# COMPSCI 311 Introduction to Algorithms

## Lecture 5: Graph Traversal

Dan Sheldon

University of Massachusetts Amherst

# Graph Traversal

Thought experiment. World social graph.

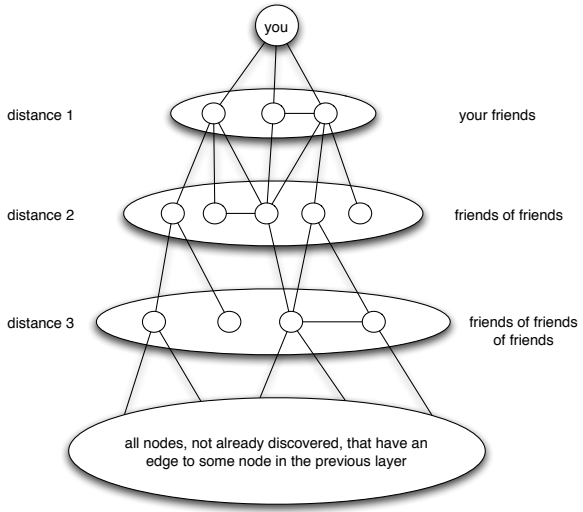
- Is it connected?
- If not, how big is largest connected component?
- Is there a path between you and Shohei Otani?

How can you tell algorithmically?

Answer: graph traversal! (BFS/DFS)

# Breadth-First Search

Explore outward from starting node $s$ by distance. "Expanding wave"



# Breadth-First Search: Layers

Explore outward from starting node $s$.

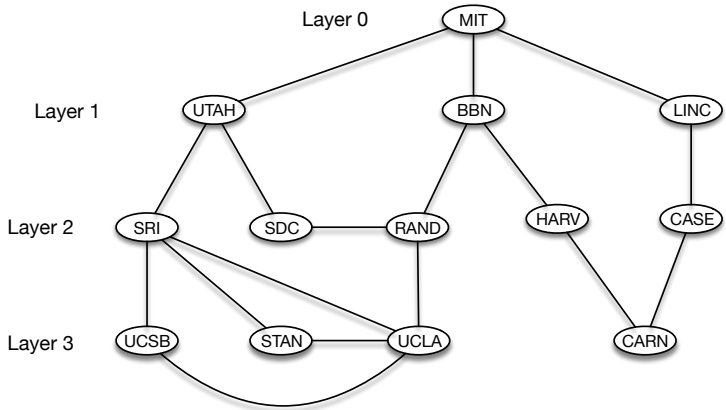
Define layer $L_i$ = all nodes at distance exactly $i$ from $s$.

**Layers**

- $L_0 = \{s\}$
- $L_1$ = nodes with edge to $L_0$
- $L_2$ = nodes with an edge to $L_1$ that don't belong to $L_0$ or $L_1$
- ...
- $L_{i+1}$ = nodes with an edge to $L_i$ that don't belong to any earlier layer.

Observation: There is a path from $s$ to $t$ if and only if $t$ appears in some layer.

# BFS Layers


Layer 0
MIT
Layer 1
UTAH
BBN
LINC
Layer 2
SRI
SDC
RAND
HARV
CASE
Layer 3
UCSB
STAN
UCLA
CARN

# BFS Implementation

BFS($s$):

```
mark s as "discovered"
L[0] ← {s}, i ← 0                                   ▷ Discover s
while L[i] is not empty do
    L[i + 1] ← empty list
    for all nodes v in L[i] do
        for all neighbors w of v do                  ▷ Explore v
            if w is not marked "discovered" then
                mark w as "discovered"               ▷ Discover w
                put w in L[i + 1]
    i ← i + 1
```

Running time? How many times does each line execute?
(For now, assume graph is connected)

# BFS Running Time

BFS($s$):

```
mark s as "discovered"                                          ▷ 1
L[0] ← {s}, i ← 0                                               ▷ 1
while L[i] is not empty do                                      ▷ ≤ n
    L[i + 1] ← empty list                                       ▷ ≤ n
    for all nodes v in L[i] do                                  ▷ n
        for all neighbors w of v do                             ▷ 2m
            if w is not marked "discovered" then                ▷ 2m
                mark w as "discovered"                          ▷ n
                put w in L[i + 1]                               ▷ n
    i ← i + 1                                                   ▷ ≤ n
```

Running time: $\Theta(m + n)$

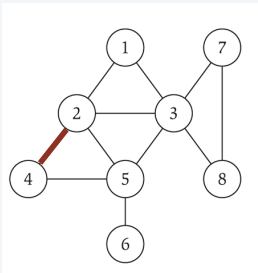
# BFS Running Time

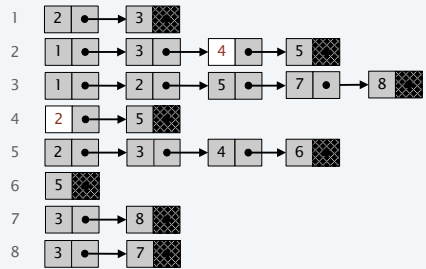
BFS running time: $\Theta(m + n)$

- Another way to think about it: "touch each node and edge" a constant number of times
- Hidden assumption: can iterate over neighbors of $v$ efficiently...

