Objectives

- Finding Connected Components
  - Breadth-first
  - Depth-first
- Implementing the algorithms

Review: Graphs

- What are the two ways to represent graphs?
- What is the space cost for the adjacency list?

Review: Connected Component

- Find all nodes ***reachable*** from *s*

In general….

```
R will consist of nodes to which s has a path
R = {s}
while there is an edge (u,v) where u∈R and v∉R
      add v to R
```

- Theorem. Upon termination, R is the connected component containing s

How can we explore the nodes?

Finding Connected Components

- Find all nodes ***reachable*** from *s*

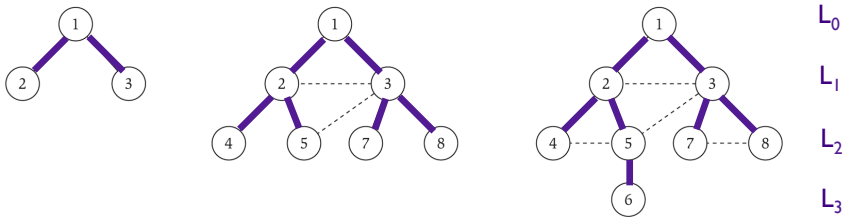
In general….

```
R will consist of nodes to which s has a path
R = {s}
while there is an edge (u,v) where u∈R and v∉R
      add v to R
```

In what order does BFS consider edges?
What is the outcome?

Review: Example of Breadth-First Search

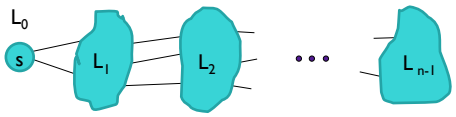
s = 1



Creates a tree
-- is a node in the graph that is not in the tree

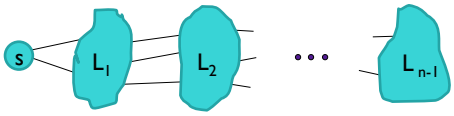
Review: Breadth-First Search

- **Intuition**. Explore outward from *s* in all possible directions (edges), adding nodes one "layer" at a time
- **Algorithm**
  - $L_0 = \{ s \}$
  - L_1 = all neighbors of L_0
  - L_2 = all nodes that have an edge to a node in L_1 and do not belong to L_0 or L_1
  - L_{i+1} = all nodes that have an edge to a node in L_i and do not belong to an earlier layer



Breadth-First Search

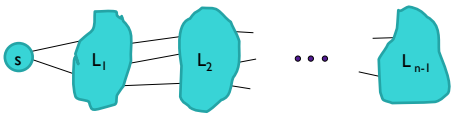
- **Theorem.**
  For each *i*, L_i consists of all nodes at distance exactly *i* from *s*.
  There is a path from s to t iff t appears in some layer.



- What does this theorem mean?
- Can we determine the distance between s and t?

Breadth-First Search

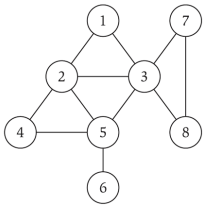
- **Theorem.** For each *i*, L_i consists of all nodes at distance exactly *i* from *s*. There is a path from *s* to *t* iff *t* appears in some layer.
  - Shortest path to *t* from *s* is the *i* from L_i that *t* is in
  - All nodes ***reachable*** from *s* are in $L_1, L_2, \ldots, L_{n-1}$



Breadth-First Search

- **Property**. Let T be a BFS tree of G = (V, E), and let (x, y) be an edge of G. Then the level of x and y *differ* by *at most* 1.

G:

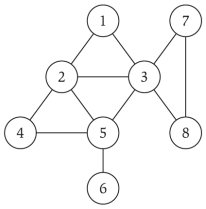


If x is in L_i,
then y must be in ???

Breadth-First Search

- **Property**. Let T be a BFS tree of G = (V, E), and let (x, y) be an edge of G. Then the level of x and y *differ* by *at most* 1.

