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?
- What is a connected component?
  - What are two ways to find a connected component?
  - How are their results similar? Different?

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?

Review: Finding Connected Components

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

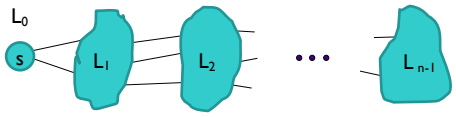
In general….

```
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?

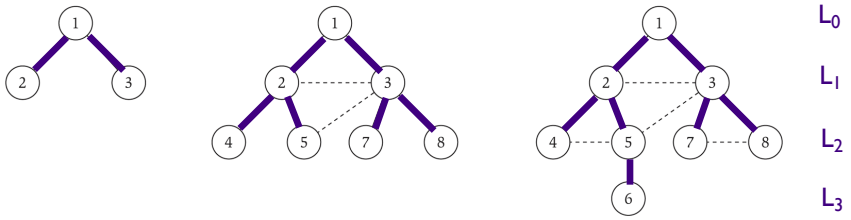
Review: Breadth-First Search

- **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



Review: Example of Breadth-First Search

s = 1

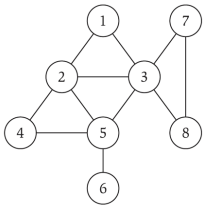


Creates a tree
-- is a node in the graph that is not in the tree

Review: Breadth-First Search

- **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.

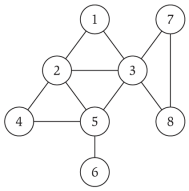
G:



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

Review: Depth-First Search

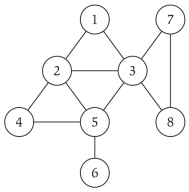


- Need to keep track of where you've been
- When reach a "dead-end" (already explored all neighbors), backtrack to node with unexplored neighbor
- **Algorithm**:

```
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)
```

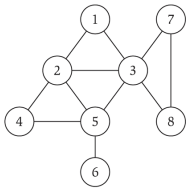
Depth-First Search

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



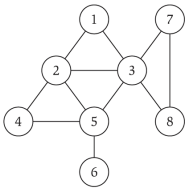
DFS vs BFS

- 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.

"equivalent" of BFS: connected nodes are at most one layer apart

What is an example of such an edge from the example we did?

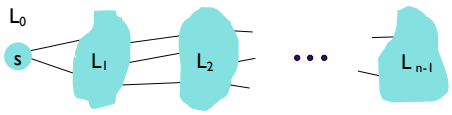
DFS Analysis

- 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, starting node
- Discovered array
- Maintain layers in separate lists, L[i]
- BFS Tree T

Implementing BFS

- Graph: Adjacency list
- Discovered array
- Maintain layers L[i], BFS Tree T

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

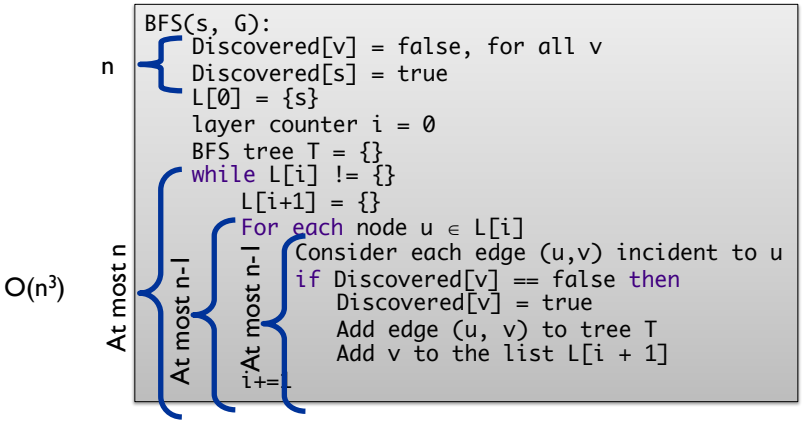
Given: s – start node, G – adjacency list