# Graph Representation: Adjacency Lists

Each node keeps **list of neighbors**




- Each edge stored twice
- Space? $\Theta(m+n)$
- Time to check if $(u, v)$ is an edge? $O(\text{degree}(u))$
  (degree = number of neighbors)
- Time to iterate over all neighbors of $v$? $O(\text{degree}(u))$

# Clicker

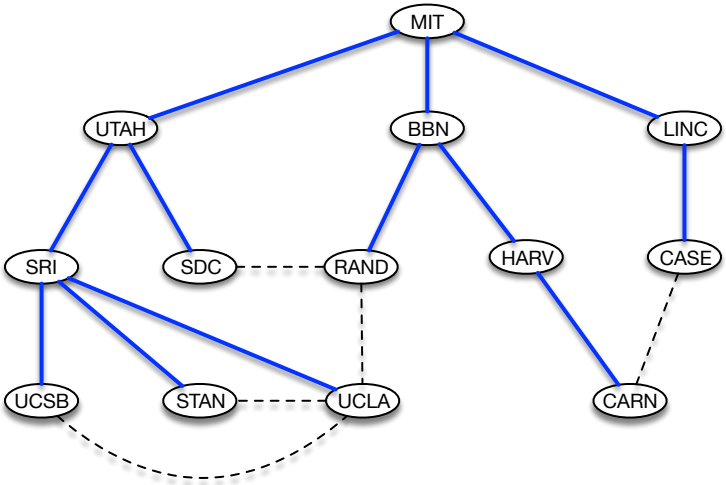
Let $q = \sum_{v \in V} \text{degree}(v)$ (this is the sum of degrees of all nodes in the graph)

Which one of the following is false?

A. $q$ is twice the number of edges
B. $q$ is $n$ times the average degree
C. $q$ is $\Theta(m + n)$ if $m \geq n$
D. None of the above

# BFS Tree

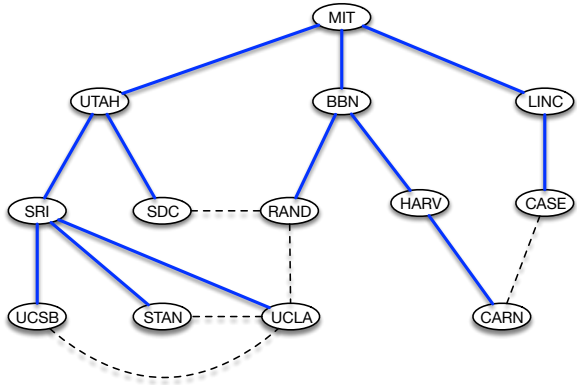
We can use BFS to make a tree. (blue: "tree edges", dashed: "non-tree edges")



# BFS Tree

BFS($s$):
  mark $s$ as "discovered"
  $L[0] \leftarrow \{s\}$, $i \leftarrow 0$
  $T \leftarrow$ empty
  **while** $L[i]$ is not empty **do**
    $L[i+1] \leftarrow$ empty list
    **for all** nodes $v$ in $L[i]$ **do**
      **for all** neighbors $w$ of $v$ **do**
        **if** $w$ is not marked "discovered" **then**
          mark $w$ as "discovered"
          put $w$ in $L[i+1]$
          put $(v, w)$ in $T$
    $i \leftarrow i+1$

# BFS Tree




**Claim**: let $T$ be the tree discovered by BFS on graph $G = (V, E)$, and let $(x, y)$ be any edge of $G$. Then the layer of $x$ and $y$ in $T$ differ by at most 1.

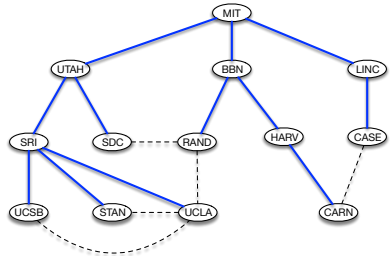
# BFS and non-tree edges

**Claim**: let $T$ be the tree discovered by BFS on graph $G = (V, E)$, and let $(x, y)$ be any edge of $G$. Then the layer of $x$ and $y$ in $T$ differ by at most 1.

**Proof**

- Let $(x, y)$ be an edge
- Assume $x$ is discovered first and placed in $L_i$
- Then $y \in L_j$ for $j \geq i$
- When neighbors of $x$ are explored, $y$ is either already in $L_i$ or $L_{i+1}$, or is discovered and added to $L_{i+1}$

# Clicker




Suppose in BFS that the nodes in each layer are explored in a different order (e.g. reverse). Which of the following are true?

