CSCI 4830 / 5722
Computer Vision

University of Colorado **Boulder**

CSCI 4830 / 5722
Computer Vision

Dr. Ioana Fleming
Spring 2019
Lecture 22

University of Colorado **Boulder**

Reminders

Submissions:

- Homework 4: Wed 3/20 at 11 pm

Readings:

- Szeliski:
  - chapter 11.5.1 (Dynamic Programming)
- P&F:
  - chapter 7.5 (Dynamic Programming)

Today

- Dynamic Programming

Stereo Reconstruction

- The Stereo Problem
  - Shape from two (or more) images
  - Biological motivation

Stereo reconstruction pipeline

What will cause errors?

- Camera calibration errors
- Poor image resolution
- Occlusions
- Violations of brightness constancy (specular reflections)
- Large motions
- Low-contrast image regions (textureless)

Stereo Correspondences

Stereo Correspondences

Dynamic Programming

- A technique for designing (optimizing) algorithms
- It can be applied to problems that can be decomposed in subproblems, but these subproblems overlap.
- Instead of solving the same subproblems repeatedly, applying dynamic programming techniques helps to solve each subproblem just once.

Algorithmic Paradigms

- Greedy. Build up a solution incrementally, myopically optimizing some local criterion.
- Divide-and-conquer. Break up a problem into two sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem.
- Dynamic programming. Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems.

Dynamic Programming

- Like divide and conquer, DP solves problems by combining solutions to subproblems.
- Unlike divide and conquer, subproblems are not independent.
  - Subproblems may share subsubproblems,
  - However, solution to one subproblem may not affect the solutions to other subproblems of the same problem. (More on this later.)
- DP reduces computation by
  - Solving subproblems in a bottom-up fashion.
  - Storing solution to a subproblem the first time it is solved.
  - Looking up the solution when subproblem is encountered again.
- Key: determine structure of optimal solutions

Dynamic Programming Applications

- Areas.
  - Bioinformatics.
  - Control theory.
  - Information theory.
  - Operations research.
  - Computer science: theory, graphics, AI, systems, ….
- Some famous dynamic programming algorithms.
  - Viterbi for hidden Markov models.
  - Unix diff for comparing two files.
  - Smith-Waterman for sequence alignment.
  - Bellman-Ford for shortest path routing in networks.
  - Cocke-Kasami-Younger for parsing context free grammars.

Dynamic Programming Examples

- String Alignment
- Binomial Coefficients
- The Integer Exact Knapsack problem
- Fibonacci numbers
- Longest Common Subsequence

String Similarity

- How similar are two strings?
  - `ocurrance`
  - `occurrence`

o	c	u	r	r	a	n	c	e	-
o	c	c	u	r	r	e	n	c	e

6 mismatches, 1 gap

o	c	-	u	r	r	a	n	c	e
o	c	c	u	r	r	e	n	c	e

1 mismatch, 1 gap

o	c	-	u	r	r	-	a	n	c	e
o	c	c	u	r	r	e	-	n	c	e

0 mismatches, 3 gaps

Edit Distance

- Applications.
  - Basis for Unix diff.
  - Speech recognition.
  - Computational biology.
- Edit distance. [Levenshtein 1966, Needleman-Wunsch 1970]
  - Gap penalty δ; mismatch penalty α_{pq}.
  - Cost = sum of gap and mismatch penalties.

C	T	G	A	C	C	T	A	C	C	T
C	C	T	G	A	C	T	A	C	A	T

$\alpha_{TC} + \alpha_{GT} + \alpha_{AG} + 2\alpha_{CA}$

-	C	T	G	A	C	C	T	A	C	C	T
C	C	T	G	A	C	-	T	A	C	A	T

$2\delta + \alpha_{CA}$

Sequence Alignment

- Goal: Given two strings $X = x_1\ x_2 \ldots x_m$ and $Y = y_1\ y_2 \ldots y_n$ find alignment of minimum cost.
- Def. An alignment M is a set of ordered pairs x_i-y_j such that each item occurs in at most one pair and no crossings.
- Def. The pair x_i-y_j and $x_{i'}$-$y_{j'}$ cross if $i < i'$, but $j > j'$.

