CSE2202
Design and Analysis of Algorithms-I

Md. Fahim Arefin

Google classroom - **m32q43s**

Md. Fahim Arefin

Reach me at fahim@cse.du.ac.bd

Quickly accessible drive link: **a.farefin.com**

How to Learn

I’d like **you to have more control over**

- how you’re learning,
- what you’re learning,
- when you’re learning,
- where you’re learning.

So no marks on attendance !

How I Teach

- You set your **own goals**,
- build on your **own interests**,
- express your **own ideas**,
- develop your **own strategies**, and
- feel a sense of control and ownership over your **own learning.**

So there will be lots of peer learning.

Recap: Data Structure

Mark Distribution

1. Group Assignment : 10 Marks
2. In course : 20 Marks
3. Final : 70 marks

Clarifications

1. In this course, the most important rule is –

   TRY YOURSELF.

2. You will be working in Java in this course.
3. Form a pair within the first week.
4. Only soft copies of assignments

Advice!

1. Don’t worry too much if you don’t like competitive programming
2. Life is a marathon, don’t worry about other people.
3. It’s about the journey, not the destination
4. Always try to think how you use algorithms in your own life

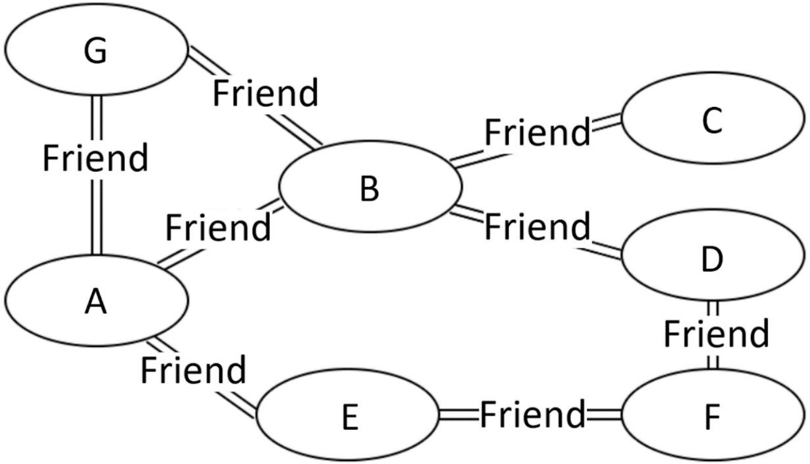
The man who asks a question is a fool for a minute, the man who does not ask is a fool for life. - Confucius

Recommended Textbooks

- **Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C., 2022. Introduction to algorithms. MIT press.**
- **Goodrich, M.T., Tamassia, R. and Goldwasser, M.H., 2013. Data structures and algorithms in Python. John Wiley & Sons Ltd.**

Graph

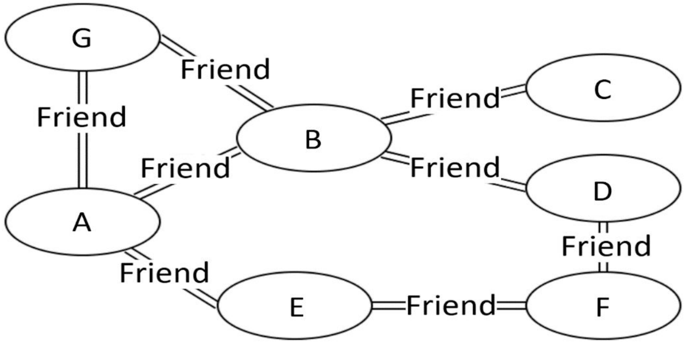
- Graph is probably the data structure that has the closest resemblance to our daily life.
- There are many types of graphs describing the relationships in real life.

