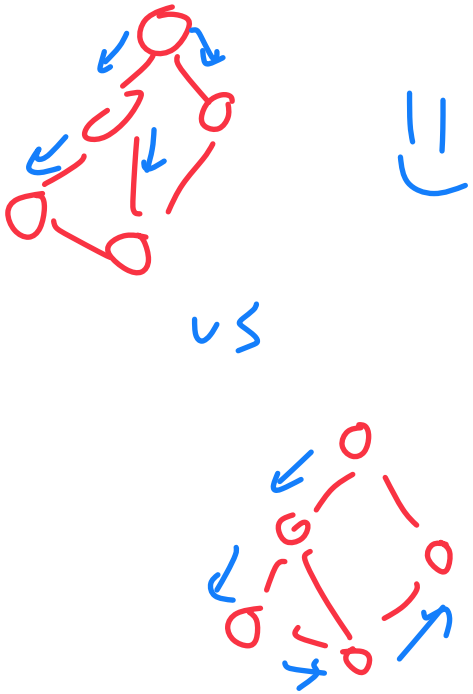
Data Structures

Graph Traversals

CS 225

Brad Solomon

April 3, 2026



UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

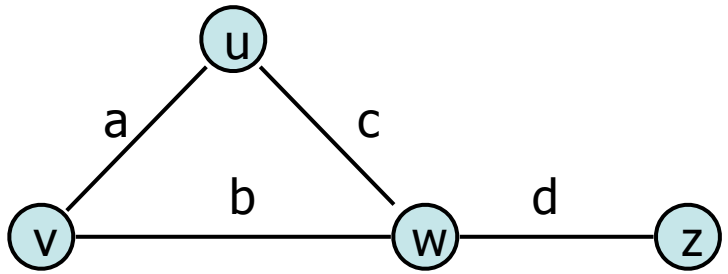
Department of Computer Science

Learning Objectives

Discuss graph traversal algorithms

Graph Implementation: Edge List $|V| = n, |E| = m$

The equivalent of an 'unordered' data structure



u
v
w
z

u	v	a
v	w	b
u	w	c
w	z	d

Vertex Storage:

An optional list of vertices

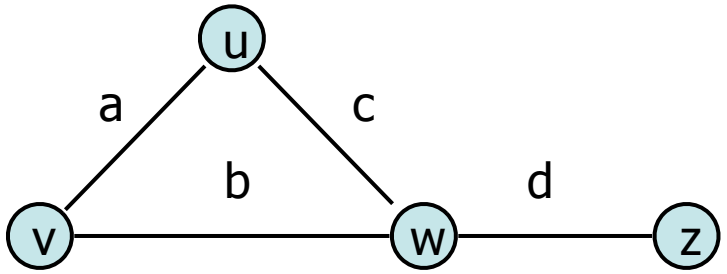
Edge Storage:

A list storing edges as (V1, V2, Weight)

Most graphs are stored as just an edge list!

Graph Implementation: Adjacency Matrix

$|V| = n, |E| = m$



u	0
v	1
w	2
z	3

	0	1	2	3
0	-	a	c	0
1		-	b	0
2			-	d
3				-

Vertex Storage:

A hash table of vertices

Implicitly or explicitly store index

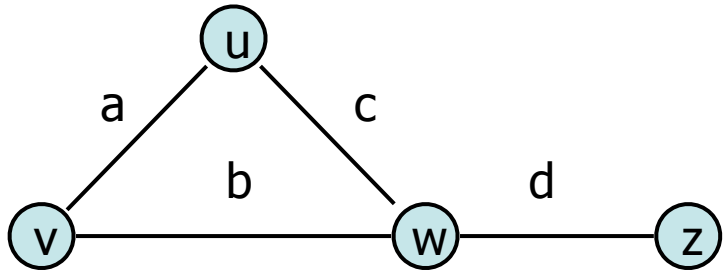
fixed size!

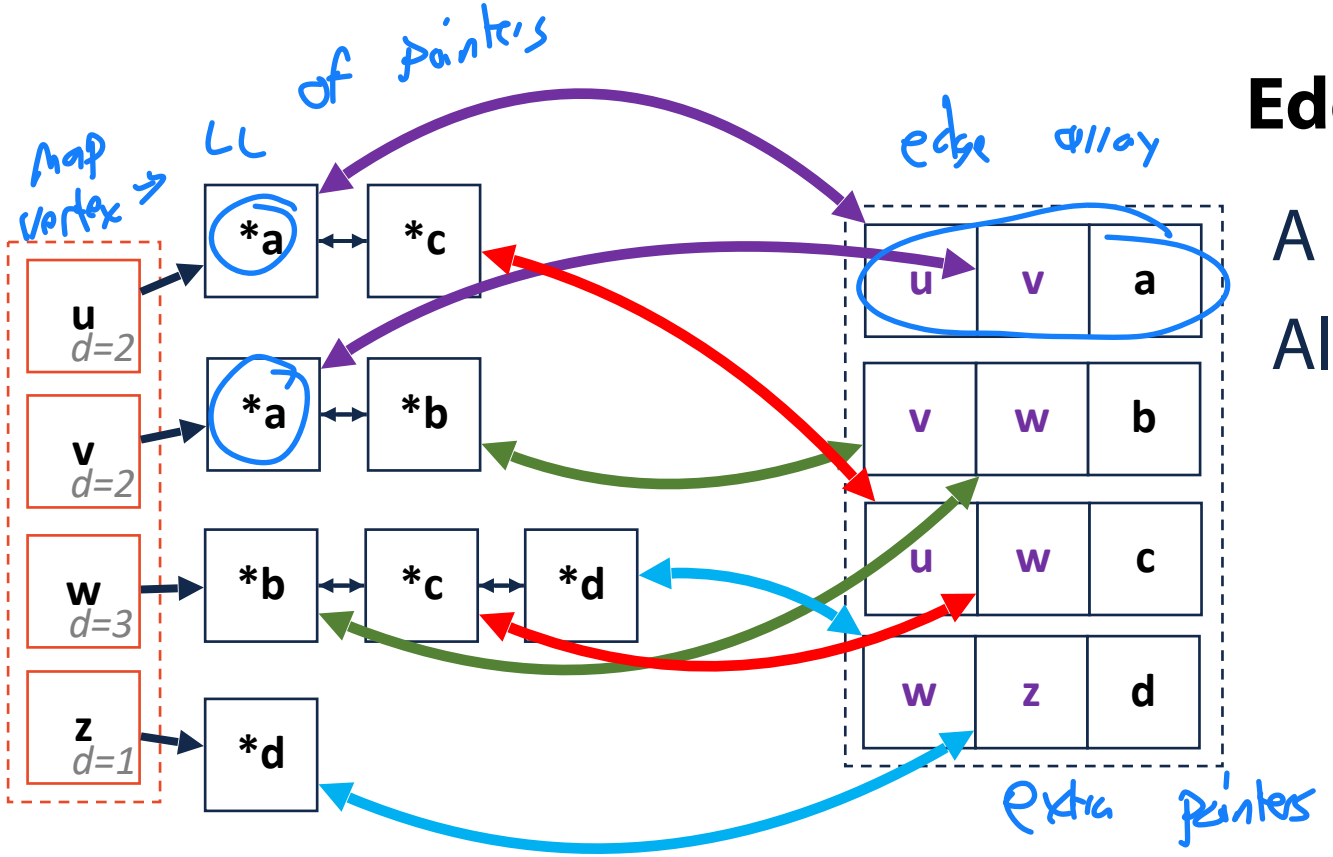
Edge Storage:

