Overflow from Wednesday

Mostly BFS

turns out we didn't get here!

Data Structures

Graph Implementation 3 & Traversals

October 30, 2026

CS 225
Brad Solomon

Happy Halloween! :)
(roughly)



UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

Learning Objectives

Review adjacency list graph implementation

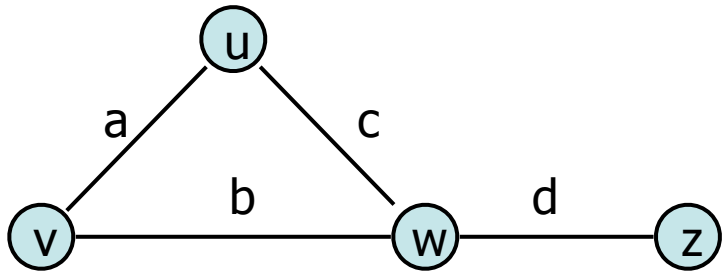
Discuss tradeoffs between all graph implementations

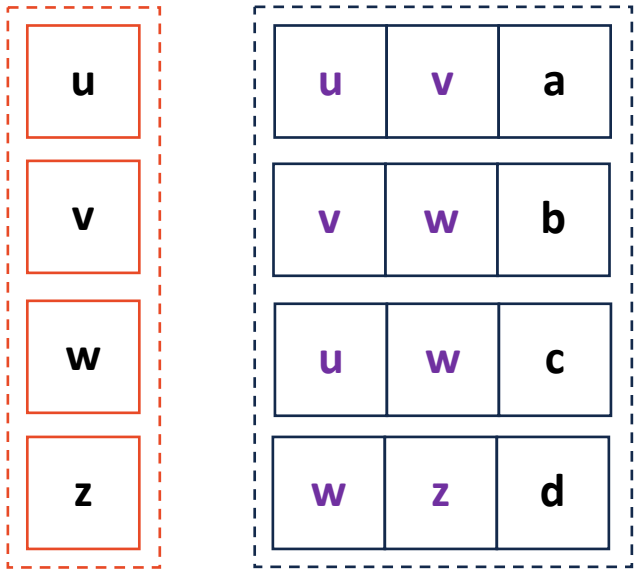
Introduce graph traversal algorithms

Graph Implementation: Edge List

`|V|= n, |E|= m`

The equivalent of an 'unordered' data structure





Vertex Storage:

An optional list of vertices

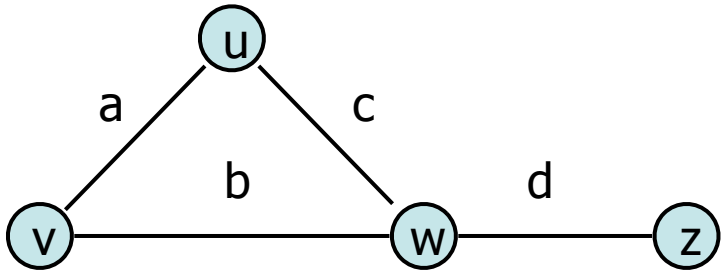
Edge Storage:

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



Vertex Storage:

A hash table of vertices

Implicitly or explicitly store index

u	0
v	1
w	2
z	3

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

Edge Storage:

A |V| x |V| matrix of edges

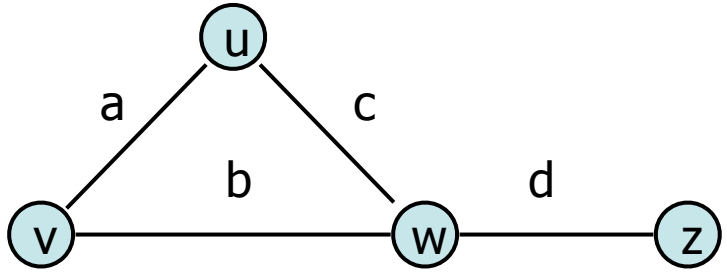
Weight is stored at position (u, v)

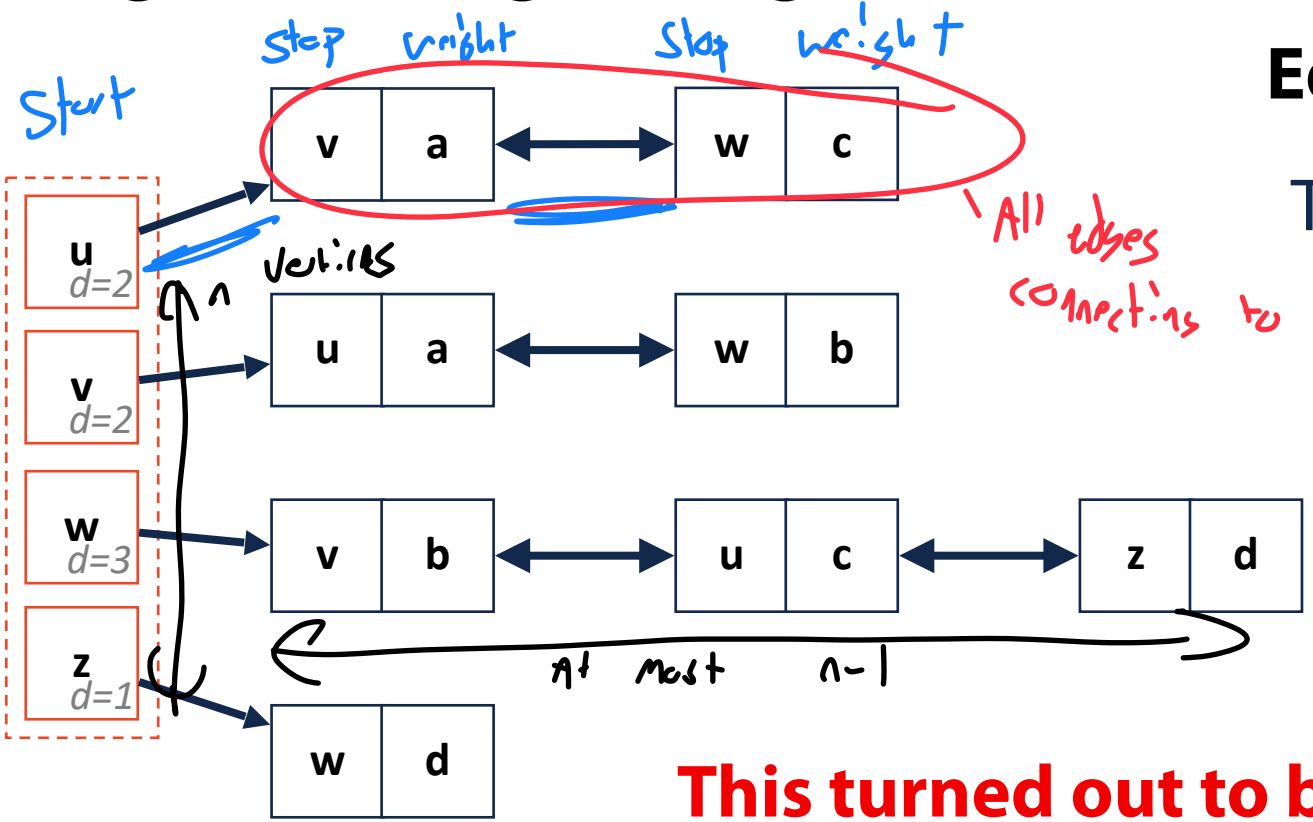
possible

↳ All edges (even empty spots) are stored *

Naive Adjacency List

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





Vertex Storage:

A size n array of pointers to linked lists

The array value is the start vertex

The first value in a node is the end vertex

Edge Storage:

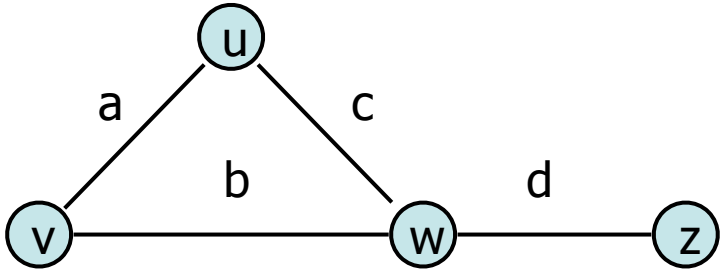
The second value in the linked list

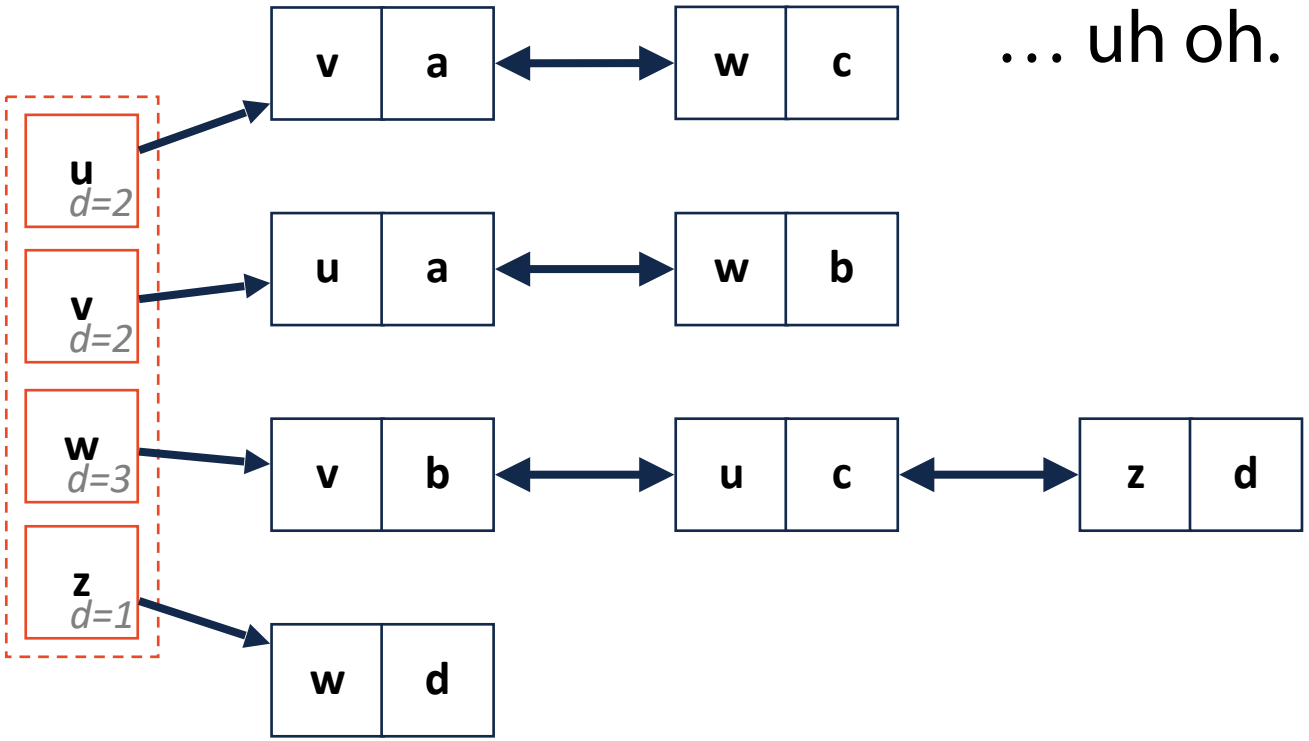
Remove Vertex → $O(n^2)$

This turned out to be a bad implementation — why?

Naive Adjacency List

|V|= n, |E|= m





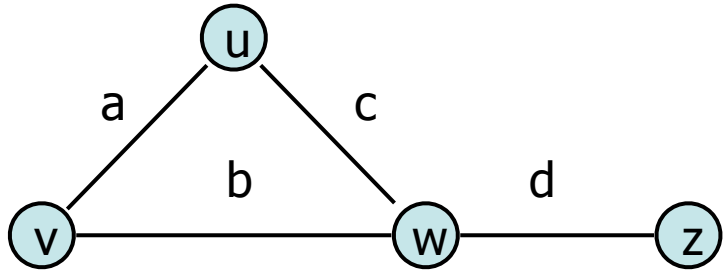
removeVertex(Vertex v):

Look up V in our list and remove all entries.

