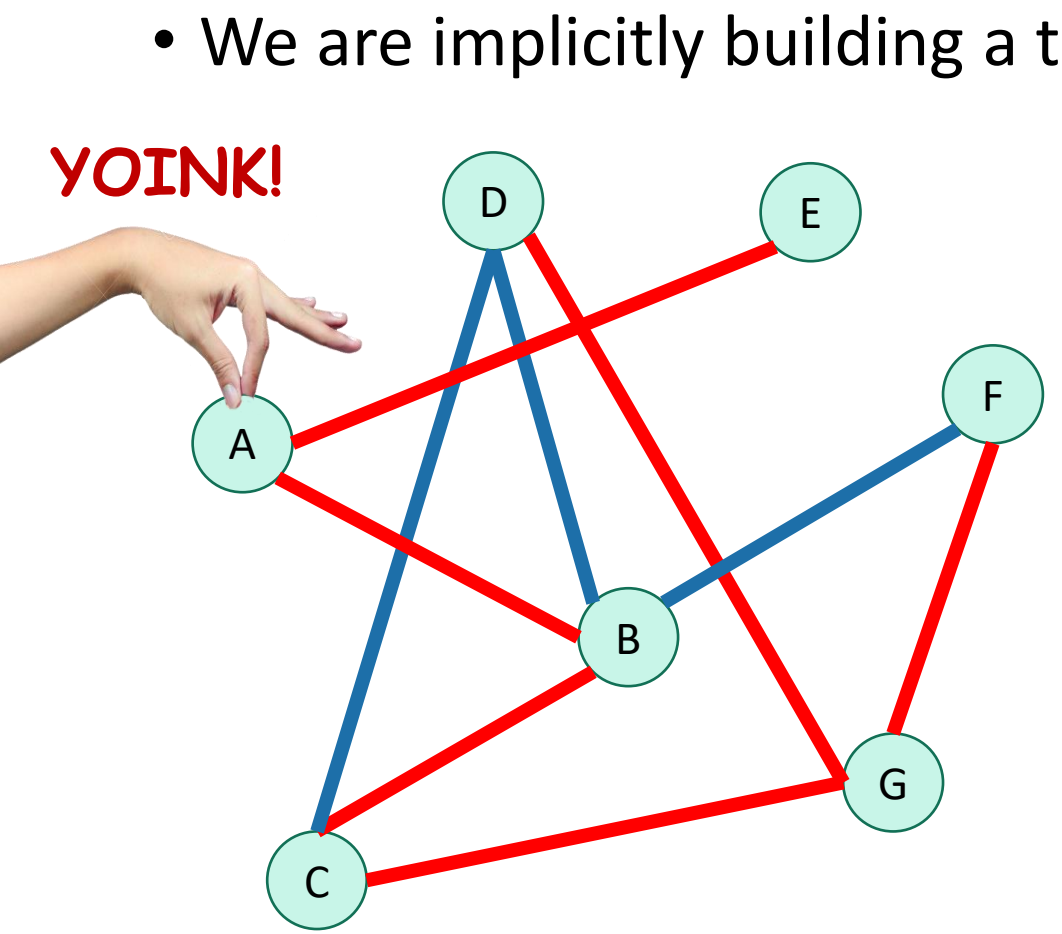
- Do it repeatedly!

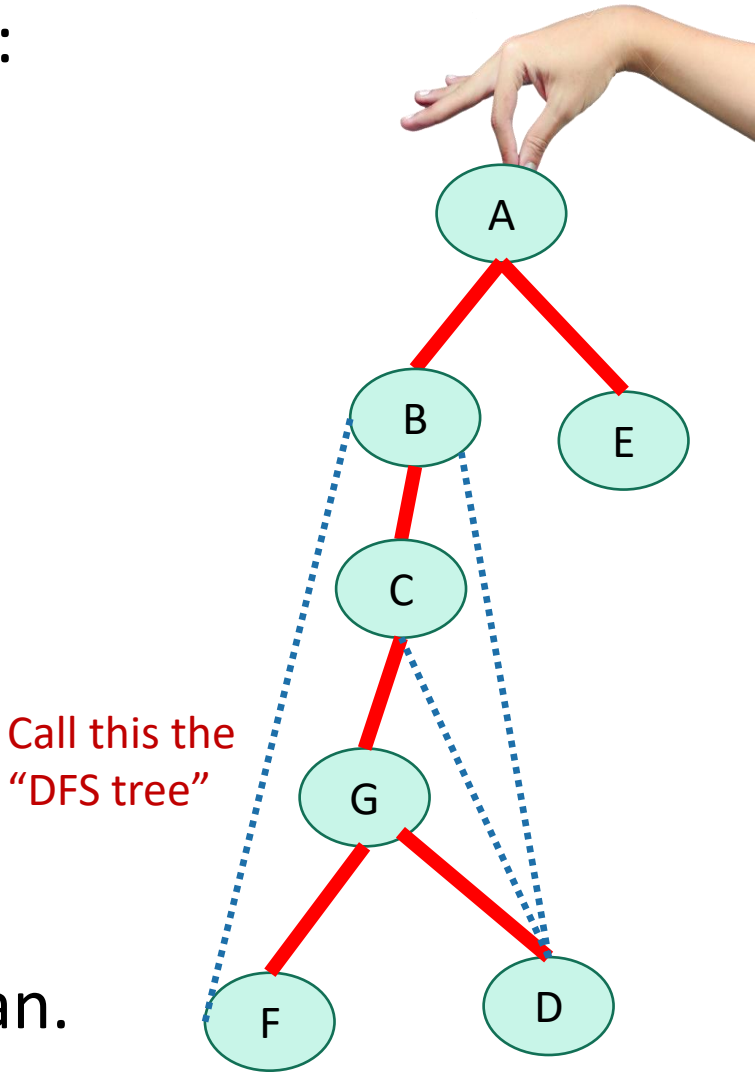




Why is it called depth-first?

- We are implicitly building a tree:





Call this the "DFS tree"

- And first we go as deep as we can.

n vertices and m edges.

Running time

To explore **just the connected component** we started in

- We look at each edge only once.
- And basically don't do anything else.
- So…

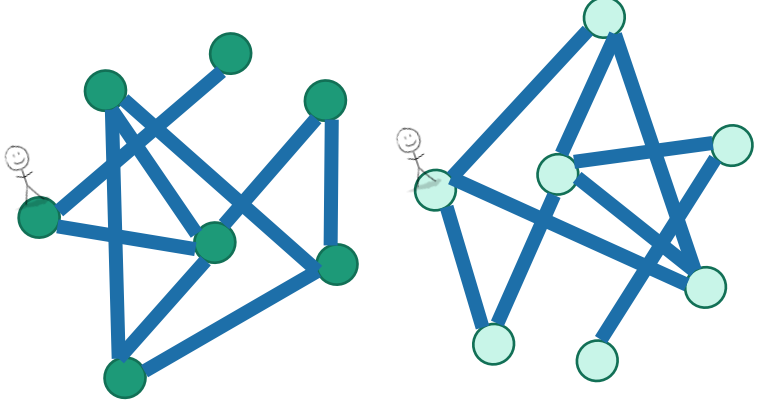
$$O(m)$$

- (Assuming we are using the linked-list representation)

n vertices and m edges.

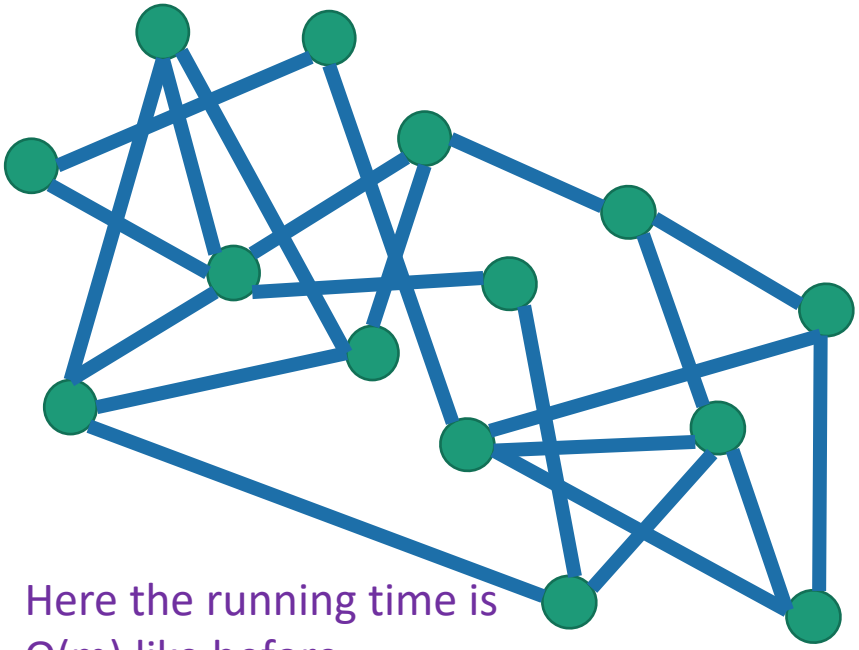
Running time

To explore the whole thing



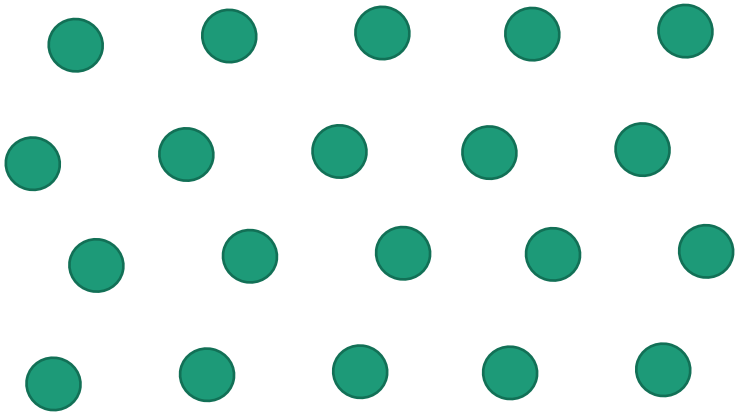
- Explore the connected components one-by-one.
- This takes time

$$O(n + m)$$



Here the running time is O(m) like before

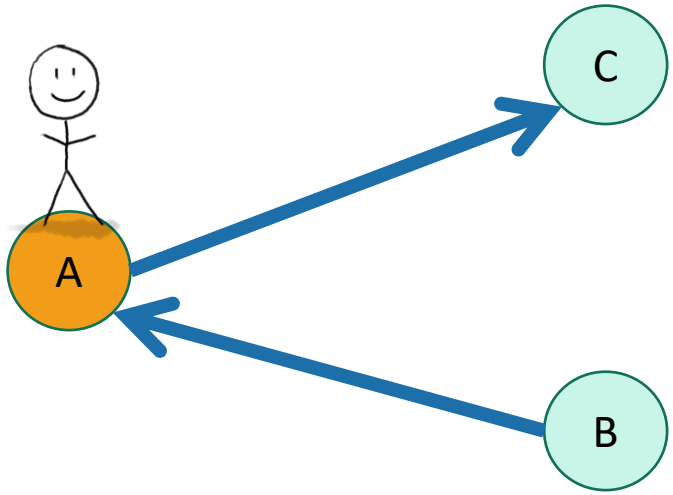
or



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

You check:

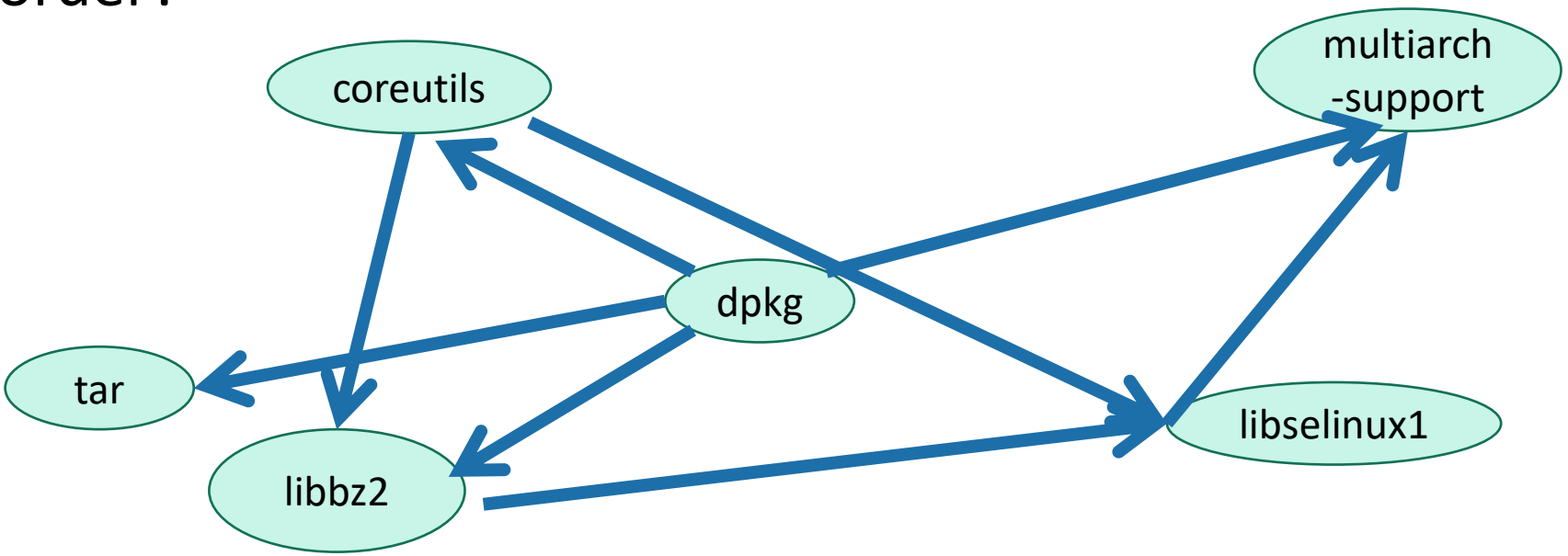
DFS works fine on directed graphs too!



Only walk to C, not to B.

Exercise

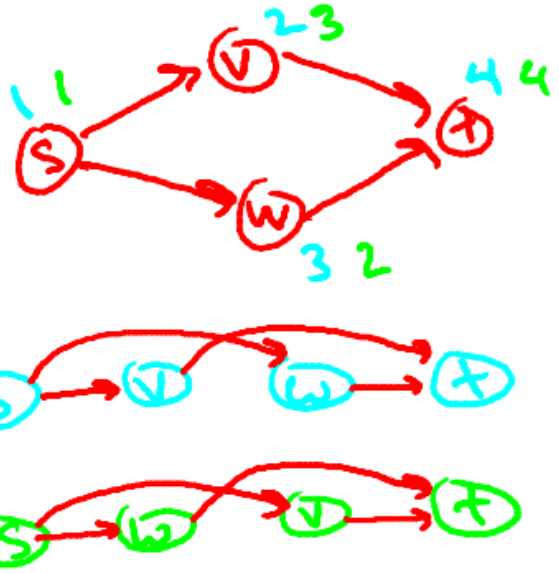
- How can you sign up for classes so that you never violate the pre-req requirements?
- More practically, given a package dependency graph, how do you install packages in the correct order?



DFS Application-Topological Ordering

Definition : A topological ordering of a directed graph G is a labeling f of G's nodes such that:

1. The f(v)'s are the set {1,2,..,n}
2. $(u, v) \in G => f(u) < f(v)$



Motivation : sequence tasks while respecting all precedence constraints.

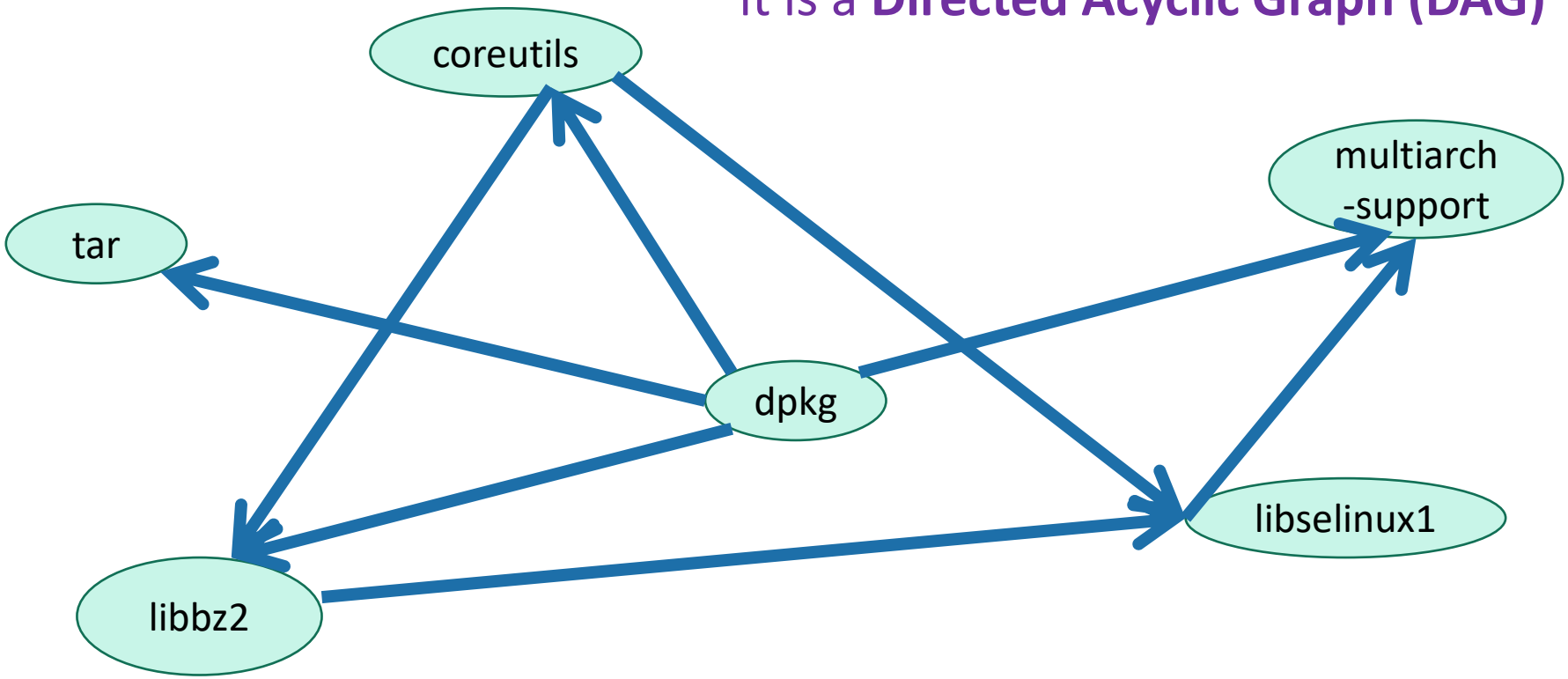
Note : G has directed cycle => no topological ordering

Theorem : no directed cycle => can compute topological ordering in O(m+n) time.

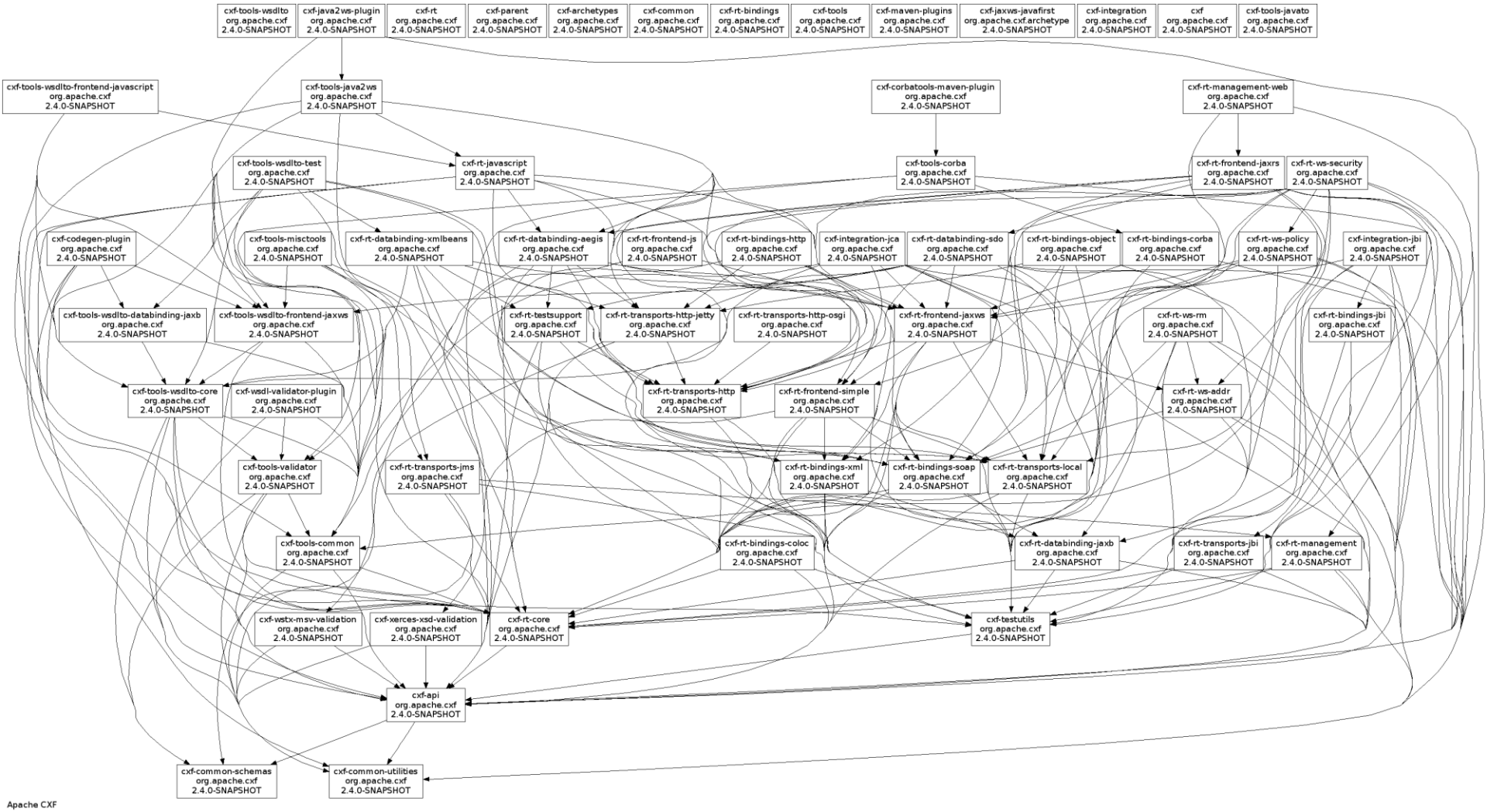
Application: topological sorting

- Question: in what order should I install packages?

