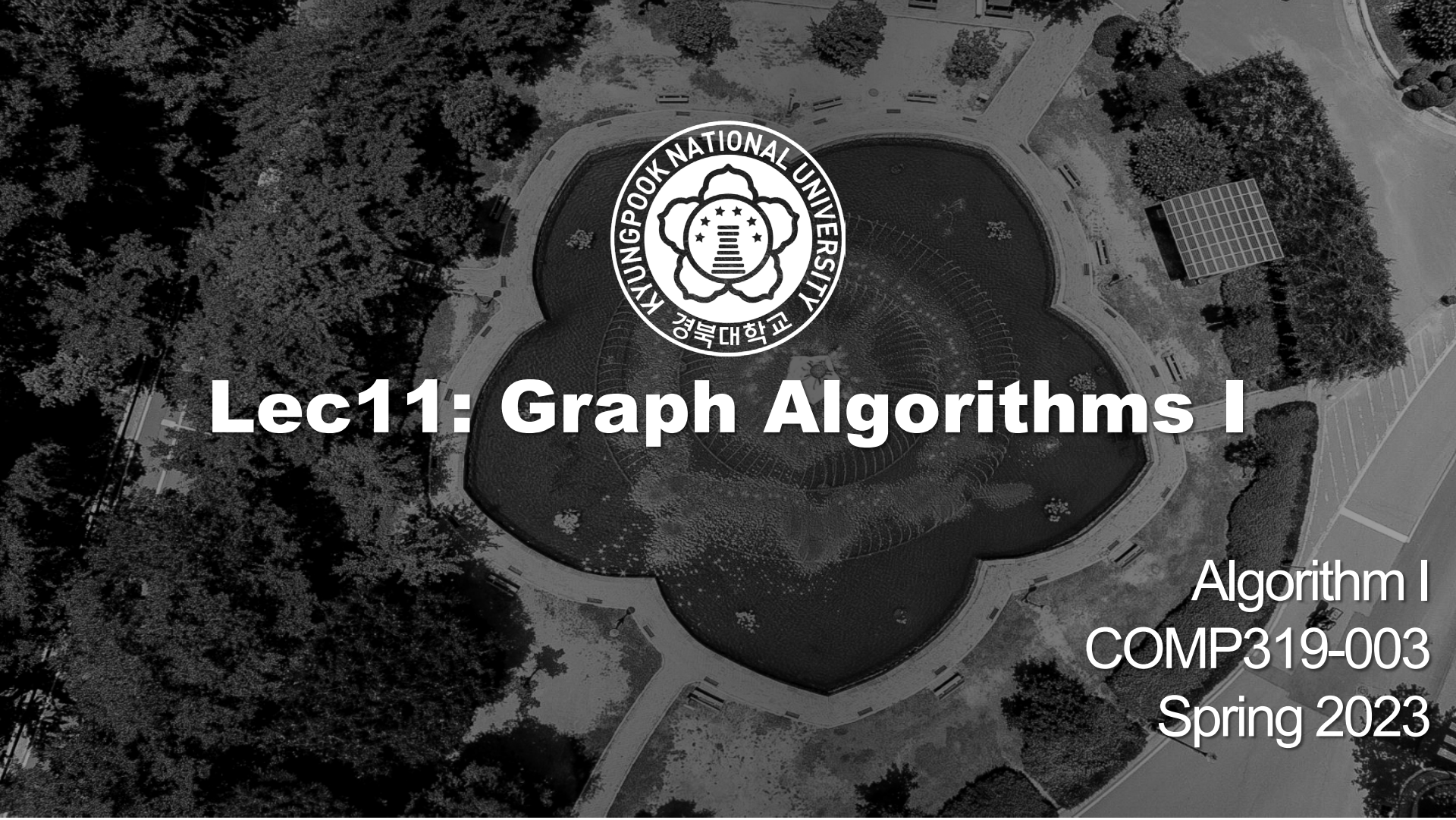


Lec11: Graph Algorithms I

Algorithm I
COMP319-003
Spring 2023

Instructor: Jiyeon Lee
School of Computer Science and Engineering
Kyungpook National University **(KNU)**

Course Overview

- Algorithmic Analysis
- Divide and Conquer
- Randomized Algorithms
- Tree Algorithms
- Hashing
- **Graph Algorithms**
- Dynamic Programming
- Greedy Algorithms
- NP-Completeness

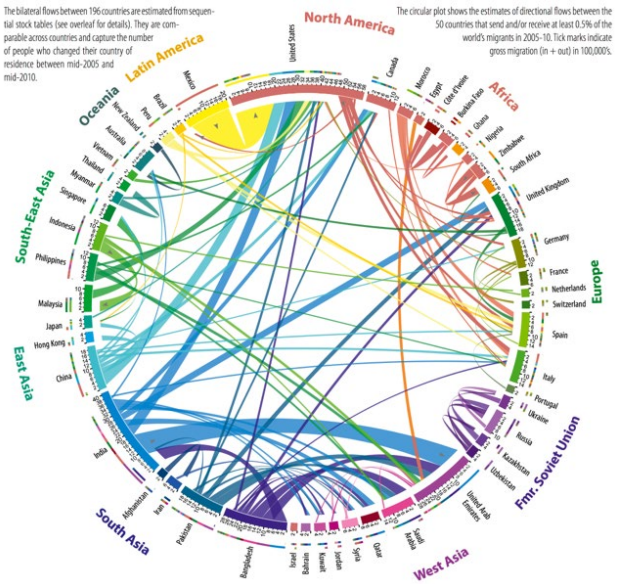
Outline

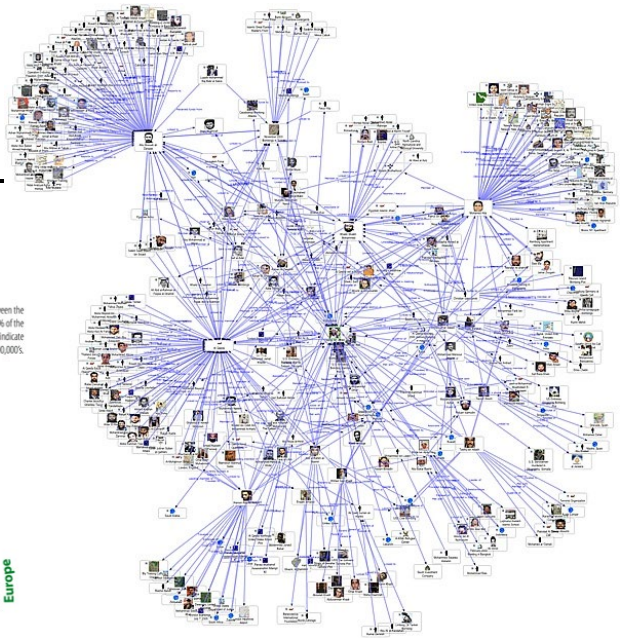
1. Graph Basics
   - Representations
   - DFS (Depth First Search)
   - BFS (Breadth First Search)

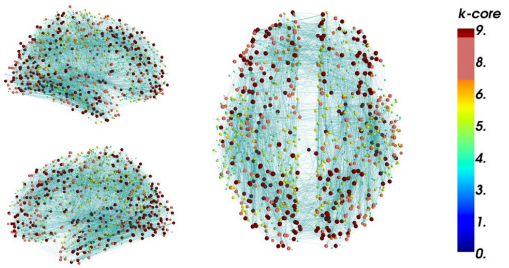
- *Reading: CLRS 22.1 – 22.4*

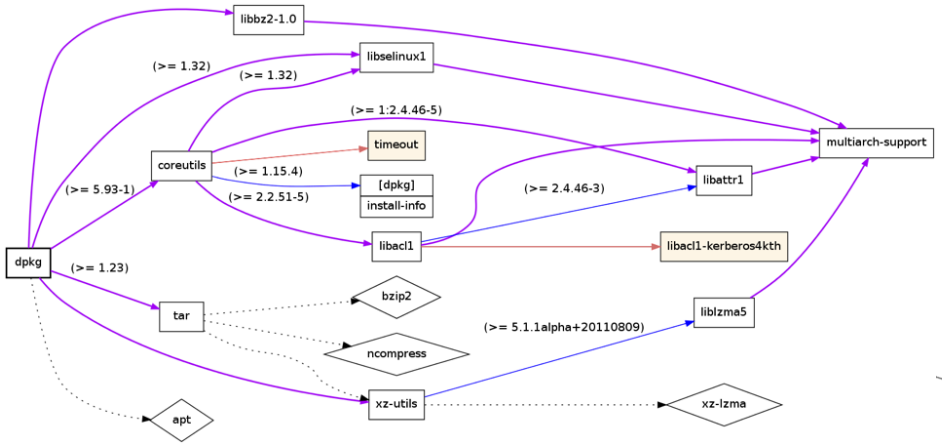
Examples of Graphs

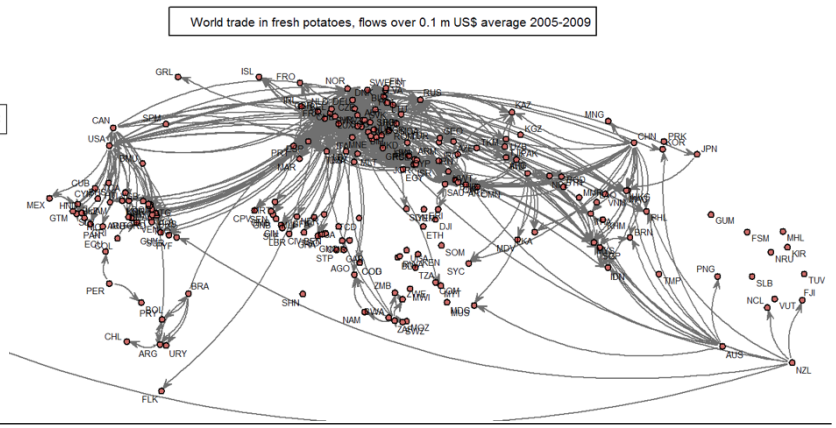
- Web
- Networks
- dependencies
- Neural connections
- Social relations
- …











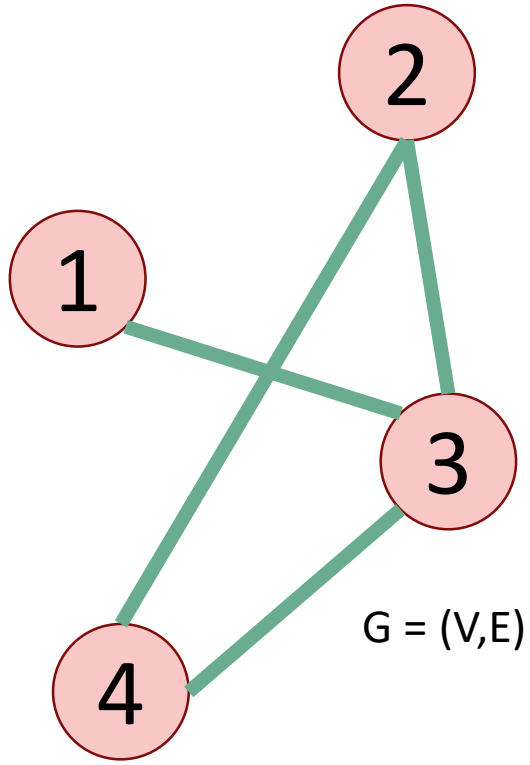
Graphs

- There are a lot of graphs.
- We want to answer questions about them.
  - **Efficient routing?**
    - Find the shortest path between two vertices
  - **Dependency?**
    - Find the topological ordering
  - **Community detection/clustering?**
    - Find strongly connected components
- This is what we'll do for the next several lectures.

Undirected Graphs

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

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

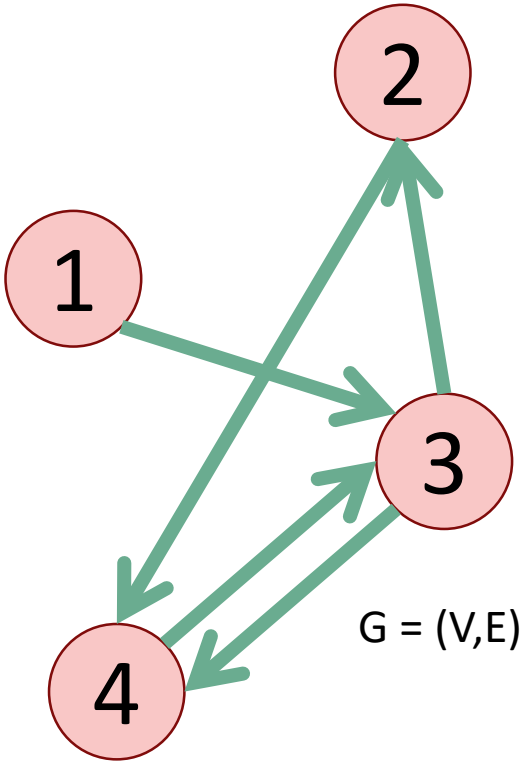


- The **degree** of vertex 4 is 2.
  - There are 2 edges coming out.
- Vertex 4's **neighbors** are 2 and 3

Directed Graphs

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

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

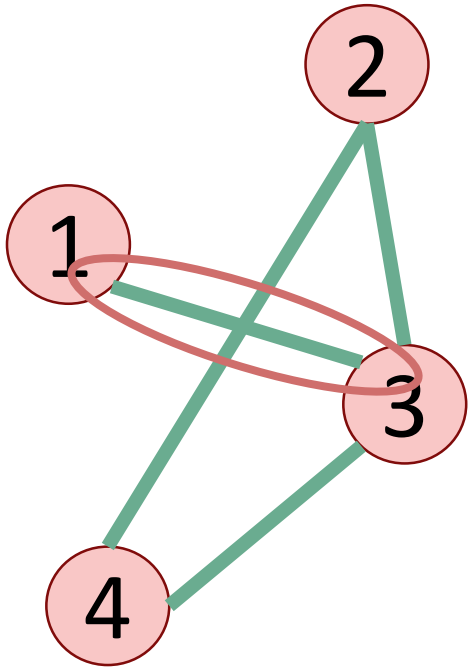


- The **in-degree** of vertex 4 is 2.
- The **out-degree** of vertex 4 is 1.
- Vertex 4's **incoming neighbors** are 2,3
- Vertex 4's **outgoing neighbor** is 3.

How do we represent graphs?

- Option 1: **adjacency matrix**

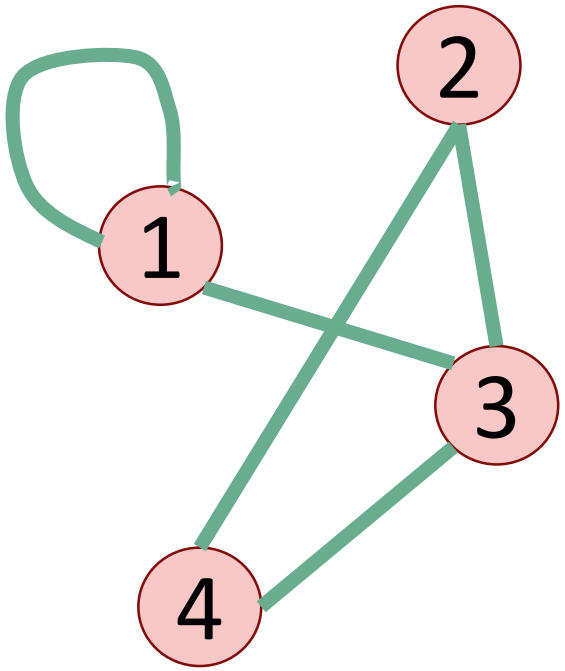
	1	2	3	4
1	0	0	1	0
2	0	0	1	1
3	1	1	0	1
4	0	1	1	0



How do we represent graphs?

- Option 1: **adjacency matrix**

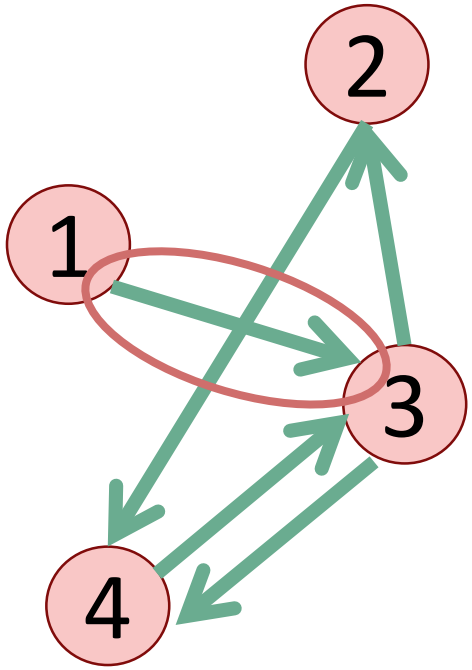
	1	2	3	4
1	1	0	1	0
2	0	0	1	1
3	1	1	0	1
4	0	1	1	0



How do we represent graphs?

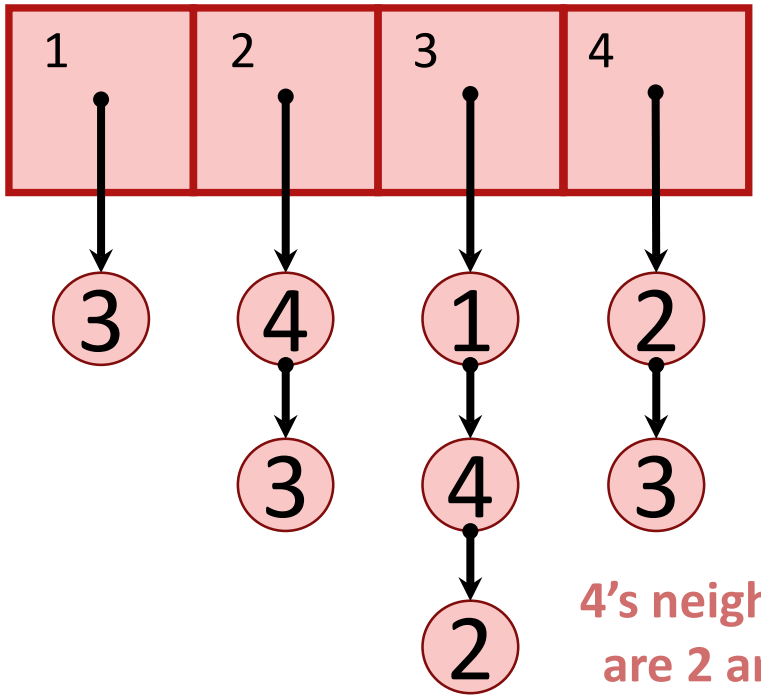
- Option 1: **adjacency matrix**

Source \ Destination	1	2	3	4
1	0	0	1	0
2	0	0	1	1
3	0	1	0	1
4	0	1	1	0

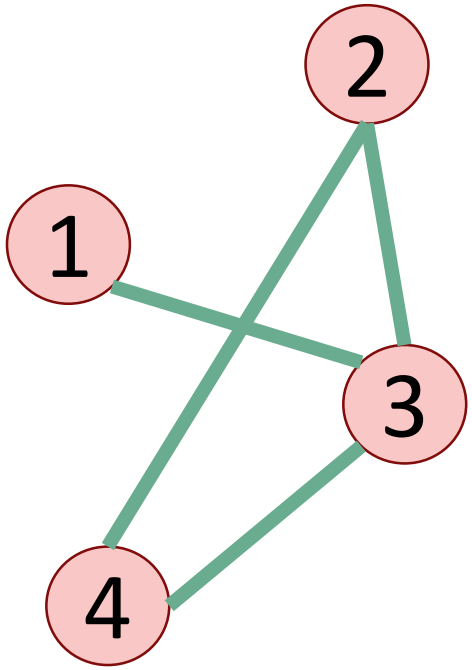


How do we represent graphs?

- Option 2: **adjacency lists.**



4's neighbors are 2 and 3



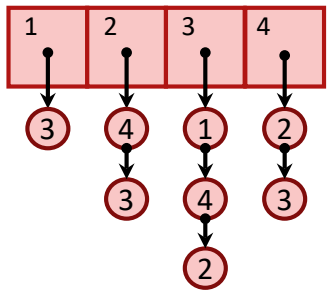
In either case

- Vertices can store other information
  - Attributes (name, IP address, …)
  - Helper information for algorithms that we will perform on the graph

- Want to be able to do the following operations:
  - **Edge Membership**: Is edge e in E?
  - **Neighbor Query**: What are the neighbors of vertex v?

Trade-offs

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

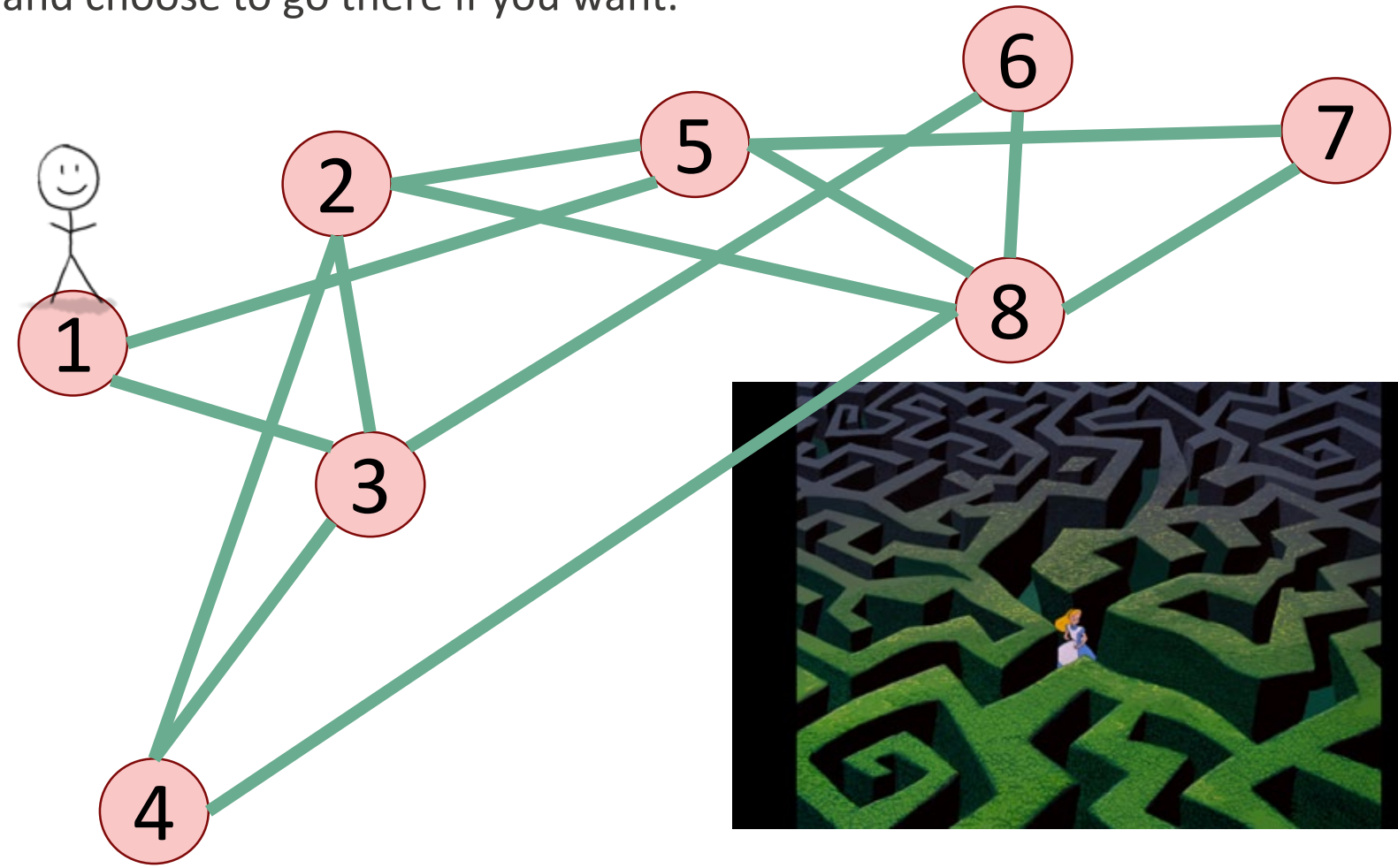
Say there are **n vertices** and **m edges.**	$\begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$	
Edge membership Is e = {v,w} in E?	O(1)	O(deg(v)) or O(deg(w))
Neighbor query Give me a list of v's neighbors.	O(n)	O(deg(v))
Space requirements	$O(n^2)$	O(n + m)

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

Depth First Search

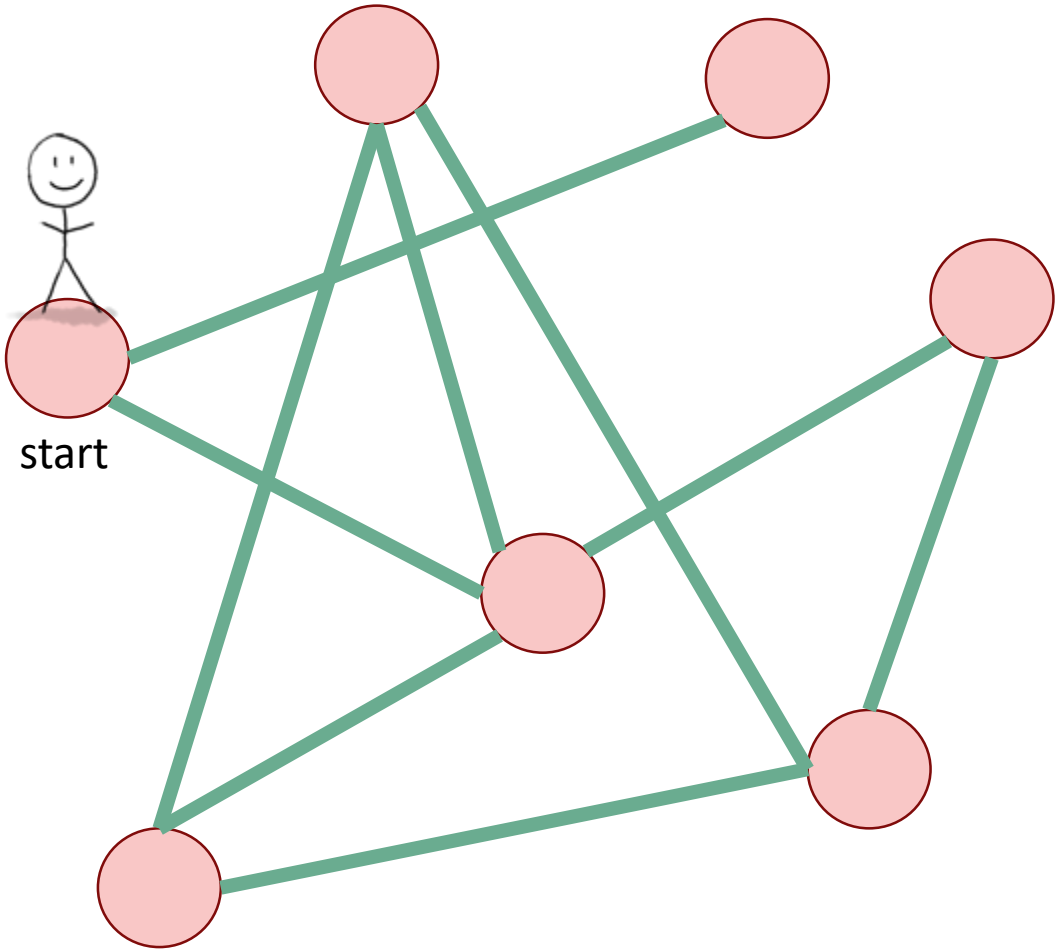
How do we explore a graph?

