# CMPSCI 311: Introduction to Algorithms

Akshay Krishnamurthy and Andrew McGregor

University of Massachusetts

Last Compiled: September 13, 2016

## Announcements

- Homework 1 released (Due 9/23 8pm)
- Discussion section tomorrow
- Updated Office Hours
  - Akshay: Tuesday 1-2pm CS 258
  - Andrew: Thursday 3-4pm CS 334
  - Archan: Friday 10-11am CS 207
  - Walter: Wednesday 3-4pm CS 207

## Plan

- Review: Asymptotics
  - $O(\cdot), \Omega(\cdot), \Theta(\cdot)$
  - Running time analysis
- Graphs
  - Motivation and definitions
  - Graph traversal
    - Breadth-First-Search (BFS)
    - Depth-First-Search
    - An Application
    - Implementation

## Review: Asymptotics

**Definition** $f(n) = O(g(n))$ if there exists $n_0, c$ such that for all $n \geq n_0$, $f(n) \leq cg(n)$.

- $g$ is an asymptotic upper bound on $f$.

**Definition** $f(n) = \Omega(g(n))$ if $g(n) = O(f(n))$.

- $g$ is an asymptotic lower bound on $f$.

**Definition** $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $g(n) = O(f(n))$.

- $g$ is an asymptotically tight bound on $f$.

## Algorithm design

- Formulate the problem precisely
- Design an algorithm to solve the problem
- Prove the algorithm is correct
- Analyze the algorithm's running time

## Running Time Analysis

Mathematical analysis of worst-case running time of an algorithm as function of input size. Why these choices?

- Mathematical: describes the *algorithm*. Avoids hard-to-control experimental factors (CPU, programming language, quality of implementation), while still being predictive.
- Worst-case: just works. ("average case" appealing, but hard to analyze)
- Function of input size: allows predictions. What will happen on a new input?

## Efficiency

When is an algorithm efficient?

Stable Matching Brute force: $\Omega(n!)$
Propose-and-Reject?: $O(n^2)$
We must have done something clever

## Polynomial Time

Working definition of efficient

**Definition**: an algorithm runs in polynomial time if the number of primitive execution steps is at most $cn^d$, where $n$ is the input size and $c$ and $d$ are constants.

- Matches practice: almost all practically efficient algorithms have this property
- Usually distinguishes a clever algorithm from a "brute force" approach ($n^d = O(2^n)$ for all constant $d$).
- Refutable: gives us a way of saying an algorithm is not efficient, or that no efficient algorithm exists.

## Plan

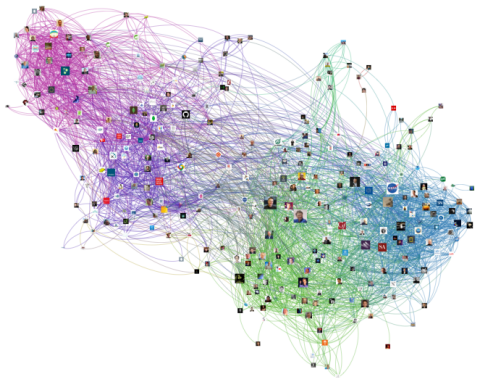
- Review: Asymptotics
  - $O(\cdot), \Omega(\cdot), \Theta(\cdot)$
  - Running time analysis
- Graphs
  - Motivation and definitions
  - Graph traversal
    - Breadth-First-Search (BFS)
    - Depth-First-Search
    - An Application
    - Implementation

## Questions

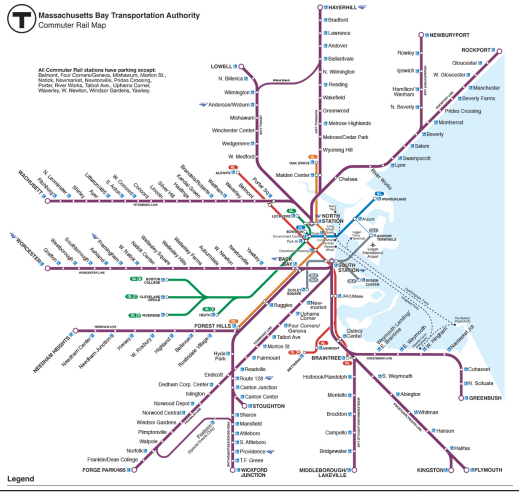
- Facebook: how many "degrees of separation" between me and Barack Obama?
- Google Maps: what is the shortest driving route from South Hadley to Florida?

Can we build algorithms to answer these questions?

