Algorithms and Data Structures for Data Science
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)

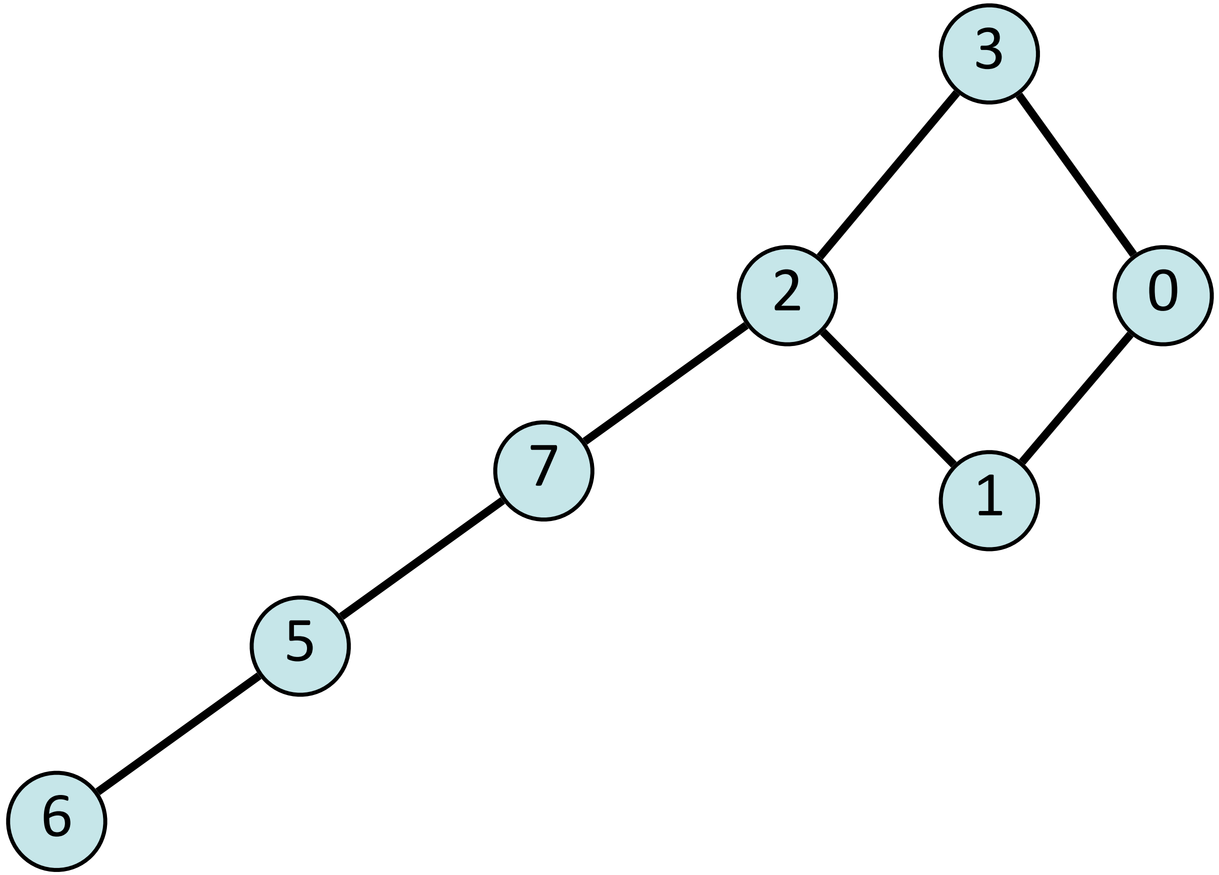
insertEdge(u, v) —> G.add_edge(u, v)

Remove

removeVertex(v) —> G.remove_node(v)

removeEdge(u, v) —> G.remove_edge(u, v)