G:



If x is in L_i, then y must be in

- L_{i-1}: y was reached before x
- L_i: a common parent of x and y was reached first
- L_{i+1}: y will be added in the next layer

Connected Component: BFS vs DFS

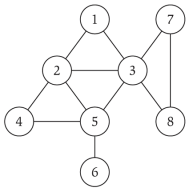
- Find all nodes ***reachable*** from *s*

In general….

```
R will consist of nodes to which s has a path
R = {s}
while there is an edge (u,v) where u∈R and v∉R
      add v to R
```

- Theorem. Upon termination, R is the connected component containing s
  - BFS = explore in order of distance from s
  - DFS = explore until hit "deadend"

Depth-First Search

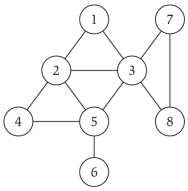


- Need to keep track of where you've been
- When reach a "dead-end" (already explored all neighbors), backtrack to node with unexplored neighbor
- **Algorithm**:

```
DFS(u):
   Mark u as "Explored" and add u to R
   For each edge (u, v) incident to u
      If v is not marked "Explored" then
            DFS(v)
```

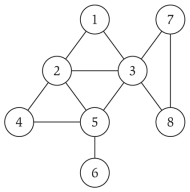
Depth-First Search

- How does DFS work on this graph?
  - Starting from node 1



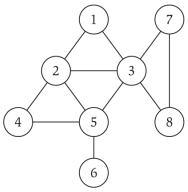
DFS vs BFS

- Compare the resulting trees



DFS vs BFS: Tree Comparison

- BFS
  - Bushy
- DFS
  - Spindly



DFS Analysis

- Let T be a depth-first search tree, let *x* and *y* be nodes in T, and let (*x*, *y*) be an edge of G that is not an edge of T. Then one of *x* or *y* is an ancestor of the other in T.

parallel of BFS: connected nodes are at most one layer apart

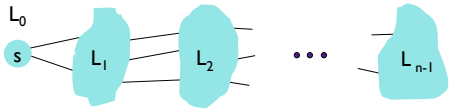
DFS Analysis

- Let T be a depth-first search tree, let *x* and *y* be nodes in T, and let (*x*, *y*) be an edge of G that is not an edge of T. Then one of *x* or *y* is an ancestor of the other in T.
- Proof.
  - Suppose that x-y is an edge in G but not in T. (From problem statement)
  - WLOG, assume that DFS reaches x before y
  - When edge x-y is considered in the DFS algorithm, we don't add it to T (from problem statement), which means that y must have been explored.
  - But, since we reached x first, y had to be discovered between invocation and end of the recursive call DFS(x)
    - i.e., y is a descendent of x

IMPLEMENTING ALGORITHMS

Review: Breadth-First Search

- **Intuition**. Explore outward from *s* in all possible directions (edges), adding nodes one "layer" at a time
- **Algorithm**
  - $L_0 = \{ s \}$
  - L_1 = all neighbors of L_0
  - L_2 = all nodes that have an edge to a node in L_1 and do not belong to L_0 or L_1
  - L_{i+1} = all nodes that have an edge to a node in L_i and do not belong to an earlier layer



Implementing BFS

- What do we need as input?
- What do we need to model?
  - How will we model that?

Implementing BFS

- Input: Graph as an adjacency list
- Discovered array
- Maintain layers in separate lists, L[i]

Implementing BFS

- Graph: Adjacency list
- Discovered array
- Maintain layers L[i]

What does this stopping condition mean?

L[i] representation?

```
BFS(s, G):
    Discovered[v] = false, for all v
    Discovered[s] = true
    L[0] = {s}
    layer counter i = 0
    BFS tree T = {}
    while L[i] != {}
        L[i+1] = {}
        for each node u ∈ L[i]
            Consider each edge (u,v) incident to u
            if Discovered[v] == false then
               Discovered[v] = true
               Add edge (u, v) to tree T
               Add v to the list L[i + 1]
        i+=1
```

