Lecture

Graphs, BFS and DFS

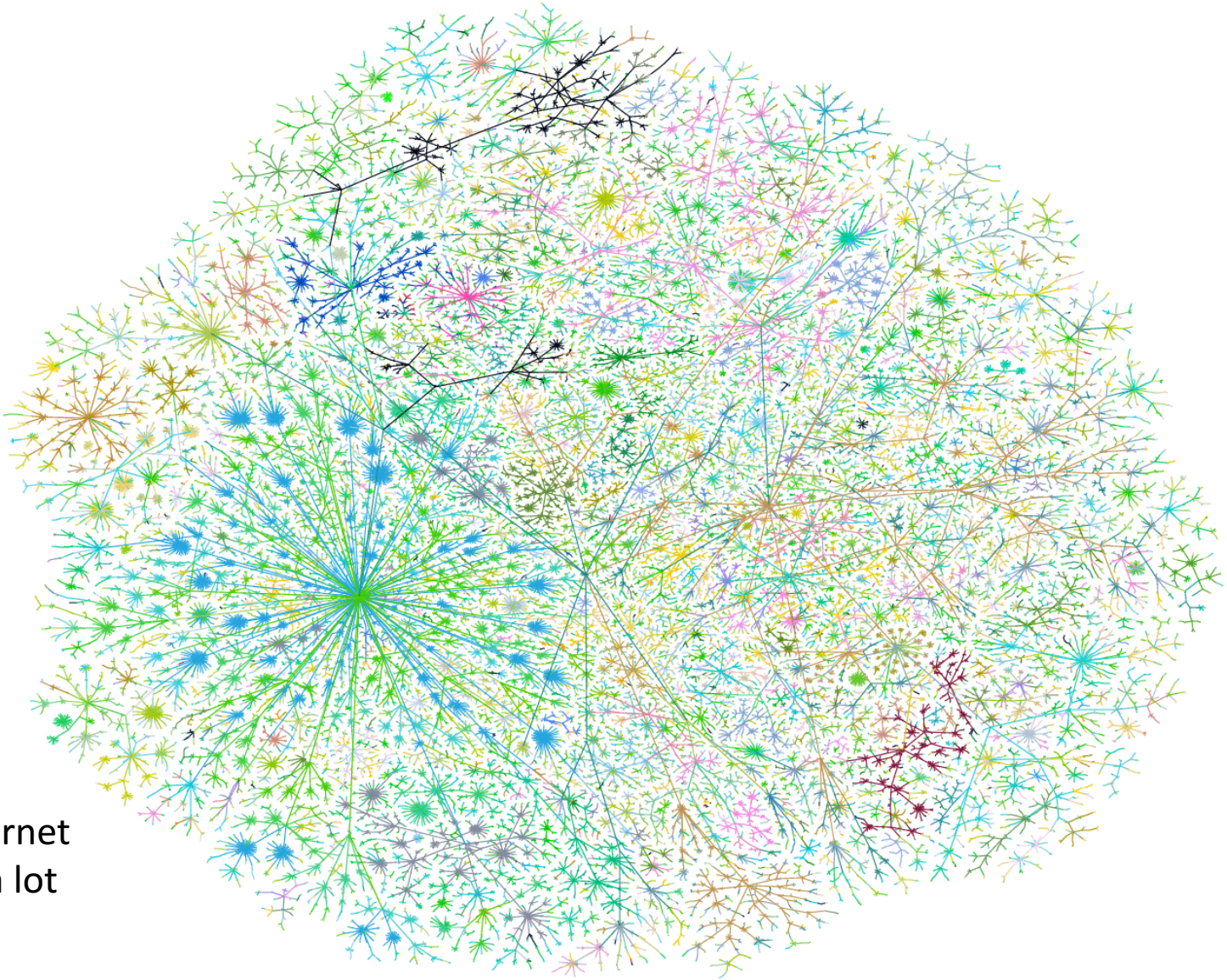
Outline

- 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?

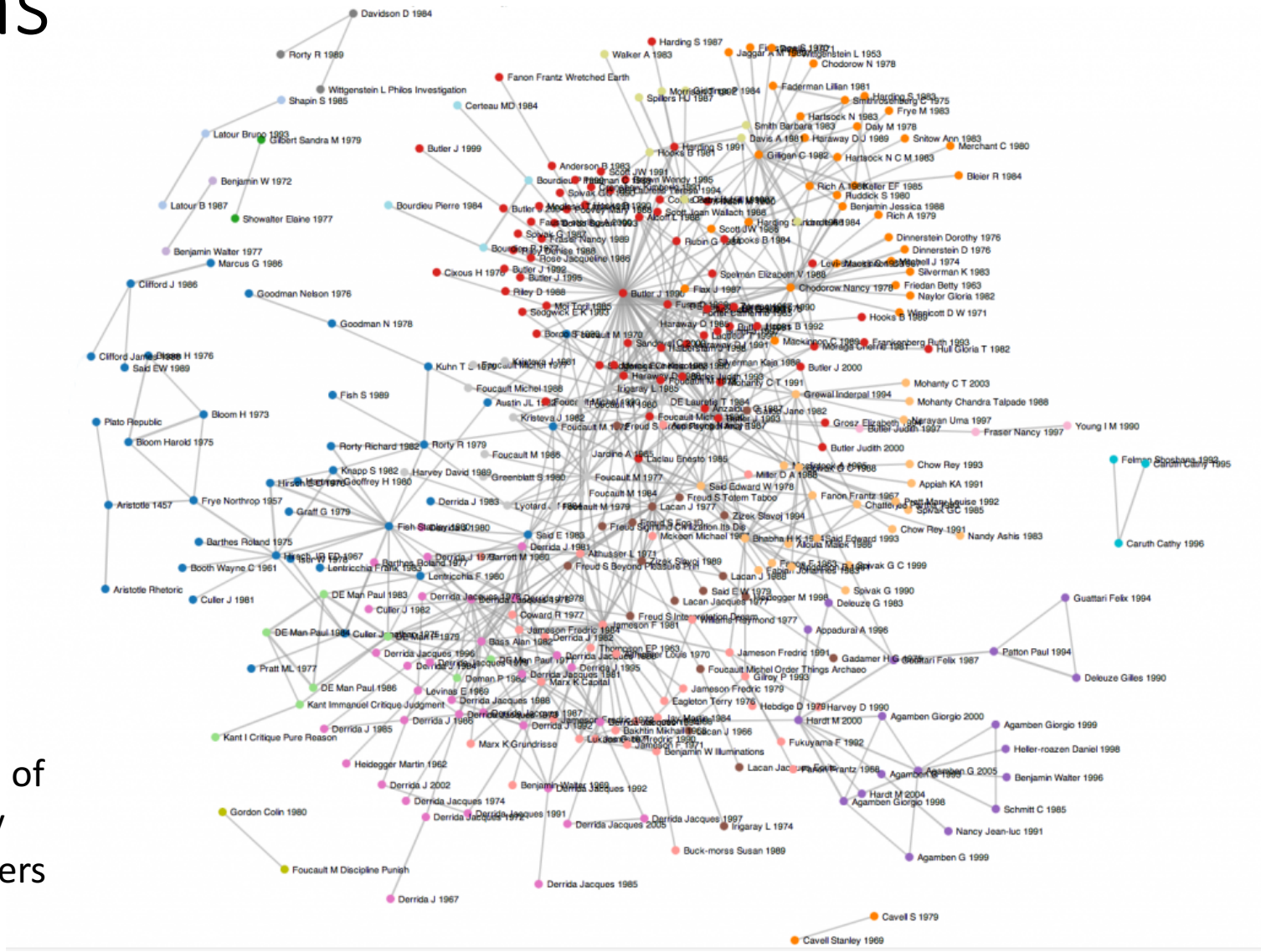
Part 0: Graphs

Graphs



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

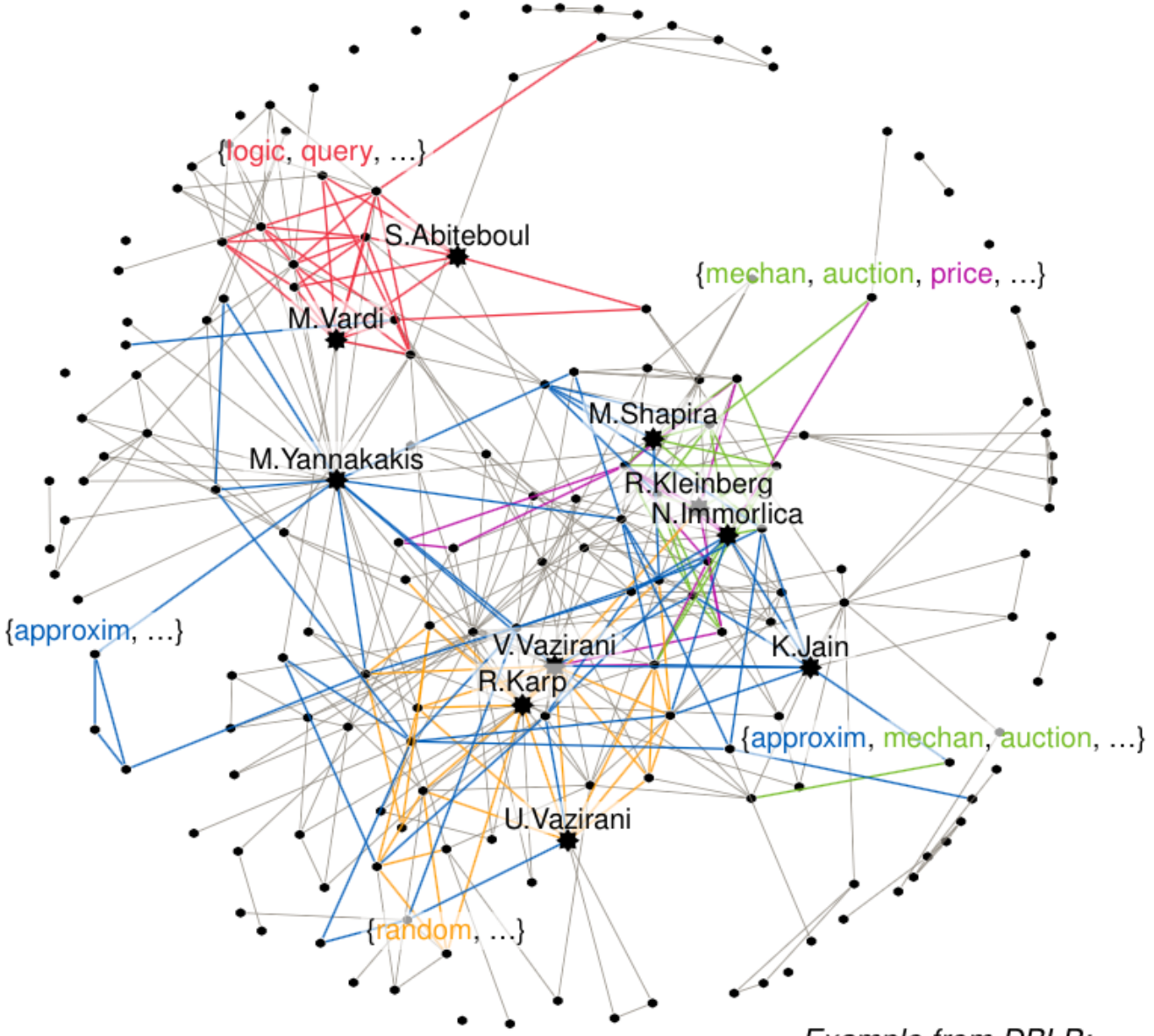
Graphs



Citation graph of literary theory academic papers

Graphs

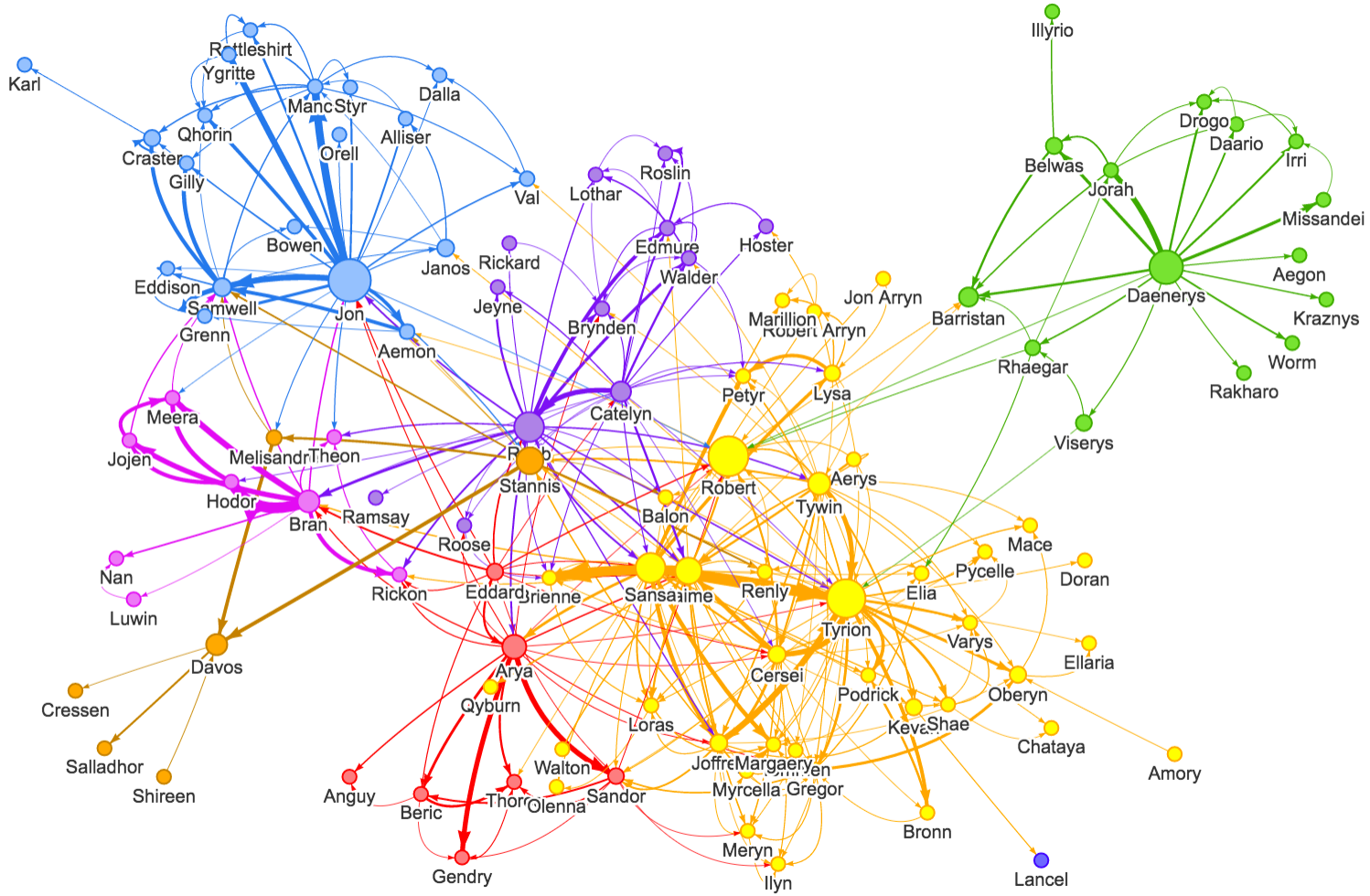
Theoretical Computer Science academic communities