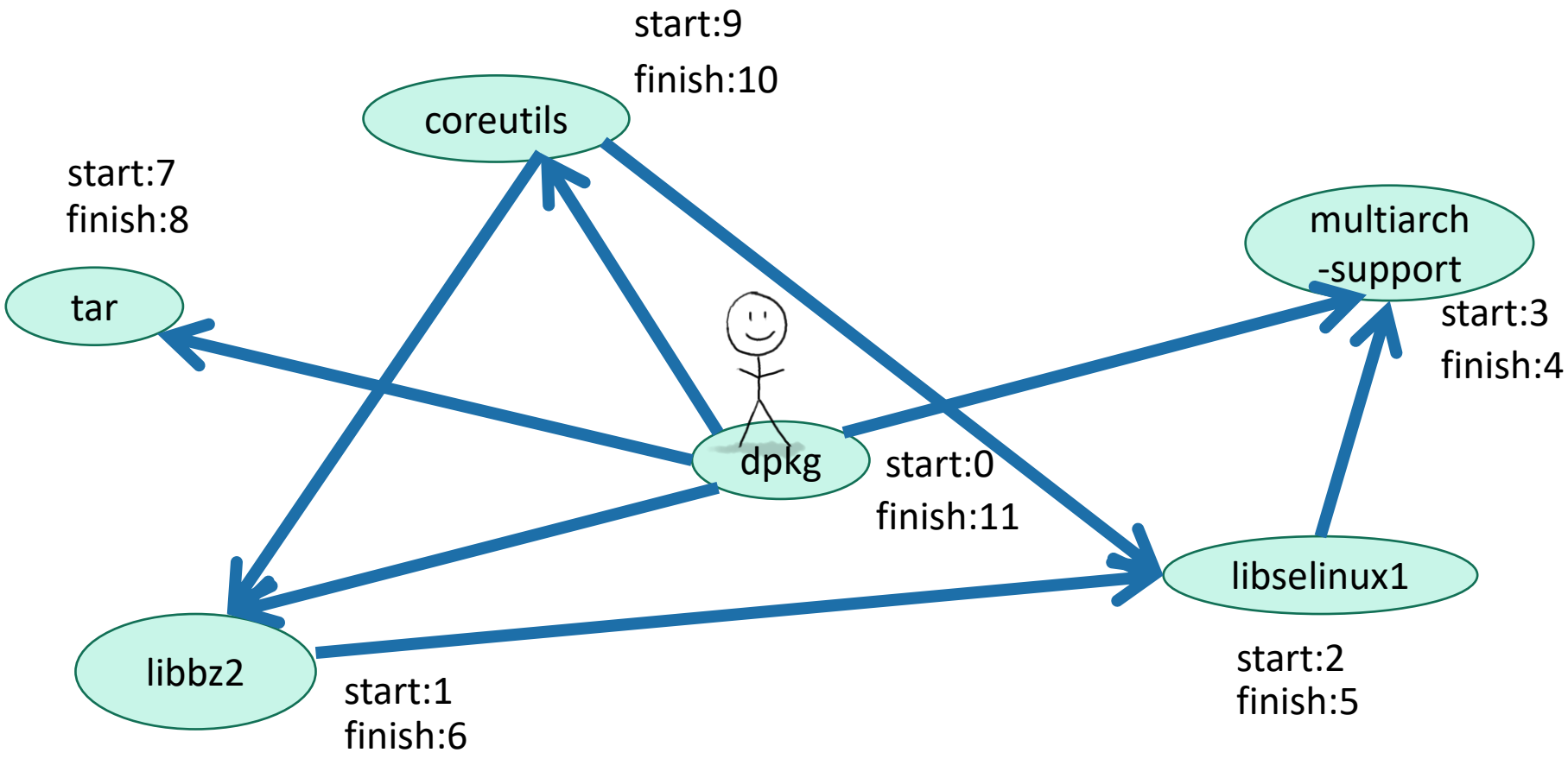
Suppose the dependency graph has no cycles:
it is a **Directed Acyclic Graph (DAG)**



Can’t always eyeball it.



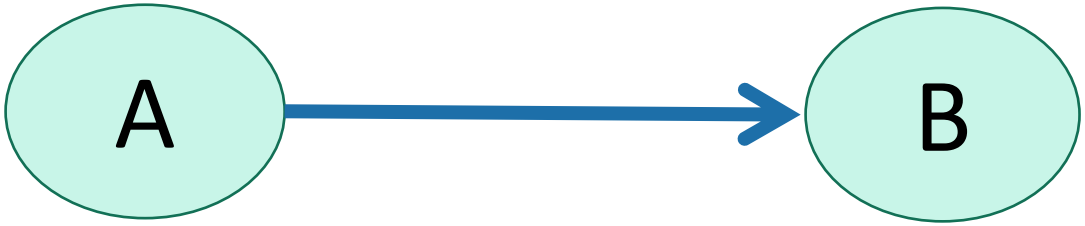
Let’s do DFS


start:9
finish:10
coreutils
start:7
finish:8
tar
multiarch
-support
start:3
finish:4
dpkg
start:0
finish:11
libselinux1
libbz2
start:1
finish:6
start:2
finish:5

Suppose the underlying graph has no cycles

Finish times seem useful

Claim: In general, we'll always have:



finish: [larger] finish: [smaller]

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

A more general statement

(this holds even if there are cycles)

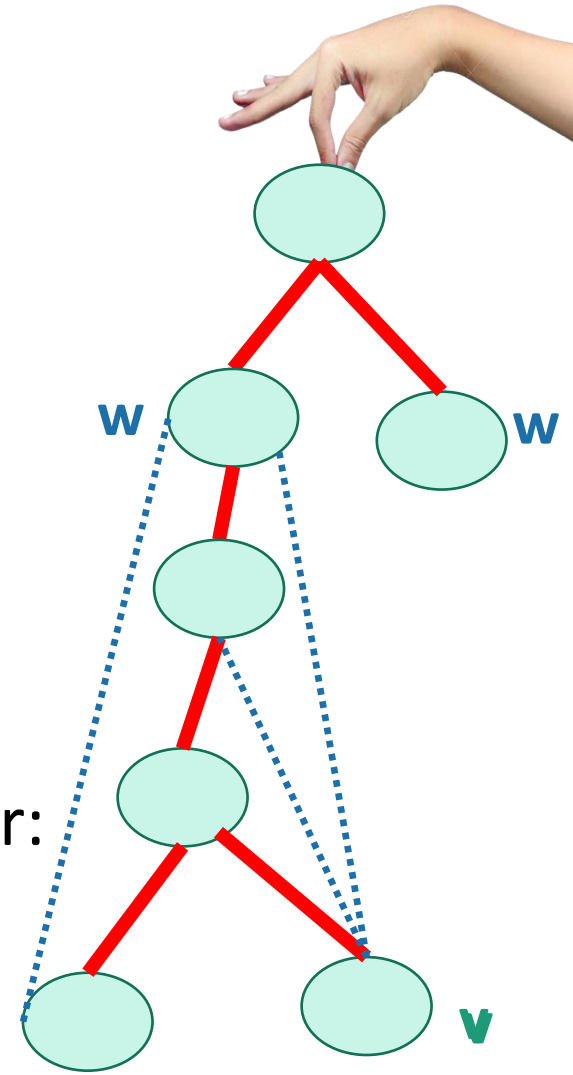
This is called the "parentheses theorem" in CLRS

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

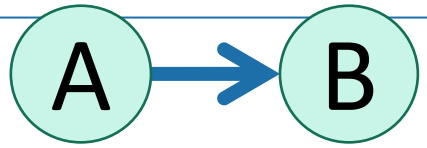
- If w is a descendant of v in this tree:

- If neither are descendants of each other:

(or the other way around)

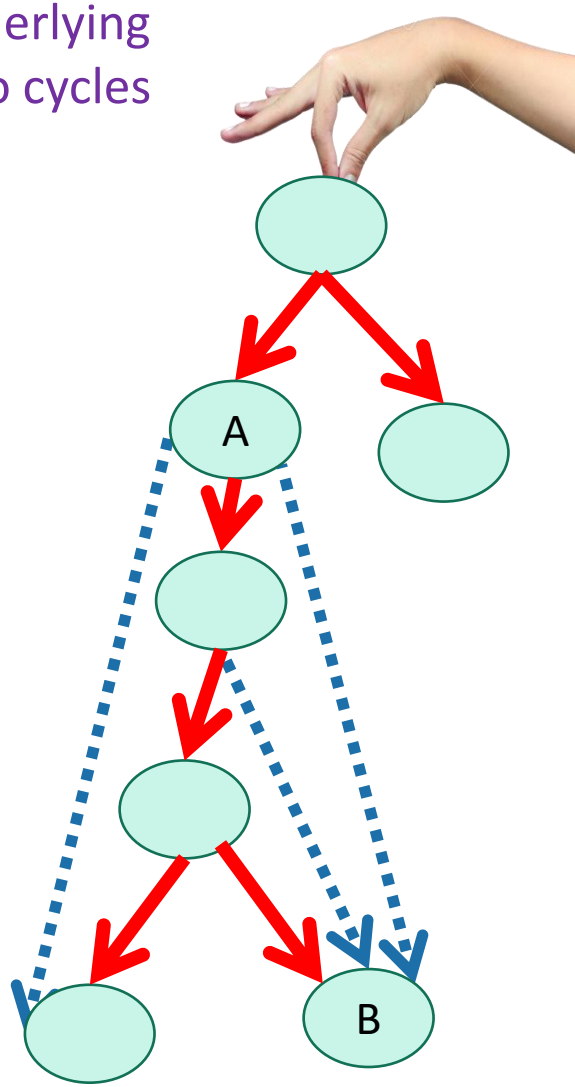


So to prove this ->

If 

Then B.finishTime < A.finishTime

Suppose the underlying graph has no cycles



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

- Then

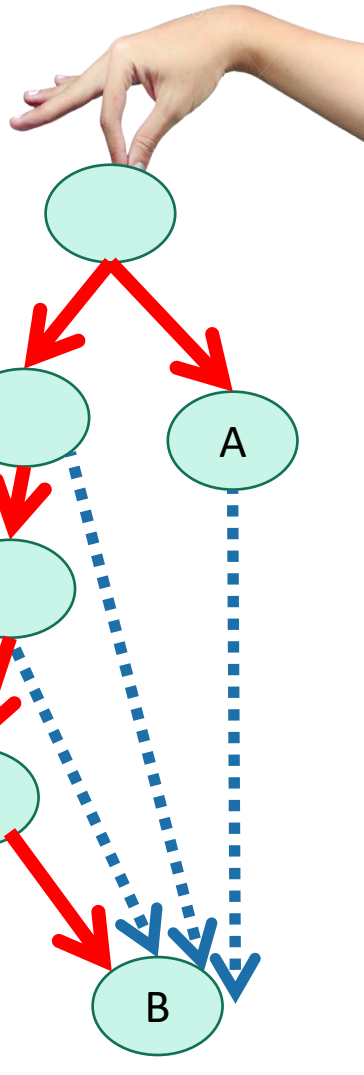
- aka, B.finishTime < A.finishTime.

So to prove this ->

If A → B

Then B.finishTime < A.finishTime

Suppose the underlying graph has no cycles



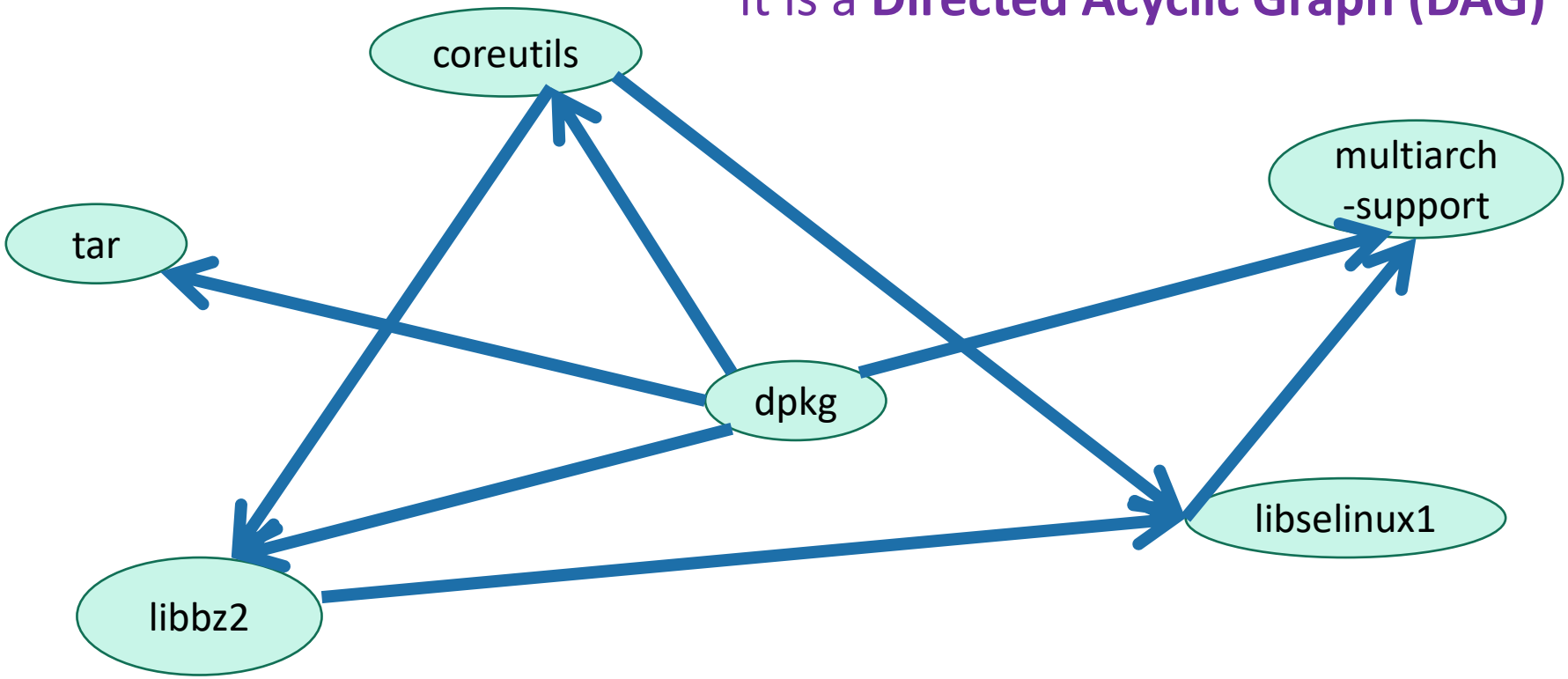
- **Case 2**: B is a NOT descendant of A in the DFS tree.
- Then we must have explored B before A.
  - Otherwise we would have gotten to B from A, and B would have been a descendant of A in the DFS tree.
- Then

- aka, B.finishTime < A.finishTime.

Back to this problem

- Question: in what order should I install packages?

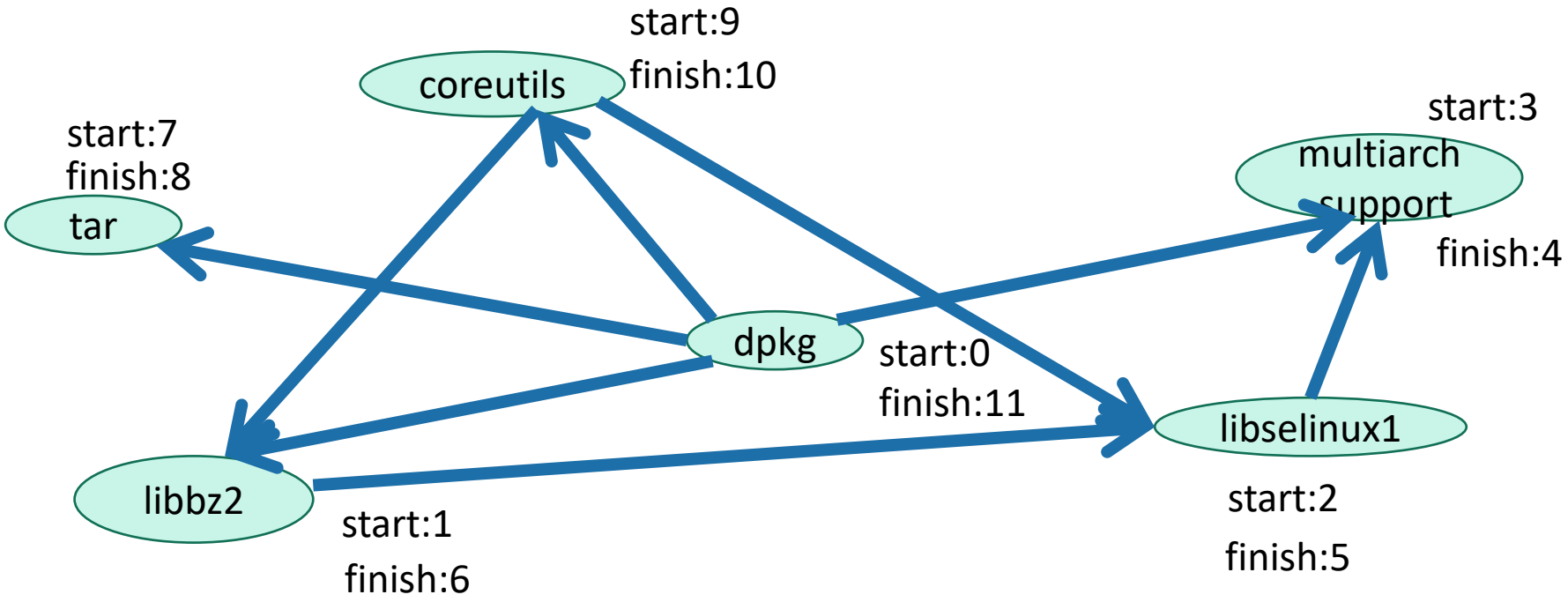
Suppose the dependency graph has no cycles:
it is a **Directed Acyclic Graph (DAG)**



In reverse order of finishing time

- Do DFS
- Maintain a list of packages, in the order you want to install them.
- When you mark a vertex as **all done**, put it at the **beginning** of the list.

- `dpkg`
- `coreutils`
- `tar`
- `libbz2`
- `libselinux1`
- `multiarch_support`

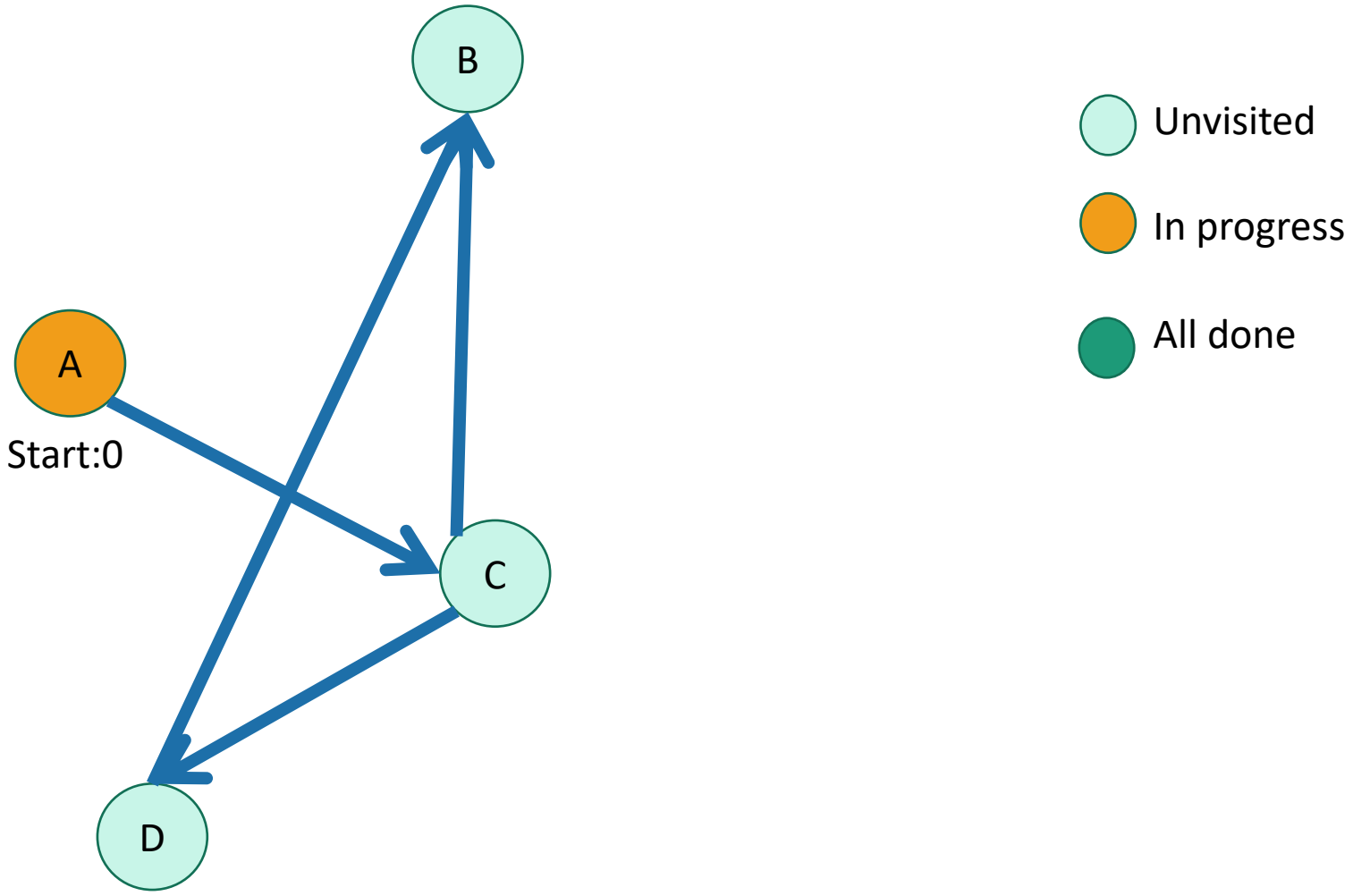


What did we just learn?

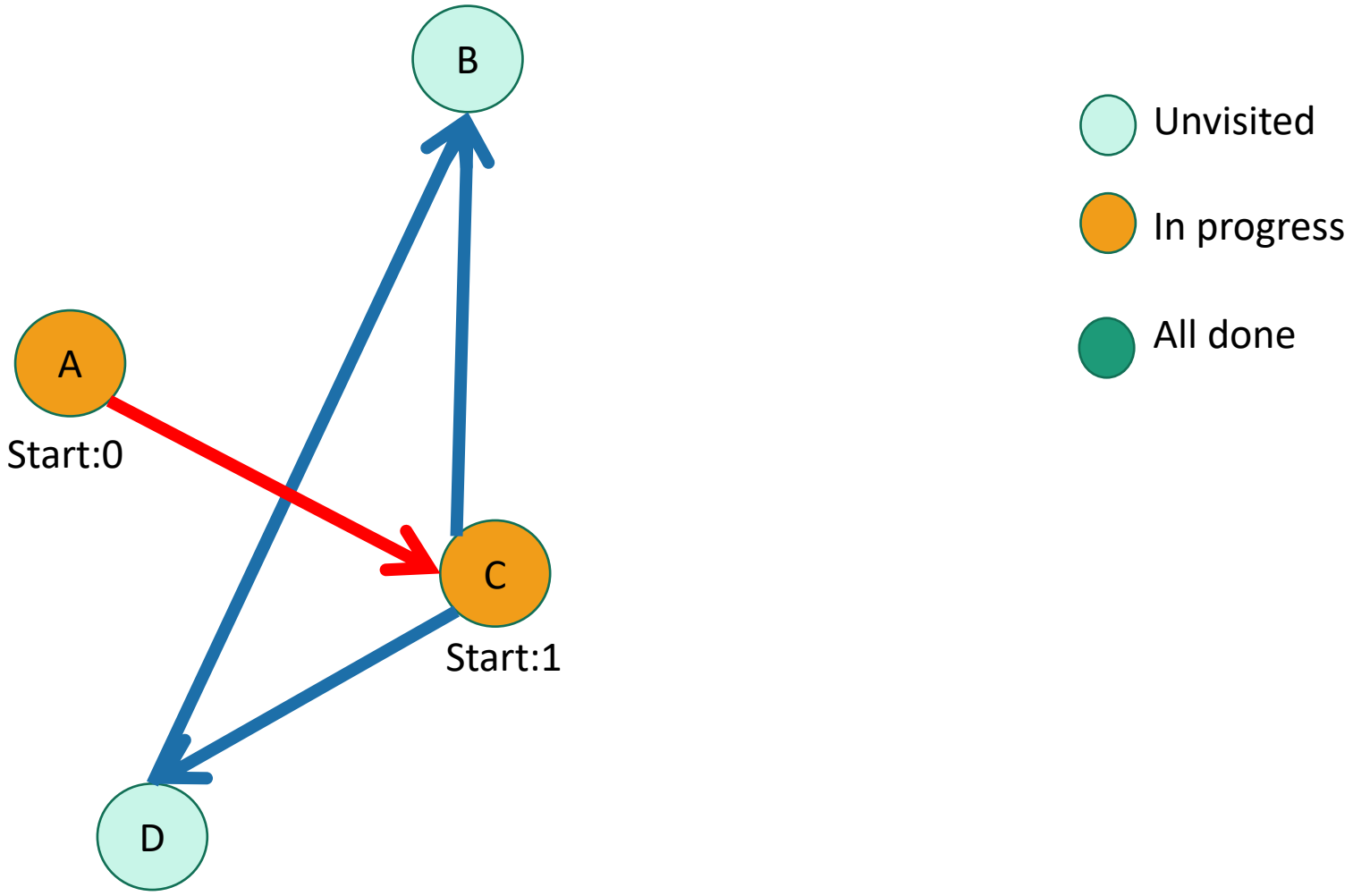
- DFS can help you solve the **TOPOLOGICAL SORTING PROBLEM**
  - That's the fancy name for the problem of finding an ordering that respects all the dependencies

- Thinking about the DFS tree is helpful.

