CS 225

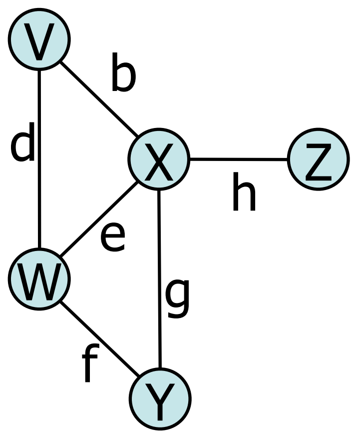
Data Structures

November 16 – Graph Implementations and Traversals

G Carl Evans

Graph ADT

Data:
- **Vertices**
- **Edges**
- **Some data structure maintaining the structure between vertices and edges.**



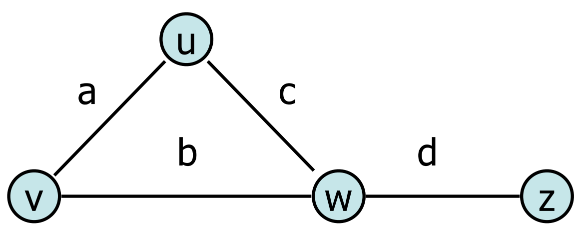
Functions:
- **insertVertex(K key);**
- **insertEdge(Vertex v1, Vertex v2, K key);**

- **removeVertex(Vertex v);**
- **removeEdge(Vertex v1, Vertex v2);**

- **incidentEdges(Vertex v);**
- **areAdjacent(Vertex v1, Vertex v2);**

- **origin(Edge e);**
- **destination(Edge e);**

Edge List



Vertex List

u
v
w
z

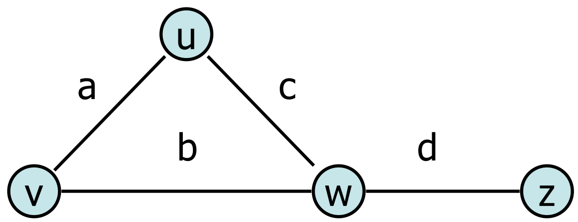
Edge List

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

Key Ideas:

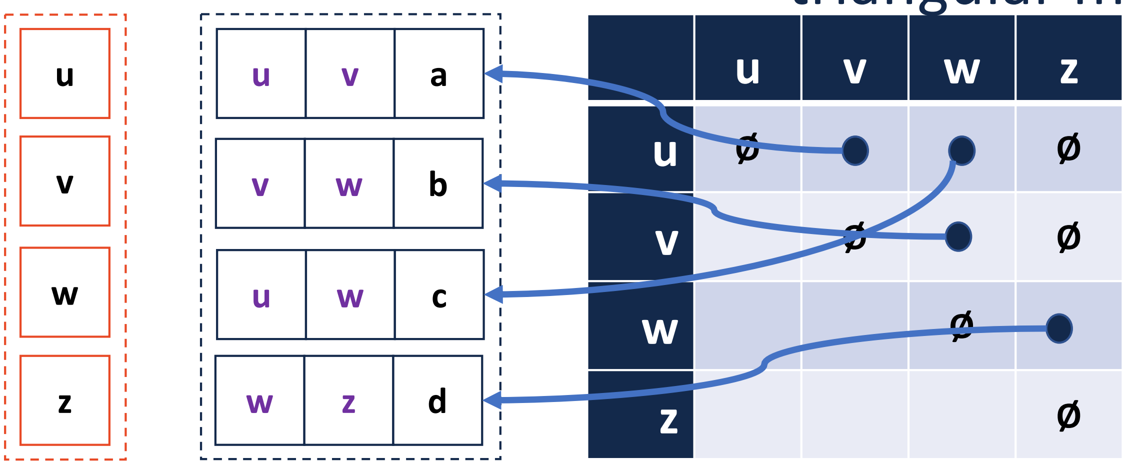
- Given a vertex, O(1) lookup in vertex list
  - Implement w/ a hash table, etc
- All basic ADT operations runs in O(m) time