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



Depth First Search

- Searching “deeper” in the graph whenever possible.



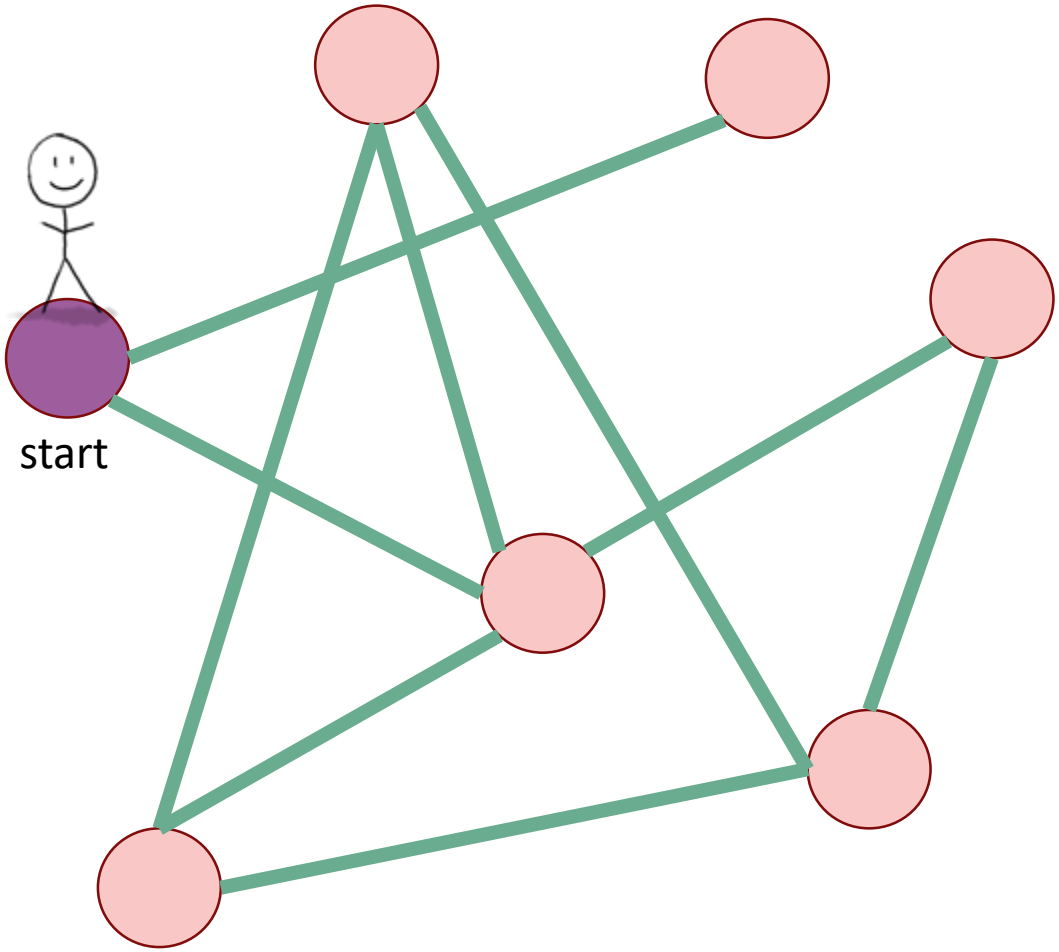
Not been there yet

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

Been there, have explored all the paths out.

Depth First Search

- Searching “deeper” in the graph whenever possible.



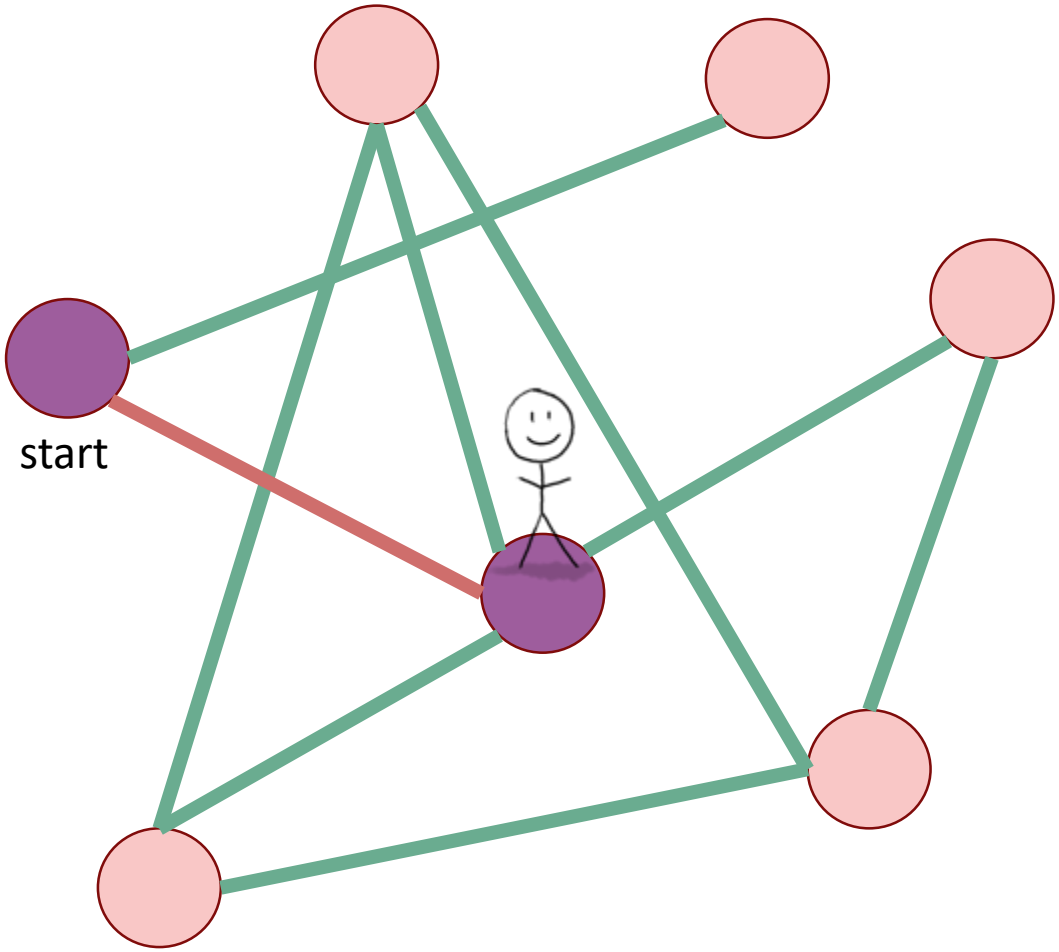
Not been there yet

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

Been there, have explored all the paths out.

Depth First Search

- Searching “deeper” in the graph whenever possible.



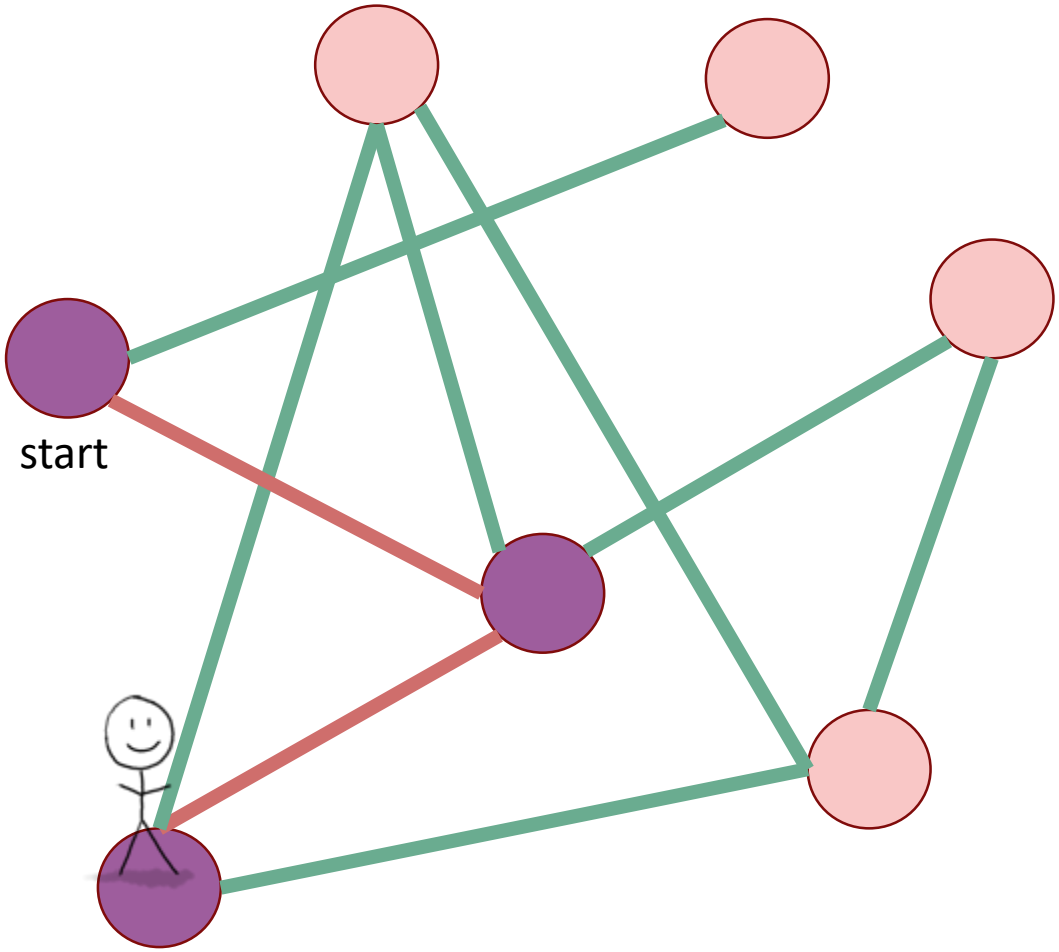
Not been there yet

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

Been there, have explored all the paths out.

Depth First Search

- Searching “deeper” in the graph whenever possible.



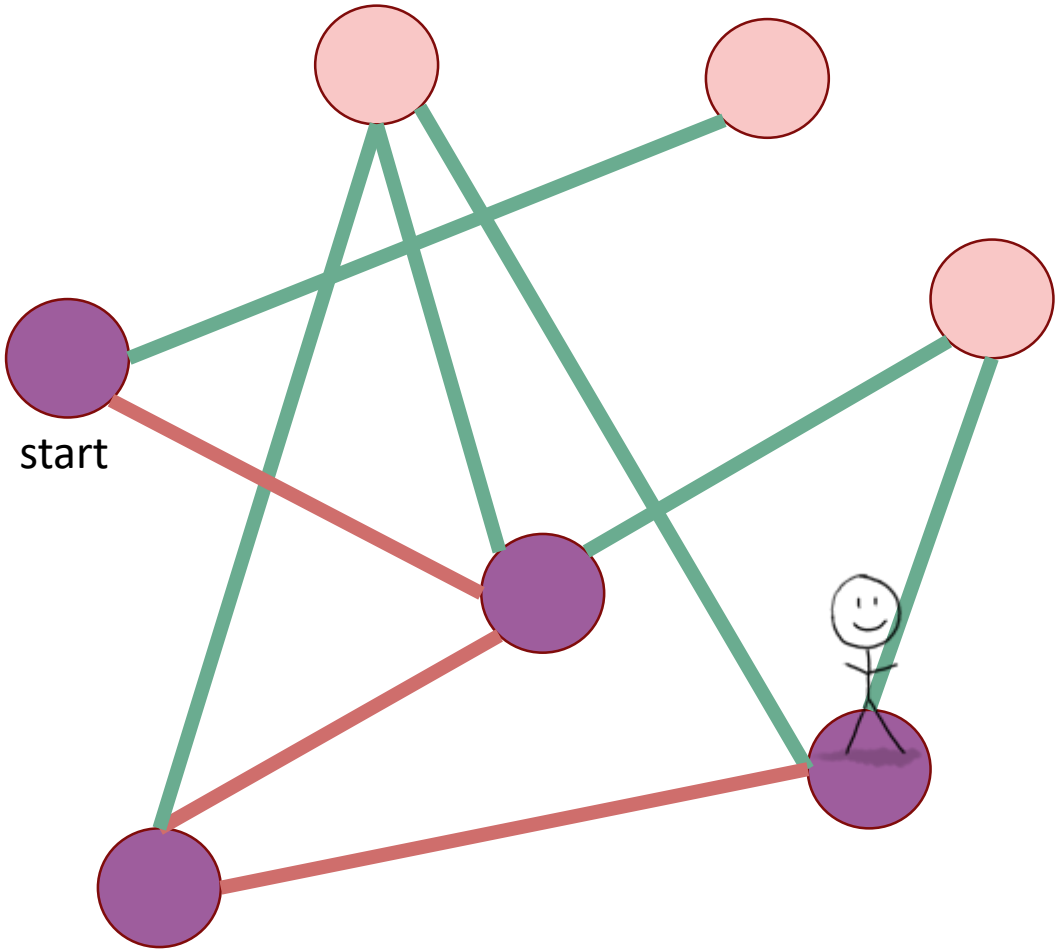
Not been there yet

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

Been there, have explored all the paths out.

Depth First Search

- Searching “deeper” in the graph whenever possible.



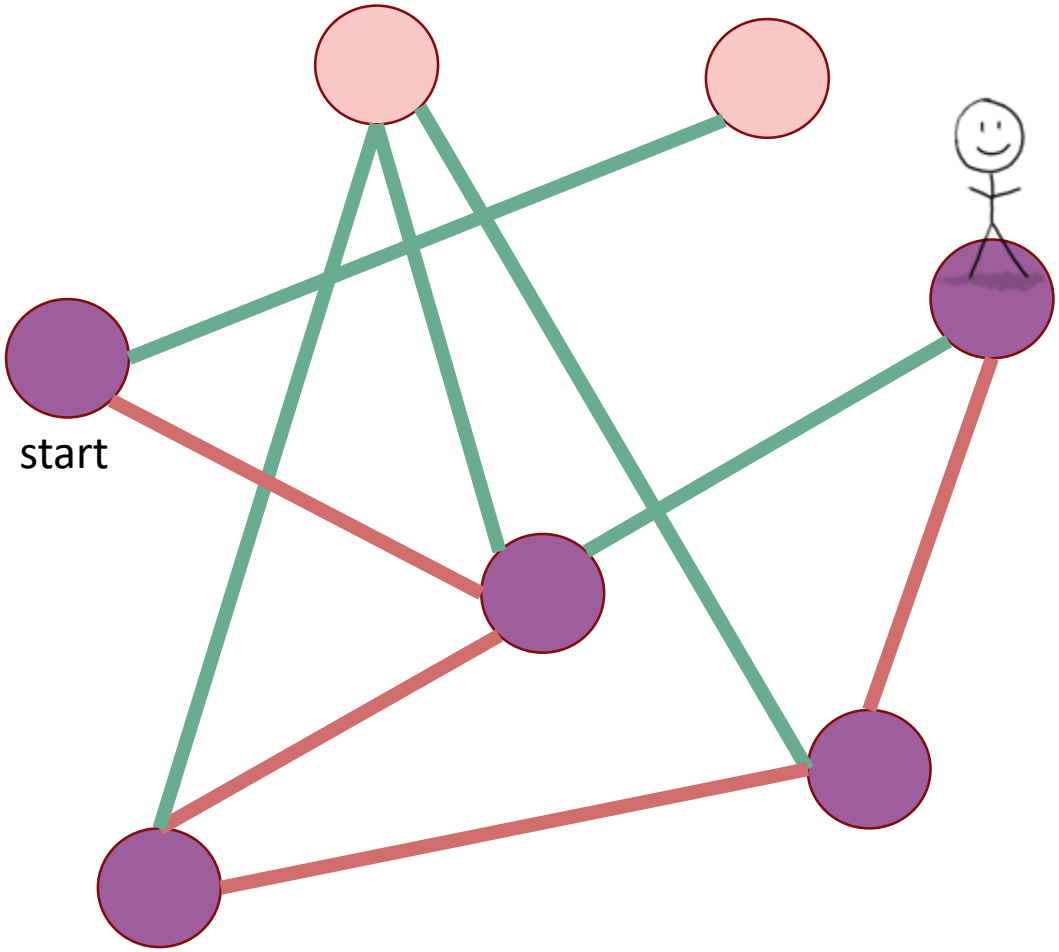
Not been there yet

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

Been there, have explored all the paths out.

Depth First Search

- Searching “deeper” in the graph whenever possible.



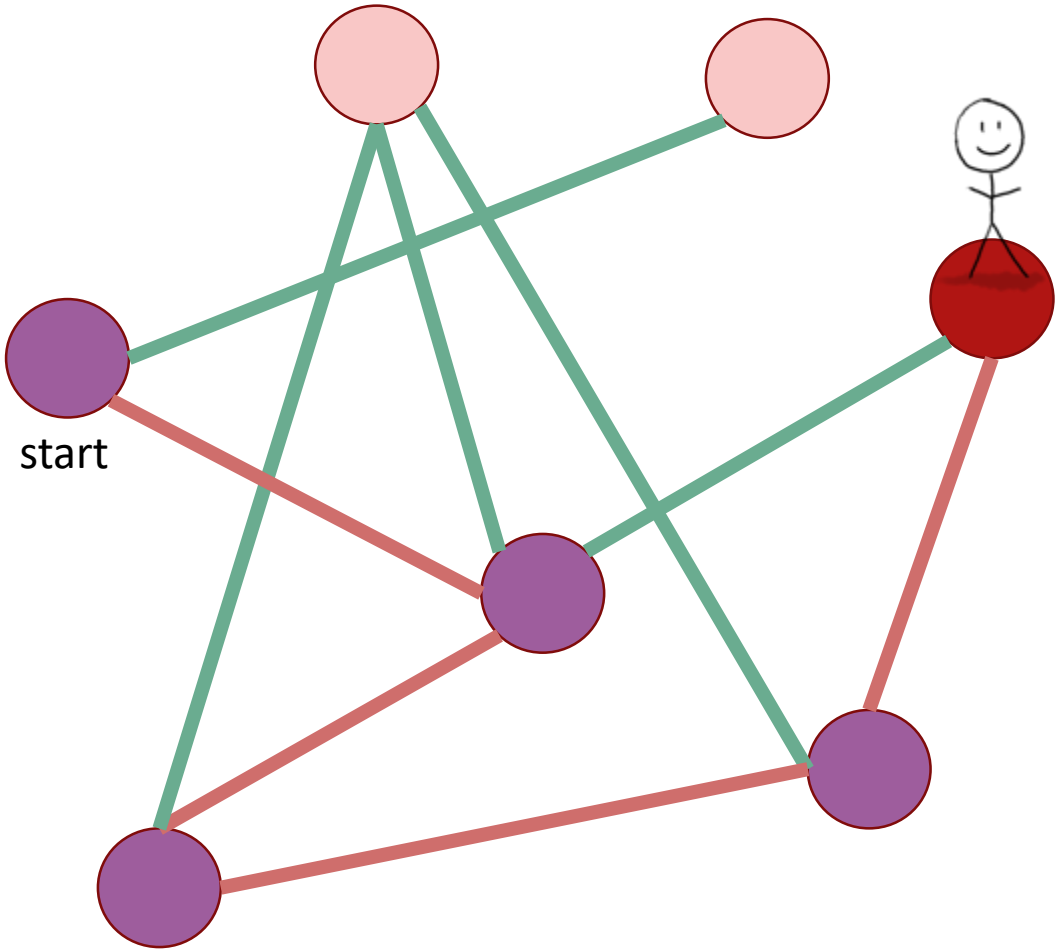
Not been there yet

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

Been there, have explored all the paths out.

Depth First Search

- Searching “deeper” in the graph whenever possible.



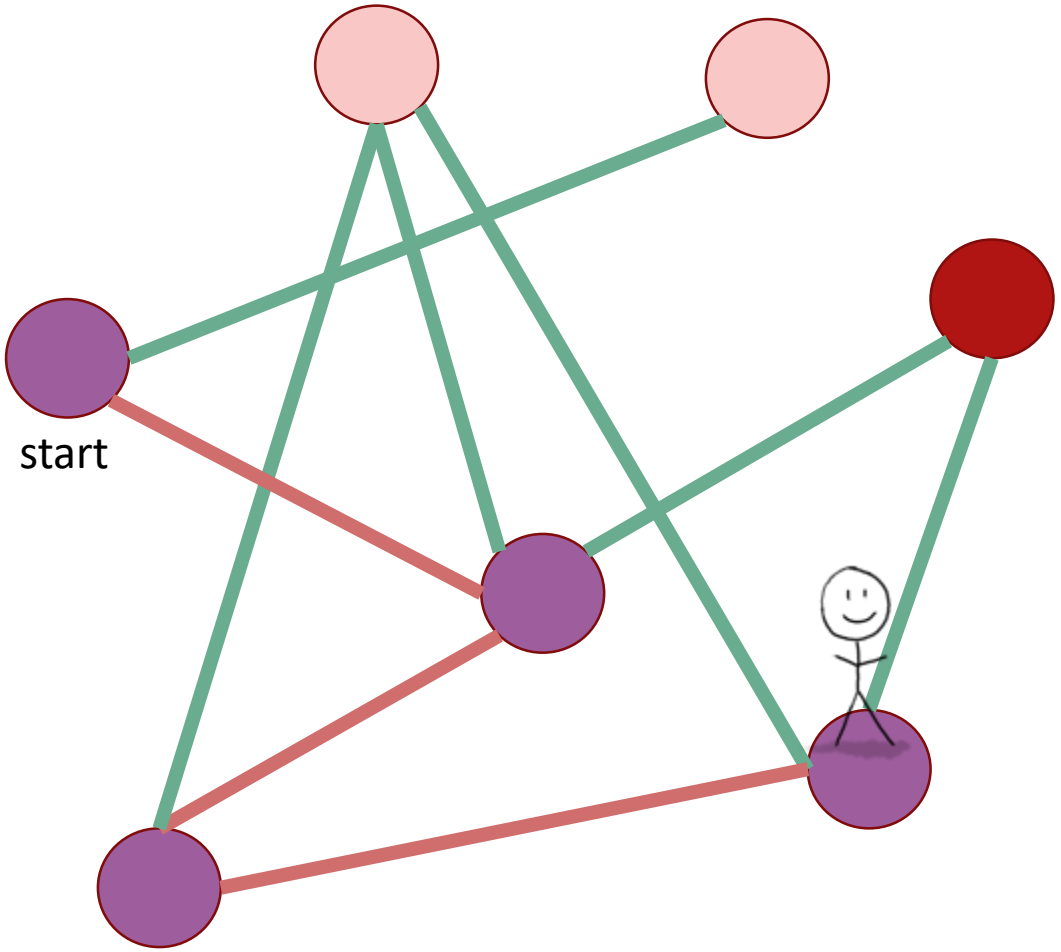
Not been there yet

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

Been there, have explored all the paths out.

Depth First Search

- Searching “deeper” in the graph whenever possible.