… And then look for V in every other list

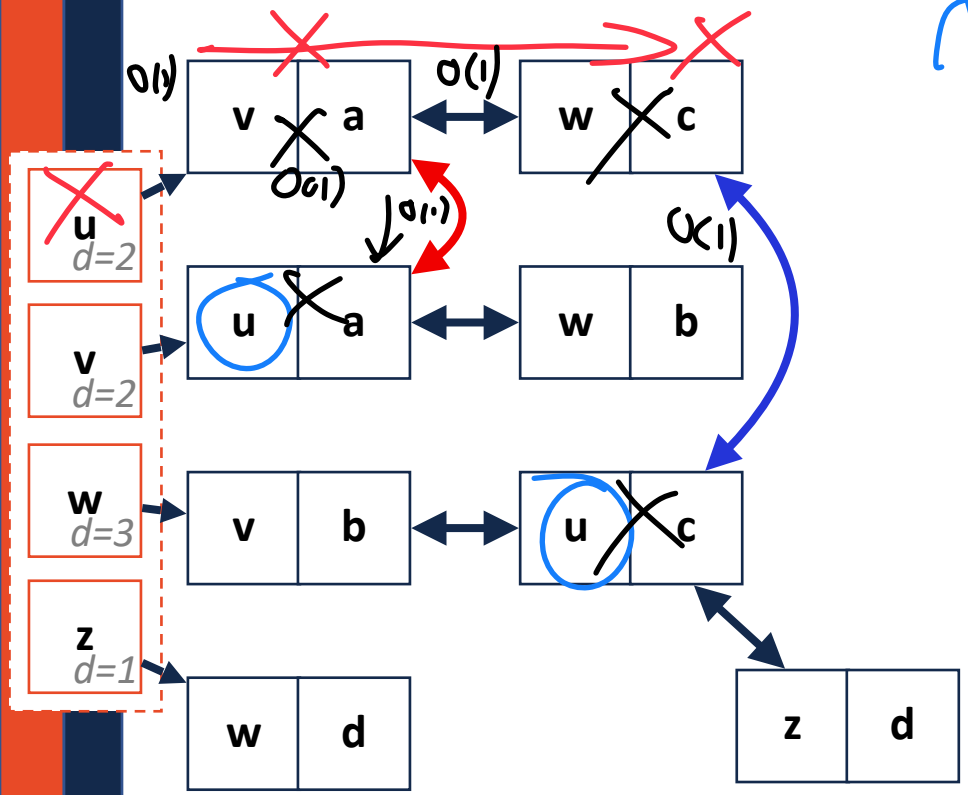
… uh oh.

Adjacency List



|V| = n, |E| = m

Add more list to our list!



Problem: Edges can exist twice in adj list

Soln: Add a bidirectional connection from

u v a to v u a

(All paired edges)

if sparse prob not O(n)

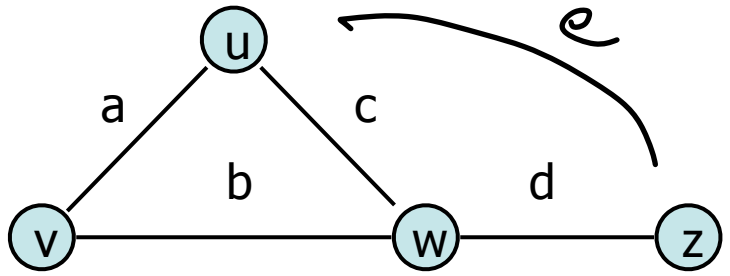
max size

O[deg(u)] → O(n)

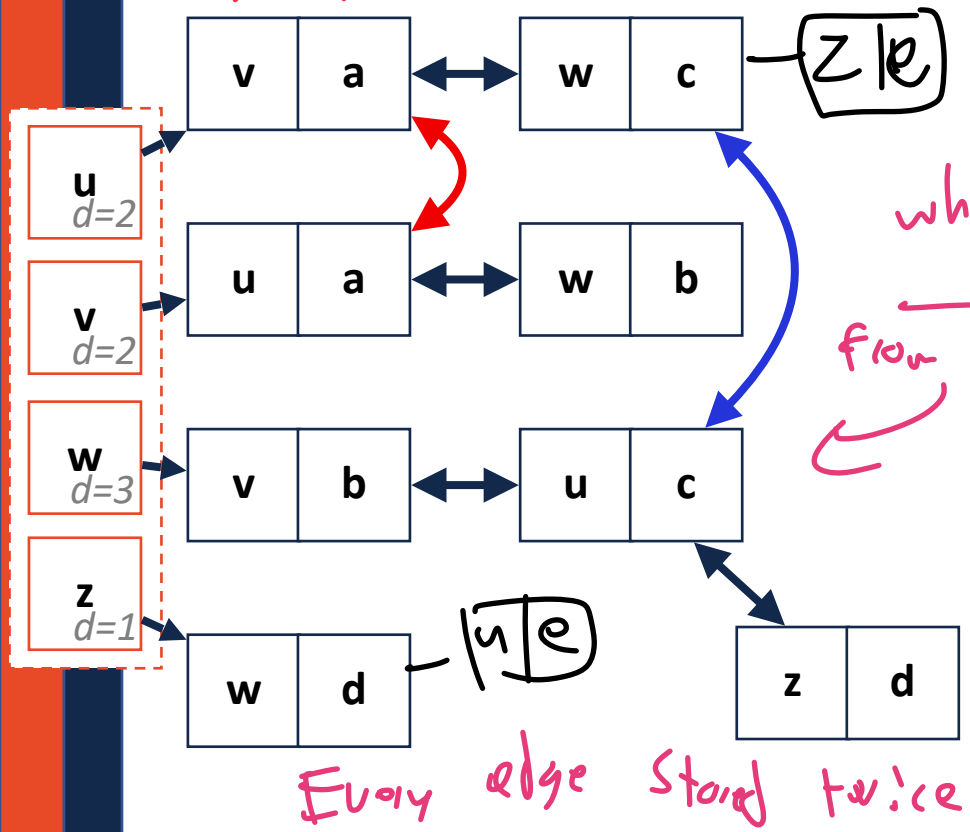
if dense

Adjacency List

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

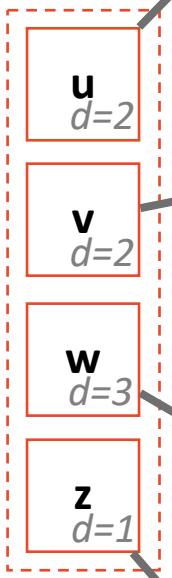


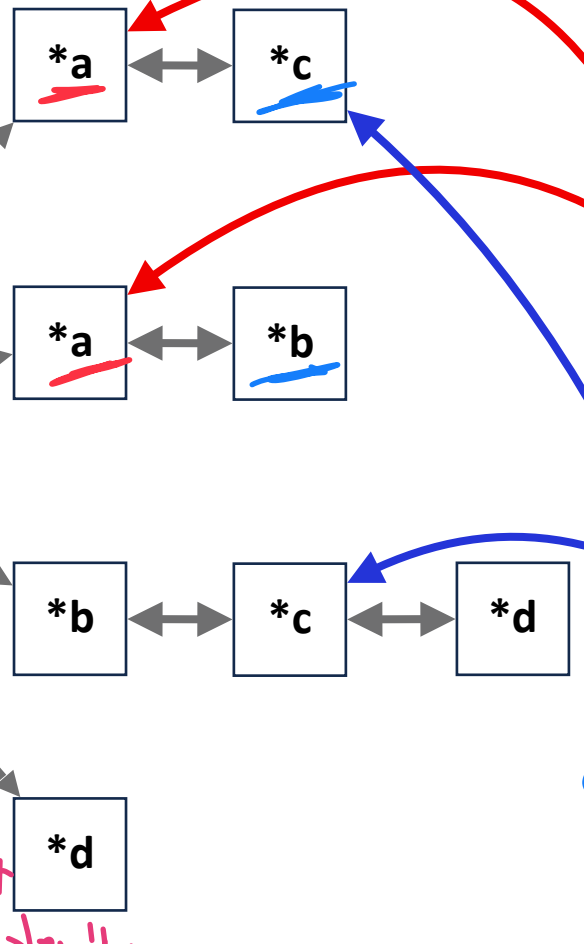
LL of
↳ stop
↳ weight



Every edge stored twice

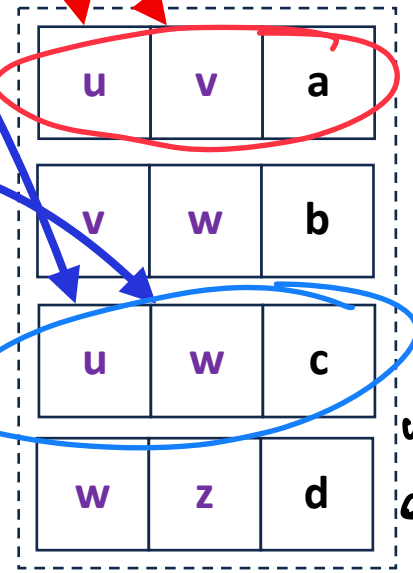
Map
start → LL





Store linked list of pointers

Keep edge list



Every edge stored once

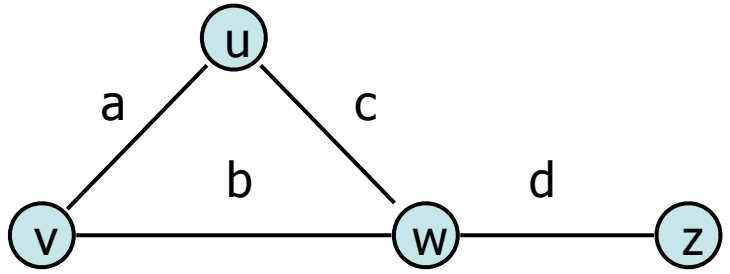
why jump from → to

Reduce storage cost

Engineering complexity efficiency

Adjacency List

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



Bidirectional

Linked list

Adj List Node:

*Prev	*Edge	*Next

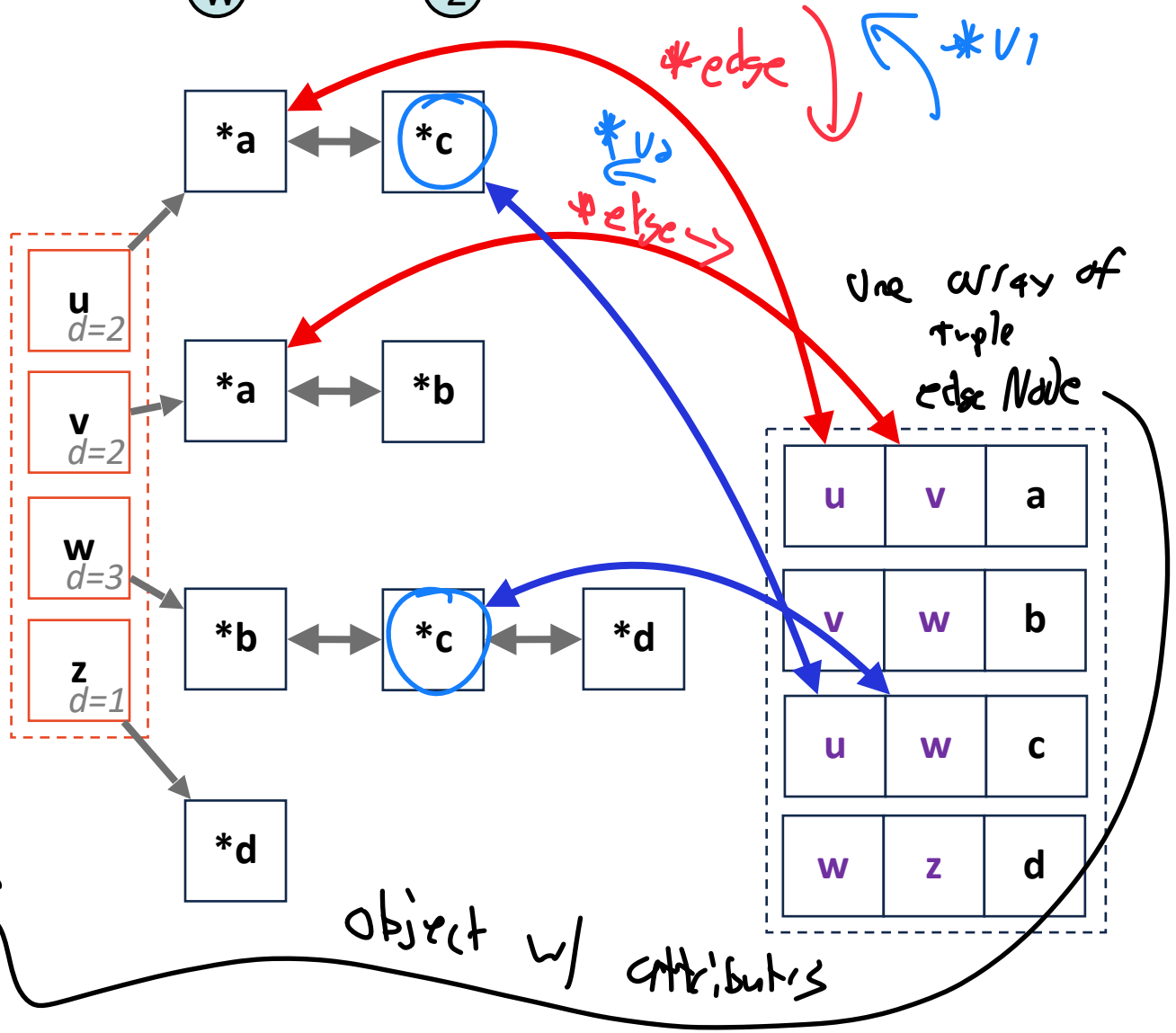
*Edge List:

