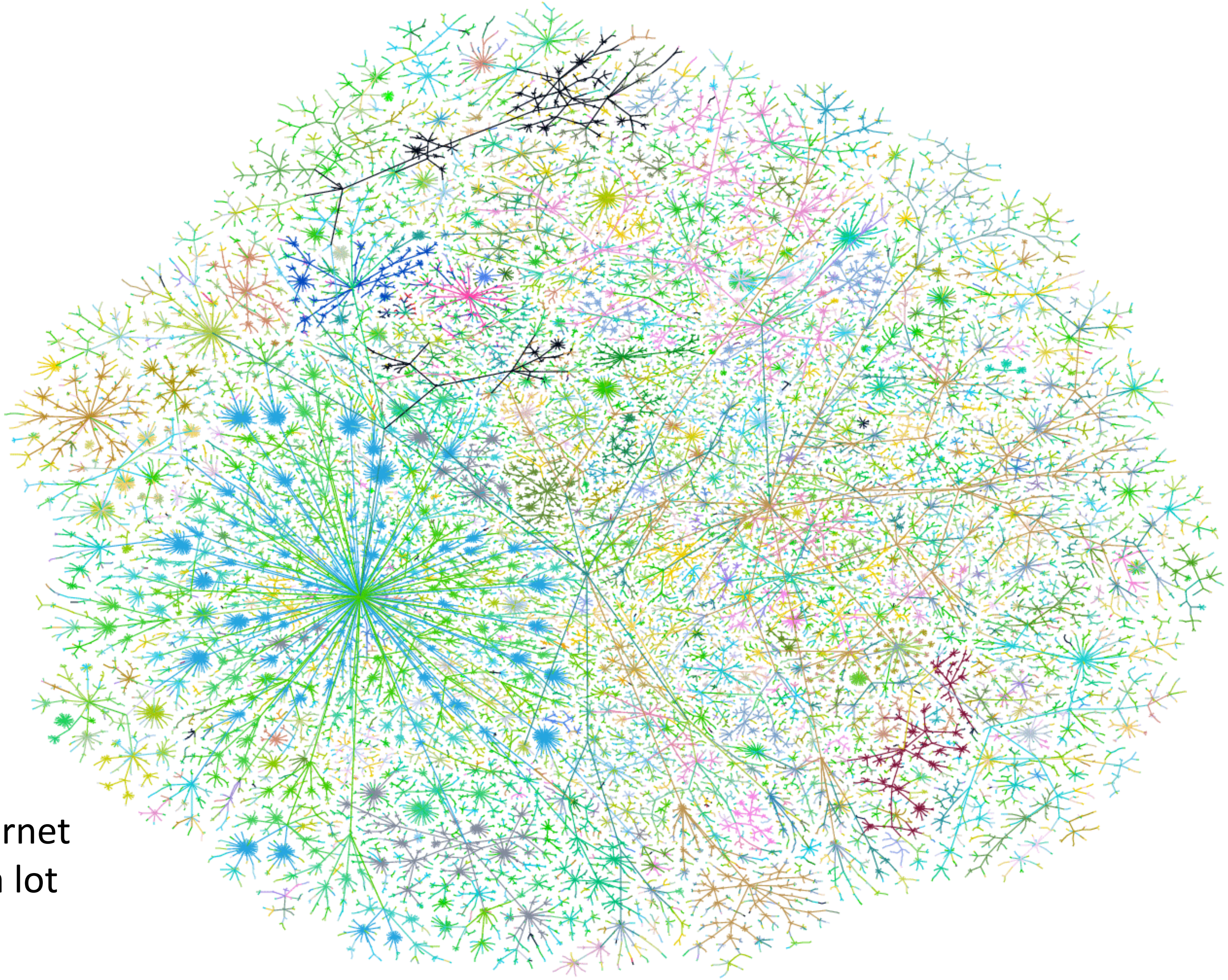
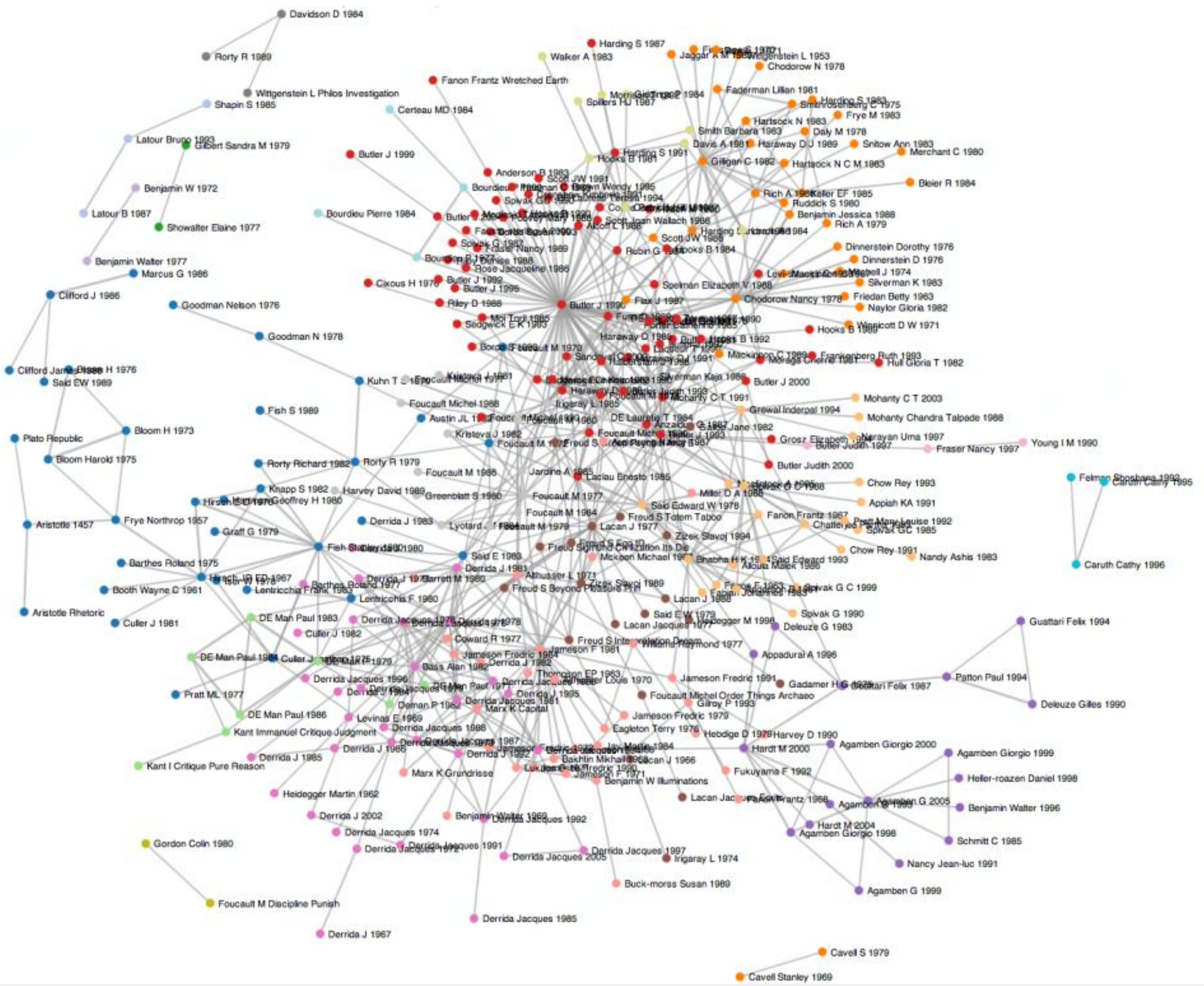
Graphs

Graphs



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

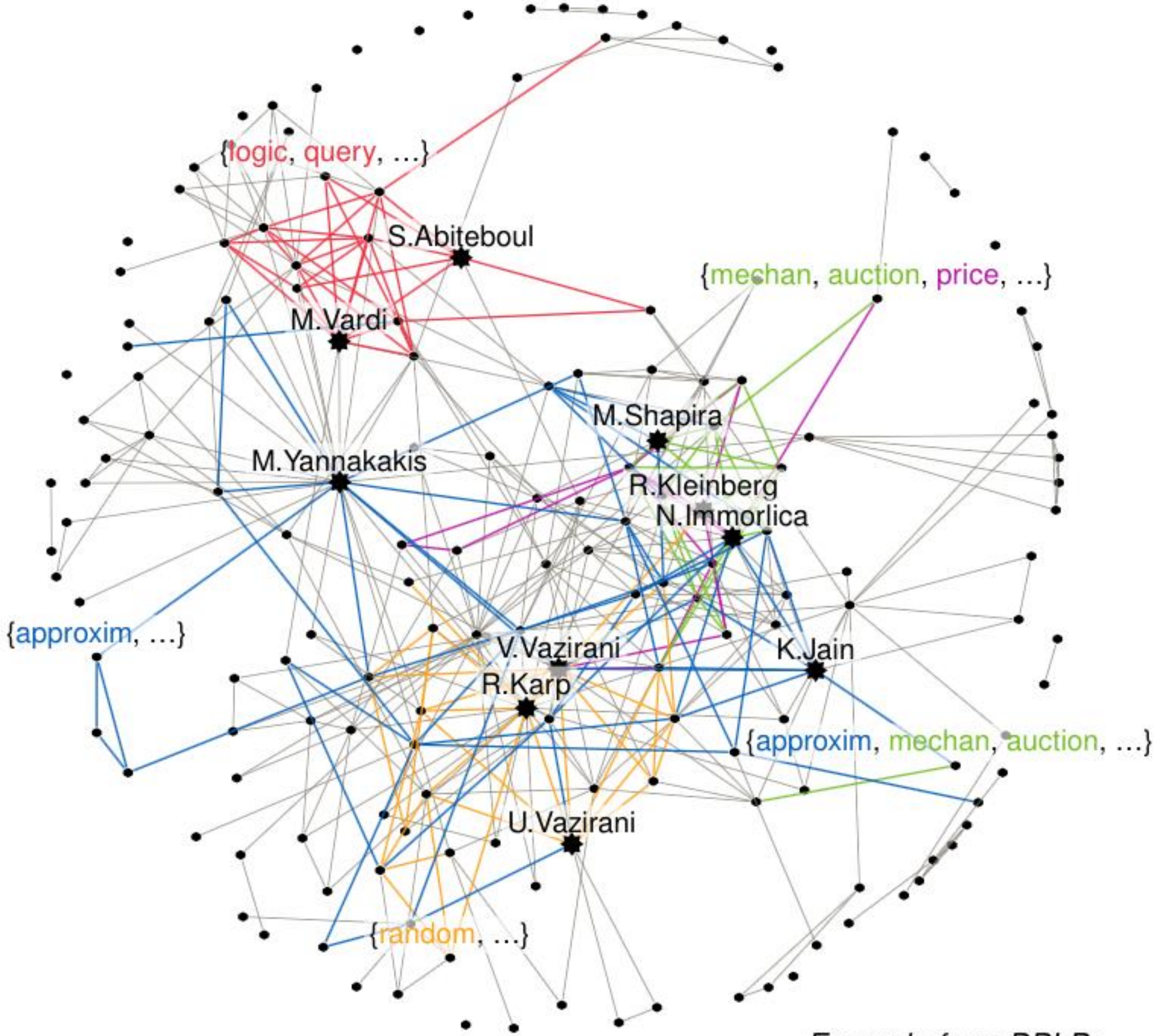
Graphs



Citation graph of literary theory academic papers

Graphs

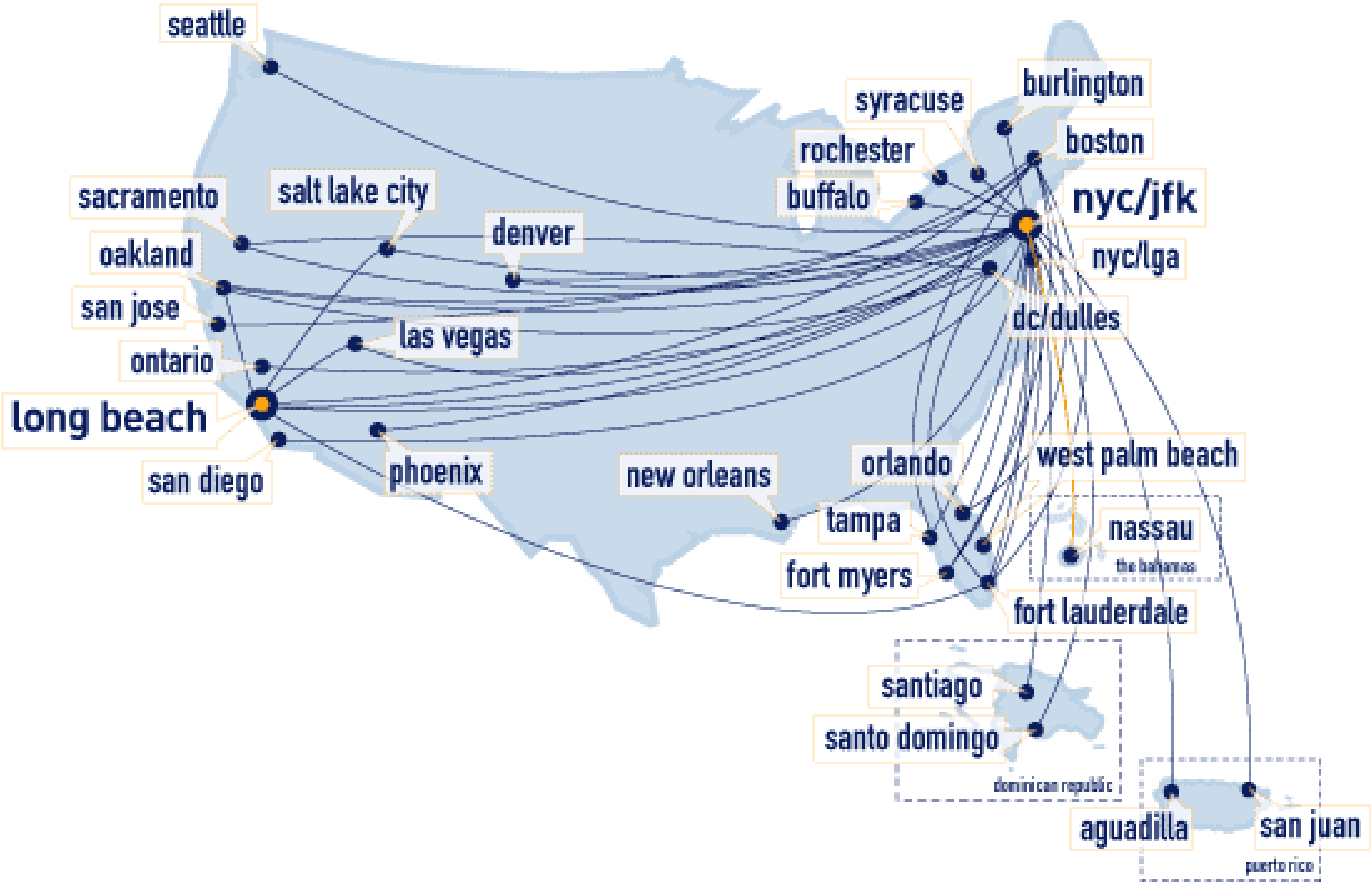
Theoretical Computer Science academic communities



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

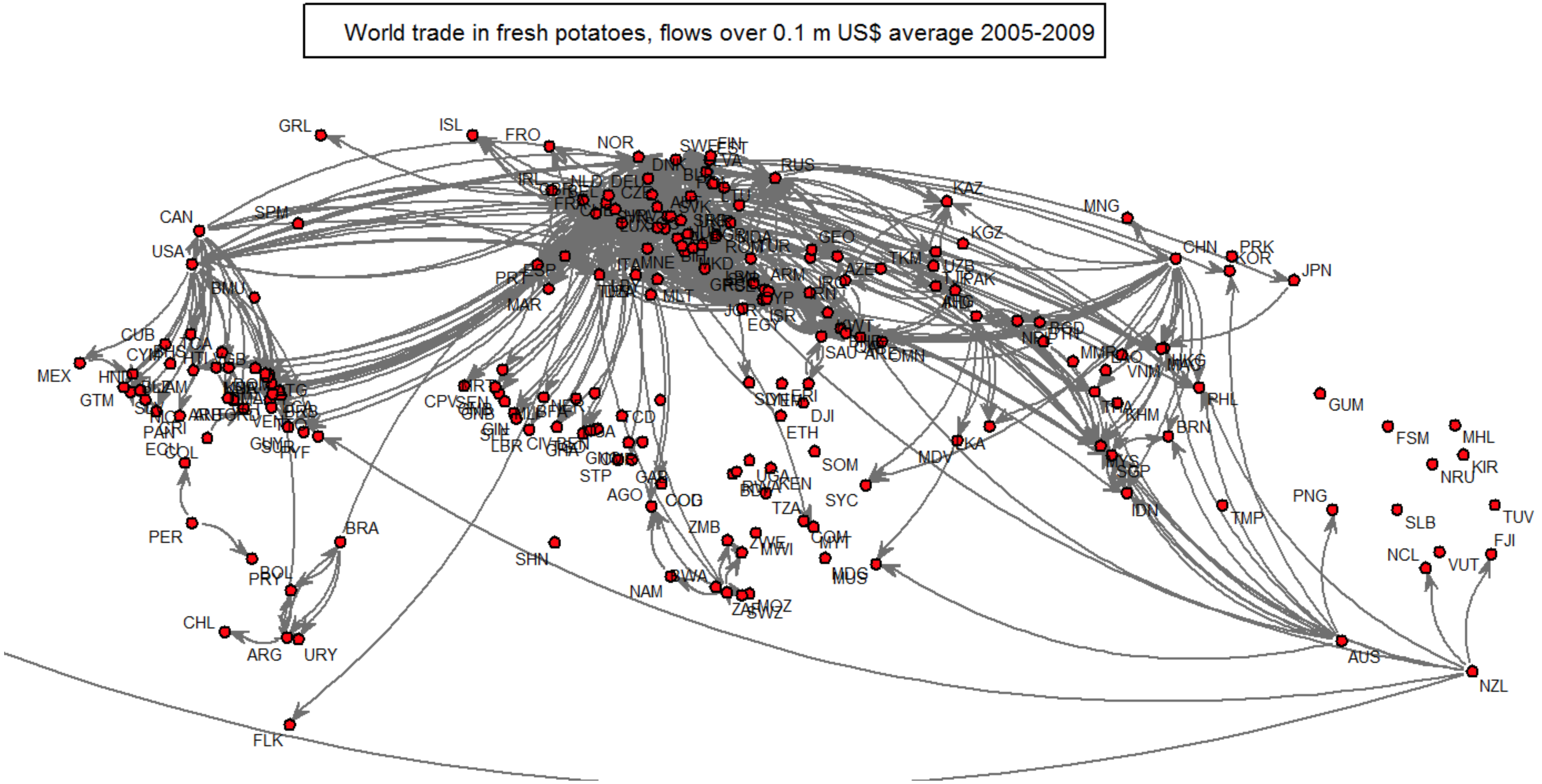
Graphs

jetblue flights



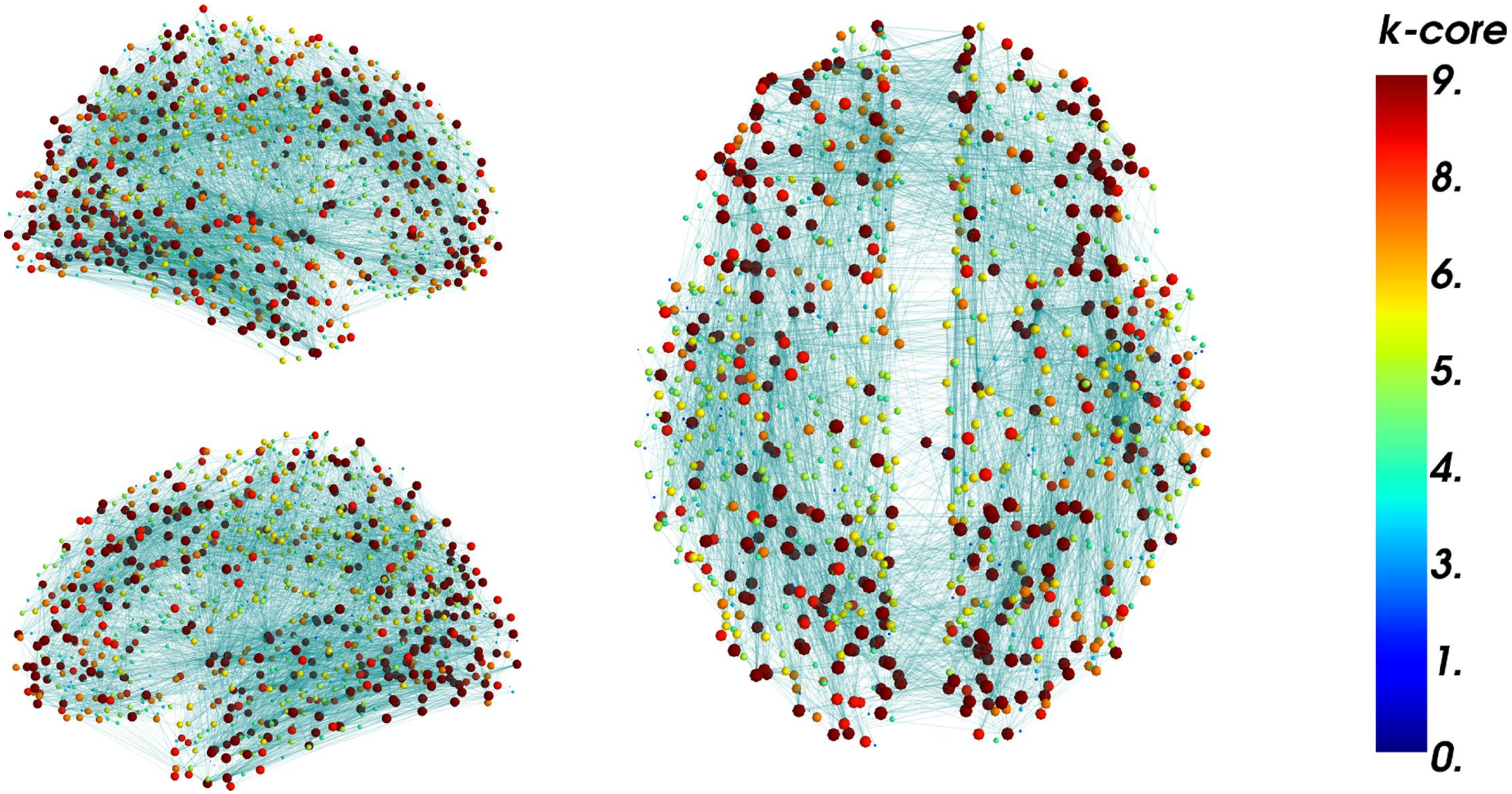
Graphs

Potato trade



Graphs

Neural connections
in the brain



Graphs

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

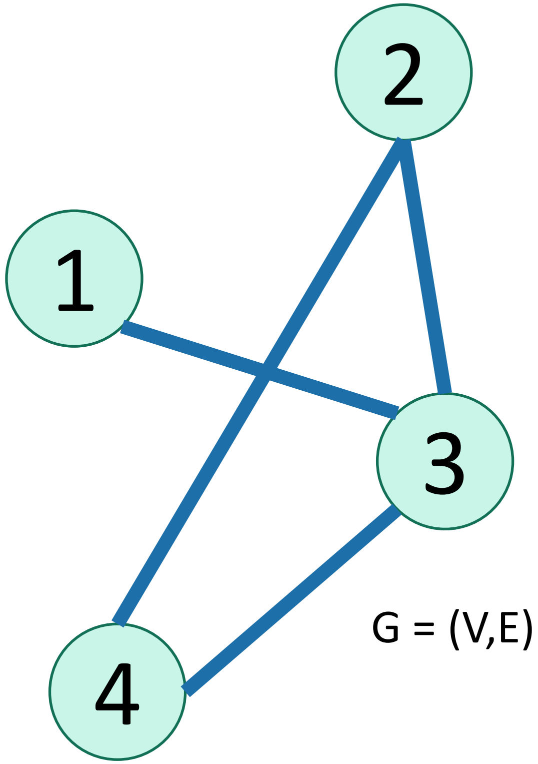
- We want to answer questions about them.
  - Efficient routing?
  - Community detection/clustering?
  - An ordering that respects dependencies?

- 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} }

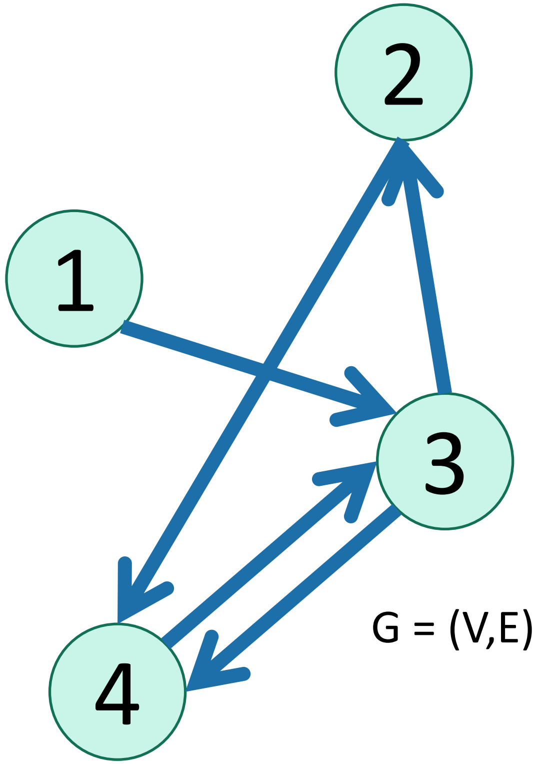


- The degree of vertex 4 is 2.
  - There are 2 edges coming out.
- Vertex 4's neighbors are 2 and 3

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

Graph Representation

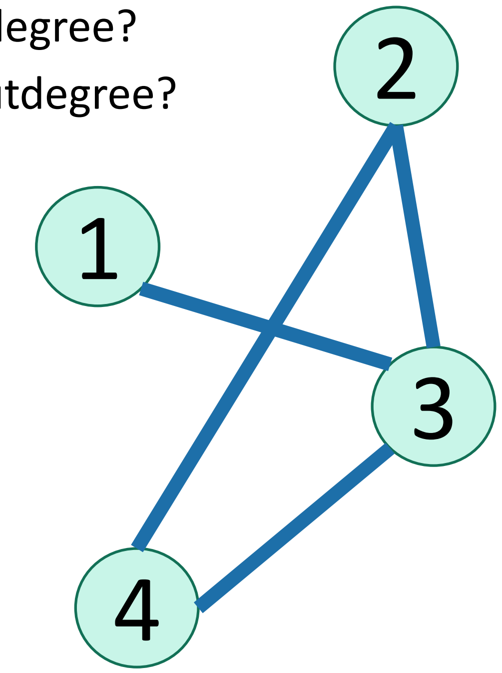
How do we represent graphs?

- Option 1: adjacency matrix

Indegree?

Outdegree?

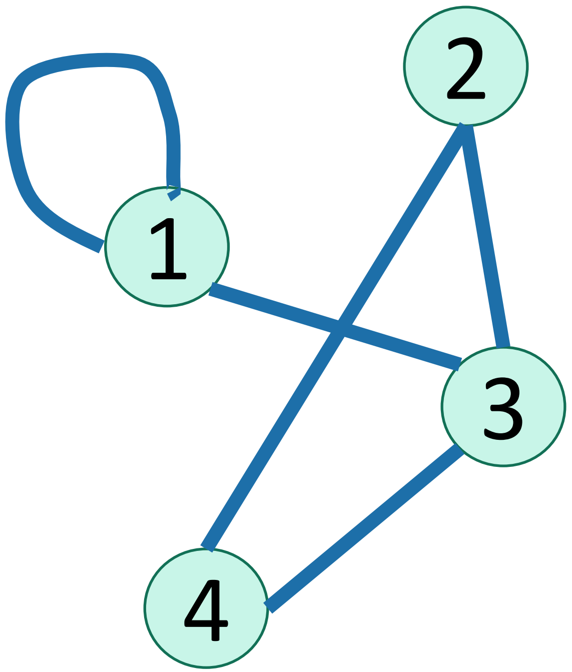
$$
\begin{array}{c c}
 & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}
\end{array}
$$



How do we represent graphs?

- Option 1: adjacency matrix

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



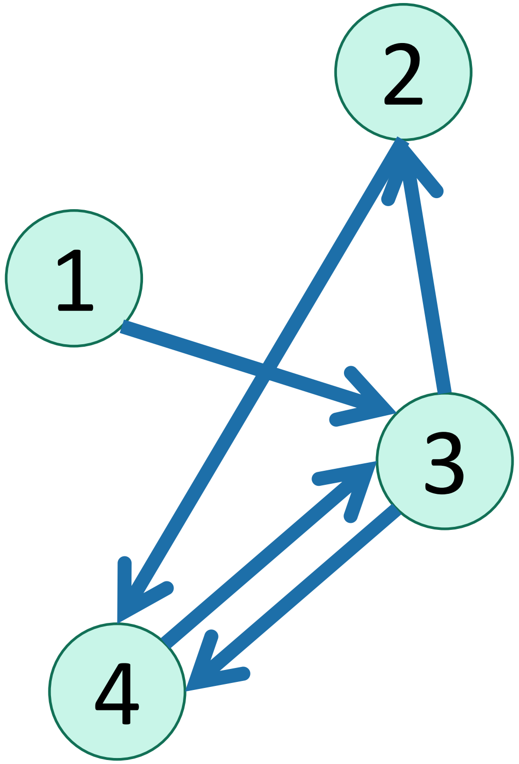
How do we represent graphs?

