Objectives

- BFS & DFS Implementations, Analysis
- Graph Applications: Bipartiteness
- Directed Graphs

Notes on Assignments

- Wiki: Any thoughts about using Dokuwiki for your notes?

Implementing DFS

- Keep nodes to be processed in a *stack*

```
DFS(s):
    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
```


Analyzing DFS

O(n+m)

```
DFS(s):
    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)
deg(u)      For each edge (u, v) incident to u
                Add v to the stack S
                Parent[v] = u
```


Analysis of Set of All Connected Components

```
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*
```

Running time: O(m+n)

But the "inner" loop was O(m+n)! How can this be?

Analysis of Set of All Connected Components

```
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*
```

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

That's m and n of the ***connected*** component, let's say m_i and n_i

So…

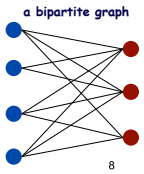
$\Sigma_i O(m_i + n_i) = O(m+n)$

Where i is the subscript of the connected component

TESTING BIPARTITENESS

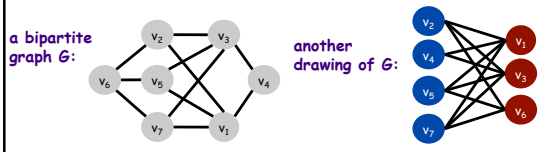
Bipartite Graphs

- 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 end and one blue end
  - Generally: vertices divided into sets X and Y
- Applications:
  - Stable marriage:
    - men = red, women = blue
  - Scheduling:
    - machines = red, jobs = blue



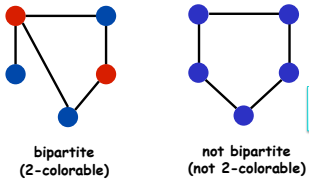
Testing Bipartiteness

- 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



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.

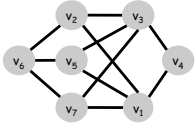


If find an odd cycle, graph is NOT bipartite

How Can We Determine if a Graph is Bipartite?

- Given a connected graph
  1. Color one node red
     - Doesn't matter which color (Why?)
  - What should we do next?

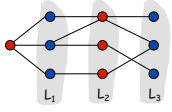
Why connected?



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

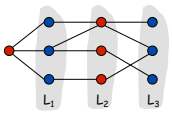
How Can We Determine if a Graph is Bipartite?

- 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



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

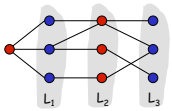


What is the running time of this algorithm? **O(n+m)**

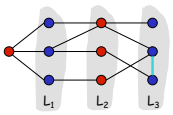
Analyzing Algorithm's Correctness

- 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):



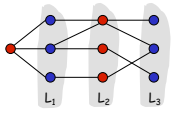
Case (ii):



Analyzing Algorithm's Correctness

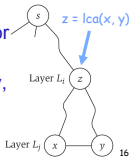
- Lemma. Let G be a connected graph, and let L_0, …, L_k be the layers produced by BFS starting at node *s*. Exactly one of the following holds:
  - (i) No edge of G joins two nodes of the same layer
    - G is bipartite
- Pf. (i)
  - Suppose no edge joins two nodes in the same layer
  - Implies all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

Case (i)



Analyzing Algorithm's Correctness

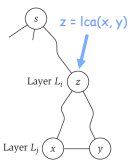
- Lemma. Let G be a connected graph, and let L_0, …, L_k be the layers produced by BFS starting at node *s*. Exactly one of the following holds:
  - (ii) An edge of G joins two nodes of the same layer → G contains an odd-length cycle and hence is not bipartite
- Pf. (ii)
  - Suppose (x, y) is an edge 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

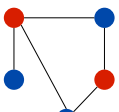
- Lemma. Let G be a connected graph, and let L_0, …, L_k be the layers produced by BFS starting at node *s*. Exactly one of the following holds:
  - (ii) An edge of G joins two nodes of the same layer → G contains an odd-length cycle and hence is not bipartite
- Pf. (ii)
  - Suppose (x, y) is an edge 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 1 + (j-i) + (j-i), which is odd

(x, y) path from y to z path from z to x

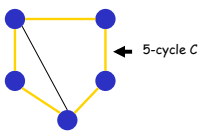


Obstruction to Bipartiteness

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



bipartite (2-colorable)

