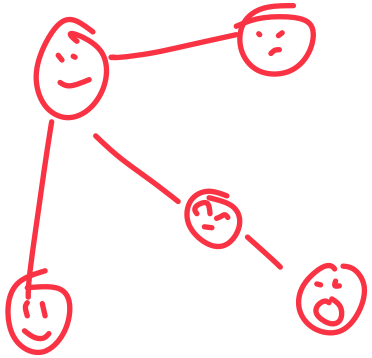
Algorithms and Data Structures for Data Science

Networkx & Graph Traversals

CS 277
Brad Solomon

April 3, 2024



UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

Learning Objectives

Practice using NetworkX to build and explore graphs

Implement breadth and depth traversals on graphs

Extend NetworkX for weighted and directed graphs

NetworkX Graph ADT

Find

- getVertices() —> list(G.nodes())
- getEdges(v) —> G[v]
- areAdjacent(u, v) —> G.has_edge(u, v)

Insert

- insertVertex(v) —> G.add_node(v)
- insertEdge(u, v) —> G.add_edge(u, v)

Remove

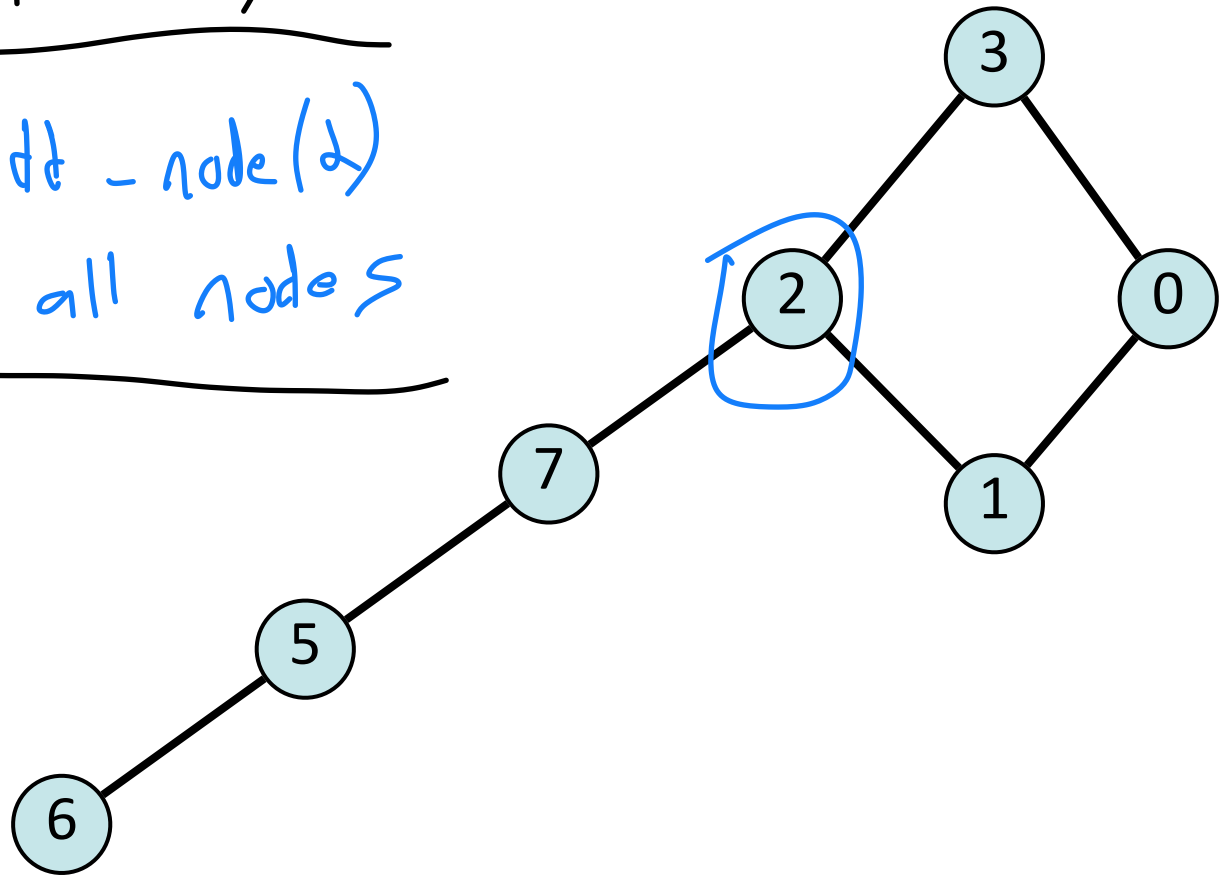
- removeVertex(v) —> G.remove_node(v)
- removeEdge(u, v) —> G.remove_edge(u, v)

Graph Practice 1: Build the following graph

Optionally:

G.add_node(2)

Add all nodes



Networkx has a graph class:

1. Make new graph object

G = nx.Graph()

2. Add every edge once

Graph Practice 1: Build the following graph

We can build a graph in NetworkX by adding edges one at a time:

```
G = nx.Graph()

G.add_edge(0, 1)

G.add_edge(1, 2)

G.add_edge(2, 3)

G.add_edge(3, 0)

G.add_edge(5, 6)

G.add_edge(5, 7)

G.add_edge(7, 2)
```

Graph Practice 1: Build the following graph

Given a list of edges, we can build the graph all at once

```
G = nx.Graph([(0, 1), (0, 3), (1, 2), (2, 3), (2, 7), (5, 6), (5, 7)])
```

Given a NumPy matrix, we can build the graph all at once

```
G = nx.Graph(<NumPy Adjacency Matrix>)
```

