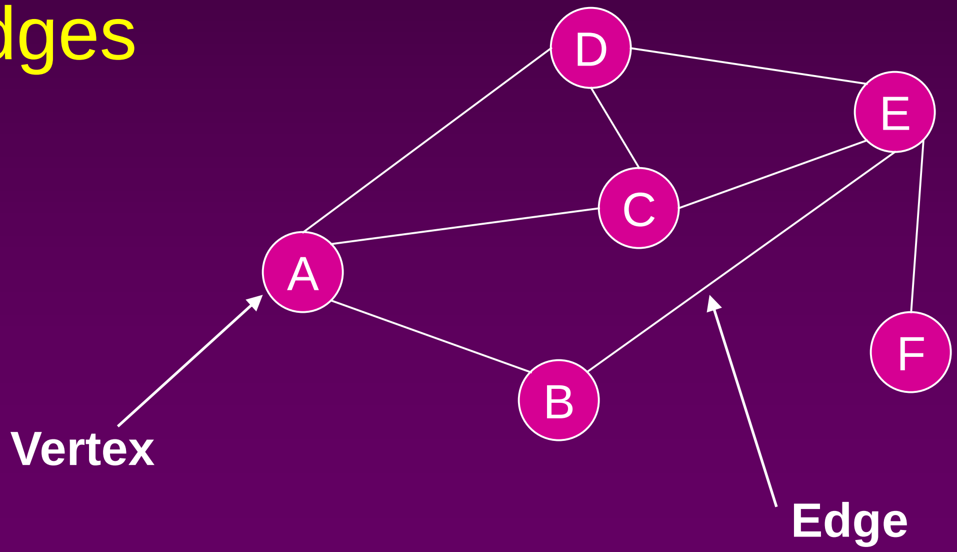
CSE 301
Combinatorial Optimization

Graph & BFS (revisit)

Graphs

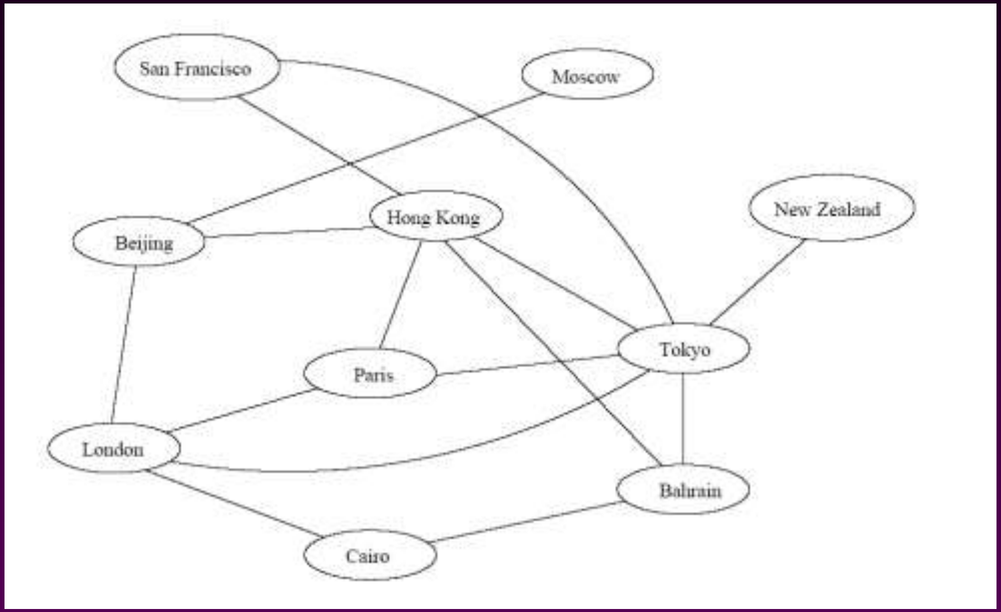
- Extremely useful tool in modeling problems
- Consist of:
  - Vertices
  - Edges



Vertices can be considered “sites” or locations.

Edges represent connections.

Application

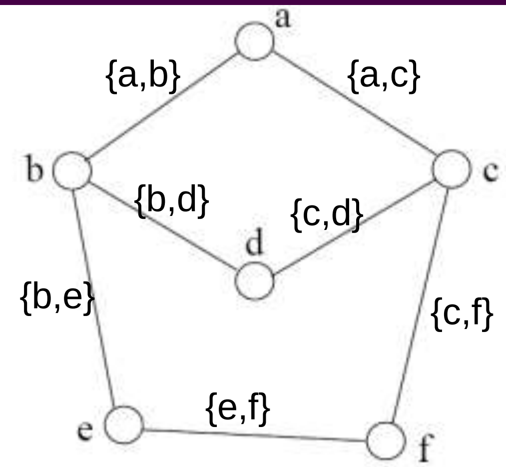


Air flight system

- Each vertex represents a city
- Each edge represents a direct flight between two cities
- A query on direct flights = a query on whether an edge exists
- A query on how to get to a location = does a path exist from A to B
- We can even associate costs to edges (weighted graphs), then ask "what is the cheapest path from A to B"

Definition

- A graph G=(V, E) consists a set of vertices, V, and a set of edges, E.
- Each edge is a pair of *(v, w)*, where v, w belongs to V
- If the pair is unordered, the graph is undirected; otherwise it is directed



$$V = \{a, b, c, d, e, f\}$$

$$E = \{\{a, b\}, \{a, c\}, \{b, d\}, \{c, d\}, \{b, e\}, \{c, f\}, \{e, f\}\}$$

An undirected graph

Definition

- Complete Graph
  - How many edges are there in an N-vertex complete graph?
- Bipartite Graph
  - What is its property? How can we detect it?
- Path
- Tour
- Degree of a vertices
  - Indegree
  - Outdegree
  - Indegree+outdegree = Even (why??)

Graph Variations

- Variations:
  - A *connected graph* has a path from every vertex to every other
  - In an *undirected graph:*
    - Edge (u,v) = edge (v,u)
    - No self-loops
  - In a *directed* graph:
    - Edge (u,v) goes from vertex u to vertex v, notated u→v

Graph Variations

- More variations:
  - A *weighted graph* associates weights with either the edges or the vertices
    - E.g., a road map: edges might be weighted w/ distance
  - A *multigraph* allows multiple edges between the same vertices
    - E.g., the call graph in a program (a function can get called from multiple points in another function)

Graphs

- We will typically express running times in terms of |E| and |V| (often dropping the |'s)
  - If $|E| \approx |V|^2$ the graph is *dense*
  - If $|E| \approx |V|$ the graph is *sparse*
- If you know you are dealing with dense or sparse graphs, different data structures may make sense

Graph Representation

- Two popular computer representations of a graph. Both represent the vertex set and the edge set, but in different ways.