A |V| x |V| matrix of edges

Weight is stored at position (u, v)

Adjacency List





Vertex Storage:

A bidirectional linked list with size variable

Each node is a pointer to edge in edge list

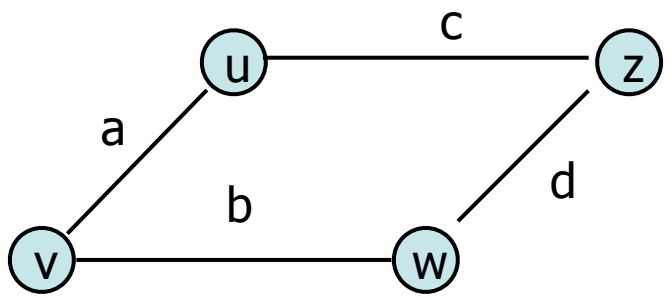
Edge Storage:

A list of (v1, v2, weight) edges

Also store pointers back to nodes

Adjacency List

|V| = n, |E| = m



removeVertex(z)

Adj List Node:

Linked list!

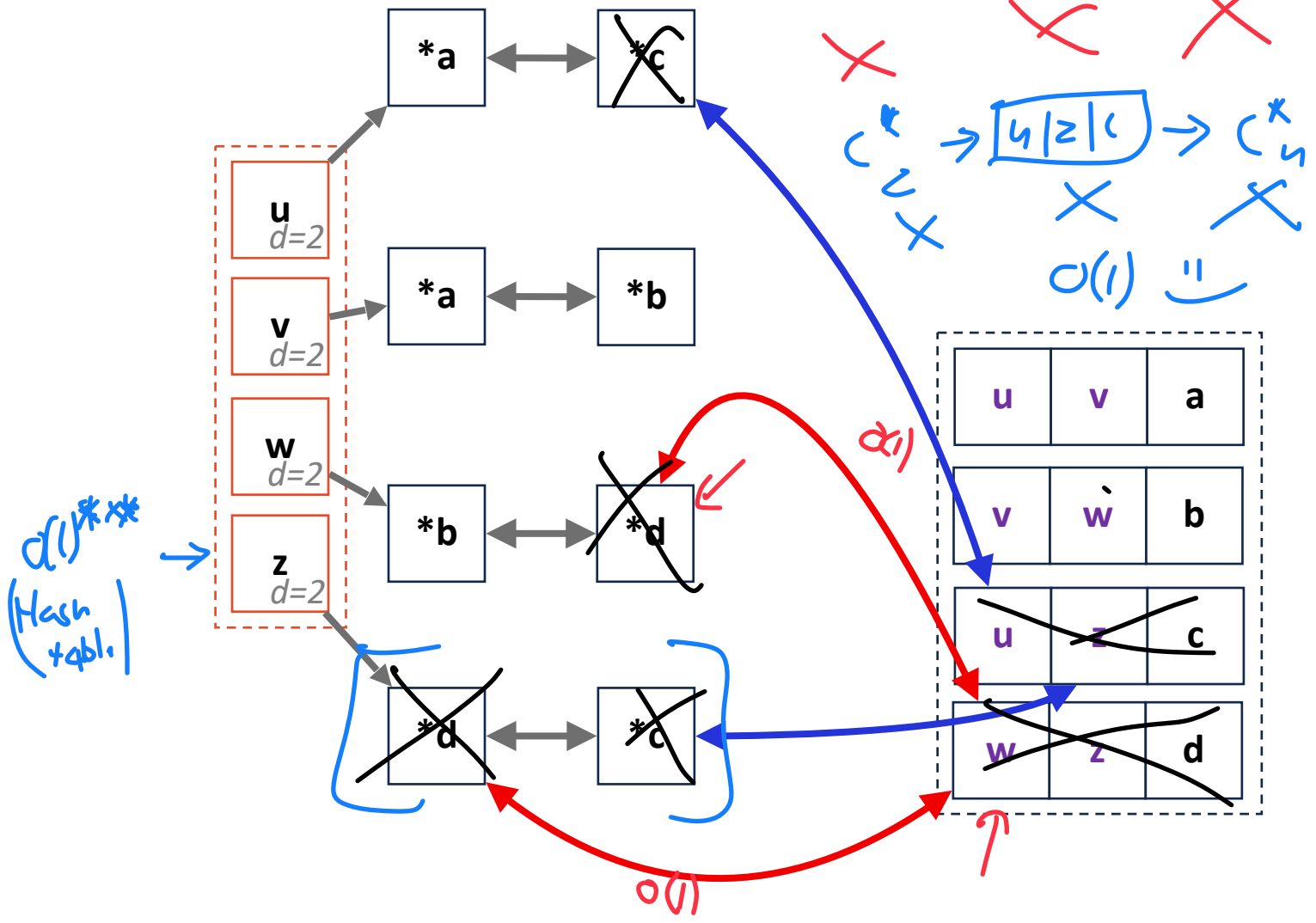
*Prev	*Edge	*Next

↳ Points to edge list

Edge List:

V1	V2	Weight
*V1	*V2	

Points to LL1 — Points to LL2



$|V| = n, |E| = m$

Expressed as O(f)	Edge List	Adjacency Matrix	Adjacency List
Space	n+m	n^2	n+m
insertVertex(v)	1*	n*	1*
removeVertex(v)	n+m	n	deg(v)
insertEdge(u, v)	1	1	1*
removeEdge(u, v)	m	1	min(deg(u), deg(v))
incidentEdges(v)	m	n	deg(v)
areAdjacent(u, v)	m	1	min(deg(u), deg(v))

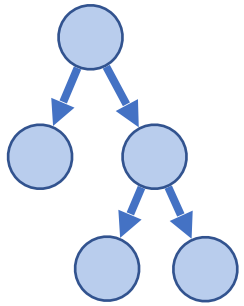
Graph Traversals

→ Solve a maze
→ Find a shortest path
→ Make a spanning tree
...
find some structural property about graph

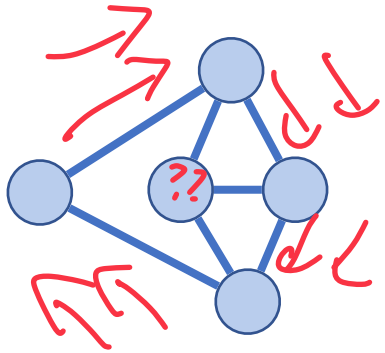
Objective: Visit every vertex and every edge in the graph.

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

Tree traversals won't work — lets compare:



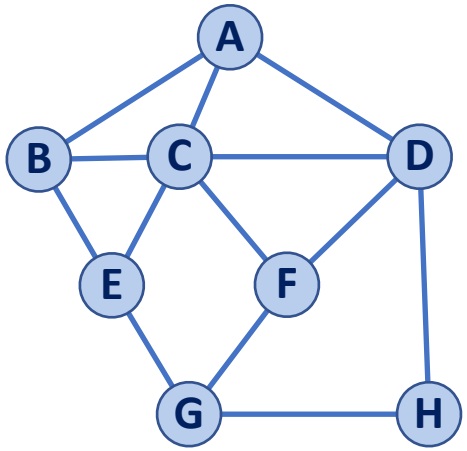
- Rooted
- Acyclic
- Notion of completion



- No clear root
- Cycles
- When have we finished?

Traversal: BFS

Not a full implementation



To complete a traversal what do we need?

As input:

1) A starting vertex

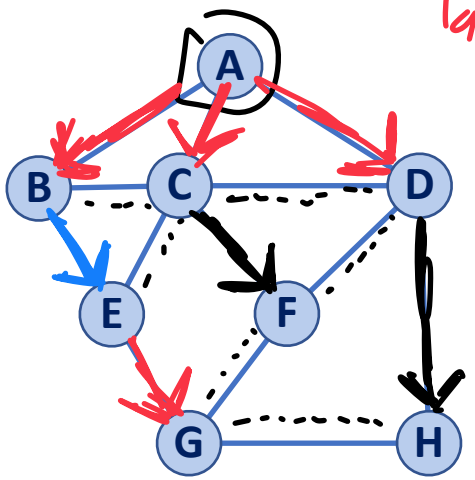
To keep track of: and edges

↳ Nodes that I have visited

↳ My current position

We are done if... all nodes & edges are here

Traversal: BFS



→ Discovery edges

.... Cross edges

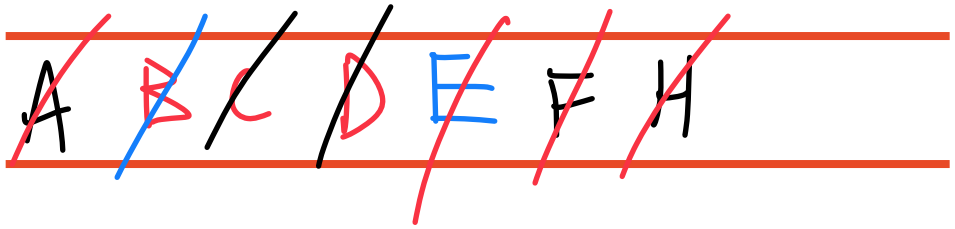
0) Initialization
- ↳ A queue (tracks things to be visited)
- labels
  - ↳ distance from start vertex
  - ↳ Predecessor: The node I left to get to me
  - for each U, store U1

1) while queue not empty:

tmp = q.dequeue()

process all children
- ↳ label dist, prev
- ↳ Add unvisited children to q

v	d	P	Adjacent Edges
A	0	—	B C D
B	1	A	~~A~~ ~~C~~ E ← C is a cross edge
C	1	A	~~A B D E~~ F
D	1	A	~~A C F~~ H
E	2	B	~~B C~~ G
F	2	C	C D G
G	3	E	E F H
H	2	D	D G



*Track 'visited' by label { unlabeled not visited
labeled has been or will be visited soon

Traversal: BFS

$|V| = n, |E| = m$

Initialize queue / dist / predecessor

While queue not empty:

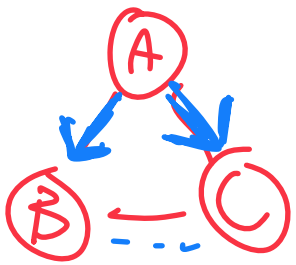
- Remove front vertex of queue
- Check if edge connects to new vertex
  - Set dist / pred if new vertex
- Add unvisited edges to queue

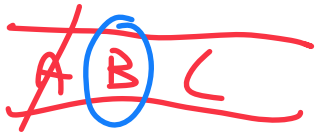
v	d	P	Adjacent Edges
A	0	-	B C D
B	1	A	A C E
C	1	A	A B D E F
D	1	A	A C F H
E	2	B	B C G
F	2	C	C D G
G	3	E	E F H
H	2	D	D G

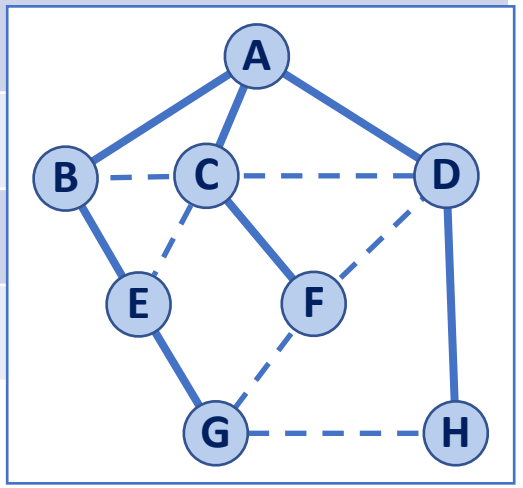
Join Code: 225

label
cross

get Adjacent (B)
↳ A, C
X X







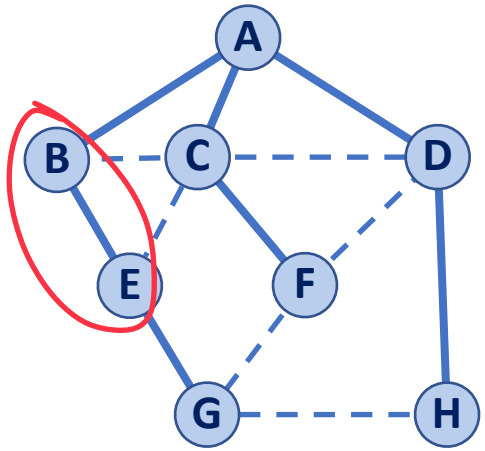
A B C D E F H G

Running time? $O(n + m)$

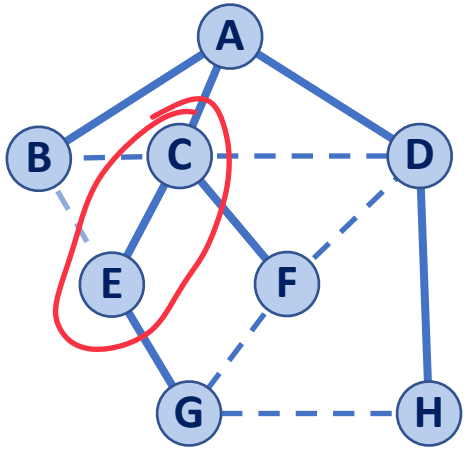
Traversal: BFS

Your graph implementation matters

Traversal depends on start position as well as order of edges at each node



v	d	P	Adjacent Edges
A	0	-	B C D
B	1	A	A C E
C	1	A	A B D E F



v	d	P	Adjacent Edges
A	0	-	C B D
B	1	A	A C E
C	1	A	A B D E F

Input: Graph, G

Output: A labeling of the edges in G as discovery or cross

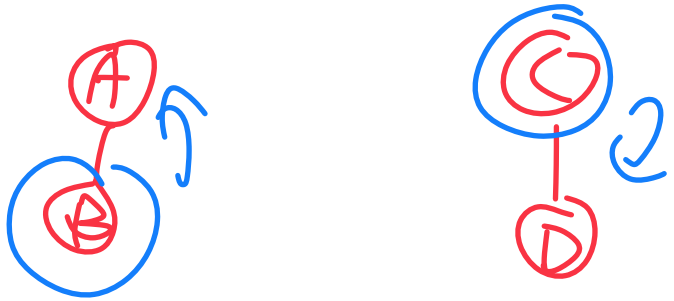
```
 1  BFS(G):
 2    foreach (Vertex v : G.vertices()):
 3      setPred(v, NULL)
 4      setDist(v, -1)
 5
 6    foreach (Edge e : G.edges()):
 7      setLabel(e, UNEXPLORED)
 8
 9    foreach (Vertex v : G.vertices()):
10      if getDist(v) == -1:
11         BFS(G, v)
```

Initialize vertices O(n)

Initialize edges O(m)

??

Multiple connected components



```
1  BFS(G):
2    foreach (Vertex v : G.vertices()):
3      setPred(v, NULL)
4      setDist(v, -1)
5
6    foreach (Edge e : G.edges()):
7      setLabel(e, UNEXPLORED)
8
9    foreach (Vertex v : G.vertices()):
10     if getDist(v) == -1:
11       BFS(G, v)
```

$$\sum_{v}^{G} \deg(v) = 2|E|$$

↳ Every edge seen twice!