Adjacency Matrix

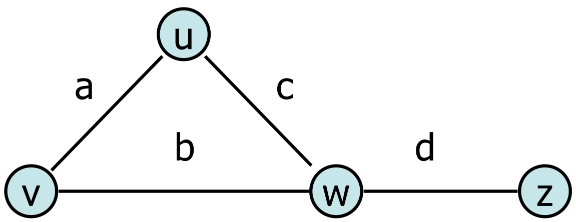


Key Ideas:

- Given a vertex, O(1) lookup in vertex list
- Given a pair of vertices (an edge), O(1) lookup in the matrix
- Undirected graphs can use an upper triangular matrix



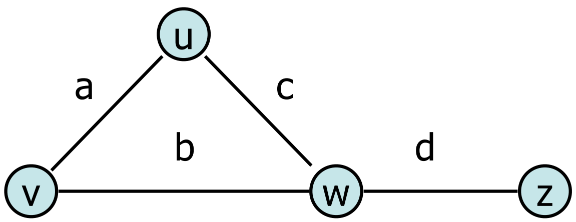
Graph Implementation: Adjacency List

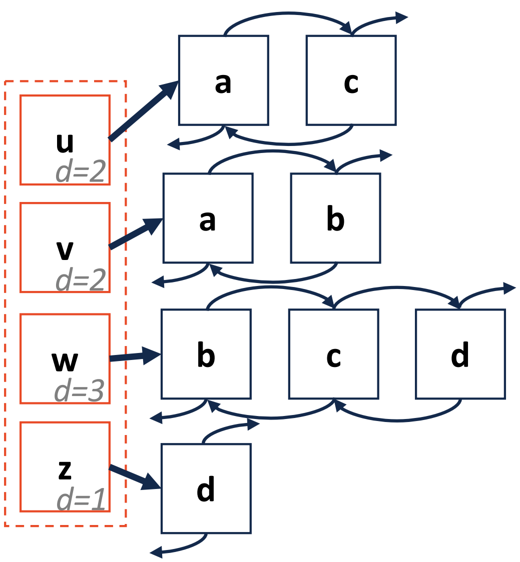


u
v
w
z

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