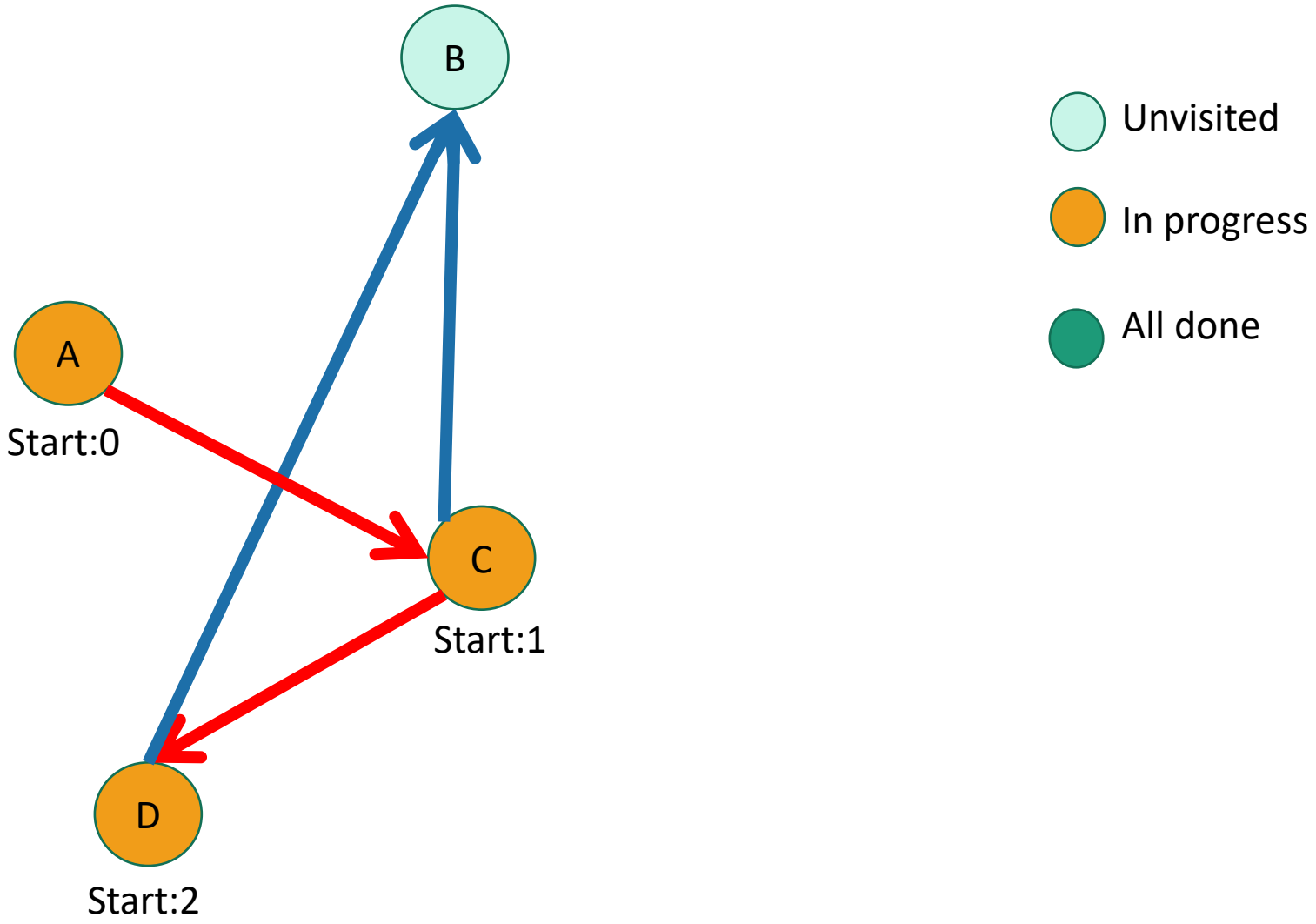
Example:



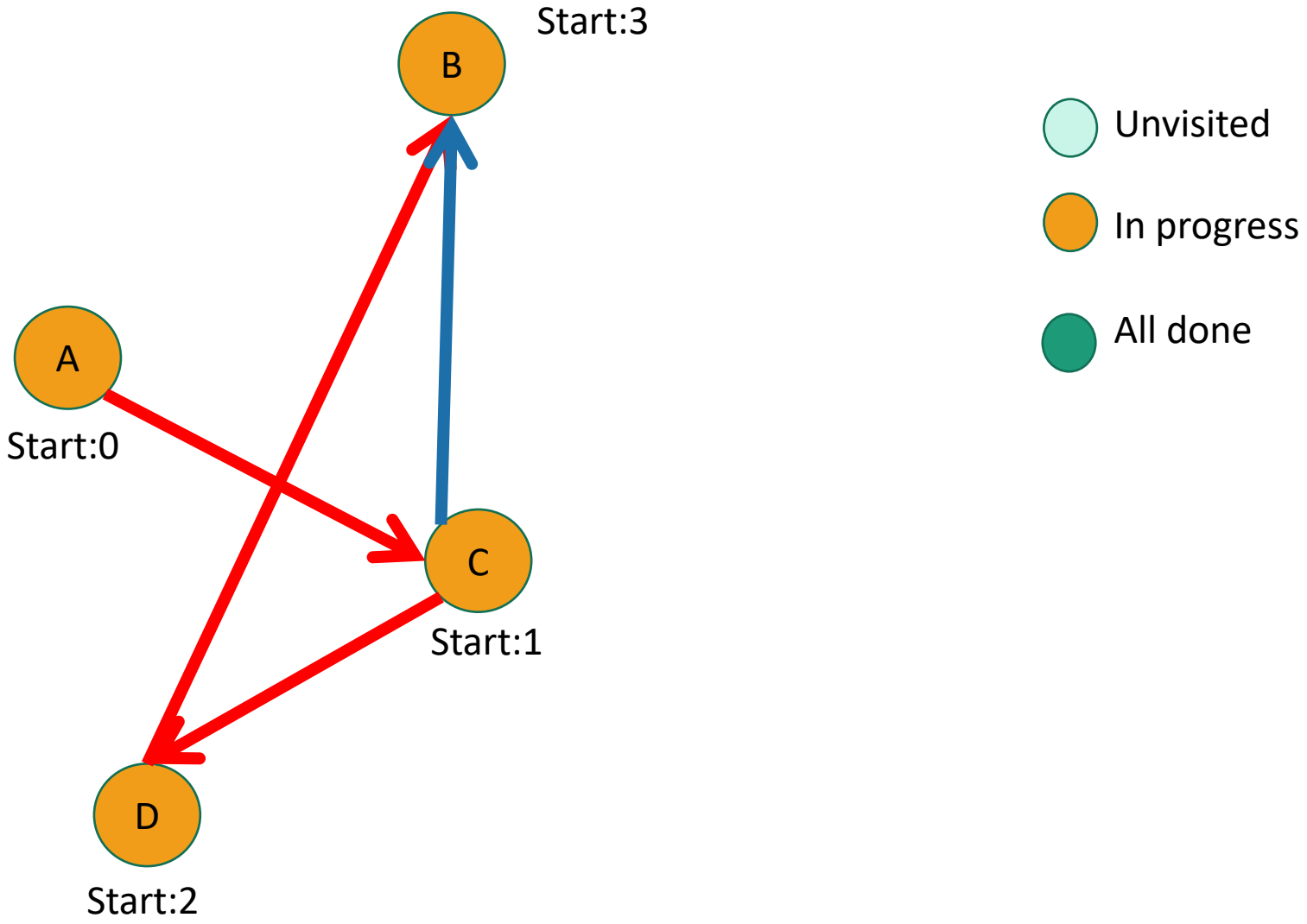
Example



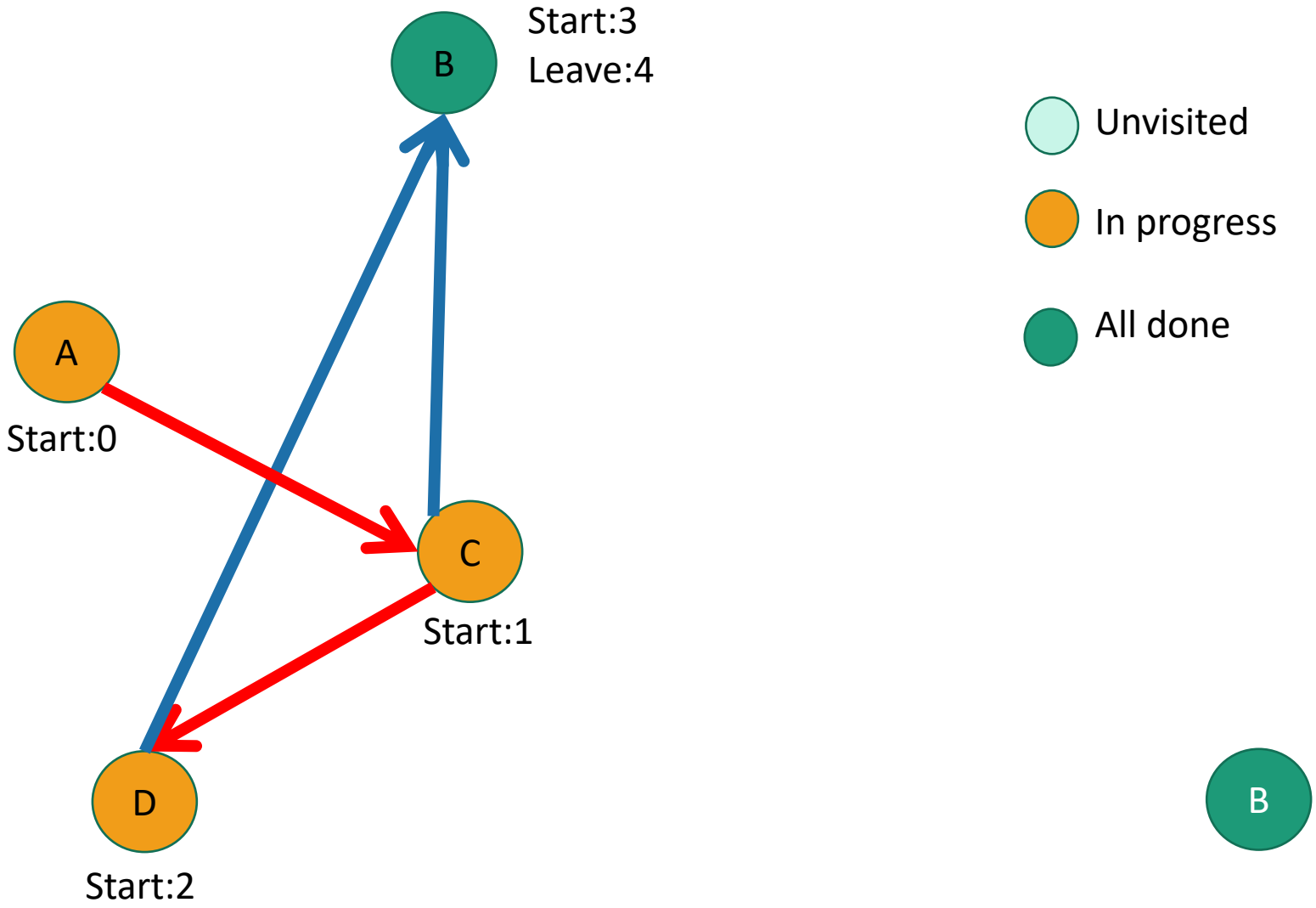
Example



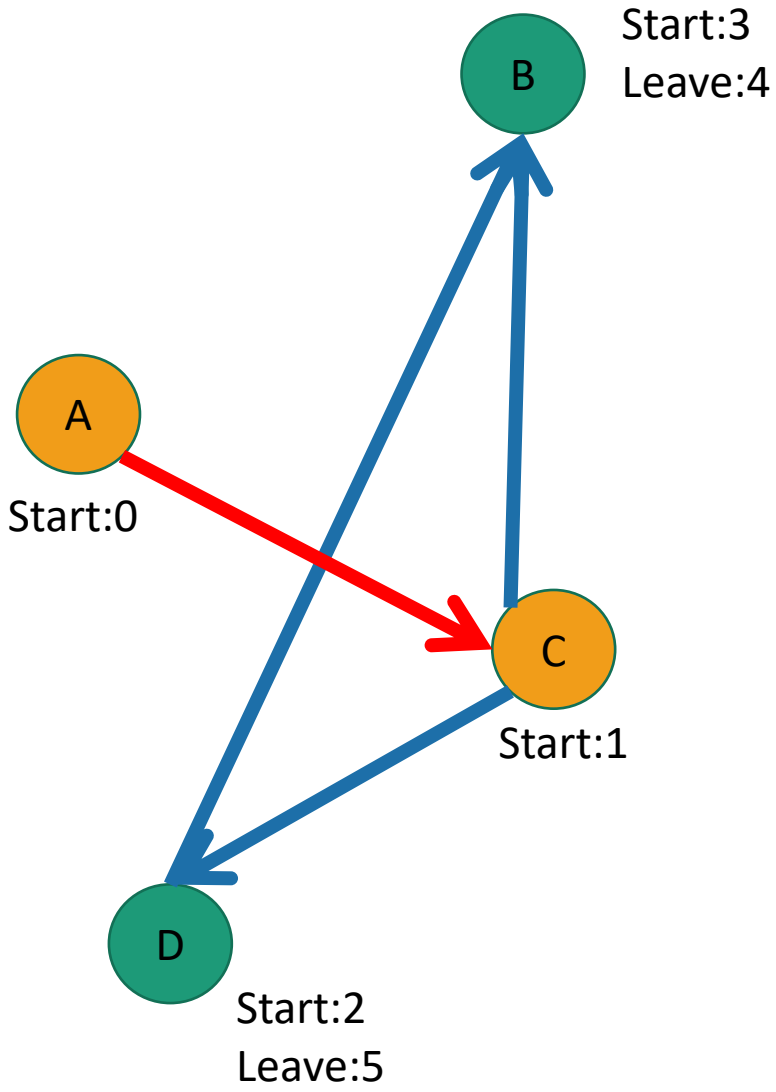
Example


Start:3
B
A
Start:0
C
Start:1
D
Start:2
Unvisited
In progress
All done

Example


B
Start:3
Leave:4
A
Start:0
C
Start:1
D
Start:2
Unvisited
In progress
All done
B

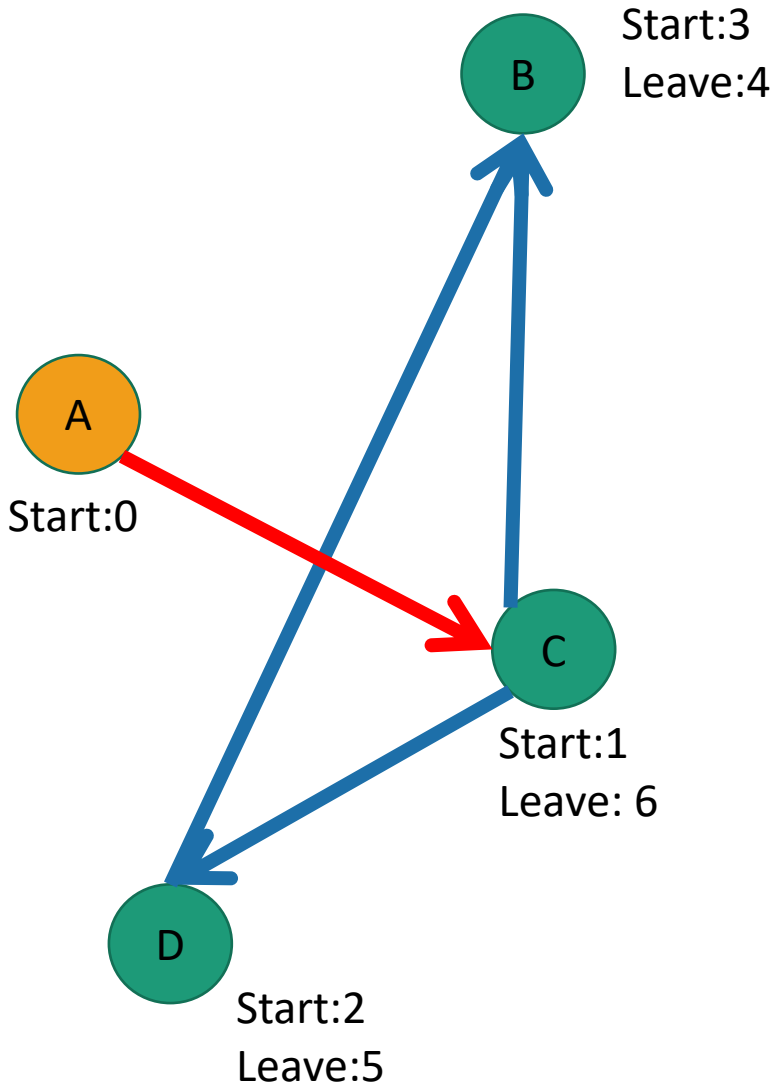
Example


B
Start:3
Leave:4
A
Start:0
C
Start:1
D
Start:2
Leave:5


Unvisited
In progress
All done


D
B

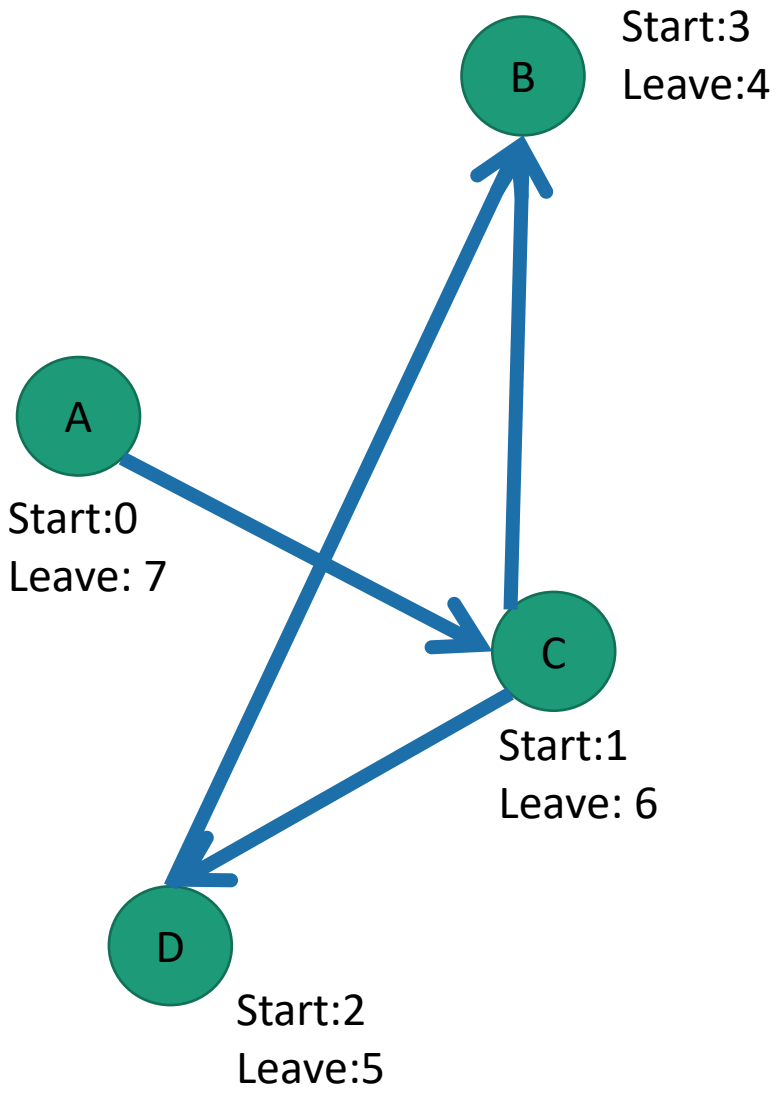
Example


B
Start:3
Leave:4
A
Start:0
C
Start:1
Leave: 6
D
Start:2
Leave:5


Unvisited
In progress
All done


C
D
B

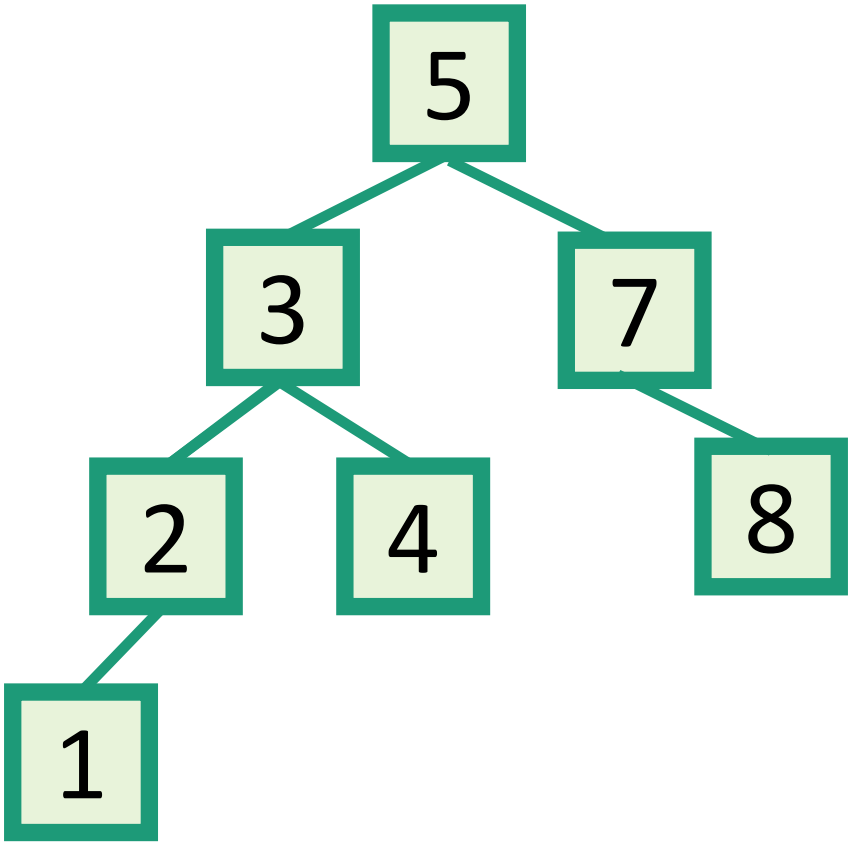
Example



Do them in this order:

Another use of DFS

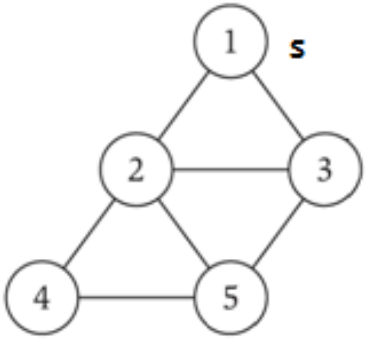
- In-order enumeration of binary search trees



Given a binary search tree, output all the nodes **in order.**

Instead of outputting a node when you are done with it, output it when you are done with the left child and before you begin the right child.

