# Elementary Graph Algorithms – contd.

## Graphs

- Motivation and Terminology
- Representations
- Traversals

## Traversing a Graph

One of the most fundamental graph problems is to traverse every edge and vertex in a graph. Applications include:

- Printing out the contents of each edge and vertex.
- Counting the number of edges.
- Identifying connected components of a graph.

For *correctness*, we must do the traversal in a systematic way so that we don't miss anything.

For *efficiency*, we must make sure we visit each edge at most twice.

## Marking Vertices

*The idea in graph traversal is that we mark each vertex when we first visit it, and keep track of what is not yet completely explored.*

For each vertex, we maintain two flags:

- ***discovered*** - have we encountered this vertex before?
- ***explored*** - have we finished exploring this vertex?

We must maintain a structure containing all the vertices we have discovered but not yet completely explored.

Initially, only a single start vertex is set to be discovered.

## 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

## Correctness of Graph Traversal

Every edge and vertex in the connected component is eventually visited.

Suppose not, i.e. there exists a vertex which was unvisited whose neighbor *was* visited. This neighbor will eventually be explored so we *would* visit it….

## Traversal Orders

The order we explore the vertices depends upon the data structure used to hold the discovered vertices yet to be fully explored:

- ***Queue*** - by storing the vertices in a first-in, first out (FIFO) queue, we explore the oldest unexplored vertices first. Thus we radiate out slowly from the starting vertex, defining a so-called *breadth-first search*.
- ***Stack*** - by storing the vertices in a last-in, first-out (LIFO) stack, we explore the vertices by constantly visiting a new neighbor if one is available; we back up only when surrounded by previously discovered vertices. This defines a so-called *depth-first search*.

## Where are these used?

- Breadth-first search
  - *What can we use BFS to calculate?*
  - A: shortest-path distance to source vertex
- Depth-first search
  - Tree edges, back edges, cross and forward edges
  - *What can we use DFS for?*
  - A: finding cycles, topological sort

## Breadth-First Search

- Breadth-first search is a method for visiting all the vertices of a graph in a sequence, based on their proximity to a designated starting vertex.
- A breadth-first search (BFS) of a graph G is a way to traverse every vertex of a connected graph by constructing a spanning tree, rooted at a given vertex r. After the BFS traversal visits a vertex v, all of the previously unvisited neighbors of v are enqueued, and then the traversal removes from the queue whatever vertex is at the front of the queue, and visits that vertex.

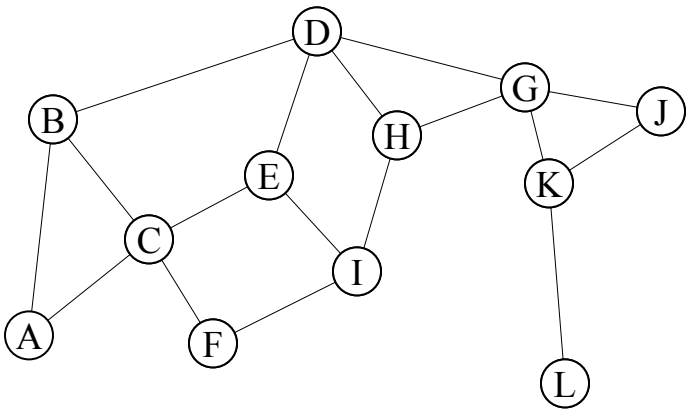
## Breadth-First Search

- "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.

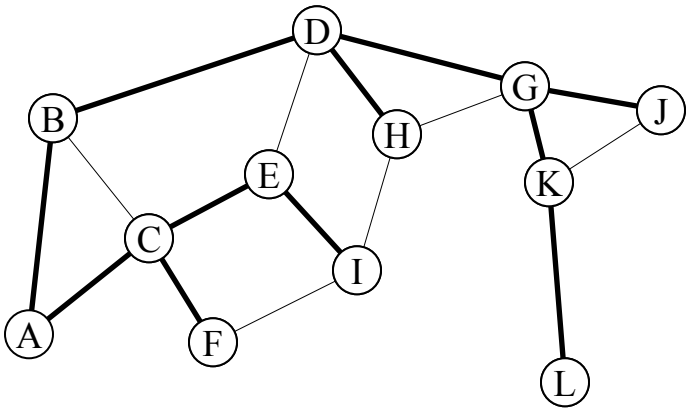
## Breadth-First Search

- Again will associate vertex "colors" to guide the algorithm
  - White vertices have not been discovered
    - All vertices start out white
  - 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