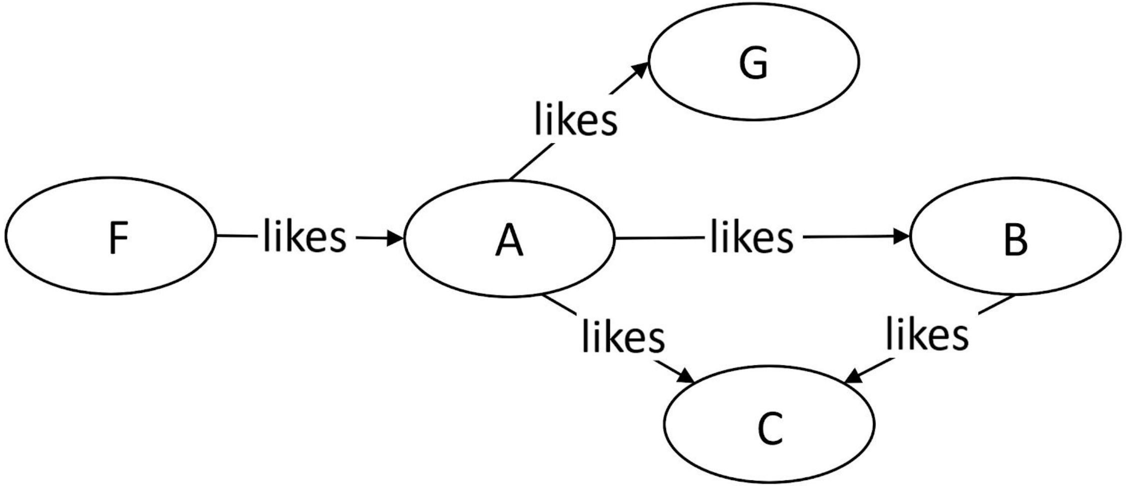


Graph Variations

- 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

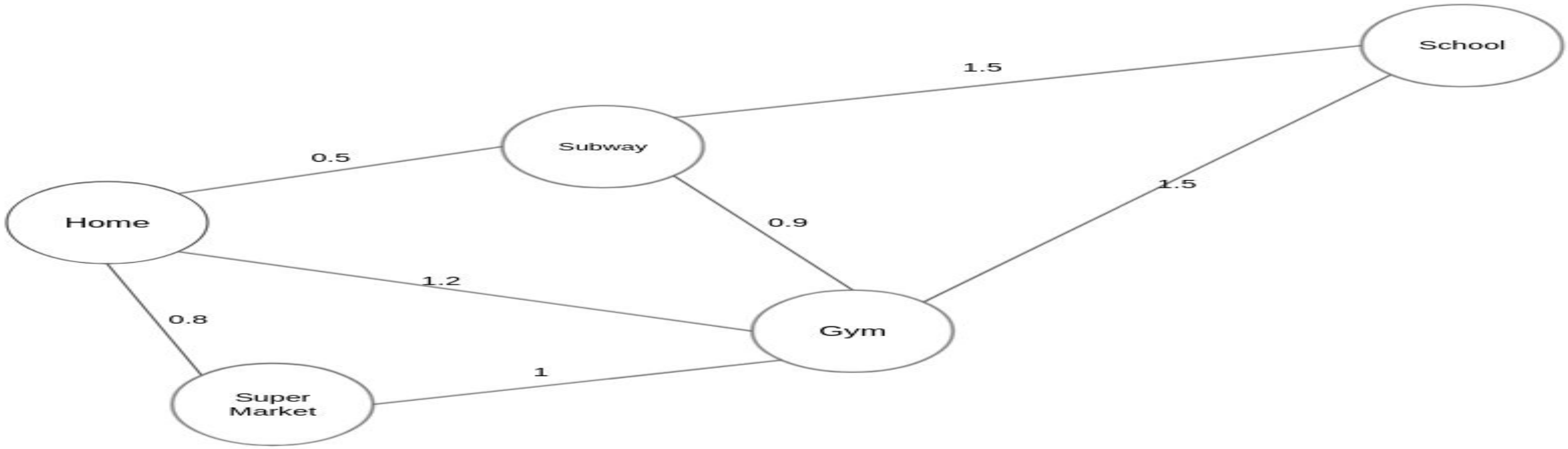
Graph Variations





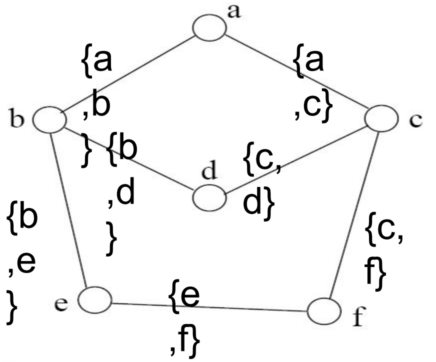
Graph Variations

- 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)



Graph – 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.



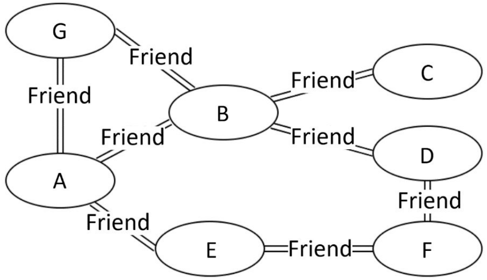
$$V = \{a, b, c, d, e, f\}$$

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

An undirected graph

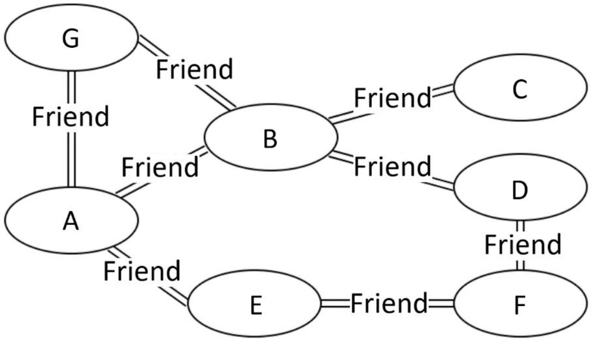
Definition

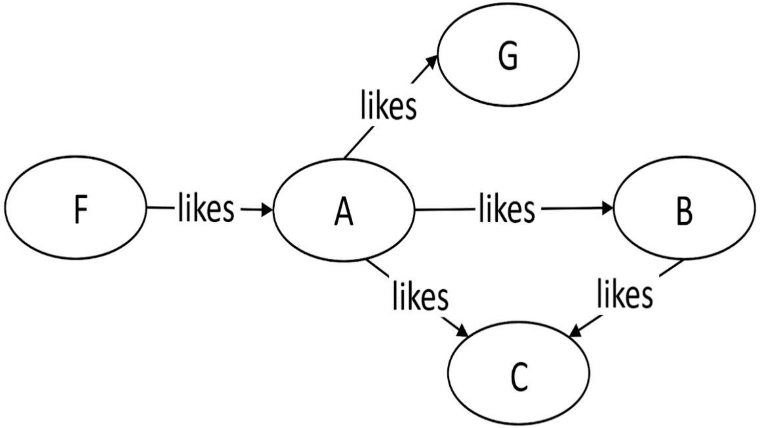
- **Path:** the sequence of vertices to go through from one vertex to another.
  - a path from A to C is [A, B, C], or [A, G, B, C], or [A, E, F, D, B, C].
- **Path Length:** the number of edges in a path.
- **Cycle:** a path where the starting point and endpoint are the same vertex.
  - [A, B, D, F, E] forms a cycle. Similarly, [A, G, B] forms another cycle.



Definition

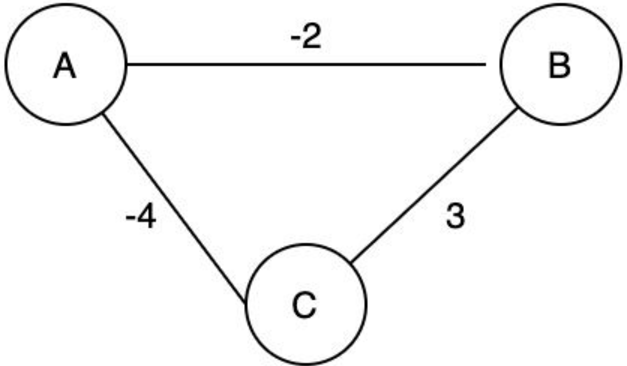
- **Degree of a Vertex:** the term “degree” applies to unweighted graphs. The degree of a vertex is the number of edges connecting the vertex.
  - the degree of vertex A is 3 because three edges are connecting it.
- **In-Degree:** “in-degree” is a concept in directed graphs. If the in-degree of a vertex is d, there are d directional edges incident to the vertex.
  - In Figure 2, A’s indegree is 1, i.e., the edge from F to A.
- **Out-Degree:** “out-degree” is a concept in directed graphs. If the
  is d, t n the
  e edges