- Option 1: adjacency matrix

Destination

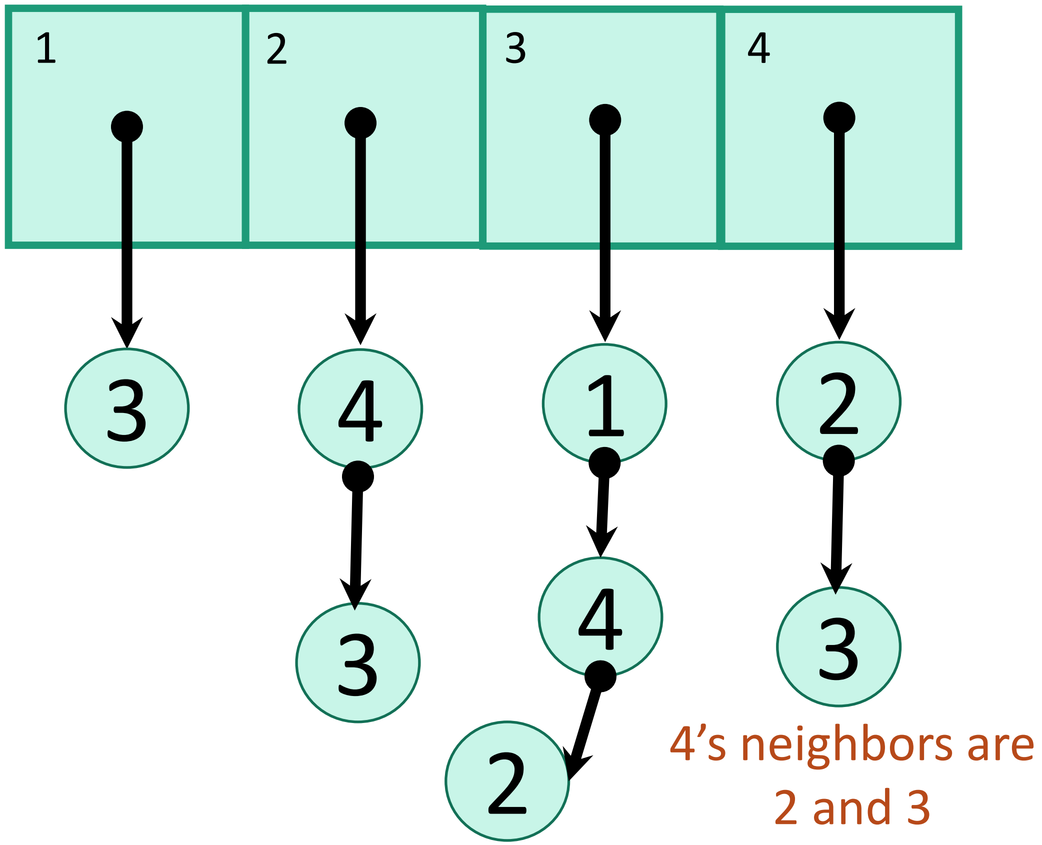
Source

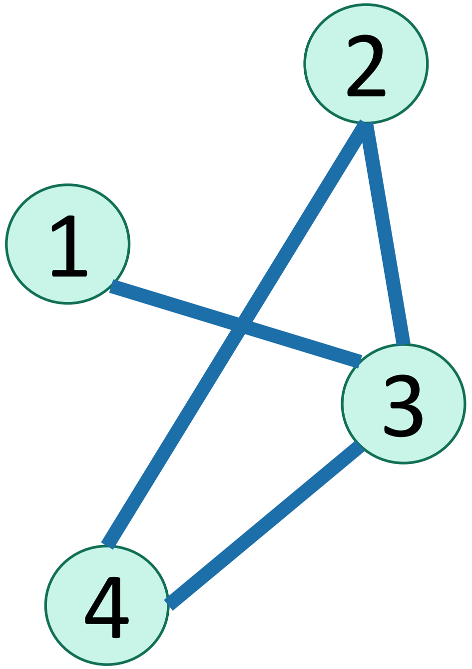
$$\begin{array}{cc} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \end{array}$$



How do we represent graphs?

- Option 2: linked lists.





How would you modify this for directed graphs?

In either case

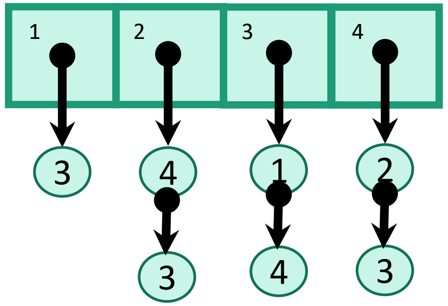
- May think of vertices storing other information
  - Attributes (name, IP address, …)
  - helper info for algorithms that we will perform on the graph
- We will want to be able to do the following ops:
  - Edge Membership: Is edge e in E?
  - Neighbor Query: What are the neighbors of vertex v?

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

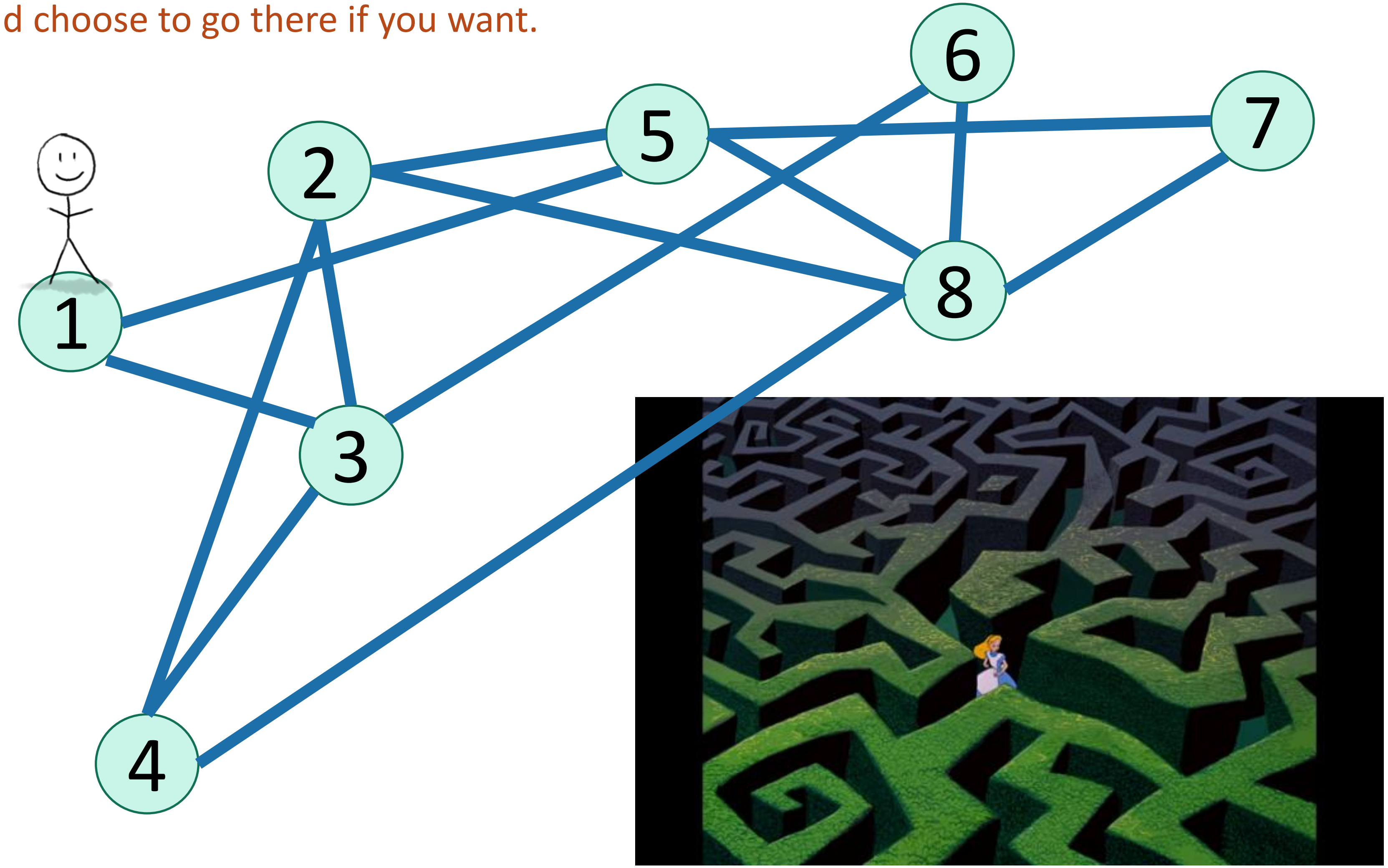
Assignment Question 1

- Design a Graph Class with following functions
  - EdgeMemberShip
  - NeighbourQuery
  - Add Edge
  - Constructor(List of Vertices)

Depth-first search

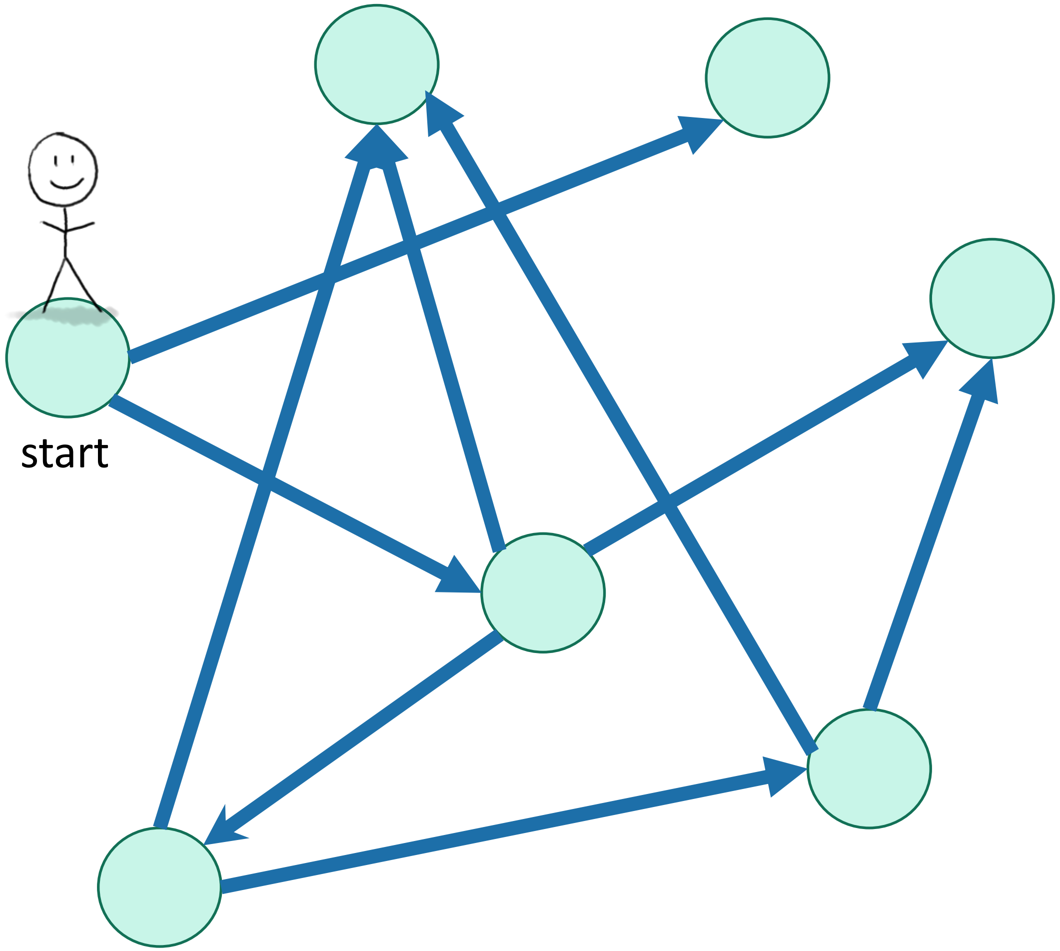
How do we explore a graph?

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



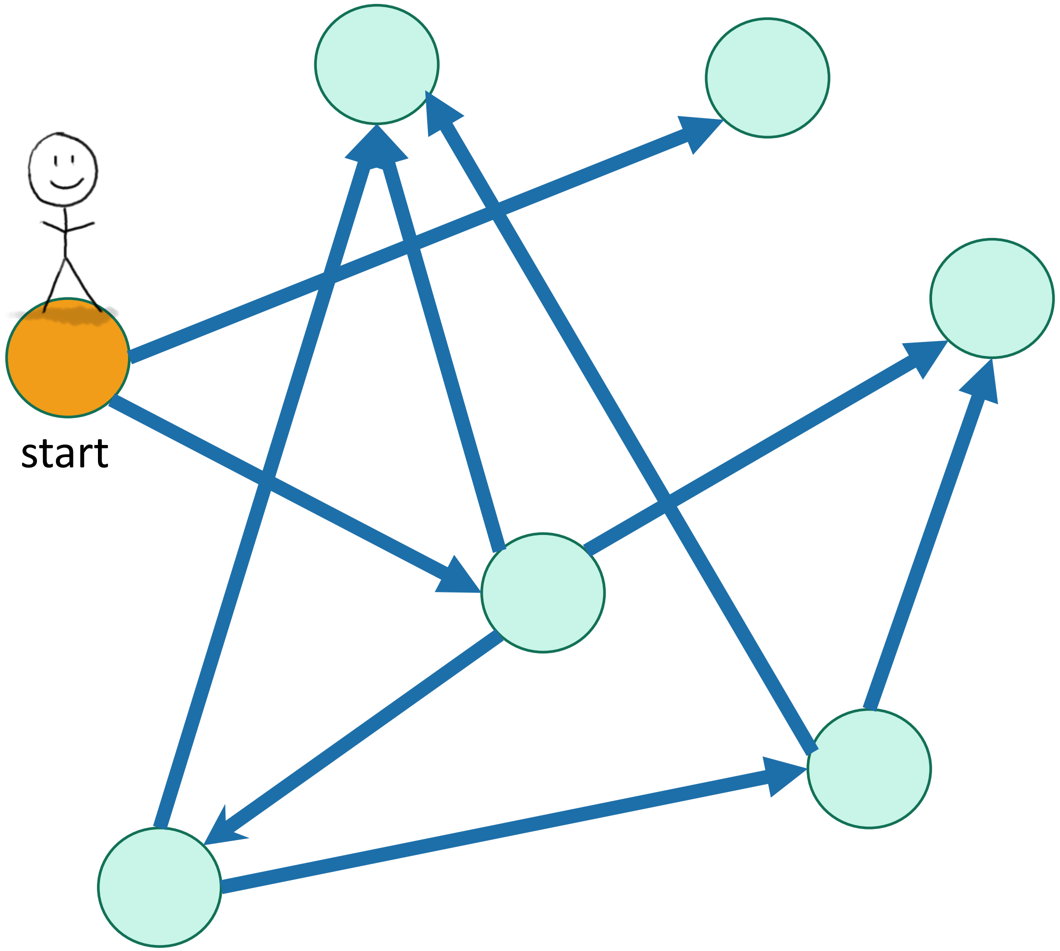
Depth First Search

Exploring a labyrinth with chalk and a piece of string



Depth First Search

Exploring a labyrinth with chalk and a piece of string



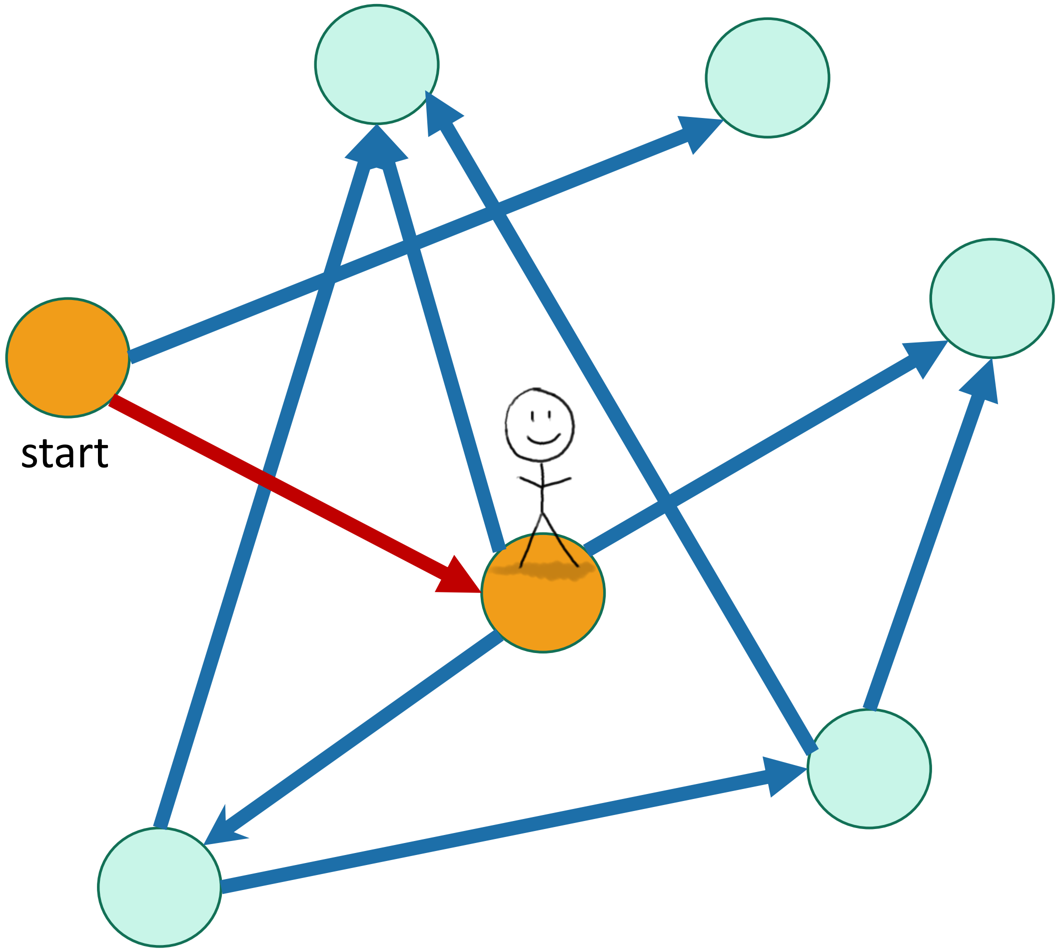
Not been there yet

Been there, haven't explored all the paths out.

Been there, have explored all the paths out.

Depth First Search

Exploring a labyrinth with chalk and a piece of string



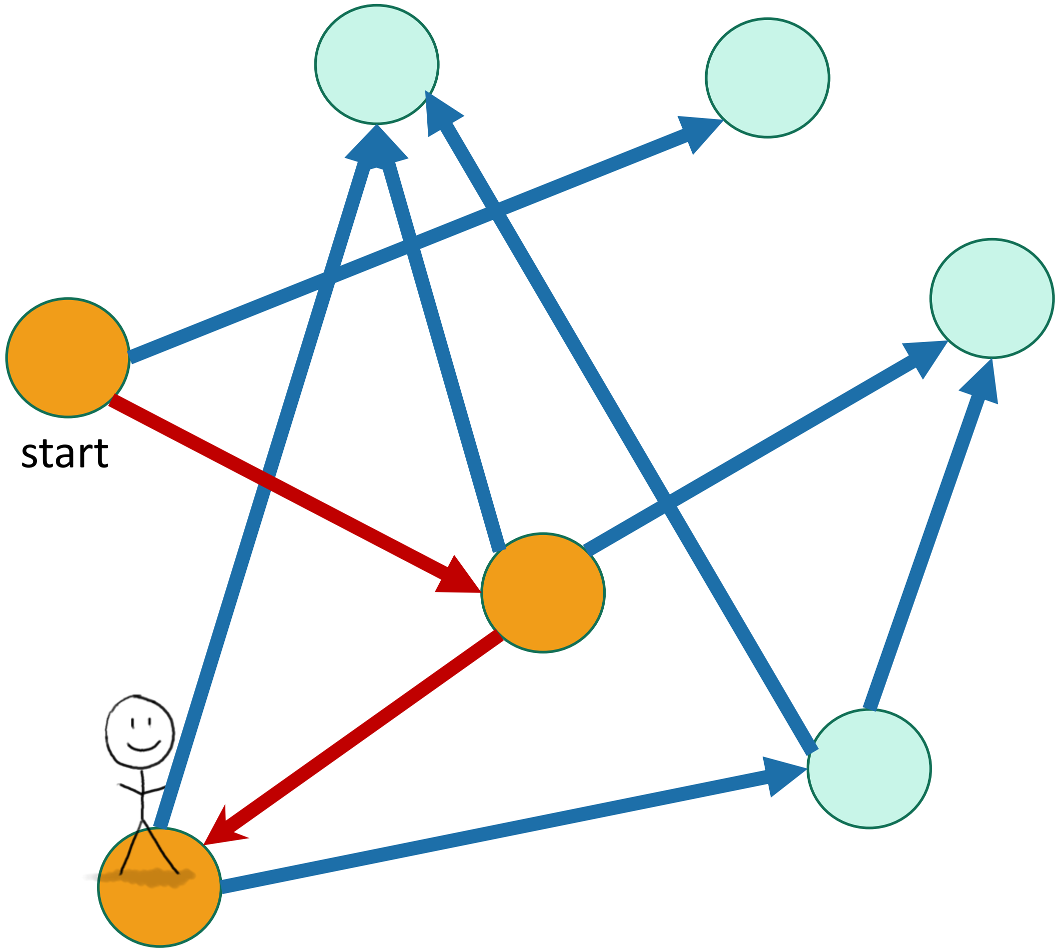
Not been there yet

Been there, haven't explored all the paths out.

Been there, have explored all the paths out.

Depth First Search

Exploring a labyrinth with chalk and a piece of string



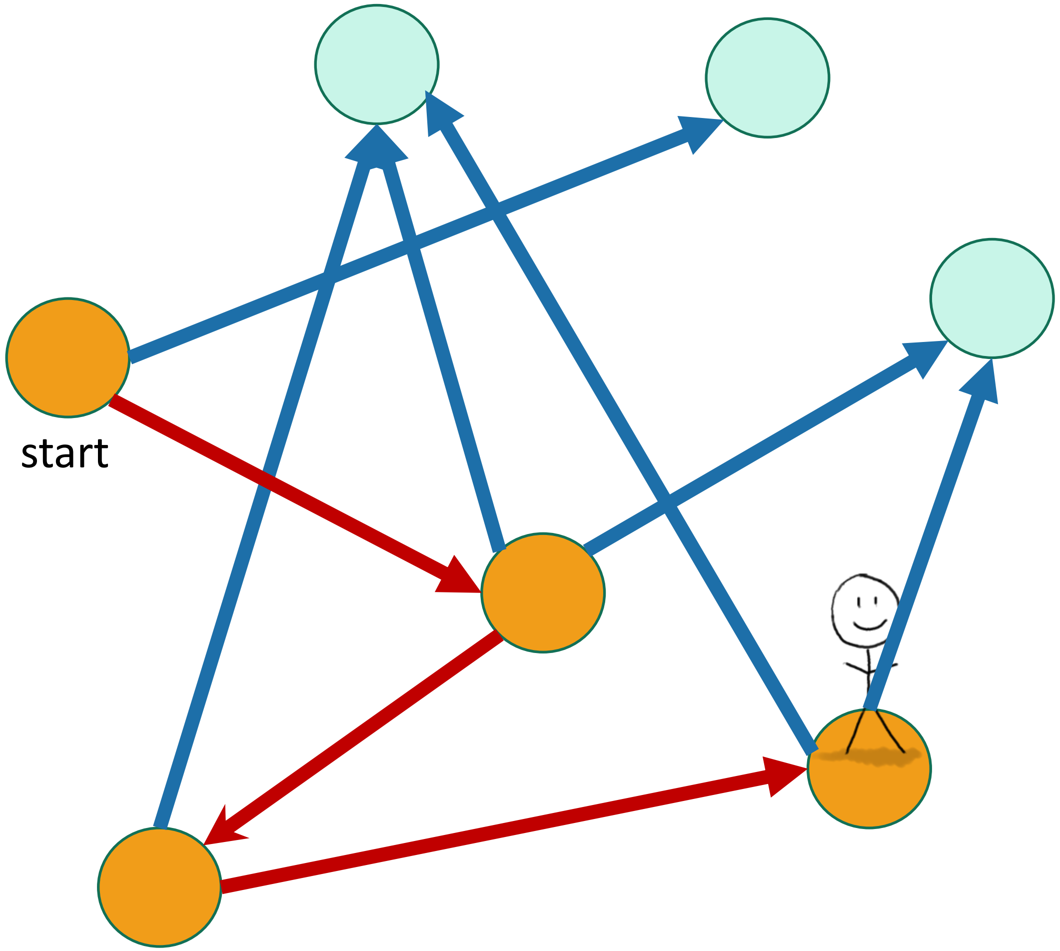
Not been there yet

Been there, haven't explored all the paths out.

Been there, have explored all the paths out.

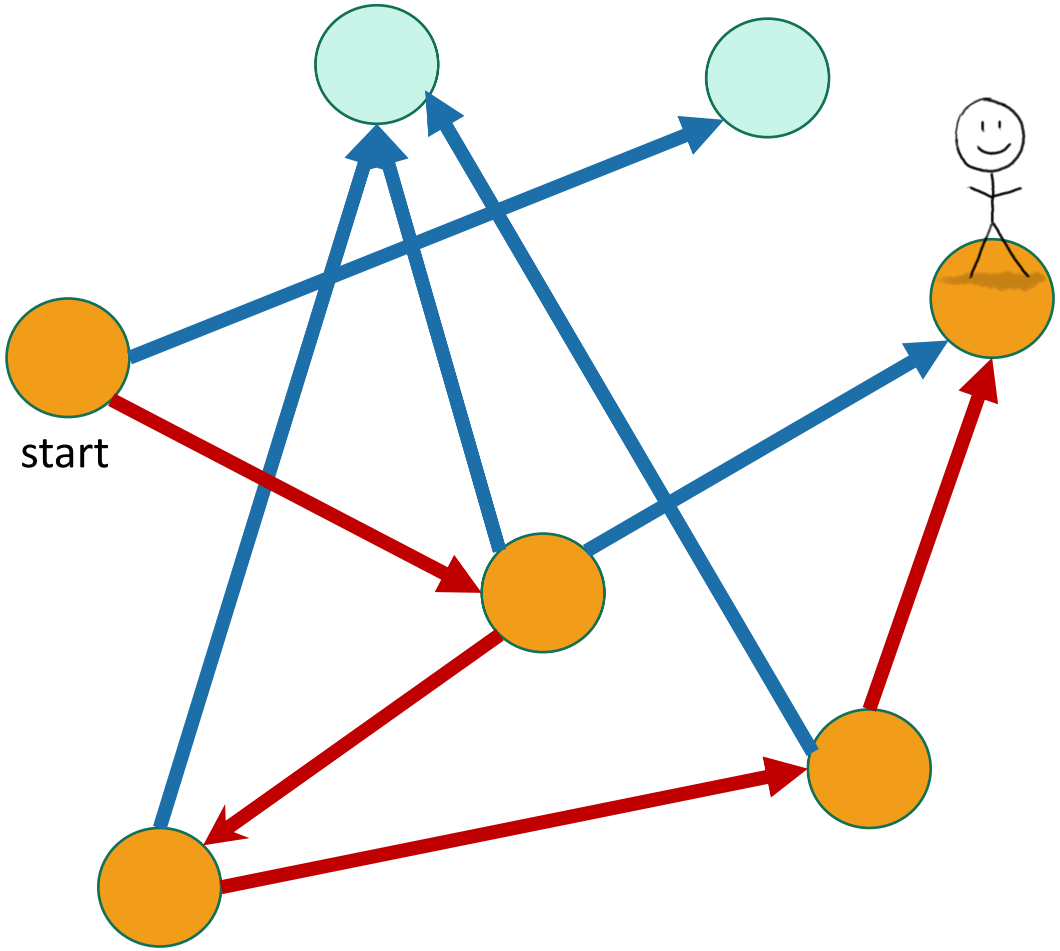
Depth First Search

Exploring a labyrinth with chalk and a piece of string



Depth First Search

Exploring a labyrinth with chalk and a piece of string



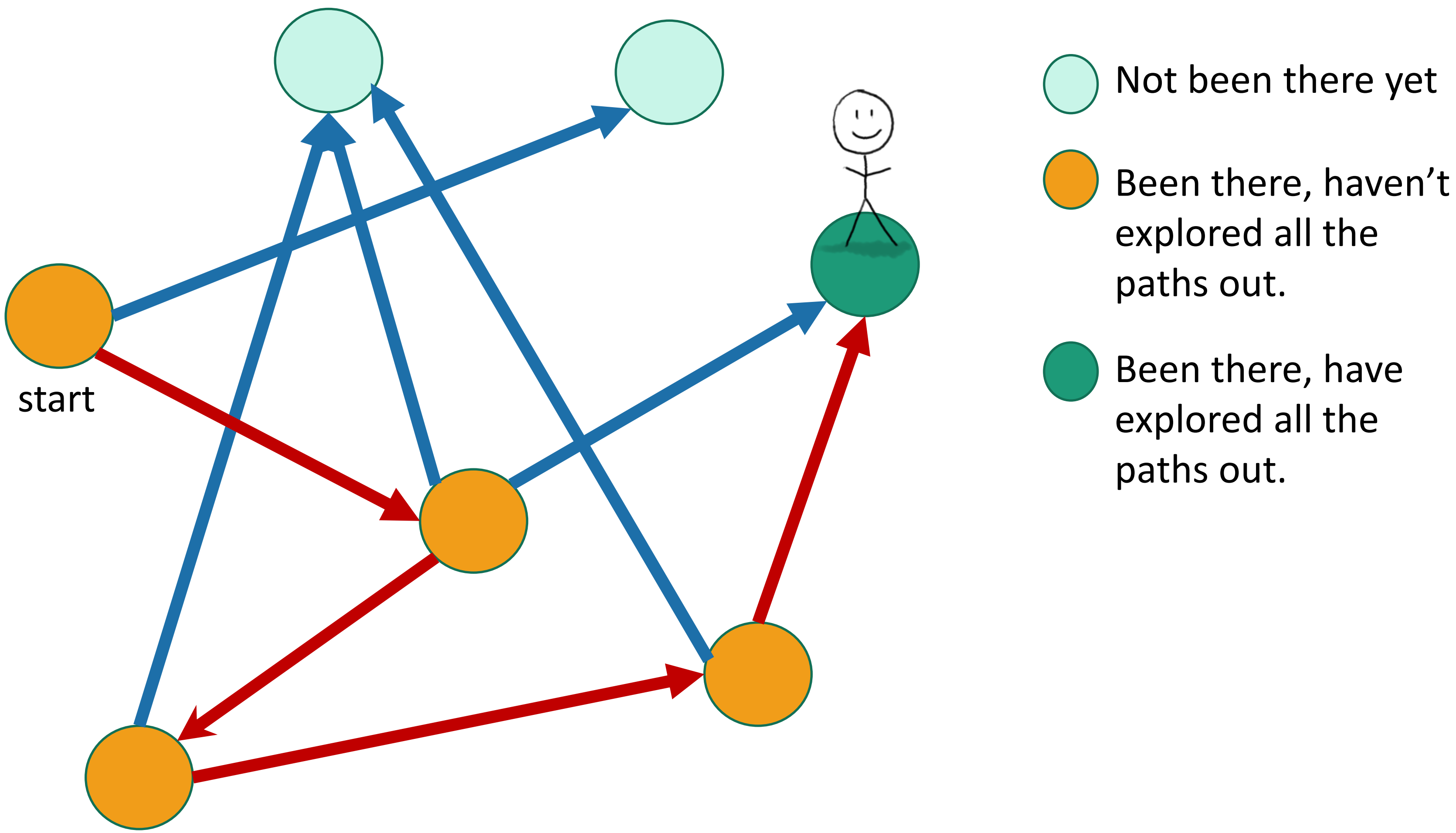
Not been there yet

Been there, haven't explored all the paths out.