DFS Construction Example



DFTree

Explored FFFFF

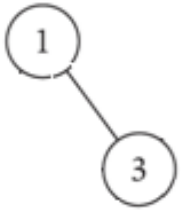
Stack nil,s

p(u),u

DFTree 

Explored TFFFF

Stack 1,3
1,2

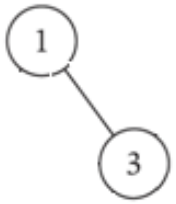
DFTree 

Explored TFTFF

Stack 3,1
3,2
3,5
1,2

DFS Construction Example

DFTree

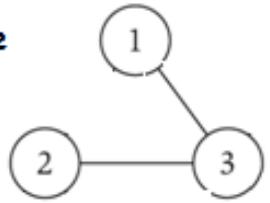


Explored TFTFF

Stack

3,2
3,5
1,2

DFTree

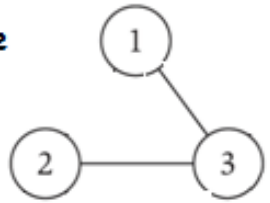


Explored TTTFF

Stack

2,1
2,3
2,4
2,5
3,5
1,2

DFTree

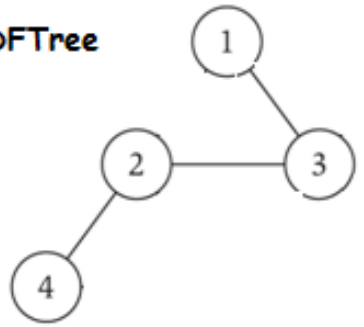


Explored TTTFF

Stack

2,4
2,5
3,5
1,2

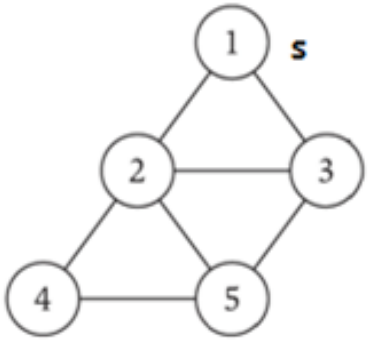
DFTree



Explored TTTTF

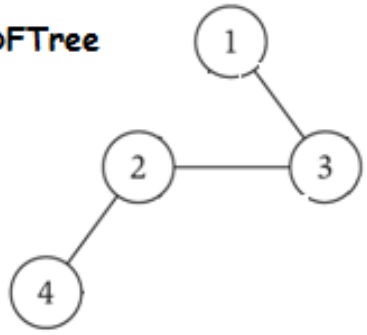
Stack

4,2
4,5
2,5
3,5
1,2



DFS Construction Example

DFTree

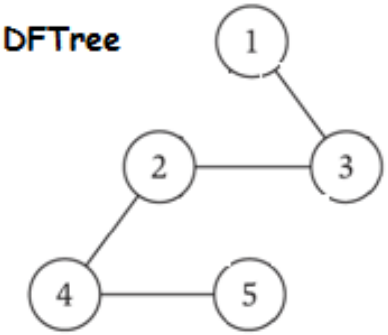


Explored TTTTF

Stack

4,5
2,5
3,5
1,2

DFTree

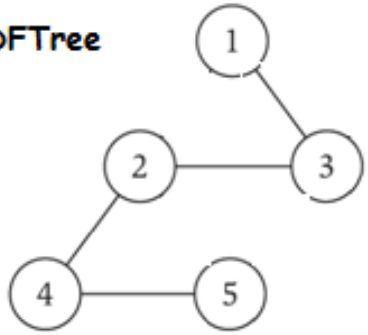


Explored TTTTT

Stack

5,2
5,3
5,4
2,5
3,5
1,2

DFTree

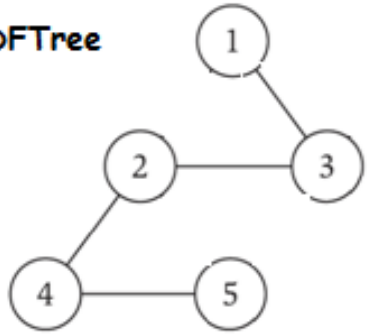


Explored TTTTT

Stack

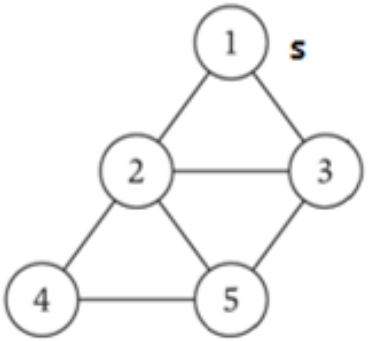
5,3
5,4
2,5
3,5
1,2

DFTree



Explored TTTTT

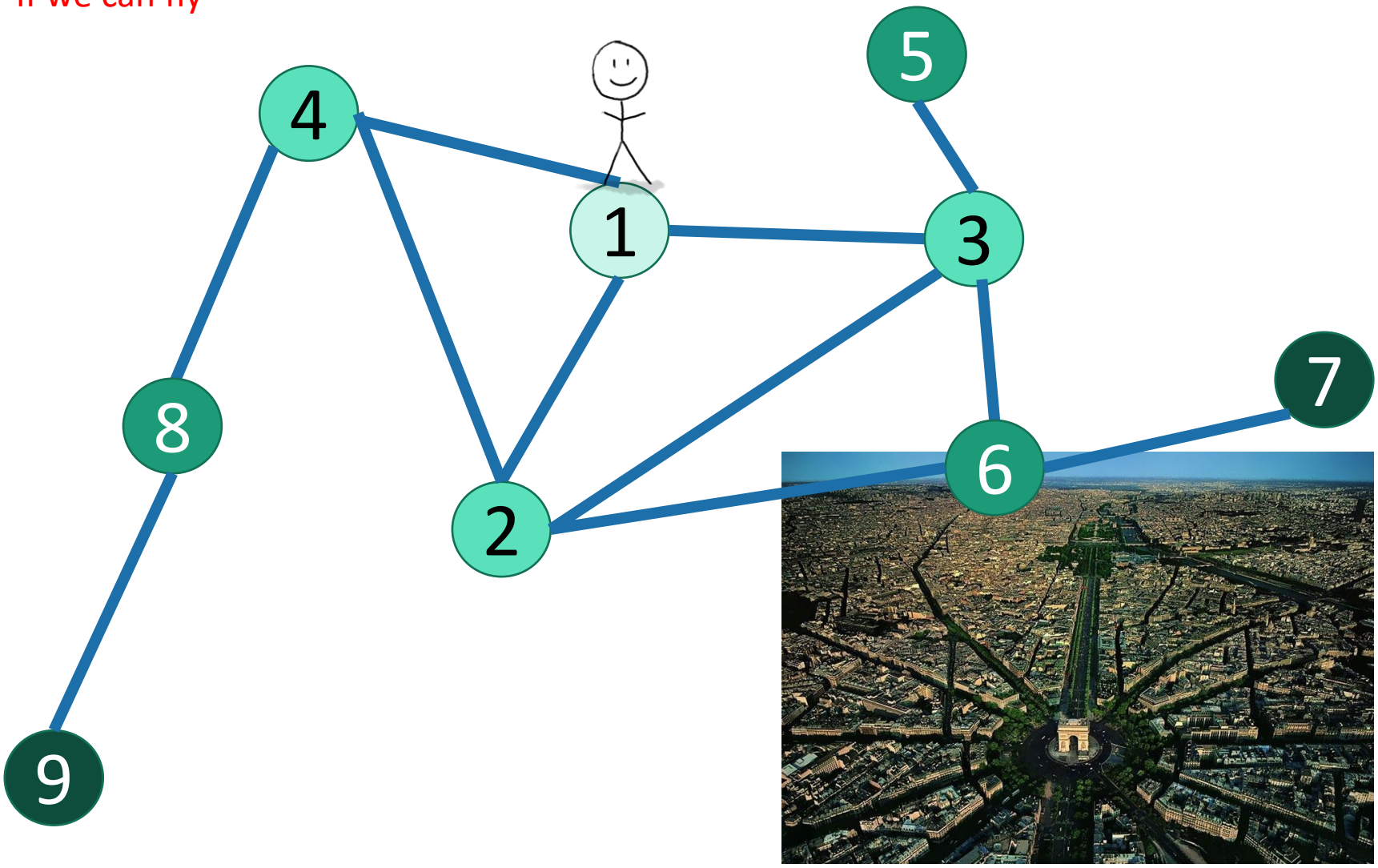
Stack



Part 2: breadth-first search

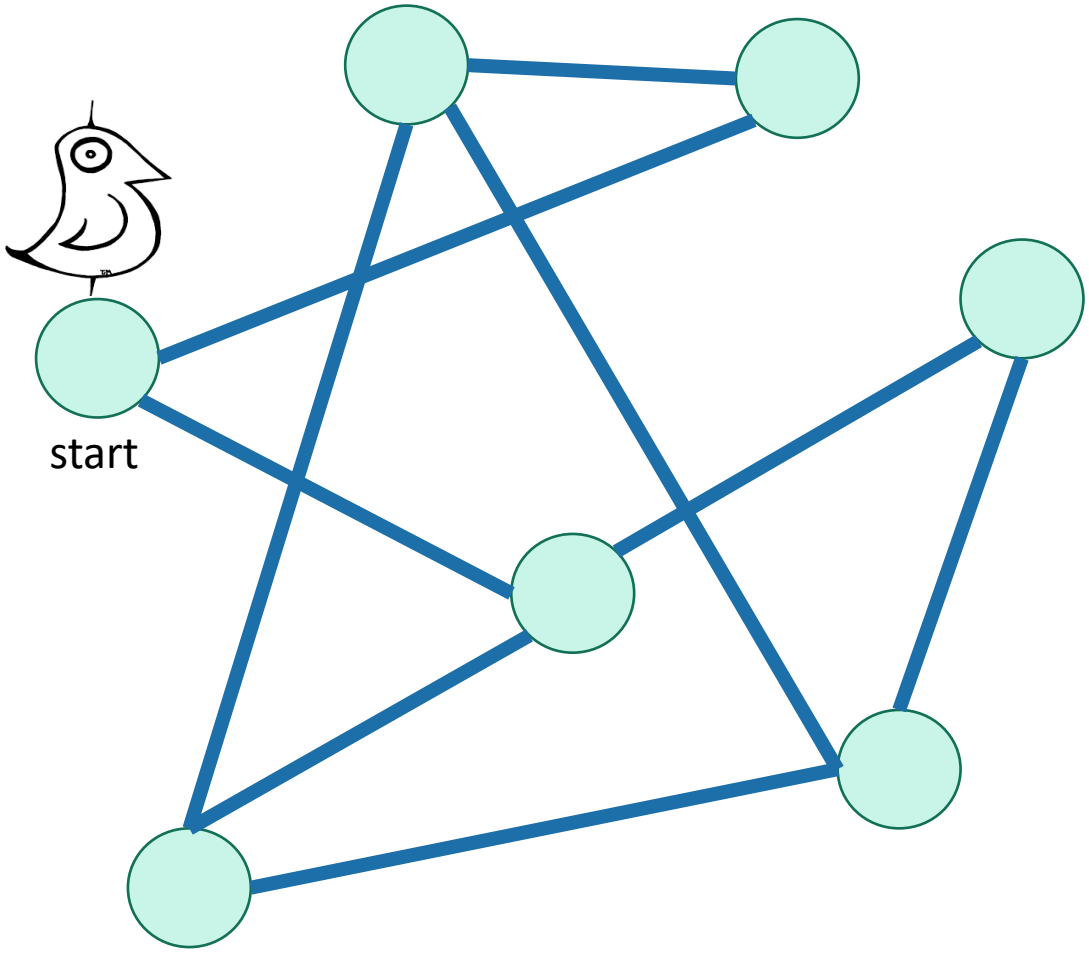
How do we explore a graph?

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

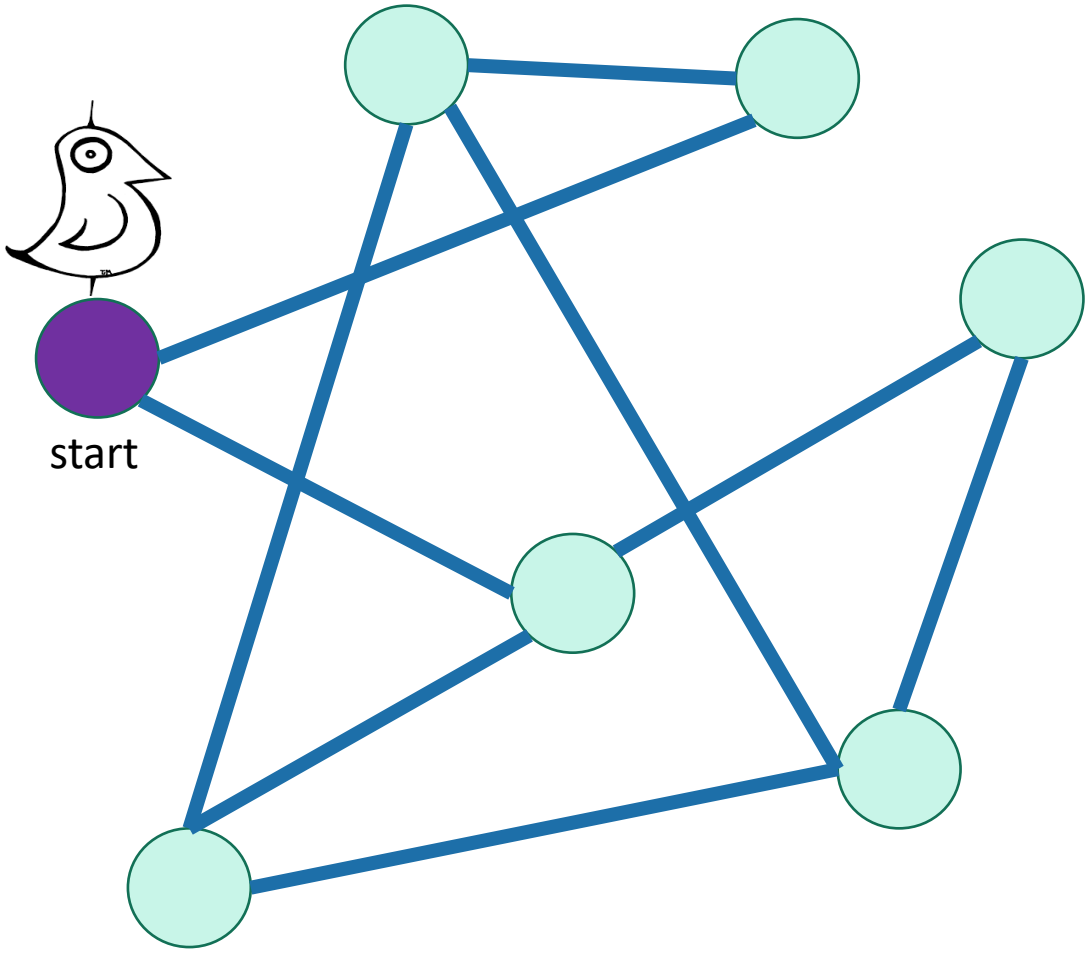
Can reach there in one step

Can reach there in two steps

Can reach there in three steps

Breadth-First Search

Exploring the world with a bird's-eye view



Not been there yet

Can reach there in zero steps

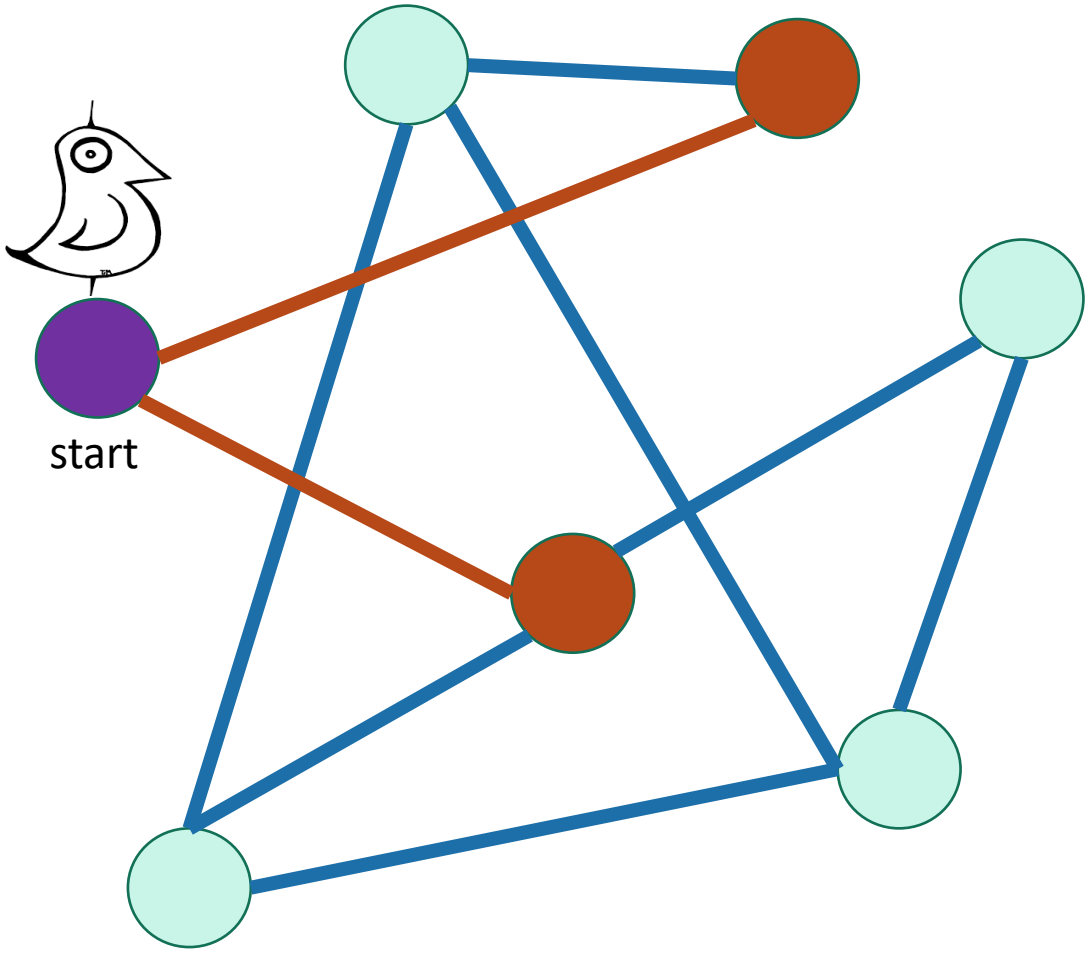
Can reach there in one step

Can reach there in two steps

Can reach there in three steps

Breadth-First Search

Exploring the world with a bird's-eye view



Not been there yet

Can reach there in zero steps

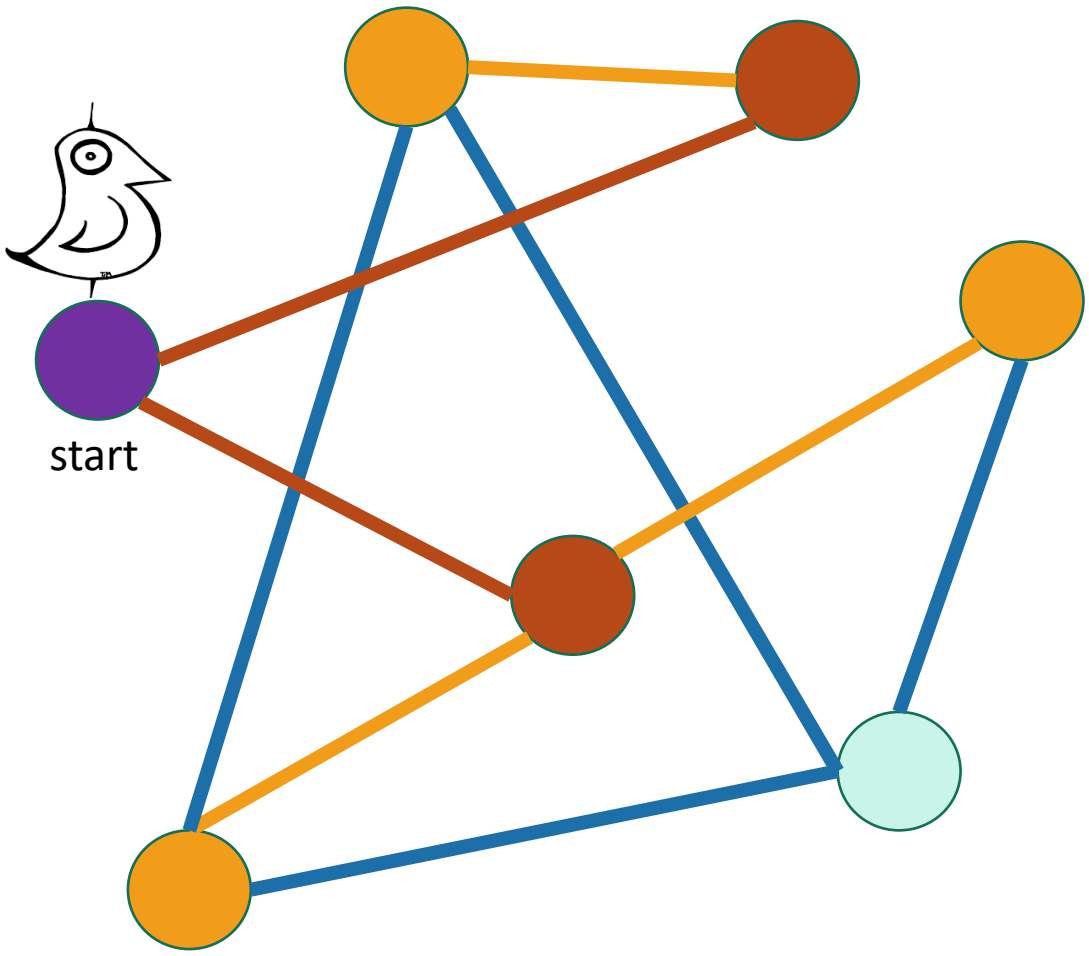
Can reach there in one step

Can reach there in two steps

Can reach there in three steps

Breadth-First Search

Exploring the world with a bird's-eye view



Not been there yet

Can reach there in zero steps

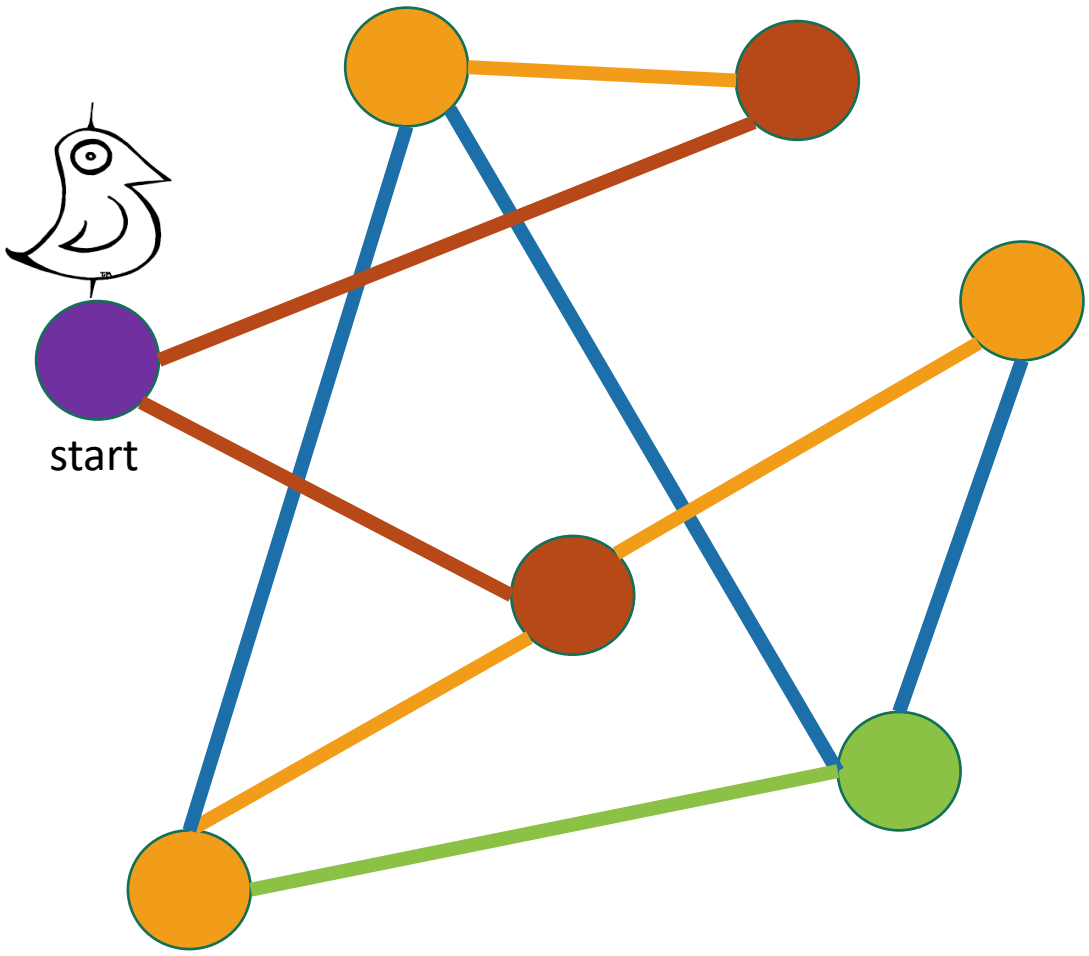
Can reach there in one step

Can reach there in two steps

Can reach there in three steps

Breadth-First Search

Exploring the world with a bird's-eye view



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!