$$\text{cost}(M) = \underbrace{\sum_{(x_i, y_j) \in M} \alpha_{x_i y_j}}_{\text{mismatch}} + \underbrace{\sum_{i : x_i \text{ unmatched}} \delta + \sum_{j : y_j \text{ unmatched}} \delta}_{\text{gap}}$$

x_1	x_2	x_3	x_4	x_5		x_6
C	T	A	C	C	-	G
-	T	A	C	A	T	G
	y_1	y_2	y_3	y_4	y_5	y_6

Ex: `CTACCG` vs. `TACATG`.

Sol: $M = x_2\text{-}y_1, x_3\text{-}y_2, x_4\text{-}y_3, x_5\text{-}y_4, x_6\text{-}y_6$

Sequence Alignment: Problem Structure

Def. OPT(i, j) = min cost of aligning strings $x_1\ x_2 \ldots x_i$ and $y_1\ y_2 \ldots y_j$.

- Case 1: OPT matches x_i-y_j.
  - pay mismatch for x_i-y_j + min cost of aligning two strings $x_1\ x_2 \ldots x_{i-1}$ and $y_1\ y_2 \ldots y_{j-1}$
- Case 2a: OPT leaves x_i unmatched.
  - pay gap for x_i and min cost of aligning $x_1\ x_2 \ldots x_{i-1}$ and $y_1\ y_2 \ldots y_j$
- Case 2b: OPT leaves y_j unmatched.
  - pay gap for y_j and min cost of aligning $x_1\ x_2 \ldots x_i$ and $y_1\ y_2 \ldots y_{j-1}$

$$OPT(i,j) = \begin{cases} j\delta & \text{if } i = 0 \\ \min \begin{cases} \alpha_{x_i y_j} + OPT(i-1, j-1) \\ \delta + OPT(i-1, j) \\ \delta + OPT(i, j-1) \end{cases} & \text{otherwise} \\ i\delta & \text{if } j = 0 \end{cases}$$

Sequence Alignment: Algorithm

```
Sequence-Alignment(m, n, x1x2...xm, y1y2...yn, δ, α) {
   for i = 0 to m
      M[0, i] = i*δ
   for j = 0 to n
      M[j, 0] = j*δ

   for i = 1 to m
      for j = 1 to n
         M[i, j] = min(α[xi, yj] + M[i-1, j-1],
                       δ + M[i-1, j],
                       δ + M[i, j-1])
   return M[m, n]
}
```

Example: Binomial Coefficients

- $(x + y)^2 = x^2 + 2xy + y^2$, coefficients are 1,2,1
- $(x + y)^3 = x^3 + 3x^2y + 3xy^2 + y^3$, coefficients are 1,3,3,1
- $(x + y)^4 = x^4 + 4x^3y + 6x^2y^2 + 4xy^3 + y^4$, coefficients are 1,4,6,4,1
- $(x + y)^5 = x^5 + 5x^4y + 10x^3y^2 + 10x^2y^3 + 5xy^4 + y^5$, coefficients are 1,5,10,10,5,1

Binomial Coefficients

- The binomial coefficient C(n, k) is the number of ways of choosing a subset of k elements from a set of n elements.
- By its definition, $\mathrm{C(n,k)} = \frac{n!}{(n-k)!*k!}$
- The repeated computation of all the factorials gets to be expensive
- Instead: $C(n,k) = \frac{n!}{(n-k)!*(k)!} = \frac{(n-1)!*n}{(n-k)!*(k)!} =$

$$= \frac{(n-1)!*(n-k)}{(n-k)!*(k)!} + \frac{(n-1)!*k}{(n-k)!*(k)!}$$

$$= C(n-1,k-1) + \frac{(n-1)!}{(n-k-1)!*(k)!} =$$

$$= C(n-1,k-1) + C(n-1,k)$$

C(n,0)=1, C(n,n)=1

- What approach can we use?