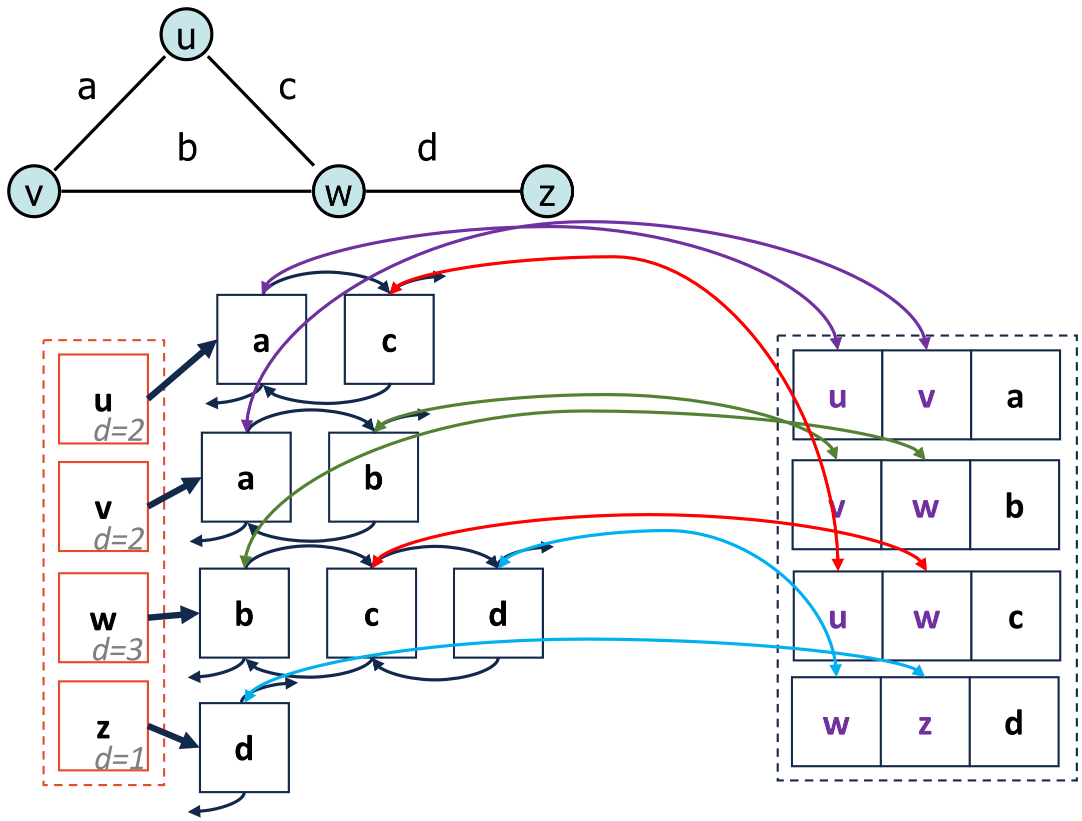
Adjacency List





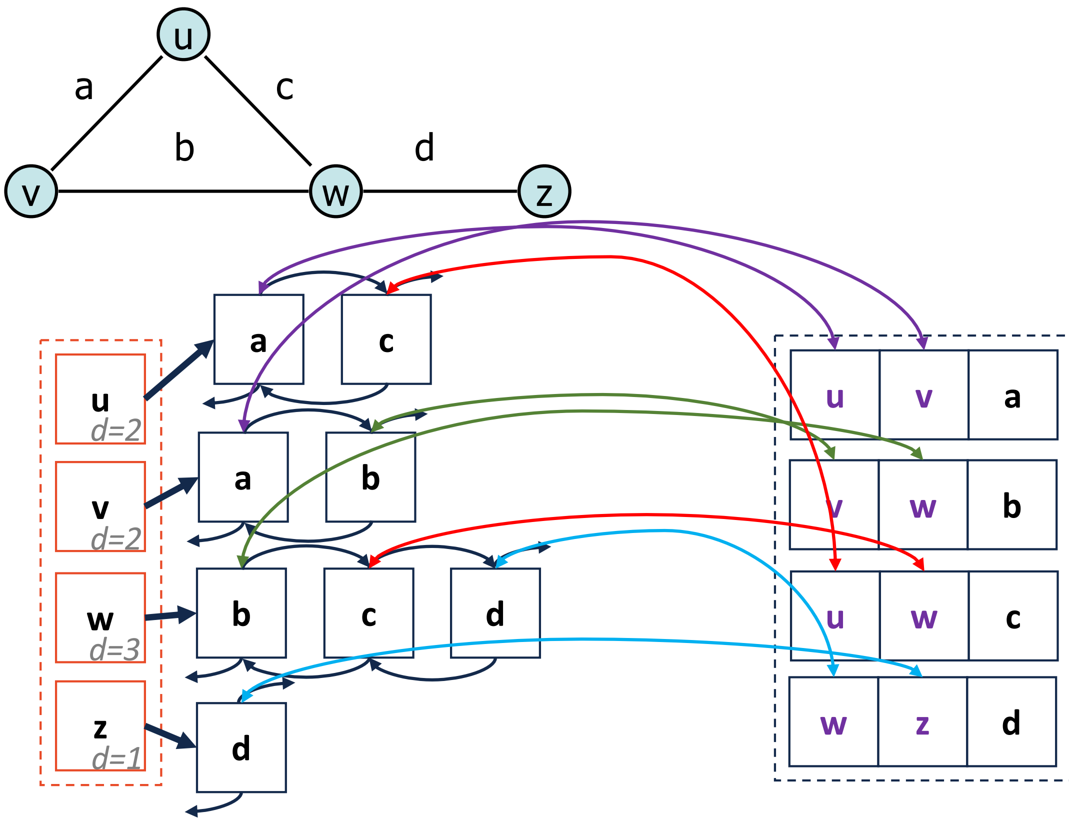
u	v	a
v	w	b
u	w	c
w	z	d

Adjacency List



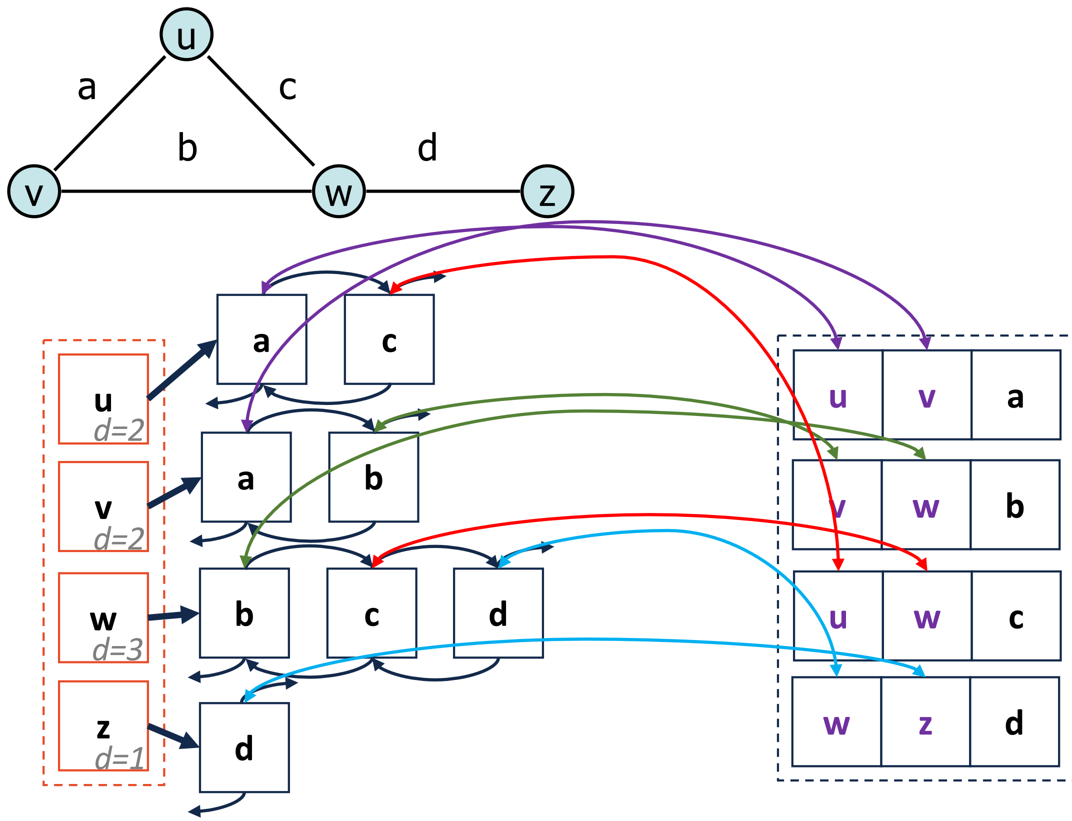
Adjacency List

insertVertex(K key):