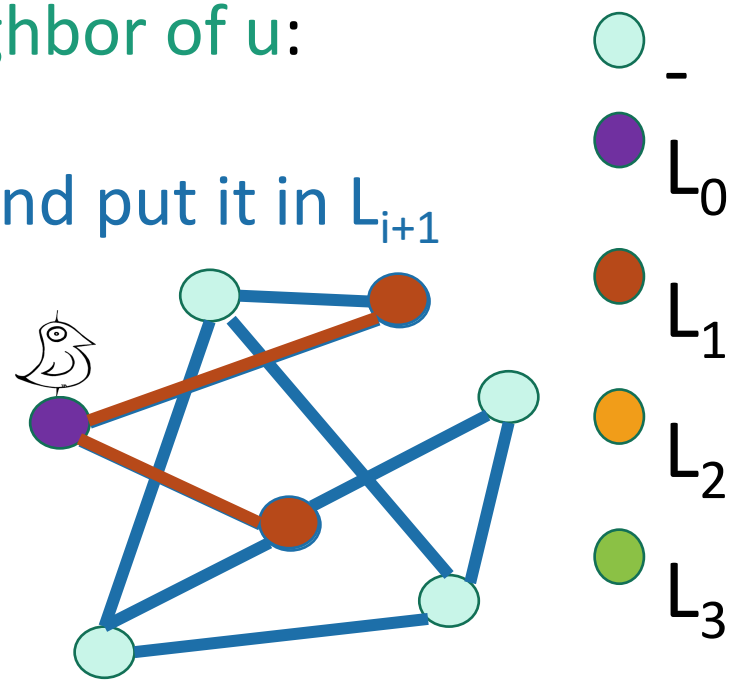
Breadth-First Search

Exploring the world with pseudocode

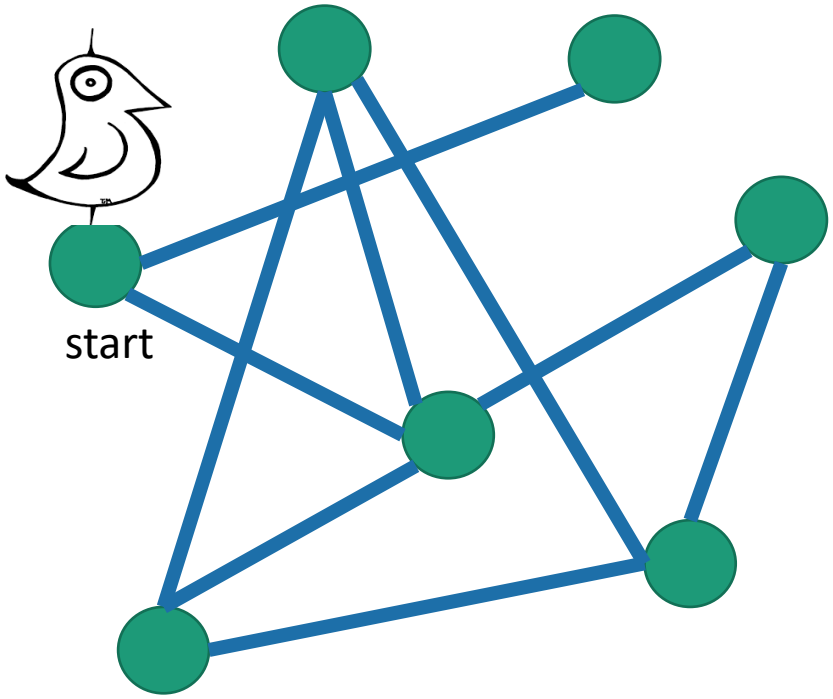
- Set L_i = [] for i=1,…,n
- L_0 = {w}, where w is the start node
- **For** i = 0, …, n-1:
  - **For** u in L_i:
    - **For** each v which is a neighbor of u:
      - **If** v isn't yet visited:
        - mark v as visited, and put it in L_{i+1}

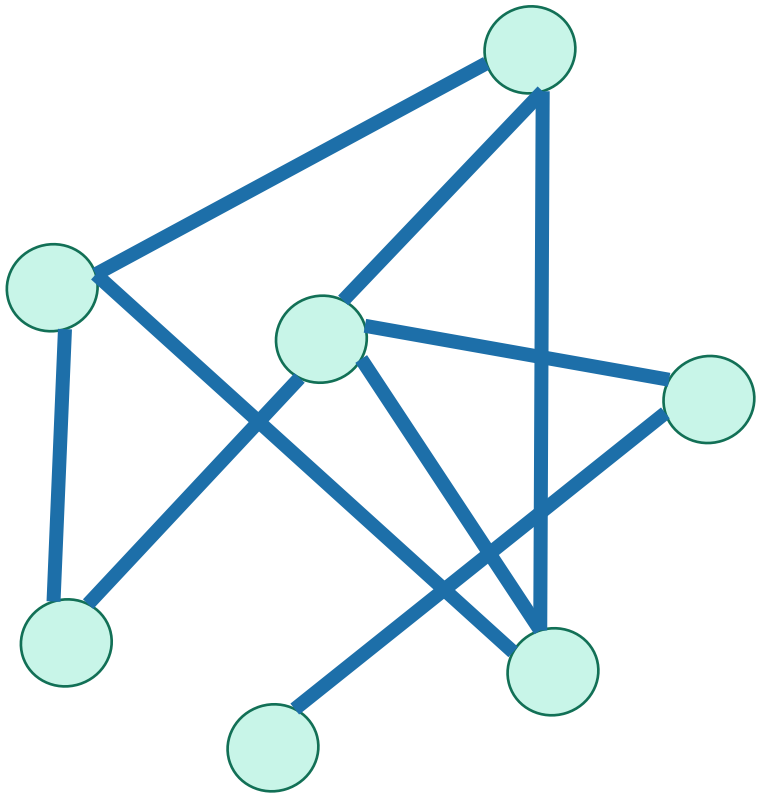
L_i is the set of nodes we can reach in i steps from w

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

To explore the whole thing

- Explore the connected components one-by-one.
- Same argument as DFS: running time is

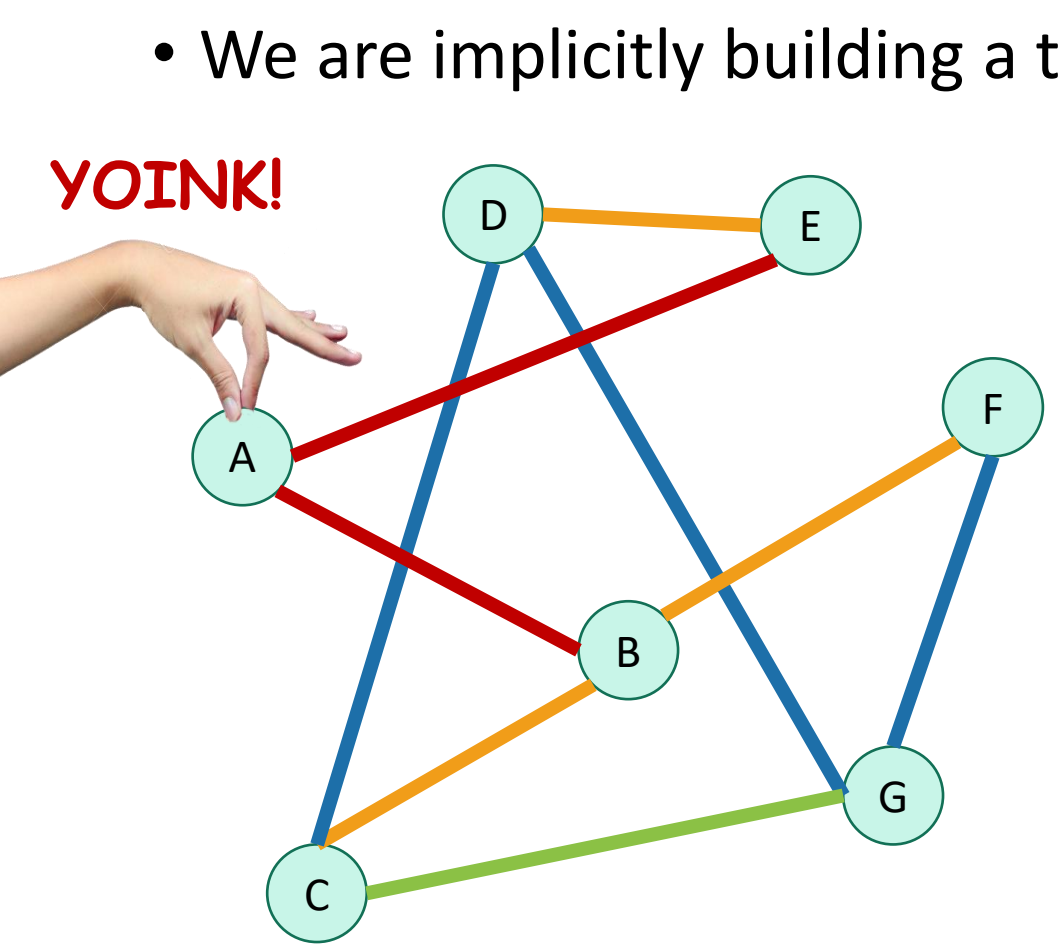
$$O(n + m)$$

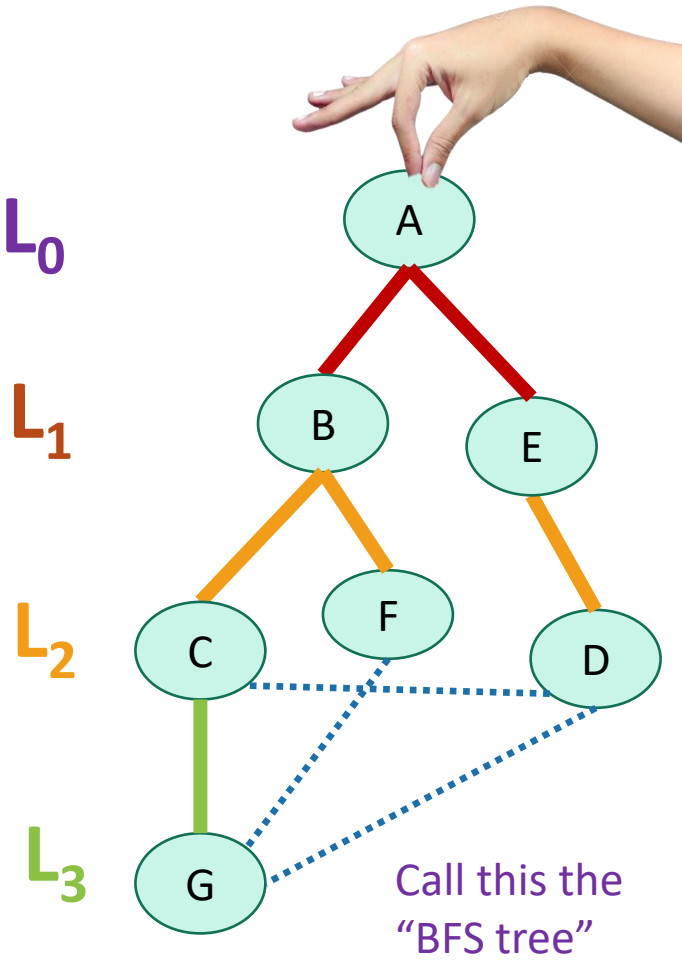
Verify these!

- Like DFS, BFS also works fine on directed graphs.

Why is it called breadth-first?

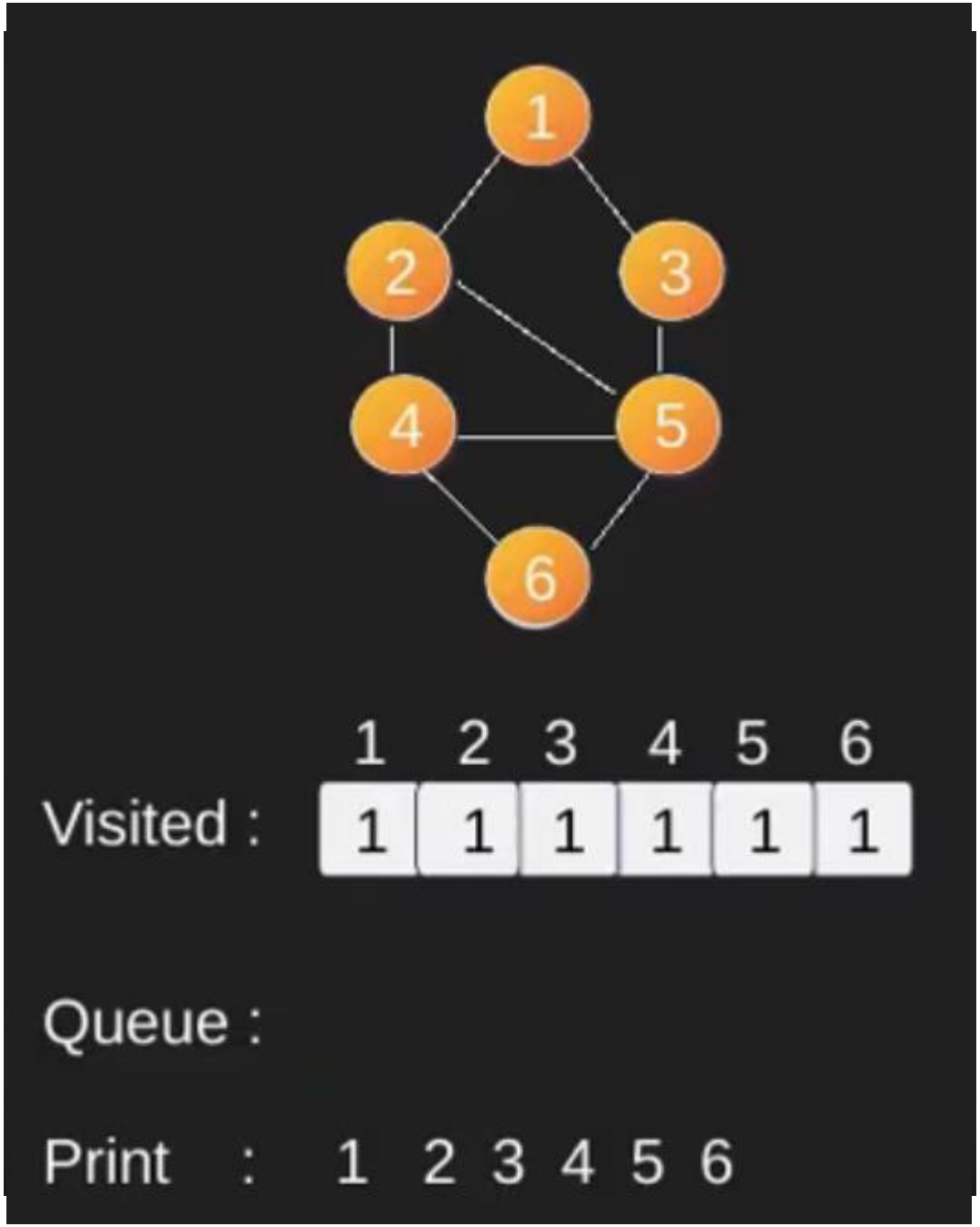
- We are implicitly building a tree:





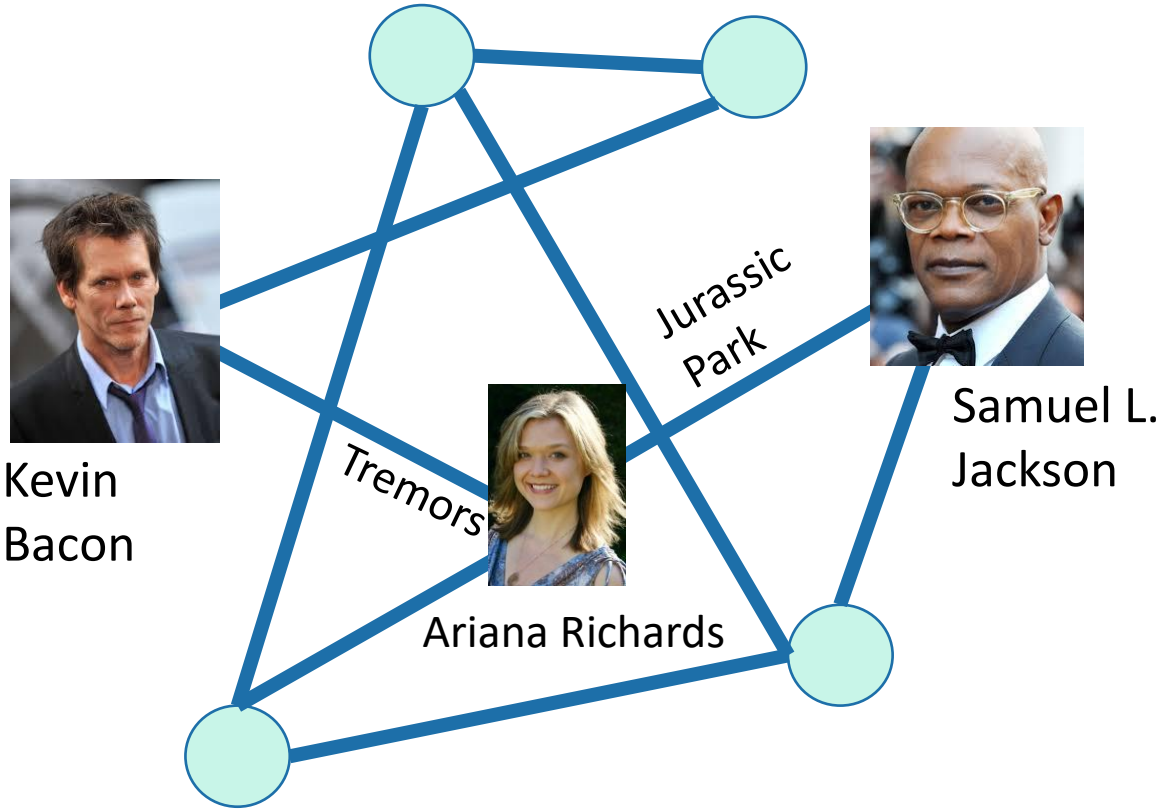
- And first we go as broadly as we can.

Breadth First Search- Example

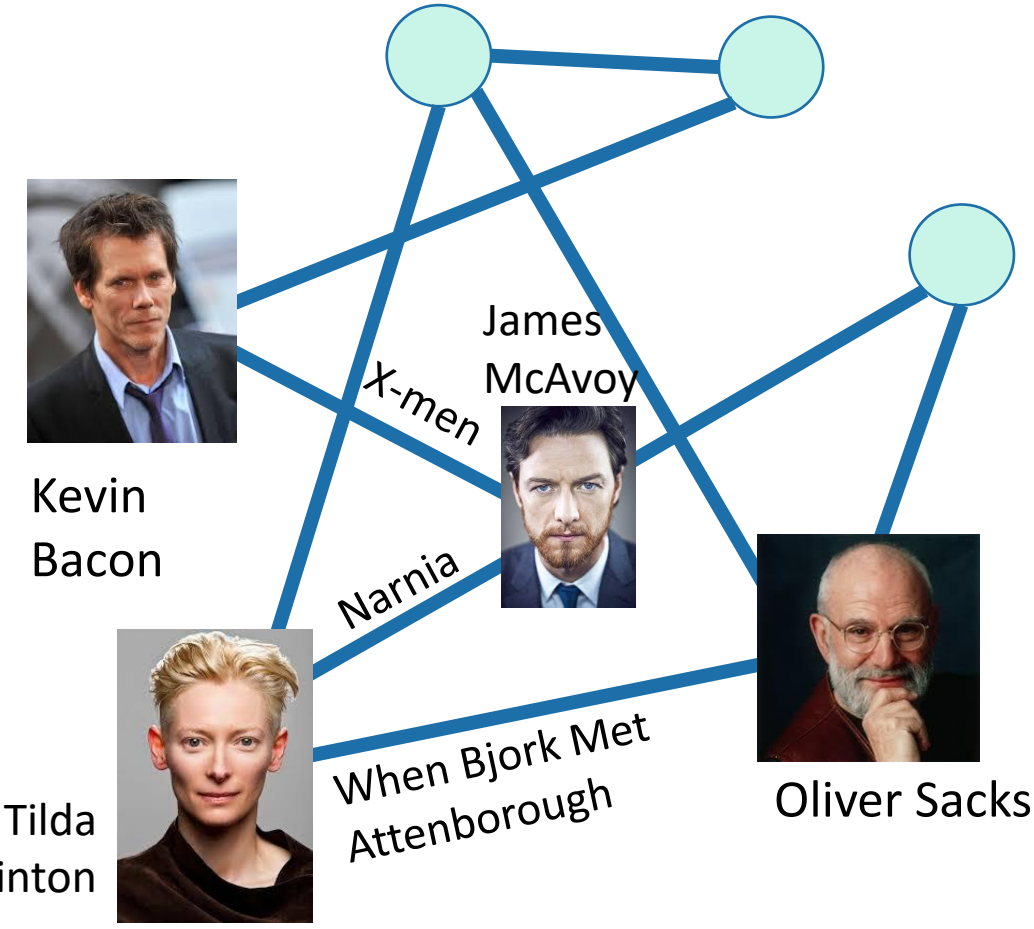


Bacon number

- What Samuel L. Jackson's Bacon number?



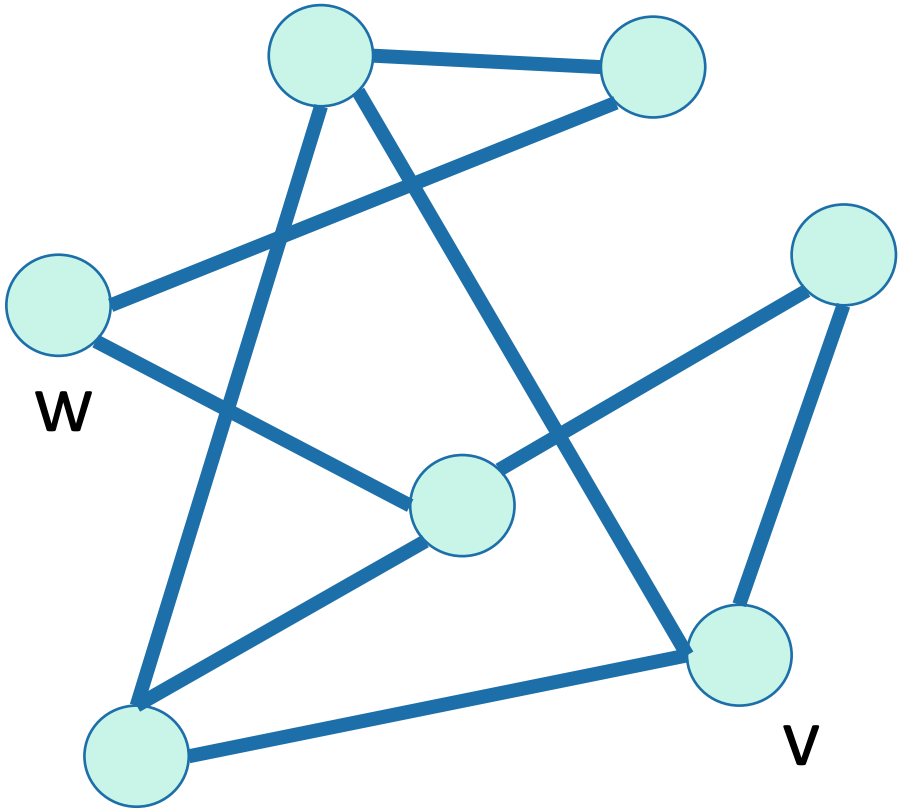
Bacon number



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

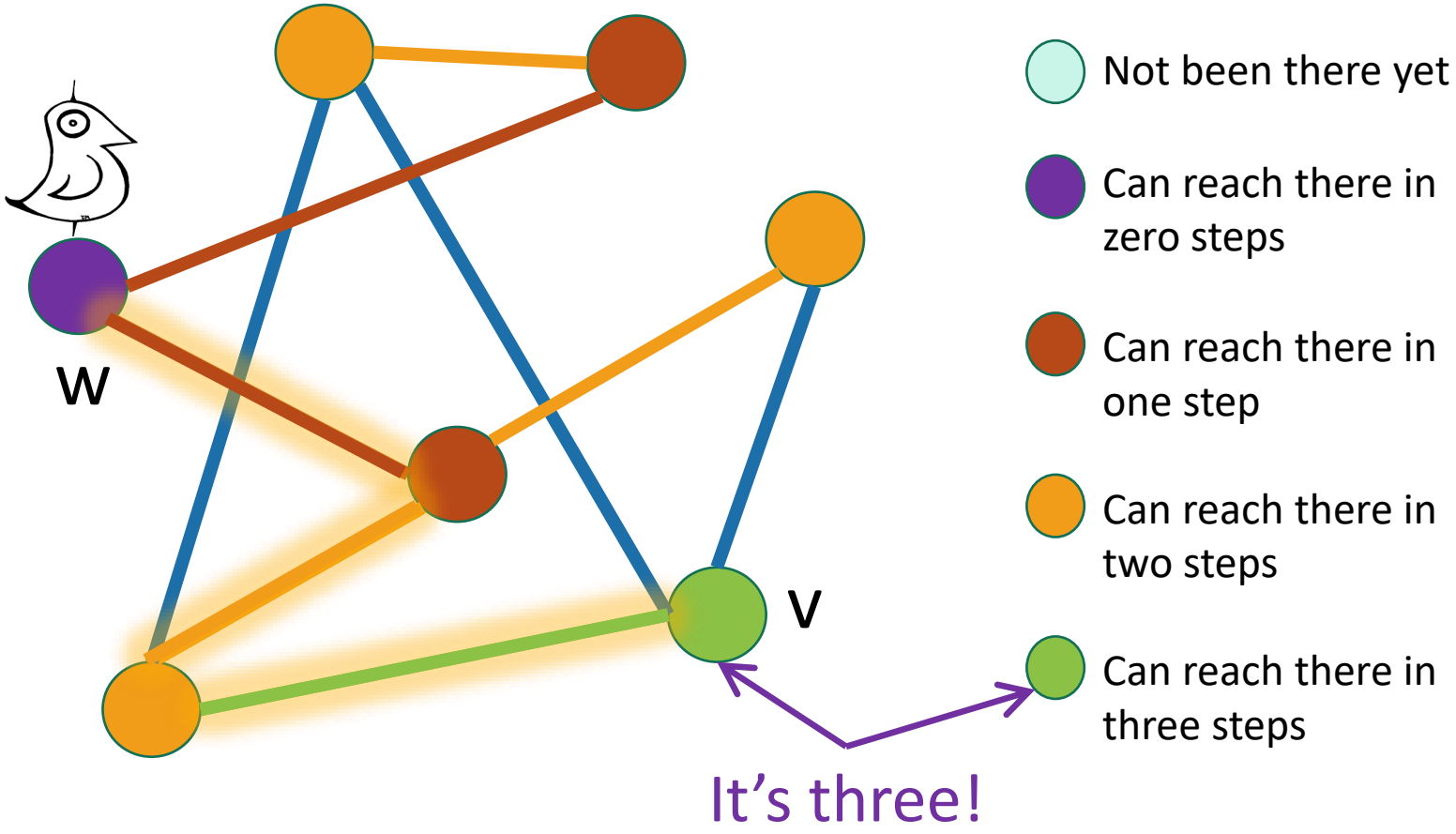
Application: shortest path

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



Application: shortest path

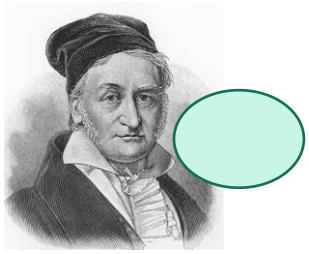
- How long is the shortest path between w and v?



