Lecture 35

Graphs, BFS & DFS

December 14, 2021

WHAT ARE GRAPHS?

Some examples & some terminology

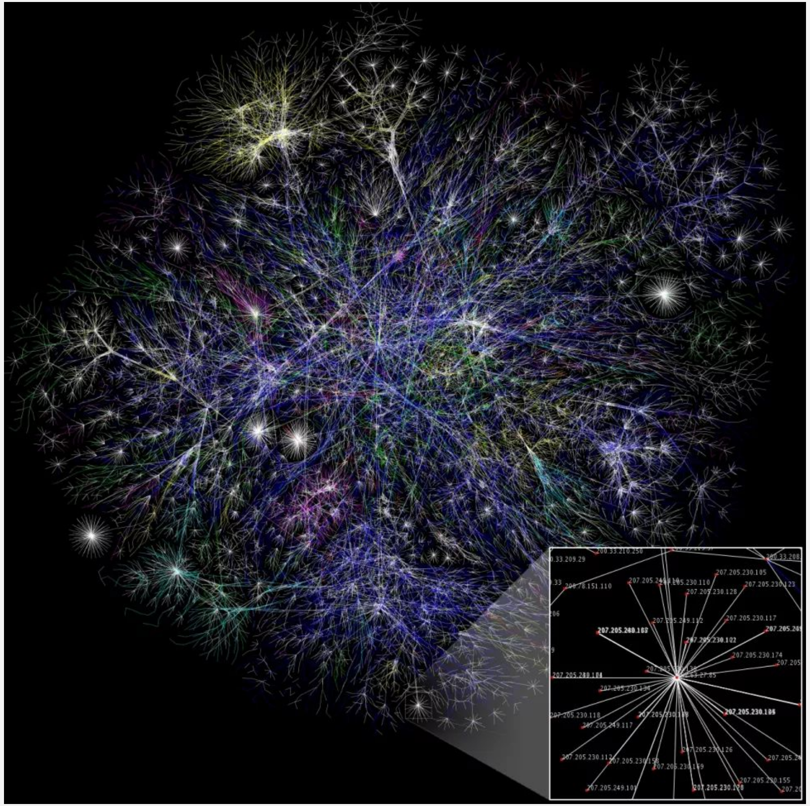
RELATION TO BINARY TREES

- Binary trees provide a very useful way of representing relationships in which a **hierarchy** exists.
- A node is pointed to by at most one other node (its **parent**), and each node points to at most two other nodes (its **children**).
- If we remove the restriction that each node can have at most two children, we have a general tree.
- If we also **remove the restriction that each node may have only one parent node**, we have a data structure called a **Graph**.

RELATION TO BINARY TREES

- A data structure that consists of a set of **nodes** and a set of **edges** that relate the nodes to one another.
- Vertex: A node in graph.
- Edge (arc): A pair of vertices representing a connection between two nodes in graph.

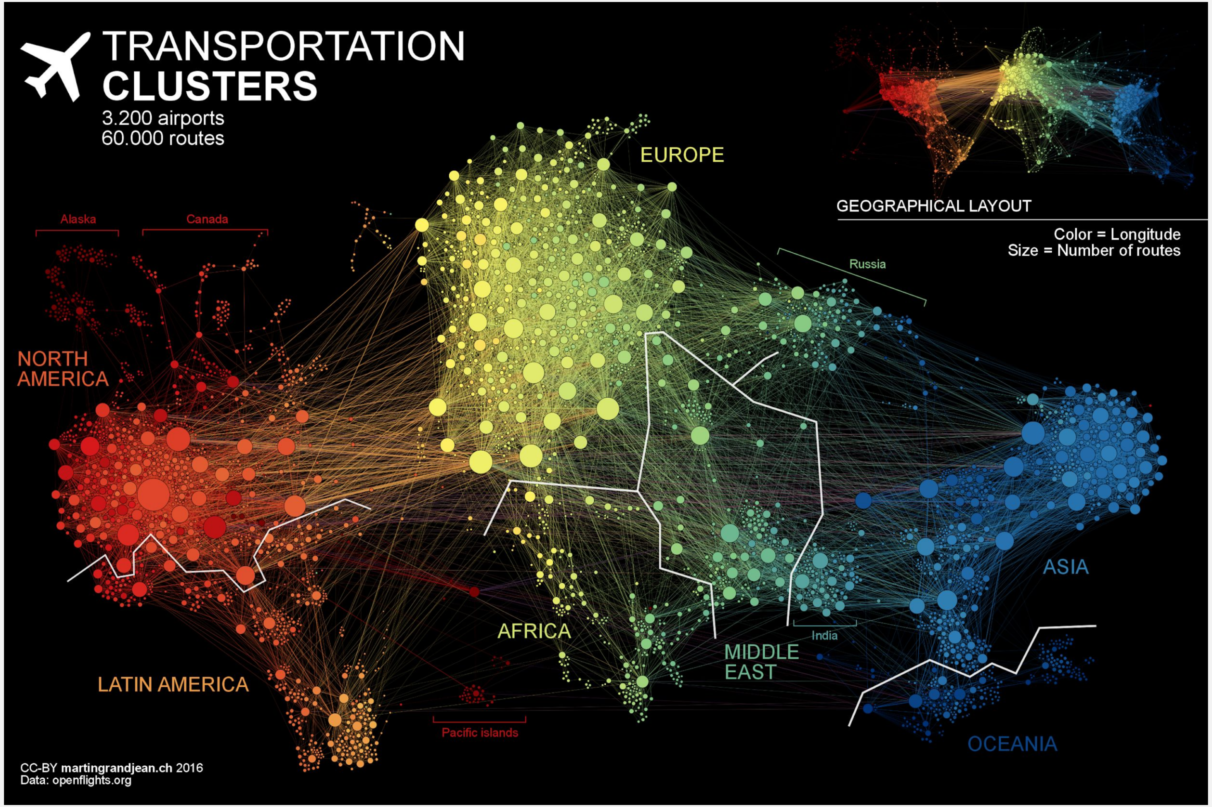
GRAPH EXAMPLES



Partial graph of the Internet (in 2005), where each “node” is an IP address, and the “edges” between them reveal connectivity delays (shorter lines = closer IP addresses)

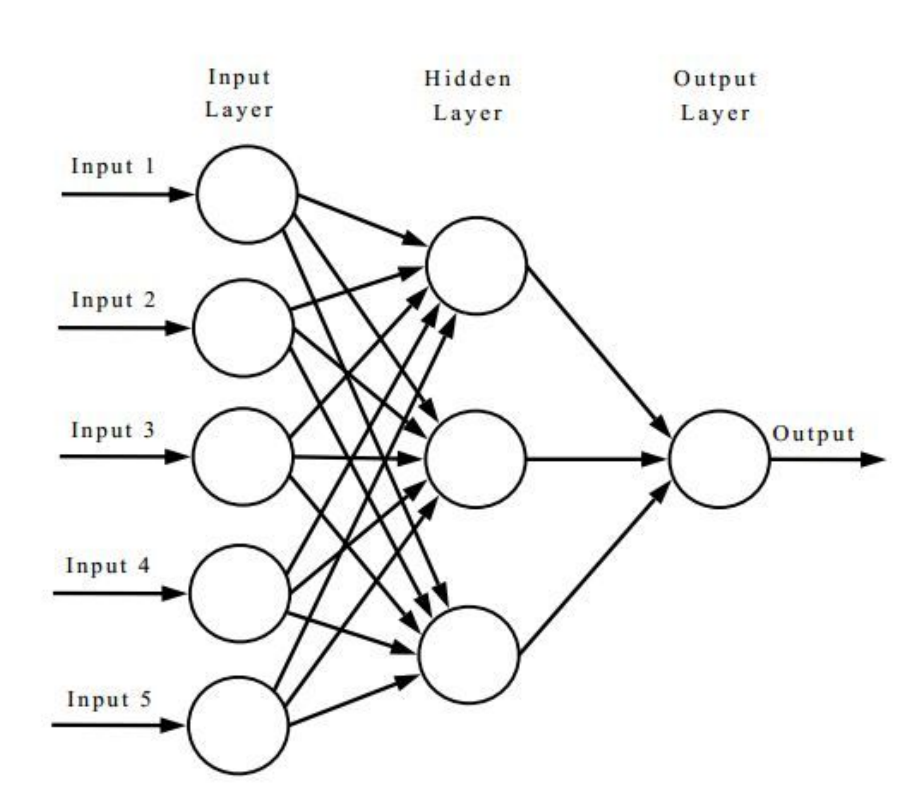
GRAPH EXAMPLES

Each "node" is an airport, and flight routes are represented by the "edge" in between them



GRAPH EXAMPLES

Neural networks! Each "node" represents a module of the neural network, and "edge" represent output/input relationships



GRAPH EXAMPLES

Graph of characters in the third book of Game of Thrones, where each "node" is a character, and "edge" reveal frequency of interaction (i.e. 2 names appearing within 15 words of one another).

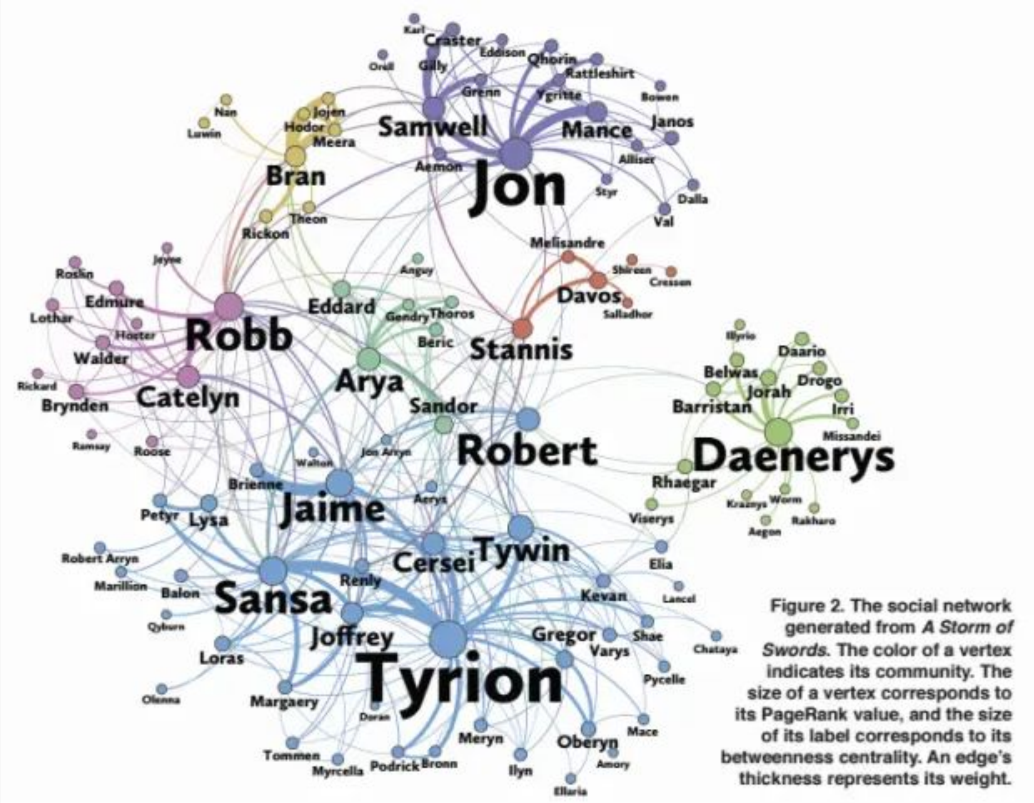
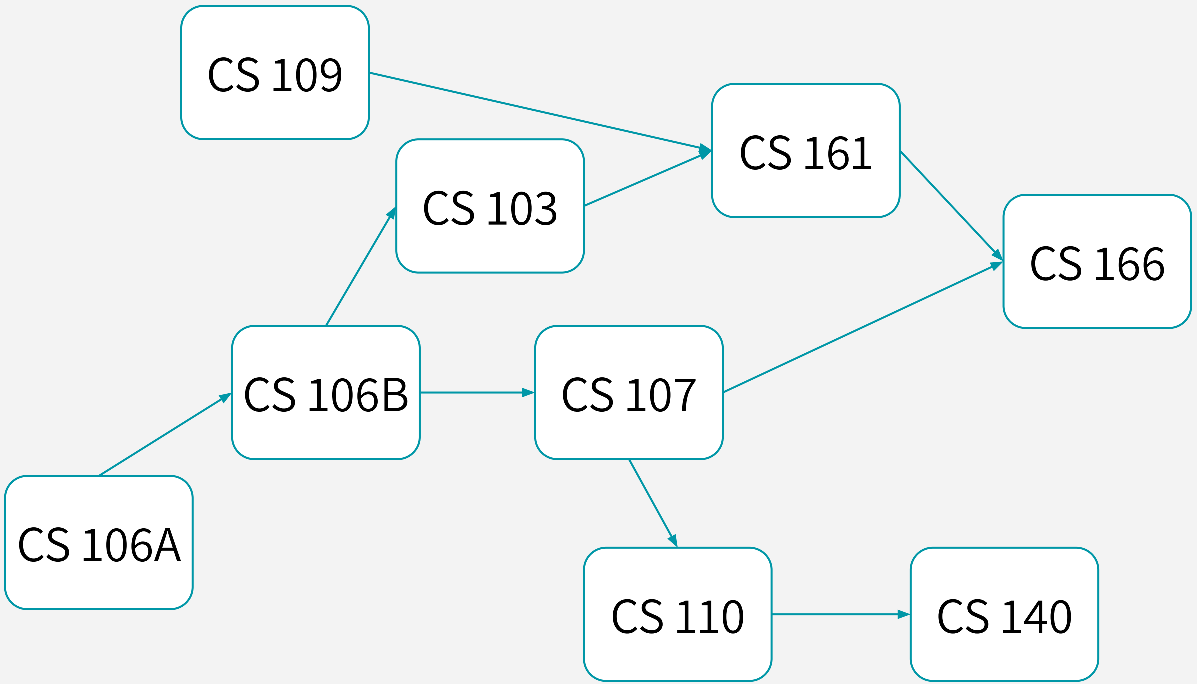


Figure 2. The social network generated from *A Storm of Swords*. The color of a vertex indicates its community. The size of a vertex corresponds to its PageRank value, and the size of its label corresponds to its betweenness centrality. An edge's thickness represents its weight.

GRAPH EXAMPLES

CS prerequisites!
“nodes” are classes
and an “edge” from
class A to class B
means “class B
depends on class A”



WHAT ARE GRAPHS USED FOR?

- There are a lot of diverse problems that can be represented as graphs, and we want to answer questions about them
- For example:
    - How do we most efficiently route packets across the internet?
    - Are there natural “clusters” or “communities” in a graph?
    - Which character(s) are least related with ______?
    - How should I sign up for classes without violating pre-req constraints?

But first off, some terminology!

WHAT ARE GRAPHS USED FOR?

- Recall (Graph): A set of vertices V and a set of edges E
  - V(G) is a finite, nonempty set of vertices
  - E(G) is a set of edges (written as pairs of vertices)
- G = (V, E)
- To specify the set of vertices, we list them in set notation.

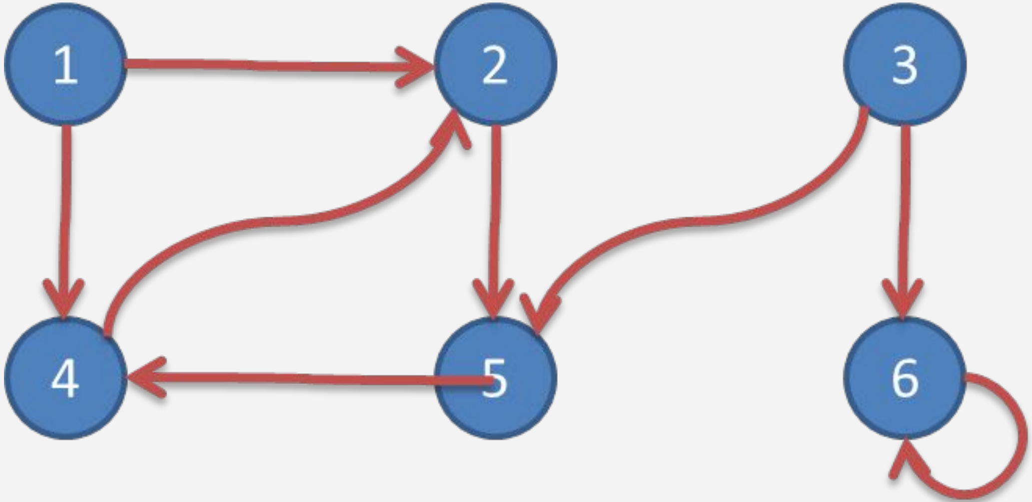
  V = {1, 2, 3, 4, 5, 6}

  E = {(1, 2), (1, 4), (2, 5), (3, 6), (3, 5), (4, 2), (5, 4), (6, 6)}

GRAPHS INTRODUCTION

V = {1, 2, 3, 4, 5, 6}

E = {(1, 2), (1, 4), (2, 5), (3, 6), (3, 5), (4, 2), (5, 4), (6, 6)}



GRAPHS INTRODUCTION

Undirected graph

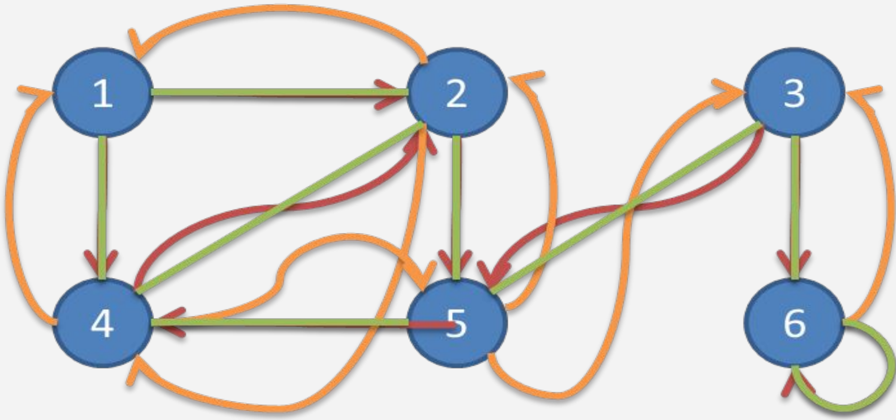
- A special graph
- If $v_i \neq v_j$, $<v_i, v_j> \in E$, $<v_j, v_i> \in E$

E.g. **V** = {**1, 2, 3, 4, 5, 6**}

E = {**(1, 2), (1, 4), (2, 5), (3, 6), (3, 5), (4, 2), (5, 4), (6, 6)**}

∪ {(2, 1), (4, 1), (5, 2), (6, 3), (5, 3), (2, 4), (4, 5)}

Use a undirected line to indicate a pair of edges or a self edge

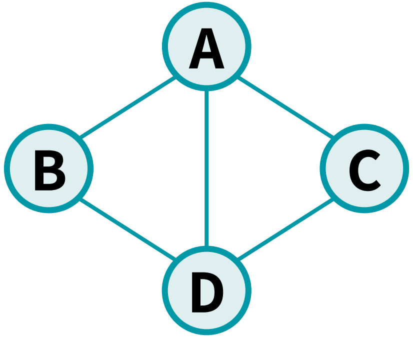


2 KINDS OF GRAPHS

UNDIRECTED GRAPHS

An undirected graph has
a set of vertices (V) & a set of edges (E)

Formally,
G = (V, E)



V = {A, B, C, D}
E = { (A, B), (A, C), (A, D), (B, D), (C, D)}

The road which links Karachi with HyderAbad also links HyderAbad with Karachi. ***The road has no direction***.

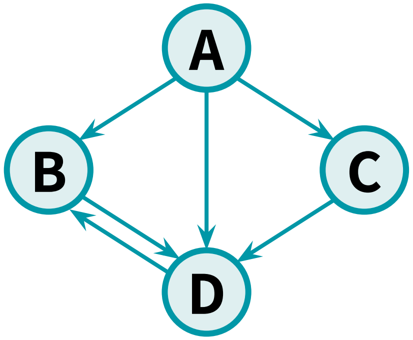
These are graphs where edges aren't assigned weights, or all edges are assumed to have the same weight.

2 KINDS OF GRAPHS

A flight from Karachi to Singapore does not guarantee a flight from Karachi to Singapore.

DIRECTED GRAPHS

A directed graph has
a set of vertices (V) & a set of **DIRECTED** edges (E)



Formally,
G = (V, E)

V = {A, B, C, D}
E = { [A, B], [A, C], [A, D], [B, D], [C, D], [D, B]}

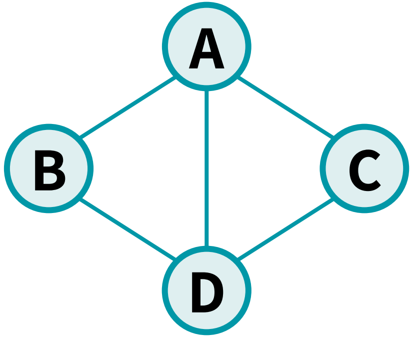
2 KINDS OF GRAPHS

A directed graph is also known as **digraph**

UNDIRECTED GRAPHS

An undirected graph has
a set of vertices (V) & a set of edges (E)

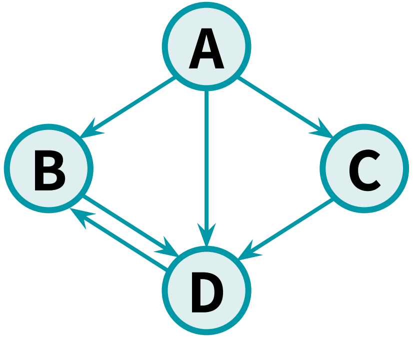
Formally,
G = (V, E)



The **degree** of vertex D is 3
Vertex D's **neighbors** are A, B, and C

DIRECTED GRAPHS

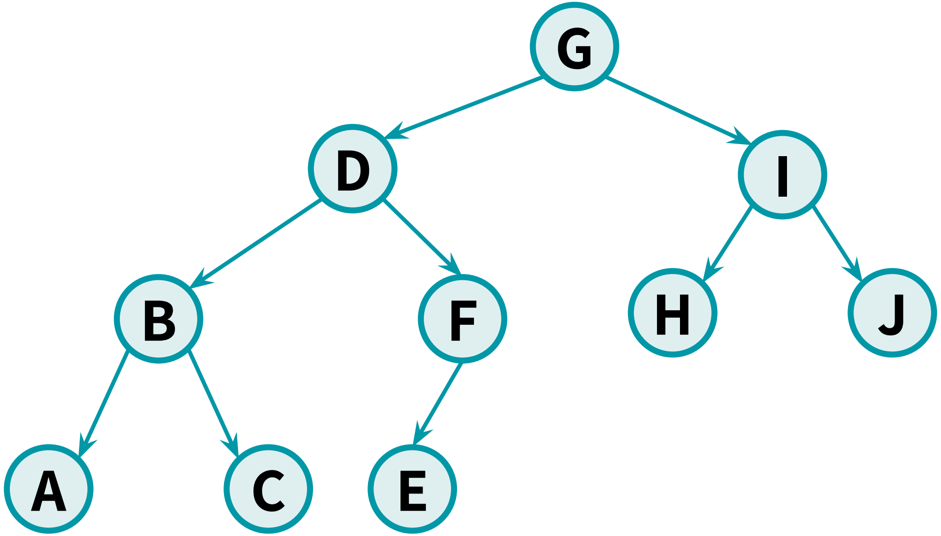
A directed graph has
a set of vertices (V) & a set of **DIRECTED** edges (E)



Formally,
G = (V, E)

The **in-degree** of vertex D is 3. The **out-degree** of vertex D is 1.
Vertex D's **incoming neighbors** are A, B, & C
Vertex D's **outgoing neighbor** is B

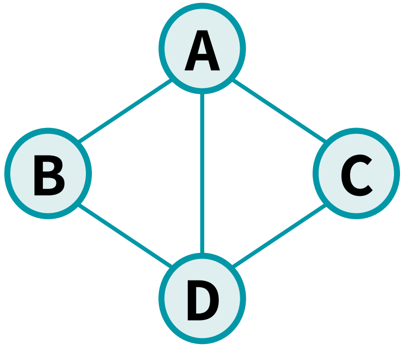
2 KINDS OF GRAPHS



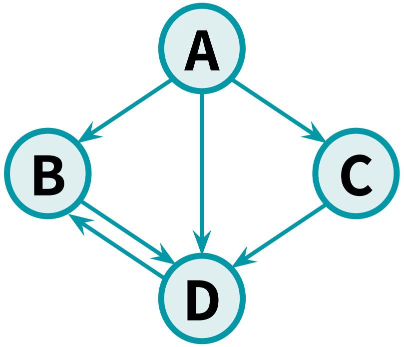
V = {A, B, C, D, E, F, G, H, I, J}
E = {(G, D), (G, I), (D, B), (D, F), (I, H), (I, J), (B, A), (B, C), (F, E)}

WHAT ARE GRAPHS USED FOR?