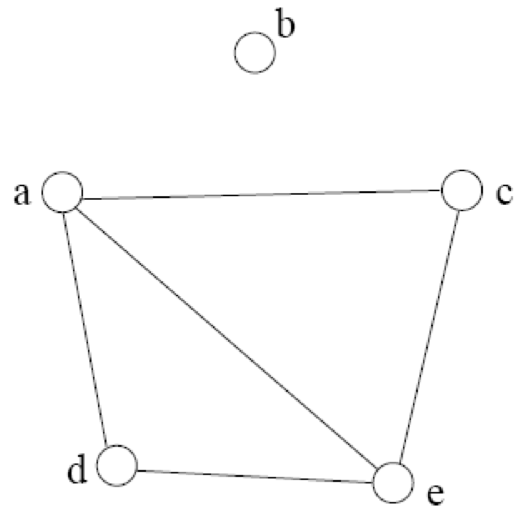
1. Adjacency Matrix

   Use a 2D matrix to represent the graph

1. Adjacency List

   Use a 1D array of linked lists

Adjacency Matrix



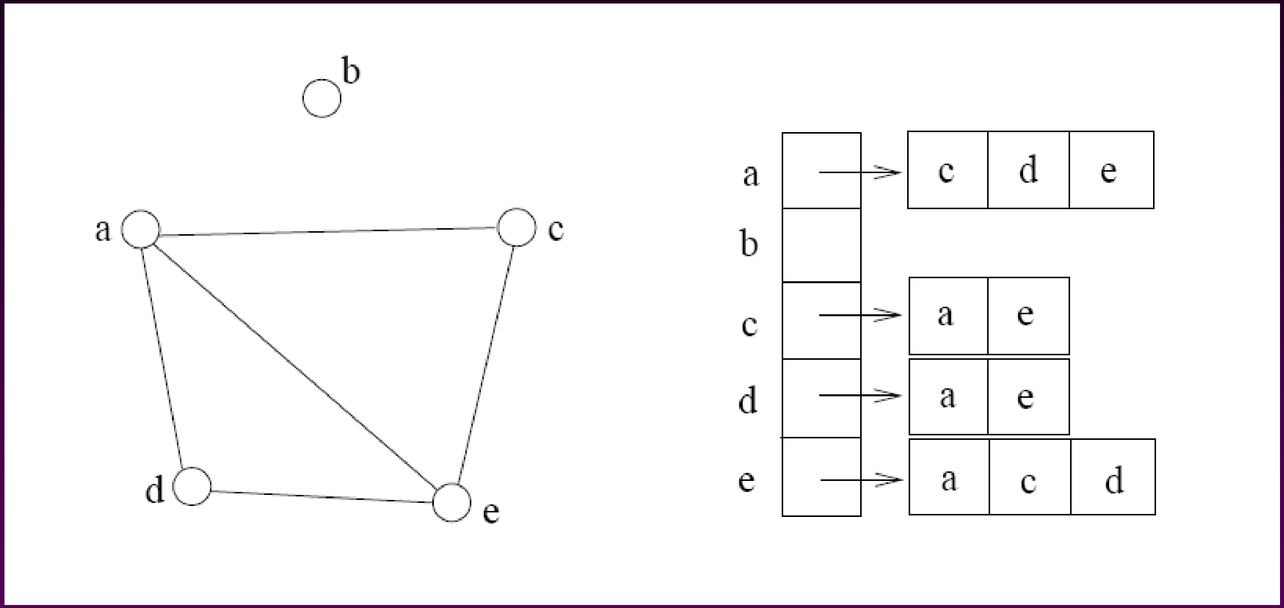
	a	b	c	d	e
a	0	0	1	1	1
b	0	0	0	0	0
c	1	0	0	0	1
d	1	0	0	0	1
e	1	0	1	1	0

- 2D array A[0..n-1, 0..n-1], where ***n*** is the number of vertices in the graph
- Each row and column is indexed by the vertex id
  - e,g a=0, b=1, c=2, d=3, e=4
- *A[i][j]=1* if there is an edge connecting vertices *i* and *j*; otherwise, *A[i][j]=0*
- The storage requirement is $\Theta(n^2)$. It is not efficient if the graph has few edges. An adjacency matrix is an appropriate representation if the graph is dense: $|E|=\Theta(|V|^2)$
- We can detect in O(1) time whether two vertices are connected.

Simple Questions on Adjacency Matrix

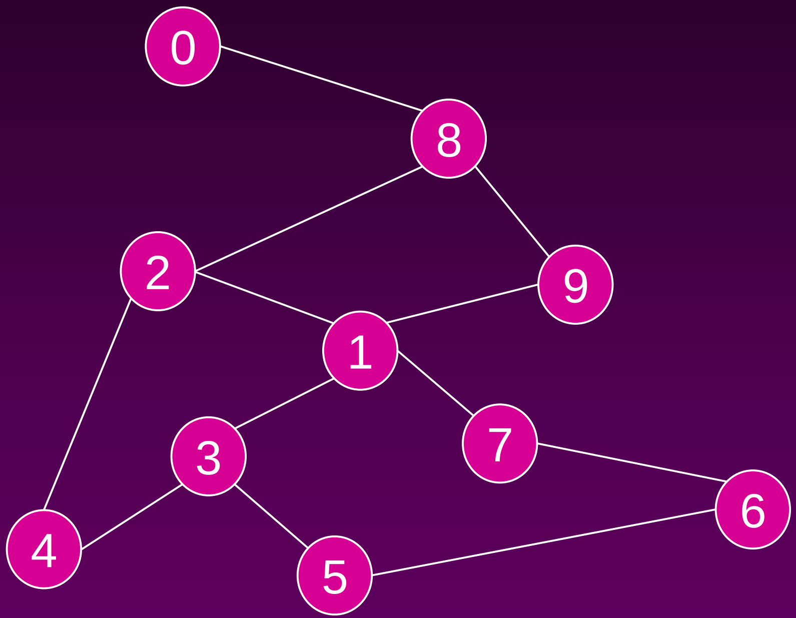
- Is there a direct link between A and B?
- What is the indegree and outdegree for a vertex A?
- How many nodes are directly connected to vertex A?
- Is it an undirected graph or directed graph?
- Suppose ADJ is an NxN matrix. What will be the result if we create another matrix ADJ2 where ADJ2=ADJxADJ?

Adjacency List



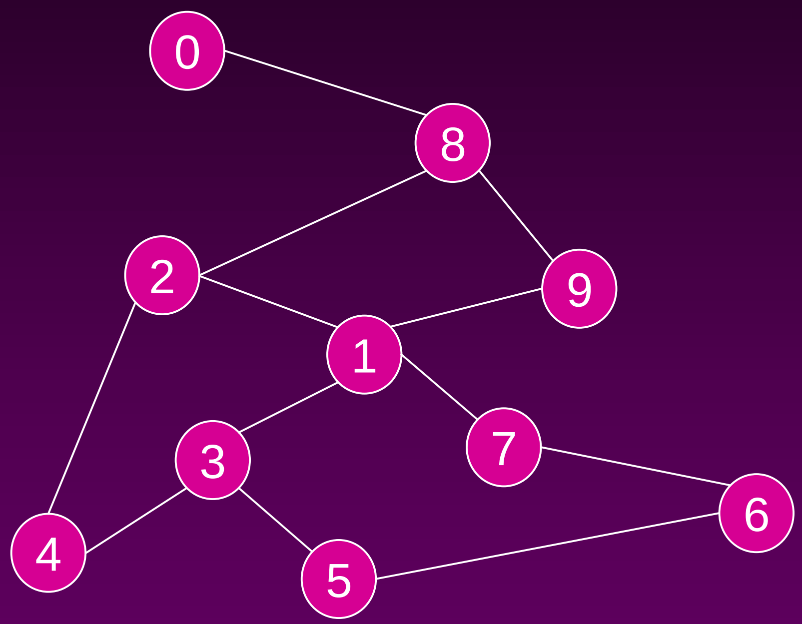
- If the graph is not dense, in other words, sparse, a better solution is an adjacency list
- The adjacency list is an array A[0..n-1] of lists, where n is the number of vertices in the graph.
- Each array entry is indexed by the vertex id
- Each list *A[i]* stores the ids of the vertices adjacent to vertex *i*