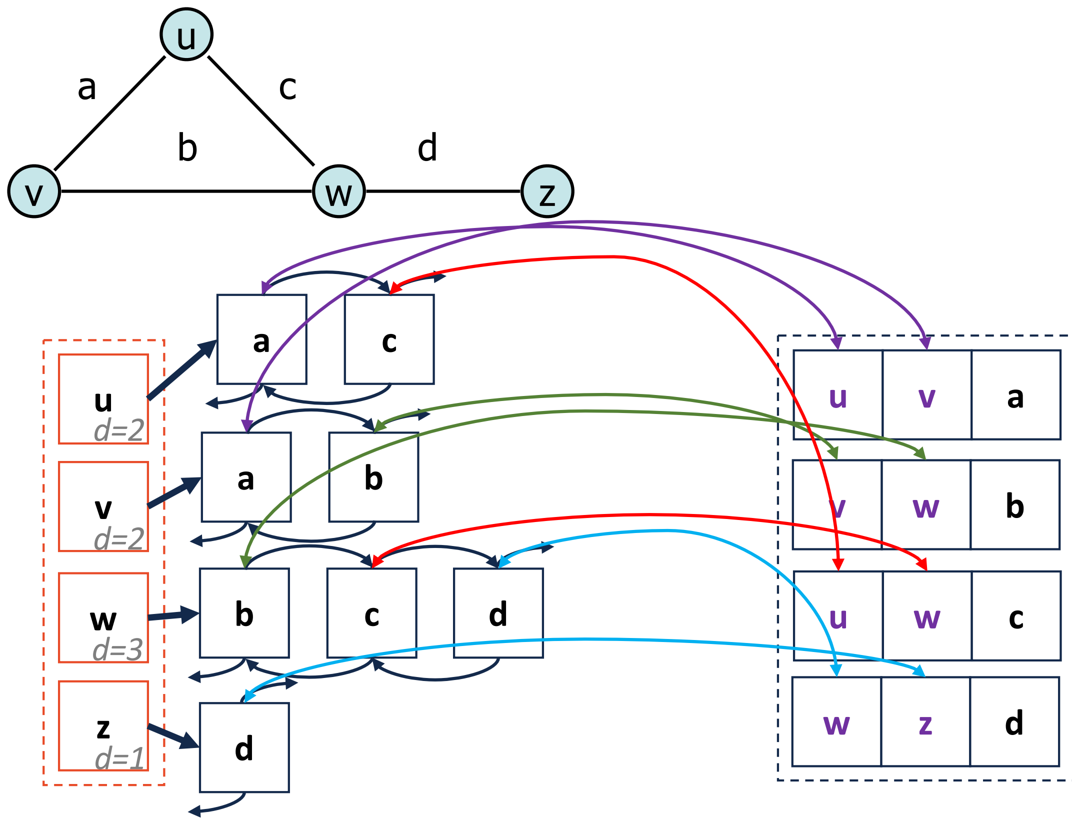
Adjacency List

removeVertex(Vertex v):



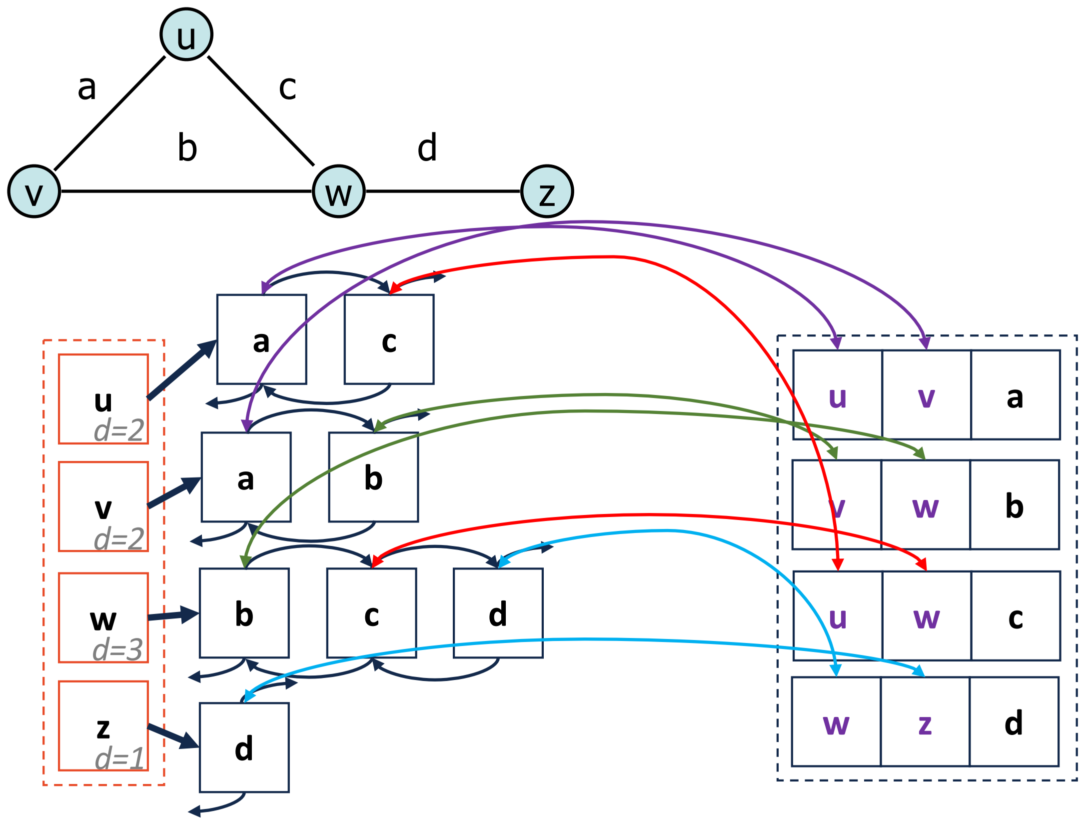
Adjacency List

incidentEdges(Vertex v):