- If two vertices in a graph are connected by an edge, they are said to be, **adjacent**.
- If the vertices are connected by a directed edge, then the first vertex is said to be **adjacent to** the second, and the second vertex is said to be **adjacent from.**



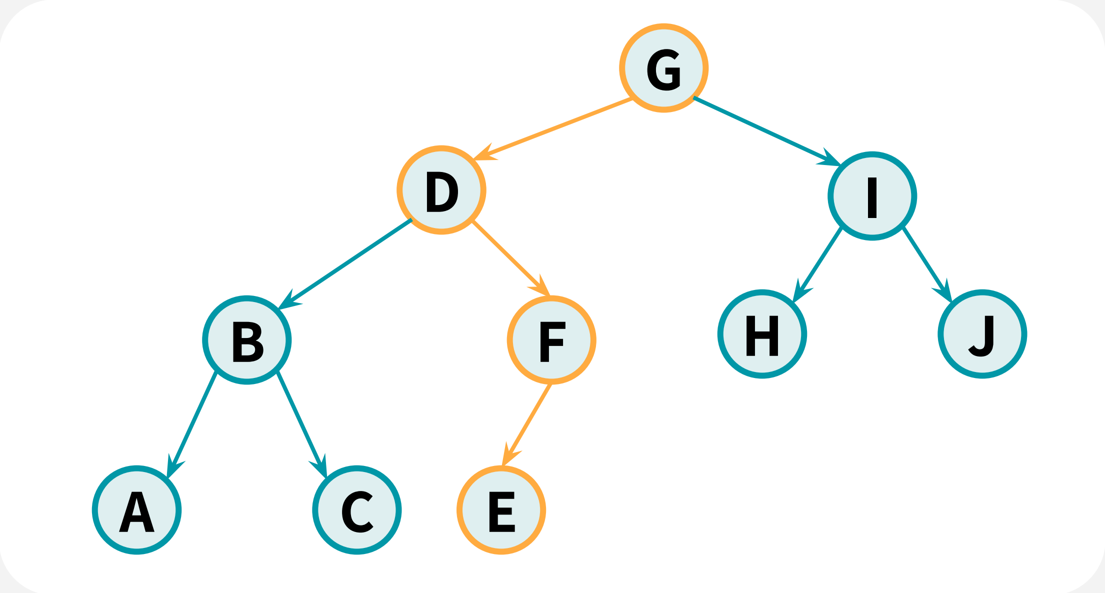
Graph A



Graph B

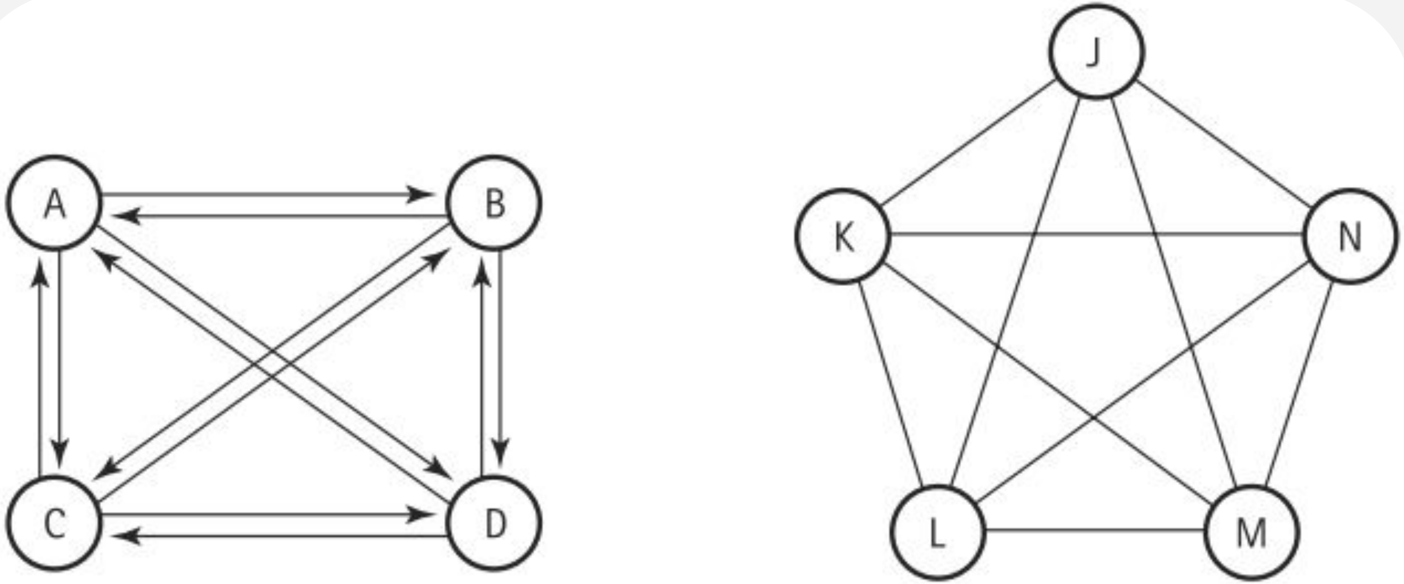
GRAPH TERMINOLOGIES

- A **path** from one vertex to another consists of a sequence of vertices that connect them. For a path to exist, an uninterrupted sequence of edges must go from the first vertex, through any number of vertices, to the second vertex.



GRAPH TERMINOLOGIES

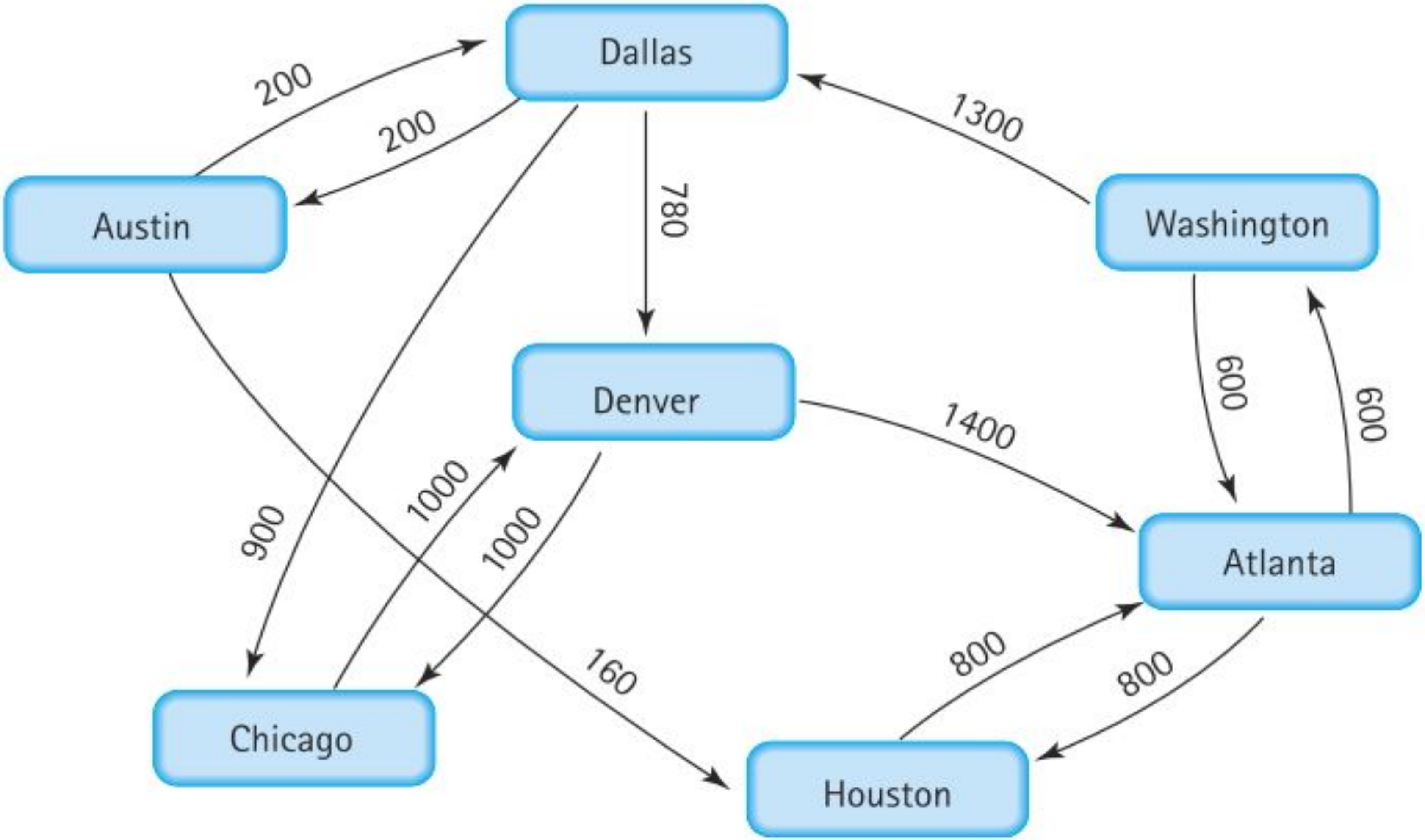
- **Complete Graph** is such a graph where every vertex is adjacent to every other vertex.



(a) Complete directed graph. (b) Complete undirected graph.

GRAPH TERMINOLOGIES

In a **Weighted Graph,** each edge carries a value. Weighted graphs can be used to represent applications in which the **value of the connection between the vertices is important, not just the existence of a connection.**



Dallas
Austin
Washington
Denver
Atlanta
Chicago
Houston
200
200
1300
780
600
600
1400
900
1000
1000
160
800
800

GRAPH TERMINOLOGIES

Adjacency Matrix

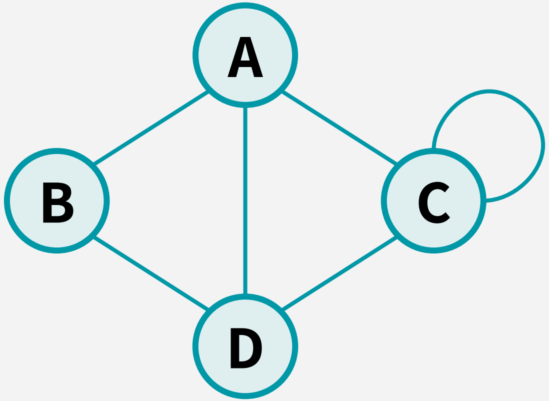
For a graph with **V** vertices, a **V x V** table that shows the existence of all edges in the graph.

A simple way to represent V(graph), the vertices in the graph, is with an array where the elements are of the type of the vertices (VertexType).

Simply its a two-dimensional array of edge in the graph values (weights).

GRAPH REPRESENTATIONS

OPTION 1: **ADJACENCY MATRIX**



(An undirected graph)

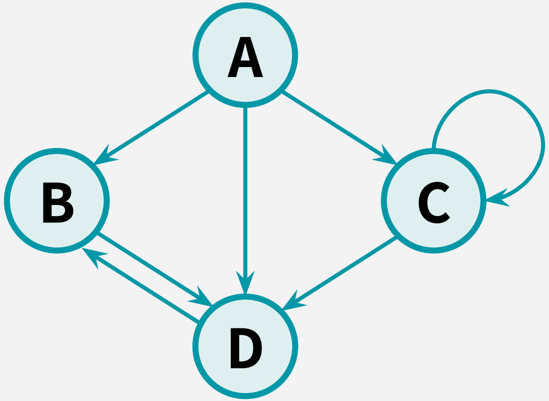
(destination)

(source)

$$\begin{array}{cc} & \begin{array}{cccc} \mathbf{A} & \mathbf{B} & \mathbf{C} & \mathbf{D} \end{array} \\ \begin{array}{c} \mathbf{A} \\ \mathbf{B} \\ \mathbf{C} \\ \mathbf{D} \end{array} & \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

GRAPH REPRESENTATIONS

OPTION 1: **ADJACENCY MATRIX**



(A directed graph)

(destination)

(source)

$$\begin{array}{c} \\ A \\ B \\ C \\ D \end{array}\begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix} \end{array}$$

[0]	"Atlanta "
[1]	"Austin "
[2]	"Chicago "
[3]	"Dallas "
[4]	"Denver "
[5]	"Houston "
[6]	"Washington"
[7]	
[8]	
[9]	

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
[0]	0	0	0	0	0	800	600	•	•	•
[1]	0	0	0	200	0	160	0	•	•	•
[2]	0	0	0	0	1000	0	0	•	•	•
[3]	0	200	900	0	780	0	0	•	•	•
[4]	1400	0	1000	0	0	0	0	•	•	•
[5]	800	0	0	0	0	0	0	•	•	•
[6]	600	0	0	1300	0	0	0	•	•	•
[7]	•	•	•	•	•	•	•	•	•	•
[8]	•	•	•	•	•	•	•	•	•	•
[9]	•	•	•	•	•	•	•	•	•	•

ADJACENCY MATRIX

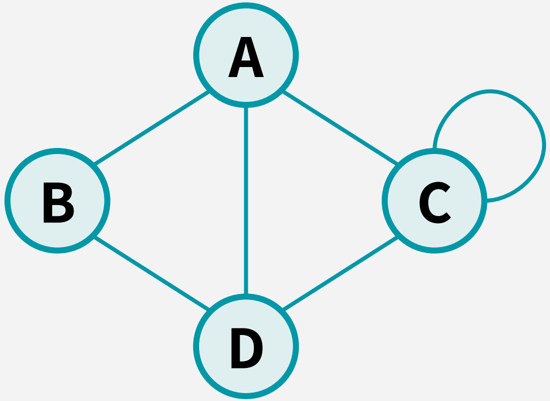
- The advantages to representing the edges in a graph with an adjacency matrix relate to its speed and simplicity.
- Given the indexes of two vertices, determining the existence (or the weight) of an edge between them is an **$O(1)$** operation.
- The problem with adjacency matrices is that their use of space is **$O(V^2)$**,
- If the maximum number of vertices is large than adjacency matrices may waste of a lot of space.

ADJACENCY LIST

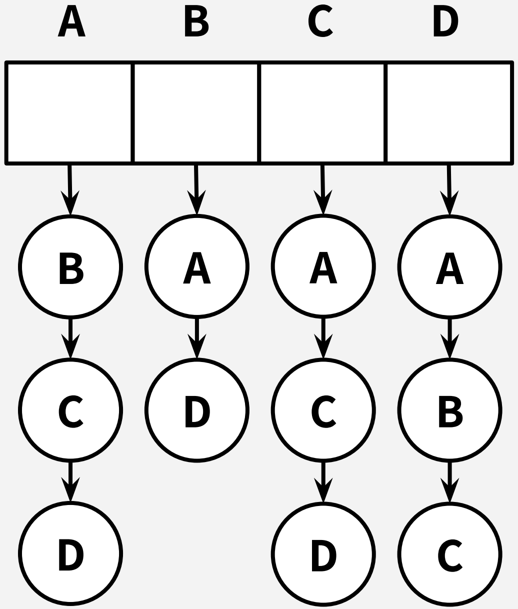
- Recall, we have tried to **save space by allocating memory as we need it** at run time, using linked structures.
- We can use similar approach for Graphs.
- Adjacency Lists are linked structure, **one list per vertex**, **that identify the vertices to which each vertex is connected.**

GRAPH REPRESENTATIONS

OPTION 2: **ADJACENCY LISTS**



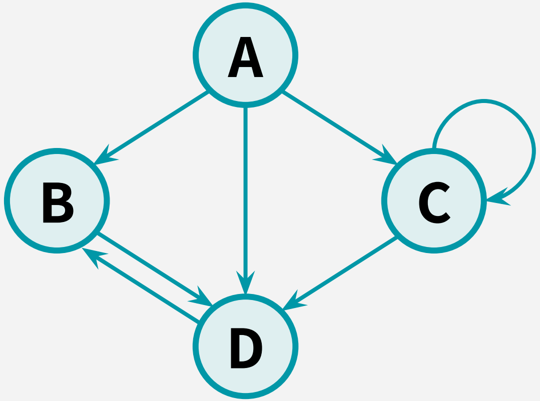
(An undirected graph)



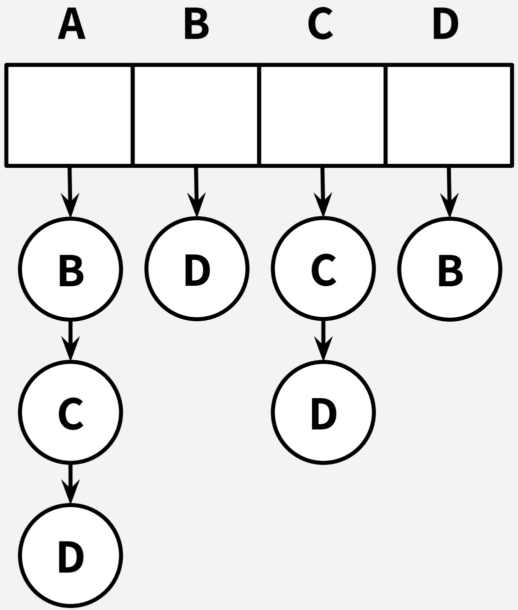
Each list stores a node's neighbors

GRAPH REPRESENTATIONS

OPTION 2: **ADJACENCY LISTS**

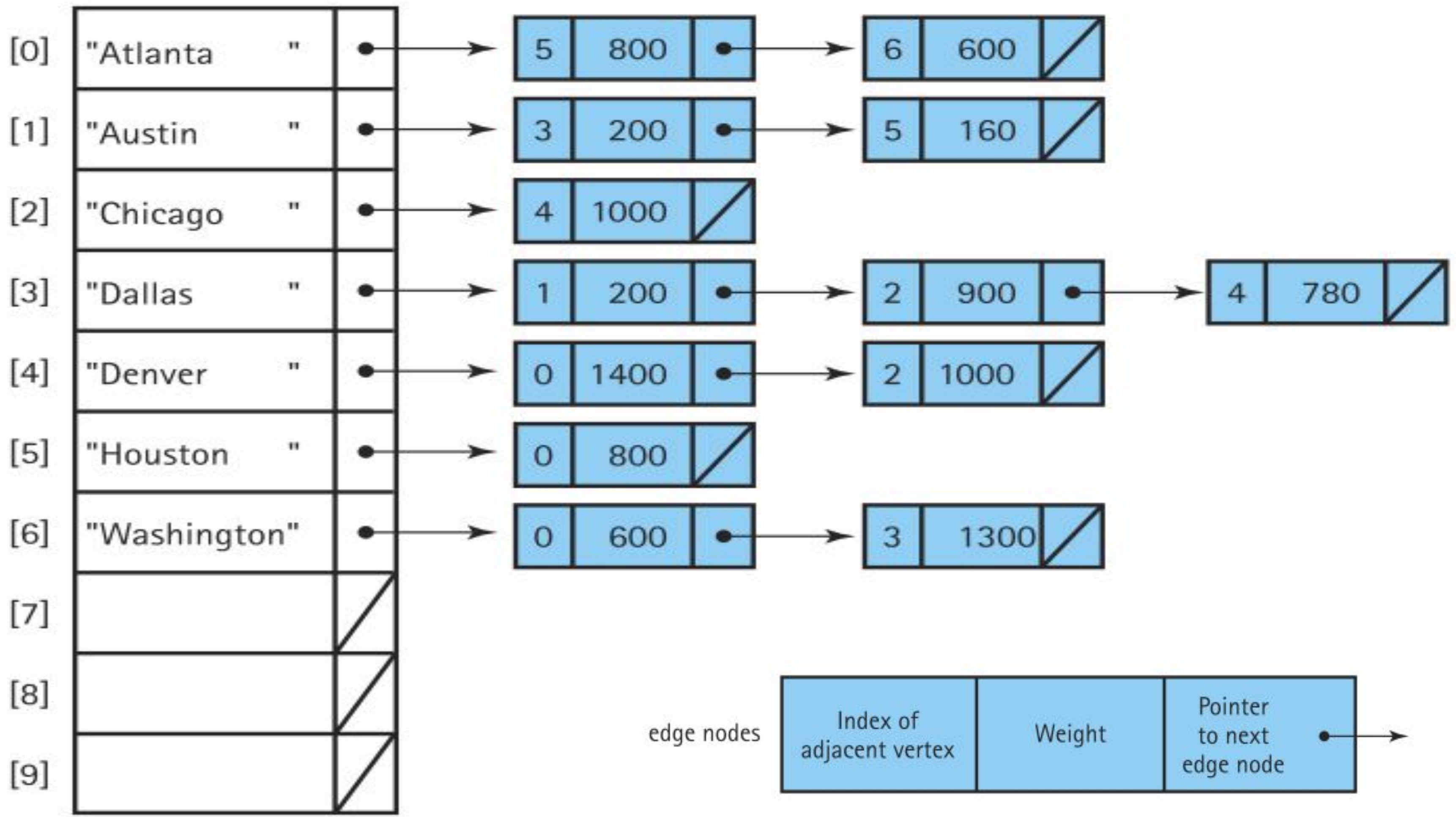


(A directed graph)