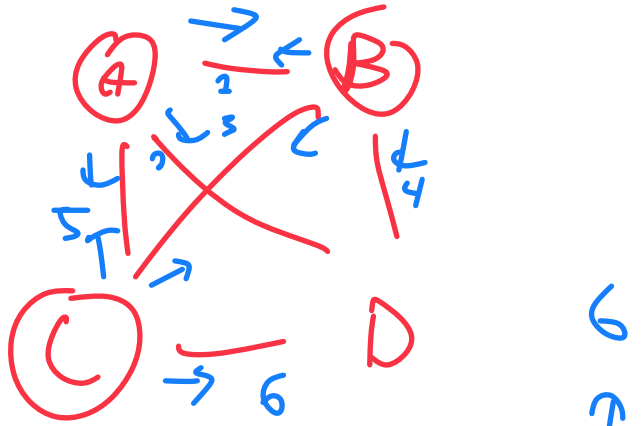
$O(n+m)$

```
12 BFS(G, v):
13   Queue q
14   setDist(v, 0)
15   q.enqueue(v)
16
17   while !q.empty():
18     v = q.dequeue()
19
20     foreach (Vertex w : G.adjacent(v)):
21       if( getDist(w) == -1):
22         setLabel((v, w), DISCOVERY)
23         setPred(w, v)
24         setDist(w, v + 1)
25         q.enqueue(w)
26       else:
27         setLabel((v, w), CROSS)
```

initialize

← This will happen once per vertex

every edge in every vertex



6

$3+3+3+3=12$

```
1  BFS(G):
2    foreach (Vertex v : G.vertices()):
3      setPred(v, NULL)
4      setDist(v, -1)
5
6    foreach (Edge e : G.edges()):
7      setLabel(e, UNEXPLORED)
8
9    foreach (Vertex v : G.vertices()):
10     if getDist(v) == -1:
11       BFS(G, v)
```

Count pt

Count connected components?

↳ Yes but... we need one modification

Add count to line 11

Cycle Detection?

↳ Yes

A single cross edge says we have at least one cycle

```
12 BFS(G, v):
13   Queue q
14   setDist(v, 0)
15   q.enqueue(v)
16
17   while !q.empty():
18     v = q.dequeue()
19
20     foreach (Vertex w : G.adjacent(v)):
21       if( getDist(w) == -1):
22         setLabel((v, w), DISCOVERY)
23         setPred(w, v)
24         setDist(w, v + 1)
25         q.enqueue(w)
26       else:
27         setLabel((v, w), CROSS)
```

Cross if already visited!

BFS Observations

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

same dest

↳ path length 2

A → D → H

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

↳ we don't know!

v	d	P	Adjacent Edges
A	0	-	C B D
B	1	A	A C E
C	1	A	B A D E F
D	1	A	A C F H
E	2	C	B C G
F	2	C	C D G
G	3	E	E F H
H	2	D	D G

BFS Observations

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

A-D-H (see by walking backwards!)

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

Can't tell by BFS!

↳ why?

v	d	P	Adjacent Edges
A	0	-	**C B D**
B	1	**A**	**A C E**
C	1	**A**	**B A D E F**
D	1	**A**	**A C F H**
E	2	**C**	**B C G**
F	2	**C**	**C D G**
G	3	**E**	**E F H**
H	2	**D**	**D G**

↑ All in relation to start vertex!

BFS Observations

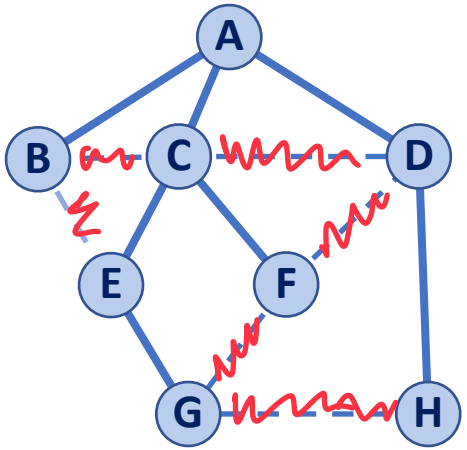
If my node has distance **d**, do I know anything about the nodes connected by a **cross edge**?

↳ A cycle

What structure is made from **discovery edges**?

↳ Spanning tree!

v	d	P	Adjacent Edges
A	0	-	C B D
B	1	A	A C E
C	1	A	B A D E F
D	1	A	A C F H
E	2	C	B C G
F	2	C	C D G
G	3	E	E F H
H	2	D	D G



BFS Observations

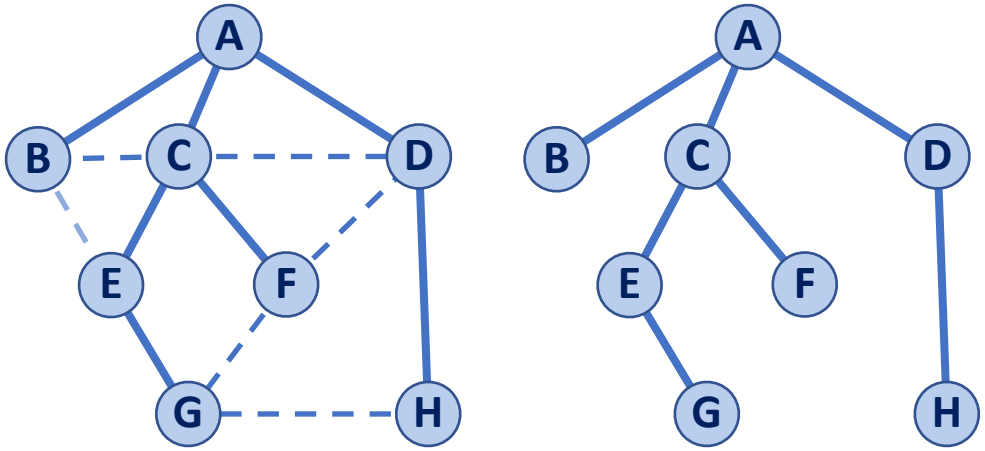
If my node has distance **d**, do I know anything about the nodes connected by a **cross edge**?

A cross edge tells us we have at least one cycle

What structure is made from **discovery edges**?

