Objectives

- Wrap up: Implementing BFS and DFS
- Graph Application: Bipartite Graphs

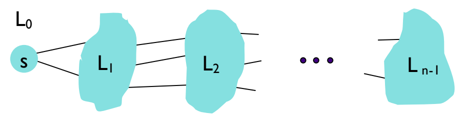
Turn in your problem set

Review

- What are two ways to find a connected component?
  - How are their results similar? Different?
- What was the runtime for BFS?
- What is the runtime for DFS?
  - Review what you have so far…

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

Analysis: Tighter Bound

n

O(n^2)

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

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

```
    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] != {}                        O(deg(u))
  ⎪       L[i+1] = {}
  ⎪       For each node u ∈ L[i]
  ⎪           Consider each edge (u,v) incident to u
At most n     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
```

→ O(n+m)

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

Analyzing
Finding All Connected Components

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

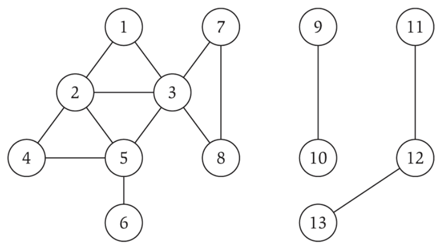
```
R* = set of connected components (a set of sets)
while there is a node that does not belong to R*
    select s not in R*
    R = {s}
    while there is an edge (u,v) where u∈R and v∉R
        add v to R

    Add R to R*
```

But the inner loop is O(m+n)!
How can this RT be possible?

Claim: Running time is O(m+n)

Consider: Finding All Connected Components for This Graph



- What would the process look like?
- What is the runtime for the major steps?

Set of All Connected Components

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

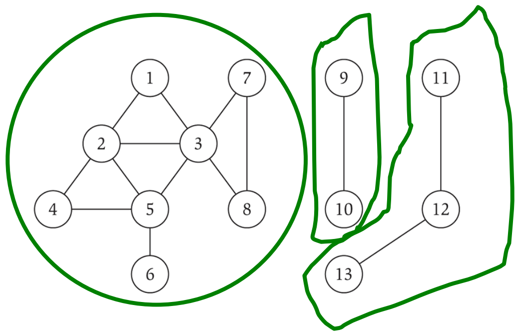
```
R* = set of connected components (a set of sets)
while there is a node that does not belong to R*
    select s not in R*
    R = {s}
    while there is an edge (u,v) where u∈R and v∉R
        add v to R
    Add R to R*
```

Imprecision in the running time of inner loop: O(m+n)

But that's m and n of the ***connected*** component, let's say m_i and n_i.
$\Sigma_i O(m_i + n_i) = O(m+n)$

Where *i* is the subscript of the connected component

Consider: Finding All Connected Components for This Graph



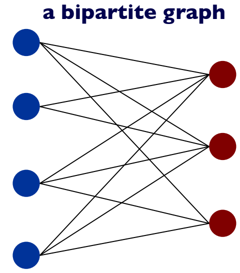
- Find each connected component
  - Runtime: that connected component's nodes and edges

BIPARTITE GRAPHS

Bipartite Graphs

- Def. An undirected graph G = (V, E) is ***bipartite*** if the nodes can be colored red or blue such that every edge has one red and one blue end
  - Generally: vertices divided into sets X and Y
- Applications:
  - Stable matching:
    - men = red, women = blue
  - Scheduling:
    - machines = red, jobs = blue

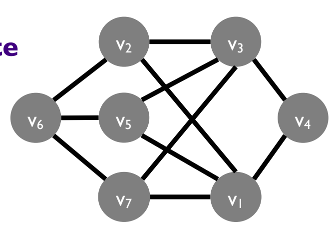
a bipartite graph



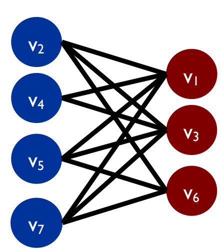
Testing Bipartiteness

- Given a graph G, is it bipartite?
- Many graph problems become:
  - Easier if underlying graph is bipartite (e.g., matching)
  - Tractable if underlying graph is bipartite (e.g., independent set)
- Before designing an algorithm, need to understand structure of bipartite graphs

a bipartite graph G:



another drawing of G:



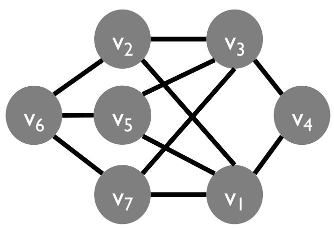
How Can We Determine if a Graph is Bipartite?