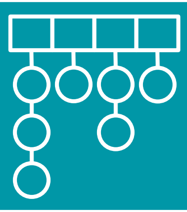
Tracks *outgoing neighbors.*

(You could also do the same for incoming neighbors as well)

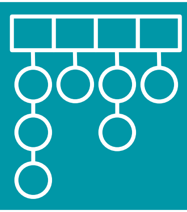


[0] "Atlanta " 5 800 6 600
[1] "Austin " 3 200 5 160
[2] "Chicago " 4 1000
[3] "Dallas " 1 200 2 900 4 780
[4] "Denver " 0 1400 2 1000
[5] "Houston " 0 800
[6] "Washington" 0 600 3 1300
[7]
[8]
[9]
edge nodes
Index of adjacent vertex
Weight
Pointer to next edge node

GRAPH REPRESENTATIONS

For a graph $G = (V, E)$ where $\|V\| = \mathbf{n}$, and $\|E\| = \mathbf{m}$	$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$	
EDGE MEMBERSHIP Is $e = \{v, w\}$ in E?	**O(1)**	
NEIGHBOR QUERY Give me v's neighbors	**O(n)**	
SPACE REQUIREMENTS	**$O(n^2)$**	

GRAPH REPRESENTATIONS

For a graph $G = (V, E)$ where $\lvert V \rvert = \mathbf{n}$, and $\lvert E \rvert = \mathbf{m}$	$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$	
EDGE MEMBERSHIP Is $e = \{v, w\}$ in E?	**O(1)**	**O(deg(v))** or **O(deg(w))**
NEIGHBOR QUERY Give me v's neighbors	**O(n)**	**O(deg(v))**
SPACE REQUIREMENTS	**$O(n^2)$**	**O(n + m)**

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

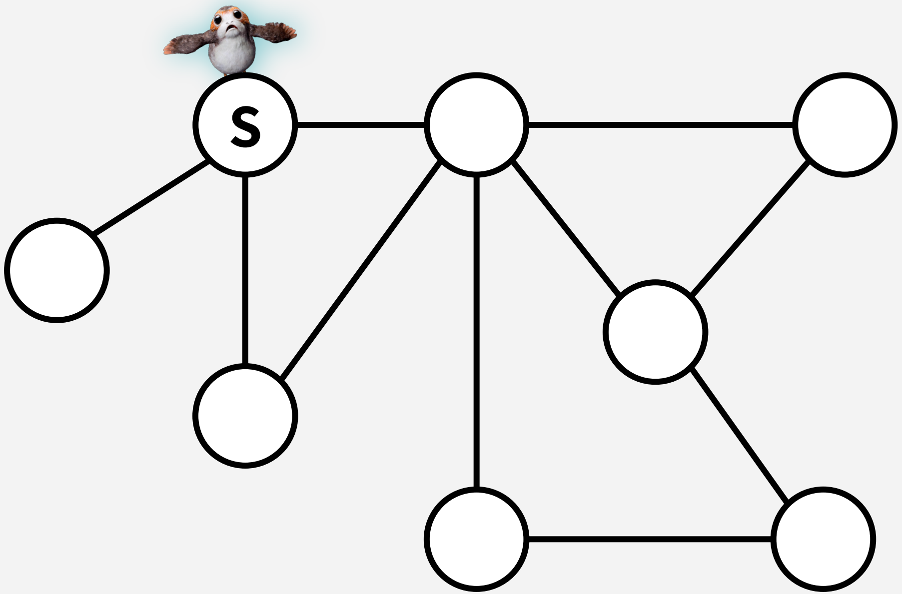
We'll assume this representation, unless otherwise stated.

BREADTH-FIRST SEARCH

One way to explore a graph!

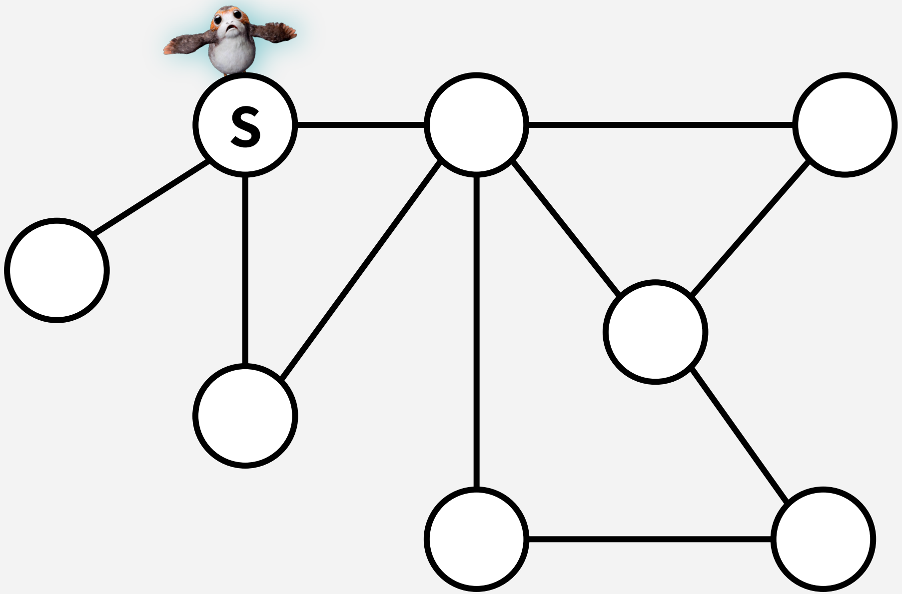
BREADTH-FIRST SEARCH

An analogy:
A bird is exploring a labyrinth from above (with a bird's eye view)



BREADTH-FIRST SEARCH

An analogy:
A bird is exploring a labyrinth from above (with a bird's eye view)



unvisited

Layer 0: reachable in 0 steps

Layer 1: reachable in 1 step

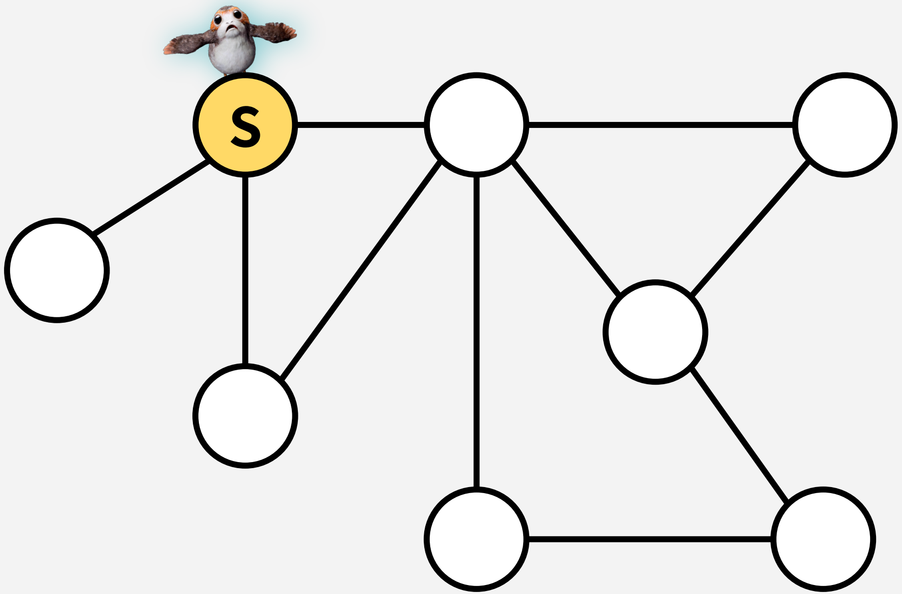
Layer 2: reachable in 2 steps

Layer 3: reachable in 3 steps

BREADTH-FIRST SEARCH

An analogy:

A bird is exploring a labyrinth from above (with a bird's eye view)



unvisited

Layer 0: reachable in 0 steps

Layer 1: reachable in 1 step

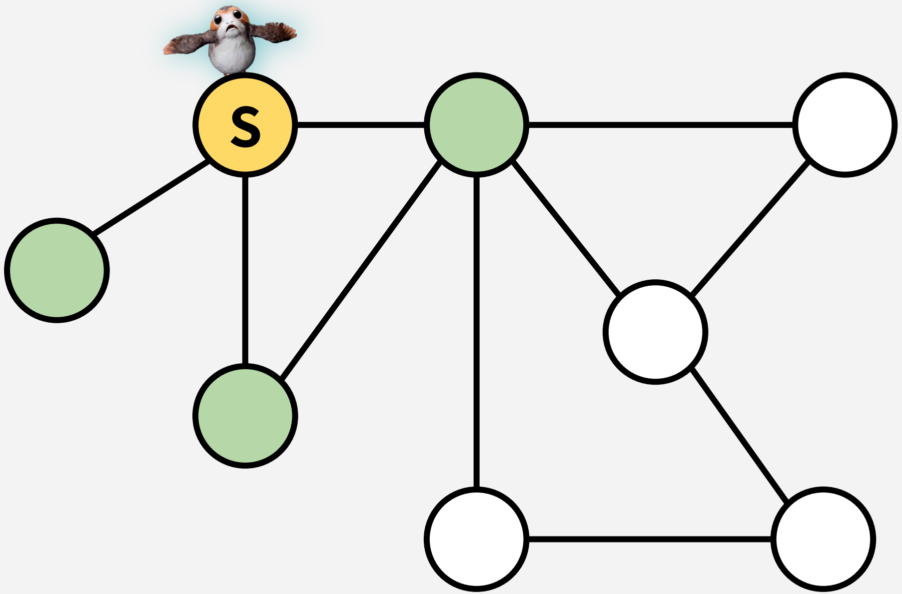
Layer 2: reachable in 2 steps

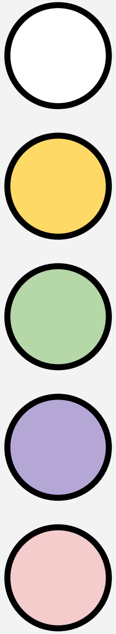
Layer 3: reachable in 3 steps

BREADTH-FIRST SEARCH

An analogy:

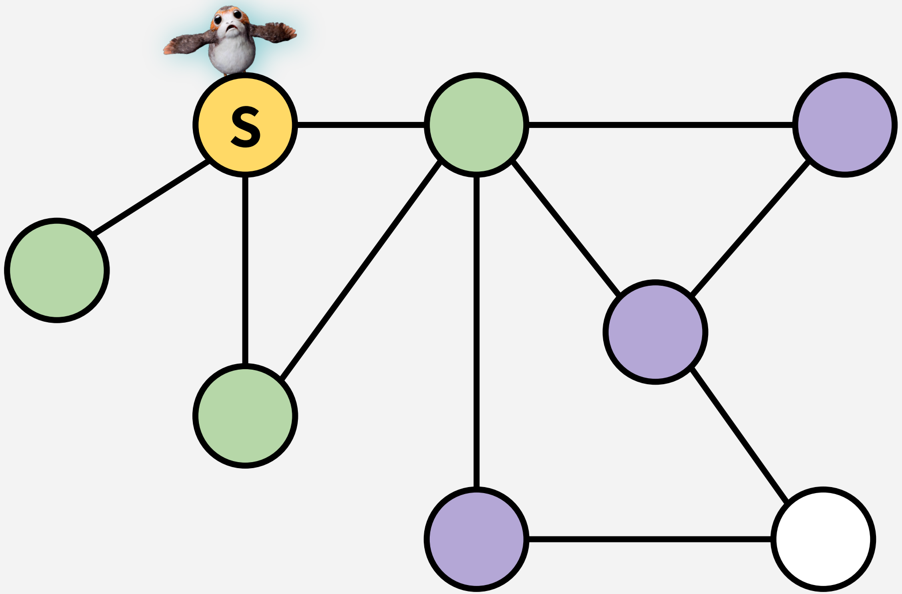
A bird is exploring a labyrinth from above (with a bird's eye view)





BREADTH-FIRST SEARCH

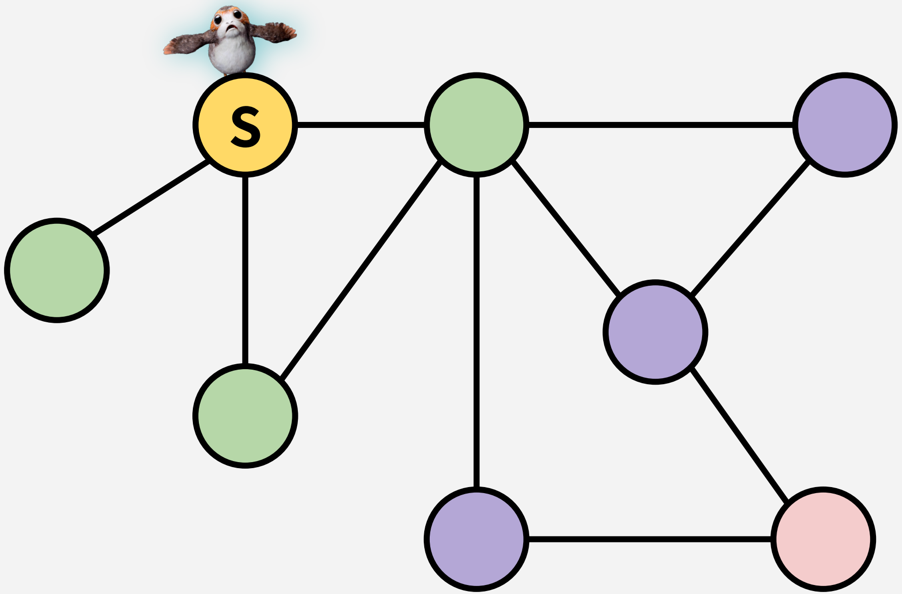
An analogy:
A bird is exploring a labyrinth from above (with a bird's eye view)



BREADTH-FIRST SEARCH

An analogy:

A bird is exploring a labyrinth from above (with a bird's eye view)



unvisited

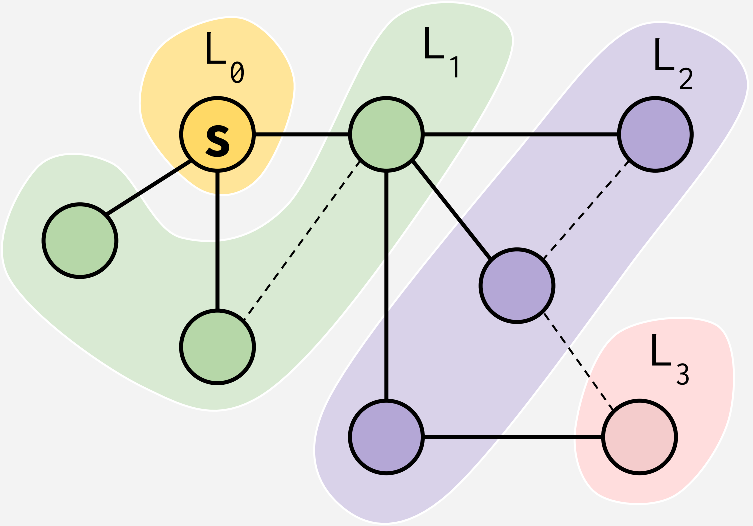
Layer 0: reachable in 0 steps

Layer 1: reachable in 1 step

Layer 2: reachable in 2 steps

Layer 3: reachable in 3 steps

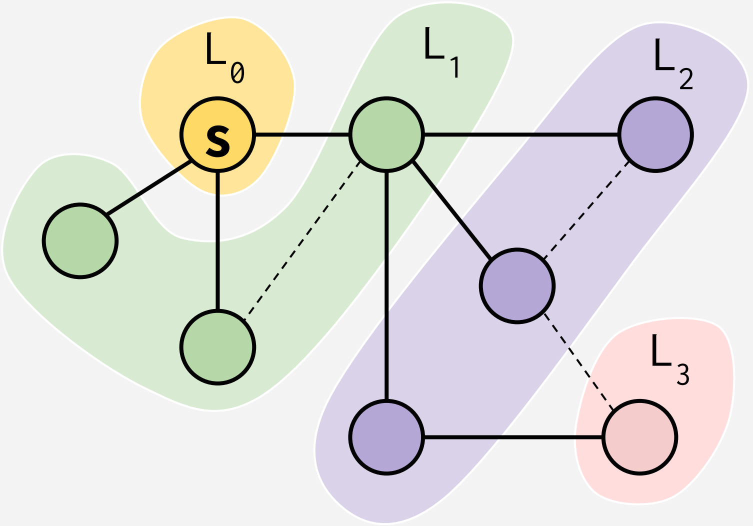
BREADTH-FIRST SEARCH




```
BFS(s):
    Set L_i = [] for i = 0, ..., n-1
    L_0 = s
    for i = 0, ..., n-1:
        for u in L_i:
            for v in u.neighbors:
                if v not yet visited:
                    mark v as visited
                    add v to L_i+1
```

L_i = The set of nodes we can reach in i steps from s

BREADTH-FIRST SEARCH

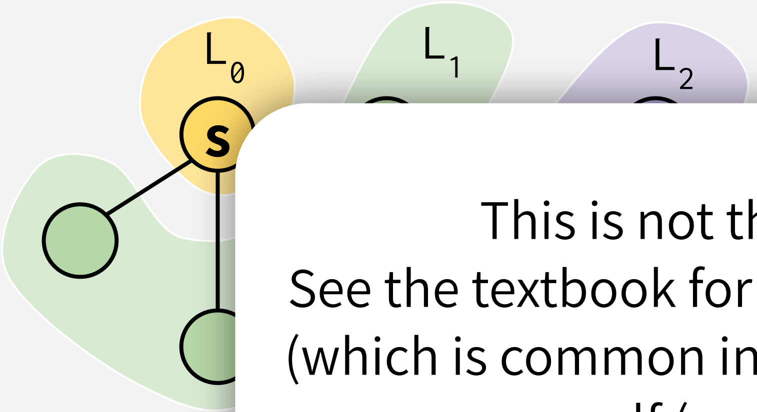


L_i = The set of nodes we can reach in i steps from s

```
BFS(s):
    Set L_i = [] for i = 0, ..., n-1
    L_0 = s
    for i = 0, ..., n-1:
        for u in L_i:
            for v in u.neighbors:
                if v not yet visited:
                    mark v as visited
                    add v to L_i+1
```

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

BREADTH-FIRST SEARCH




```
BFS(s):
                                       ..., n-1
                                          rs:
                                         sited:
                                         sited
```

This is not the only way to write BFS!
See the textbook for a version that uses a FIFO **queue** (which is common in the real world), and try writing it yourself (great for interview practice)

L_i = The set of nodes we can reach in i steps from s

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

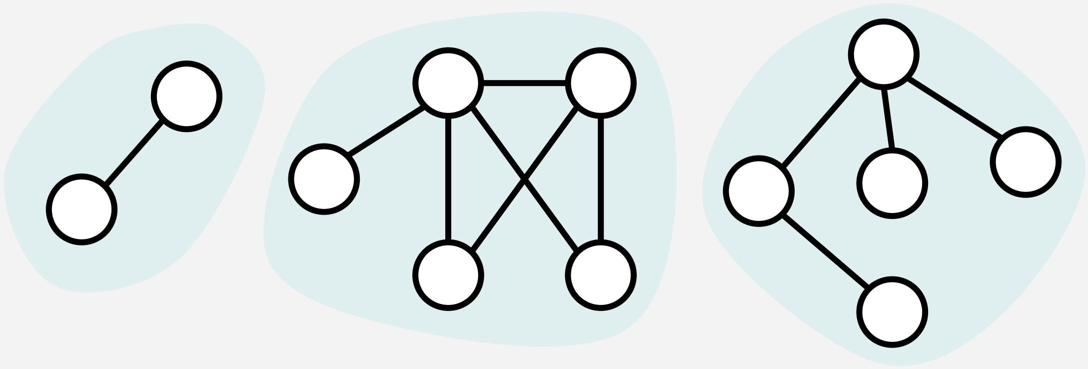
BREADTH-FIRST SEARCH

BFS finds all the nodes reachable from the starting point!

BREADTH-FIRST SEARCH

BFS finds all the nodes reachable from the starting point!

In undirected graphs, this is equivalent to finding the node's **connected component.**



BREADTH-FIRST SEARCH

Why is it called breadth-first?

We are implicitly building a **tree**!

(It's a tree because we never revisit a node)

We go as "broadly" as we can when building each layer of the tree

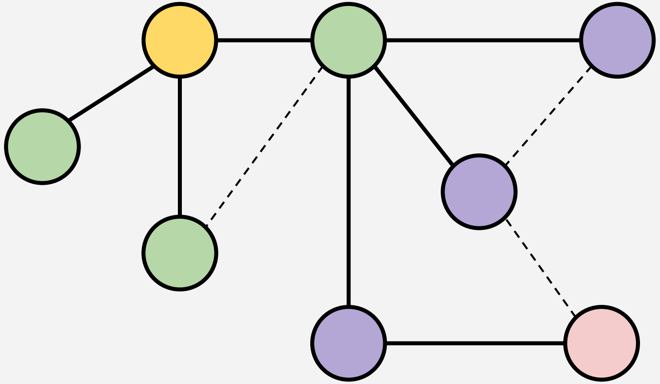
BREADTH-FIRST SEARCH

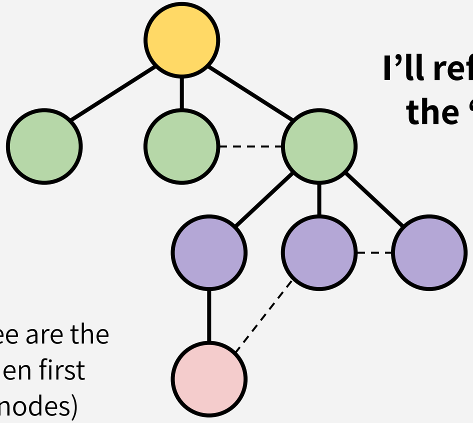
Why is it called breadth-first?

We are implicitly building a **tree**!

(It's a tree because we never revisit a node)

We go as "broadly" as we can when building each layer of the tree





I'll refer to this as the "BFS tree"

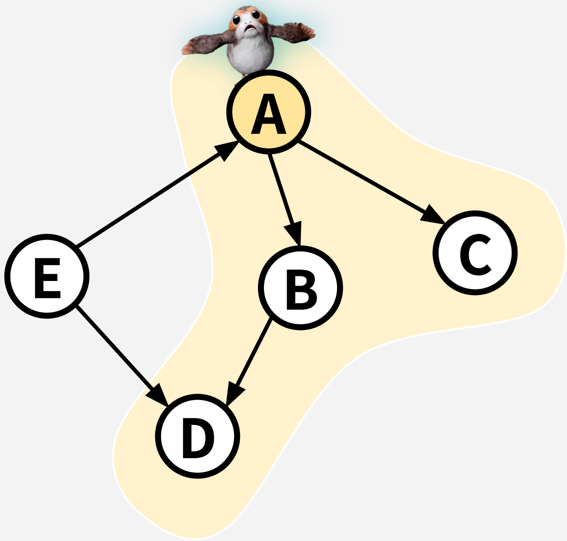
(Edges in the BFS tree are the ones traversed when first finding unvisited nodes)

BREADTH-FIRST SEARCH

BFS works fine on directed graphs too!