Been there, have explored all the paths out.

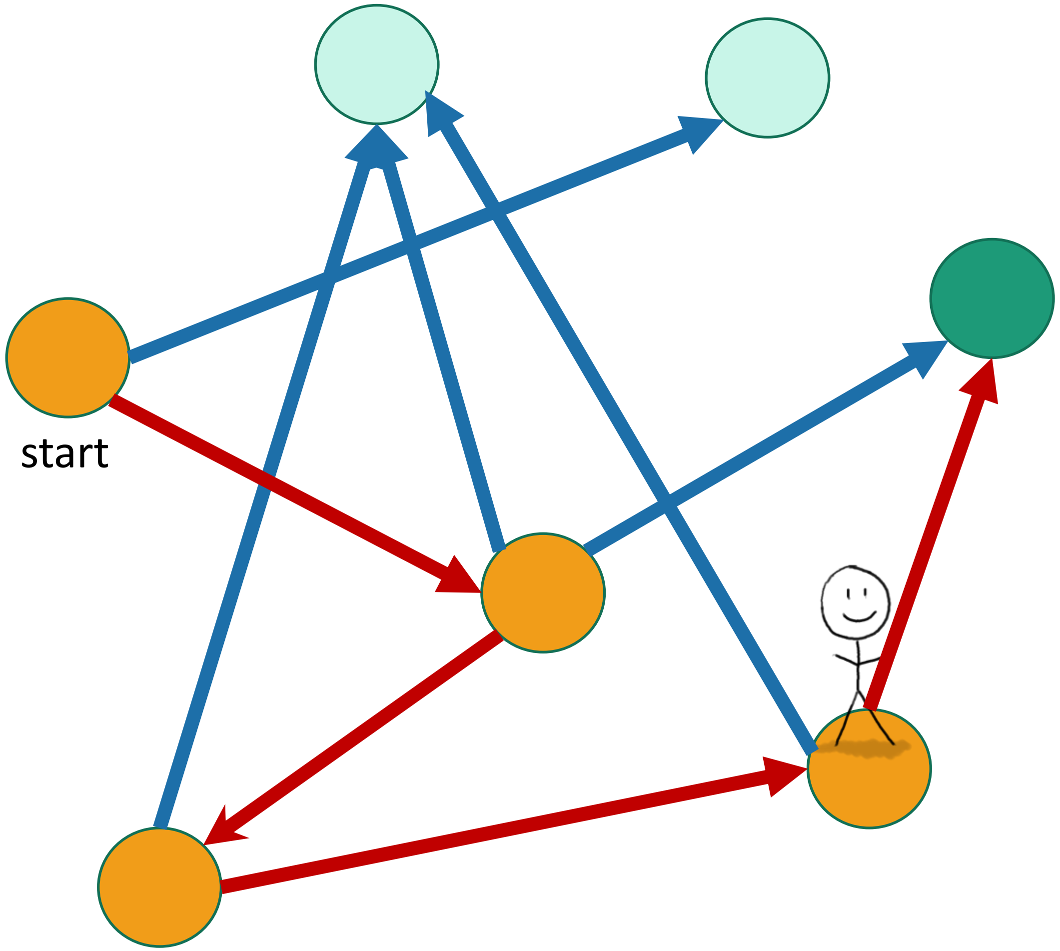
Depth First Search

Exploring a labyrinth with chalk and a piece of string



Depth First Search

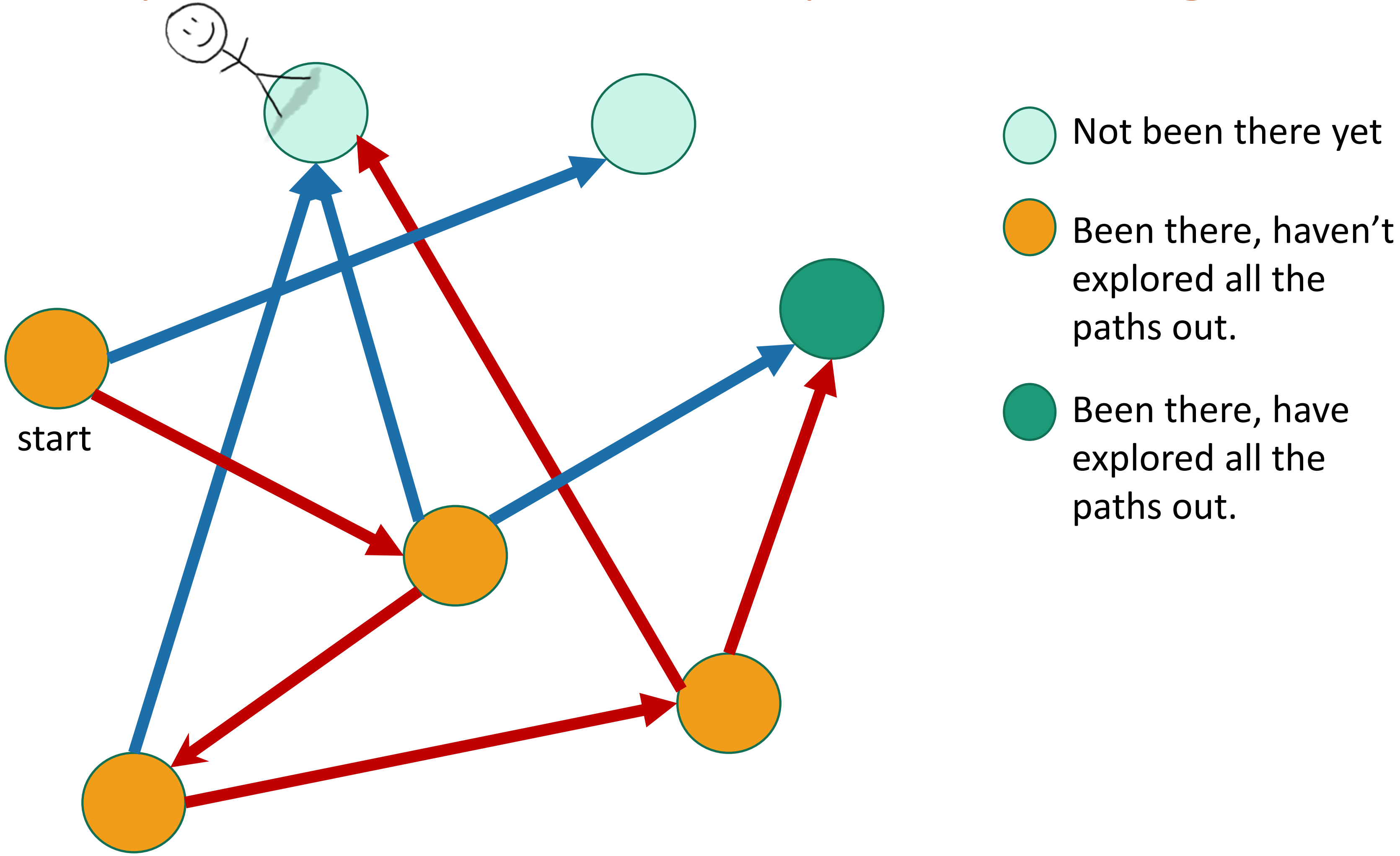
Exploring a labyrinth with chalk and a piece of string



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

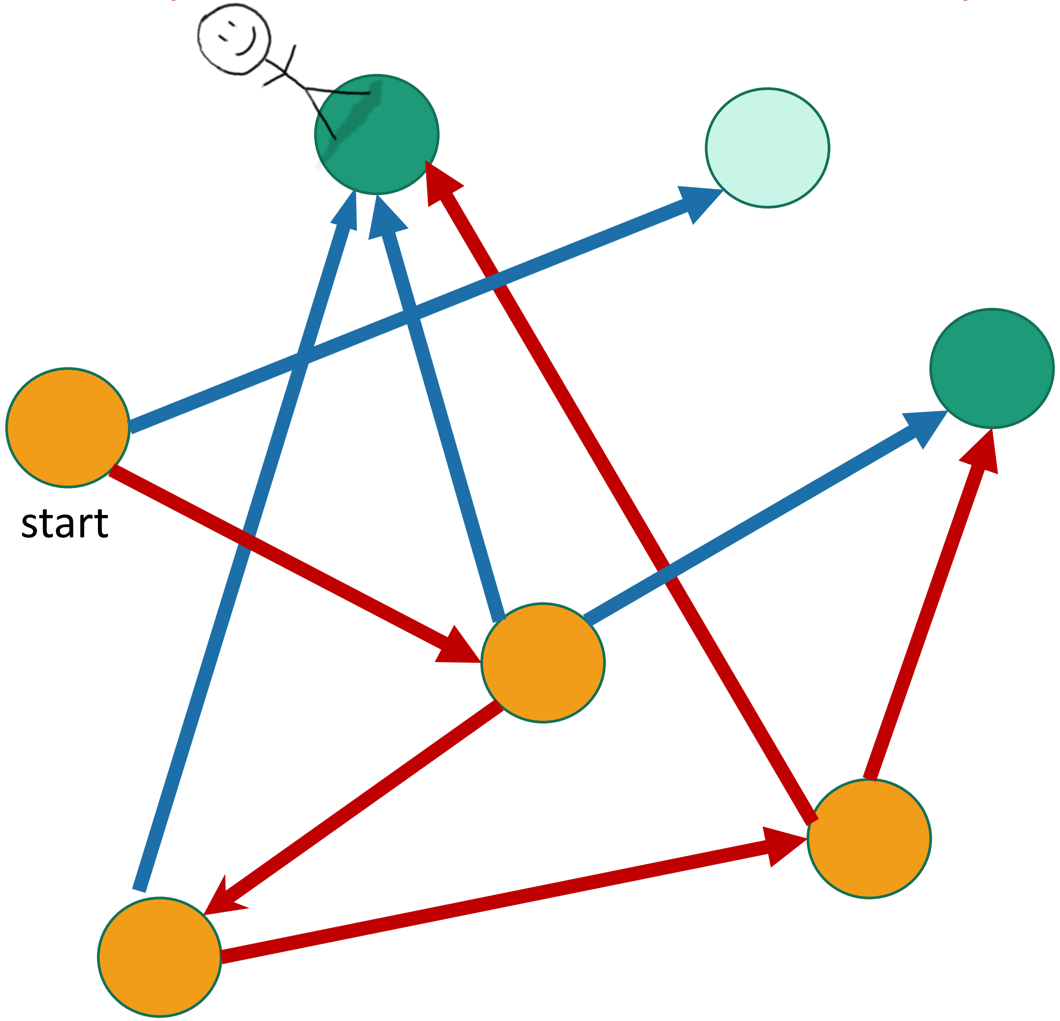
Depth First Search

Exploring a labyrinth with chalk and a piece of string



Depth First Search

Exploring a labyrinth with chalk and a piece of string



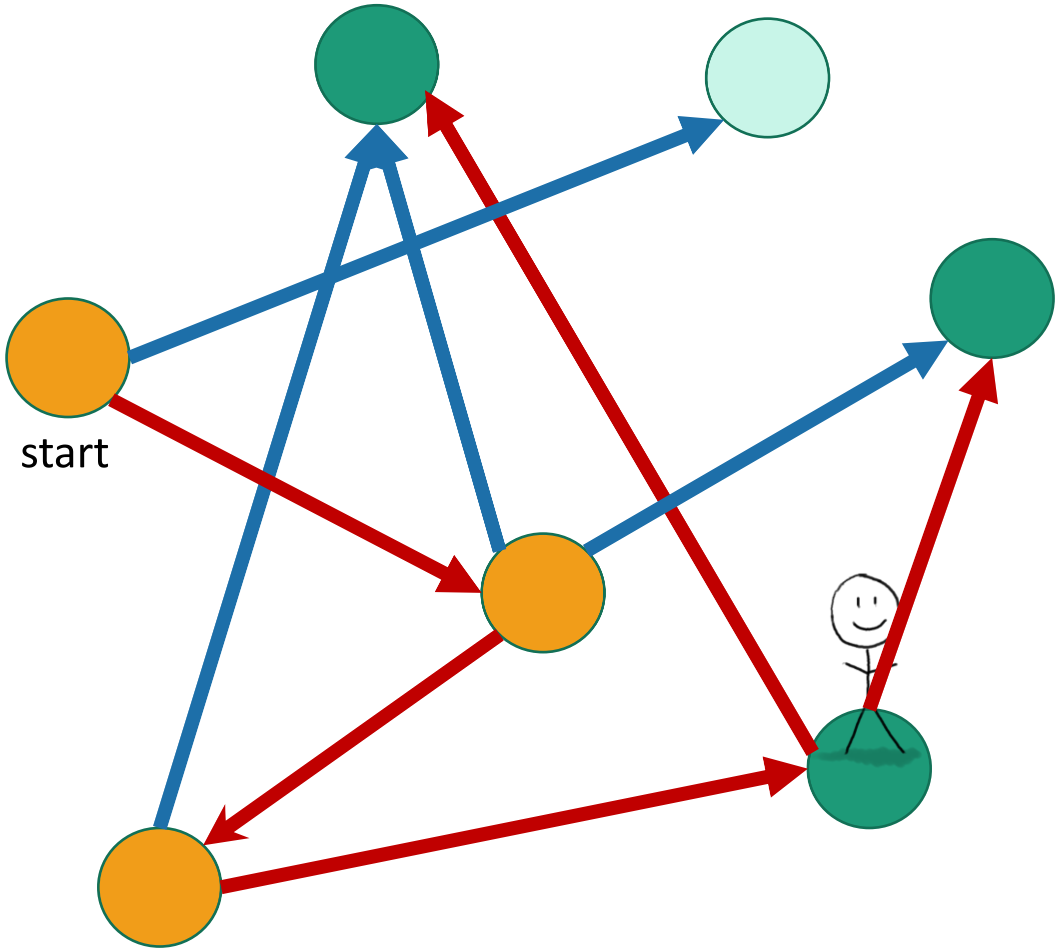
Not been there yet

Been there, haven't explored all the paths out.

Been there, have explored all the paths out.

Depth First Search

Exploring a labyrinth with chalk and a piece of string



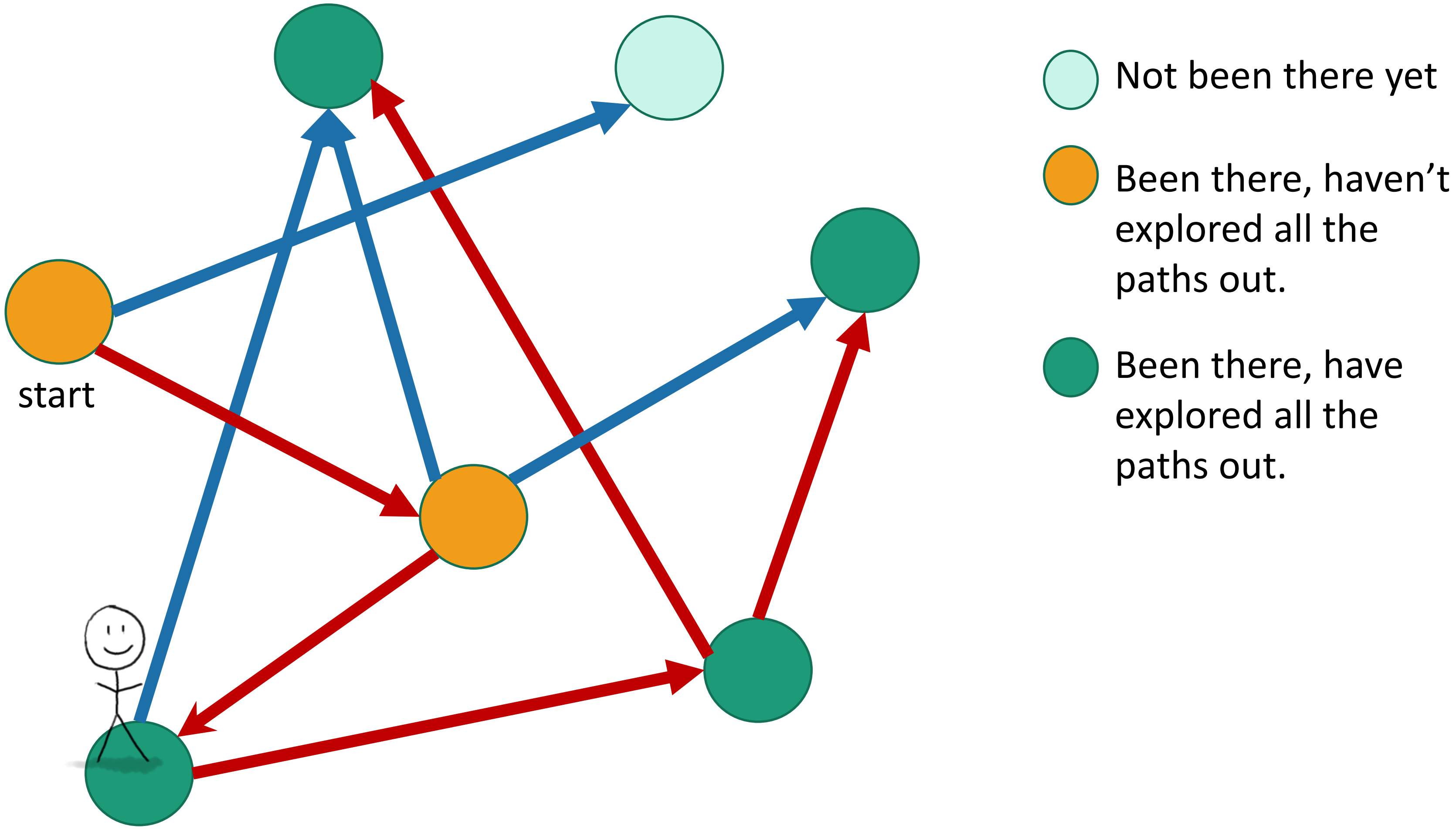
Not been there yet

Been there, haven't explored all the paths out.

Been there, have explored all the paths out.

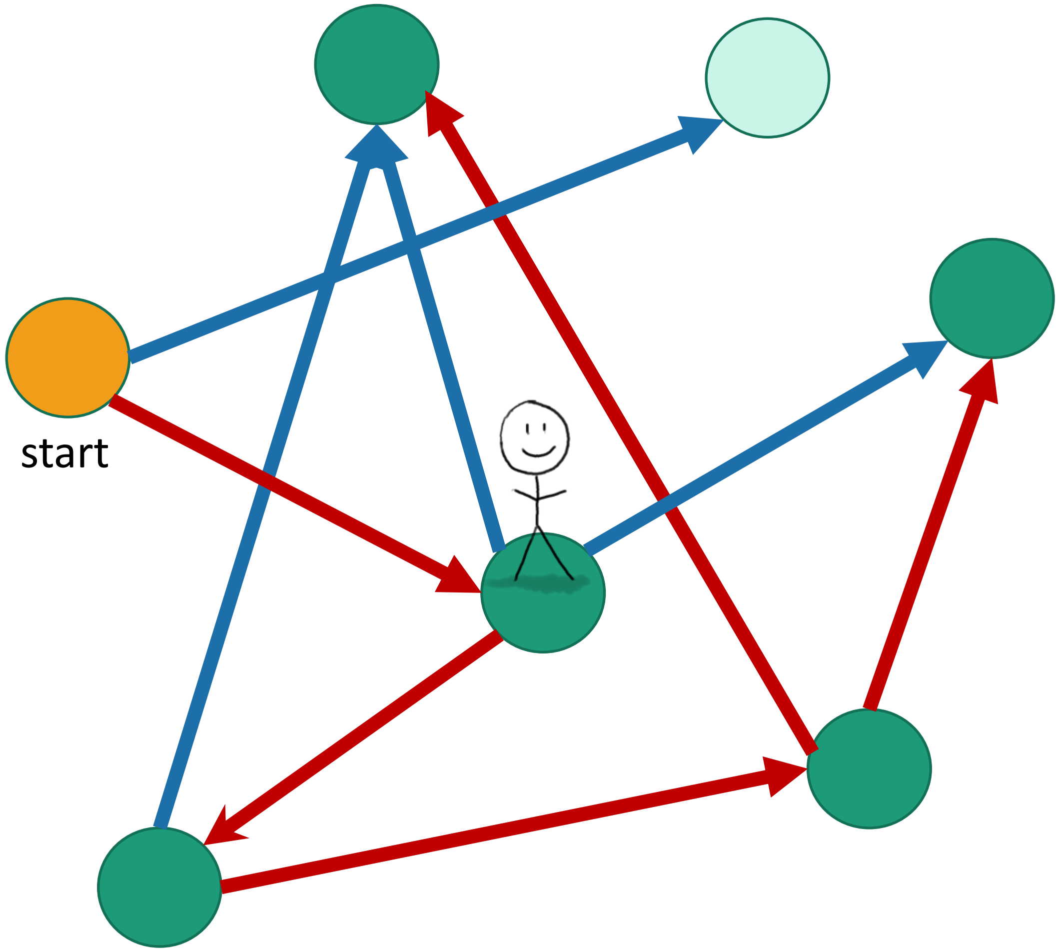
Depth First Search

Exploring a labyrinth with chalk and a piece of string



Depth First Search

Exploring a labyrinth with chalk and a piece of string



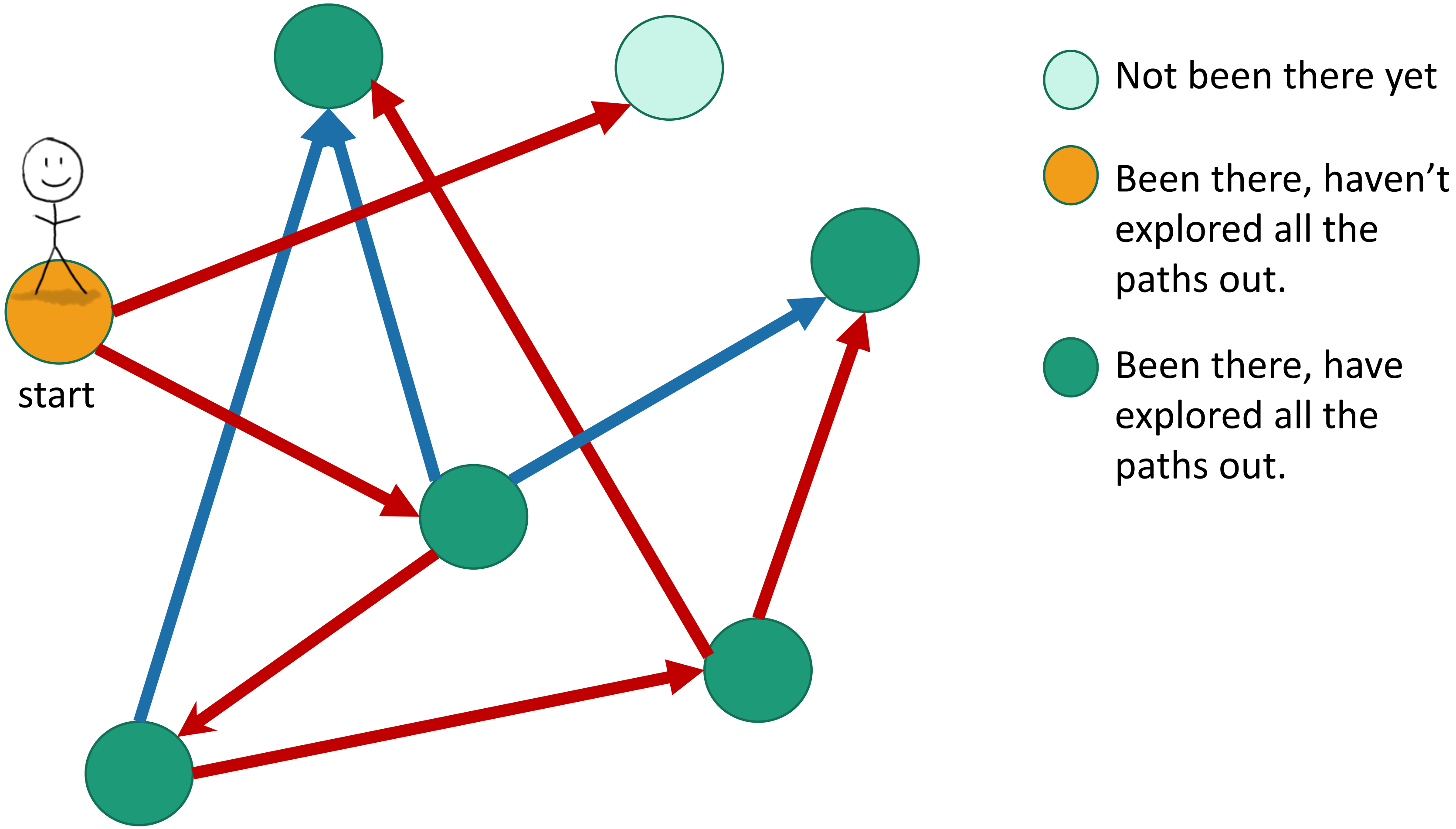
Not been there yet

Been there, haven't explored all the paths out.

Been there, have explored all the paths out.