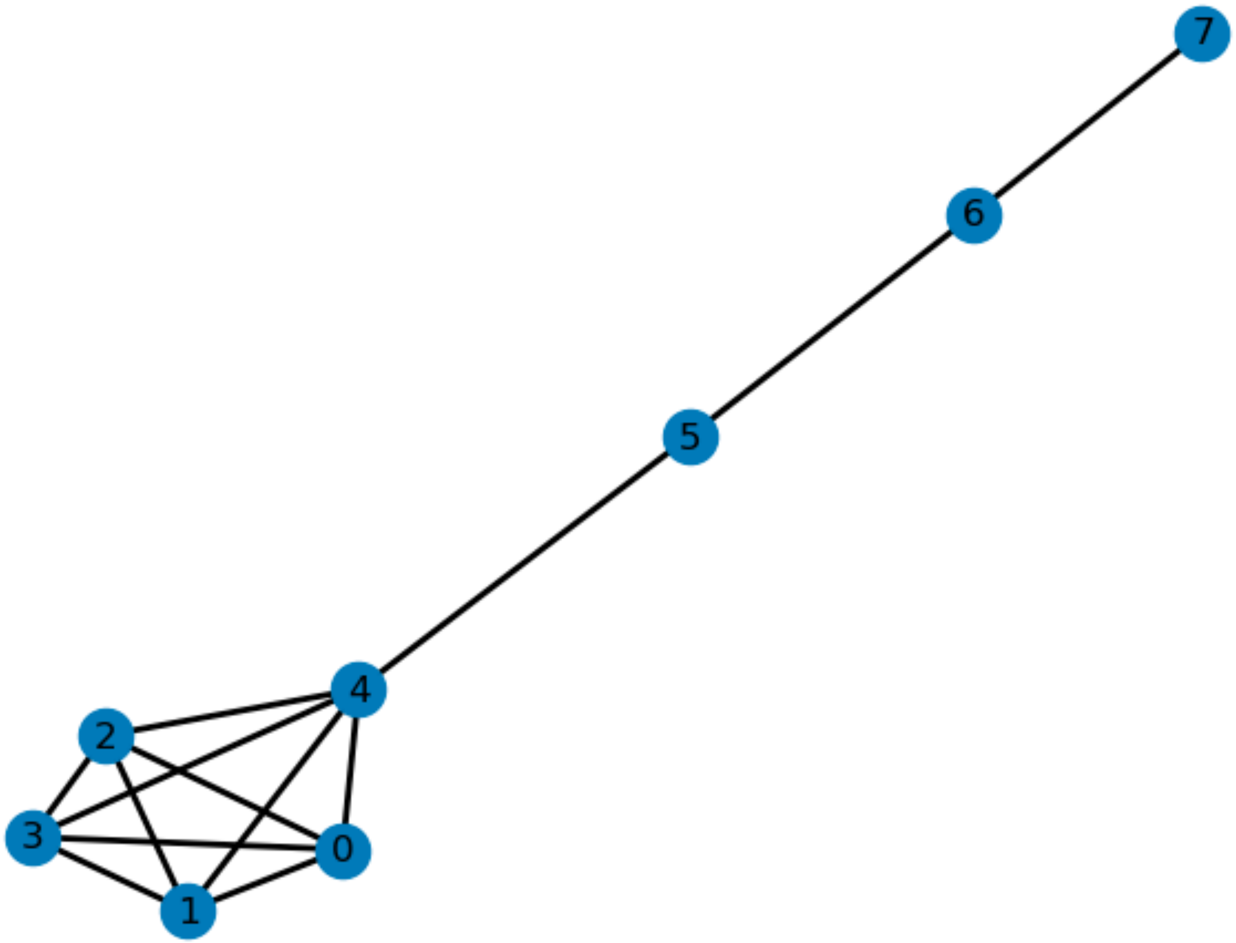
Given a file in format edge list or adjacency list

```
G = nx.read_edgelist(<edgeList file>)

G = nx. read_adjlist(<adjList file>)
```

Why not adjacency matrix?

Graph Practice 2: Remove all odd vertices

lollipop graph()

