CS590/CPE590

Graphs | Graph Algorithms

Kazi Lutful Kabir

Spring 2023

Graphs

- Trees are limited in that a data structure can only have one parent
- Graphs overcome this limitation
- Graphs were being studied long before computers were invented
- Graphs algorithms run
  - large communication networks
  - the software that makes the Internet function
  - programs to determine optimal placement of components on a silicon chip
- Graphs describe
  - roads maps
  - airline routes
  - course prerequisites

Graph Applications

- Graphs can be used to:
  - determine if one node in a network is connected to all the others
  - map out multiple course prerequisites (a solution exists if the graph is a directed graph with no cycles)
  - find the shortest route from one city to another (least cost or shortest path in a weighted graph)
  - Scheduling problems (such as: class scheduling, exam scheduling)

Graph

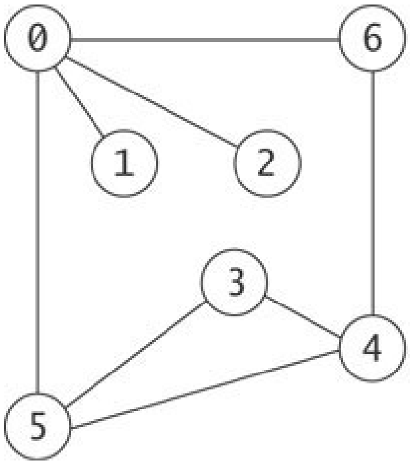
- A graph is a set of vertices and edges: **G = <V, E>**
- Edges indicate that there is some form of connection between a pair of vertices.
  - An edge can be represented as a pair of vertices: **$e = \langle v_1, v_2 \rangle$**
- Graphs can be weighted (each edge would have a weight associated with it) or unweighted.

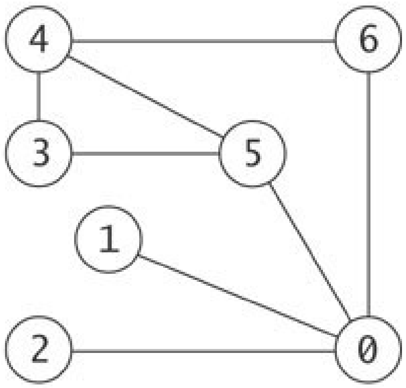
Definition

- Graphs can be directed (meaning each edge might have an arrow indicating the flow) or undirected
  - For a given edge, **$e = <v_1, v_2>$**, the order of the vertices v_1 and v_2 in the pair may or may not matter
  - If **$e = <v_1, v_2>$** is an edge from v_1 to v_2 then **v_2 is said to be adjacent to v_1**
  - The degree of a vertex is the number of edges adjacent to it.
- Example: a map with one vertex for each city and one edge for each inter-city road
  - Edge weights represent distances
  - Edges are unidirectional to represent one-way roads
  - Two-way roads are represented by a pair of edges

Visual Representation of Graphs

- The physical layout of the vertices and their labeling is not relevant

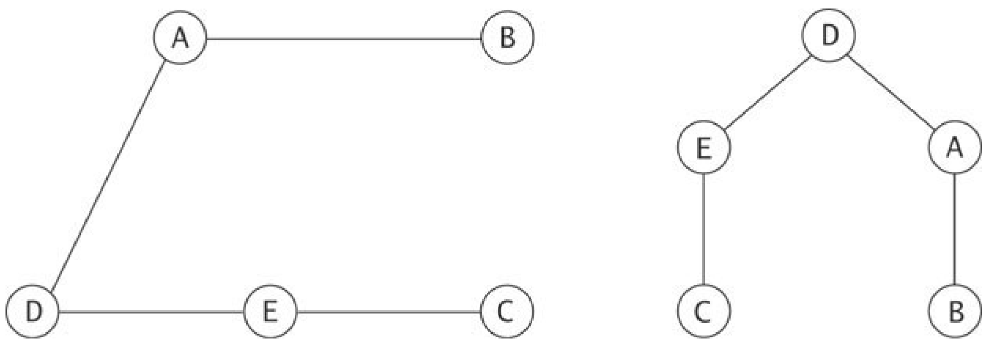




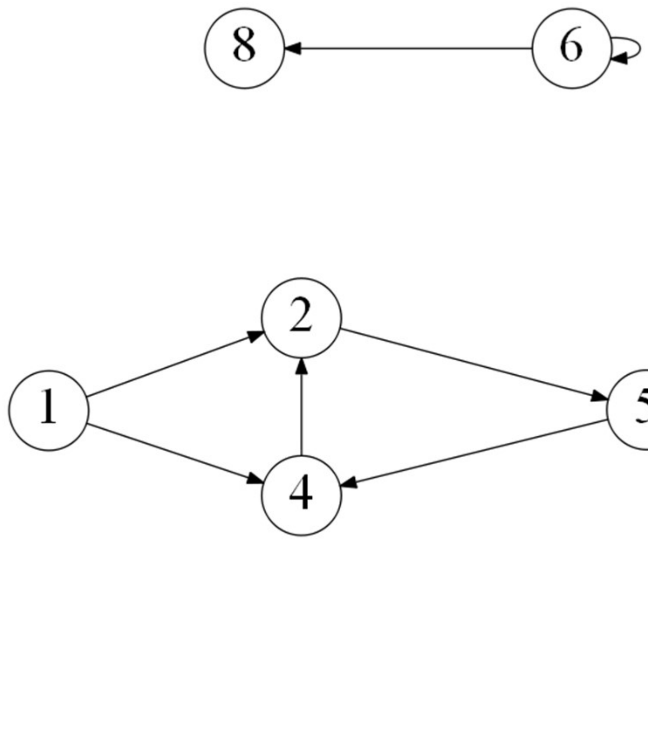
V = {0, 1, 2, 3, 4, 5, 6}
E = {{0, 1}, {0, 2}, {0, 5}, {0, 6}, {3, 5}, {3, 4}, {4, 5}, {4, 6}}

Relationship between Graphs and Trees

- A tree is a special case of a graph
- Any graph that is
  - Connected
  - contains no cycles can be viewed as a tree by making one of the vertices the root

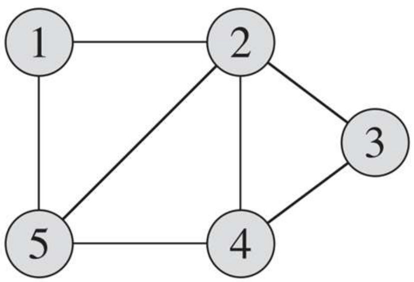


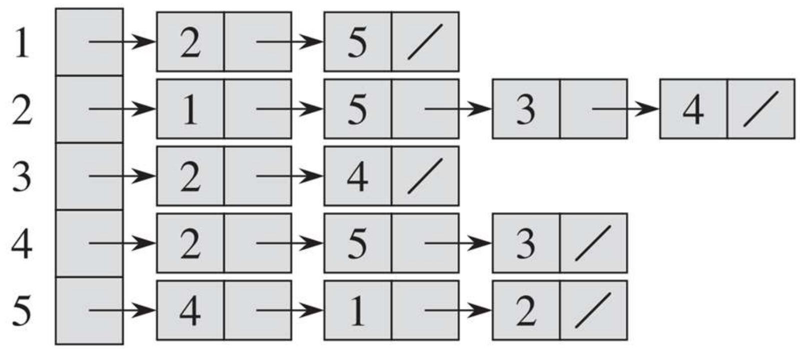
Sample Graph (One Graph!)



Graph Representation

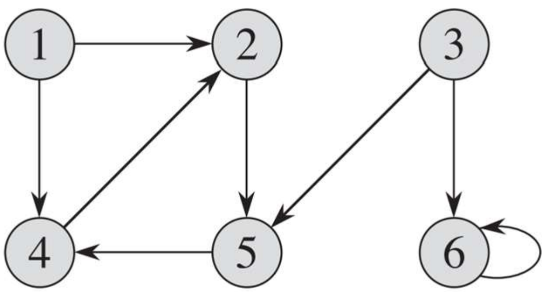
Adjacency Matrix vs Adjacency List (undirected graph)