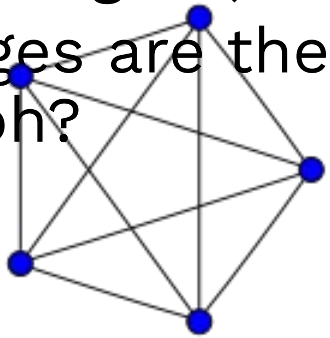
Definition

- **Connectivity**: if there exists at least one path between two vertices, these two vertices are connected.
  - A and C are connected because there is at least one path connecting them.
- **Negative Weight Cycle:** In a “weighted graph”, if the sum of the weights of all edges of a cycle is a negative value, it is a negative weight cycle.
  - In the Figure the sum of weights is -3



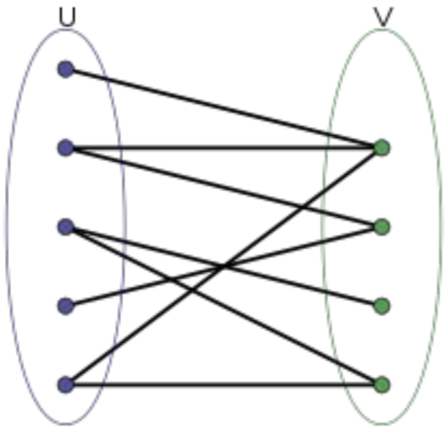
Definition

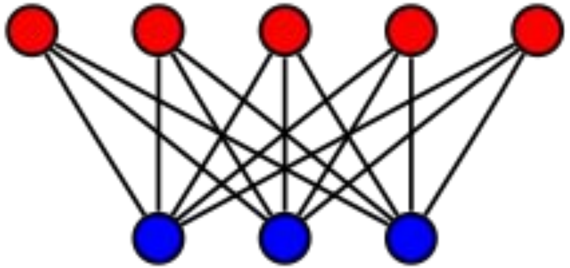
- Complete Graph
  - a complete graph is a simple undirected graph in which every pair of distinct vertices is connected by a unique edge.
  - A complete digraph is a directed graph in which every pair of distinct vertices is connected by a pair of unique edges (one in each direction).
  - How many edges are there in an N-vertex complete graph?



Definition

- Bipartite Graph
- a bipartite graph (or bigraph) is a graph whose vertices can be divided into two disjoint and independent sets





Definition

- 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.

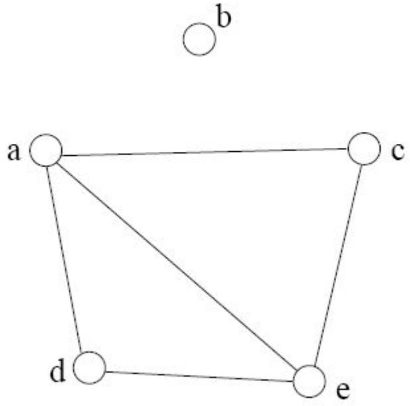
1. Adjacency Matrix

   Use a 2D matrix to represent the graph

2. Adjacency List

   Use a 1D array of linked lists

Adjacency Matrix

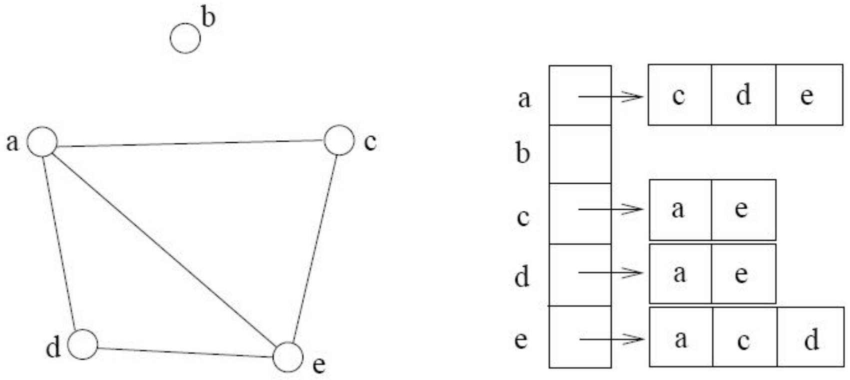


	a	b	c	d	e
a	0	0	1	1	1
b	0	0	0	0	0
c	1	0	0	0	1
d	1	0	0	0	1
e	1	0	1	1	0

Simple Questions on Adjacency Matrix

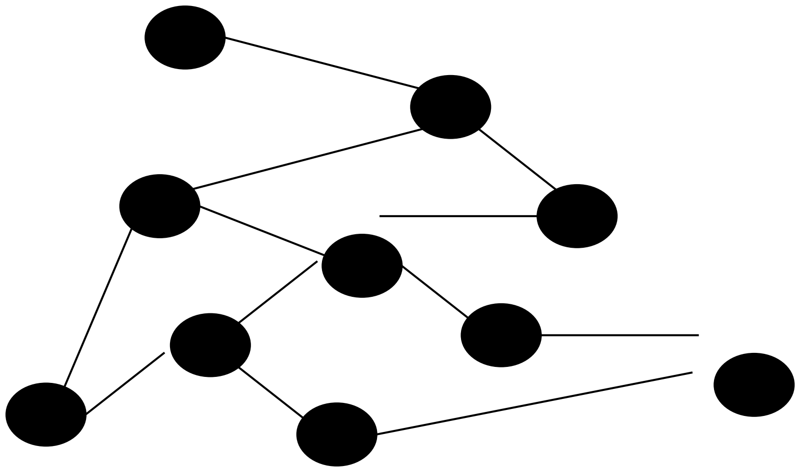
- Is there a direct link between A and B?
- What is the indegree and outdegree for a vertex A?
- How many nodes are directly connected to vertex A?
- Is it an undirected graph or directed graph?
- Suppose ADJ is an NxN matrix. What will be the result if we create another matrix ADJ2 where ADJ2=ADJxADJ?