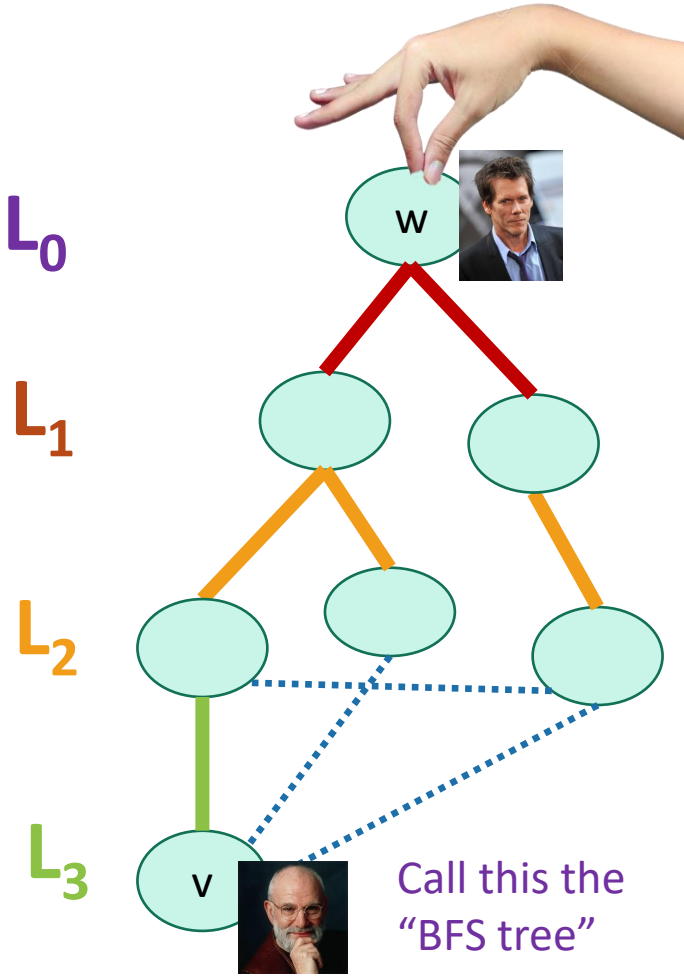
To find the distance between w and all other vertices v

The **distance** between two vertices is the length of the shortest path between them.

- 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.

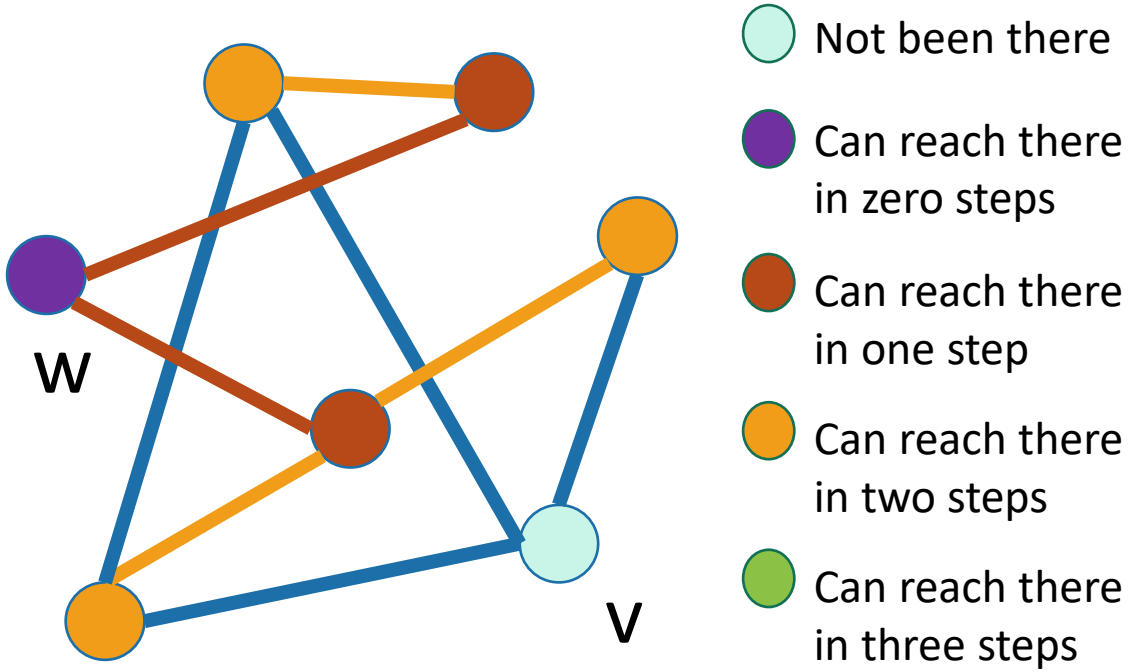


Gauss has no Bacon number



Call this the "BFS tree"

Proof idea

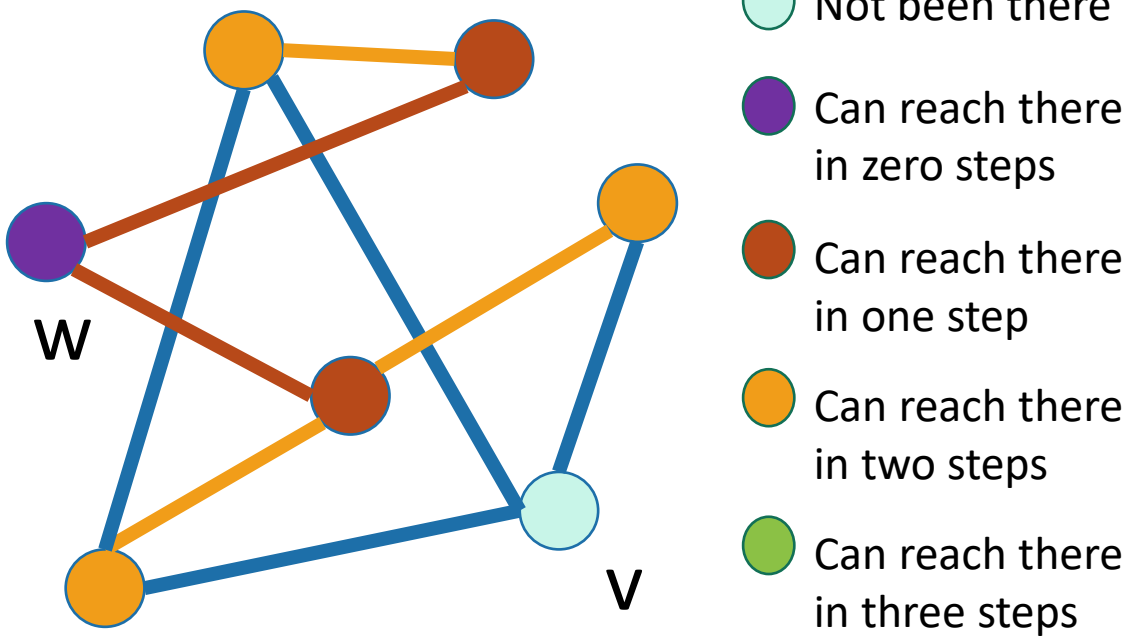

w
v
Not been there
Can reach there in zero steps
Can reach there in one step
Can reach there in two steps
Can reach there in three steps

Proof idea

Just the idea…see CLRS for details!

- Suppose by **induction** it's true for vertices in L_0, L_1, L_2
  - For all $i < 3$, the vertices in L_i have distance i from v.
- **Want to show**: it's true for vertices of distance 3 also.
  - aka, the shortest path between w and v has length 3.

- **Well, it has distance at most 3**
  - Since we just found a path of length 3
- **And it has distance at least 3**
  - Since if it had distance $i < 3$, it would have been in L_i.



n vertices and m edges.

What did we just learn?

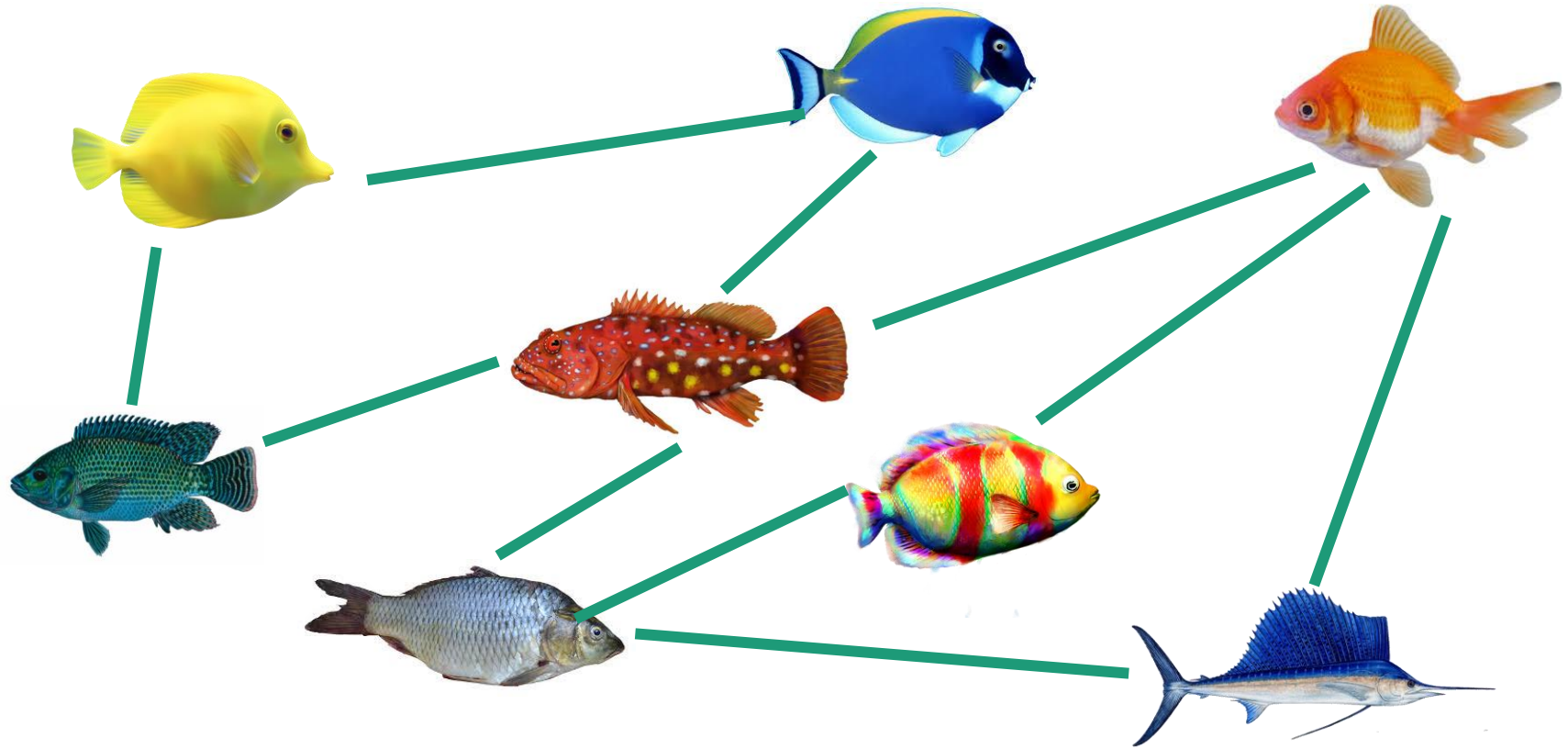
- The BFS tree is useful for computing distances between pairs of vertices.
- We can find the shortest path between u and v in time O(m).

The BSF tree is also helpful for:

- Testing if a graph is bipartite or not.

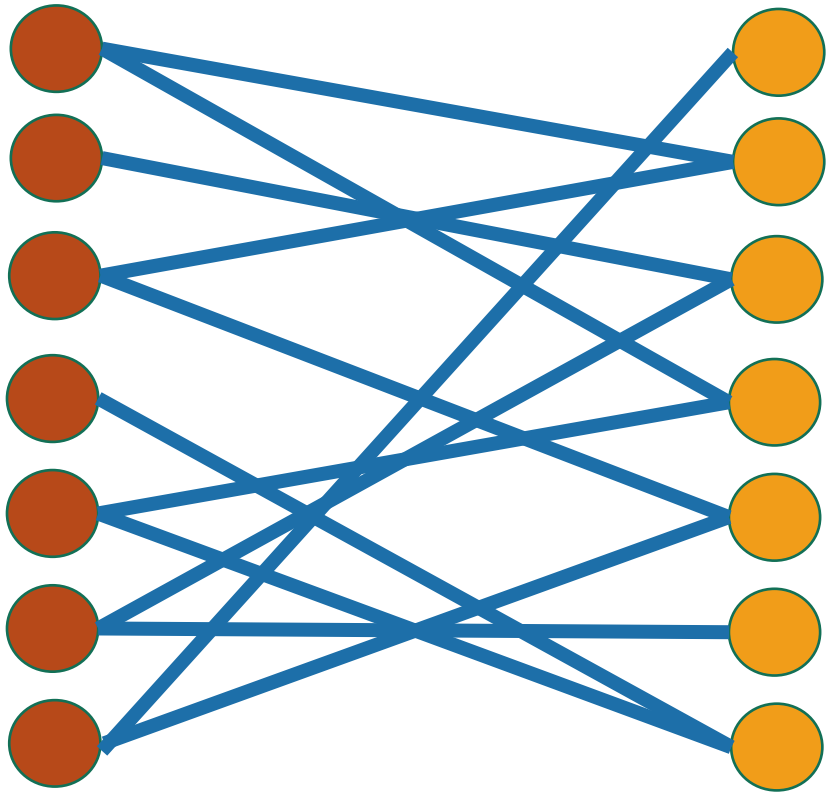
Fish Example

- Some pairs of species will fight if put in the same tank.
- You only have two tanks.
- Connected fish will fight.



Application: testing if a graph is bipartite

- Bipartite means it 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

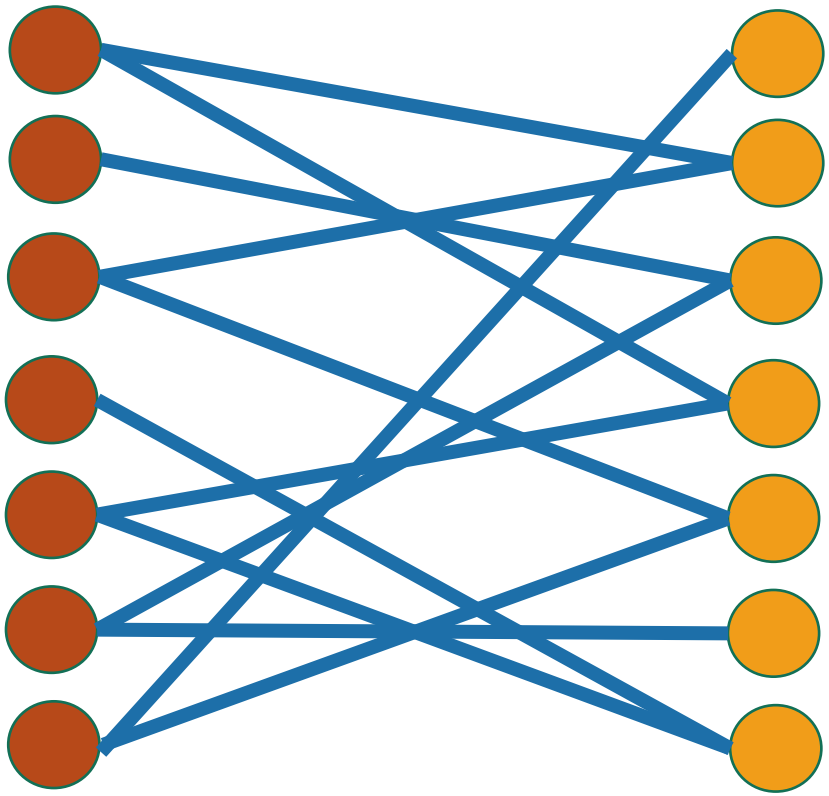
Example:

are students

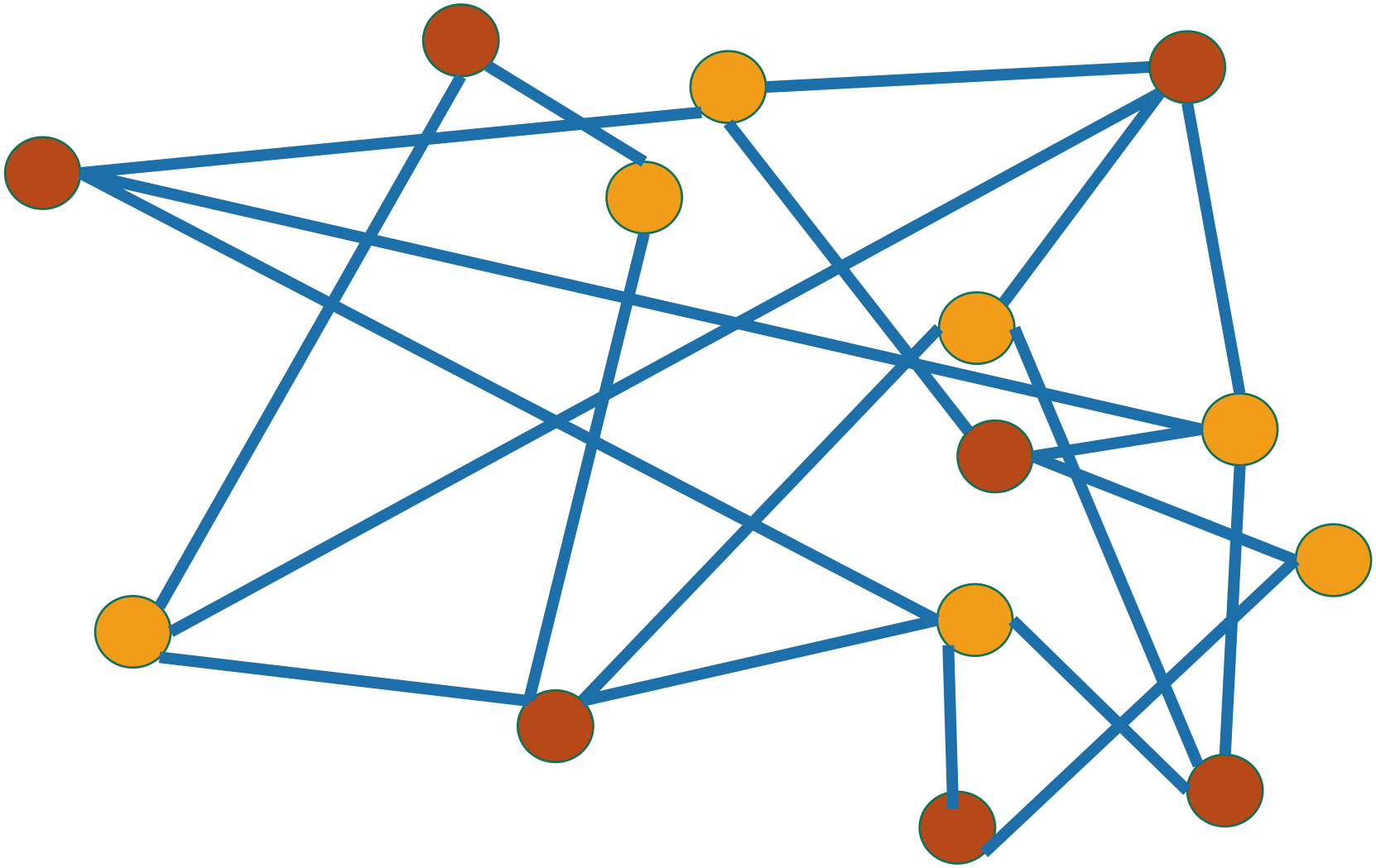
are classes

if the student is enrolled in the class

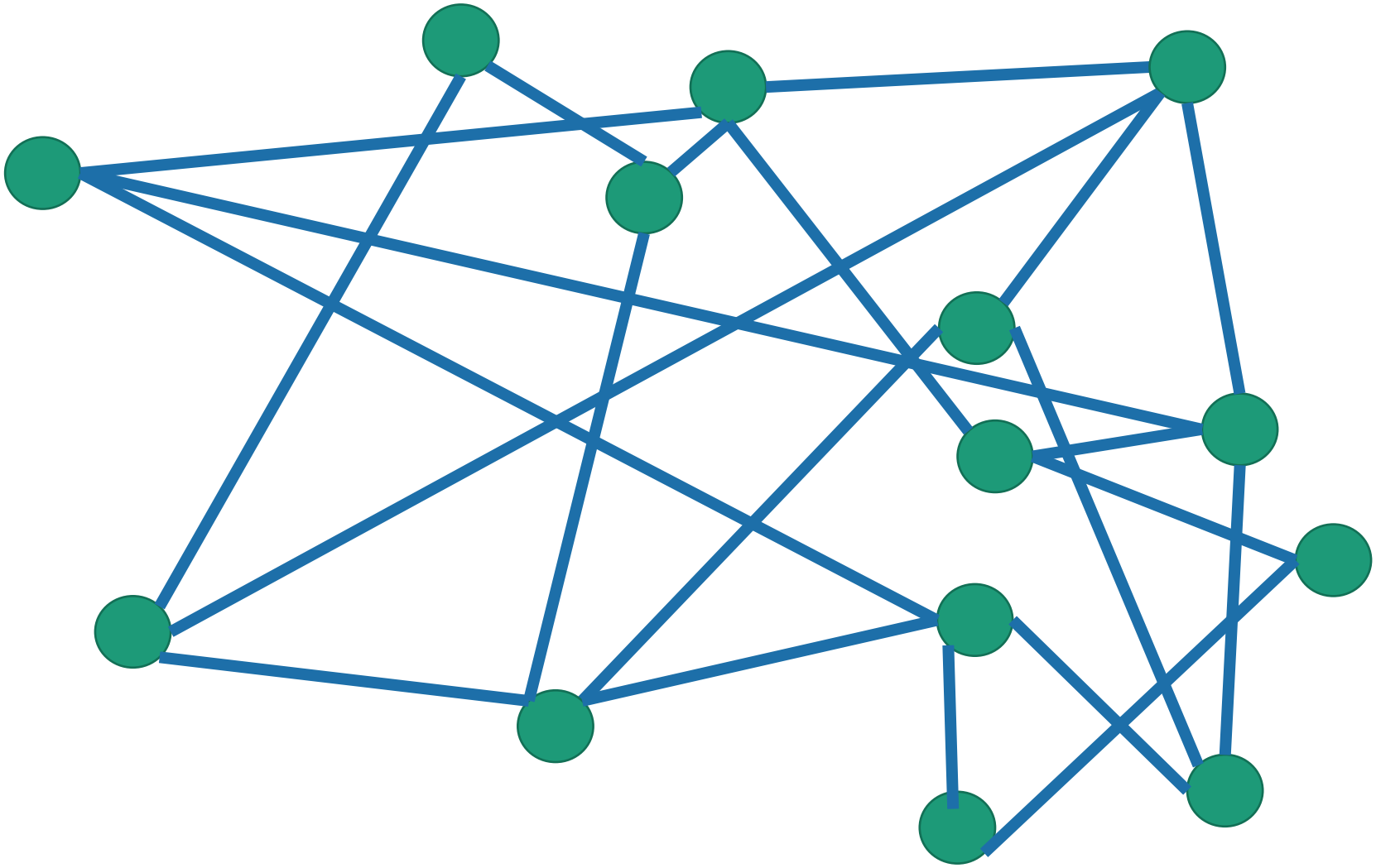
Is this graph bipartite?



