COMPSCI 311 Introduction to Algorithms

Lecture 3: Asymptotic Complexity
Graphs and Breadth-First Search

Marius Minea

University of Massachusetts Amherst

slides credit: Dan Sheldon, Akshay Krishnamurthy, Andrew McGregor

30 January 2019

Big-Θ

Definition: the function $T(n)$ is $\Theta(f(n))$ if there exist positive constants c_1, c_2 and n_0 such that

$$c_1 f(n) \le T(n) \le c_2 f(n) \text{ for all } n \ge n_0$$

f is an asymptotically tight bound of T

Equivalent Definition: the function $T(n)$ is $\Theta(f(n))$ if it is both $O(f(n))$ and $\Omega(f(n))$.

Example. $f(n) = 32n^2 + 17n + 1$

- $f(n)$ is $\Theta(n^2)$
- $f(n)$ is neither $\Theta(n)$ nor $\Theta(n^3)$

Clicker Question 1

Which of the following implies that $f(n)$ is $\Theta(g(n))$:

A. $f(n)$ is $\Theta(g(n))$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ for some constant $0 < c < \infty$

B. $f(n)$ is $\Theta(g(n))$ if there exist constants $c_1, c_2 > 0$ such that $c_1 \cdot g(n) \le f(n) \le c_2 \cdot g(n)$ for infinitely many n

C. Both A and B

D. Neither A nor B

Big-Θ example

How do we correctly compare the running time of these algorithms?

Algorithm **foo**

```
for  $i$= 1 to $n$ do
    for  $j$= 1 to $n$ do
        do something...
    end for
end for
```

Algorithm **bar**

```
for  $i$= 1 to $n$ do
    for  $j$= 1 to $n$ do
        for  $k$= 1 to $n$ do
            do something else..
        end for
    end for
end for
```

Answer: **foo** is $\Theta(n^2)$ and **bar** is $\Theta(n^3)$.
They do not have the same asymptotic running time.

Additivity Revisited

Suppose f and g are two (non-negative) functions and f is $O(g)$

Old version: Then $f + g$ is $O(g)$

New version: Then $f + g$ is $\Theta(g)$

Example:

$$\underbrace{n^2}_{g} + \underbrace{42n + n\log n}_{f} \text{ is } \Theta(n^2)$$

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

Question: Is it $\Omega(n^2)$?

Polynomial Time

Definition: an algorithm runs in polynomial time if its running time is $O(n^d)$ for some constant d

- Examples

These are polynomial time:
$f_1(n) = n$
$f_2(n) = 4n + 100$
$f_3(n) = n\log(n) + 2n + 20$
$f_4(n) = 0.01n^2$
$f_5(n) = n^2$
$f_6(n) = 20n^2 + 2n + 3$

Not polynomial time:
$f_7(n) = 2^n$
$f_8(n) = 3^n$
$f_9(n) = n!$

Why Polynomial Time ?

Why is this a good definition of efficiency?

- Matches practice: almost all practically efficient algorithms have this property.
- Usually distinguishes a clever algorithm from a “brute force” approach.
- Refutable: gives us a way of saying an algorithm is not efficient, or that no efficient algorithm exists.

Exponential time

An algorithm is *exponential time* if it is $O(2^{n^k})$ for some $k > 0$

Useful fact: (Stirling’s approximation)

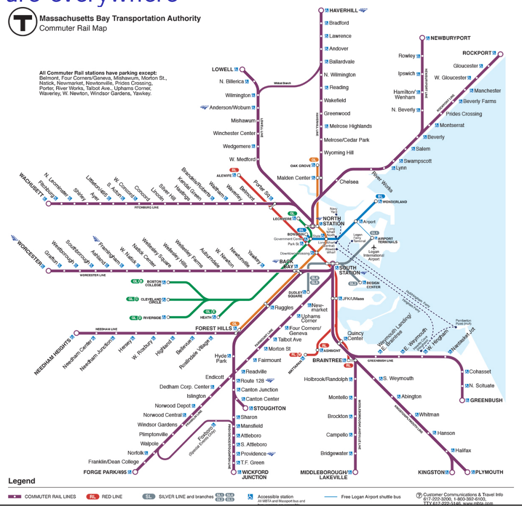
$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \qquad \text{(ratio tends to 1)}$$

Exercise: What can you claim from here for big-O (and later big-Θ)?

Review: Asymptotics

Property	Definition / terminology
$f(n)$ is $O(g(n))$	$\exists c, n_0$ s.t. $f(n) \le cg(n)$ for all $n \ge n_0$ g is an asymptotic upper bound on f
$f(n)$ is $\Omega(g(n))$	$\exists c, n_0$ s.t. $f(n) \ge cg(n)$ for all $n \ge n_0$ Equivalently: $g(n)$ is $O(f(n))$ g is an asymptotic lower bound on f
$f(n)$ is $\Theta(g(n))$	$f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$ g is an asymptotically tight bound on f

Graphs are everywhere

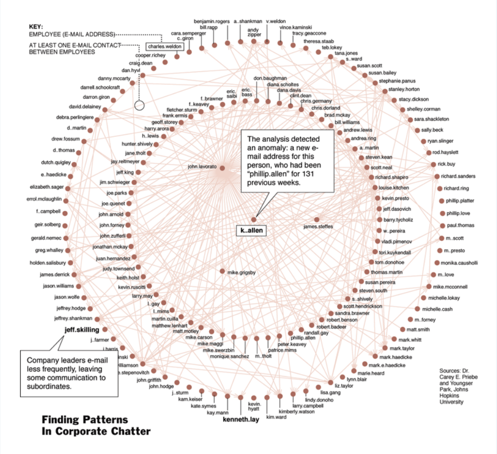


Some graphs

- Transportation networks: hubs, links, routes
- Communication networks: routing, how many hops, latency/throughput?
- Information networks:
  WWW, what are important/authoritative pages?
- Social networks: study interaction dynamics, find influencers?

How do we build algorithms to answer these questions?

One week of Enron emails



The analysis detected an anomaly: a new e-mail address for this person, who had been "phillip.allen" for 131 previous weeks.

Company leaders e-mail less frequently, leaving some communication to subordinates.

Finding Patterns In Corporate Chatter

Sources: Dr. Carey E. Priebe and Youngser Park, Johns Hopkins University

slide credit: Kevin Wayne / Pearson

Framingham heart study

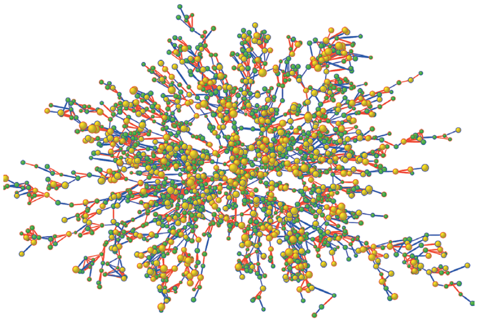


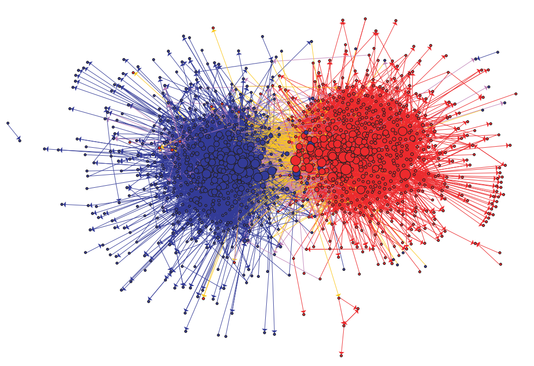
Figure 1. Largest Connected Subcomponent of the Social Network in the Framingham Heart Study in the Year 2000. Each circle (node) represents one person in the data set. There are 2200 persons in this subcomponent of the social network. Circles with red borders denote women, and circles with blue borders denote men. The size of each circle is proportional to the person's body-mass index. The interior color of the circles indicates the person's obesity status: yellow denotes an obese person (body-mass index, ≥30) and green denotes a nonobese person. The colors of the ties between the nodes indicate the relationship between them: purple denotes a friendship or marital tie and orange denotes a familial tie.

"The Spread of Obesity in a Large Social Network over 32 Years" by Christakis and Fowler in New England Journal of Medicine, 2007

slide credit: Kevin Wayne / Pearson

Political blogosphere graph

Node = political blog; edge = link.



The Political Blogosphere and the 2004 U.S. Election: Divided They Blog, Adamic and Glance, 2005

slide credit: Kevin Wayne / Pearson

More applications

- Network science
  - random graphs: various evolution models
  - scale-free, small world
- Analyzing graph evolution in time
  - fake news
  - botnets
- Analyzing programs
  - control flow graph, function call graph
  - state space search (also in games): compute reachable states (configurations) is an error state reachable?

Graphs

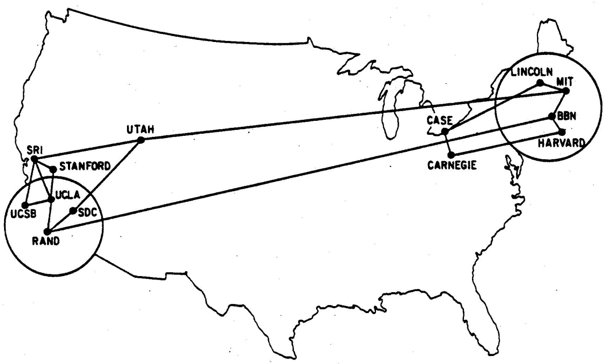
A graph is a mathematical representation of a network

- Set of nodes (vertices) V
- Set of pairs of nodes (edges) E (a *relation*)

Graph $G = (V, E)$

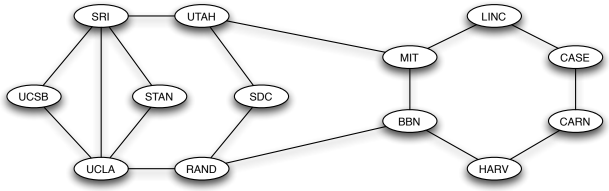
Notation: $n = |V|$, $m = |E|$ (almost always used)

Example: Internet in 1970



Definitions: Edge, Path

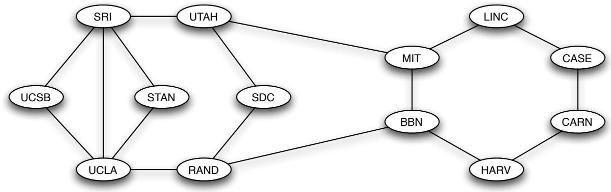
Edge $e = \{u, v\}$ — but usually written $e = (u, v)$
u and v are *neighbors*, *adjacent*, *endpoints* of e
e is *incident* to u and v



A **path** is a sequence $P = v_1, v_2, \ldots, v_{k-1}, v_k$ such that each consecutive pair v_i, v_{i+1} is joined by an edge in G

Called: path "from v_1 to v_k". Or: a v_1–v_k path

Clicker Question 2



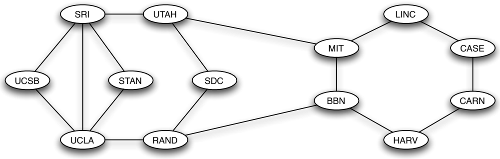
Q: Which is not a path?

1. UCSB - SRI - UTAH
2. BBN - HARV - BBN - MIT
3. STAN - SRI - UCLA - UCSB - SRI
4. All of the above are paths

Simple path, distance, cycle

- **Simple path**: path where all vertices are distinct
  - Exercise. Prove: If there is a path from u to v then there is a simple path from u to v.
- **Distance** from u to v: minimum number of edges in a u–v path
- **(Simple) Cycle**: path $v_1, \ldots, v_{k-1}, v_k$ where
  - $v_1 = v_k$
  - First $k - 1$ nodes distinct
  - All edges distinct

Connected components



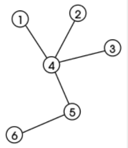
Connected graph = graph with paths between every pair of vertices.

- **Connected component**: maximal subset of nodes such that a path exists between each pair in the set
- maximal = if a new node is added to the set, there will no longer be a path between each pair

Trees

Tree = a connected graph with no cycles

Q: Is this equivalent to trees seen in Data Structures?
A: More or less.



Tree properties
Let G be an undirected graph with n nodes.
Then any two of the following statements implies the third:

- G is connected
- G does not contain a cycle
- G has $n - 1$ edges

Rooted tree: tree with parent-child relationship

- Pick root r and "orient" all edges away from root
- Parent of v = predecessor on path from r to v

Directed Graphs

- **Directed graph** $G = (V, E)$
    - *Directed* edge $e = (u, v)$ is now an *ordered* pair
    - e *leaves* u (source) and *enters* v (sink)
- **Directed path, cycle**: same as before, but with directed edges
- **Strongly connected**: directed graph with directed path between every pair of vertices
- Note: graphs *undirected* if not otherwise specified

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 <*some famous person*>?

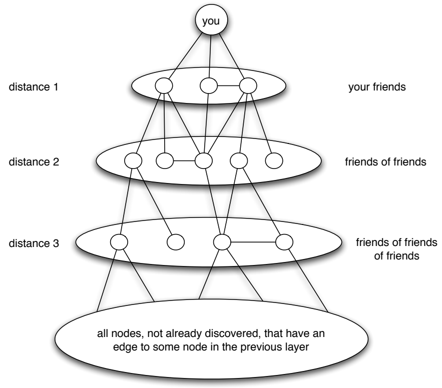
"Six degrees of separation" (everyone connected in at most 6 links?)
Erdös number: coauthorship of scientific papers

How can you tell algorithmically?

Answer: graph traversal! (BFS/DFS)

Breadth-First Search

Explore outward from starting node 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.

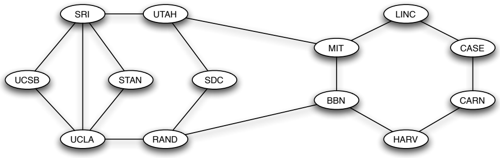
Layers

- $L_0 = \{s\}$
- 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.

Clicker Question 3

How many nodes are in layer 2, starting a BFS from UTAH ?



A) 4

B) 5

C) 6

D) None of the above

BFS Implementation

```
BFS($s$):
  mark $s$ as "discovered"
  $L[0] \leftarrow \{s\}$, $i \leftarrow 0$                         ▷ Discover $s$
  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                ▷ Explore $v$
              if $w$ is not marked "discovered" then
                  mark $w$ as "discovered"               ▷ Discover $w$
                  put $w$ in $L[i+1]$
              end if
          end for
      end for
      $i \leftarrow i+1$
  end while
```

Running time? How many times does each line execute?

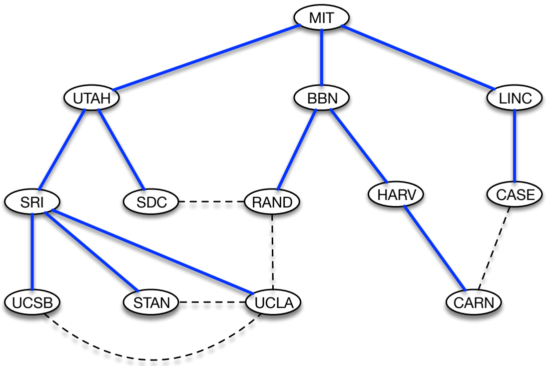
BFS Running Time

```
BFS($s$):
    mark $s$ as "discovered"                                  ▷ $1$
    $L[0] \leftarrow \{s\}$, $i \leftarrow 0$                 ▷ $1$
    while $L[i]$ is not empty do
        $L[i+1] \leftarrow$ empty list                        ▷ $\leq n$
        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 $L[i+1]$                       ▷ $n$
                end if
            end for
        end for
        $i \leftarrow i+1$                                    ▷ $\leq n$
    end while
```

Running time: $O(m+n)$. Hidden assumption: can iterate over neighbors of v efficiently... OK pending data structure.

BFS Tree

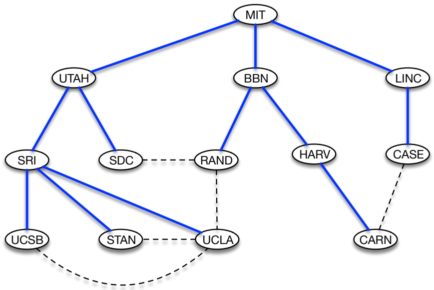
We can use BFS to make a tree. (blue: "tree edges", dashed: "non-tree edges")



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 $T$
                end if
            end for
        end for
        $i \leftarrow i+1$
    end while
```

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 is discovered and added to L_{i+1}

Exploring *all* Connected Components

How to explore entire graph even if it is disconnected?

```
while there is some unexplored node $s$ do
    BFS($s$)                                     ▷ Run BFS starting from $s$.
    Extract connected component containing $s$
end while
```

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

Running time? Does it change?