A. The nodes that appear in each layer may change

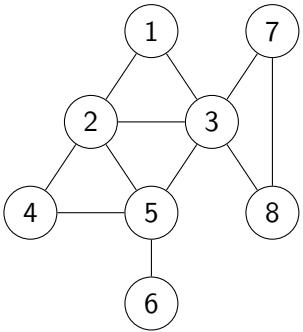
B. The BFS tree may change

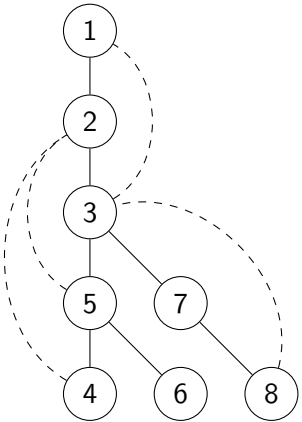
C. Both A and B

D. Neither A nor B

# Depth-First Search

Depth-first search (DFS): keep exploring from the most recently added node until you have to backtrack.




Dotted edges: to already explored nodes

# DFS: Recursive Implementation

DFS($u$)

mark $u$ as "explored"
**for all** edges $(u, v)$ **do**
**if** $v$ is not "explored" **then**
call DFS($v$) recursively

# DFS: Running Time

How to analyze if algorithm is recursive? Same: count executions of each line, across *all* recursive calls

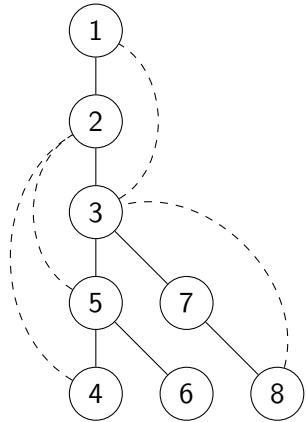
DFS($u$)

```
mark u as "explored"                                   ▷ n
for all edges (u, v) do                                ▷ 2m
    if v is not "explored" then                        ▷ 2m
        call DFS(v) recursively                        ▷ n
```

Running time: $O(m+n)$ same as BFS

# DFS Tree

$T \leftarrow$ empty
DFS($u$)
  mark $u$ as "explored"
  **for all** edges $(u, v)$ **do**
    **if** $v$ is not "explored" **then**
      put $(u, v)$ in $T$
      call DFS($v$) recursively




**Claim:** Non-tree edges lead to (indirect) **ancestors**

# DFS: Non-tree edges lead to ancestors

**Claim:** Let $T$ be the tree discovered by DFS, and let $(x, y)$ be an edge of $G$ that is not in $T$. Then one of $x$ or $y$ is an ancestor of the other.

**Proof:**

- Let $x$ be the first of the two nodes explored
- Is $y$ explored at beginning of DFS($x$)? No.
- At some point *during* DFS($x$), we examine the edge $(x, y)$. Is $y$ explored then? Yes, otherwise we would put $(x, y)$ in $T$
- $\Rightarrow$ $y$ was explored *during* DFS($x$)
- $\Rightarrow$ $y$ is a descendant of $x$

# Generic Traversal Implementations

Generic approach: maintain set of explored nodes and discovered nodes

- Explored = have seen this node and explored its outgoing edges
- Discovered = the "frontier". Have seen the node, but not explored its outgoing edges.

# Generic Graph Traversal

Let $A$ = data structure of discovered nodes

Traverse($s$)

```
put s in A
while A is not empty do
    take a node v from A
    if v is not marked "explored" then
        mark v "explored"
        for each edge (v, w) incident to v do
            put w in A                          ▷ w is discovered
```

BFS: $A$ is a queue (FIFO)  DFS: $A$ is a stack (LIFO)

# Clicker

```
put s in A
while A is not empty do
    take a node v from A
    if v is not marked "explored" then
        mark v "explored"
        for each edge (v, w) incident to v do
            put w in A                                  ▷ w is discovered
```

Suppose we run this traversal code and every node is marked explored before it terminates. Which of the following is false?

A. Every node is marked "explored" exactly once.
B. A single node could be put into $A$ more than once.
C. If $w \neq s$, the number of times that node $w$ is put into $A$ is degree$(w)$.
D. It's possible that there exist nodes $x$ and $y$ with no path from $x$ to $y$.

# Exploring *all* Connected Components

How to explore entire graph even if it is disconnected?

**while** there is some unexplored node $s$ **do**
  Traverse($s$) ▷ Run BFS/DFS starting from $s$.
  Extract connected component containing $s$

Running time? Still $O(m + n)$

- Traversal of each component takes time proportional to the numbers of nodes + edges in *that* component

**Advice**: usually OK to assume graph is connected. State if you are doing so and why it does not trivialize the problem.

# Summary

- Graph traversal by BFS/DFS: basic algorithmic primitive used in many other algorithms
  - Is there a path from $u$ to $v$?
  - Find all connected components
  - Produce trees with different properties, sometimes useful in algorithms
- $\Theta(m + n)$ time
- Different versions of general exploration strategy