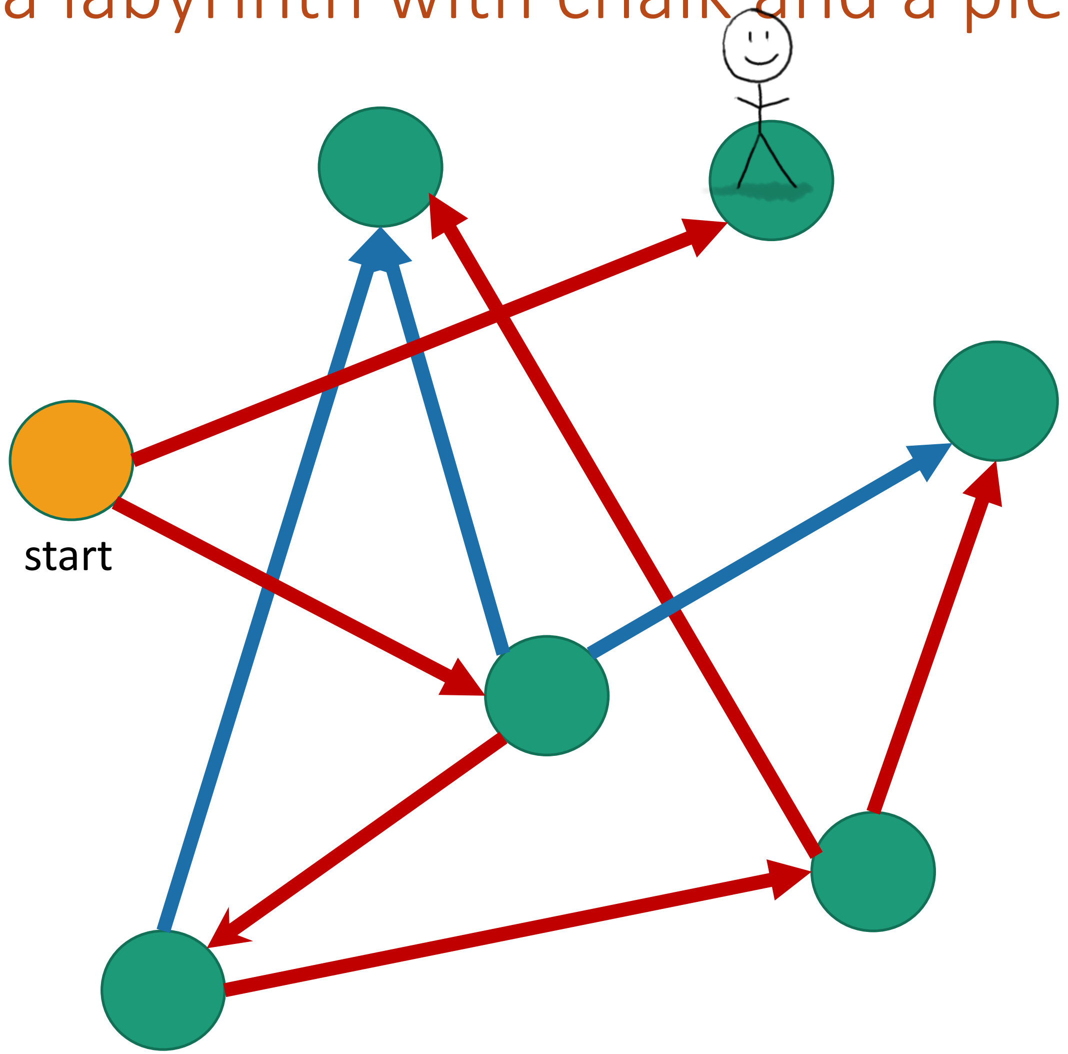
Depth First Search

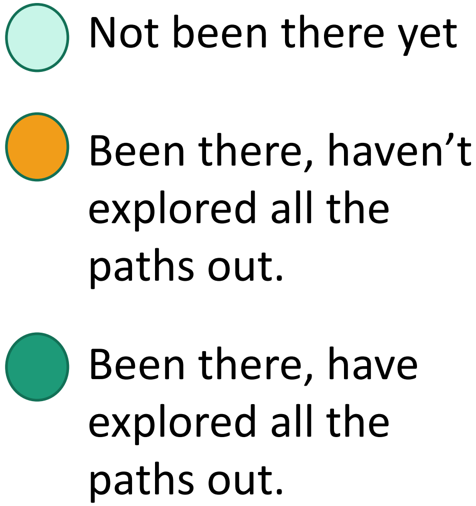
Exploring a labyrinth with chalk and a piece of string



Depth First Search

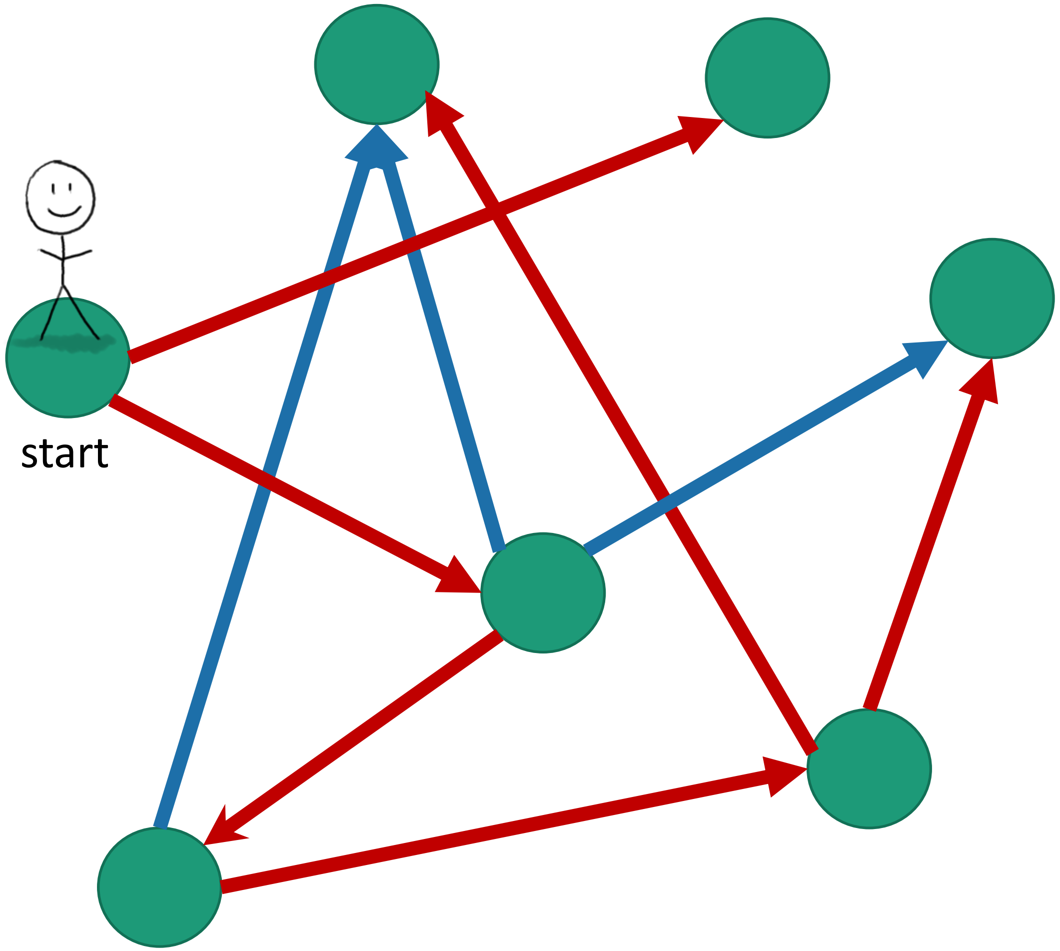
Exploring a labyrinth with chalk and a piece of string





Depth First Search

Exploring a labyrinth with chalk and a piece of string



Not been there yet

Been there, haven't explored all the paths out.

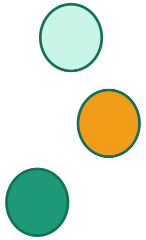
Been there, have explored all the paths out.

Labyrinth:
explored!

Depth First Search

Exploring a labyrinth with pseudocode

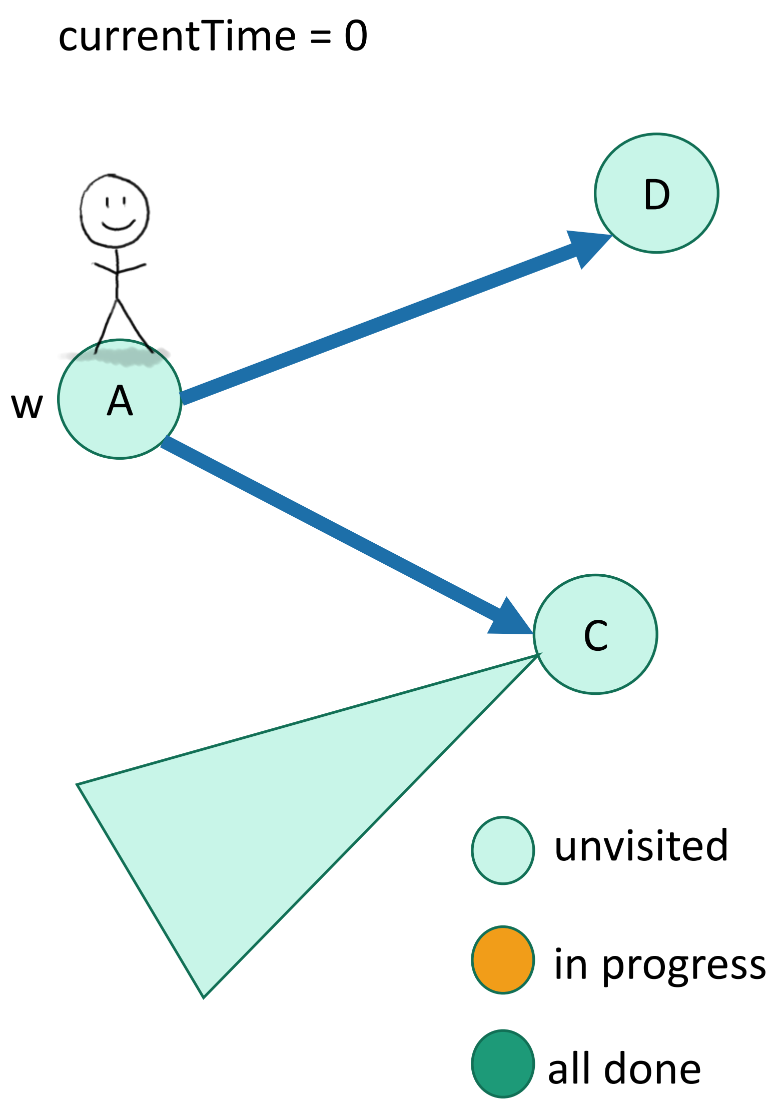
- Each vertex keeps track of whether it is:
  - Unvisited
  - In progress
  - All done



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

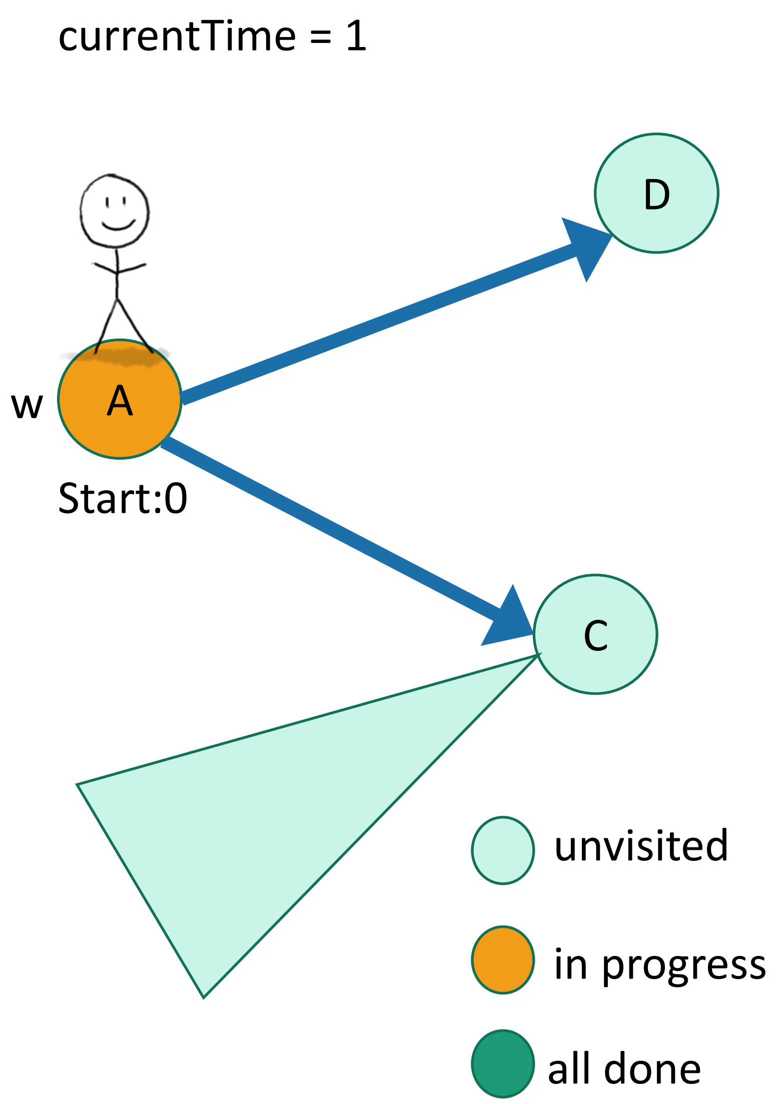
You might have seen other ways to implement DFS than what we are about to go through. This way has more bookkeeping, but more intuition – also, the bookkeeping will be useful later!

Depth First Search



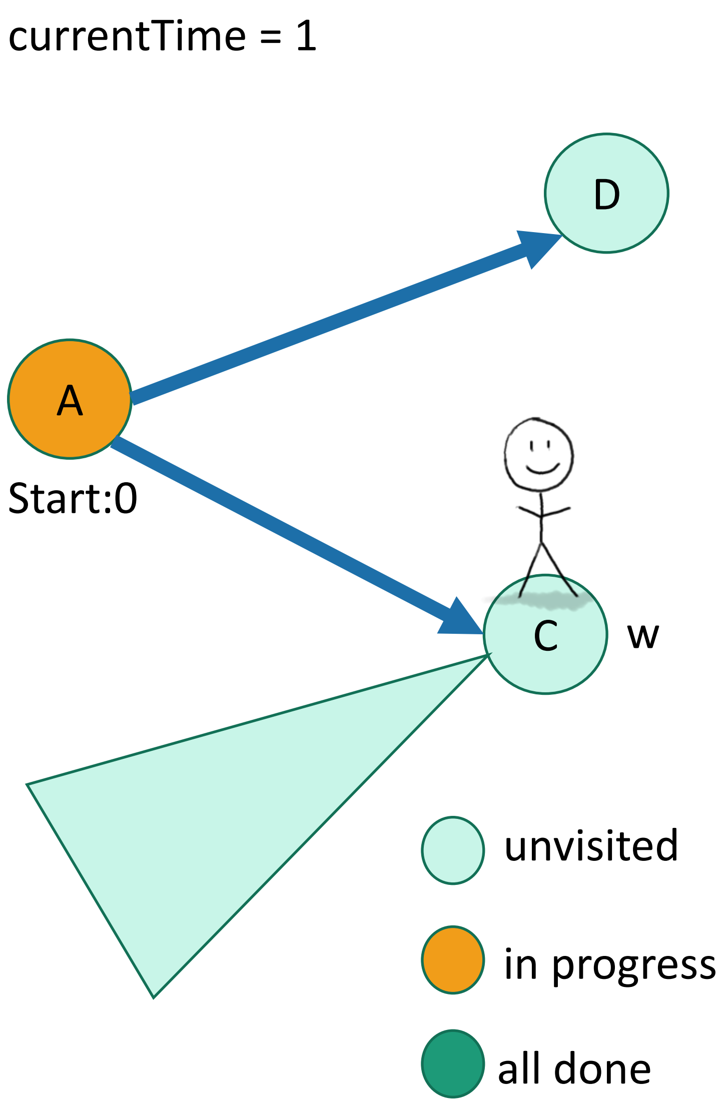
- **DFS**(w, currentTime):
  - w.entryTime = currentTime
  - currentTime ++
  - Mark w as in progress.
  - **for** v in w.neighbors:
    - **if** v is unvisited:
      - currentTime
        = **DFS**(v, currentTime)
      - currentTime ++
  - w.finishTime = currentTime
  - Mark w as **all done**
  -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

Depth First Search



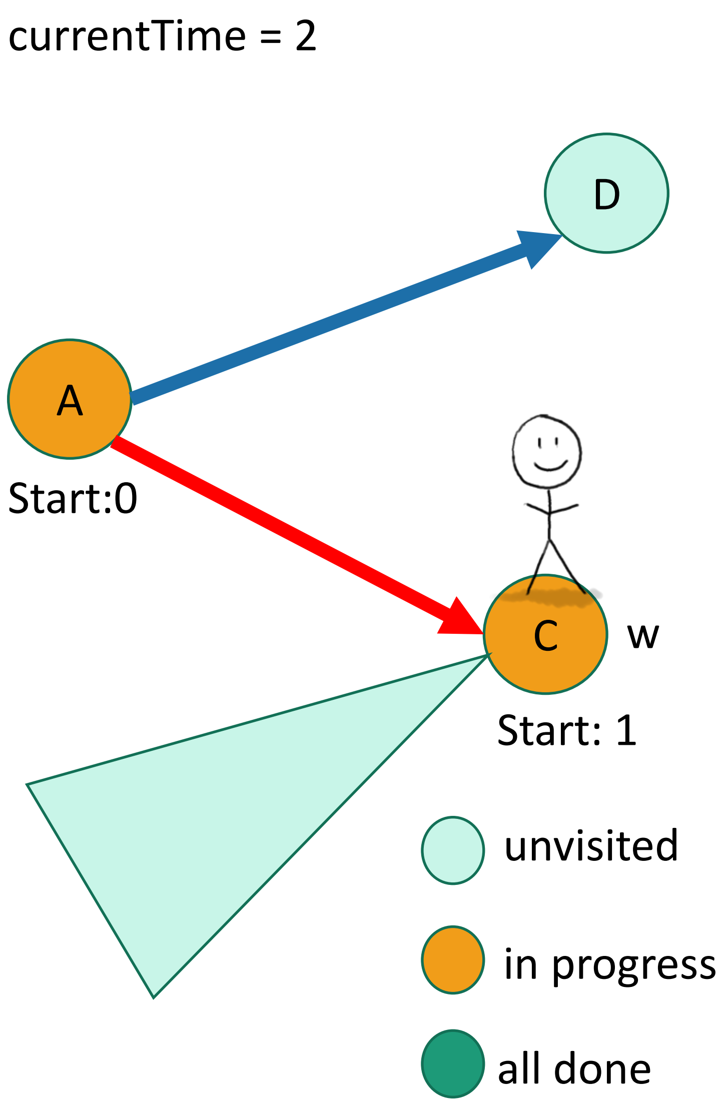
- **DFS**(w, currentTime):
  - w.entryTime = currentTime
  - currentTime ++
  - Mark w as in progress.
  - **for** v in w.neighbors:
    - **if** v is unvisited:
      - currentTime
        = **DFS**(v, currentTime)
      - currentTime ++
  - w.finishTime = currentTime
  - Mark w as **all done**
  -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

Depth First Search



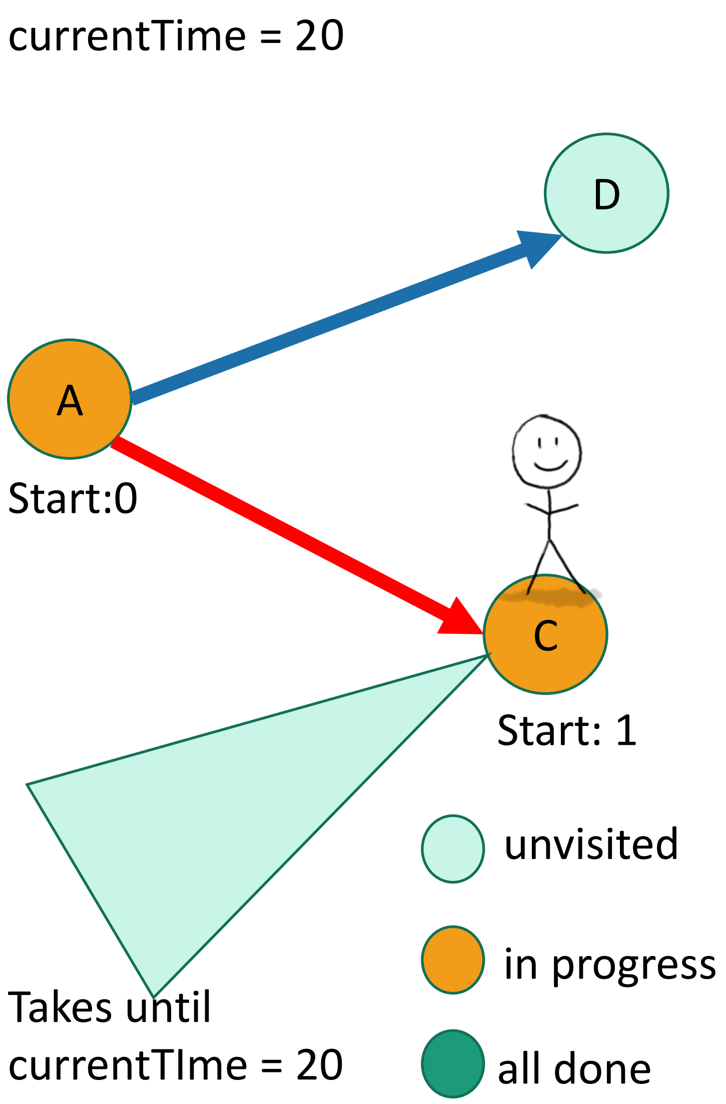
- **DFS**(w, currentTime):
  - w.entryTime = currentTime
  - currentTime ++
  - Mark w as in progress.
  - **for** v in w.neighbors:
    - **if** v is unvisited:
      - currentTime = **DFS**(v, currentTime)
      - currentTime ++
  - w.finishTime = currentTime
  - Mark w as **all done**
  - **return**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

Depth First Search



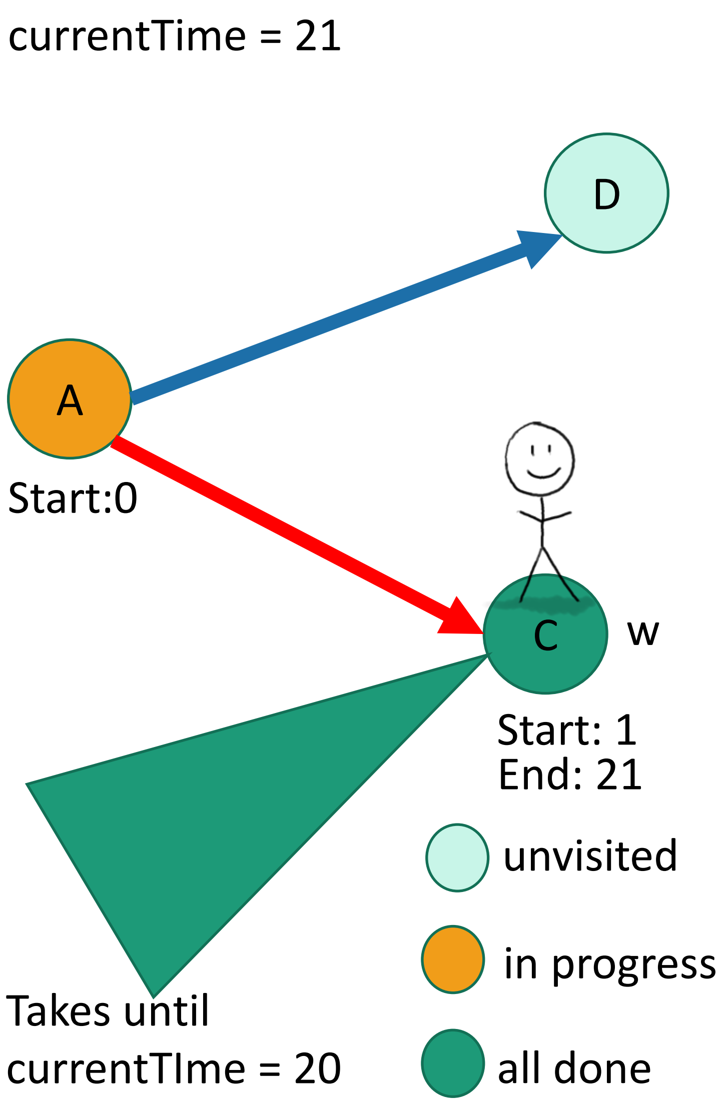
- **DFS**(w, currentTime):
  - w.entryTime = currentTime
  - currentTime ++
  - Mark w as **in progress**.
  - **for** v in w.neighbors:
    - **if** v is **unvisited**:
      - currentTime = **DFS**(v, currentTime)
      - currentTime ++
  - w.finishTime = currentTime
  - Mark w as **all done**
  -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

Depth First Search



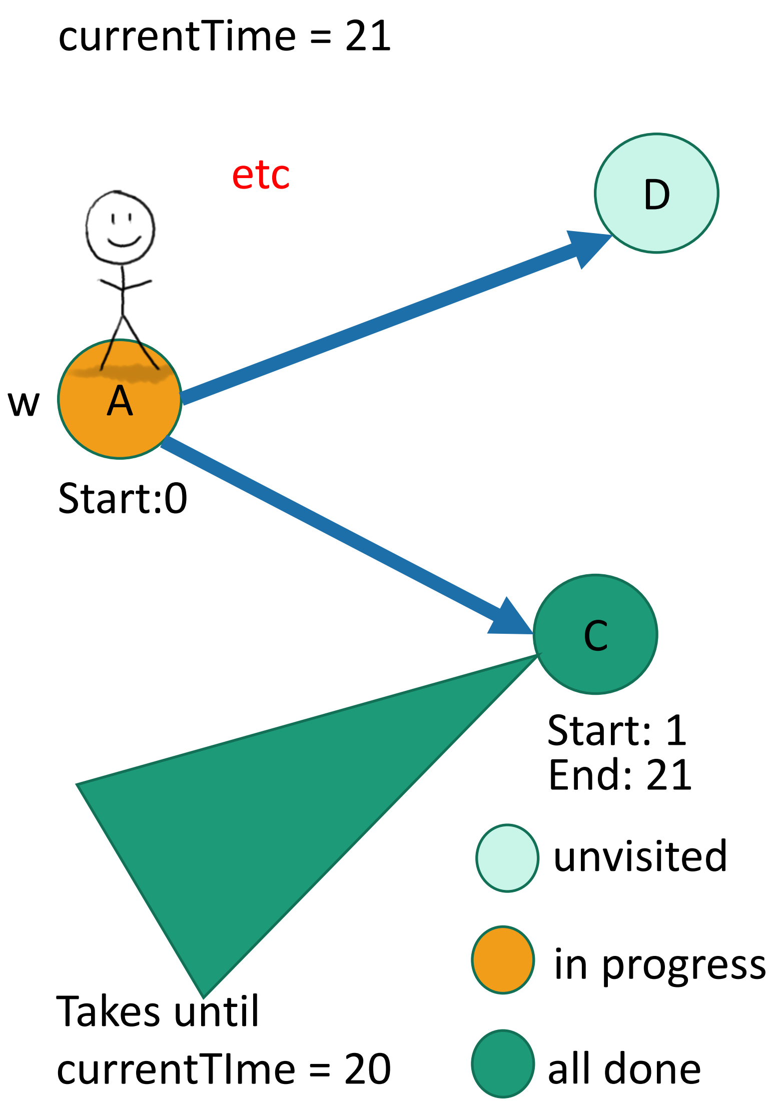
- **DFS**(w, currentTime):
  - w.entryTime = currentTime
  - currentTime ++
  - Mark w as in progress.
  - **for** v in w.neighbors:
    - **if** v is unvisited:
      - currentTime
        = **DFS**(v, currentTime)
      - currentTime ++
  - w.finishTime = currentTime
  - Mark w as **all done**
  -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

Depth First Search



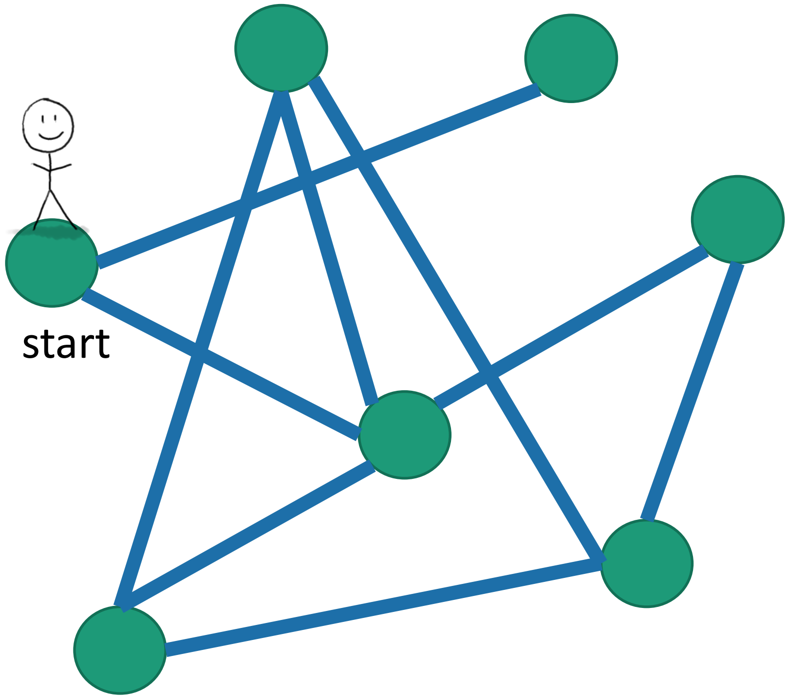
- **DFS**(w, currentTime):
  - w.startTime = currentTime
  - currentTime ++
  - Mark w as **in progress**.
  - **for** v in w.neighbors:
    - **if** v is **unvisited**:
      - currentTime
        = **DFS**(v, currentTime)
      - currentTime ++
  - w.finishTime = currentTime
  - Mark w as **all done**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

Depth First Search

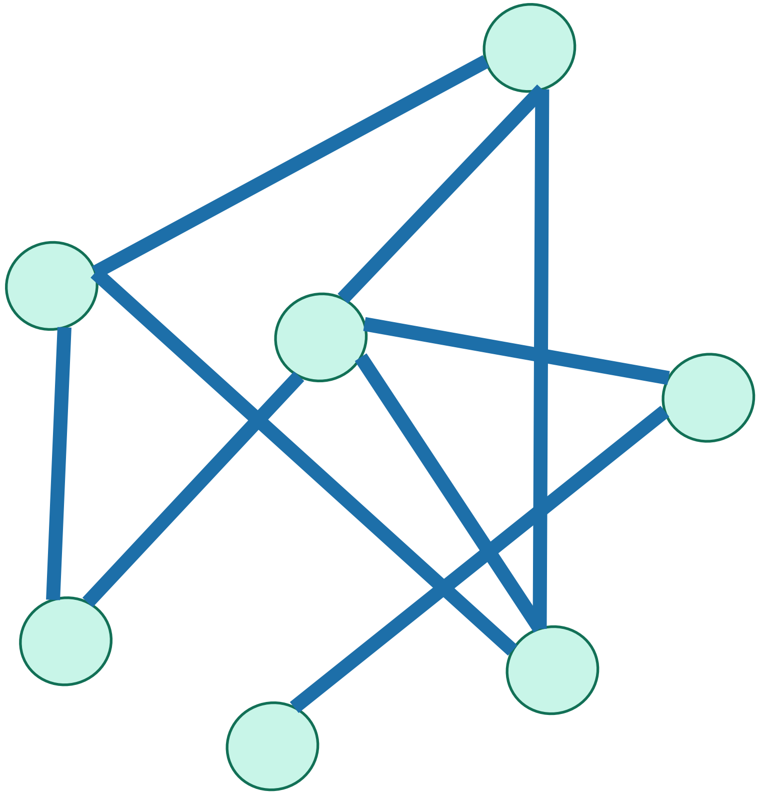


- **DFS**(w, currentTime):
  - w.startTime = currentTime
  - currentTime ++
  - Mark w as **in progress**.
  - **for** v in w.neighbors:
    - **if** v is **unvisited**:
      - currentTime
        = **DFS**(v, currentTime)
      - currentTime ++
  - w.finishTime = currentTime
  - Mark w as **all done**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

