IE170: Algorithms in Systems Engineering: Lecture 33

Jeff Linderoth

Department of Industrial and Systems Engineering
Lehigh University

April 25, 2007

What We've Learned – Part One

1. Summation Formulae, Induction and Bounding
2. How to compare functions: $o, \omega, O, \Omega, \Theta$
3. How to count the running time of algorithms
4. How to solve recurrences that occur when we do (3)
5. Data Structures
   - Hash
   - Binary Search Trees
   - Heaps

What We've Learned – Part Deux

- Dynamic Programming (15.[1,3])
- Greedy Algorithms (16.[1,2])
- Graphs and Search (22.*)
- Spanning Trees (23.*)
- (Single Source) Shortest Paths (24.[1,2,3])
- (All Pairs) Shortest Paths (25.[1,2])
- Max Flow (26.[1,2,3])

Stuff To Know: EVERYTHING!

DP and Greedy

- Develop (and potentially solve small) problems via DP
- Activity Selection (or related problems): Greedy Works

Graphs

- BFS, DFS, and Analysis.
- Classifying edges in directed and undirected graphs
- Topological Sorting
- Finding Strongly Connected Components

Spanning Trees

- Kruskal's Algorithm (and analysis)
- Prim's Algorithm (and analysis)

More Stuff To Know...

Single Source Shortest Paths

- Distance Labels and RELAX
- Path Relaxation Property
- Bellman-Ford Algorithm
  - How to do it
  - When (Why?) it works
  - Analysis
- SSSP Dag
  - How to do it
  - When (Why?) it works
  - Analysis
- Dijkstra's Algorithm
  - How to do it
  - When (Why?) it works
  - Analysis

Even More Stuff To Know...

All Pairs Shortest Paths

- Analogue to Matrix Multiplication
- Floyd-Warshall
  - How to do it?
  - When (Why?) it works?
  - Analysis

Flows

- What is a flow?
- What is a cut?
- What is MFMC Theorem?
- How to create residual graph G_f?
- How to do Augmenting Paths algorithm (Ford Fulkerson/Edmonds Karp)
- Analysis

What We've Learned, Part Trois

- Matrix Review.
  - Linear (in)dependence, positive definiteness, singularity, range, null-space, etc.
- Matrix manipulation: Matrix Multiplication
- Solving Triangular Systems
- Cholesky Factorization (Least Squares)
- Gaussian Elimination
  - Relationship to LU-factorization
- $PA = LU$

O, Ω, Θ definitions

$$\Theta(g) = \{f : \exists\, c_1, c_2, n_0 > 0 \text{ such that } c_1 g(n) \leq f(n) \leq c_2 g(n)\ \forall n \geq n_0\}$$

$$\Omega(g) = \{f \mid \exists \text{ constants } c, n_0 > 0 \text{ s.t. } 0 \leq cg(n) \leq f(n)\ \forall n \geq n_0\}$$

$$O(g) = \{f \mid \exists \text{ constants } c, n_0 > 0 \text{ s.t. } f(n) \leq cg(n)\ \forall n \geq n_0\}$$

o, ω Notation

$$f \in o(g) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

$$f \in \omega(g) \Leftrightarrow g \in o(f) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

$$f \in \Theta(g) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = c$$

- $f \in o(g) \Rightarrow f \in O(g) \setminus \Theta(g)$.
- $f \in \omega(g) \Rightarrow f \in O(g) \setminus \Theta(g)$.
- $f \in \Theta(g) \Leftrightarrow g \in \Theta(f)$

Remember This!

The Upshot!

- $f \in O(g)$ is like "$f \leq g$,"
- $f \in \Omega(g)$ is like "$f \geq g$,"
- $f \in o(g)$ is like "$f < g$,"
- $f \in \omega(g)$ is like "$f > g$," and
- $f \in \Theta(g)$ is like "$f = g$."

Functions

- Polynomials f of degree k are in $\Theta(n^k)$.
- Exponential functions always grow faster than polynomials
- Polylogarithmic functions always grow more slowly than polynomials.

Count 'em Up

- You should be able to look at a short code module, and write down how many times each line is done.
- Like the InsertionSort, MergeSort, and Towers of Hanoi examples in class.
- If the algorithm is recursive, you should be able to look at the recurrence and compute its running time

Analyzing Recurrences

Deep Thoughts

To understand recursion, we must first understand recursion

- General methods for analyzing recurrences
  - Substitution
  - Master Theorem
- When we analyze a recurrence, we may not get or need an exact answer, only an asymptotic one

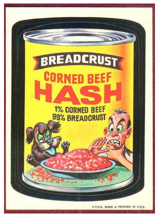
The Master Theorem

- Most recurrences that we will be interested in are of the form

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ aT(n/b) + f(n) & n > 1 \end{cases}$$

- The Master Theorem tells us how to analyze recurrences of this form.
- If $f \in O(n^{\log_b a - \epsilon})$, for some constant $\epsilon > 0$, then $T \in \Theta(n^{\log_b a})$.
- If $f \in \Theta(n^{\log_b a})$, then $T \in \Theta(n^{\log_b a} \lg n)$.
- If $f \in \Omega(n^{\log_b a + \epsilon})$, for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and $n > n_0$, then $T \in \Theta(f)$.

More on Hash



- In a hash table the number of keys stored is small relative to the number of possible keys
- A hash table is an array. Given a key k, we don't use k as the index into the array – rather, we have a hash function h, and we use $h(k)$ as an index into the array.
- Given a "universe" of keys K.
  - Think of K as all the words in a dictionary, for example
- $h : K \to \{0, 1, \ldots m - 1\}$, so that $h(k)$ gets mapped to an integer between 0 and $m - 1$ for every $k \in K$
- We say that k hashes to $h(k)$

Storing Binary Trees

Array

- The root is stored in position 0.
- The children of the node in position i are stored in positions $2i + 1$ and $2i + 2$.
- This determines a unique storage location for every node in the tree and makes it easy to find a node's parent and children.
- Using an array, the basic operations can be performed very efficiently.

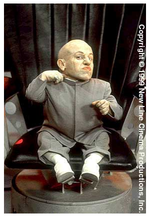
Binary Search Tree

- A binary search tree is a data structrue that is conceptualized as a binary tree, but has one additional property:

Binary Search Tree Property

If y is in the left subtree of x, then $k(y) \leq k(x)$

Short Is Beautiful



- SEARCH() takes $O(h)$
- MINIMUM(), MAXIMUM() also take $O(h)$
- Slightly less obvious is that INSERT(), DELETE() also take $O(h)$
- Thus we would like to keep out binary search trees "short" (h is small).

Sorted

- We saw in the lab that the Java Tree Set allowed you to iterate through the list in sorted order. How long does it take to do this?

```
INORDER-TREE-WALK(x)
if x ≠ NIL
  then INORDER-TREE-WALK(ℓ(x))
       print k(x)
       INORDER-TREE-WALK(r(x))
```

- What is running time of this algorithm?

Operations

SUCCESSOR(x)

- How would I know "next biggest" element?
- If right subtree is not empty: MINIMUM($r(x)$)
- If right subtree is empty: Walk up tree until you make the first "right" move

INSERT(x)

- Just walk down the tree and put it in. It will go "at the bottom"

DELETE()

- If 0 or 1 child, deletion is fairly easy
- If 2 children, deletion is made easier by the following fact:

Binary Search Tree Property

- If a node has 2 children, then
  - its successor will not have a left child
  - its predecessor will not have a right child

Heaps

- Heaps are a bit like binary search trees, however, they enforce a different property

Heap Property: Children are Horrible!

- In a max-heap, the key of the parent node is always at least as big as its children:

$$k(p(x)) \geq k(x) \quad \forall x \neq root$$

Heapify

HEAPIFY(x)

1. Find largest of $k(x)$, $k(\ell(x))$, $k(r(x))$
2. If $k(x)$ is largest, you are done
3. Swap x with largest node, and call HEAPIFY() on the new subtree

- $\Rightarrow$ HEAPIFY a node in $O(\lg n)$
- Alternatively, HEAPIFY node of height h is $O(h)$
- Building a heap out of an array of size n takes $O(n)$

Operations on a Heap

- The node with the highest key is always the root.
- To delete a record
  - Exchange its record with that of a leaf.
  - Delete the leaf.
  - Call `heapify()`.
- To add a record
  - Create a new leaf.
  - Exchange the new record with that of the parent node if it has a higher key.
  - This is like insertion sort – just move it up the path...
  - Continue to do this until all nodes have the heap property.
  - Note that we can change the key of a node in a similar fashion.