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

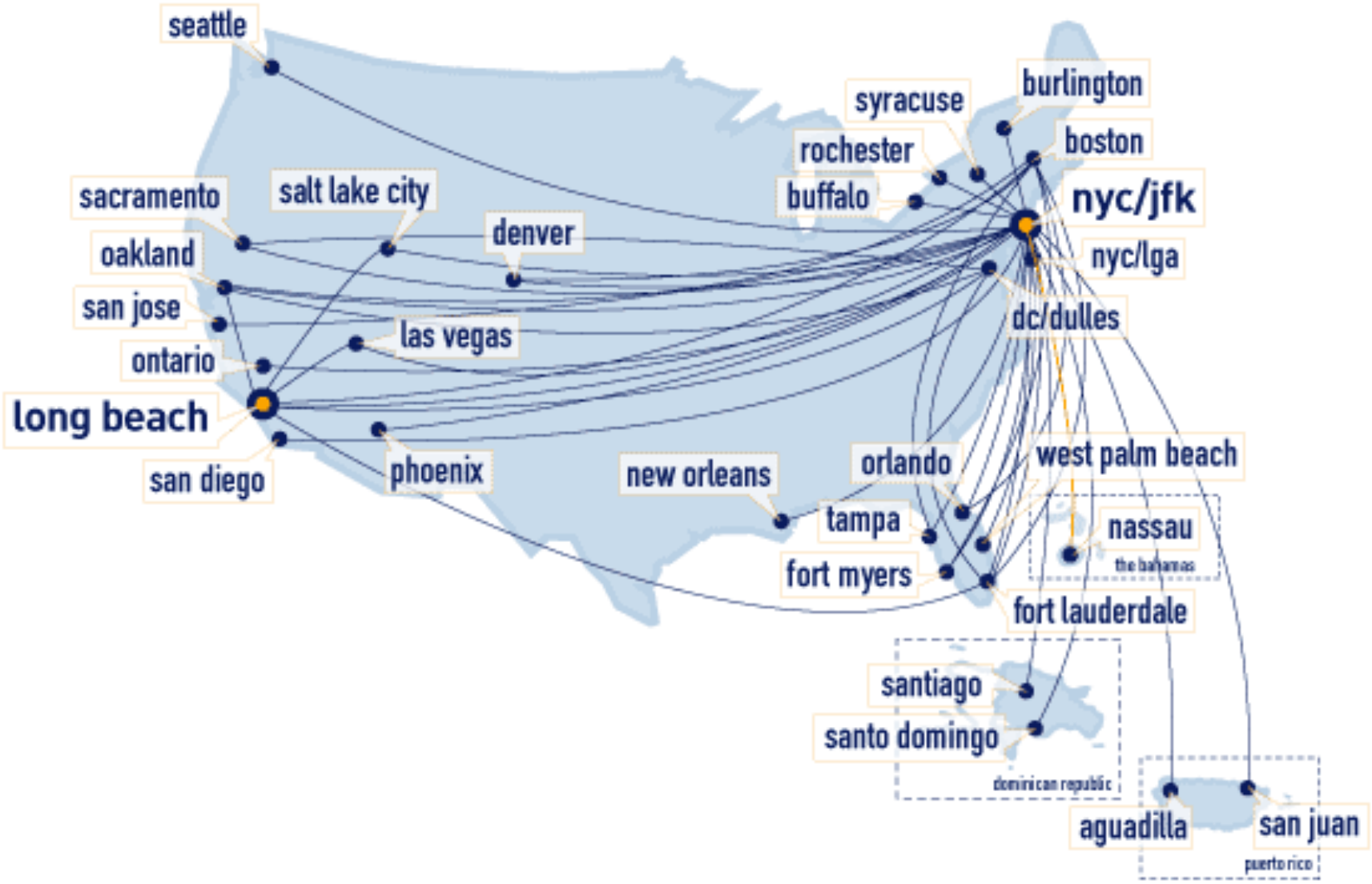
Graphs

Game of Thrones Character Interaction Network



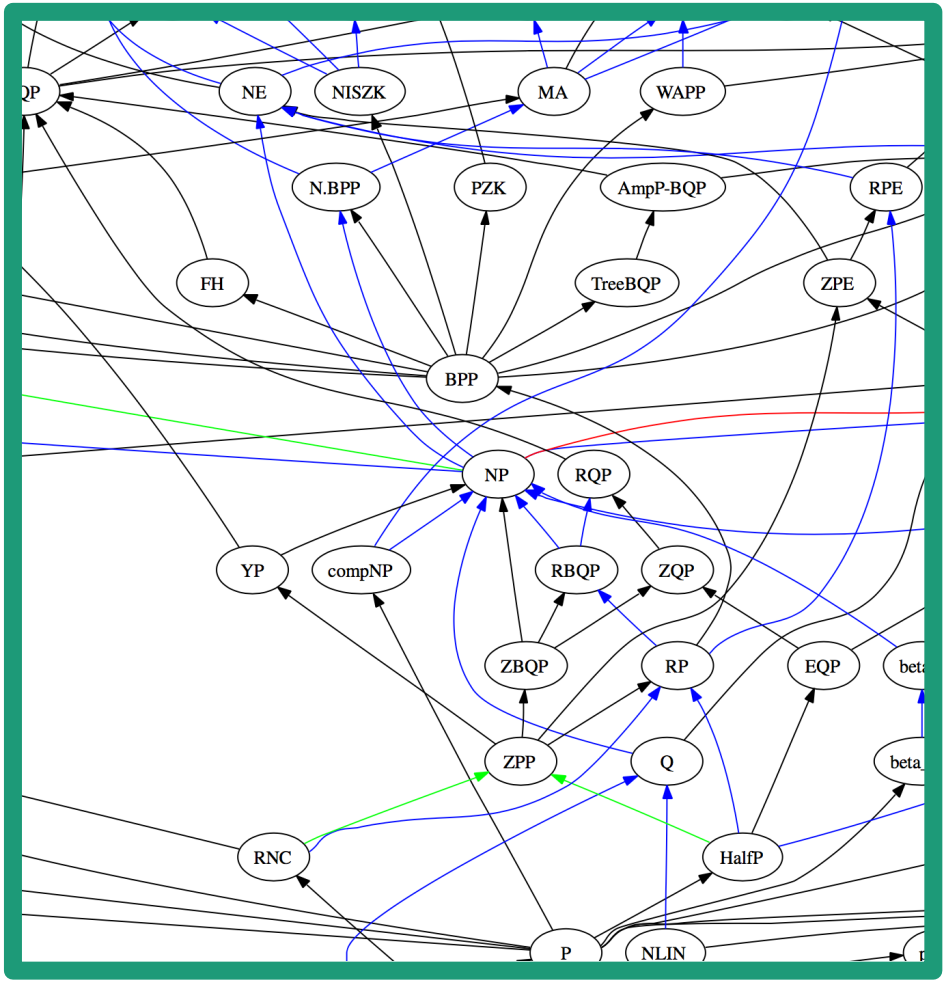
Graphs

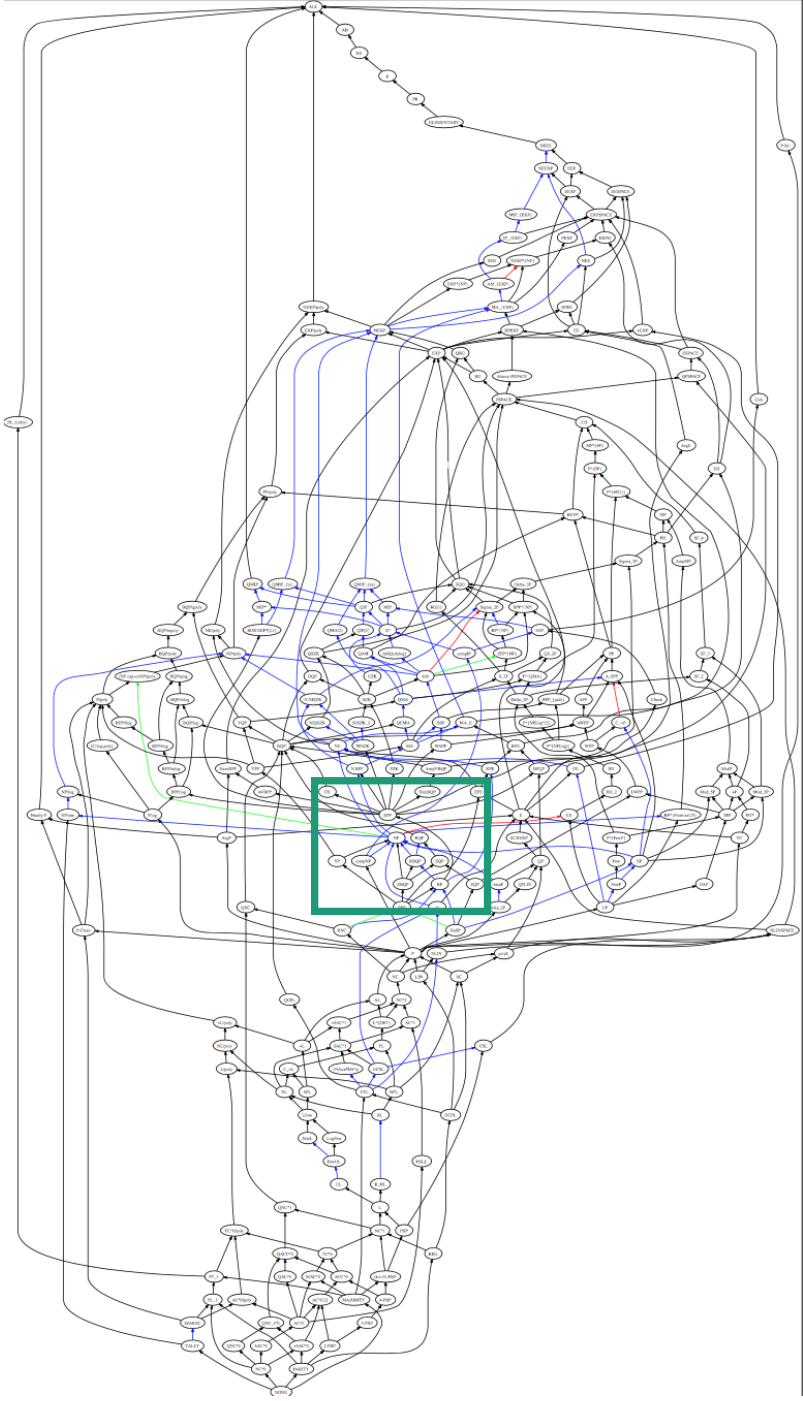
jetblue flights



Graphs

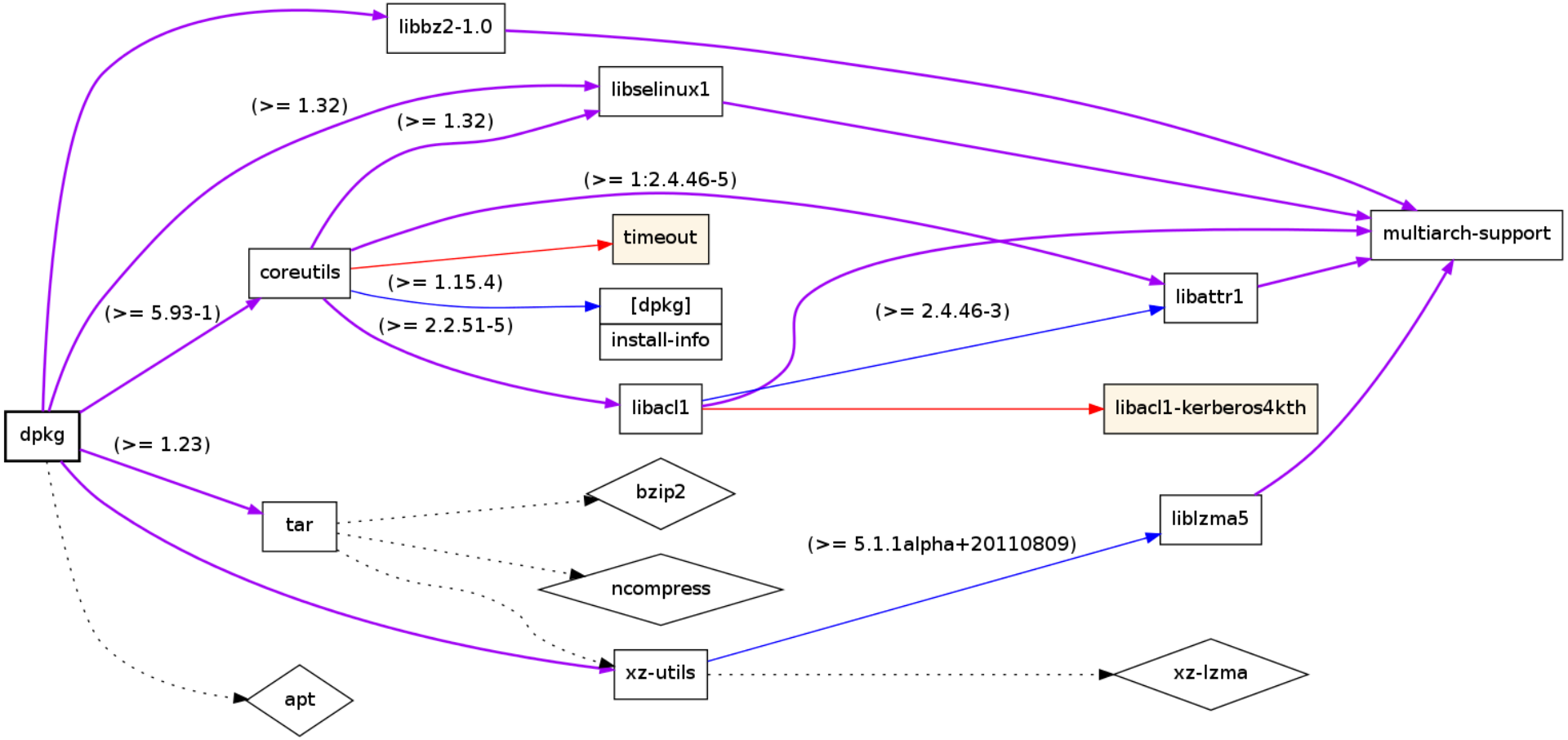
Complexity Zoo
containment graph





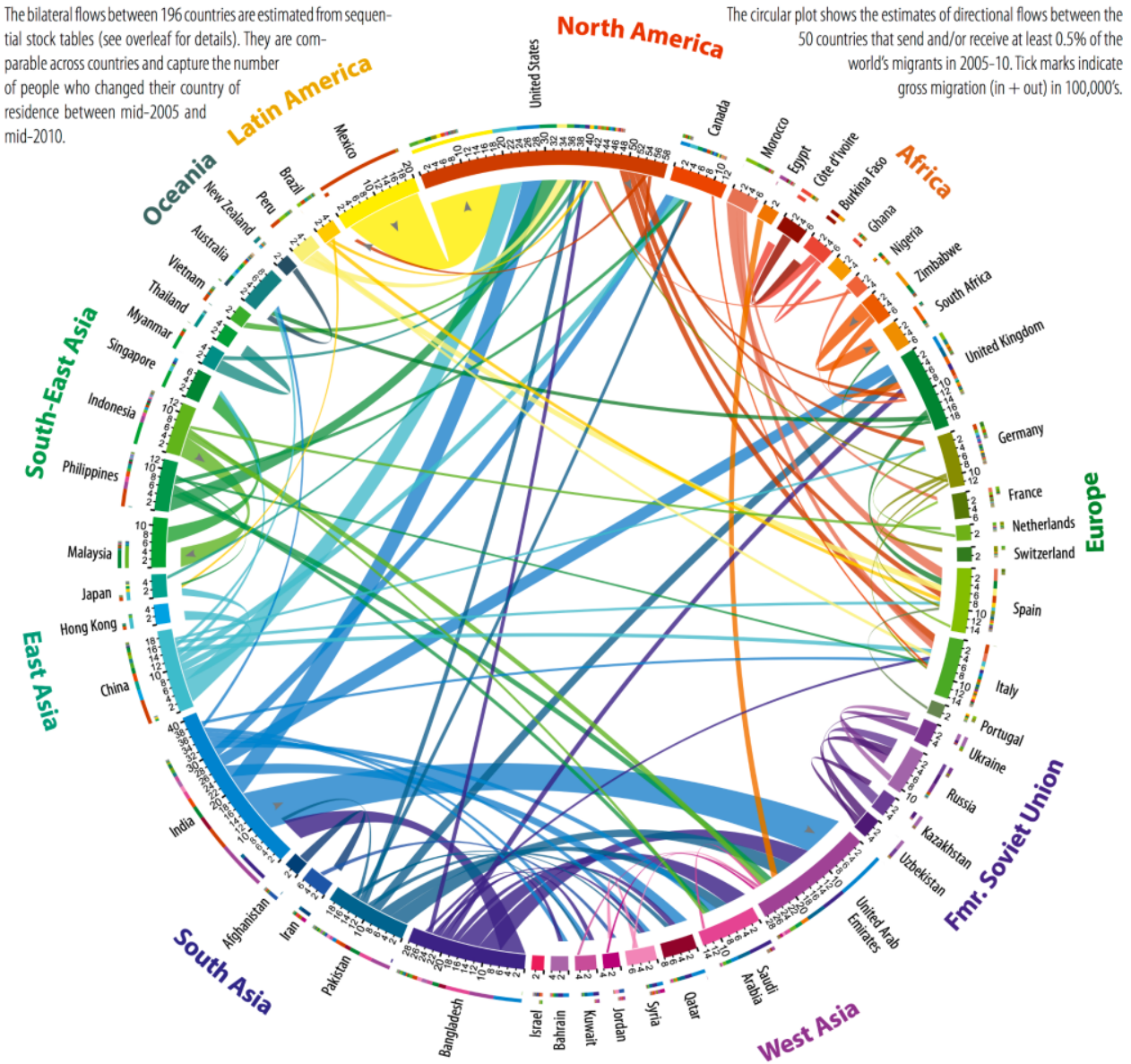
Graphs

debian dependency (sub)graph



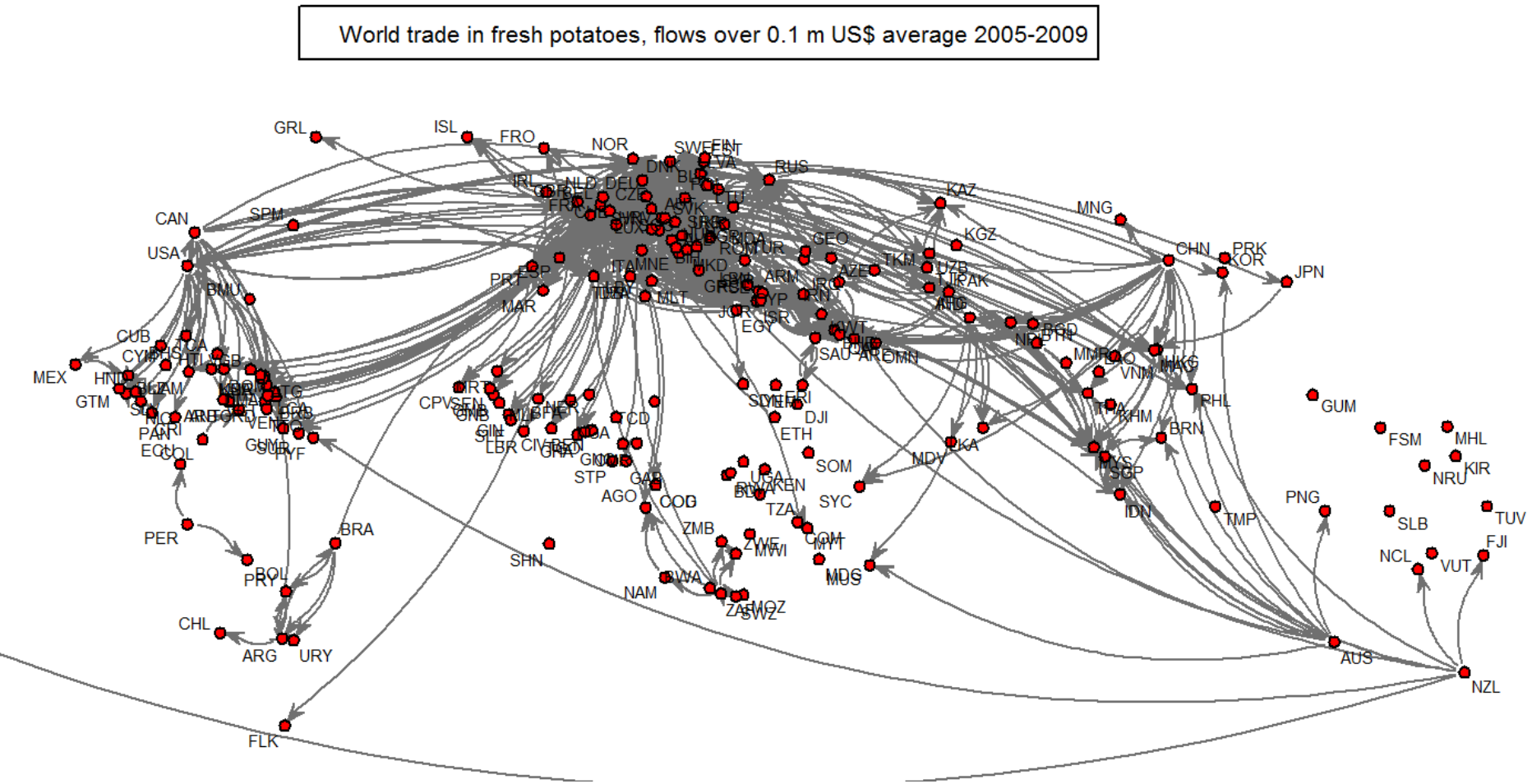
Graphs

Immigration flows



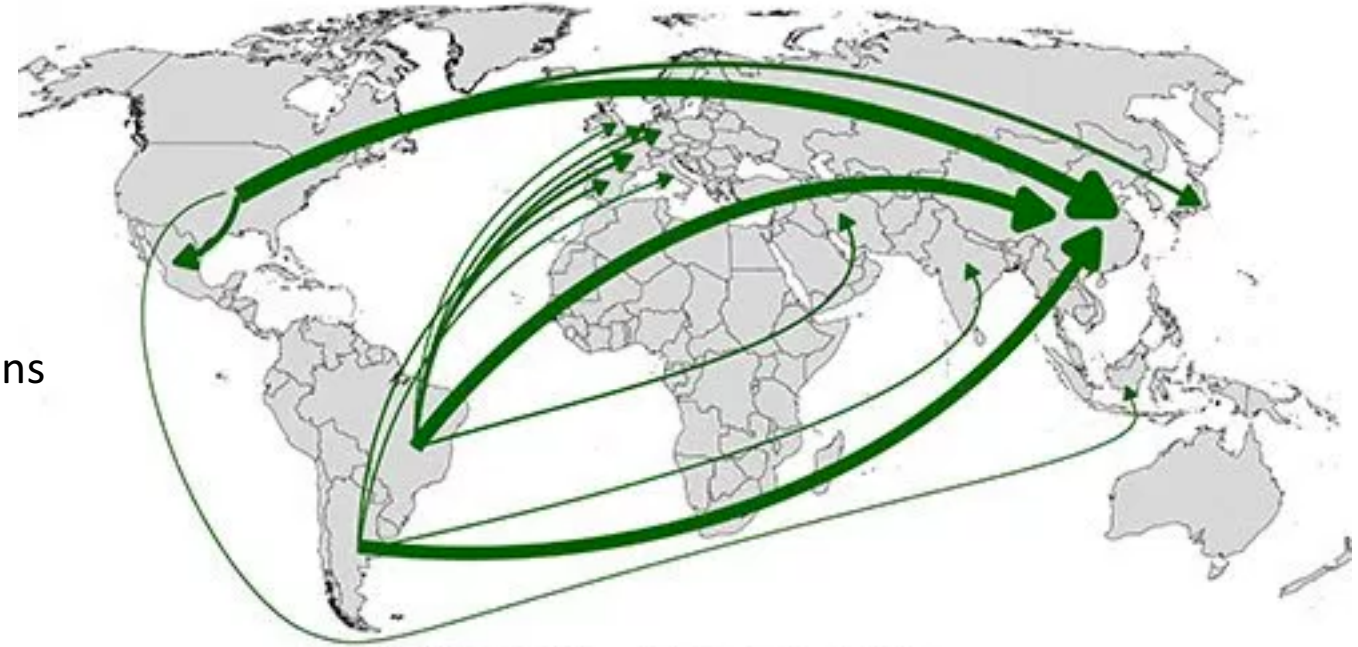
Graphs

Potato trade

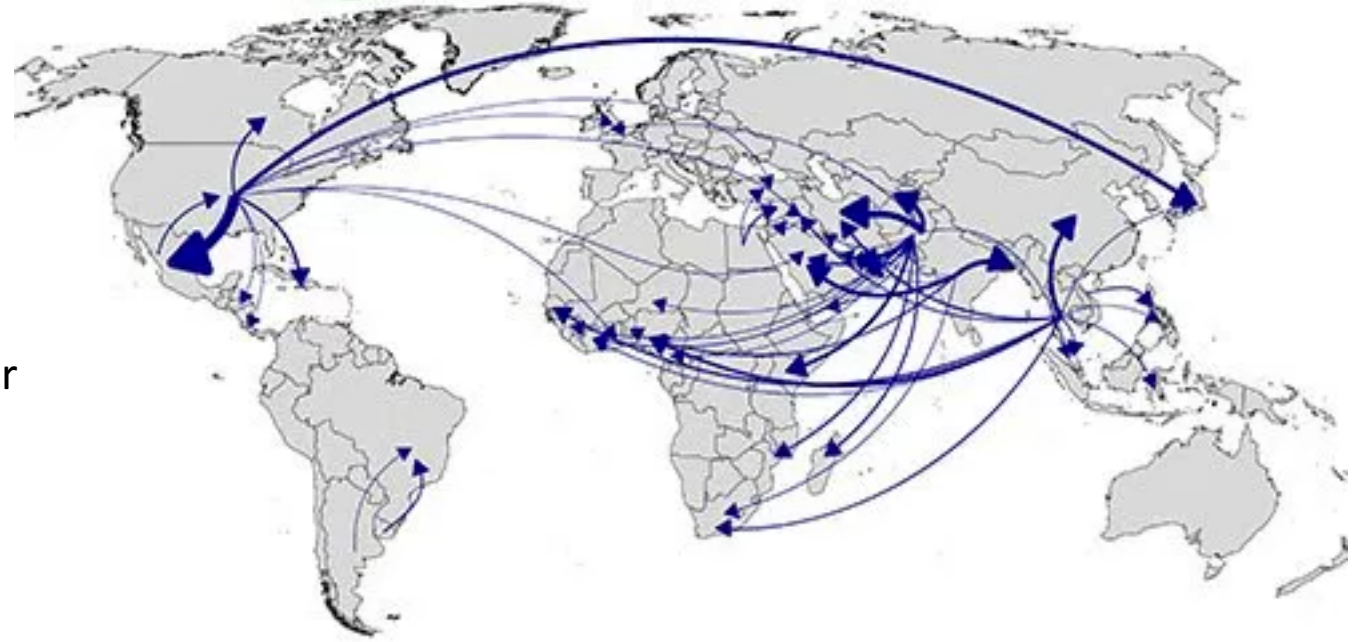


Graphs

Soybeans

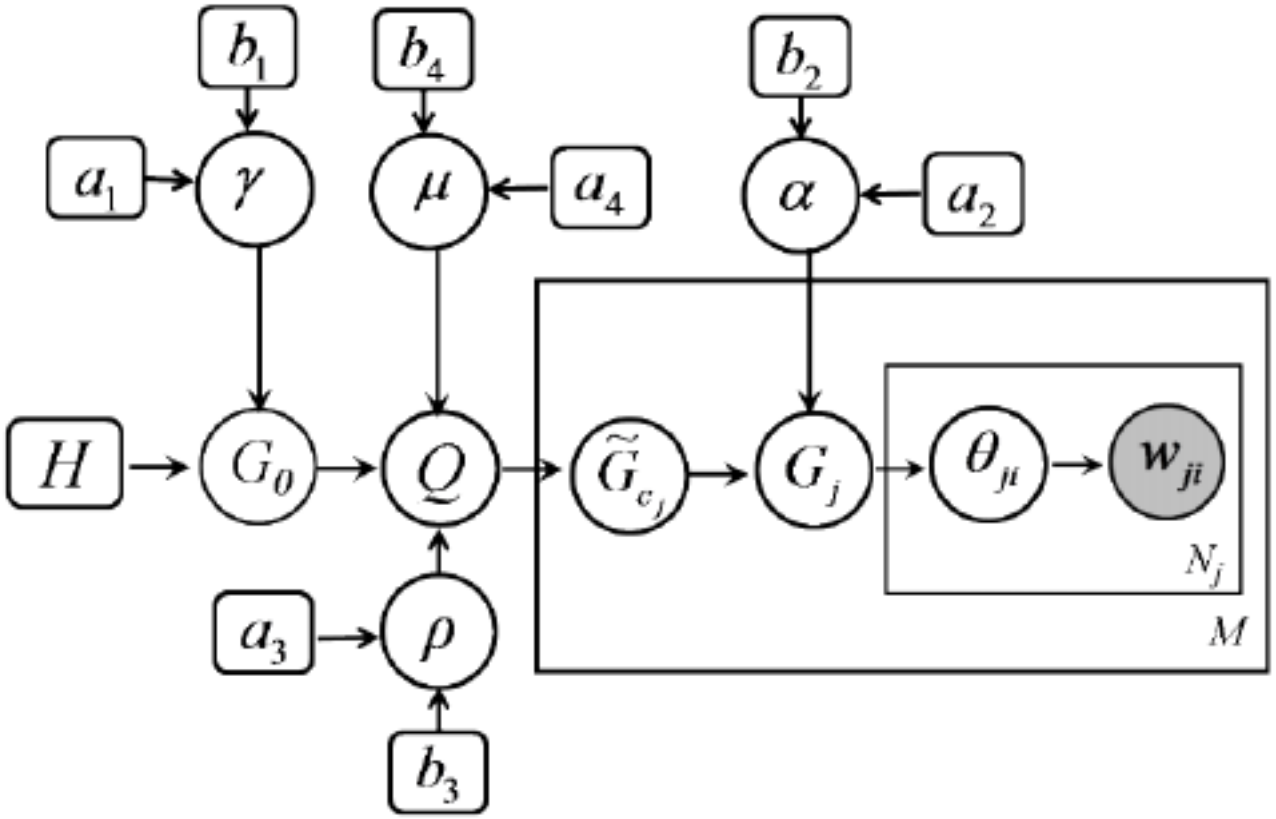


Water



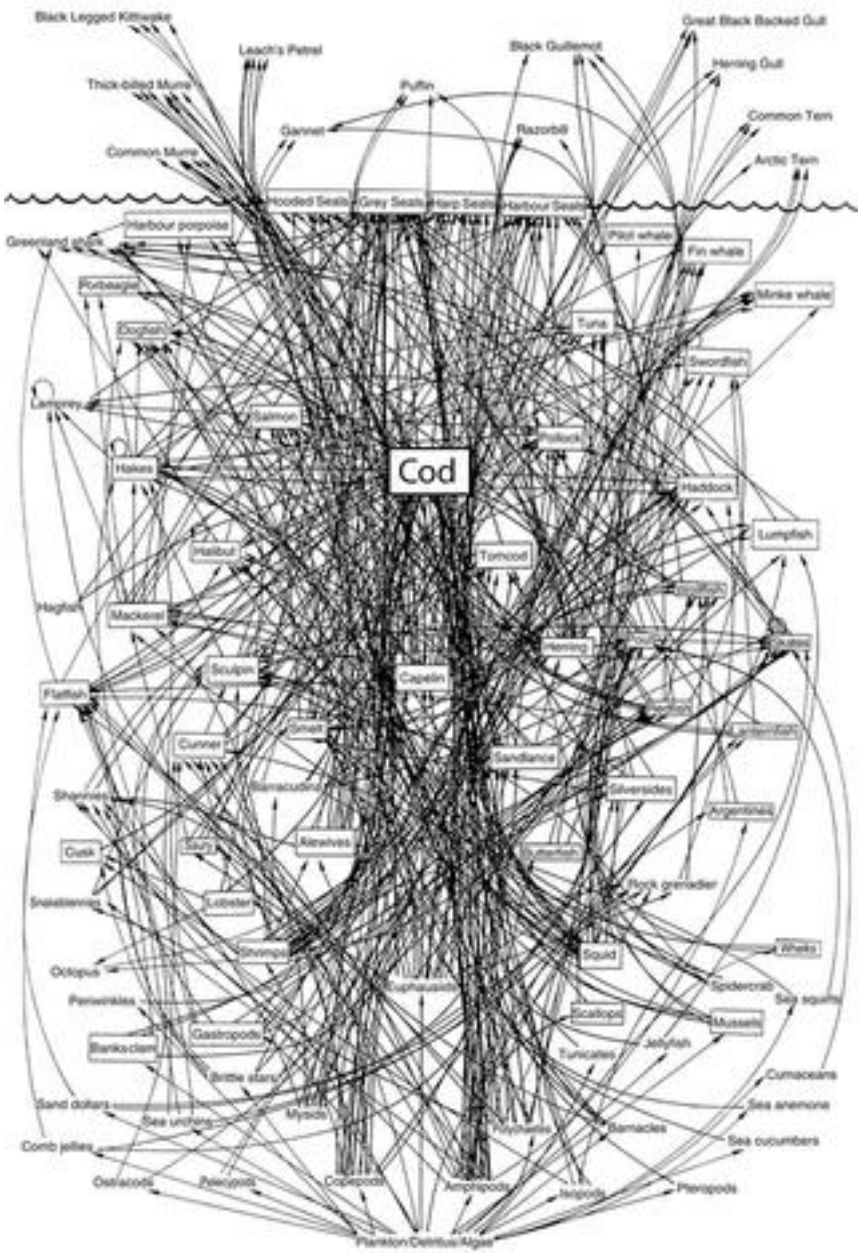
Graphs

Graphical models



Graphs

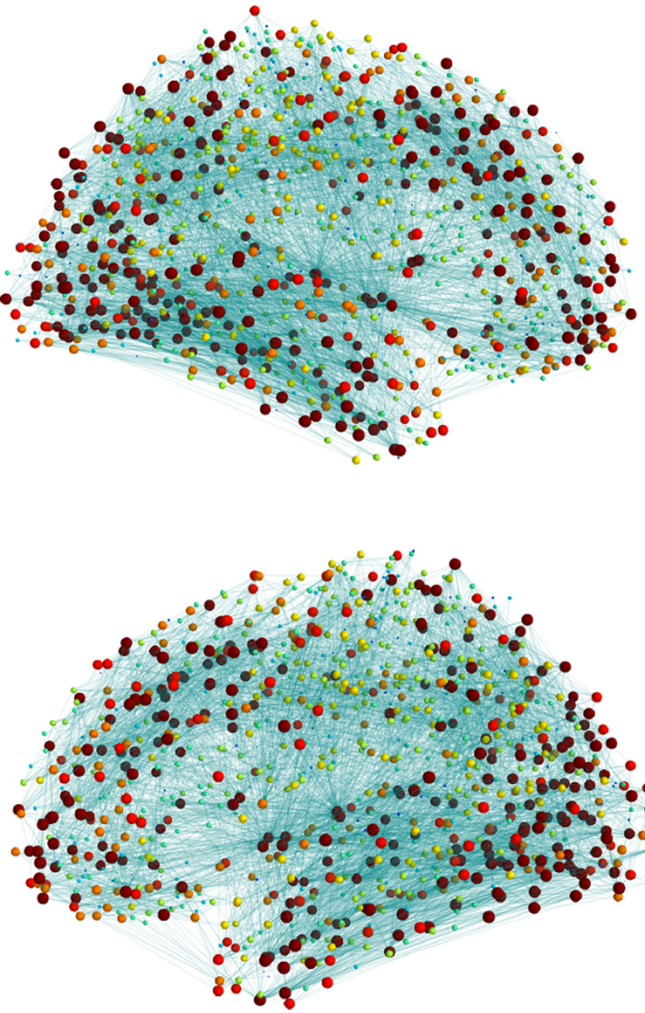
What eats what in the Atlantic ocean?

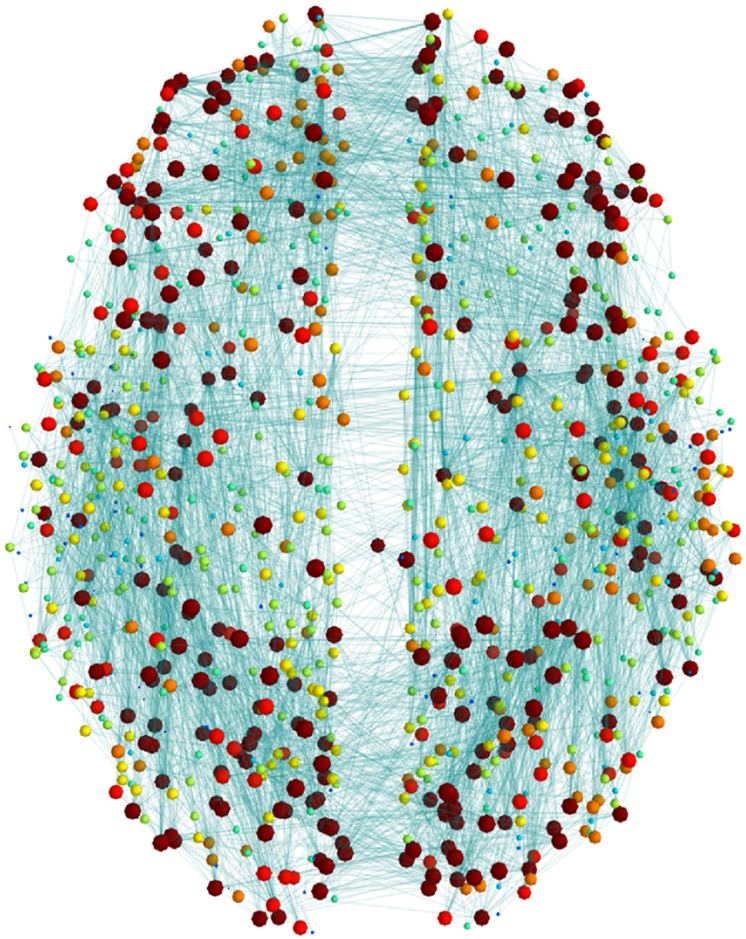


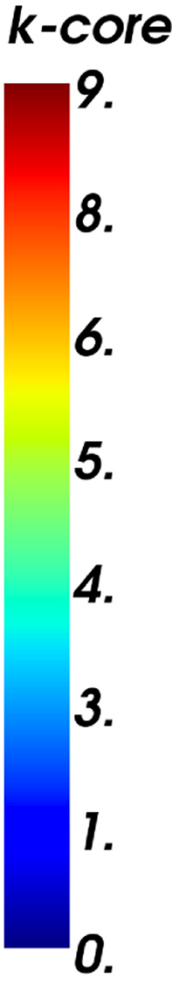
A simplified food web for the Northwest Atlantic. © IMMA

Graphs

Neural connections
in the brain







Graphs

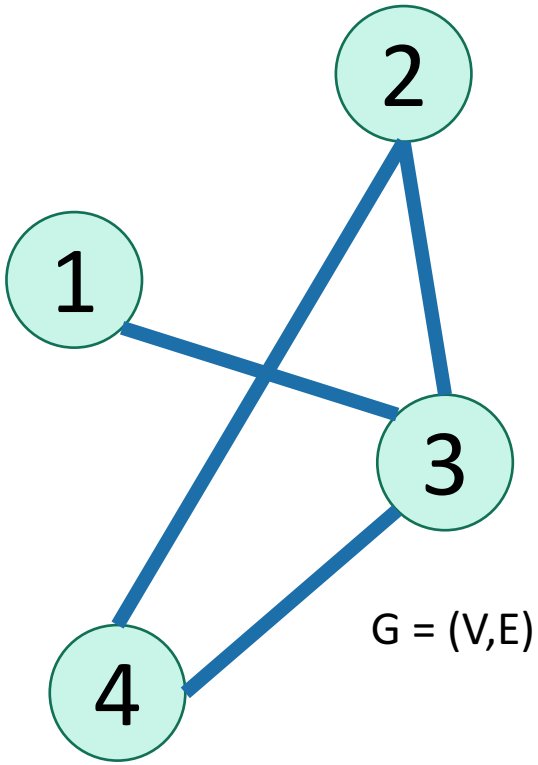
- **There are a lot of graphs.**

- We want to answer questions about them.
  - Efficient routing?
  - Community detection/clustering?
  - An ordering that respects dependencies?

- 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)

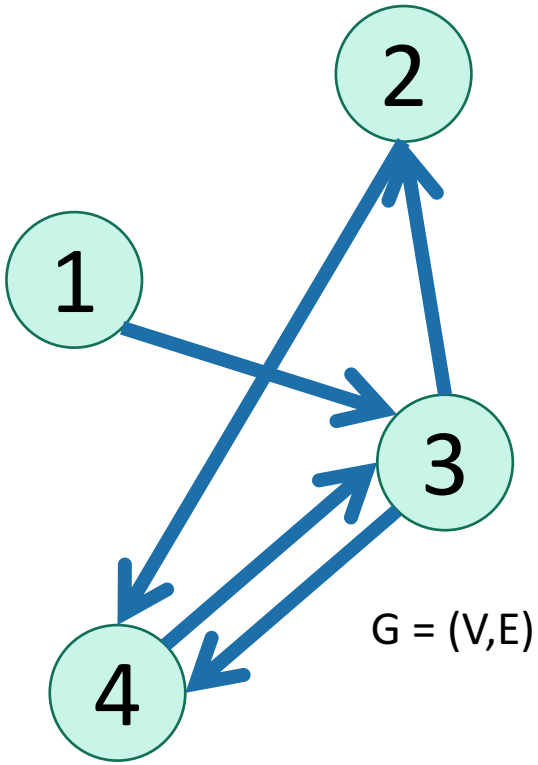


- Example
  - V = {1,2,3,4}
  - E = { {1,3}, {2,4}, {3,4}, {2,3} }

- 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)



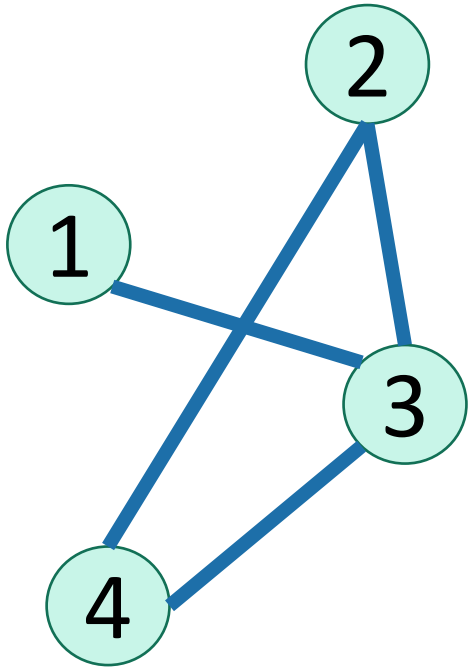
- Example
  - V = {1,2,3,4}
  - E = { (1,3), (2,4), (3,4), (4,3), (3,2) }

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

How do we represent graphs?

- Option 1: adjacency matrix

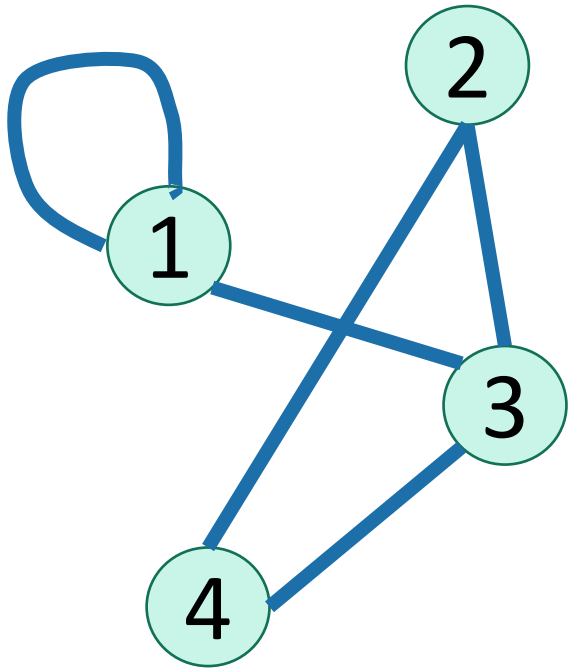
	1	2	3	4
1	0	0	1	0
2	0	0	1	1
3	1	1	0	1
4	0	1	1	0



How do we represent graphs?

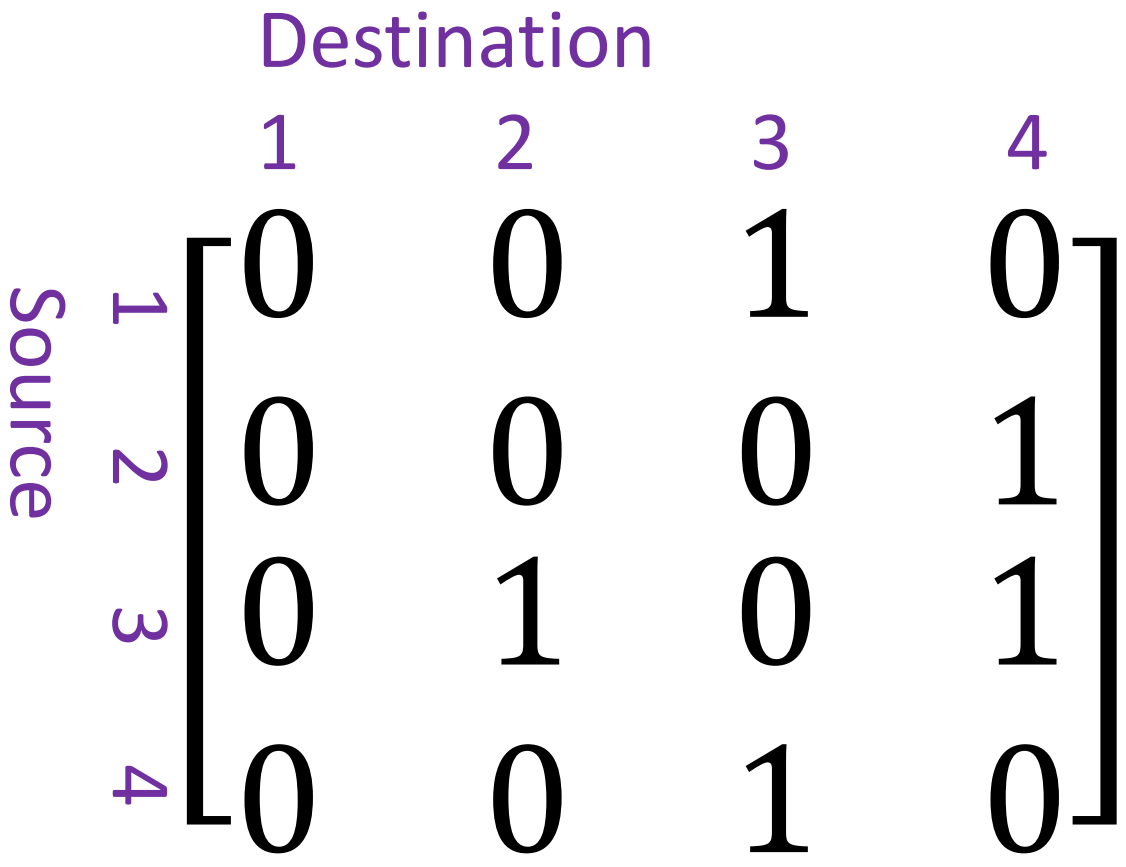
- Option 1: adjacency matrix

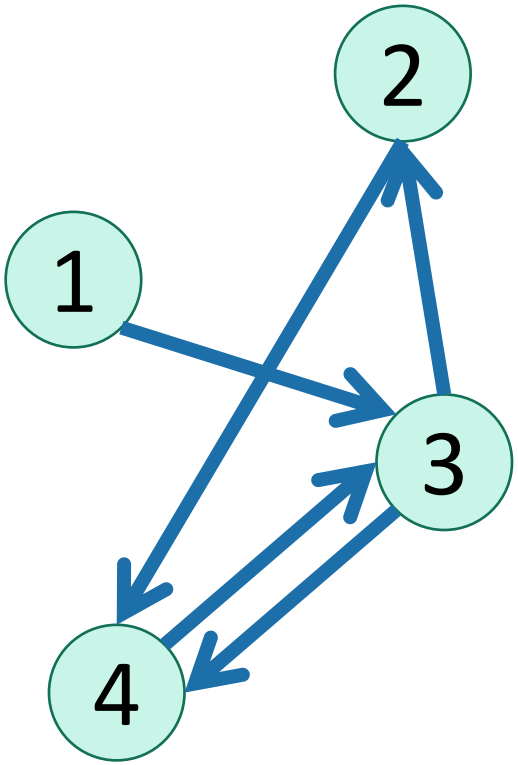
	1	2	3	4
1	1	0	1	0
2	0	0	1	1
3	1	1	0	1
4	0	1	1	0



How do we represent graphs?

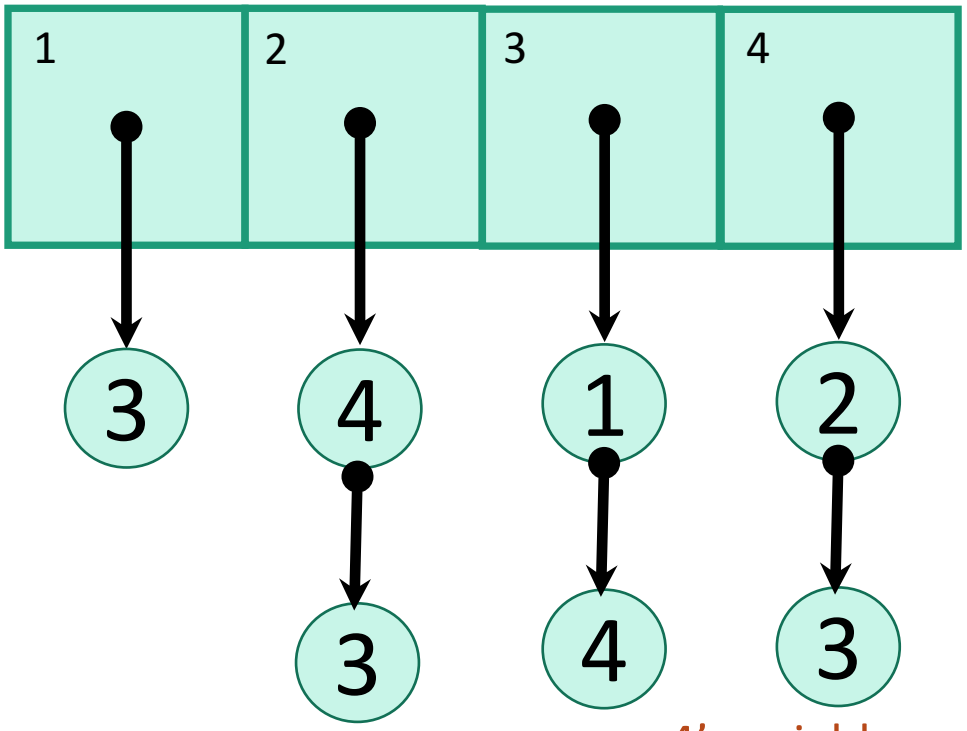
- Option 1: adjacency matrix

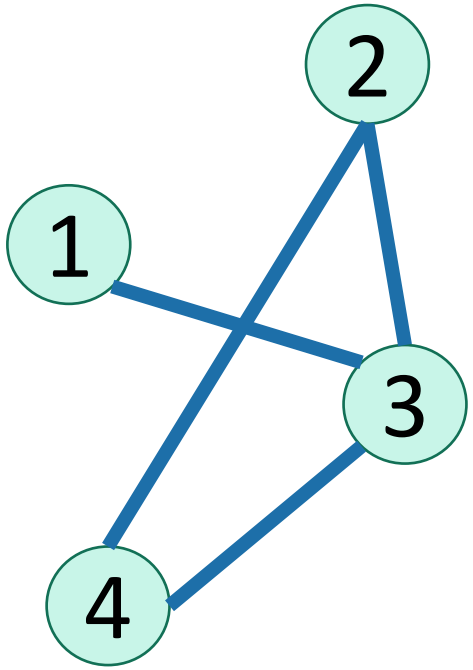




How do we represent graphs?

- Option 2: linked lists.





4's neighbors are
2 and 3

In either case

- May think of vertices storing other information
  - Attributes (name, IP address, …)
  - helper info for algorithms that we will perform on the graph

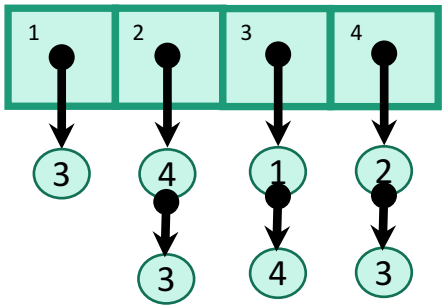
- We will want to be able to do the following ops:
  - Edge Membership: Is edge e in E?
  - Neighbor Query: What are the neighbors of vertex v?

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

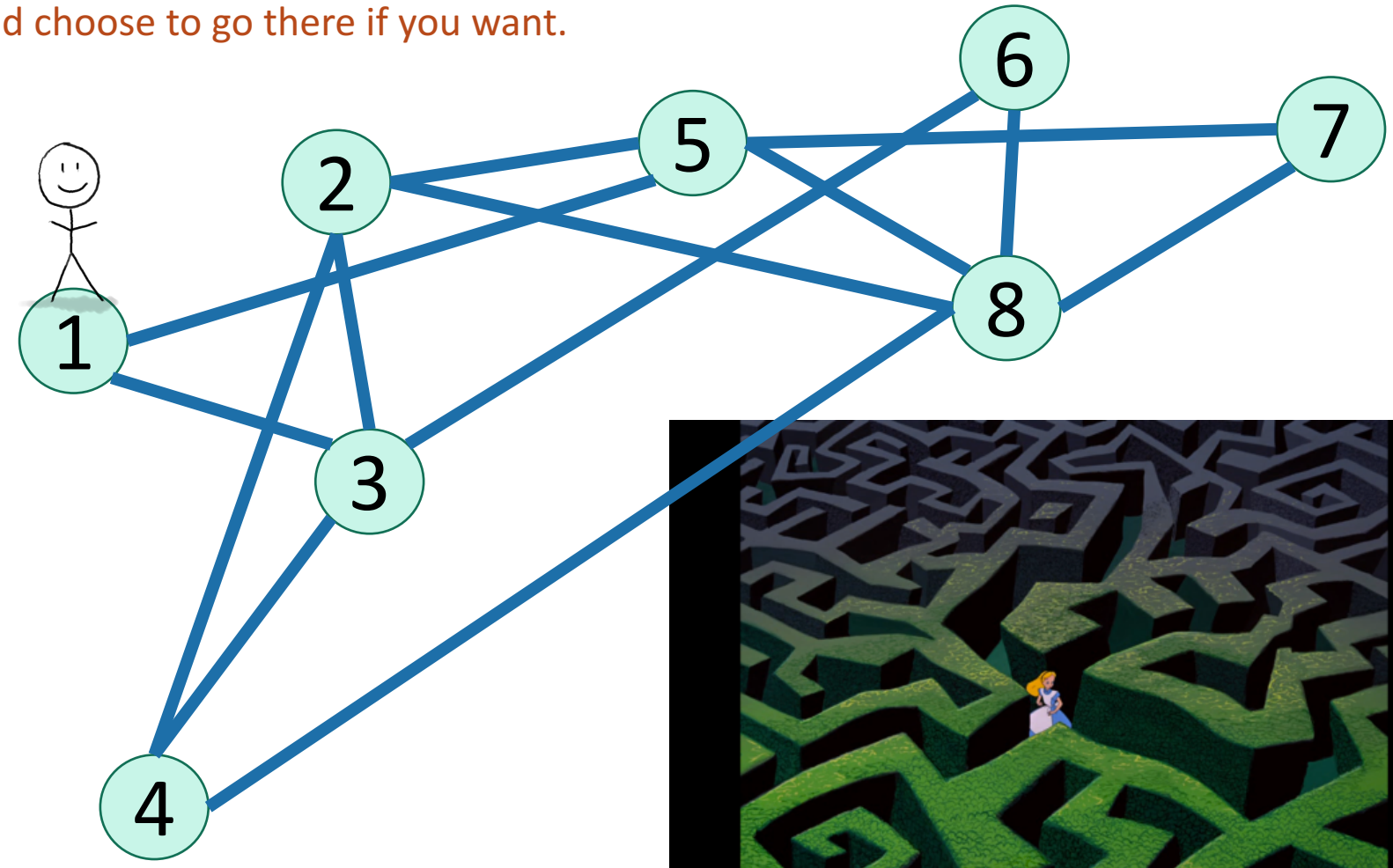


Edge membership Is e = {v,w} in E?	O(1)	O(deg(v)) or O(deg(w))
Neighbor query Give me v's neighbors.	O(n)	O(deg(v))
Space requirements	$O(n^2)$	O(n + m) **We'll assume this representation for the rest of the class**

Part 1: Depth-first search

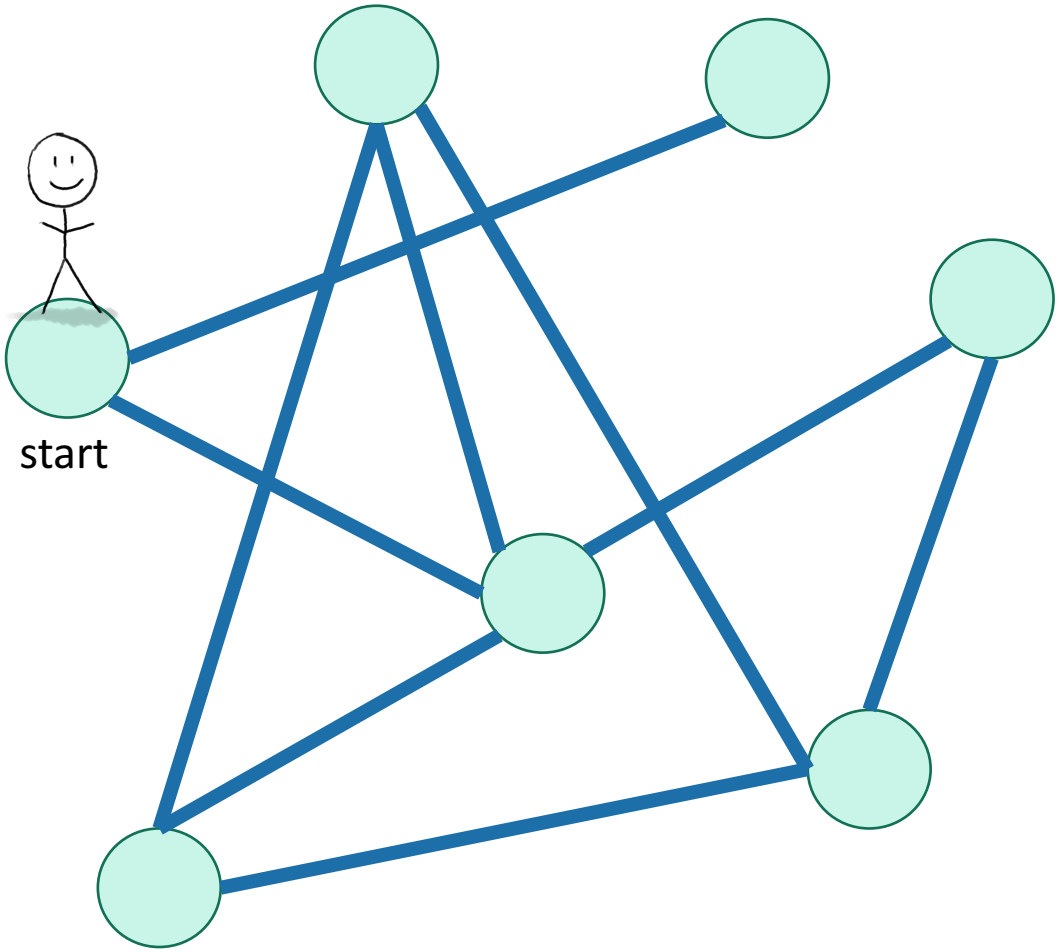
How do we explore a graph?