DFS finds all the nodes reachable from the starting point



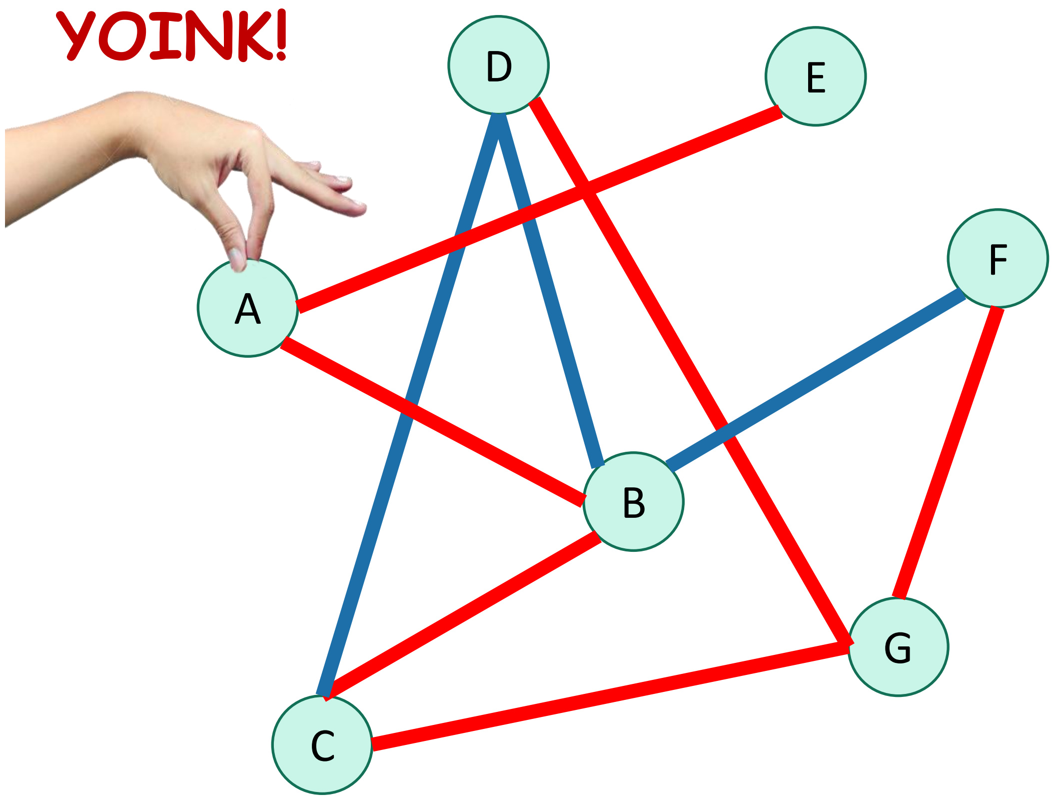
In an undirected graph, this is called a **connected component.**

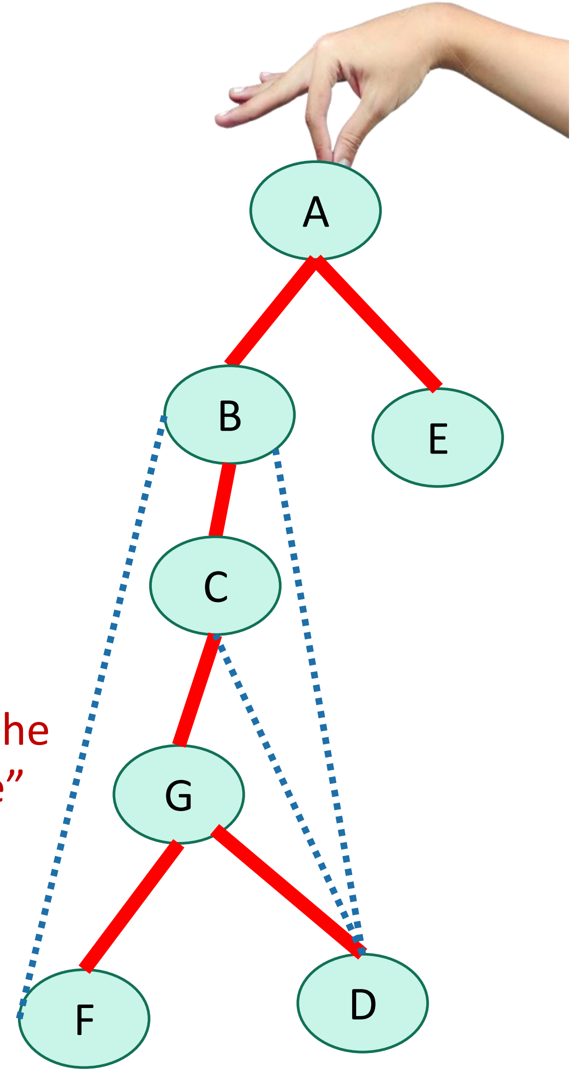


One application: finding connected components.

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)

Verify this formally!

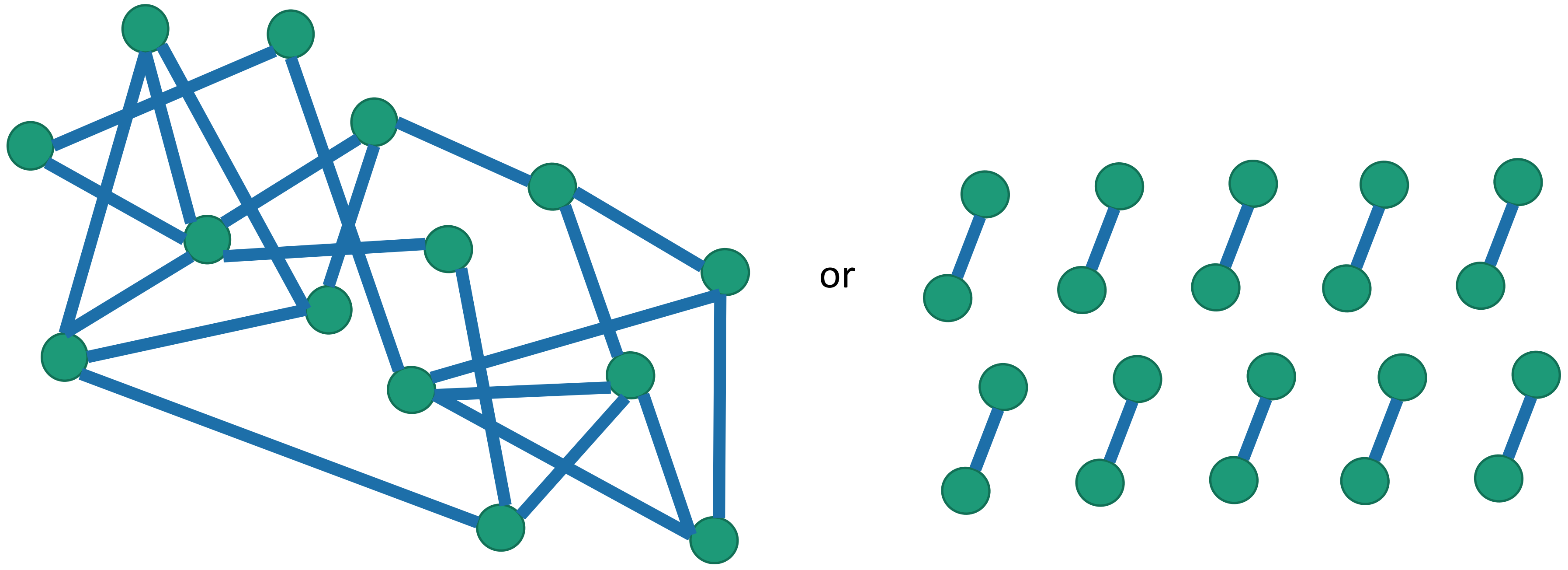
Ollie the over-achieving ostrich

Running time

To explore the whole thing

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

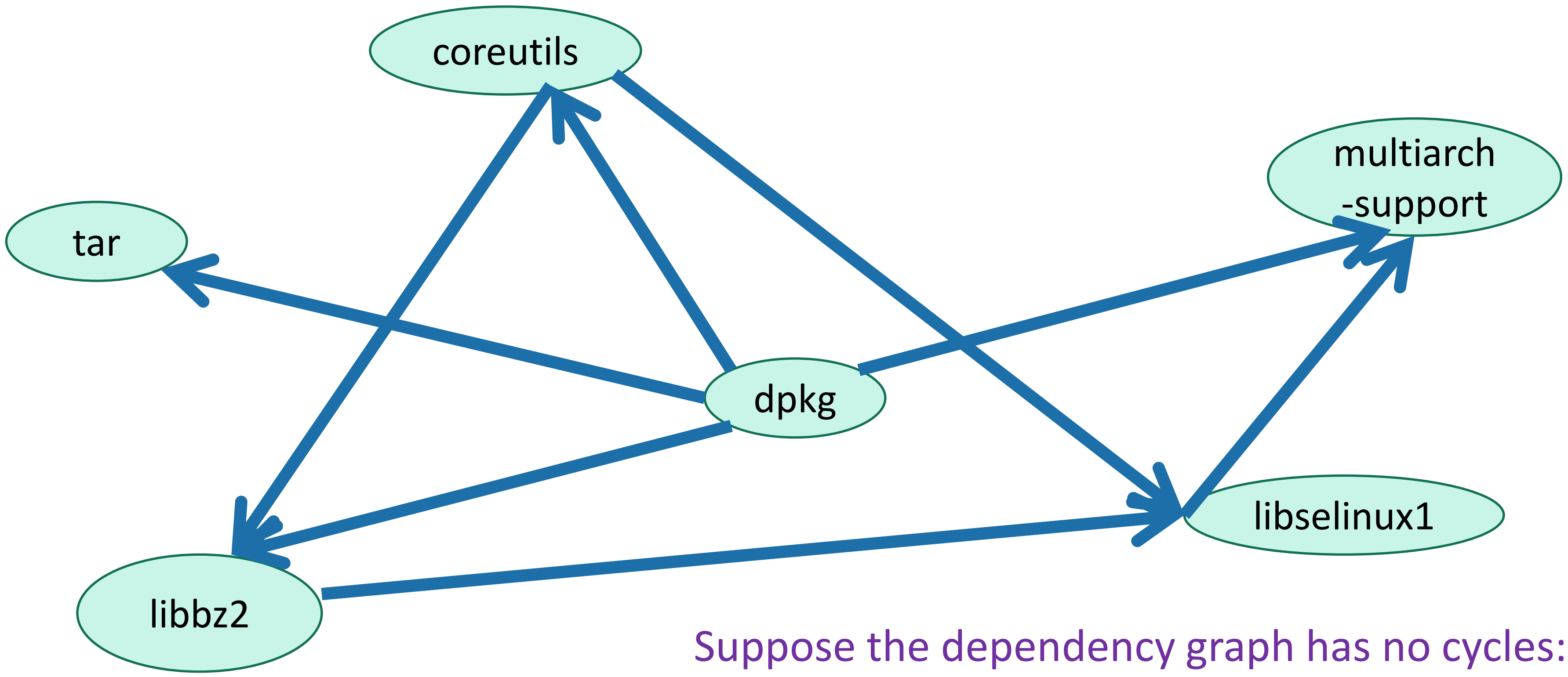
$$O(n + m)$$



Applications of DFS

topological sorting: Application of DFS

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

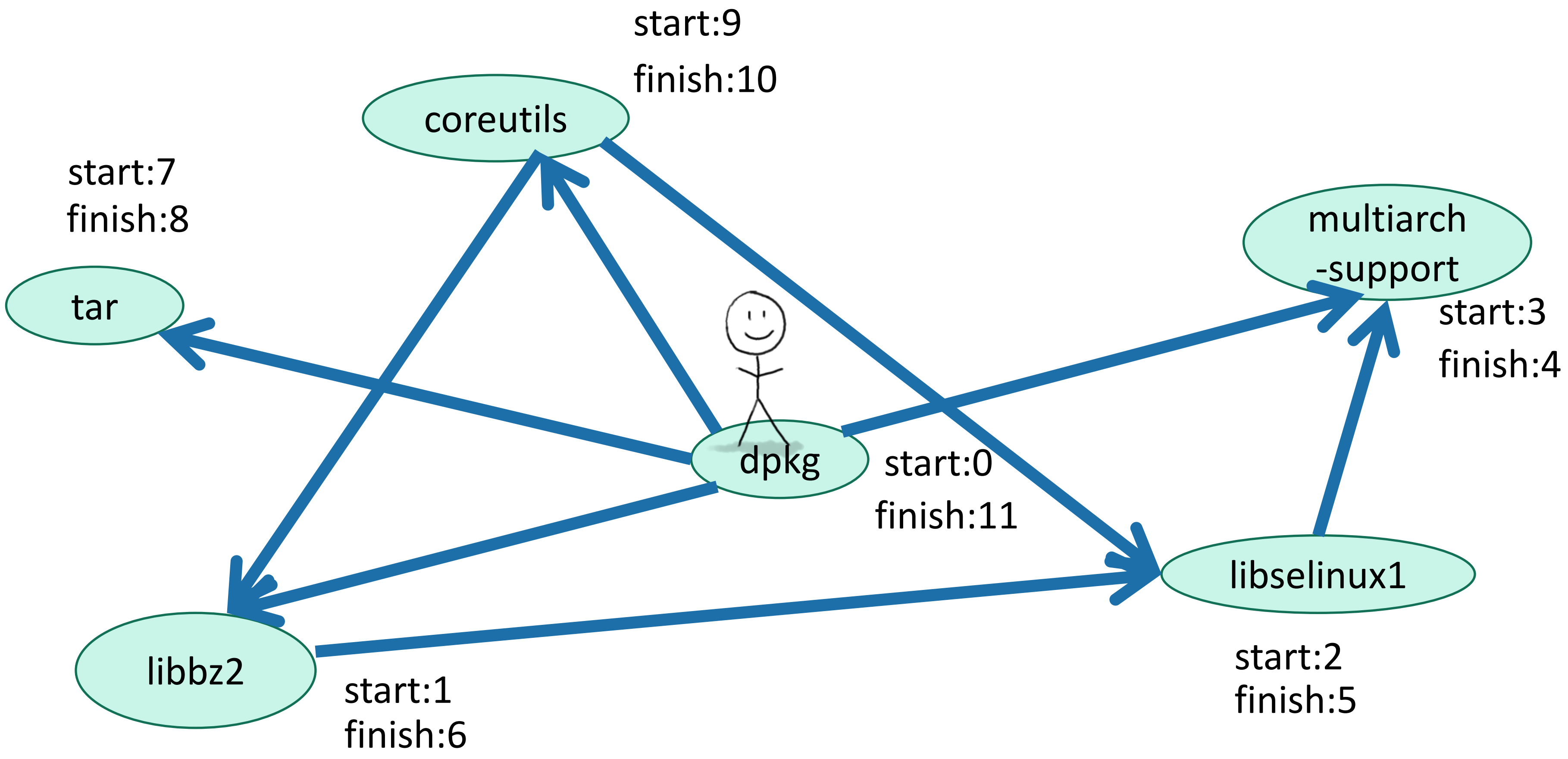


Suppose the dependency graph has no cycles:
it is a **Directed Acyclic Graph (DAG)**

Let’s do DFS

Observations:

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



Suppose the underlying graph has no cycles

This is not an accident

Claim: In general, we'll always have:

finish: [larger]

finish: [smaller]

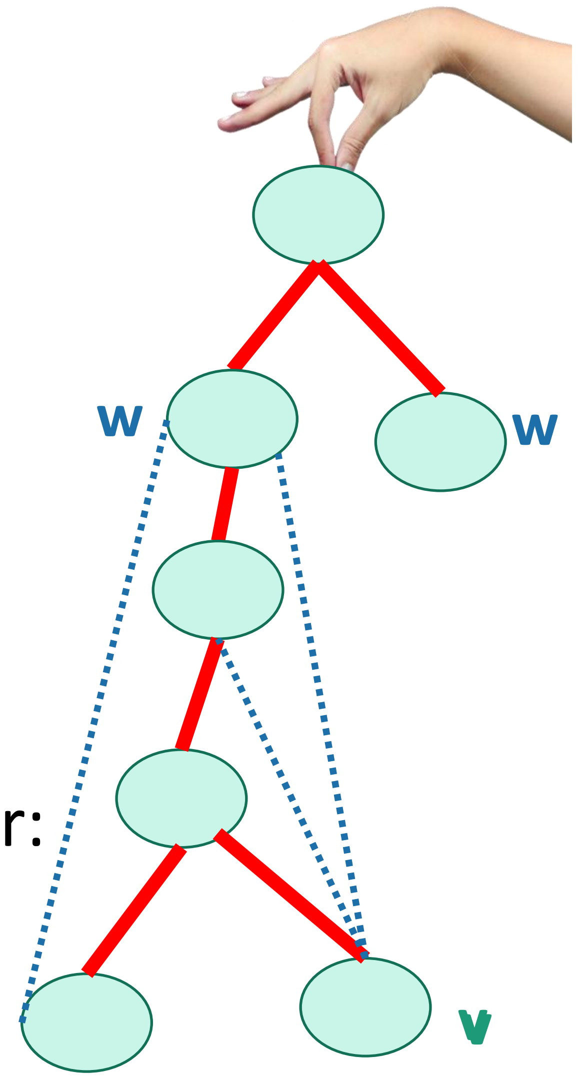
To understand why, let's go back to that DFS tree.

A more general statement

(this holds even if there are cycles)

This is called the “parentheses theorem” in CLRS

(check this statement carefully!)



- 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:

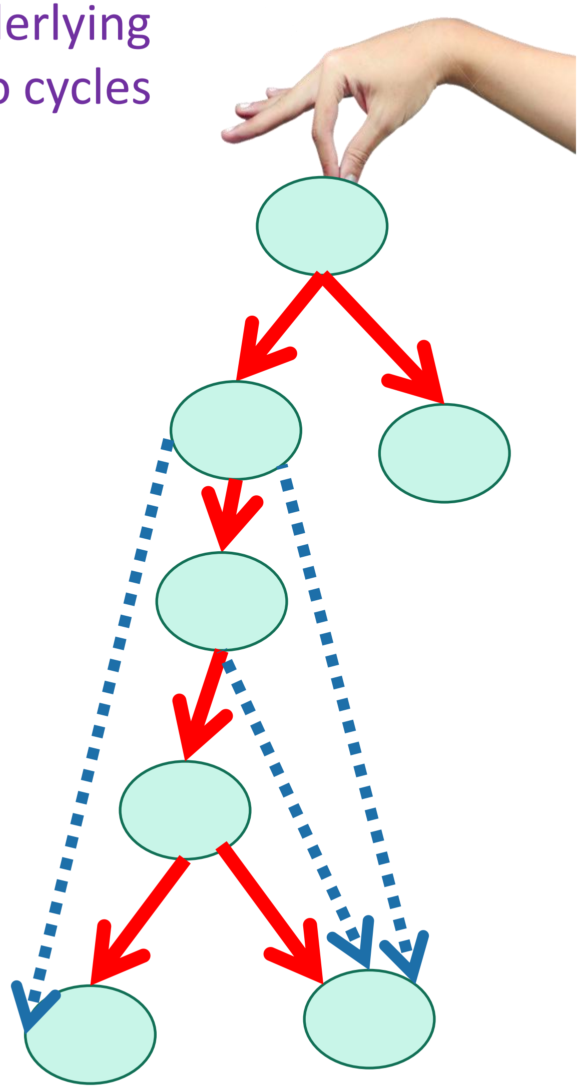
(or the other way around)

So to prove this ->

If A → B

Then B.finishTime < A.finishTime

Suppose the underlying graph has no cycles



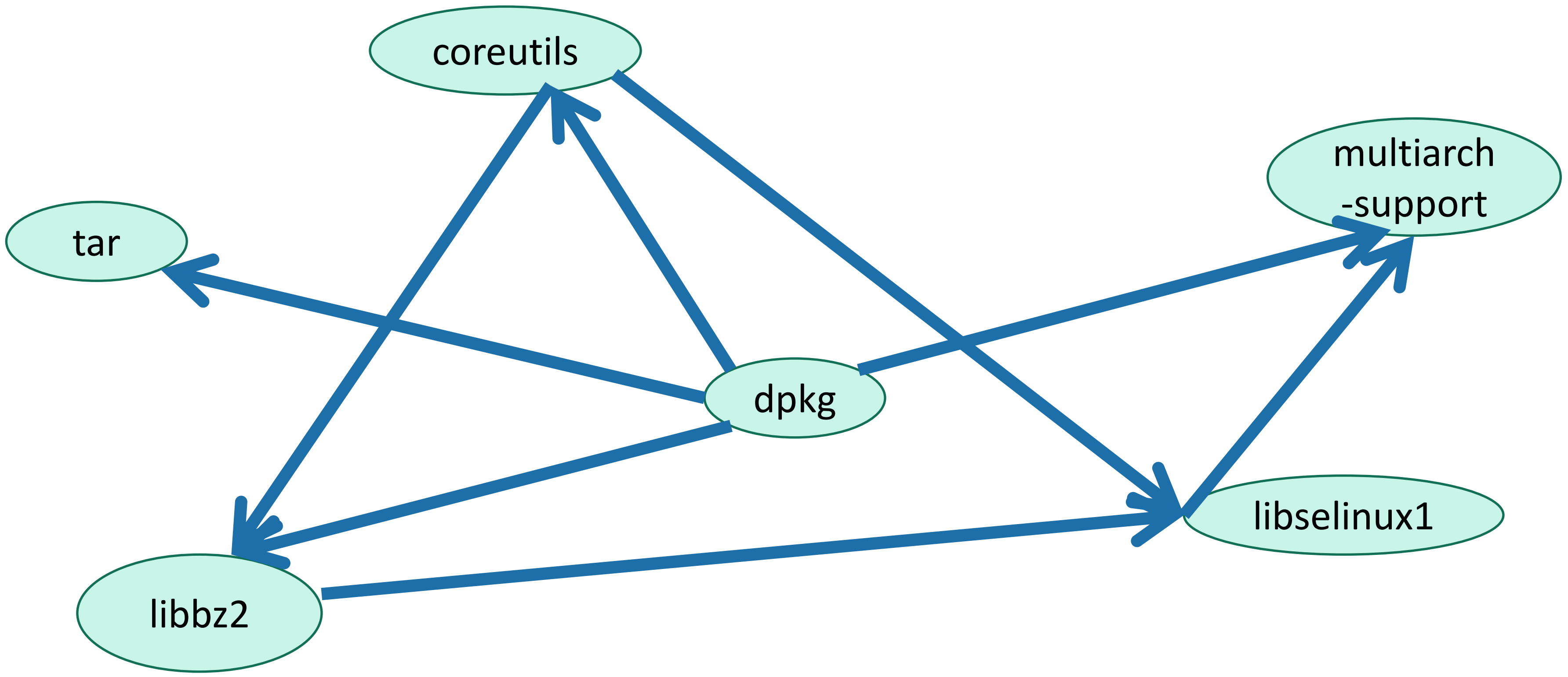
- Since the graph has no cycles, B must be a descendent of A in that tree.
  - All edges go down the tree.

- Then

- aka, B.finishTime < A.finishTime.

Back to this problem

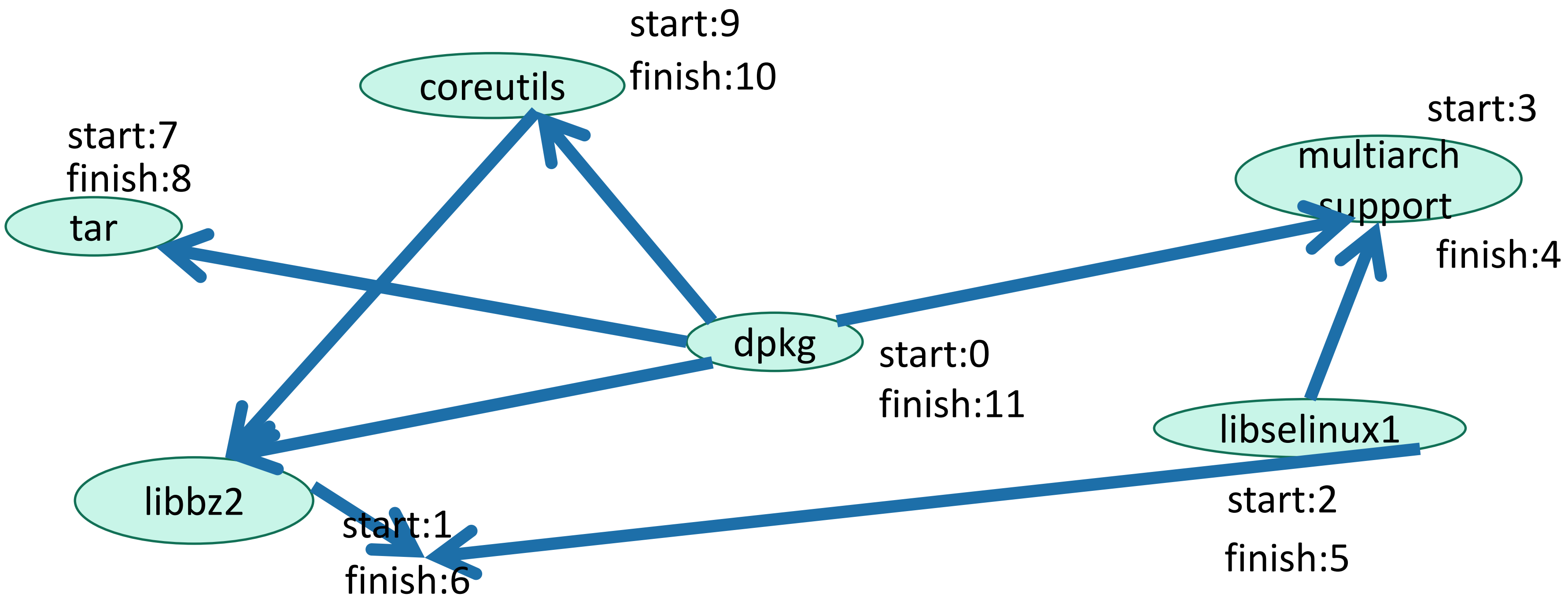
- Example: package dependency graph
- Question: in what order should I install packages?



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`

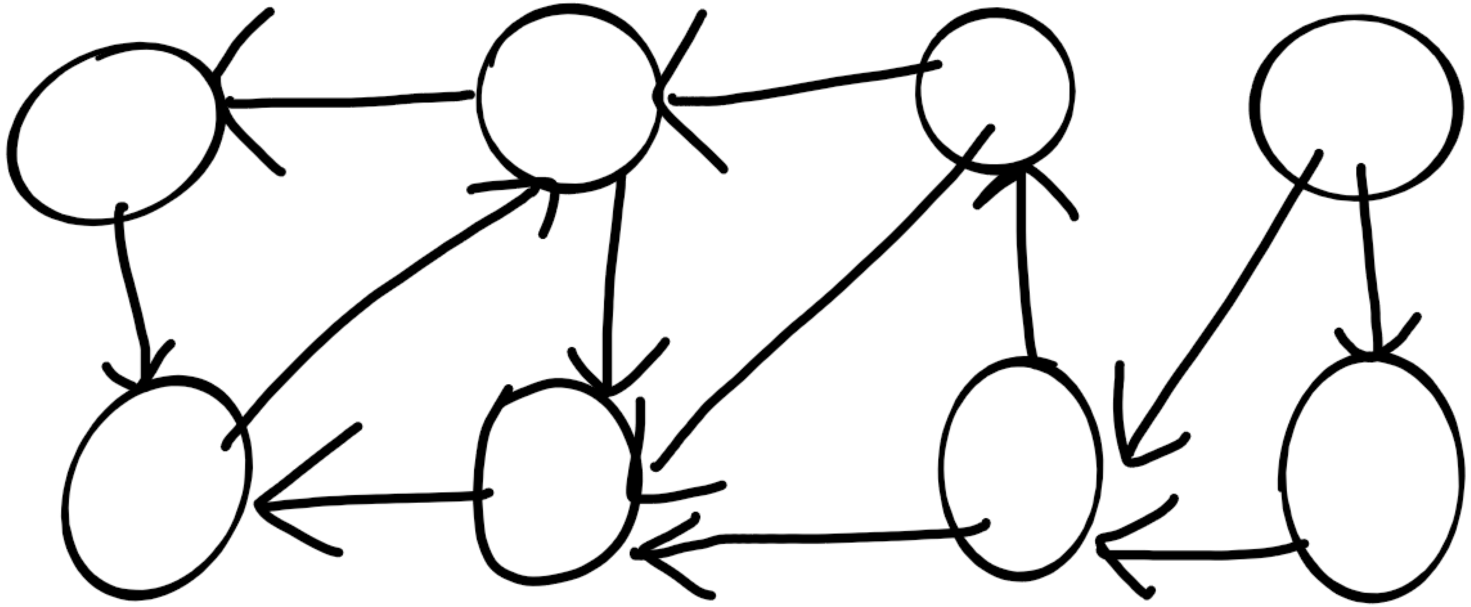


What did we just learn?

- 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.