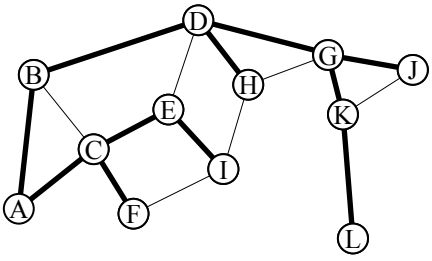
## Graph Traversal - Example

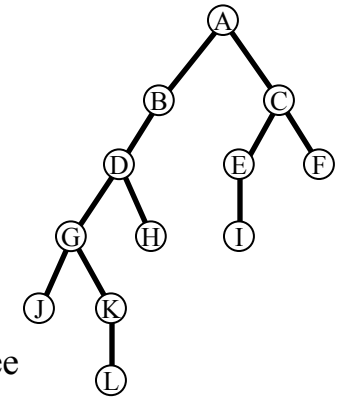


## Breadth-First Search Tree



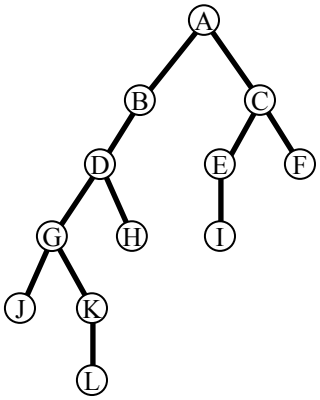

## Key BFS Tree Property




Every root-leaf path in a BFS tree has the fewest number of edges possible.

# Questions about BFS tree




Every root-leaf path in a BFS tree has the fewest number of edges possible.

- Suppose we are only given a BFS tree. Can we infer information about other potential edges?
- Edge (B,C)?
- Edge (C,D)?
- Edge (C,H)?

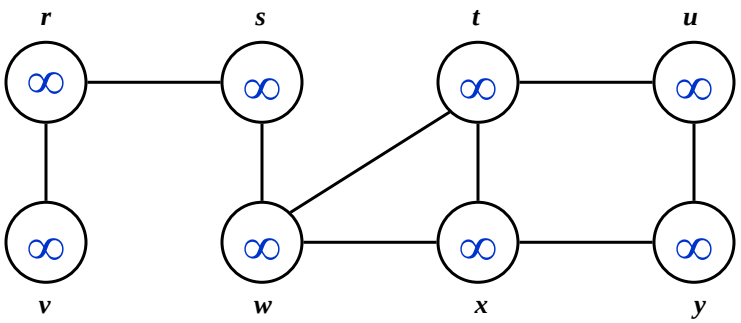
# Breadth-First Search

```
BFS(G, s) {
    initialize vertices;
    Q = {s};        // Q is a queue (duh); initialize to s
    while (Q not empty) {
        u = RemoveTop(Q);
        for each v ∈ u->adj {
            if (v->color == WHITE)
                v->color = GREY;
                v->d = u->d + 1;
                v->p = u;
                Enqueue(Q, v);
        }
        u->color = BLACK;
    }
}
```