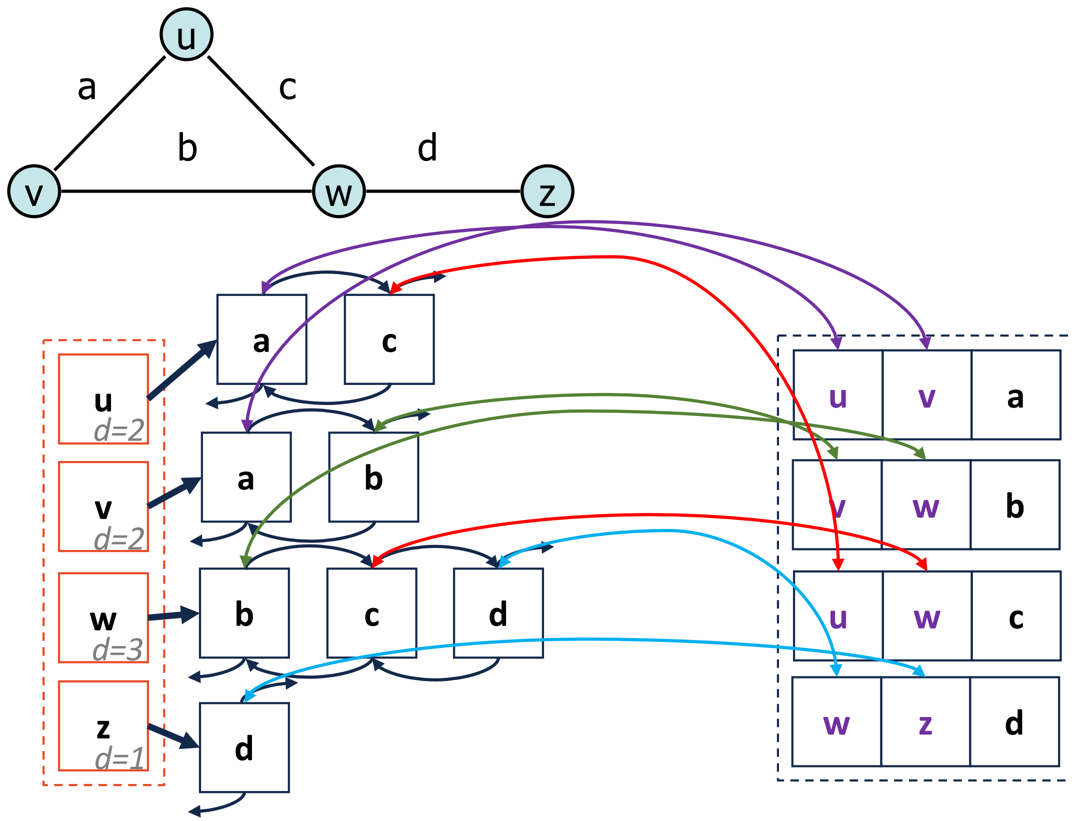
Adjacency List

areAdjacent(Vertex v1, Vertex v2):



Adjacency List

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



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

mp_traversals and mp_mazes

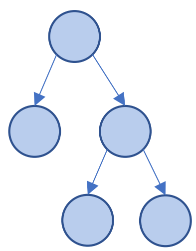
- mp_traversals
  - Vertex Set : The pixels are the vertices
  - Edge Set : There is an edge between every n/s/e/w pixel unless the color change exceeds the tolerance
  - There are several graphs here depending on the tolerance
- mp_mazes
  - Vertex Set: The squares in the maze are the vertices
  - Edge Set: There is an edge between two vertices if canTravel() returns true
  - Once the maze is made this graph is a spanning tree of the graph with canTravel() returning true.

Traversal:

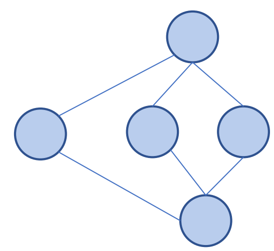
Objective: Visit every vertex and every edge in the graph.

Purpose: Search for interesting sub-structures in the graph.