V1	V2	Weight
*V1	*V2	

This Node

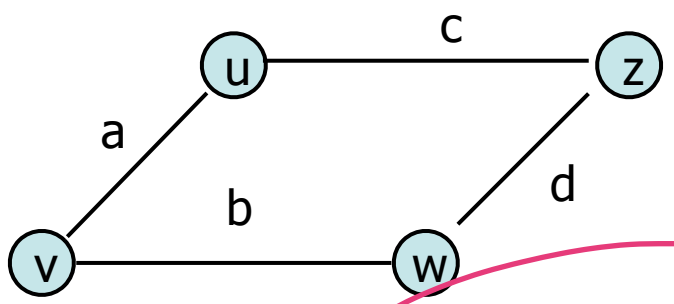
An edge list Node

Two new Pointers



Adjacency List

|V|= n, |E|= m



Adj List Node:

*Prev	*Edge	*Next

Edge List:

V1	V2	Weight
*V1	*V2	

Imp choice

visually only top 3

arrows are this!

removeVertex(z)

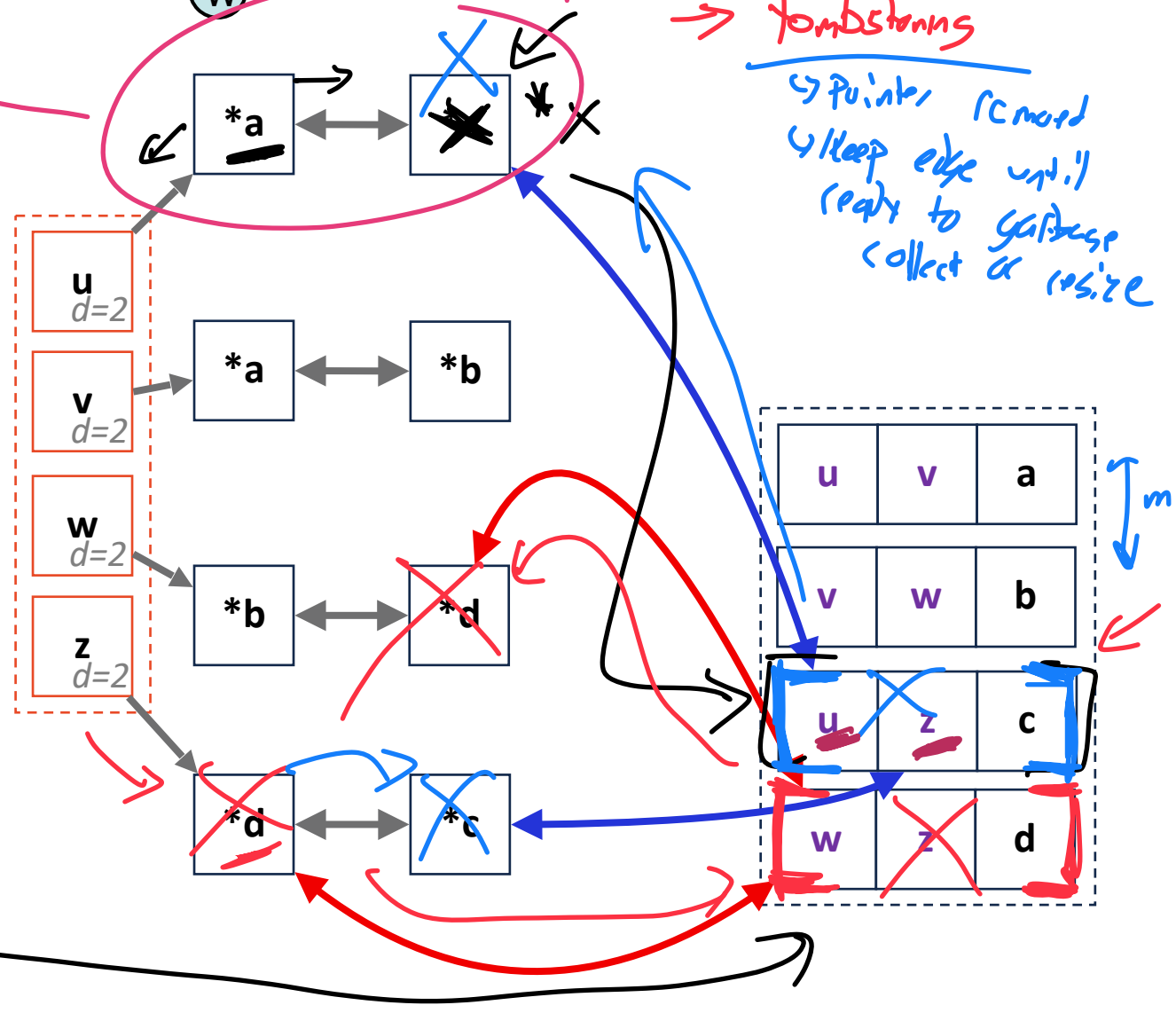
Problem: Removal from array O(m)

Soln: → Swap & remove

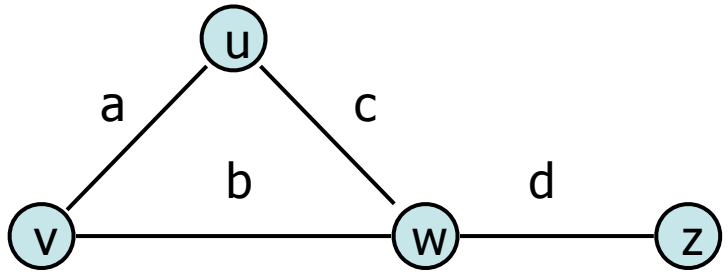
→ tombstoning

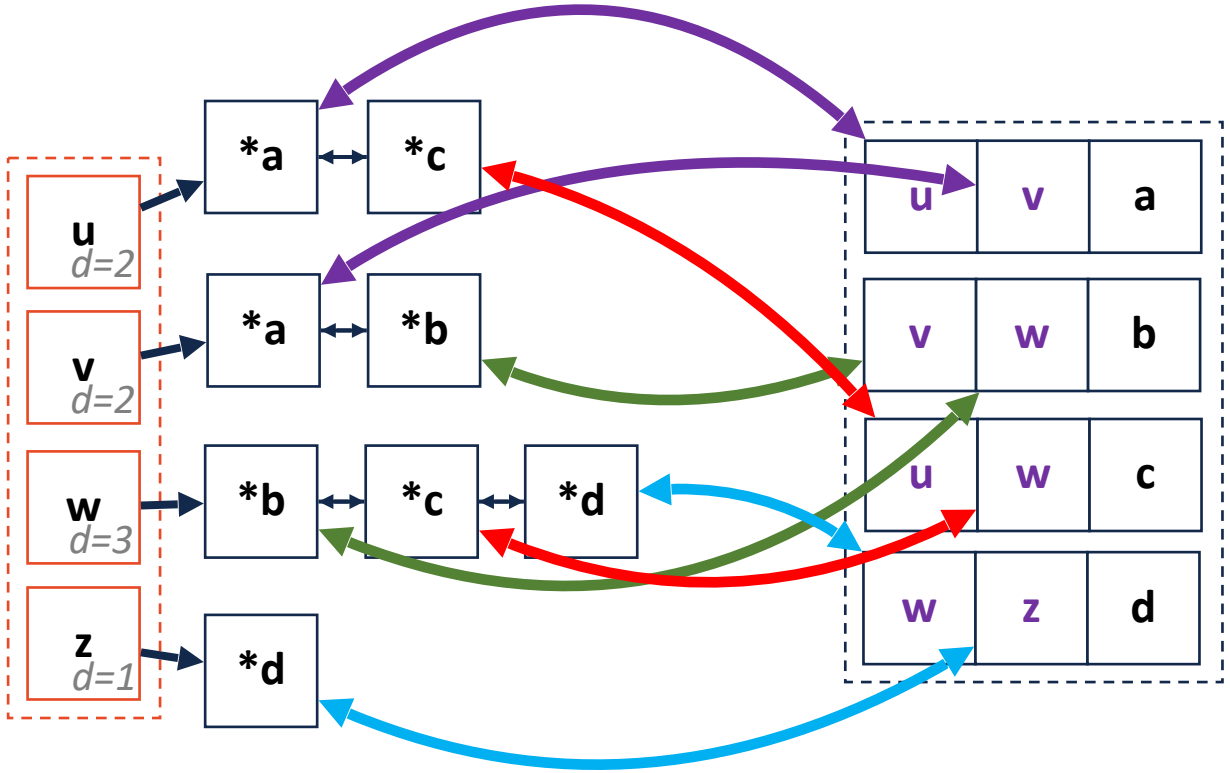
↳ Pointers removed

↳ keep edge until ready to garbage collect or resize



Adjacency List





Vertex Storage:

A map with start vertex key and LL value

Each linked list is bidirectional with size variable

Each node is a pointer to edge in edge list

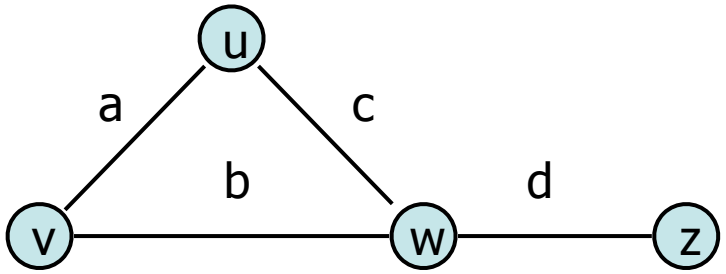
Edge Storage:

A list of (v1, v2, weight) edges

Also store pointers back to nodes

↑ 2 extra pointers per edge

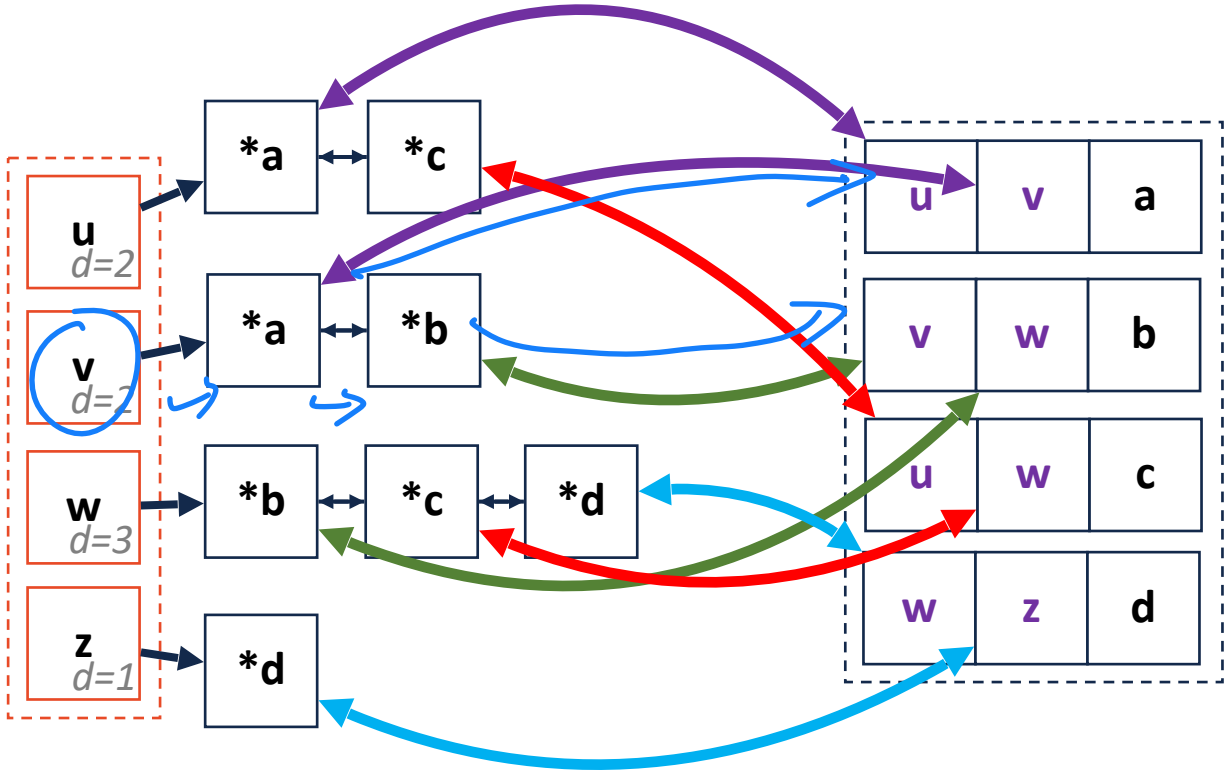
Adjacency List



getEdges(Vertex v):