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



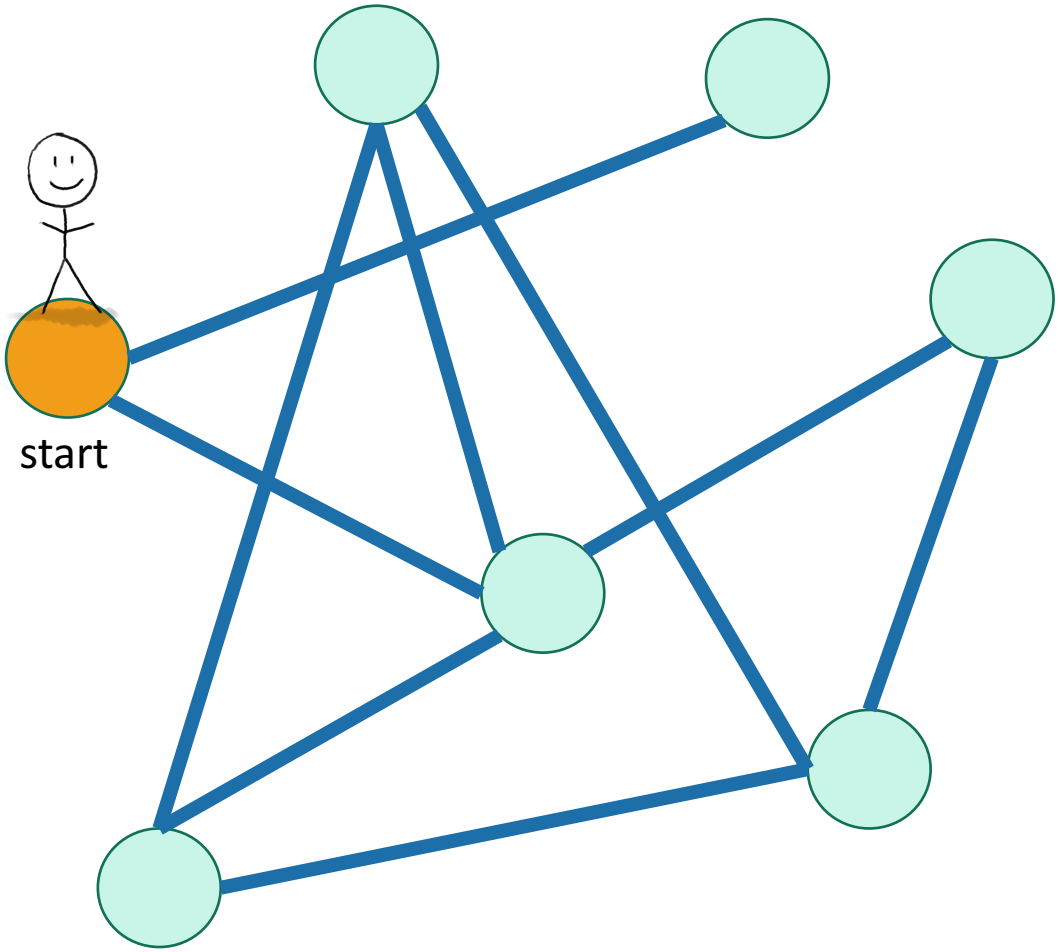
Not been there yet

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



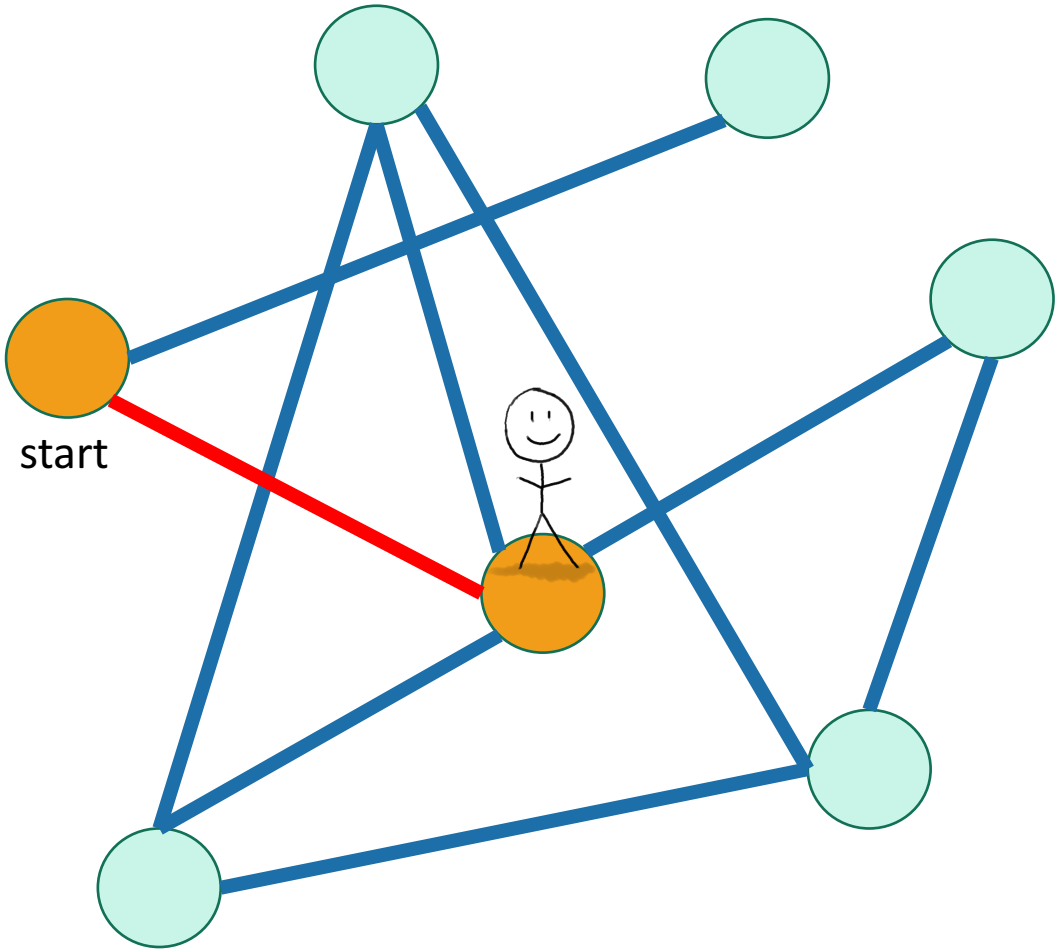
Not been there yet

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



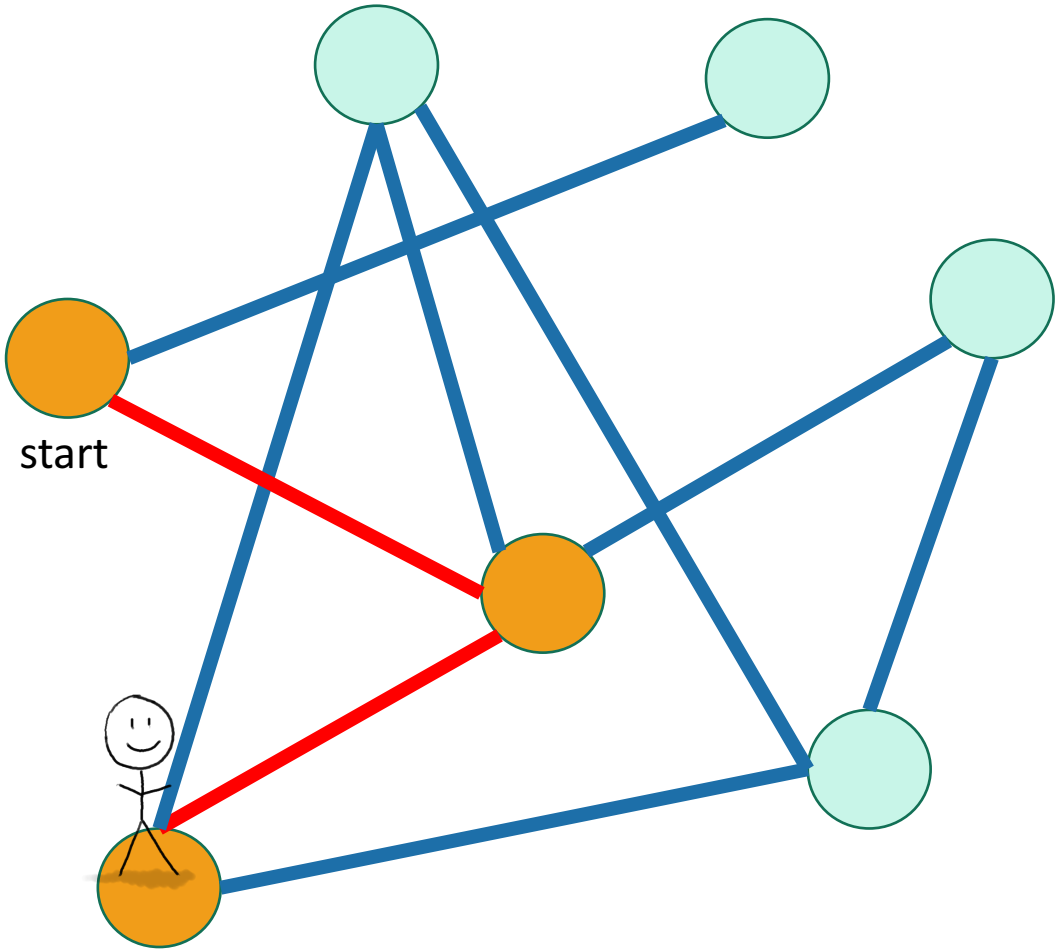
Not been there yet

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



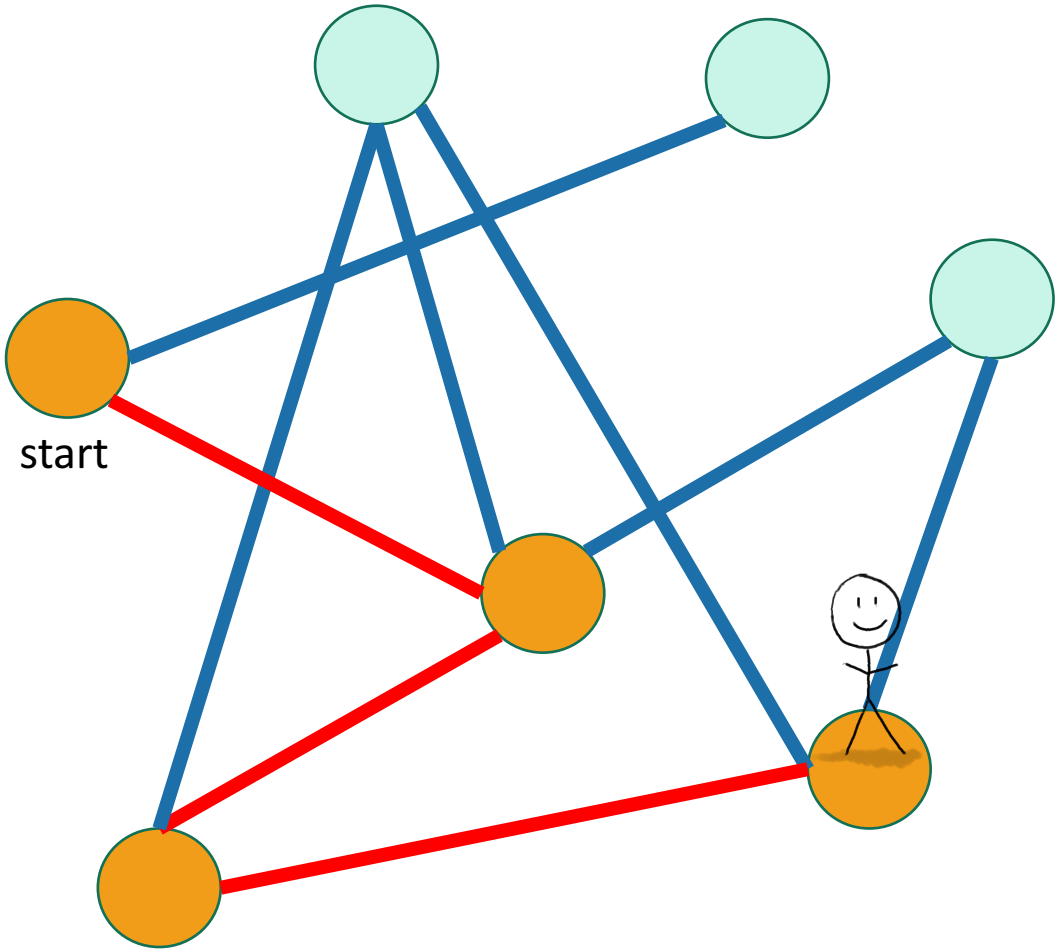
Not been there yet

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



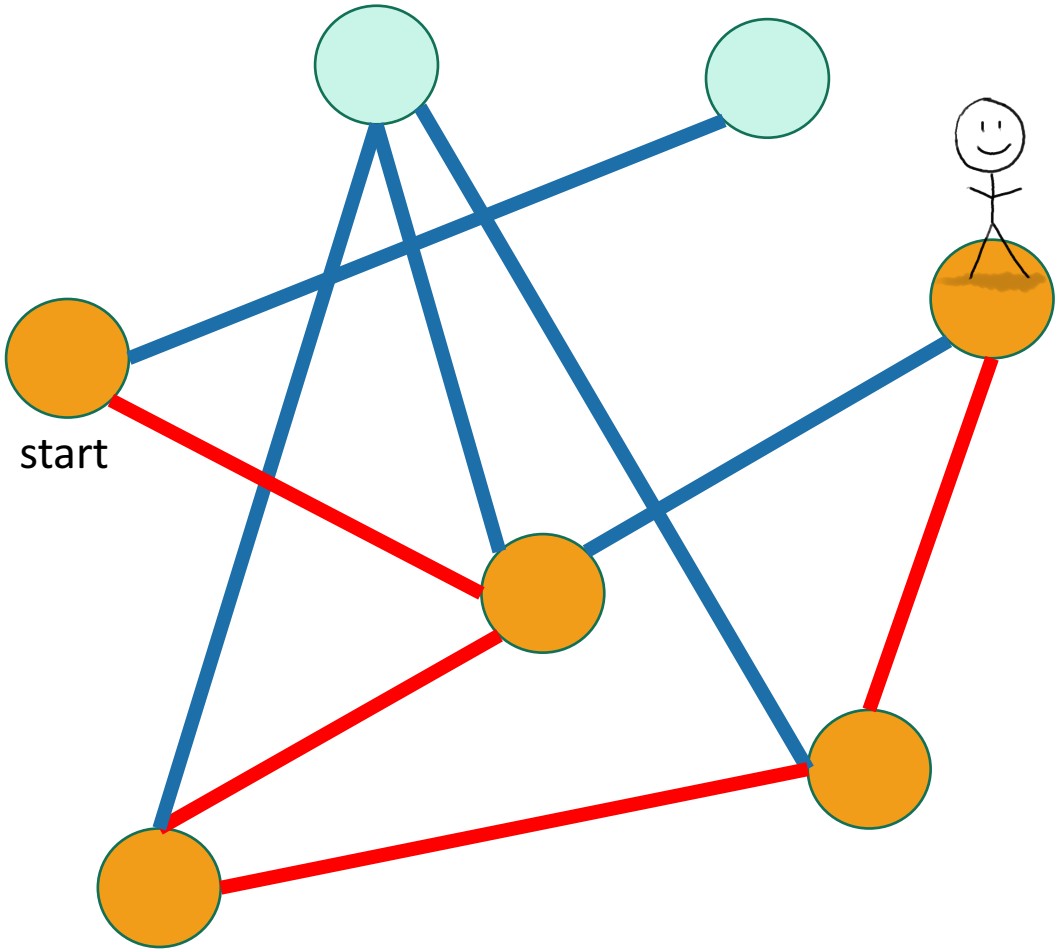
Not been there yet

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



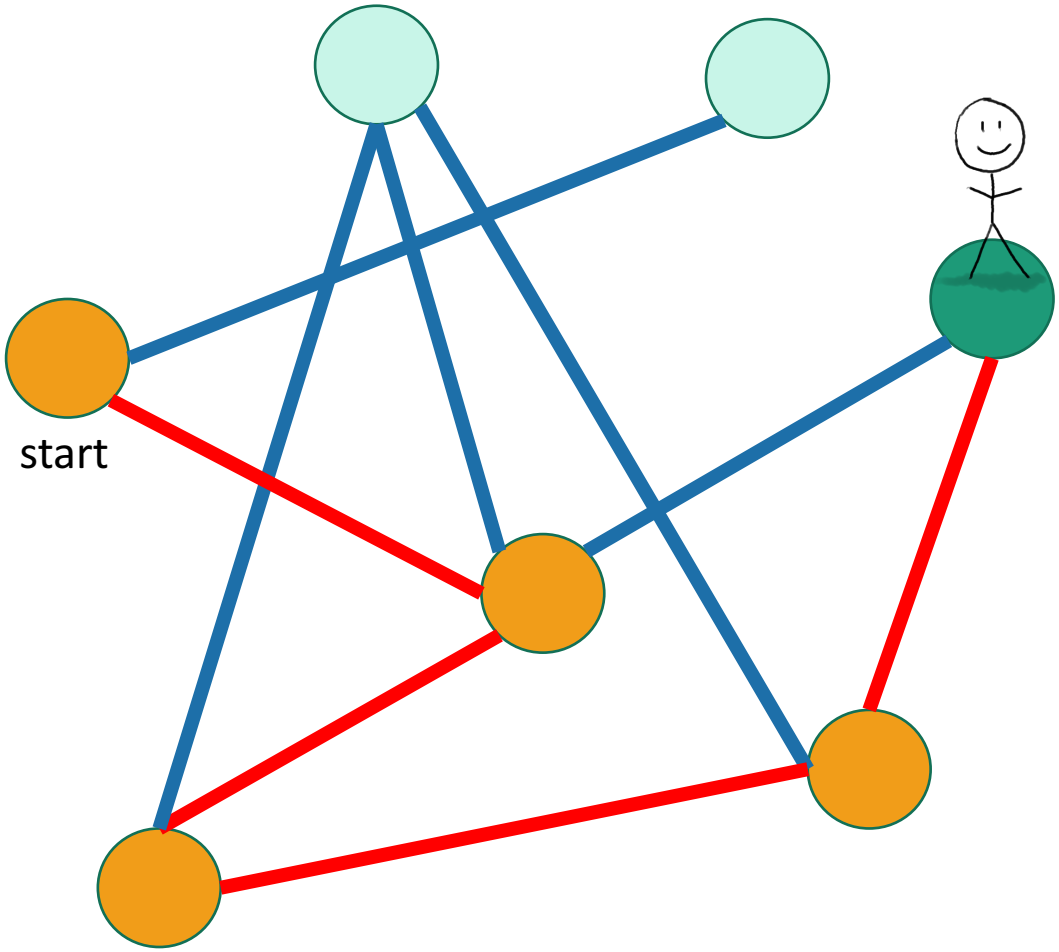
Not been there yet

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



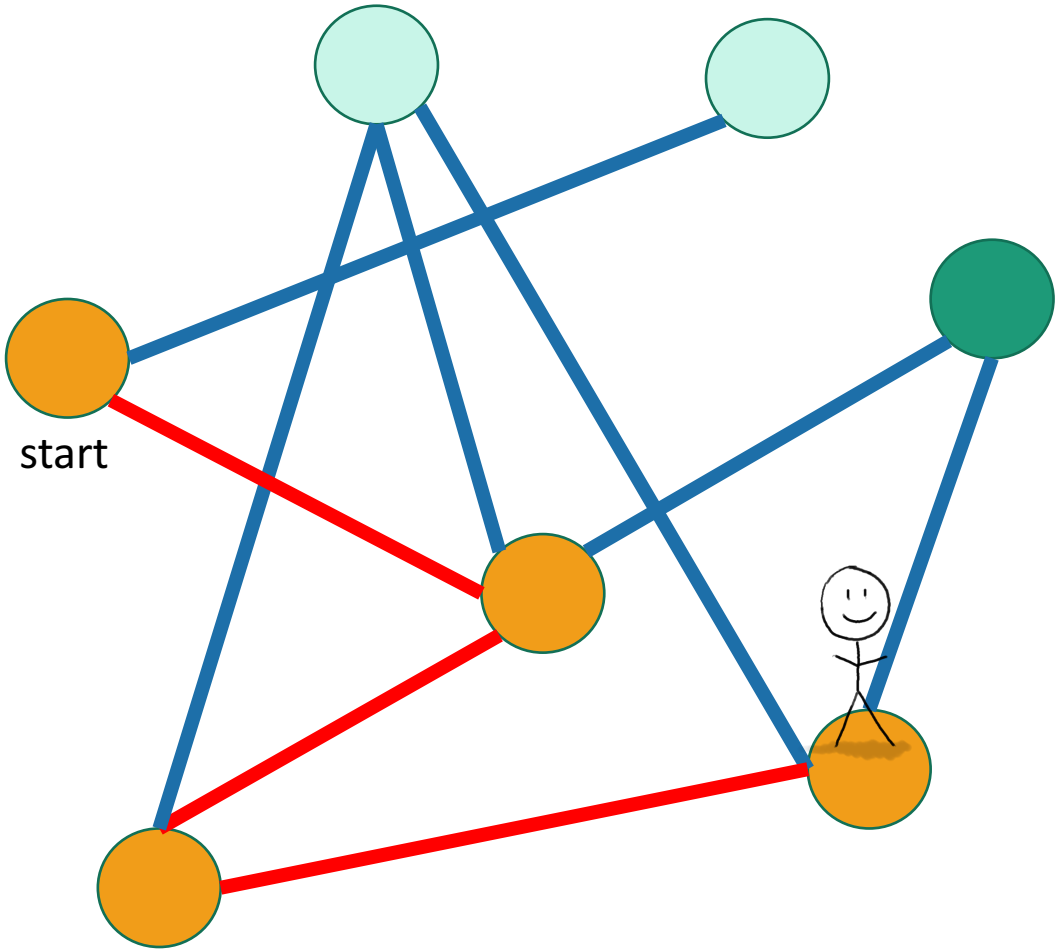
Not been there yet

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



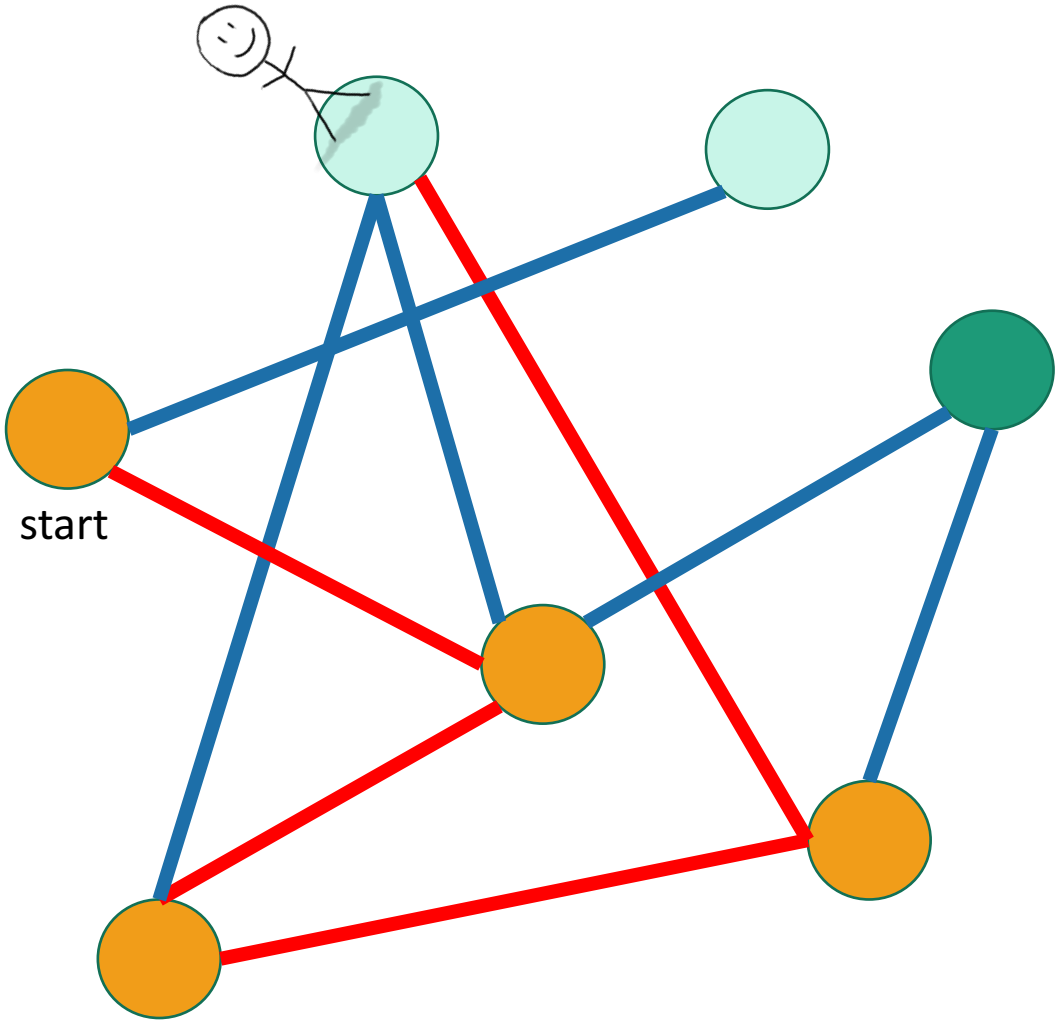
Not been there yet

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



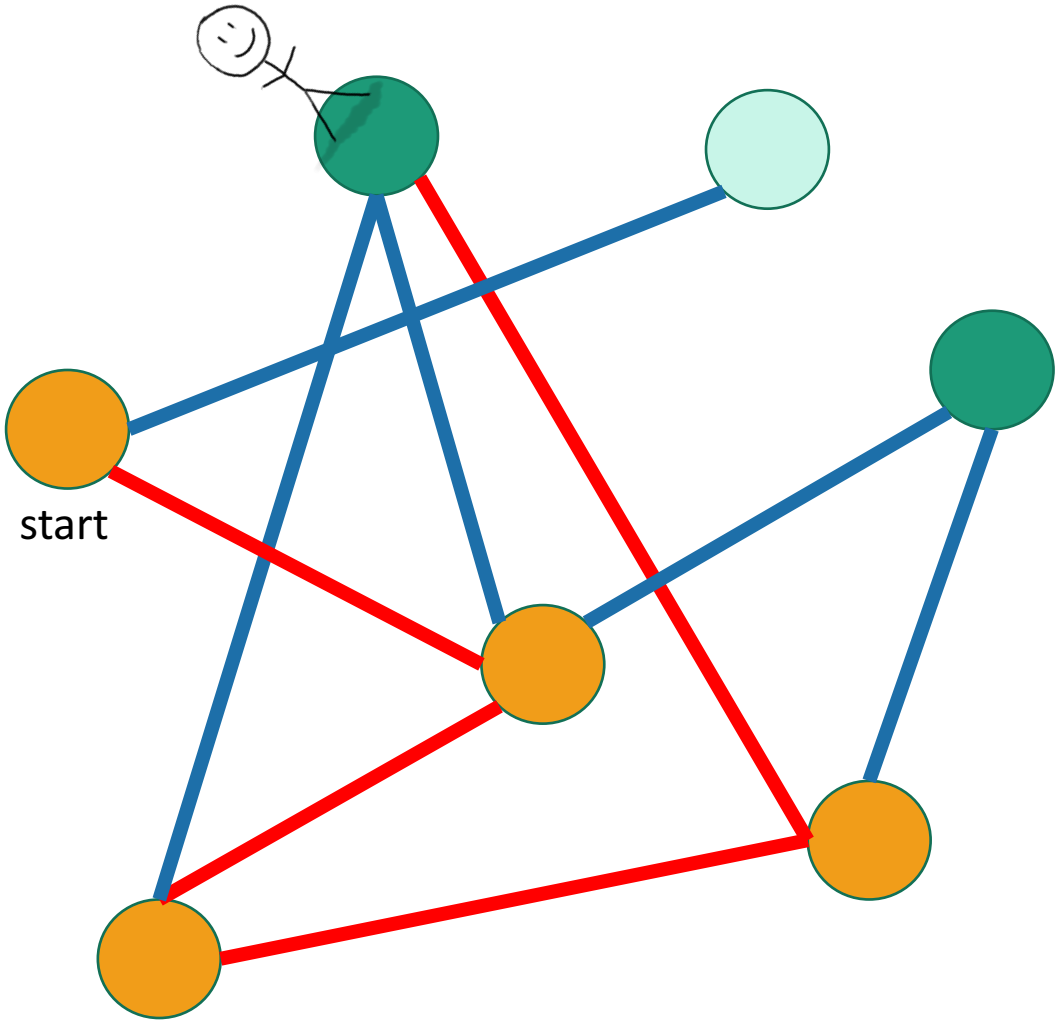
Not been there yet

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



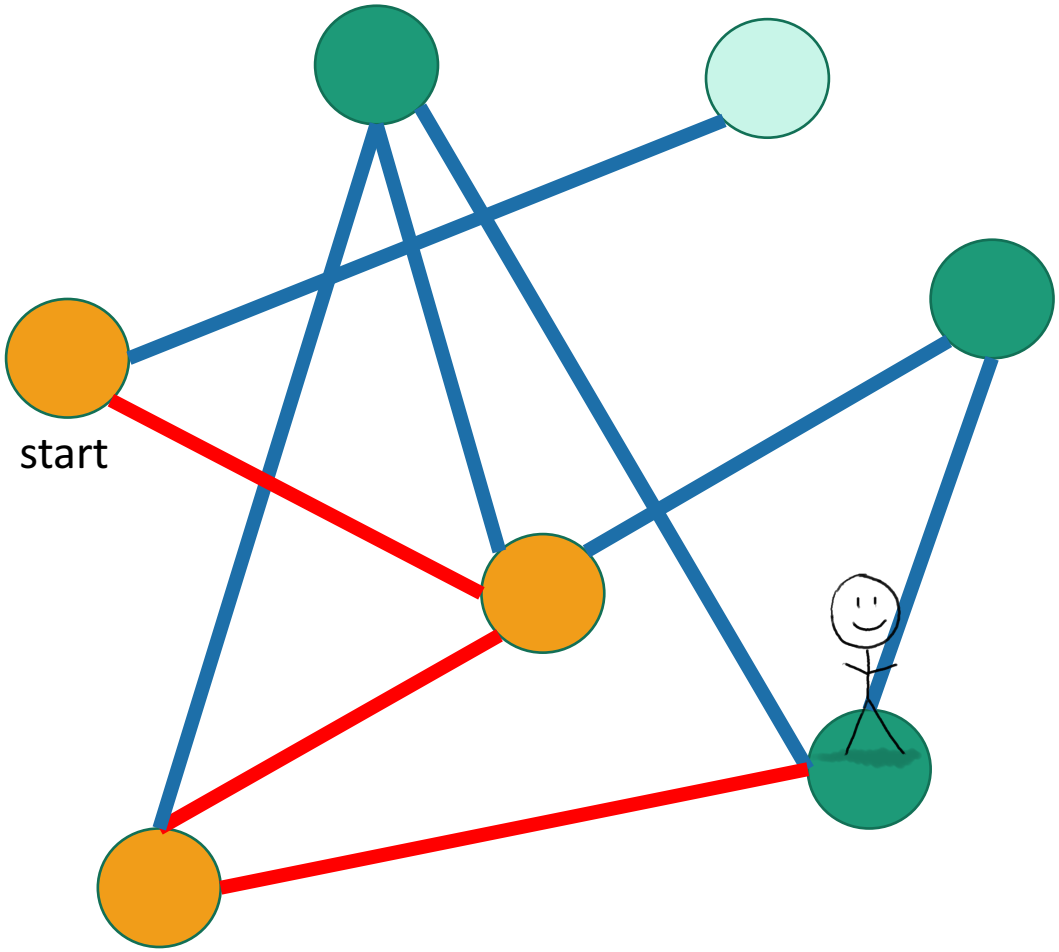
Not been there yet

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



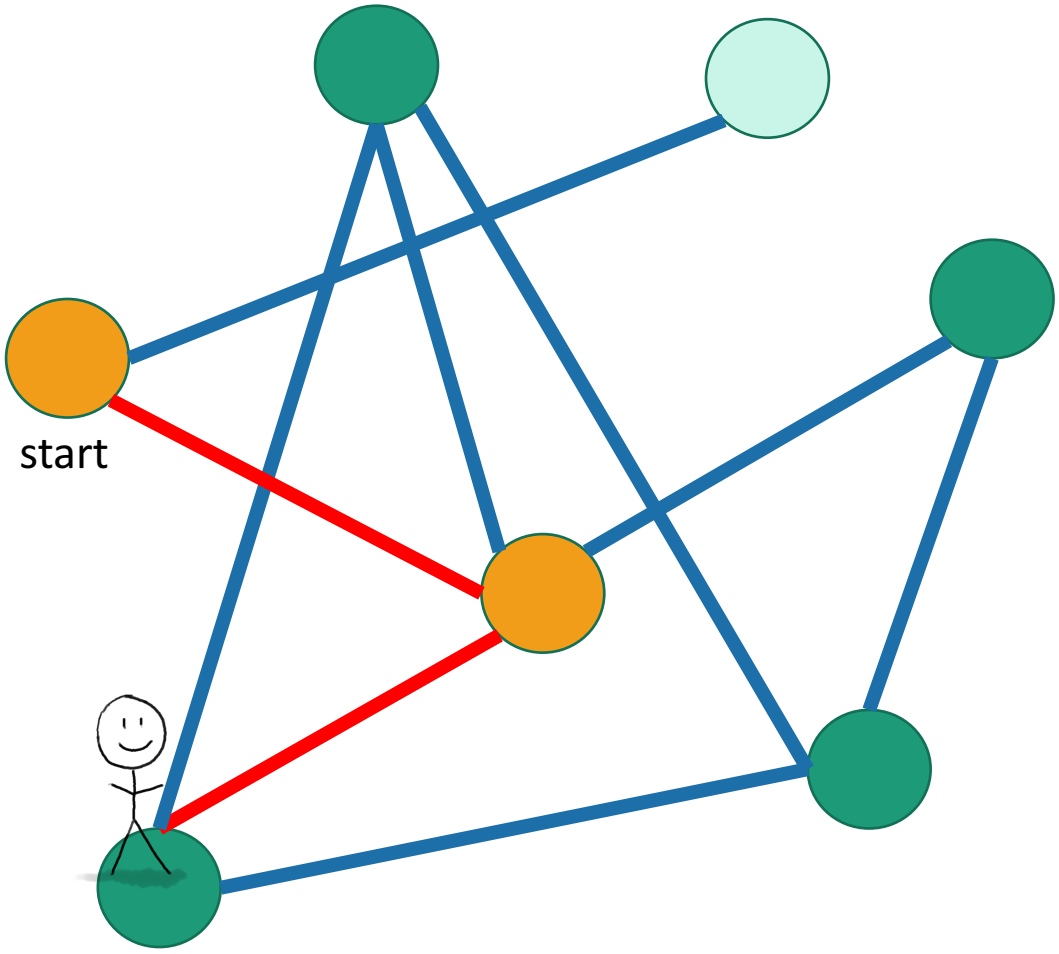
Not been there yet

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



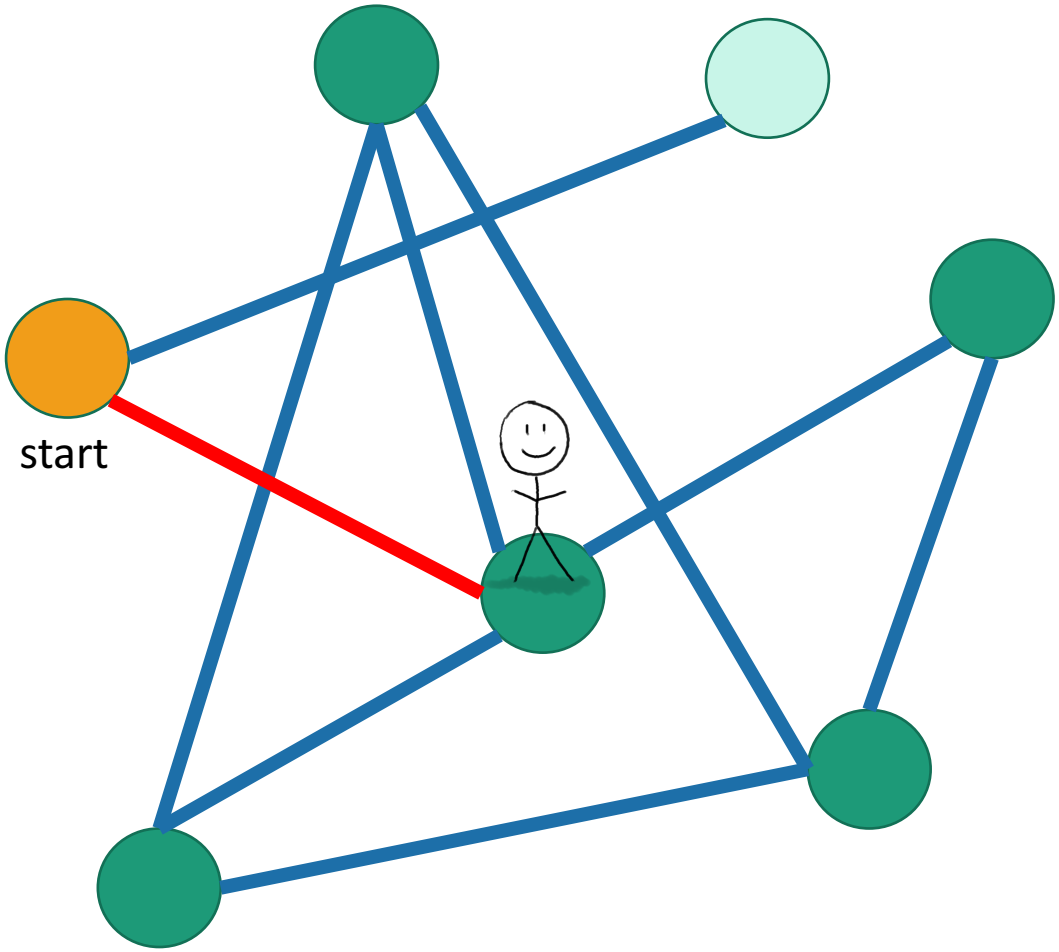
Not been there yet

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



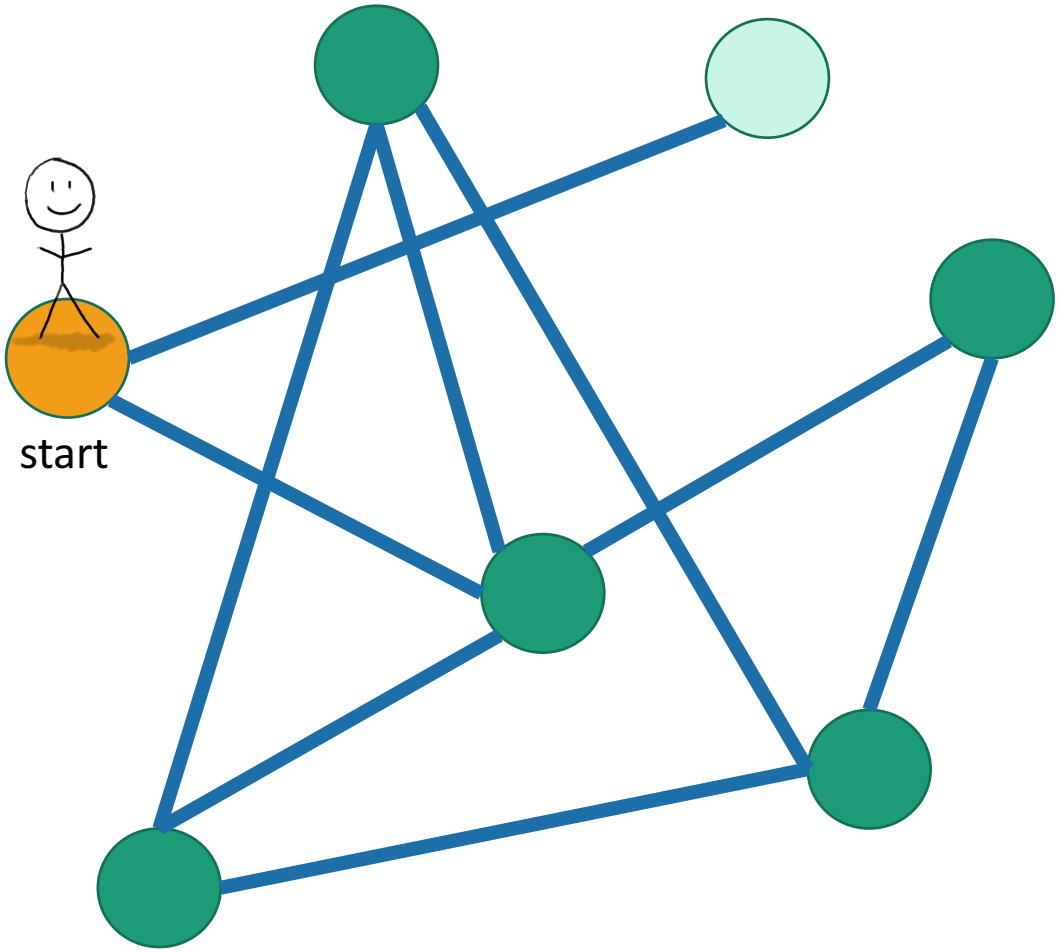
Not been there yet

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



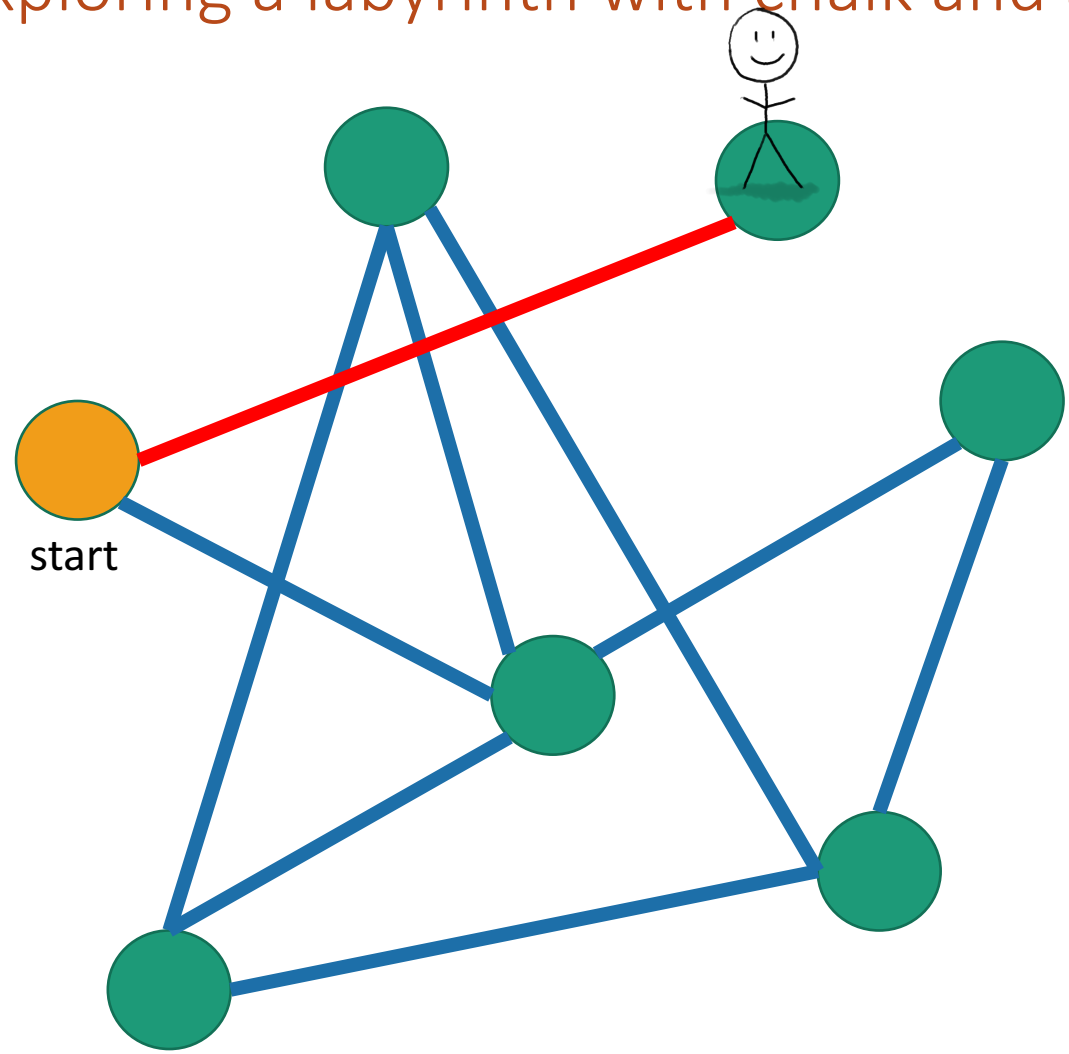
Not been there yet

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



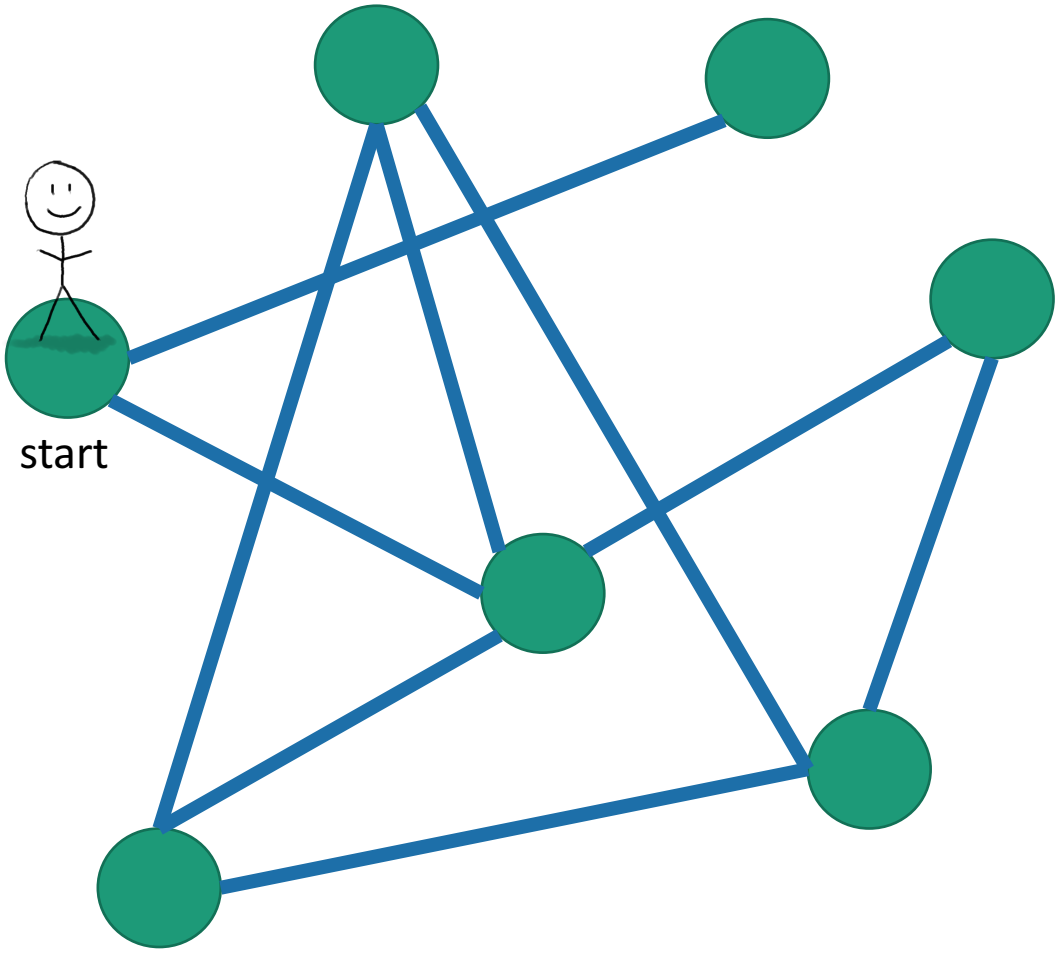
Not been there yet

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

- **DFS**(w, currentTime):
  - w.entryTime = 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

Each node is going to keep track of whether it's unvisited, in progress, or all done.

We'll also keep track of the time at which we started and finished with that node.

Depth First Search

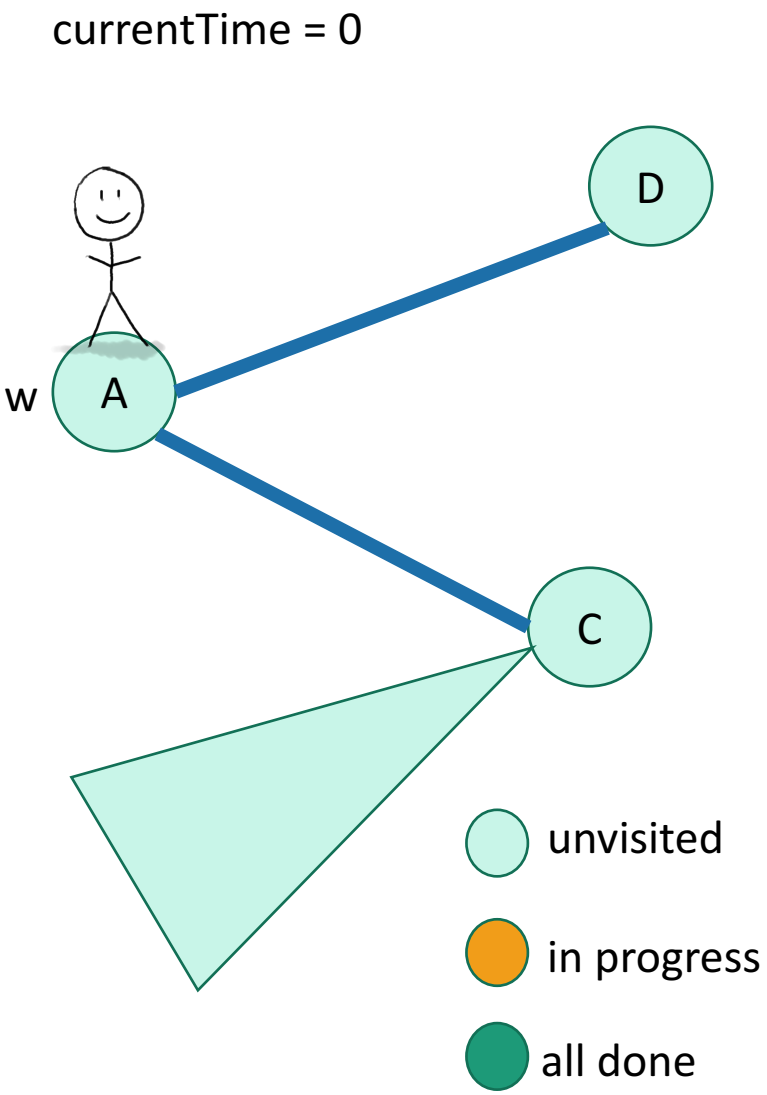
Exploring a labyrinth with pseudocode

- 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**.

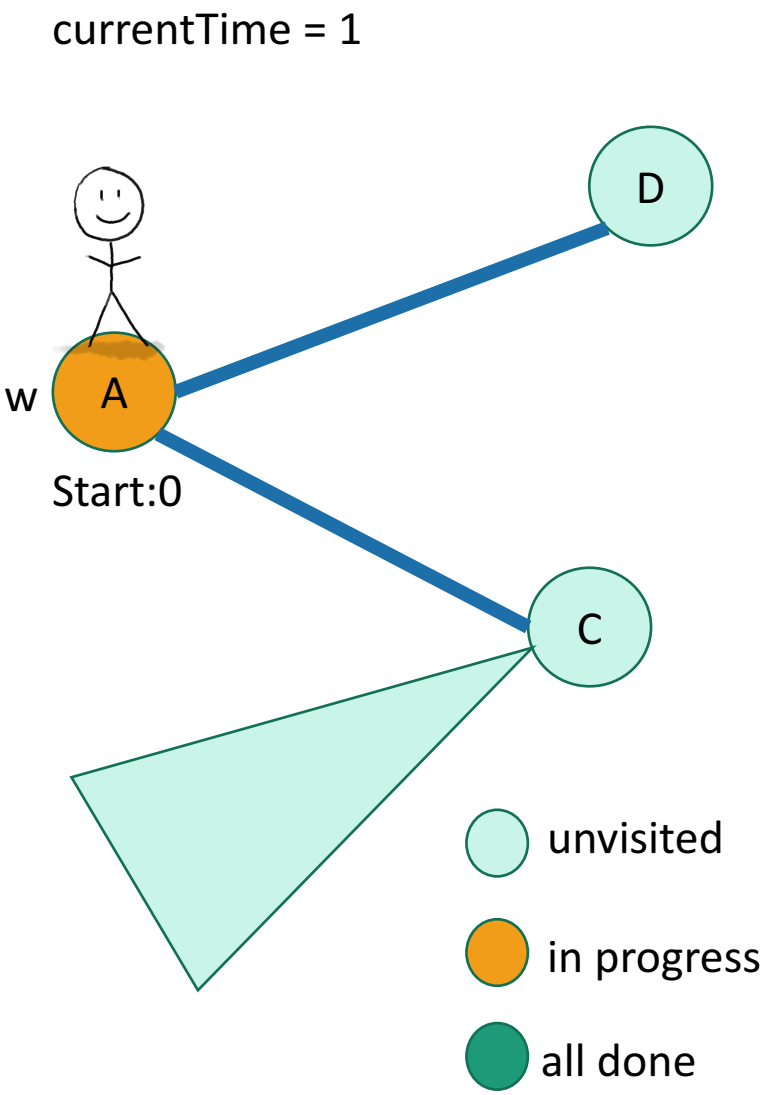
You might have seen other ways to implement DFS than what we are about to go through. This way has more bookkeeping, but more intuition – also, the bookkeeping will be useful later!