Adjacency List



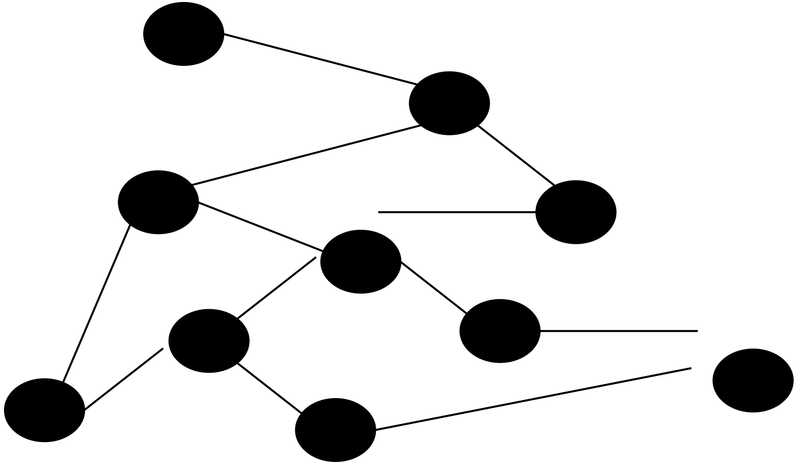
- If the graph is not dense, in other words, sparse, a better solution is an adjacency list

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



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

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

Storage of Adjacency List

- 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

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

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

Adjacency List vs. Matrix

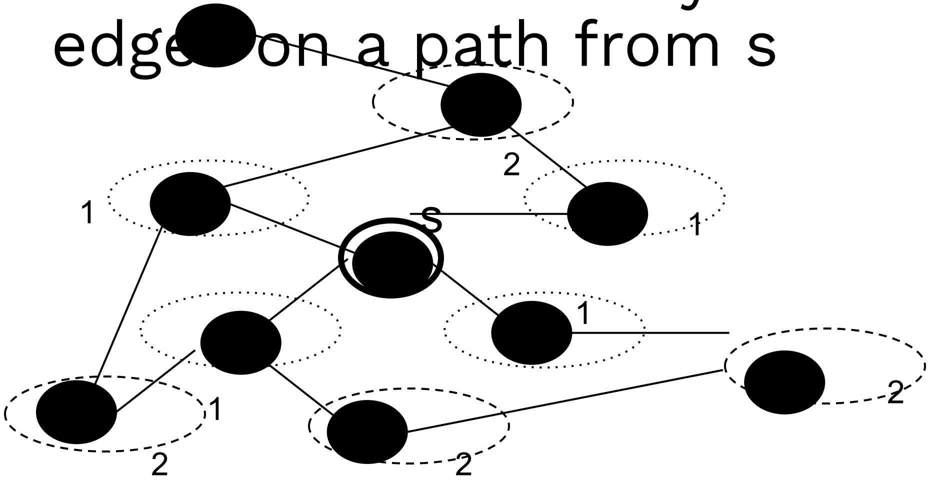
- **Adjacency List**
  - More compact than adjacency matrices if graph has few edges
  - Requires more time to find if an edge exists
- **Adjacency Matrix**
  - Always require n^2 space
    - This can waste a lot of space if the number of edges are sparse
  - Can quickly find if an edge exists

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

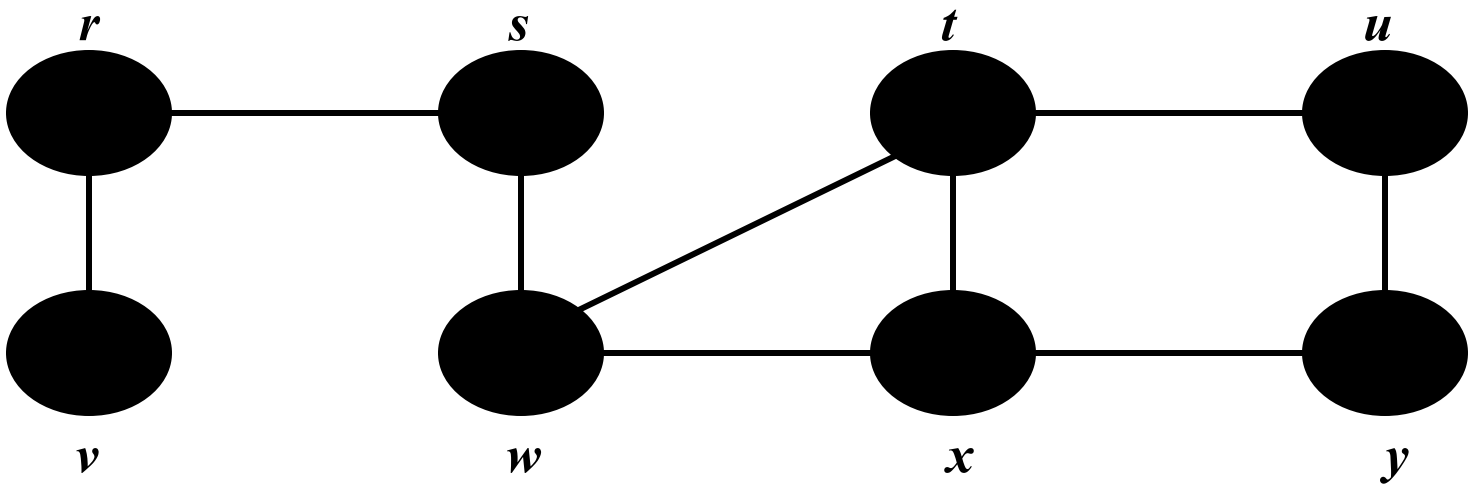
Breadth-First Search: The Code

```
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);
```