## Networks



## Networks



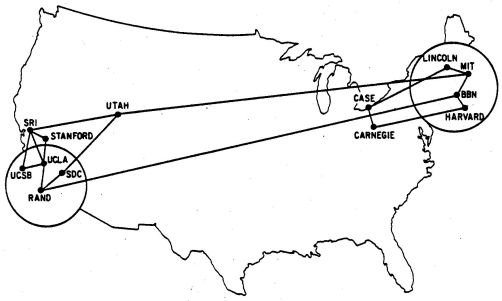
## Graphs

A graph is a mathematical representation of a network

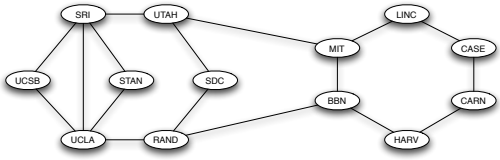
- Set of nodes (vertices) $V$
- Set of pairs of nodes (edges) $E$

Graph $G = (V, E)$

## Example: Internet in 1970

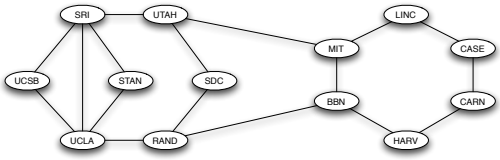



## Example: Internet in 1970



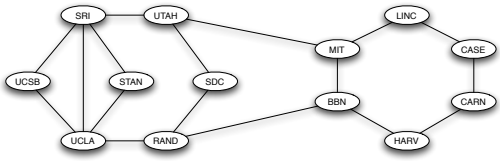

**Definitions**:
Edge $e = (u, v)$. Neighbor, incident, endpoints

## Example: Internet in 1970



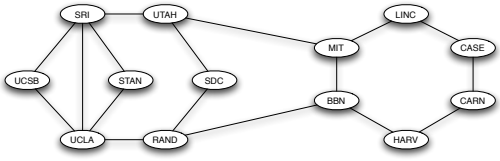

**Definitions**:
Path, cycle, path length, distance between two nodes

## Example: Internet in 1970




**Definitions**:
Connected. Connected components.

## Example: Internet in 1970




**Definitions**:
Tree = a connected undirected graph that does not contain a cycle
Rooted vs. unrooted trees

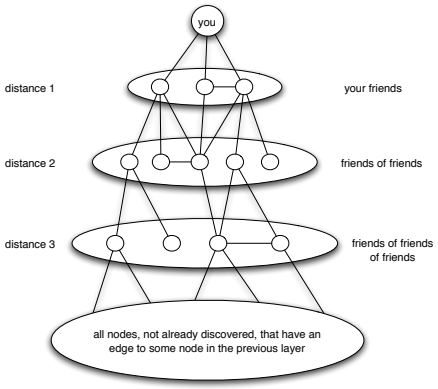
## Graph Traversal

Thought experiment. World social graph. Is it connected? Is there a path between you and Barack Obama? How can you tell?

Answer: graph traversal! (BFS/DFS)

## Breadth First Search

Traverse graph by exploring outward from starting node by distance. "Expanding wave"




## Breadth-First Search: Layers

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

**Layers**

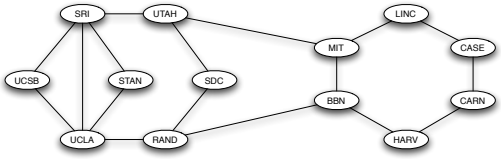
- $L_0 = \{s\}$
- $L_1$ = all neighbors of $L_0$
- $L_2$ = all nodes with an edge to $L_1$ that don't belong to $L_0$ or $L_1$
- ...
- $L_{i+1}$ = nodes with an edge to $L_i$ that don't belong to any earlier layer.

$$L_{i+1} = \{v : \exists (u, v) \in E, u \in L_i, v \notin (L_0 \cup \ldots \cup L_i)\}$$

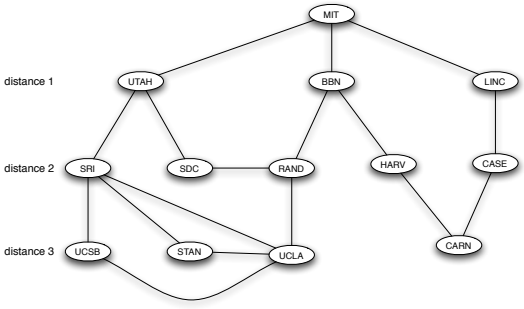
Observation: There is a path from $s$ to $t$ if and only if $t$ appears in some layer.