Depth First Search



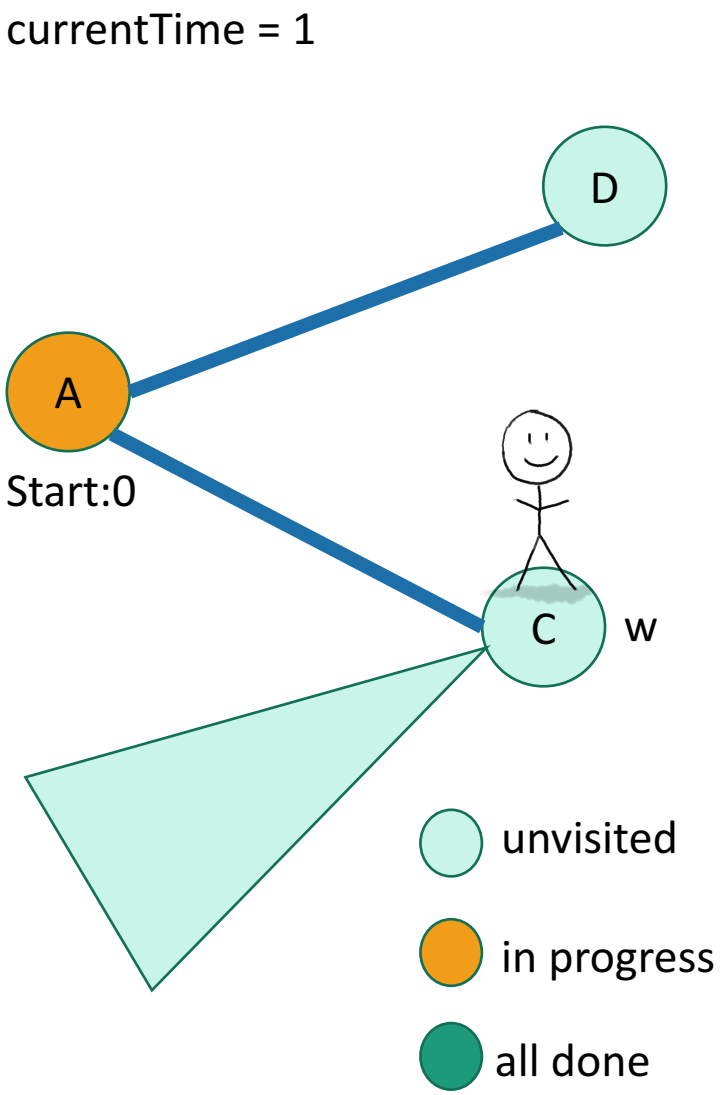
- **DFS**(w, currentTime):
  - w.entryTime = currentTime
  - currentTime ++
  - Mark w as **in progress**.
  - **for** v in w.neighbors:
    - **if** v is **unvisited**:
      - currentTime = **DFS**(v, currentTime)
      - currentTime ++
  - w.finishTime = currentTime
  - Mark w as **all done**
  -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

Depth First Search



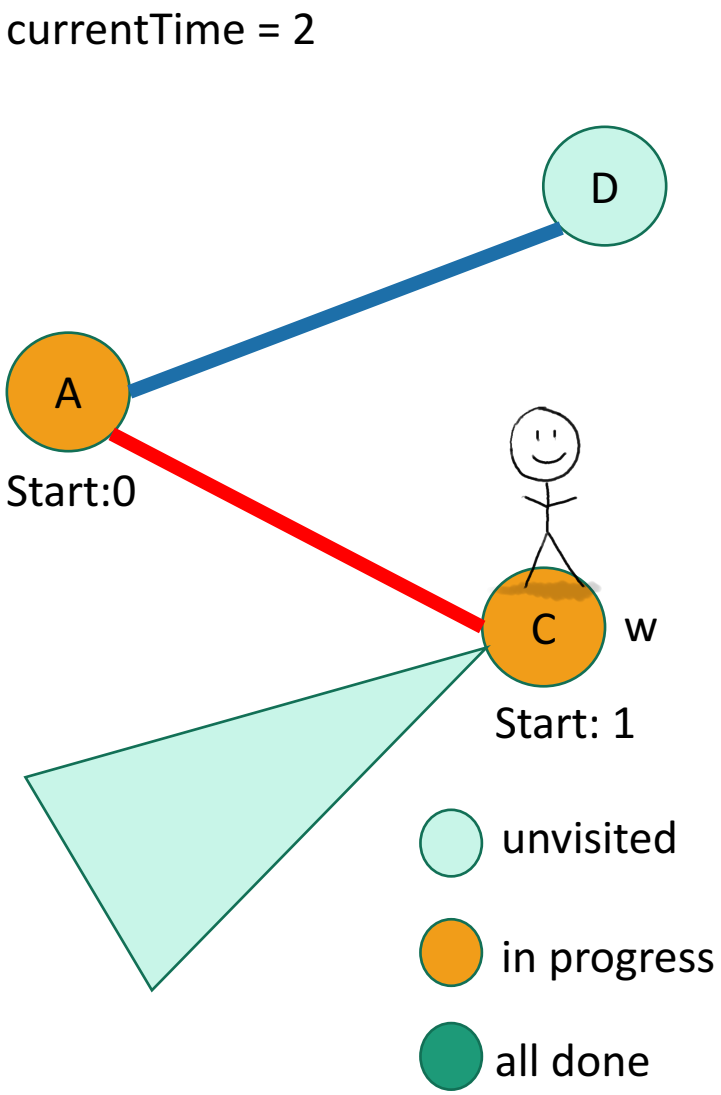
- **DFS**(w, currentTime):
  - w.entryTime = currentTime
  - currentTime ++
  - Mark w as **in progress**.
  - **for** v in w.neighbors:
    - **if** v is **unvisited**:
      - currentTime = **DFS**(v, currentTime)
      - currentTime ++
  - w.finishTime = currentTime
  - Mark w as **all done**
  -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

Depth First Search



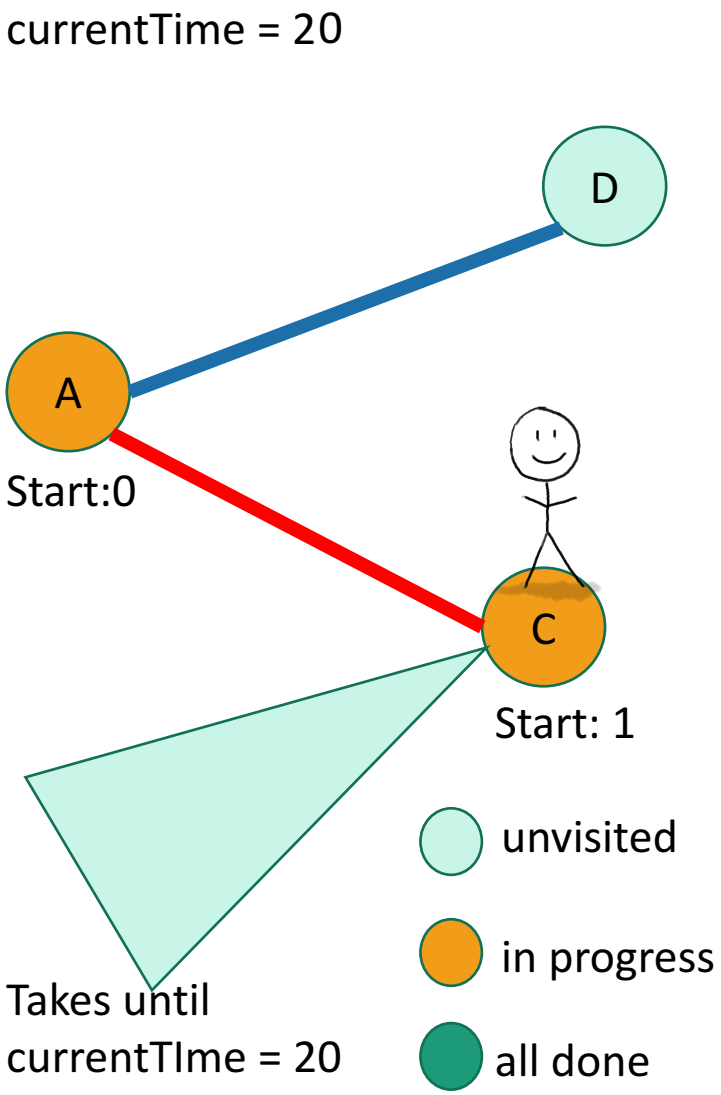
- **DFS**(w, currentTime):
  - w.entryTime = currentTime
  - currentTime ++
  - Mark w as **in progress**.
  - **for** v in w.neighbors:
    - **if** v is **unvisited**:
      - currentTime = **DFS**(v, currentTime)
      - currentTime ++
  - w.finishTime = currentTime
  - Mark w as **all done**
  -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

Depth First Search



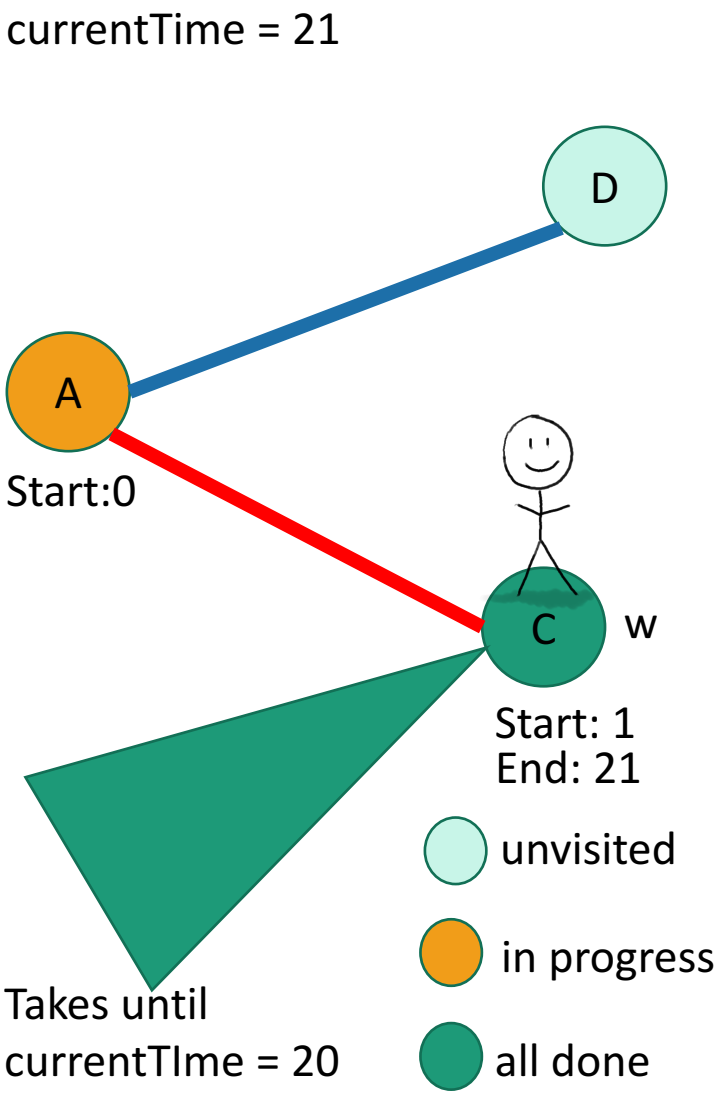
- **DFS**(w, currentTime):
  - w.entryTime = currentTime
  - currentTime ++
  - Mark w as **in progress**.
  - **for** v in w.neighbors:
    - **if** v is **unvisited**:
      - currentTime = **DFS**(v, currentTime)
      - currentTime ++
  - w.finishTime = currentTime
  - Mark w as **all done**
  -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

Depth First Search



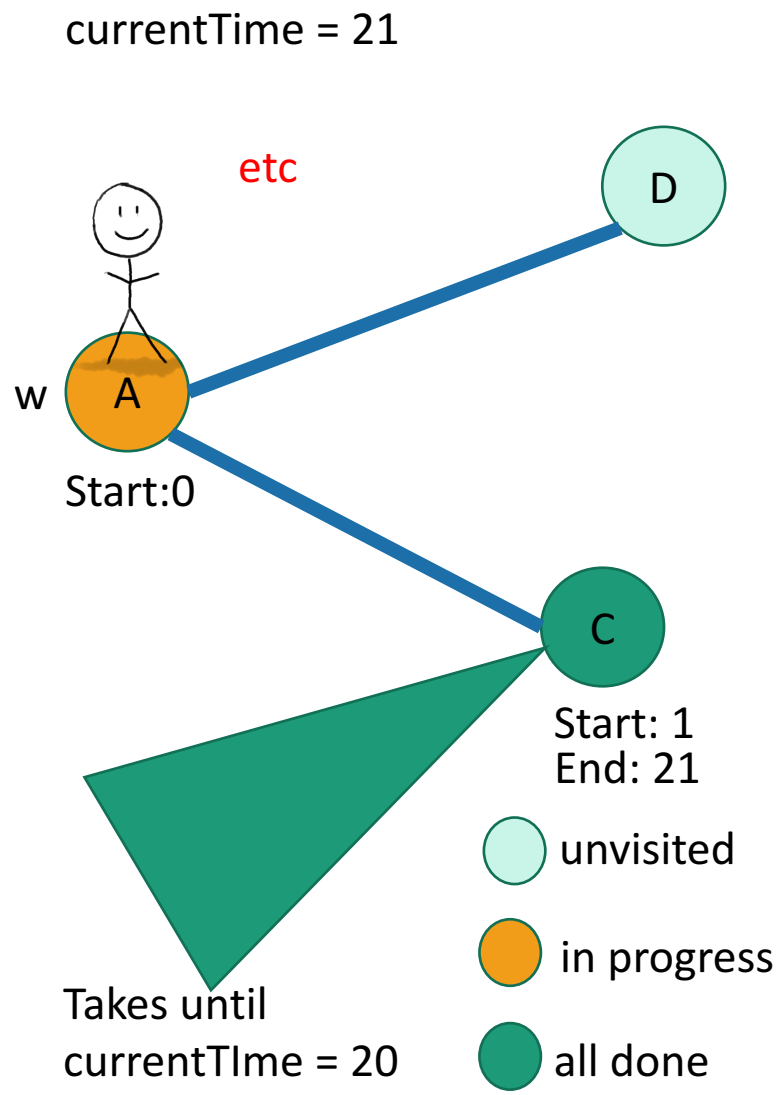
- **DFS**(w, currentTime):
  - w.entryTime = currentTime
  - currentTime ++
  - Mark w as in progress.
  - **for** v in w.neighbors:
    - **if** v is unvisited:
      - currentTime = **DFS**(v, currentTime)
      - currentTime ++
  - w.finishTime = currentTime
  - Mark w as **all done**
  - **return**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

Depth First Search



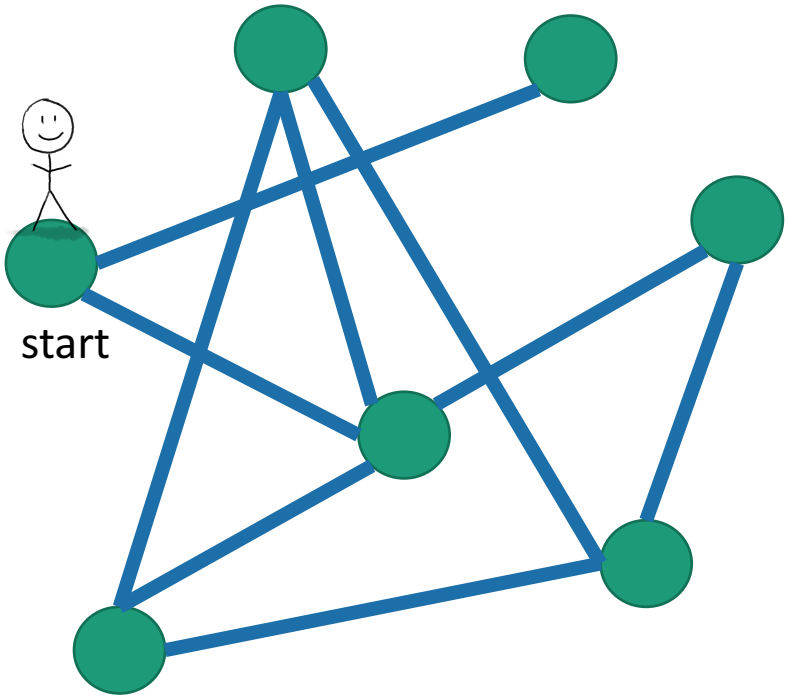
- **DFS**(w, currentTime):
  - w.startTime = currentTime
  - currentTime ++
  - Mark w as in progress.
  - **for** v in w.neighbors:
    - **if** v is unvisited:
      - currentTime = **DFS**(v, currentTime)
      - currentTime ++
  - w.finishTime = currentTime
  - Mark w as **all done**
  - **return**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

Depth First Search

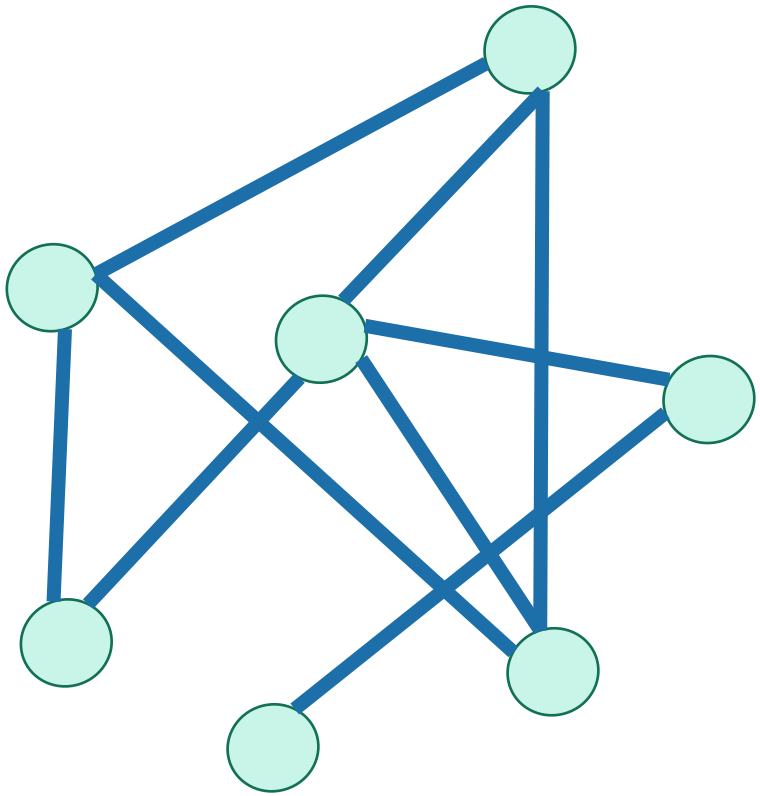


- **DFS**(w, currentTime):
  - w.startTime = currentTime
  - currentTime ++
  - Mark w as **in progress**.
  - **for** v in w.neighbors:
    - **if** v is **unvisited**:
      - currentTime = **DFS**(v, currentTime)
      - currentTime ++
  - w.finishTime = currentTime
  - Mark w as **all done**
  -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

DFS finds all the nodes reachable from the starting point



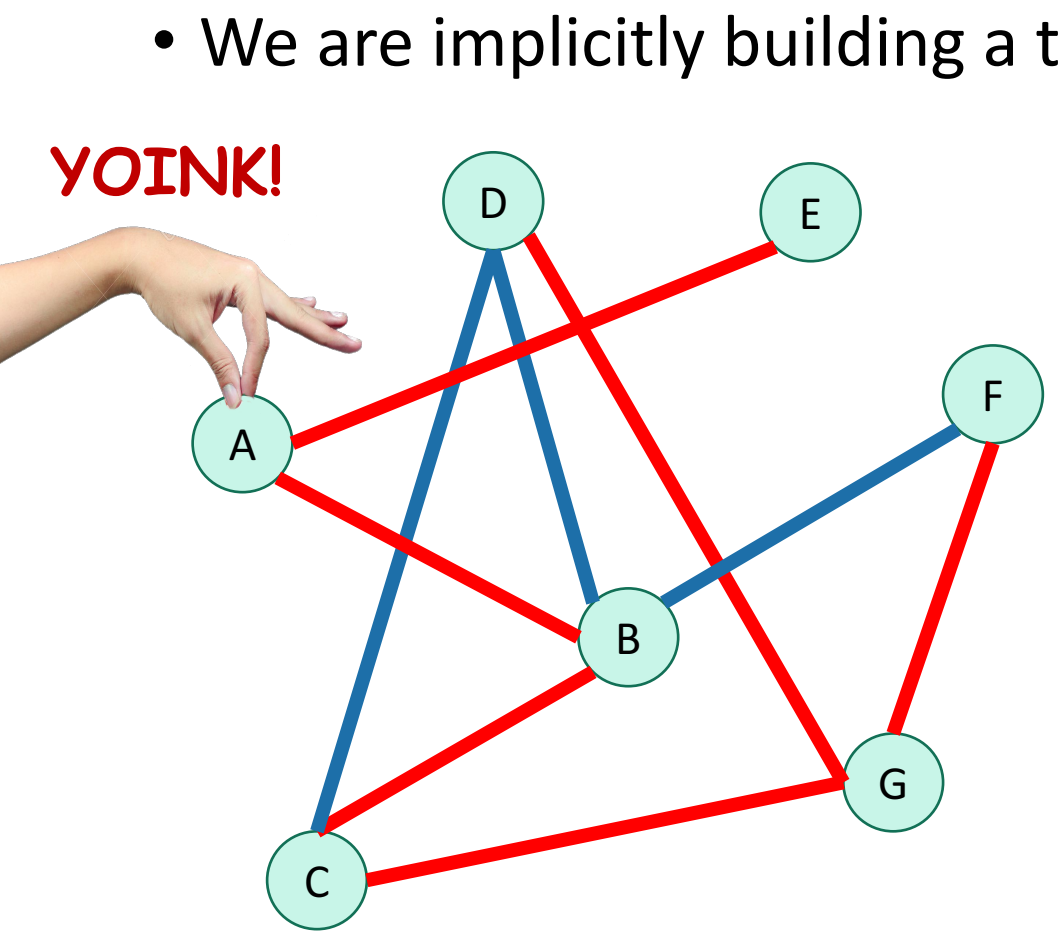
In an undirected graph, this is called a **connected component.**

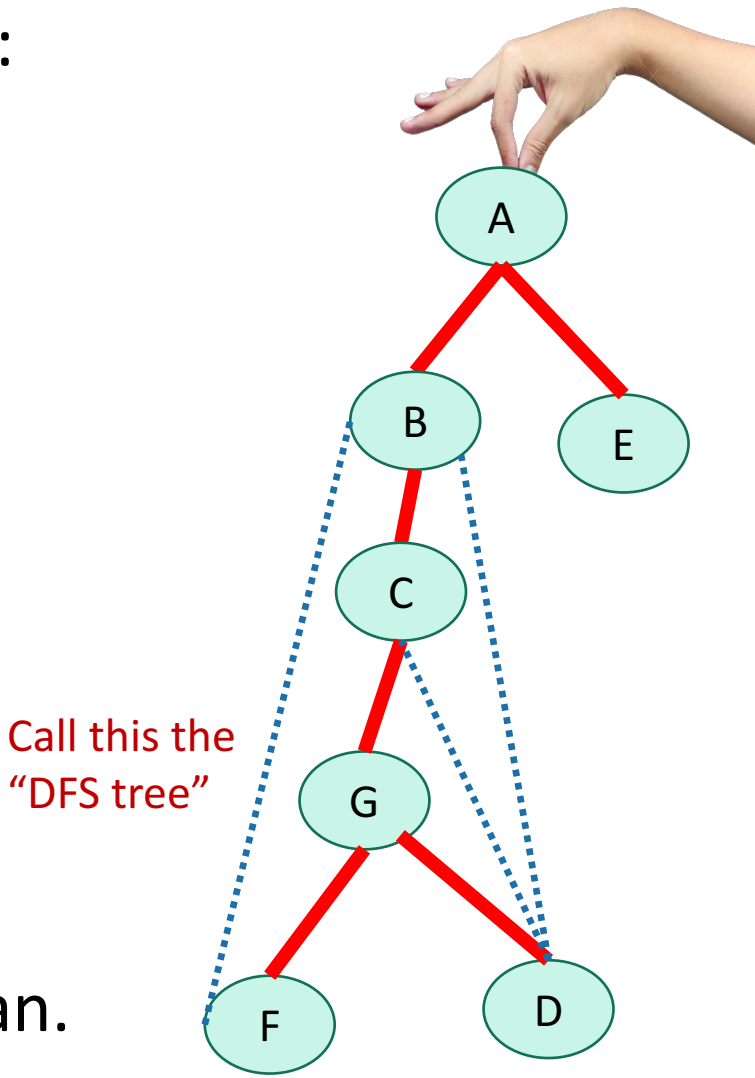


One application: finding connected components.

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.

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…

$$O(m)$$

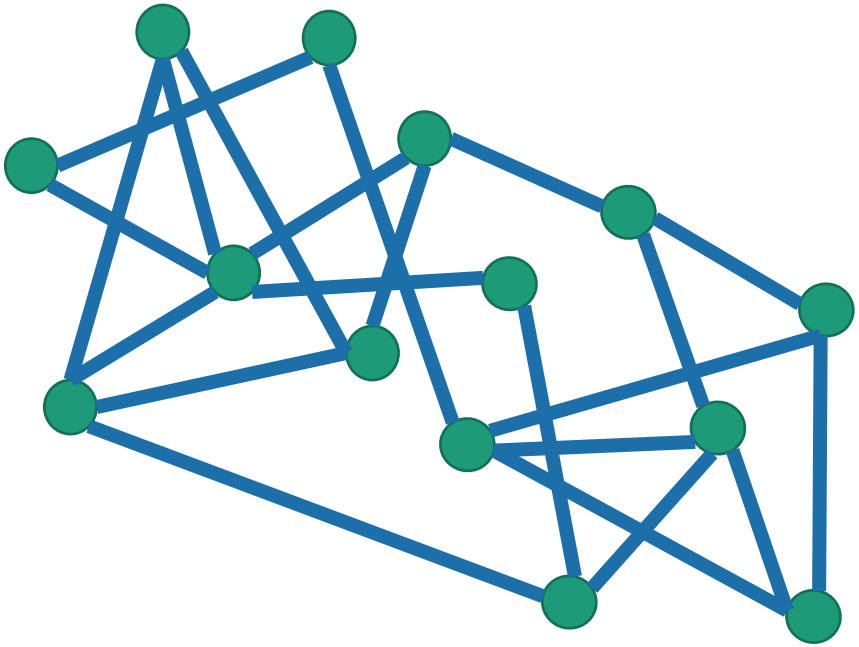
- (Assuming we are using the linked-list representation)

Running time

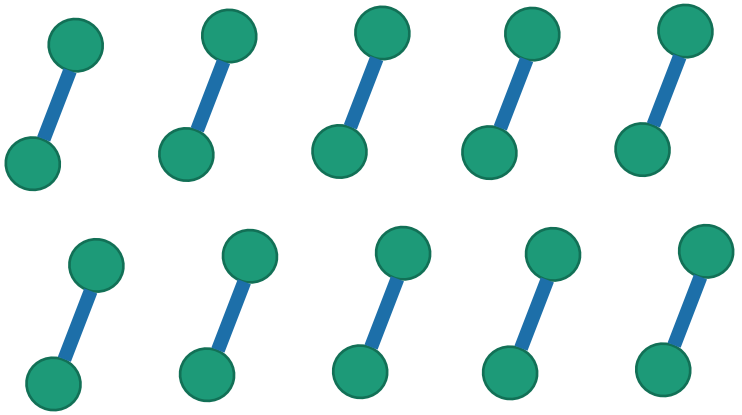
To explore the whole thing

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

$$O(n + m)$$

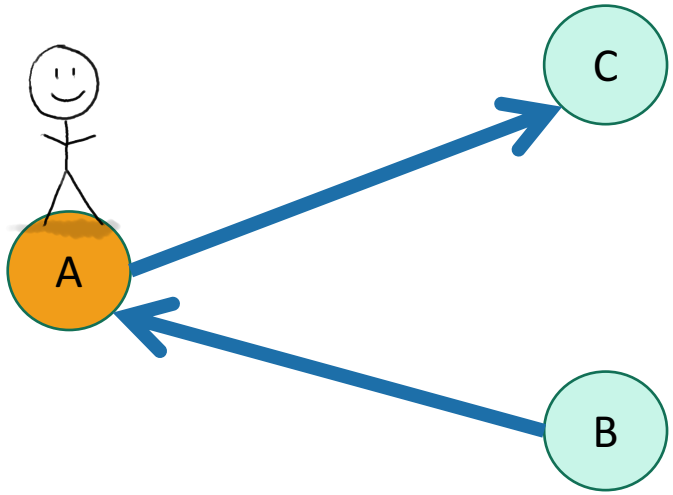


or



You check:

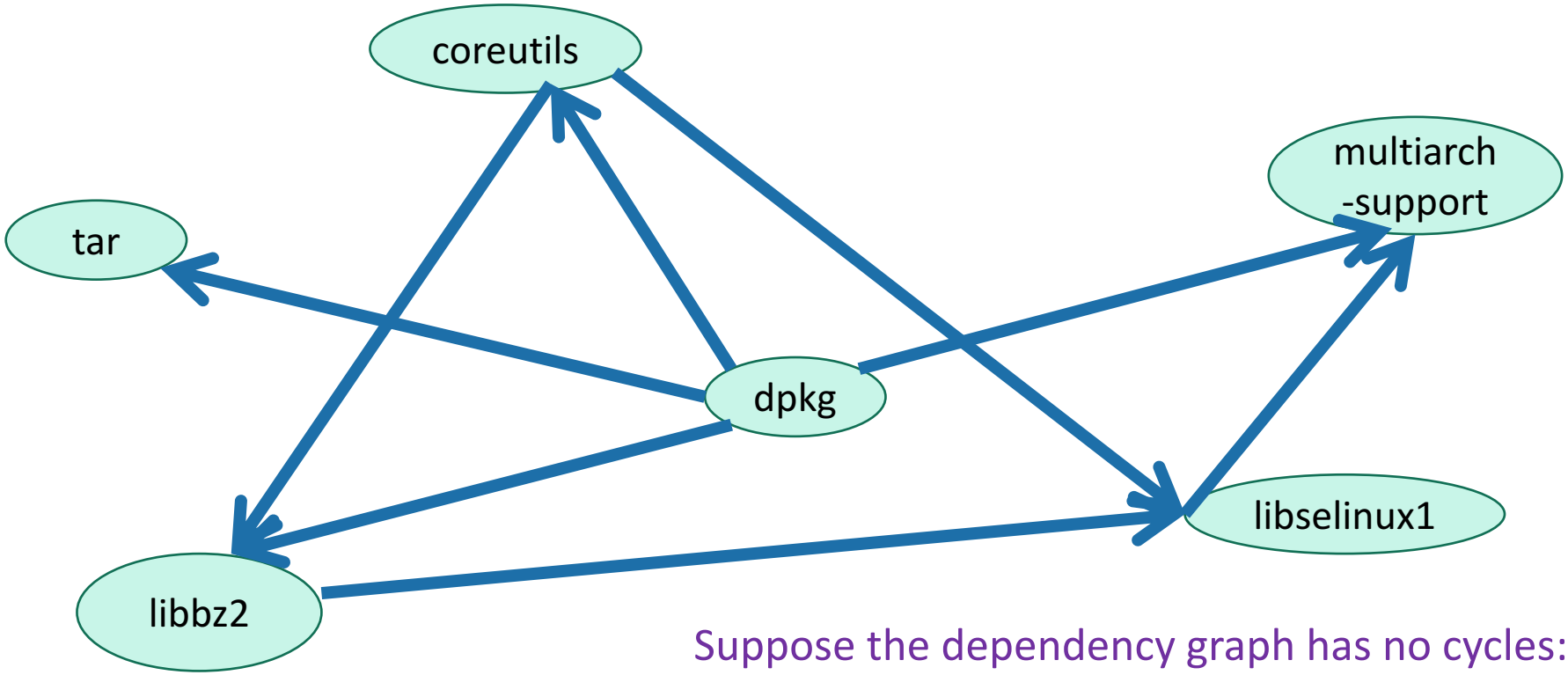
DFS works fine on directed graphs too!



Only walk to C, not to B.

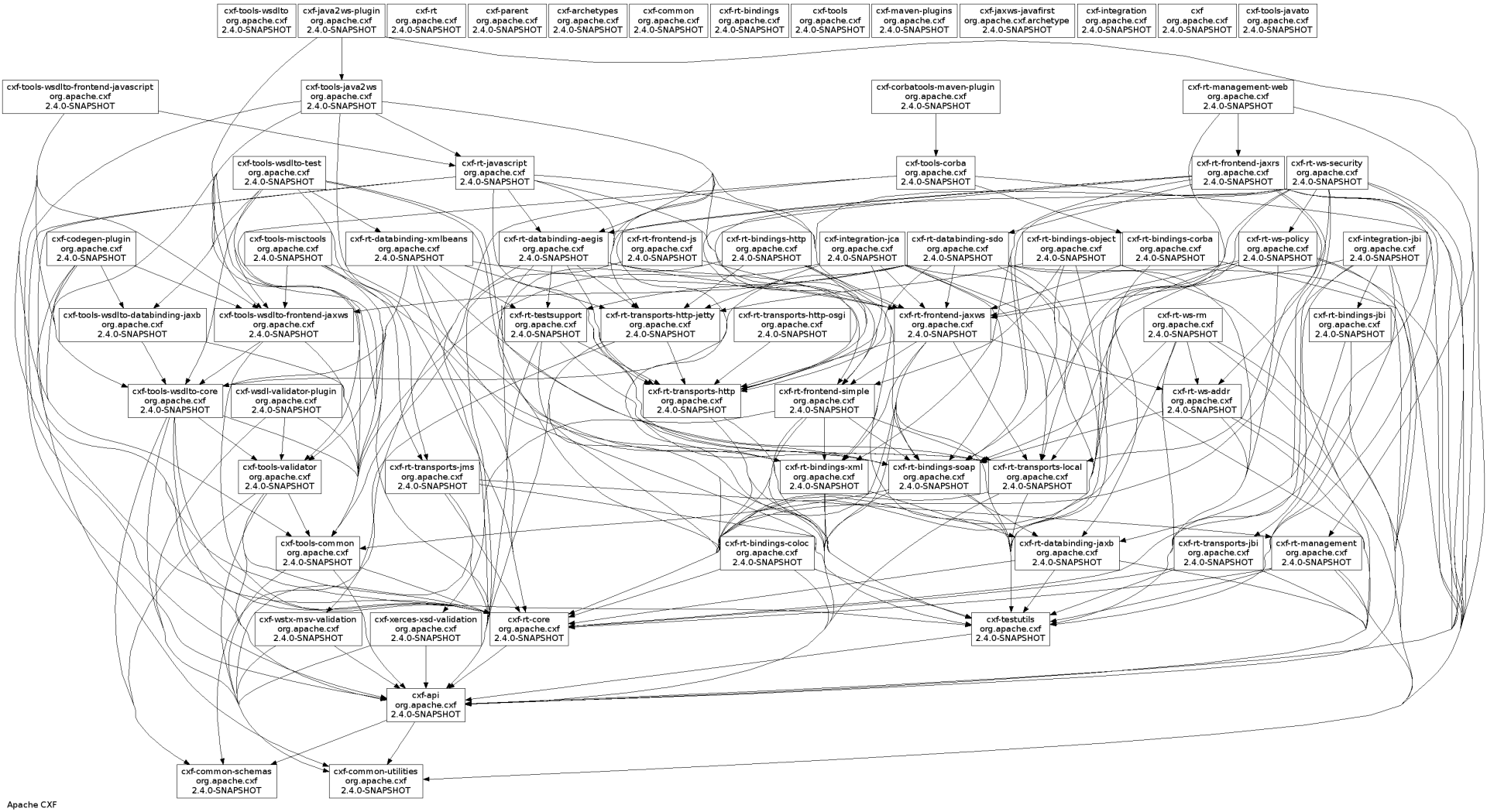
Application: topological sorting

- Example: package dependency graph
- Question: in what order should I install packages?



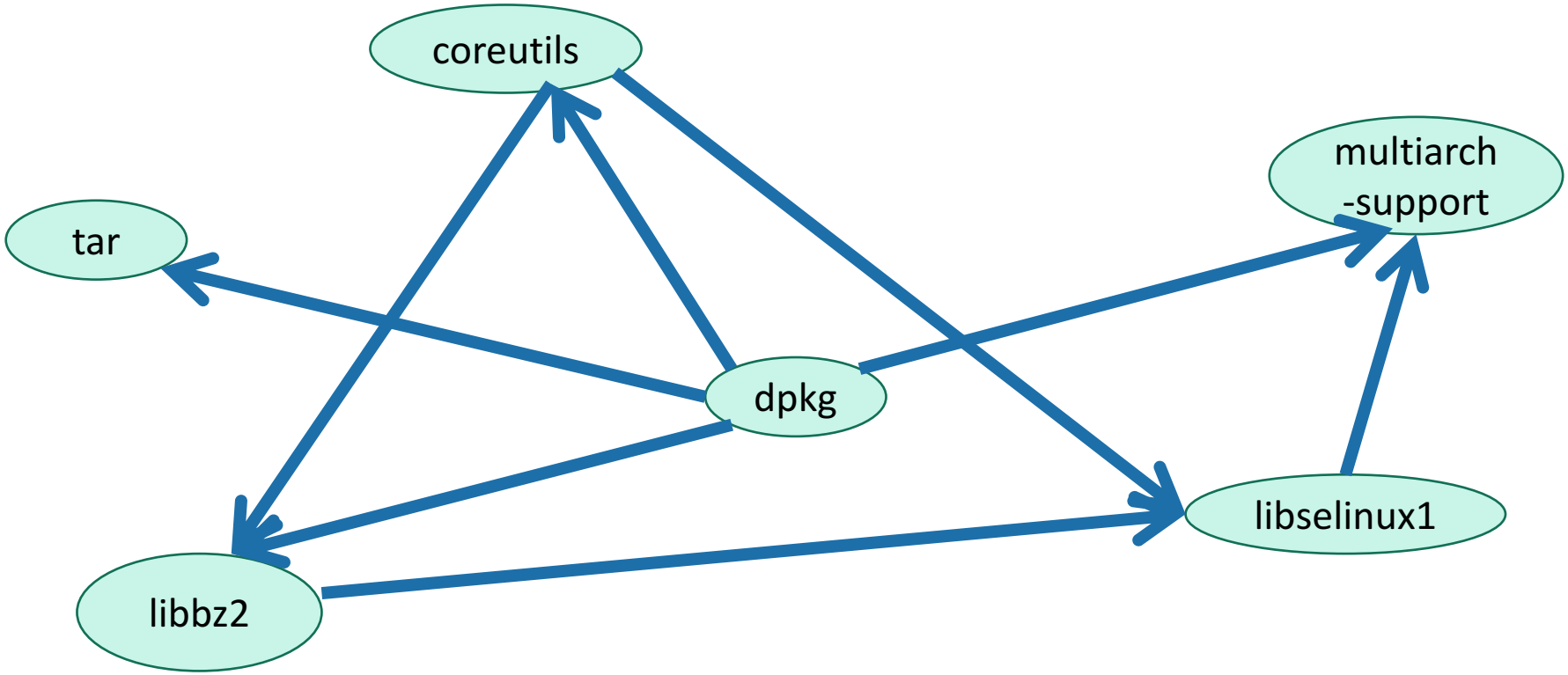
Suppose the dependency graph has no cycles:
it is a **Directed Acyclic Graph (DAG)**

Can’t always eyeball it.



Application: topological sorting

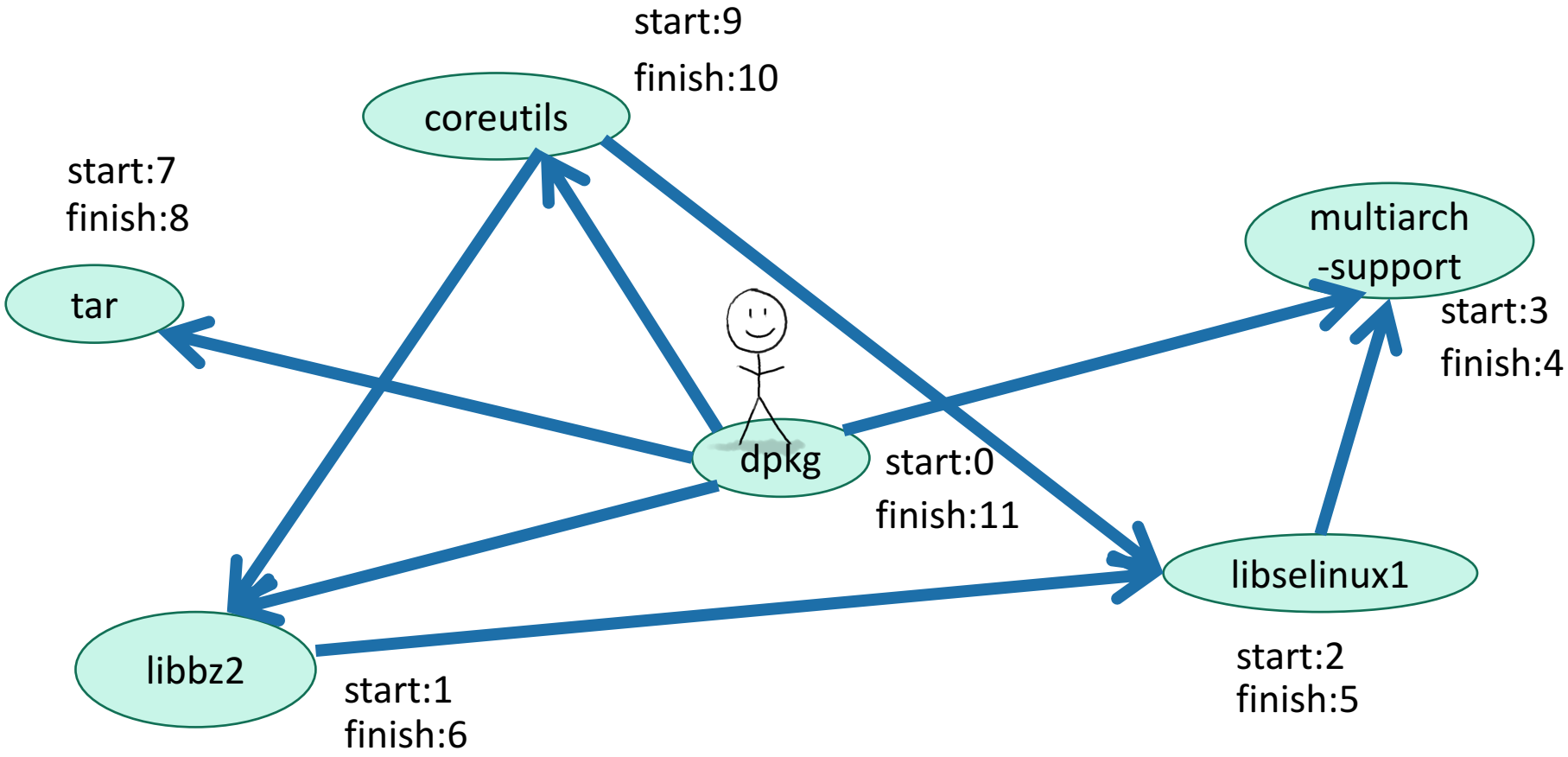
- Example: package dependency graph
- Question: in what order should I install packages?