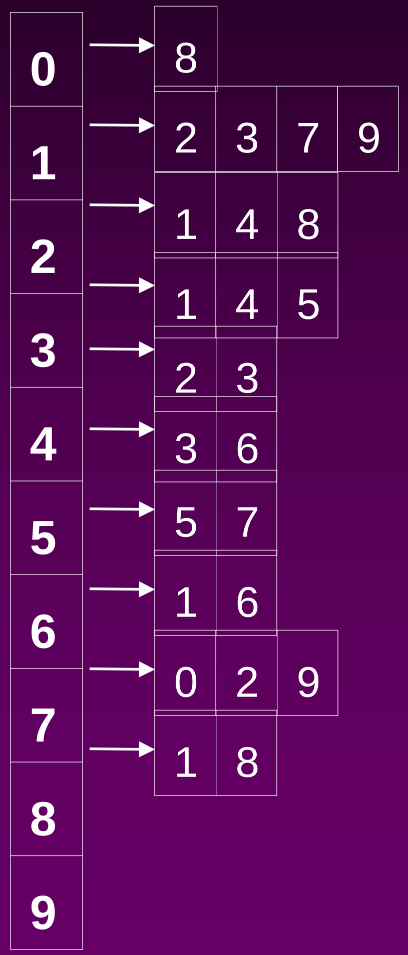
Adjacency Matrix Example



	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	1	0
1	0	0	1	1	0	0	0	1	0	1
2	0	1	0	0	1	0	0	0	1	0
3	0	1	0	0	1	1	0	0	0	0
4	0	0	1	1	0	0	0	0	0	0
5	0	0	0	1	0	0	1	0	0	0
6	0	0	0	0	0	1	0	1	0	0
7	0	1	0	0	0	0	1	0	0	0
8	1	0	1	0	0	0	0	0	0	1
9	0	1	0	0	0	0	0	0	1	0

Adjacency List Example





Storage of Adjacency List

- The array takes up $\Theta(n)$ space
- Define degree of *v*, deg(*v*), to be the number of edges incident to *v*. Then, the total space to store the graph is proportional to:

$$\sum_{\text{vertex } v} \deg(v)$$

- An edge *e={u,v}* of the graph contributes a count of 1 to deg(*u*) and contributes a count 1 to deg(*v*)
- Therefore, $\Sigma_{\text{vertex } v}\deg(v) = 2m$, where *m* is the total number of edges
- In all, the adjacency list takes up $\Theta(n+m)$ space
  - If $m = O(n^2)$ (i.e. dense graphs), both adjacent matrix and adjacent lists use $\Theta(n^2)$ space.
  - If $m = O(n)$, adjacent list outperform adjacent matrix

- However, one cannot tell in O(1) time whether two vertices are connected

Adjacency List vs. Matrix

- **Adjacency List**
  - More compact than adjacency matrices if graph has few edges
  - Requires more time to find if an edge exists

- **Adjacency Matrix**
  - Always require n^2 space
    - This can waste a lot of space if the number of edges are sparse
  - Can quickly find if an edge exists

Path between Vertices

- A path is a sequence of vertices (v_0, v_1, v_2,… v_k) such that:
  - For $0 \le i < k$, $\{v_i, v_{i+1}\}$ is an edge

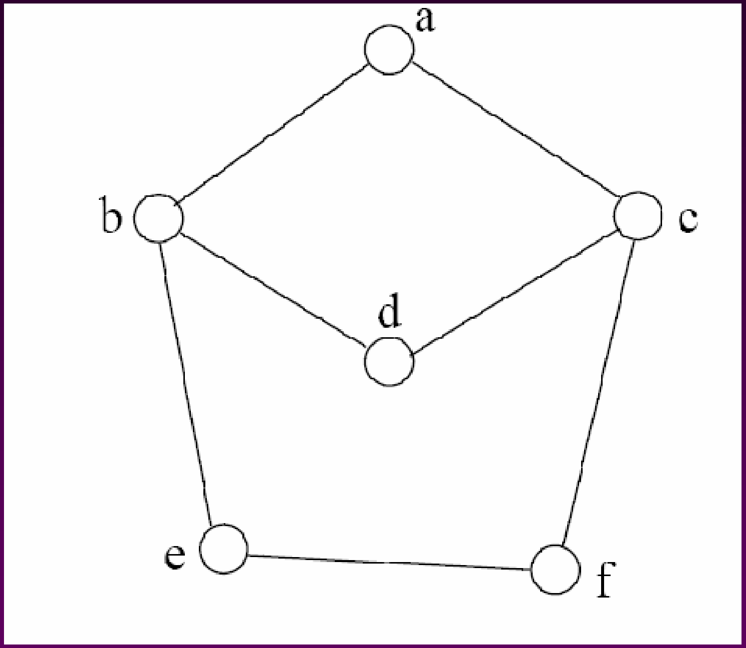
Note: a path is allowed to go through the same vertex or the same edge any number of times!

- The length of a path is the number of edges on the path

Types of paths

- A path is simple if and only if it does not contain a vertex more than once.
- A path is a cycle if and only if $v_0 = v_k$
  - The beginning and end are the same vertex!
- A path contains a cycle as its sub-path if some vertex appears twice or more

Path Examples



Are these paths?

Any cycles?

What is the path's length?

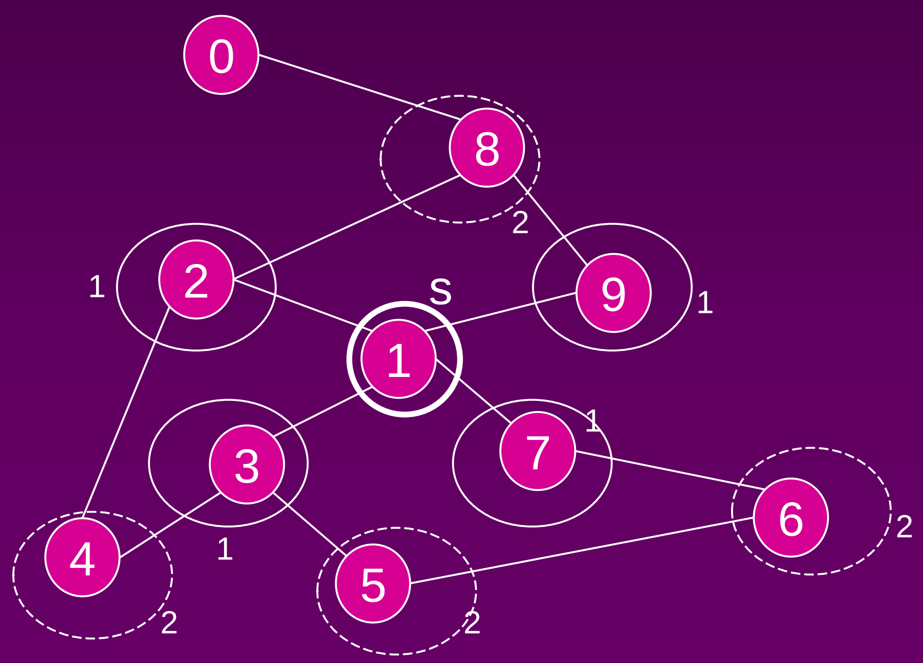
1. {a,c,f,e}
1. {a,b,d,c,f,e}
1. {a, c, d, b, d, c, f, e}
2. {a,c,d,b,a}
1. {a,c,f,e,b,d,c,a}

Graph Traversal

- Application example
  - Given a graph representation and a vertex **s** in the graph
  - Find paths from **s** to other vertices
- Two common graph traversal algorithms
  - Breadth-First Search (BFS)
    - Find the shortest paths in an unweighted graph
  - Depth-First Search (DFS)
    - Topological sort
    - Find strongly connected components

BFS and Shortest Path Problem

- Given any source vertex ***s***, BFS visits the other vertices at increasing distances away from s. In doing so, BFS discovers paths from s to other vertices
- What do we mean by “distance”? The number of edges on a path from s



Example

Consider s=vertex 1

Nodes at distance 1?
2, 3, 7, 9

Nodes at distance 2?
8, 6, 5, 4

Nodes at distance 3?
0

Graph Searching

- Given: a graph G = (V, E), directed or undirected
- Goal: methodically explore every vertex and every edge
- Ultimately: build a tree on the graph
  - Pick a vertex as the root
  - Choose certain edges to produce a tree
  - Note: might also build a *forest* if graph is not connected

Breadth-First Search

- "Explore" a graph, turning it into a tree
  - One vertex at a time
  - Expand frontier of explored vertices across the *breadth* of the frontier
- Builds a tree over the graph
  - Pick a *source vertex* to be the root
  - Find ("discover") its children, then their children, etc.