- Given a connected graph

Why connected?

1. Color one node red
    - Doesn't matter which color (Why?)

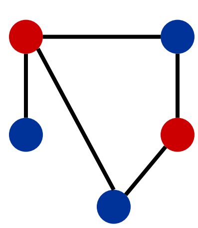
➢ What should we do next?



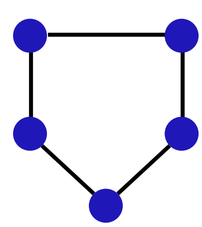
- How will we know when we're finished?
- What does this process sound like?

An Obstruction to Bipartiteness

- Lemma. If a graph G is bipartite, it cannot contain an odd-length cycle.



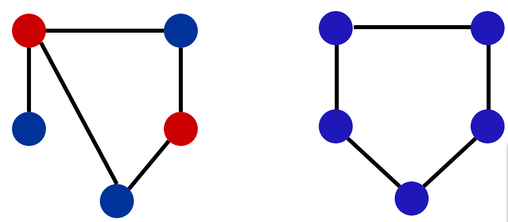
bipartite (2-colorable)



not bipartite (not 2-colorable)

An Obstruction to Bipartiteness

- Lemma. If a graph G is bipartite, it cannot contain an odd-length cycle.
- Pf. Not possible to 2-color the odd cycle, let alone G.



If find an odd cycle, graph is NOT bipartite

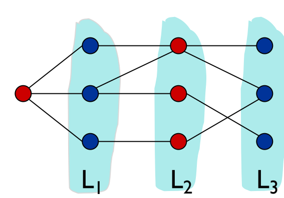
bipartite (2-colorable)

not bipartite (not 2-colorable)

How Can We Determine if a Graph is Bipartite?

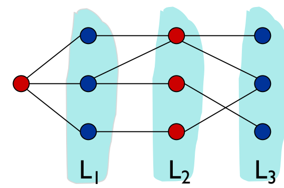
- Given a connected graph
  - Color one node red
    - Doesn't matter which color (Why?)
  - What should we do next?
- How will we know that we're finished?
- What does this process sound like?
  - BFS: alternating colors, layers

How can we implement the algorithm?



Implementing Algorithm

- Modify BFS to have a Color array
- When add v to list L[i+1]
  - Color[v] = red if i+1 is even
  - Color[v] = blue if i+1 is odd



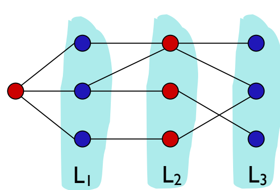
What is the running time of this algorithm? **O(n+m)**

Marks a change in how we think about algorithms
Starting to apply known algorithms to solve new problems

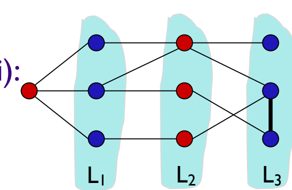
Analyzing Algorithm's Correctness

- Lemma. Let G be a connected graph, and let L_0, …, L_k be the layers produced by BFS starting at node *s*. Exactly one of the following holds:
  - (i) No edge of G joins two nodes of the same layer
    - ⇒ G is bipartite
  - (ii) An edge of G joins two nodes of the same layer
    - ⇒ G contains an odd-length cycle and hence is not bipartite

Case (i):

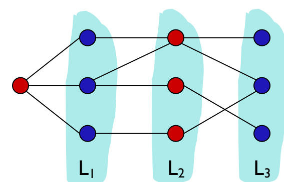


Case (ii):



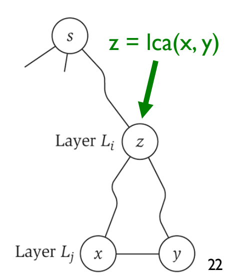
Analyzing Algorithm's Correctness

- Lemma. Let G be a connected graph, and let L_0, …, L_k be the layers produced by BFS starting at node *s*. Exactly one of the following holds:
  - (i) No edge of G joins two nodes of the same layer
    - ⇒ G is bipartite
- Pf. (i)
  - Suppose no edge joins two nodes in the same layer
  - Implies all edges join nodes on adjacent level
  - Bipartition
    - red = nodes on odd levels
    - blue = nodes on even levels