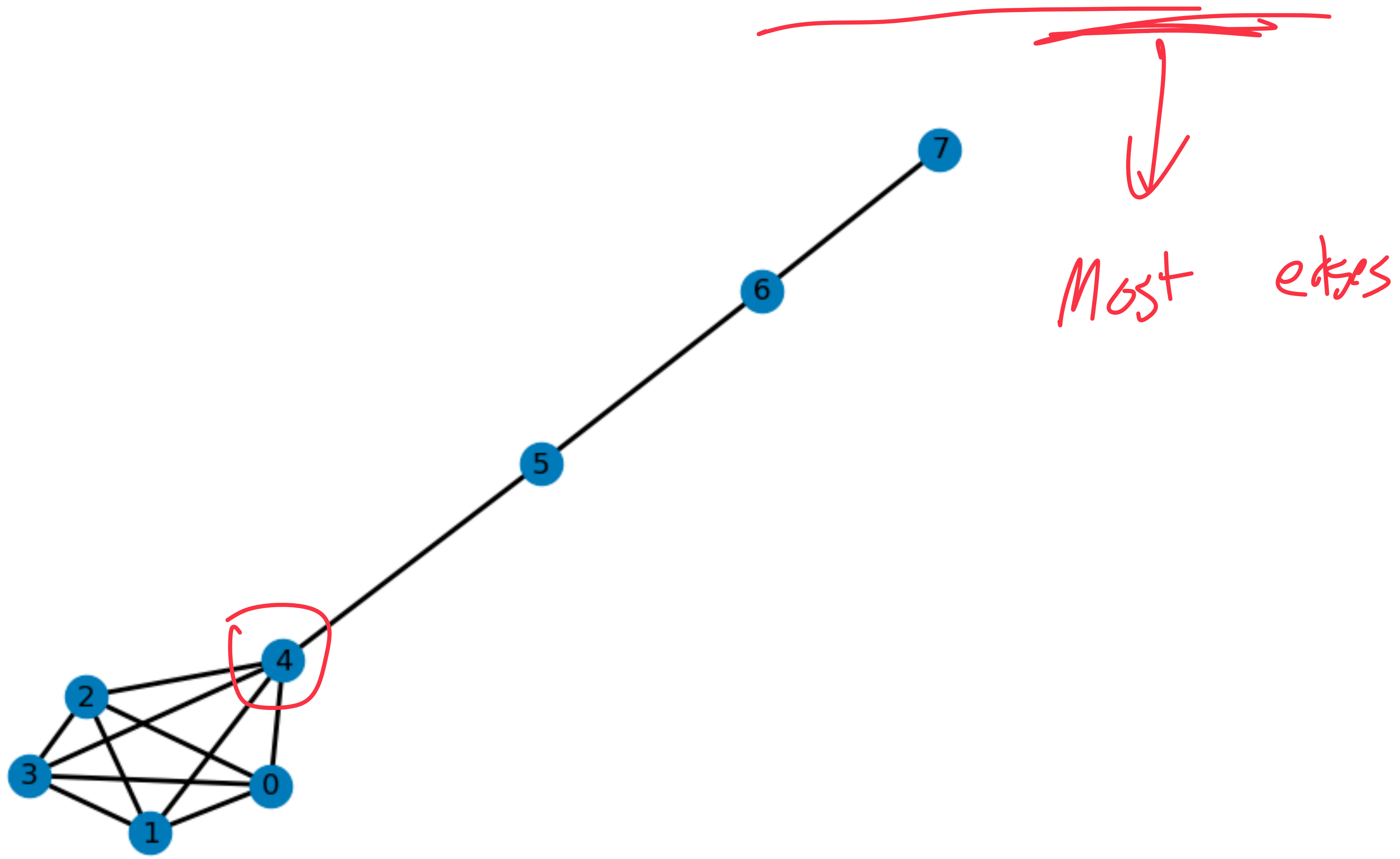


Graph Practice 2: Remove all odd vertices

G.nodes() by default returns a **dictionary**.

```
nodes = list(G.nodes())

for n in nodes:

    if n % 2 == 1:

        G.remove_node(n)
```

Graph Practice 3: Find the highest degree vertex



Graph Practice 3: Find the highest degree vertex

We can build a graph in NetworkX by adding edges one at a time:

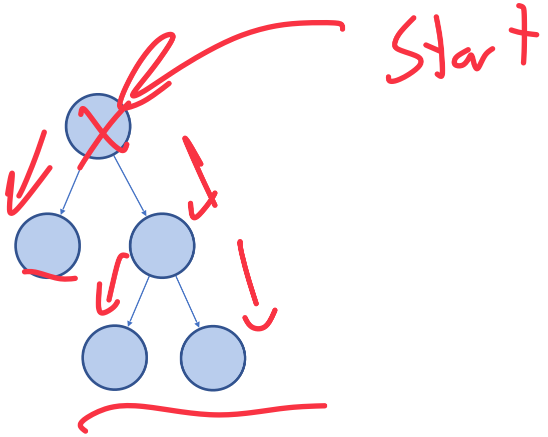
```
max = -1
v = None

for n in G.nodes():

    if len(G[n].keys()) > max:

        max = len(G[n].keys())

        v = n

print(v, max)
```

Graph Traversals

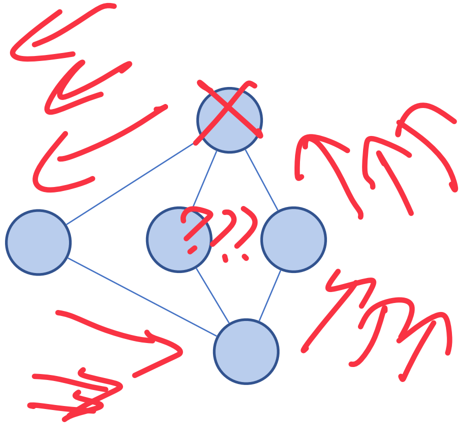
There is no clear order in a graph (even less than a tree!)

How can we systematically go through a complex graph in the fewest steps?

Tree traversals won't work — lets compare:

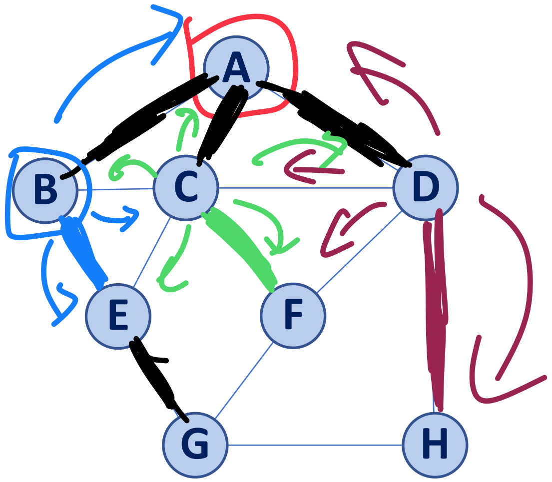


- Rooted
- Acyclic
- Clear base cases ('doneness')



- Arbitrary starting point
- Can have cycles
- Must track visited nodes directly

Simple BFS Traversal



1) Create a queue and a visit list

2) Initialize both to contain our start

3) While queue not empty:

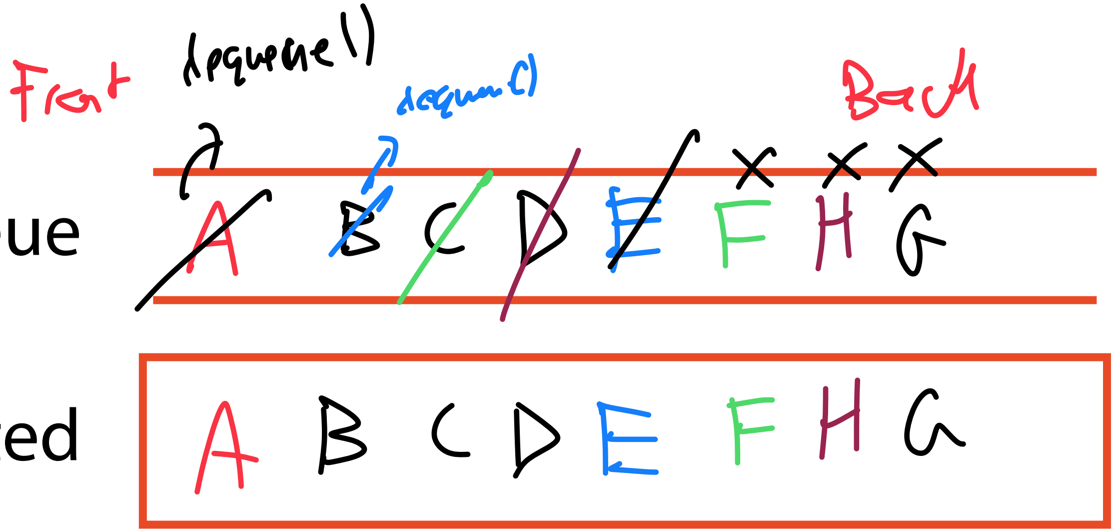
Remove front vertex of queue

Check if each edge has been seen before

Add unvisited edges to queue (and list)

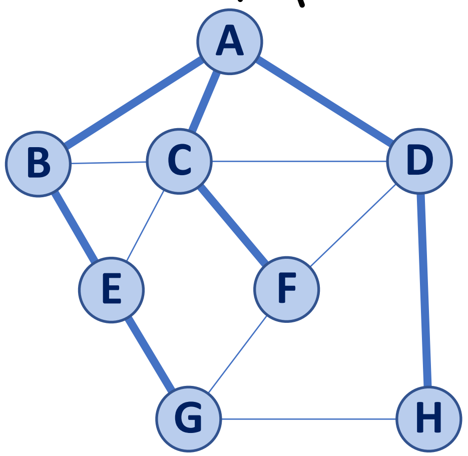
Queue

Visited



Simple BFS Traversal

$|V| = n \quad |E| = m$



1) Create a queue and a visit list

2) Initialize both to contain our start

3) While queue not empty:

   Remove front vertex of queue

   Check if each edge has been seen before

   Add unvisited edges to queue (and list)

Back ??? Front

Queue: ~~G H F E D C B A~~

Visited: A B C D E F H G

What is my runtime?

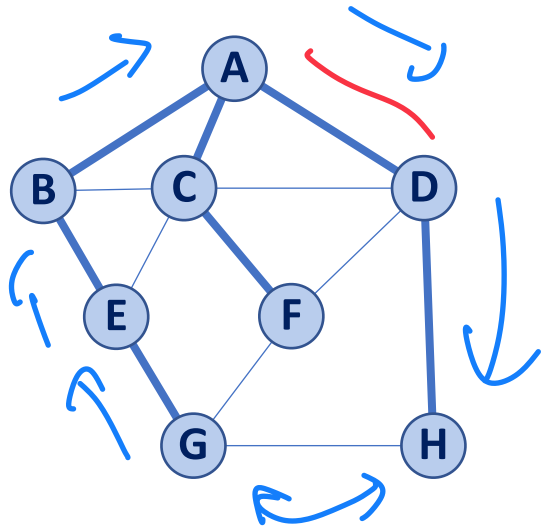
$O(n + m)$

n ↑ every vertex once

m ↑ every edge twice

Simple BFS Traversal

shortest path from start to anything



From G to H shortest path?

1 not 5!

What is the shortest distance from **A** to **H**?

A → D → H

2

What is the shortest path from **A** to **H**?

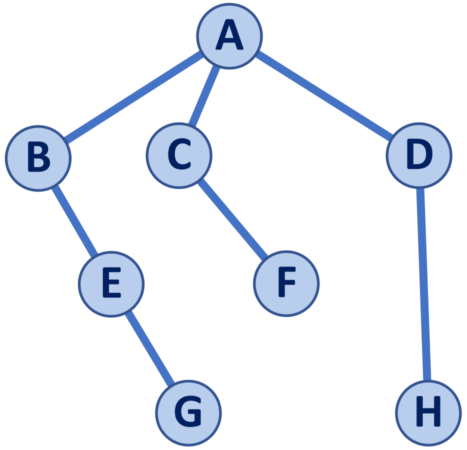
What is the shortest path from **A** to **F**?

A - C - F

What is the shortest distance from **A** to **F**?

2

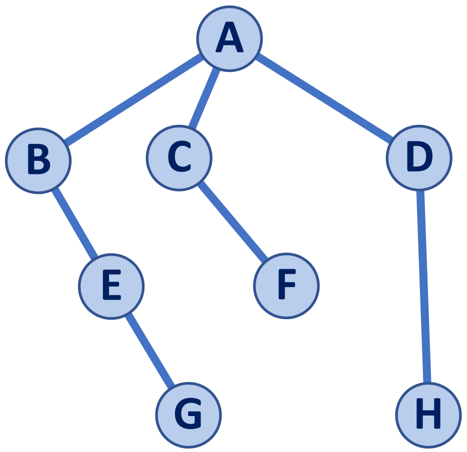
Simple BFS Traversal



acyclic graph

What data structure is this? a tree!