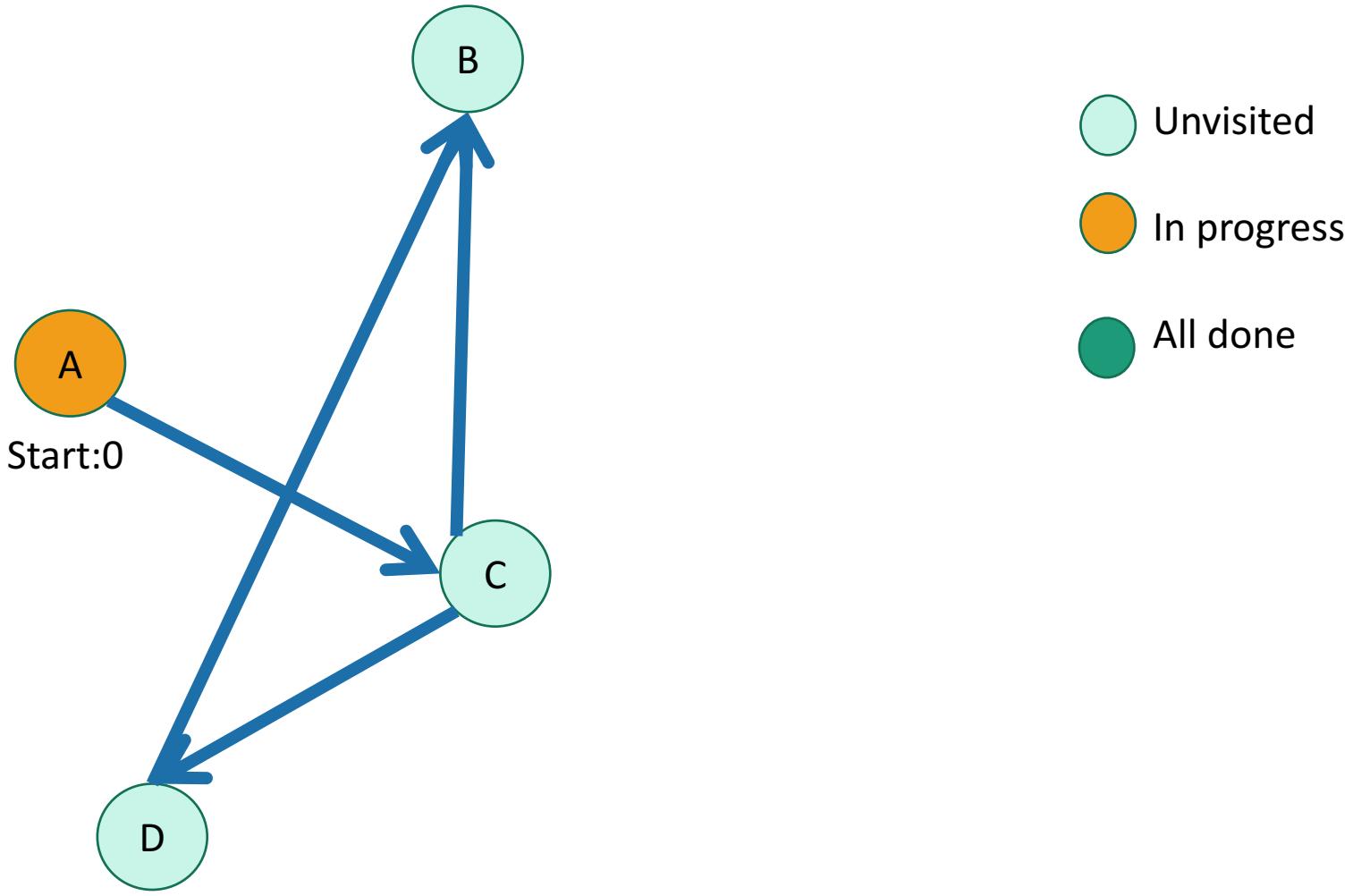
Let's do DFS

Observations:

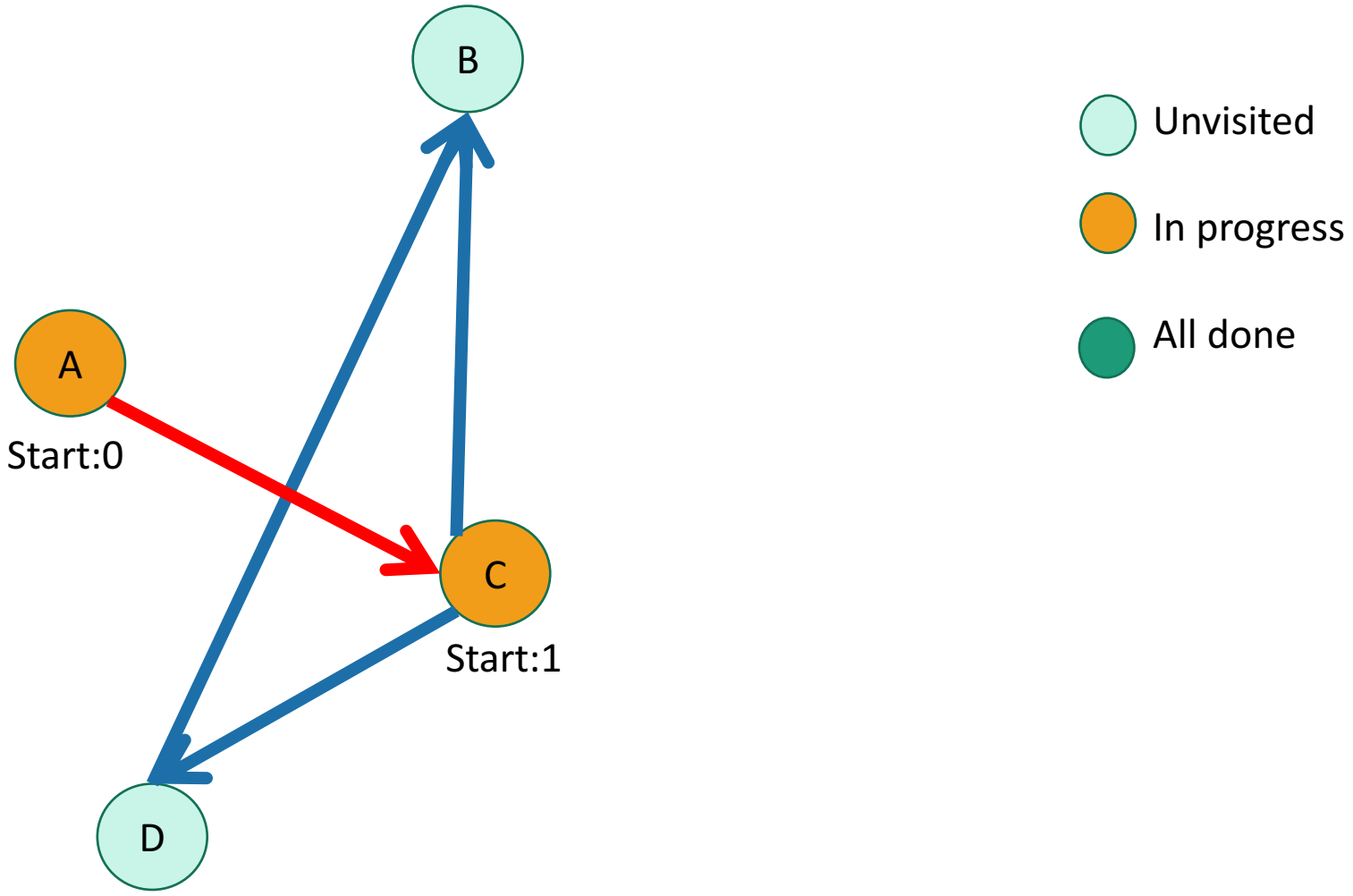
- The start times don't seem that useful.
- But the packages we should include earlier have larger **finish times.**



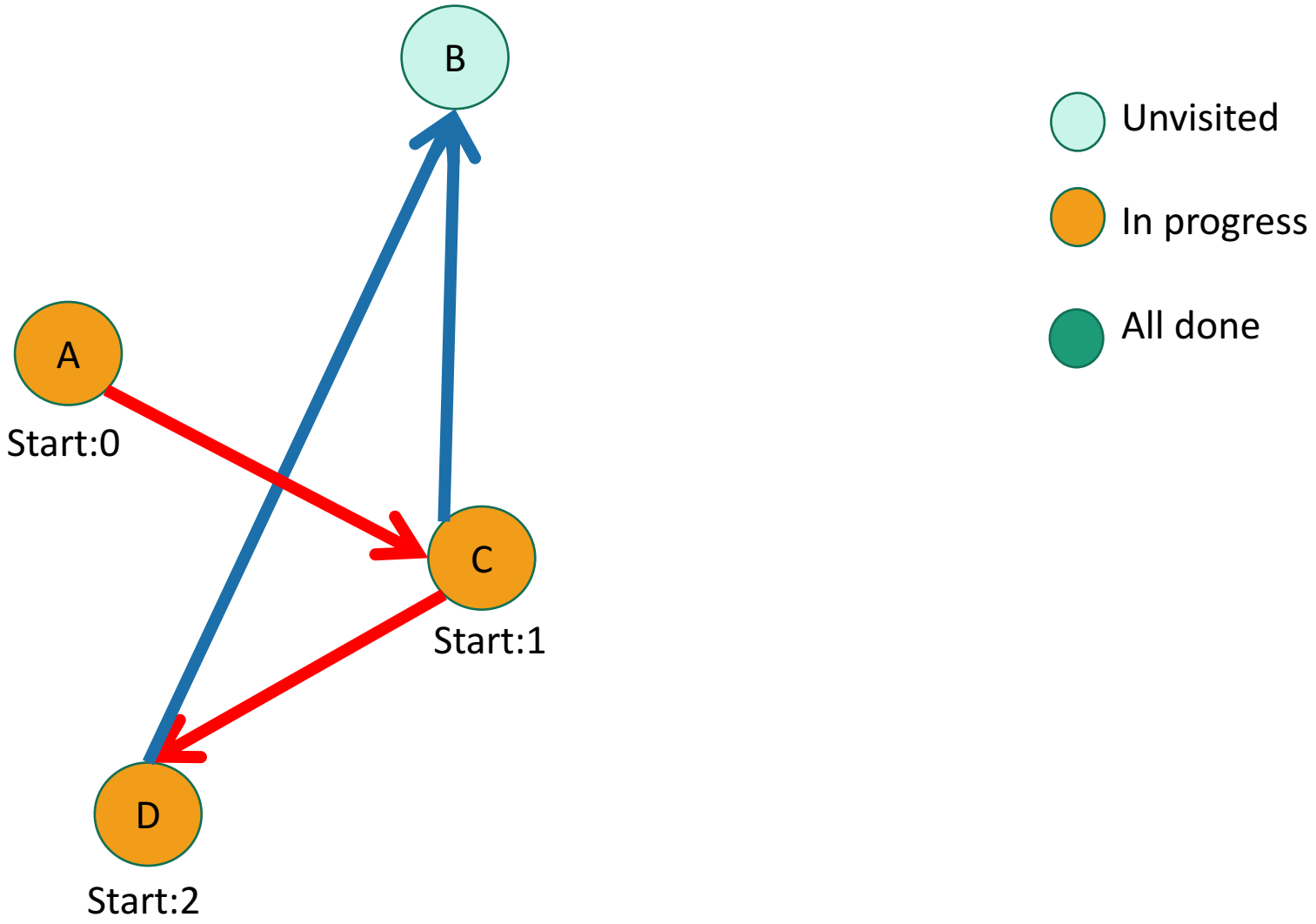
Example:



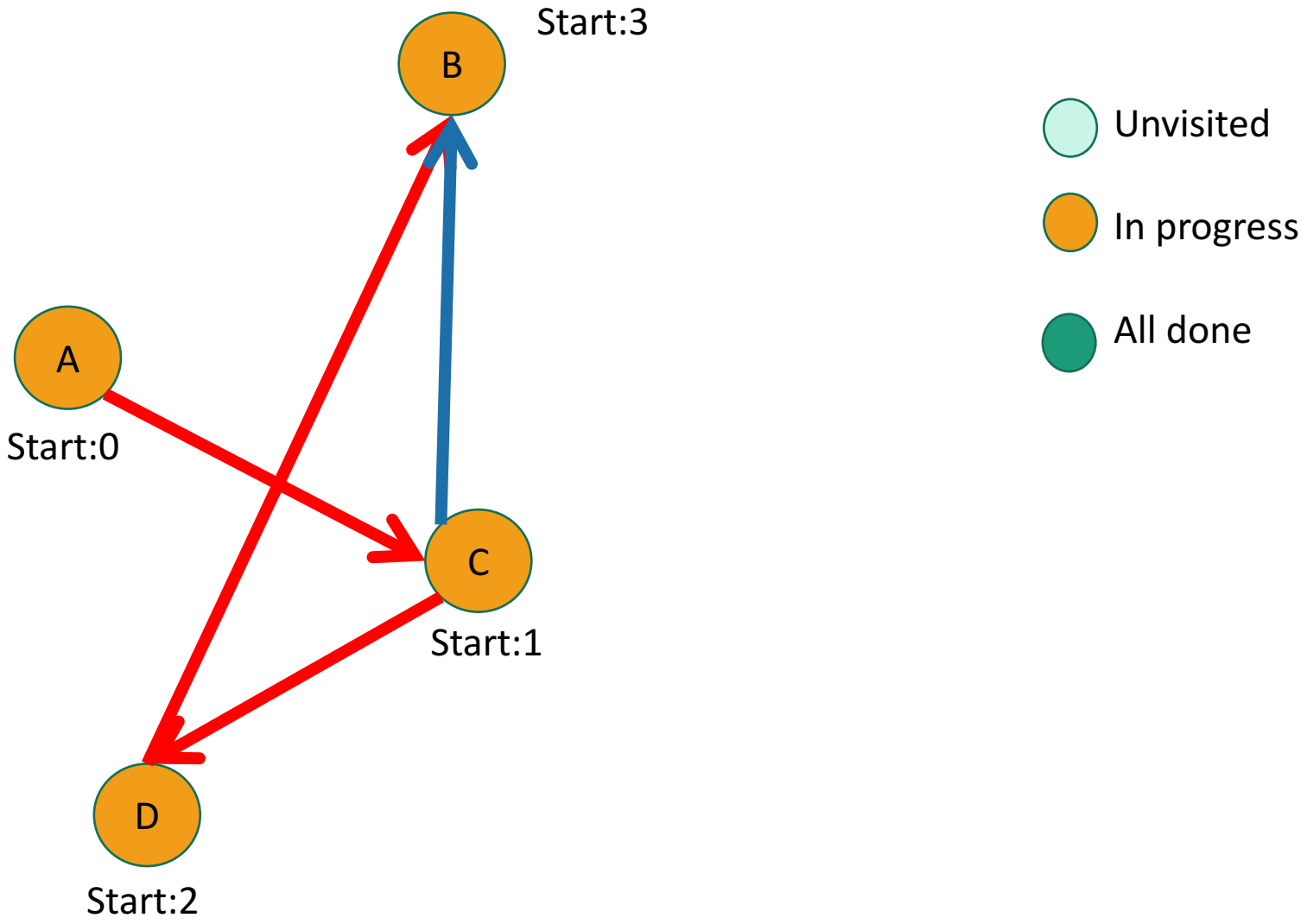
Example



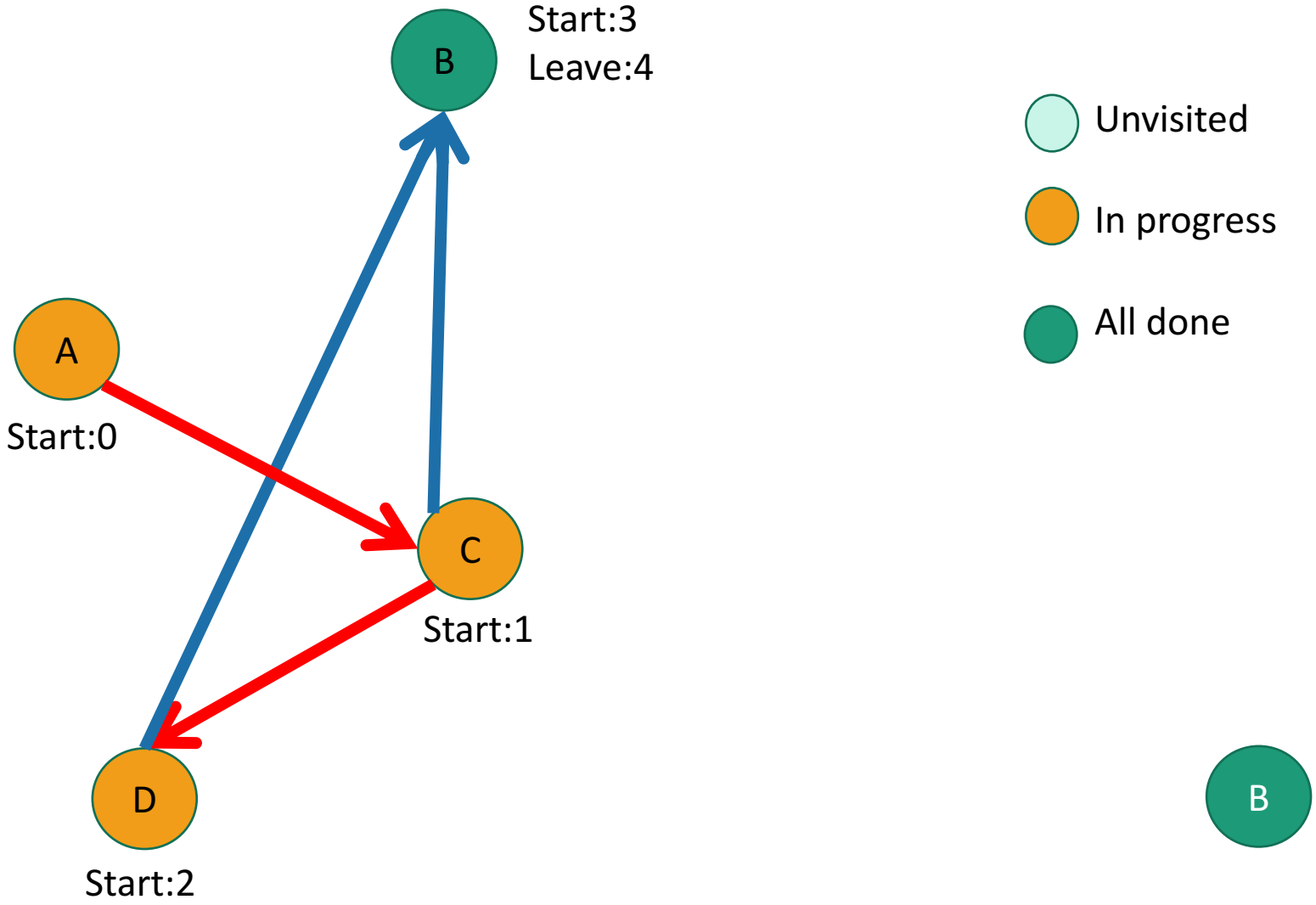
Example



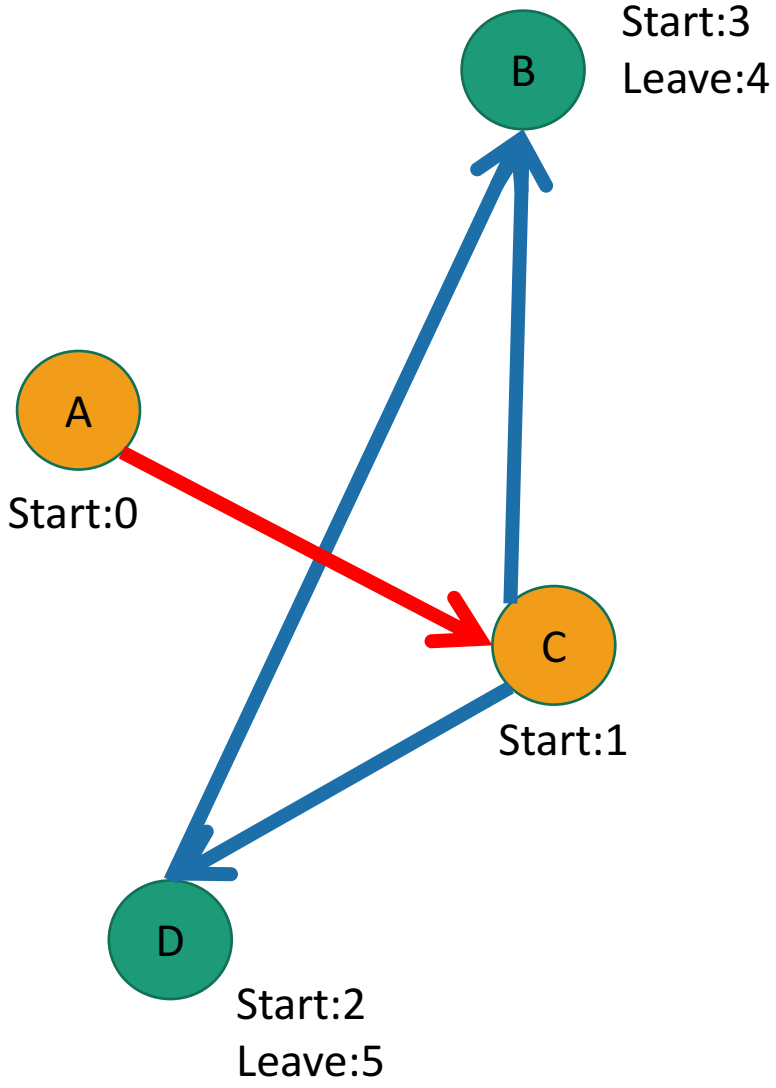
Example


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

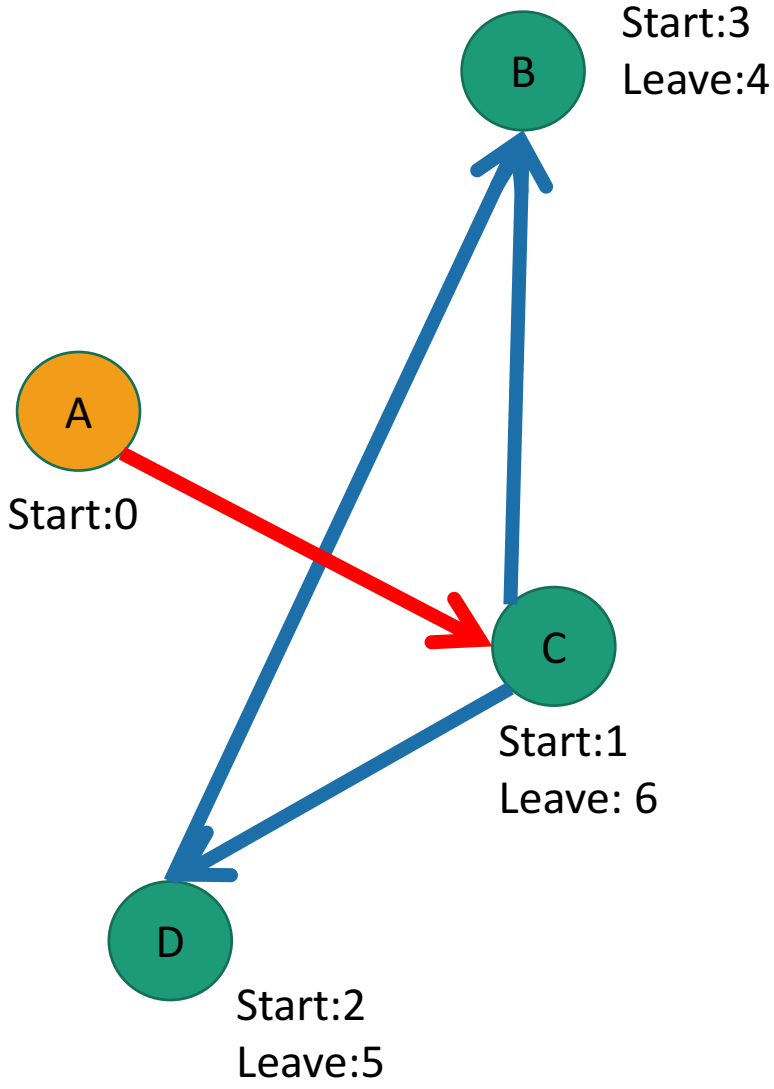
Example


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


Unvisited
In progress
All done


D
B

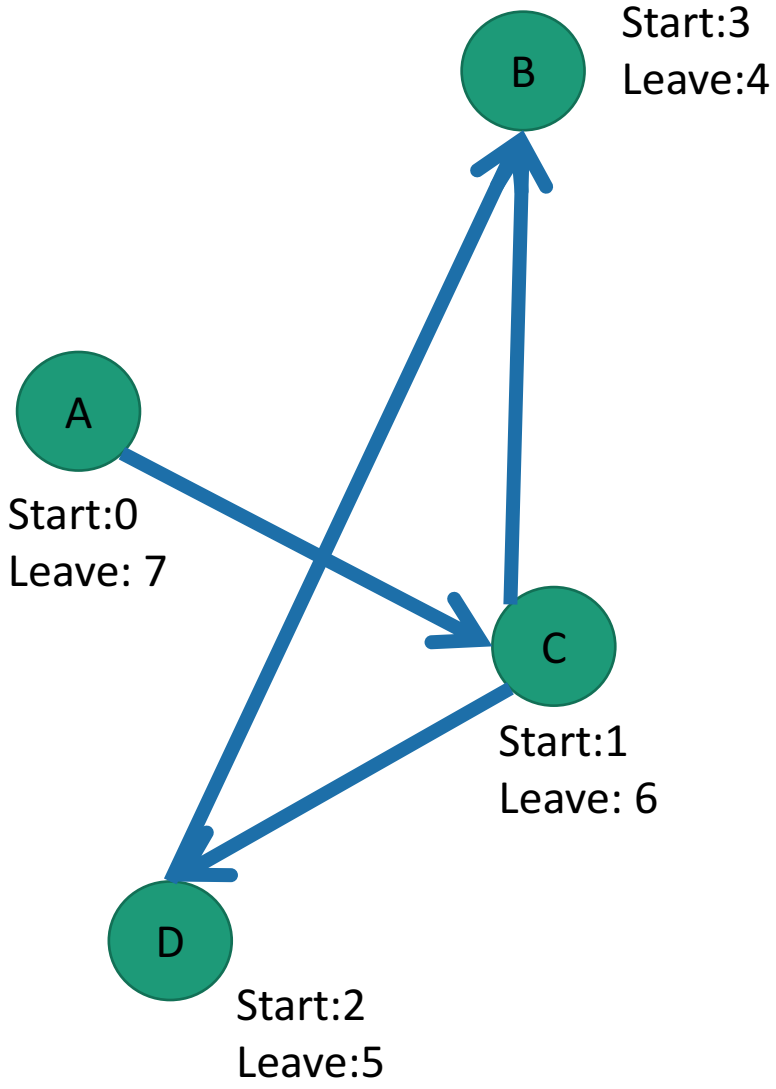
Example


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:

Suppose the underlying graph has no cycles

This is not an accident

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)

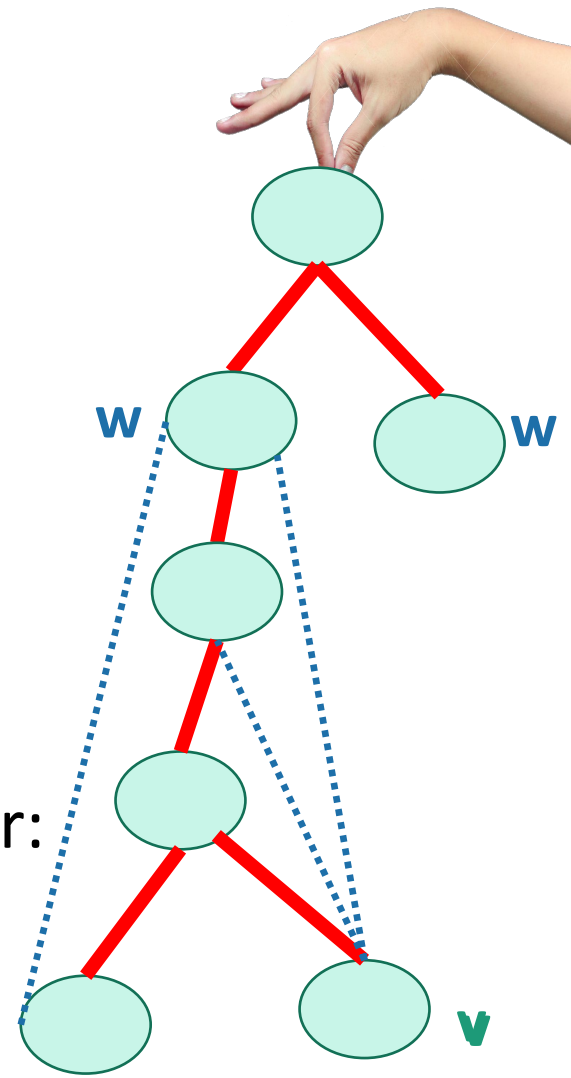
This is called the "parentheses theorem" in CLRS

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

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

- If neither are descendents of each other:

(or the other way around)

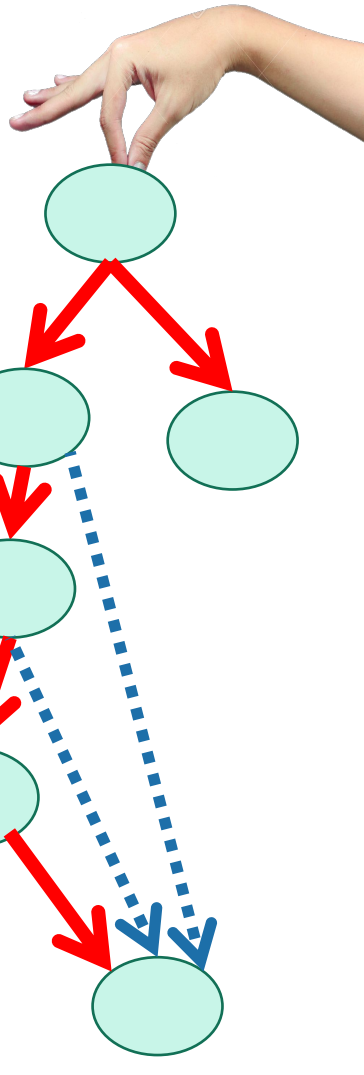


So to prove this ->

If A → B

Then B.finishTime < A.finishTime

Suppose the underlying graph has no cycles

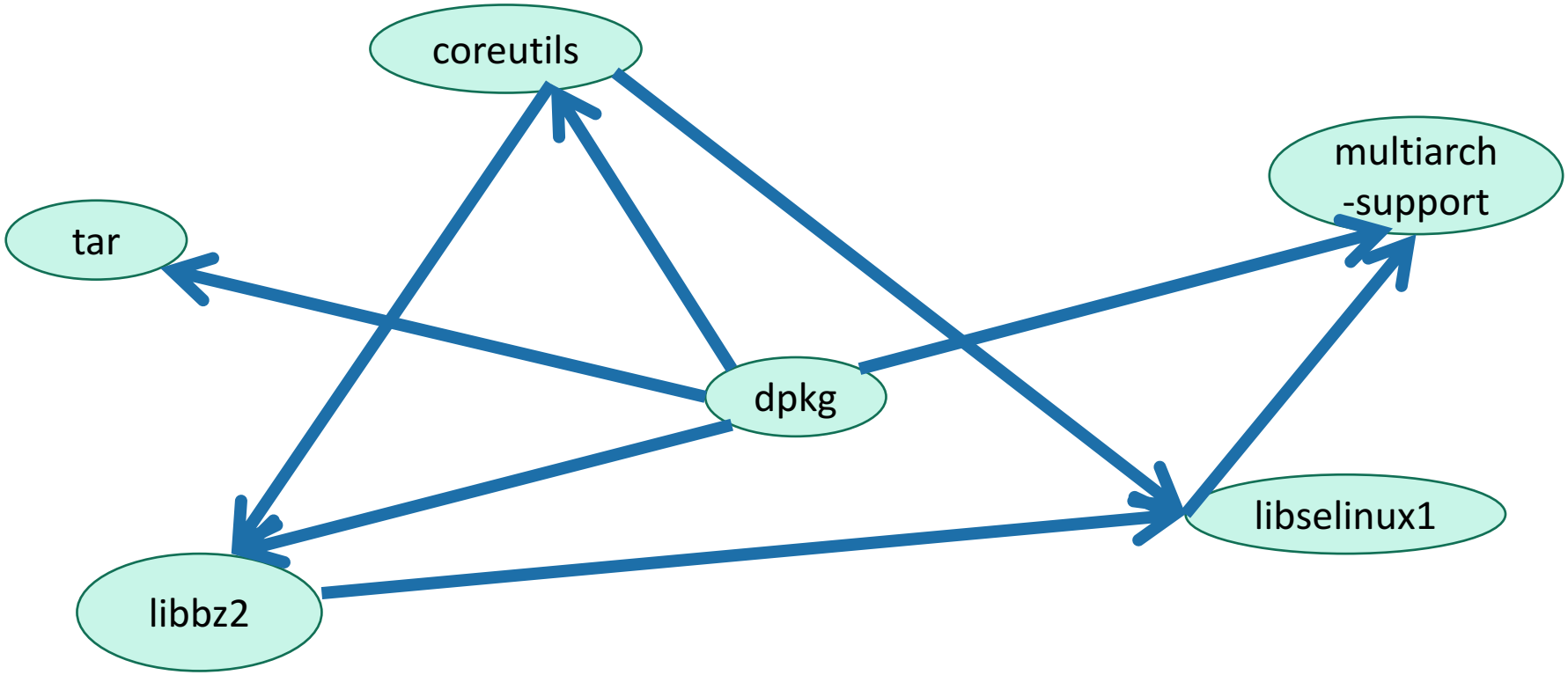


- Since the graph has no cycles, B must be a descendent of A in that tree.
  - All edges go down the tree.
- Then

- aka, B.finishTime < A.finishTime.

Back to this problem

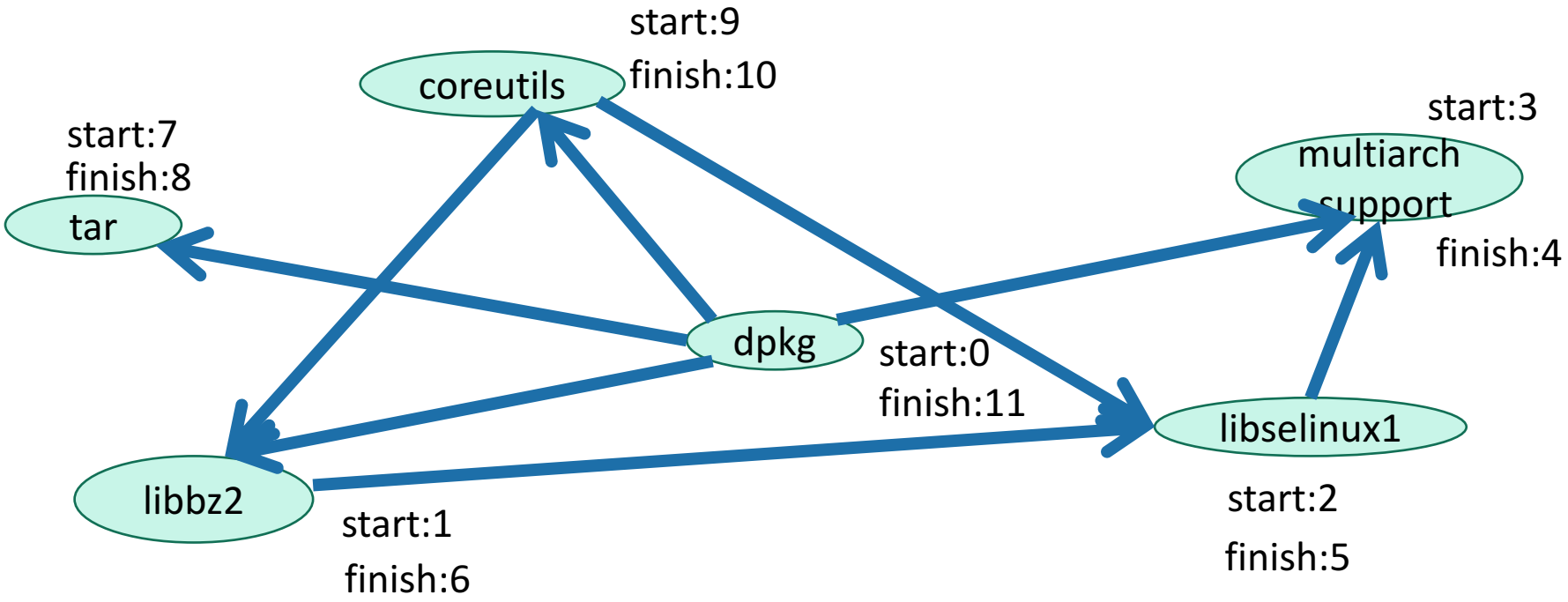
- Example: package dependency graph
- Question: in what order should I install packages?



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`



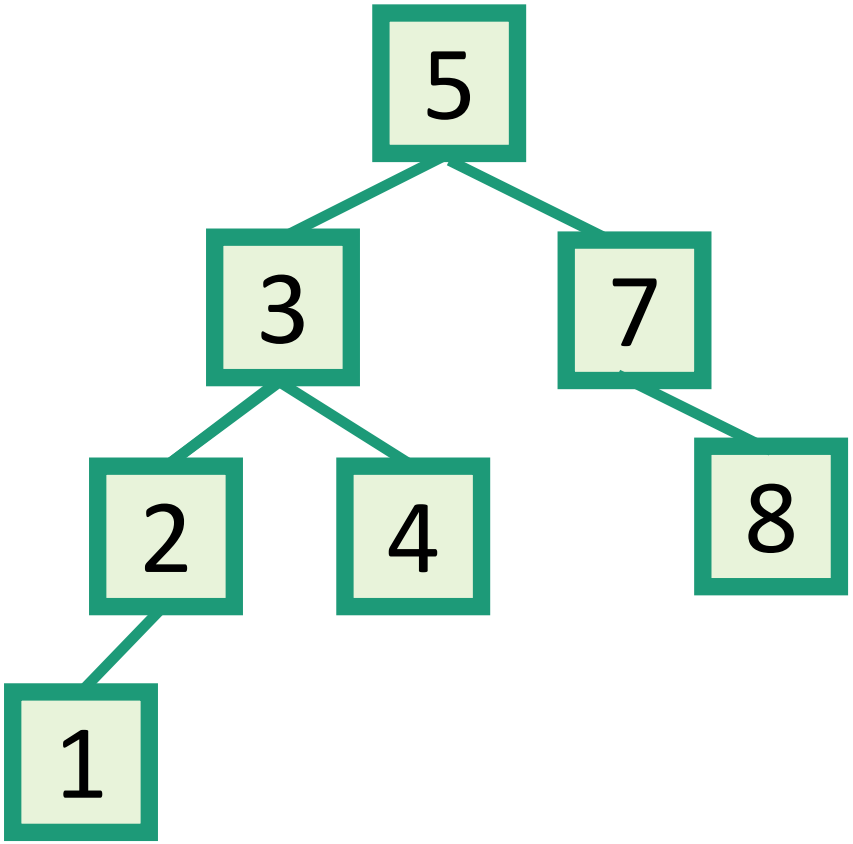
What did we just learn?

- 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.

Another use of DFS

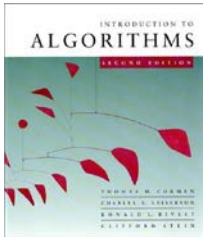
- In-order enumeration of binary search trees



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.

Part 2: breadth-first search



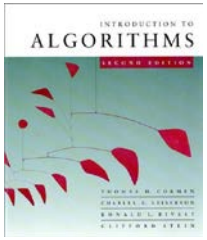
Unweighted graphs

Suppose that $w(u, v) = 1$ for all $(u, v) \in E$.
Can Dijkstra's algorithm be improved?

- Use a simple FIFO queue instead of a priority queue.

Breadth-first search

```
while Q ≠ ∅
    do u ← DEQUEUE(Q)
        for each v ∈ Adj[u]
            do if d[v] = ∞
                    then d[v] ← d[u] + 1
                         ENQUEUE(Q, v)
```



Unweighted graphs

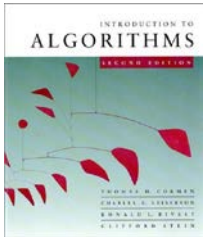
Suppose that $w(u, v) = 1$ for all $(u, v) \in E$.
Can Dijkstra's algorithm be improved?

- Use a simple FIFO queue instead of a priority queue.