From a start node x, BFS would find all nodes ***reachable*** from x.

(In directed graphs, "connected component" isn't as well defined… more on that later!)



Verify this on your own: running BFS from A would still find all nodes reachable from A (E isn't reachable from A in this directed graph).

BREADTH-FIRST SEARCH

What are some applications of BFS?

Finding a node's connected component (just run BFS)!

Single-source shortest paths!

DEPTH-FIRST SEARCH

One way to explore a graph!

BFS vs. DFS

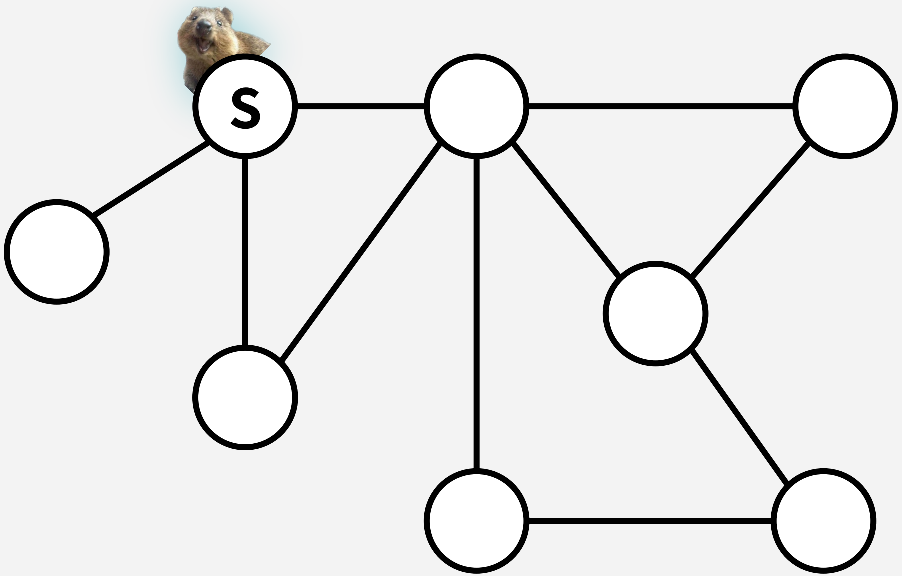
Literally just BREADTH vs DEPTH:

While BFS first explores the nodes closest to the “source” and then moves outwards in layers, DFS goes as far down a path as it can before it comes back to explore other options.

DEPTH-FIRST SEARCH

An analogy:

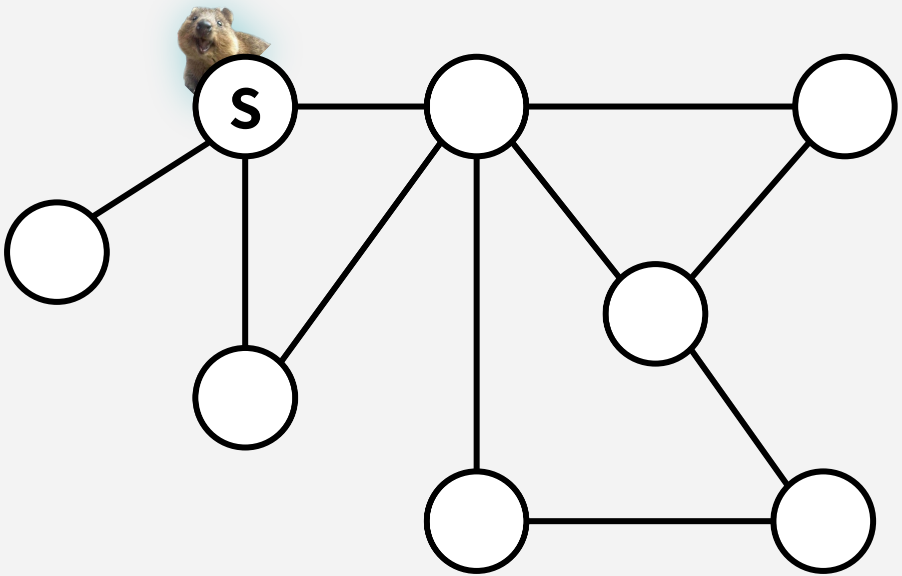
A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



DEPTH-FIRST SEARCH

An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



unvisited

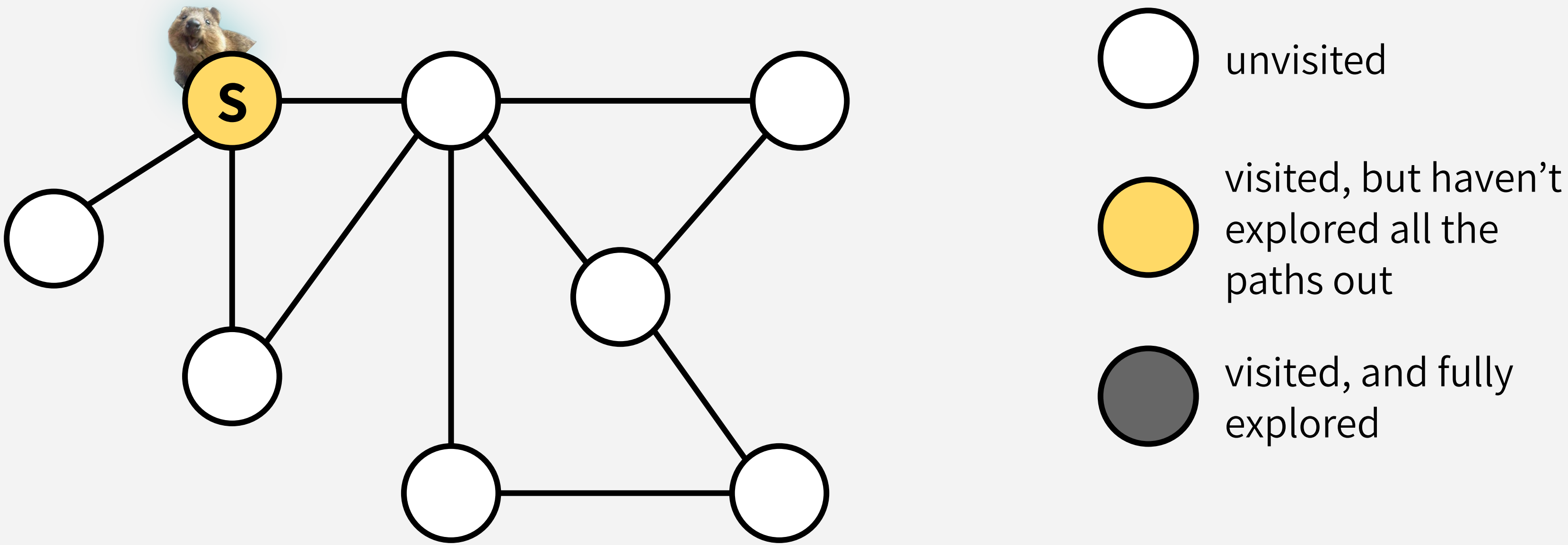
visited, but haven't explored all the paths out

visited, and fully explored

DEPTH-FIRST SEARCH

An analogy:

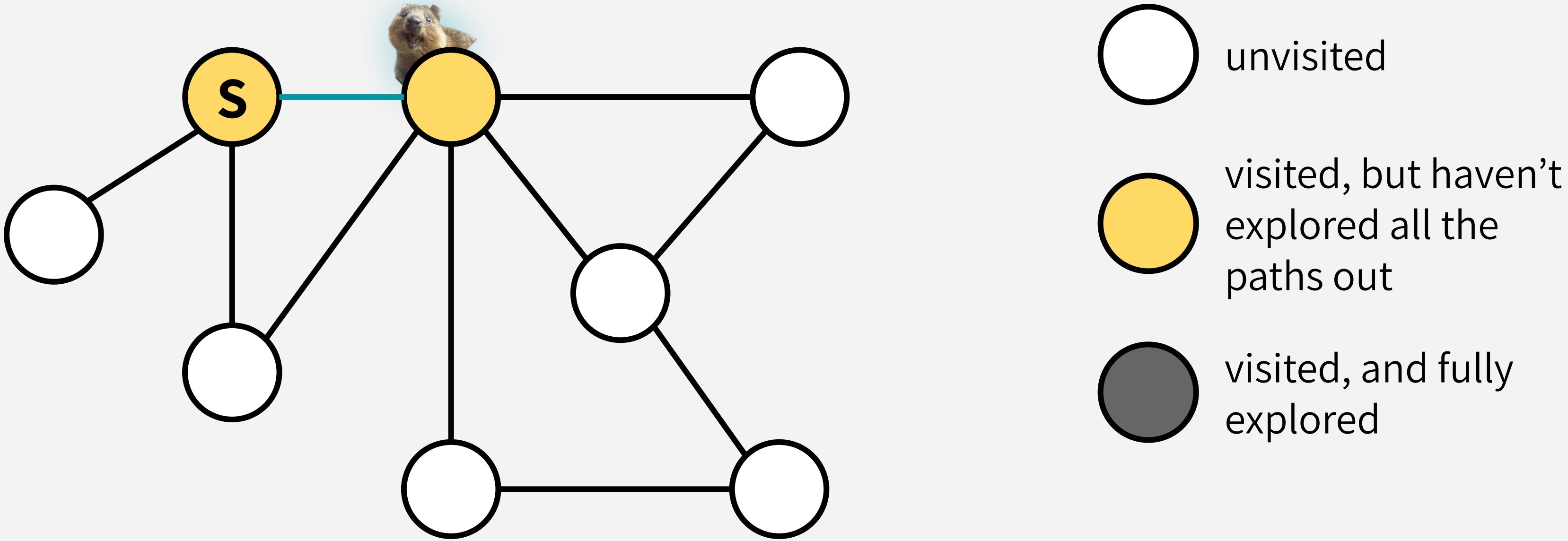
A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



DEPTH-FIRST SEARCH

An analogy:

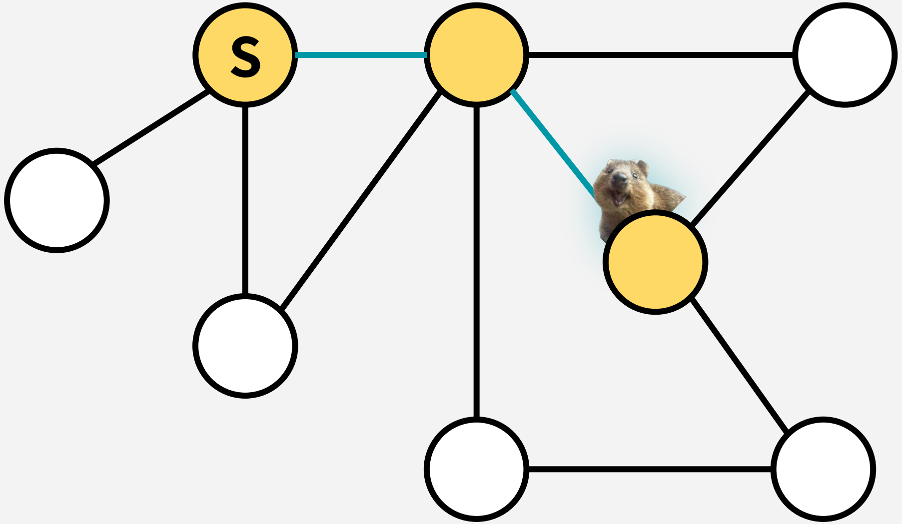
A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)

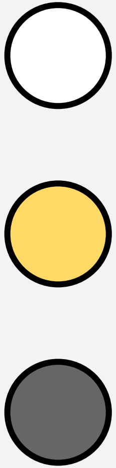


DEPTH-FIRST SEARCH

An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)

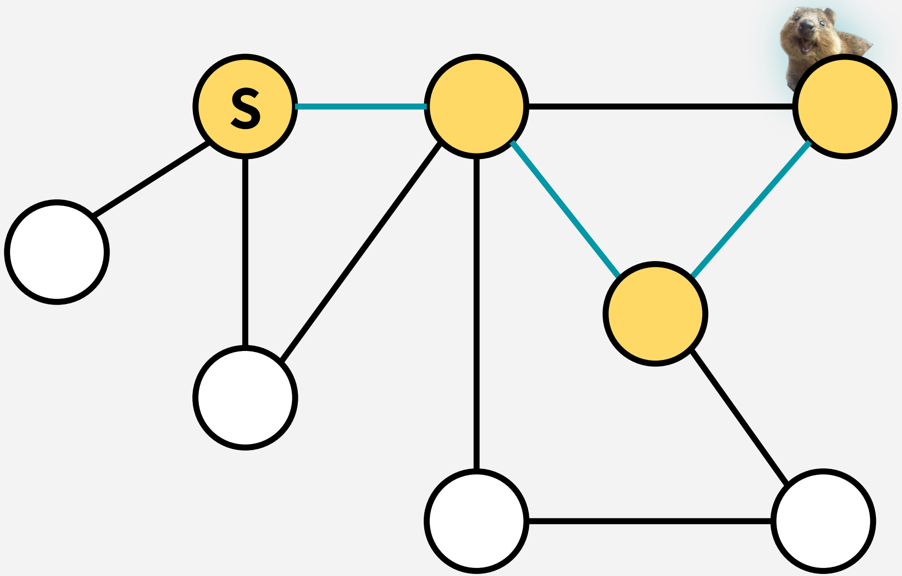




DEPTH-FIRST SEARCH

An analogy:

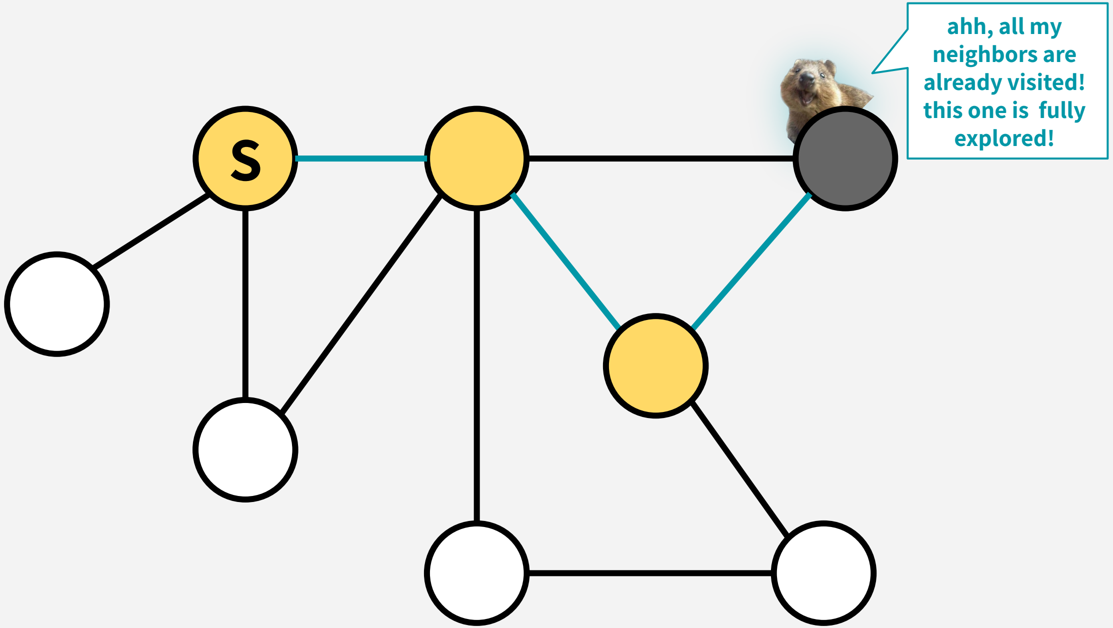
A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



DEPTH-FIRST SEARCH

An analogy:

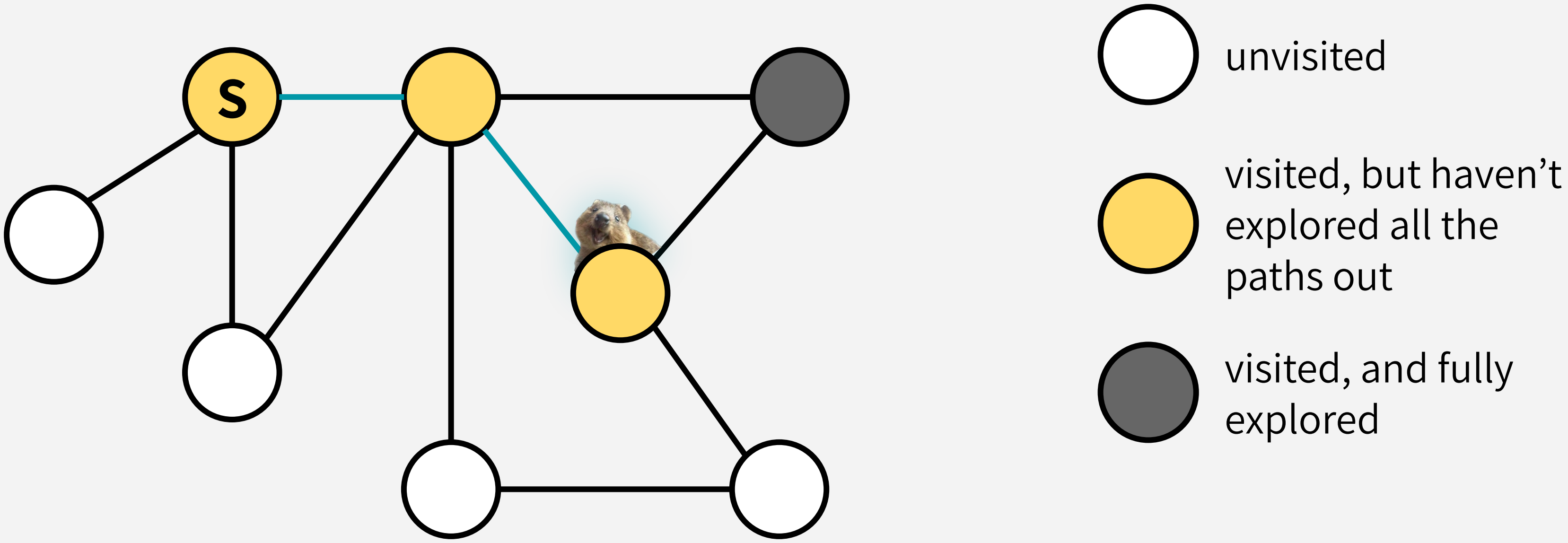
A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



DEPTH-FIRST SEARCH

An analogy:

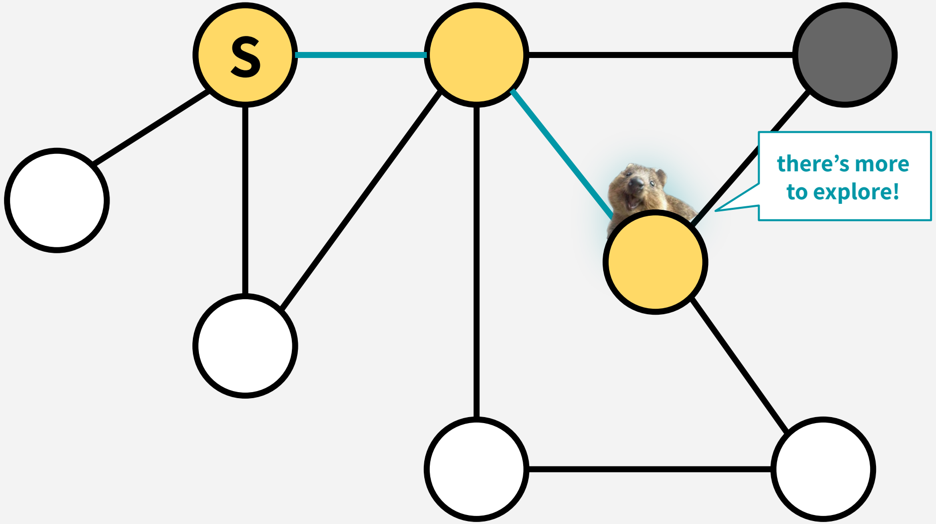
A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



DEPTH-FIRST SEARCH

An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



unvisited

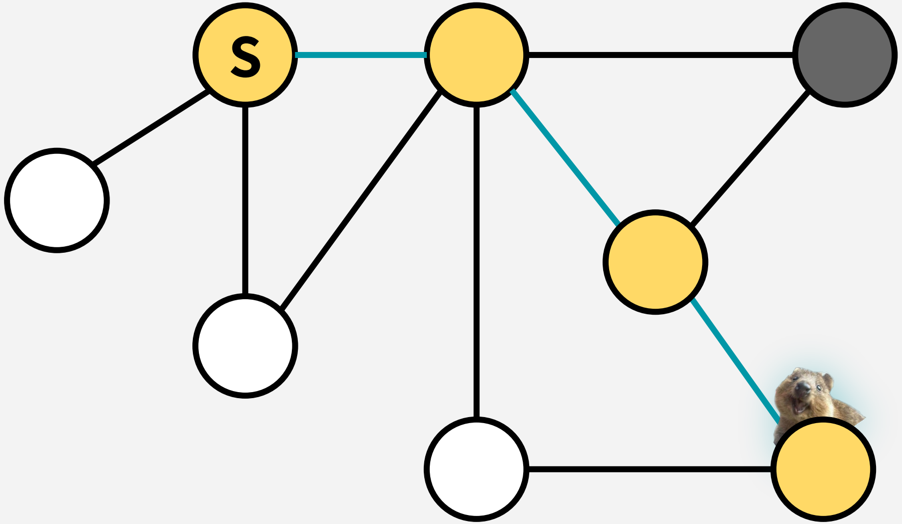
visited, but haven't explored all the paths out

visited, and fully explored

DEPTH-FIRST SEARCH

An analogy:

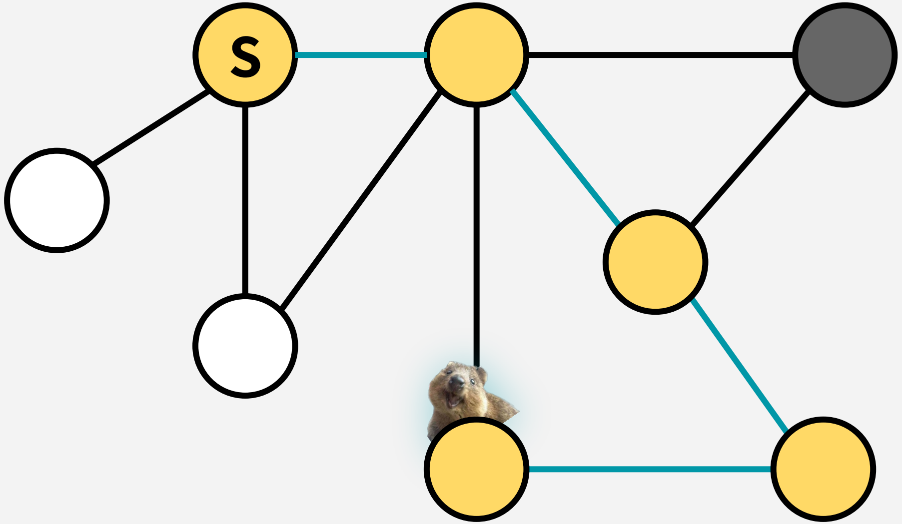
A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)