Look up vertex list (and walk across it)

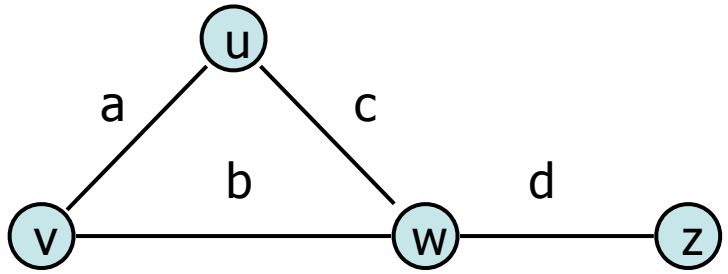
[Each edge is a pointer lookup away]



All individual steps are O(1)

$O[\deg(v)] \sim O(n)$

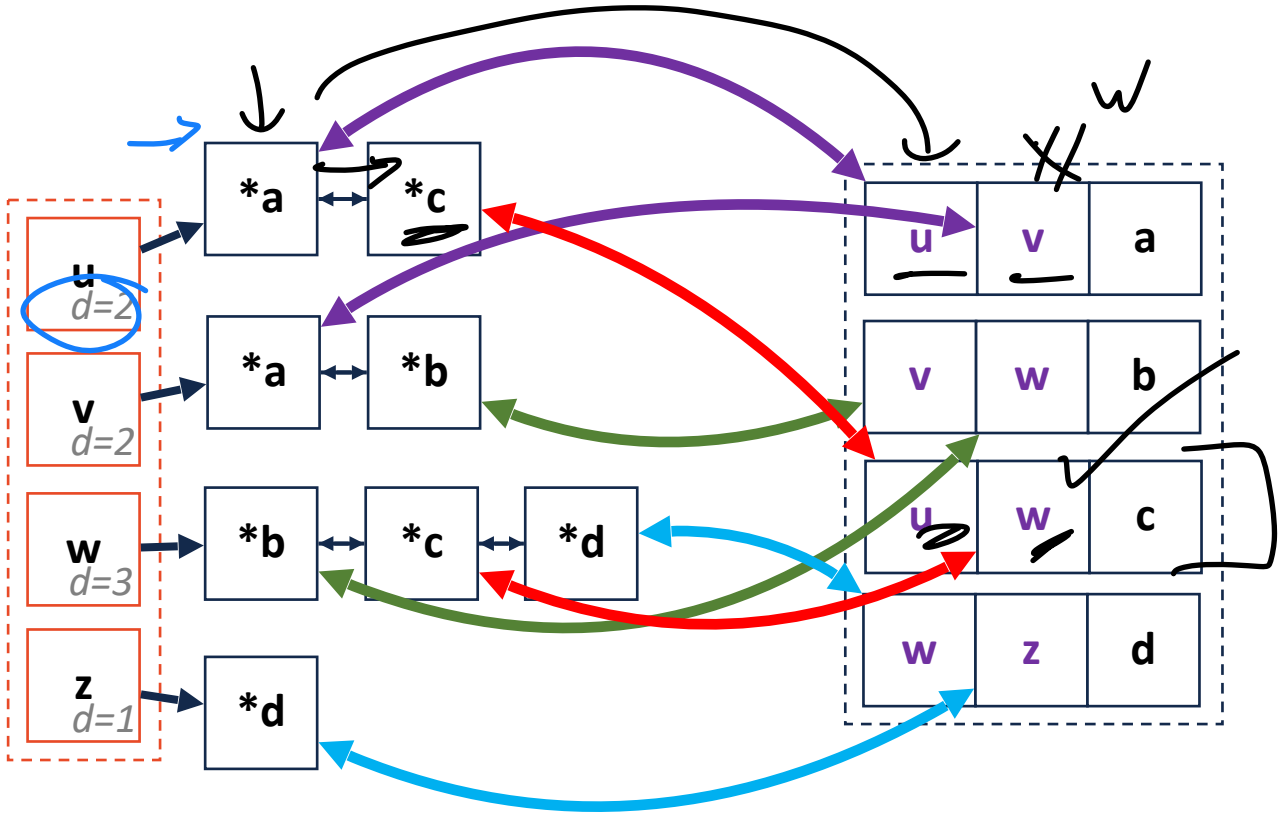
Adjacency List



areAdjacent(Vertex v1, Vertex v2):

Look up min-degree vertex list

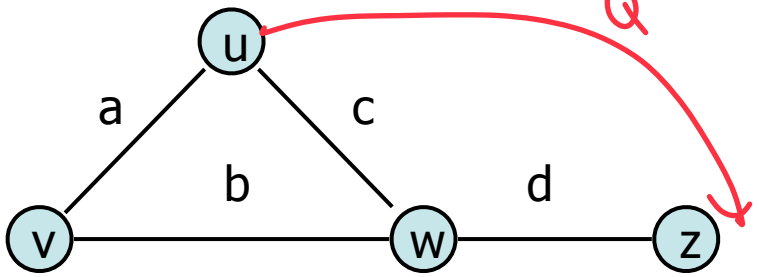
Search for other vertex across list

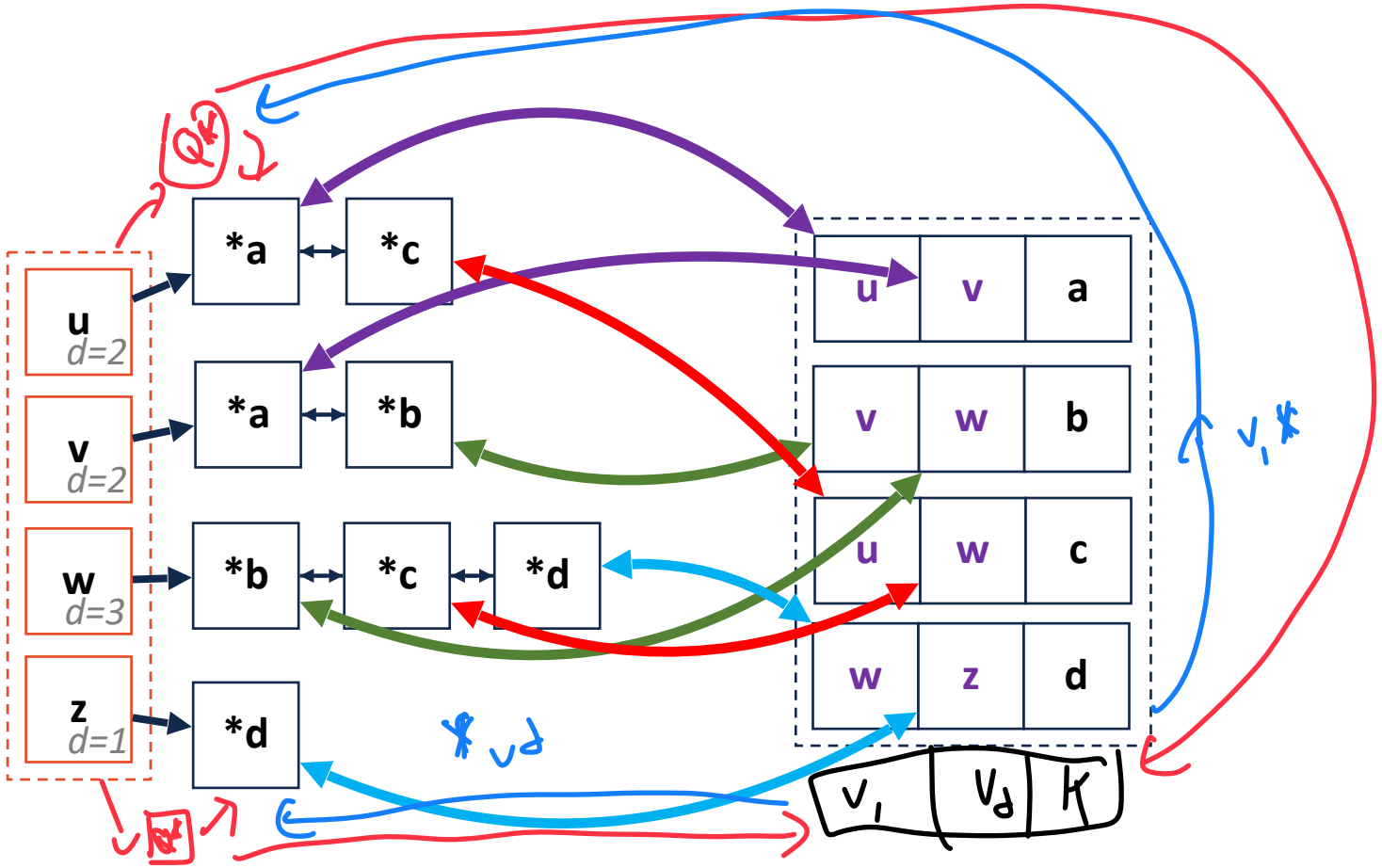


→ return True or C

O[min (deg(v1), deg(v2))]

Adjacency List





insertEdge(Vertex v1, Vertex v2, K key):

