MA/CSSE 473 Day 27

Dynamic Programming

Binomial Coefficients

Warshall's algorithm

(Optimal BSTs)

Student questions?

Dynamic programming

- Used for problems with recursive solutions and overlapping subproblems
- Typically, we save (memoize) solutions to the subproblems, to avoid recomputing them.
- Previously seen example: Fib(n)

Dynamic Programming Example

- Binomial Coefficients:
- C(n, k) is the coefficient of x^k in the expansion of $(1+x)^n$
- C(n,0) = C(n, n) = 1.
- If 0 < k < n, C(n, k) = C(n-1, k) + C(n-1, k-1)
- Can show by induction that the "usual" factorial formula for C(n, k) follows from this recursive definition.
  - An upcoming homework problem.
- If we don't cache values as we compute them, this can take a lot of time, because of duplicate (overlapping) computation.

Computing a binomial coefficient

Binomial coefficients are coefficients of the binomial formula:
$(a + b)^n = C(n,0)a^n b^0 + . . . + C(n,k)a^{n-k}b^k + . . . + C(n,n)a^0 b^n$

Recurrence: $C(n,k) = C(n-1,k) + C(n-1,k-1)$ for $n > k > 0$
$C(n,0) = 1, \quad C(n,n) = 1$ for $n \geq 0$

Value of $C(n,k)$ can be computed by filling in a table:

	0	1	2 . . .	k-1	k
0	1				
1	1	1			
.					
.					
.					
n-1				$C(n-1,k-1)$	$C(n-1,k)$
n					$C(n,k)$

Computing *C*(*n*, *k*):

ALGORITHM *Binomial*(*n*, *k*)

```
//Computes C(n, k) by the dynamic programming algorithm
//Input: A pair of nonnegative integers n ≥ k ≥ 0
//Output: The value of C(n, k)
for i ← 0 to n do
    for j ← 0 to min(i, k) do
        if j = 0 or j = i
            C[i, j] ← 1
        else C[i, j] ← C[i − 1, j − 1] + C[i − 1, j]
return C[n, k]
```

Time efficiency: Θ(*nk*)

Space efficiency: Θ(*nk*)

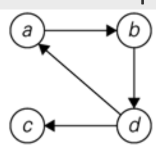
If we are computing C(n, k) for many different n and k values, we could cache the table between calls.

Elementary Dyn. Prog. problems

- These are in Section 8.1 of Levitin
- Simple and straightforward.
- I am going to have you read them on your own.
  - Coin-row
  - Change-making
  - Coin Collection

Transitive closure of a directed graph

- We ask this question for a given directed graph G: for each of vertices, (A,B), is there a path from A to B in G?
- Start with the boolean adjacency matrix A for the n-node graph G. A[i][j] is 1 if and only if G has a directed edge from node i to node j.
- The **transitive closure** of G is the boolean matrix T such that T[i][j] is 1 iff there is a nontrivial directed path from node i to node j in G.
- If we use boolean adjacency matrices, what does M^2 represent? M^3?
- In boolean matrix multiplication, + stands for **or**, and * stands for **and**



(a)

$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array}\begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

(b)

$$T = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array}\begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

(c)

FIGURE 8.2 (a) Digraph. (b) Its adjacency matrix. (c) Its transitive closure.

Transitive closure *via* multiplication

- Again, using + for **or**, we get

  $T = M + M^2 + M^3 + \ldots$

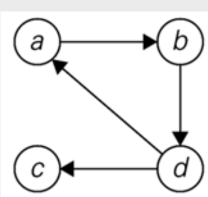
- Can we limit it to a finite operation?
- We can stop at M^{n-1}.
  - How do we know this?
- Number of numeric multiplications for solving the whole problem?