Graph Practice 1: Build the following graph



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>)
```

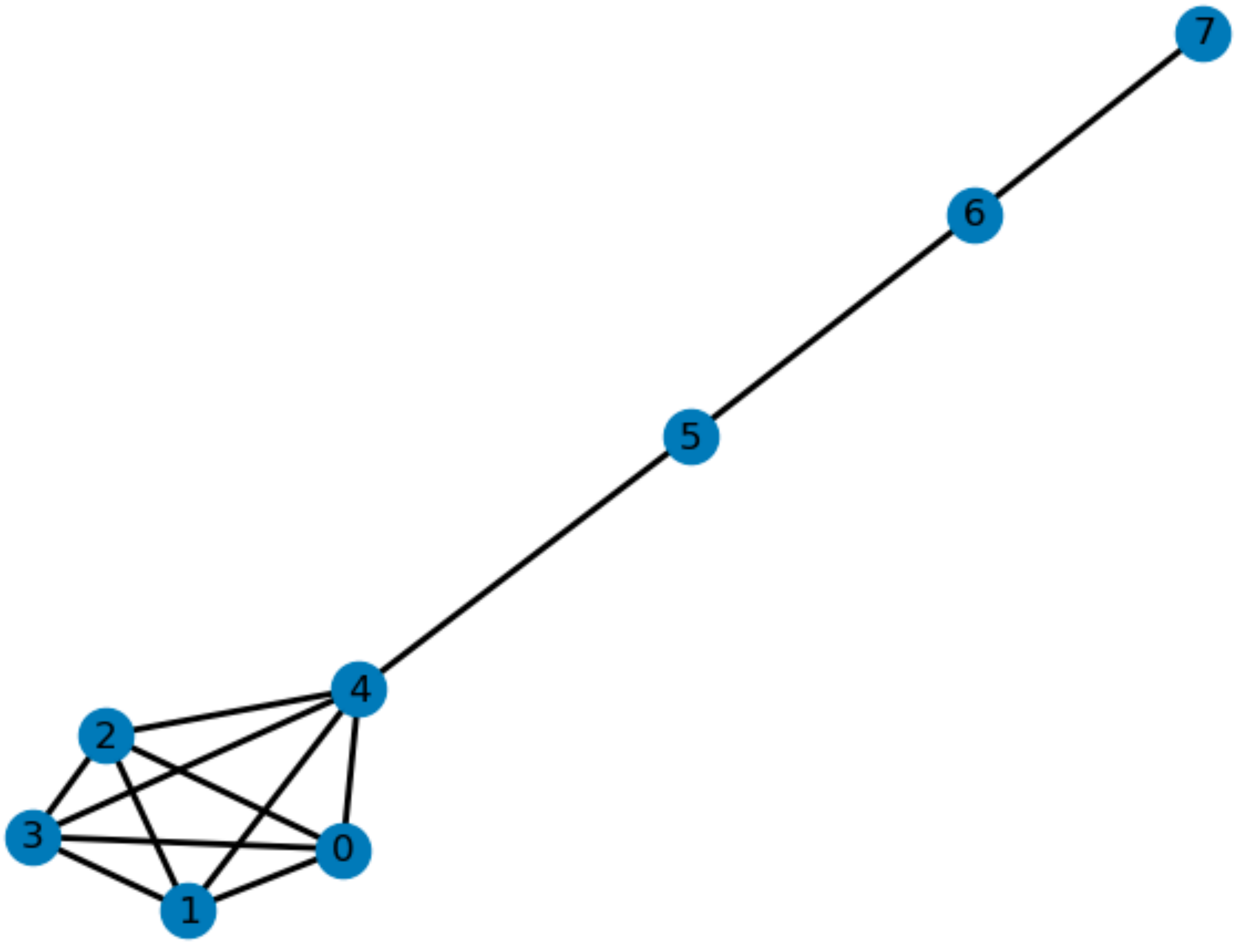
Given a file in format edge list or adjacency list

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

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