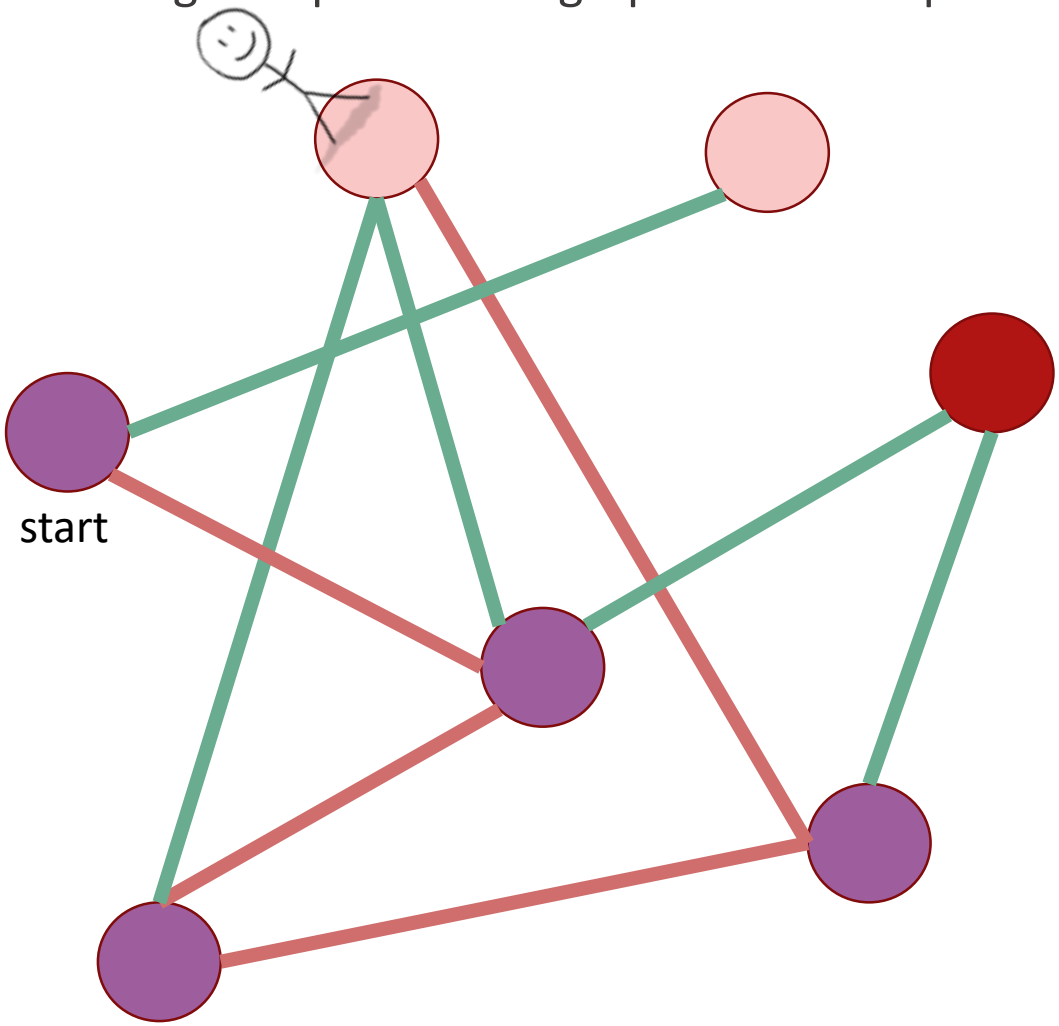
Not been there yet

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

Been there, have explored all the paths out.

Depth First Search

- Searching “deeper” in the graph whenever possible.



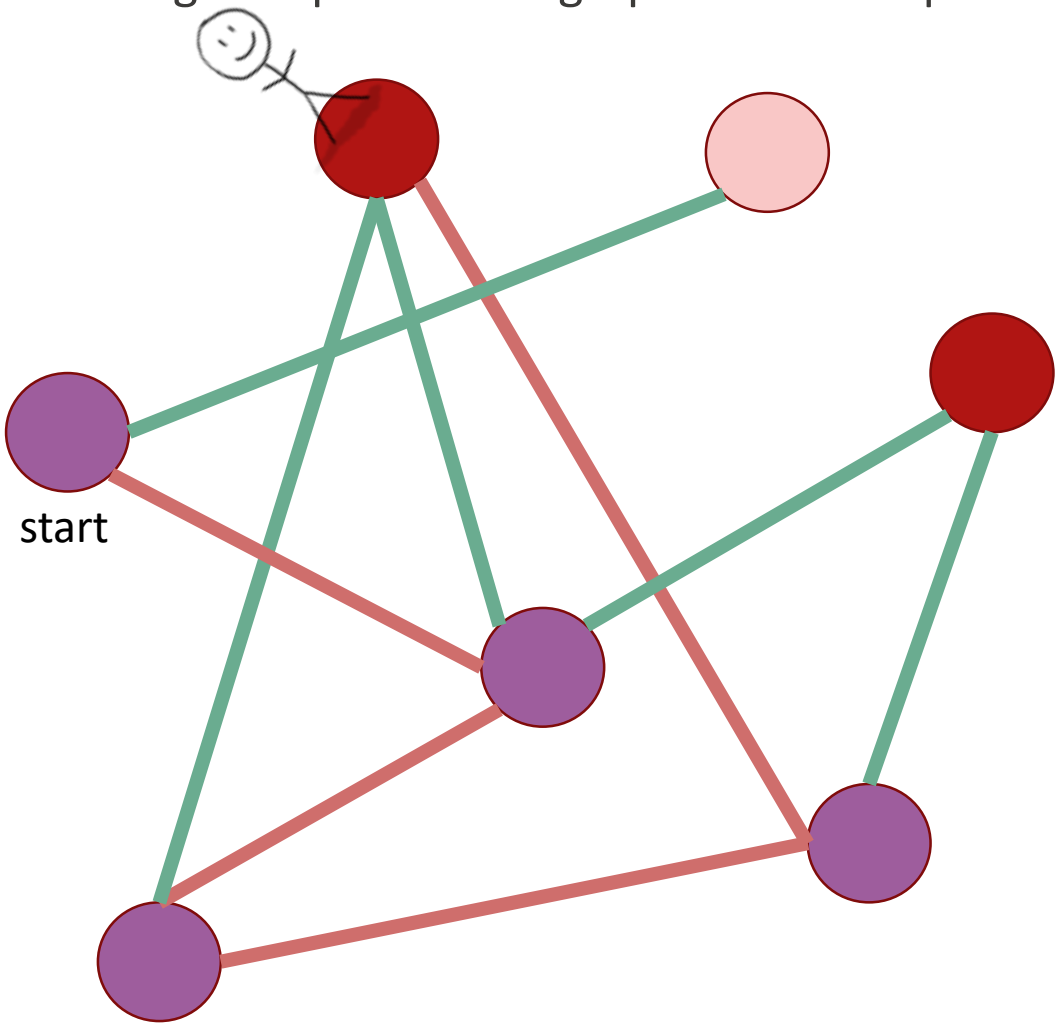
Not been there yet

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

Been there, have explored all the paths out.

Depth First Search

- Searching “deeper” in the graph whenever possible.



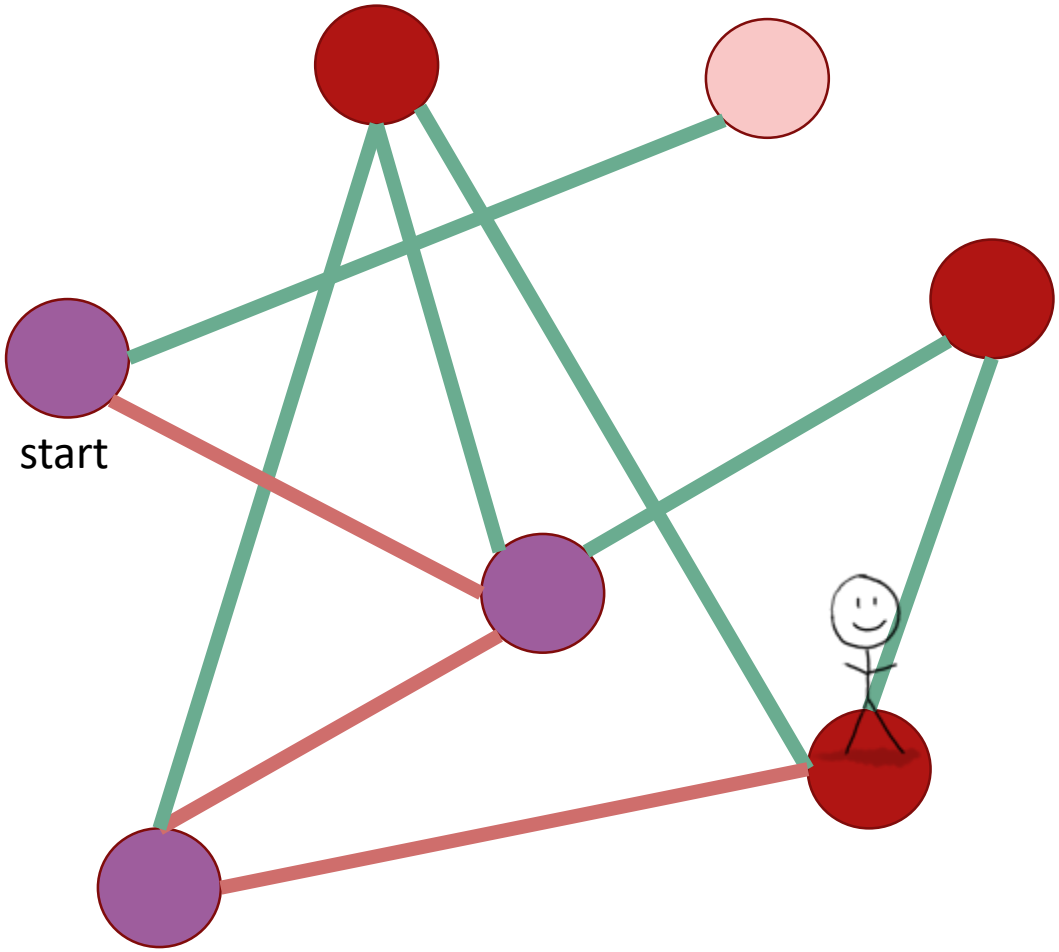
Not been there yet

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

Been there, have explored all the paths out.

Depth First Search

- Searching “deeper” in the graph whenever possible.



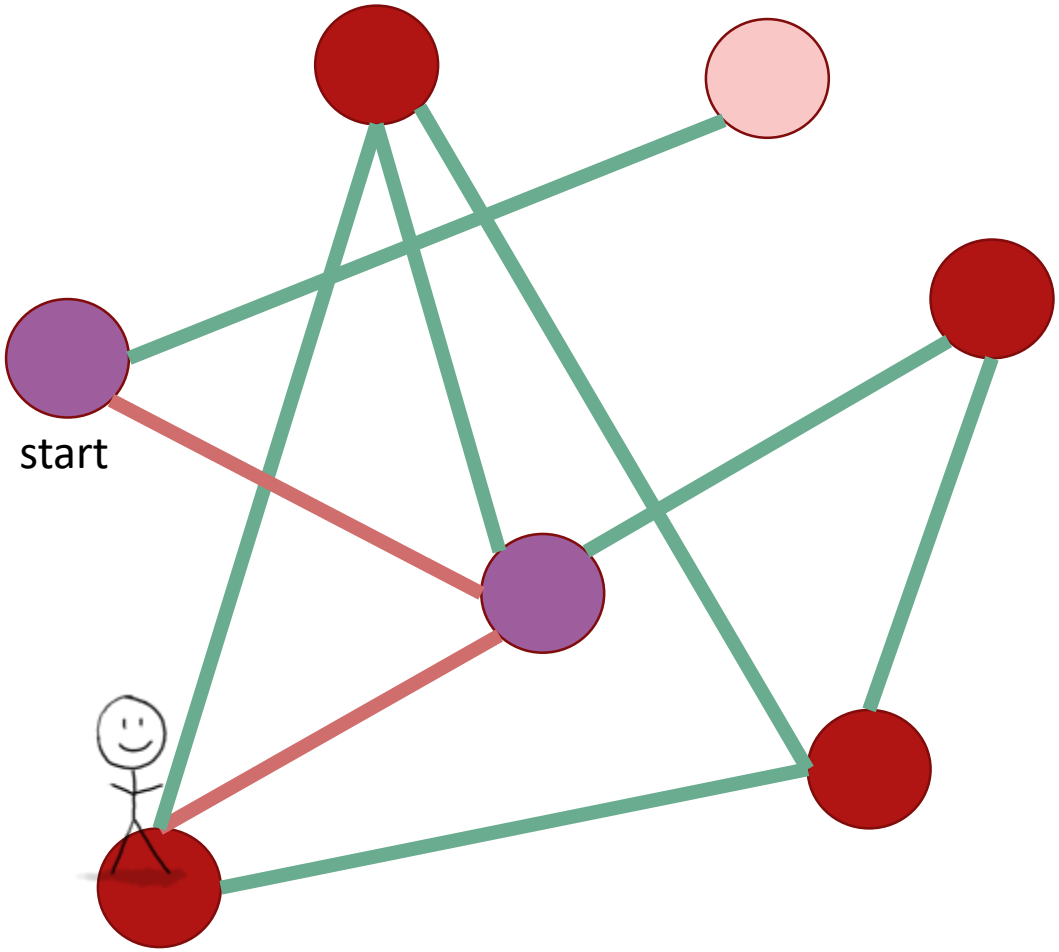
Not been there yet

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

Been there, have explored all the paths out.

Depth First Search

- Searching “deeper” in the graph whenever possible.



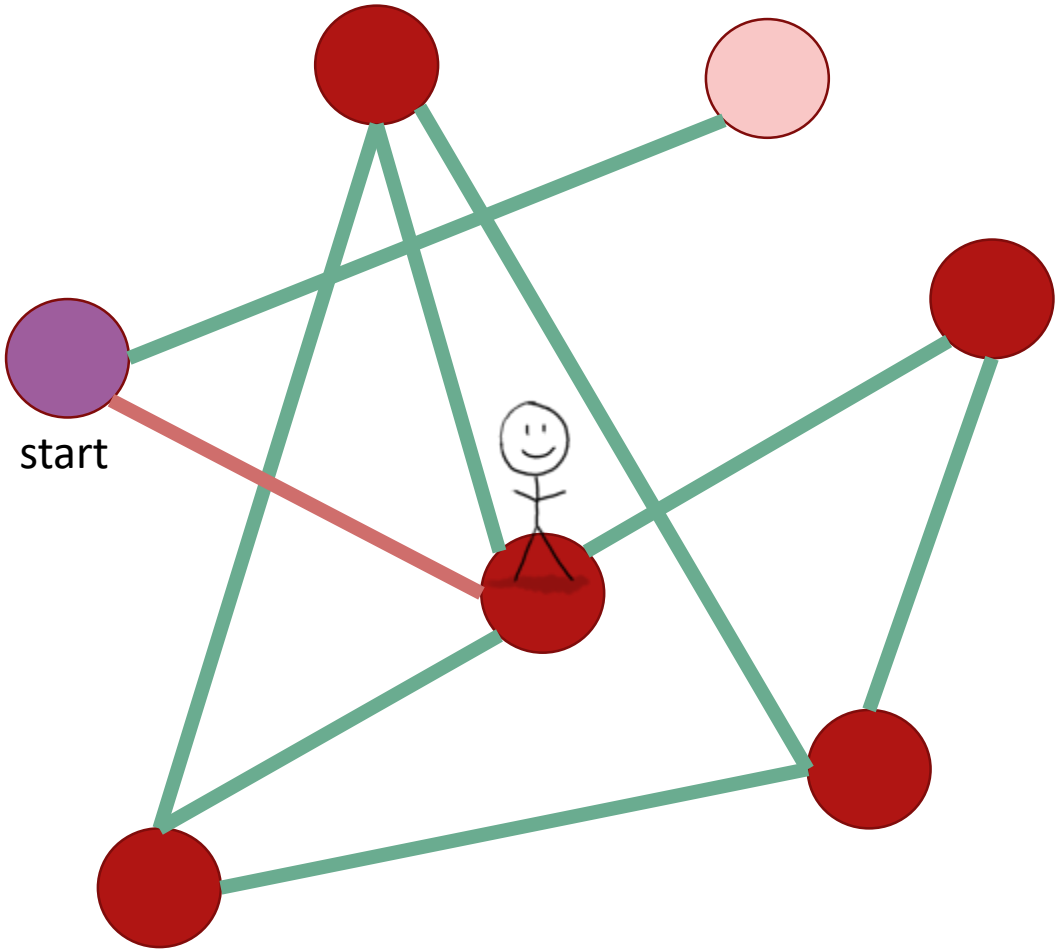
Not been there yet

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

Been there, have explored all the paths out.

Depth First Search

- Searching “deeper” in the graph whenever possible.



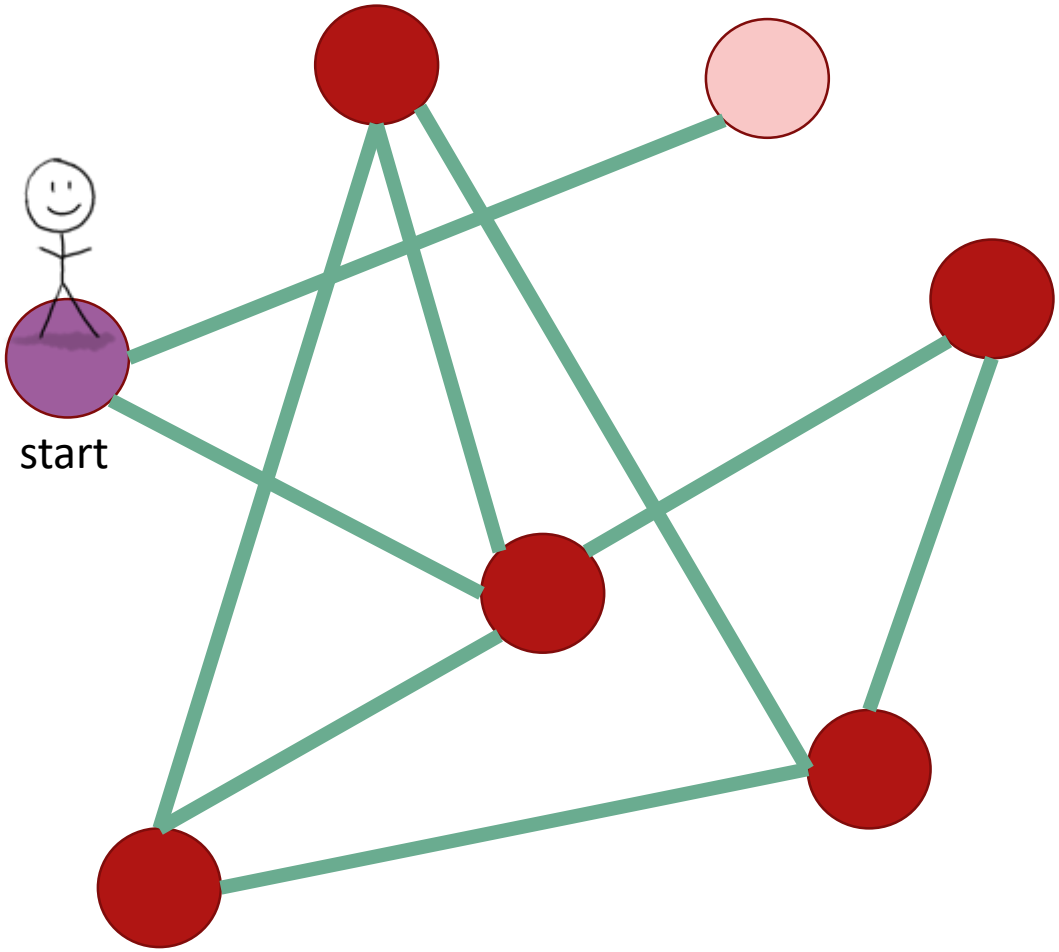
Not been there yet

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

Been there, have explored all the paths out.

Depth First Search

- Searching “deeper” in the graph whenever possible.



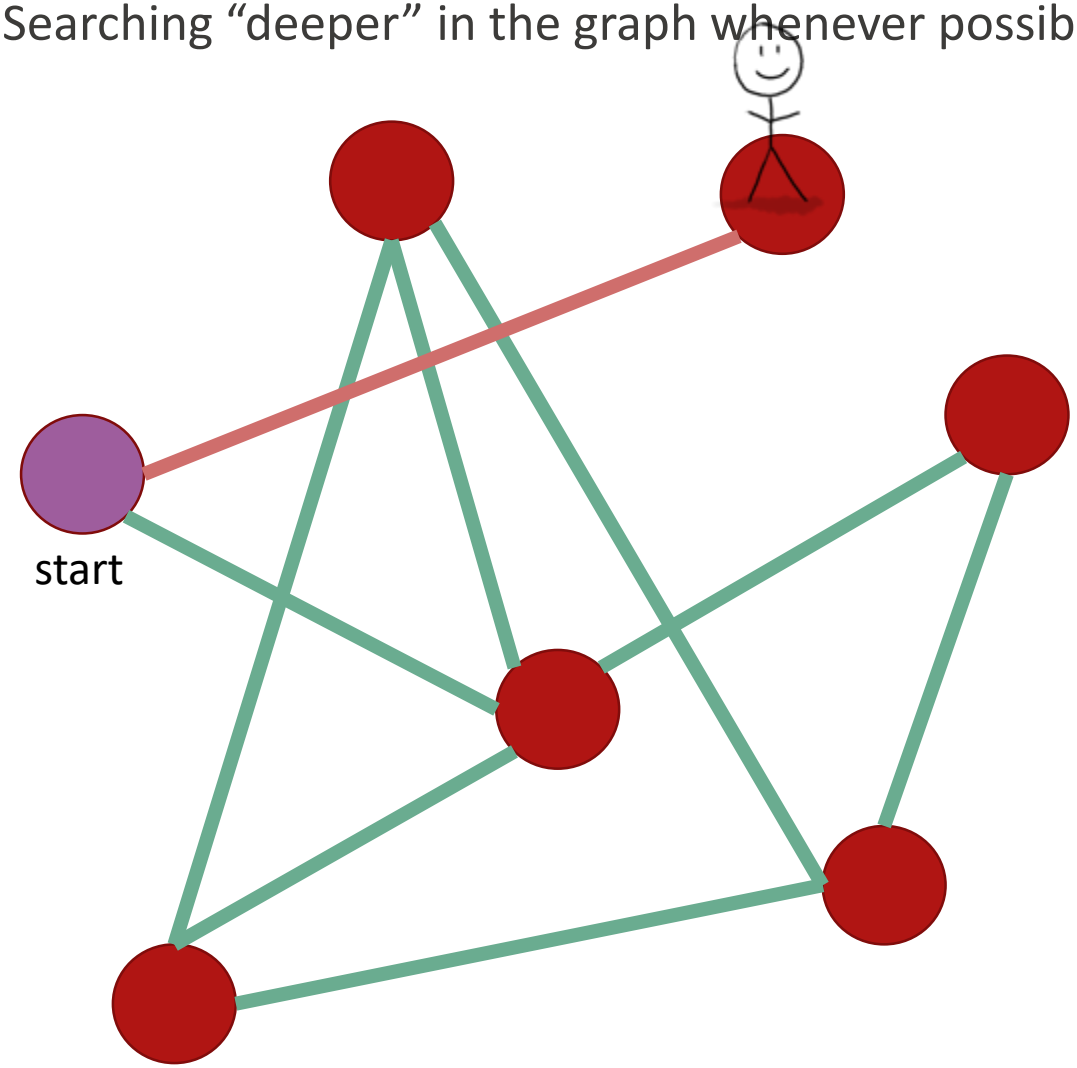
Not been there yet

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

Been there, have explored all the paths out.

Depth First Search

- Searching “deeper” in the graph whenever possible.



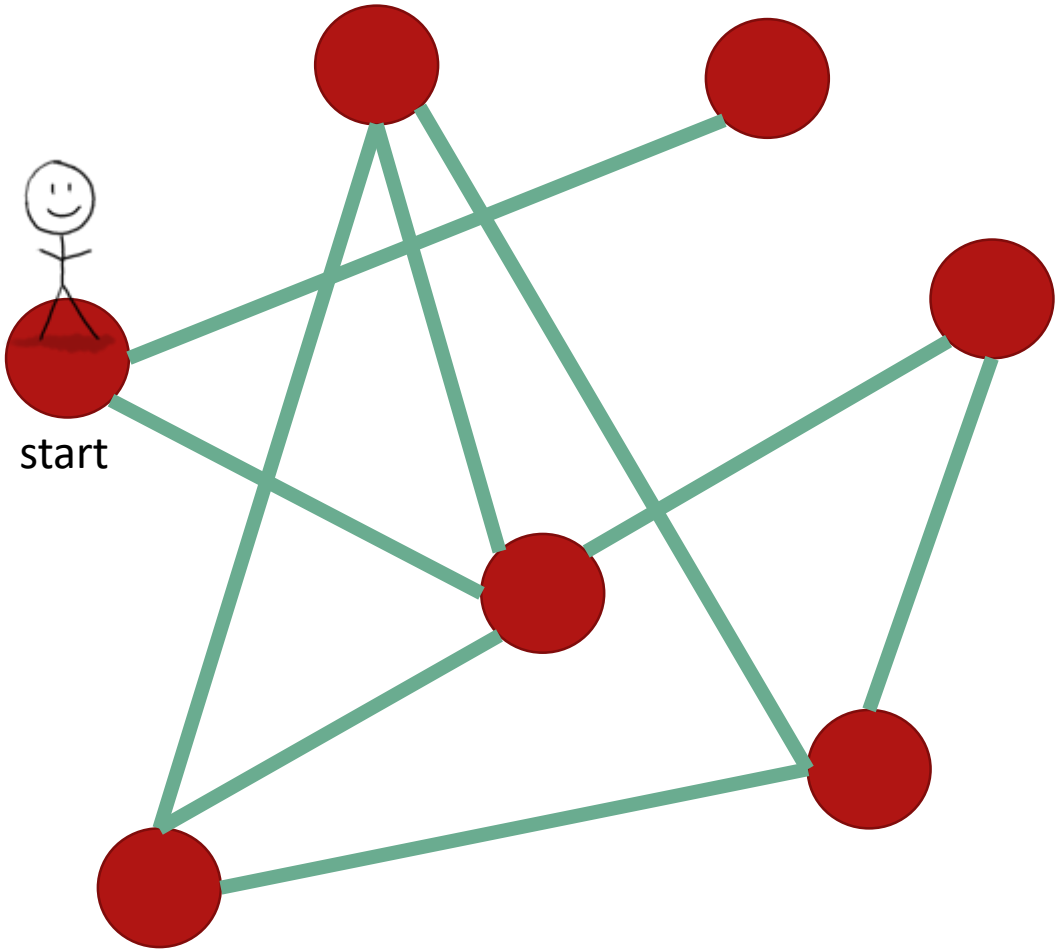
Not been there yet

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

Been there, have explored all the paths out.