↳ Add edge to edge list O(1)*

↳ Add overhead indexing

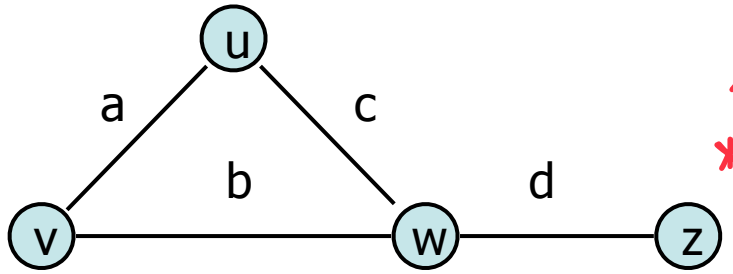
↳ Add new node to LL v1

LL v2

$v_1 = u$

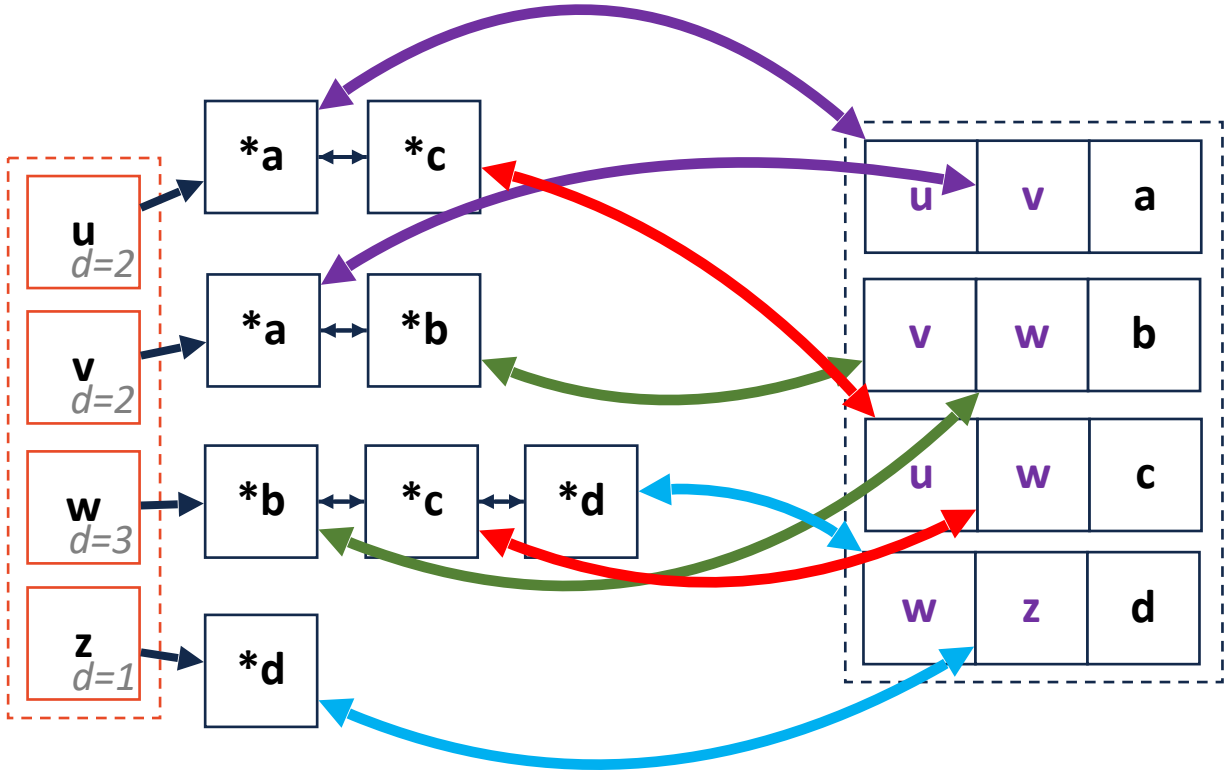
$v_2 = z$

Adjacency List



insertEdge(Vertex v1, Vertex v2, K key):

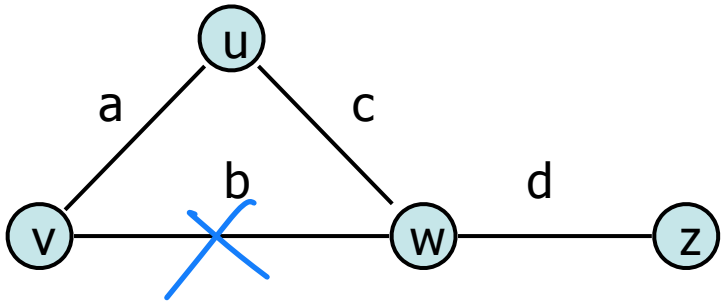
- Add edge to edge list
- Add node to each vertex list
- Connect all three with pointers

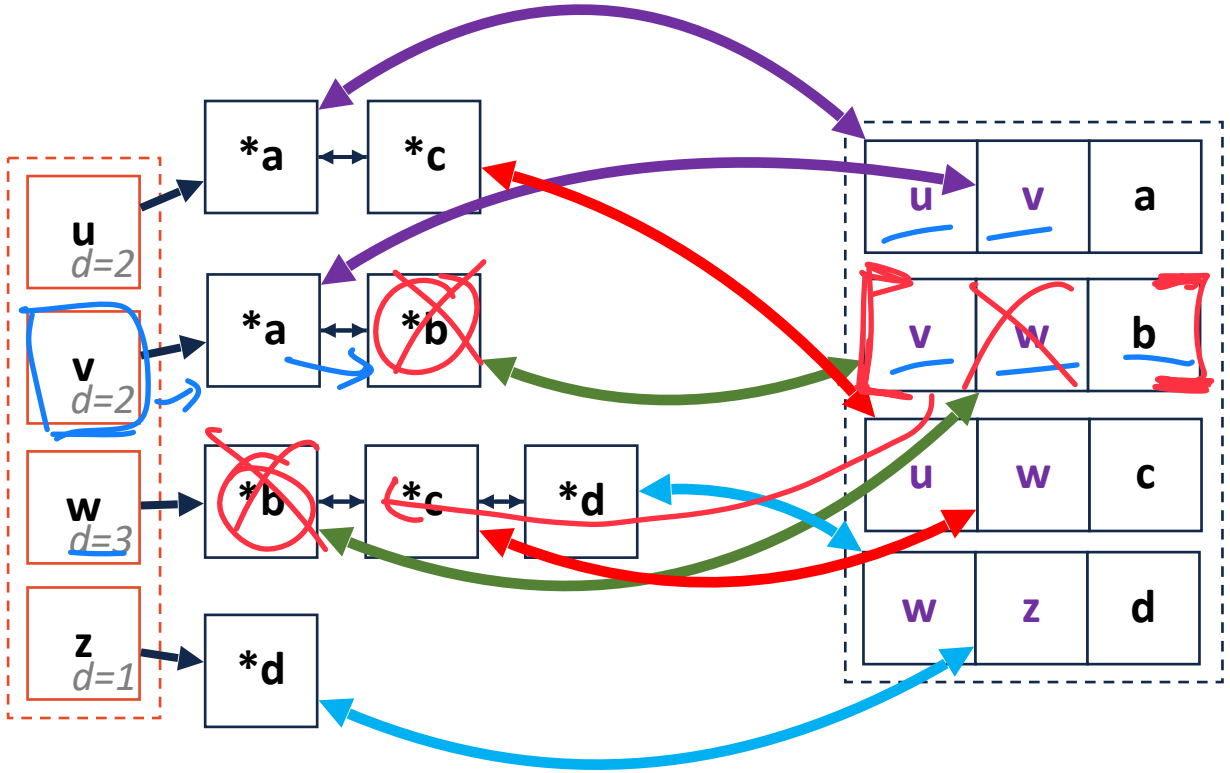


Join Code: 225

What are my Big Os?

Adjacency List

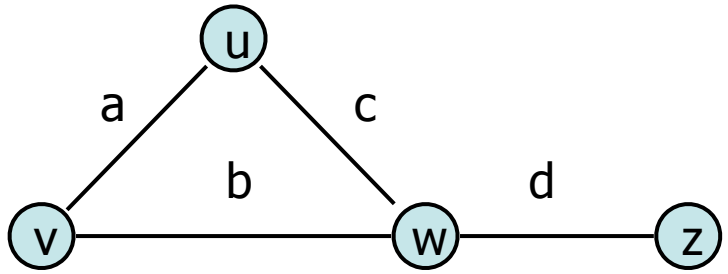




removeEdge(Vertex v1, Vertex v2):

find edge in min degree LL

Adjacency List



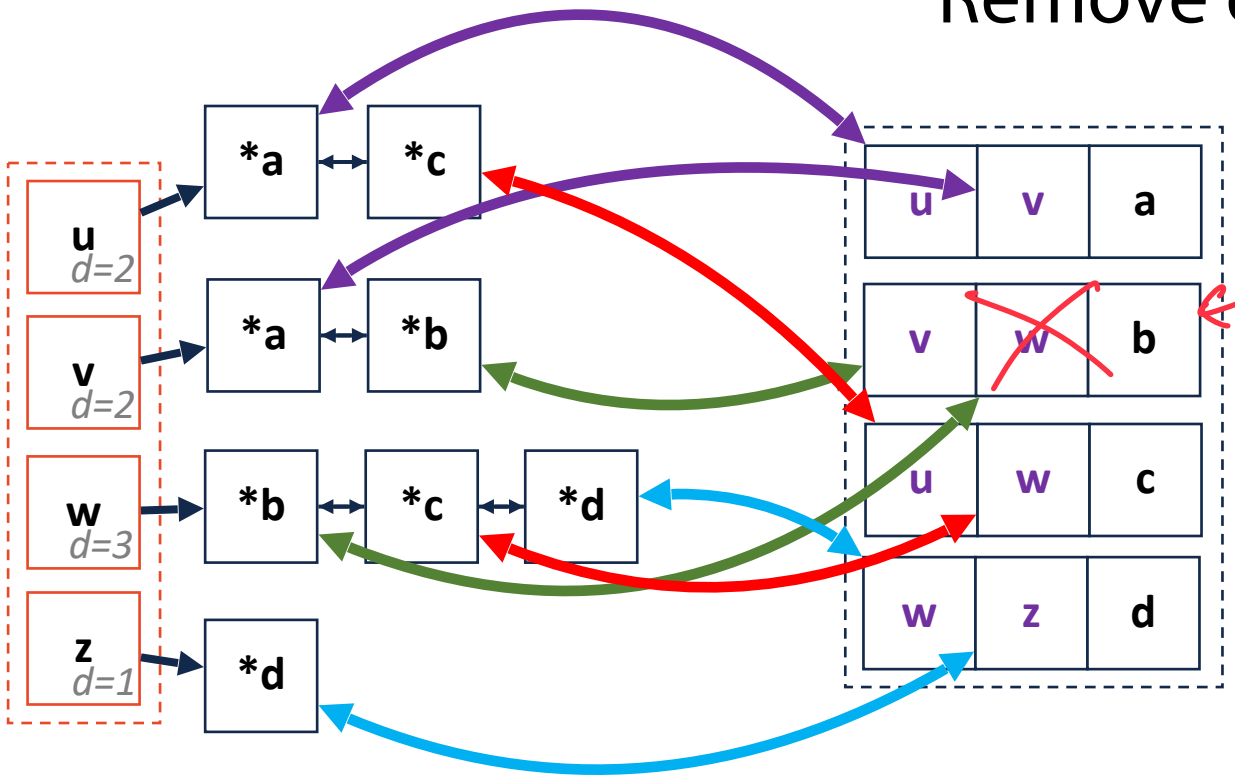
removeEdge(Vertex v1, Vertex v2):

Search min-degree vertex list $O(\deg(v))$

Remove mirrored entry using pointers $O(1)$

Remove edge from edge list $O(1)$

Remove entry from vertex list $O(1)$



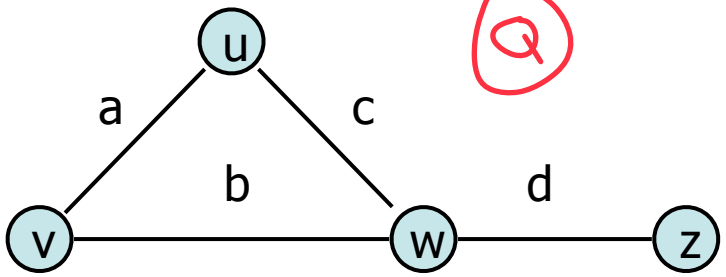
tombstoned away

$O[\deg(v)]$

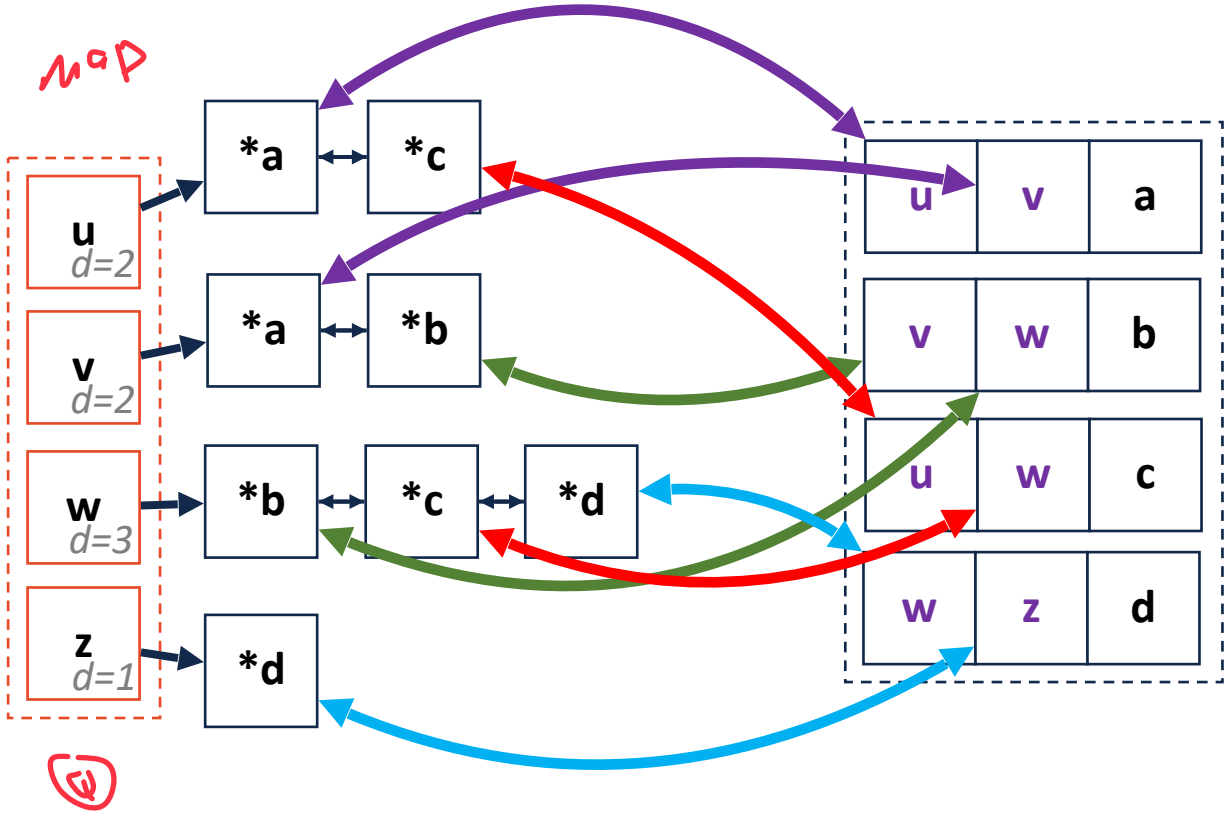
Join Code: 225

What are my Big Os?