We've seen traversal before ....but it's different:

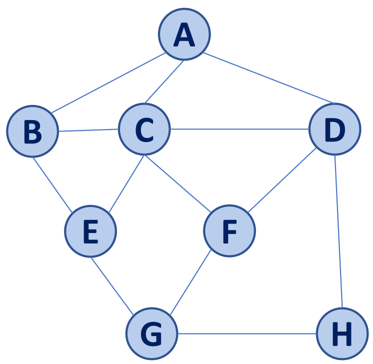


- Ordered
- Obvious Start
-

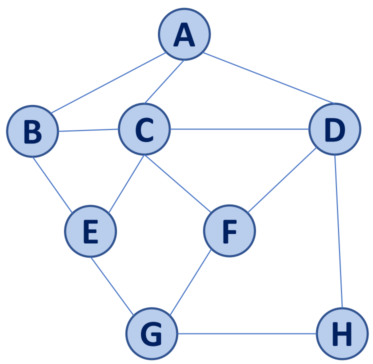


-
-
-

Traversal: BFS

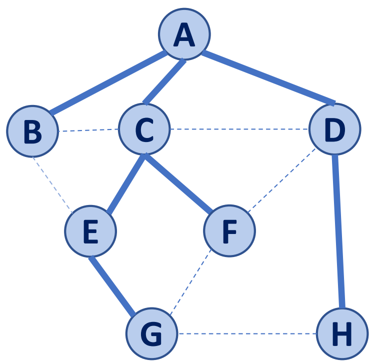


Traversal: BFS

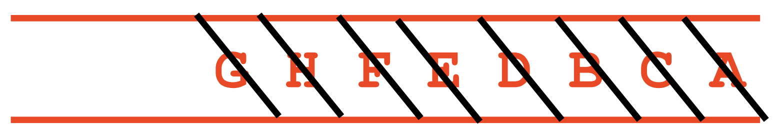


v	d	P	Adjacent Edges
A			
B			
C			
D			
E			
F			
G			
H			

Traversal: BFS



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



```
BFS(G):
  Input: Graph, G
  Output: A labeling of the edges on
      G as discovery and cross edges

  foreach (Vertex v : G.vertices()):
    setLabel(v, UNEXPLORED)
  foreach (Edge e : G.edges()):
    setLabel(e, UNEXPLORED)
  foreach (Vertex v : G.vertices()):
    if getLabel(v) == UNEXPLORED:
      BFS(G, v)
```

```
BFS(G, v):
  Queue q
  setLabel(v, VISITED)
  q.enqueue(v)

  while !q.empty():
    v = q.dequeue()
    foreach (Vertex w : G.adjacent(v)):
      if getLabel(w) == UNEXPLORED:
         setLabel(v, w, DISCOVERY)
         setLabel(w, VISITED)
         q.enqueue(w)
      elseif getLabel(v, w) == UNEXPLORED:
         setLabel(v, w, CROSS)
```

BFS Analysis

Q: Does our implementation handle disjoint graphs? If so, what code handles this?

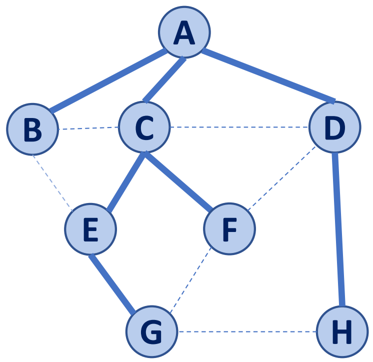
- ***How do we use this to count components?***

Q: Does our implementation detect a cycle?

- ***How do we update our code to detect a cycle?***

Q: What is the running time?

Running time of BFS



While-loop at **:19**?

For-loop at **:21**?

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

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

```
BFS(G):
  Input: Graph, G
  Output: A labeling of the edges on
      G as discovery and cross edges

  foreach (Vertex v : G.vertices()):
    setLabel(v, UNEXPLORED)
  foreach (Edge e : G.edges()):
    setLabel(e, UNEXPLORED)
  foreach (Vertex v : G.vertices()):
    if getLabel(v) == UNEXPLORED:
       BFS(G, v)
```

```
BFS(G, v):
  Queue q
  setLabel(v, VISITED)
  q.enqueue(v)

  while !q.empty():
    v = q.dequeue()
    foreach (Vertex w : G.adjacent(v)):
      if getLabel(w) == UNEXPLORED:
         setLabel(v, w, DISCOVERY)
         setLabel(w, VISITED)
         q.enqueue(w)
      elseif getLabel(v, w) == UNEXPLORED:
         setLabel(v, w, CROSS)
```

