Objectives

- Wrapping up implementing BFS and DFS
- Graph Application: Bipartite Graphs

Review: Comparing BFS vs DFS

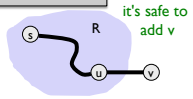
- What do they do?
- How are their outcomes different?
- When would we want to use one over the other?

Review: Comparing BFS vs DFS

- What do they do?
  - Techniques for finding connected components
    - Create a tree of connected components
  - Other uses as well
- How are their outcomes different?
  - BFS: shortest path; bushy tree
  - DFS: spindly tree
- When would we want to use one over the other?
  - BFS: Shortest path
  - DFS: what you'd do in a maze (can't split)

Review: A General Algorithm for Finding Connected Component

```
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
```



- *R* will be the **connected component** containing s
- Algorithm is underspecified
  - BFS and DFS say how to consider edges

Analysis of Connected Components

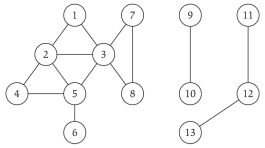
- For any two nodes *s* and *t* in a graph, their connected components are either identical or disjoint
- Proof?

Analysis of Connected Components

- For any two nodes *s* and *t* in a graph, their connected components are either identical or disjoint
- Proof sketch:
  - (i) There is a path between *s* and *t* → same set of connected components
  - (ii) There is no path between *s* and *t* → disjoint set of connected components

Set of All Connected Components

- How can we find set of ***all*** connected components of graph?



Set of All Connected Components

- How can we find set of all connected components of graph?

```
R* = set of connected components
while there is a node that does not belong to R*

    select s not in R*

    R = {s}

    while there is an edge (u,v) where u∈R and v∉R
       add v to R


    Add R to R*
```

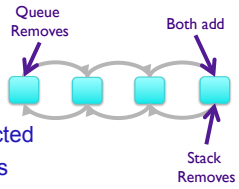
Find s's connected component

IMPLEMENTATION & ANALYSIS

Queues and Stacks

- How are queues and stacks similar?
- How are queues and stacks different?

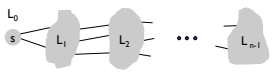
Queues and Stacks



- Both: doubly linked list
  - Always take first on list
  - Difference in where extracted
  - Have first and last pointers
  - Done in constant time
- Queue: FIFO
  - First in, first out
- Stack: LIFO
  - Last in, first out

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 in separate lists, 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
```


Analysis

```
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

$O(n^3)$

n

At most n

At most n-1

At most n-1

```
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
```


Analysis: Tighter Bound

$O(n^2)$

n

At most n

At most n-1

```
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
```

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

Analysis: Even Tighter Bound

```
BFS(s, G):
    Discovered[v] = false, for all v                    } n
    Discovered[s] = true
    L[0] = {s}
    layer counter i = 0
    BFS tree T = {}
    while L[i] != {}                                    } At most n
        L[i+1] = {}
        For each node u ∈ L[i]
            Consider each edge (u,v) incident to u      ← O(deg(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
```

$\Sigma_{u \in V} \deg(u) = 2m$

→ 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
```

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                      } O(n)
    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)
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 2m = O(m) times

Looking Ahead

- Reading Chapter 3
  - 3.1-3.3
- Problem Set 3 due before class on Friday