Breadth-First Search

- Every vertex of a graph contains a color at every moment:
  - White vertices have not been discovered
    - All vertices start with white initially
  - Grey vertices are discovered but not fully explored
    - They may be adjacent to white vertices
  - Black vertices are discovered and fully explored
    - They are adjacent only to black and gray vertices
- Explore vertices by scanning adjacency list of grey vertices

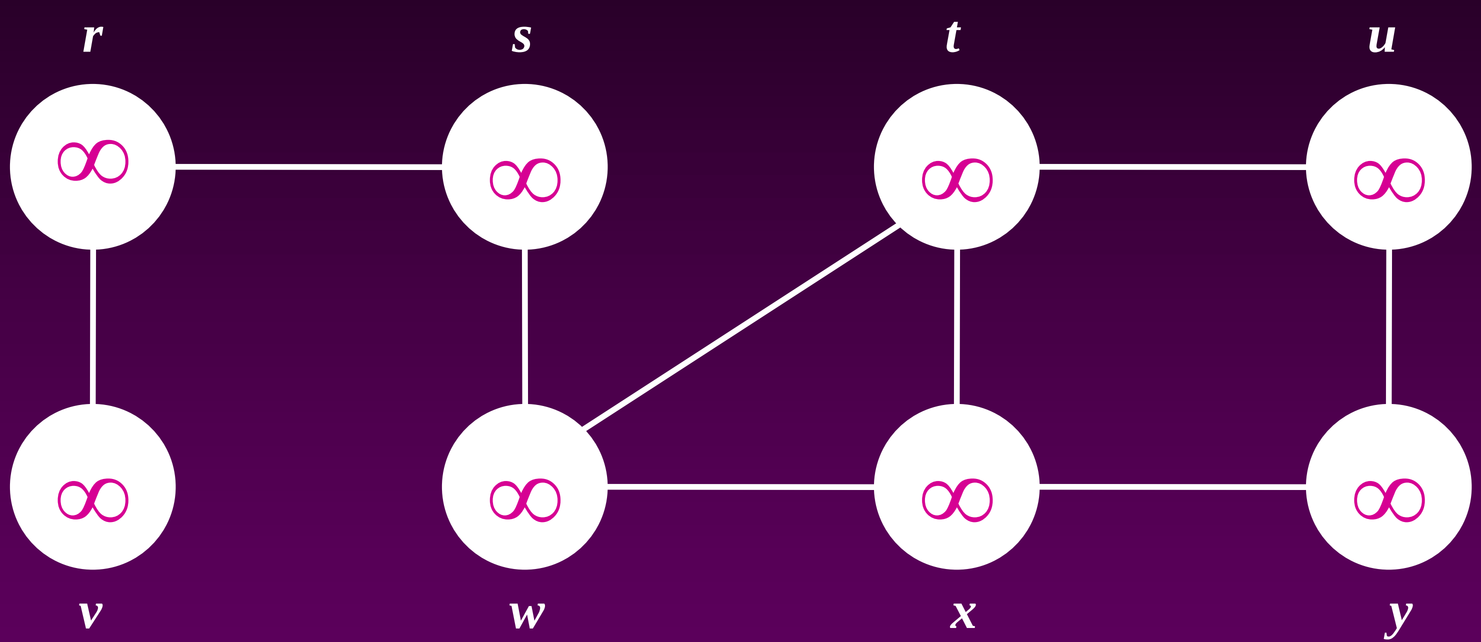
Breadth-First Search: The Code

```
Data: color[V], prev[V],d[V]

BFS(G) // starts from here
{
    for each vertex u ∈ V-
   {s}
    {
        color[u]=WHITE;
       prev[u]=NIL;
       d[u]=inf;
    }
    color[s]=GRAY;
   d[s]=0; prev[s]=NIL;
   Q=empty;
   ENQUEUE(Q,s);
```

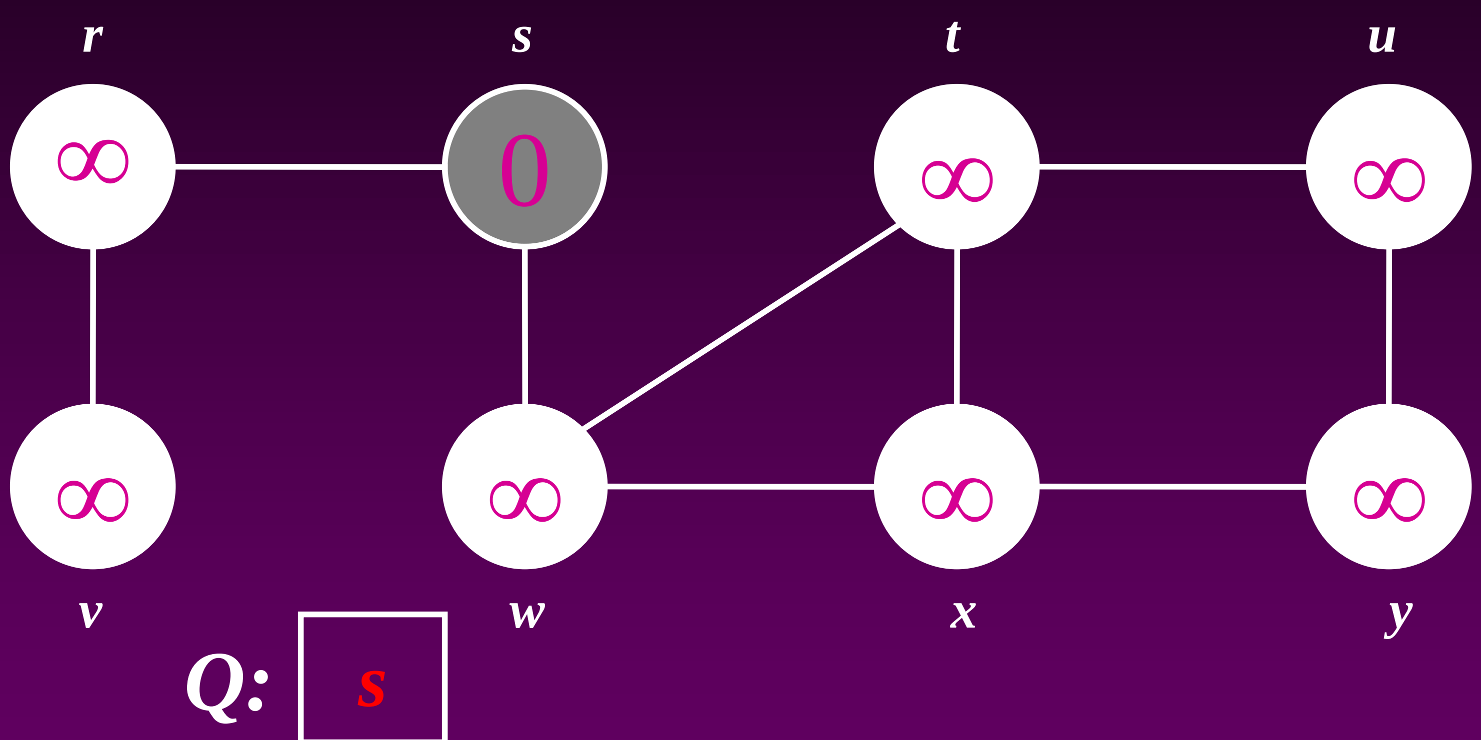
```
While(Q not empty)
{
   u = DEQUEUE(Q);
   for each v ∈ adj[u]{
     if (color[v] ==
  WHITE){
         color[v] = GREY;
         d[v] = d[u] + 1;
         prev[v] = u;
         Enqueue(Q, v);
     }
   }
   color[u] = BLACK;
 }
}
```