Simple BFS Traversal

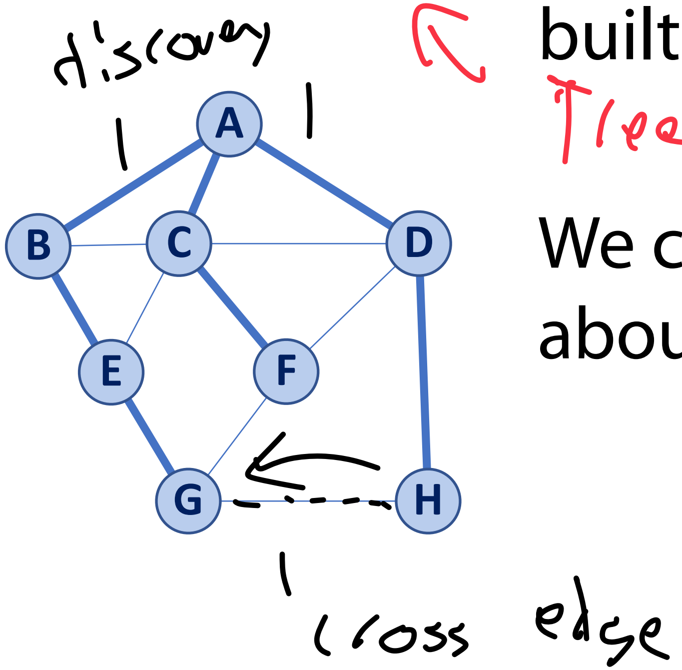


A **minimum spanning tree** is a tree formed by a subset of graph edges such that all vertices are connected with the smallest total possible edge weight

= 1 ??

On an unweighted, undirected graph this MST can be built by tracking **discovery** edges during a BFS traversal

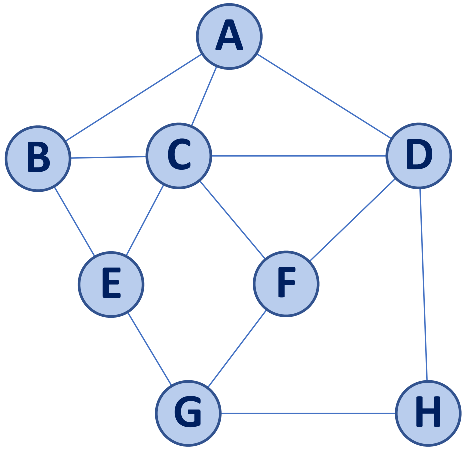
Tree encodes shortest path from A to anything

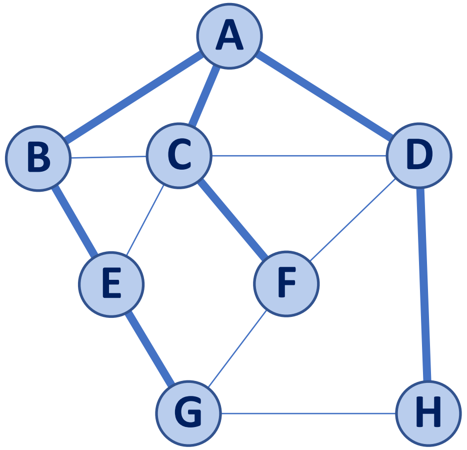


We call the remaining edges **cross** edges. What can I say about a graph with at least one **cross** edge?

↳ There is a cycle

Traversal: BFS



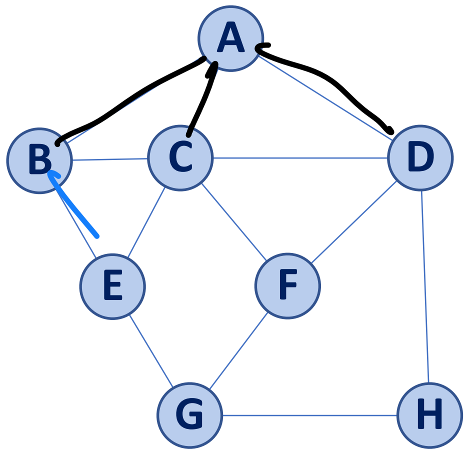


If we modify our BFS traversal algorithm, we can track both **distances** and **discovery edges!**

shortest path

cycles or not

Traversal: BFS



Replace 'visited' list with a **distance** and a **previous**

When we add to queue, record **previous**.

When we process vertex from queue, record **distance**.

"Unvisited" vertices have neither **distance** or **previous**

Vertex	Distance	Previous
A	0	None
B	0+1 = 1	A
C	0+1 = 1	A
D	0+1 = 1	A
E	1+1 = 2	B
F	2	C
G	3	E
H	2	D

F(ront)?

Queue

A

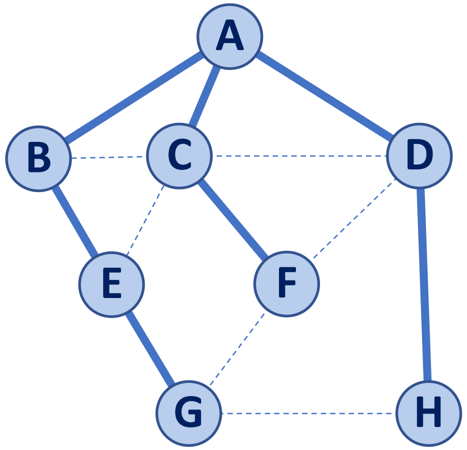
Traversal: BFS

Replace 'visited' list with a **distance** and a **previous**

When we add to queue, record **previous**.

When we process vertex from queue, record **distance**.

"Unvisited" vertices have neither **distance** or **previous**



Vertex	Distance	Previo
A	0	-
B	1	**A**
C	1	**A**
D	1	**A**
E	2	**B**
F	2	**C**
G	3	**E**
H	2	**D**

Shortest path

Queue: G H F E D C B A

BFS Traversal using NetworkX

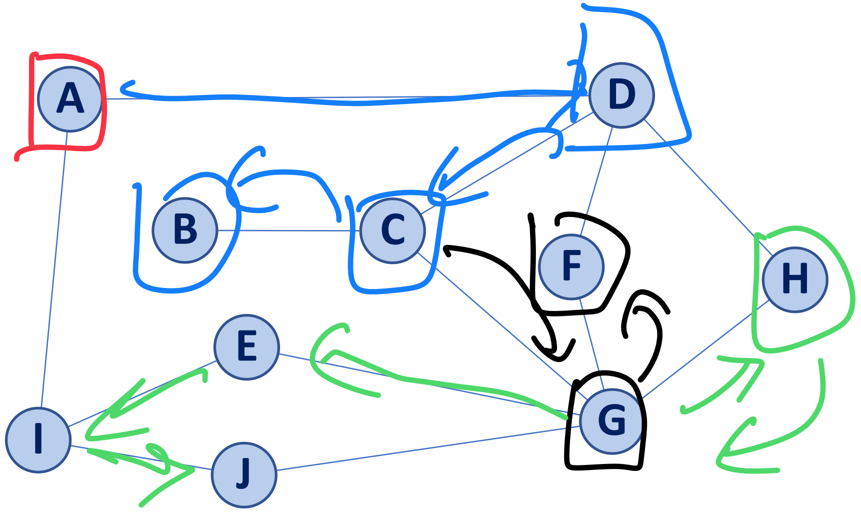
There are many different methods for running a BFS (different output):

```
G = nx.random_regular_graph(3, 6)

print(list(nx.bfs_edges(G, 0)))

print(list(nx.bfs_predecessors(G, 0)))

print(nx.descendants_at_distance(G, 0, 0))

print(nx.descendants_at_distance(G, 0, 1))

print(nx.descendants_at_distance(G, 0, 2))

print(nx.descendants_at_distance(G, 0, 3))

T = nx.bfs_tree(G, 0)
```

2 different traversal outputs

distance traversal output

Traversal: DFS



1) Create stack, initialize w/ start

while stack not empty:

- use top() to look at "current"
- add unvisited child to stack "push"
  - ↳ Mark as visited

If no unvisited children pop()

Visited:

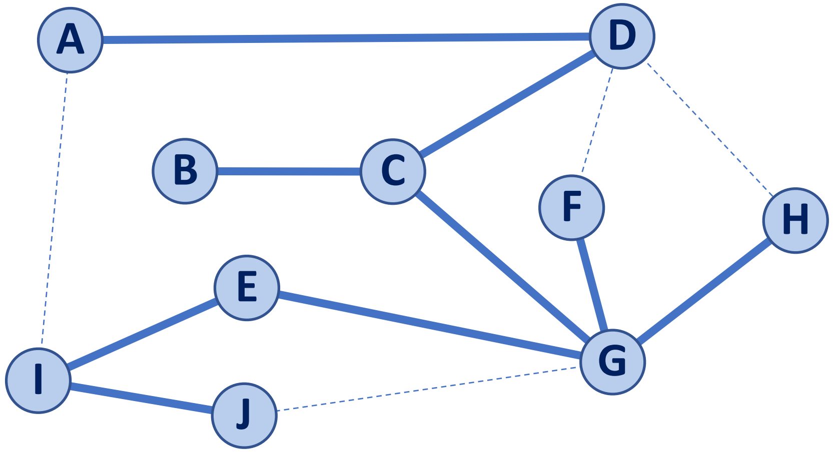
A D C B G F H E I J

Stack:

A D C B G F H E I J

Top

Traversal: DFS



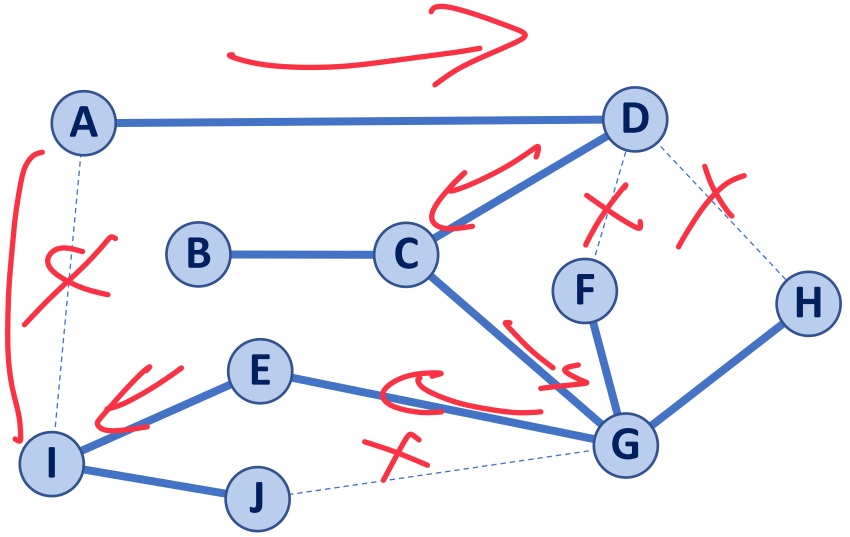
1) Create a stack and a visit list
2) Initialize both to contain our start
3) While stack not empty:

   Use **top()** to look at current vertex

   If no unvisited children, **pop()**

   Otherwise, **push()** the first unvisited child

Traversal: DFS



Discovery Edge

Back Edge

Do we still make a spanning tree?

↳ Yes! All traversals make spanning trees *

* unweighted graphs

Does distance have meaning here?

No, no distance meaning

Do our edge labels have meaning here?

↳ Yes; Back edge ≡ cross edge is a cycle

DFS Traversal using NetworkX

What can the BFS do that the DFS cannot do?

```
G = nx.random_regular_graph(3, 6)


print(list(nx.dfs_edges(G, 0)))


print(list(nx.dfs_predecessors(G, 0)))


print(list(nx.find_cycle(G, 0)))


T = nx.dfs_tree(G, 0)
```

Path

Traversal & return

Dictionary of pred
previous

cycle detection!

Spanning Tree

DFS vs BFS Runtime

DFS: $O(n+m)$

Use Cases: Spanning tree
Detect cycles

Peak Memory Cost:
↳ longest path

BFS: $O(n+m)$

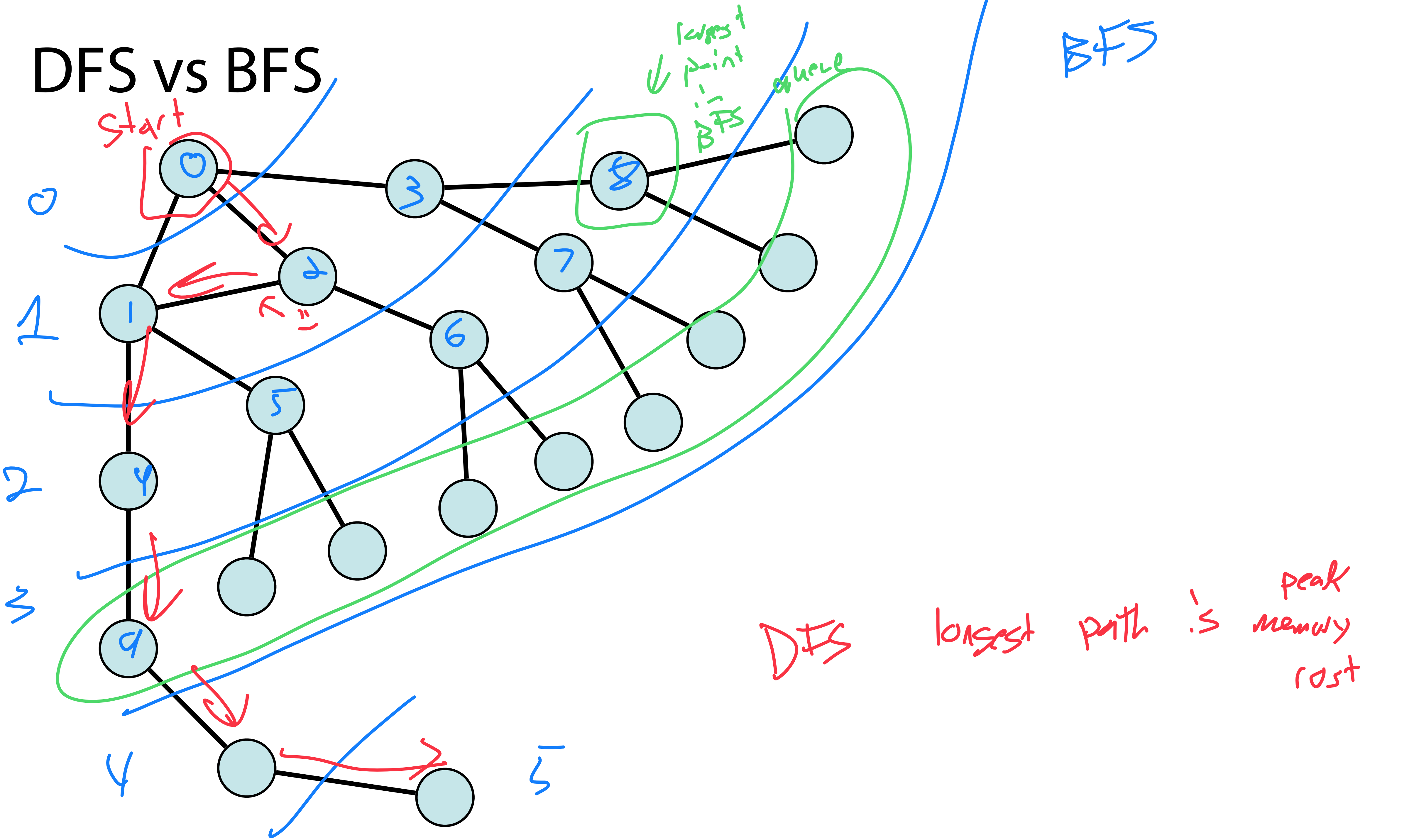
Use Cases: Spanning tree
Cycles
Shortest path

Peak Memory Cost:
↳ Bad peak memory ≡ width of graph

In real world longest path is usually shorter than width of graph

Outside class scope:
Much more flexible in use cases
or "choice of next"

DFS vs BFS

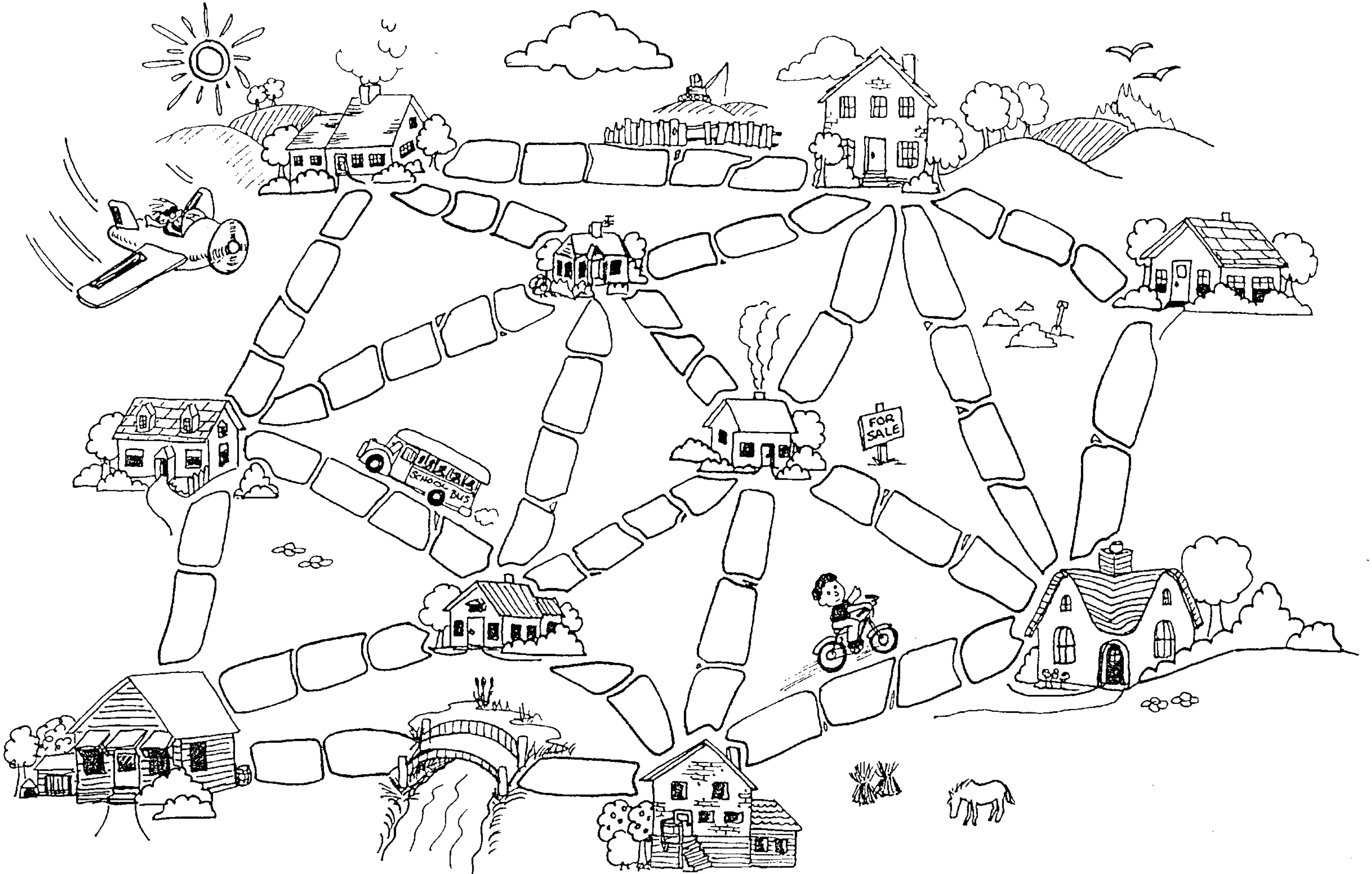

start
longest point in BFS
queue
BFS
DFS longest path 's peak memory cost

Where do we go from here?

Consider: How does our implementation change for weights?

for directed edges?

Consider: What are some common graph algorithms / uses?


SCHOOL BUS
FOR SALE

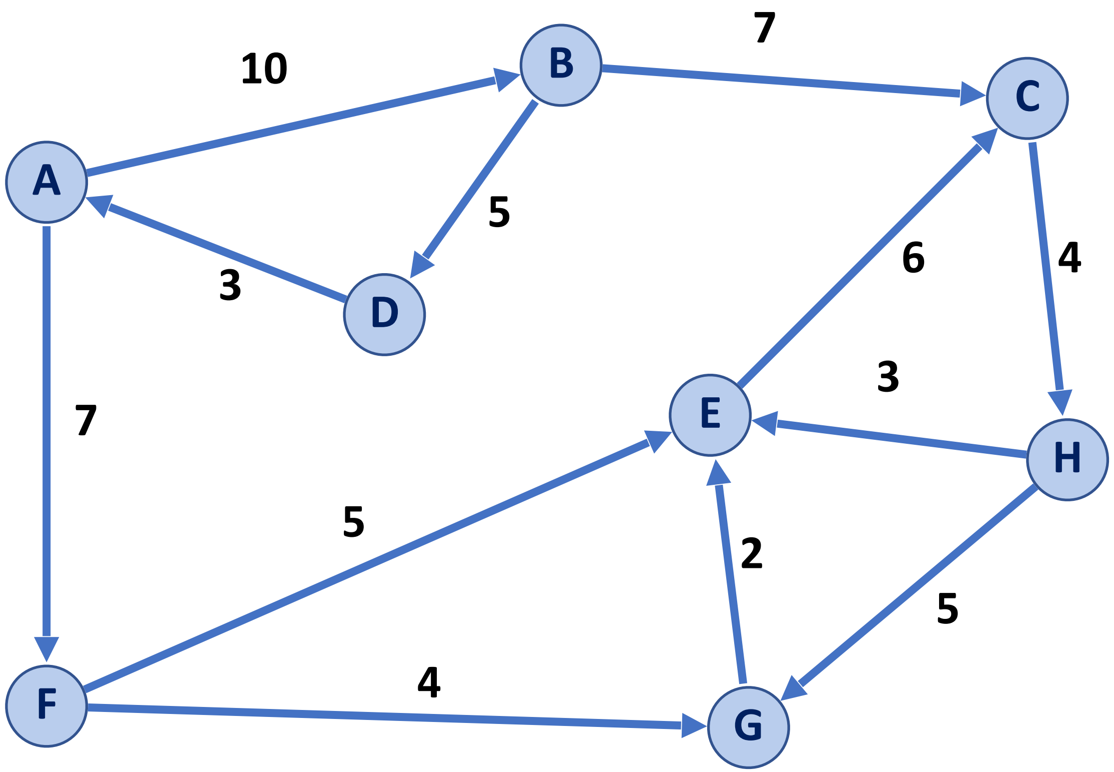
Graph Practice 4: Build a weighted graph

We can build a graph in NetworkX by adding edges one at a time:

with weights!

```
G = nx.Graph()

G.add_edge(0, 1, weight=5)

G.add_edge(1, 2, weight=1)

G.add_edge(2, 3, weight=2)

G.add_edge(3, 0, weight=6)

G.add_edge(5, 6, weight=3)

G.add_edge(5, 7, weight=7)

G.add_edge(7, 2, weight=2)
```

Weighted Directed Graphs



Graph Practice 5: Build a directed weighted graph

We can build a directed graph in NetworkX by using a DiGraph() object:

```
G = nx.DiGraph()

G.add_edge(0, 1, weight=5)

G.add_edge(1, 2, weight=1)

G.add_edge(2, 3, weight=2)

G.add_edge(3, 0, weight=6)

G.add_edge(5, 6, weight=3)

G.add_edge(5, 7, weight=7)

G.add_edge(7, 2, weight=2)
```

with weights!