BFS Analysis

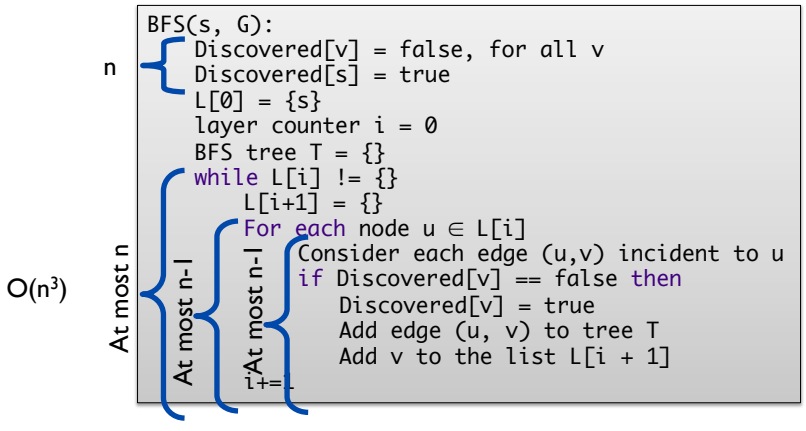
Given: s – start node, G – adjacency list

```
BFS(s, G):
    Discovered[v] = false, for all v
    Discovered[s] = true
    L[0] = {s}
    layer counter i = 0
    BFS tree T = {}
    while L[i] != {}
        L[i+1] = {}
        For each node u ∈ L[i]
            Consider each edge (u,v) incident to u
            if Discovered[v] == false then
                Discovered[v] = true
                Add edge (u, v) to tree T
                Add v to the list L[i + 1]
        i+=1
```

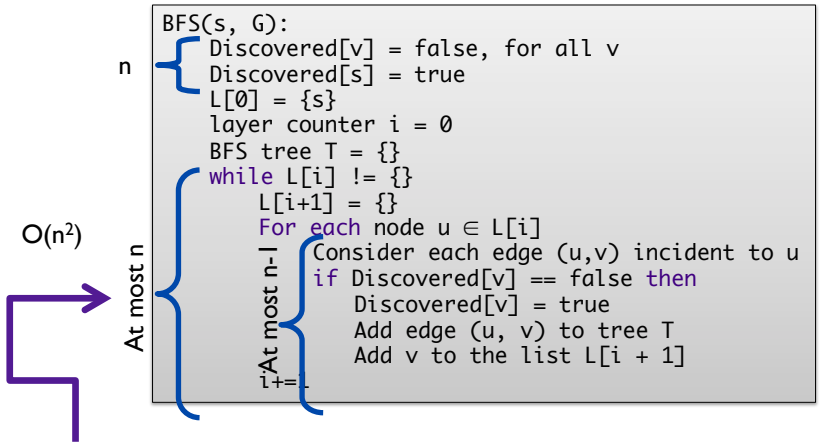
- L[i] representation? List, queue, or stack
  - Doesn't matter because algorithm can consider nodes in any order

What is the running time?

Analysis

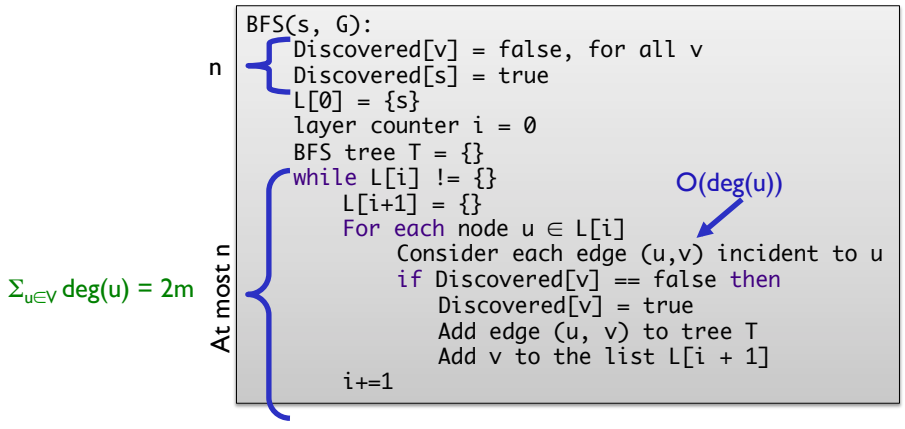


Analysis: Tighter Bound



Because we're going to look at each node at most once

Analysis: Even Tighter Bound



→ O(n+m)

Looking Ahead

- PS3 due Friday