Depth First Search

- Searching “deeper” in the graph whenever possible.



explored!

Not been there yet

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

Been there, have explored all the paths out.

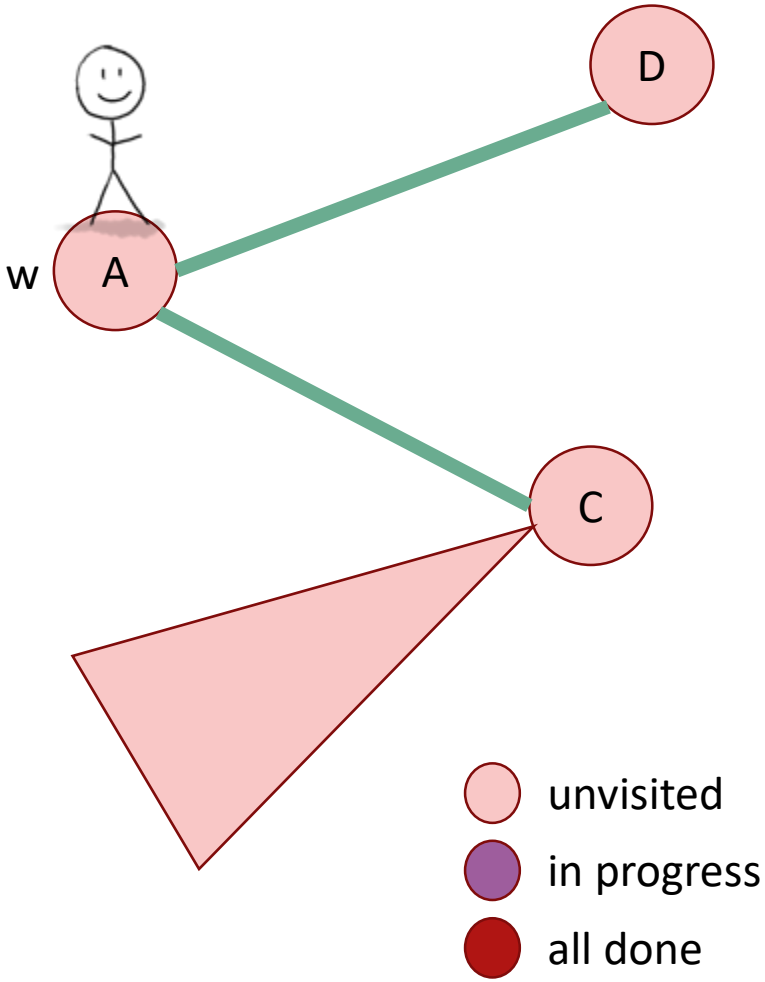
Depth First Search

- Searching “deeper” in the graph whenever possible.
- 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**.
  - Note:
    - This is not the only way to write DFS.
    - This way has more bookkeeping. (It will be useful later!)

Depth First Search

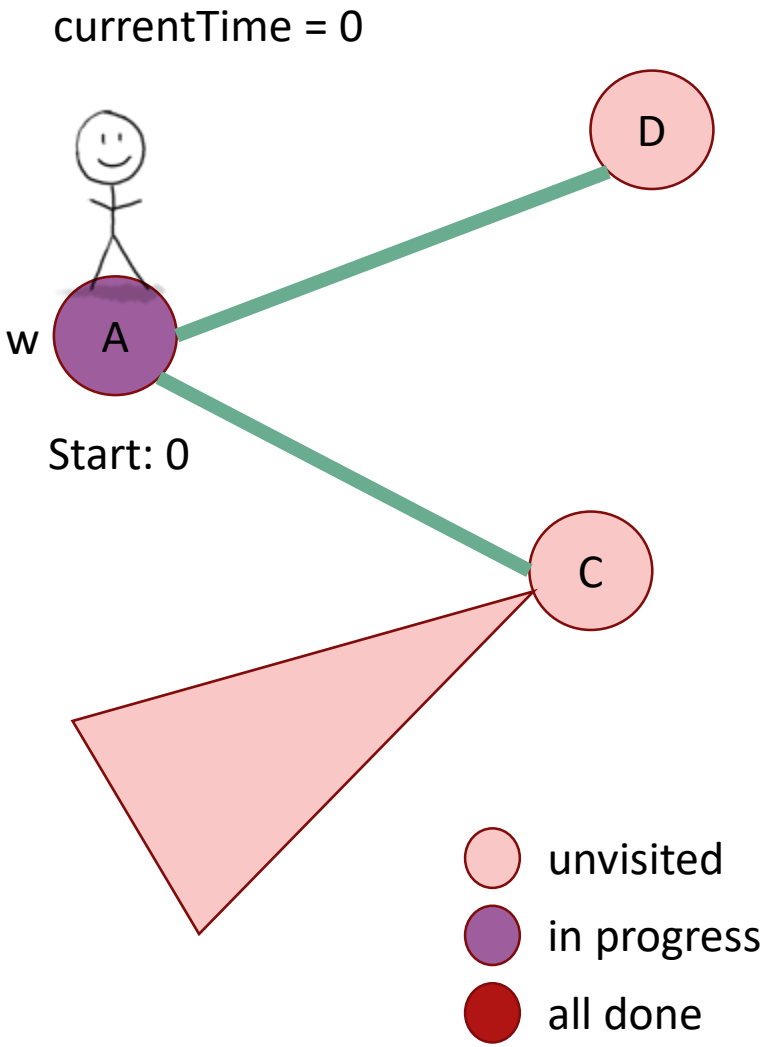
currentTime = 0



w: input vertex

```
DFS(w, currentTime):
    w.startTime = currentTime
    currentTime += 1
    Mark w as in progress
    for v in w.neighbors:
        if v is unvisited:
            currentTime =
                DFS(v, currentTime)
            currentTime += 1
    w.finishTime = currentTime
    Mark w as all done
    return currentTime
```

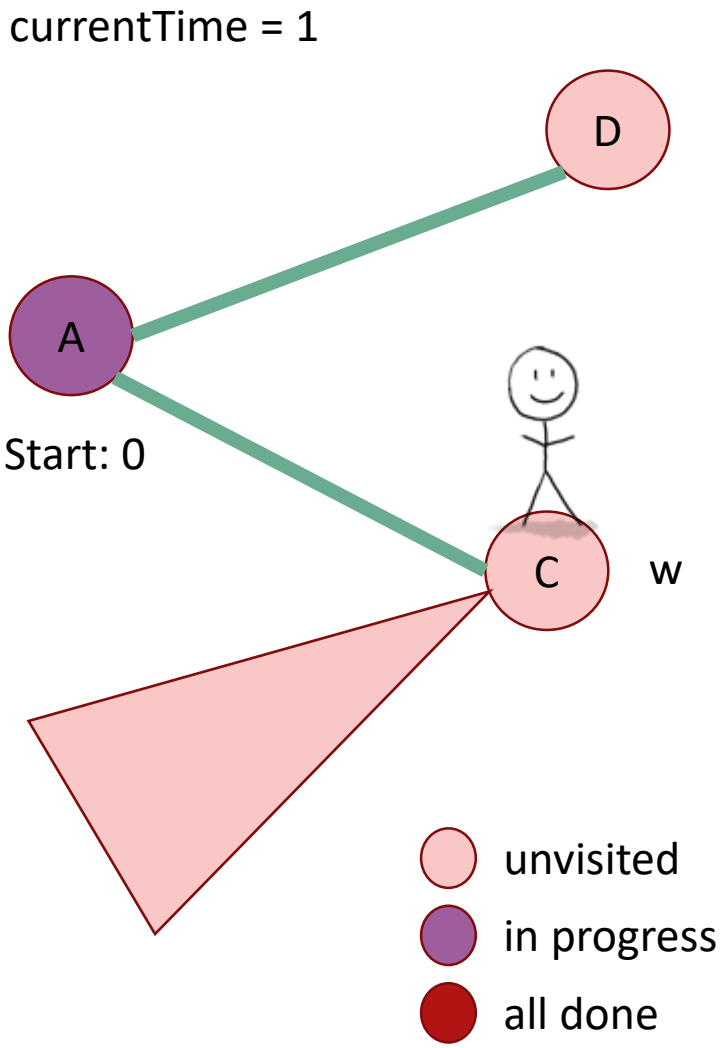
Depth First Search



w: input vertex

```
DFS(w, currentTime):
    w.startTime = currentTime
    currentTime += 1
    Mark w as in progress
    for v in w.neighbors:
        if v is unvisited:
            currentTime =
                DFS(v, currentTime)
            currentTime += 1
    w.finishTime = currentTime
    Mark w as all done
    return currentTime
```

Depth First Search

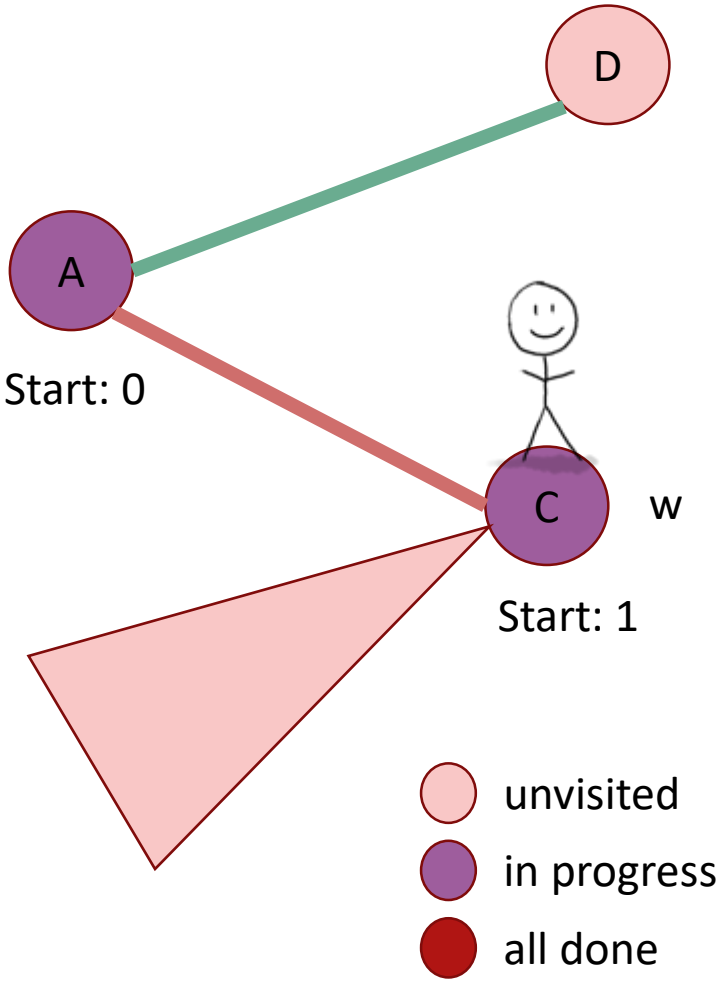


w: input vertex

```
DFS(w, currentTime):
    w.startTime = currentTime
    currentTime += 1
    Mark w as in progress
    for v in w.neighbors:
        if v is unvisited:
            currentTime =
                DFS(v, currentTime)
            currentTime += 1
    w.finishTime = currentTime
    Mark w as all done
    return currentTime
```

Depth First Search

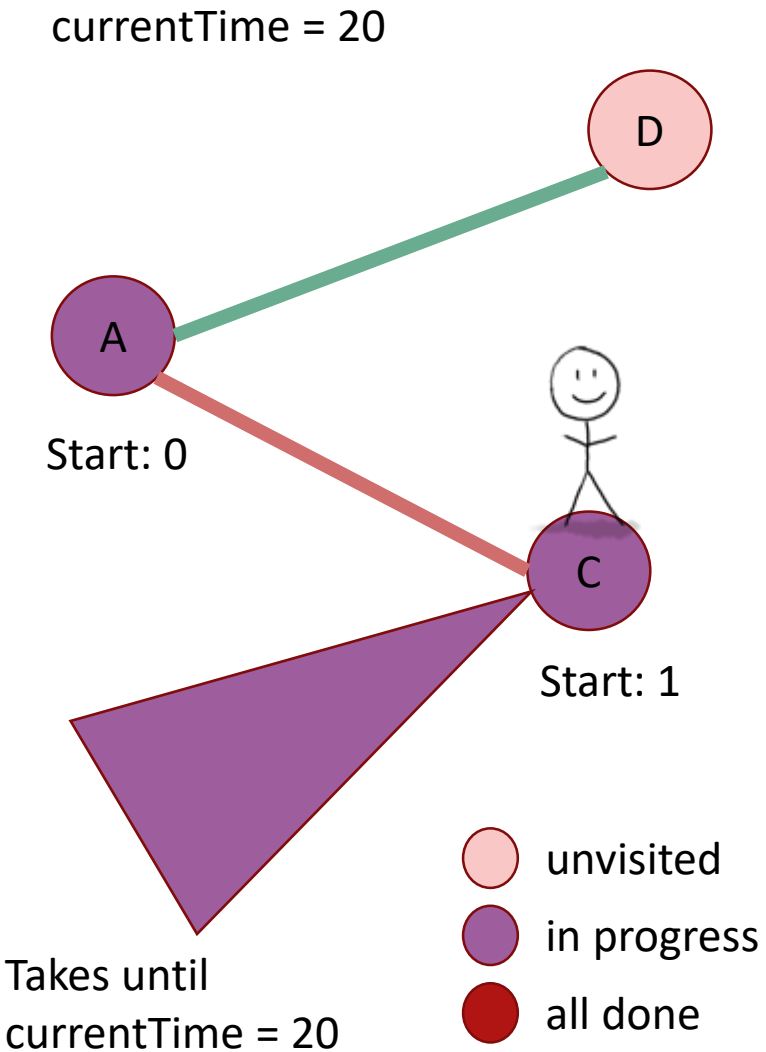
currentTime = 1



w: input vertex

```
DFS(w, currentTime):
    w.startTime = currentTime
    currentTime += 1
    Mark w as in progress
    for v in w.neighbors:
        if v is unvisited:
            currentTime =
                DFS(v, currentTime)
            currentTime += 1
    w.finishTime = currentTime
    Mark w as all done
    return currentTime
```

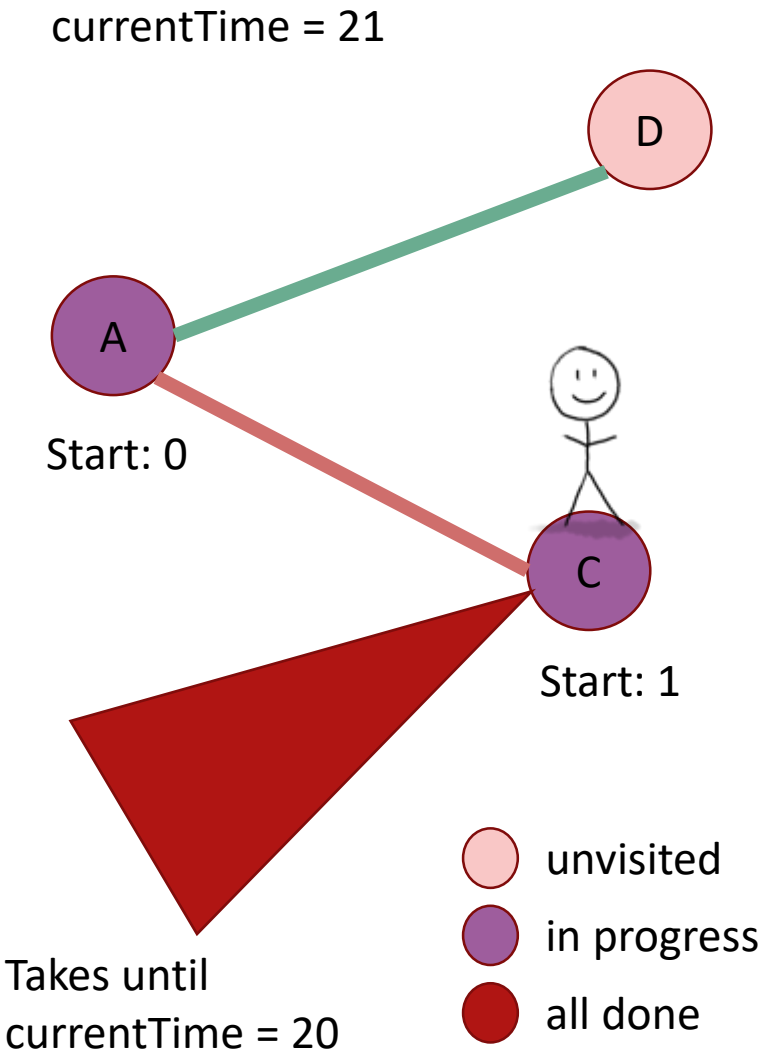
Depth First Search



w: input vertex

```
DFS(w, currentTime):
    w.startTime = currentTime
    currentTime += 1
    Mark w as in progress
    for v in w.neighbors:
        if v is unvisited:
            currentTime =
                DFS(v, currentTime)
            currentTime += 1
    w.finishTime = currentTime
    Mark w as all done
    return currentTime
```

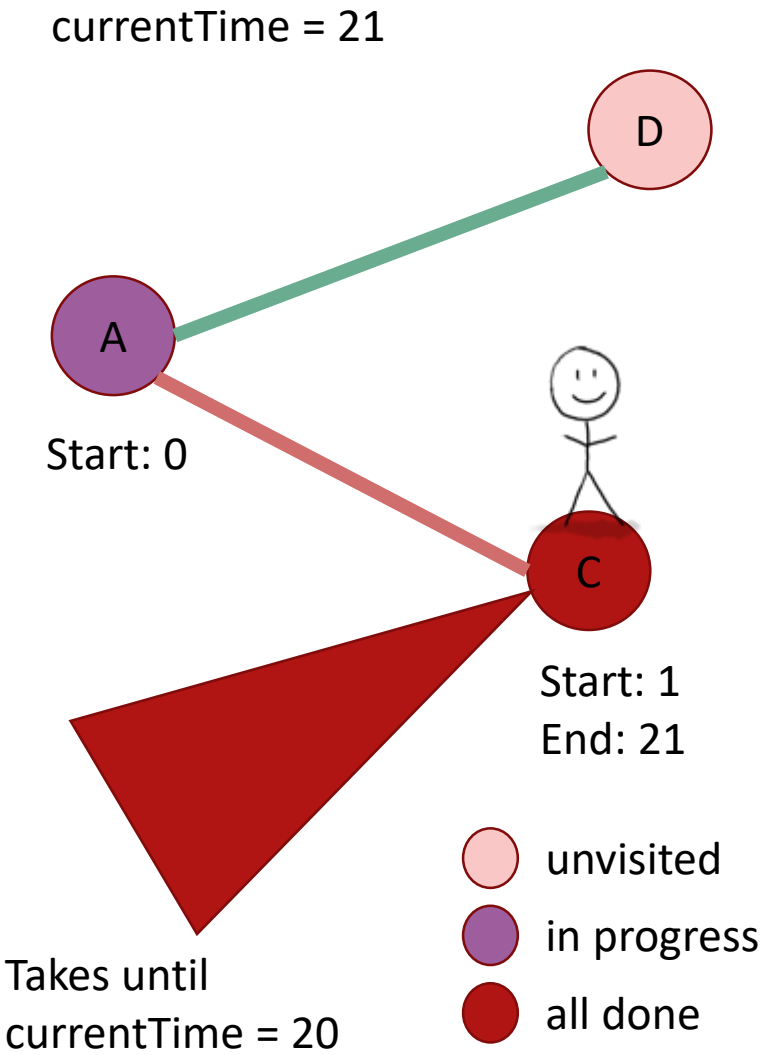
Depth First Search



w: input vertex

```
DFS(w, currentTime):
    w.startTime = currentTime
    currentTime += 1
    Mark w as in progress
    for v in w.neighbors:
        if v is unvisited:
            currentTime =
                DFS(v, currentTime)
            currentTime += 1
    w.finishTime = currentTime
    Mark w as all done
    return currentTime
```

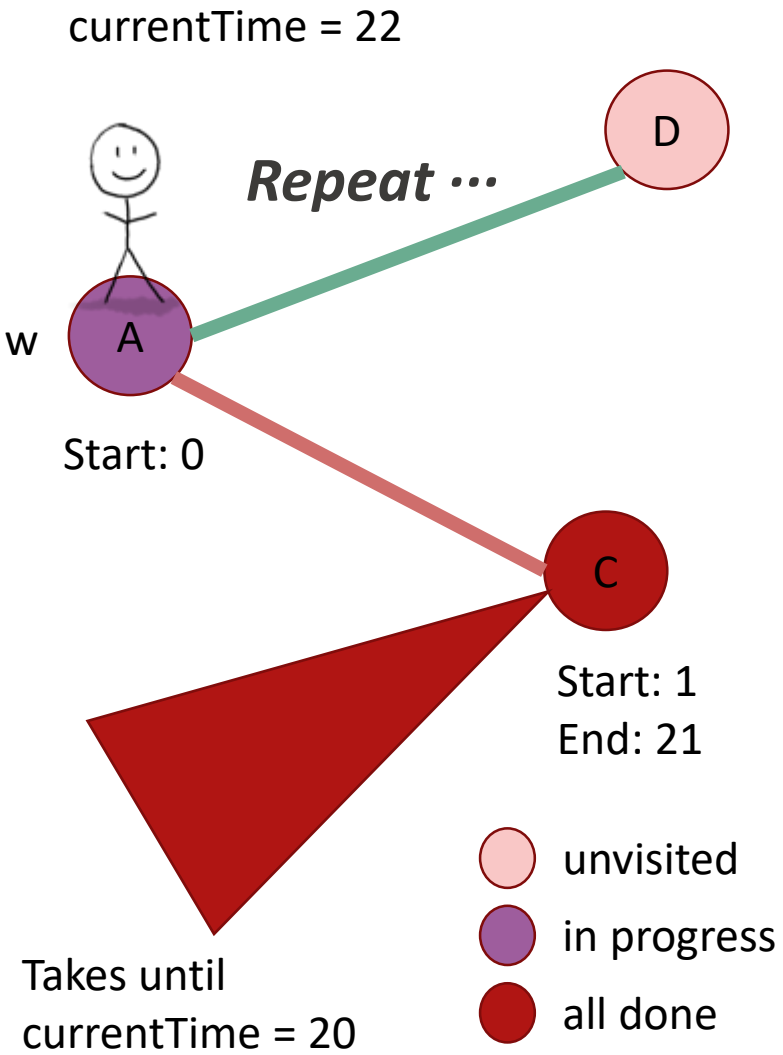
Depth First Search



w: input vertex

```
DFS(w, currentTime):
    w.startTime = currentTime
    currentTime += 1
    Mark w as in progress
    for v in w.neighbors:
        if v is unvisited:
            currentTime =
                DFS(v, currentTime)
            currentTime += 1
    w.finishTime = currentTime
    Mark w as all done
    return currentTime
```

Depth First Search

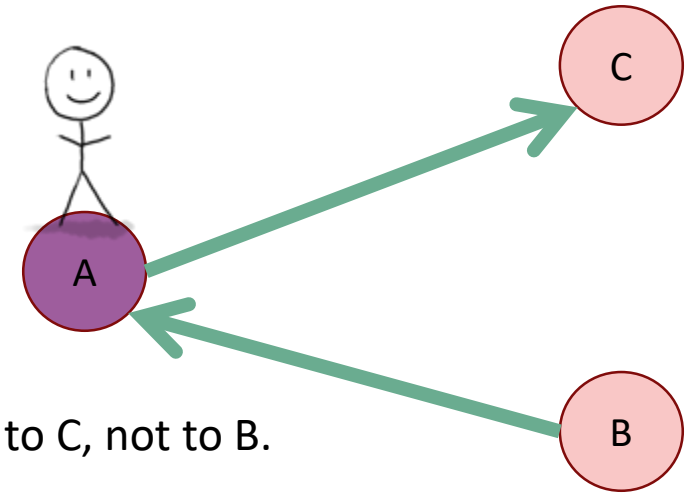


w: input vertex

```
DFS(w, currentTime):
    w.startTime = currentTime
    currentTime += 1
    Mark w as in progress
    for v in w.neighbors:
        if v is unvisited:
            currentTime =
                DFS(v, currentTime)
            currentTime += 1
    w.finishTime = currentTime
    Mark w as all done
    return currentTime
```

You check:

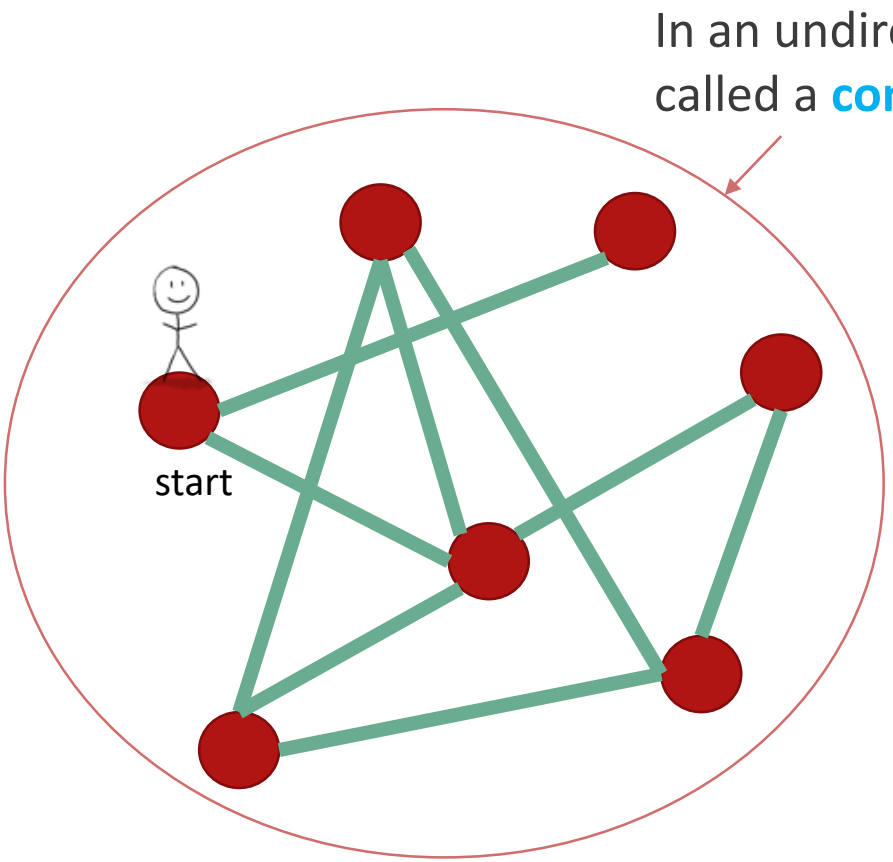
- DFS works fine on directed graphs too!