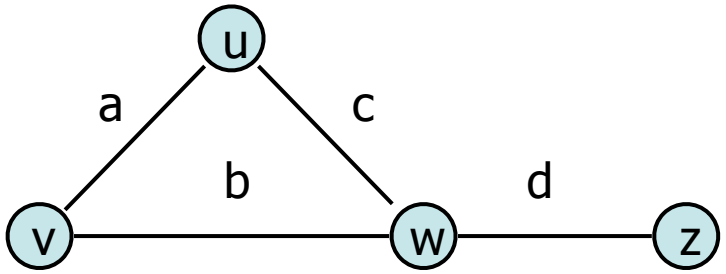
Adjacency List

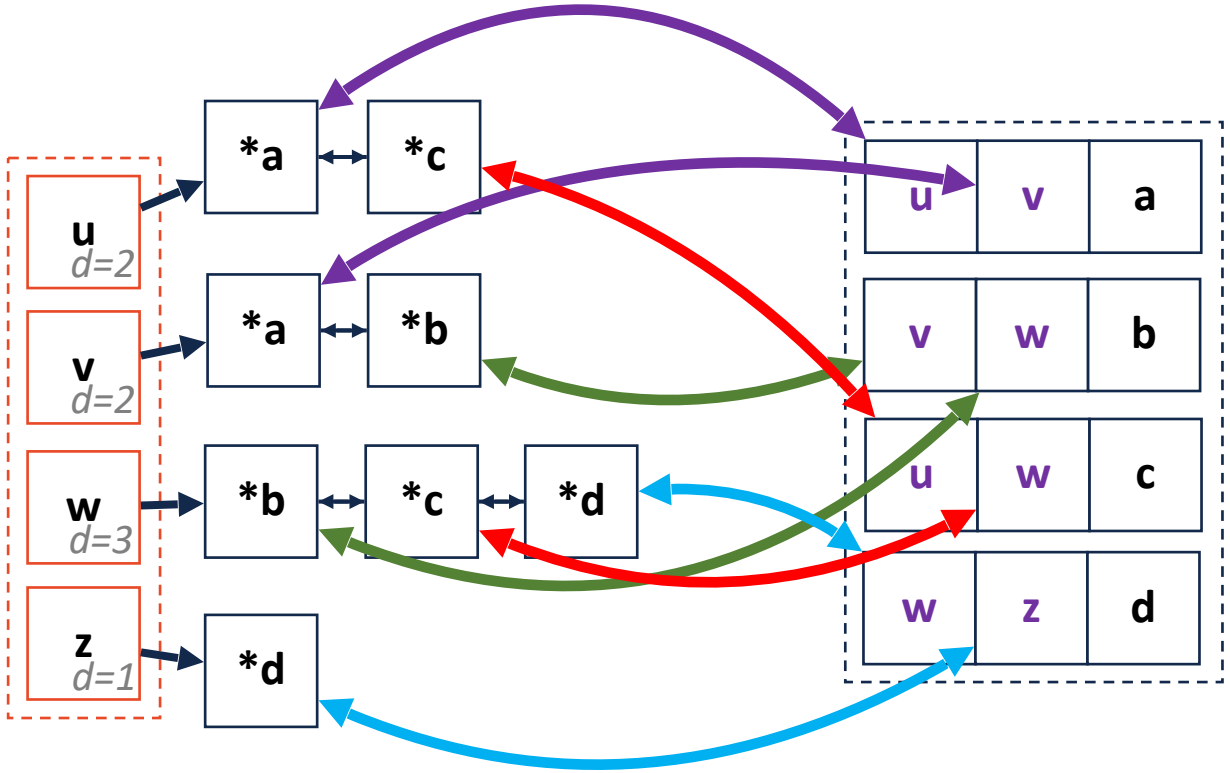


insertVertex(K key):



Adjacency List



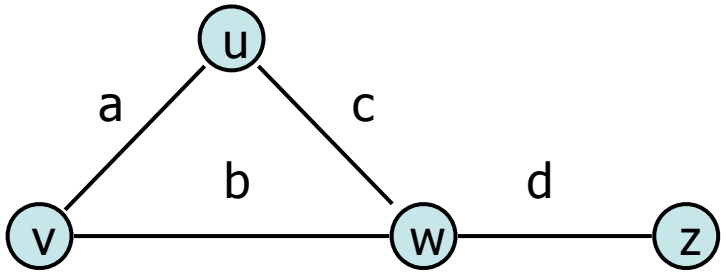


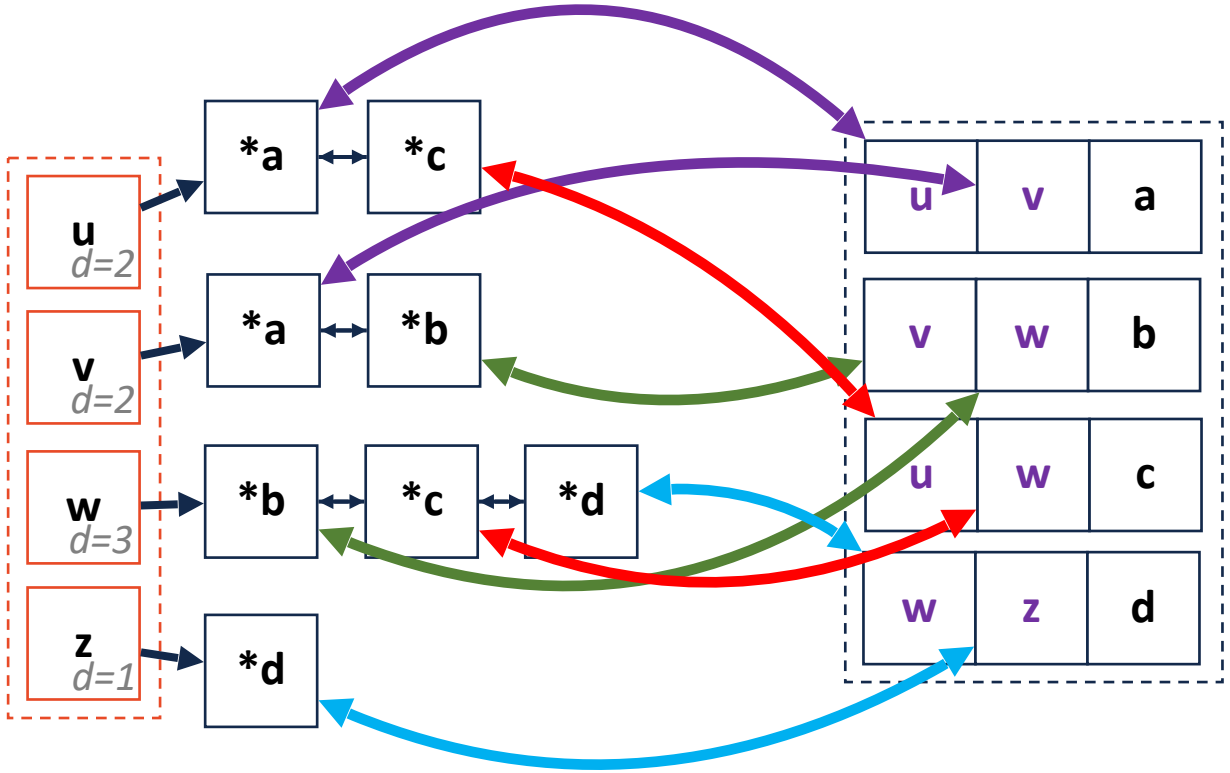
insertVertex(K key):

Add new empty list to vertex list.



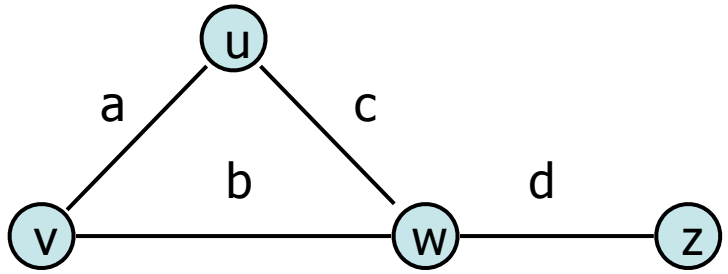
Adjacency List





removeVertex(Vertex v):

Adjacency List



removeVertex(Vertex v):

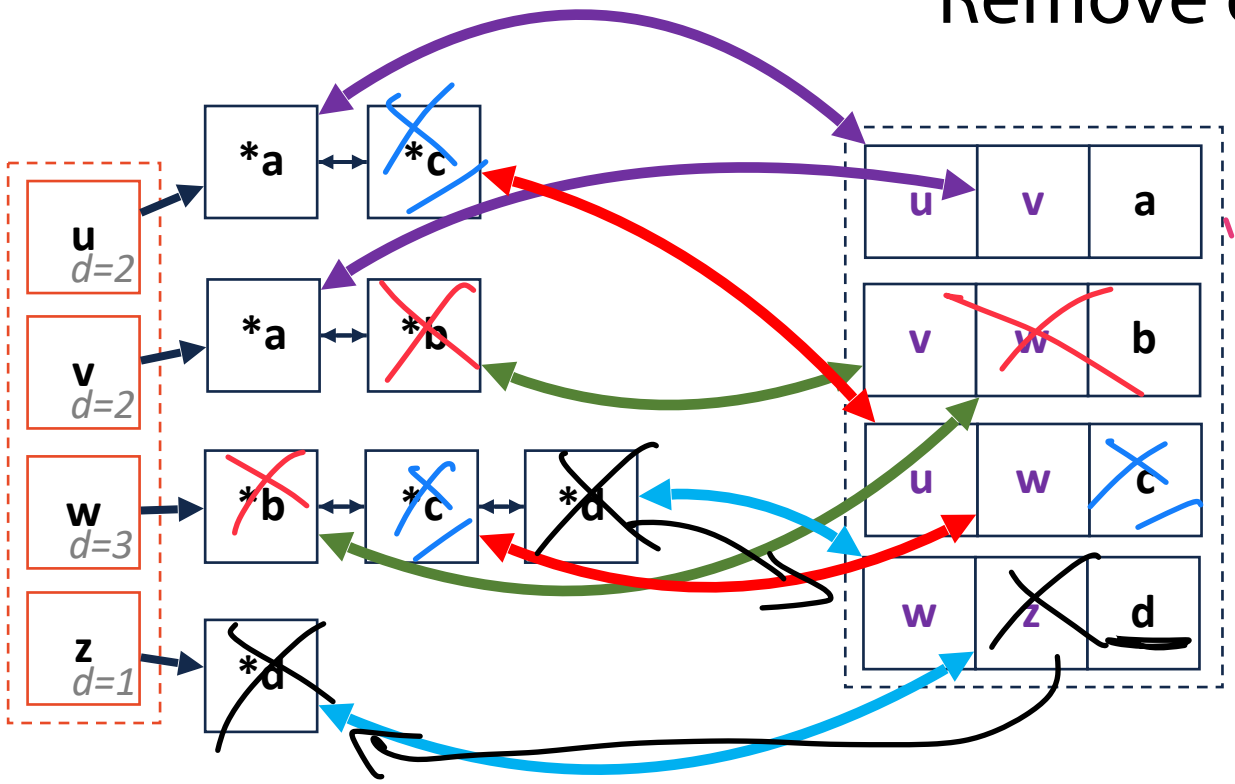
Look up vertex in vertex list

Walk across list

Remove mirrored entry using pointers

Remove edge from edge list

Remove entry from vertex list



Join Code: 225

What are my Big Os?