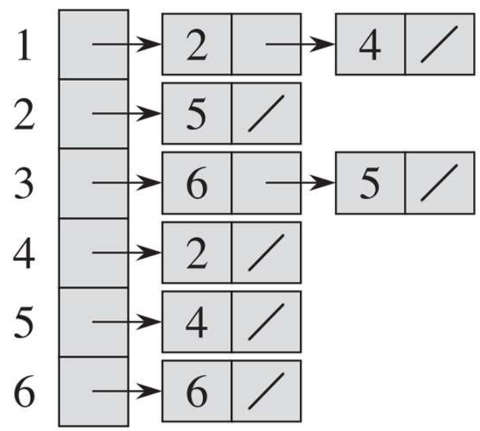




	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

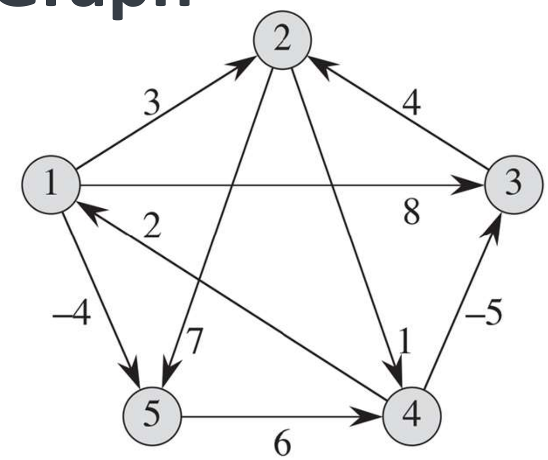
Directed Graph





	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Weighted & Directed Graph



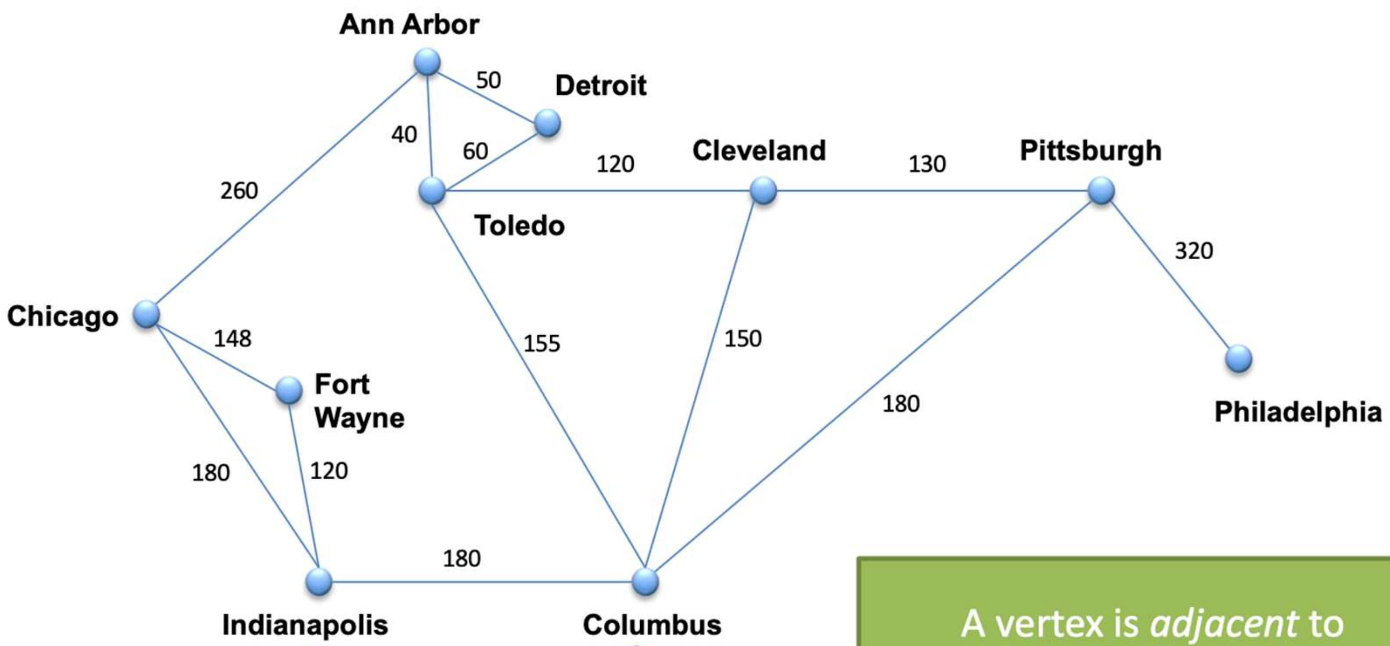
$$\begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

Adjacency Matrix for Weighted & Directed Graph

Graph Representations

	Adjacency Matrix	Adjacency List
Run time for determining if there is an edge between two vertices	$\theta(1)$	$O(d)$ Where d is the degree of the vertex (i.e. number of edges)
Run time for determining all vertices adjacent to a given vertex	$\theta(V)$	$\theta(d)$ -- where d is the degree of the vertex (i.e. number of edges) -- The number of edges incident to a vertex v
Space requirement	$\theta(V^2)$	$\theta(V + E)$
When to use	Small graphs Dense graphs	Large graphs Sparse graphs

Paths and Cycles

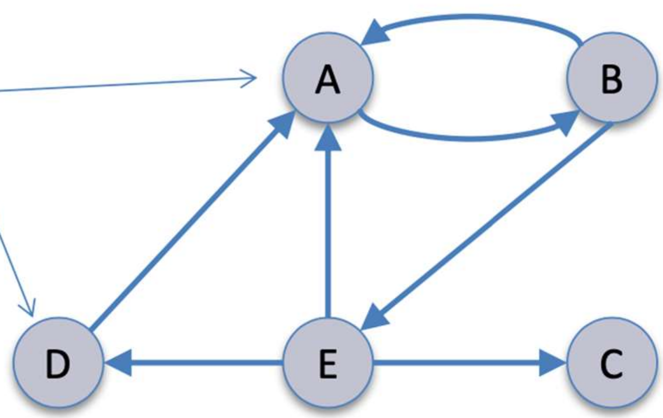


Indianapolis is adjacent to Columbus

A vertex is *adjacent* to another vertex if there is an edge to it from that other vertex

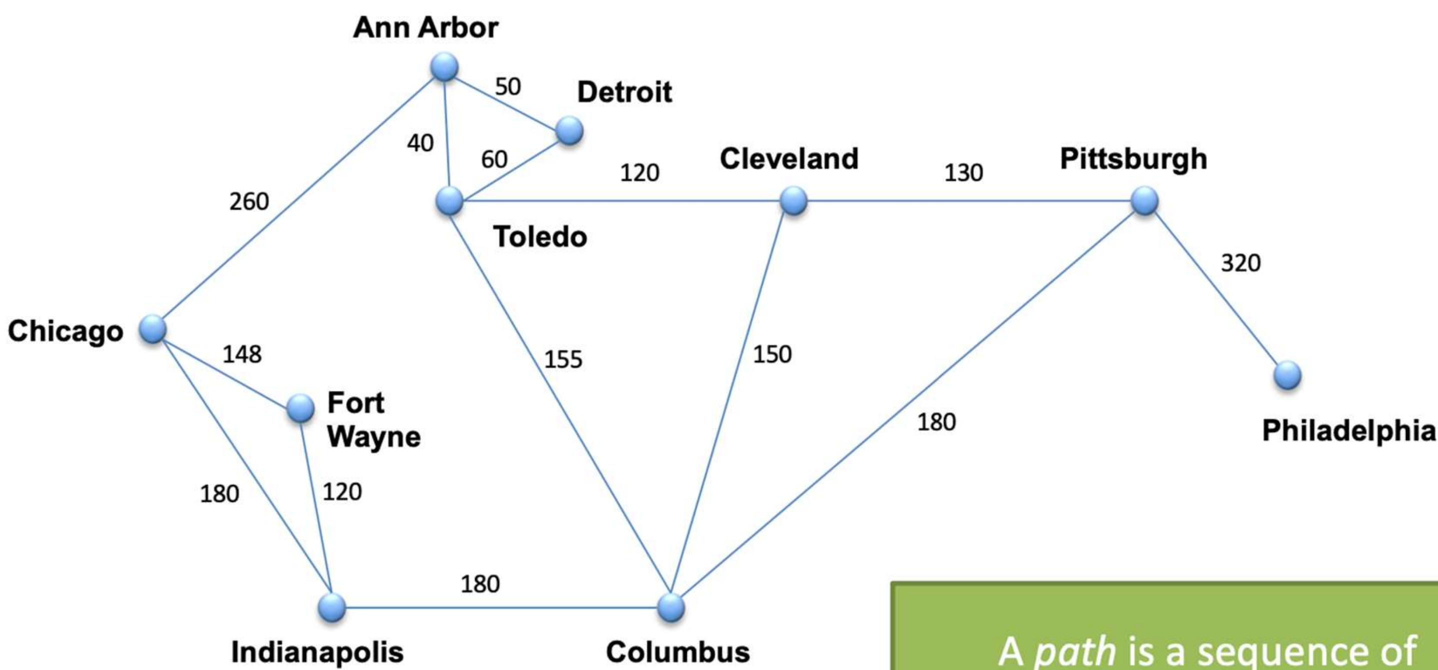
Paths and Cycles (cont.)

A is adjacent to D, but D is NOT adjacent to A



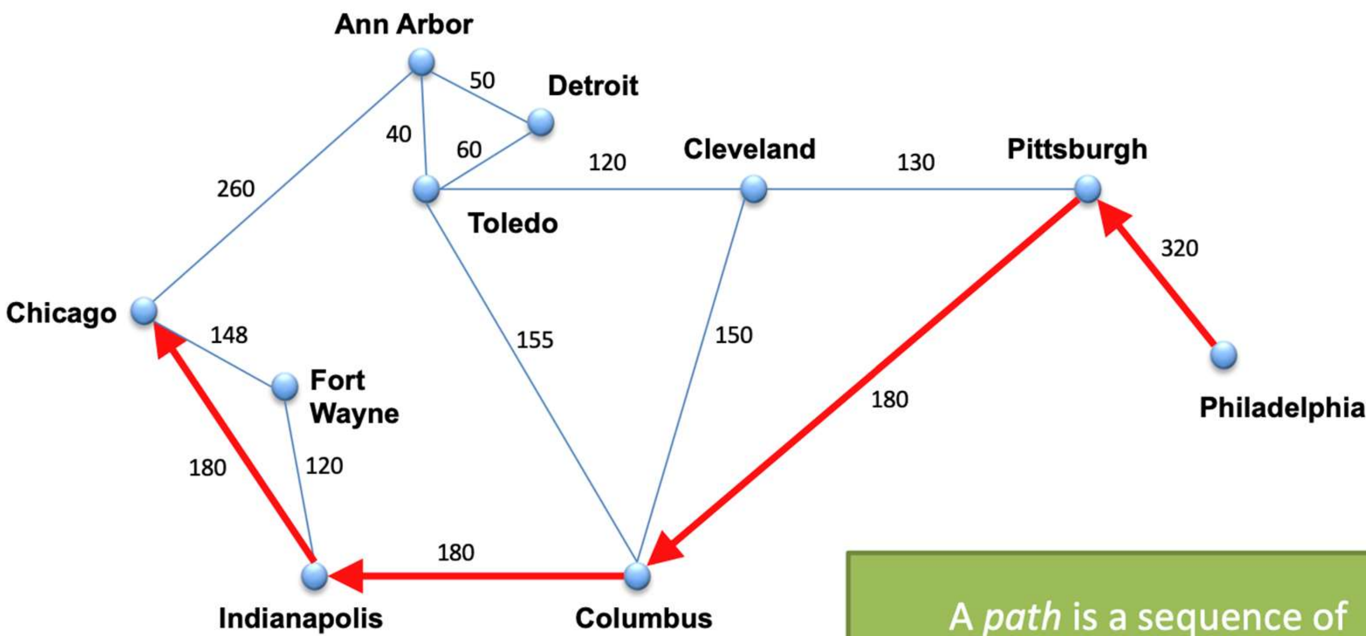
A vertex is *adjacent* to another vertex if there is an edge to it from that other vertex

Paths and Cycles (cont.)