Warshall's Algorithm for Transitive Closure

- Similar to binomial coefficients algorithm
- Assume that the vertices have been numbered
  $v_1, v_2, \ldots, v_n,$
- Graph represented by a boolean adjacency matrix M.
- Numbering is arbitrary, but is fixed throughout the algorithm.
- Define the boolean matrix $R^{(k)}$ as follows:
  - $R^{(k)}[i][j]$ is 1 iff there is a path from v_i to v_j in the directed graph that has the form $v_i = w_0 \rightarrow w_1 \rightarrow \ldots \rightarrow w_s = v_j$, where
    - s >=1, and
    - for all t = 1, …, s-1, the w_t is v_m for some $m \leq k$
      i.e, none of the intermediate vertices are numbered higher than k
- What is $R^{(0)}$?
- Note that the transitive closure T is $R^{(n)}$

$R^{(k)}$ example

- $R^{(k)}[i][j]$ is 1 iff there is a path in the directed graph
  $v_i = w_0 \rightarrow w_1 \rightarrow \ldots \rightarrow w_s = v_j$, where
  - s >1, and
  - for all t = 2, …, s-1, the w_t is v_m for some $m \leq k$
- **Example:** assuming that the node numbering is in alphabetical order, calculate $R^{(0)}$, $R^{(1)}$, and $R^{(2)}$



$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

Quickly Calculating $R^{(k)}$

- Back to the matrix multiplication approach:
  - How much time did it take to compute $A^k[i][j]$, once we have A^{k-1}?
- Can we do better when calculating $R^{(k)}[i][j]$ from $R^{(k-1)}$?
- How can $R^{(k)}[i][j]$ be 1?
  - either $R^{(k-1)}[i][j]$ is 1, or
  - there is a path from v_i to v_k that uses no vertices numbered higher than v_{k-1}, and a similar path from v_k to v_j.
- Thus $R^{(k)}[i][j]$ is

  $R^{(k-1)}[i][j]$ **or** ($R^{(k-1)}[i][k]$ **and** $R^{(k-1)}[k][j]$)
- Note that this can be calculated in constant time if we already have the three vales from the right-hand side.
- Time for calculating $R^{(k)}$ from $R^{(k-1)}$?
- Total time for Warshall's algorithm?

Code and example on next slides

ALGORITHM *Warshall*($A[1..n, 1..n]$)

```
//Implements Warshall's algorithm for computing the transitive closure
//Input: The adjacency matrix A of a digraph with n vertices
//Output: The transitive closure of the digraph
R(0) ← A
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            R(k)[i, j] ← R(k−1)[i, j] or (R(k−1)[i, k] and R(k−1)[k, j])
return R(n)
```

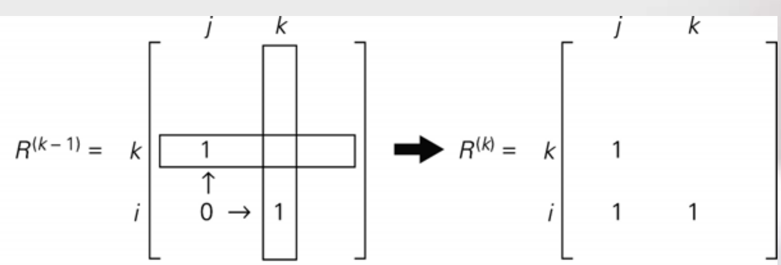
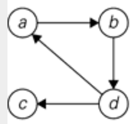


FIGURE 8.3 Rule for changing zeros in Warshall's algorithm



$R^{(0)} =$

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	0	1	0

Ones reflect the existence of paths with no intermediate vertices ($R^{(0)}$ is just the adjacency matrix); boxed row and column are used for getting $R^{(1)}$.

$R^{(1)} =$

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	**1**	1	0

Ones reflect the existence of paths with intermediate vertices numbered not higher than 1, i.e., just vertex *a* (note a new path from *d* to *b*); boxed row and column are used for getting $R^{(2)}$.

$R^{(2)} =$

	a	b	c	d
a	0	1	0	**1**
b	0	0	0	1
c	0	0	0	0
d	1	1	1	**1**

Ones reflect the existence of paths with intermediate vertices numbered not higher than 2, i.e., *a* and *b* (note two new paths); boxed row and column are used for getting $R^{(3)}$.

$R^{(3)} =$

	a	b	c	d
a	0	1	0	1
b	0	0	0	1
c	0	0	0	0
d	1	1	1	1

Ones reflect the existence of paths with intermediate vertices numbered not higher than 3, i.e., *a*, *b*, and *c* (no new paths); boxed row and column are used for getting $R^{(4)}$.