Classification of Edges

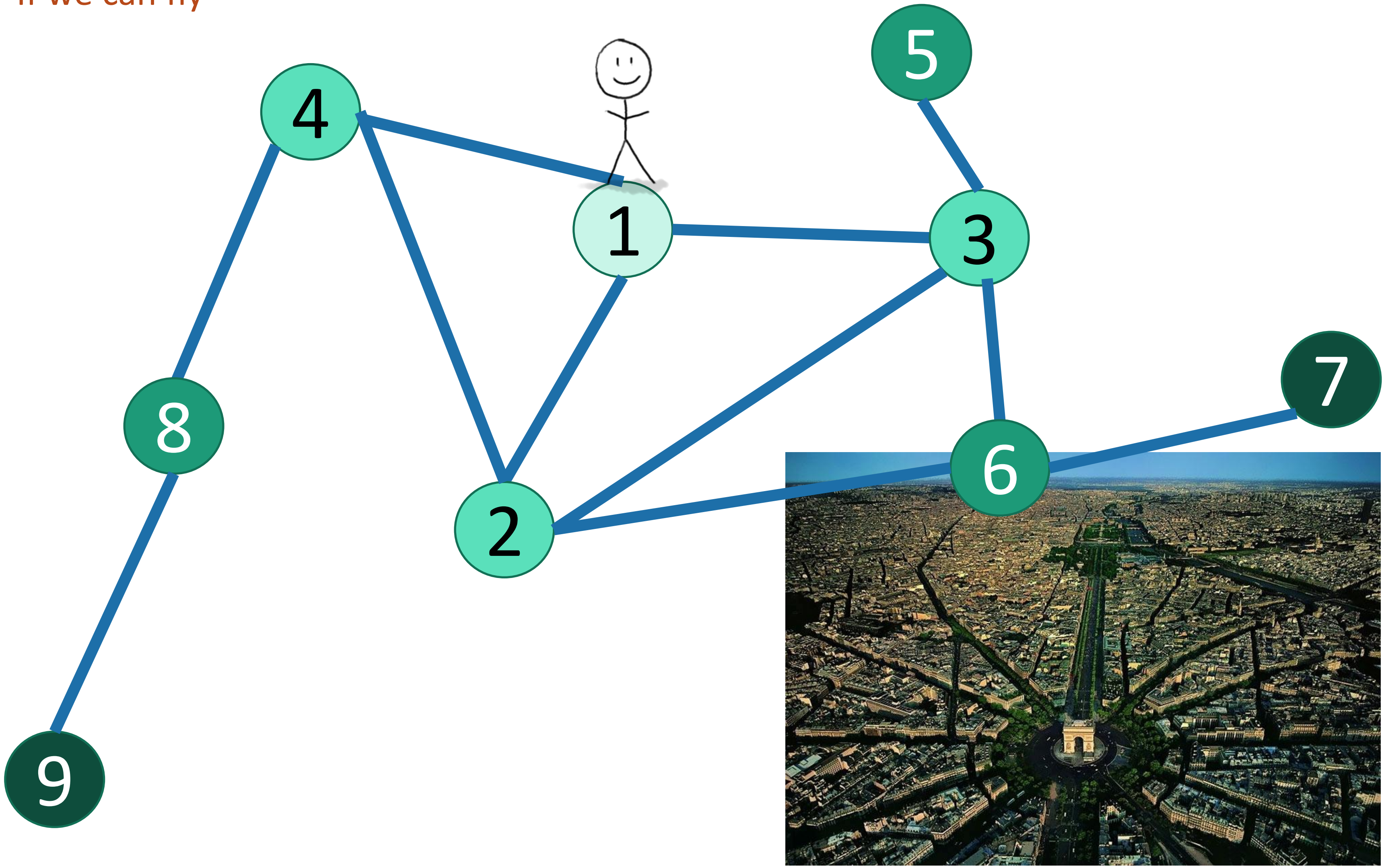
- Tree Edge
- Back Edge
- Forward Edge
- Cross Edge



Breadth-First Search

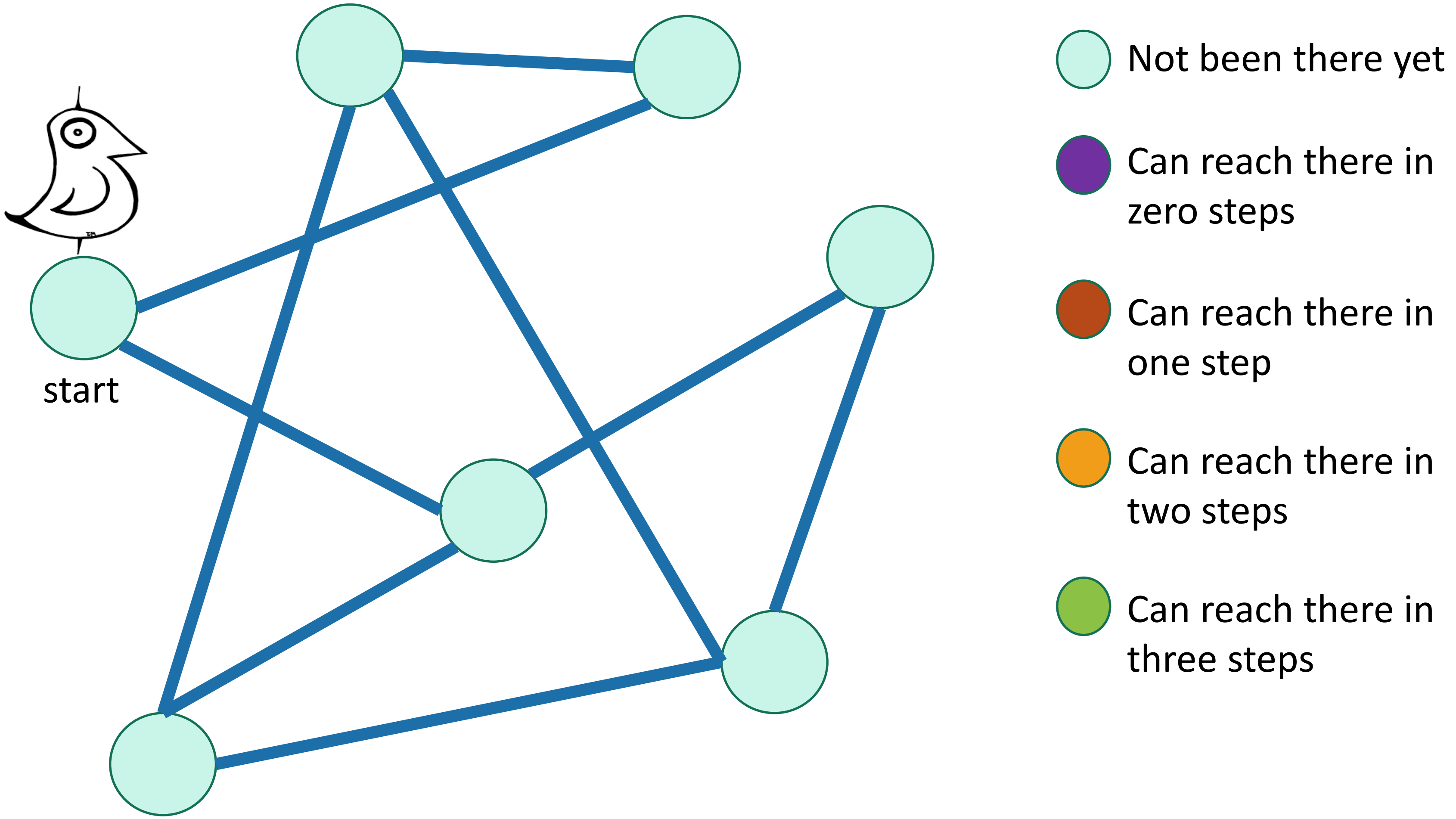
How do we explore a graph?

If we can fly



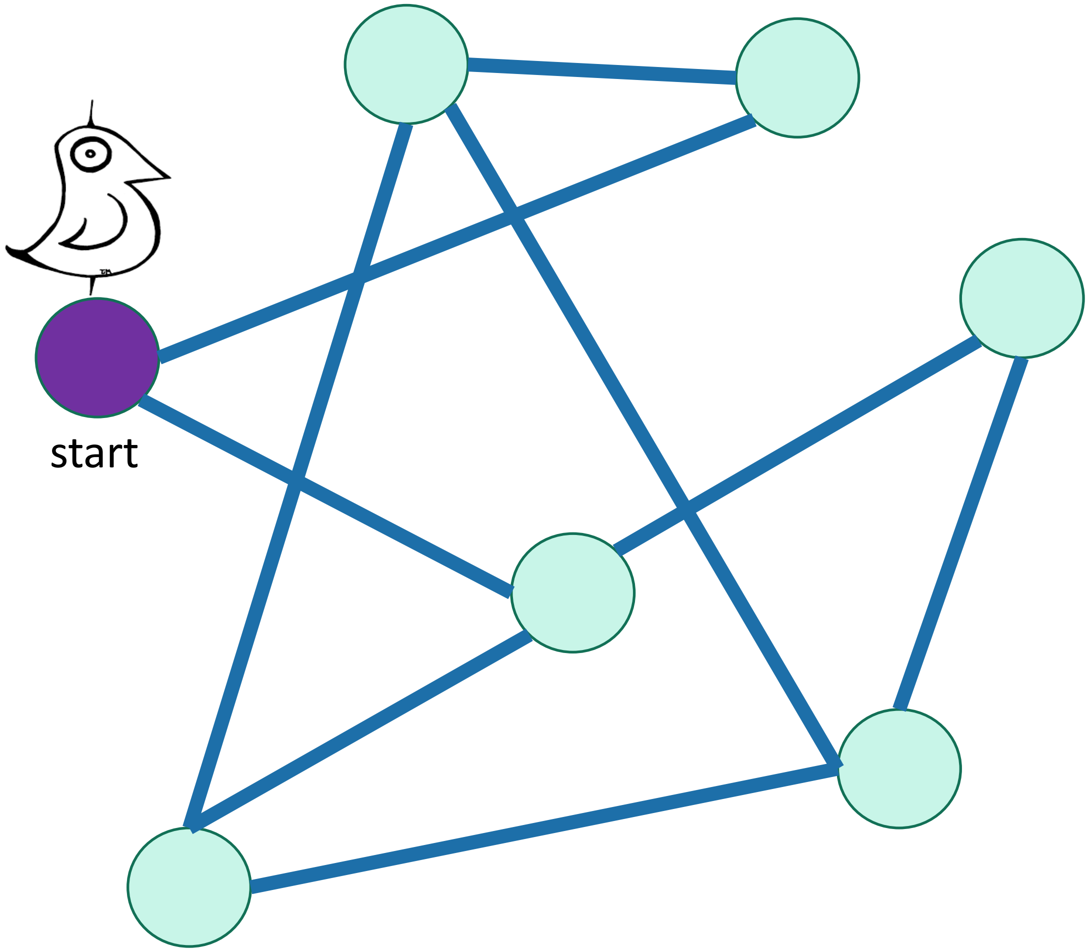
Breadth-First Search

Exploring the world with a bird's-eye view



Breadth-First Search

Exploring the world with a bird's-eye view



Not been there yet

Can reach there in zero steps

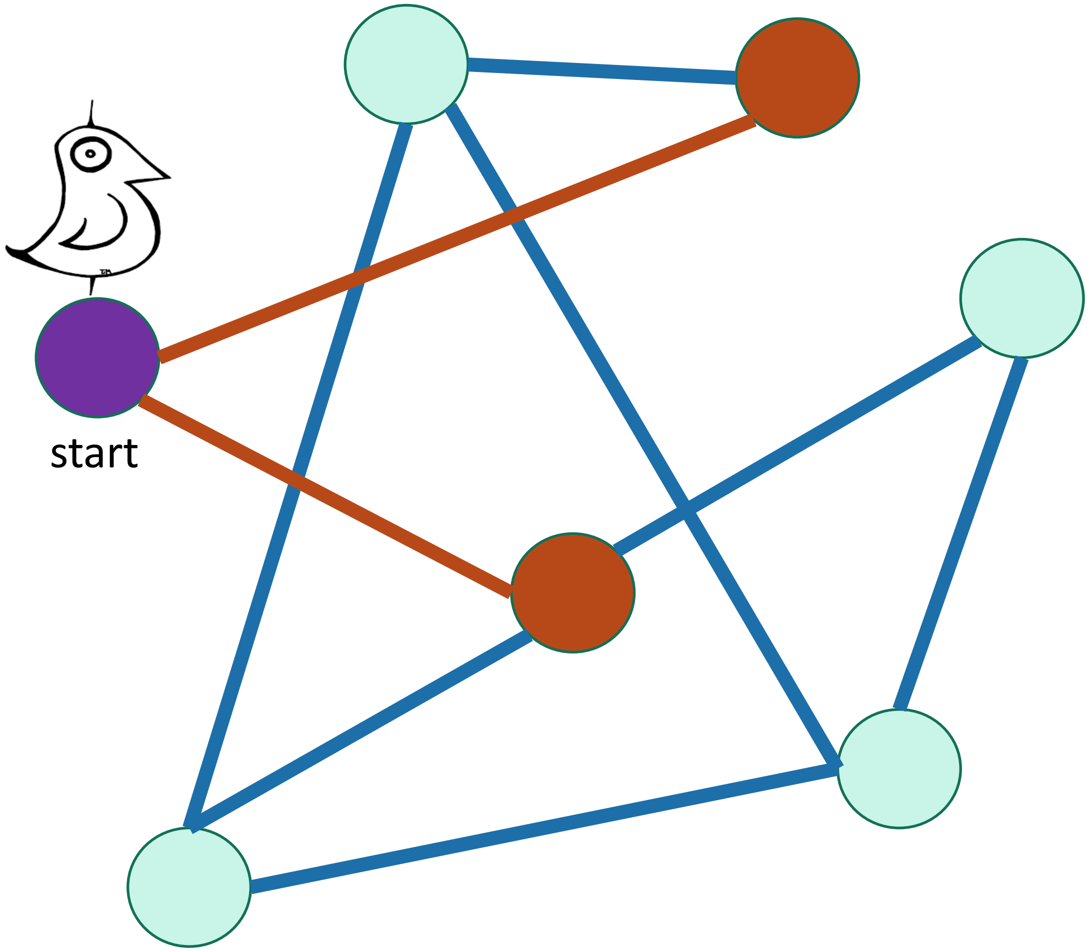
Can reach there in one step

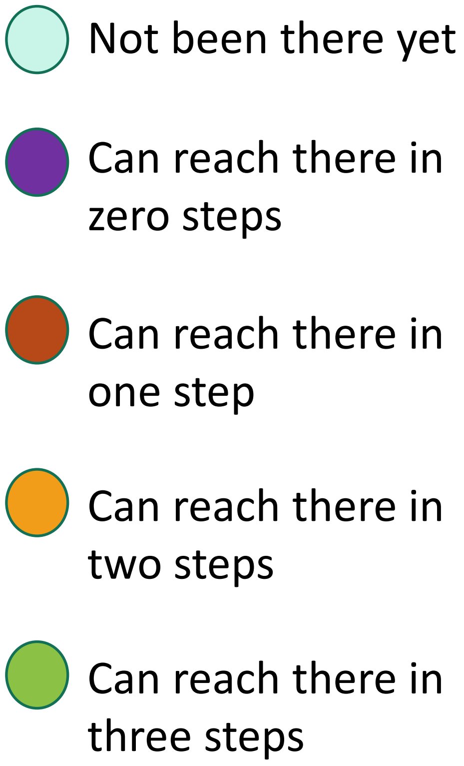
Can reach there in two steps

Can reach there in three steps

Breadth-First Search

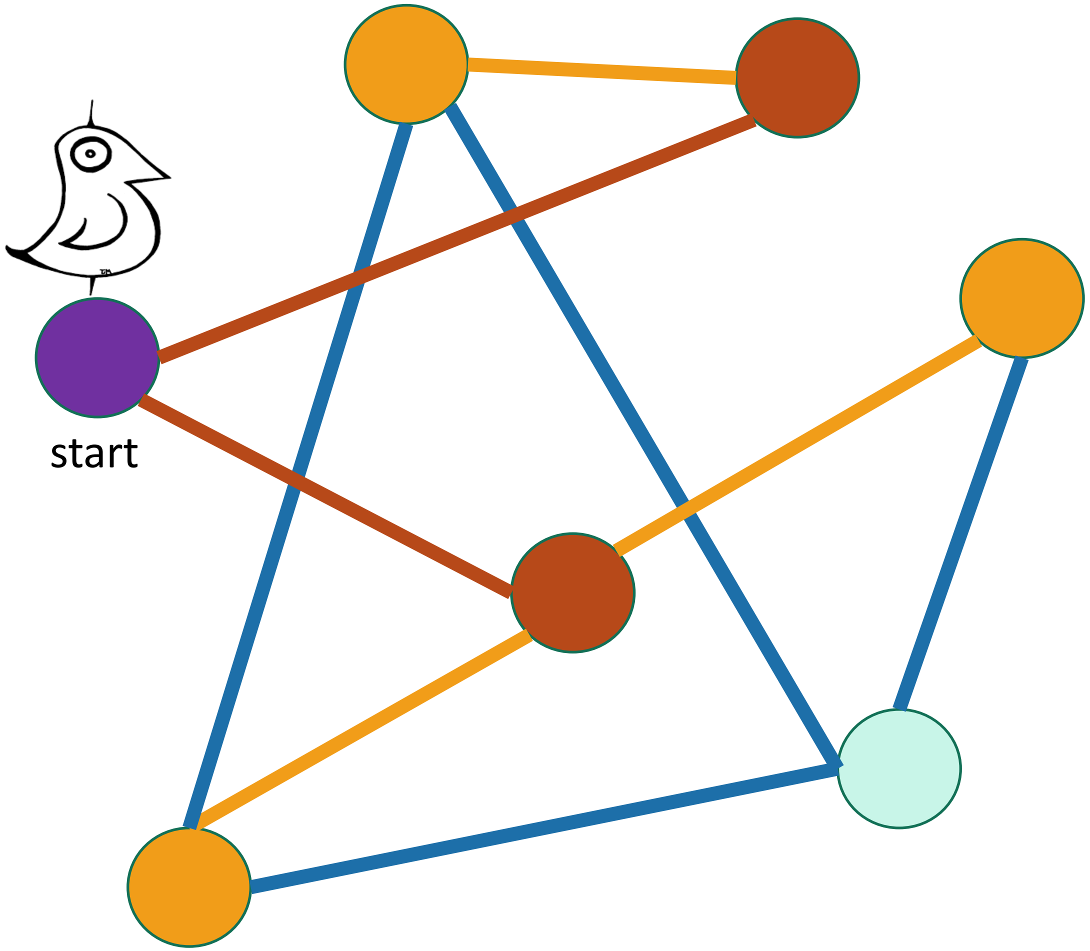
Exploring the world with a bird's-eye view





Breadth-First Search

Exploring the world with a bird's-eye view



Not been there yet

Can reach there in zero steps

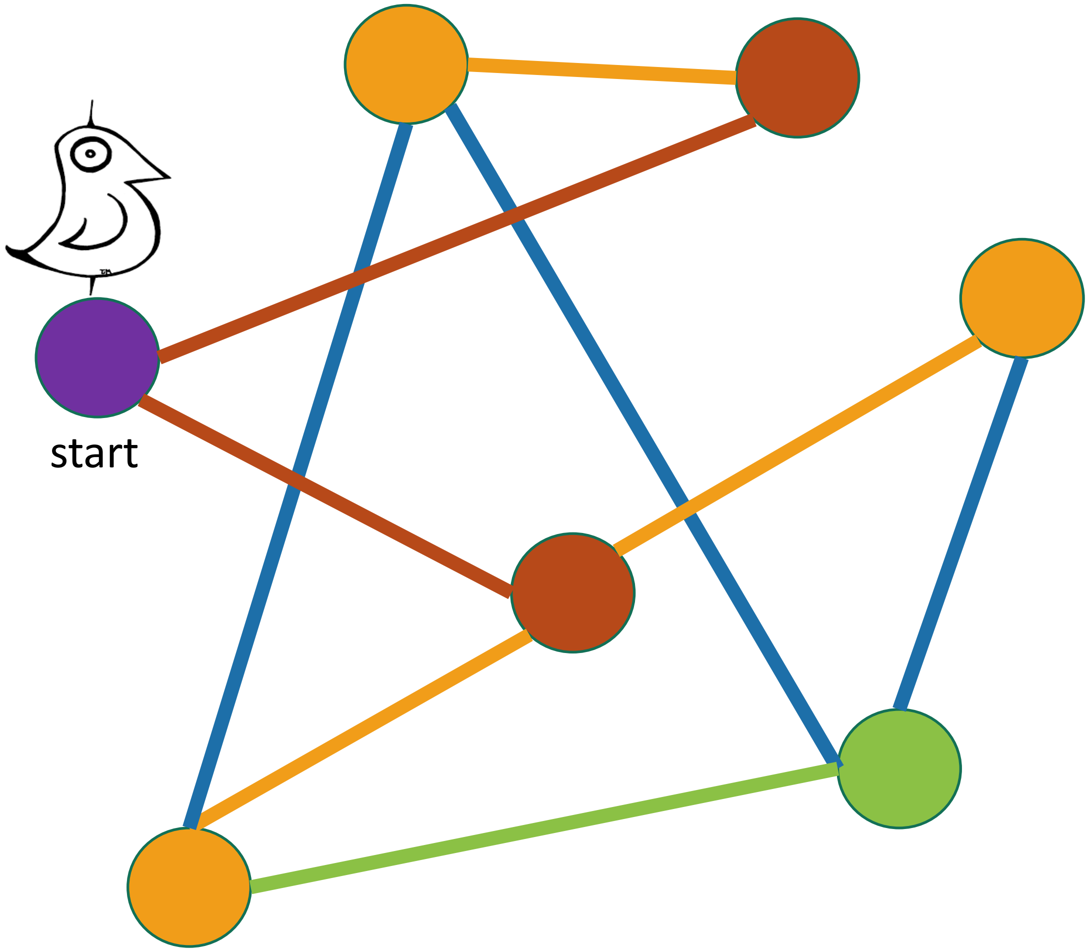
Can reach there in one step

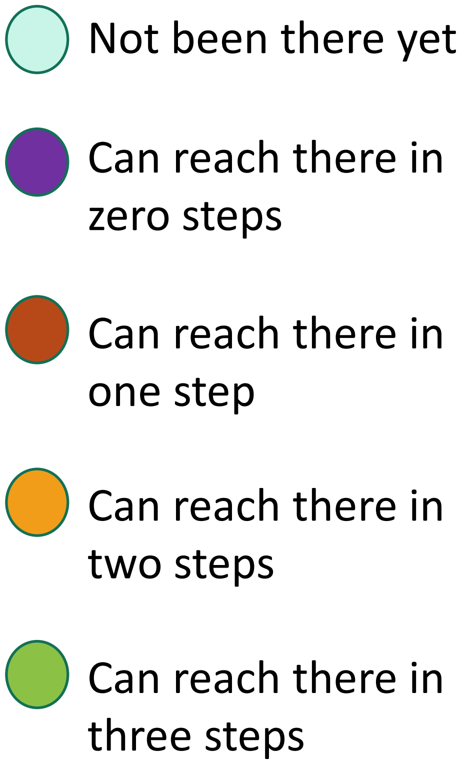
Can reach there in two steps

Can reach there in three steps

Breadth-First Search

Exploring the world with a bird's-eye view





World:

explored!

Same disclaimer as for DFS: you may have seen other ways to implement this, this will be convenient for us.

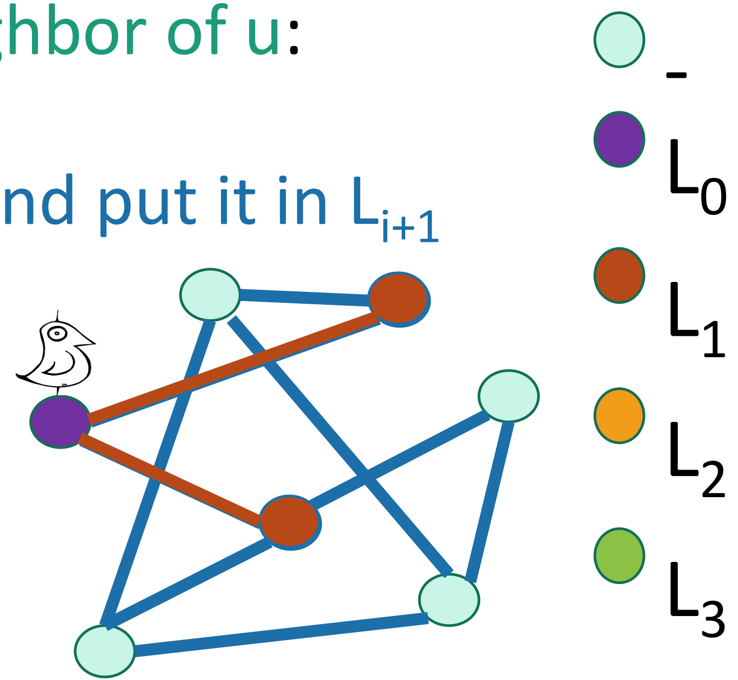
Breadth-First Search

Exploring the world with pseudocode

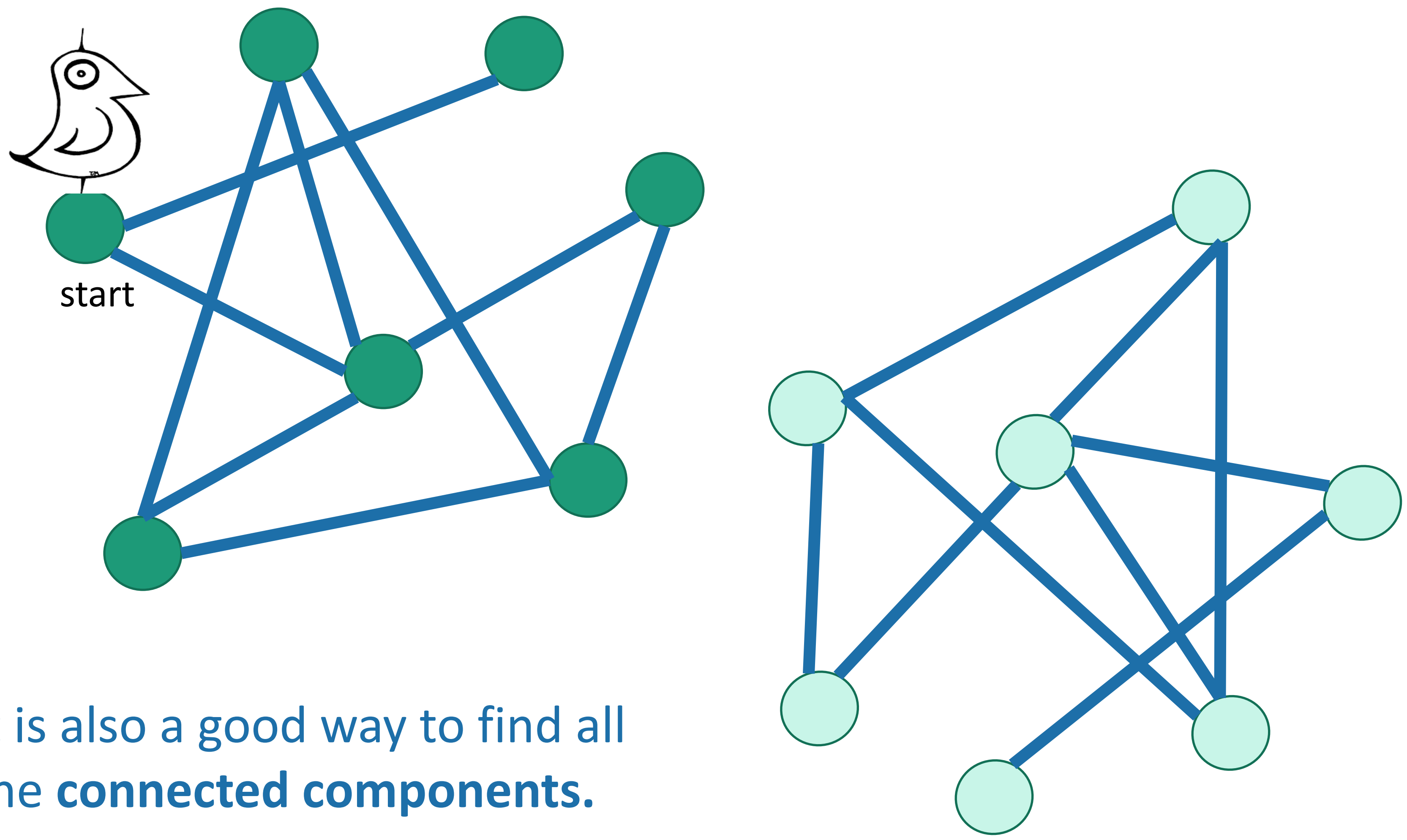
- Set L_i = {} for i=1,…,n
- L_0 = {w}, where w is the start node
- **For** i = 0, …, n-1:
  - **For** u in L_i:
    - **For** each v which is a neighbor of u:
      - **If** v isn't yet visited:
        - mark v as visited, and put it in L_{i+1}

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

$$O(n + m)$$

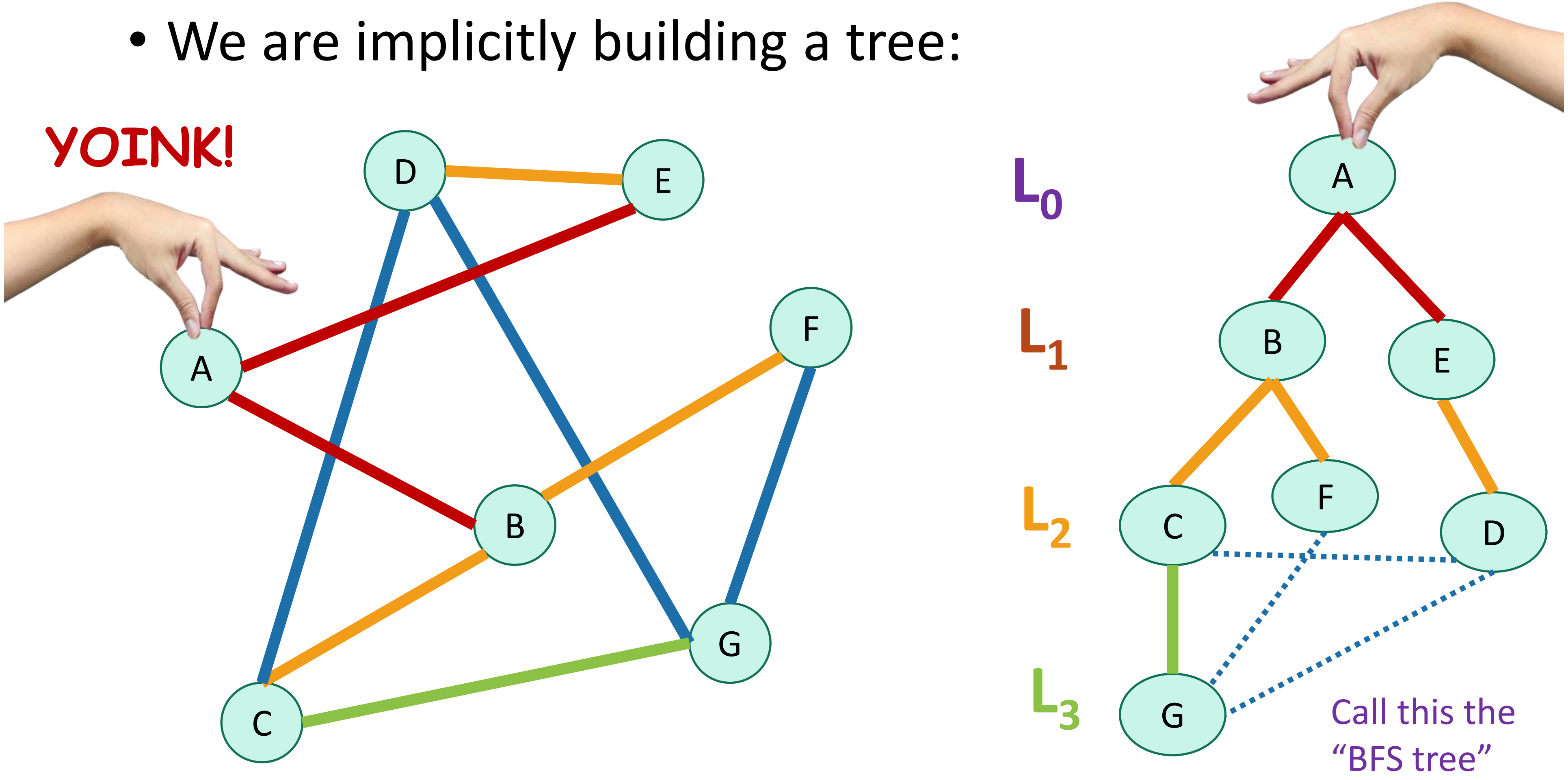
Verify these!