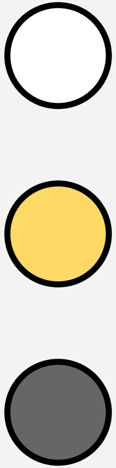


DEPTH-FIRST SEARCH

An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)

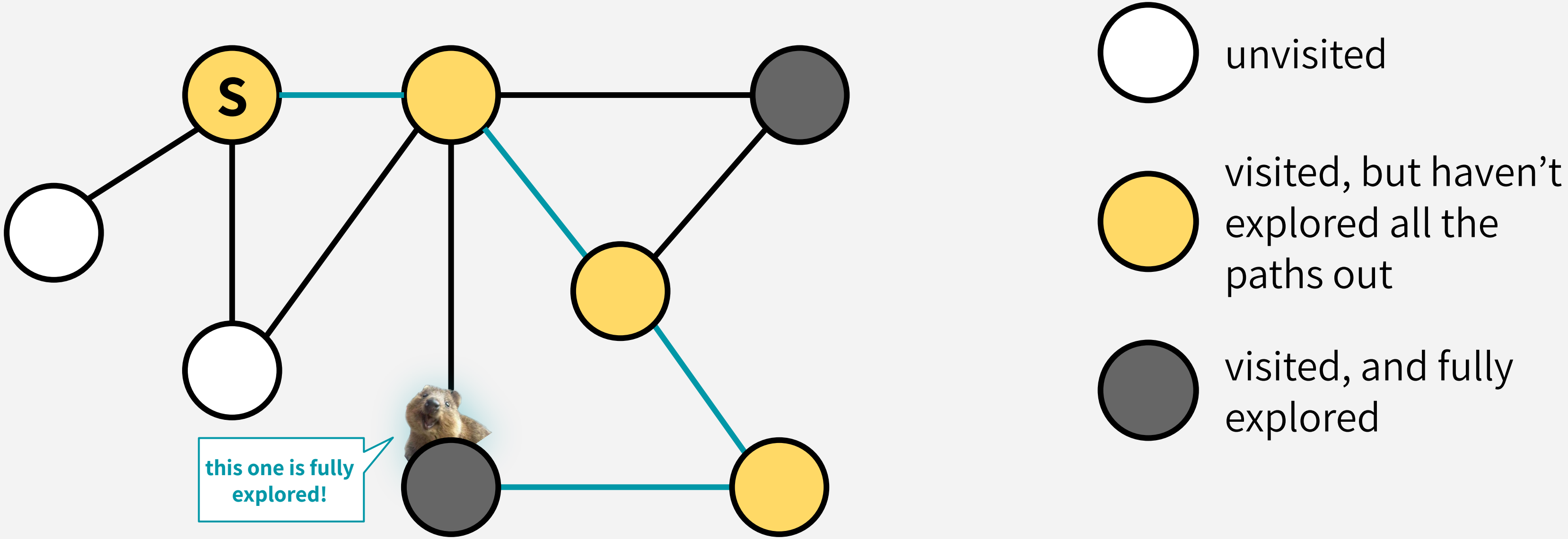




DEPTH-FIRST SEARCH

An analogy:

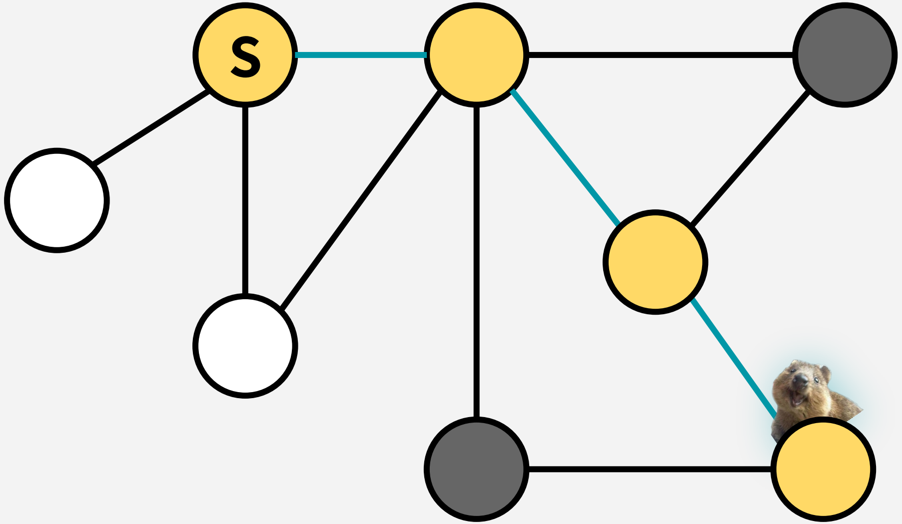
A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



DEPTH-FIRST SEARCH

An analogy:

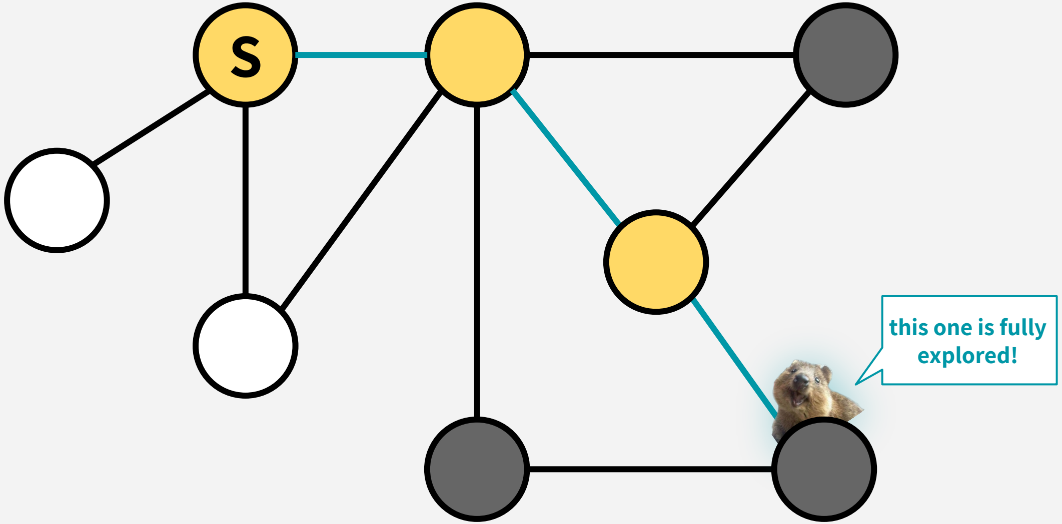
A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



DEPTH-FIRST SEARCH

An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)

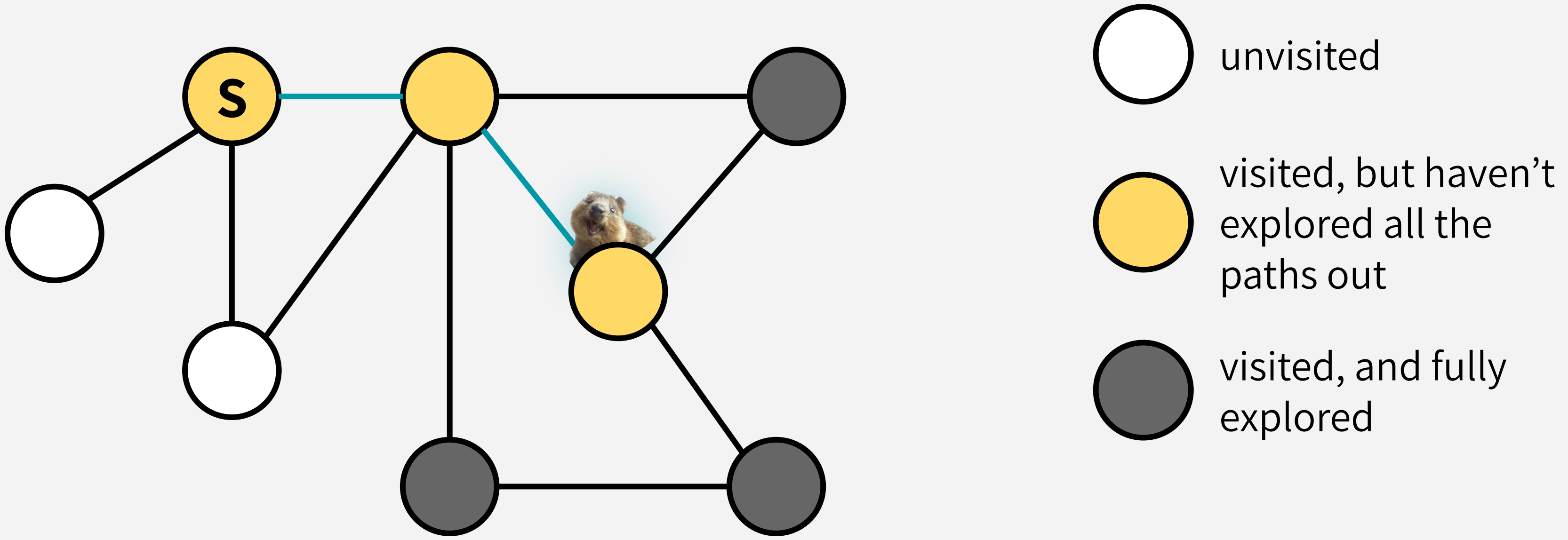


- unvisited
- visited, but haven't explored all the paths out
- visited, and fully explored

DEPTH-FIRST SEARCH

An analogy:

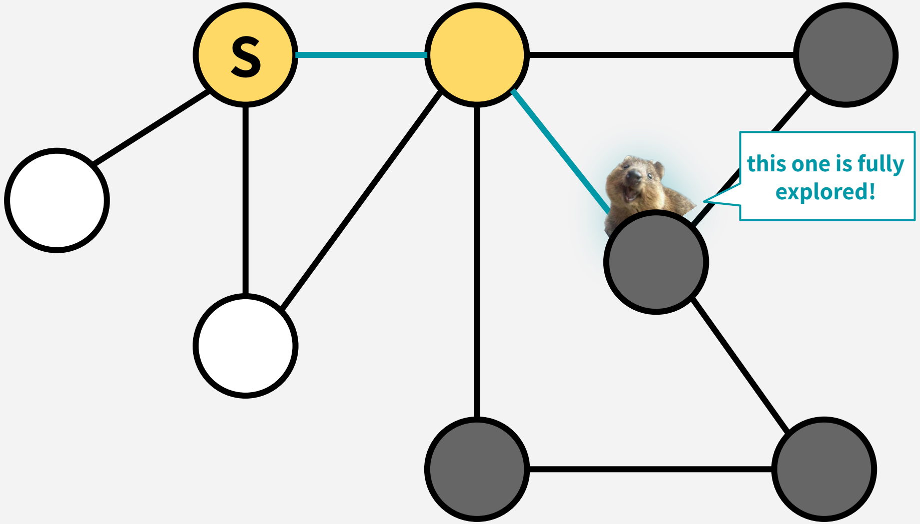
A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



DEPTH-FIRST SEARCH

An analogy:

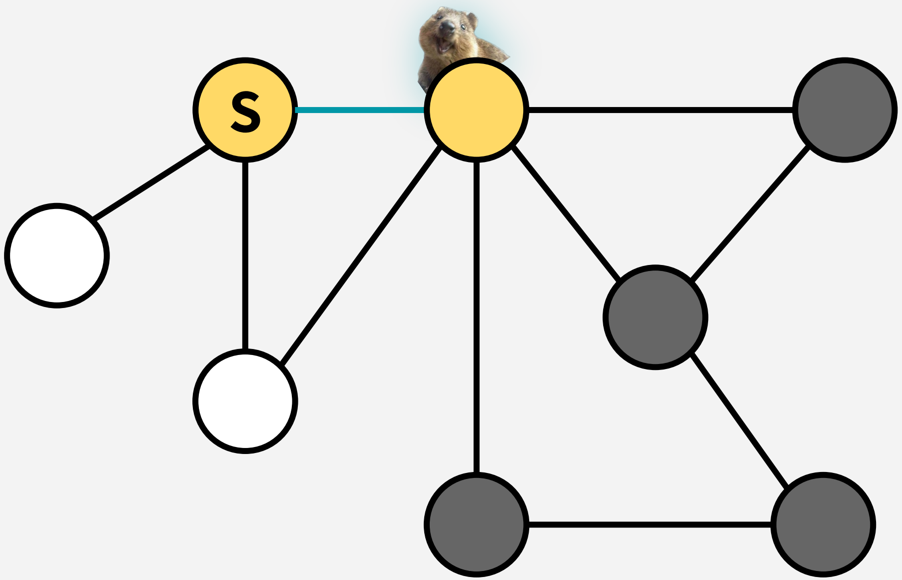
A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



DEPTH-FIRST SEARCH

An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



unvisited

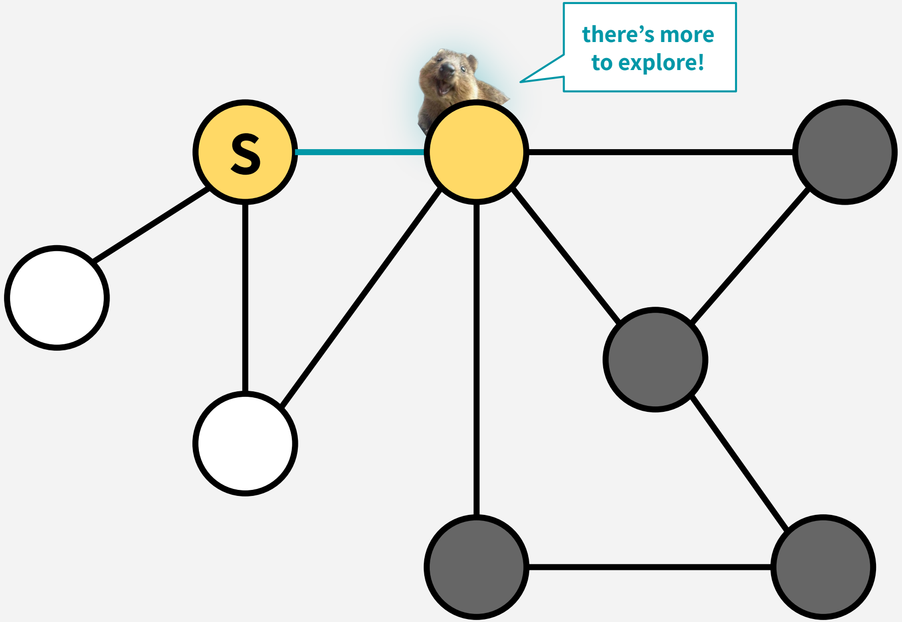
visited, but haven't explored all the paths out

visited, and fully explored

DEPTH-FIRST SEARCH

An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



unvisited

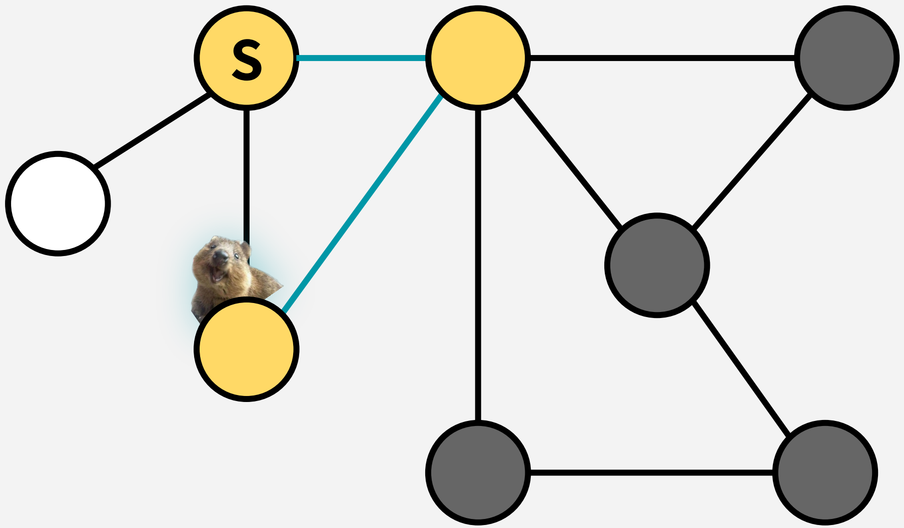
visited, but haven't explored all the paths out

visited, and fully explored

DEPTH-FIRST SEARCH

An analogy:

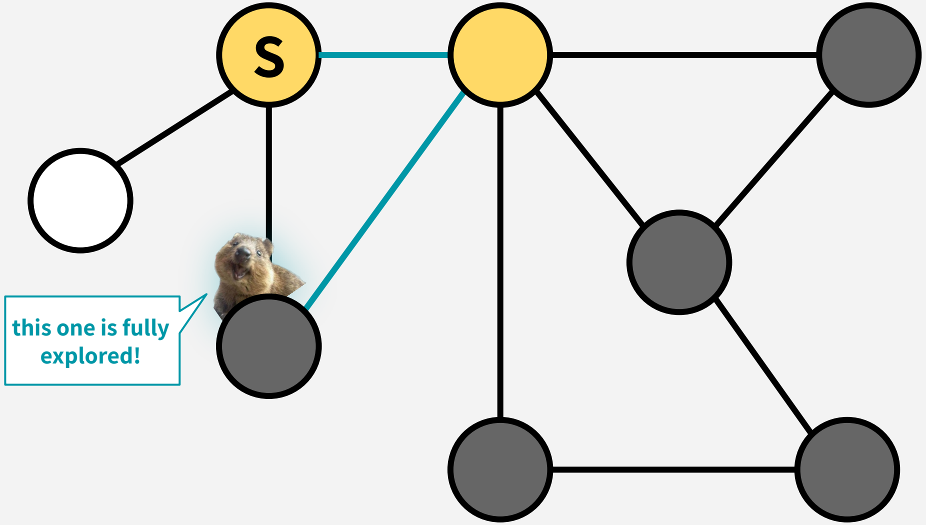
A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



DEPTH-FIRST SEARCH

An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



unvisited

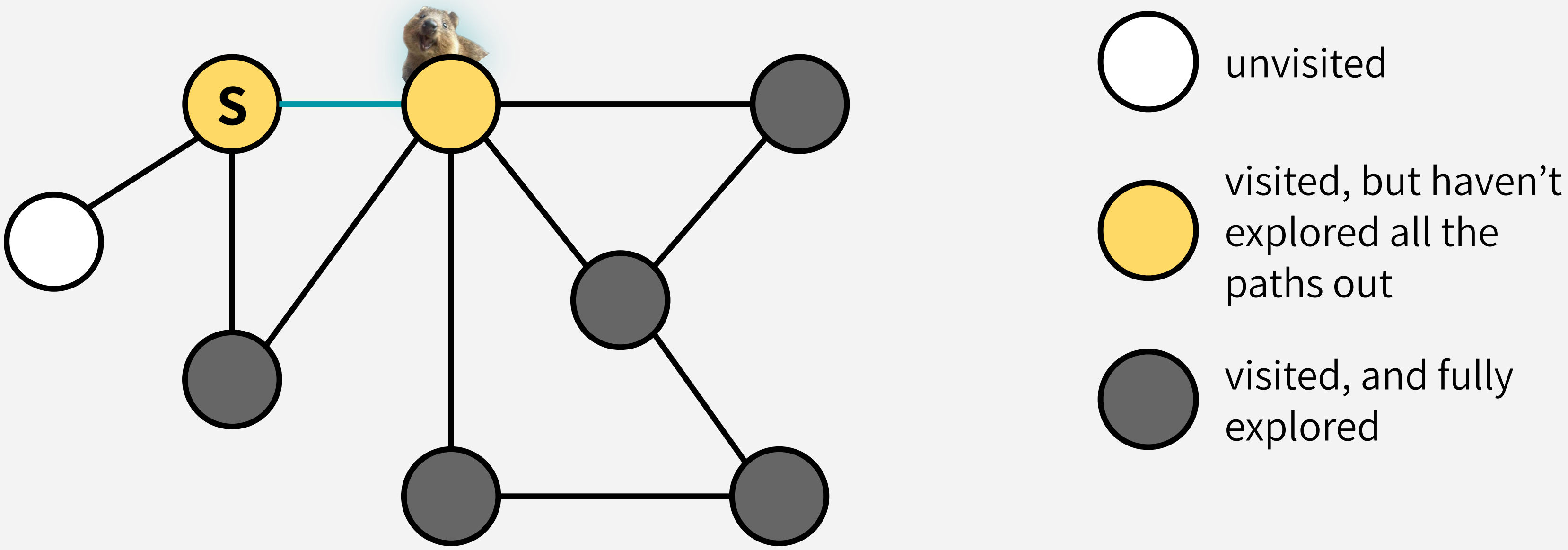
visited, but haven't explored all the paths out

visited, and fully explored

DEPTH-FIRST SEARCH

An analogy:

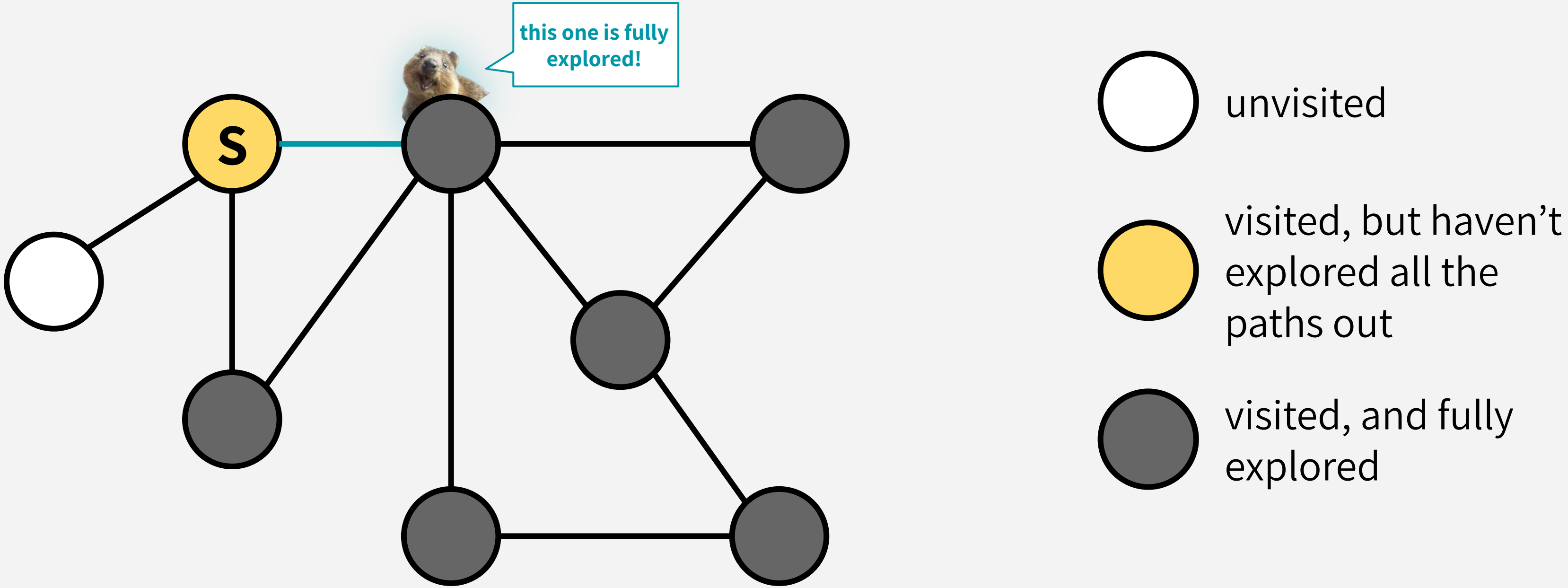
A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