$R^{(4)} =$

	a	b	c	d
a	**1**	1	**1**	1
b	**1**	**1**	**1**	1
c	0	0	0	0
d	1	1	1	1

Ones reflect the existence of paths with intermediate vertices numbered not higher than 4, i.e., *a*, *b*, *c*, and *d* (note five new paths).

FIGURE 8.4 Application of Warshall's algorithm to the digraph shown. New ones are in bold.

Floyd's algorithm

- All-pairs shortest path
- A **network** is a graph whose edges are labeled by (usually) non-negative numbers. We store those edge numbers as the values in the adjacency matrix for the graph
- A **shortest path** from vertex u to vertex v is a path whose edge sum is smallest.
- Floyd's algorithm calculates the shortest path from u to v for each pair (u, v) od vertices.
- It is so much like Warshall's algorithm, that I am confident you can quickly get the details from the textbook after you understand Warshall's algorithm.

Dynamic Programming Example

OPTIMAL BINARY SEARCH TREES

Warmup: Optimal linked list order

- Suppose we have n distinct data items $x_1, x_2, \ldots, x_n$ in a linked list.
- Also suppose that we know the probabilities $p_1, p_2, \ldots, p_n$ that each of these items is the item we'll be searching for.
- Questions we'll attempt to answer:
  - What is the expected number of probes before a successful search completes?
  - How can we minimize this number?
  - What about an unsuccessful search?

Examples

- $p_i = 1/n$ for each i.
  - What is the expected number of probes?
- $p_1 = ½, p_2 = ¼, …, p_{n-1} = 1/2^{n-1}, p_n = 1/2^{n-1}$
  - expected number of probes:

$$\sum_{i=1}^{n-1} \frac{i}{2^i} + \frac{n}{2^{n-1}} = 2 - \frac{1}{2^{n-1}} < 2$$

- What if the same items are placed into the list in the opposite order?

$$\sum_{i=2}^{n} \frac{i}{2^{n+1-i}} + \frac{1}{2^{n-1}} = n - 1 + \frac{1}{2^{n-1}}$$

- The next slide shows the evaluation of the last two summations in Maple.
  - Good practice for you? prove them by induction

Calculations for previous slide

```
> sum(i/2^i, i=1..n-1)+ n/2^(n-1);
```

$$-2\left(\frac{1}{2}\right)^n n - 2\left(\frac{1}{2}\right)^n + 2 + \frac{n}{2^{(n-1)}}$$

```
> simplify(%);
```

$$-2^{(1-n)} + 2$$

```
> sum(i/2^(n+1-i), i=2..n) + 1/2^(n-1);
```

$$n - 1 + \frac{1}{2^{(n-1)}}$$

What if we don't know the probabilities?

1. Sort the list so we can at least improve the average time for unsuccessful search
2. Self-organizing list:
   - Elements accessed more frequently move toward the front of the list; elements accessed less frequently toward the rear.
   - Strategies:
     - Move ahead one position (exchange with previous element)
     - Exchange with first element
     - Move to Front (only efficient if the list is a linked list)

- What we are actually likely to know is frequencies in previous searches.
- Our best estimate of the probabilities will be proportional to the frequencies, so we can use frequencies instead of probabilities.

Optimal Binary Search Trees

- Suppose we have n distinct data keys K_1, K_2, …, K_n (in increasing order) that we wish to arrange into a Binary Search Tree
- Suppose we know the probabilities that a successful search will end up at K_i and the probabilities that the key in an unsuccessful search will be larger than K_i and smaller than K_{i+1}
- This time the expected number of probes for a successful or unsuccessful search depends on the shape of the tree and where the search ends up
- General principle?

Example

- For now we consider only successful searches, with probabilities A(0.2), B(0.3), C(0.1), D(0.4).
- How many different ways to arrange these into a BST? Generalize for N distinct values.
- What would be the worst-case arrangement for the expected number of probes?
  - For simplicity, we'll multiply all of the probabilities by 10 so we can deal with integers.
- Try some other arrangements:
  Opposite, Greedy, Better, Best?