Only walk to C, not to B.

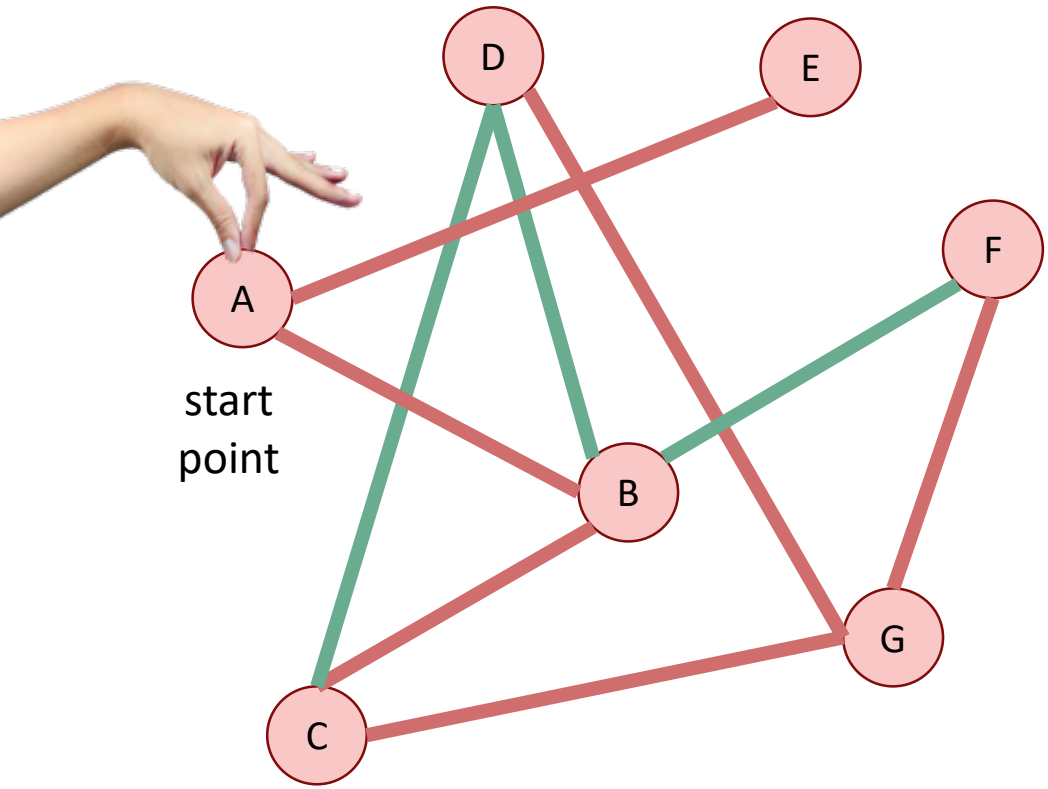
Depth First Search

- DFS finds all the nodes reachable from the starting point

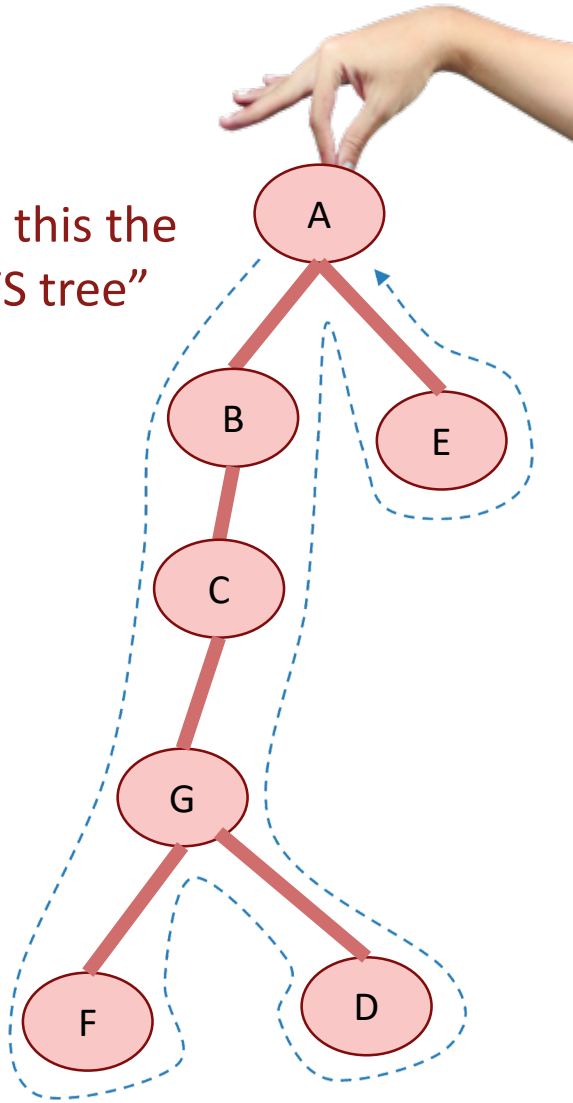


Why is it called depth-first?

- We are implicitly building a tree:



Call this the
"DFS tree"



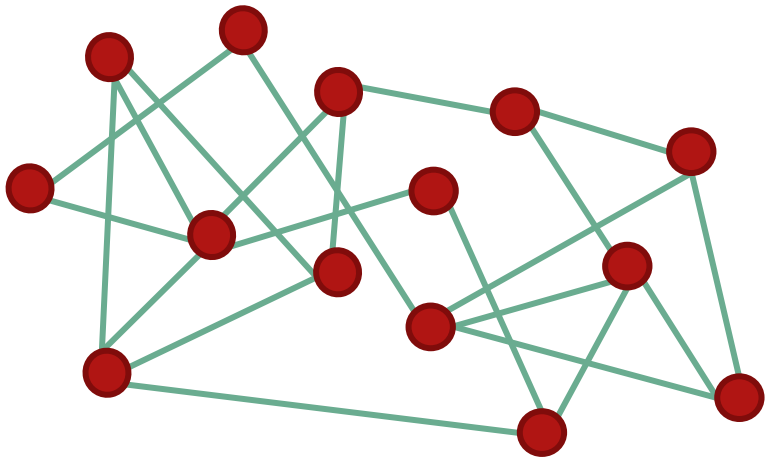
We first go as
deep as we can.

Running time

- **To explore just the connected component we started in**
  - We look at each edge at most twice.
    - Once from each of its endpoints
  - And basically, we don't do anything else.
  - So, **O(m)**

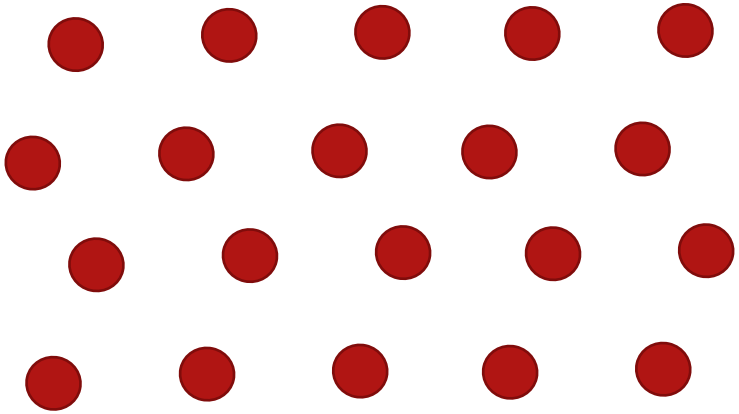
Running time

- **To explore the whole graph**
  - Explore the connected components one-by-one.
  - This takes time **O(n + m)**



Here the running time is O(m) like before

or



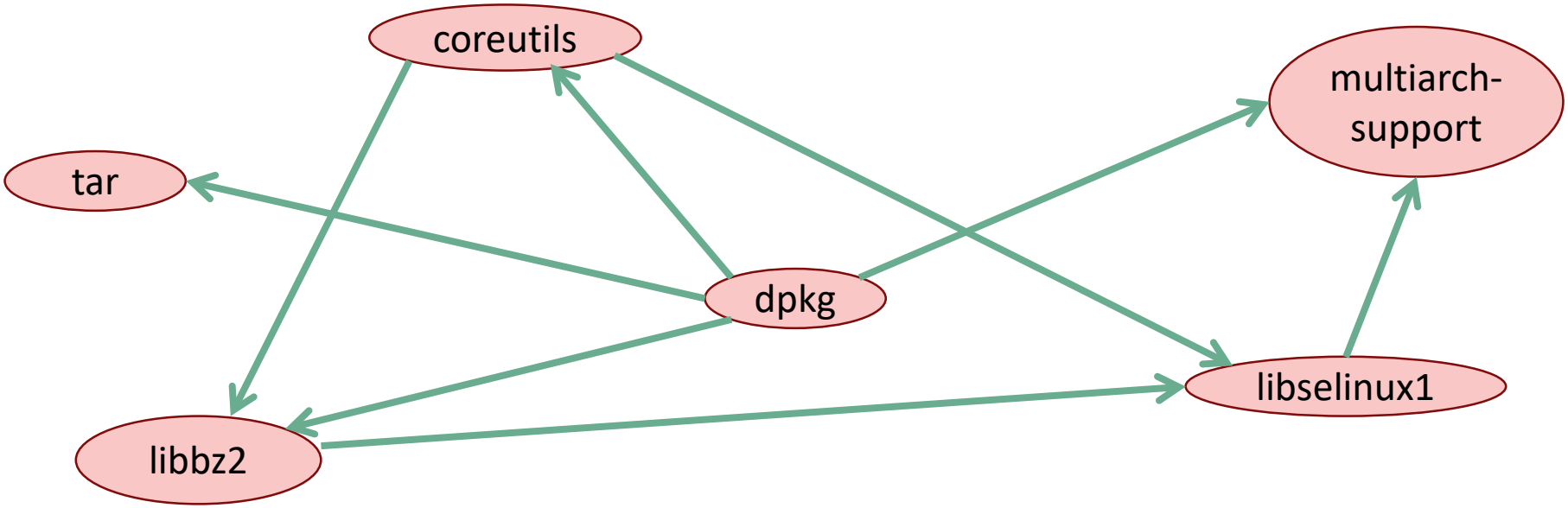
Here m=0 but it still takes time O(n) to explore the graph.

Topological Ordering

Topological Ordering

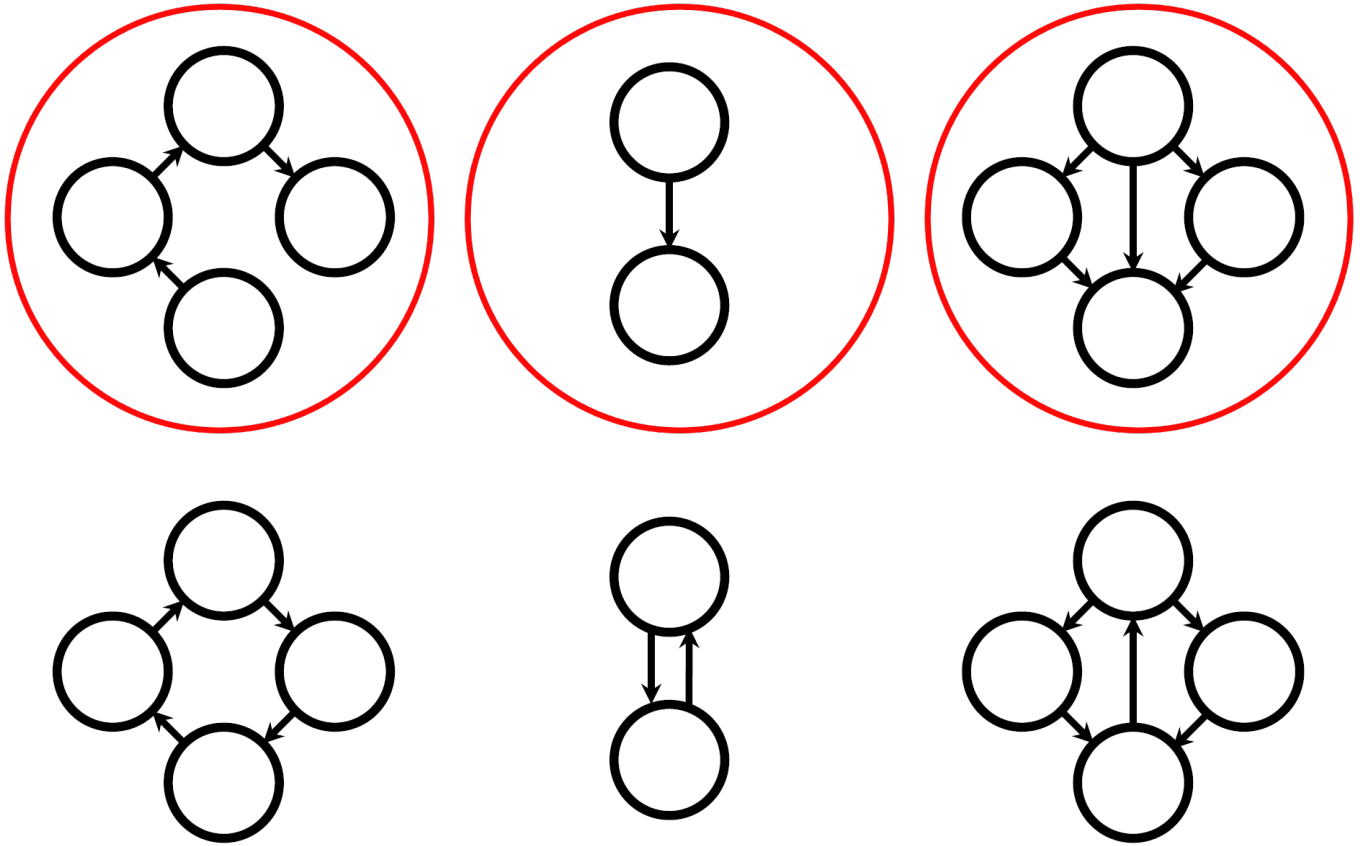
- Q. Given a package dependency graph, how do you install packages in the correct order?

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



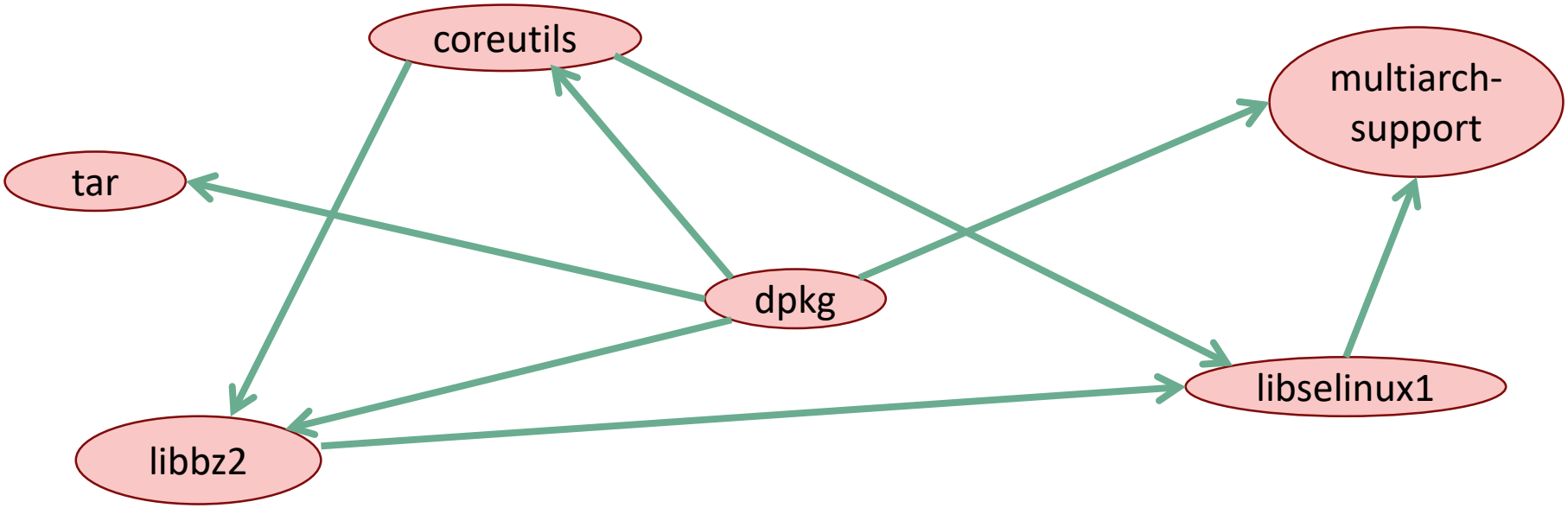
Wait.. What is a DAG?

- DAG is a **directed graph with no directed cycles**.
- Which of these graphs are valid DAGs?

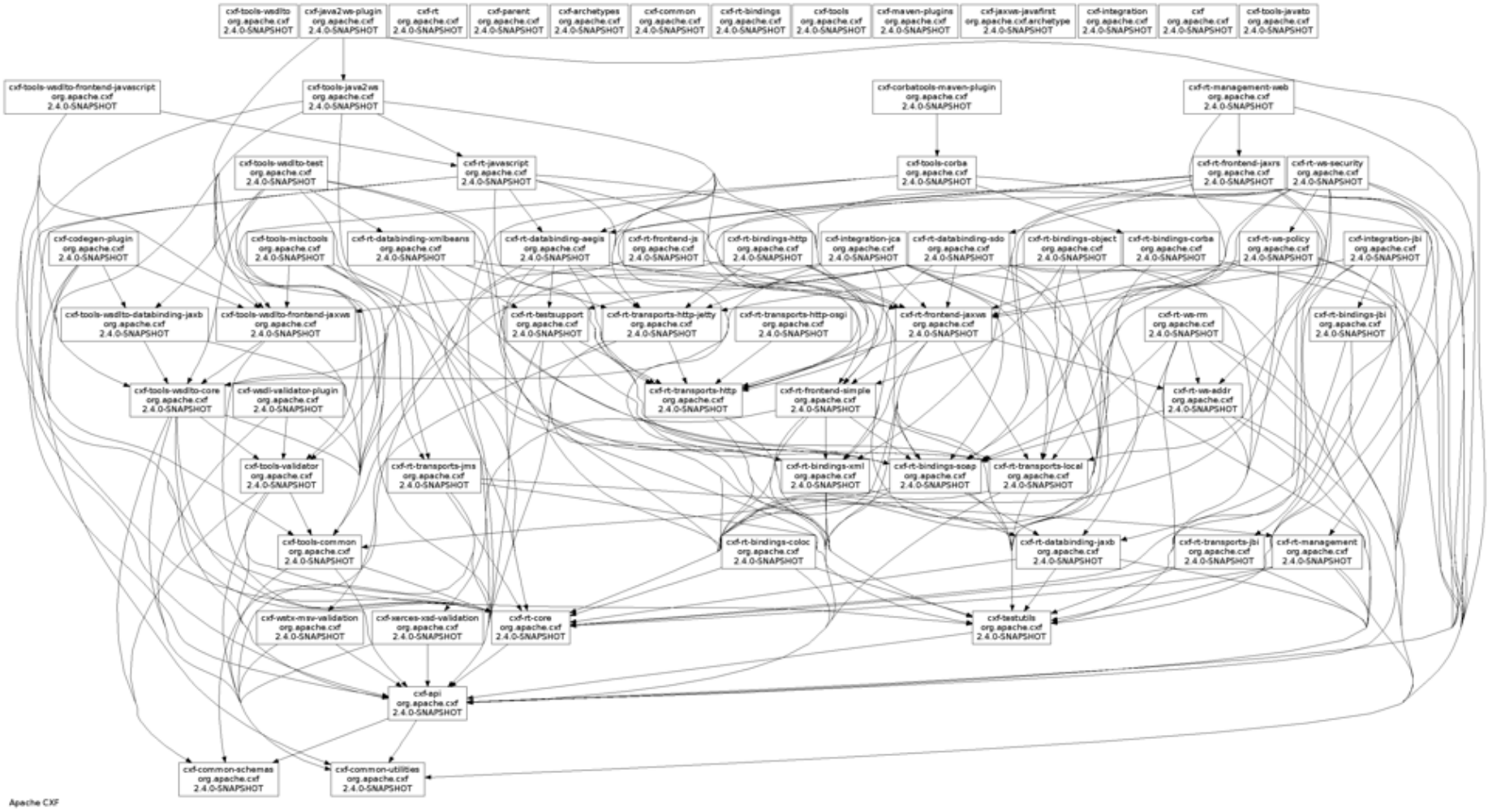


Topological Ordering

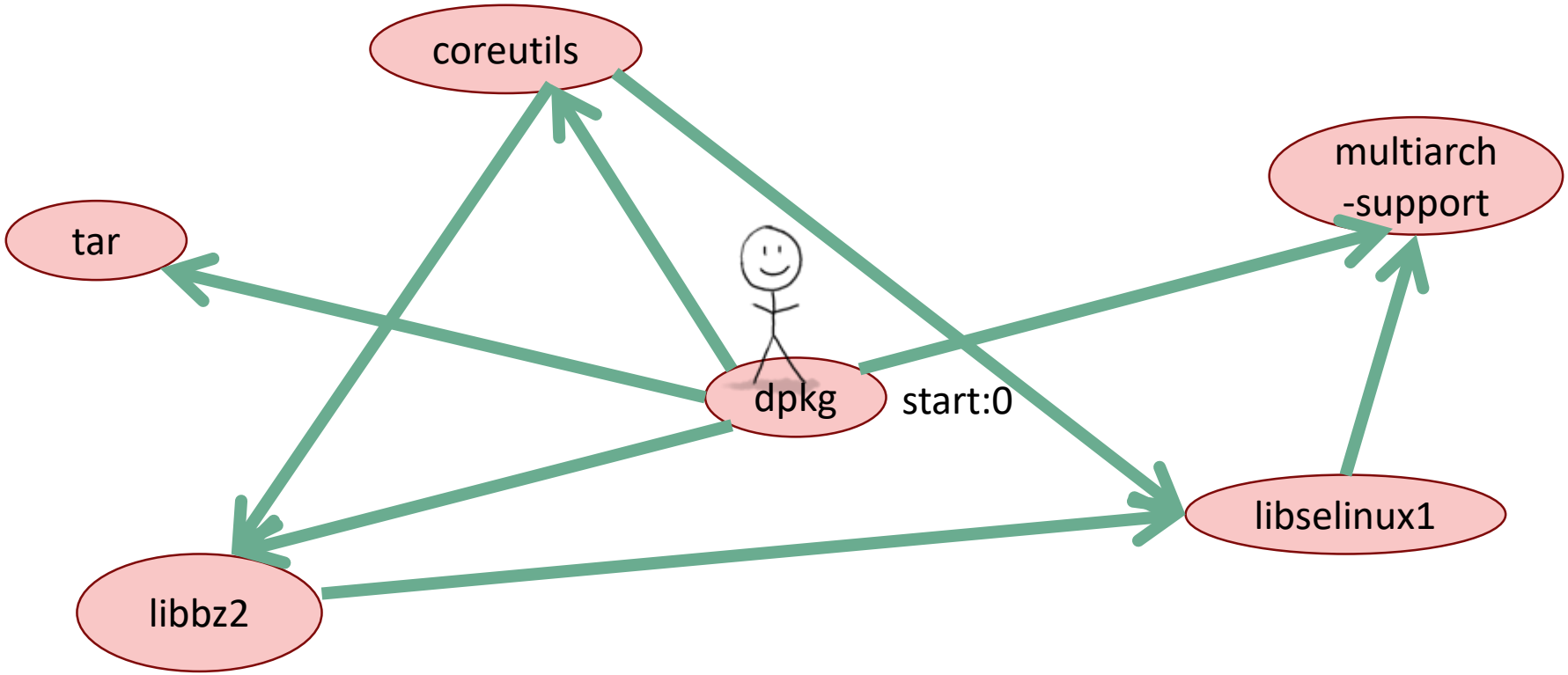
- Application of DFS
- **Find an ordering of vertices in a DAG** so that all of the dependency requirements are met.
  - If v comes before w in the ordering, there is not an edge from w to v.