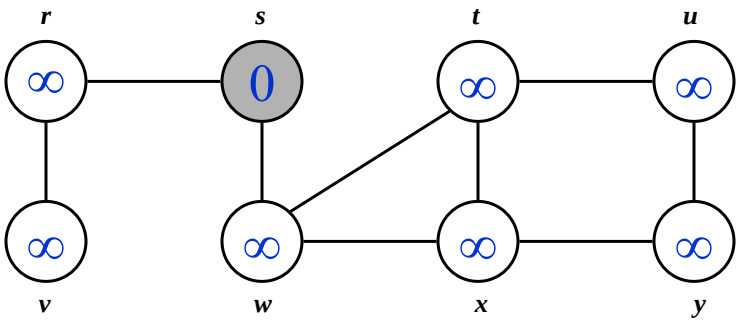
*What does* `v->d` *represent?*

*What does* `v->p` *represent?*

## Breadth-First Search: Example

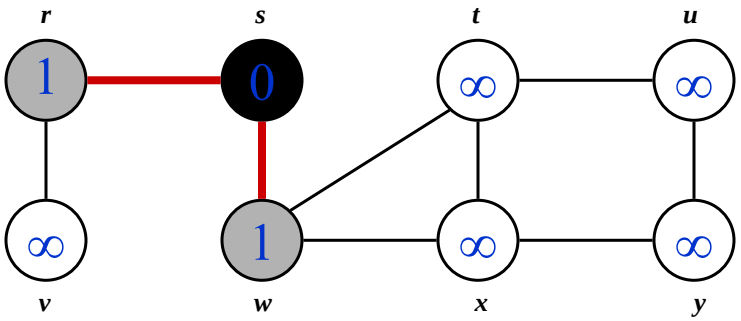



## Breadth-First Search: Example



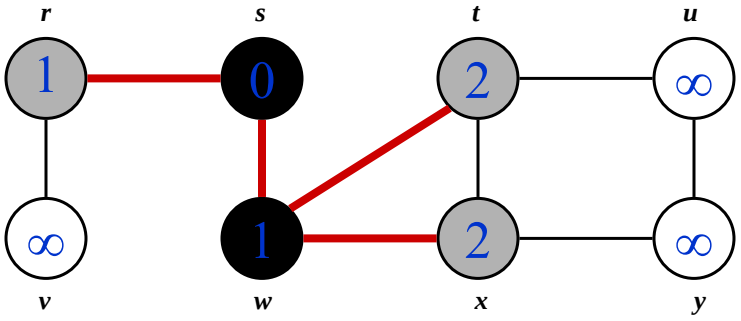

***Q:*** ***s***

## Breadth-First Search: Example



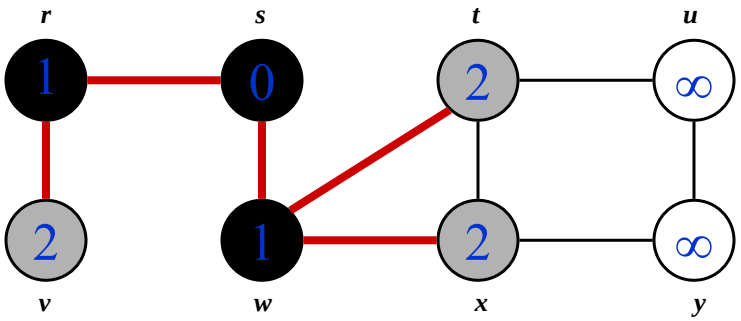

*Q:* | *w* | *r* |

## Breadth-First Search: Example



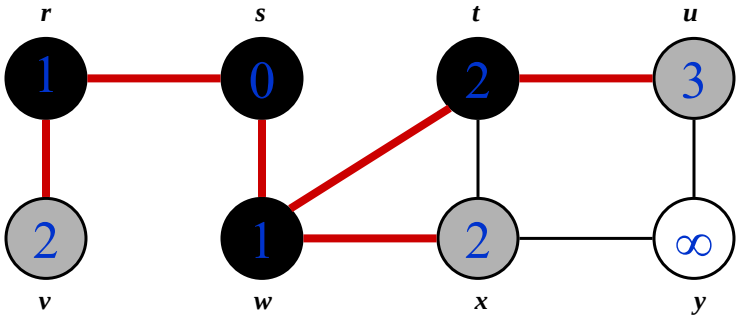

*Q:* | *r* | *t* | *x* |

## Breadth-First Search: Example



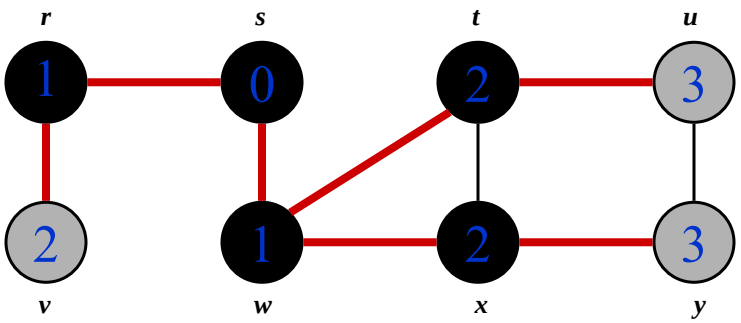

$Q:$ | $t$ | $x$ | $v$ |

## Breadth-First Search: Example



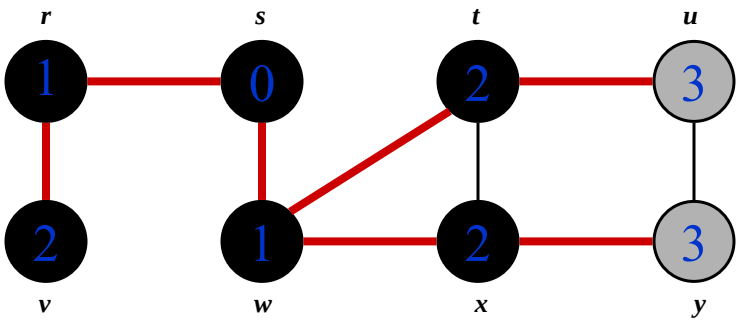

$Q:$ | $x$ | $v$ | $u$ |

## Breadth-First Search: Example



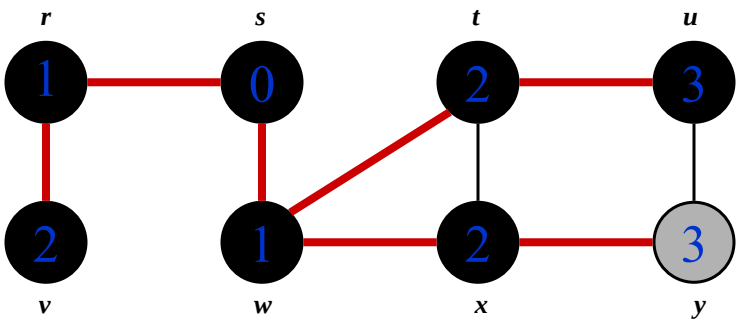

*Q:* | *v* | *u* | *y* |

## Breadth-First Search: Example



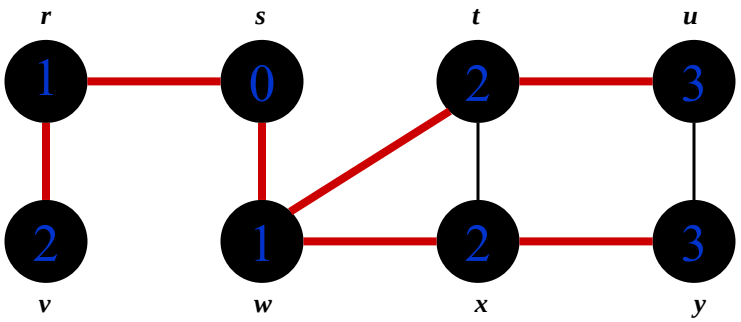

*Q:* | *u* | *y* |

## Breadth-First Search: Example




*Q:* y

## Breadth-First Search: Example




*Q:* Ø

# BFS: The Code Again

```
BFS(G, s) {
    initialize vertices;
    Q = {s};
    while (Q not empty) {
        u = RemoveTop(Q);
        for each v ∈ u->adj {
            if (v->color == WHITE)
                v->color = GREY;
                v->d = u->d + 1;
                v->p = u;
                Enqueue(Q, v);
        }
        u->color = BLACK;
    }
}
```

*Touch every vertex: O(V)*

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

*So v = every vertex that appears in some other vert's adjacency list*

*What will be the running time?*

**Total running time: O(V+E)**

# BFS: The Code Again

```
BFS(G, s) {
    initialize vertices;
    Q = {s};
    while (Q not empty) {
        u = RemoveTop(Q);
        for each v ∈ u->adj {
            if (v->color == WHITE)
                v->color = GREY;
                v->d = u->d + 1;
                v->p = u;
                Enqueue(Q, v);
        }
        u->color = BLACK;
    }
}
```

*What will be the storage cost in addition to storing the tree?*

**Total space used: O(max(degree(v))) = O(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 $\infty$ if v not reachable from s
  - Proof given in the book (p. 597-601)
- 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