Breadth-First Search: Example



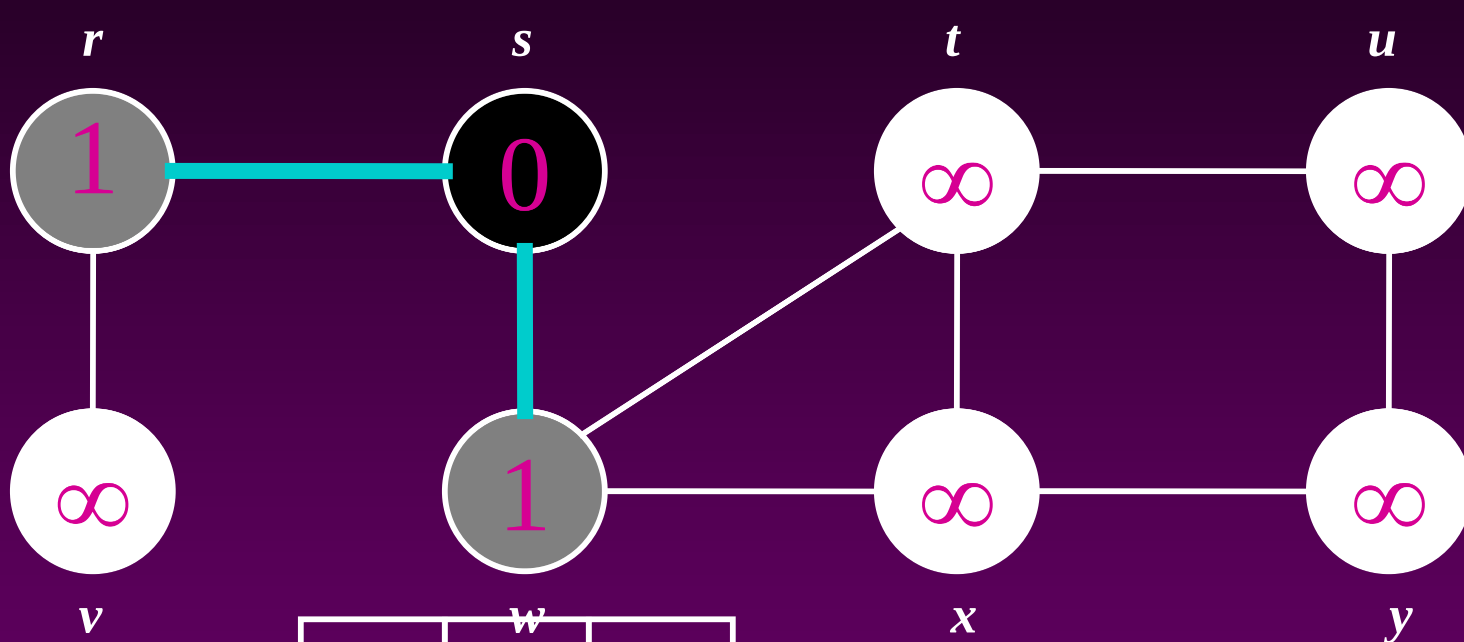
Vertex	r	s	t	u	v	w	x	y
color	W	W	W	W	W	W	W	W
d	∞	∞	∞	∞	∞	∞	∞	∞