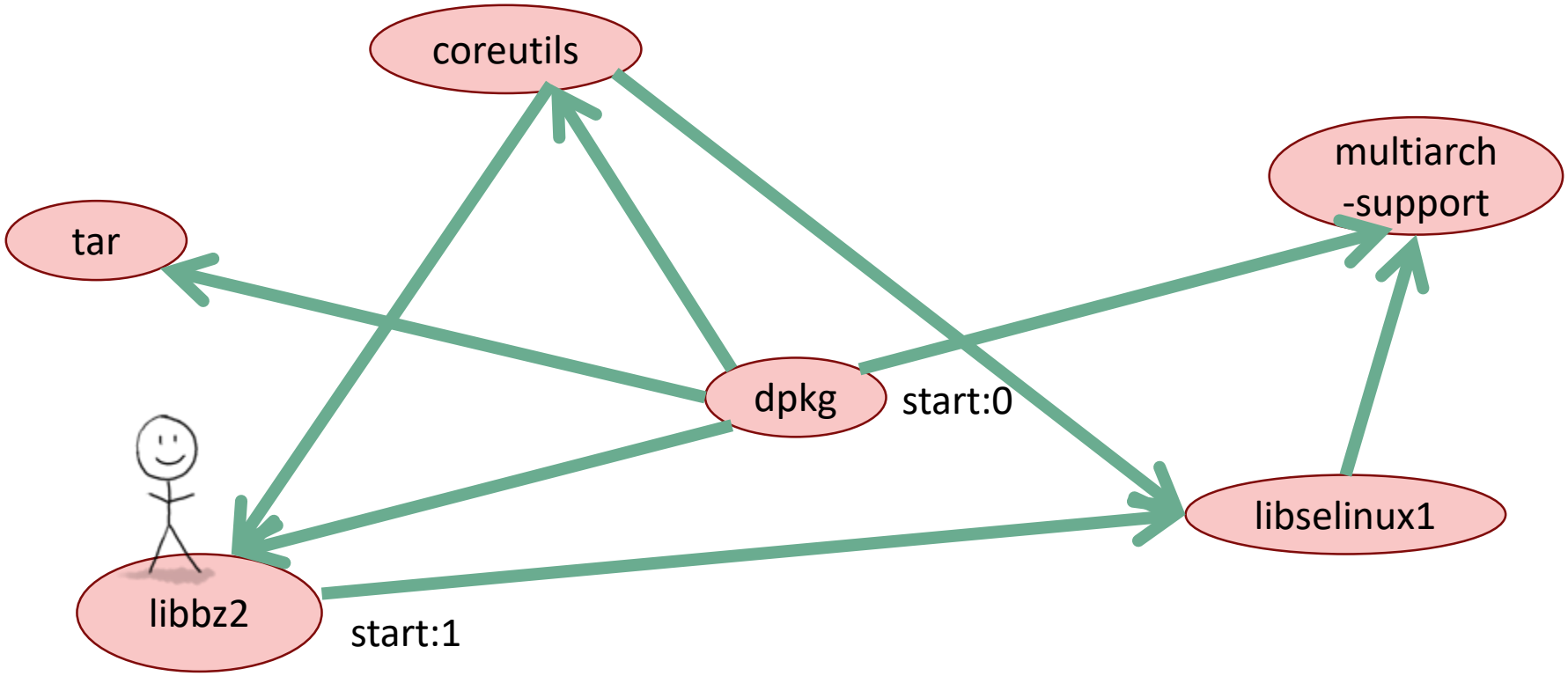
Can’t always eyeball it.



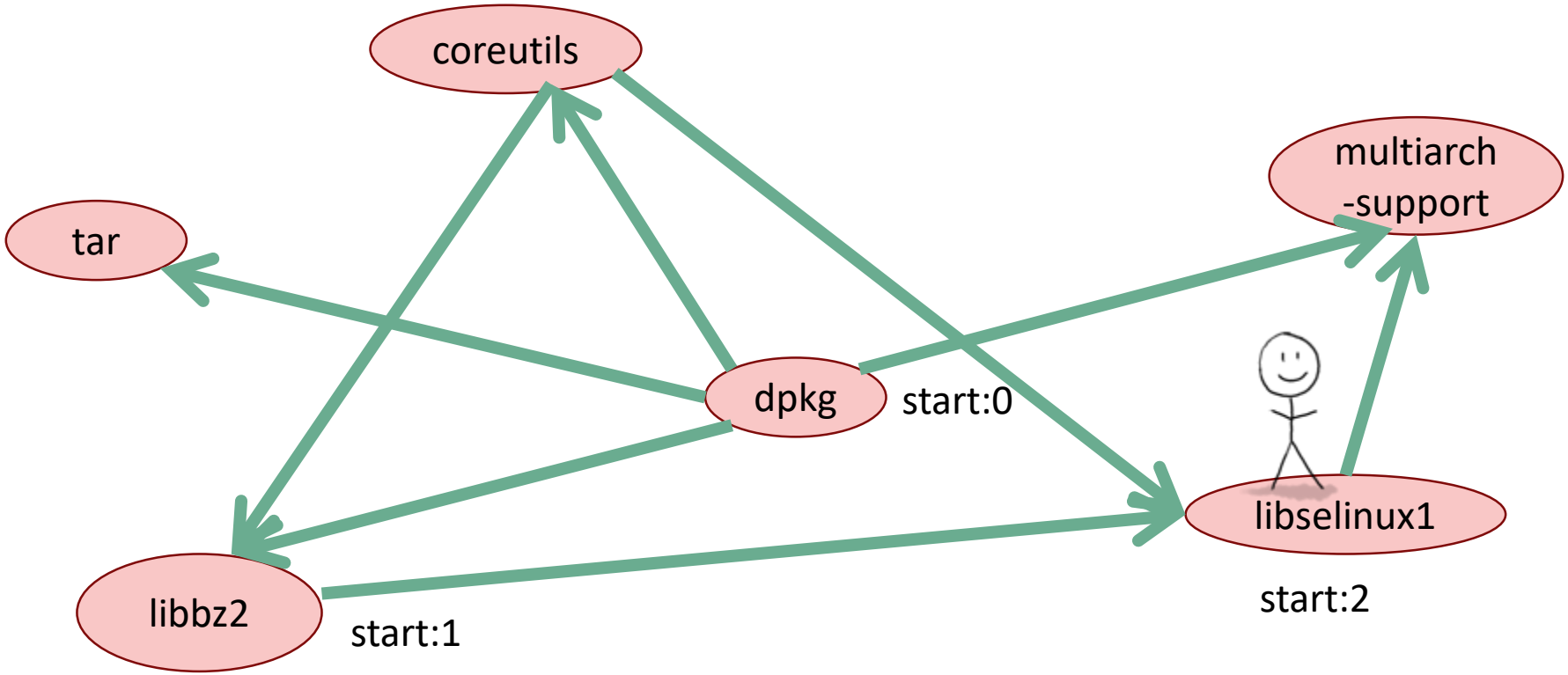
Let's do DFS


coreutils
multiarch
-support
tar
dpkg
start:0
libselinux1
libbz2

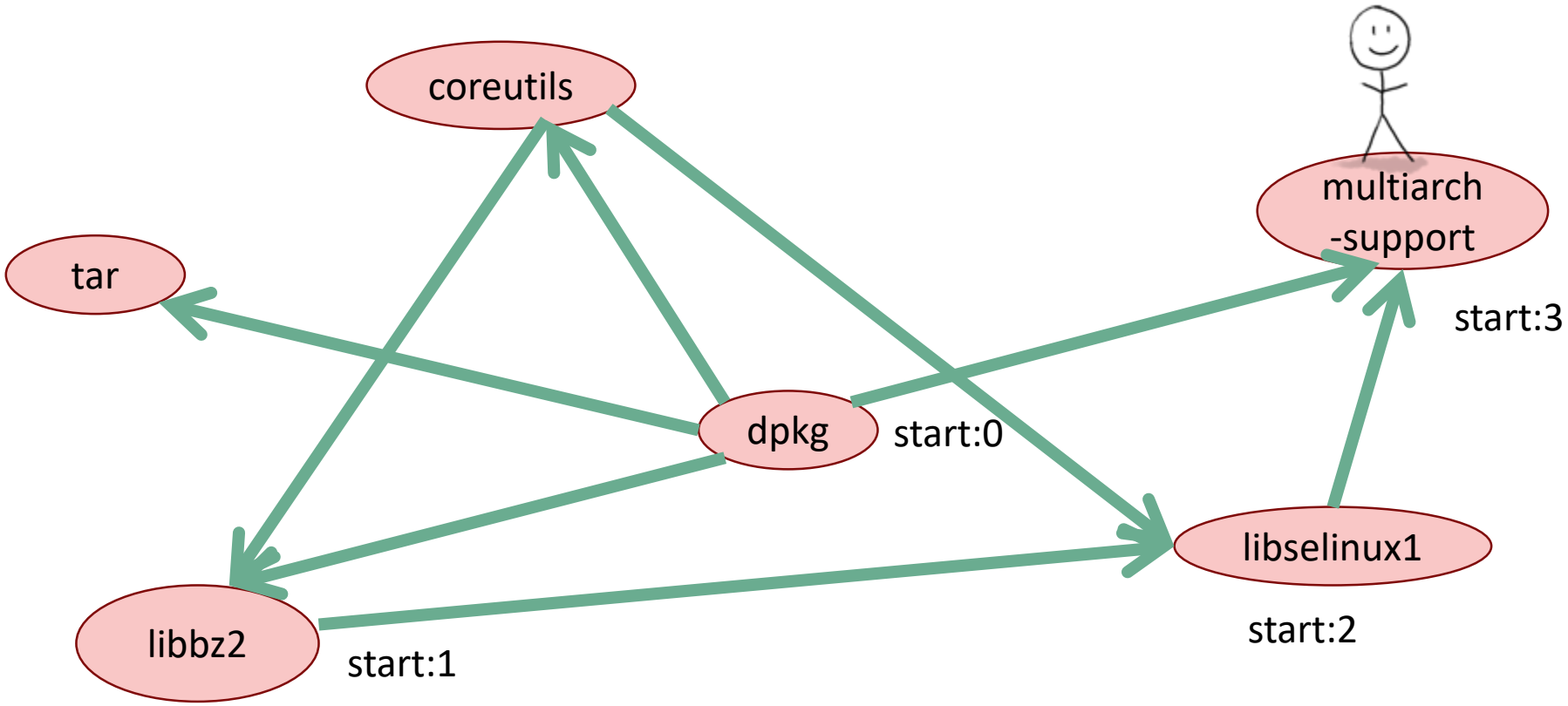
Let’s do DFS


coreutils
multiarch
-support
tar
dpkg
start:0
libselinux1
libbz2
start:1

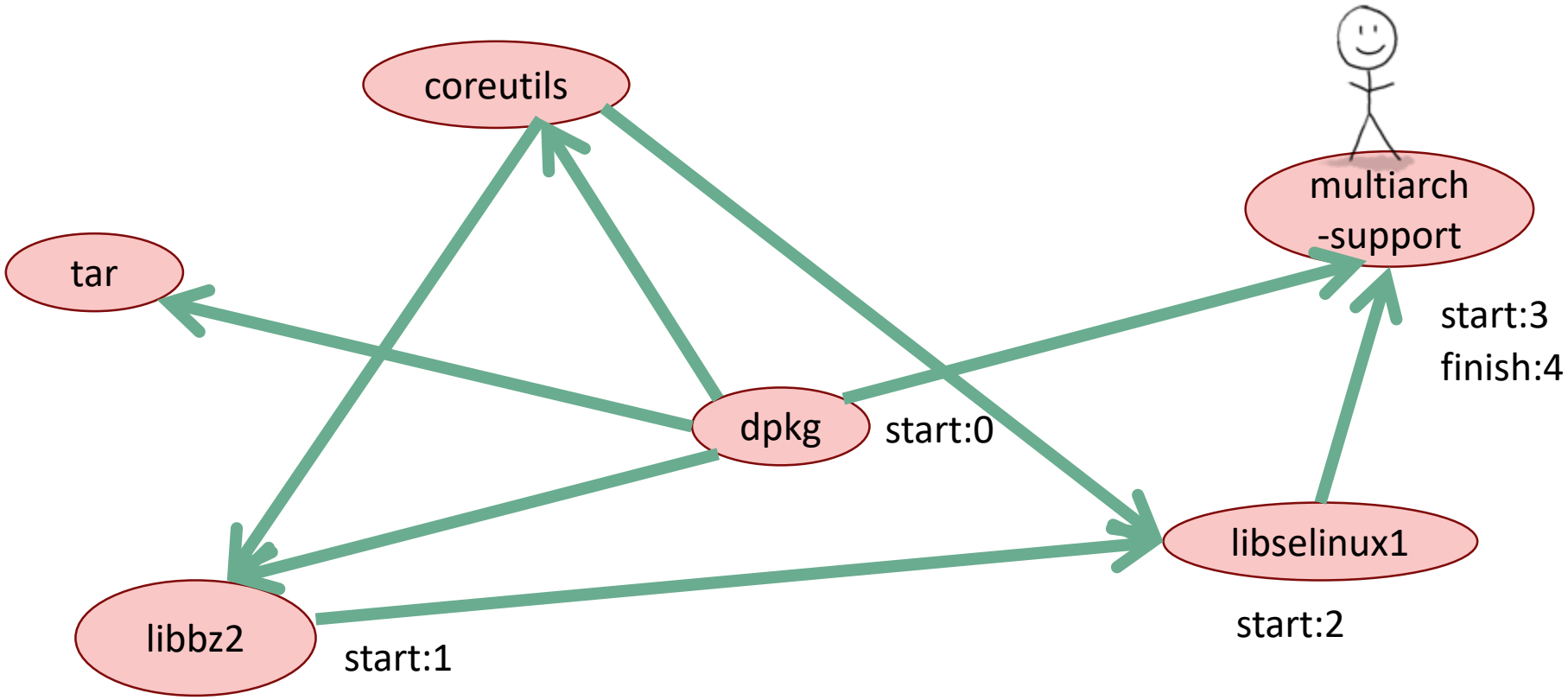
Let's do DFS


coreutils
multiarch
-support
tar
dpkg
start:0
libselinux1
libbz2
start:1
start:2

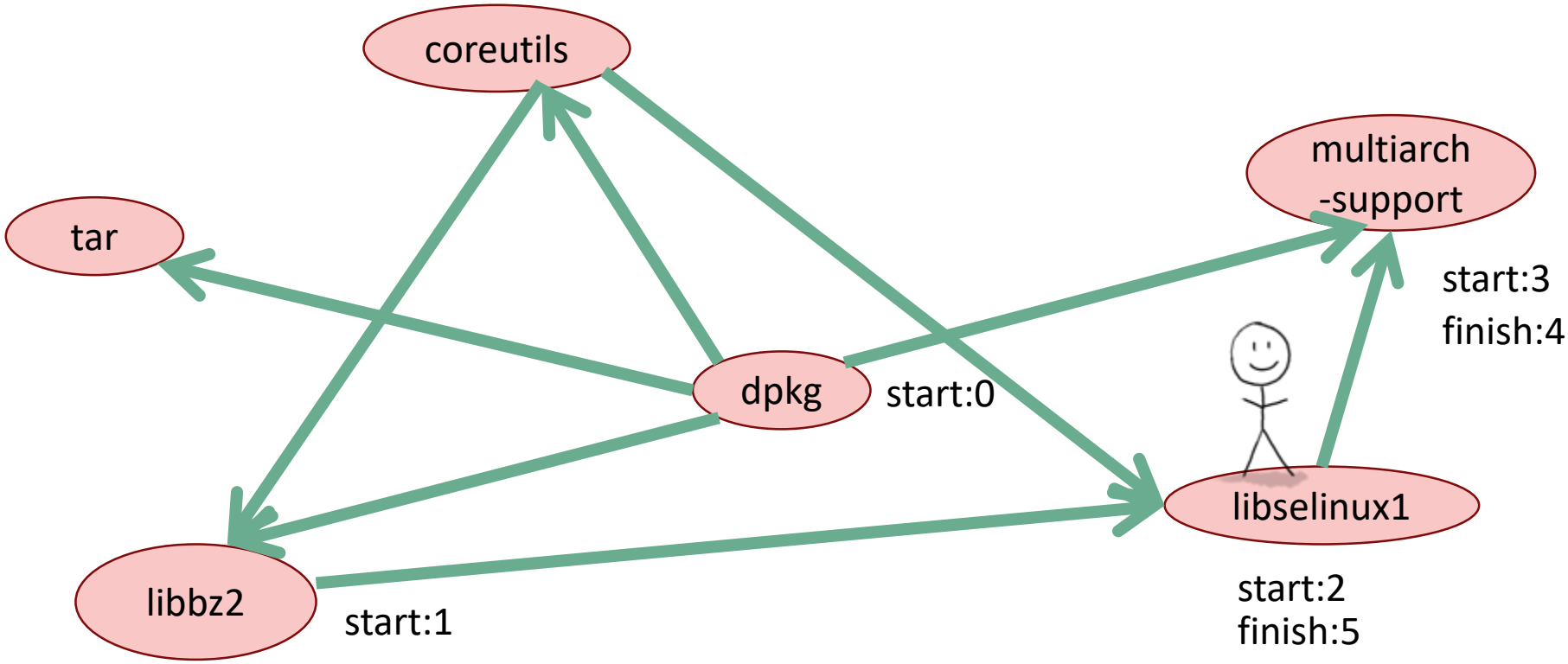
Let's do DFS


coreutils
tar
dpkg
start:0
multiarch
-support
start:3
libselinux1
start:2
libbz2
start:1

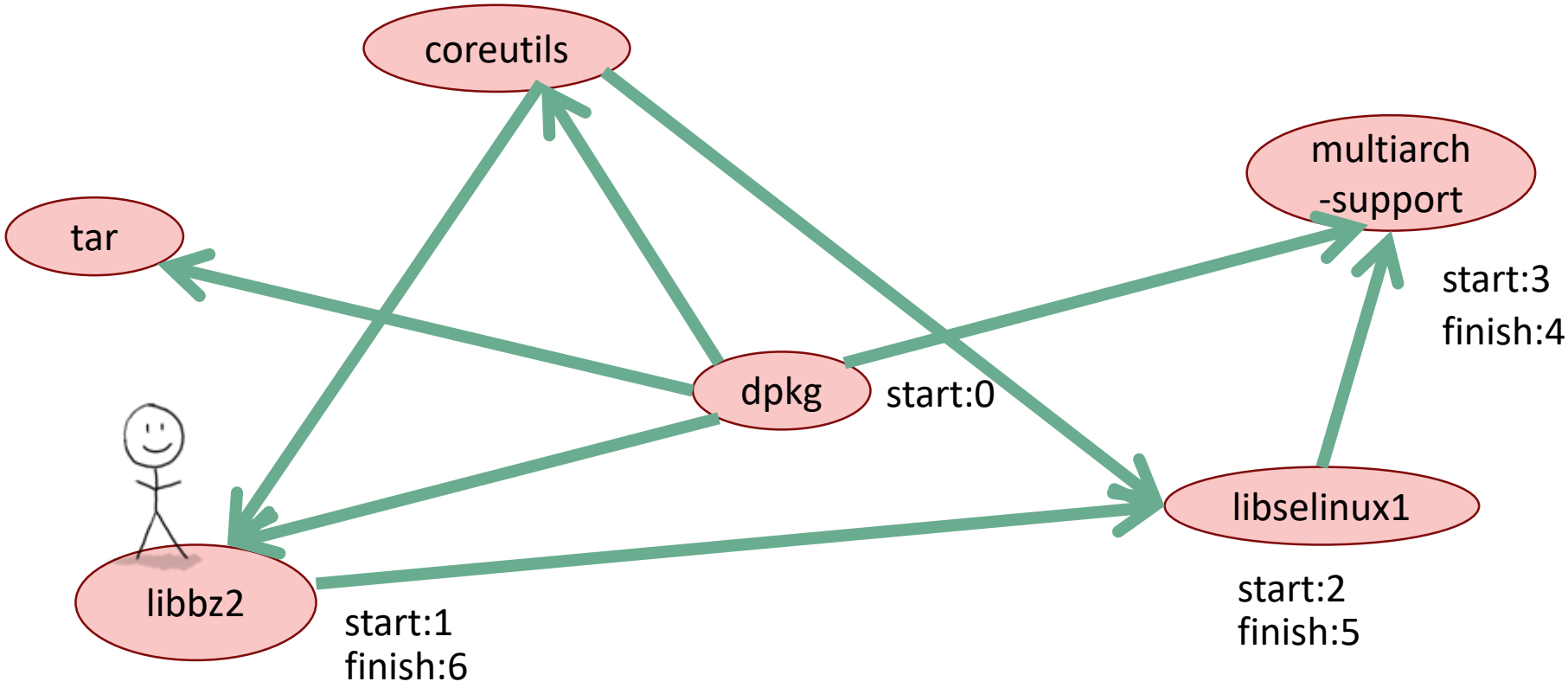
Let's do DFS


coreutils
tar
dpkg
start:0
multiarch
-support
start:3
finish:4
libselinux1
start:2
libbz2
start:1

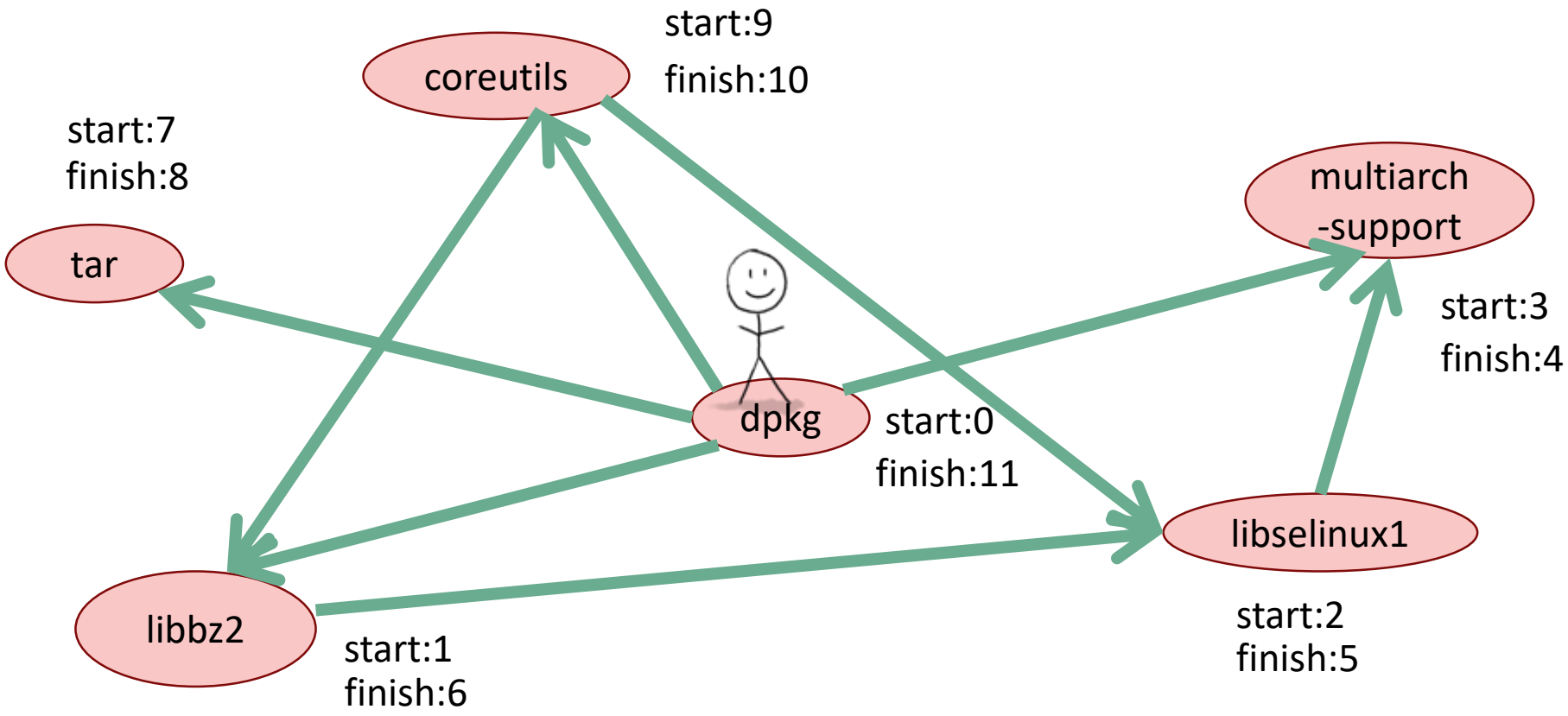
Let's do DFS


coreutils
multiarch
-support
tar
start:3
finish:4
dpkg
start:0
libselinux1
libbz2
start:1
start:2
finish:5

Let's do DFS


coreutils
multiarch
-support
tar
start:3
finish:4
dpkg
start:0
libselinux1
libbz2
start:2
finish:5
start:1
finish:6

Let's do DFS



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

Finish times seem useful

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

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

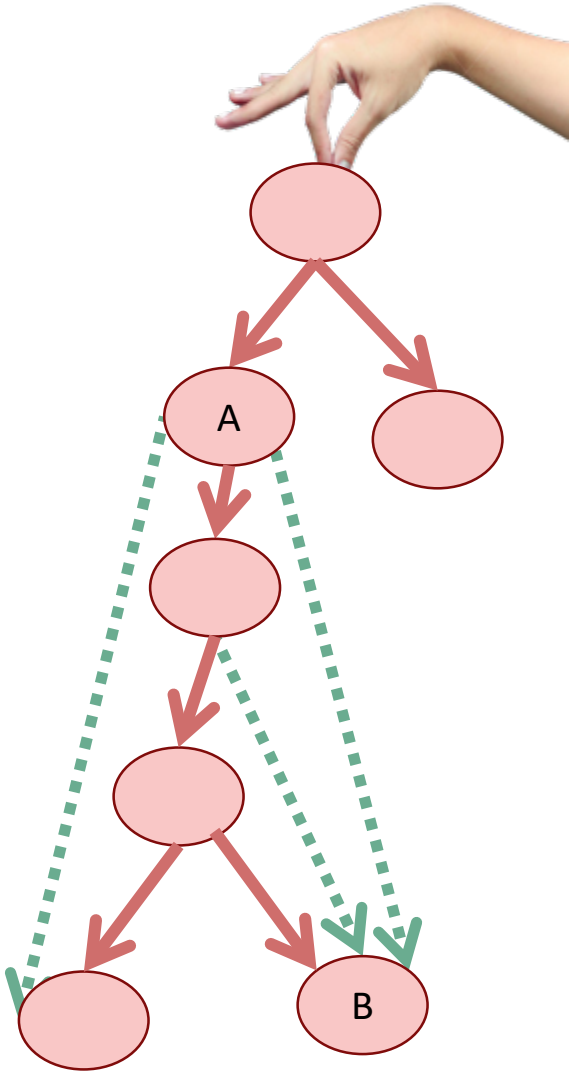
Proof the claim

Suppose the underlying graph has no cycles

If A → B, then **B.finishTime < A.finishTime**

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

- aka, **B.finishTime < A.finishTime**.



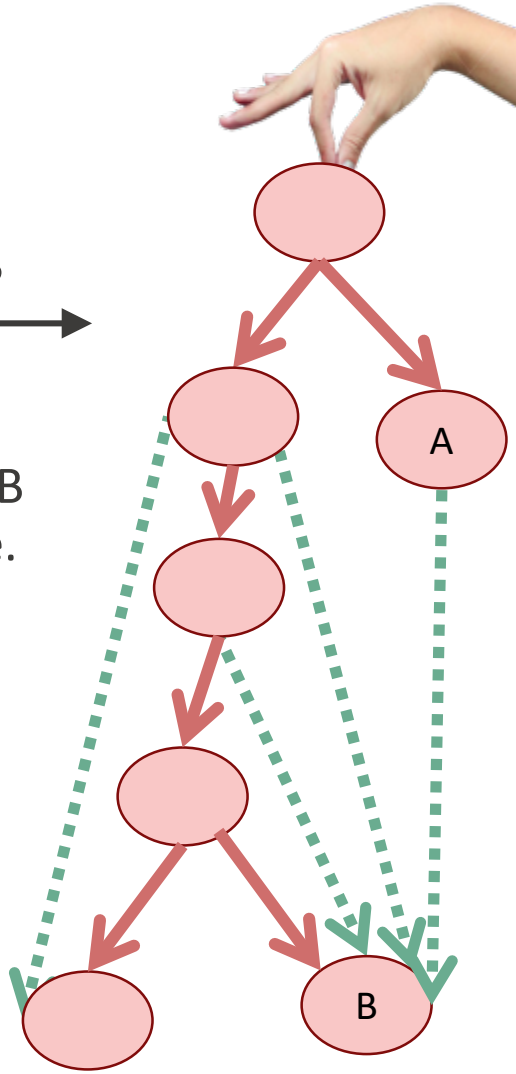
Proof the claim

Suppose the underlying graph has no cycles

If A → B, then **B.finishTime < A.finishTime**

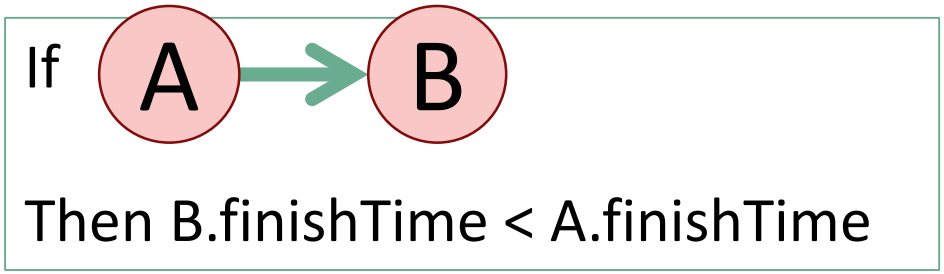
- **Case 2:** B is a NOT descendant of A in the DFS tree.
  - Notice that A can't be a descendant of B in the DFS tree or else there'd be a cycle; so it looks like this →
- Then we must have explored B before A.
  - Otherwise we would have gotten to B from A, and B would have been a descendant of A in the DFS tree.
- Then

- aka, **B.finishTime < A.finishTime**.



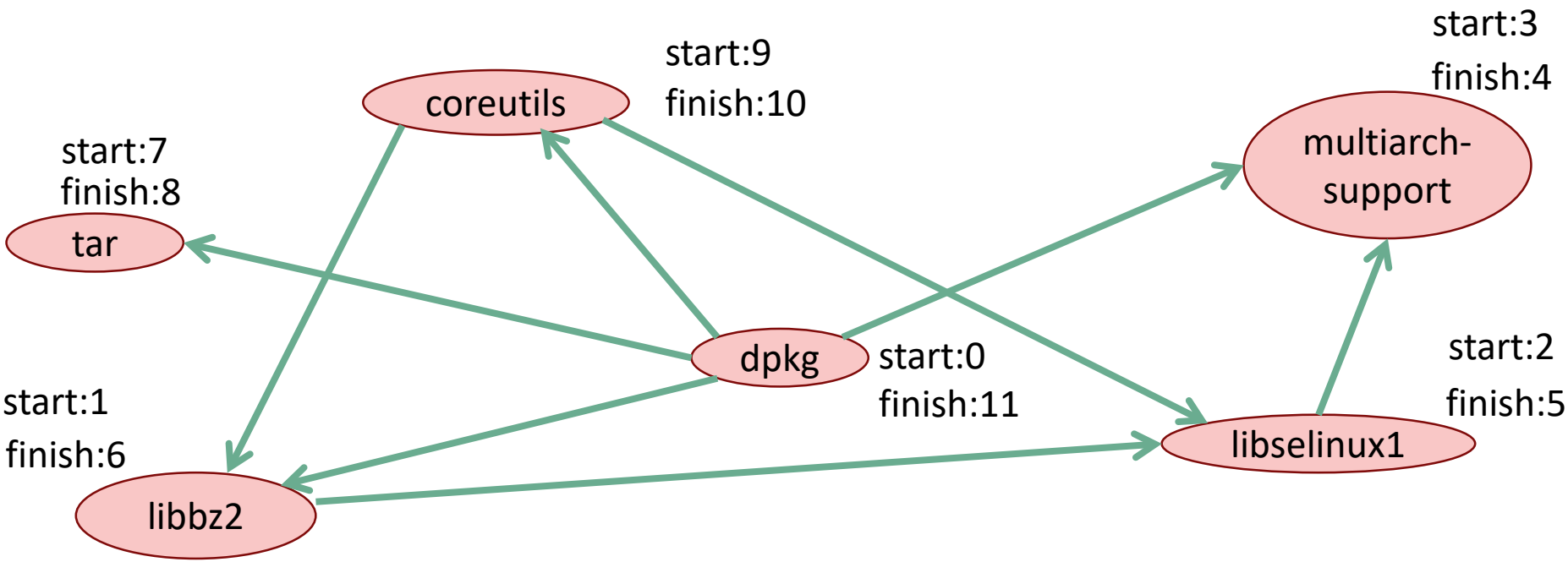
Theorem

- If we run DFS on a directed acyclic graph,



Back to topological sorting

- In what order should I install packages?

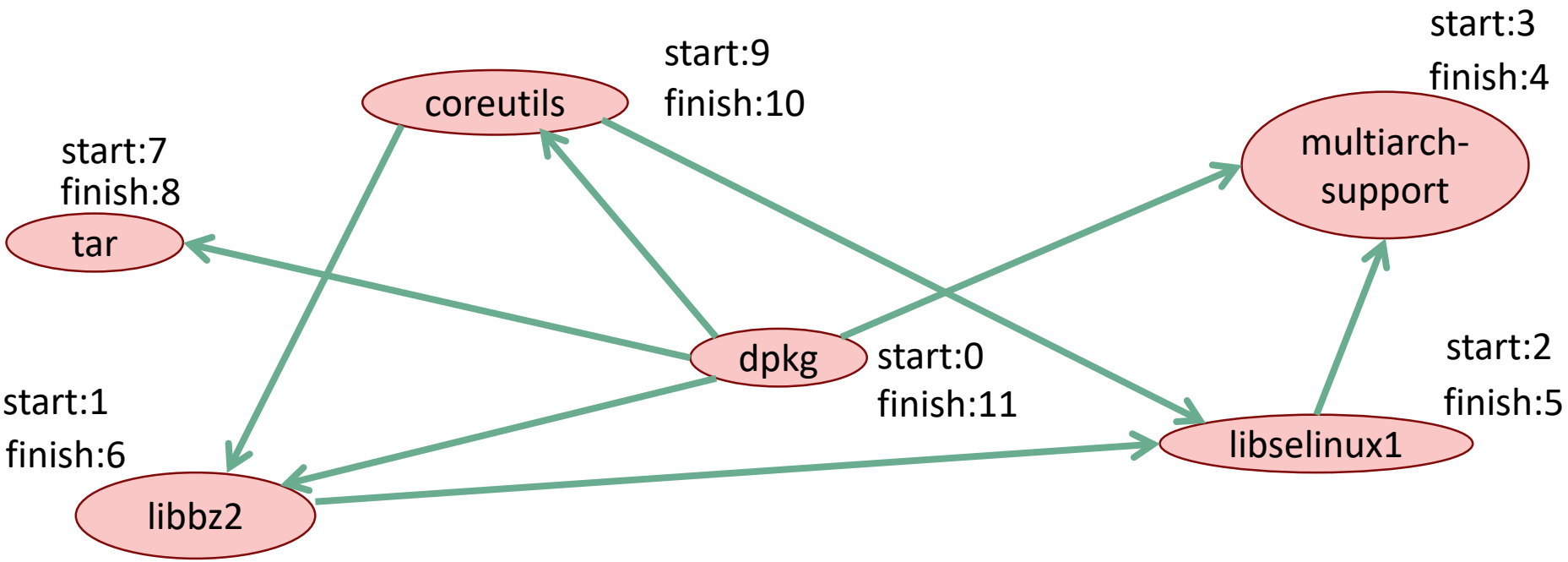


- **In reverse order** of finishing time in DFS!

Topological Sorting

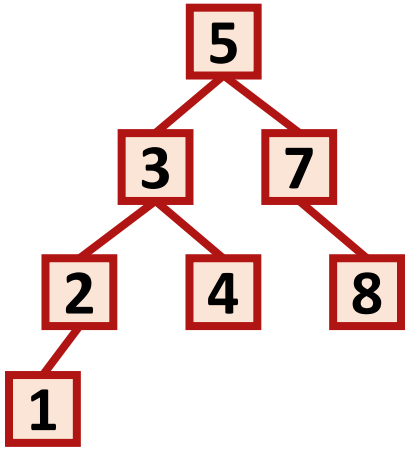
- Do DFS
- When you mark a vertex as **all done**, put it at the **beginning** of the list.

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



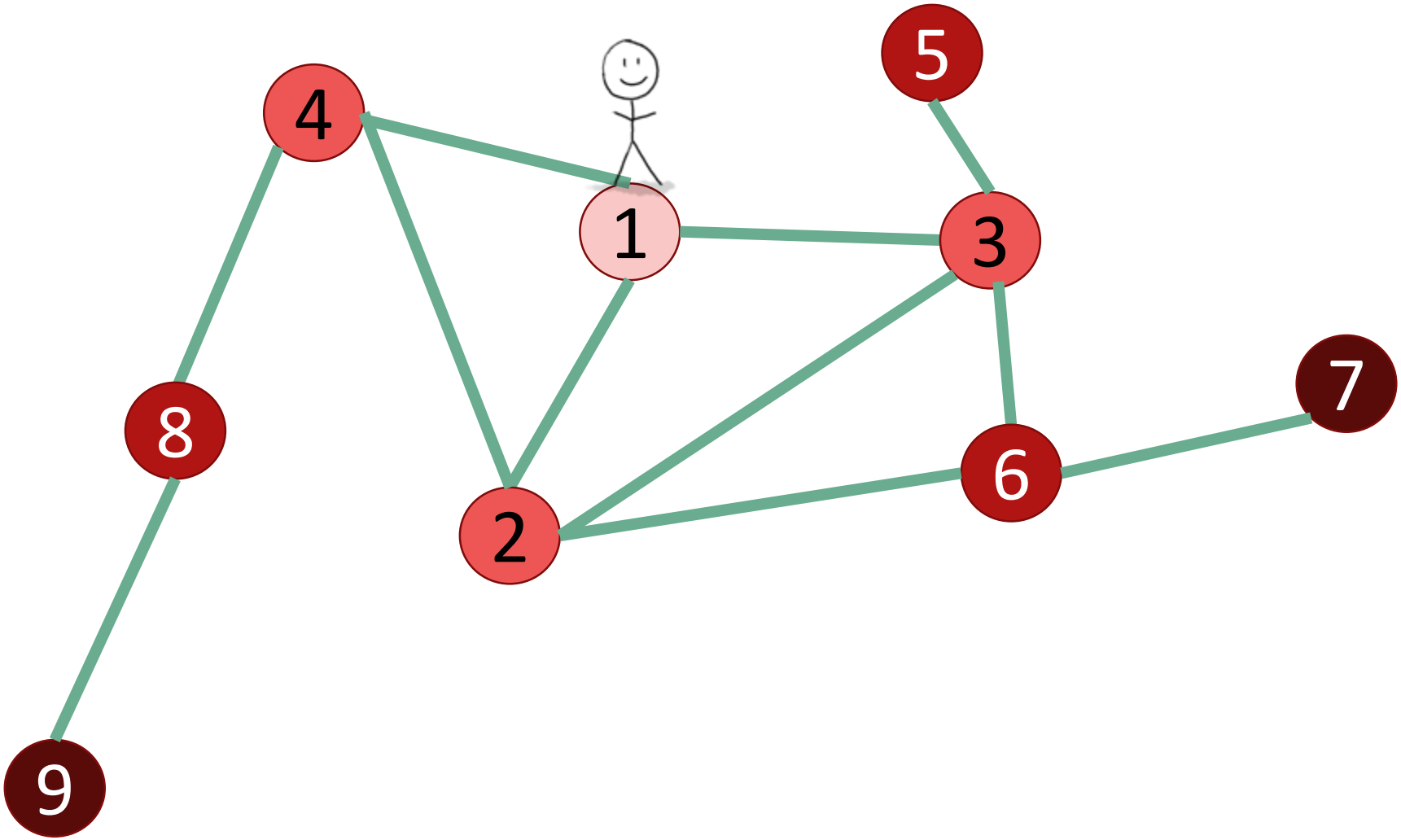
What did we just learn?

- DFS can help you solve the **topological sorting problem**
- Another use of DFS: **In-order enumeration** of binary search trees

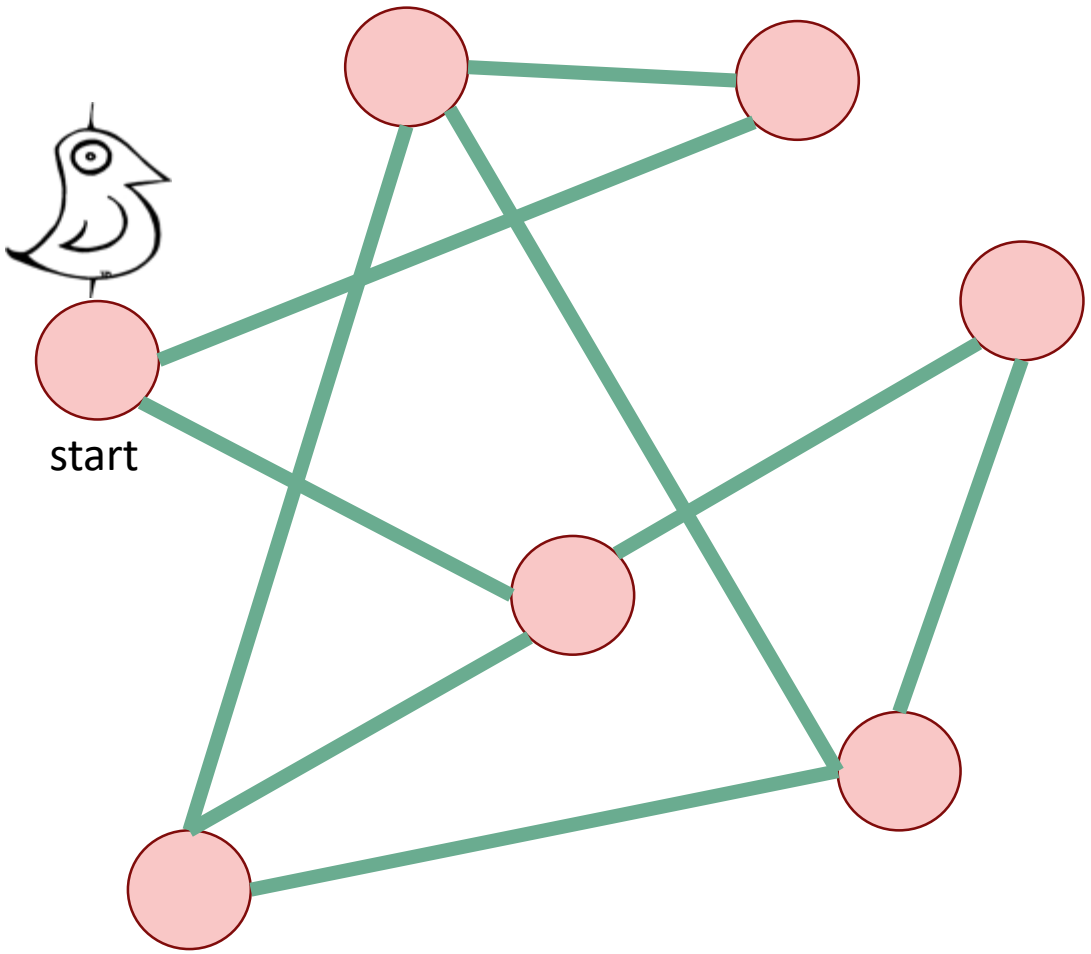


Breadth-First Search

How do we explore a graph?