Why not adjacency matrix?

Graph Practice 2: Remove all odd vertices

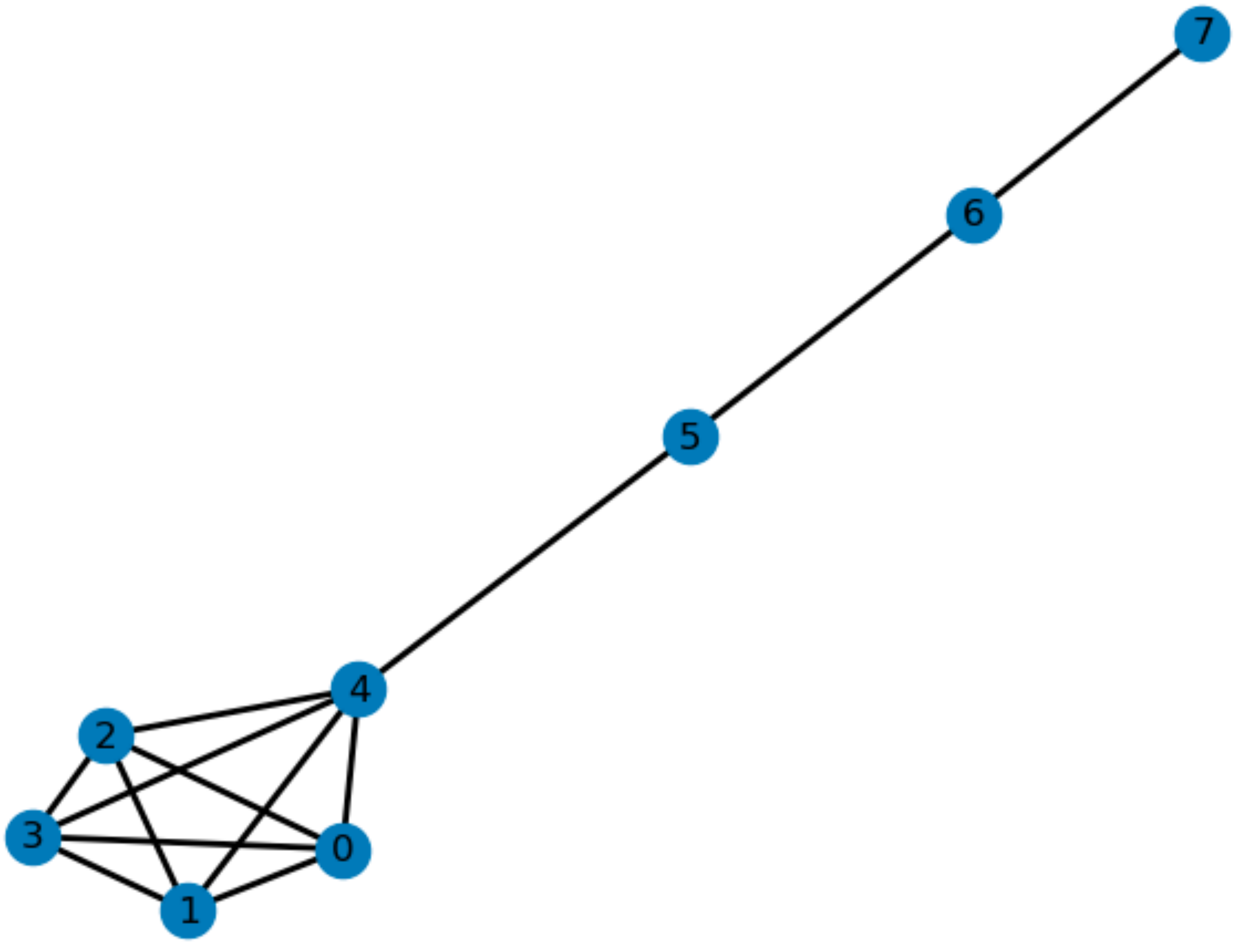


Graph Practice 2: Remove all odd vertices

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:

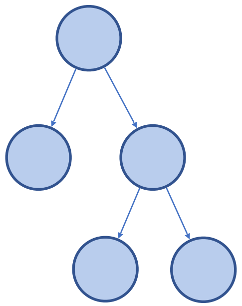
```
max = -1
v = None