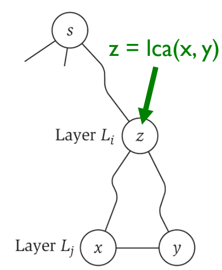
Analyzing Algorithm's Correctness

- Lemma. Let G be a connected graph, and let L_0, …, L_k be the layers produced by BFS starting at node *s*. Exactly one of the following holds:
  - (ii) An edge of G joins two nodes of the same layer → G contains an odd-length cycle and hence is not bipartite
- Pf. (ii)
  - Suppose (x, y) is an edge with x, y in same level L_j.
  - Let z = lca(x, y) = lowest common ancestor
  - Let L_i be level containing z
  - Consider cycle that takes edge from x to y, then path y →z, then path from z → x



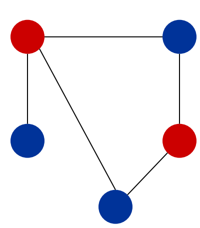
Analyzing Algorithm's Correctness

- Lemma. Let G be a connected graph, and let L_0, ..., L_k be the layers produced by BFS starting at node *s*. Exactly one of the following holds:
  - (ii) An edge of G joins two nodes of the same layer → G contains an odd-length cycle and hence is not bipartite
- Pf. (ii)
  - Suppose (x, y) is an edge with x, y in same level L_j.
  - Let z = lca(x, y)=lowest common ancestor
  - Let L_i be level containing z
  - Consider cycle that takes edge from x to y, then path y → z, then path z → x
  - Its length is $\underbrace{1}_{(x,\,y)} + \underbrace{(j-i)}_{\text{path from y to z}} + \underbrace{(j-i)}_{\text{path from z to x}}$, which is odd

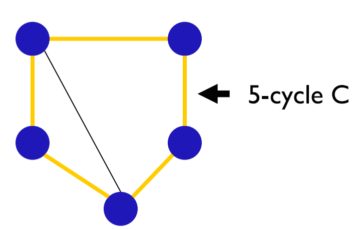


An Obstruction to Bipartiteness

- Corollary. A graph G is bipartite *iff* it contains no odd length cycle.



bipartite
(2-colorable)



not bipartite
(not 2-colorable)

Graph Summary So Far

- What do we know about graphs?

Graph Summary So Far

- What do we know about graphs?
  - Representation: Adjacency List, Space O(n+m)
  - Connectivity
    - BFS, DFS – O(n+m)
- Can apply BFS for Bipartite – O(n+m)

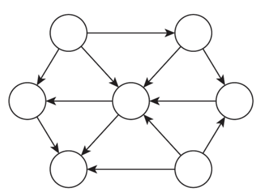
Second verse, similar to the first.

But directed!

DIRECTED GRAPHS

Directed Graphs G = (V, E)

- Edge (u, v) goes from node u to node v



- Example: Web graph - hyperlink points from one web page to another
  - Directedness of graph is crucial
  - Modern web search engines exploit hyperlink structure to rank web pages by importance

Representing Directed Graphs

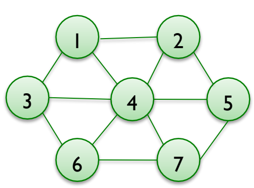
- For each node, keep track of
  - Out edges (where links go)
  - In edges (from where links come in)
  - Space required?
- Could only store *out* edges
  - Figure out *in* edges with increased computation/time
  - Useful to have both *in* and *out* edges

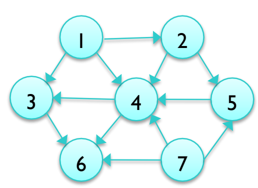
Rock Paper Scissors Lizard Spock

CONNECTIVITY IN DIRECTED GRAPHS

Graph Search

- How does ***reachability*** change with directed graphs?

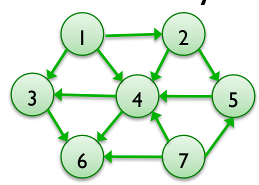




- Example: Web crawler
  1. Start from web page s.
  2. Find all web pages linked from s, either directly or indirectly.

Graph Search

- Directed reachability. Given a node *s*, find all nodes reachable from *s*.
- Directed *s*-*t* shortest path problem. Given two nodes *s* and *t*, what is the length of the shortest path between *s* and *t*?
  - Not necessarily the same as t→s shortest path
- Graph search. BFS and DFS extend naturally to directed graphs
  - Trace through out edges
  - Run in O(m+n) time



Problem

- Find all nodes with paths **to** *s*
  - Rather than paths from *s* to other nodes

Problem/Solution

- Problem. Find all nodes with paths **to** *s*
- Solution. Run BFS on *in edges* instead of out edges