Binomial Coefficients – RecursionTree for C(5,2)

Binomial Coefficients – Recursive Solution

```
long C(int n, int k) {
  if ((k==0) || (k==n))
      return 1;
  else
      return C(n - 1, k) + C(n - 1, k - 1);
}
```

Optimization level 1: Memoization

- We can speed up the ***recursive*** algorithm by writing down the results of the recursive calls and looking them up again if we need them later.
- In this way we do not compute again a recursive call that was already computed before, just take the result from a ***table***
- This process was called ***memoization***
  - ***Memoization*** (not memorization!): the term comes from *memo (memorandum)*, since the technique consists of recording a value so that we can look it up later.

Binomial Coefficients – Table Filling Order

- result[i][j] stores the value of C(i,j)
- Table has n+1 rows and k+1 columns, k<=n
- Initialization: C(i,0)=1 and C(i,i)=1 for i=1 to n

Binomial Coefficients – Order (cont)

- result[i][j] stores the value of C(i,j)
- Rest of entries (i,j), for i=2 to n and j = 1 to i-1 are computed using entry (i-1, j-1) and (i-1, j)

Binomial Coefficients – DP

```
int[][]result;
int BinCoef(int n, int k) {
result = new int [n + 1][n + 1];
  int i, j;
  % loop to fill in the 1st column and diagonal of 1s
  for (i=0; i<=n; i++) {
      result[i][0]=1;
      result[i][i]=1;
      }
  %loop for the rest of the table
  for (i=2; i<=n; i++)
    for(j=1; j<i; j++)
      result[i][j]=result[i-1][j-1]+result[i-1][j];

  return result[n][k];
}
```

Time: O(n*n) (or O(n*k))
Memory: O(n*n) (or O(n*k))

Solution by dynamic programming

n	c(n,0)	c(n,1)	c(n,2)	c(n,3)	c(n,4)	c(n,5)	c(n,6)
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1	4	6	4	1		
5	1	5	10	10	5	1	
6	1	6	15	20	15	6	1

- Each row depends only on the preceding row
- Only linear space and quadratic time are needed
- This algorithm is known as **Pascal's Triangle**

Optimization level 3:
Memory Efficient Dynamic Programming

- In many dynamic programming algorithms, it may be not necessary to retain all intermediate results through the entire computation.
- Every step (every subproblem) depends usually on a reduced set of subproblems, not all other subproblems
- We replace the big table storing the results of all subproblems by some smaller buffers that are reused during the computation

Binomial Coefficients – Reduce Memory Complexity

- At every iteration for i, we compute the values of a row using the values of the row before it
- Two buffers of the length of a row are enough
- The buffers are *reused* after each iteration

Binomial Coefficients –

Memory Efficient Dynamic Programming

```
long C(int n, int k) {

long[] result1 = new long[n + 1];
long[] result2 = new long[n + 1];
result1[0] = 1;
result1[1] = 1;

for (int i = 2; i <= n; i++) {
    result2[0] = 1;
    for (int j = 1; j < i; j++)
        result2[j] = result1[j - 1] + result1[j];
    result2[i] = 1;
   long[] auxi = result1;
   result1 = result2;
   result2 = auxi;
   }
return result1[k];
}
```

Time: O(n*n) (or O(n*k))
Memory: O(n) (or O(k))

The principle of optimality, I

- Dynamic programming is a technique for finding an *optimal* solution
- The principle of optimality applies if the optimal solution to a problem always contains optimal solutions to all subproblems
- Example: Consider the problem of making N¢ with the fewest number of coins
  - Either there is an N¢ coin, or
  - The set of coins making up an optimal solution for N¢ can be divided into two nonempty subsets, n_1¢ and n_2¢
    - ***If either subset, n_1¢ or n_2¢, can be made with fewer coins, then clearly N¢ can be made with fewer coins, hence solution was not optimal***