Storage

Great for edits

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



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* / n^2	deg(v)
insertEdge(u, v)	1	1	1*
removeEdge(u, v)	m	1	min(deg(u), deg(v))
~~incidentEdges(v)~~ Get	m	n	deg(v)
areAdjacent(u, v)	m	1	min(deg(u), deg(v))

dense ≈

sparse ☺

dense case

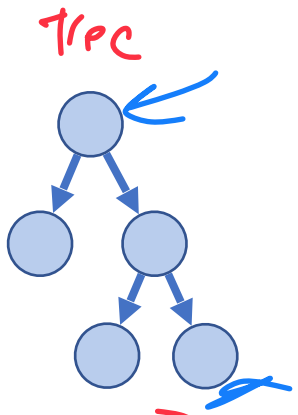
sparse ☺

Graph Traversals

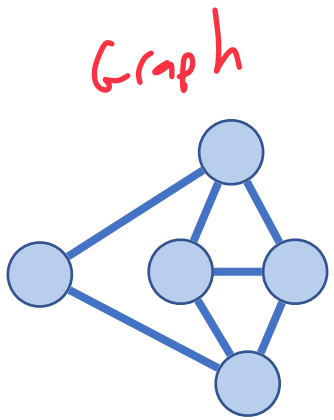
Objective: Visit every vertex and every edge in the graph.

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

Tree traversals won't work — lets compare:



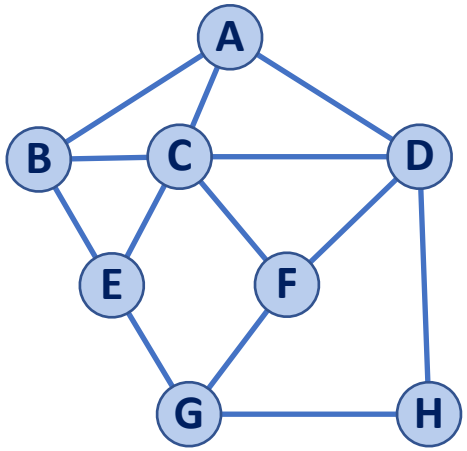
- Rooted
- Acyclic
-



No clear start
clear end

- No clear root
- Potentially many cycles
-

Traversal: BFS



To complete a traversal what do we need?

As input:

↳ a start vertex

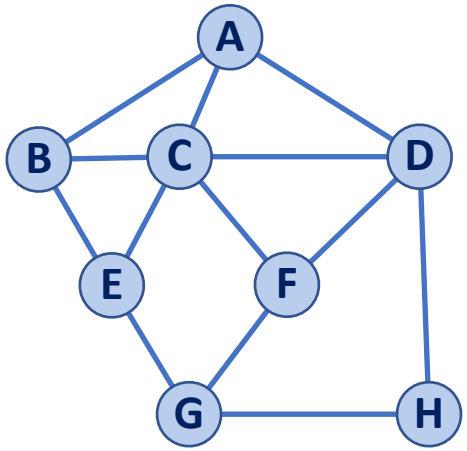
To keep track of:

(to know when to stop)

↳ A list of visited edges (or visited nodes)

↳ A work list or queue to track current state

Traversal: BFS



distance

predecessor

Matrix

v	d	P	Adjacent Edges
A			B C D
B			A C E
C			A B D E F
D			A C F H
E			B C G
F			C D G
G			E F H
H			D G

Stopped here!

Queue

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

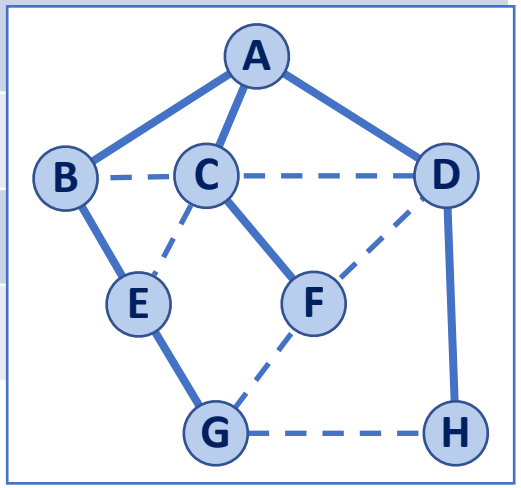
Set dist / pred if new vertex

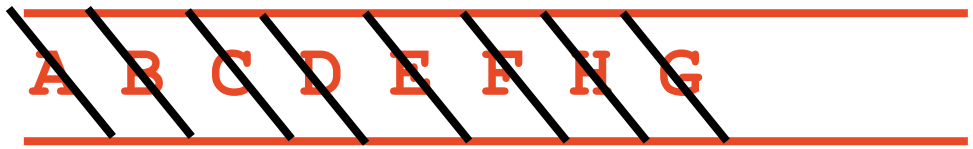
Add unvisited edges to queue

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

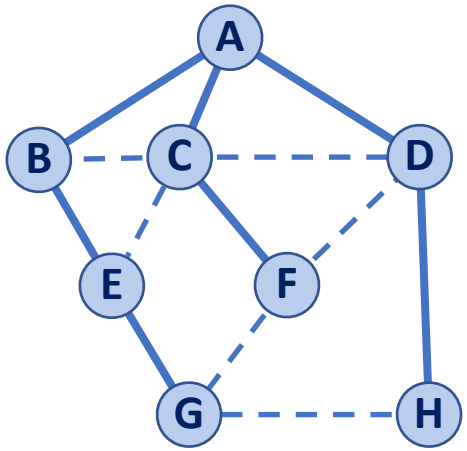




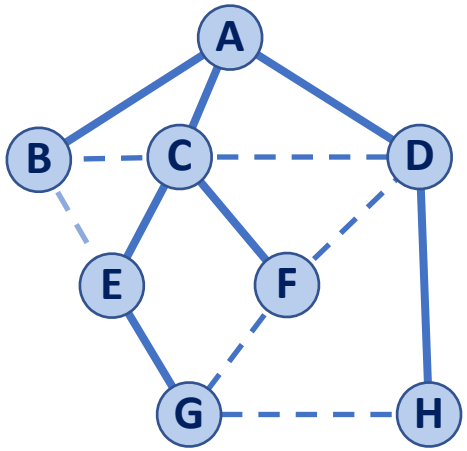
Running time?

Traversal: BFS

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)
```

```
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)
```

```
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)
```

```
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 connected components?

Cycle Detection?

```
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)
```

BFS Observations

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

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

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!

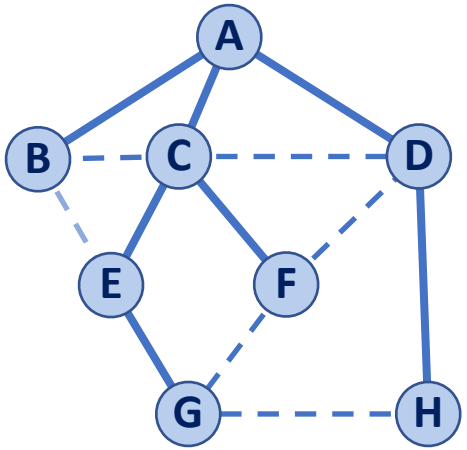
v	d	P	Adjacent Edges
A	0	-	**C B D**
B	1	**A**	**A C E**
C	1	**A**	**B A D E F**
D	1	**A**	**A C F H**
E	2	**C**	**B C G**
F	2	**C**	**C D G**
G	3	**E**	**E F H**
H	2	**D**	**D G**

BFS Observations

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

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

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

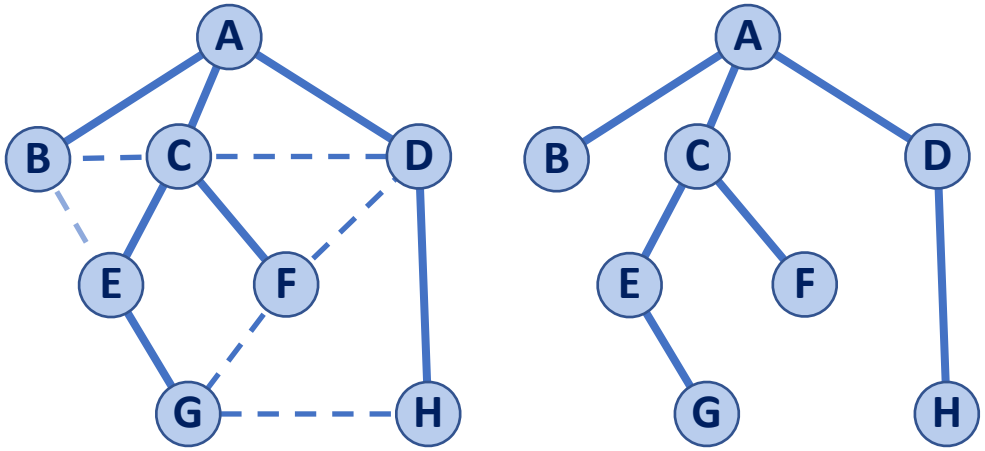
If my node has distance **d**, do I know anything about the nodes connected by a **cross edge**?

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

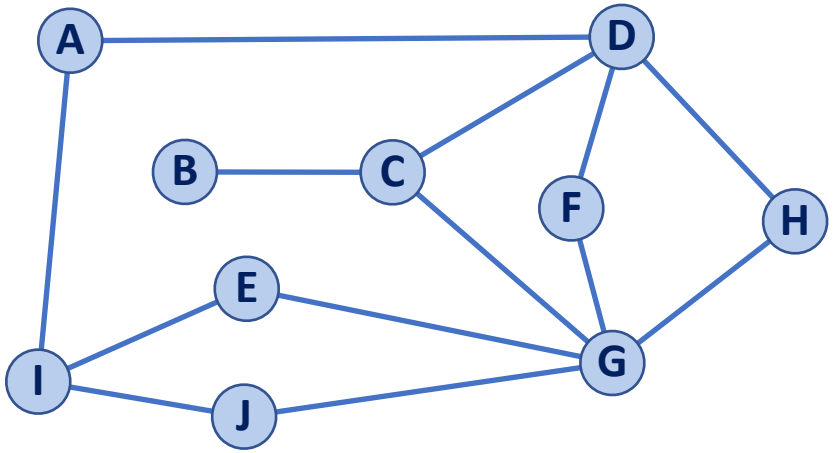


1. BFS can be used to count components
2. BFS can be used to detect cycles

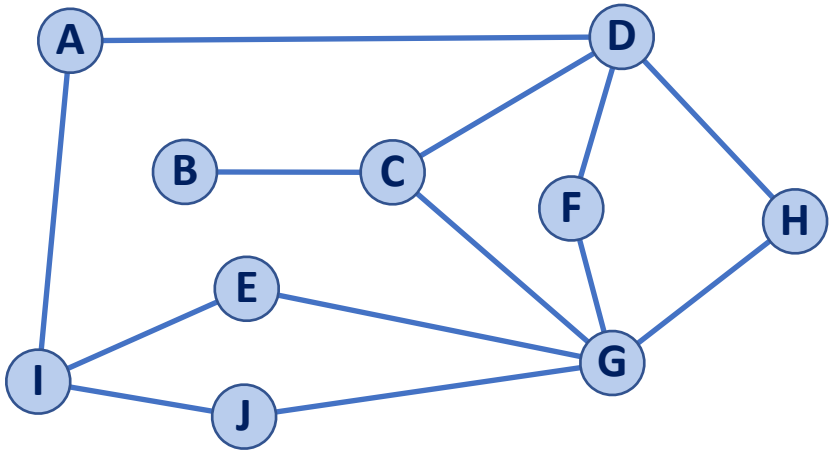
3. The BFS ‘distance’ value is always the shortest distance from source to any vertex (and the discovery edges form a spanning tree)