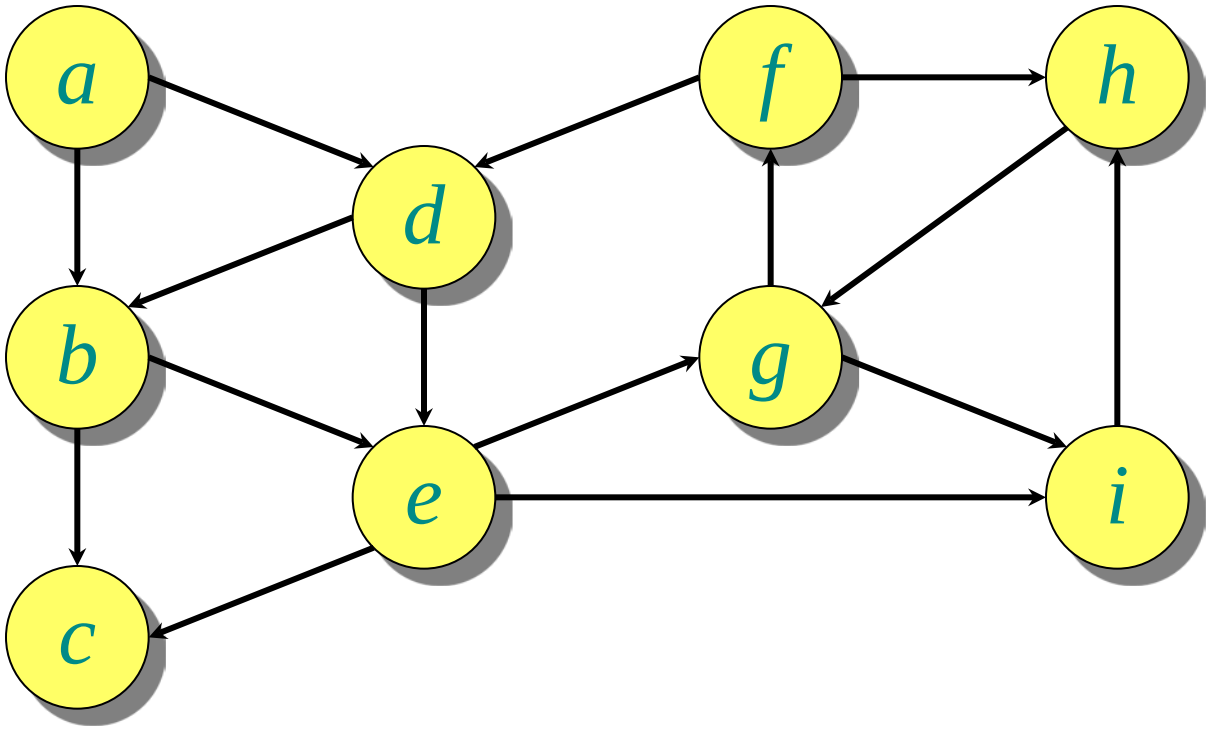
Breadth-first search

while $Q \neq \varnothing$
  do $u \leftarrow$ DEQUEUE(Q)
    for each $v \in Adj[u]$
      do if $d[v] = \infty$
        then $d[v] \leftarrow d[u] + 1$
          ENQUEUE(Q, v)

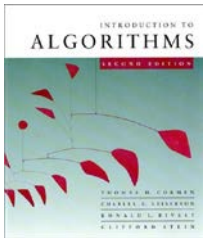
Analysis: Time $= O(V + E)$.



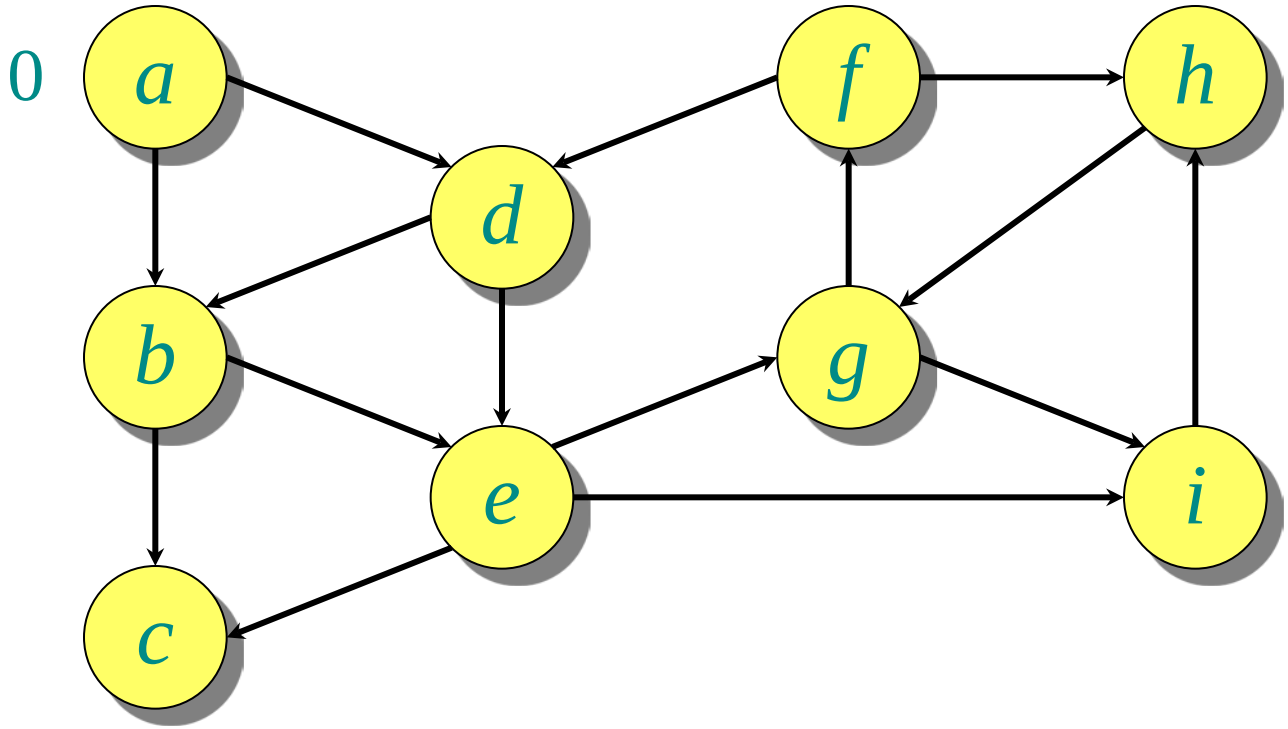
Example of breadth-first search



Q:

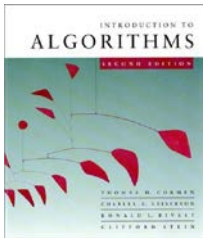


Example of breadth-first search

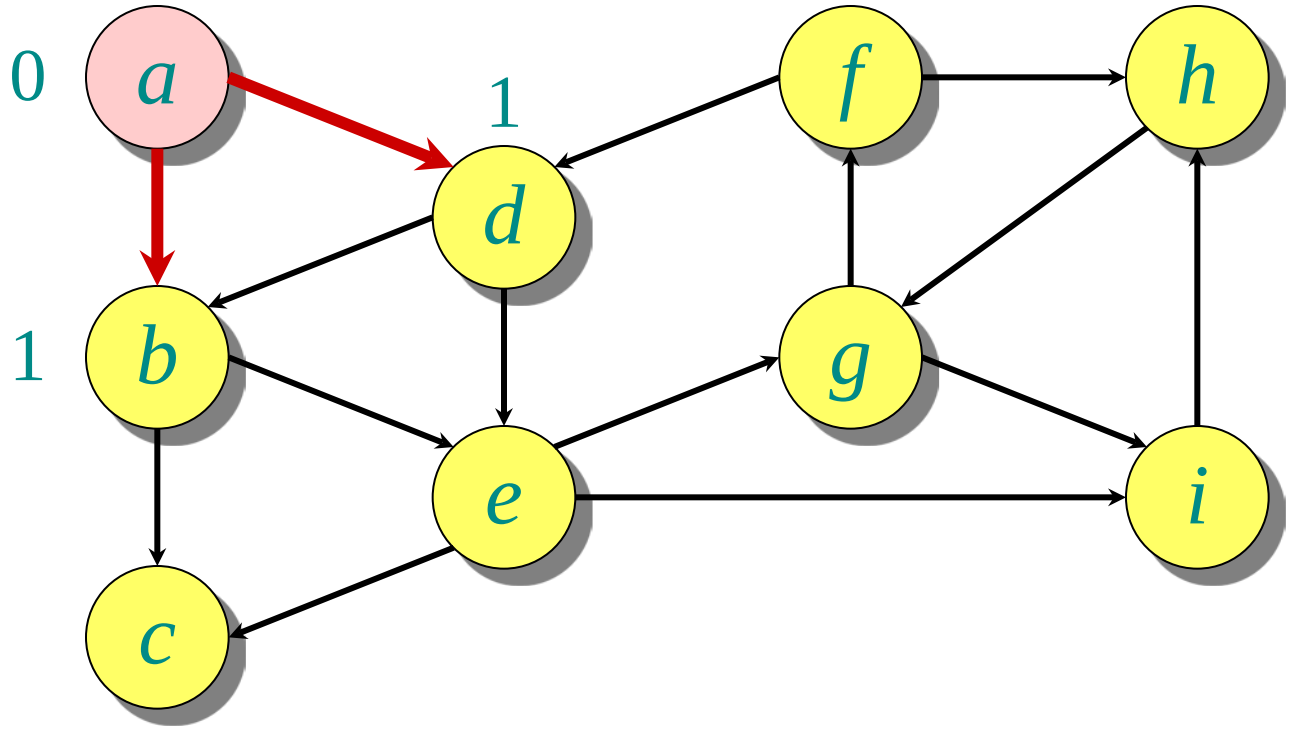


0

Q: a

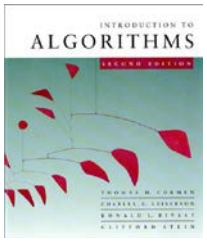


Example of breadth-first search

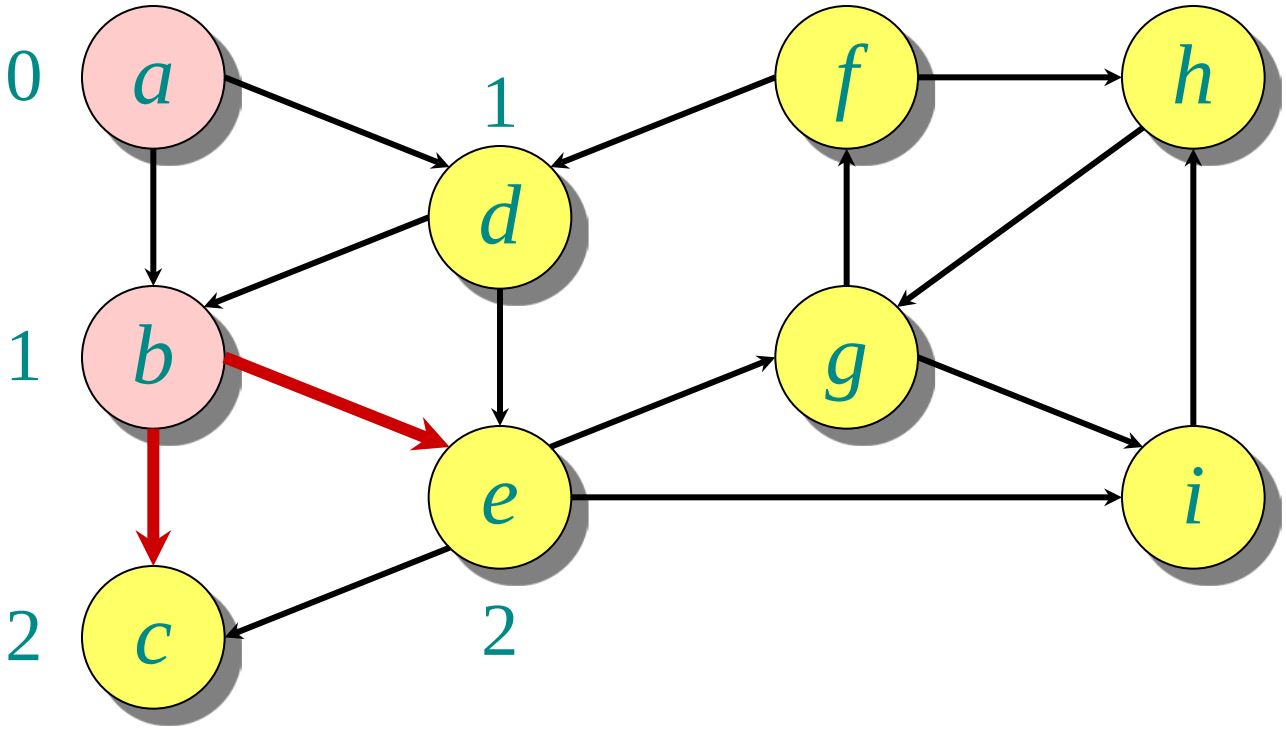


1 1

Q: a b d

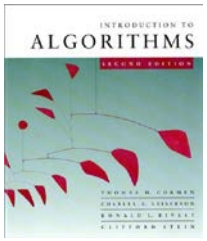


Example of breadth-first search

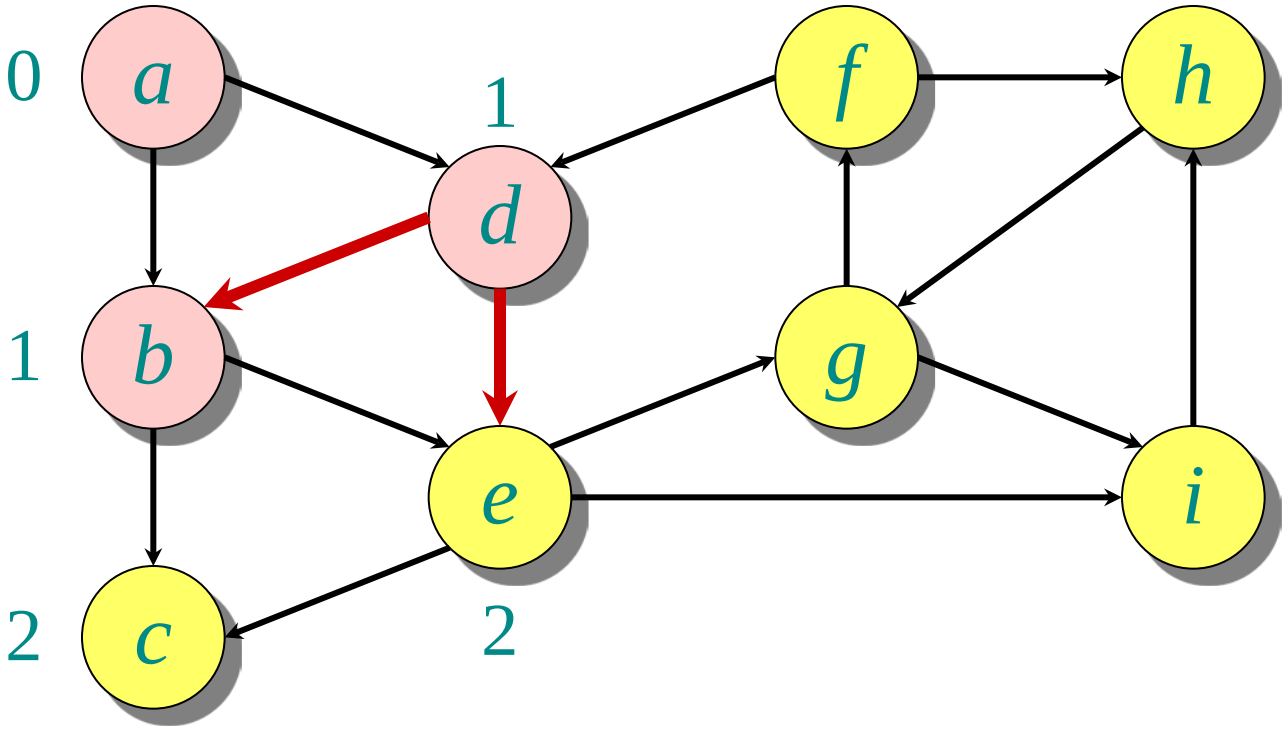


1 2 2

Q: *a* *b* *d* *c* *e*

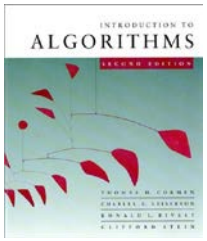


Example of breadth-first search

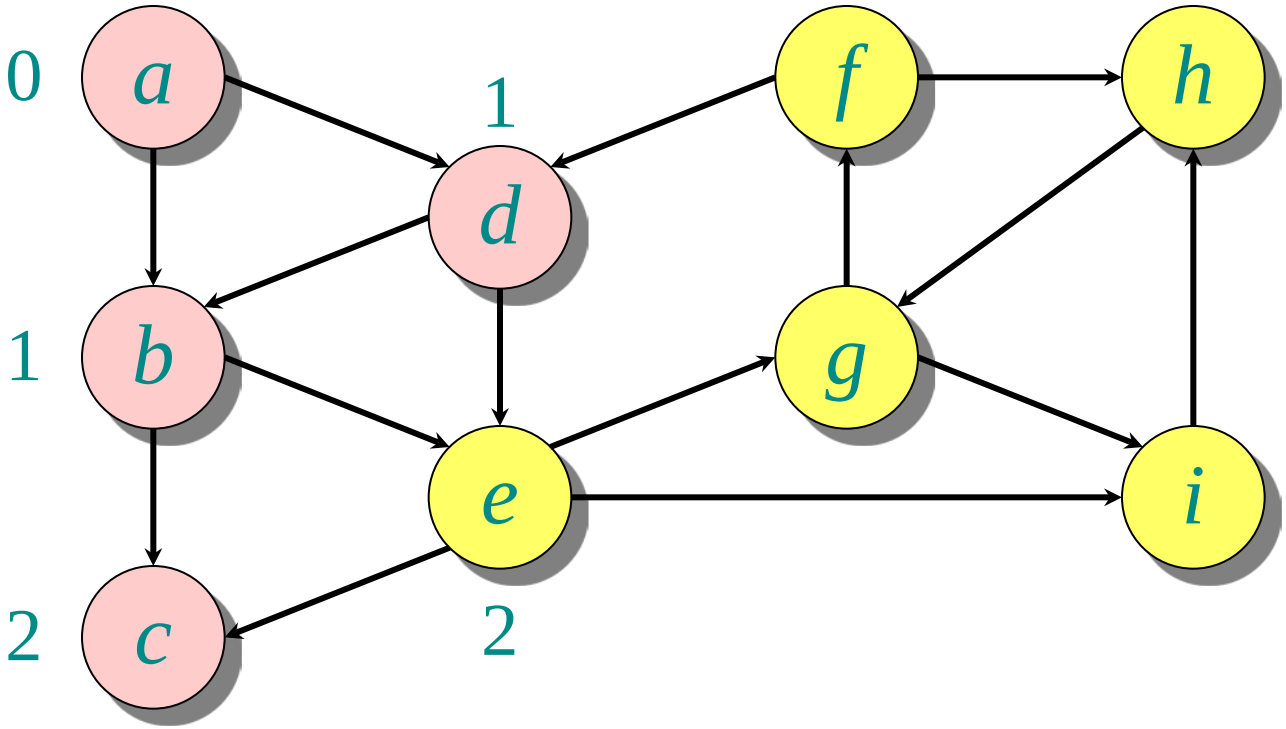


2 2

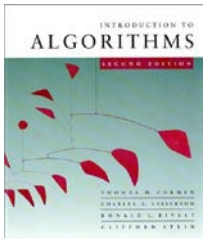
Q: a b d c e



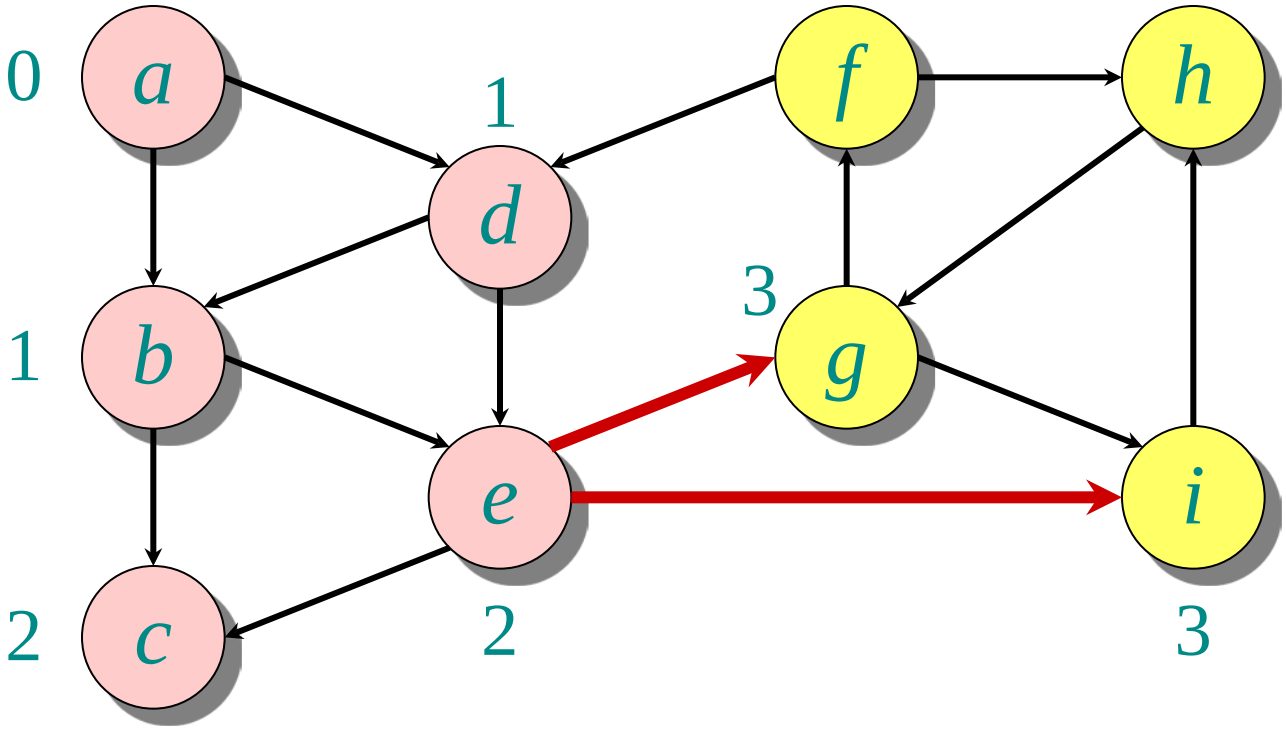
Example of breadth-first search



Q: a b d c e

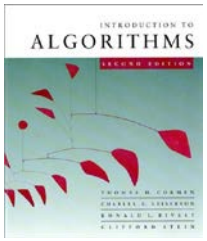


Example of breadth-first search

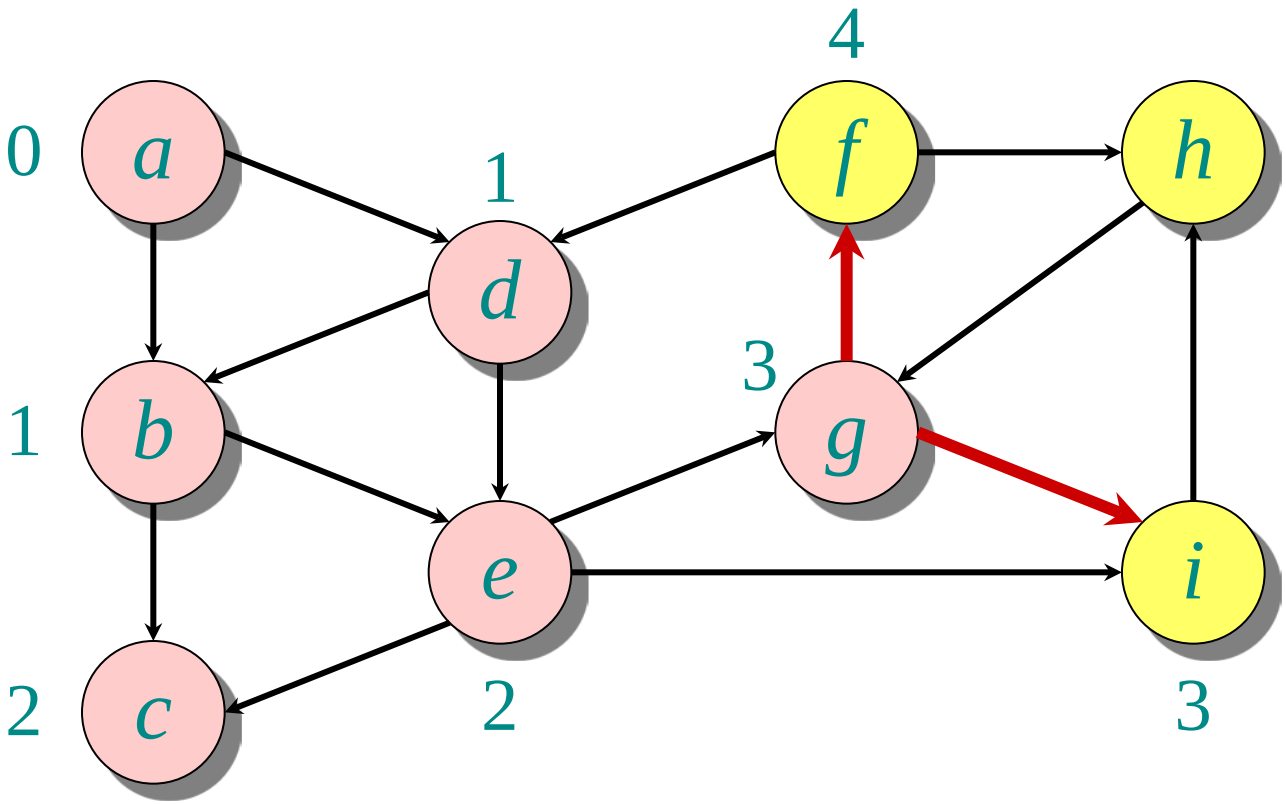


3 3

Q: a b d c e g i

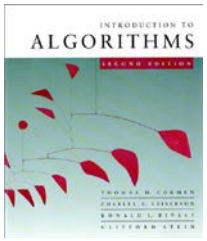


Example of breadth-first search

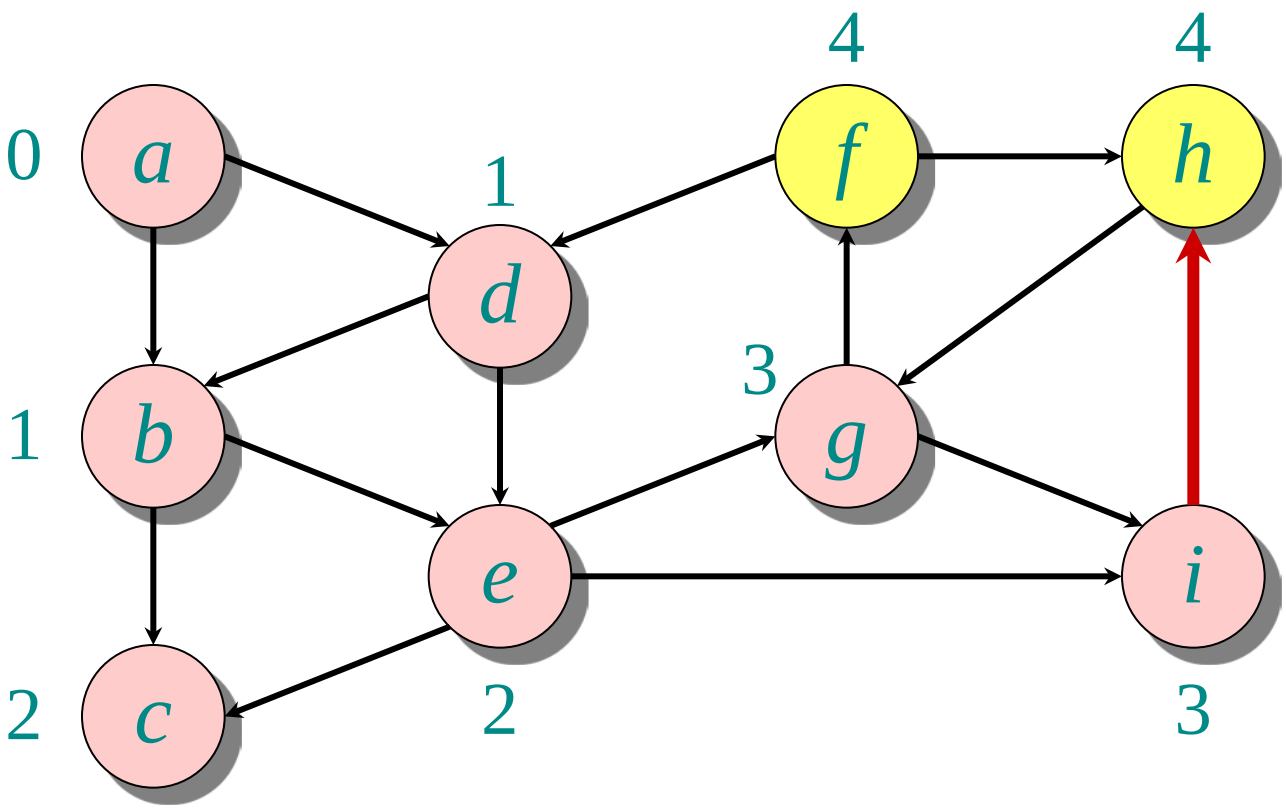


3 4

Q: a b d c e g i f

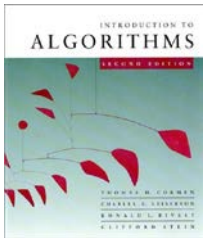


Example of breadth-first search

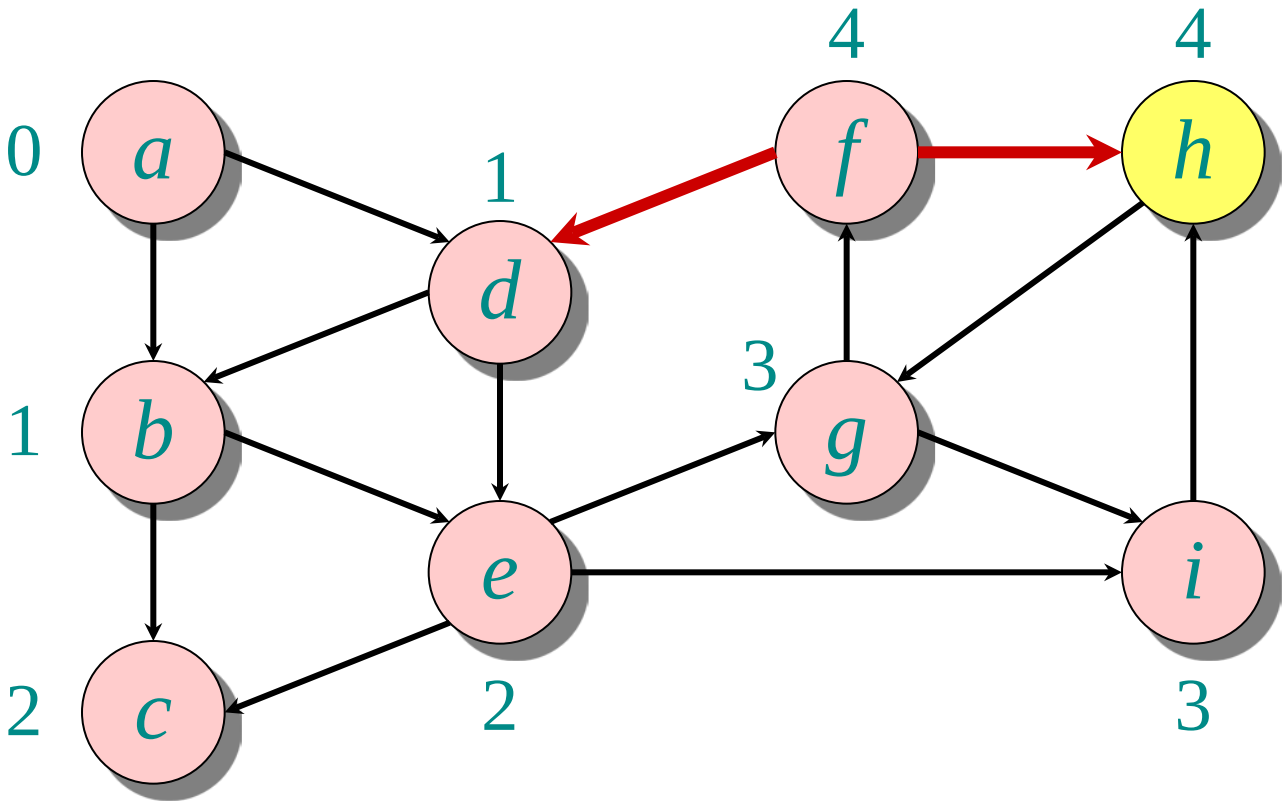


Q: a b d c e g i f h

4 4

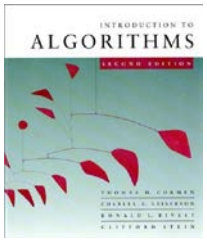


Example of breadth-first search

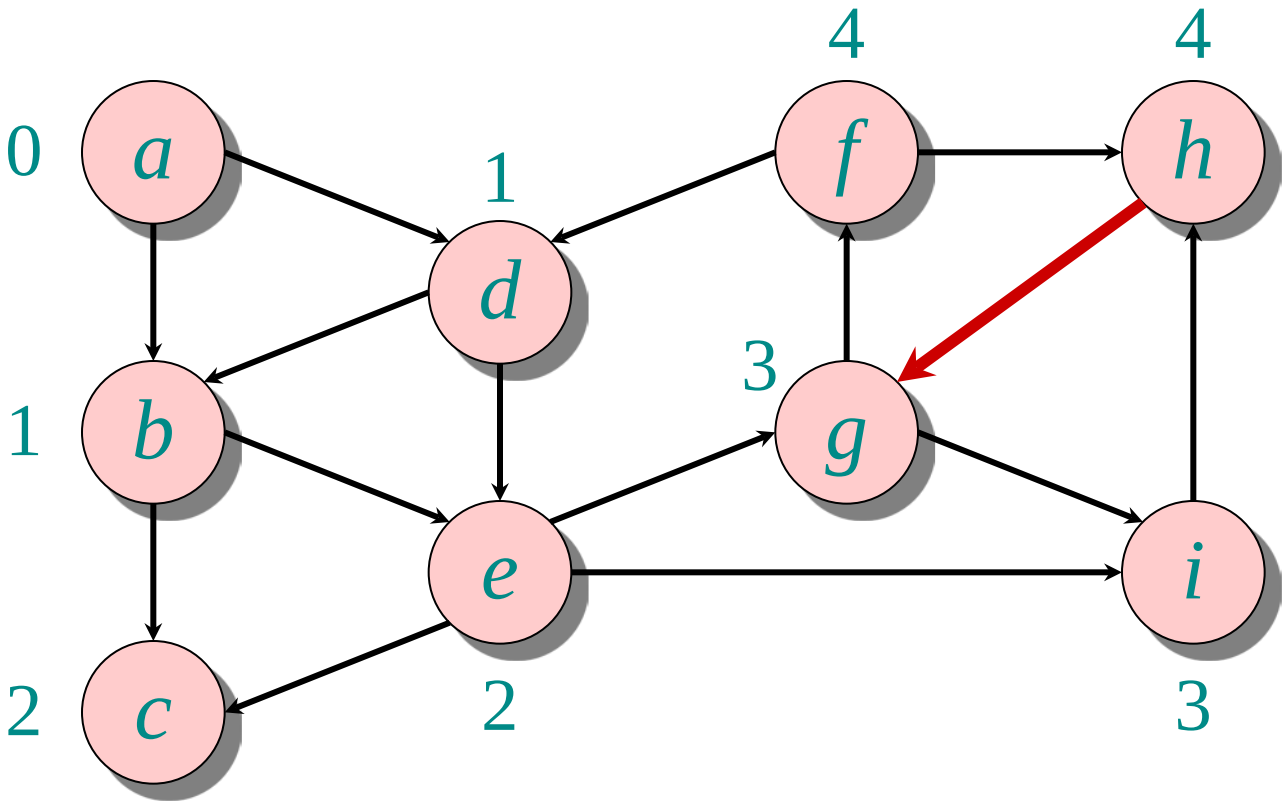


4

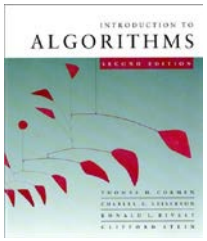
Q: a b d c e g i f h



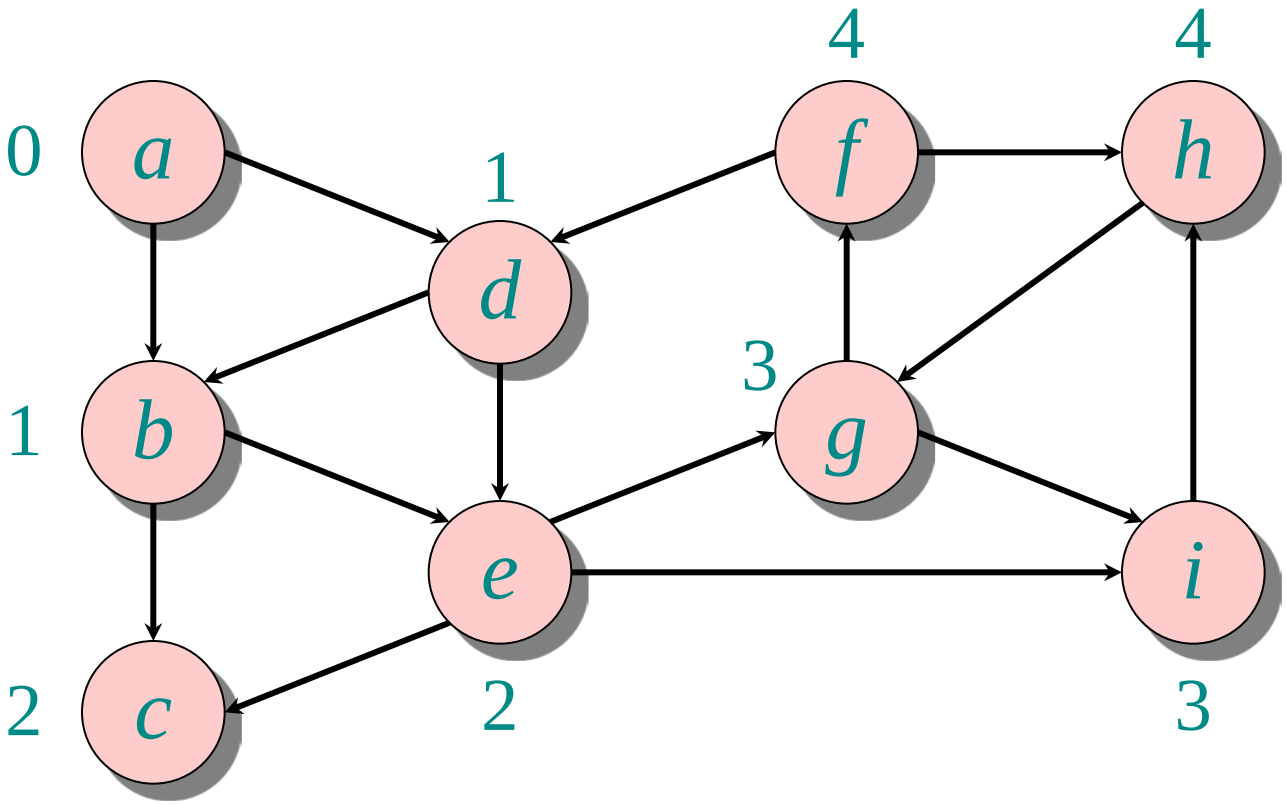
Example of breadth-first search



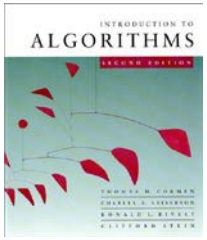
Q: *a b d c e g i f h*



Example of breadth-first search



Q: *a b d c e g i f h*



Correctness of BFS

```
while Q ≠ ∅
    do u ← DEQUEUE(Q)
        for each v ∈ Adj[u]
            do if d[v] = ∞
                    then d[v] ← d[u] + 1
                         ENQUEUE(Q, v)
```

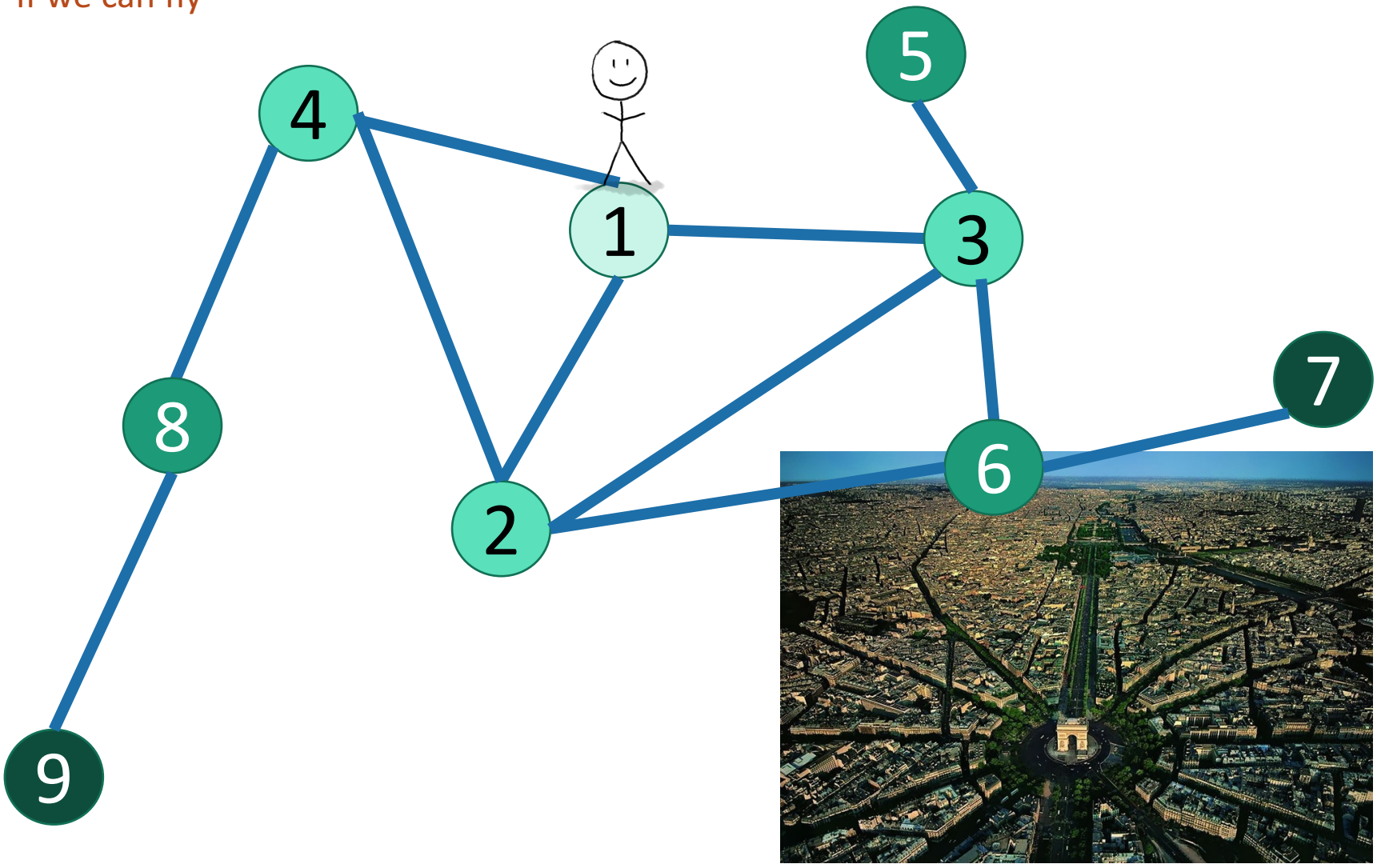
Key idea:

The FIFO Q in breadth-first search mimics the priority queue Q in Dijkstra.

- **Invariant:** v comes after u in Q implies that $d[v] = d[u]$ or $d[v] = d[u] + 1$.