DEPTH-FIRST SEARCH

An analogy:

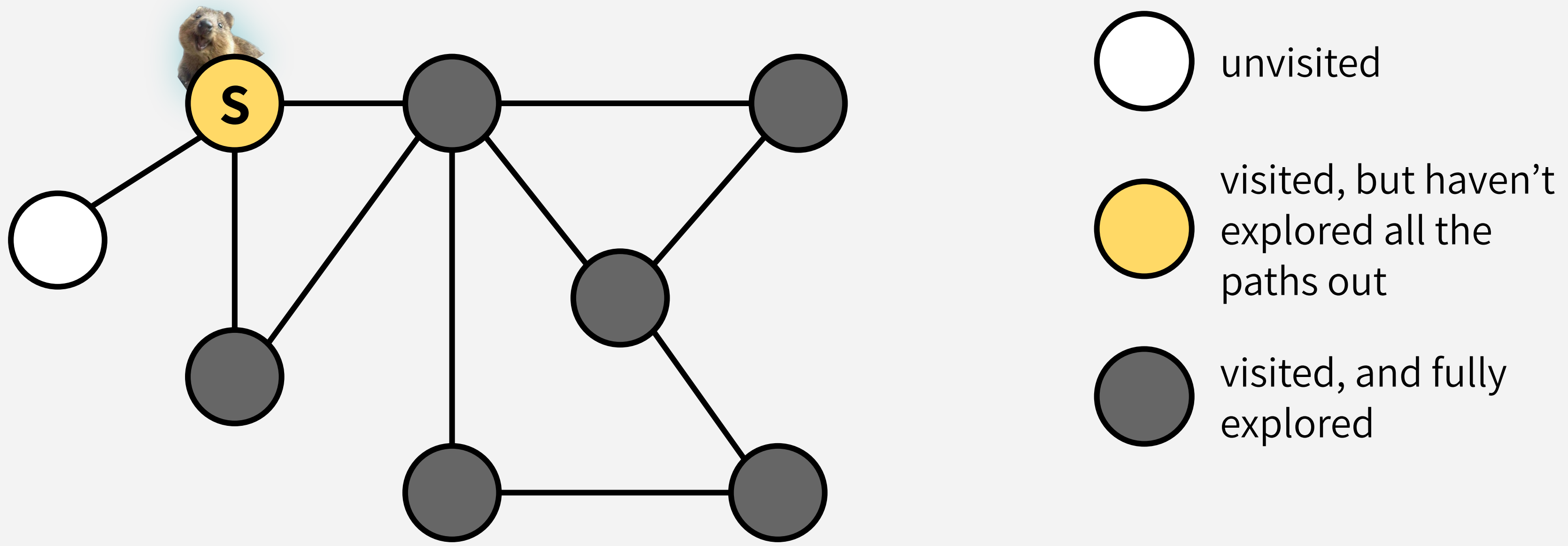
A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



DEPTH-FIRST SEARCH

An analogy:

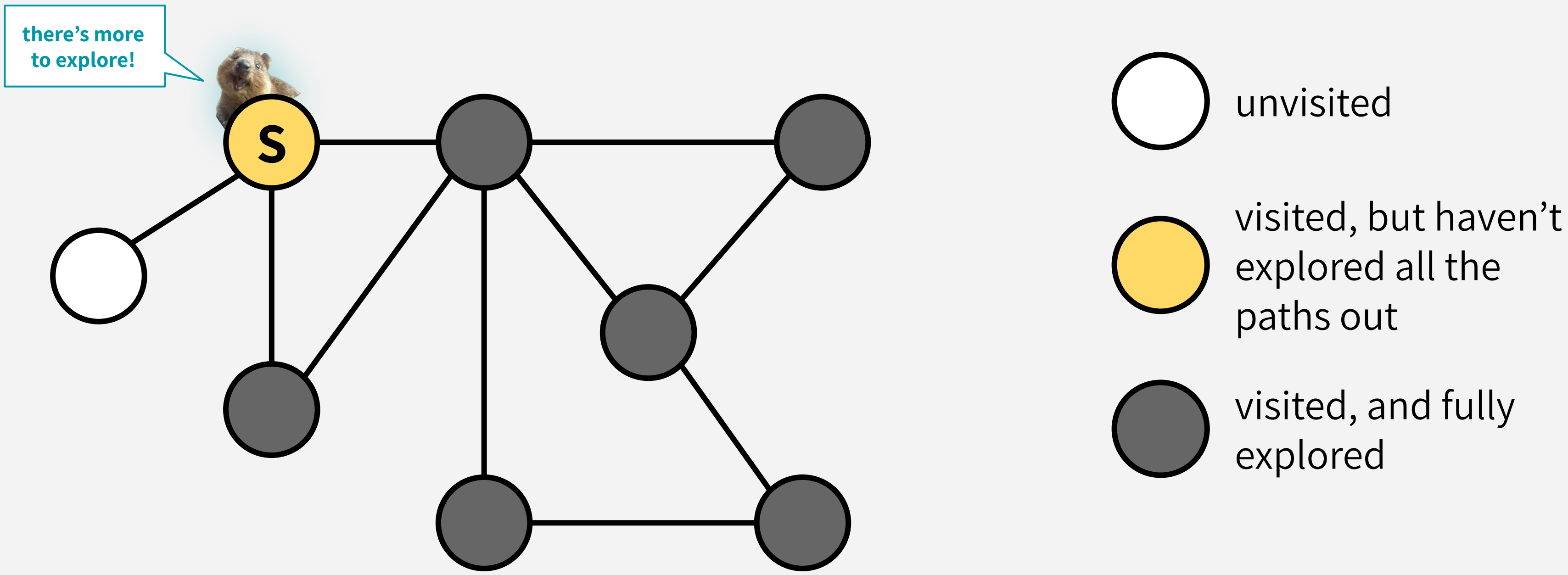
A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



DEPTH-FIRST SEARCH

An analogy:

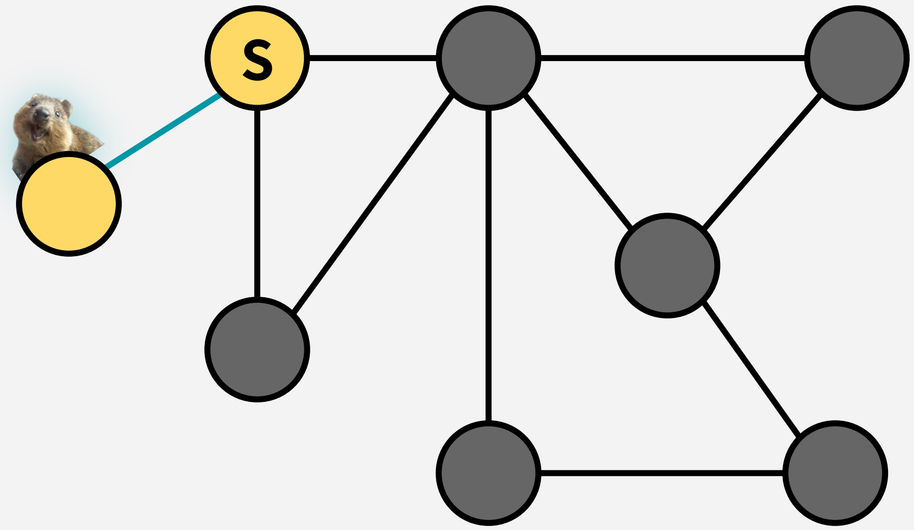
A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



DEPTH-FIRST SEARCH

An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



unvisited

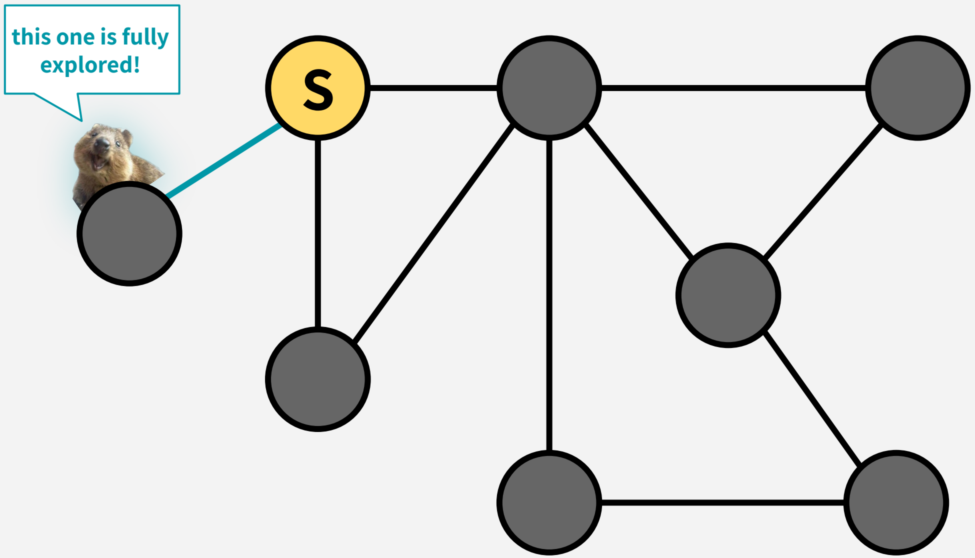
visited, but haven't explored all the paths out

visited, and fully explored

DEPTH-FIRST SEARCH

An analogy:

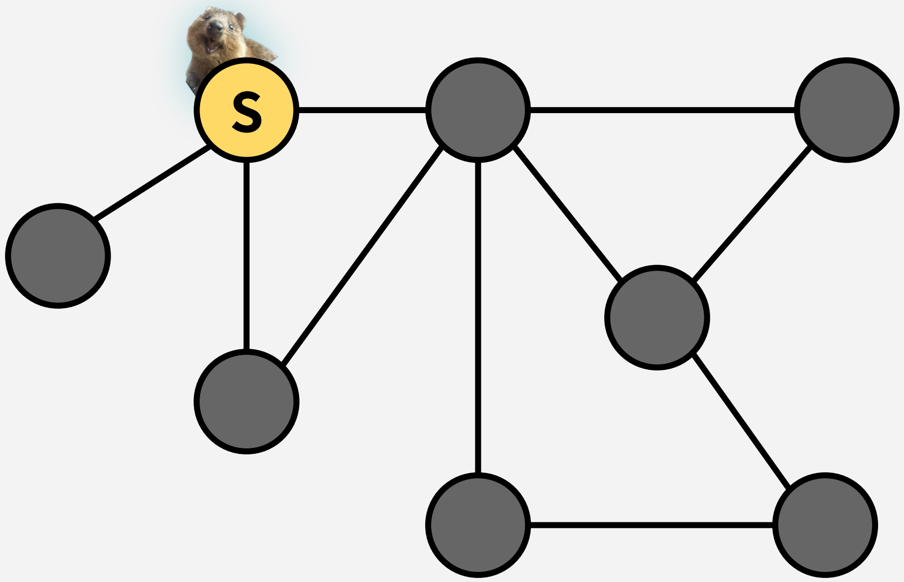
A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



DEPTH-FIRST SEARCH

An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



unvisited

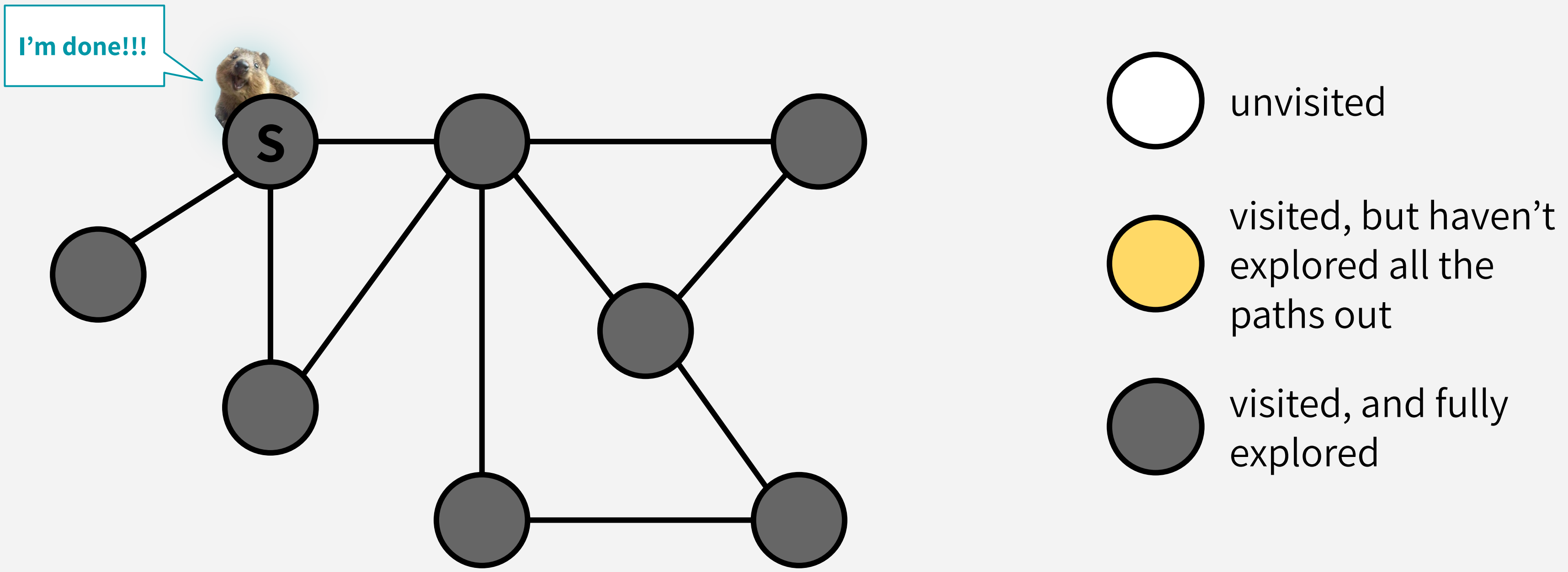
visited, but haven't explored all the paths out

visited, and fully explored

DEPTH-FIRST SEARCH

An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



DEPTH-FIRST SEARCH

An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)

I'm done!!!

In addition to keeping track of the visited status of nodes, we're going to keep track of:

The time we first enter it, i.e. mark it as .

The time we finish it, i.e. mark it as .

You've probably seen other ways to implement DFS, all this extra bookkeeping will be useful for us later!

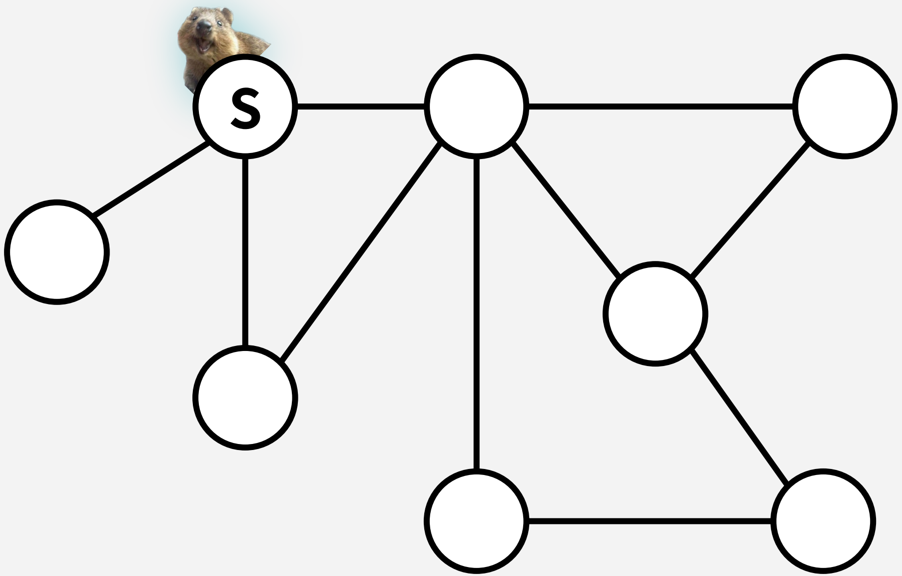
ut haven't
all the

nd fully

DEPTH-FIRST SEARCH

An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)

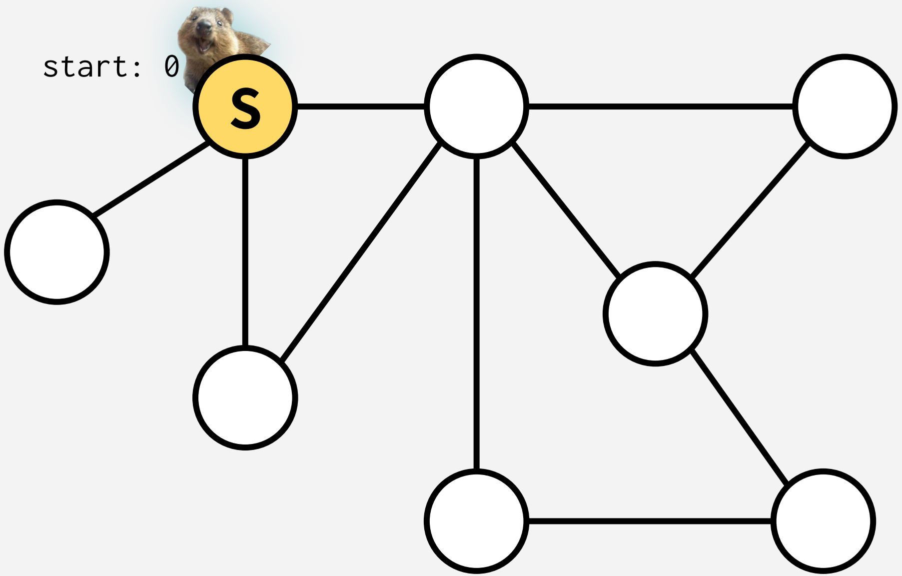



```
DFS(w, currTime):
    w.start = currTime
    currTime++
    mark w as visited
    for v in w.neighbors:
        if v is unvisited:
            currTime =
               DFS(v, currTime)
            currTime++
    w.finish = currTime
    mark w as finished
    return currTime
```

DEPTH-FIRST SEARCH

An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)

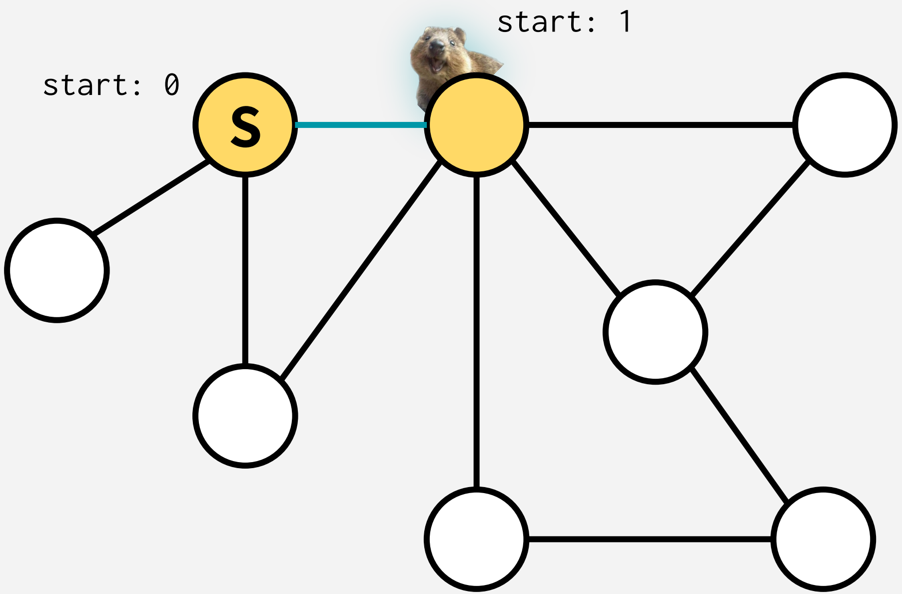



```
DFS(w, currTime):
    w.start = currTime
    currTime++
    mark w as visited
    for v in w.neighbors:
        if v is unvisited:
            currTime =
                DFS(v, currTime)
            currTime++
    w.finish = currTime
    mark w as finished
    return currTime
```

DEPTH-FIRST SEARCH

An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)

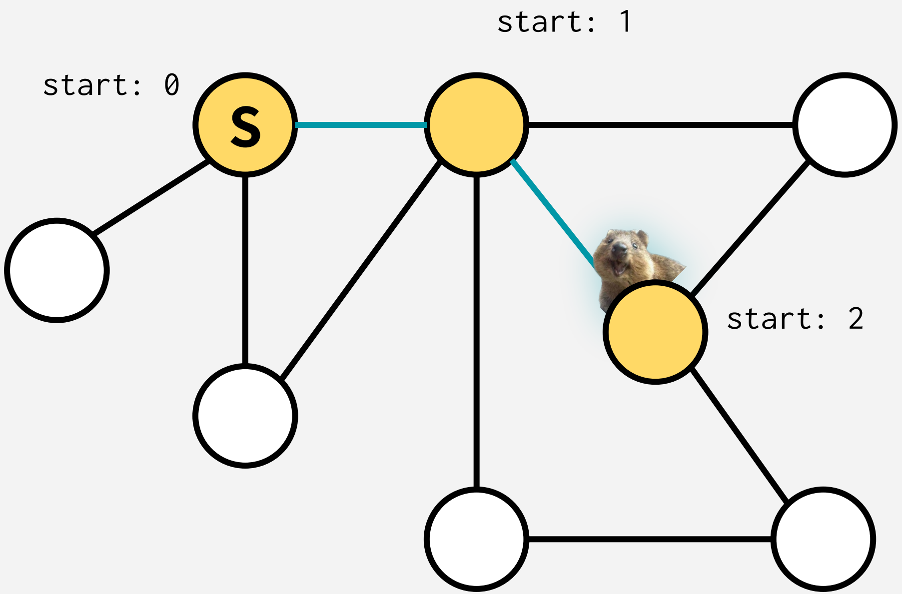



```
DFS(w, currTime):
    w.start = currTime
    currTime++
    mark w as visited
    for v in w.neighbors:
        if v is unvisited:
            currTime =
                DFS(v, currTime)
            currTime++
    w.finish = currTime
    mark w as finished
    return currTime
```