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

Traversal: DFS



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

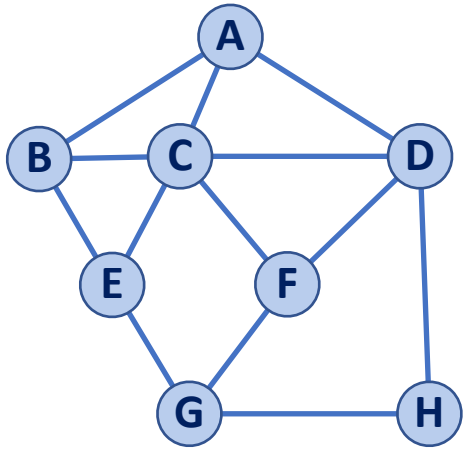
If no unvisited children

stack.pop()

Stack

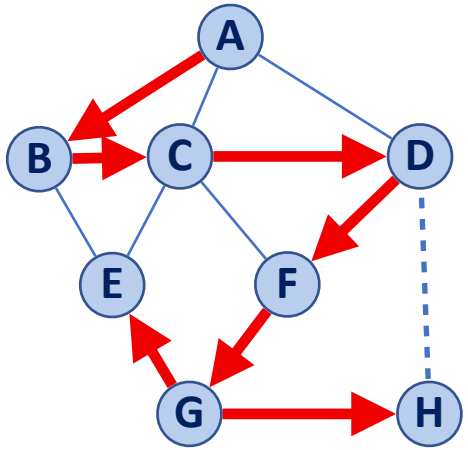


```
1  DFS(G):
2    foreach (Vertex v : G.vertices()):
3      setPred(v, NULL)
4      setDist(v, -1)
5
6    foreach (Edge e : G.edges()):
7      setLabel(e, UNEXPLORED)
8
9    foreach (Vertex v : G.vertices()):
10     if getDist(v) == -1:
11       DFS(G, v)
```




```
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)
```



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

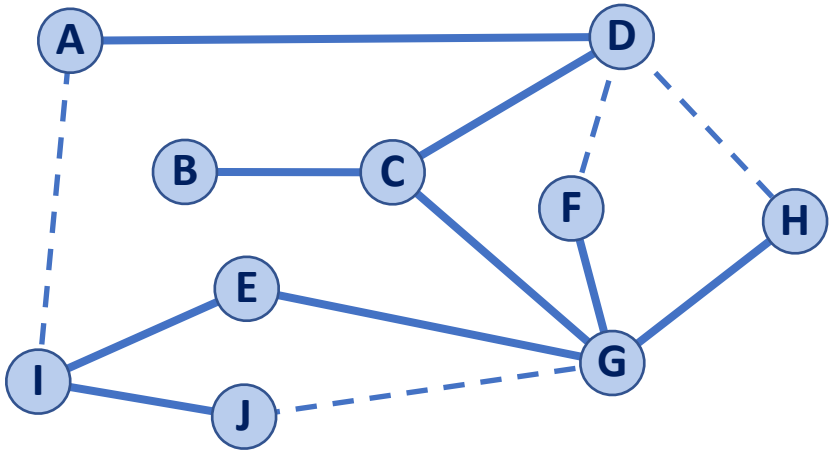
Running time?

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?

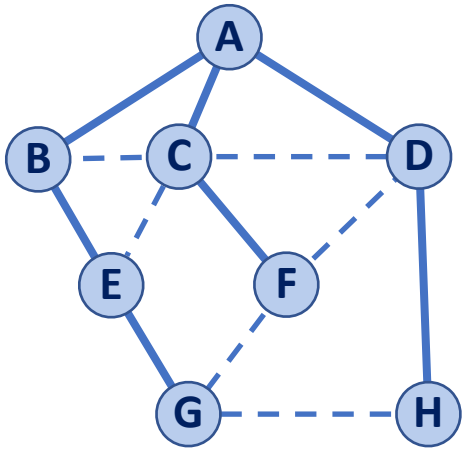
Does distance have meaning here?

Do our edge labels have meaning here?

Efficiency: DFS vs BFS

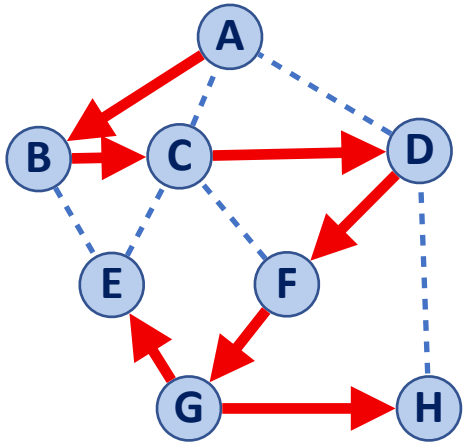
$|V| = n, |E| = m$

BFS:



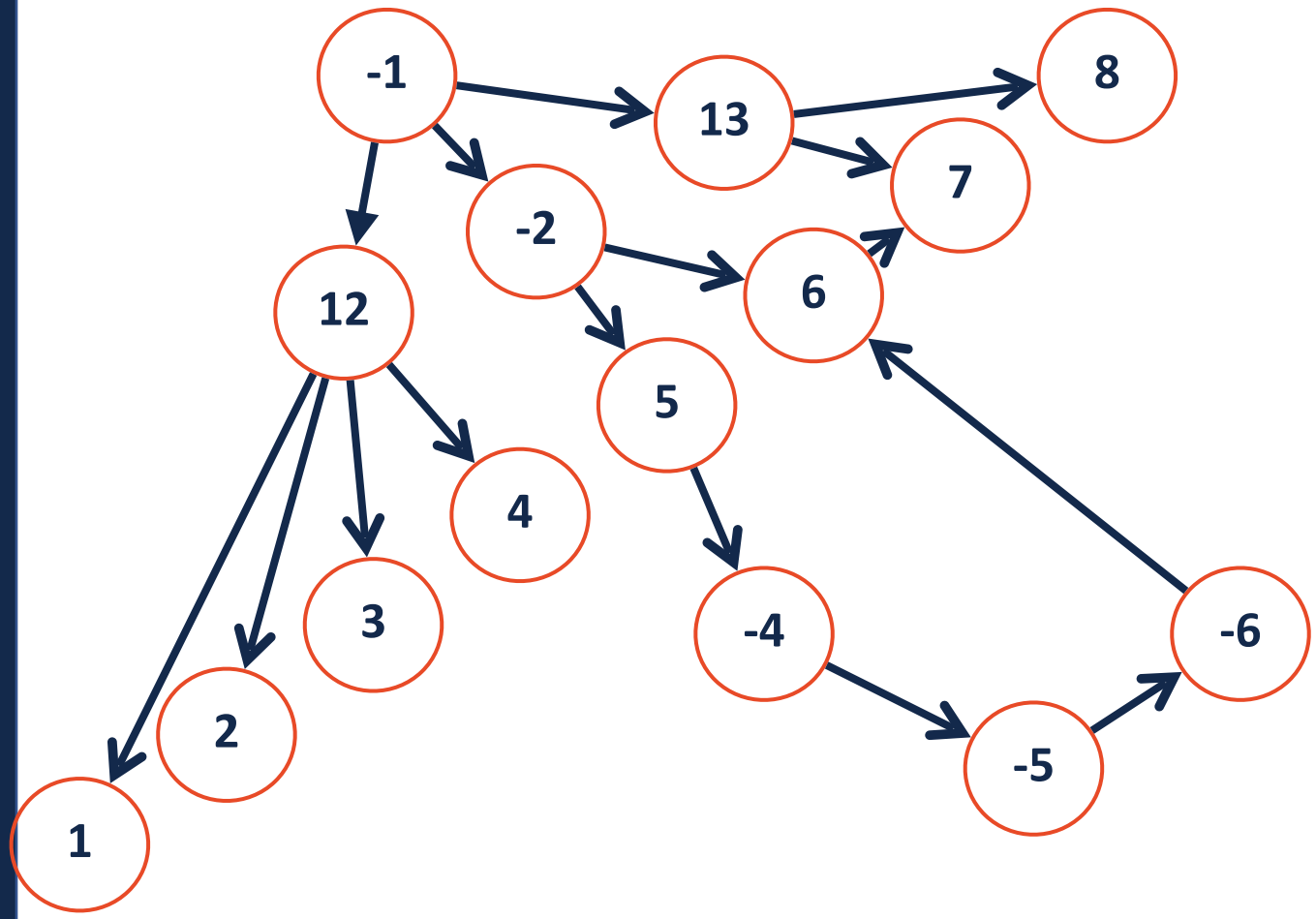
A B C D E F H G

DFS:



A B C D F G E H

Space Efficiency: DFS vs BFS



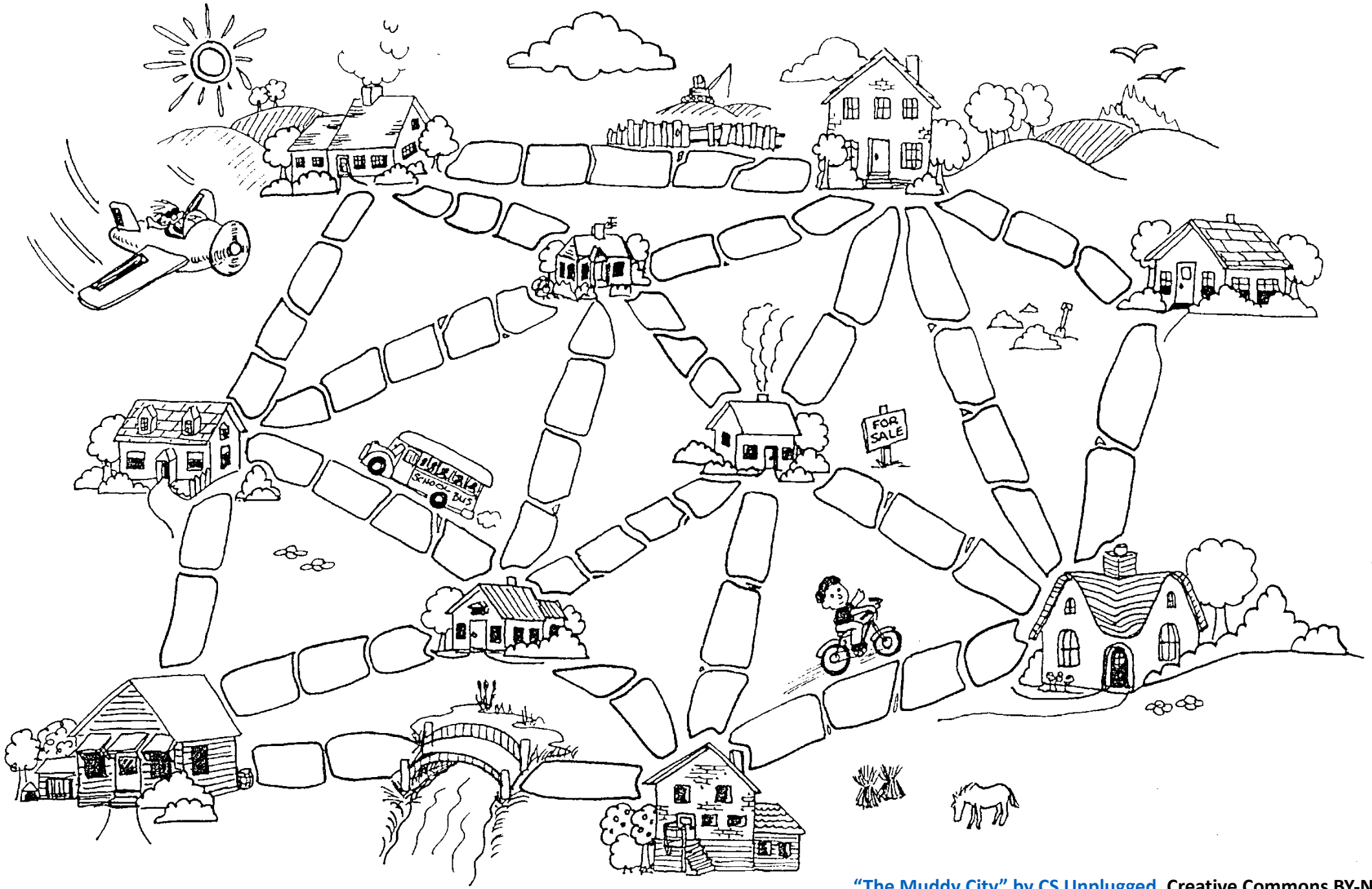
Summary: DFS and BFS

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



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



"The Muddy City" by CS Unplugged, Creative Commons BY-NC-SA 4.0

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