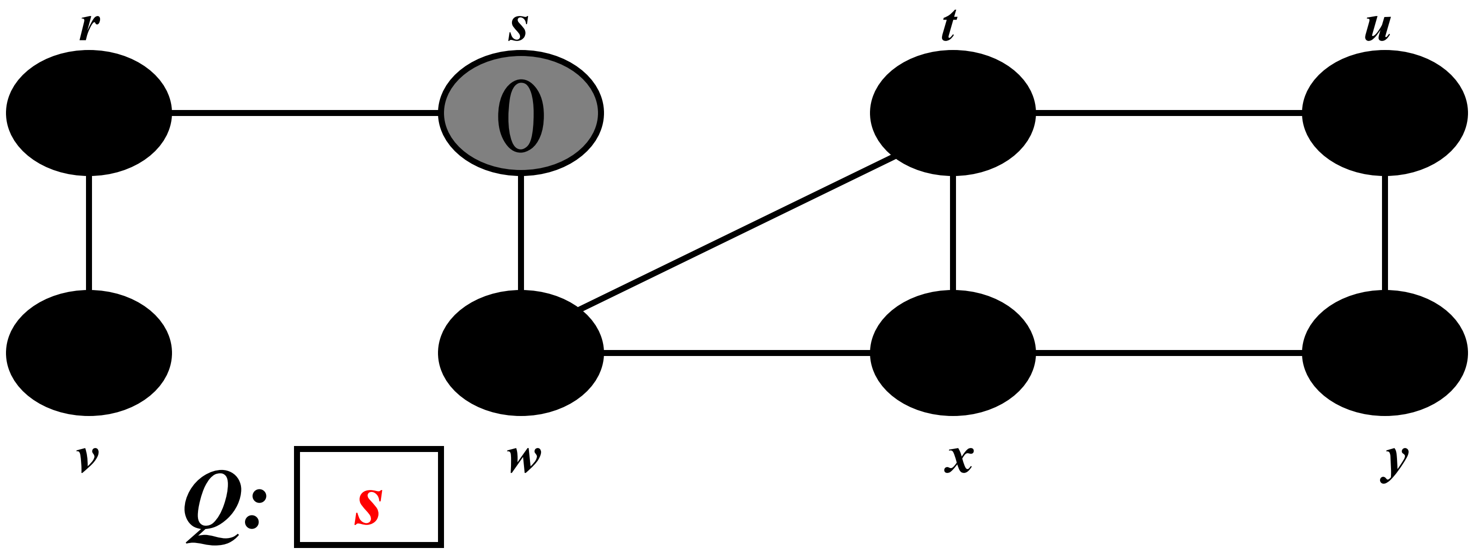
```
While(Q not empty)
{
  u = DEQUEUE(Q);
  for each v ∈ adj[u]{
    if (color[v] == WHITE){
        color[v] = GREY;
        d[v] = d[u] + 1;
        prev[v] = u;
        Enqueue(Q, v);
    }
  }
  color[u] = BLACK;
}
}
```

Breadth-First Search: Example



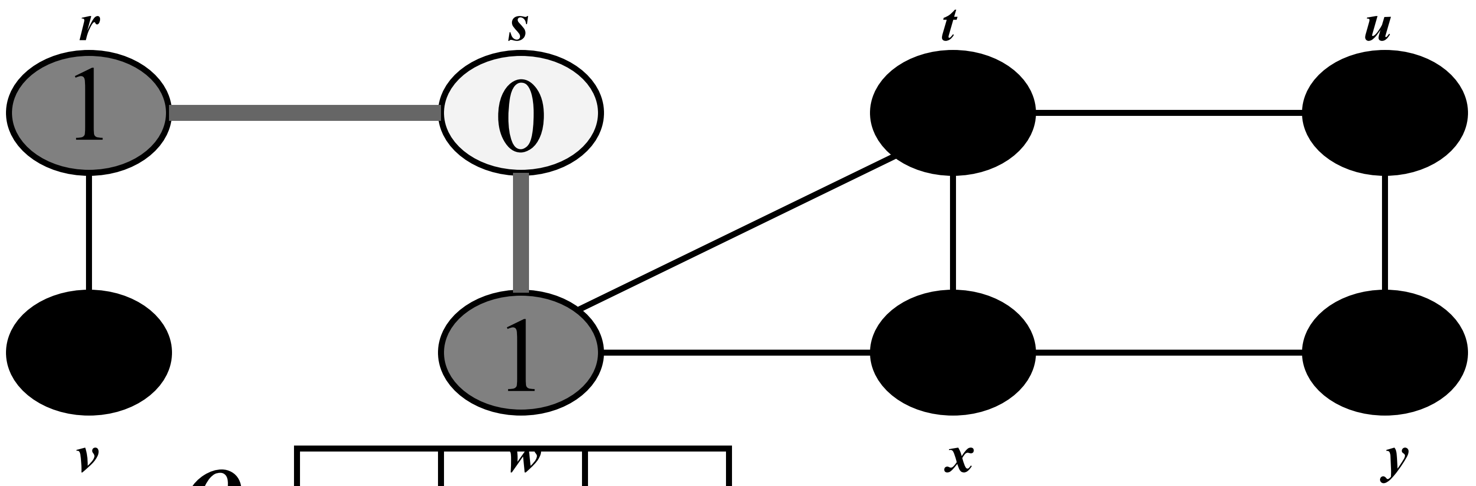
Vertex	r	s	t	u	v	w	x	y
color	W	W	W	W	W	W	W	W
d	∞	∞	∞	∞	∞	∞	∞	∞
prev	nil	nil	nil	nil	nil	nil	nil	nil

Breadth-First Search: Example



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

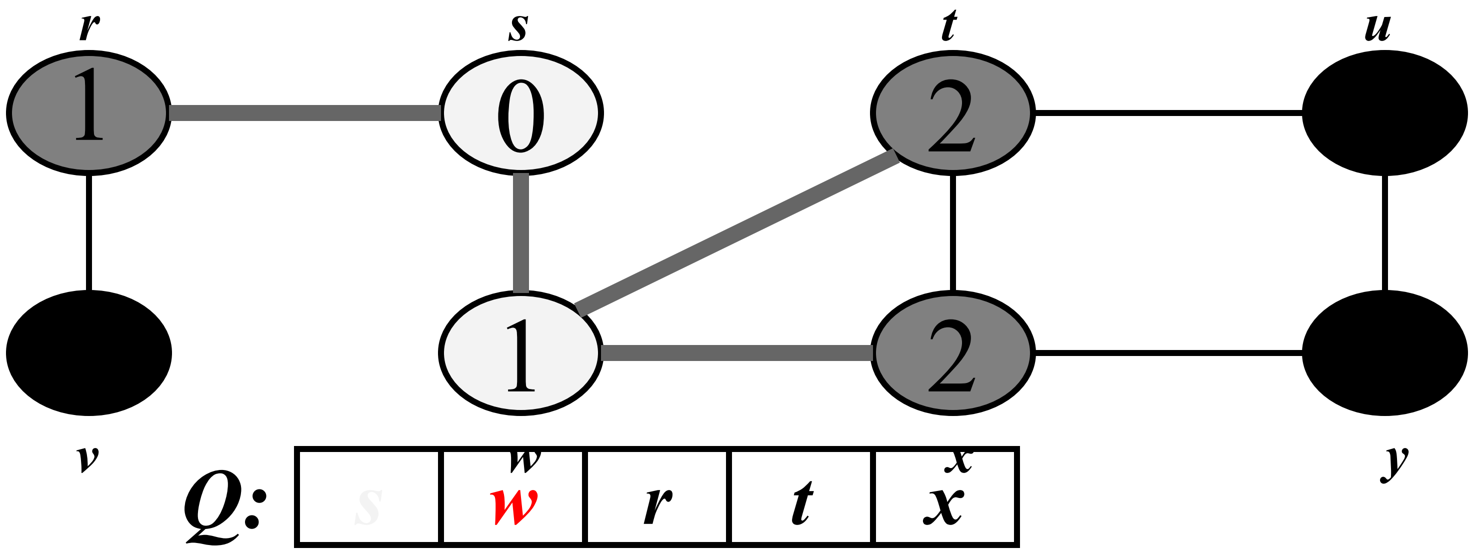
Breadth-First Search: Example



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

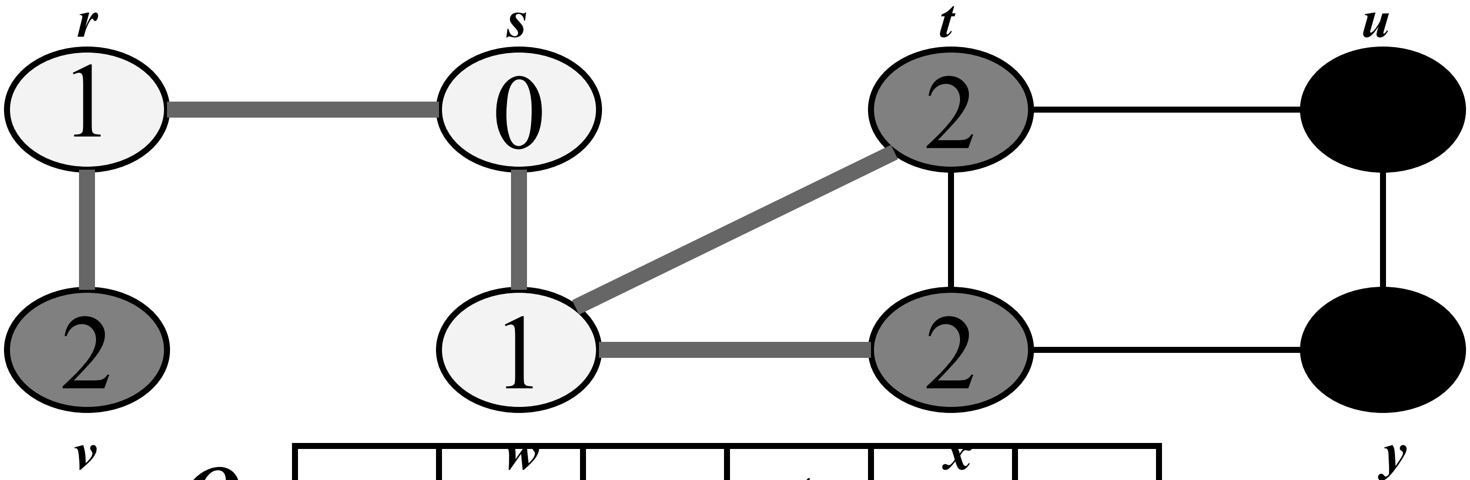
vertex	r	s	t	u	v	w	x	y
Color	G	B	W	W	W	G	W	W
d	1	0	∞	∞	∞	1	∞	∞
prev	s	nil	nil	nil	nil	s	nil	nil

Breadth-First Search: Example



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

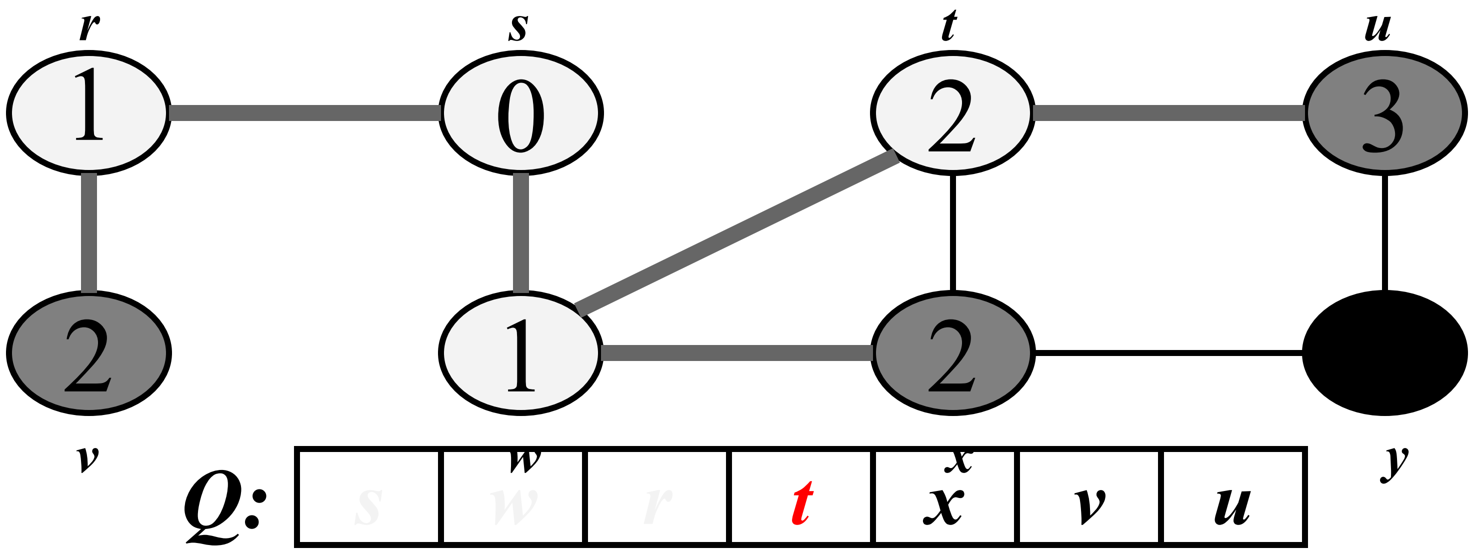
Breadth-First Search: Example



Q: s w r t x v

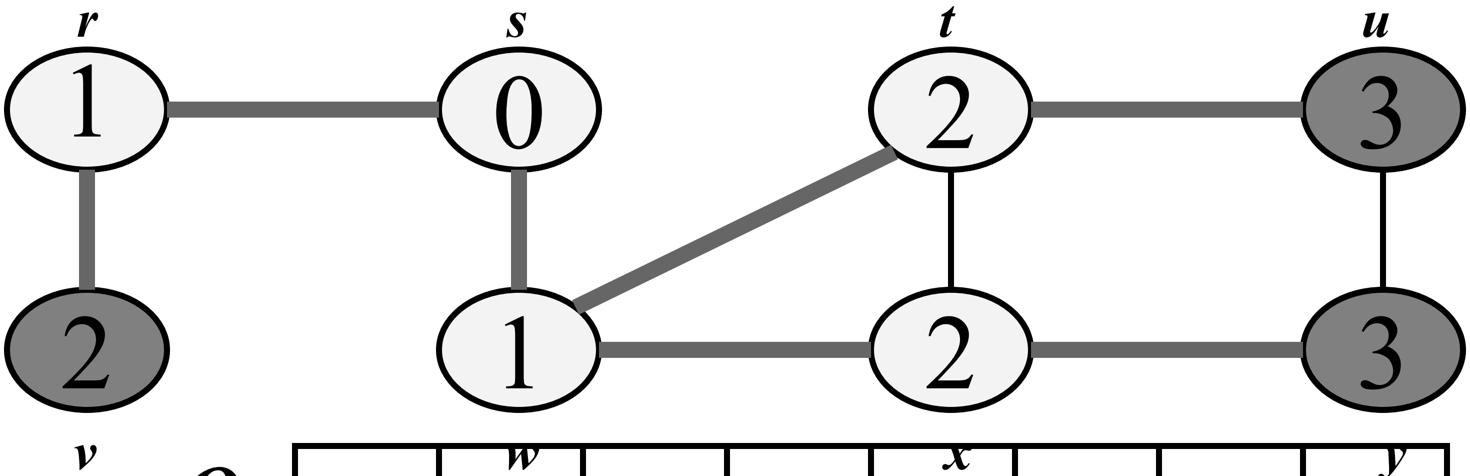
vertex	r	s	t	u	v	w	x	y
Color	**B**	B	G	W	**G**	B	G	W
d	**1**	0	2	∞	**2**	1	2	∞
prev	**s**	nil	w	nil	**r**	s	w	nil

Breadth-First Search: Example



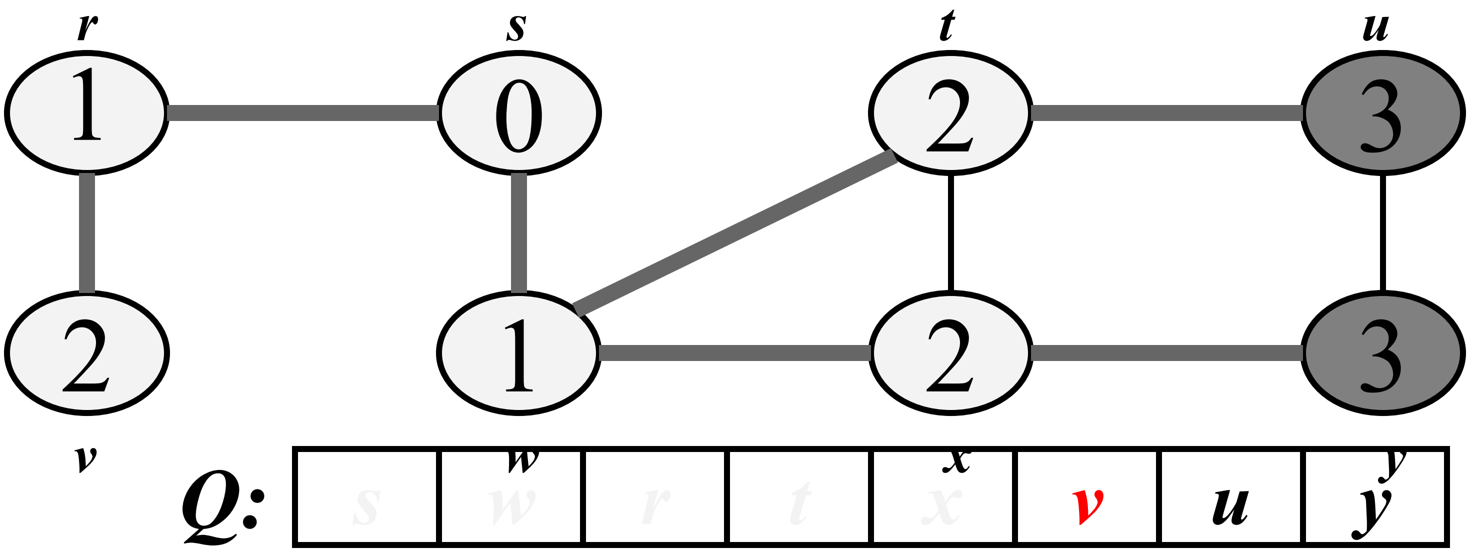
vertex	r	s	t	u	v	w	x	y
Color	B	B	**B**	**G**	G	B	G	W
d	1	0	**2**	**3**	2	1	2	∞
prev	s	nil	**w**	**t**	r	s	w	nil

