Objectives

- BFS & DFS Implementations, Analysis
- Graph Application: Bipartiteness

Soap Opera Proofs

- “It’s the only thing that makes sense.”

Problem Set #1

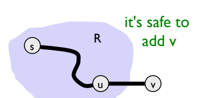
- $\sqrt{2n} < n + 10$

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: Finding Connected Components

```
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
```



DFS and BFS say what order we look at the edges.

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)

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 a graph?

Set of All Connected Components

- How can we find set of ***all*** connected components of a 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*
```

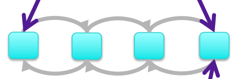

IMPLEMENTATION & ANALYSIS

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



Implementing BFS

- Graph: 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]

```
BFS(s):
   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
```

What does this stopping condition mean?

L[i] as a queue or stack?

Analysis

```
BFS(s):
   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] as a queue or stack?
  - Doesn't matter because algorithm can consider nodes in any order

What is the running time?

Analysis

```
BFS(s):
   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
```

n

At most n

At most n-1

$O(n^2)$

Analysis: Tighter Bound

```
BFS(s):
   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
```

n

At most n

O(deg(u))

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

$\rightarrow O(n+m)$

Implementing DFS

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
```


Set of All Connected Components

- How can we find set of all connected components of 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 was O(m+n)! How can this RT be possible?

Running time: O(m+n)

Set of All Connected Components

- How can we find set of all connected components of 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 . Therefore, $\Sigma_i O(m_i + n_i) = O(m+n)$

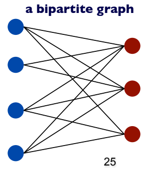
Where i is the subscript of the connected component

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 marriage:
    - men = red, women = blue
  - Scheduling:
    - machines = red, jobs = blue

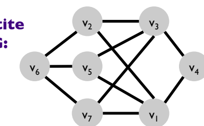
a bipartite graph



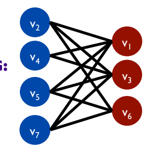
Testing Bipartiteness

- 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:

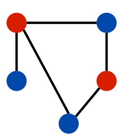


another drawing of G:

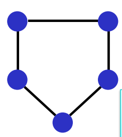


An Obstruction to Bipartiteness

- 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.



bipartite (2-colorable)



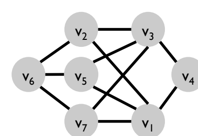
not bipartite (not 2-colorable)

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?

Reminders

- Friday: Problem Set 2 due