A spanning tree

v	d	P	Adjacent Edges
A	0	-	C B D
B	1	A	A C E
C	1	A	B A D E F
D	1	A	A C F H
E	2	C	B C G
F	2	C	C D G
G	3	E	E F H
H	2	D	D G



BFS Observations

1. BFS can be used to count components
2. BFS can be used to detect cycles

3. The BFS 'distance' value is always the shortest distance from source to any vertex (and the discovery edges form a spanning tree)

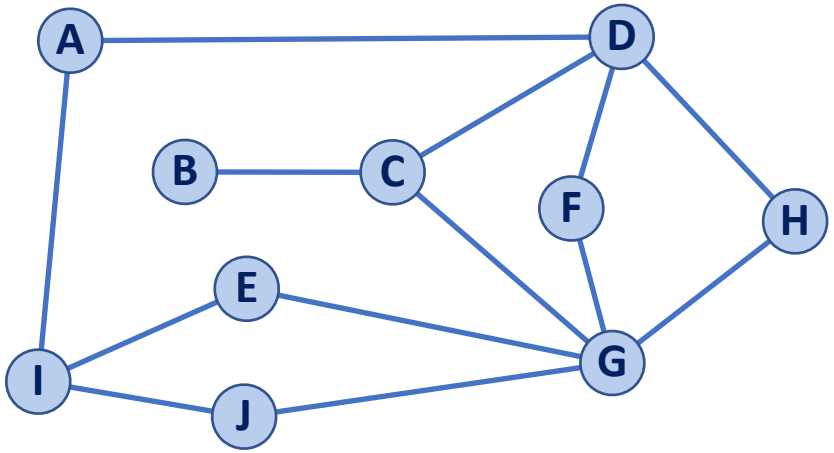
↳ shortest path!

4. The endpoints of a cross edge never differ in distance by more than 1 ($|d(u) - d(v)| = 1$)

↳ Trivia point but think about why!

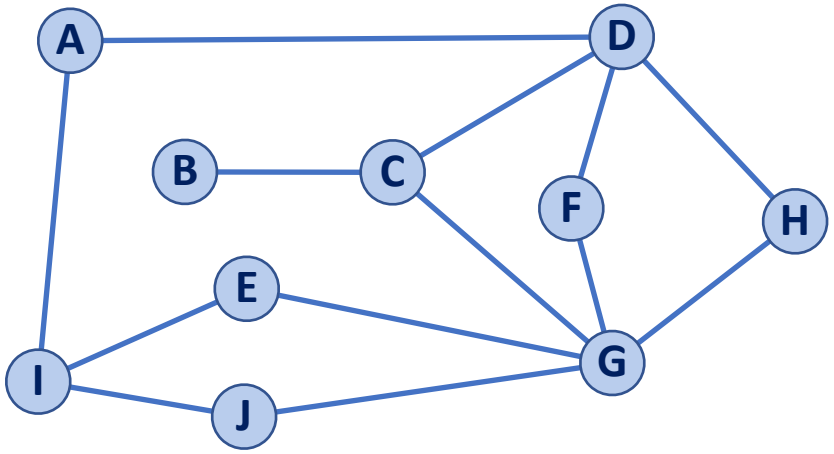
Proof by contradiction

Traversal: DFS



The start to BFS's queue

Traversal: DFS



Initialize dist / pred / stack

All dist null (start node dist 0)

All pred -1 (start node pred -1)

Stack loaded with start node

While stack not empty

tmp=stack.peek()

Process one unvisited child

Add to stack

dist = tmp.dist+1

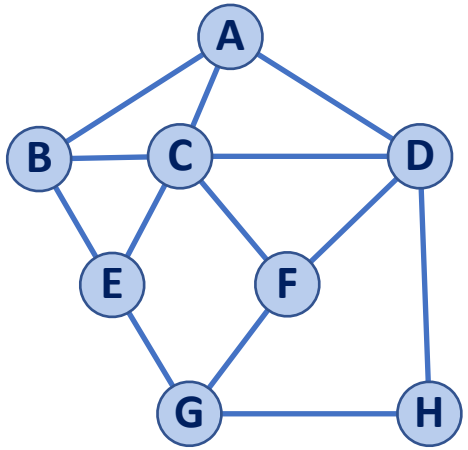
pred = tmp

If no unvisited children

stack.pop()

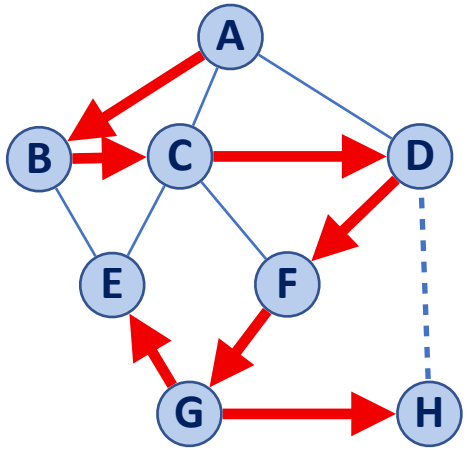
Stack

```
1  DFS(G):
2    foreach (Vertex v : G.vertices()):
3      setPred(v, NULL)
4      setDist(v, -1)
5
6    foreach (Edge e : G.edges()):
7      setLabel(e, UNEXPLORED)
8
9    foreach (Vertex v : G.vertices()):
10     if getDist(v) == -1:
11       DFS(G, v)
```




```
12 DFS(G, v):
13
14     foreach (Vertex w : G.adjacent(v)):
15         if( getDist(w) == -1):
16             setLabel((v, w), DISCOVERY)
17             setPred(w, v)
18             setDist(w, v + 1)
19             DFS(G, w)
20         else:
21             setLabel((v, w), BACK)
```

```
12 DFS(G, v):
13
14     foreach (Vertex w : G.adjacent(v)):
15         if( getDist(w) == -1):
16             setLabel((v, w), DISCOVERY)
17             setPred(w, v)
18             setDist(w, v + 1)
19             DFS(G, w)
20         else:
21             setLabel((v, w), BACK)
```