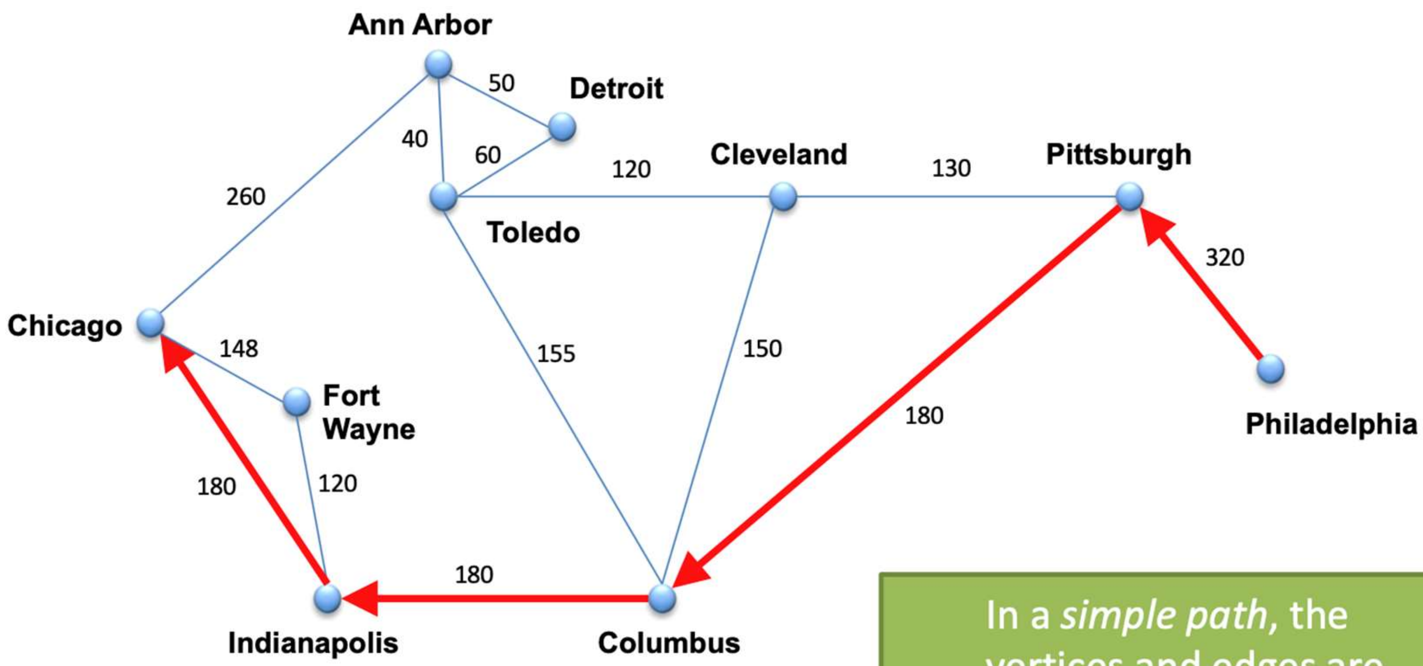
A *path* is a sequence of vertices in which each successive vertex is adjacent to its predecessor

Paths and Cycles (cont.)



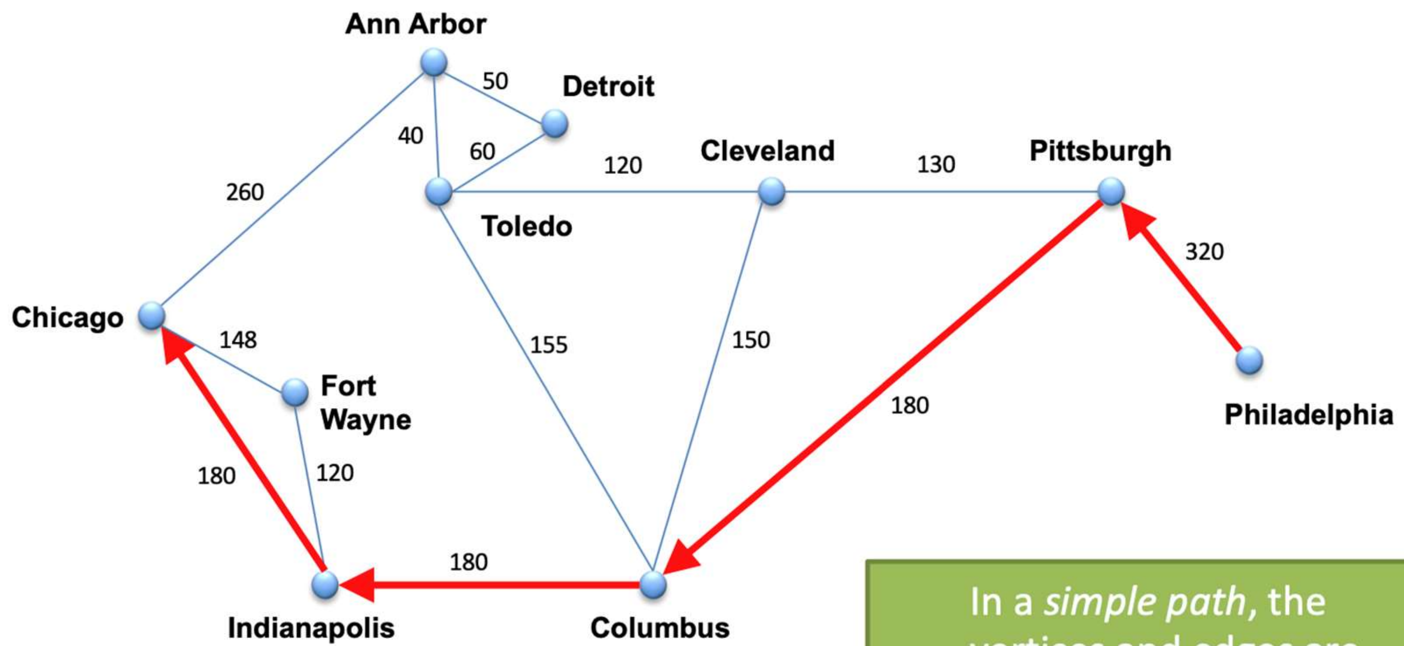
A *path* is a sequence of vertices in which each successive vertex is adjacent to its predecessor

Paths and Cycles (cont.)



In a *simple path*, the vertices and edges are distinct except that the first and last vertex may be the same

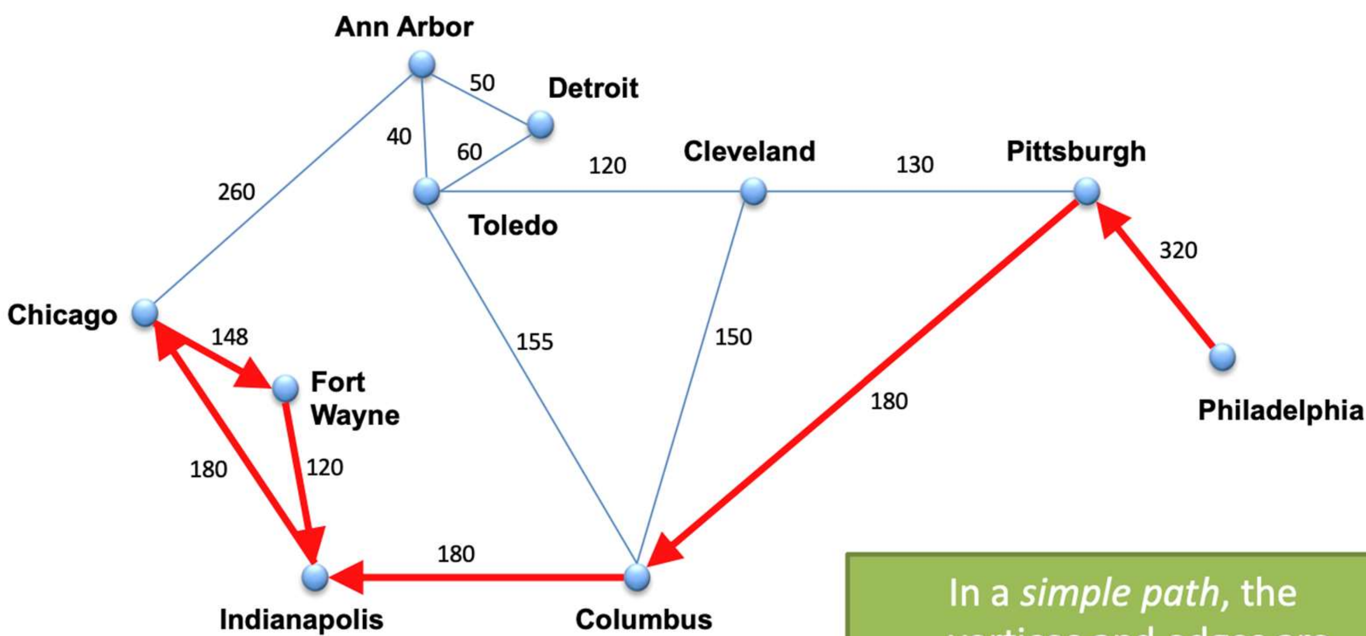
Paths and Cycles (cont.)