BFS Observations

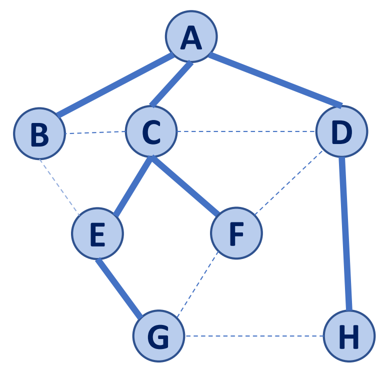
Q: What is a shortest path from **A** to **H**?

Q: What is a shortest path from **E** to **H**?

Q: How does a cross edge relate to **d**?

Q: What structure is made from discovery edges?

d	p	v	Adjacent
0	A	A	C B D
1	A	B	A C E
1	A	C	B A D E F
1	A	D	A C F H
2	C	E	B C G
2	C	F	C D G
3	E	G	E F H
2	D	H	D G



BFS Observations

Obs. 1: Traversals can be used to count components.

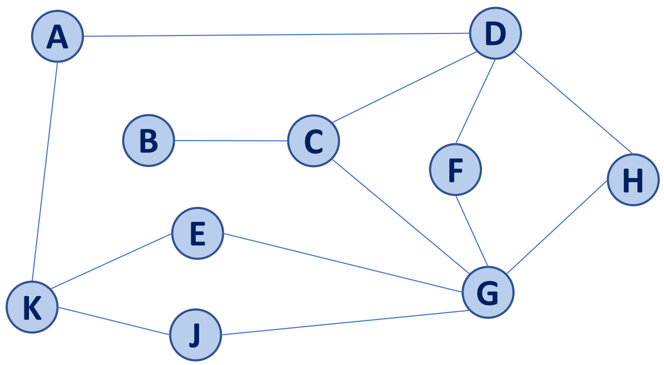
Obs. 2: Traversals can be used to detect cycles.

Obs. 3: In BFS, **d** provides the shortest distance to every vertex.

Obs. 4: In BFS, the endpoints of a cross edge never differ in distance, **d**, by more than 1:

$$|d(u) - d(v)| = 1$$

Traversal: DFS



```
BFS(G):
  Input: Graph, G
  Output: A labeling of the edges on
      G as discovery and cross edges

  foreach (Vertex v : G.vertices()):
    setLabel(v, UNEXPLORED)
  foreach (Edge e : G.edges()):
    setLabel(e, UNEXPLORED)
  foreach (Vertex v : G.vertices()):
    if getLabel(v) == UNEXPLORED:
       BFS(G, v)
```

```
BFS(G, v):
  Queue q
  setLabel(v, VISITED)
  q.enqueue(v)

  while !q.empty():
    v = q.dequeue()
    foreach (Vertex w : G.adjacent(v)):
      if getLabel(w) == UNEXPLORED:
         setLabel(v, w, DISCOVERY)
         setLabel(w, VISITED)
         q.enqueue(w)
      elseif getLabel(v, w) == UNEXPLORED:
         setLabel(v, w, CROSS)
```

```
DFS(G):
  Input: Graph, G
  Output: A labeling of the edges on
      G as discovery and back edges

  foreach (Vertex v : G.vertices()):
    setLabel(v, UNEXPLORED)
  foreach (Edge e : G.edges()):
    setLabel(e, UNEXPLORED)
  foreach (Vertex v : G.vertices()):
    if getLabel(v) == UNEXPLORED:
       DFS(G, v)
```

```
DFS(G, v):
~~  Queue q~~
  setLabel(v, VISITED)
~~  q.enqueue(v)~~

~~  while !q.empty():~~
~~    v = q.dequeue()~~
    foreach (Vertex w : G.adjacent(v)):
      if getLabel(w) == UNEXPLORED:
         setLabel(v, w, DISCOVERY)
         ~~setLabel(w, VISITED)~~
         DFS(G, w)
      elseif getLabel(v, w) == UNEXPLORED:
         setLabel(v, w, BACK)
```

Running time of DFS

Labeling:

- Vertex:

- Edge:

Queries:

- Vertex:

- Edge: