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

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?

Big-Θ
Definition: the function $T(n)$ is $Θ(f(n))$ if there exist positive constants $c_1$, $c_2$ and $n_0$ such that
\[ c_1 f(n) \leq T(n) \leq c_2 f(n) \text{ for all } n \geq n_0 \]

$f$ is an asymptotically tight bound of $T$

Equivalent Definition: the function $T(n)$ is $Θ(f(n))$ if it is both $O(f(n))$ and $Ω(f(n))$.

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

▶ $f(n)$ is $Θ(n^2)$
▶ $f(n)$ is neither $Θ(n)$ nor $Θ(n^3)$

Clicker Question 1
Which of the following implies that $f(n)$ is $Θ(g(n))$:

A. $f(n)$ is $Θ(g(n))$ if $\lim_{n \to \infty} \frac{f(n)}{g(n)} = c$ for some constant $0 < c < \infty$
B. $f(n)$ is $Θ(g(n))$ if there exist constants $c_1$, $c_2 > 0$ such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for infinitely many $n$
C. Both A and B
D. Neither A nor B

Running Time Analysis
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 $Θ(g)$

Example:

\[ \frac{n^2 + 42n + n \log n}{f} \] is $Θ(n^2)$

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 $Θ(g)$

Example:

\[ \frac{n^2 + 42n + n \log n}{f} \] is $Θ(n^2)$

Big-Θ example

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

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

Algorithm bar
for $i=1$ to $n$
for $j=1$ to $n$
do something else..
end for
end for

do something else..
end for

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

Big-Θ
Definition: the function $T(n)$ is $Θ(f(n))$ if there exist positive constants $c_1$, $c_2$ and $n_0$ such that
\[ c_1 f(n) \leq T(n) \leq c_2 f(n) \text{ for all } n \geq n_0 \]

$f$ is an asymptotically tight bound of $T$

Equivalent Definition: the function $T(n)$ is $Θ(f(n))$ if it is both $O(f(n))$ and $Ω(f(n))$.

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

▶ $f(n)$ is $Θ(n^2)$
▶ $f(n)$ is neither $Θ(n)$ nor $Θ(n^3)$

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 $Θ(g)$

Example:

\[ \frac{n^2 + 42n + n \log n}{f} \] is $Θ(n^2)$

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 $O(nd)$ 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:

- $f_7(n) = 2n$
- $f_8(n) = 3n$
- $f_9(n) = n!$

Exponential time

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

Useful fact: (Stirling’s approximation)

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

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

Review: Asymptotics

<table>
<thead>
<tr>
<th>Property</th>
<th>Definition / terminology</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(n)$ is $O(g(n))$</td>
<td>$\exists c, n_0 \text{ s.t. } f(n) \leq cg(n) \text{ for all } n \geq n_0$ $g$ is an asymptotic upper bound on $f$</td>
</tr>
<tr>
<td>$f(n)$ is $\Omega(g(n))$</td>
<td>$\exists c, n_0 \text{ s.t. } f(n) \geq cg(n) \text{ for all } n \geq n_0$ Equivalently: $g(n)$ is $O(f(n))$ $g$ is an asymptotic lower bound on $f$</td>
</tr>
<tr>
<td>$f(n)$ is $\Theta(g(n))$</td>
<td>$f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$ $g$ is an asymptotically tight bound on $f$</td>
</tr>
</tbody>
</table>

Graphs are everywhere
One week of Enron emails

Framingham heart study

The Spread of Obesity in a Large Social Network over 32 Years

by Christakis and Fowler in New England Journal of Medicine, 2007

How do we build algorithms to answer these questions?

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)
Chapter 2. Graphs

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 $P = v_1, v_2, \ldots, v_k$ 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 $v_1 \rightarrow v_k$ path

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 $u \rightarrow v$ path
- (Simple) Cycle: path $v_1, \ldots, v_k$ where
  - $v_1 = v_k$
  - First $k - 1$ nodes distinct
  - All edges distinct

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?)
Erdős number: coauthorship of scientific papers

How can you tell algorithmically?

Answer: graph traversal! (BFS/DFS)

Directed Graphs

Directed Graphs

Breadth-First Search: Layers

Explore outward from starting nodes.

Define layer $L_i = \text{all nodes at distance exactly } i \text{ from } s$.

Layers

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

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

Breadth-First Search: Layers

Explore outward from starting node $s$.

Define layer $L_i = \text{all nodes at distance exactly } i \text{ from } s$.

Layers

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

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

BFS Implementation

BFS($s$):

1. mark $s$ as "discovered"
2. $L[0] \leftarrow \{ s \}$, $i \leftarrow 0$
3. while $L[i]$ is not empty do
   - $L[i+1] \leftarrow \text{empty list}$
   - 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 $L[i+1]$
     - end if
   - end for
   - $i \leftarrow i + 1$
   - $L[i+1] \leftarrow \text{empty list}$
   - for all nodes $v$ in $L[i]$ do
     - if $v$ is not marked "discovered" then
       - mark $v$ as "discovered"
       - $L[i+1] \leftarrow \text{empty list}$
     - end if
   - end for
4. end while

Running time? How many times does each line execute?

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 Running Time

BFS(s):
mark s as "discovered"
L[0] ← {s}, i ← 0
while L[i] is not empty do
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]
        end if
    end for
end for
i ← i + 1
end while

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

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 ∈ L_j for j ≥ i
- When neighbors of x are explored, y is either already in L_i, or is discovered and added to L_{i+1}

Running time? Does it change?

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?