This path is a simple path

In a *simple path*, the vertices and edges are distinct except that the first and last vertex may be the same

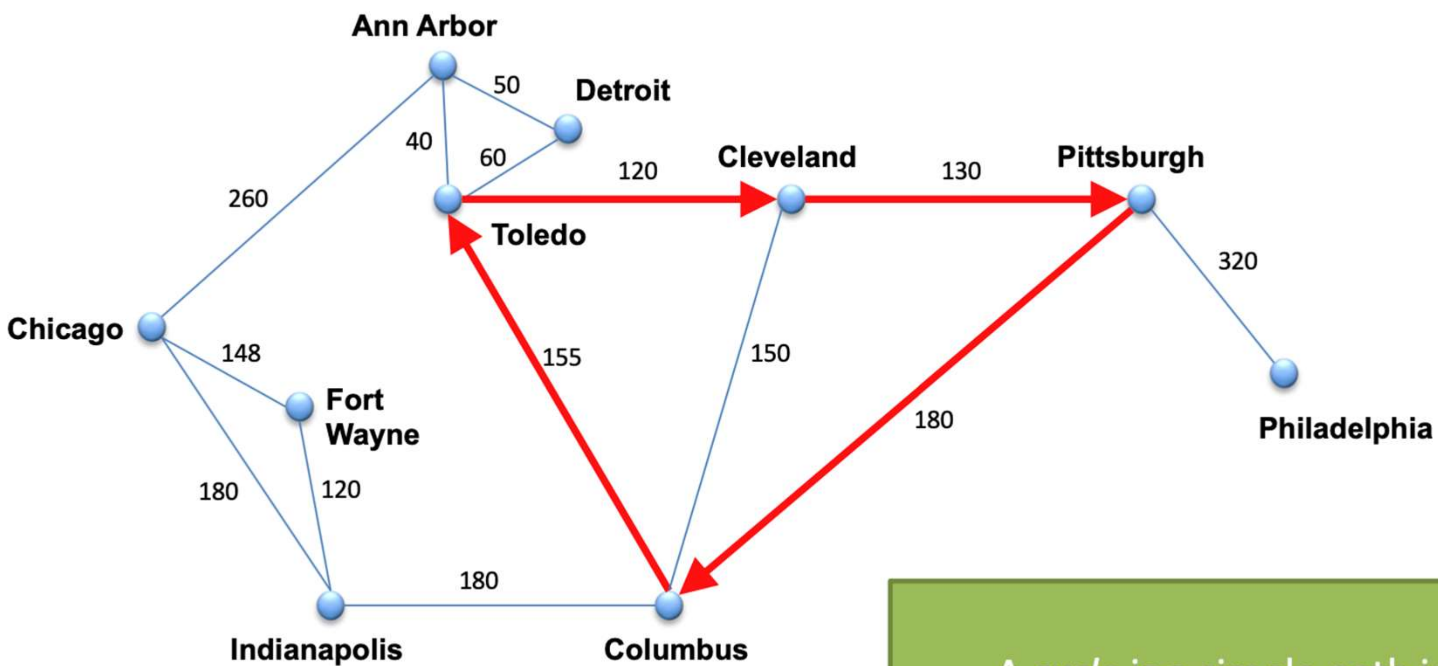
Paths and Cycles (cont.)



In a *simple path*, the vertices and edges are distinct except that the first and last vertex may be the same

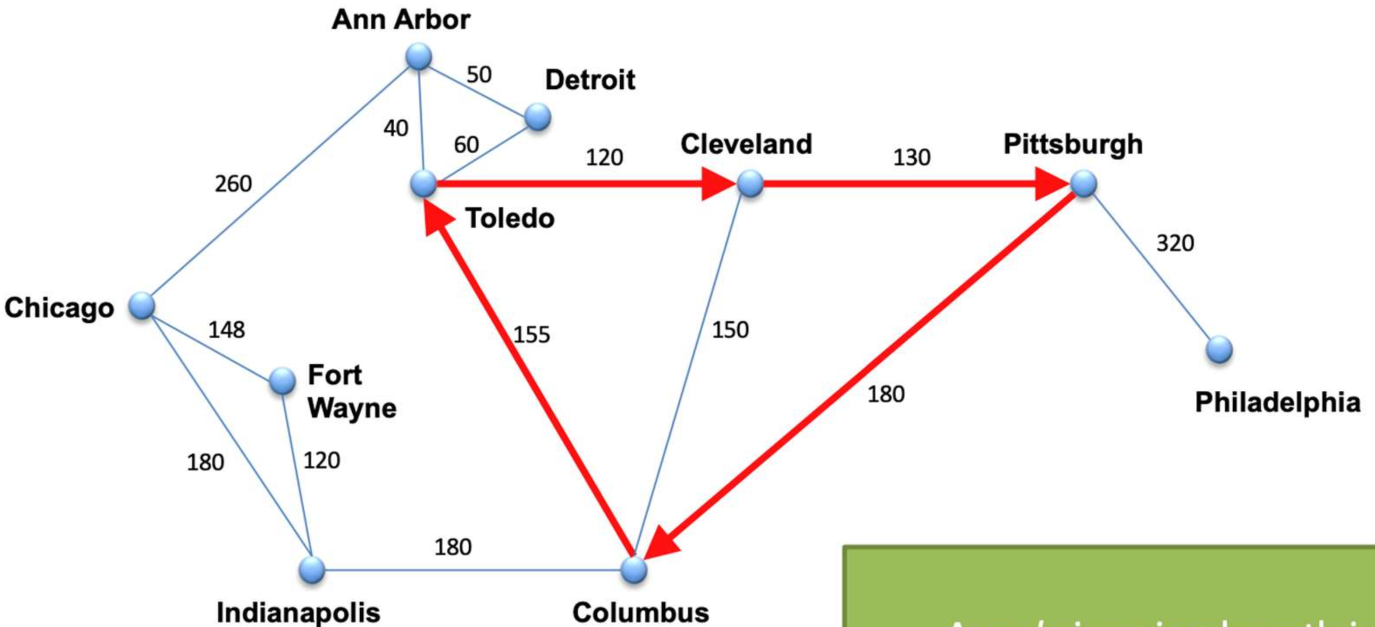
This path is NOT a simple path

Paths and Cycles (cont.)



A *cycle* is a simple path in which only the first and final vertices are the same

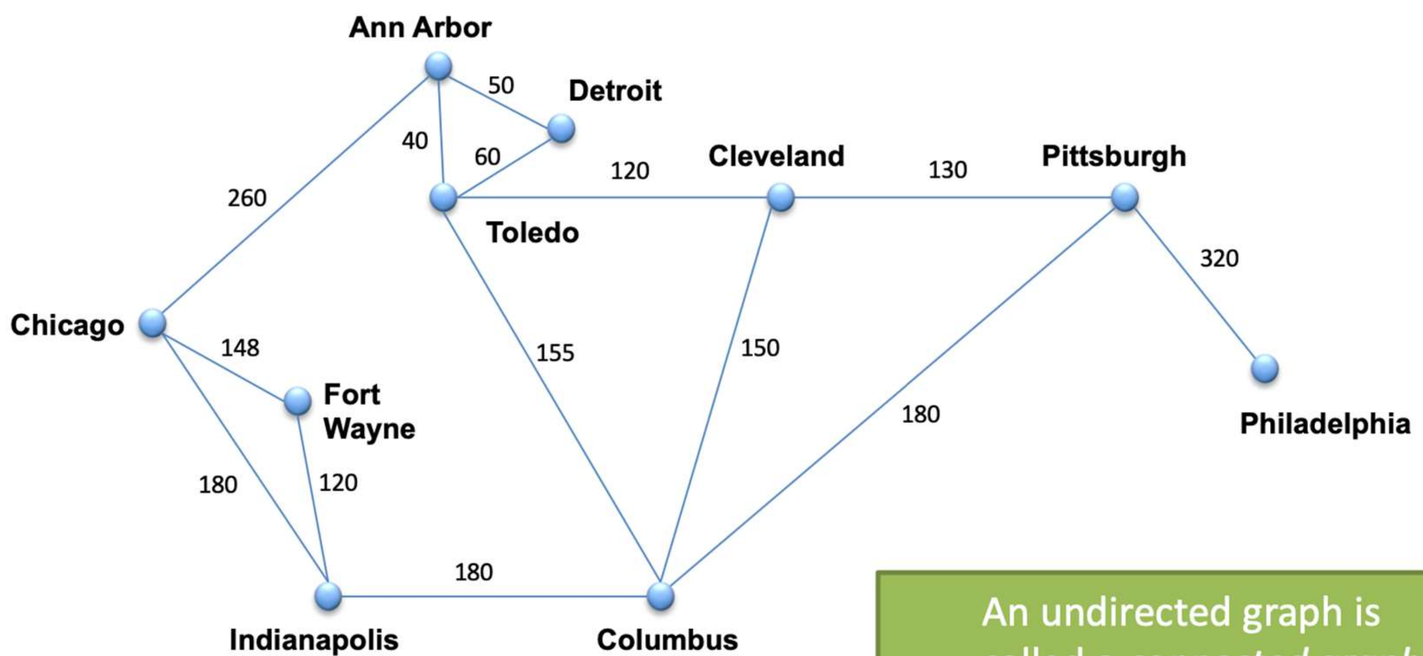
Paths and Cycles (cont.)