Breadth-First Search: Example



vertex	r	s	t	u	v	w	x	y
Color	W	G	W	W	W	W	W	W
d	∞	0	∞	∞	∞	∞	∞	∞

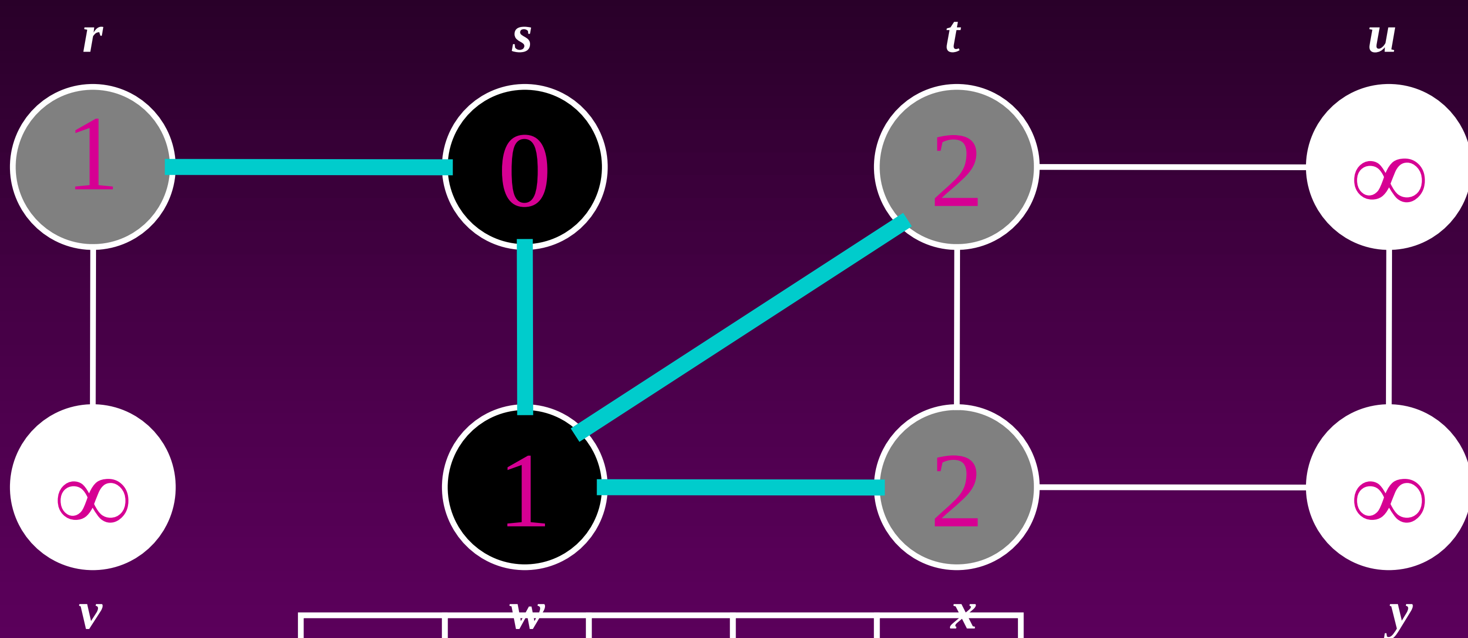
Breadth-First Search: Example



Q: | *s* | *w* | *r* |

vertex	r	s	t	u	v	w	x	y
Color	G	B	W	W	W	G	W	W
d	1	0	∞	∞	∞	1	∞	∞

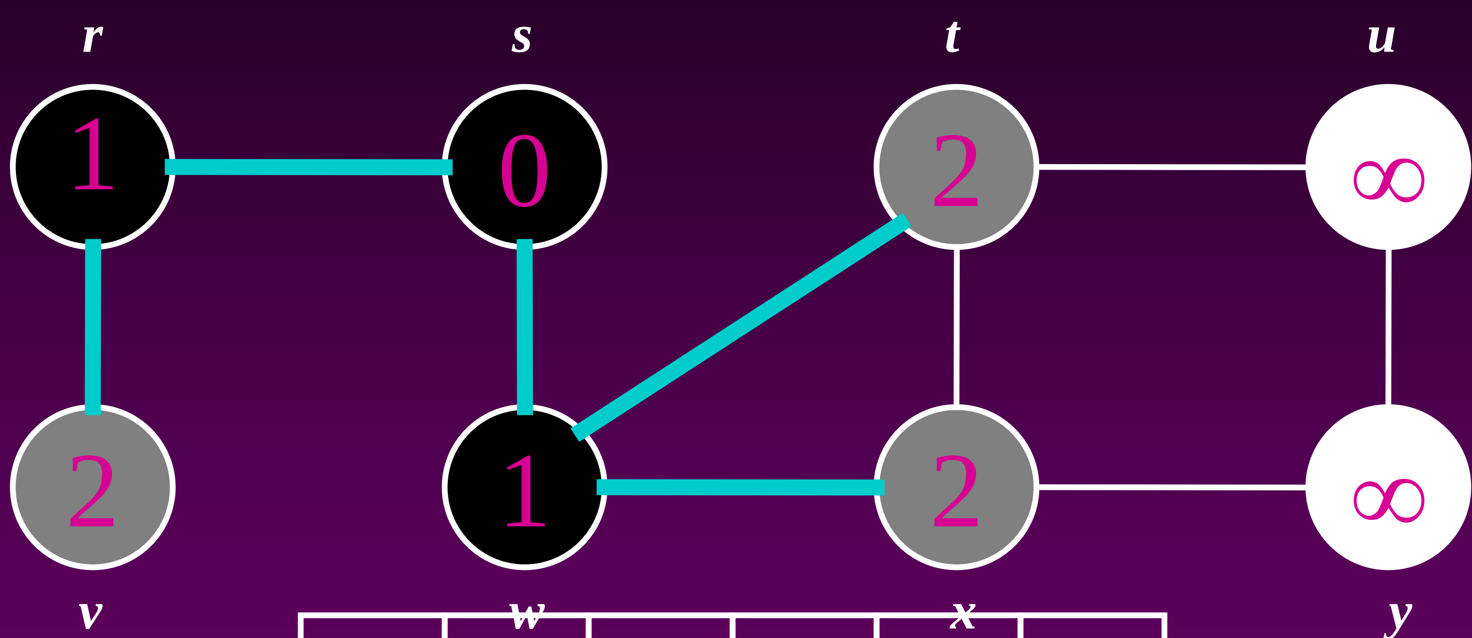
Breadth-First Search: Example



Q: | *s* | *w* | *r* | *t* | *x* |

vertex	r	s	t	u	V	w	X	y
Color	G	B	**G**	W	W	**B**	**G**	W
d	1	0	**2**	∞	∞	**1**	**2**	∞