$|V| = n, |E| = m$

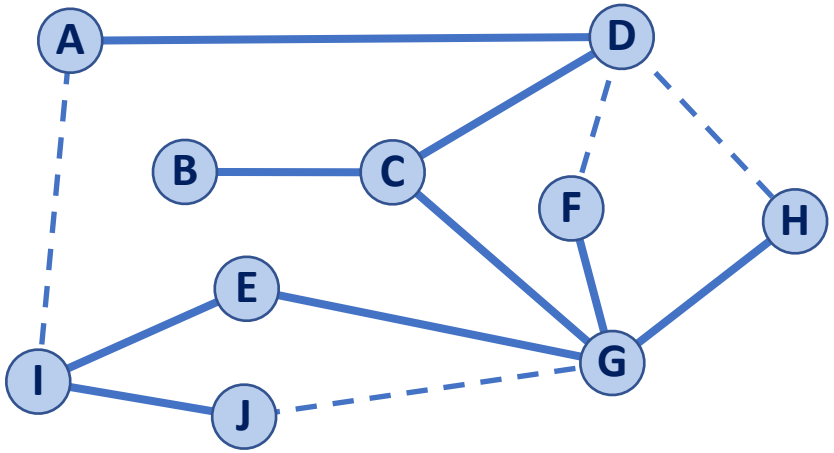
Running time?

v	d	P	Adjacent Edges
A	0	-	B C D
B	1	A	A C E
C	2	B	A B D E F
D	3	C	A C F H
E	6	G	B C G
F	4	D	C D G
G	5	F	E F H
H	6	G	D G

State of stack when H is added:

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

Traversal: DFS



Discovery Edge

Back Edge

Do we still make a spanning tree?

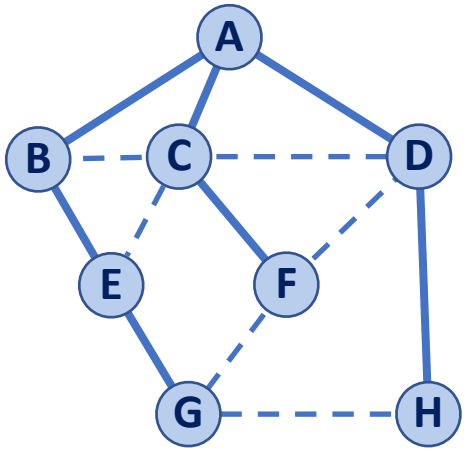
Does distance have meaning here?

Do our edge labels have meaning here?

Efficiency: DFS vs BFS

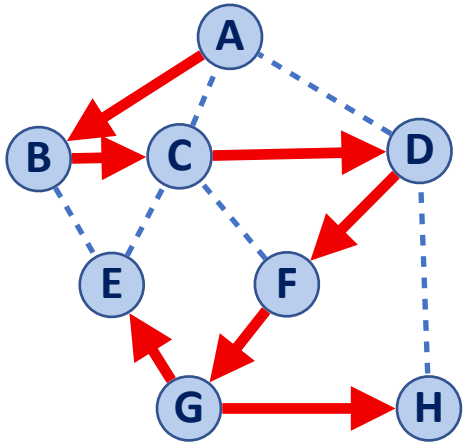
$|V| = n, |E| = m$

BFS:



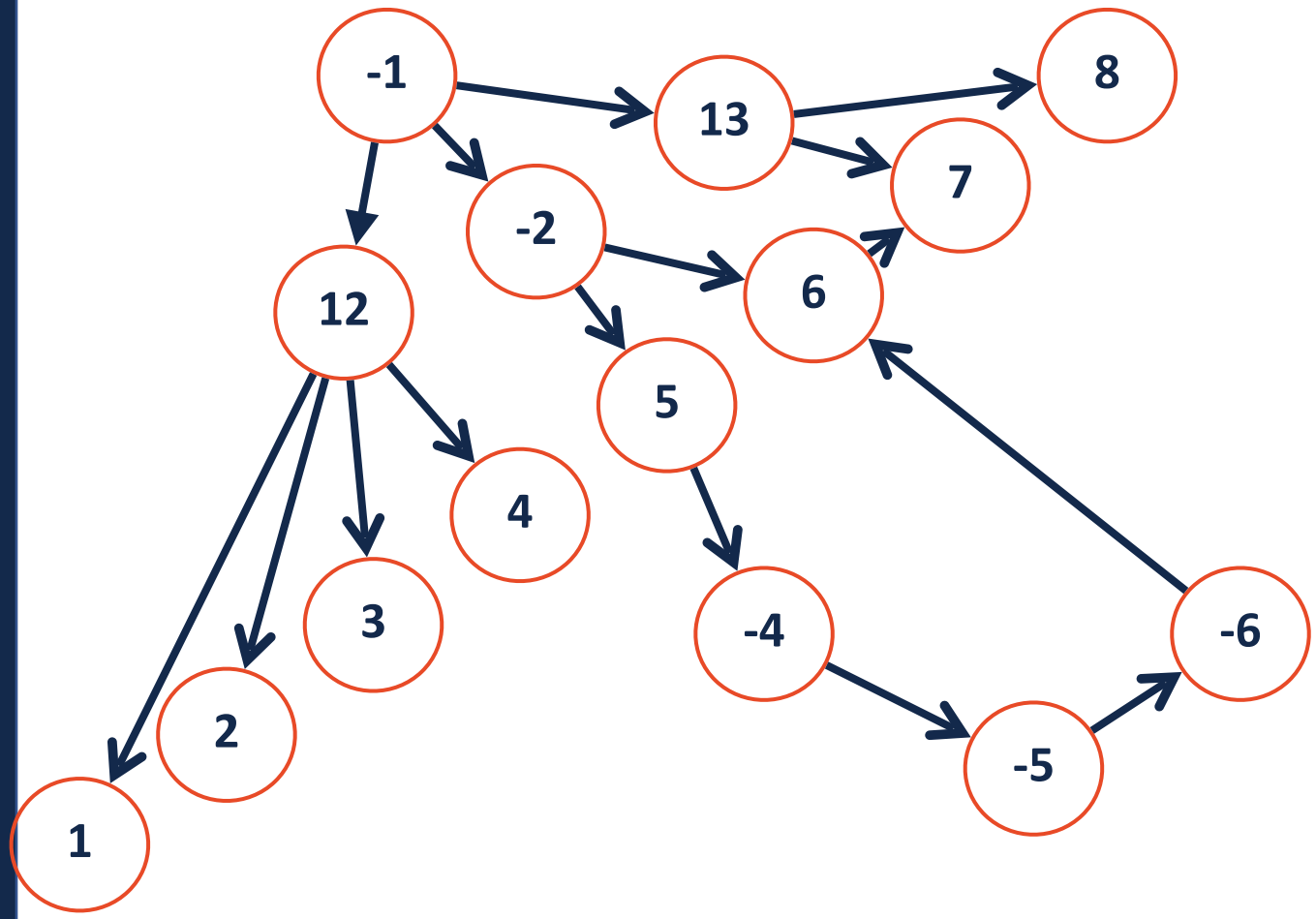
A B C D E F H G

DFS:



A B C D F G E H

Space Efficiency: DFS vs BFS

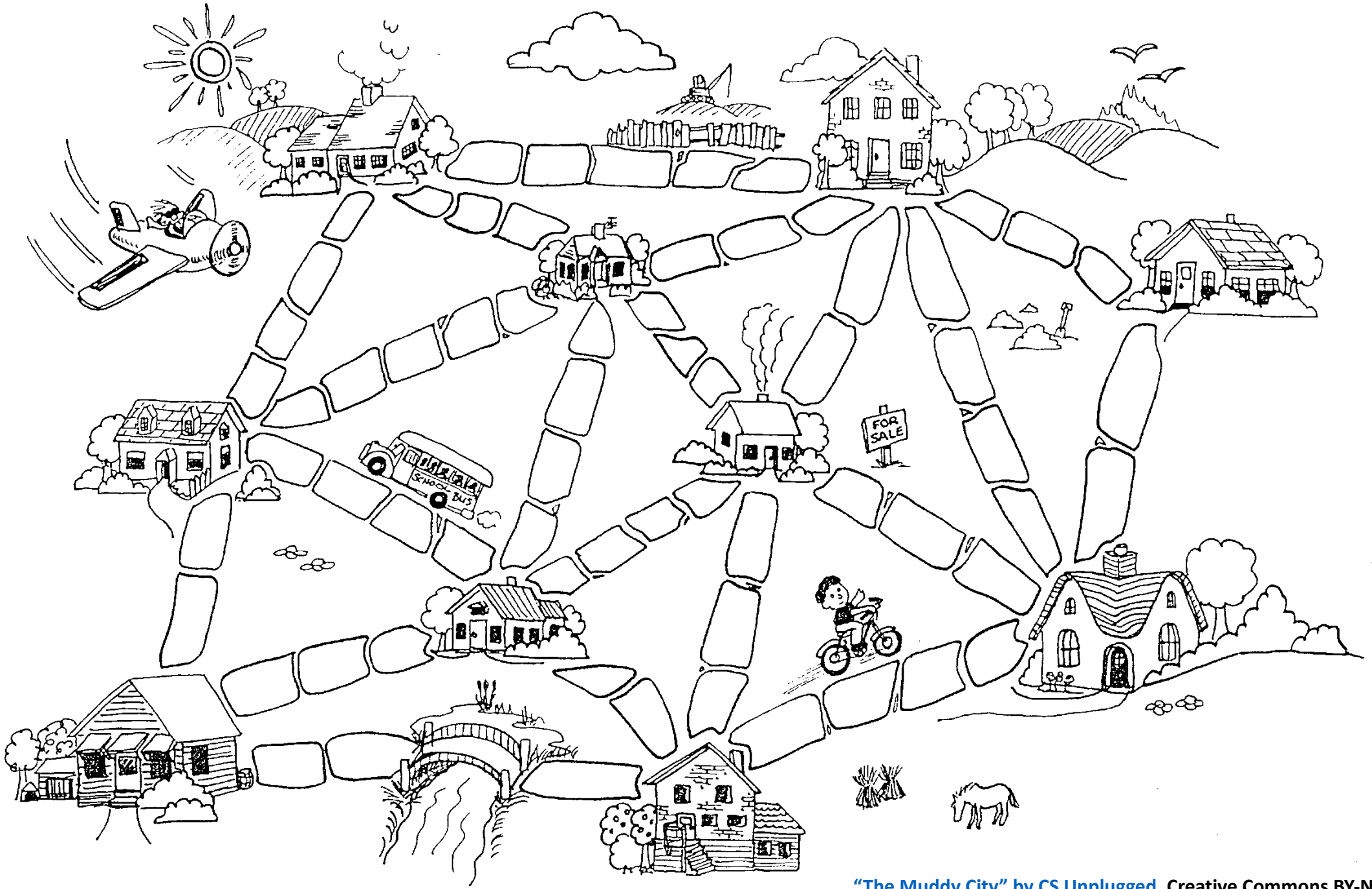


Summary: DFS and BFS

$|V| = n, |E| = m$

Both are **O(n+m)** traversals! They label every edge and every node

BFS	**DFS**
Solves unweighted MST	Solves unweighted MST
Solves shortest path	
Solves cycle detection	Solves cycle detection
Memory bounded by width	Memory bounded by longest path



Minimum Spanning Tree Algorithms

Input: Connected, undirected graph **G** with edge weights (unconstrained, but must be additive)

Output: A graph G' with the following properties:

- G' is a spanning graph of G
- G' is a tree (connected, acyclic)
- G' has a minimal total weight among all spanning trees