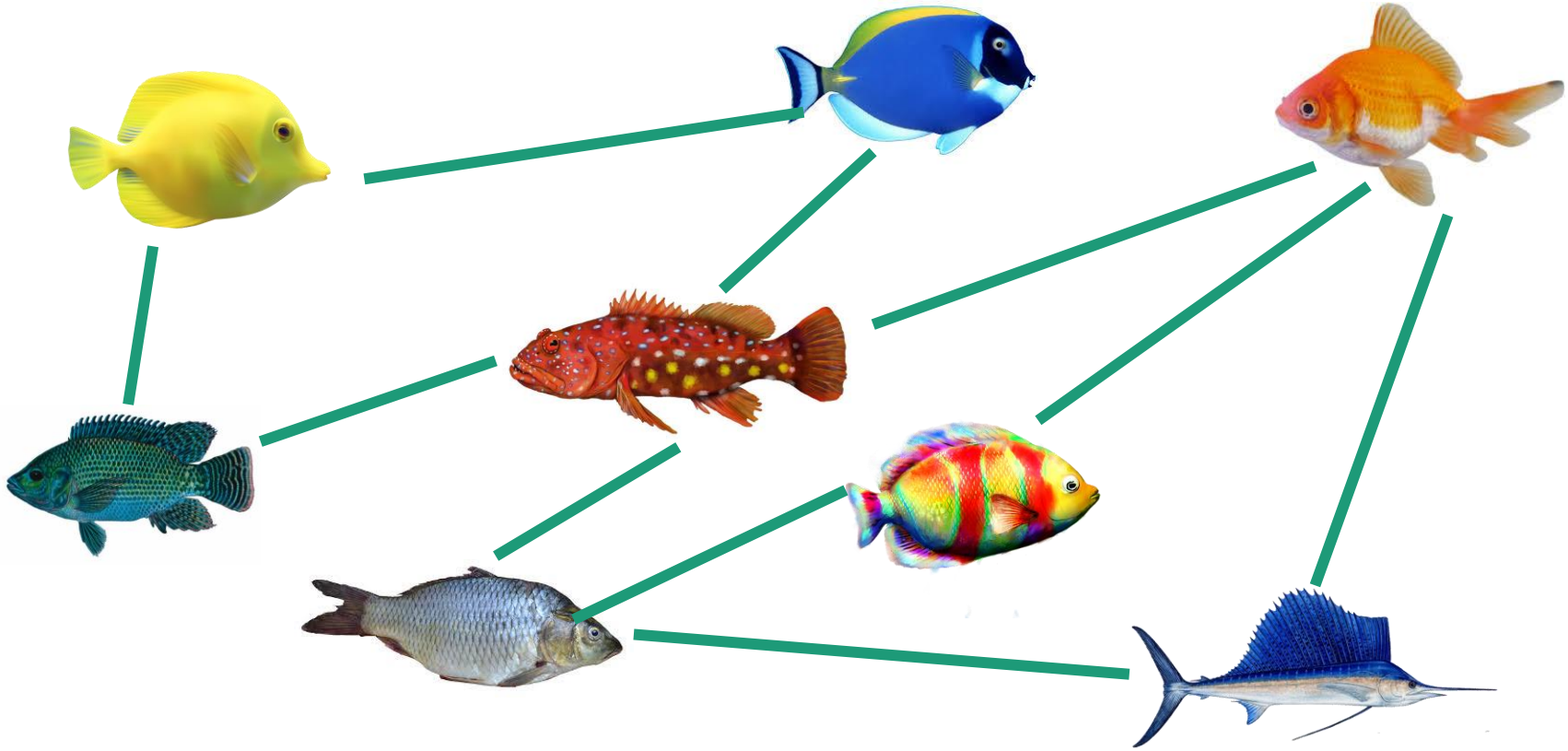
How about this one?



How about this one?

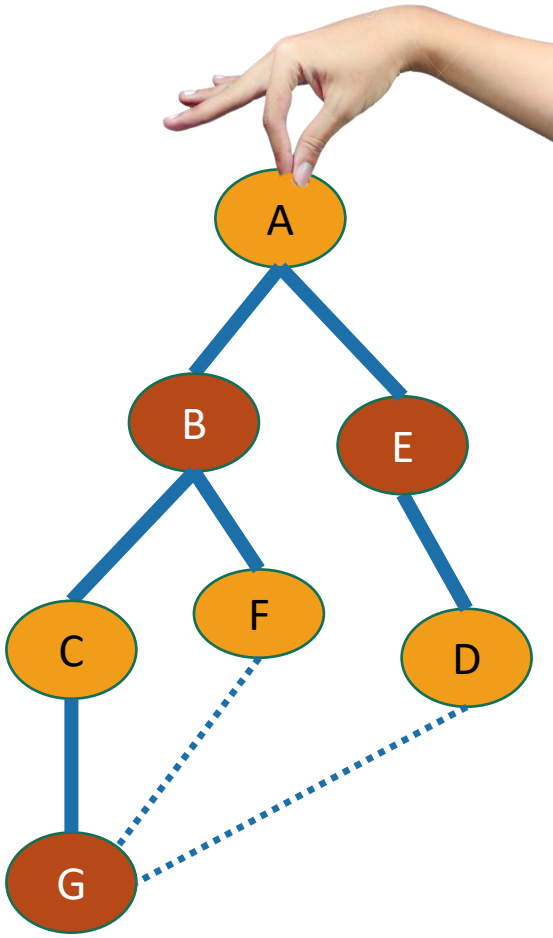


This one?



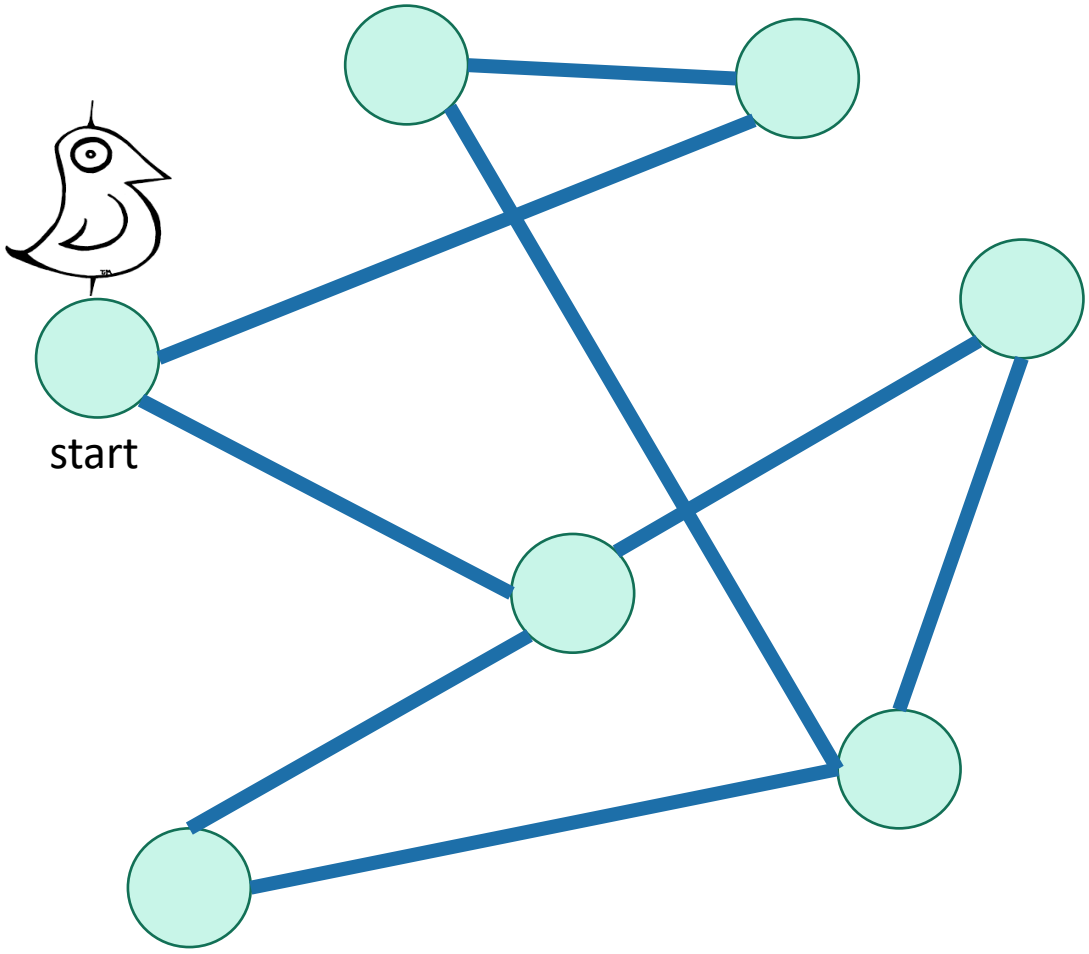
Solution using BFS

- 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

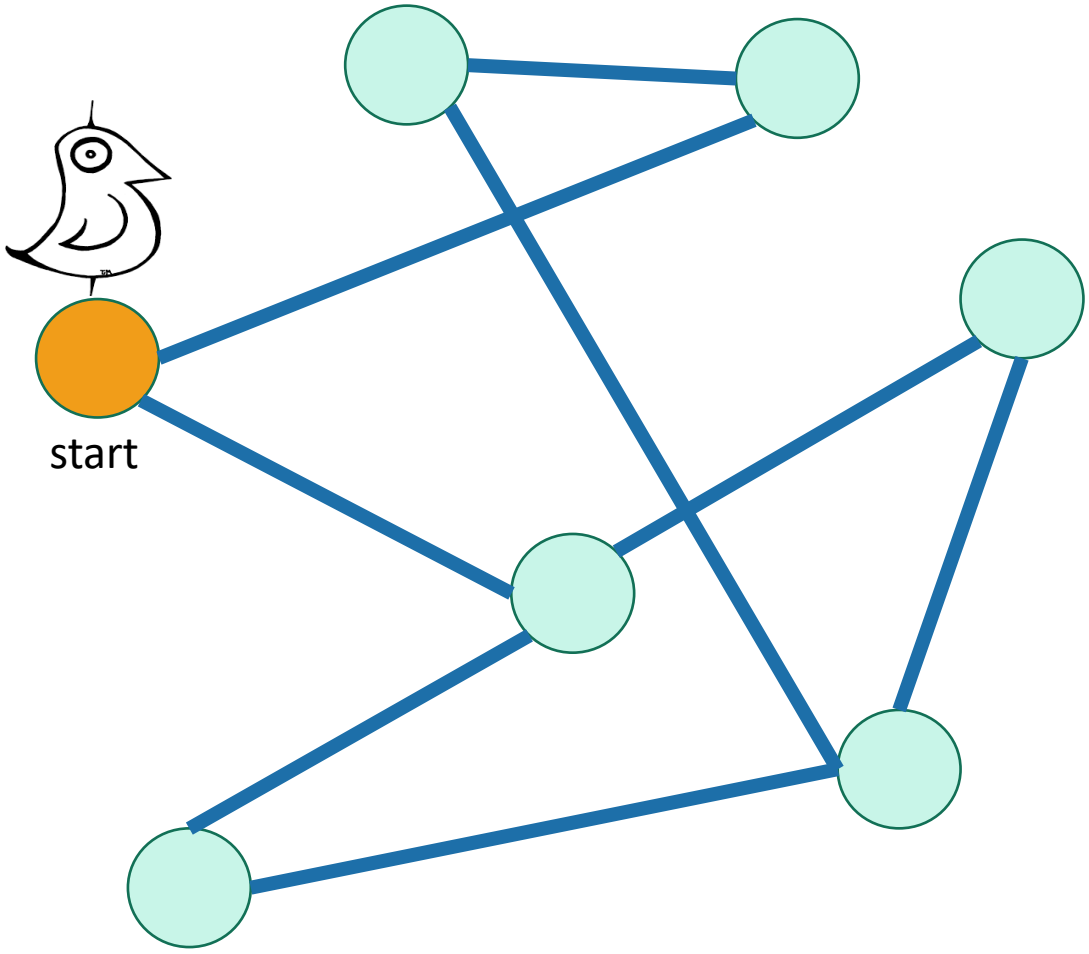
Can reach there in one step

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

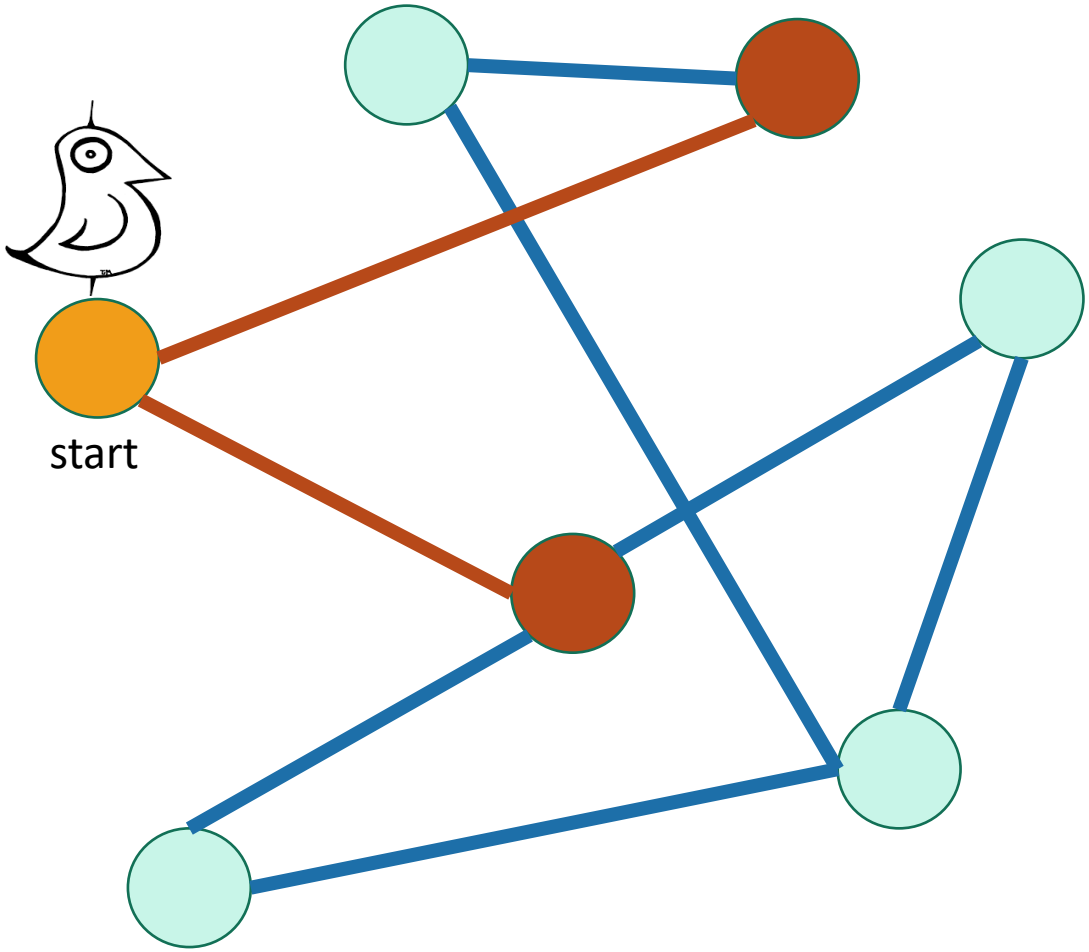
Can reach there in one step

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

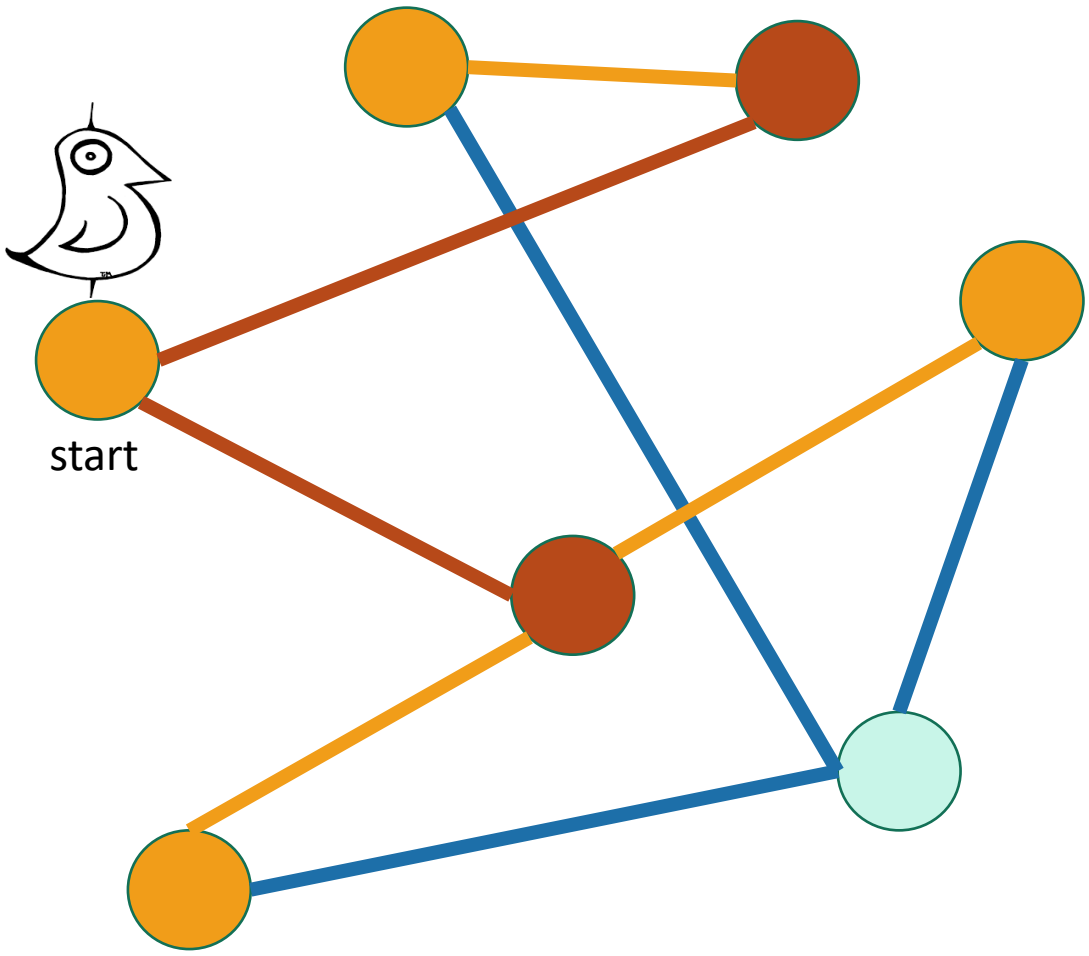
Can reach there in one step

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

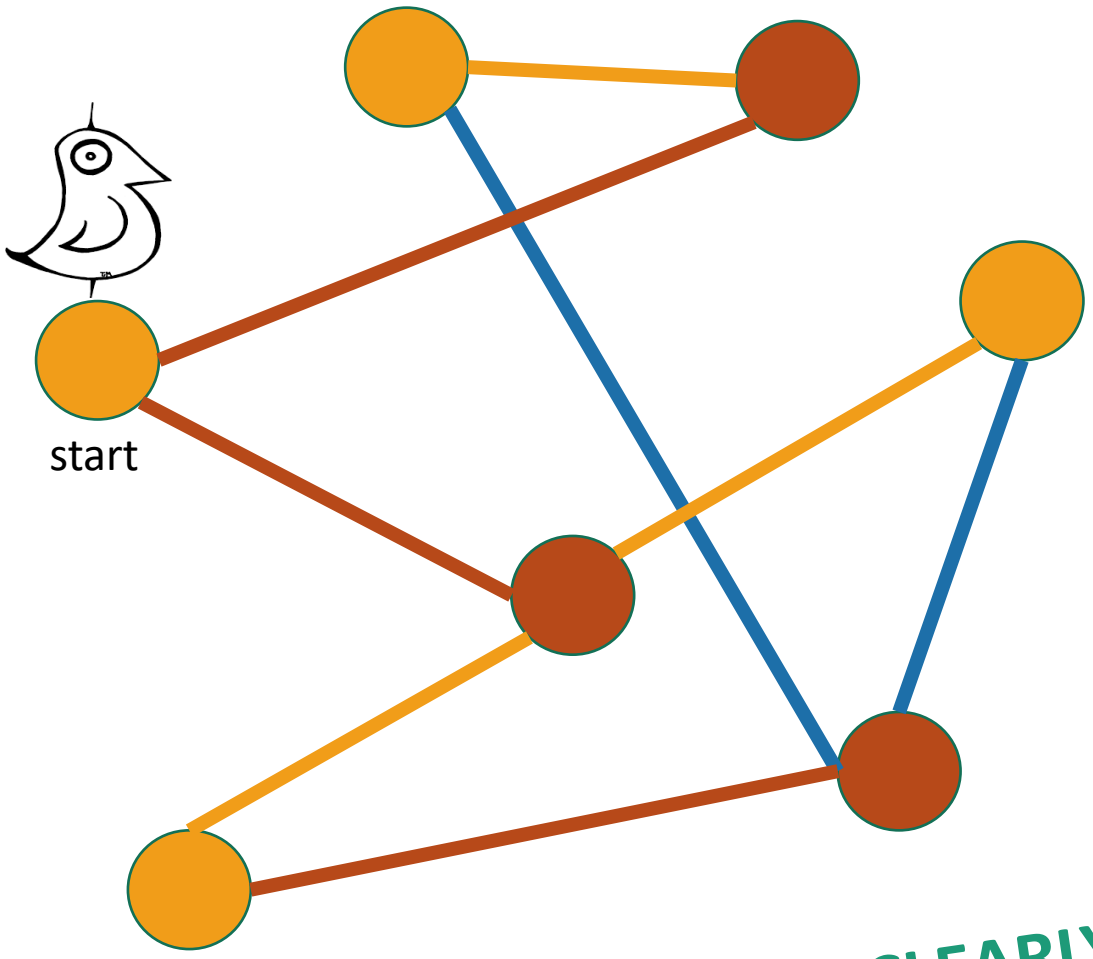
Can reach there in one step

Can reach there in two steps

Can reach there in three steps

Breadth-First Search

For testing bipartite-ness



Not been there yet

Can reach there in zero steps

Can reach there in one step

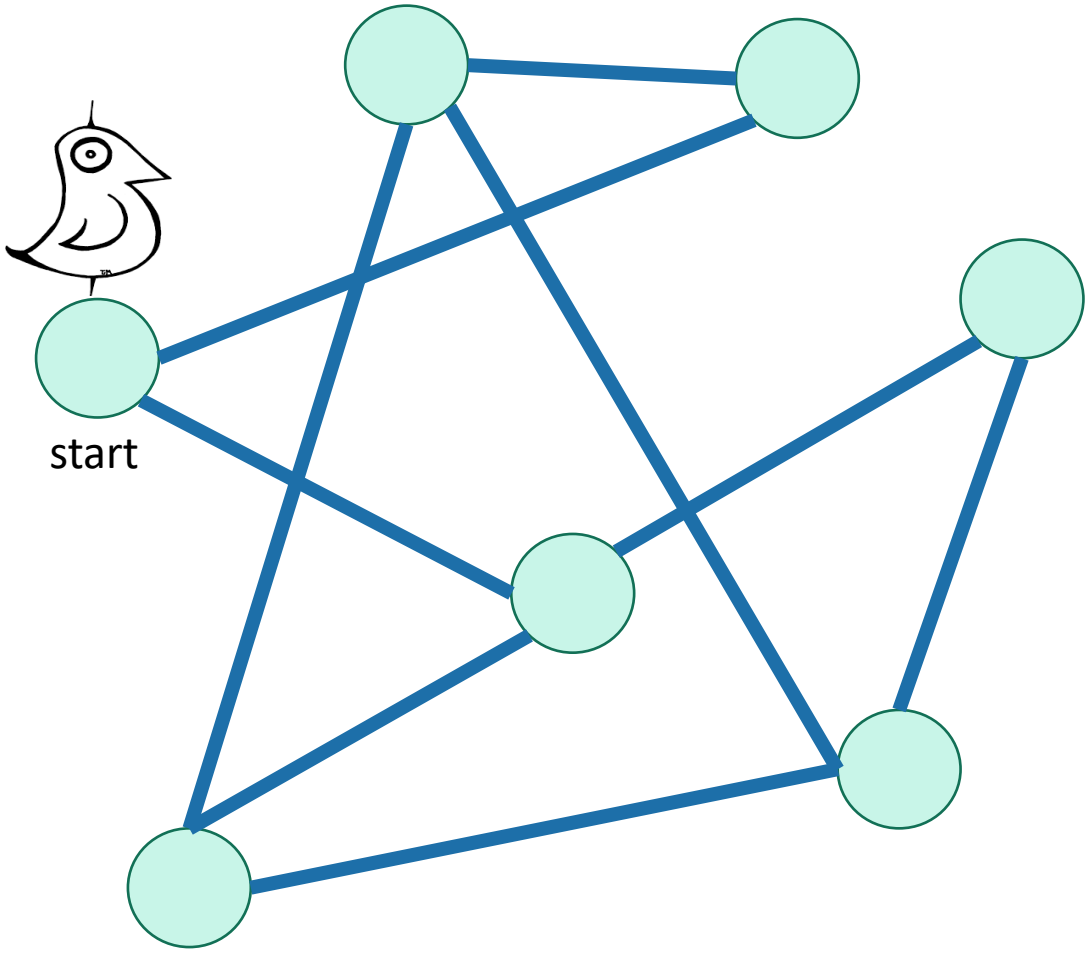
Can reach there in two steps

Can reach there in three steps

CLEARLY BIPARTITE!