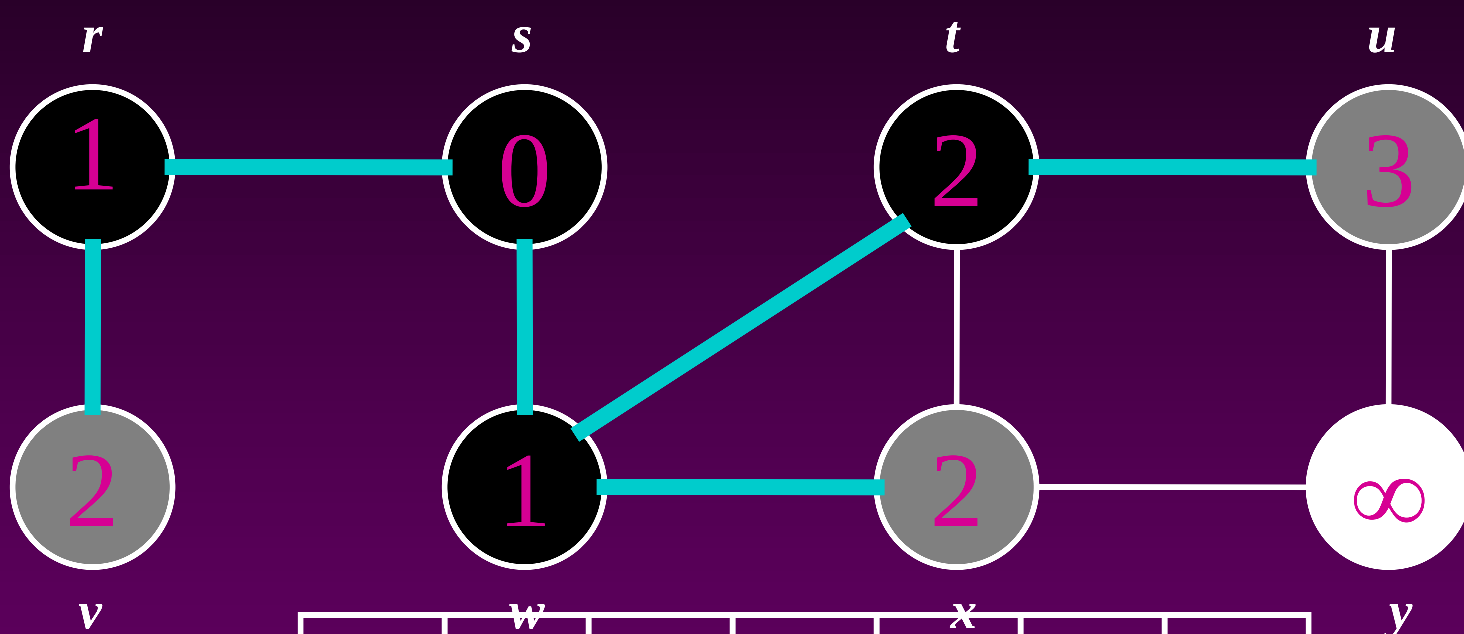
Breadth-First Search: Example



Q: | *s* | *w* | *r* | *t* | *x* | *v* |

vertex	r	s	t	u	v	w	x	y
Color	**B**	B	G	W	**G**	B	G	W
d	**1**	0	2	∞	**2**	1	2	∞

Breadth-First Search: Example

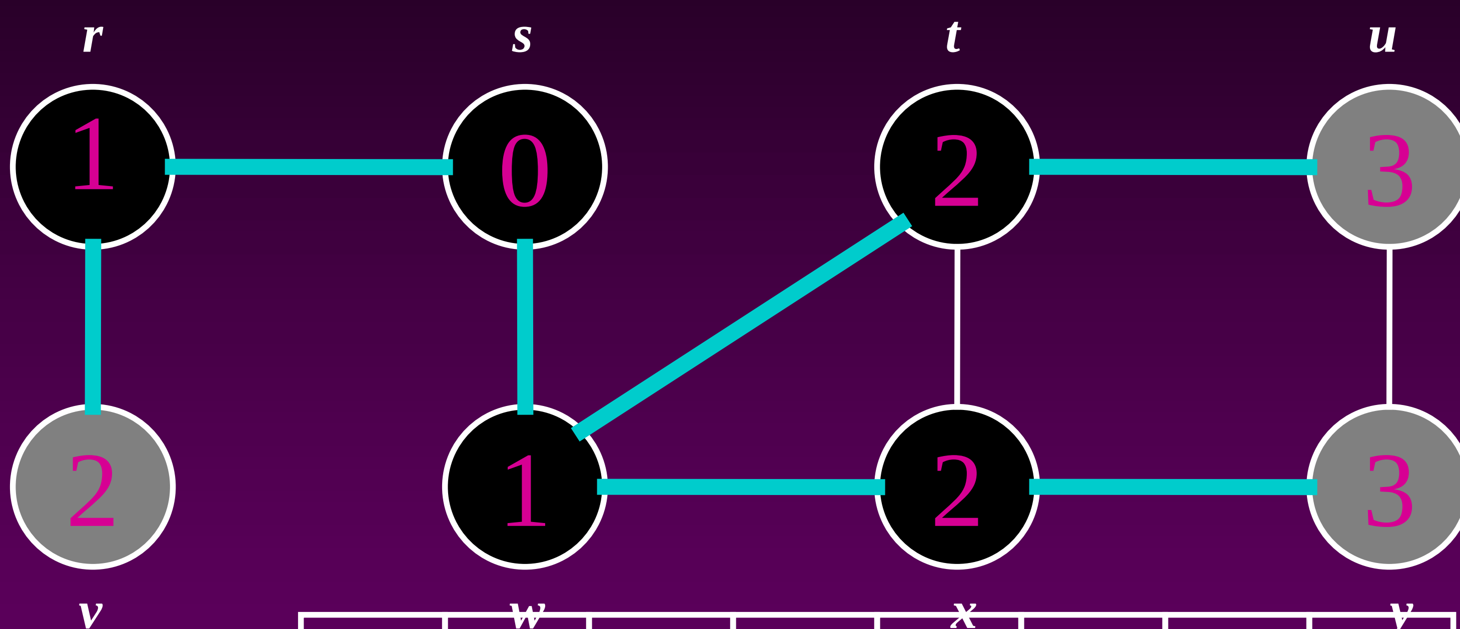


Q:

s	*w*	*r*	*t*	*x*	*v*	*u*

vertex	r	s	t	u	v	w	x	y
Color	B	B	**B**	**G**	G	B	G	W
d	1	0	**2**	**3**	2	1	2	∞

Breadth-First Search: Example

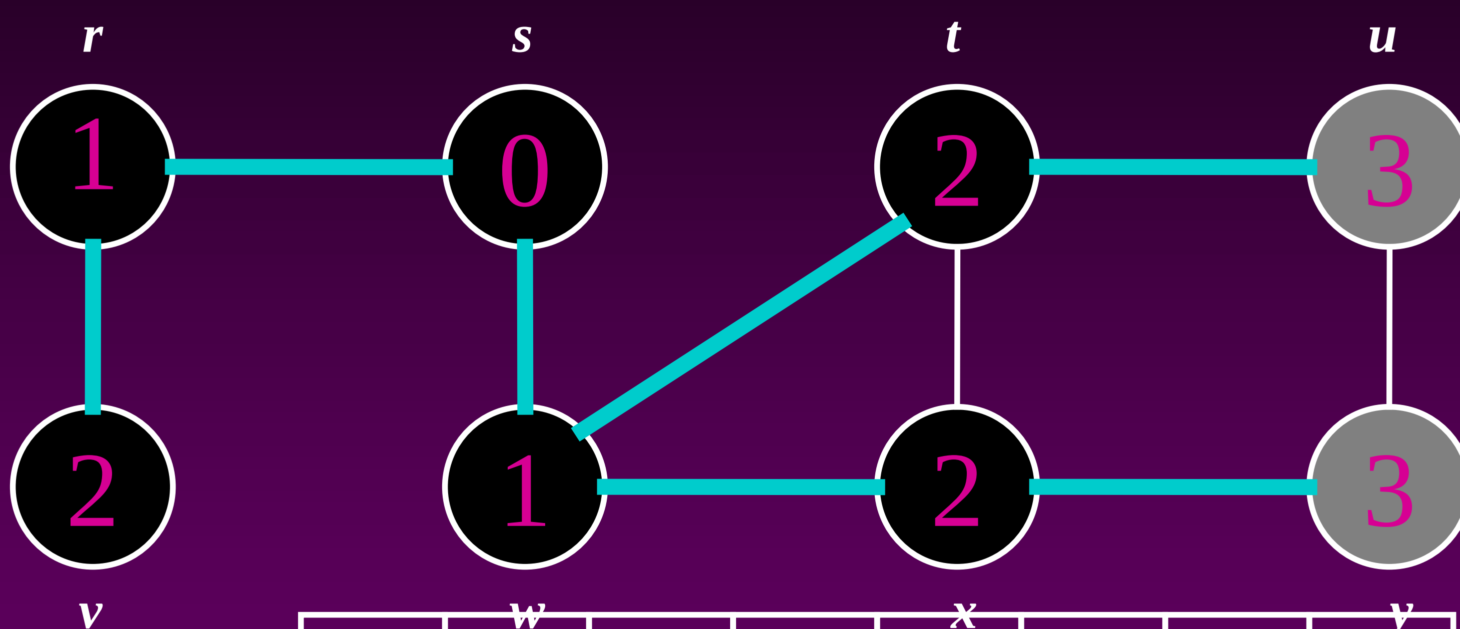


Q:

s	*w*	*r*	*t*	*x*	*v*	*u*	*y*

vertex	r	s	t	u	v	w	x	y
Color	B	B	B	G	G	B	**B**	**G**
d	1	0	2	3	2	1	**2**	**3**
prev	s	nil	w	t	r	s	**w**	**x**

Breadth-First Search: Example

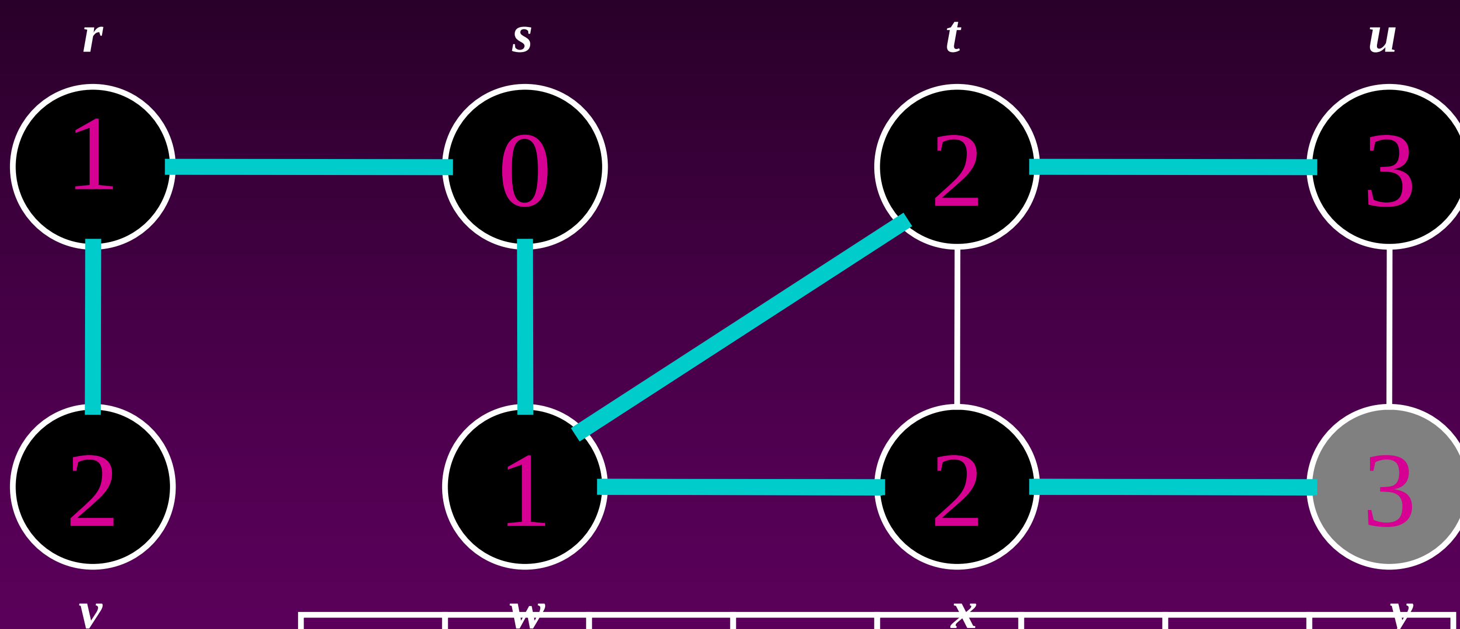


Q:

s	*w*	*r*	*t*	*x*	*v*	*u*	*y*

vertex	r	s	t	u	v	w	x	y
Color	B	B	B	G	**B**	B	B	G
d	1	0	2	3	**2**	1	2	3
prev	s	nil	w	t	**r**	s	w	x