## BFS

Exercise: draw the BFS layers for a BFS starting from MIT

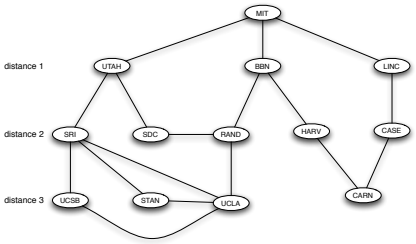



## BFS Tree




We can use BFS to make a tree.

## 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.
Proof on board

## 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**

- Suppose $x \in L_i$ and $y \in L_j$ with $i < j - 1$ but edge $(x, y)$ exists.
- When BFS visits $x$, either $y$ is already discovered or not.
  - If $y$ is already discovered, then $j \leq i$. Contradiction.
  - Otherwise since $(x, y) \in E$, $y$ is added to $L_{i+1}$. Contradiction.

## A More General Strategy

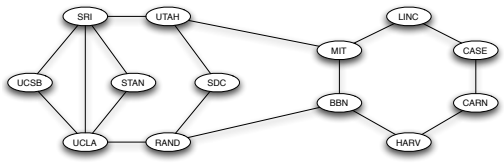
To explore the connected component, add any node $v$ for which

- $(u, v)$ is an edge
- $u$ is explored, but $v$ is not

Picture on board

## DFS

Depth-first search: keep exploring from the most recently added node until you have to backtrack.
**Example**.




## Recursive DFS

DFS($u$)
  Mark $u$ as "Explored"
  **for** each edge $(u, v)$ incident to $u$ **do**
    **if** $v$ is not marked "Explored" **then**
      Recursively invoke DFS($v$)
    **end if**
  **end for**

Example on board

## DFS Tree

**Claim**: let $T$ be a depth-first search tree for graph $G = (V, E)$, and let $(x, y)$ be an edge that is in $G$ but not $T$ (a "non-tree edge"). Then either $x$ is an ancestor of $y$ or $y$ is an ancestor of $x$ in $T$.
proof on board

## DFS and Non-tree edges

**Claim**: let $T$ be a depth-first search tree for graph $G = (V, E)$, and let $(x, y)$ be an edge that is in $G$ but not $T$ (a "non-tree edge"). Then either $x$ is an ancestor of $y$ or $y$ is an ancestor of $x$ in $T$.
**Proof**

- Suppose not and suppose that $x$ is reached first by DFS.
- Before leaving $x$, we must examine $(x, y)$.
- Since $(x, y) \notin T$, $y$ must have been explored by this time.
- But $y$ was not explored when we arrived at $x$ by assumption.
- Thus $y$ was explored during the execution of DFS($x$).
- Implies $x$ is ancestor of $y$.

## Using Graph Traversal

**Definition**: the connected component $C(v)$ of node $v$ is the set of all nodes with a path to $v$.

Easy claim: for any two nodes $s$ and $t$ either $C(s) = C(t)$, or $C(s)$ and $C(t)$ are disjoint.

Picture/example on board

## Finding Connected Components

Traverse entire graph even if not connected.
Extract connected components.

**while** There is some unexplored node $s$ **do**
  BFS($s$) ▷ Run BFS starting from $s$.
  Extract connected component $C(s)$.
**end while**

Running time?
What's the running time of BFS?

## Summary So Far

- Graph – definitions
- Graph traversals – BFS, DFS, and some properties
- Finding connected components

- Next – Implementation and run-time analysis.

## Representing a graph

*Adjacency List* Representation.

- Nodes numbered $1, \ldots, n$.
- `Adj`$[v]$ points to a list of all of $v$'s neighbors.
- Example

## Implementing BFS

Maintain set of explored nodes and discovered

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

Picture on board

## BFS Implementation

Let $A$ = Queue of discovered nodes (FIFO)
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 $v$ as "explored"
      **for** each edge $(v, w)$ incident to $v$ **do**
        Put $w$ in $A$ ▷ $w$ is discovered
      **end for**
    **end if**
  **end while**

Note: one part of this algorithm seems really dumb. Why?
Can put multiple copies of a node in $A$. ("Rediscover it many times")

## BFS Implementation

Let $A$ = Queue of discovered nodes (FIFO)
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 $v$ as "explored"
      **for** each edge $(v, w)$ incident to $v$ **do**
        Put $w$ in $A$ ▷ $w$ is discovered
      **end for**
    **end if**
  **end while**

Is this BFS?

## Summary

**Definitions**

- $G = (V, E), n = |V|, m = |E|$
- neighbor, incident, cycle, path, connected

**BFS and DFS**

- Two ways to traverse a graph, each produces a tree
- BFS tree: shallow and wide ("bushy")
- DFS tree: deep and narrow ("scraggly")
- Connected Components