DEPTH-FIRST SEARCH

An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)

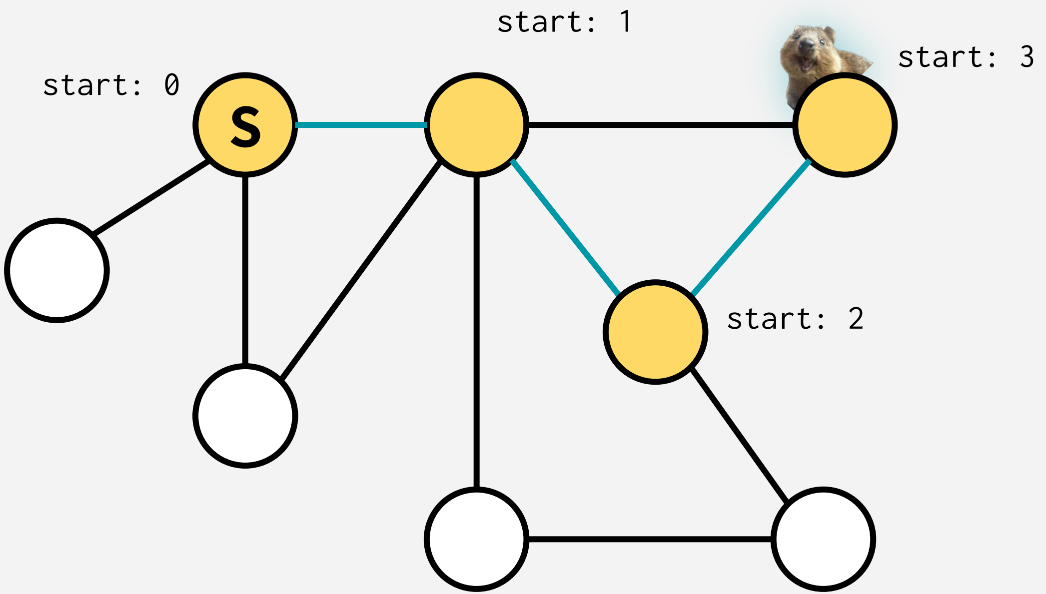



```
DFS(w, currTime):
    w.start = currTime
    currTime++
    mark w as visited
    for v in w.neighbors:
        if v is unvisited:
            currTime =
                DFS(v, currTime)
            currTime++
    w.finish = currTime
    mark w as finished
    return currTime
```

DEPTH-FIRST SEARCH

An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)

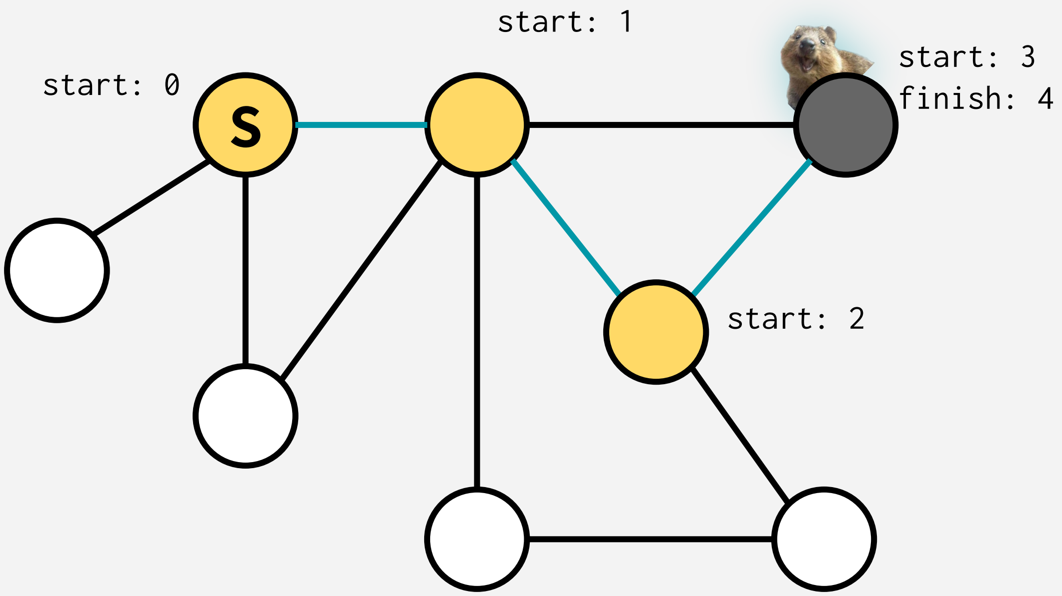



```
DFS(w, currTime):
    w.start = currTime
    currTime++
    mark w as visited
    for v in w.neighbors:
        if v is unvisited:
            currTime =
                DFS(v, currTime)
            currTime++
    w.finish = currTime
    mark w as finished
    return currTime
```

DEPTH-FIRST SEARCH

An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)

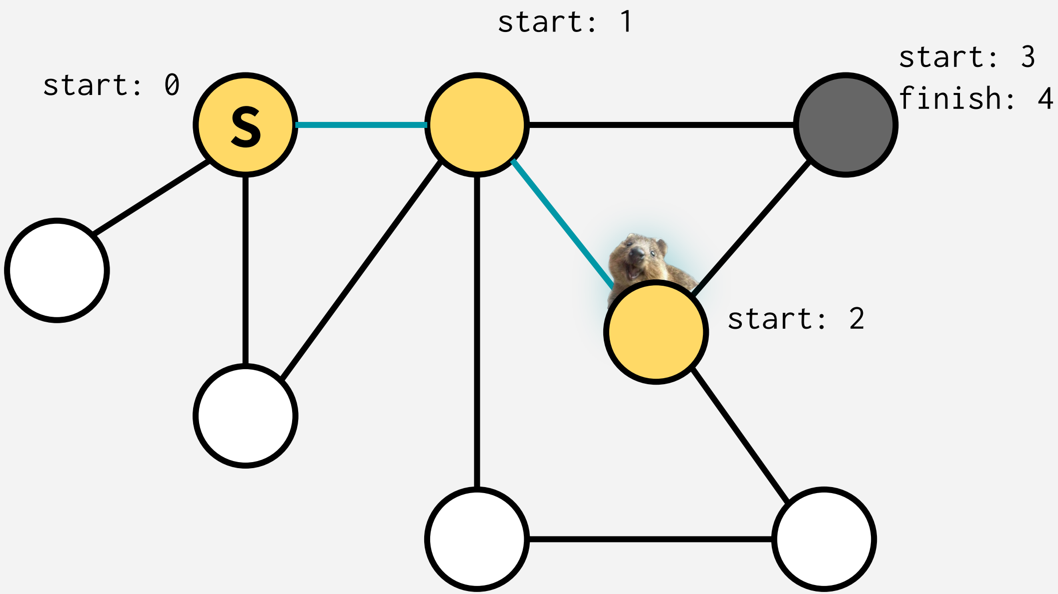



```
DFS(w, currTime):
    w.start = currTime
    currTime++
    mark w as visited
    for v in w.neighbors:
        if v is unvisited:
            currTime =
                DFS(v, currTime)
            currTime++
    w.finish = currTime
    mark w as finished
    return currTime
```

DEPTH-FIRST SEARCH

An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)

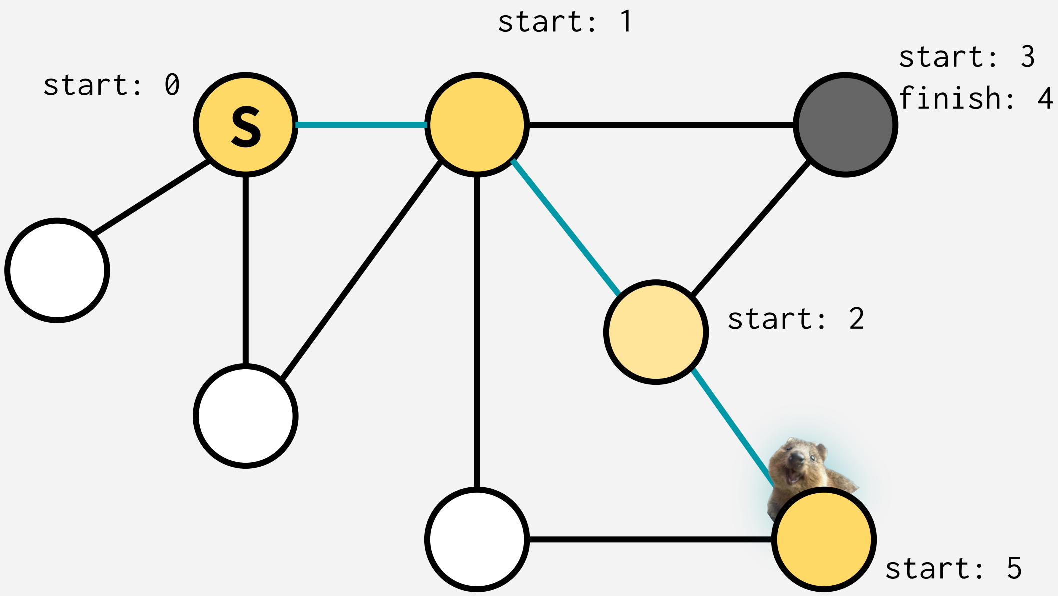



```
DFS(w, currTime):
    w.start = currTime
    currTime++
    mark w as visited
    for v in w.neighbors:
        if v is unvisited:
            currTime =
                DFS(v, currTime)
            currTime++
    w.finish = currTime
    mark w as finished
    return currTime
```

DEPTH-FIRST SEARCH

An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)

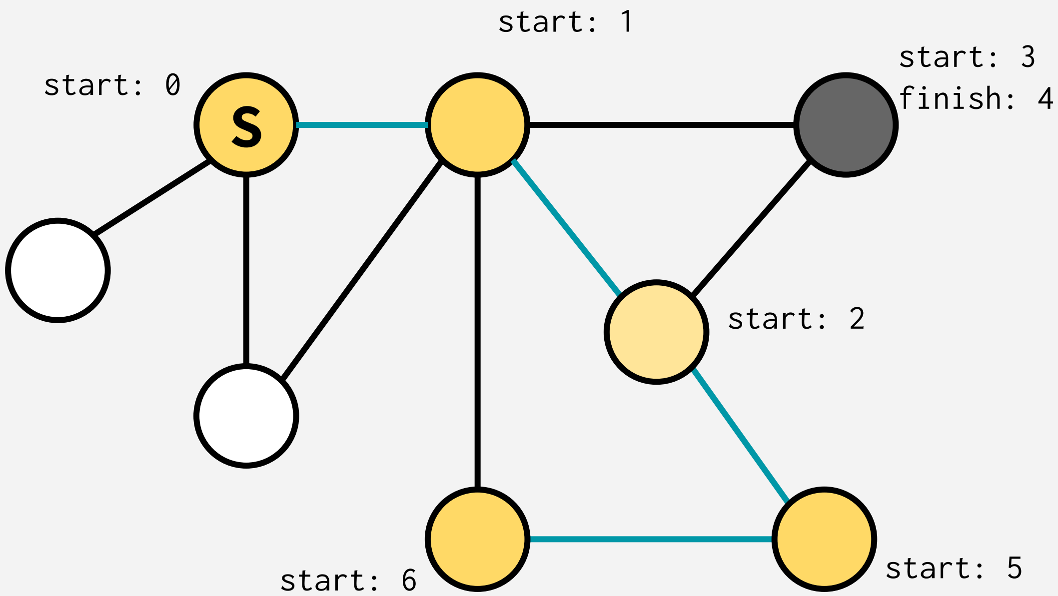



```
DFS(w, currTime):
    w.start = currTime
    currTime++
    mark w as visited
    for v in w.neighbors:
        if v is unvisited:
            currTime =
                DFS(v, currTime)
            currTime++
    w.finish = currTime
    mark w as finished
    return currTime
```

DEPTH-FIRST SEARCH

An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)

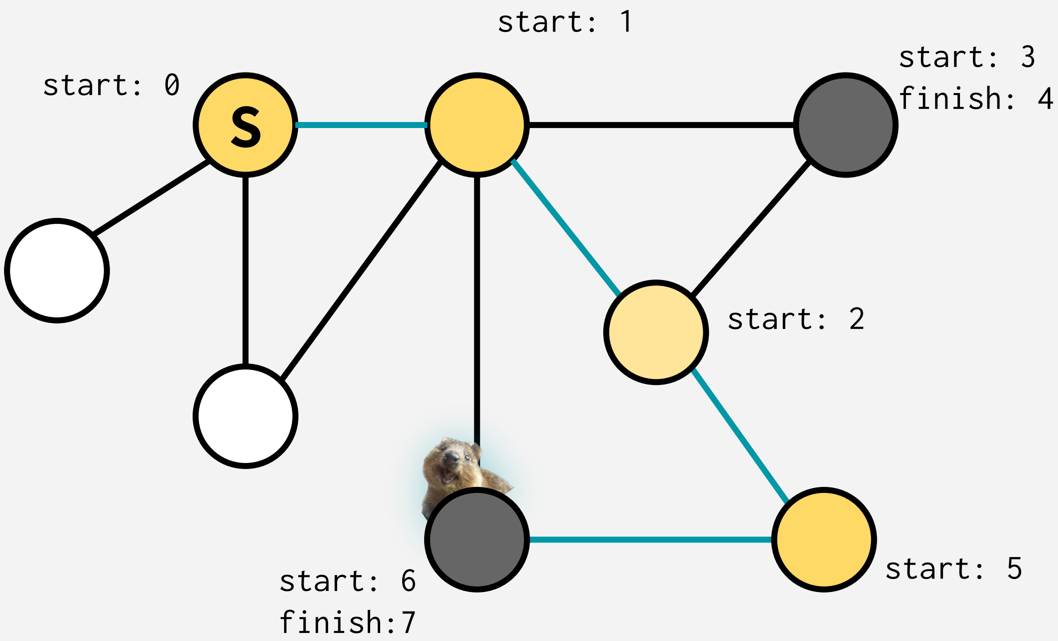



```
DFS(w, currTime):
    w.start = currTime
    currTime++
    mark w as visited
    for v in w.neighbors:
        if v is unvisited:
            currTime =
                DFS(v, currTime)
            currTime++
    w.finish = currTime
    mark w as finished
    return currTime
```

DEPTH-FIRST SEARCH

An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)

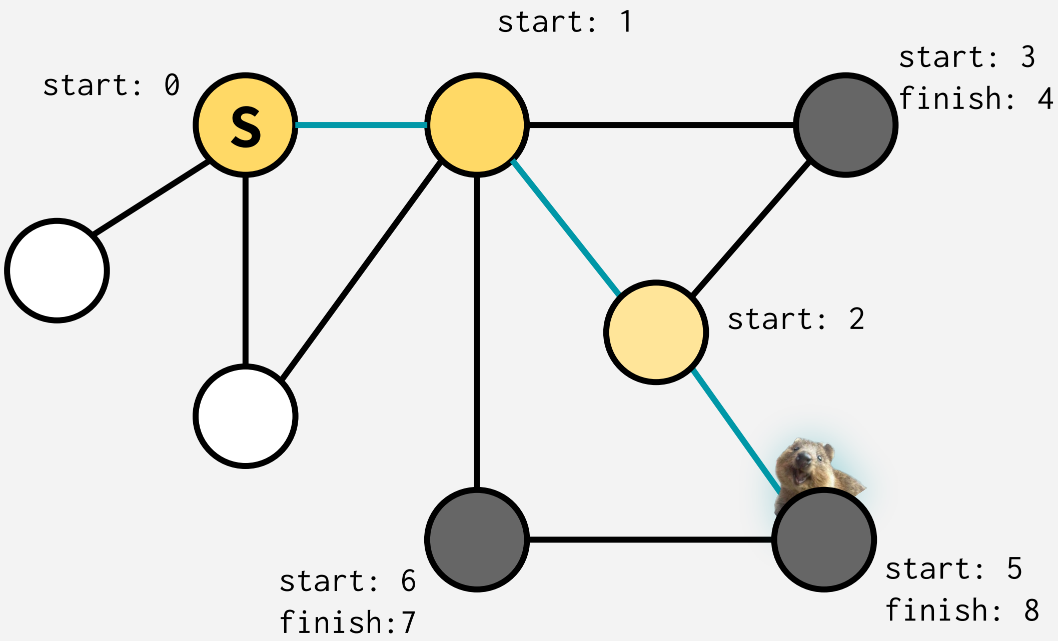



```
DFS(w, currTime):
    w.start = currTime
    currTime++
    mark w as visited
    for v in w.neighbors:
        if v is unvisited:
            currTime =
                DFS(v, currTime)
            currTime++
    w.finish = currTime
    mark w as finished
    return currTime
```

DEPTH-FIRST SEARCH

An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)

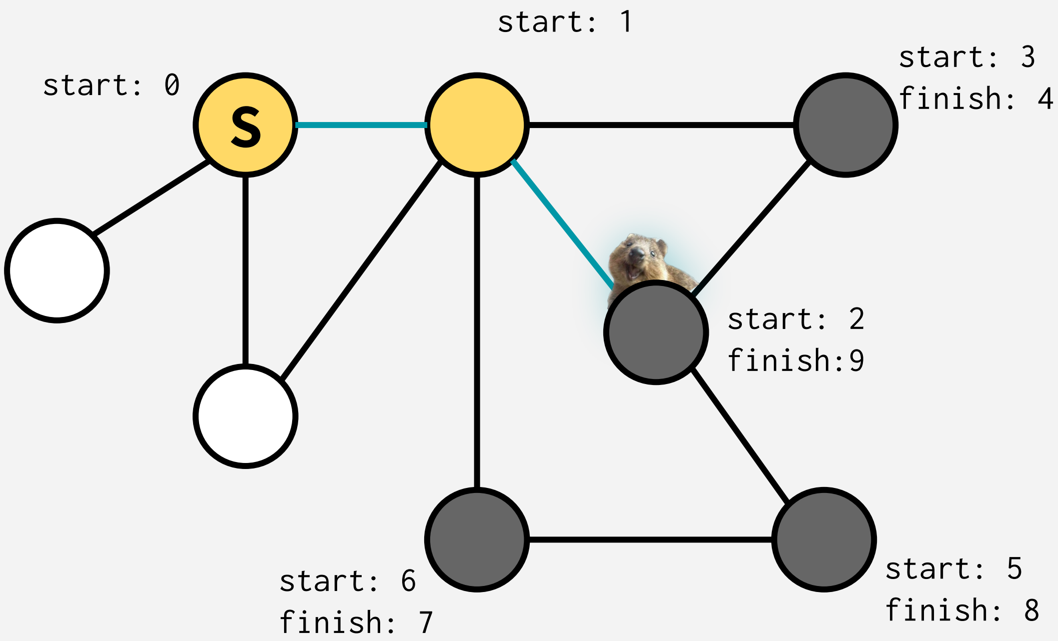



```
DFS(w, currTime):
    w.start = currTime
    currTime++
    mark w as visited
    for v in w.neighbors:
        if v is unvisited:
            currTime =
                DFS(v, currTime)
            currTime++
    w.finish = currTime
    mark w as finished
    return currTime
```

DEPTH-FIRST SEARCH

An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)

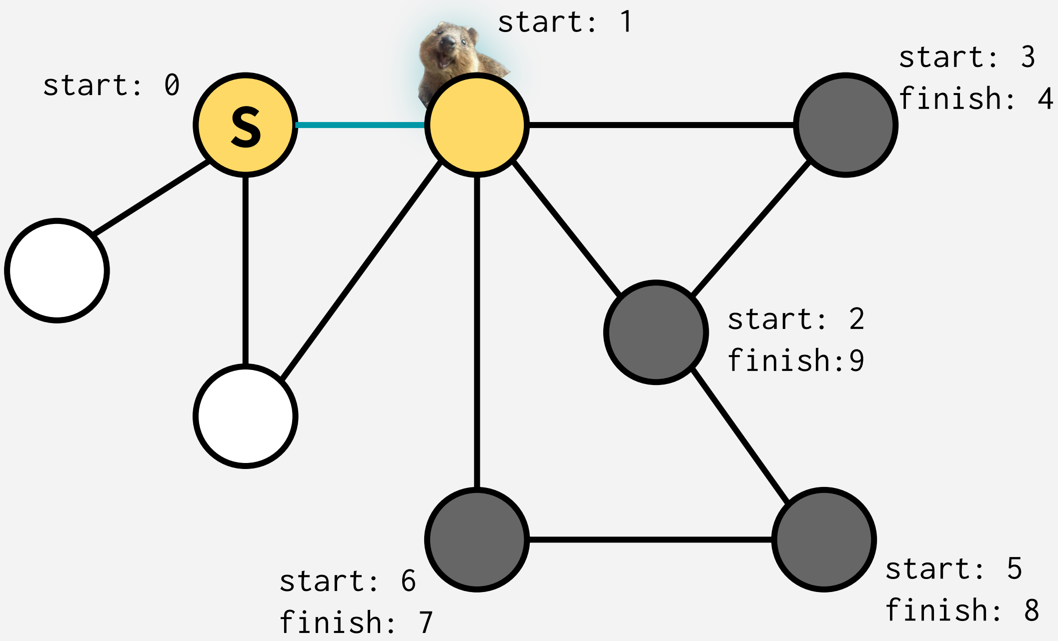



```
DFS(w, currTime):
    w.start = currTime
    currTime++
    mark w as visited
    for v in w.neighbors:
        if v is unvisited:
            currTime =
               DFS(v, currTime)
            currTime++
    w.finish = currTime
    mark w as finished
    return currTime
```

DEPTH-FIRST SEARCH

An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)