for n in G.nodes():

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

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

        v = n

print(v, max)
```

Graph Traversals

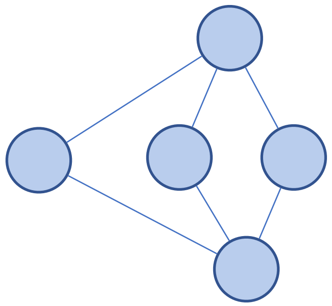
There is no clear order in a graph (even less than a tree!)

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

Tree traversals won't work — lets compare:



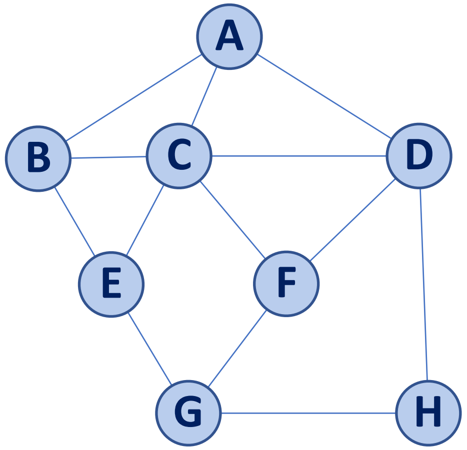
- Rooted
- Acyclic
- Clear base cases ('doneness')



- 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)

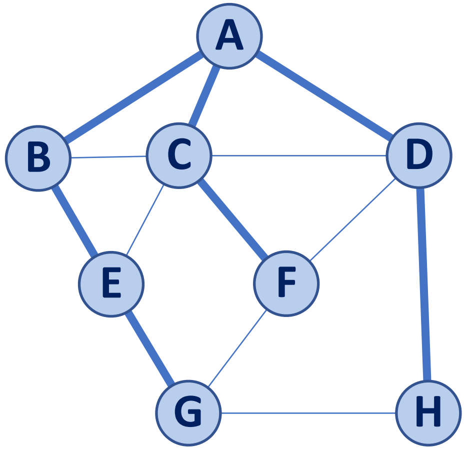


Queue

Visited

Simple BFS Traversal

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)

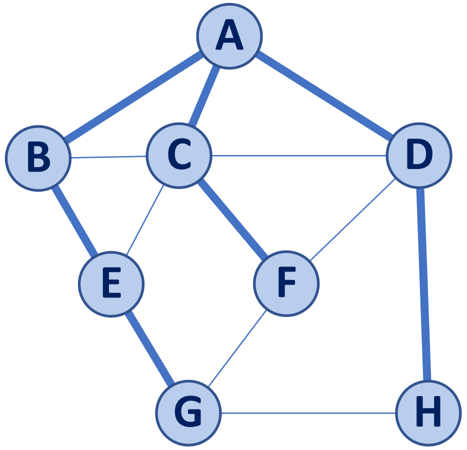


What is my runtime?

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

Visited A B C D E F H G

Simple BFS Traversal



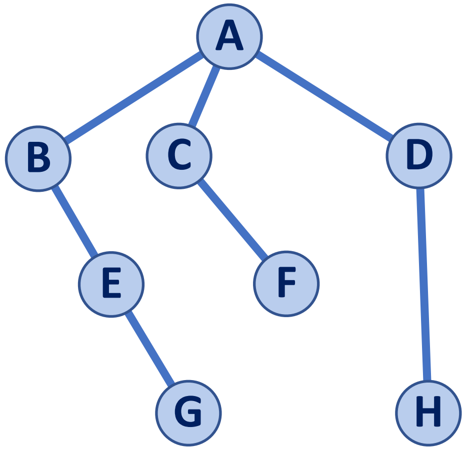
What is the shortest distance from **A** to **H**?

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

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

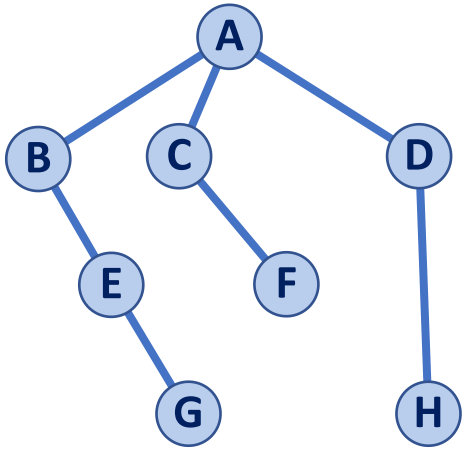
What is the shortest distance from **A** to **F**?

Simple BFS Traversal



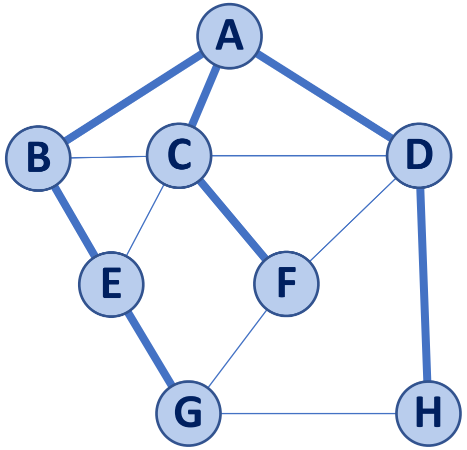