Ollie the over-achieving ostrich

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

Why is it called breadth-first?

- We are implicitly building a tree:

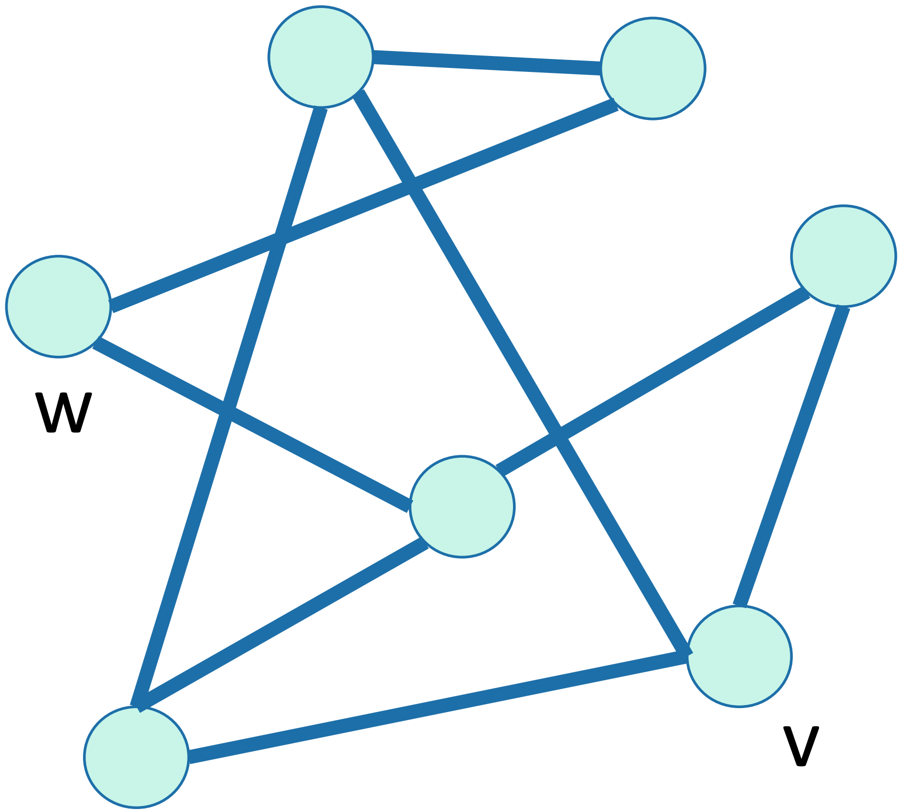


- And first we go as broadly as we can.

Applications of BFS

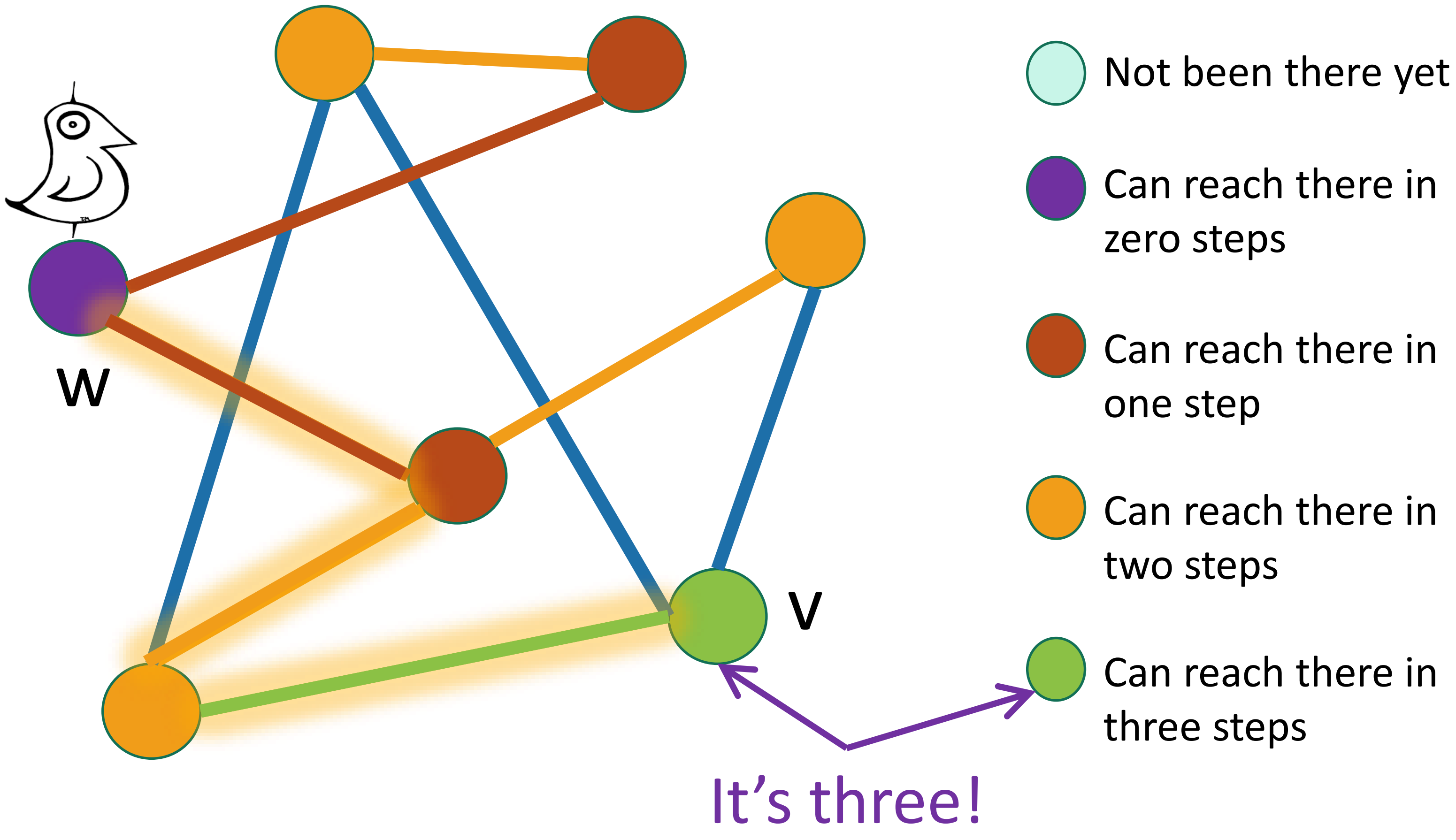
Application: shortest path

- 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 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?

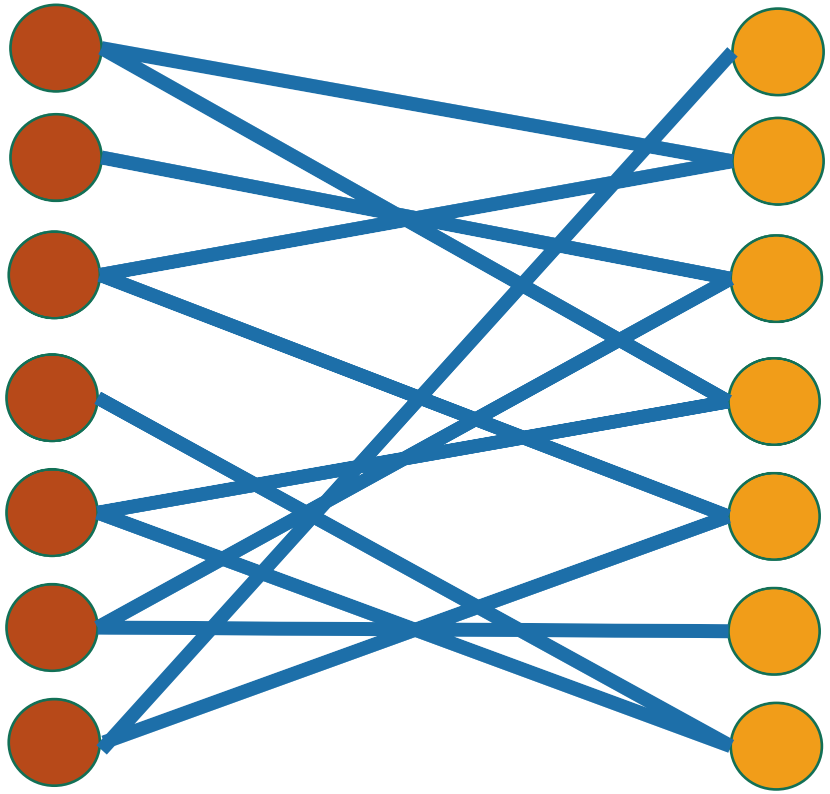
- The BFS tree is useful for computing distances between pairs of vertices.
- We can find the shortest path between u and v in time O(m).

The BSF tree is also helpful for:

- Testing if a graph is bipartite or not.

Application: testing if a graph is bipartite

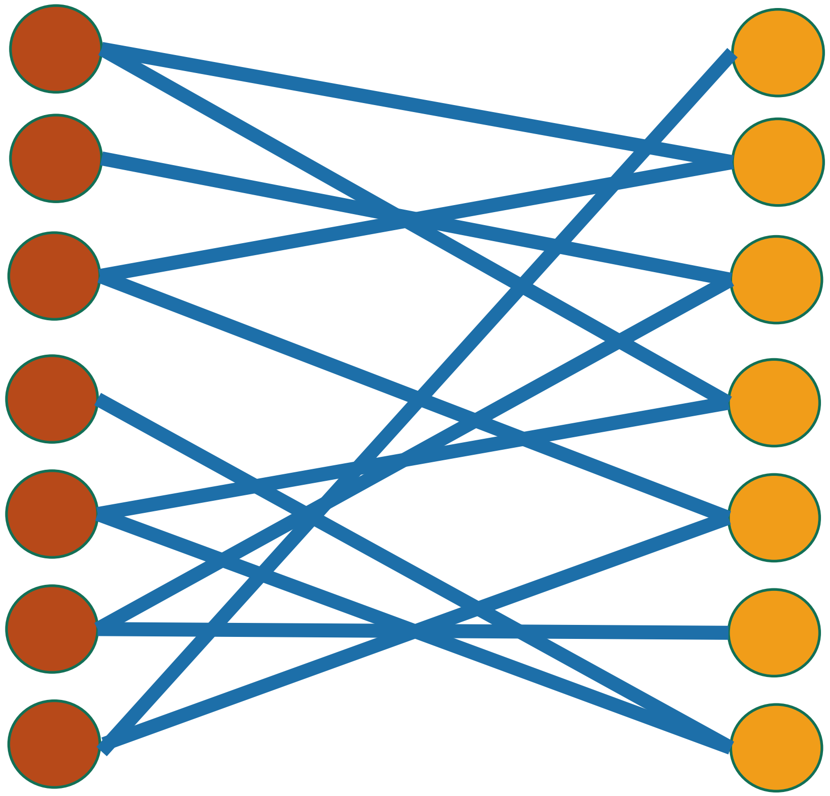
- Bipartite means it looks like this:



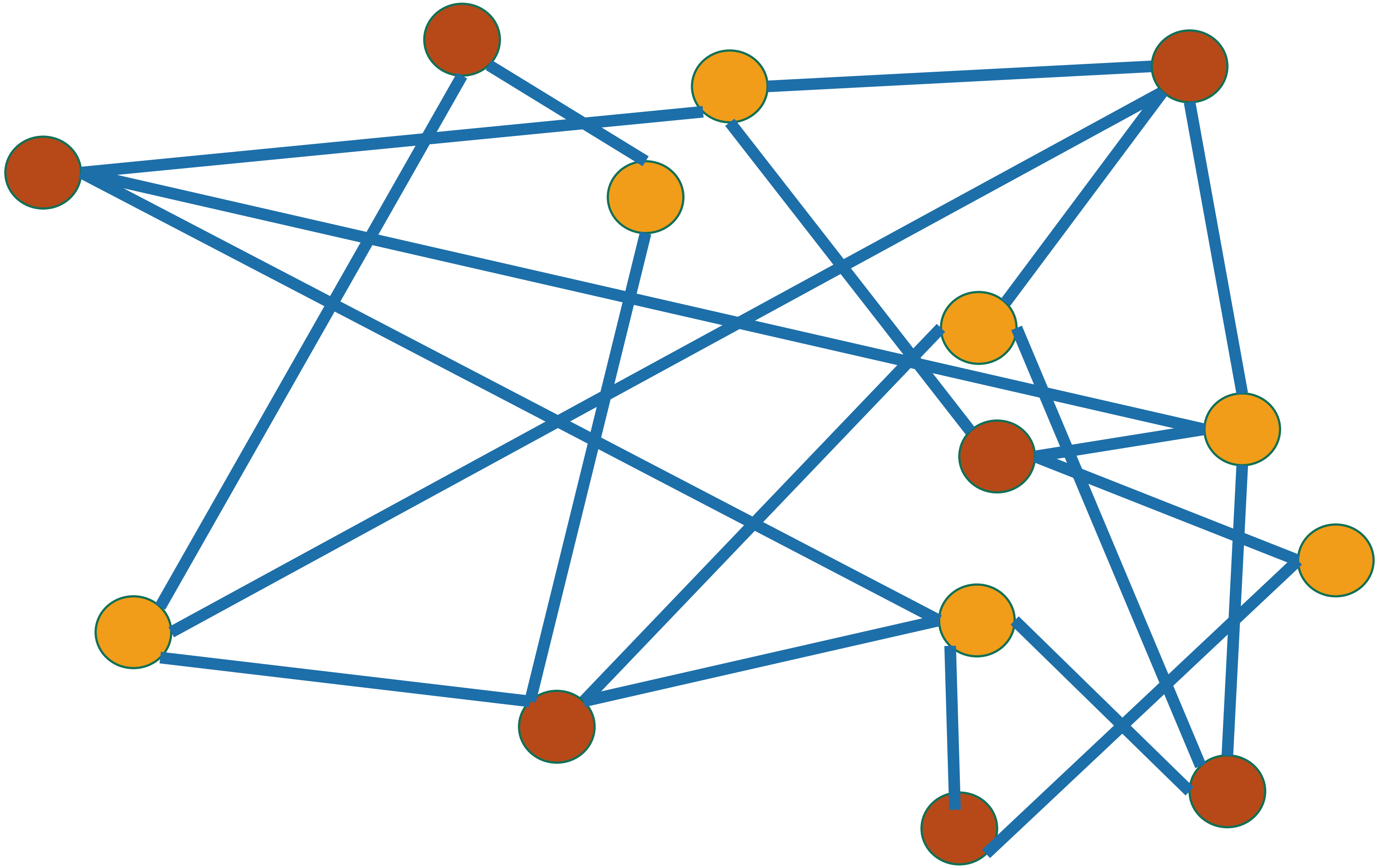
Can color the vertices red and orange so that there are no edges between any same-colored vertices

Example:
are students
are classes
if the student is enrolled in the class

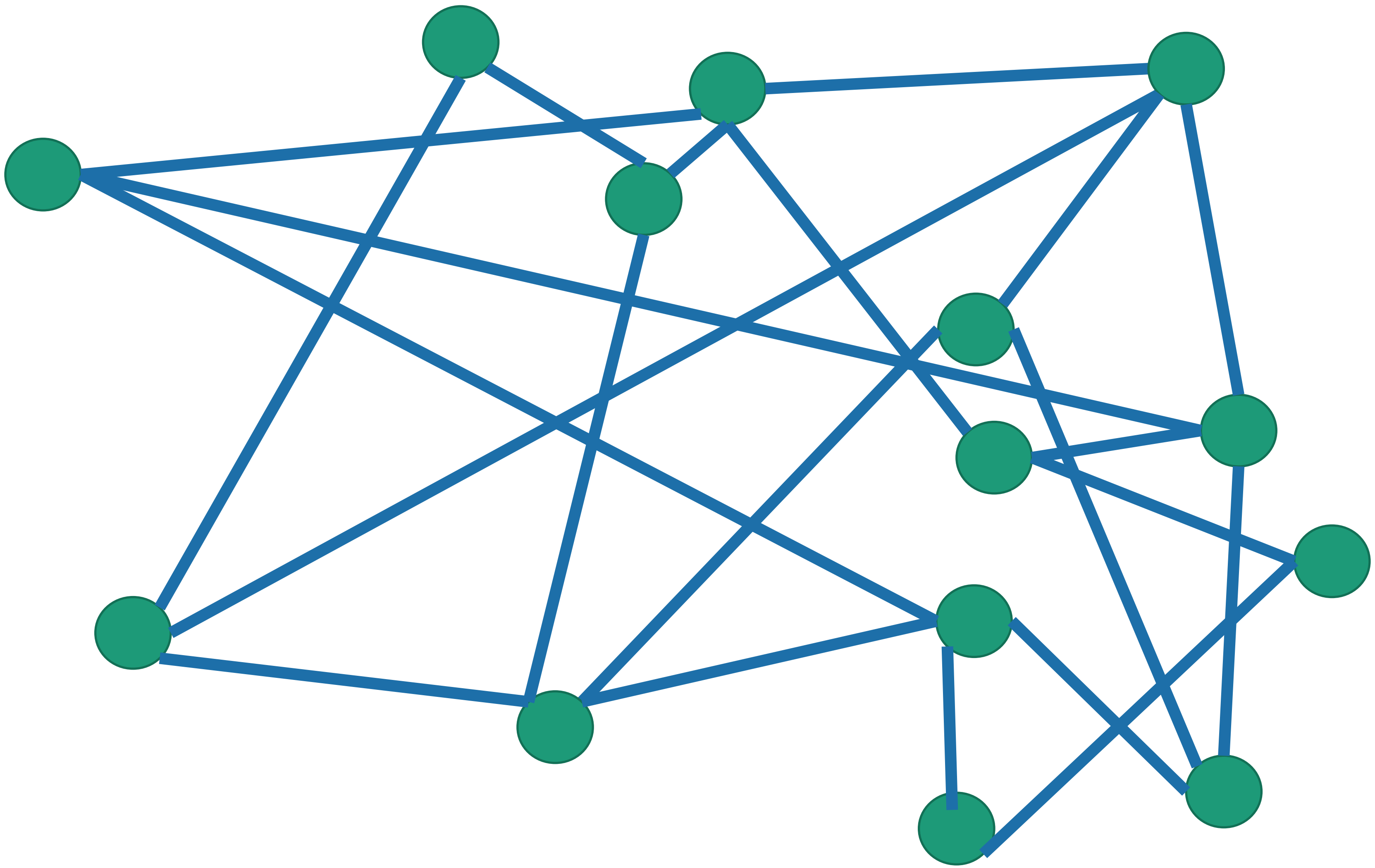
Is this graph bipartite?



How about this one?

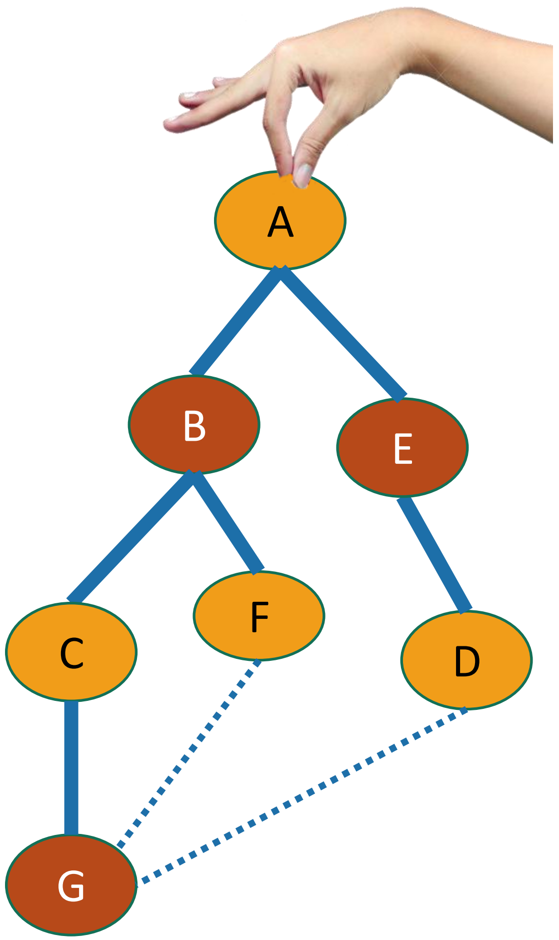


How about this one?



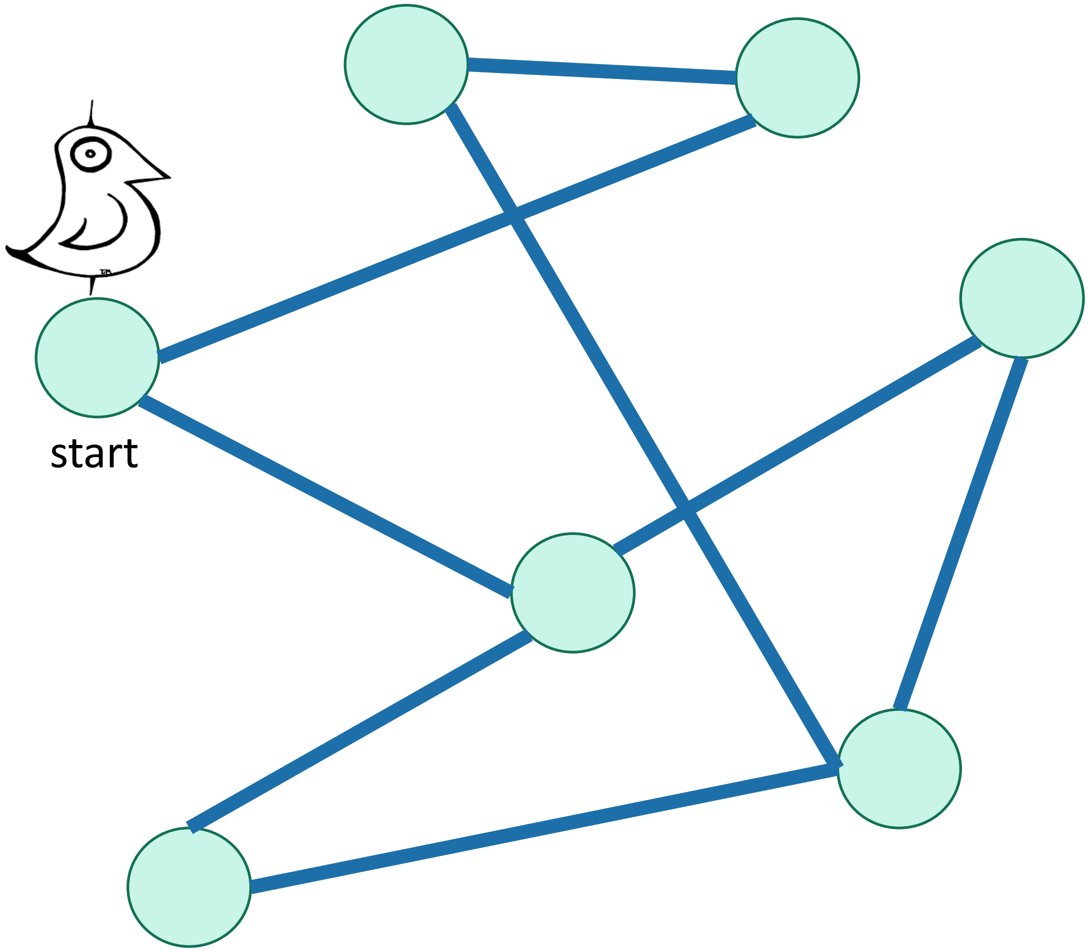
Solution using BFS

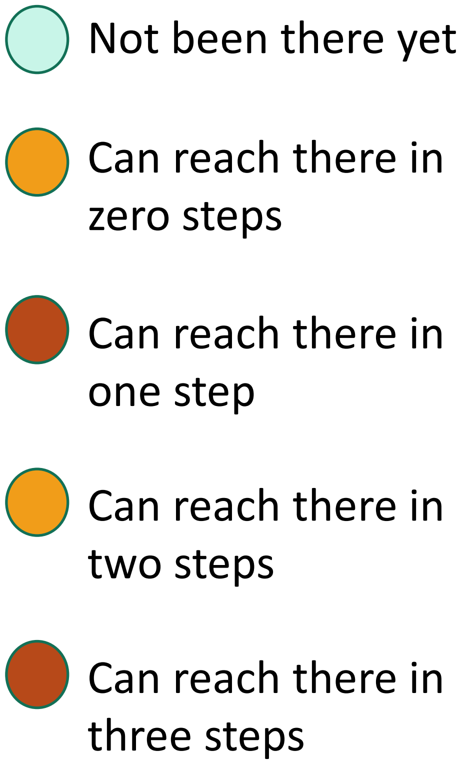
- Color the levels of the BFS tree in alternating colors.
- If you ever color a node so that you never color two connected nodes the same, then it is bipartite.
- Otherwise, it's not.