```
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
```

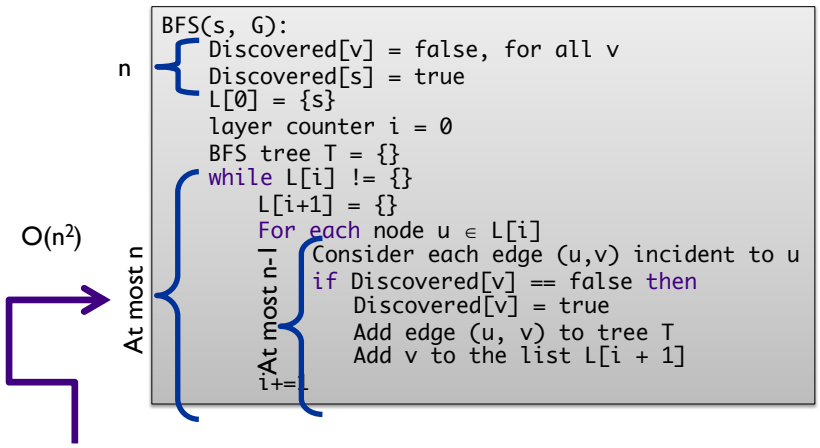
- L[i] representation? List, queue, or stack
  - Doesn't matter because algorithm can consider nodes in any order

What is the running time?

Analysis

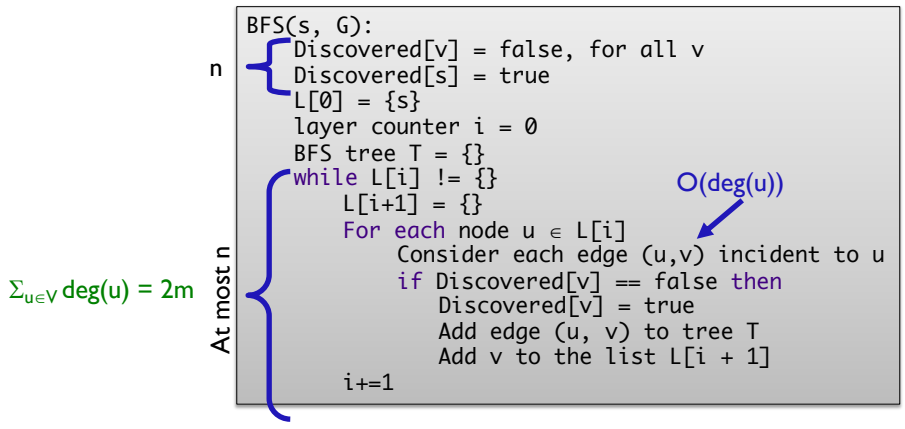


Analysis: Tighter Bound



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

Analysis: Even Tighter Bound



→ O(n+m)

Implementing DFS

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

```
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)
```


Implementing DFS

- Keep nodes to be processed in a *stack*

```
DFS(s, G):
   Initialize S to be a stack with one element s
   Explored[v] = false, for all v
   Parent[v] = 0, for all v
   DFS tree T = {}
   while S != {}
      Take a node u from S
      if Explored[u] = false
         Explored[u] = true
         Add edge (u, Parent[u]) to T (if u ≠ s)
         for each edge (u, v) incident to u
            Add v to the stack S
            Parent[v] = u
```

What is the runtime?
How many times is a node added/removed from the stack?

Analyzing DFS

O(n+m)

```
DFS(s, G):
    Initialize S to be a stack with one element s
    Explored[v] = false, for all v
    Parent[v] = 0, for all v                    } O(n)
    DFS tree T = {}
    while S != {}
        Take a node u from S
        if Explored[u] = false
            Explored[u] = true
            Add edge (u, Parent[u]) to T (if u ≠ s)
deg(u)      for each edge (u, v) incident to u
                Add v to the stack S
                Parent[v] = u
```

A node is added/removed from the stack 2* deg(u)
All nodes are added 2m = O(m) times

Looking Ahead

- PS3 due Friday