Breadth-First Search



Not been there yet

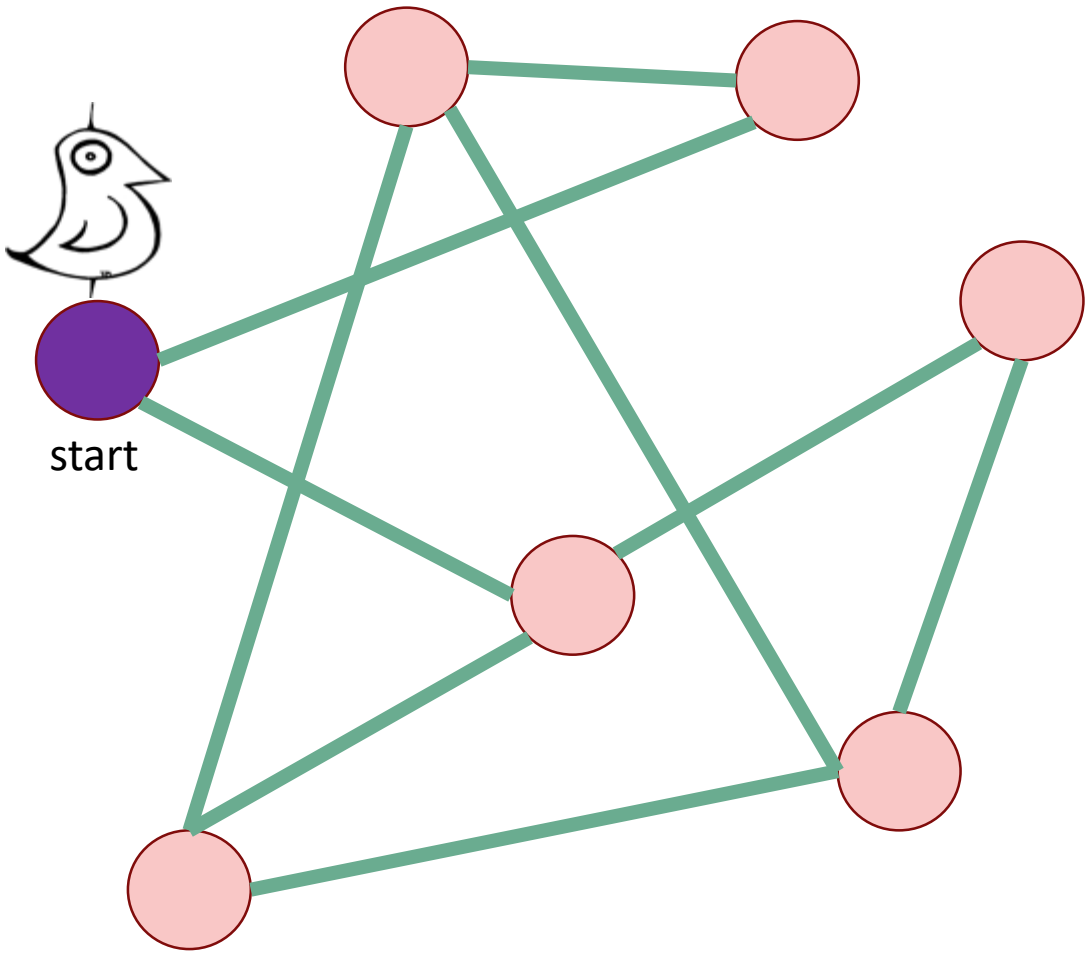
Can reach there in zero steps

Can reach there in one step

Can reach there in two steps

Can reach there in three steps

Breadth-First Search



Not been there yet

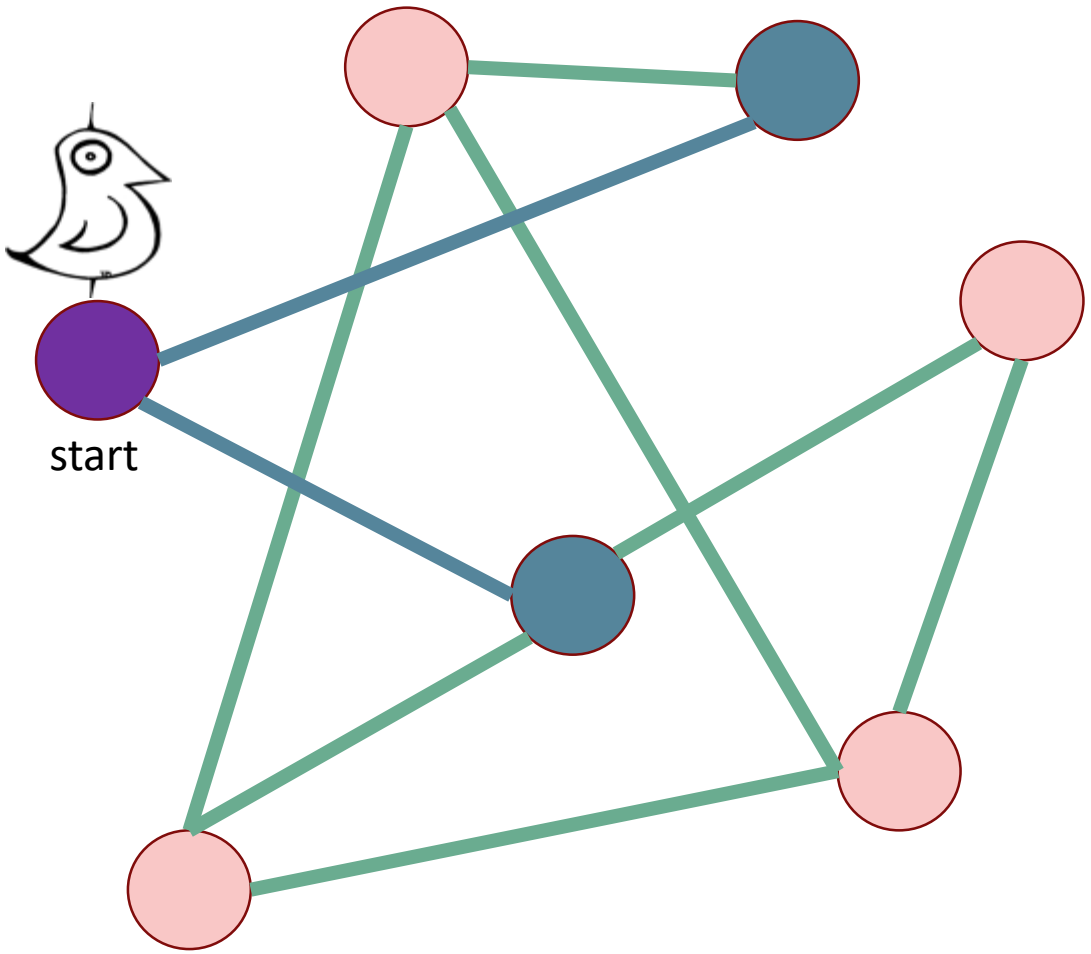
Can reach there in zero steps

Can reach there in one step

Can reach there in two steps

Can reach there in three steps

Breadth-First Search



Not been there yet

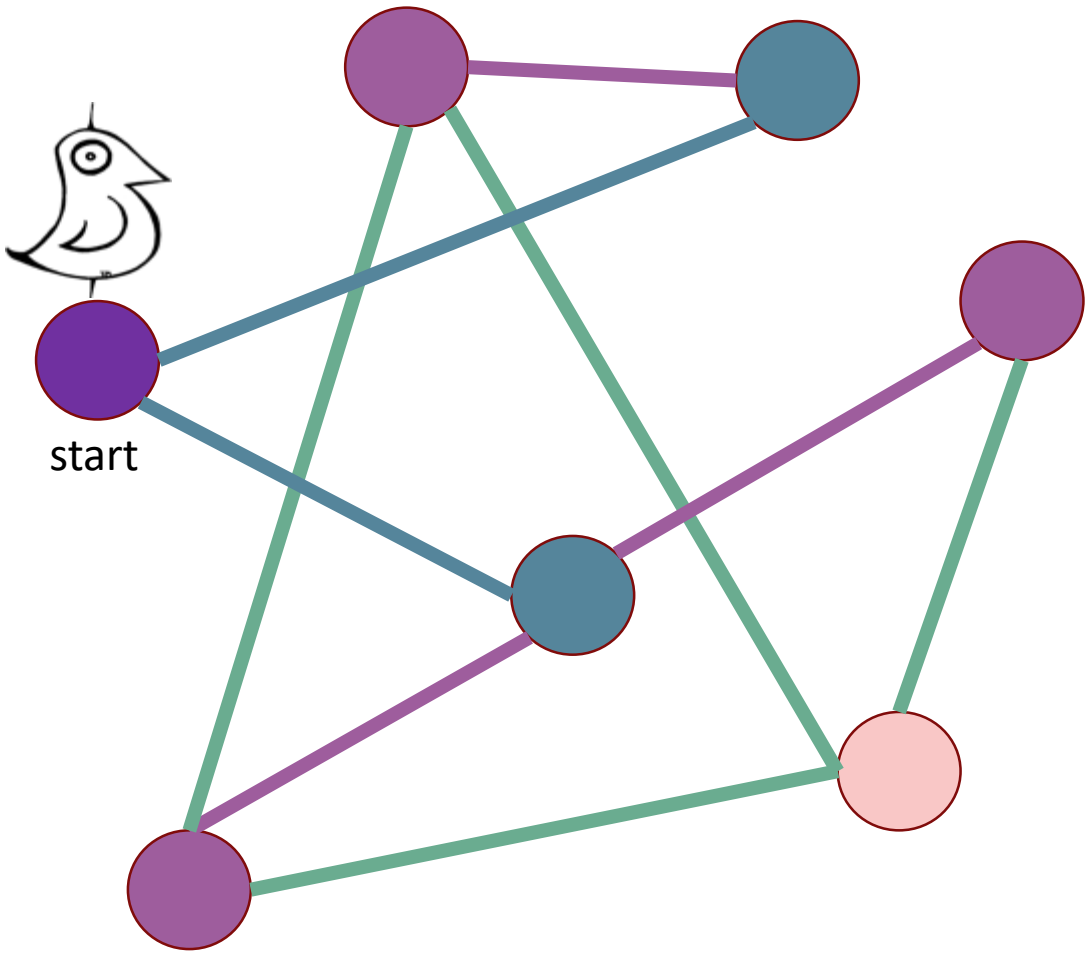
Can reach there in zero steps

Can reach there in one step

Can reach there in two steps

Can reach there in three steps

Breadth-First Search



Not been there yet

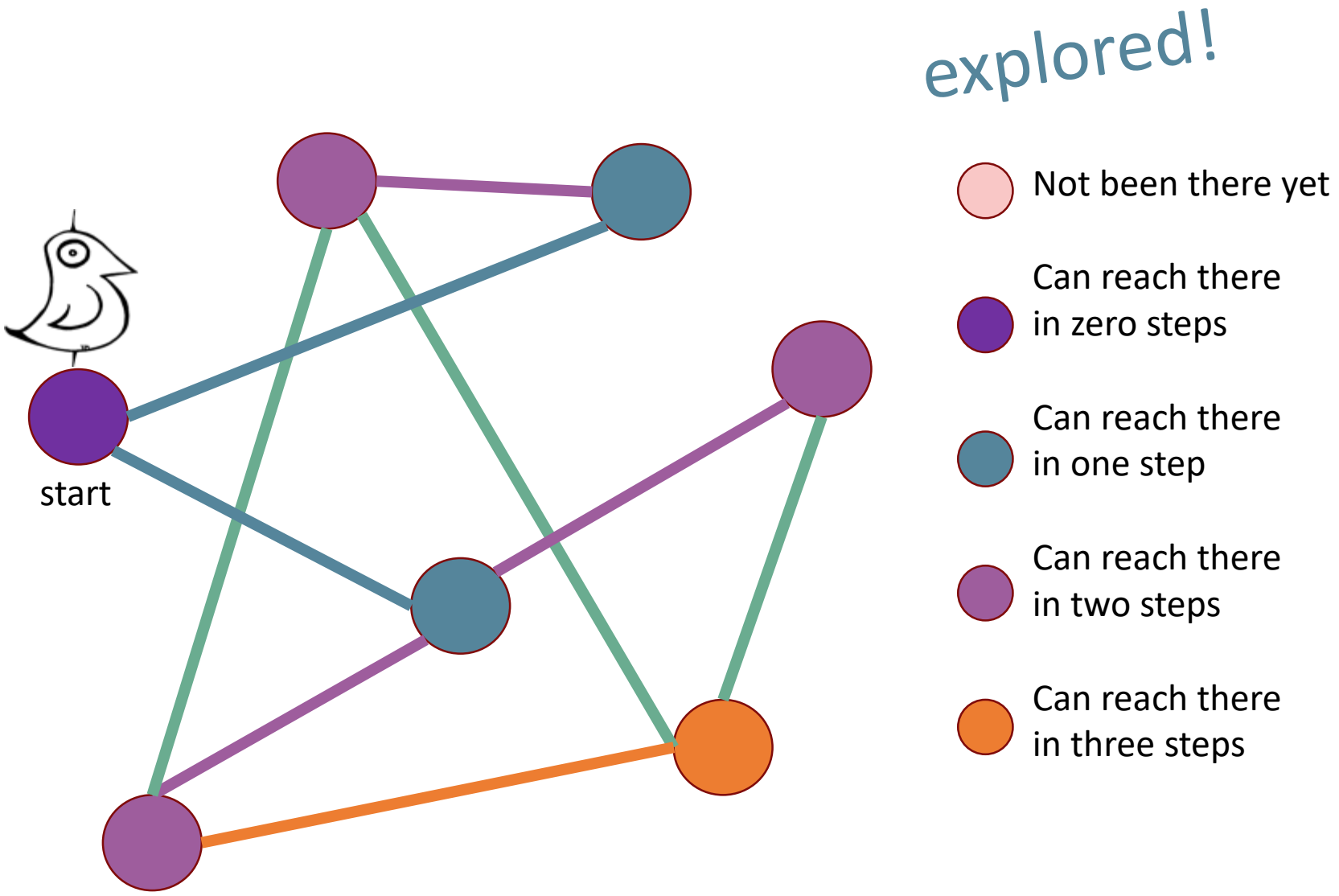
Can reach there in zero steps

Can reach there in one step

Can reach there in two steps

Can reach there in three steps

Breadth-First Search

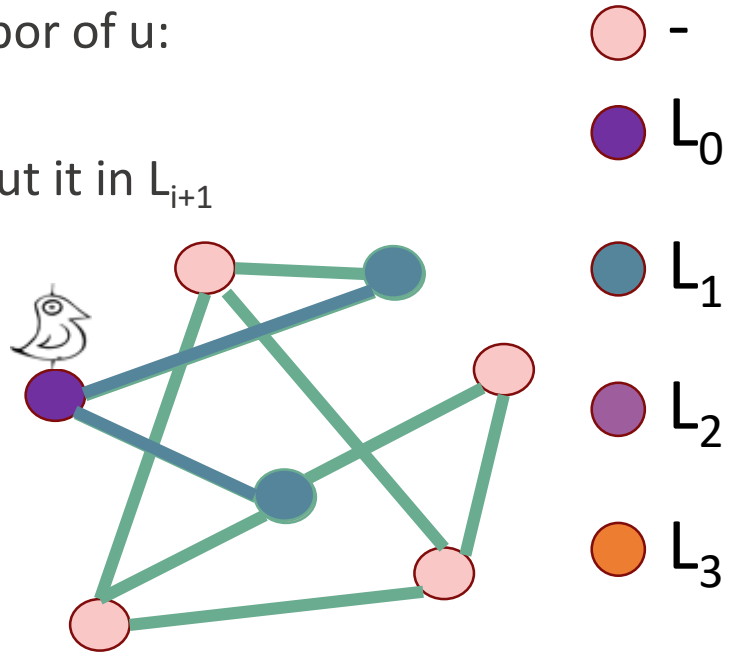


Breadth-First Search

- Set L_i = [] for i=1,…,n
- L_0 = [w], where w is the start node
- Mark w as visited
- **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}

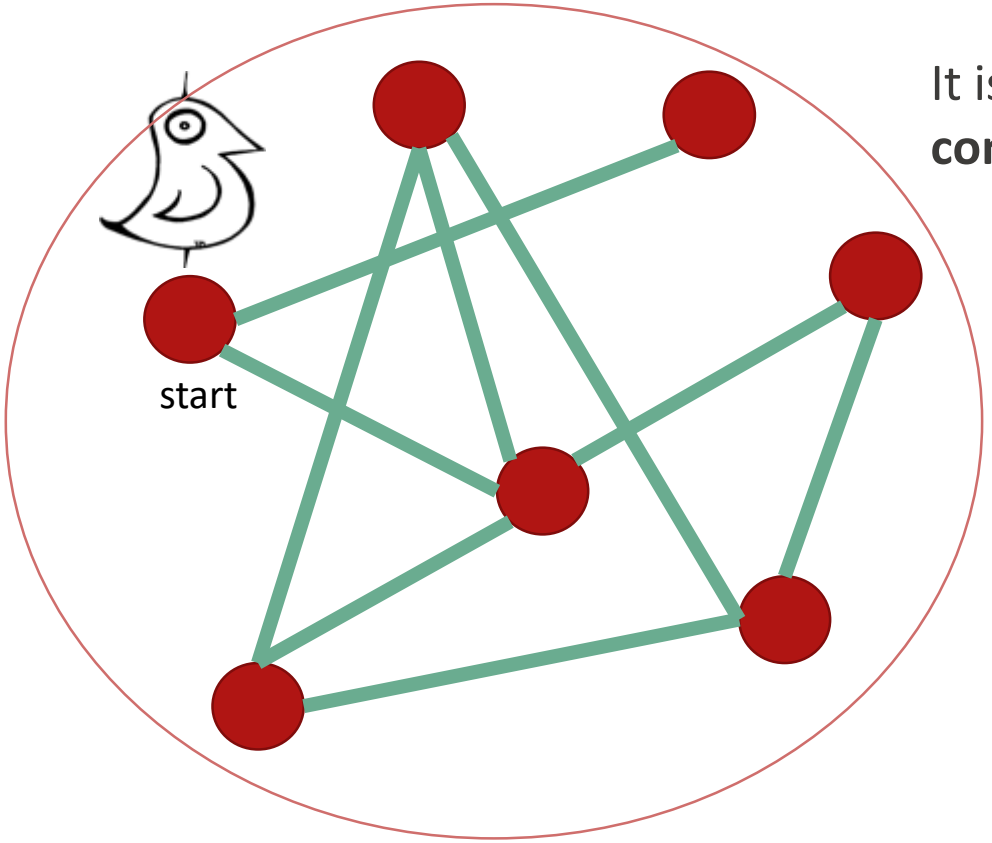
L_i is the set of nodes we can reach in i steps from w

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



Breadth-First Search

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



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

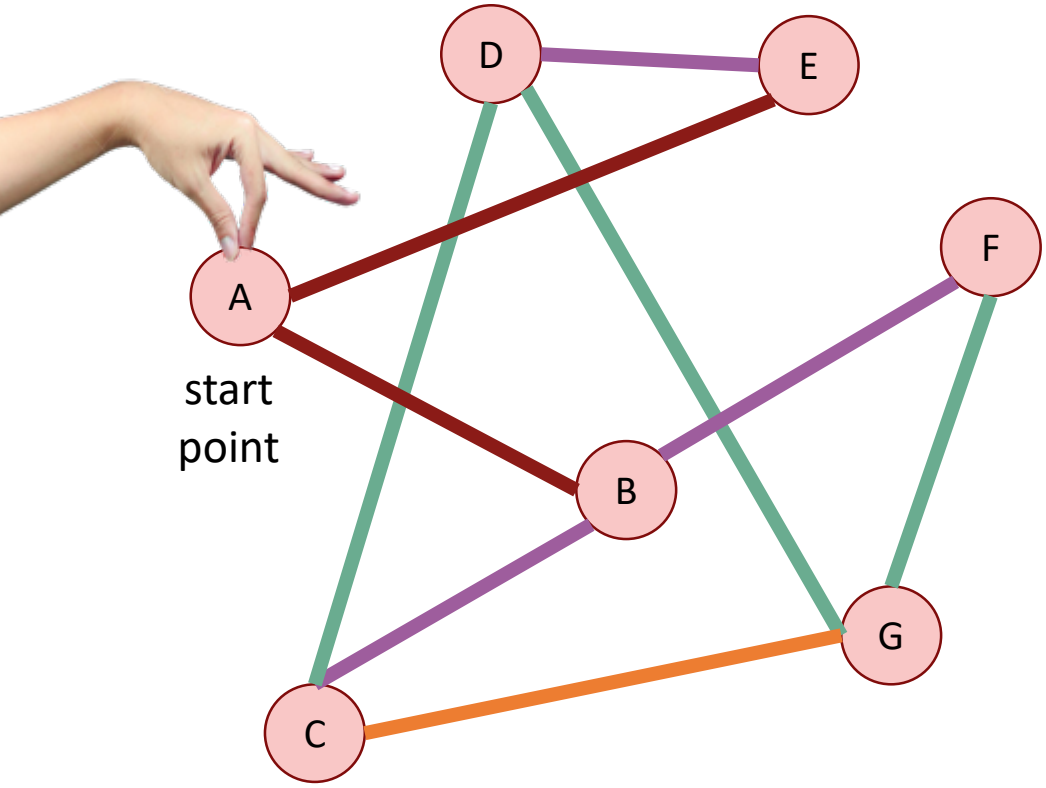
- Like DFS, BFS also works fine on directed graphs.

Running time

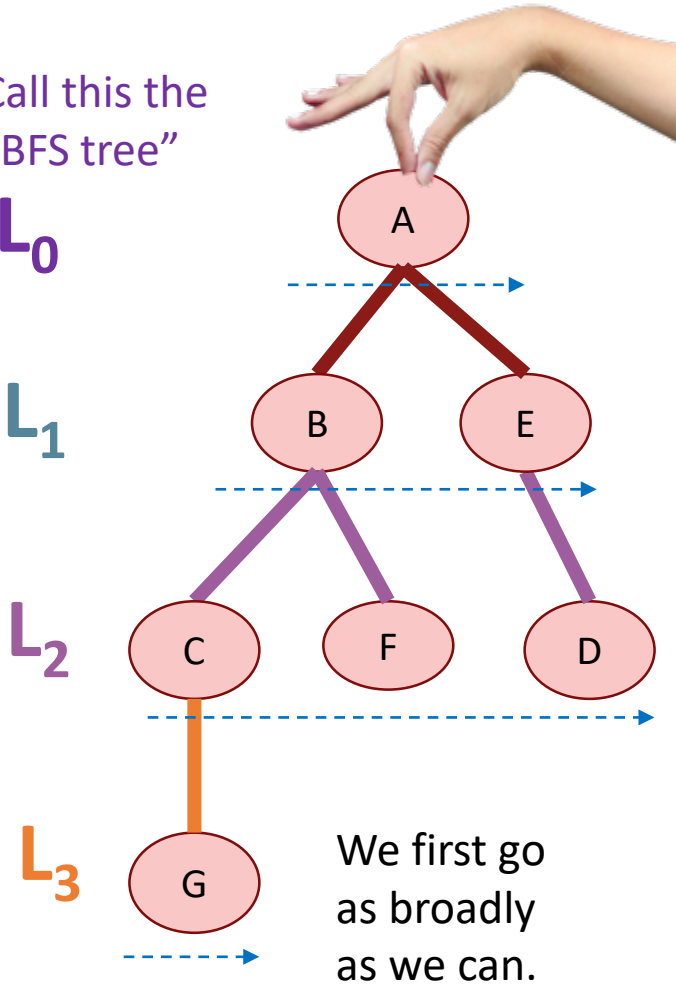
- To explore the whole graph, explore the connected components one-by-one.
- Same argument as DFS: BFS running time is **O(n + m)**

Why is it called breadth-first?

- We are implicitly building a tree:



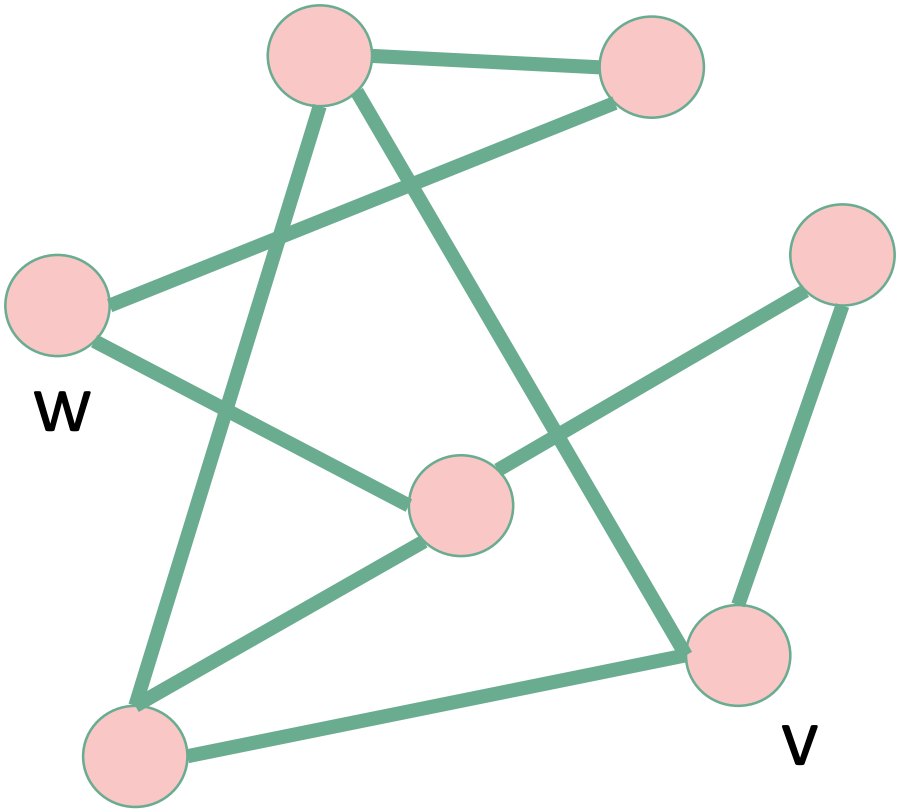
Call this the
"BFS tree"



We first go as broadly as we can.

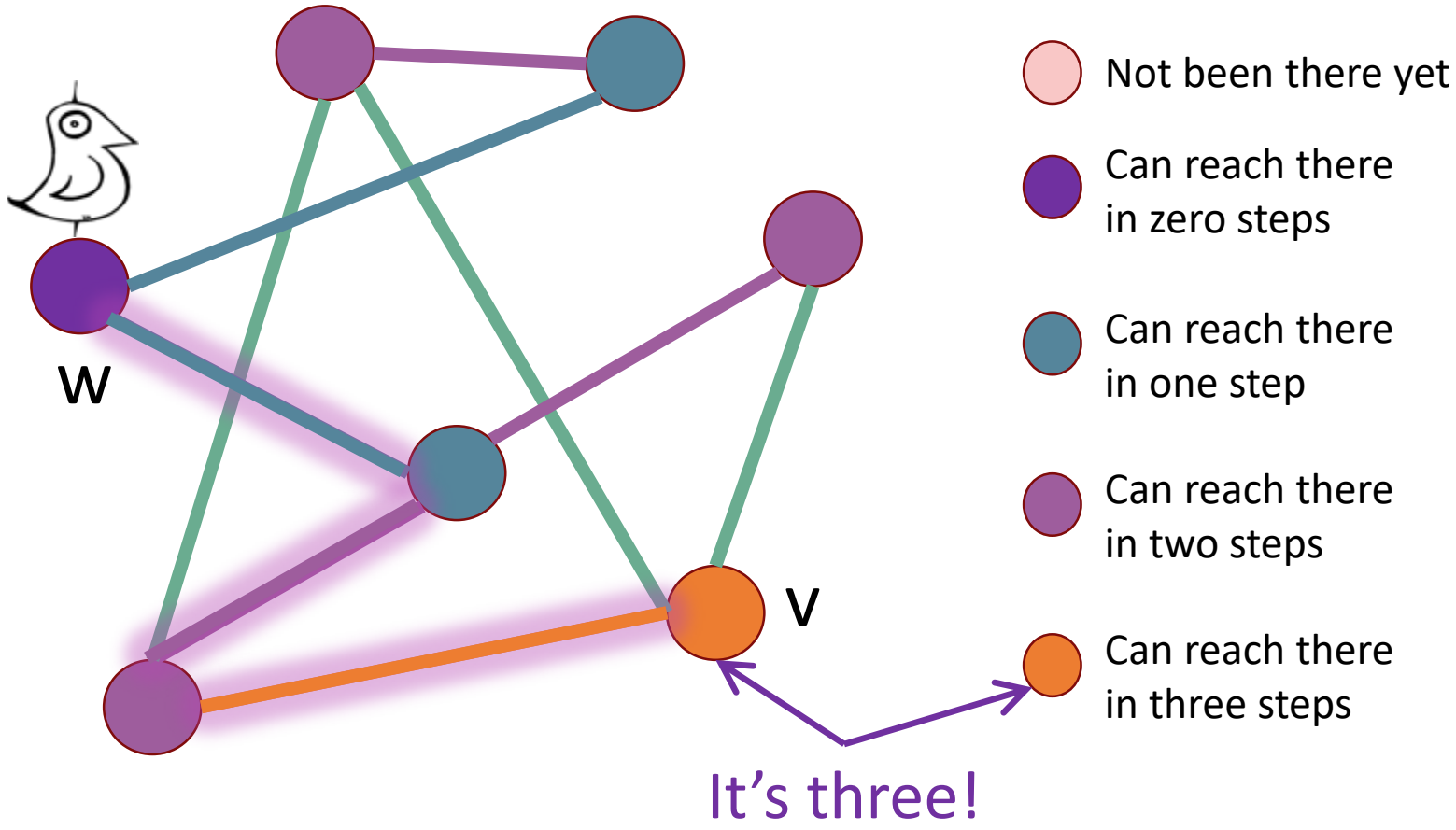
Application of BFS: shortest path

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



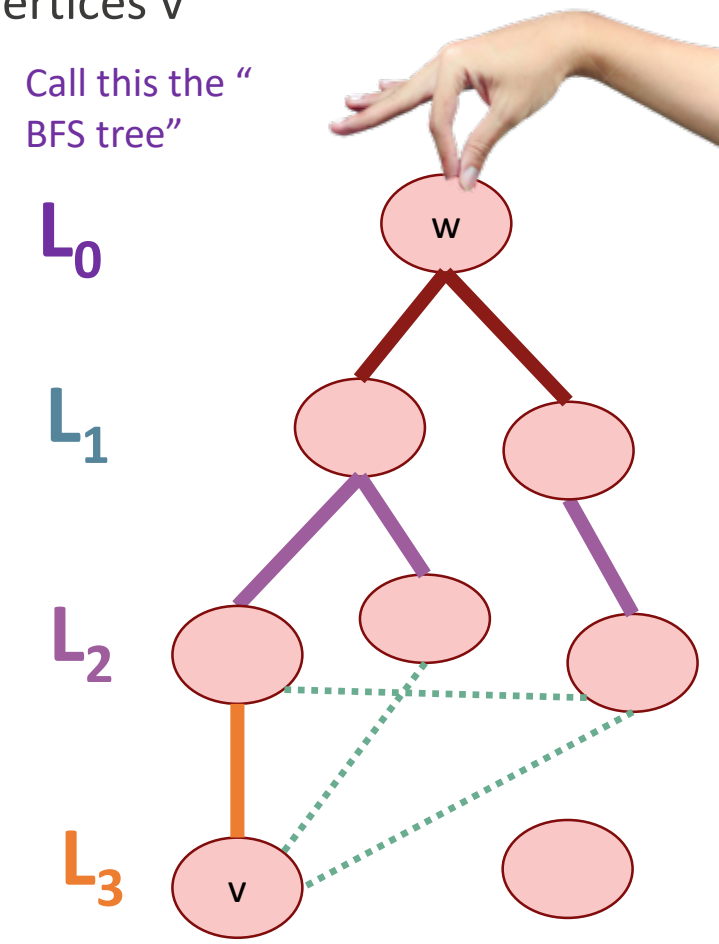
Application of BFS: shortest path

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



Finding distance

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



What have we learned?

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

Recap

- **Depth-first search**
  - Useful for topological sorting
  - Also in-order traversals of BSTs
- **Breadth-first search**
  - Useful for finding shortest paths

- **Both DFS, BFS:**
  - Useful for exploring graphs, finding connected components, etc.

Still open (next few lectures)

- We can now find components in undirected graphs.
  - What if we want to find strongly connected components in directed graphs?
- How can we find shortest paths in weighted graphs?

- Next Time:
  - Strongly Connected Components


Any Question?