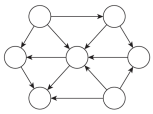


not bipartite (not 2-colorable)

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

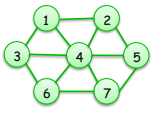
Representing Directed Graphs

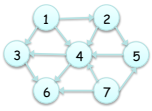
- For each node, keep track of
  - Out edges (where links go)
  - In edges (from where links come in)
- Could just keep out edges
  - Get in edges with increased computation/time
  - Useful to have both in and out edges

CONNECTIVITY IN DIRECTED GRAPHS

Graph Search

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

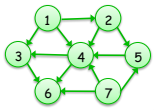




- Example: Web crawler
  1. Start from web page s.
  2. Find all web pages linked from s, either directly or indirectly.

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

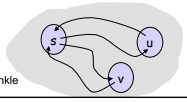
- Rather than paths from *s* to other nodes, find all nodes with paths to *s*

Problem/Solution

- Problem. Rather than paths from *s* to other nodes, find all nodes with paths to *s*
- Solution. Run BFS on *in edges* instead of out edges

Strong Connectivity

- Def. Node *u* and *v* are ***mutually reachable*** if there is a path from *u* → *v* and also a path from *v* → *u*
- Def. A graph is ***strongly connected*** if every pair of nodes is mutually reachable
- Lemma. Let *s* be any node. G is strongly connected ***iff*** every node is reachable from *s* and *s* is reachable from every node



Strong Connectivity

- If *u* and *v* are mutually reachable and *v* and *w* are mutually reachable, then *u* and *w* are mutually reachable

Strong Connectivity

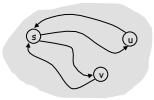
- If *u* and *v* are mutually reachable and *v* and *w* are mutually reachable, then *u* and *w* are mutually reachable.
- Proof. We need to show that there is a path from *u* → *w* and from *w* → *u*.
  - By defn of mutually reachable
    - there is a path *u* → *v* & a path *v* → *u,*
    - a path *v* → *w,* and a path *w* → *v*
  - Take path *u*→*v* and then from *v* → w
    - Path from *u*→*w*
  - Similarly for *w*→*u*

Strong Connectivity

- Def. A graph is strongly connected if every pair of nodes is mutually reachable
- Lemma. Let *s* be any node. G is ***strongly connected*** *iff* every node is reachable from *s* and *s* is reachable from every node.
  - 1st prove ⇒
  - 2nd prove ⇐
    - for any nodes *u* and *v,* is there a path *u*→*v* and *v*→*u* ?

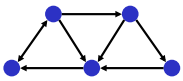
Strong Connectivity

- Def. A graph is strongly connected if every pair of nodes is mutually reachable
- Lemma. Let s be any node. G is ***strongly connected*** *iff* every node is reachable from s, and s is reachable from every node.
- Pf. ⇒ Follows from definition of strongly connected
- Pf. ⇐ For any nodes *u* and *v*, make path *u*→*v* and *v*→*u*
  - *u*→*v* : concatenating *u*→*s* with *s*→*v*
  - *v* →*u*: concatenate *v*→*s* with *s*→*u*

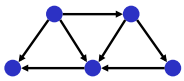


Strong Connectivity Problem

- Determine if G is strongly connected in O(m + n) time



strongly connected



not strongly connected

Hint: Can we leverage any algorithms we know have O(m+n) time?

Strong Connectivity: Algorithm

- Theorem. Can determine if G is strongly connected in O(m + n) time.
- Pf.
  - Pick any node *s*
  - Run BFS from *s* in G
  - Run BFS from *s* in Grev
  - Return true *iff* **all** nodes reached in both BFS executions
  - Correctness follows immediately from previous lemma
    - All reachable from one node, *s* is reached by all

reverse orientation of every edge in G
Or, the BFS using the *in* edges

PS1 Feedback

- Problem 1: Looking for an induction proof but I didn't really get that
- Problem 2: Straightforward adaptation of definitions
  - Trying to get you to review the definitions and get more comfortable with them
- Problems 3 & 4: Similar to one of the solved exercises
  - Take logs of functions to help see pattern
- Problem 5: Your solutions weren't quite right
  - Often going backwards
  - My drawing to try to trace through your algorithms
  - Analyze the running times of your solutions

Assignments

- Finish reading Chapter 3
  - Wikis for Wednesday
- For next Friday: Problem Set 3