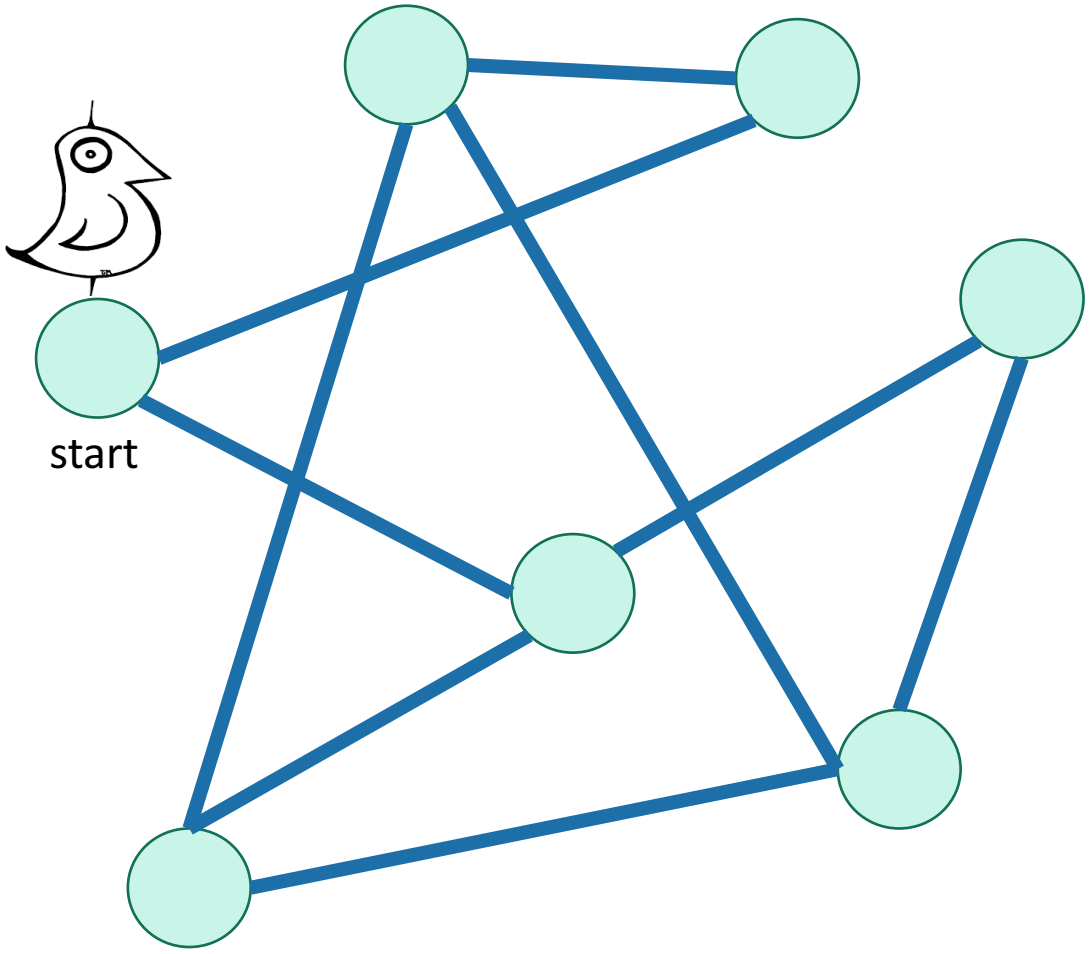
How do we explore a graph?

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

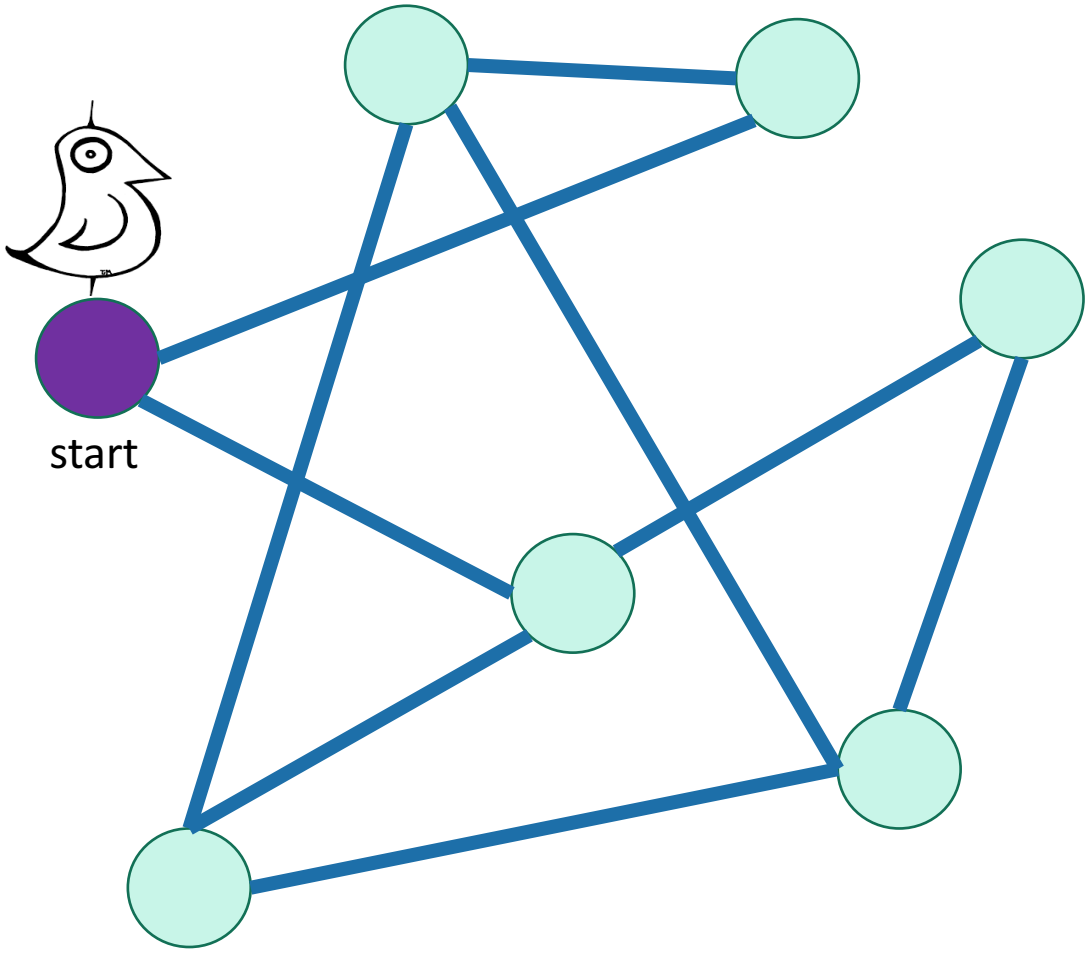
Can reach there in one step

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

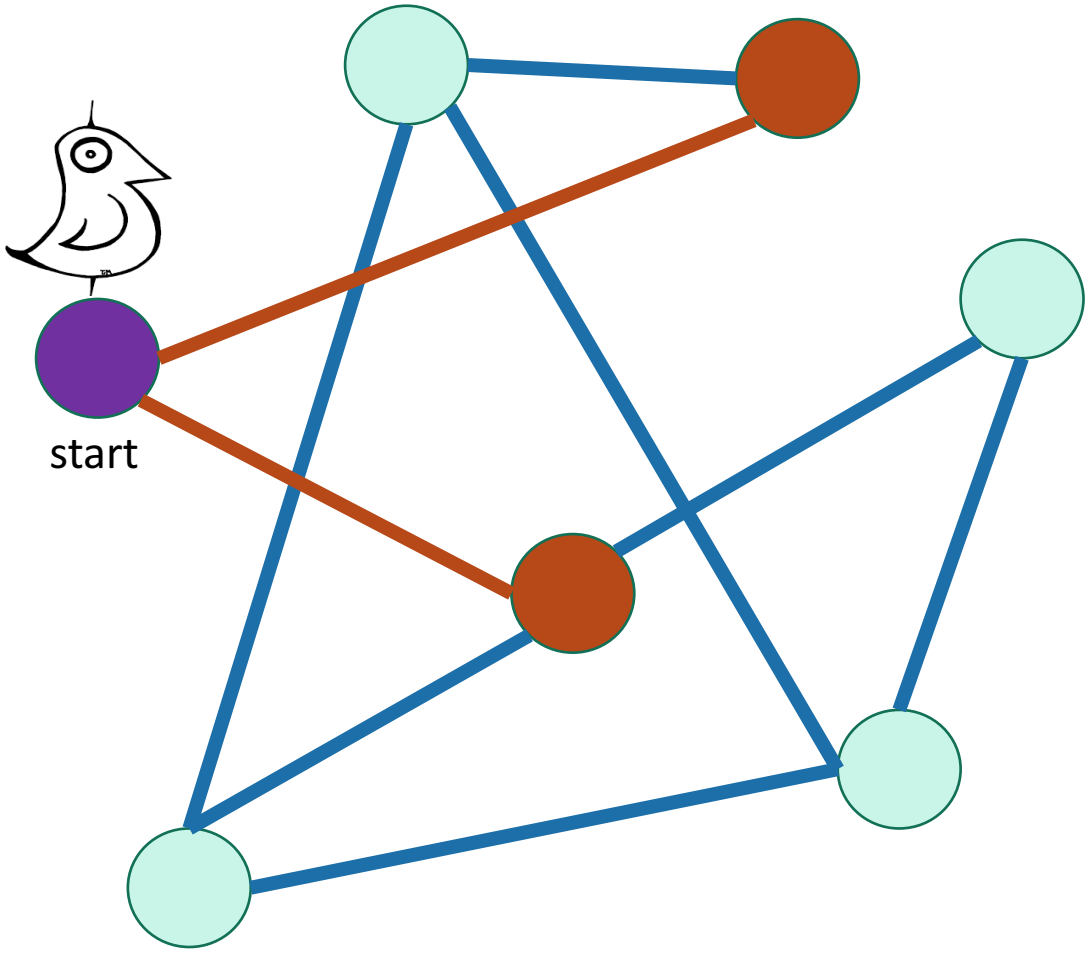
Can reach there in one step

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

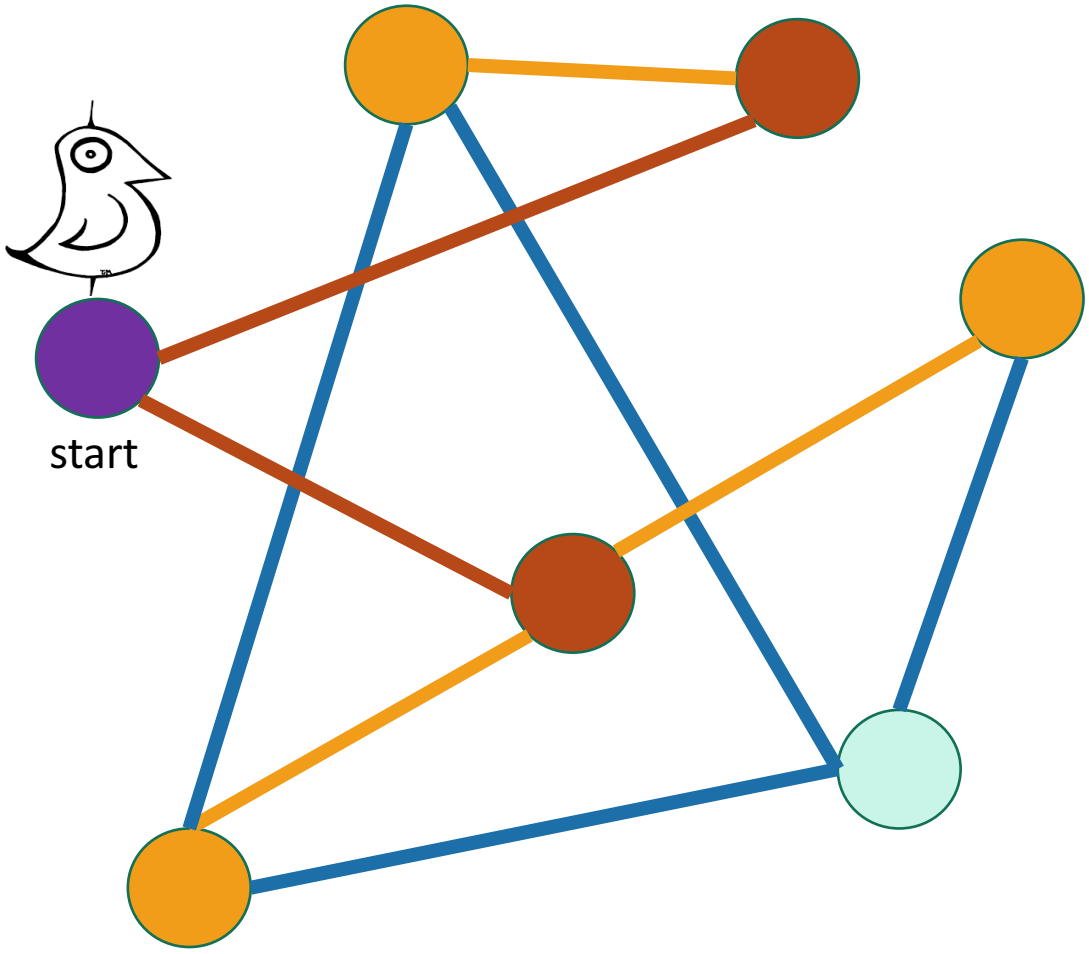
Can reach there in one step

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

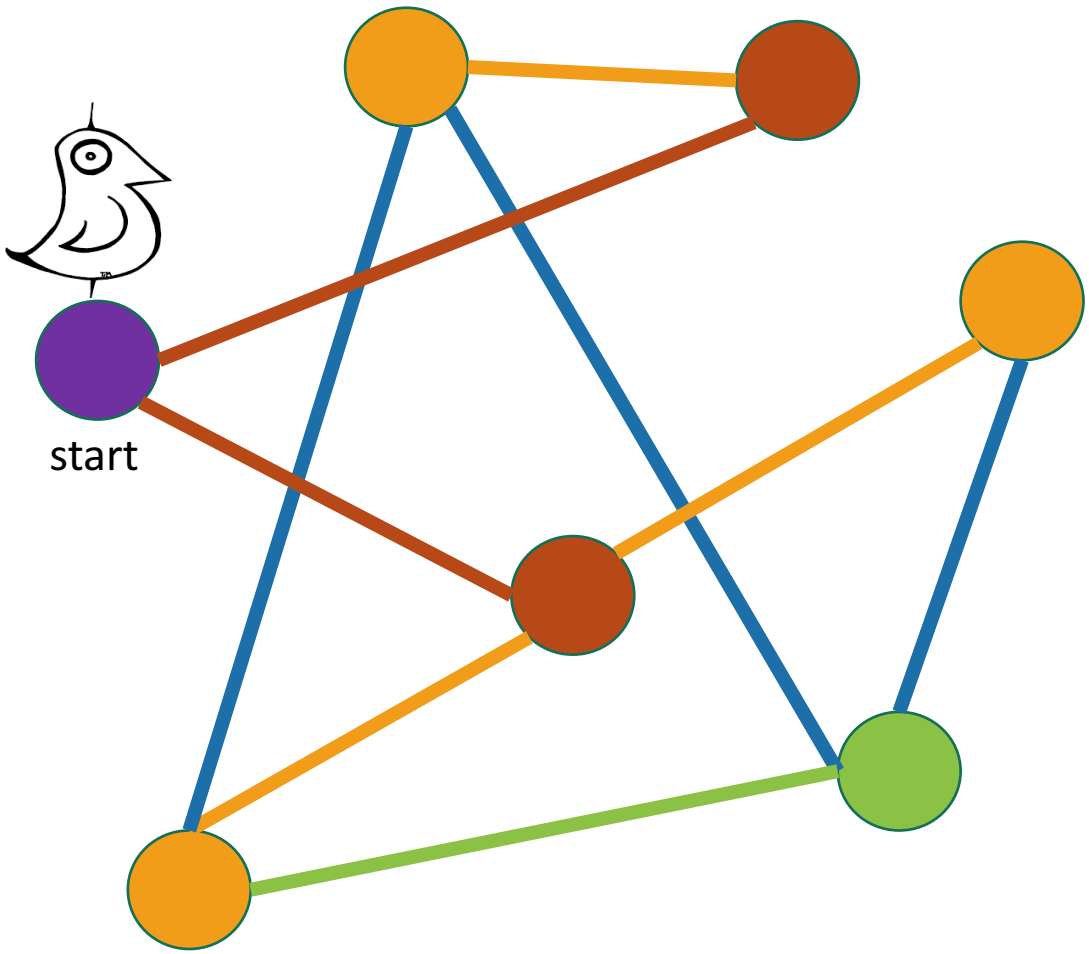
Can reach there in one step

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

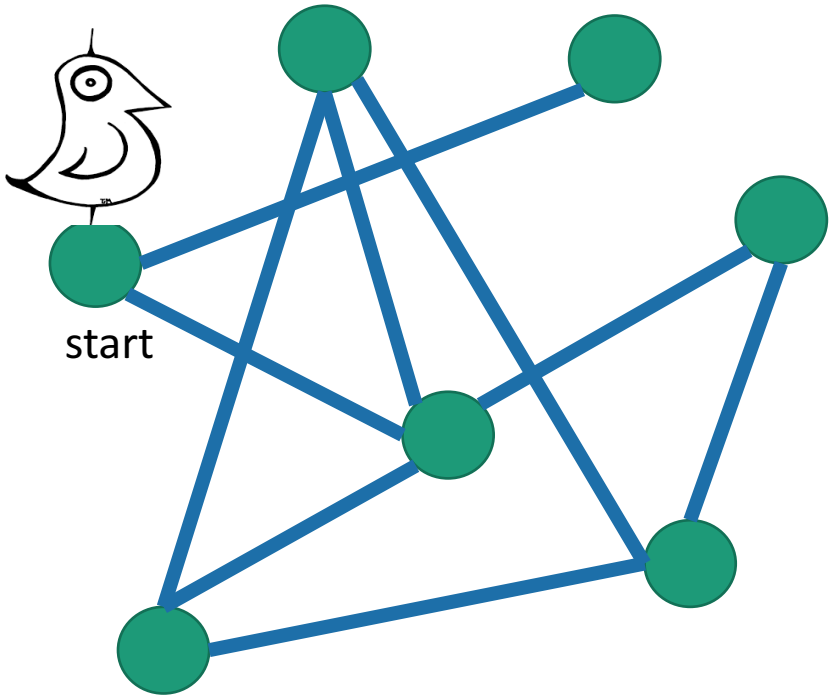
Can reach there in two steps

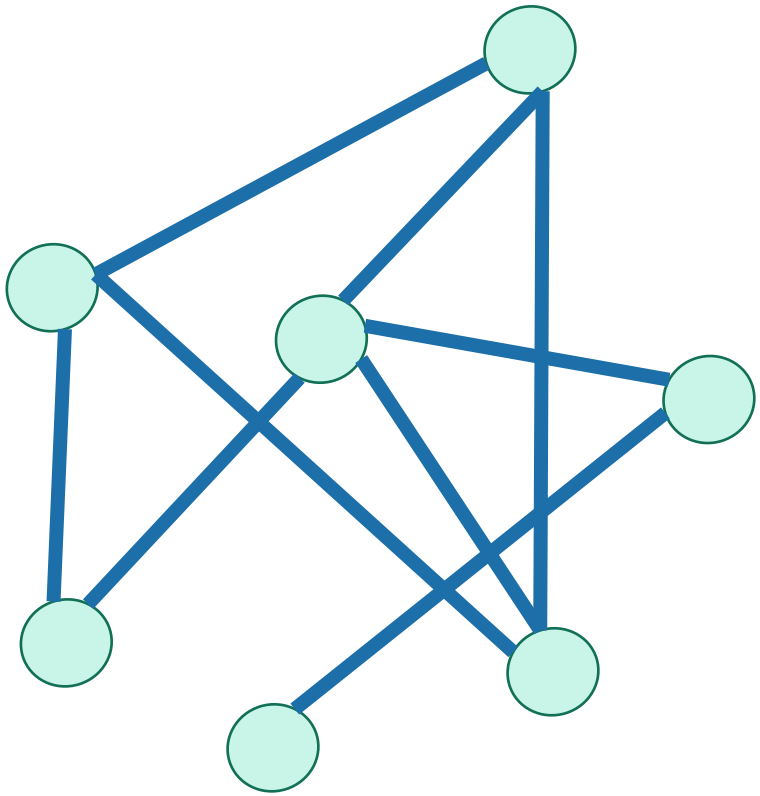
Can reach there in three steps

World:

EXPLORED!

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

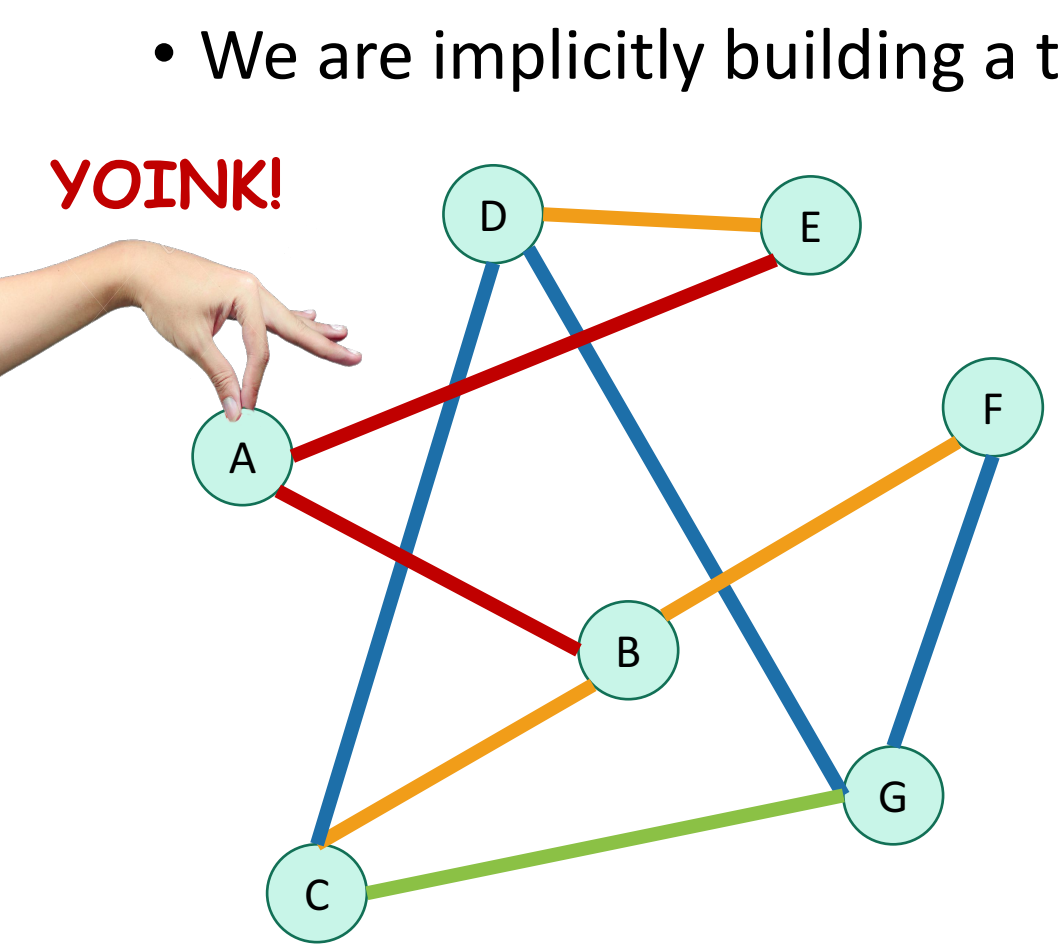
- Explore the connected components one-by-one.
- Same argument as DFS: running time is

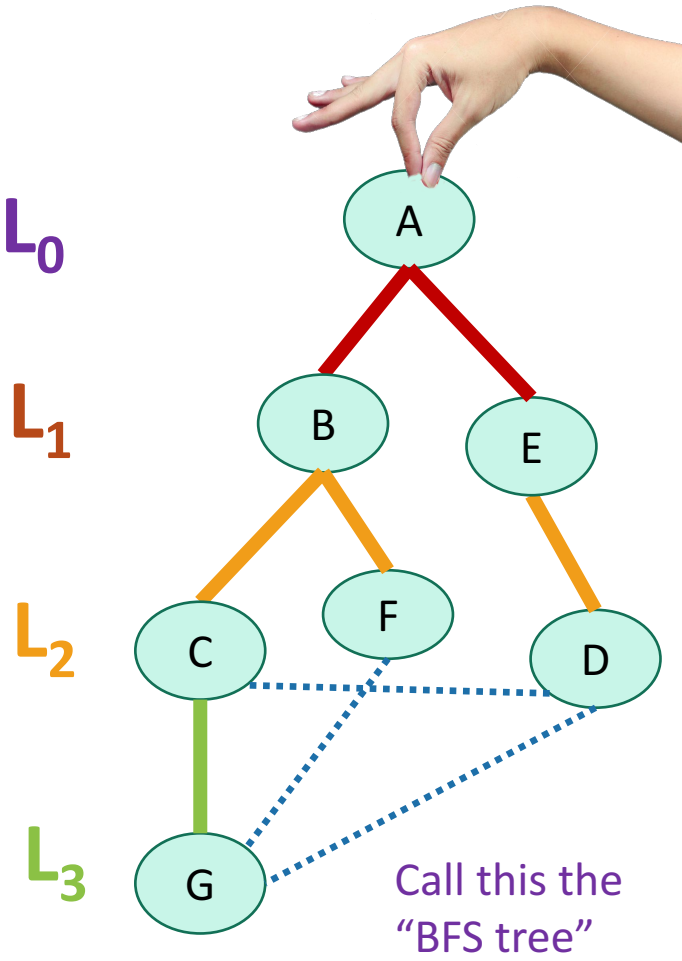
$$O(n + m)$$

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

Why is it called breadth-first?

- We are implicitly building a tree:

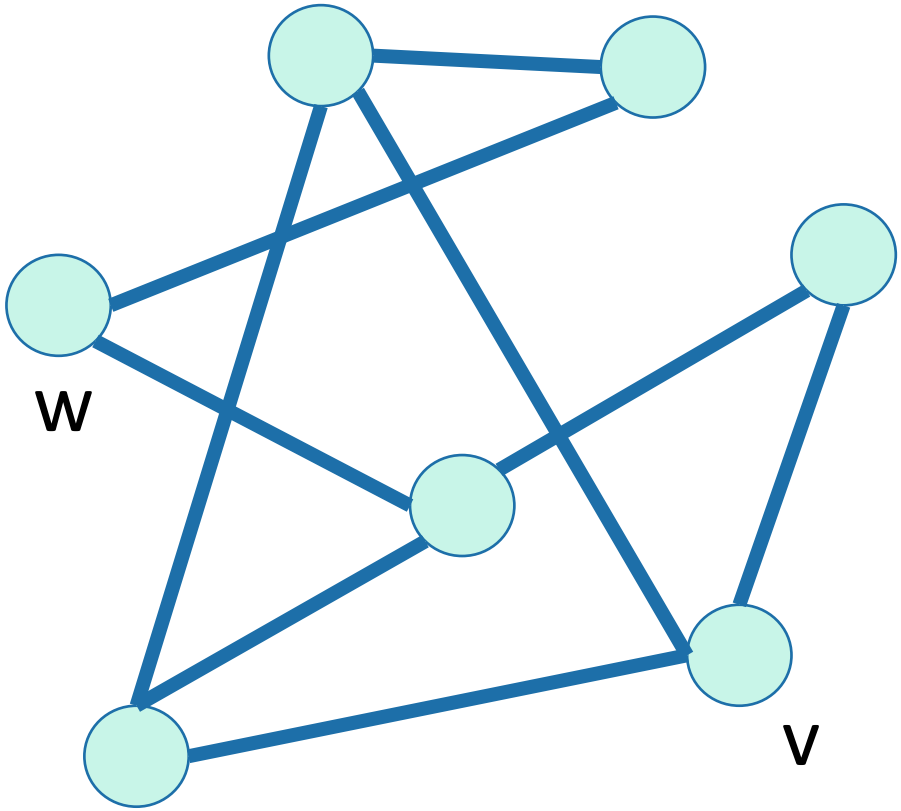




- And first we go as broadly as we can.

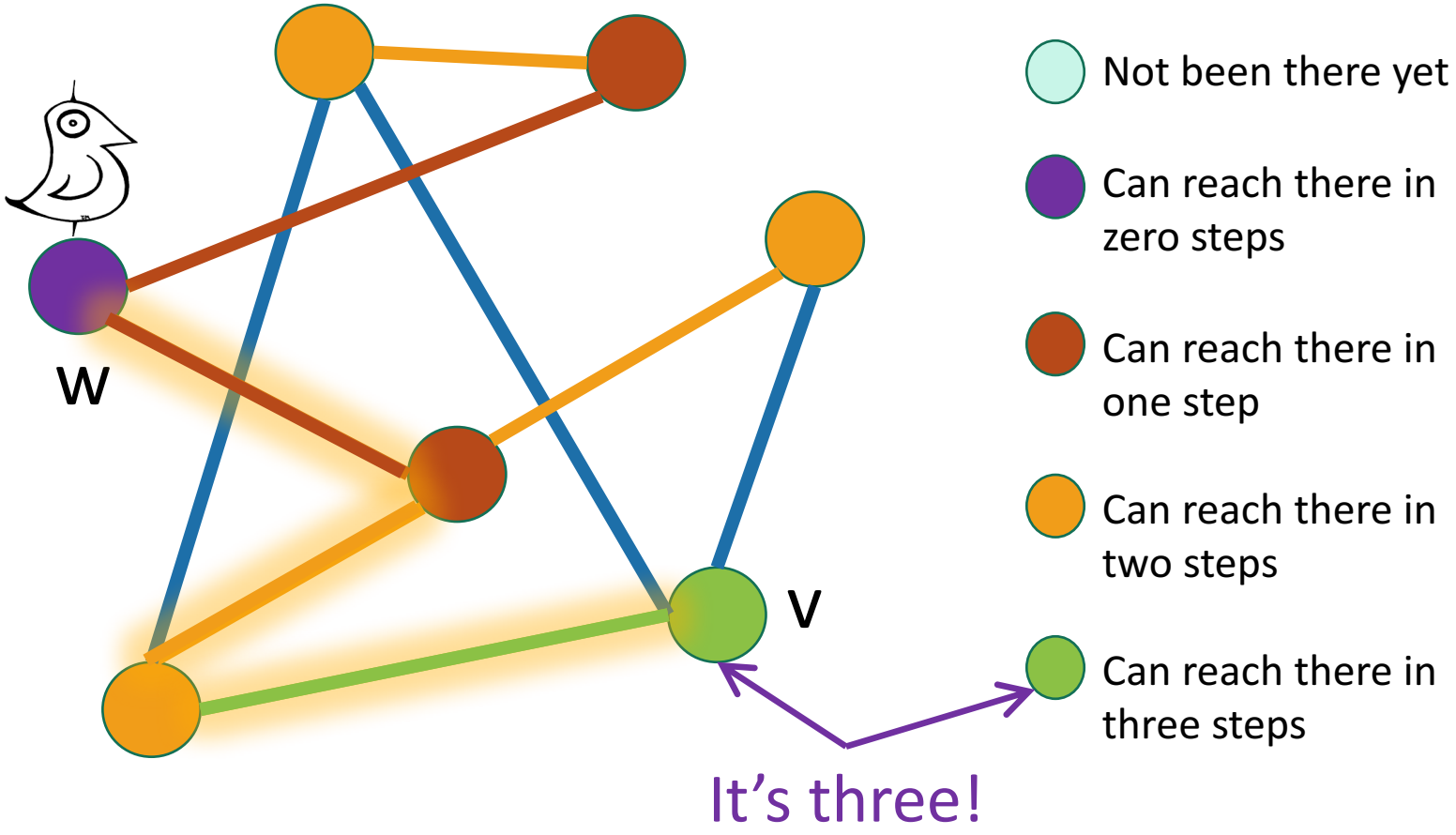
Application: shortest path

- 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 DFS starting at w
- For all v in L_i (the i'th level of the BFS tree)
  - 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.

What did we just learn?

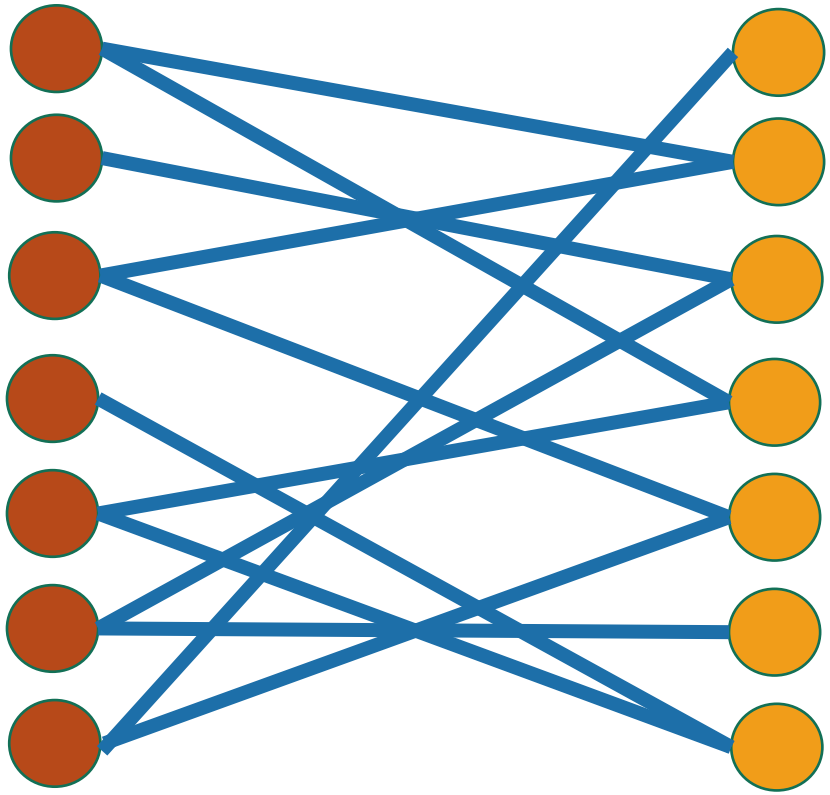
- The BFS tree is useful for computing distances between pairs of vertices.
- We can find the shortest path between u and v in time O(m).

The BSF tree is also helpful for:

- Testing if a graph is bipartite or not.

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

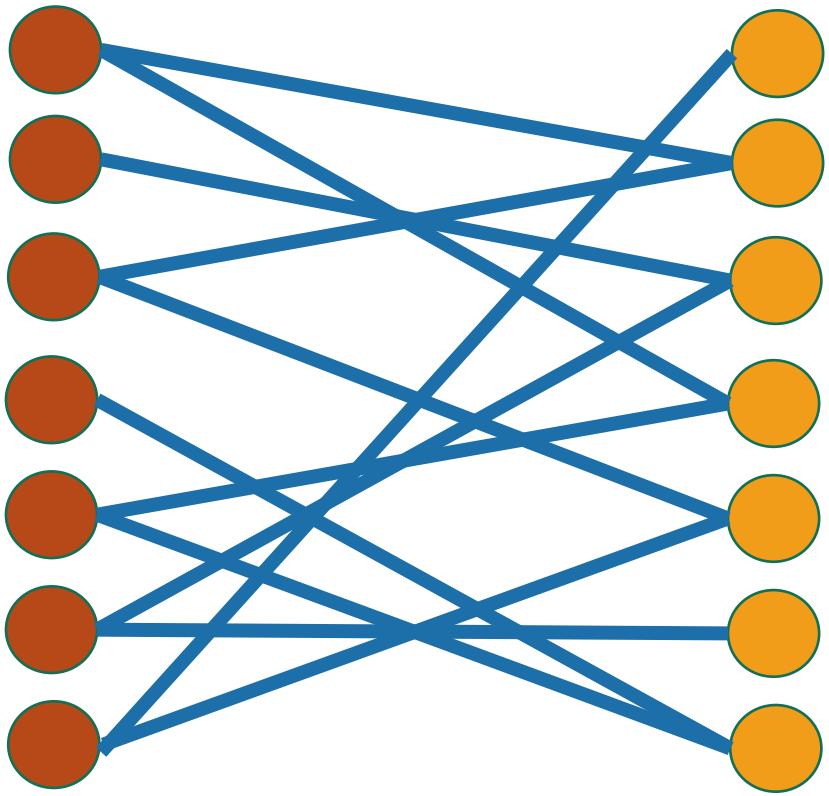
Example:

are students

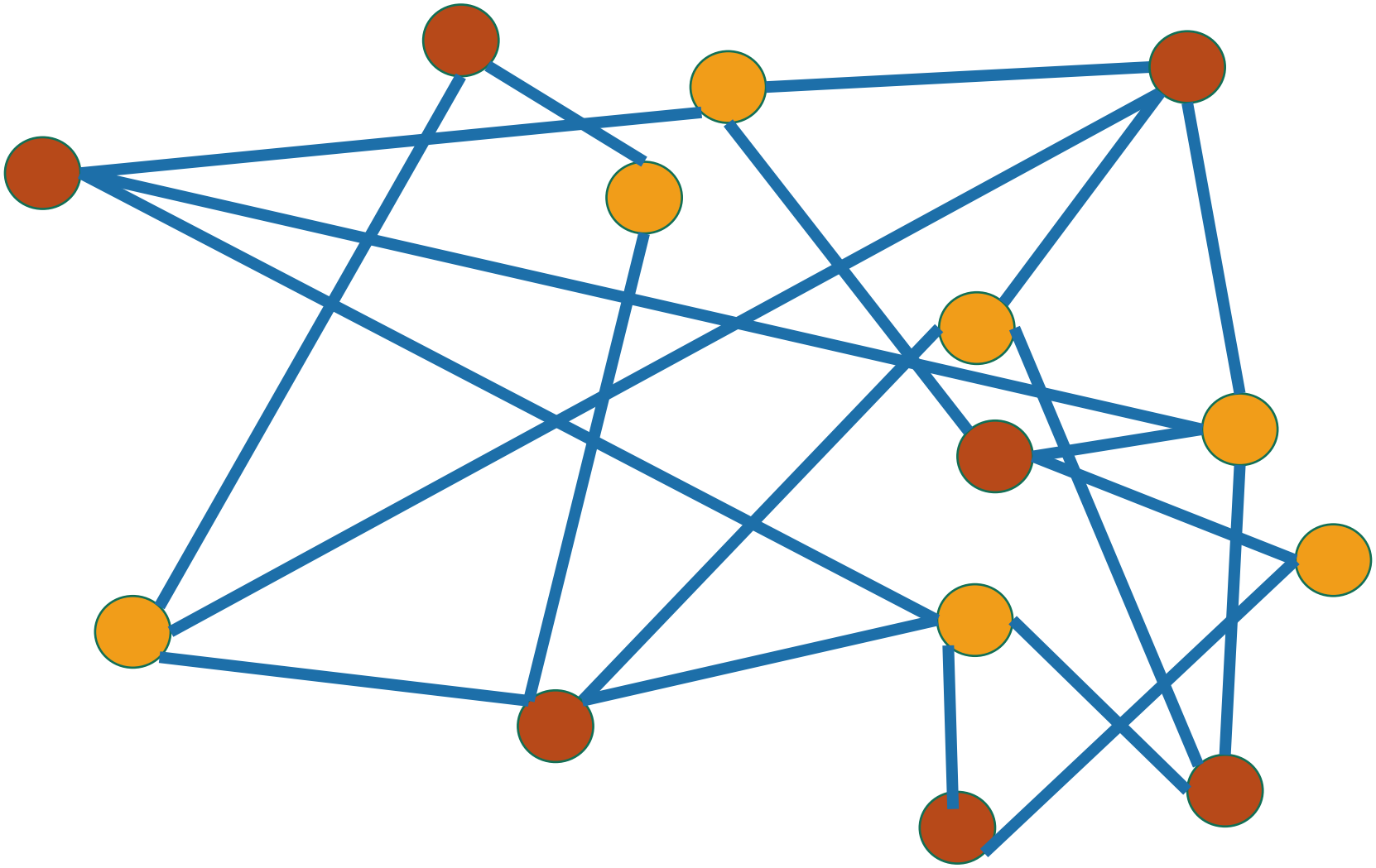
are classes

if the student is enrolled in the class

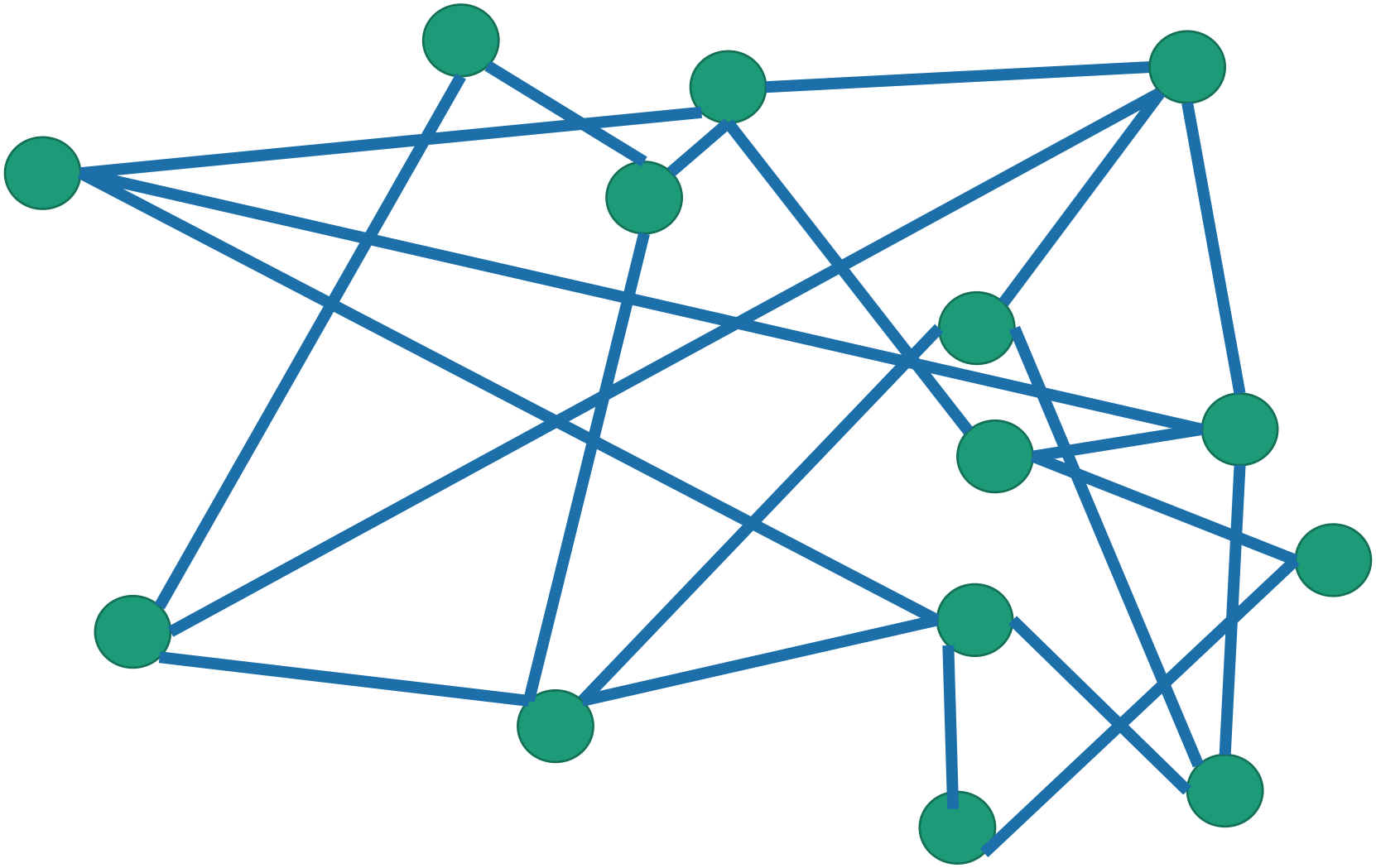
Is this graph bipartite?