In an undirected graph a cycle must contain at least three distinct vertices
Pittsburgh ⟶ Columbus ⟶ Pittsburgh
is not a cycle

A *cycle* is a simple path in which only the first and final vertices are the same

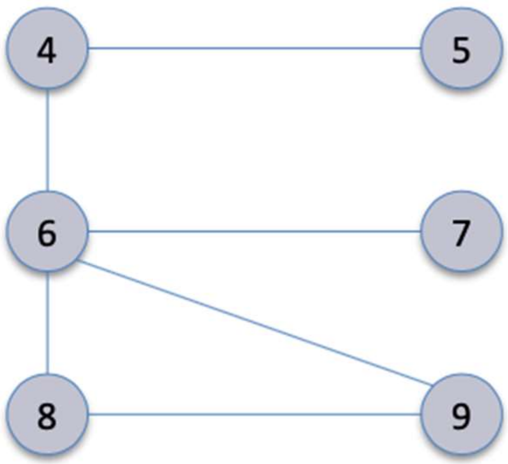
Connected Graph



This graph is a connected graph

An undirected graph is called a *connected graph* if there is a path from every vertex to every other vertex

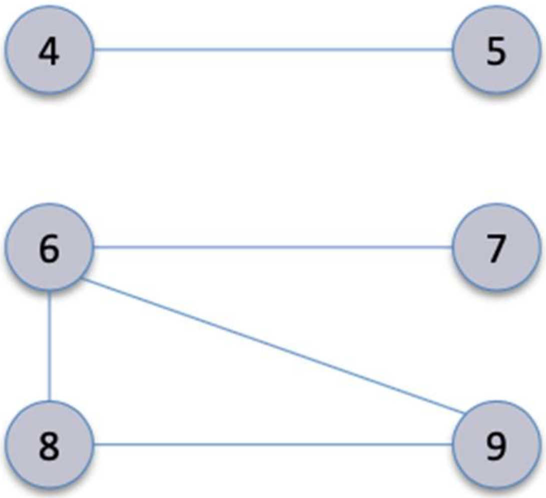
Connected Graph



An undirected graph is called a *connected graph* if there is a path from every vertex to every other vertex

This graph is a connected graph

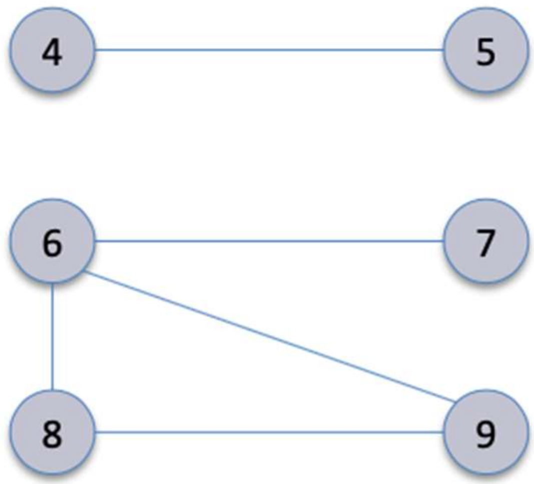
Connected Components



If a graph is not connected, it is considered *unconnected*, but still consists of *connected components*

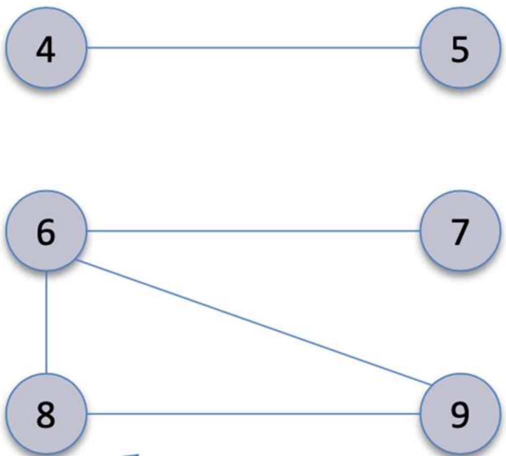
Connected Components

{4, 5} are connected components



If a graph is not connected, it is considered *unconnected*, but will still consist of *connected components*

Connected Components

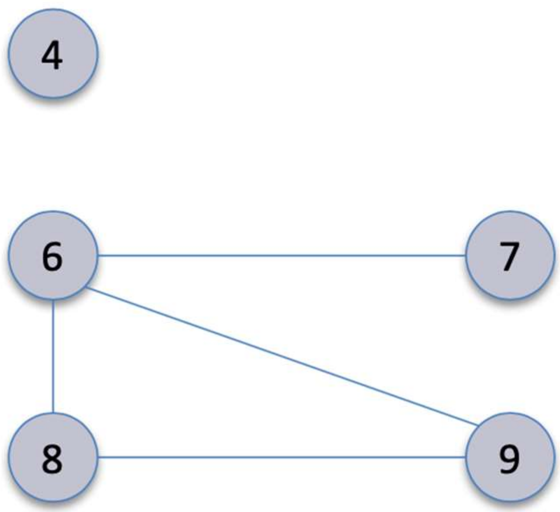


{6, 7, 8, 9} are connected components

If a graph is not connected, it is considered *unconnected*, but will still consist of *connected components*

Connected Components

A single vertex with no edge is also considered a connected component



If a graph is not connected, it is considered *unconnected*, but will still consist of *connected components*

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

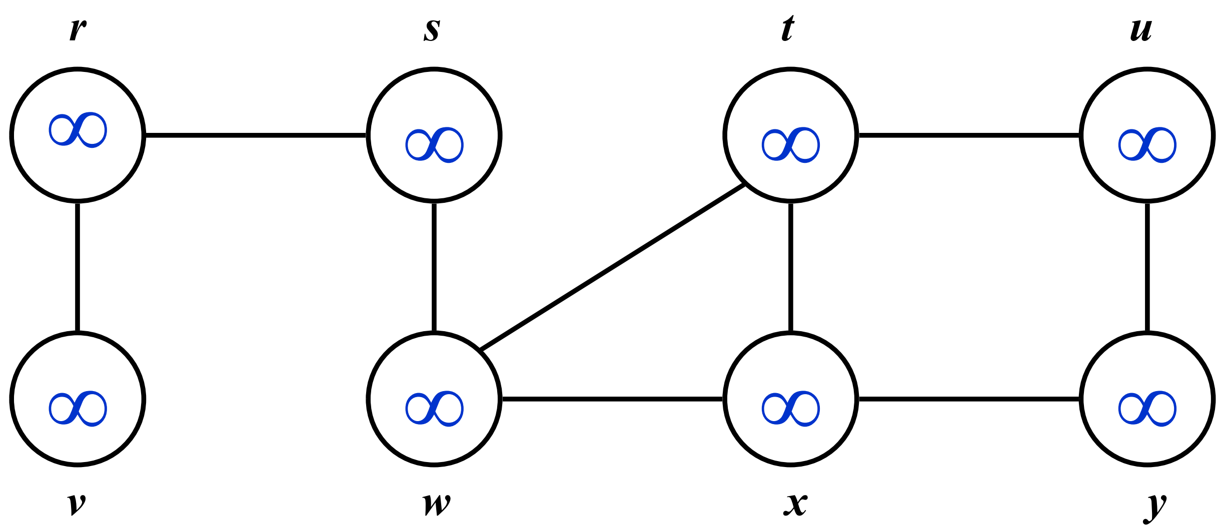
- There are two standard graph traversal techniques:
  - Breadth-First Search (BFS)
  - Depth-First Search (DFS)

Breadth-First Search

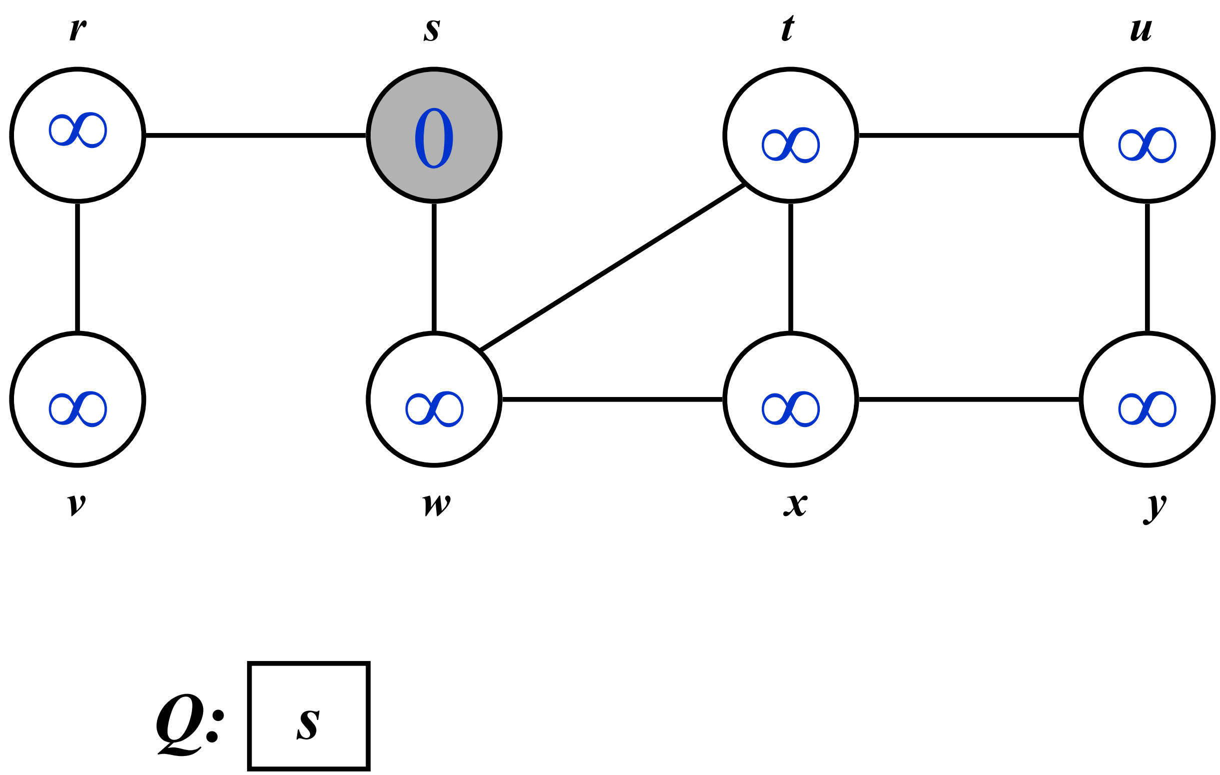
- “Explore” a graph, turning it into a tree
  - One vertex at a time
  - Expand frontier of explored vertices across the *breadth* of the frontier
- Builds a tree over the graph
  - Pick a *source vertex* to be the root
  - Find (“discover”) its children, then their children, etc.

```
BFS(G, s)
for each vertex u ∈ V[G] − {s}
      do color[u] ← WHITE
         d[u] ← ∞
         π[u] ← NIL
color[s] ← GRAY
d[s] ← 0
π[s] ← NIL
Q ← Ø
ENQUEUE(Q, s)
while Q ≠ Ø
      do u ← DEQUEUE(Q)
         for each v ∈ Adj[u]
             do if color[v] = WHITE
                   then color[v] ← GRAY
                        d[v] ← d[u] + 1
                        π[v] ← u
                        ENQUEUE(Q, v)
         color[u] ← BLACK
```

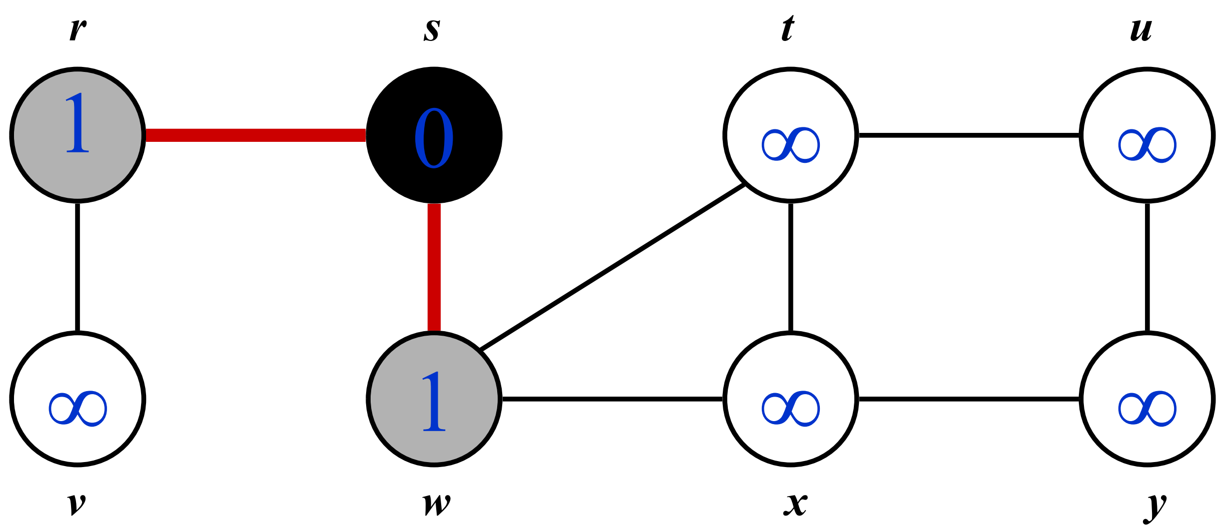
Breadth-First Search: Example



Breadth-First Search: Example

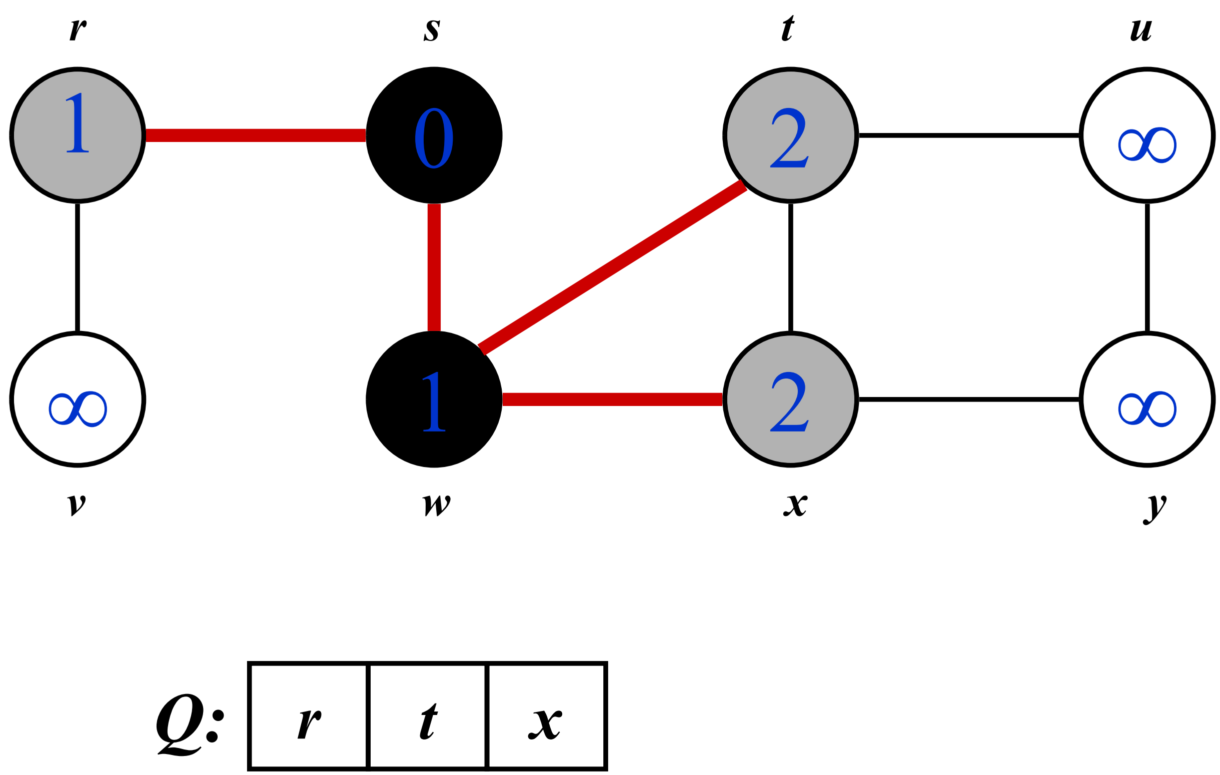


Breadth-First Search: Example

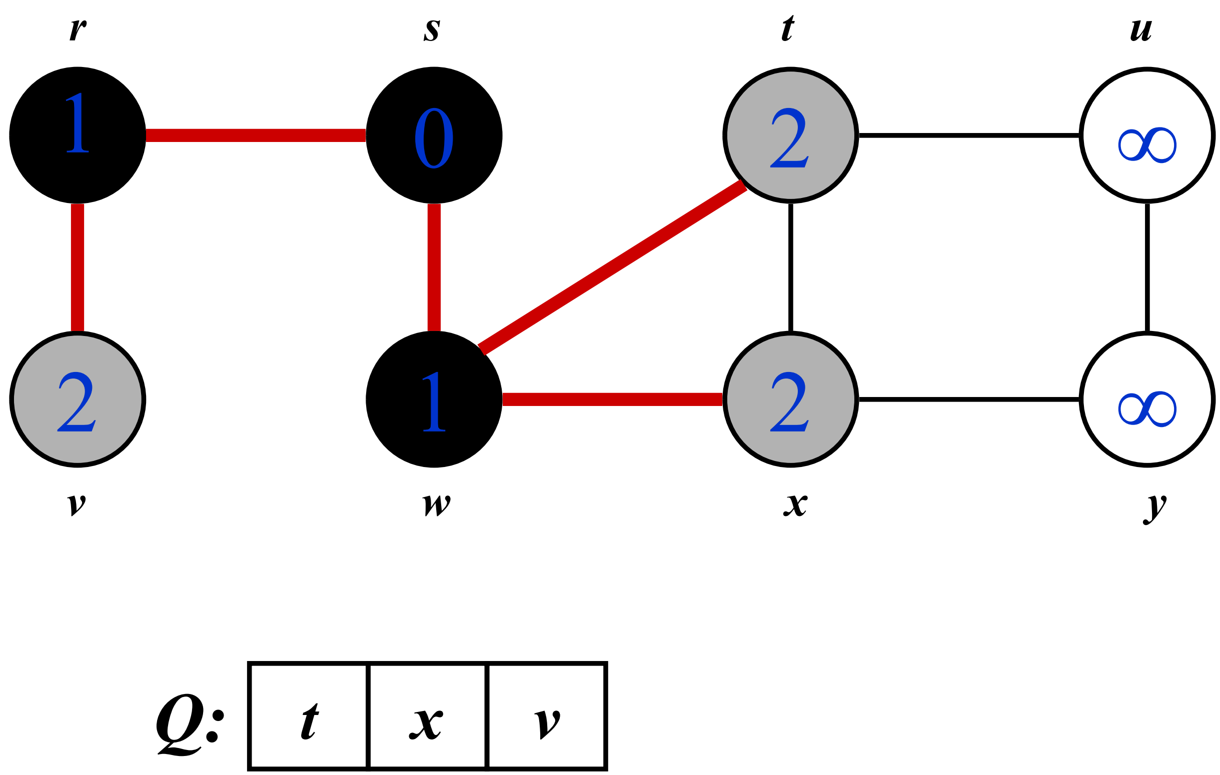


Q: | *w* | *r* |

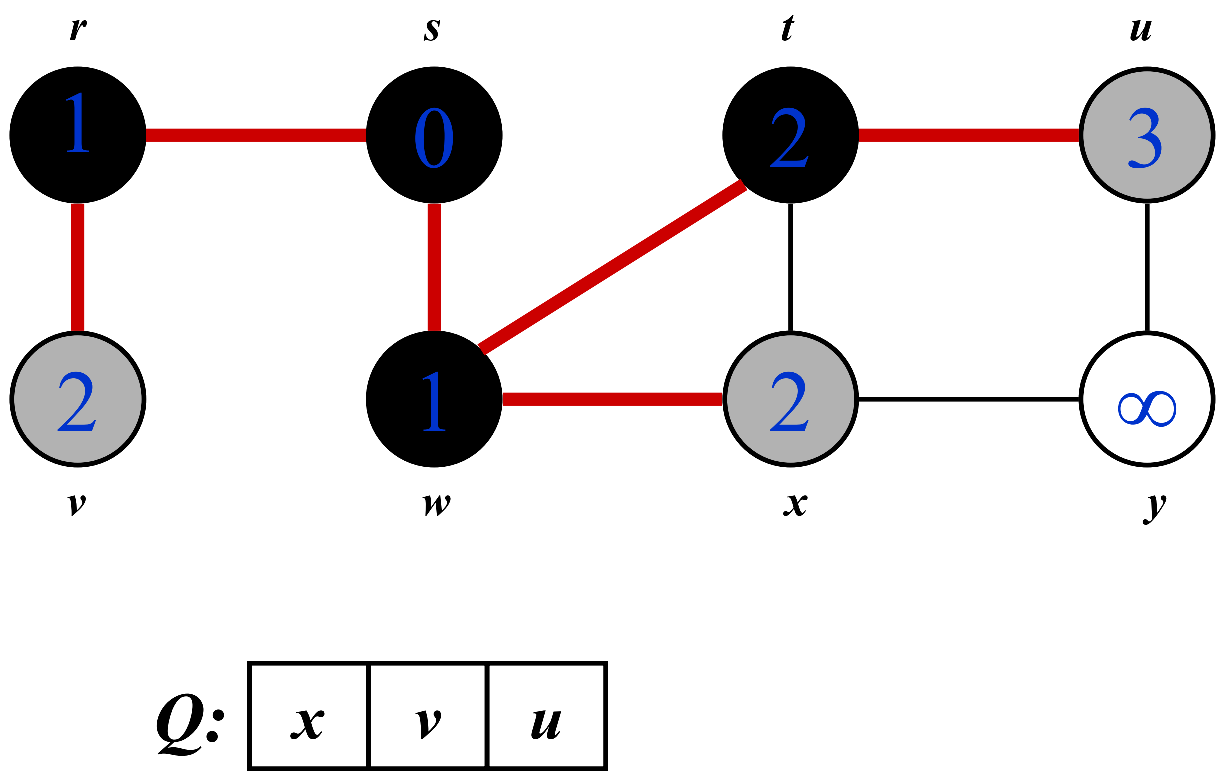
Breadth-First Search: Example



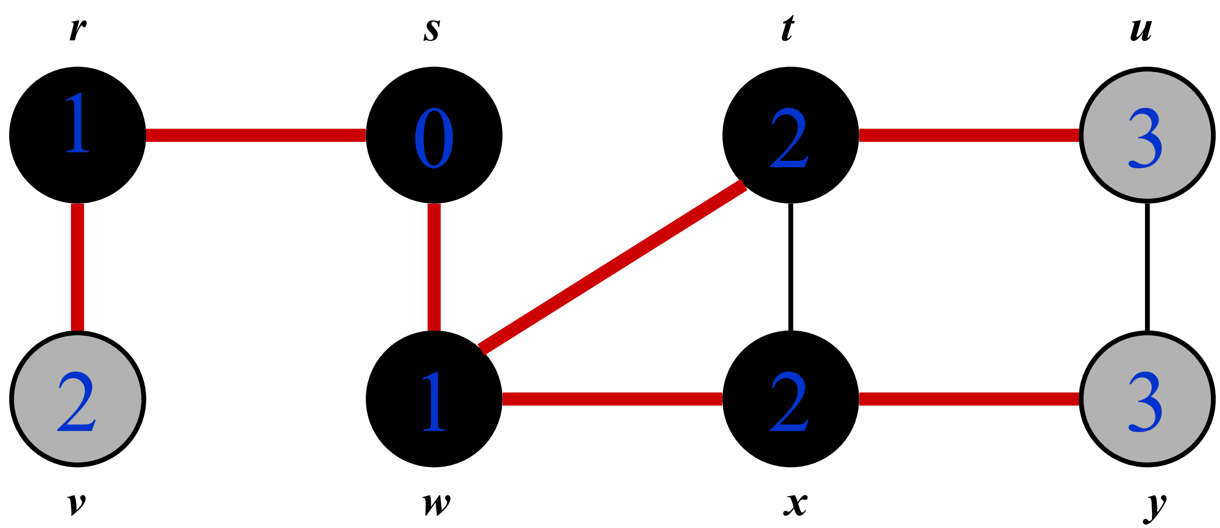
Breadth-First Search: Example



Breadth-First Search: Example

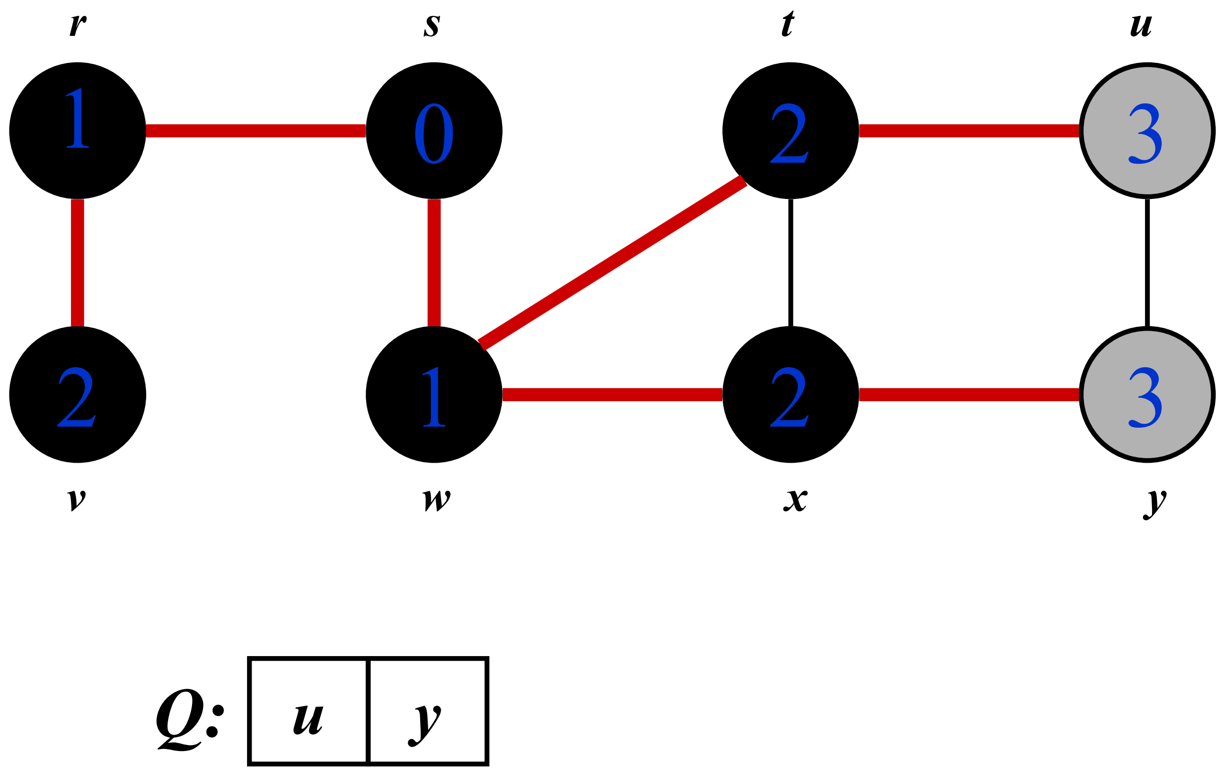


Breadth-First Search: Example

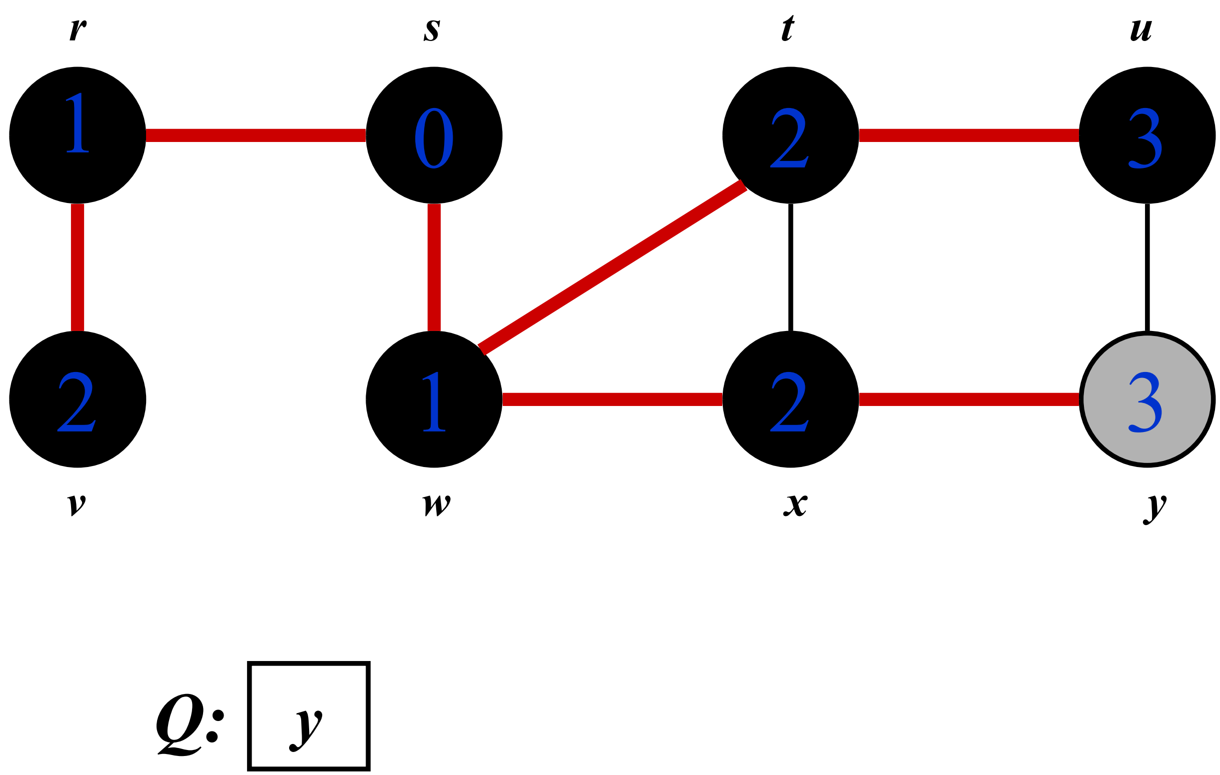


Q: | *v* | *u* | *y* |

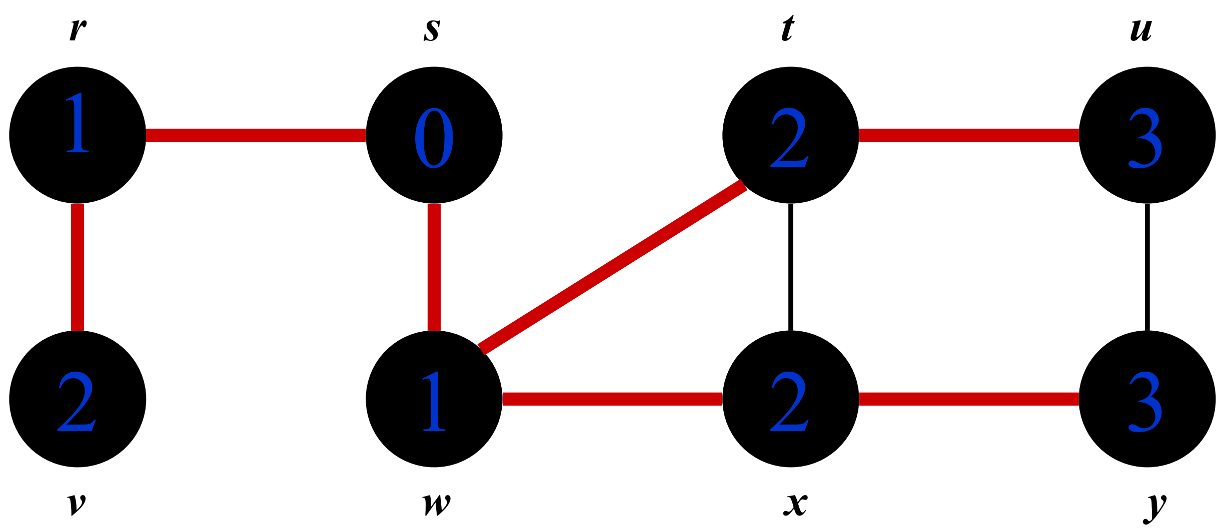
Breadth-First Search: Example



Breadth-First Search: Example



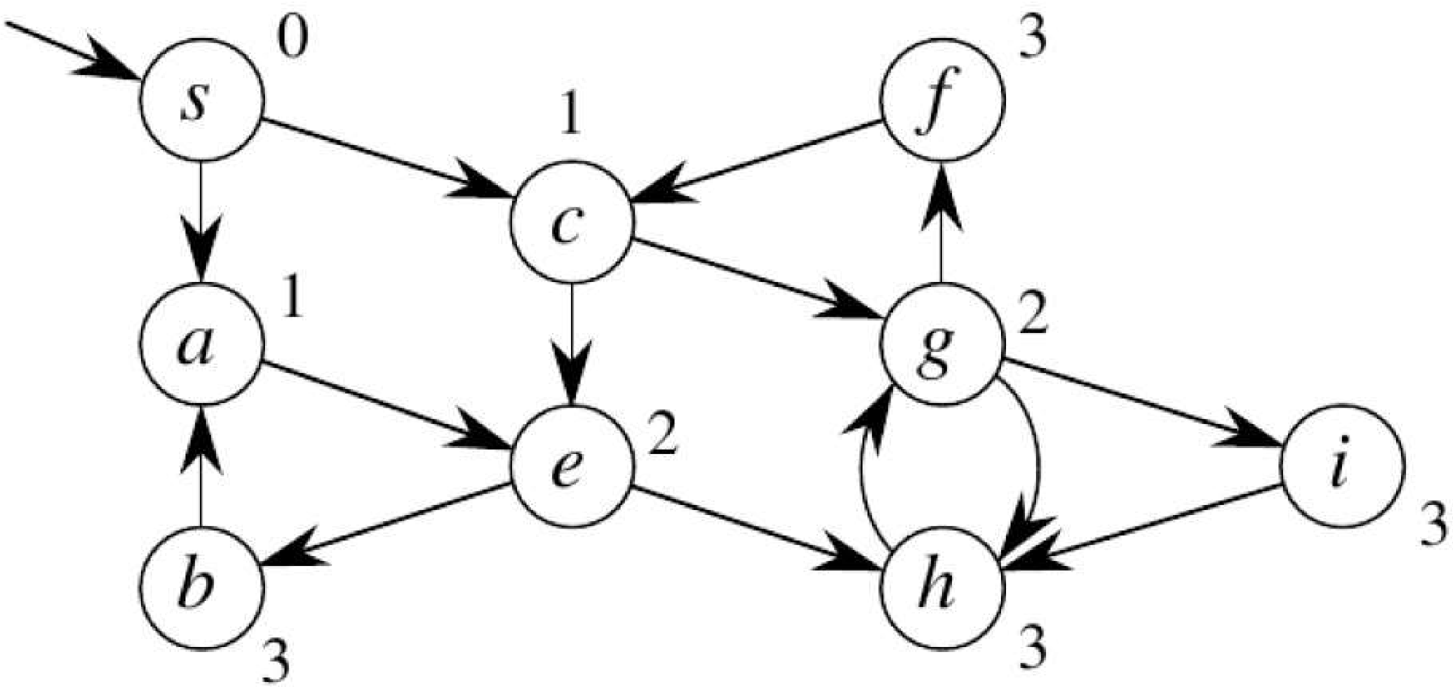
Breadth-First Search: Example



Q: Ø

BFS

Example: directed graph



BFS Running Time

- The operations of enqueuing and dequeuing take constant time, and so the total time devoted to queue operations is O(V). Because the procedure scans the adjacency list of each vertex only when the vertex is dequeued, it scans each adjacency list at most once (as every vertex and every edge will be explored in the worst case).

- Since the sum of the lengths of all the adjacency lists is |E|, the total time spent in scanning adjacency lists is O(E). The overhead for initialization is O(V), thus the total running time of the BFS procedure is O(V+E). Thus, breadth-first search runs in time linear in the size of the adjacency-list representation of G

  - **Adjacency list: $\mathbf{O}(V + E)$**

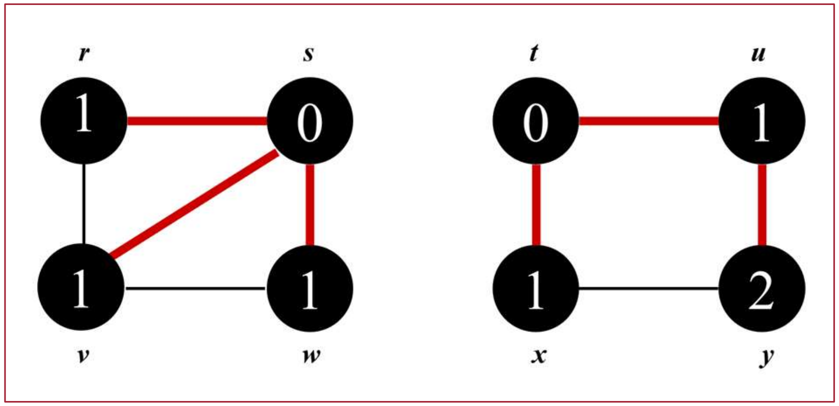
Breadth-First Search: Properties

- 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*

- BFS can find the number of disconnected components in undirected graphs

Number-of-Connected-Components(Undirected-graph G)

```
  Mark all nodes as unexplored
  // Consider that nodes are labeled from 1 to |v|
  for i = 1 to n // Consider that |v|=n
    if i is yet to be explored
      BFS(G, i)
```



Contents of this presentation are partially adapted from
My CS385 (Fall2022)
and from
Prof. In Suk Jang CS590 (Summer 2021 Lecture-10)
and are also based on
Book Chapter- 22, Introduction to Algorithms by *Cormen, Leiserson, Rivest,* & *Stein*

STEVENS
INSTITUTE OF TECHNOLOGY
1870

THANK **YOU**

Stevens Institute of Technology