Breadth-First Search

For testing bipartite-ness



Not been there yet

Can reach there in zero steps

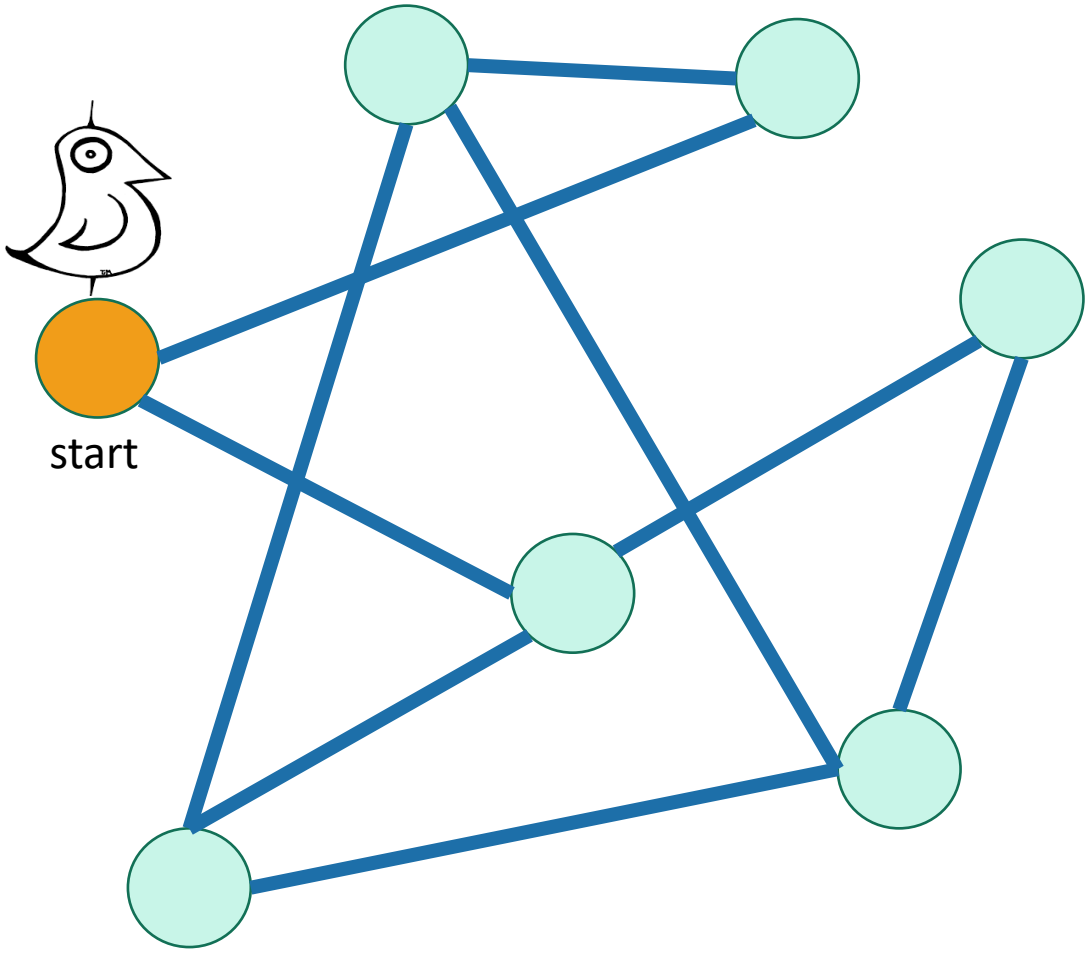
Can reach there in one step

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

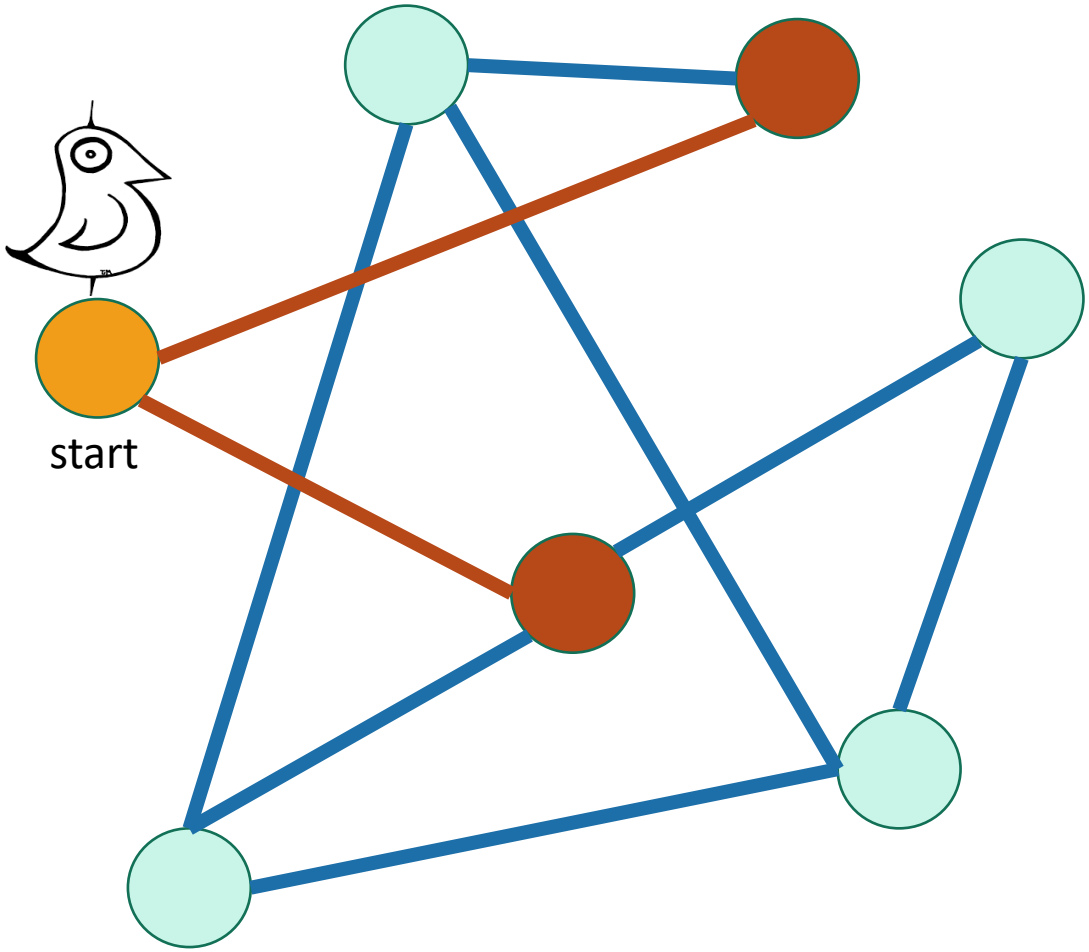
Can reach there in one step

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

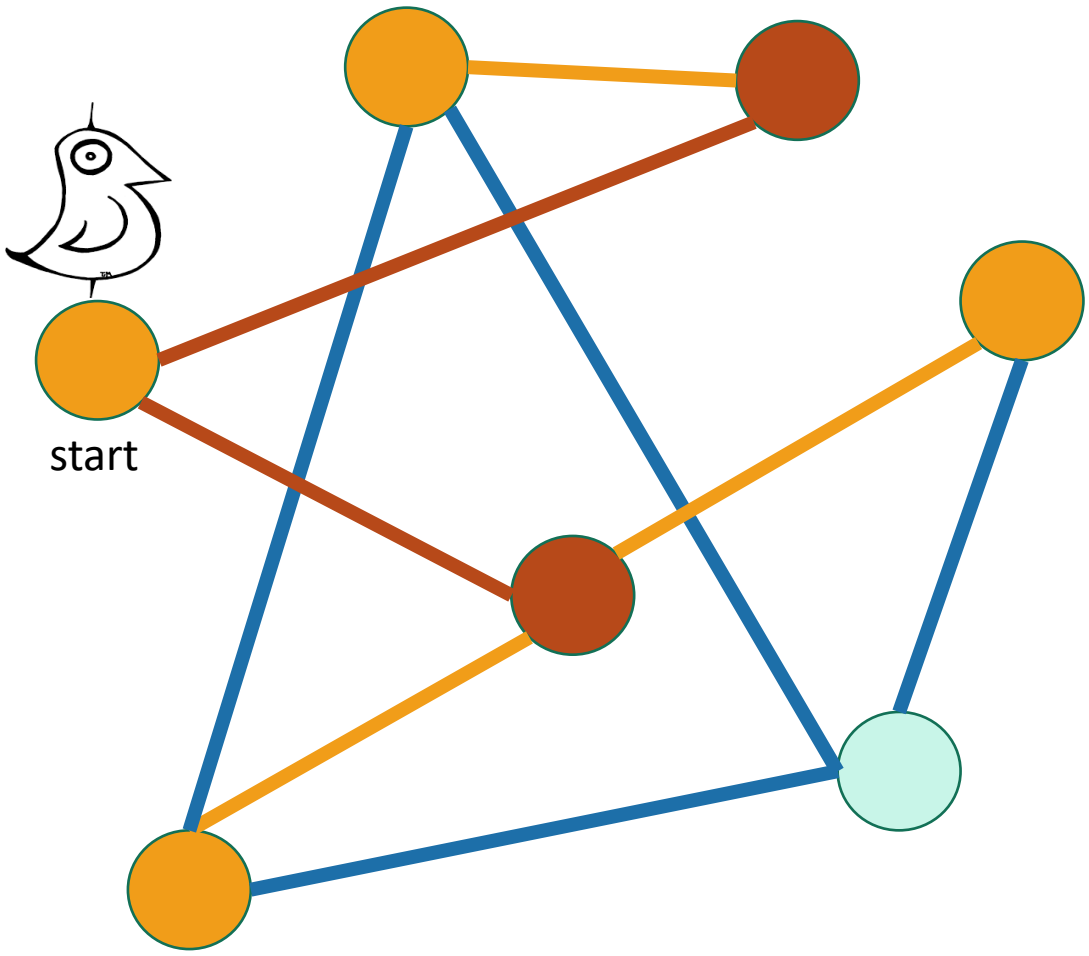
Can reach there in one step

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

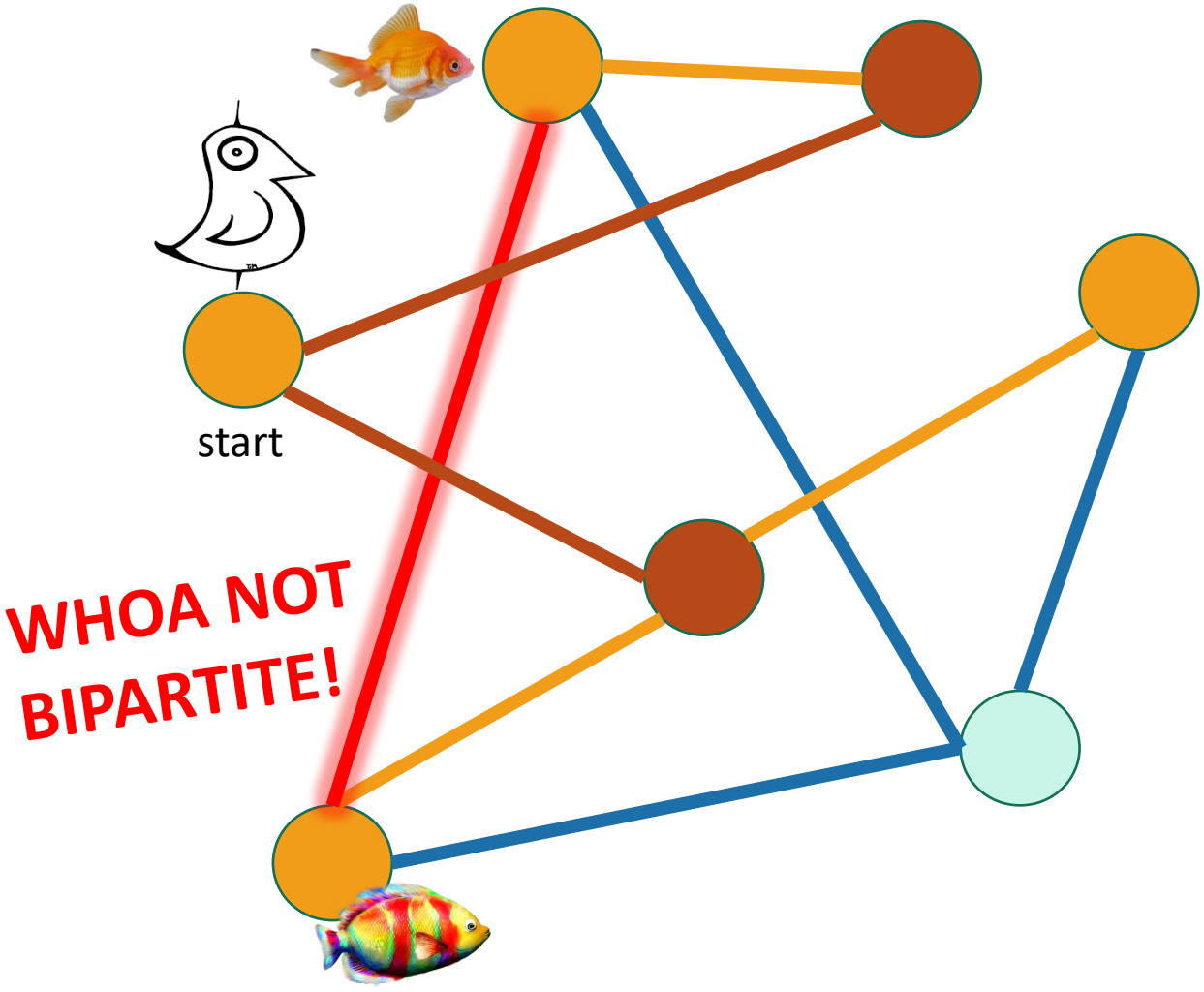
Can reach there in one step

Can reach there in two steps

Can reach there in three steps

Breadth-First Search

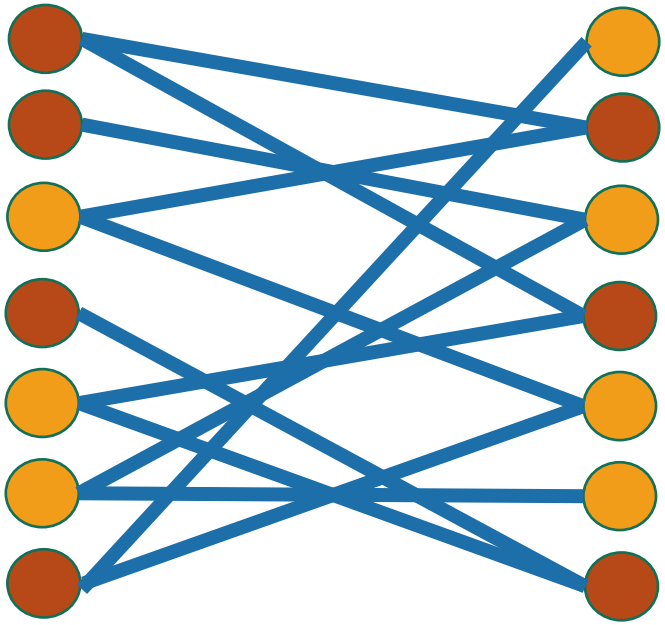
For testing bipartite-ness



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

Hang on now.

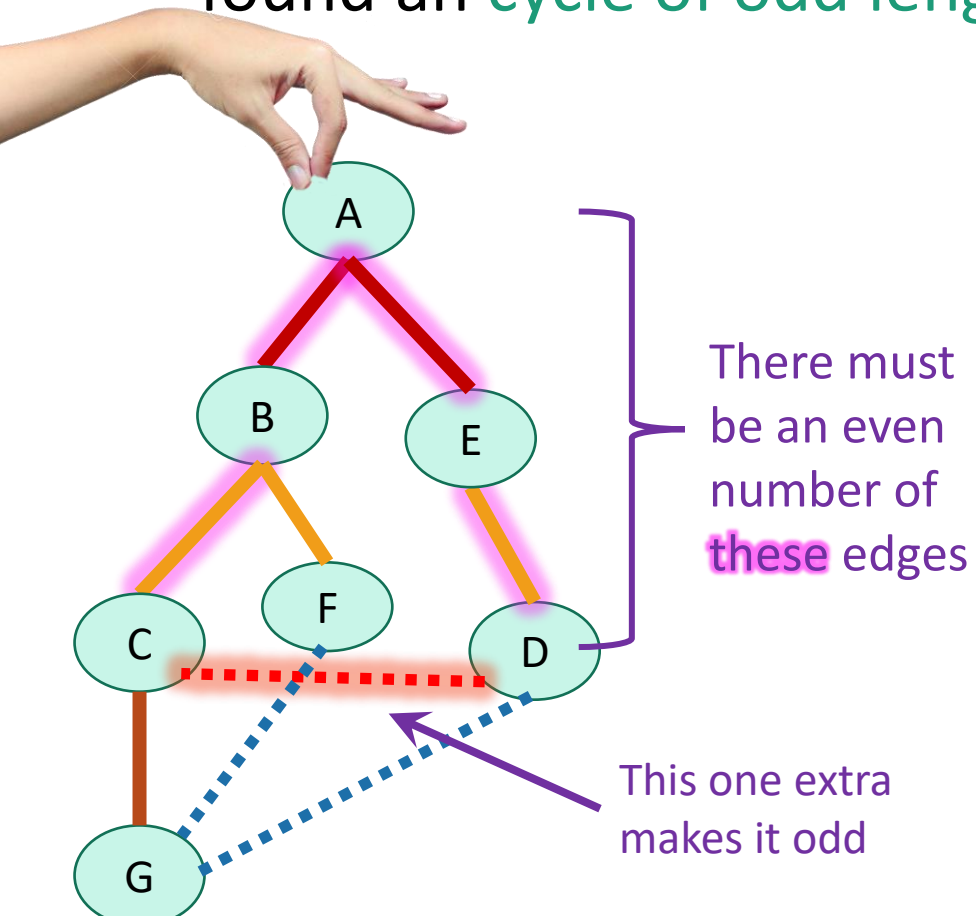
- Just because **this** coloring doesn't work, why does that mean that there is **no** coloring that works?

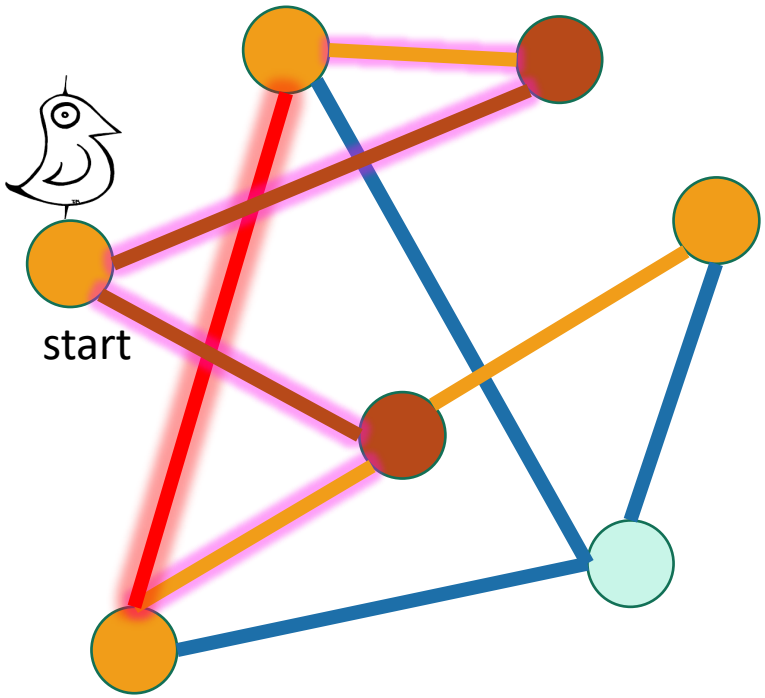


I can come up with plenty of bad colorings on this legitimately bipartite graph…

Some proof required

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

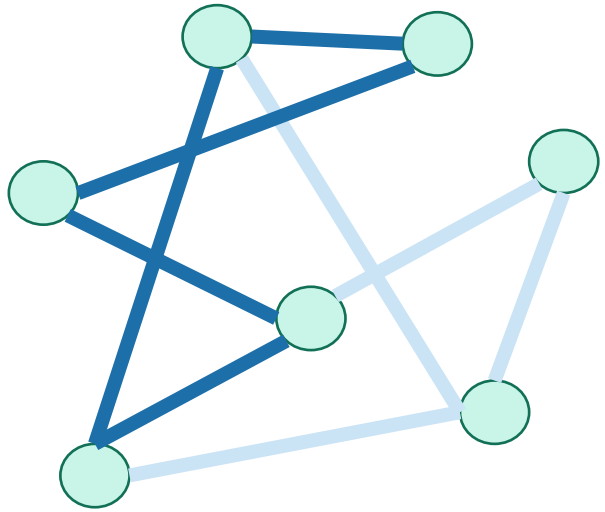




Some proof required

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



n vertices and m edges.

What did we just learn?

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

- 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 LECTURE

- Greedy Algorithms
- Interval Scheduling
- Interval Partitioning
- Shortest Paths in a Graph (Dijkstra)

Week	Date	Topic
1	21-Feb	Introduction. Some representative problems
2	28-Feb	Stable Matching
3	7-Mar	Basics of algorithm analysis.
4	14-Mar	Graphs (Project 1 announced)
5	21-Mar	Greedy algorithms-I
6	28-Mar	Greedy algorithms-II
7	4-Apr	Divide and conquer (Project 2 announced)
8	11-Apr	Dynamic Programming I
9	18-Apr	Dynamic Programming II
10	25-Apr	Network Flow-I (Project 3 announced)
11	2-May	**Midterm**
12	9-May	Network Flow II
13	16-May	NP and computational intractability-I
14	23-May	NP and computational intractability-II