Time for Heap Operations

CREATE	$O(n)$
MAXIMUM	$\Theta(1)$
HEAPIFY	$O(\lg n)$, or $O(h)$
EXTRACT-MAX	$O(\lg n)$
HEAP-INCREASE-KEY	$O(\lg n)$
INSERT	$O(\lg n)$

Heap Sort

- Suppose the list of items to be sorted are in an array of size n
- The heap sort algorithm is as follows.
  1. Put the array in heap order as described above.
  2. In the i^{th} iteration, exchange the item in position 0 with the item in position $n - i$ and call `heapify()`.
- What is the running time? $\Theta(n \lg n)$

Dynamic Programming

Dynamic Programming in a Nutshell

1. Characterize the structure of an optimal solution
2. Recursively define the value of an optimal solution
3. Compute the value of an optimal solution "from the bottum up"
4. Construct optimal solution (if required)

Examples

- Assembly Line Balancing
- Lot Sizing

Assembly Line Balancing

- Let $f_i(j)$ be the fastest time to get through $S_{ij}\ \forall i = 1,2\ \forall j = 1,2,\dots n$

$$
\begin{aligned}
f^* &= \min(f_1(n) + x_1, f_2(n) + x_2) \\
f_1(1) &= e_1 + a_{11} \\
f_2(1) &= e_2 + a_{21} \\
f_1(j) &= \min(f_1(j-1) + a_{1j}, f_2(j-1) + t_{2,j-1} + a_{1j}) \\
f_2(j) &= \min(f_2(j-1) + a_{2j}, f_1(j-1) + t_{1,j-1} + a_{2j})
\end{aligned}
$$

Lot Sizing

- Let $f_t(s)$: be the minimum cost of meeting demands from $t, t+1, \dots T$ (t until the end) if s units are in inventory at the beginning of period t

$$f_t(s) = \min_{x \in 0,1,2,\dots} \{c_t(x) + h_t(s + x - d_t) + f_{t+1}(s + x - d_t)\}.$$

Greedy

- Greedy is not always optimal!
- But it sometimes works:

Activity Selection

- Let $S_{ij} \subseteq \mathcal{A}$ be the set of activities that start after activity i needs to finish and before activity j needs to start:

$$S_{ij} \stackrel{\text{def}}{=} \{k \in S \mid f_i \leq s_k, f_k \leq s_j\}$$

- Let's assume that we have sorted the activities such that

$$f_1 \leq f_2 \leq \cdots \leq f_n$$

- Schedule jobs in $S_{0,n+1}$

- c_{ij} be the size of a maximum-sized subset of mutually compatible jobs in S_{ij}.
- If $S_{ij} = \emptyset$, then $c_{ij} = 0$
- If $S_{ij} \neq \emptyset$, then $c_{ij} = c_{ik} + 1 + c_{kj}$ for some $k \in S_{ij}$. We pick the $k \in S_{ij}$ that maximizes the number of jobs:

$$c_{ij} = \begin{cases} 0 & \text{if } S_{ij} = \emptyset \\ \max_{k \in S_{ij}} c_{ik} + c_{kj} + 1 & \text{if } S_{ij} \neq \emptyset \end{cases}$$

- Note we need only check $i < k < j$

To Solve S_{ij}

1. Choose $m \in S_{ij}$ with the earliest finish time. The Greedy Choice
2. Then solve problem on jobs S_{mj}

Graphs!

- Adjacency List, Adjacency Matrix
- Breadth First Search
- Depth First Search

BFS

- Input: Graph $G = (V, E)$, source node $s \in V$
- Output: $d(v)$, distance (smallest # of edges) from s to v $\forall v \in V$
- Output: $\pi(v)$, predecessor of v on the shortest path from s to v

BFS

$\text{BFS}(V, E, s)$

for each u **in** $V \setminus \{s\}$
do $d(u) \leftarrow \infty$
  $\pi(u) \leftarrow \text{NIL}$
$d[s] \leftarrow 0$
$Q \leftarrow \emptyset$
$\text{ADD}(Q, s)$
while $Q \neq \emptyset$
do $u \leftarrow \text{POLL}(Q)$
  for each v **in** $Adj[u]$
  do if $d[v] = \infty$
    then $d[v] \leftarrow d[u] + 1$
      $\pi[v] = u$
      $\text{ADD}(Q, v)$

DFS

DFS

- Input: Graph $G = (V, E)$
- Output: Two timestamps for each node $d(v)$, $f(v)$,
- Output: $\pi(v)$, predecessor of v
  - not on shortest path necessarily

```
DFS(V, E)
for each u in V
do color(u) ← GREEN
    π(u) ← NIL
time ← 0
for each u in V
do if color[u] = GREEN
      then DFS-VISIT(u)
```


DFS (Visit Node—Recursive)

```
DFS-VISIT(u)
color(u) ← YELLOW
d[u] ← time++
for each v in Adj[u]
do if color[v] = GREEN
      then π[v] ← u
           DFS-VISIT(v)

color(u) ← RED
f[u] = time++
```


Classifying Edges in the DFS Tree

Given a DFS Tree G_π, there are four type of edges (u, v)

1. **Tree Edges**: Edges in E_π. These are found by exploring (u, v) in the DFS procedure
2. **Back Edges**: Connect u to an ancestor v in a DFS tree
3. **Forward Edges**: Connect u to a descendent v in a DFS tree
4. **Cross Edges**: All other edges. They *can* be edges in the same DFS tree, or can cross trees in the DFS forest G_π

Modifying DFS to Classify Edges

- DFS can be modified to classify edges as it encounters them...
- Classify $e = (u, v)$ based on the color of v when e is first explored...
- GREEN: Indicates Tree Edge
- YELLOW: Indicates Back Edge
- RED: Indicates Forward or Cross Edge

Stuff You Can Do with DFS

Topological Sort: The Whole Algorithm

1. DFS search the graph
2. List vertices in order of decreasing finishing time

Strongly Connected Components

1. Call $\mathrm{DFS}(G)$ to topologically sort G
2. Compute G^T
3. Call $\mathrm{DFS}(G^T)$ but consider vertices in topologically sorteded order (from G)
4. Vertices in each tree of depth-first forest for SCC

Spanning Tree

Kruskal's Algorithm

1. Start with each vertex being its own component
2. Merge two components into one by choosing the light edge that connects them
3. Scans the set of edges in increasing order of weight

Prim's Algorithm

- Builds one tree, so A is always a tree
- Let V_A be the set of vertices on which A is incident
- Start from an arbitrary root r
- At each step find a light edge crossing the cut $(V_A, V \setminus V_A)$

Kruskal's Algorithm

```
KRUSKAL(V, E, w)
A ← ∅
for each v in V
do MAKE-SET(v)
SORT(E, w)
for each (u, v) in (sorted) E
do if FIND-SET(u) ≠ FIND-SET(v)
      then A ← A ∪ {(u, v)}
            UNION(u, v)return A
```


Pseudocode for Prim

```
PRIM(V, E, w, r)
Q ← ∅
for each u ∈ V
do key[u] ← ∞
   π[u] ← NILINSERT(Q, u)
   key[r] = 0
while Q ≠ ∅
do u ← EXTRACT-MIN(Q)
   for each v ∈ Adj[u]
   do if v ∈ Q and w_uv < key[v]
         then π[v] ← u
              key[v] = w_uv
```

Shortest Paths

- (Single Source) shortest-path algorithms produce a label: $d[v] = \delta(s, v)$.
- Initially $d[v] = \infty$, reduces as the algorithm goes, so always $d[v] \geq \delta(s, v)$
- Also produce labels $\pi[v]$, predecessor of v on a shortest path from s.

Relax!

- The algorithms work by improving (lowering) the shortest path estimate $d[v]$
- This operation is called relaxing an edge (u, v)
- Can we improve the shortest-path estimate for v by going through u and taking (u, v)?

```
RELAX(u, v, w)
if d[v] > d[u] + w_uv
   then d[v] ← d[u] + w_uv
        π[v] ← u
```


Lemma, Lemma, Lemma

Path Relaxation Property

Let $P = \{v_0, v_1, \ldots v_k\}$ be a shortest path from $s = v_0$ to v_k. If the edges (v_0, v_1), (v_1, v_2), (v_{k-1}, v_k) are relaxed in that order, (there can be other relaxations in-between), then $d[v_k] = \delta(s, v_k)$

Bellman-Ford Algorithm

- Works with Negative-Weight Edges
- Returns true is there are no negative-weight cycles reachable from s, false otherwise

```
BELLMAN-FORD(V, E, w, s)
INIT-SINGLE-SOURCE(V, s)
for i ← 1 to |V| − 1
do for  each (u, v) in E
      do RELAX(u, v, w)
for  each (u, v) in E
do if d[v] > d[u] + w_uv
        then return False

return True
```

SSSP Dag

DAG-SHORTEST-PATHS(V, E, s, w)

```
Init-Single-Source(V, s)
topologically sort the vertices
for each u in topologically sorted V
do for each v ∈ Adj[u]
    do RELAX(u, v, w)
```


Dijkstra

DIJKSTRA(V, E, w, s)

```
Init-Single-Source(V, s)
S ← ∅
Q ← V
while Q ≠ ∅
do u ← Extract-Min(Q)
    S ← S ∪ {u}
    for each v ∈ Adj[u]
    do Relax(u, v, w)
```

- Dijkstra's Algorithm Runs in $O(E \lg V)$, with a binary heap implementation.

All Pairs Shortest Paths

- The output of an all pairs shortest path algorithm is a matrix $D = (d)_{ij}$, where $d_{ij} = \delta(i, j)$
- DP: $\ell_{ij}^{(m)}$ be the shortest path from $i \in V$ to $j \in V$ that uses $\leq m$ edges

$$\ell_{ij}^{(m)} = \min_{1 \leq k \leq n} (\ell_{ik}^{(m-1)} + w_{kj})$$

EXTEND(L, W)

```
create (n × n) matrix L'
for i ← 1 to n
do for j ← 1 to n
    do ℓ'_ij ← ∞
        for k ← 1 to n
        do ℓ'_ij ← min(ℓ'_ij, ℓ_ik + w_kj)
```

- This is just like matrix multiplication.
- We can speed this up.

Floyd Warshall

- Floyd-Warshall Labels: Let $d_{ij}^{(k)}$ be the shortest path from i to j such that all intermediate vertices are in the set $\{1, 2, \ldots, k\}$.
- This simple obervation, immediately suggests a DP recursion

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & k = 0 \\ \min(d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}) & k \geq 1 \end{cases}$$

- We look for $D^{(n)} = (d)_{ij}^{(n)}$

Floyd-Warshall

FLOYD-WARSHALL(W)

$D^{(0)} = W$
for $k \leftarrow 1$ **to** n
do for $i \leftarrow 1$ **to** n
    do for $j \leftarrow 1$ **to** n
        do $d_{ij}^{(k)} \leftarrow \min(d_{ij}^{k-1}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$
            return $D^{(n)}$

Flows

- A net flow is a function $f : V \times V \rightarrow \mathbb{R}^{|V| \times |V|}$ that satisfies three conditions:

1. Capacity Constraints:
$$0 \leq f(u,v) \leq c(u,v)$$
2. Skew Symmetry:
$$f(u,v) = -f(v,u) \; \forall u \in V, v \in V$$
3. Flow Conservation:
$$\sum_{v \in V} f(u,v) = 0 \; \forall u \in V \setminus \{s,t\}$$

The Maximum Flow Problem

Given $G = (V, E)$. source node $s \in V$, sink node $t \in V$, edge capacities c. Find a flow whose value is maximum.

Phlow Phacts

- For any cut (S, T), $f(S, T) = |f|$
- Residual capacity of arcs given flow:
$$c_f(u,v) \stackrel{\text{def}}{=} c(u,v) - f(u,v) \geq 0.$$
- Give flow f, we can create a residual network from the flow. $G_f = (V, E_f)$, with
$$E_f \stackrel{\text{def}}{=} \{(u,v) \in V \times V \mid c_f(u,v) > 0\},$$
so that each edge in the residual network can admit a positive flow.

Max-Flow Min-Cut Theorem

The following statements are equivalent

1. f is a maximum flow
2. f admits no augmenting path. (No (s,t) path in residual network)
3. $|f| = c(S, T)$ for some cut (S, T)

FORD-FULKERSON(V, E, c, s, t)

for $i \leftarrow 1$ **to** n
do $f[u,v] \leftarrow f[v,u] \leftarrow 0$
while $\exists$ augmenting path P in G_f
do augment f by $c_f(P)$

Analysis of this? Do better algorithms exist?

What I Think is Important

1. I'd be especially happy if you could deduce the (worst-case) running time of an algorithm given the Pseudocost or the Java code.
2. Know about the Data Structures
   - Hash
   - Heap
   - Binary Search Tree
3. Other than that, know how to "do" all of the algorithms
   - BFS, DFS
   - Kruskal, Prim
   - Bellman-Ford, Floyd-Warshell, Dijkstra
   - Max Flow (Augmenting Path)
   - Cholesky, $PA = LU$

Left To Do

- Lab 12 – Least squares and homework assignment – Due @ 12PM on May 4.
- Final Exam: Sunday May 6 – 8AM –11AM. 360 Packard Lab. I'll bring the donuts.
  - You will be allowed One cheat sheet. You can write on one side of 8.5×11 inch paper.
  - (Aside: Please don't waste all your time looking things up on your cheat sheet.)
  - No calculators will be allowed.
- No Class on Friday. Please (if you can) attend Dr. Kelly Gaither's Talk:
  - Rausch Bizness College: Room 91
  - `www.lehigh.edu/computing/hpc/hpcday/2007`

- (For the most part), I really enjoyed teaching this class.
- You helped make my last semester here an enjoyable one
- Free Lunch – Jim and Greg! (See me after class to arrange time...)
- I'll be traveling next week, but please send email if you have questions!