Breadth-First Search: Example



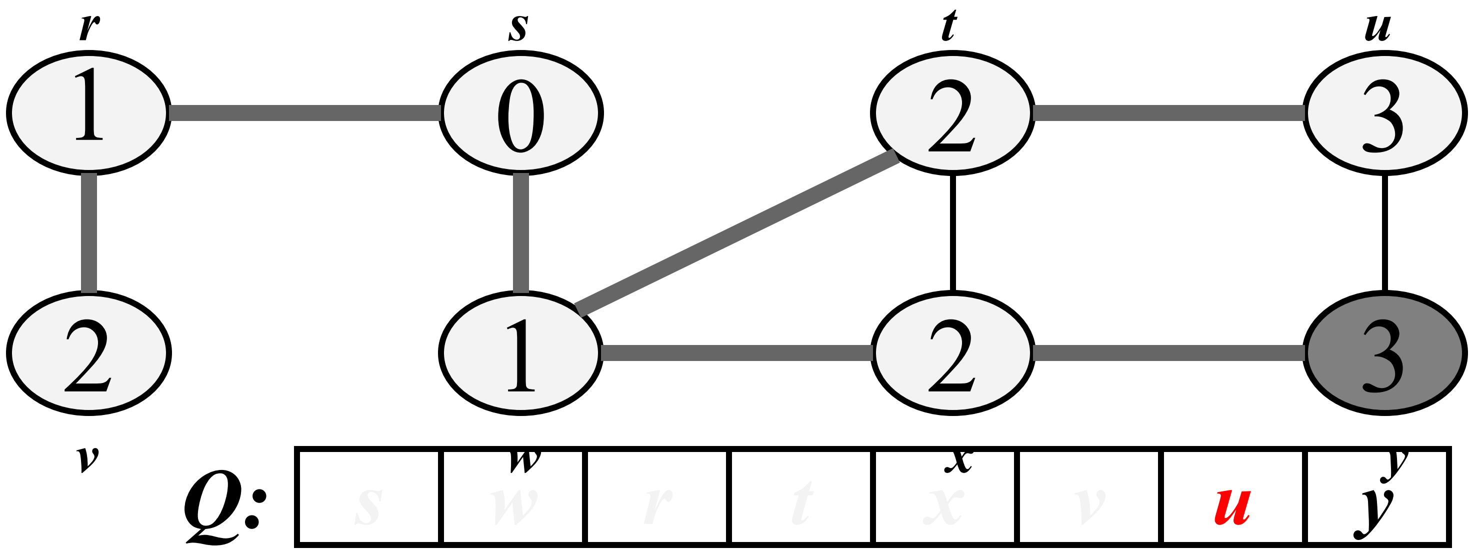
vertex	r	s	t	u	v	w	x	y
Color	B	B	B	G	G	B	**B**	**G**
d	1	0	2	3	2	1	**2**	**3**
prev	s	nil	w	t	r	s	**w**	**x**

Breadth-First Search: Example



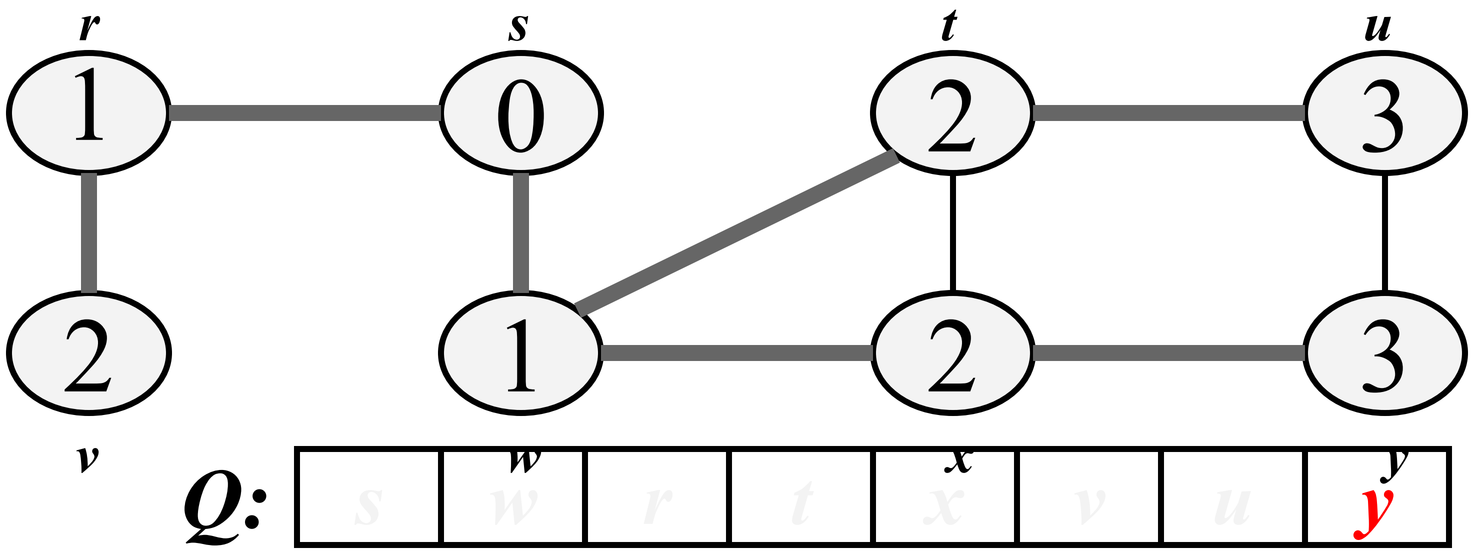
vertex	r	s	t	u	v	w	x	y
Color	B	B	B	G	**B**	B	B	G
d	1	0	2	3	**2**	1	2	3
prev	s	nil	w	t	**r**	s	w	x

Breadth-First Search: Example



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



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);
   }
}
```

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 δ(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 bipartite-ness of a graph.
- Find cycle in a graph.
- Find the connectedness of a graph.
- And many more.

Exercises on BFS

- CLRS – Chapter 22 – elementary Graph Algorithms
- Exercise you have to solve: (Page 602)
  - 22.2-7 (Wrestler)
  - 22.2-8 (Diameter)
  - 22.2-9 (Traverse)

Thanks!

Any questions?

You can find us at

- fahim@cse.du.ac.bd