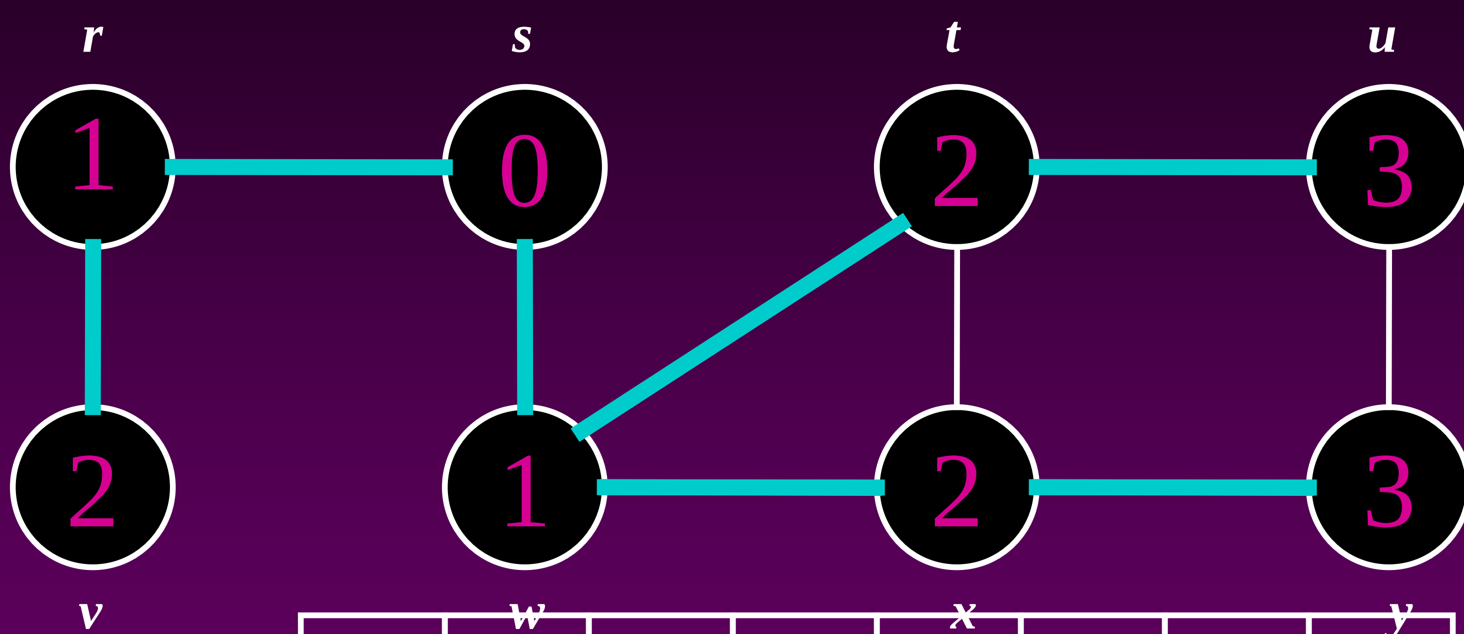
Breadth-First Search: Example



Q: | s | w | r | t | x | v | u | y |

vertex	r	s	t	u	v	w	x	y
Color	B	B	B	**B**	B	B	B	G
d	1	0	2	**3**	2	1	2	3
prev	s	nil	w	**t**	r	s	w	x

Breadth-First Search: Example



Q:								
	s	*w*	*r*	*t*	*x*	*v*	*u*	*y*

vertex	r	s	t	u	v	w	x	y
Color	B	B	B	G	B	B	B	**B**
d	1	0	2	3	2	1	2	**3**
prev	s	nil	w	t	r	s	w	**x**

BFS: The Code (again)

```
Data: color[V], prev[V],d[V]

BFS(G) // starts from here
{
    for each vertex u ∈ V-
   {s}
    {
        color[u]=WHITE;
       prev[u]=NIL;
       d[u]=inf;
    }
    color[s]=GRAY;
   d[s]=0; prev[s]=NIL;
   Q=empty;
   ENQUEUE(Q,s);
```

```
While(Q not empty)
{
   u = DEQUEUE(Q);
   for each v ∈ adj[u]{
     if (color[v] ==
  WHITE){
          color[v] = GREY;
          d[v] = d[u] + 1;
          prev[v] = u;
          Enqueue(Q, v);
     }
   }
   color[u] = BLACK;
 }
}
```

Breadth-First Search: Print Path

```
Data: color[V], prev[V],d[V]
```

```
Print-Path(G, s, v)
{
   if(v==s)
      print(s)
    else if(prev[v]==NIL)
      print(No path);
   else{
      Print-Path(G,s,prev[v]);
      print(v);
   }
}
```

Amortized Analysis

- Stack with 3 operations:
  - Push, Pop, Multi-pop
- What will be the complexity if “n” operations are performed?

BFS: Complexity

Data: color[V], prev[V],d[V]

```
BFS(G) // starts from here
{
    for each vertex u ∈ V-
   {s}
    {
        color[u]=WHITE;
        prev[u]=NIL;
        d[u]=inf;
    }
    color[s]=GRAY;
   d[s]=0; prev[s]=NIL;
   Q=empty;
   ENQUEUE(Q,s);
```

O(V)

```
While(Q not empty)
{
   u = DEQUEUE(Q);
   for each v ∈ adj[u]{
    if(color[v] == WHITE){
         color[v] = GREY;
         d[v] = d[u] + 1;
         prev[v] = u;
         Enqueue(Q, v);
     }
   }
   color[u] = BLACK;
 }
}
```

u = every vertex, but only once (Why?)

O(V)

What will be the running time?

Total running time: O(V+E)

Breadth-First Search: Properties

- BFS calculates the *shortest-path distance* to the source node
  - Shortest-path distance $\delta(s,v)$ = minimum number of edges from s to v, or ∞ if v not reachable from s
  - Proof given in the book (p. 472-5)
- BFS builds *breadth-first tree*, in which paths to root represent shortest paths in G
  - Thus can use BFS to calculate shortest path from one vertex to another in O(V+E) time

Application of BFS

- Find the shortest path in an undirected/directed unweighted graph.
- Find the bipartiteness of a graph.
- Find cycle in a graph.
- Find the connectedness of a graph.

Books

- Cormen – Chapter 22 – elementary Graph Algorithms
- Exercise you have to solve:
  - 22.1-5 (Square)
  - 22.1-6 (Universal Sink)
  - 22.2-6 (Wrestler)
  - 22.2-7 (Diameter)
  - 22.2-8 (Traverse)