What data structure is this?

Simple BFS Traversal



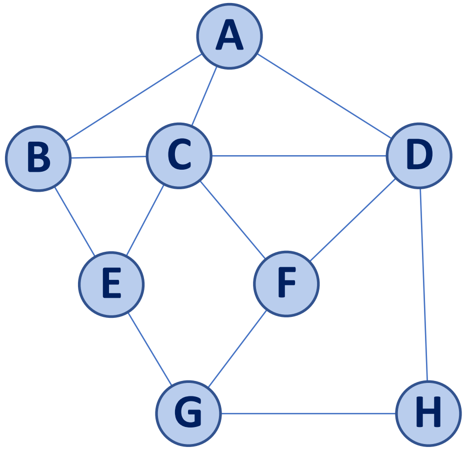
A **minimum spanning tree** is a tree formed by a subset of graph edges such that all vertices are connected with the smallest total possible edge weight

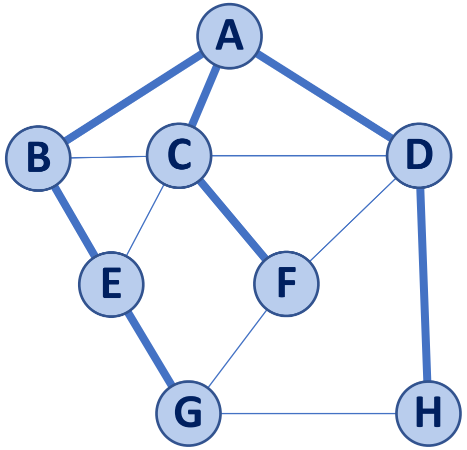
On an unweighted, undirected graph this MST can be built by tracking **discovery** edges during a BFS traversal



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

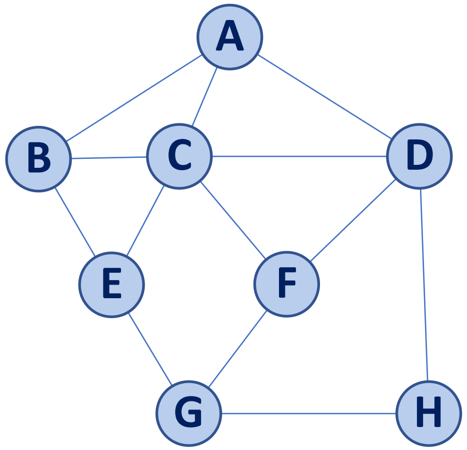
Traversal: BFS





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

Traversal: BFS



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

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

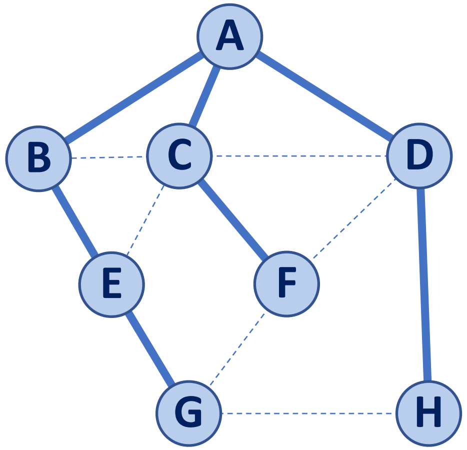
When we process vertex from queue, record **distance**.

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

Vertex	Distance	Previous
A		
B		
C		
D		
E		
F		
G		
H		

Queue

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**

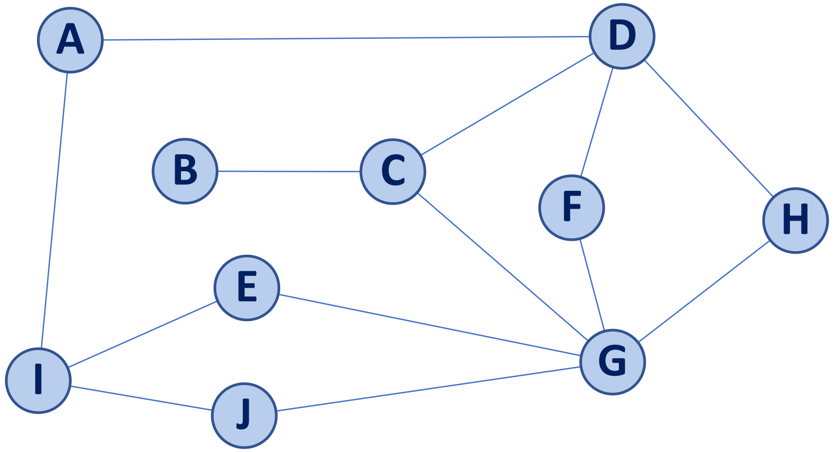
Queue ~~G~~ ~~H~~ ~~F~~ ~~E~~ ~~D~~ ~~C~~ ~~B~~ ~~A~~

BFS Traversal using NetworkX

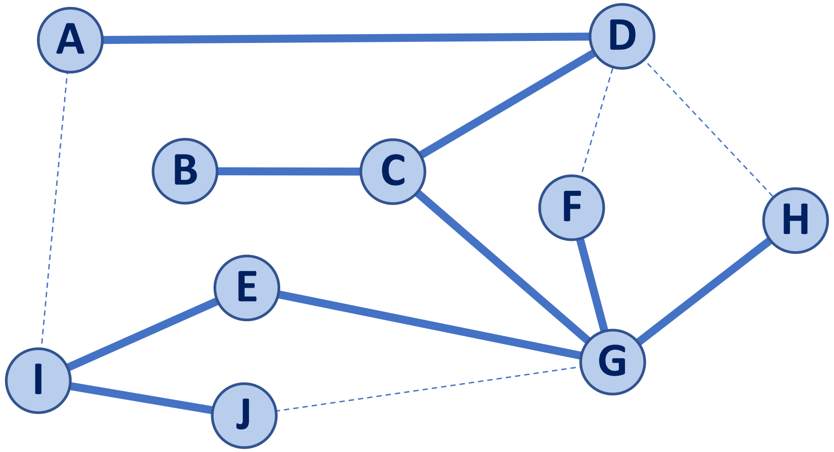
There are many different methods for running a BFS (different output):

```
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)
```

Traversal: DFS



Traversal: DFS



1) Create a stack and a visit list

2) Initialize both to contain our start

3) While stack not empty:

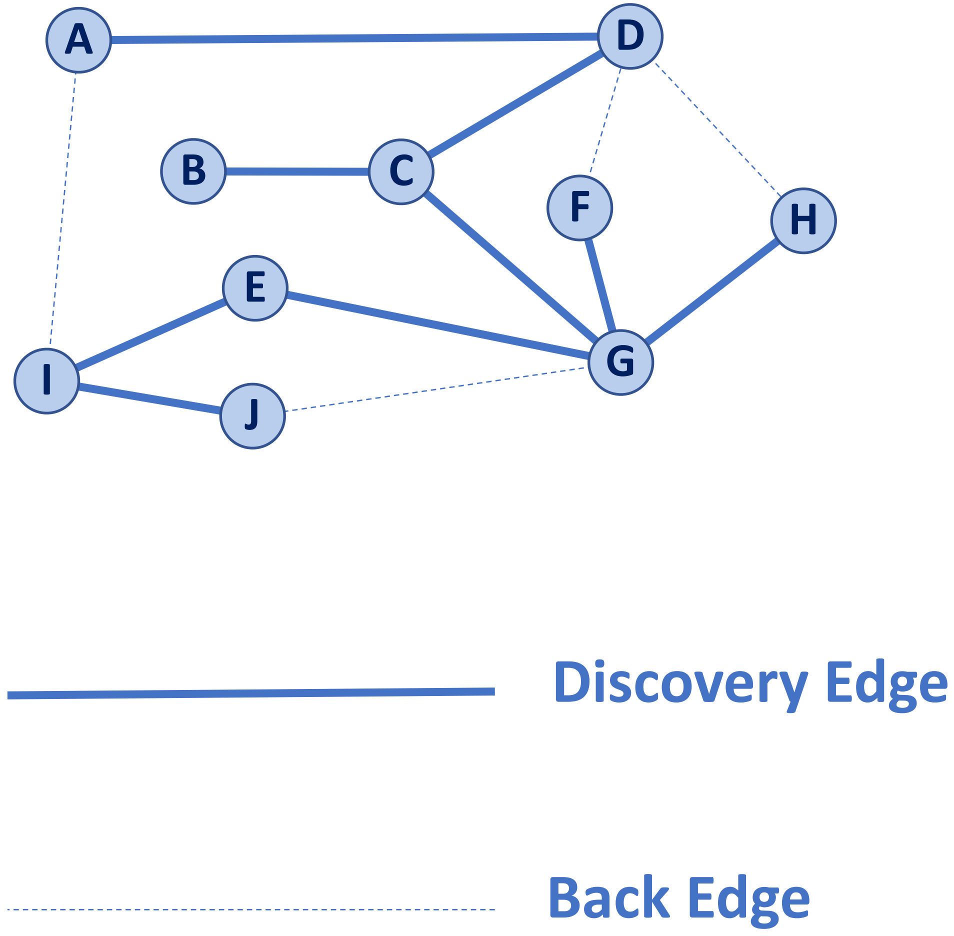
Use **top()** to look at current vertex

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

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

Traversal: DFS

Do we still make a spanning tree?



Does distance have meaning here?

Do our edge labels have meaning here?

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)
```

DFS vs BFS Runtime

DFS:

Use Cases:

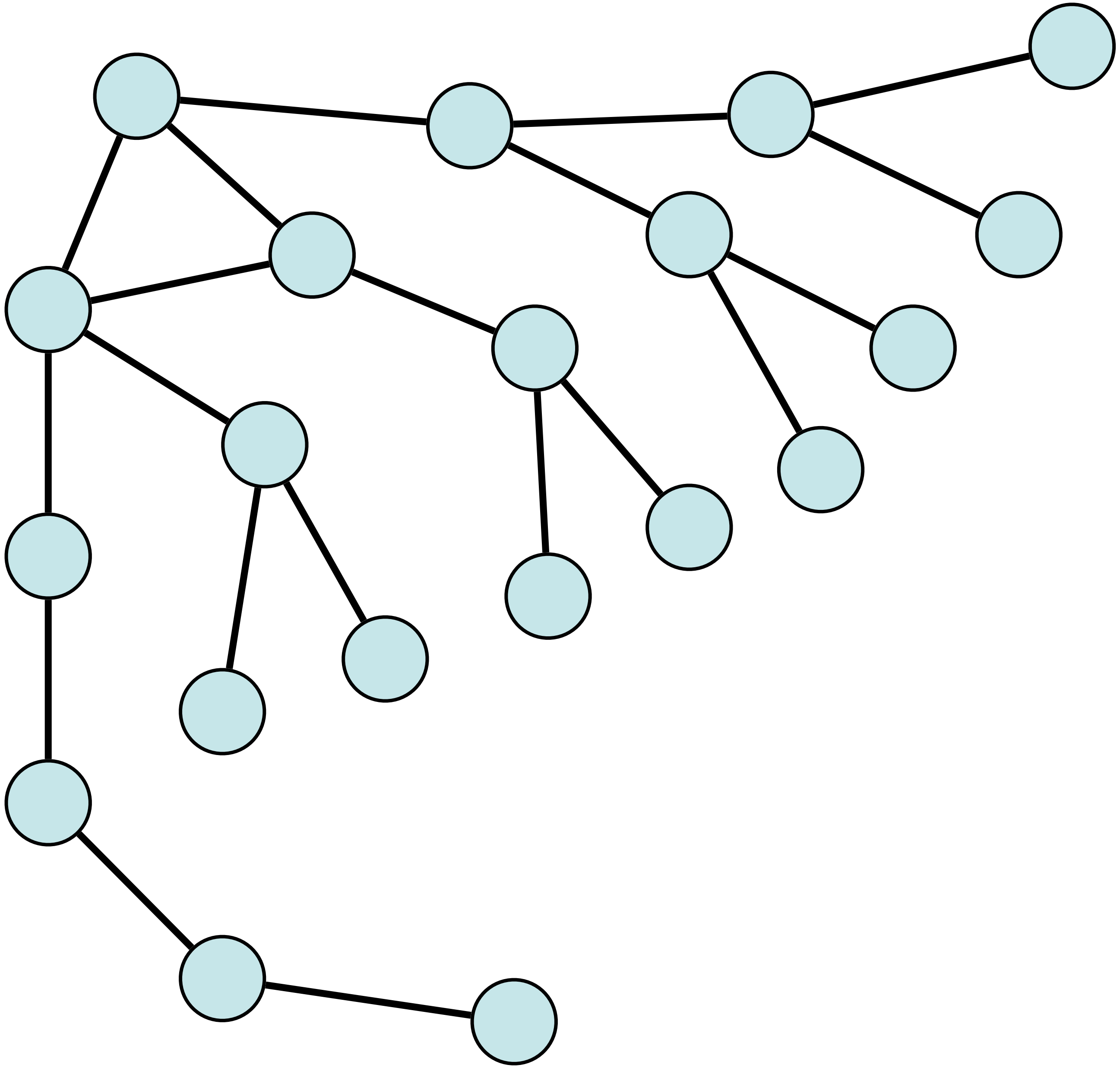
Peak Memory Cost:

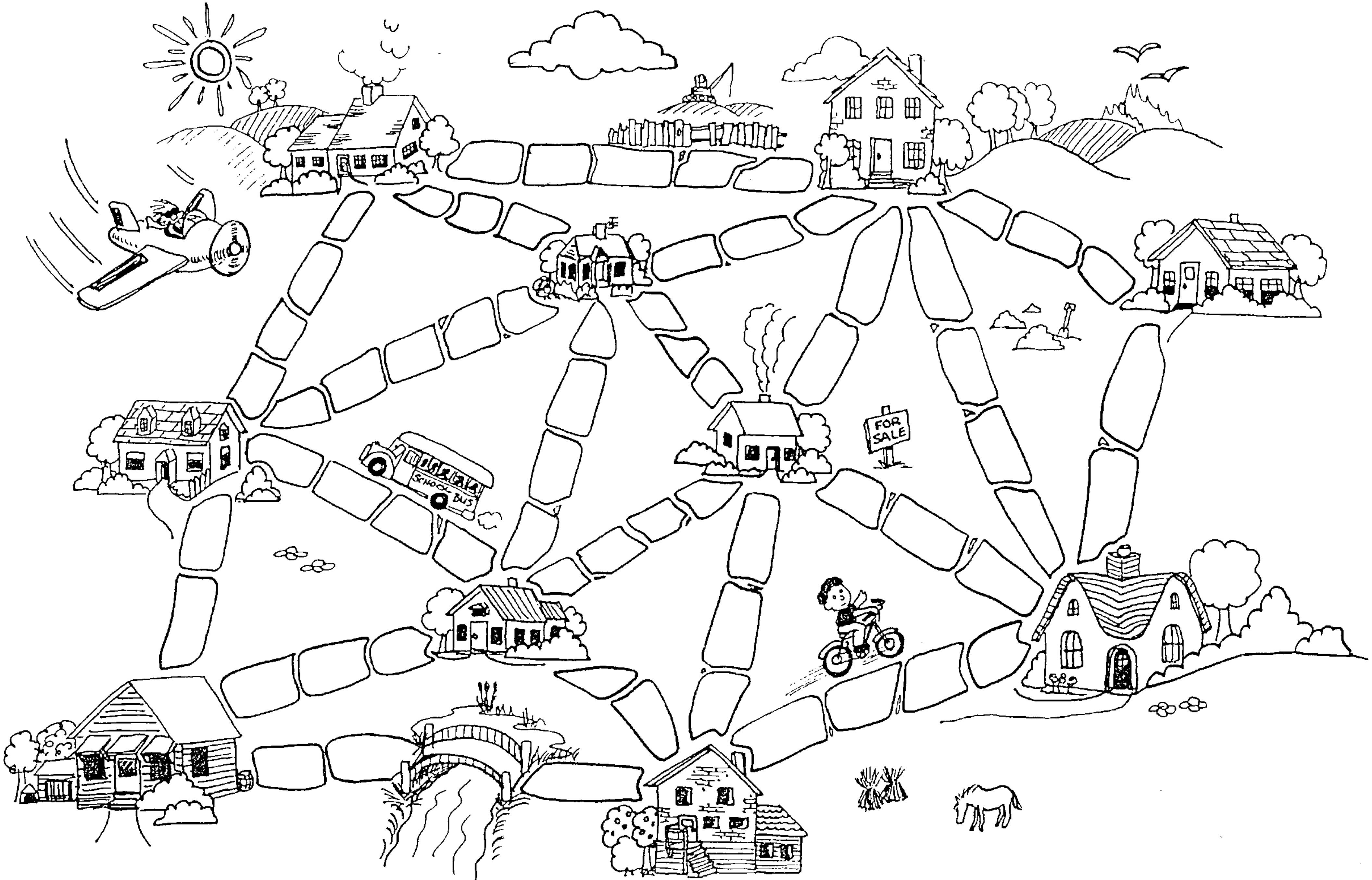
BFS:

Use Cases:

Peak Memory Cost:

DFS vs BFS




FOR SALE
SCHOOL BUS