Breadth-First Search

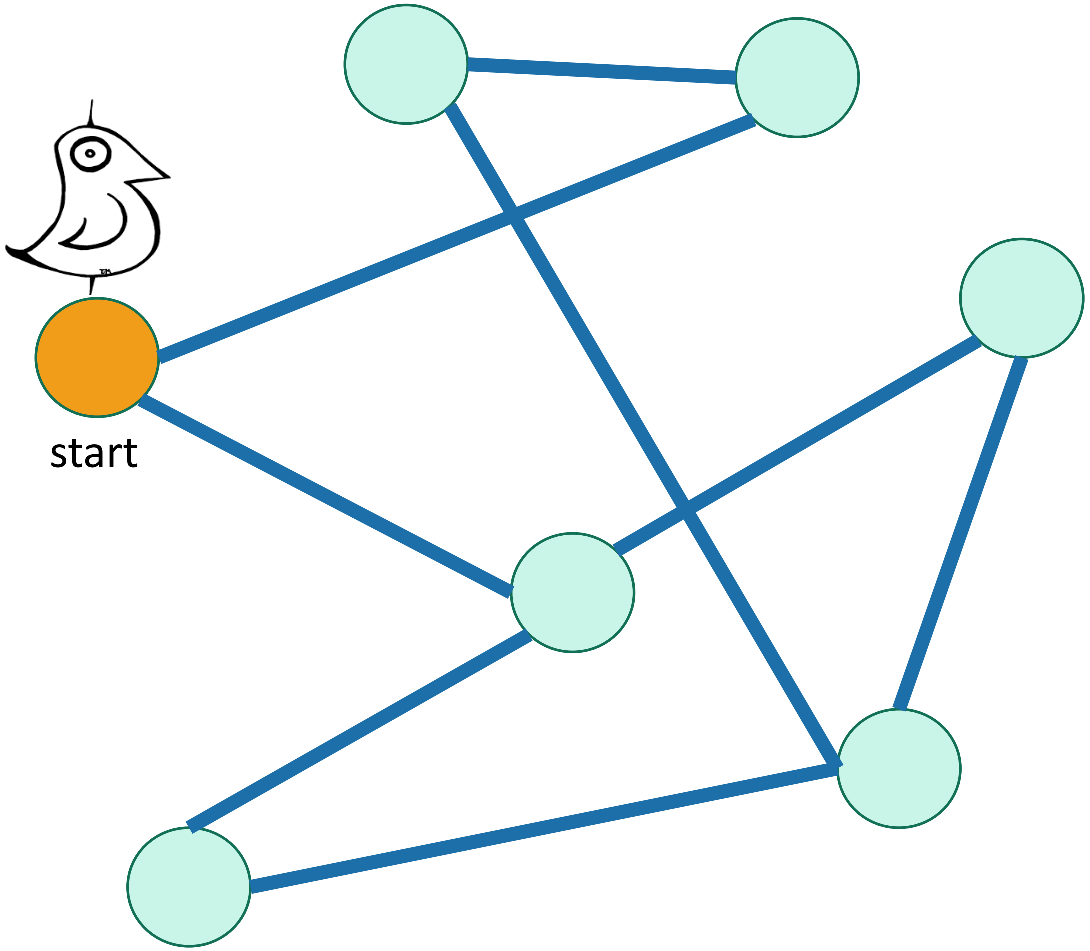
For testing bipartite-ness





Breadth-First Search

For testing bipartite-ness



Not been there yet

Can reach there in zero steps

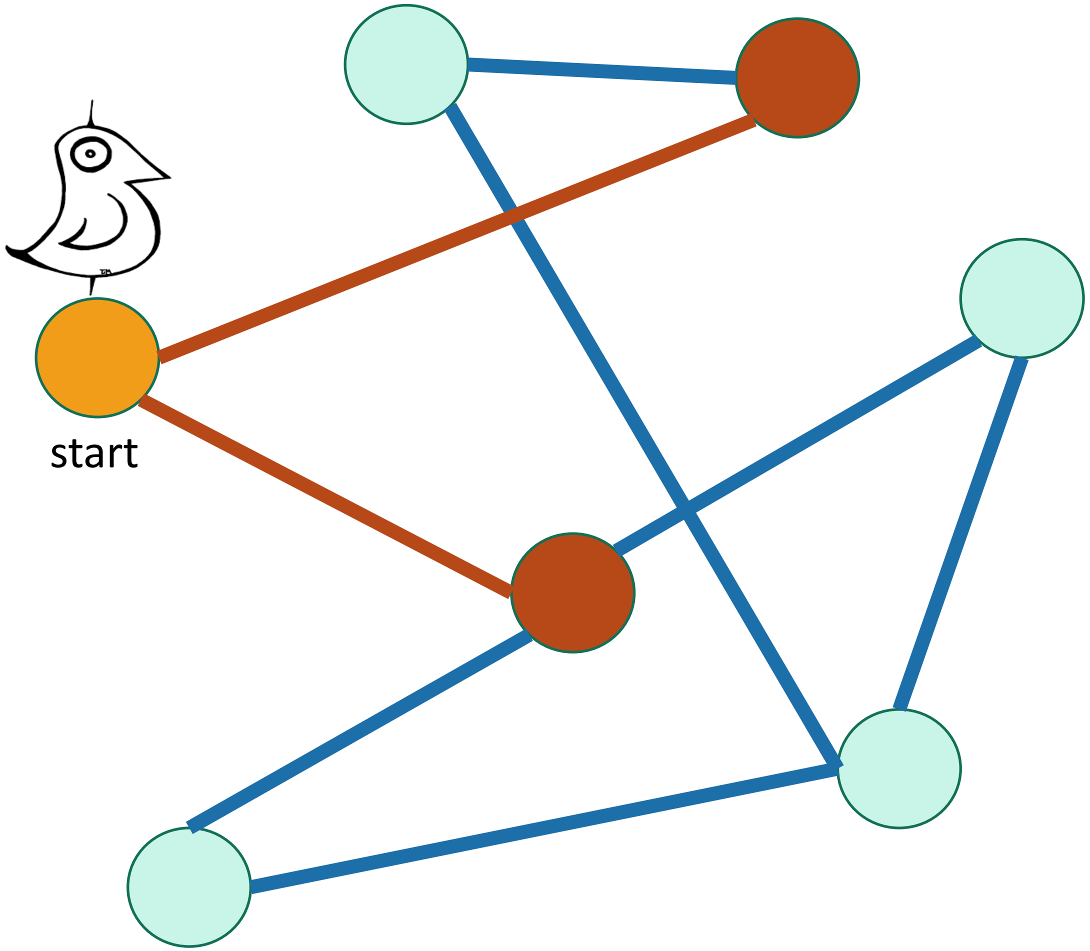
Can reach there in one step

Can reach there in two steps

Can reach there in three steps

Breadth-First Search

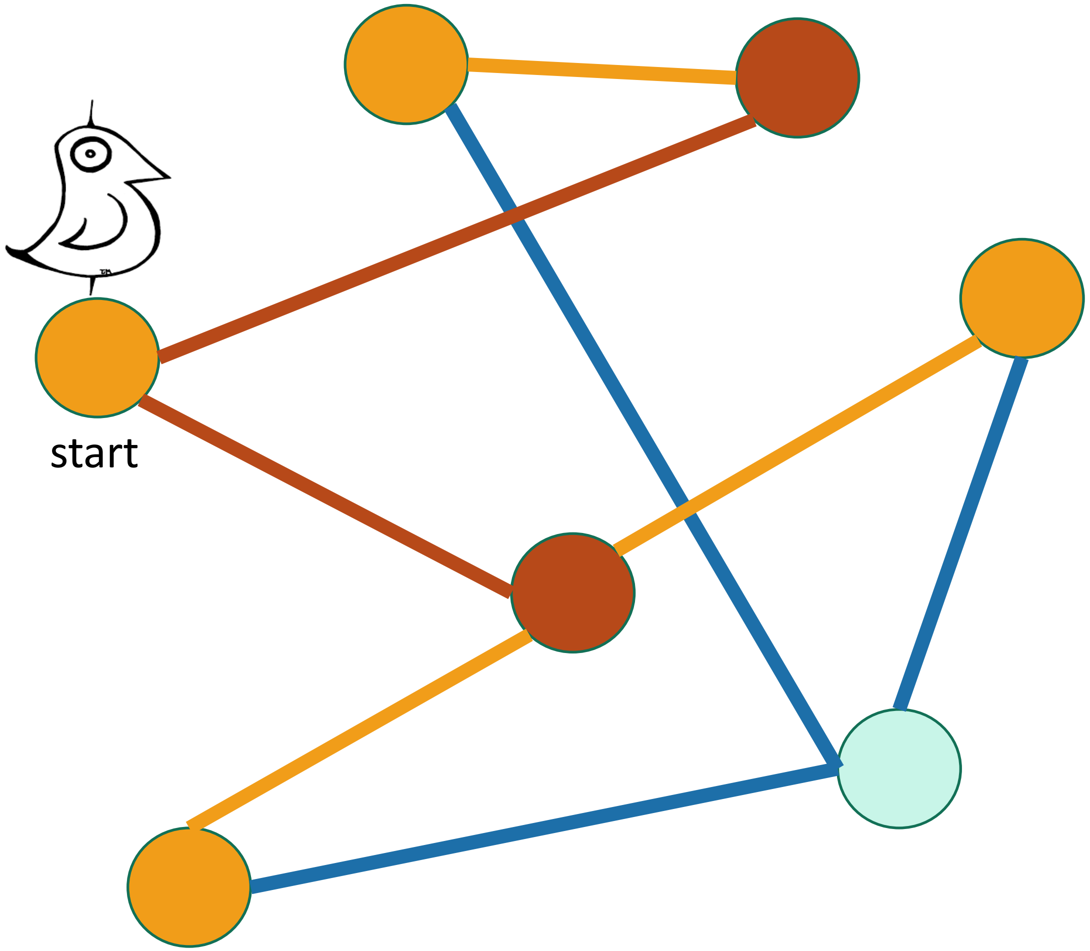
For testing bipartite-ness



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

Breadth-First Search

For testing bipartite-ness



Not been there yet

Can reach there in zero steps

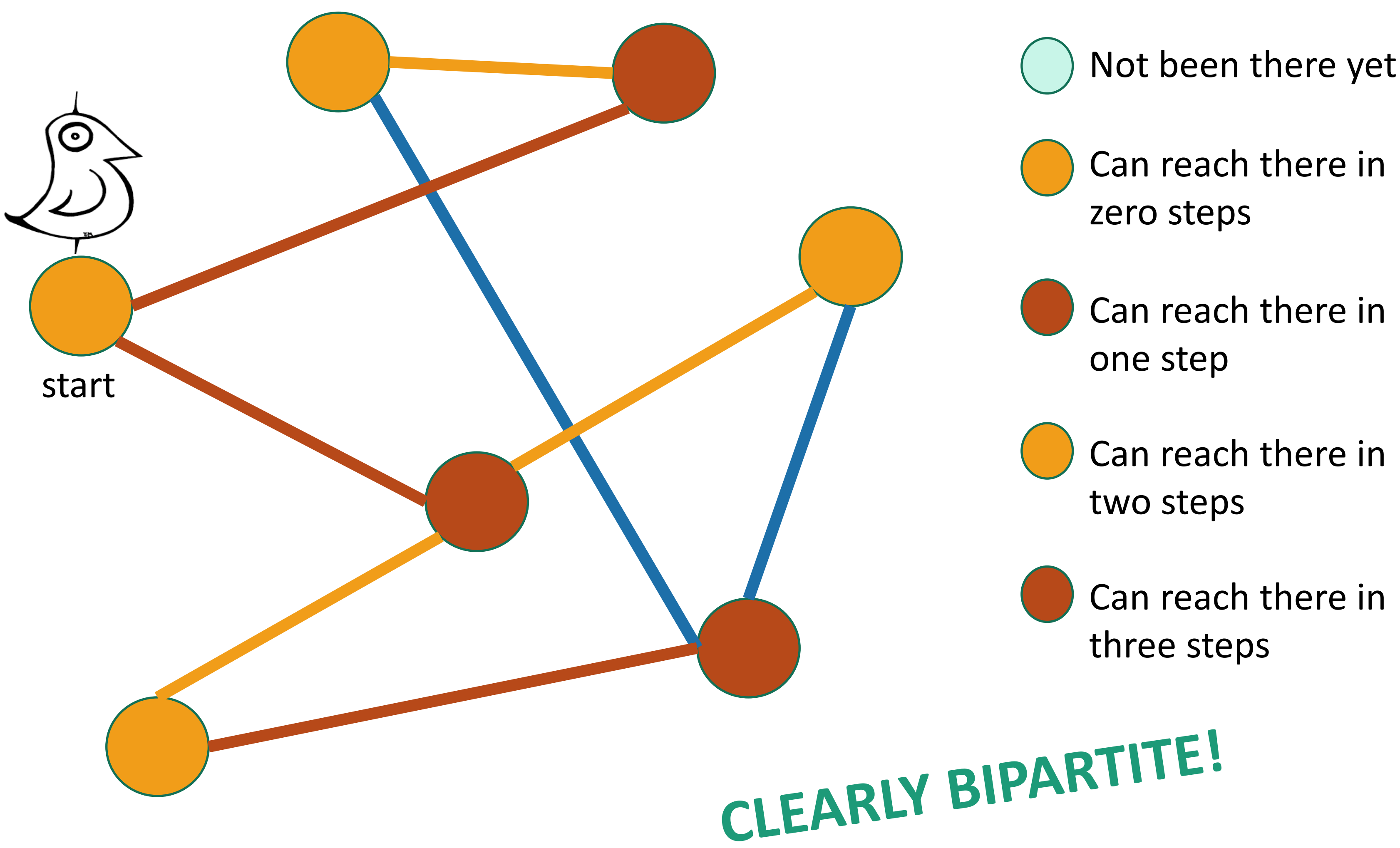
Can reach there in one step

Can reach there in two steps

Can reach there in three steps

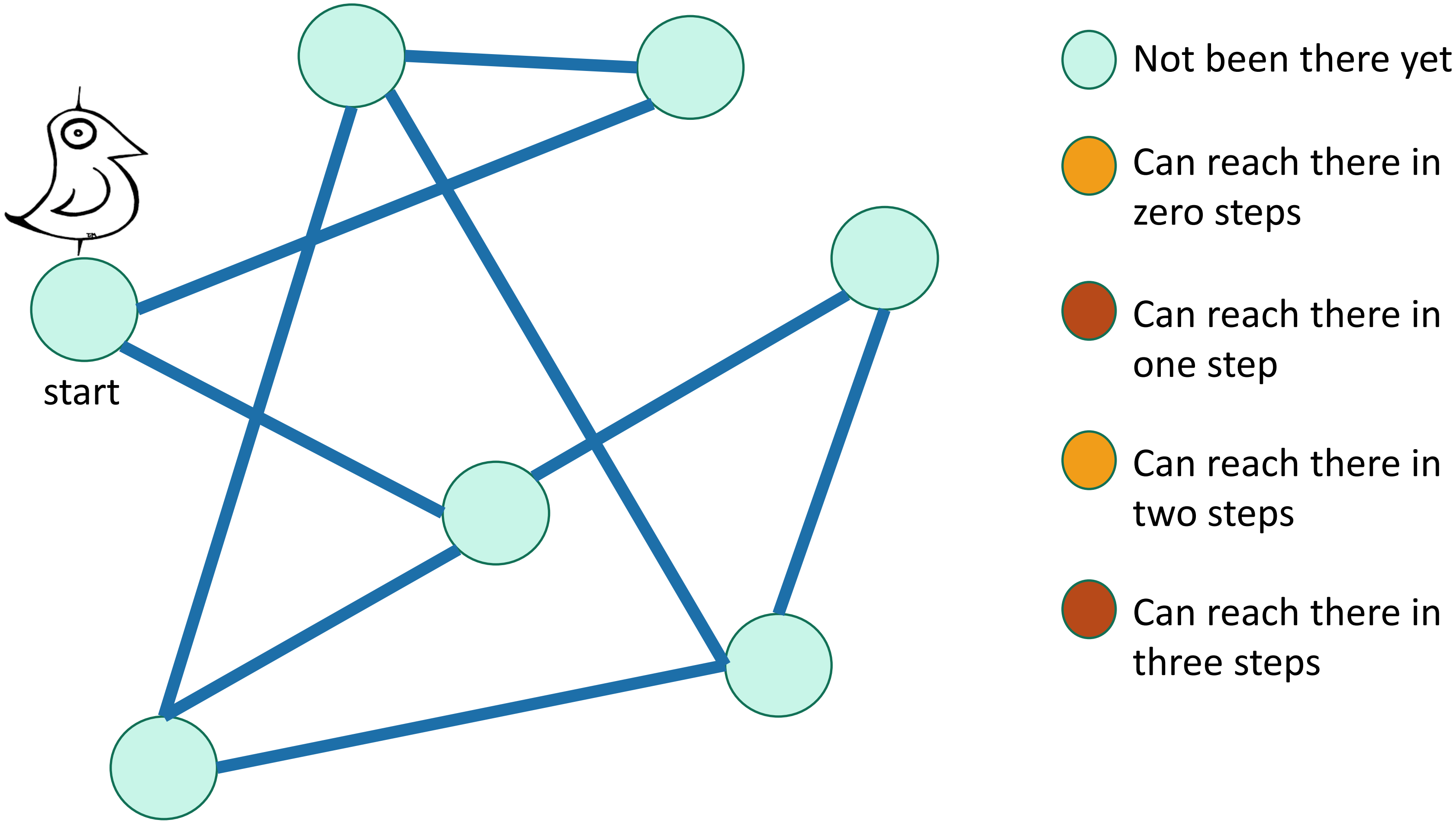
Breadth-First Search

For testing bipartite-ness



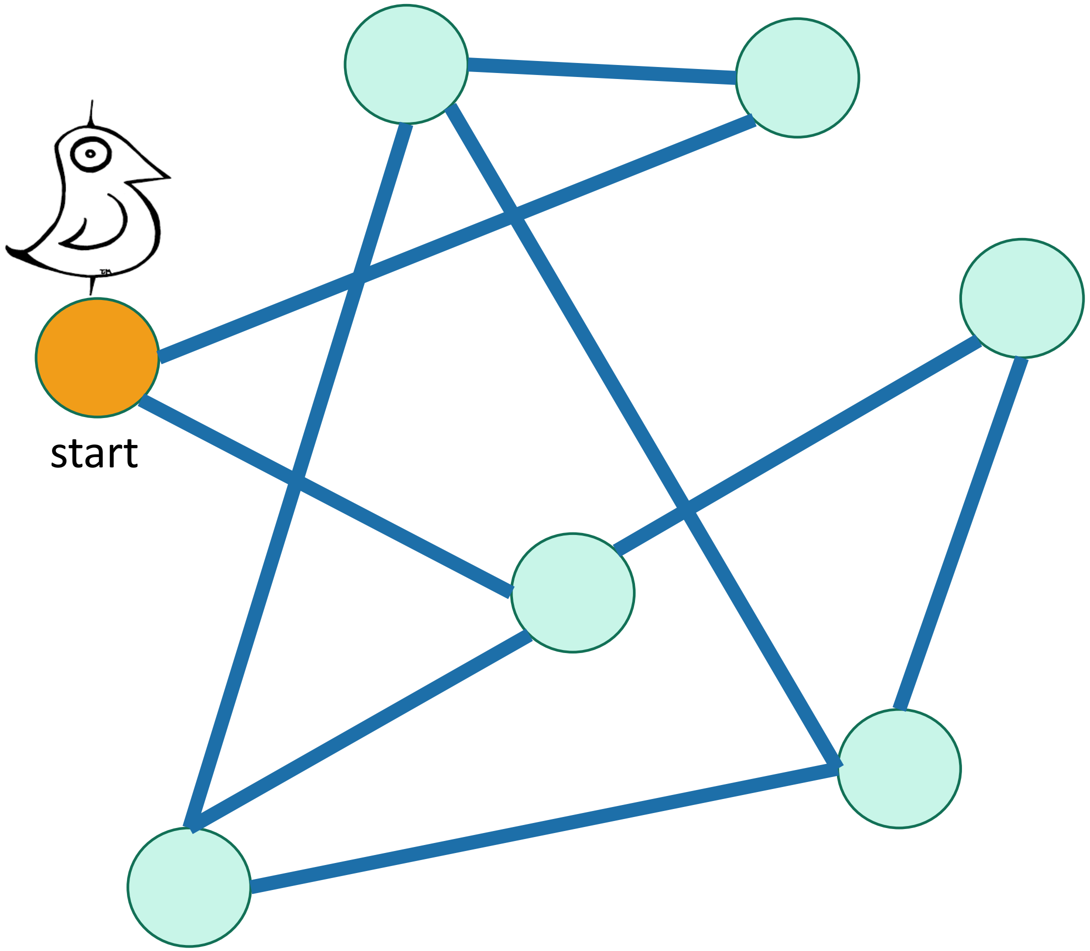
Breadth-First Search

For testing bipartite-ness



Breadth-First Search

For testing bipartite-ness



Not been there yet

Can reach there in zero steps

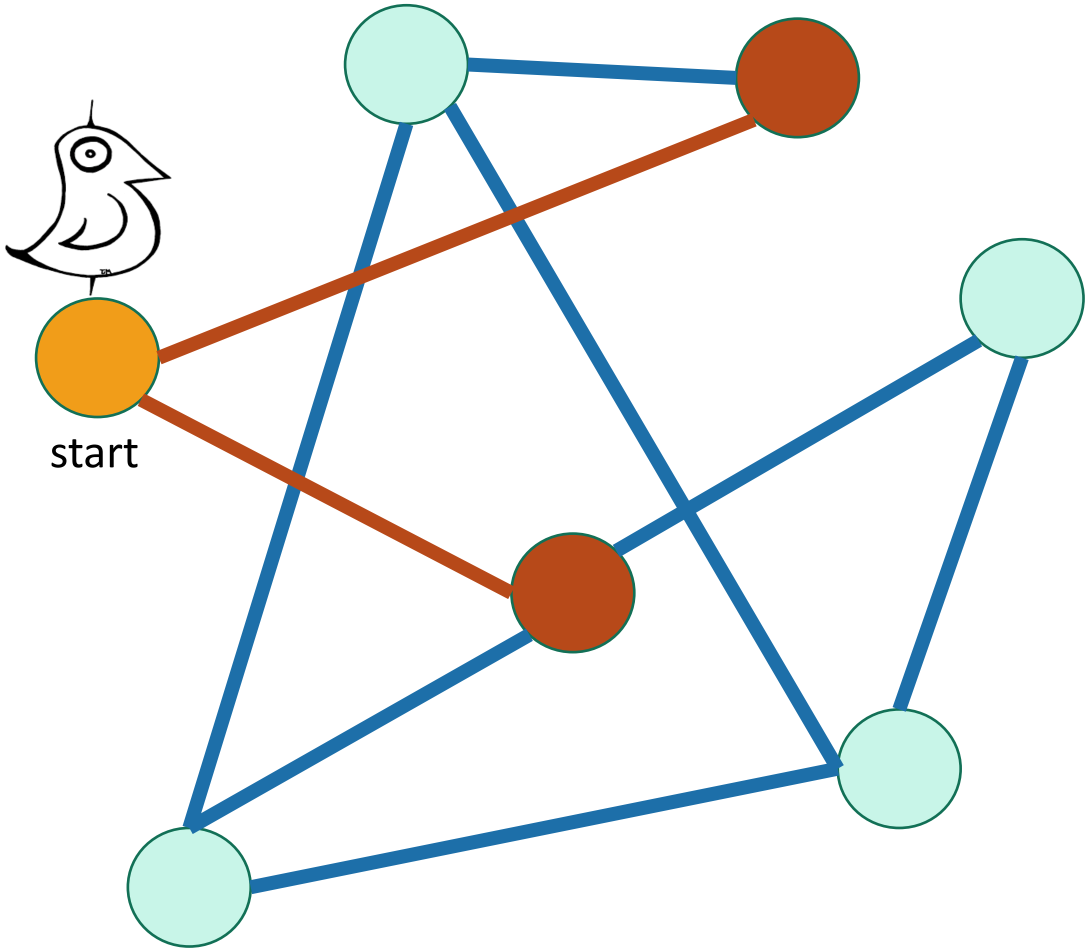
Can reach there in one step

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

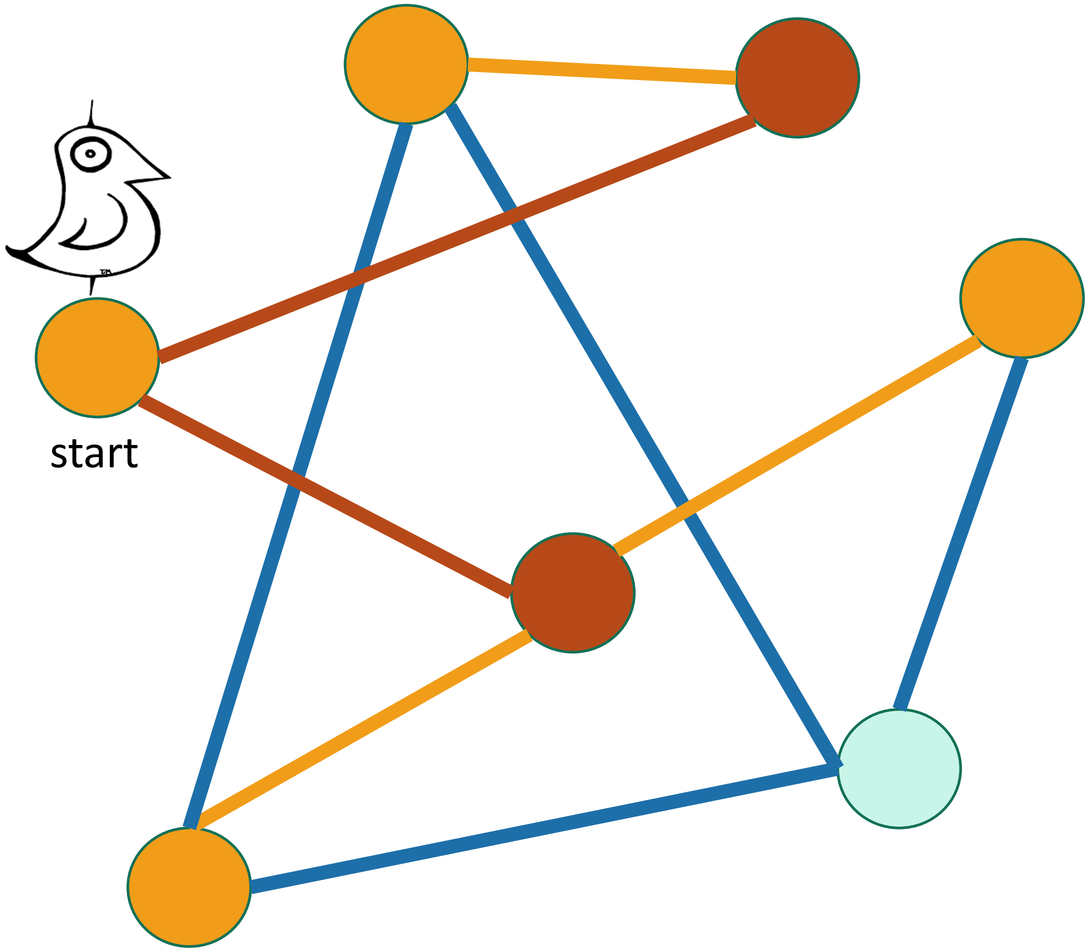
Can reach there in one step

Can reach there in two steps

Can reach there in three steps

Breadth-First Search

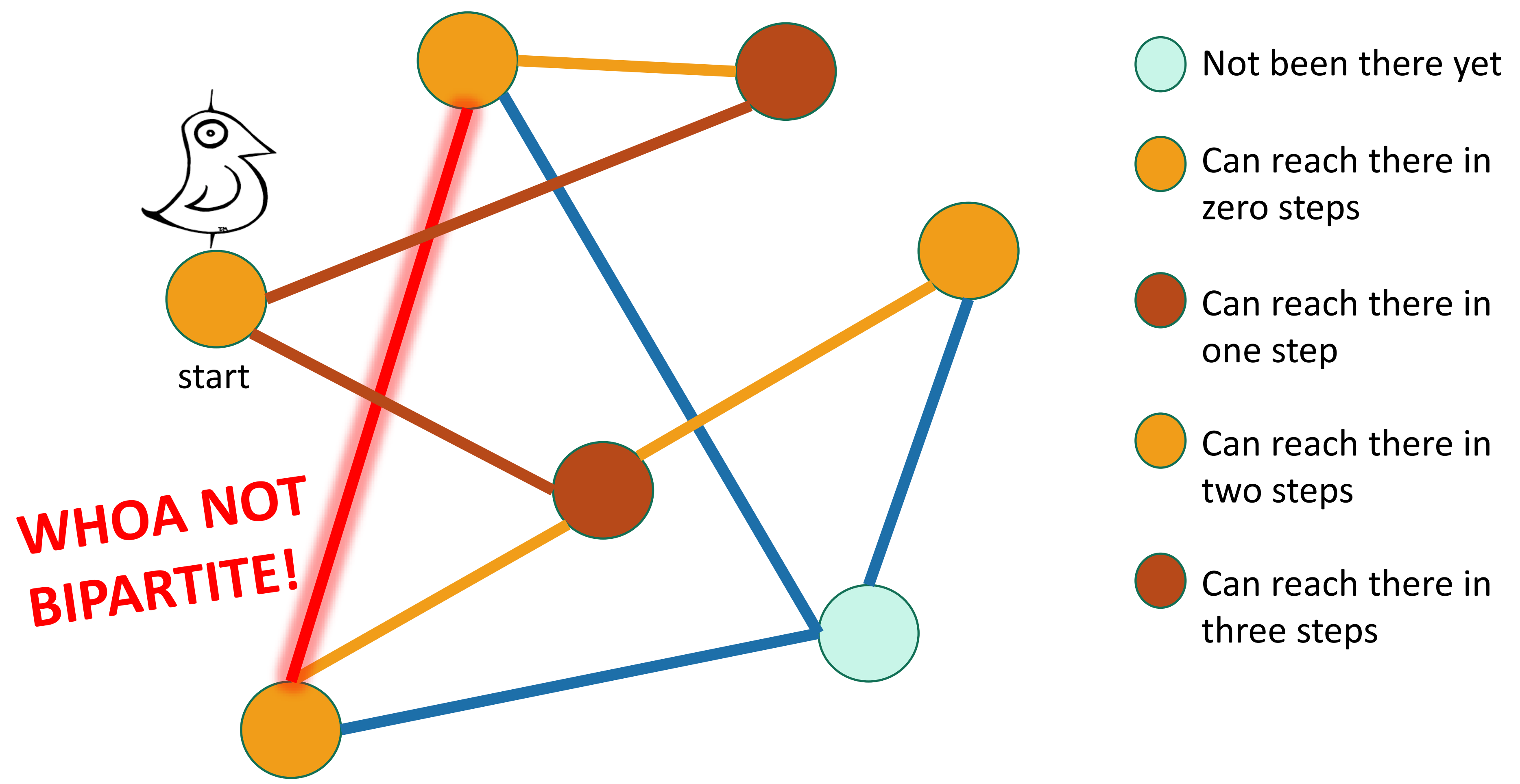
For testing bipartite-ness



- 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



What did we just learn?

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

- Consider a hash table of size m = 1000 and a corresponding hash function
  $h(k) = \lfloor m\ (kA \bmod 1) \rfloor$ for . Compute the locations to which the keys 61, 62, 63, 64, and 65 are mapped. A is equal to 0.618033