How about this one?

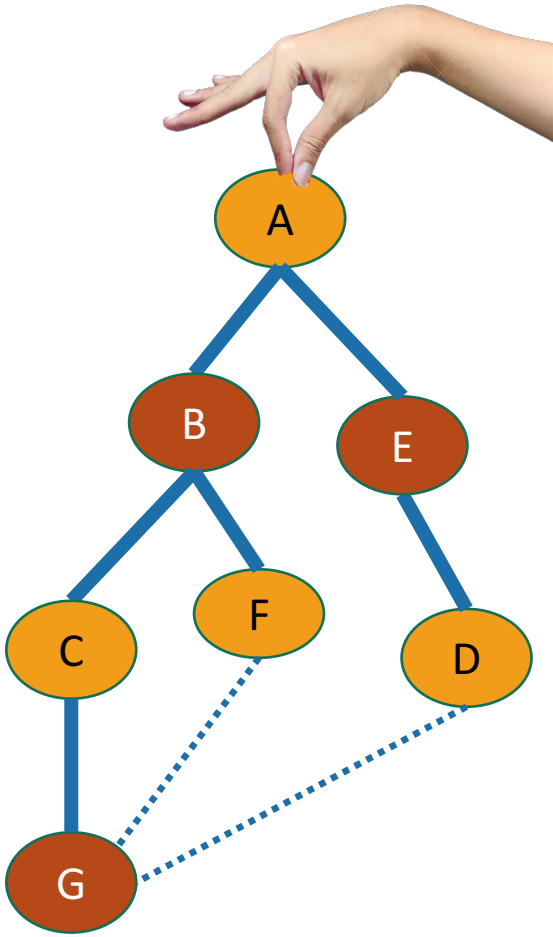


How about this one?



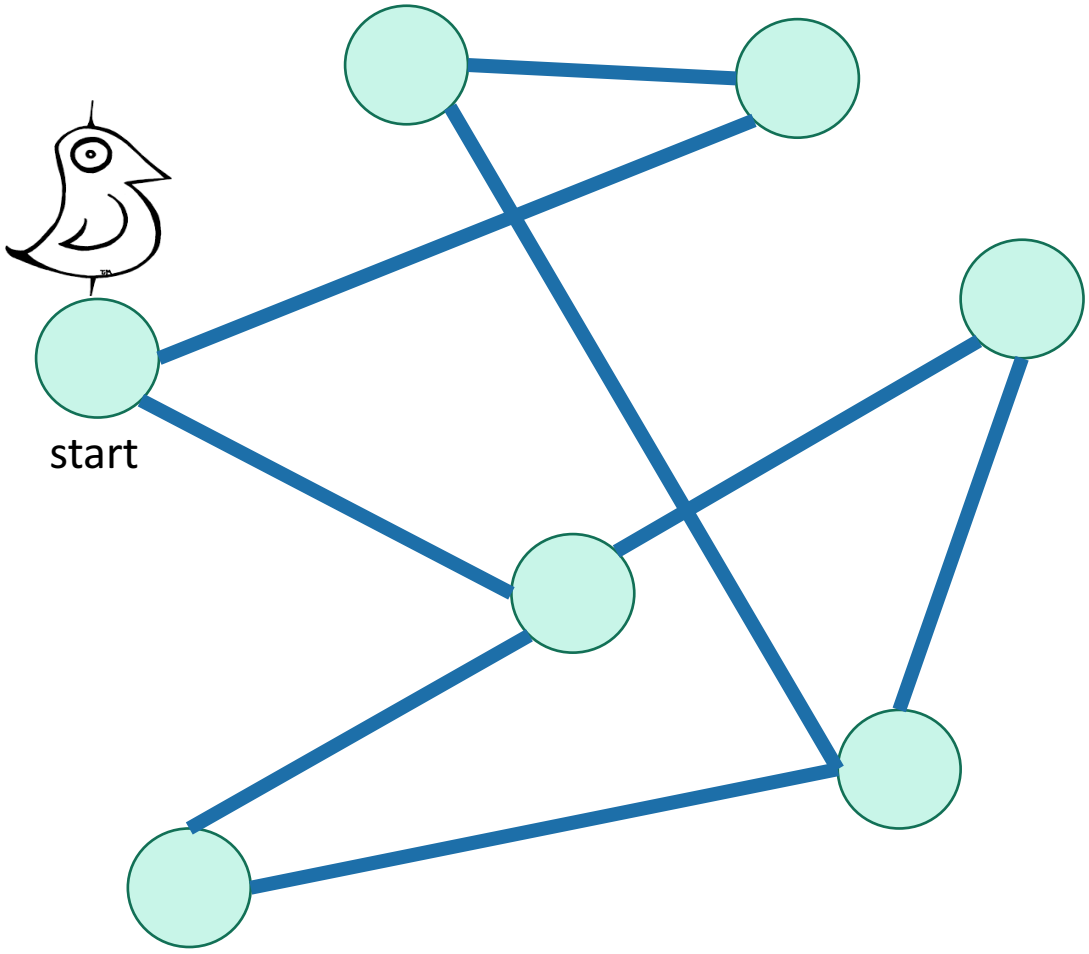
Solution using BFS

- Color the levels of the BFS tree in alternating colors.
- If you ever color a node so that you never color two connected nodes the same, 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

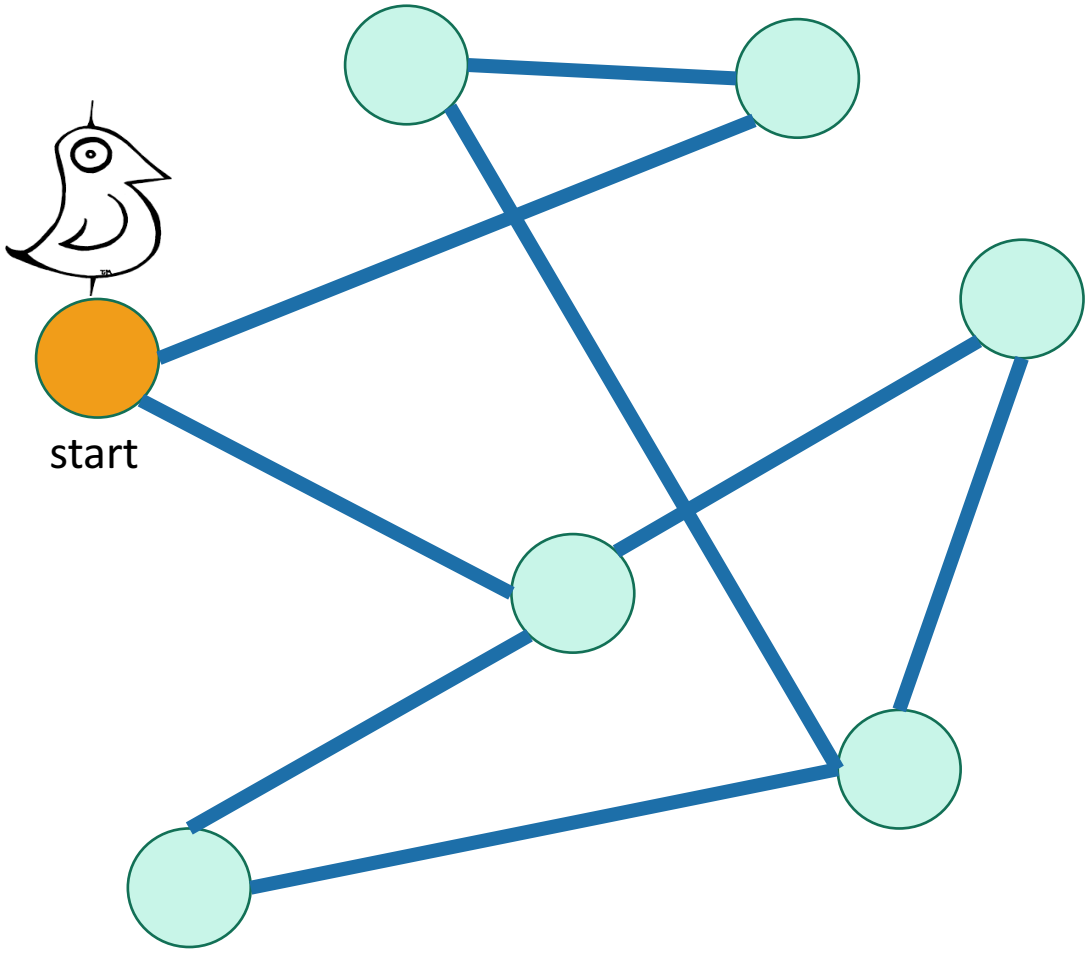
Can reach there in one step

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

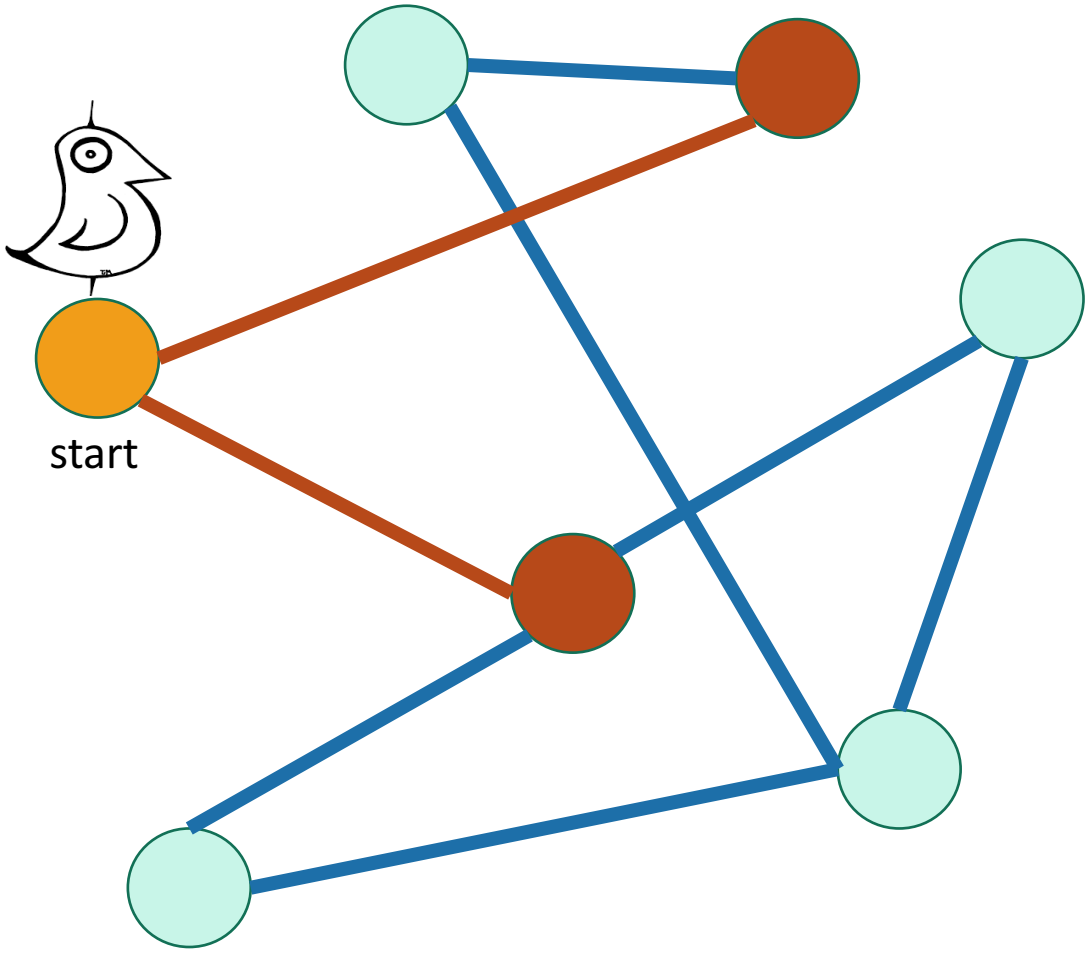
Can reach there in one step

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

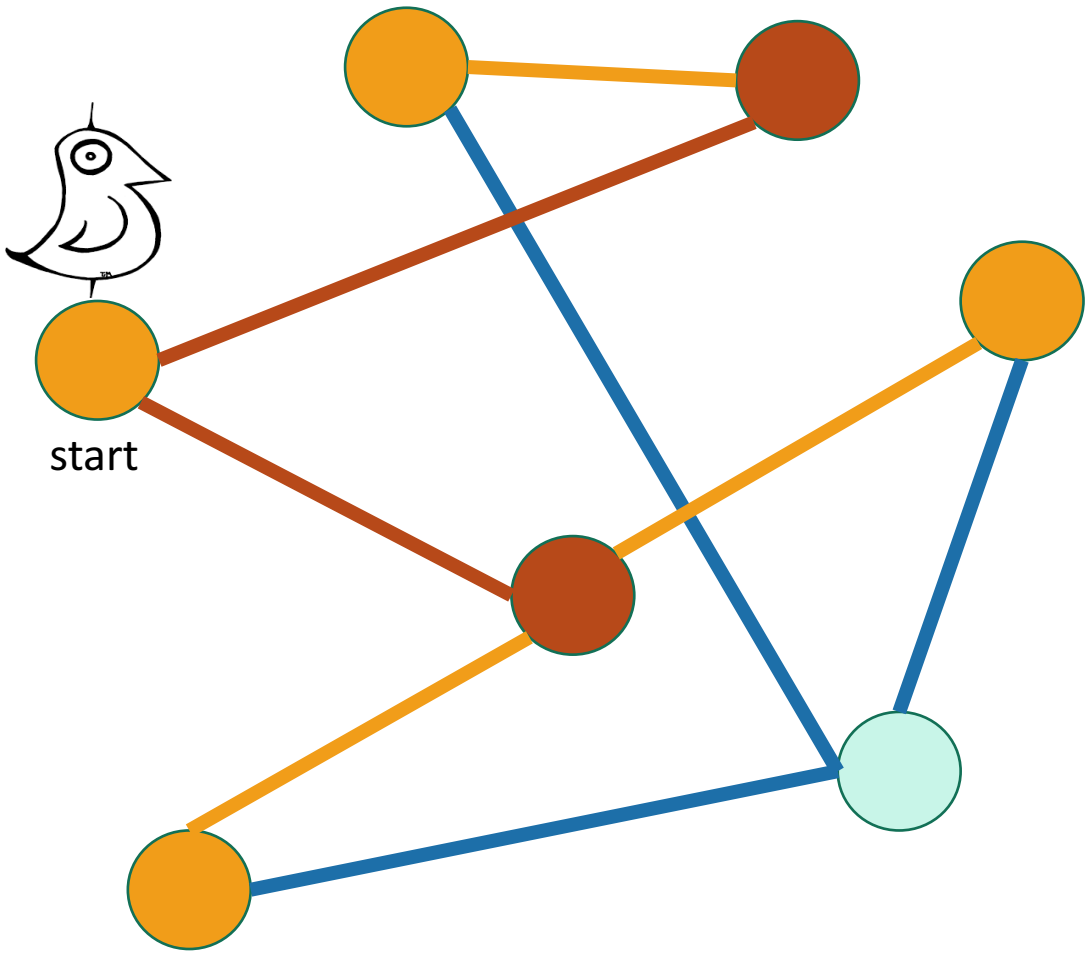
Can reach there in one step

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

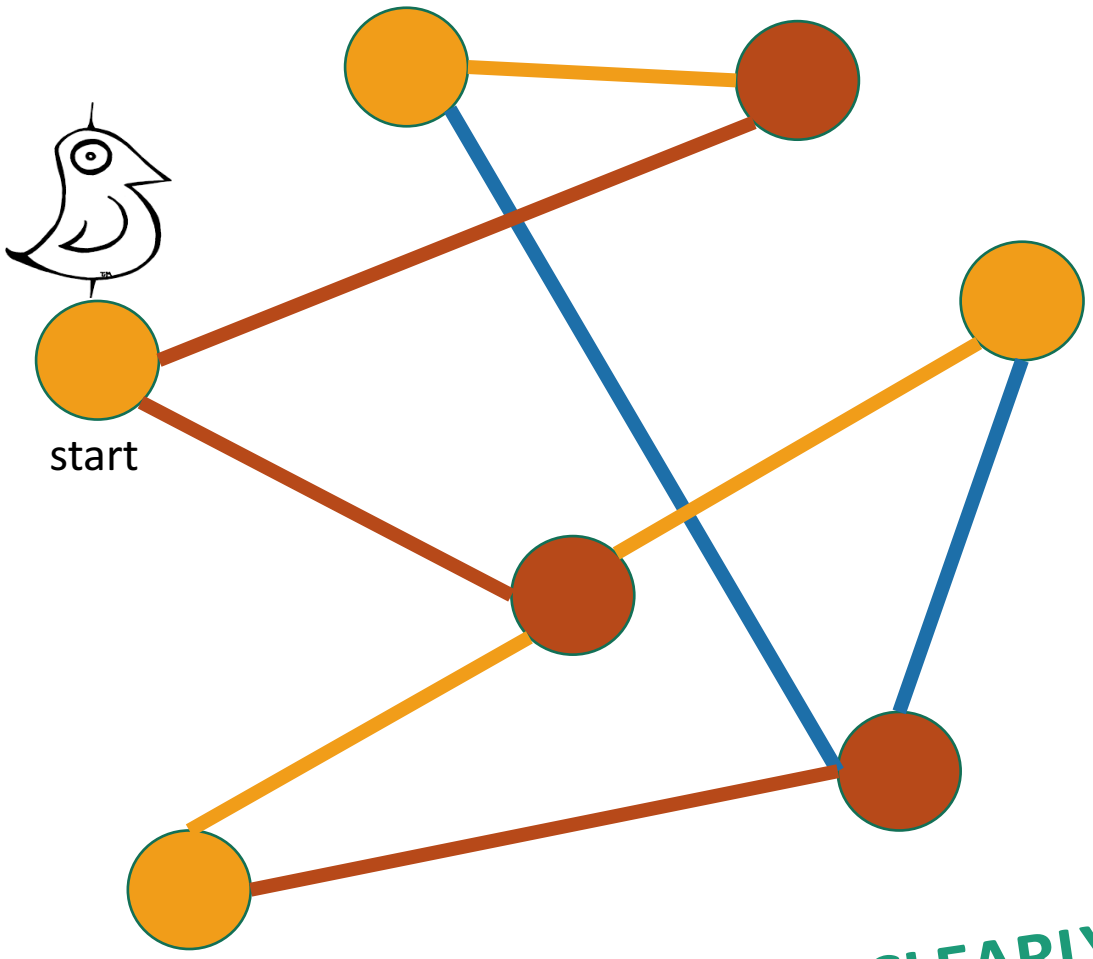
Can reach there in one step

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

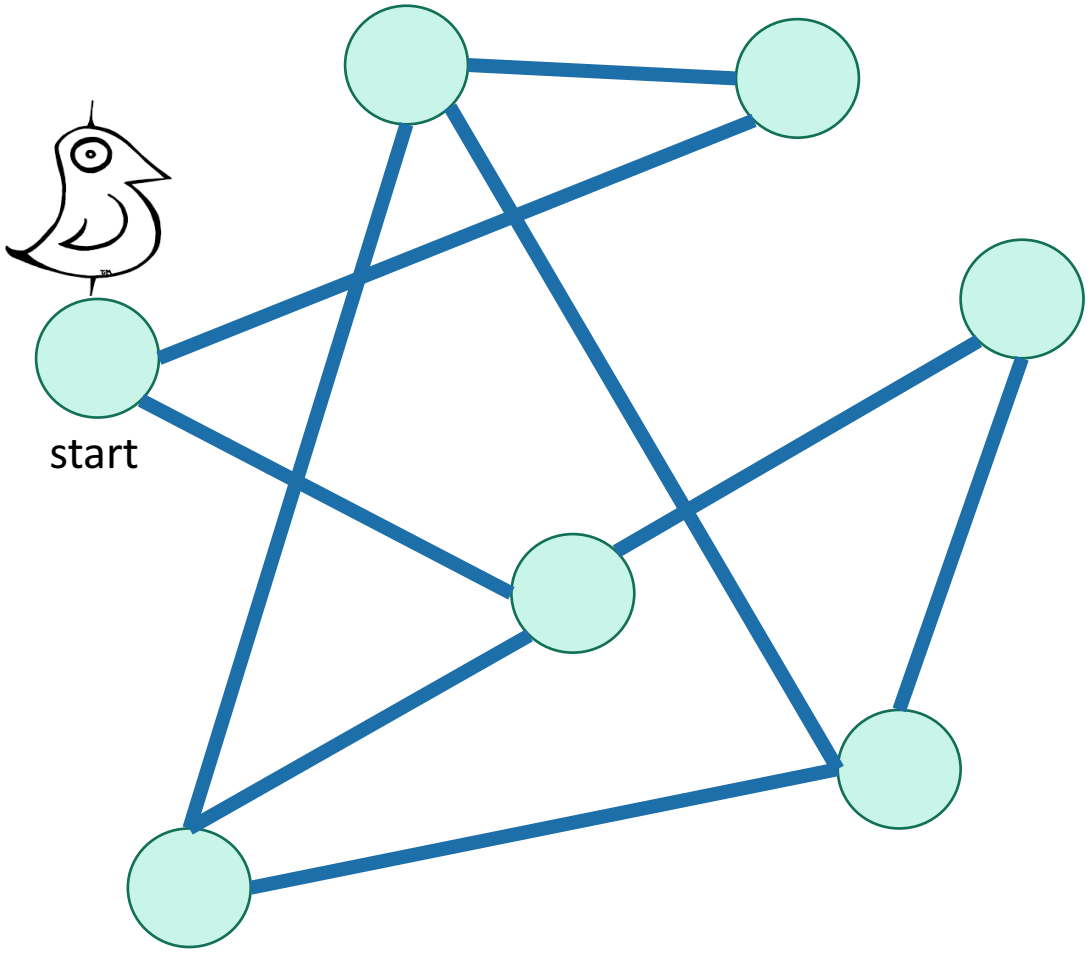
Can reach there in two steps

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

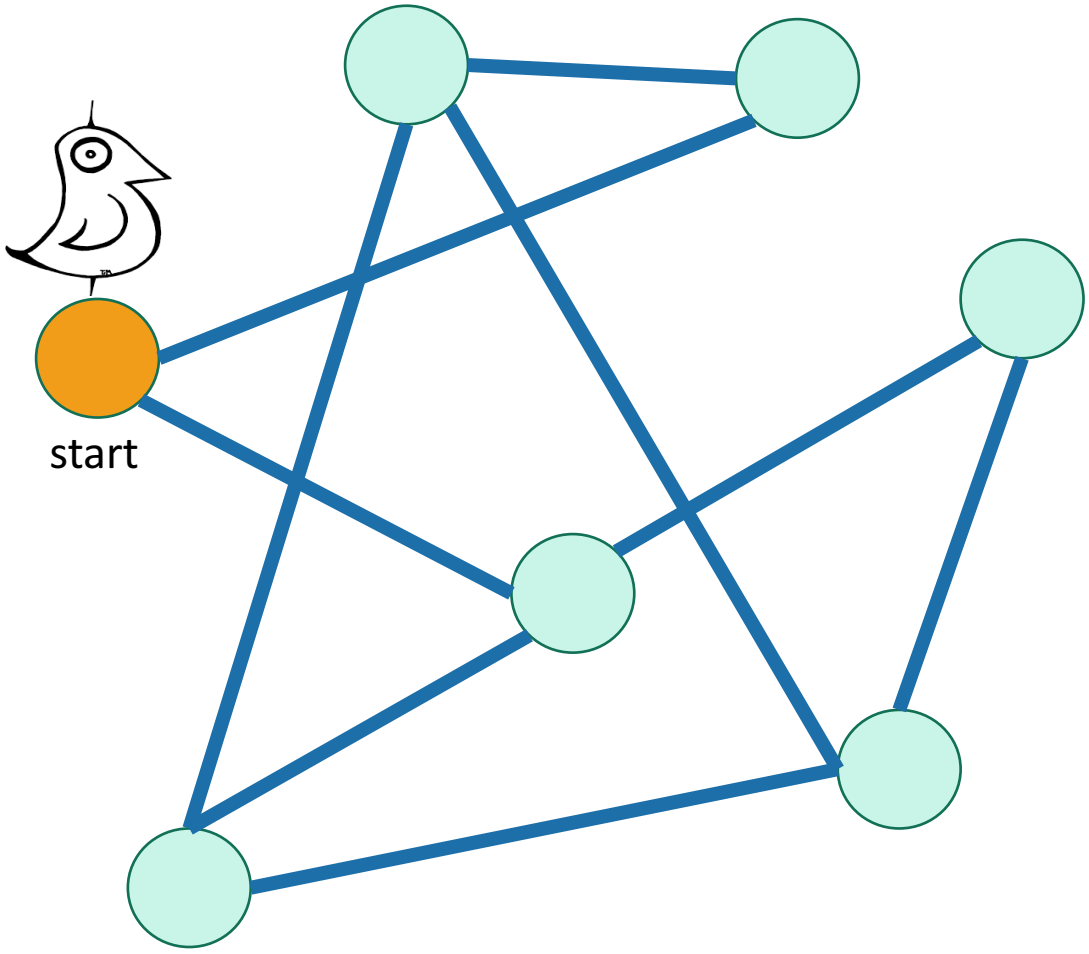
Can reach there in one step

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

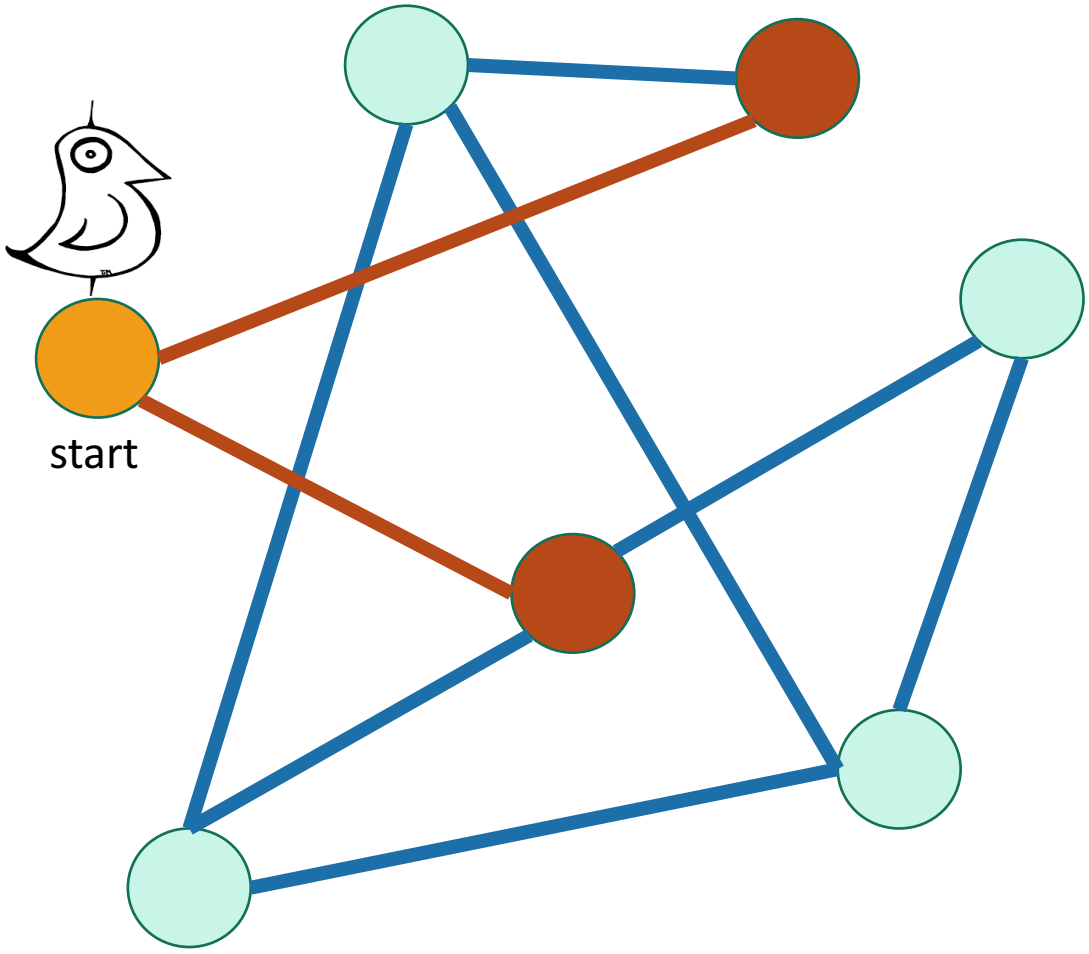
Can reach there in one step

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

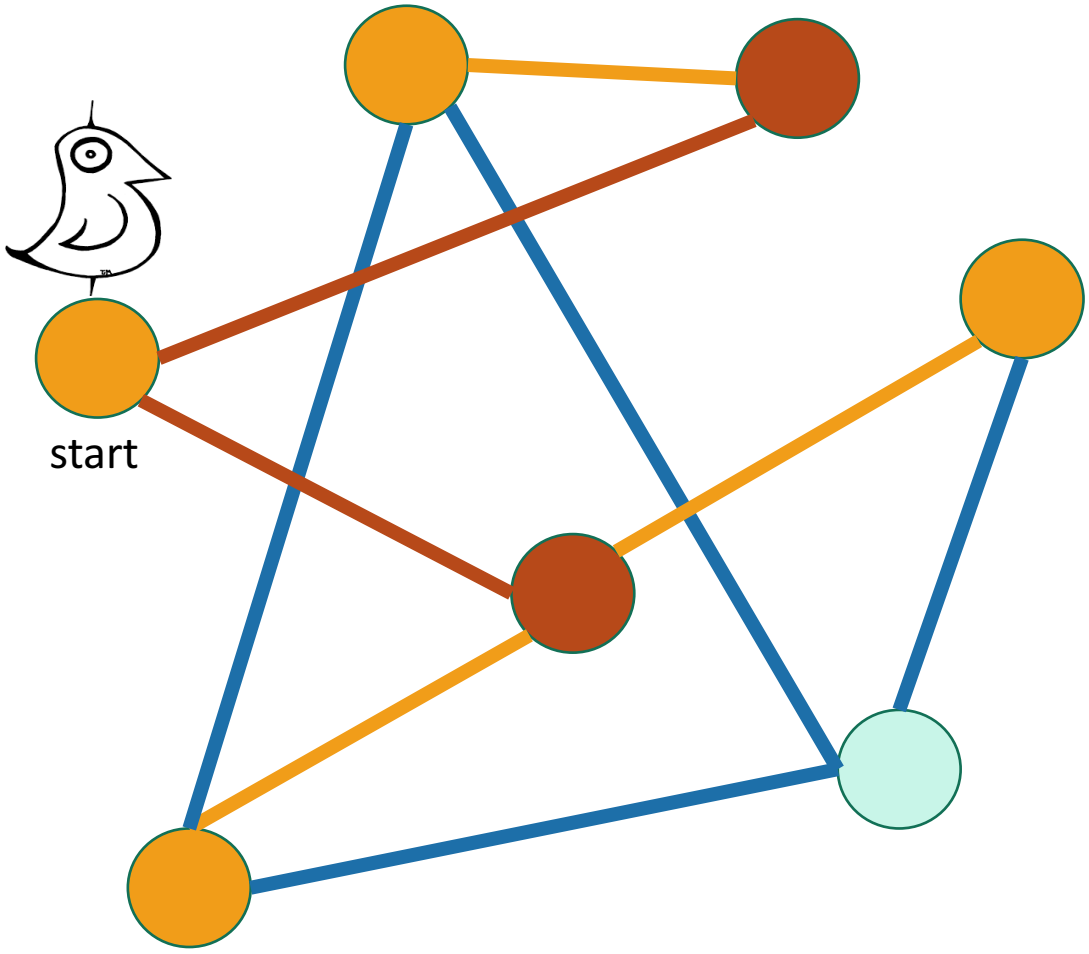
Can reach there in one step

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

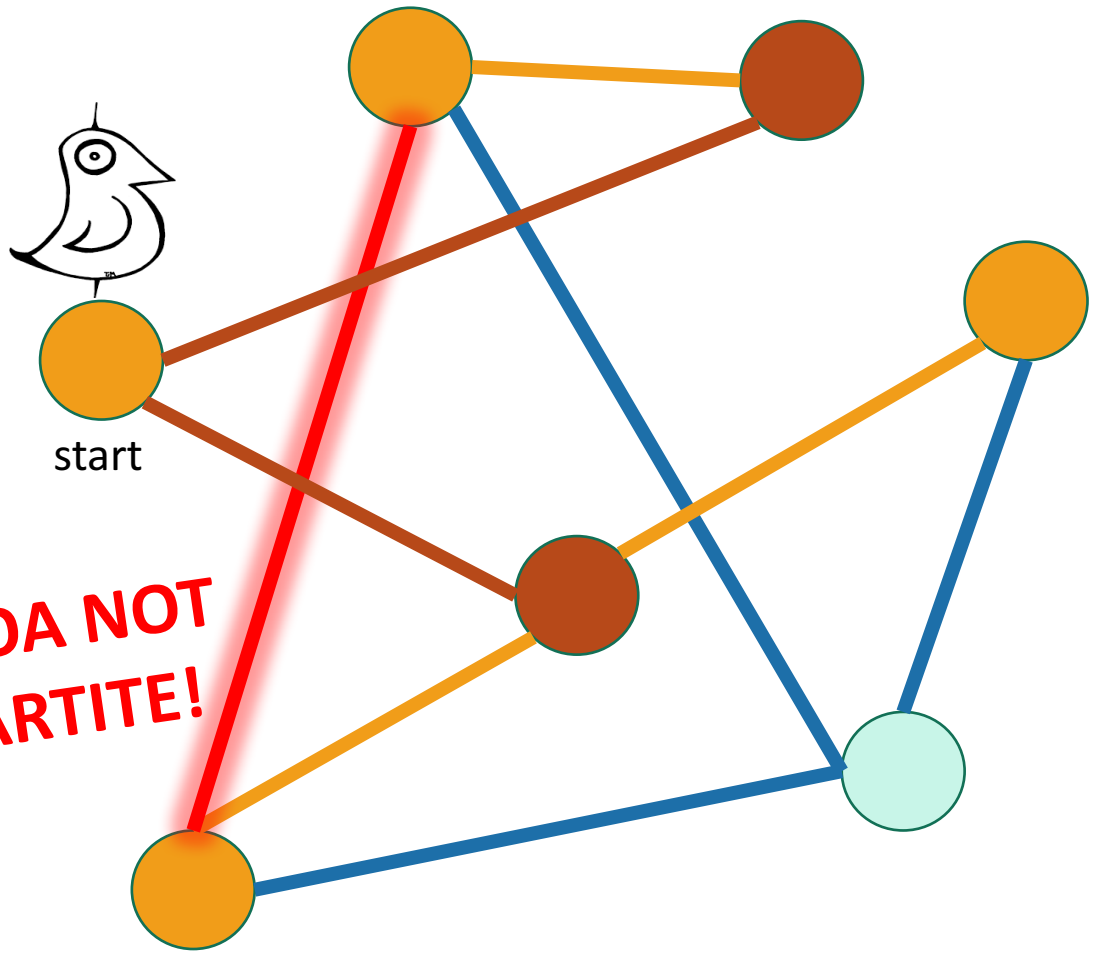
Can reach there in one step

Can reach there in two steps

Can reach there in three steps

Breadth-First Search

For testing bipartite-ness

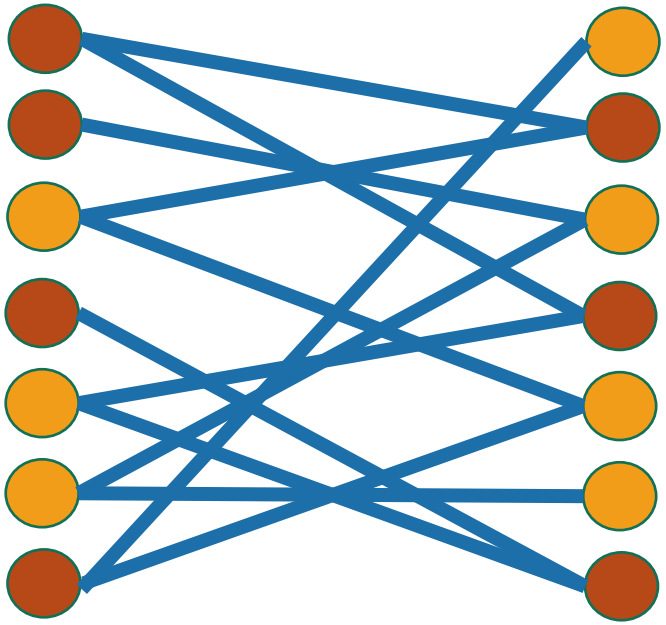


WHOA NOT BIPARTITE!

- 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.

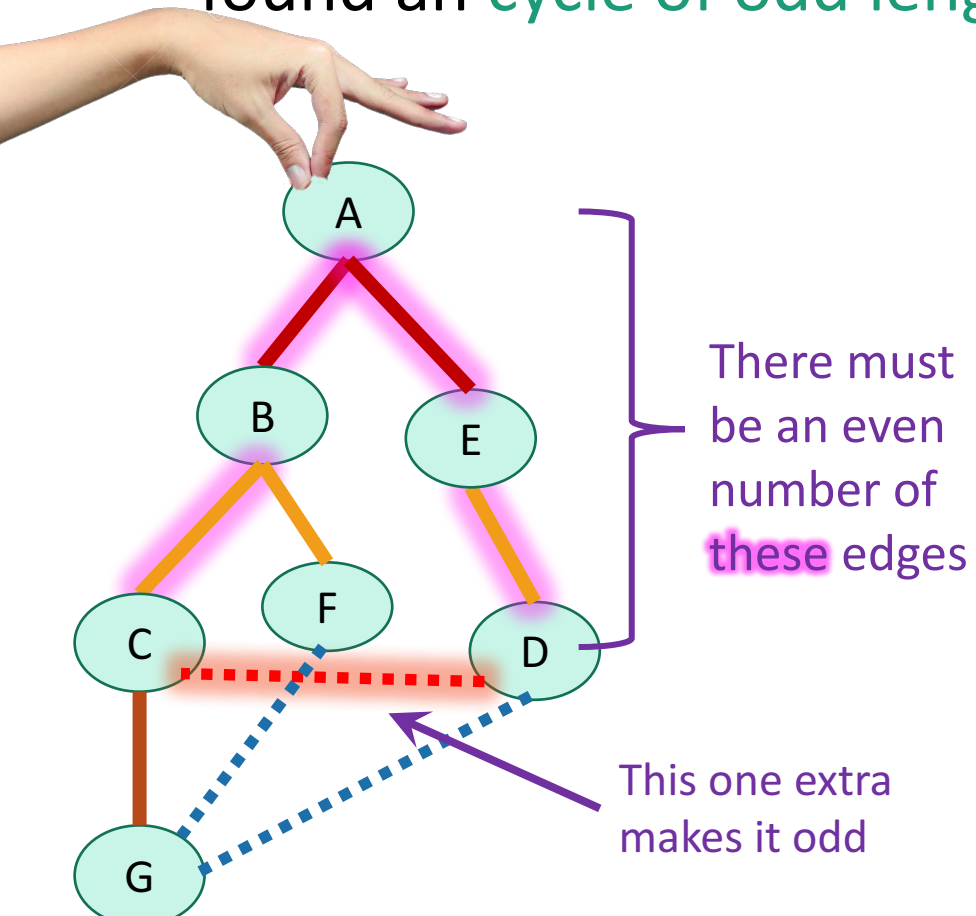
- Just because **this** coloring doesn't work, why does that mean that there is **no** coloring that works?

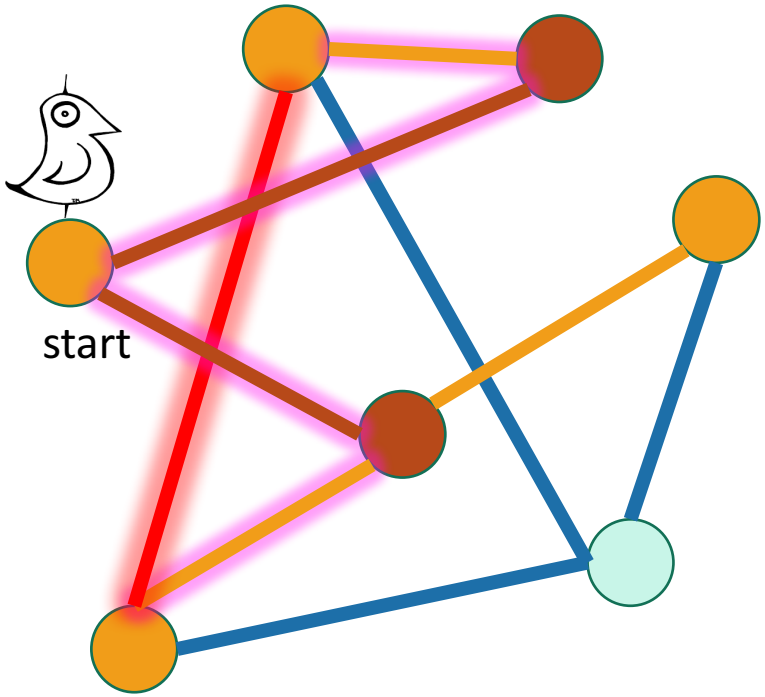


I can come up with plenty of bad colorings on this legitimately bipartite graph…

Some proof required

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

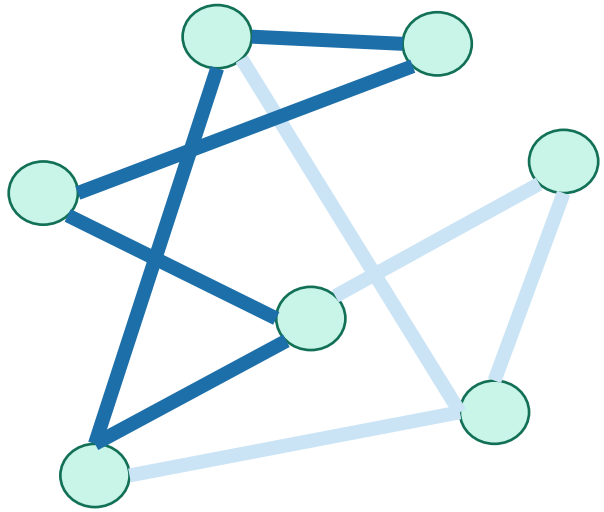




Some proof required

- 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.**



What did we just learn?

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

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