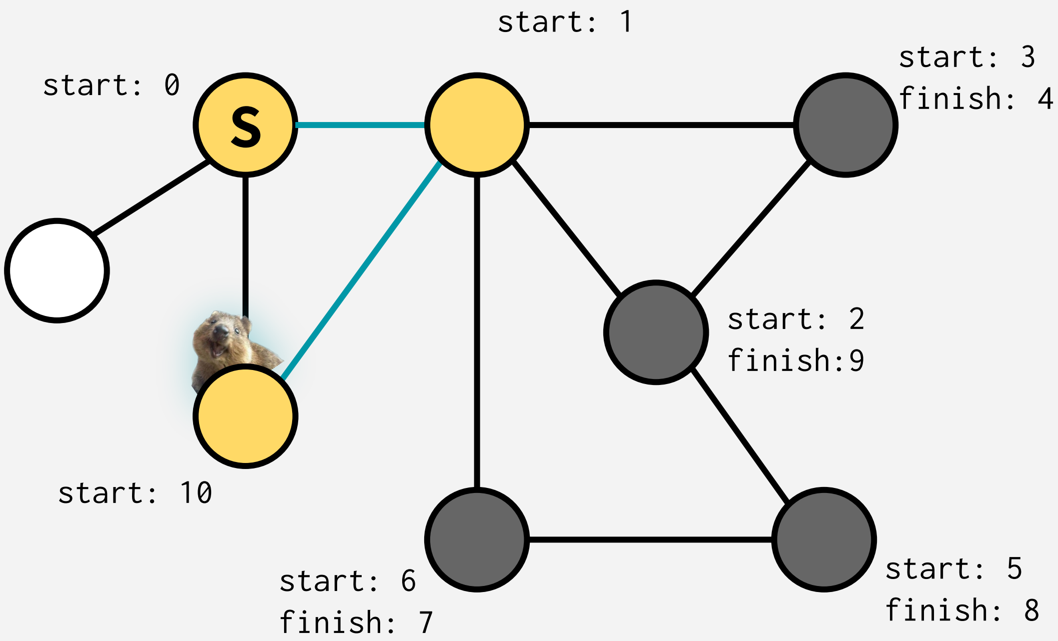



```
DFS(w, currTime):
    w.start = currTime
    currTime++
    mark w as visited
    for v in w.neighbors:
        if v is unvisited:
            currTime =
                DFS(v, currTime)
            currTime++
    w.finish = currTime
    mark w as finished
    return currTime
```

DEPTH-FIRST SEARCH

An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)

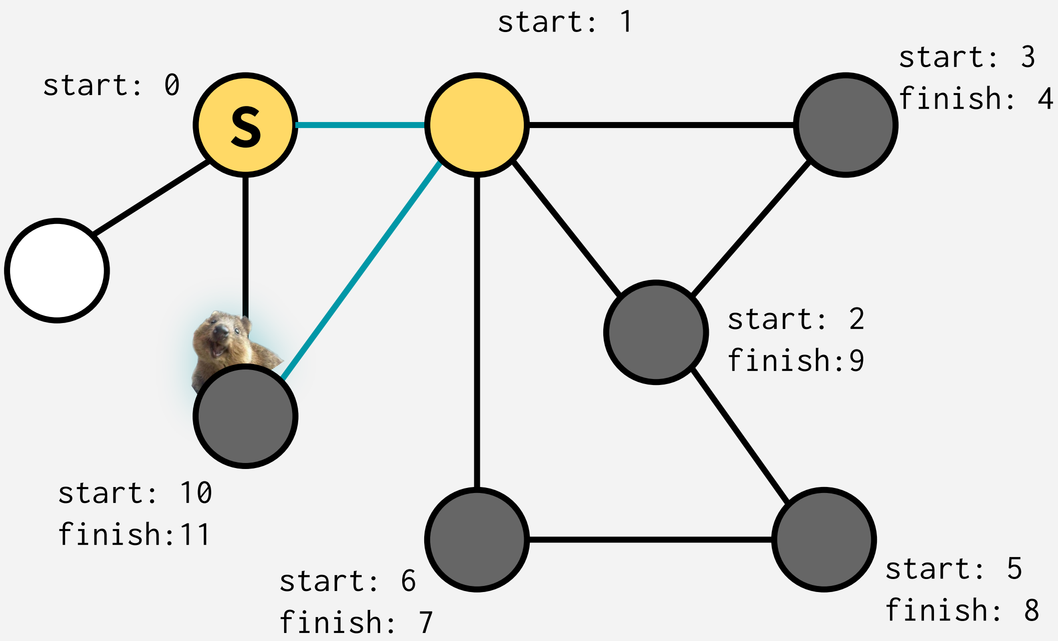



```
DFS(w, currTime):
    w.start = currTime
    currTime++
    mark w as visited
    for v in w.neighbors:
        if v is unvisited:
            currTime =
                DFS(v, currTime)
            currTime++
    w.finish = currTime
    mark w as finished
    return currTime
```

DEPTH-FIRST SEARCH

An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)

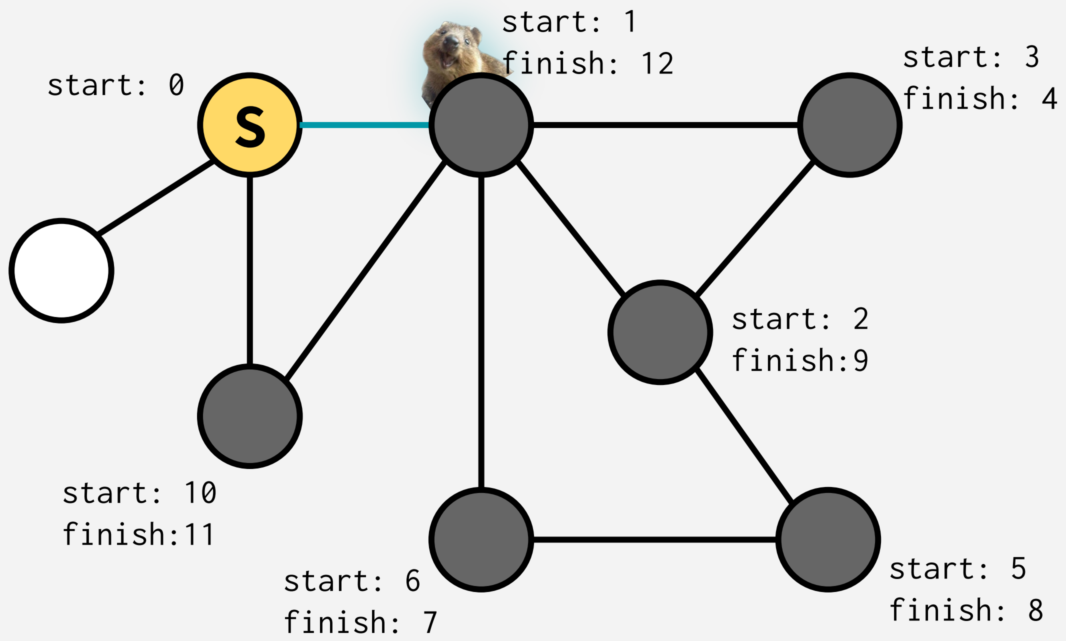



```
DFS(w, currTime):
    w.start = currTime
    currTime++
    mark w as visited
    for v in w.neighbors:
        if v is unvisited:
            currTime =
                DFS(v, currTime)
            currTime++
    w.finish = currTime
    mark w as finished
    return currTime
```

DEPTH-FIRST SEARCH

An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)

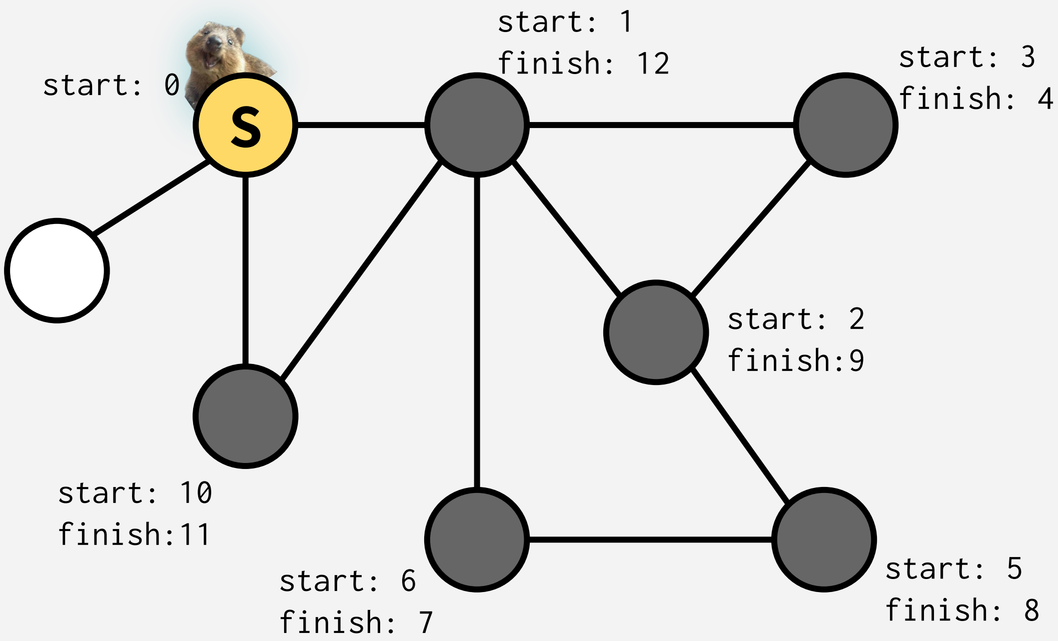



```
DFS(w, currTime):
    w.start = currTime
    currTime++
    mark w as visited
    for v in w.neighbors:
        if v is unvisited:
            currTime =
                DFS(v, currTime)
            currTime++
    w.finish = currTime
    mark w as finished
    return currTime
```

DEPTH-FIRST SEARCH

An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)

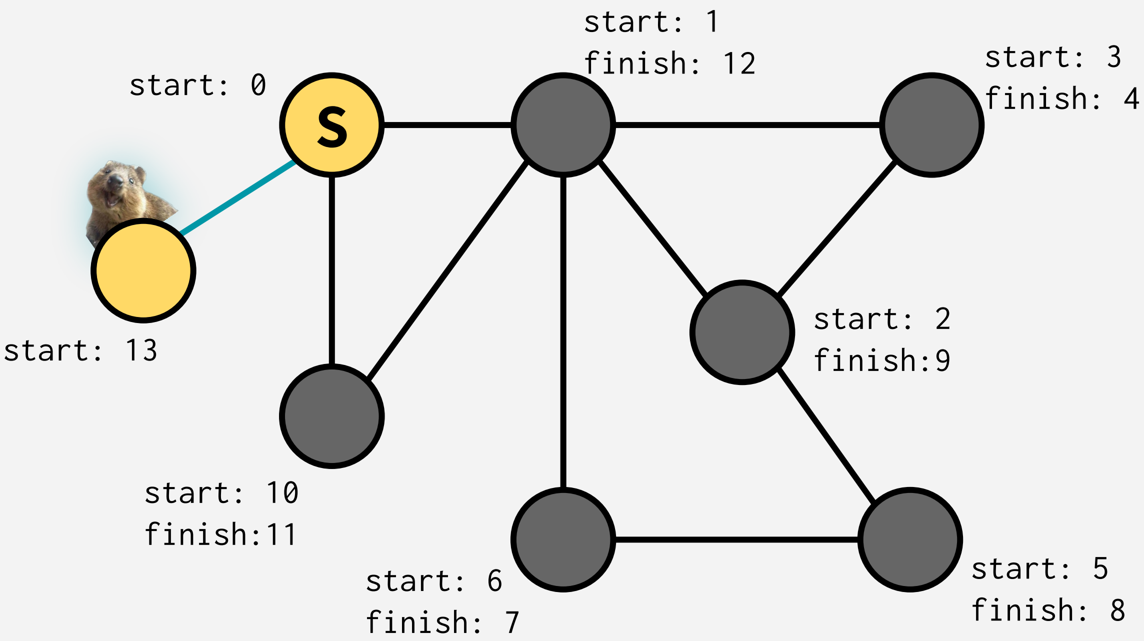



```
DFS(w, currTime):
    w.start = currTime
    currTime++
    mark w as visited
    for v in w.neighbors:
        if v is unvisited:
            currTime =
               DFS(v, currTime)
            currTime++
    w.finish = currTime
    mark w as finished
    return currTime
```

DEPTH-FIRST SEARCH

An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)

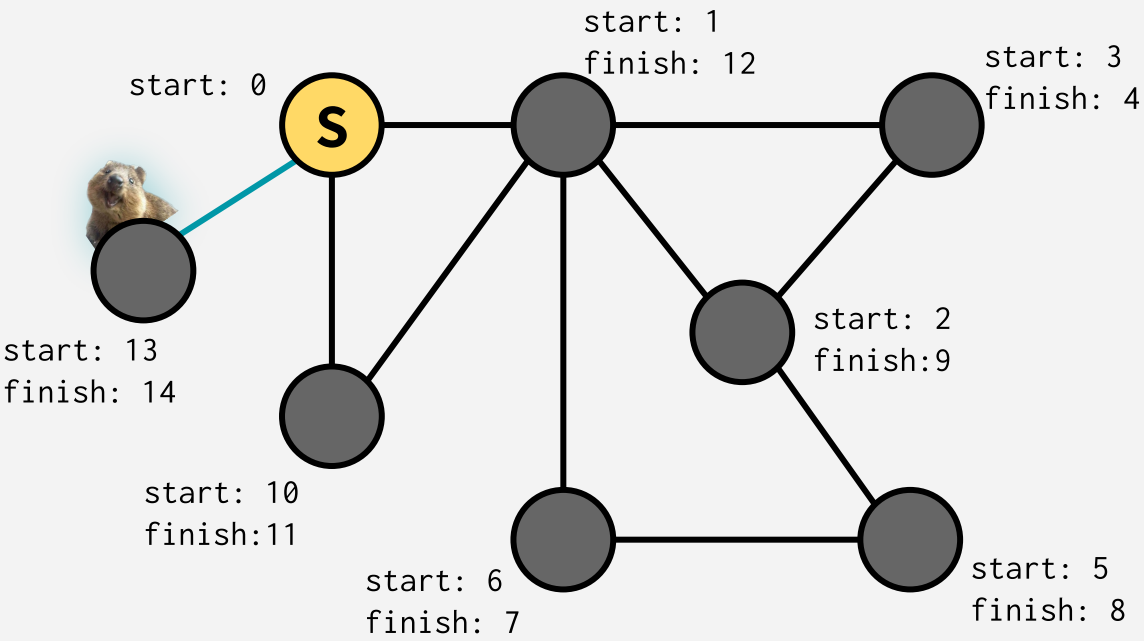



```
DFS(w, currTime):
    w.start = currTime
    currTime++
    mark w as visited
    for v in w.neighbors:
        if v is unvisited:
            currTime =
                DFS(v, currTime)
            currTime++
    w.finish = currTime
    mark w as finished
    return currTime
```

DEPTH-FIRST SEARCH

An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)

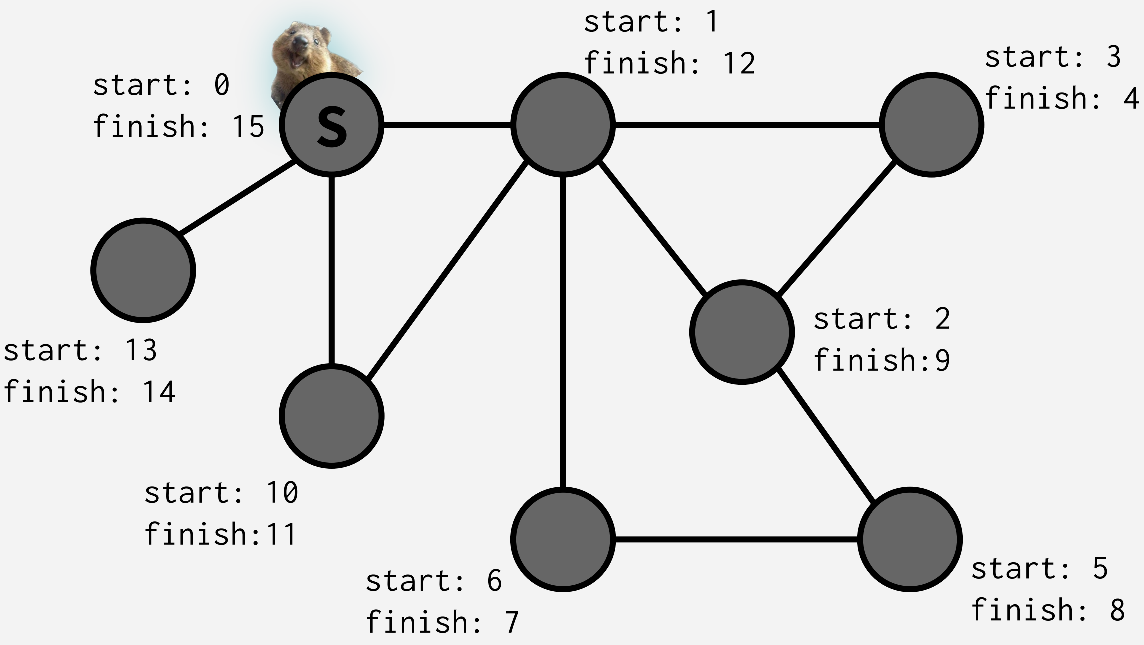



```
DFS(w, currTime):
    w.start = currTime
    currTime++
    mark w as visited
    for v in w.neighbors:
        if v is unvisited:
            currTime =
                DFS(v, currTime)
            currTime++
    w.finish = currTime
    mark w as finished
    return currTime
```

DEPTH-FIRST SEARCH

An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)




```
DFS(w, currTime):
    w.start = currTime
    currTime++
    mark w as visited
    for v in w.neighbors:
        if v is unvisited:
            currTime =
                DFS(v, currTime)
            currTime++
    w.finish = currTime
    mark w as finished
    return currTime
```

DEPTH-FIRST SEARCH

An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)

start: 1

start: 0
finish:

start: 13
finish: 14

start
finish:

start: 6
finish: 7

start: 5
finish: 8

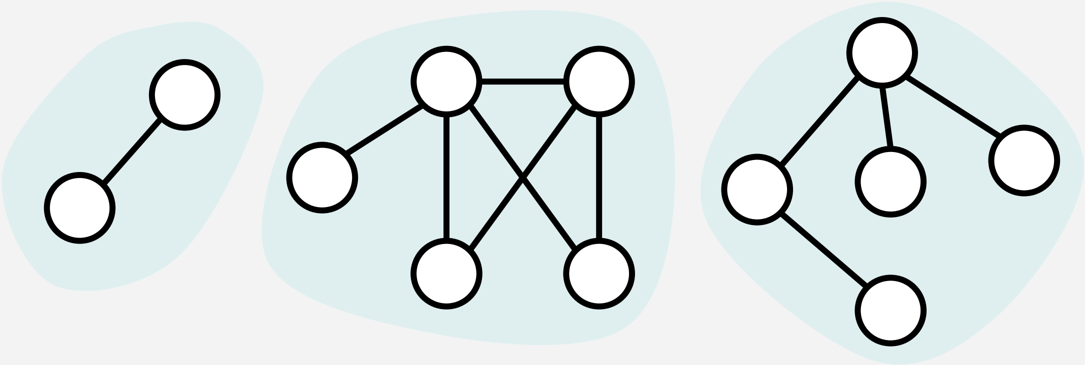
This is not the only way to write DFS!
See the textbook for an iterative version, and try writing it yourself (great for interview practice)

```
ime
ed
bors:
sited:
currTime)
Time
shed
return currTime
```

DEPTH-FIRST SEARCH

Like BFS, DFS finds all the nodes reachable from the starting point!

In undirected graphs, this is equivalent to finding a **connected component.**



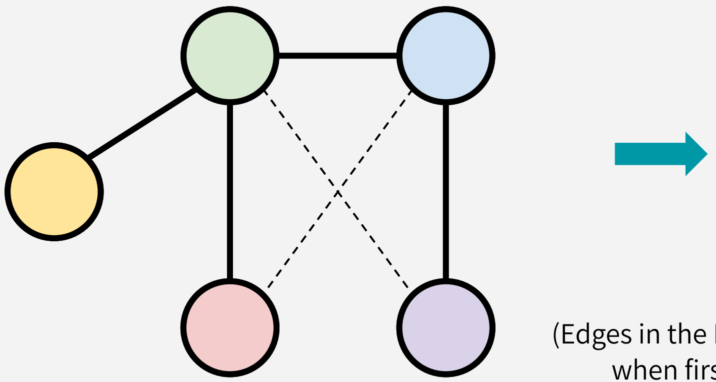
DEPTH-FIRST SEARCH

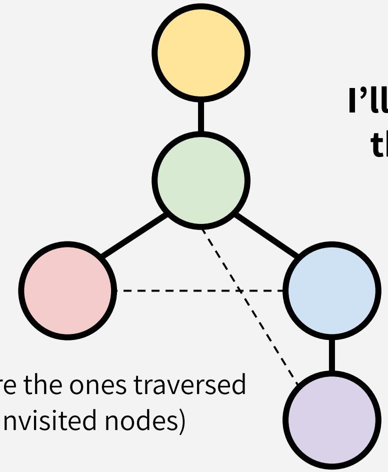
Why is it called depth-first?

We are implicitly building a **tree**!

(It's a tree because we never revisit a node)

We're going as "deep" as we can before "bubbling" back up.





I'll refer to this as the "DFS tree"

(Edges in the DFS tree are the ones traversed when first finding unvisited nodes)

DEPTH-FIRST SEARCH: RUNTIME

To explore a graph's **i^{th} connected component** (n_i nodes, m_i edges):

We visit each vertex in the CC exactly once ("visit" = "call DFS on").
At each vertex v, we:

- Do some bookkeeping: **O(1)**
- Loop over v's neighbors & check if they are visited (& then potentially make a recursive call): O(1) per neighbor → **O(deg(v))** total.

Total: $\sum_v O(\deg(v)) + \sum_v O(1) = \mathbf{O(m_i + n_i)}$

DEPTH-FIRST SEARCH: RUNTIME

To explore **the entire graph** (n nodes, m edges):

A graph might have multiple connected components! To **explore the whole graph**, we would call our DFS routine once for each connected component (note that each vertex and each edge participates in exactly one connected component). The combined running time would be:

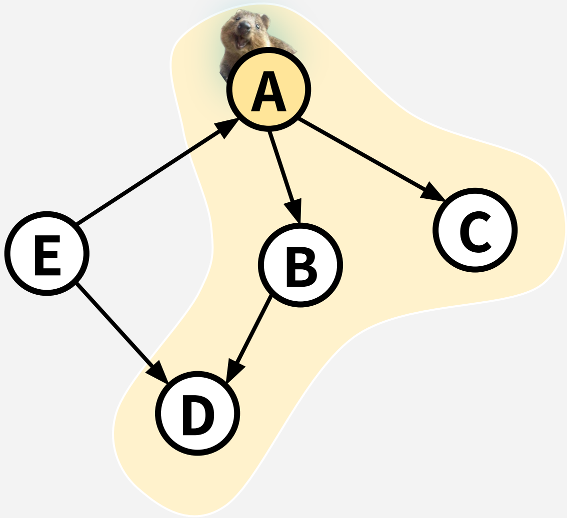
$$O(\textstyle\sum_i m_i + \sum_i n_i) = \mathbf{O(m + n)}$$

DEPTH-FIRST SEARCH

DFS works fine on directed graphs too!

From a start node x, DFS would find all nodes ***reachable*** from x.

(In directed graphs, "connected component" isn't as well defined… more on that later!)



Verify this on your own:
running DFS from A would still find all nodes reachable from A (E isn't reachable from A in this directed graph).

TRAVERSAL OF BSTs

We're actually already thinking about one application of DFS:

Given a BST, output the vertices in sorted order.

(Don't try writing your PSet solution in terms of DFS, but you can think about how DFS is closely related to this task)