The principle of optimality, II

- The principle of optimality holds if
  - Every optimal solution to a problem contains...
  - ...optimal solutions to all subproblems
- The principle of optimality does *not* say
  - If you have optimal solutions to all subproblems...
  - ...then you can combine them to get an optimal solution
- Example: In US coinage,
  - The optimal solution to 7¢ is 5¢ + 1¢ + 1¢, *and*
  - The optimal solution to 6¢ is 5¢ + 1¢, *but*
  - The optimal solution to 13¢ is *not* 5¢ + 1¢ + 1¢ + 5¢ + 1¢
- But there is *some* way of dividing up 13¢ into subsets with optimal solutions (say, 11¢ + 2¢) that will give an optimal solution for 13¢
  - Hence, the principle of optimality holds for this problem

Steps in Dynamic Programming

1. Characterize structure of an optimal solution.
2. Define value of optimal solution recursively.
3. Compute optimal solution values either top-down with caching or bottom-up in a table.
4. Construct an optimal solution from computed values.

Example – Fibonacci Numbers (recursion)

```
Function f(n)
    if n = 0
        output 0
    else
    if n = 1
        output 1
    else
        output f(n-1) + f(n-2)
```

Example – Fibonacci Numbers (recursion)

- Run time: $T(n) = T(n-1) + T(n-2)$
- This function grows as n grows
- The run time doubles as n grows and is order $O(2^n)$.
- This is a bad algorithm as there are numerous repetitive calculations.

Example – Fibonacci Numbers (DP)

Example – Fibonacci Numbers (DP)

- Dynamic programming calculates from bottom to top.
- Values are stored for later use.
- This reduces repetitive calculation.

Example – Fibonacci Numbers (DP)

```
int fibDP(int n)
{
  /* Declare an array to store Fibonacci numbers. */
  int f[n+1];
  int i;

  /* 0th and 1st number of the series are 0 and 1*/
  f[0] = 0;
  f[1] = 1;

  for (i = 2; i <= n; i++)
  {
      /* Add the previous 2 numbers in the series
         and store it */
      f[i] = f[i-1] + f[i-2];
  }

  return f[n];
}
```

Longest Common Subsequence

- ***Problem:*** Given 2 sequences, $X = \langle x_1,\ldots,x_m \rangle$ and $Y = \langle y_1,\ldots,y_n \rangle$, find the maximum common subsequence.

springtime	ncaa tournament	basketball
printing	north carolina	snoeyink

Subsequence need not be consecutive, but must be in order.

Recursive Solution

- Define $c[i, j]$ = length of LCS of X_i and Y_j .
- We want $c[m,n]$.

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max(c[i-1, j], c[i, j-1]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

This gives a recursive algorithm and solves the problem.
But does it solve it well?

Recursive Solution

$$c[\alpha,\beta] = \begin{cases} 0 & \text{if } \alpha \text{ empty or } \beta \text{ empty,} \\ c[prefix\alpha, prefix\beta]+1 & \text{if } \mathrm{end}(\alpha)=\mathrm{end}(\beta), \\ \max(c[prefix\alpha,\beta], c[\alpha, prefix\beta]) & \text{if } \mathrm{end}(\alpha)\neq\mathrm{end}(\beta). \end{cases}$$

c[springtime, printing]

c[springtim, printing] *c*[springtime, printin]

[springti, printing] [**springtim, printin**] [**springtim, printin**] [springtime, printi]

[springt, printing] [springti, printin] [springtim, printi] [springtime, print]

The LCS Problem

- The LCS problem has 2 versions:
  - The *Simple* version, requesting only to find out the length of the longest common subsequence
  - The *Complete* version, requesting to find out the sequence itself

- We discuss first the Simple version

LCS

- $X = \{x_1, \ldots x_m\}$
- $Y = \{y_1, \ldots, y_n\}$
- X_i = the prefix subsequence $\{x_1, \ldots x_i\}$
- Y_i = the prefix subsequence $\{y_1, \ldots y_i\}$
- $Z = \{z_1, \ldots z_k\}$ is the LCS of X and Y .
- LCS(i,j) = LCS of X_i and Y_j

$$LCS(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ LCS(i-1, j-1) + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max(LCS(i-1, j), LCS(i, j-1)) & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

See [CLRS] – chap 15.4

LCS – Dynamic programming

- Entries of row i=0 and column j=0 are initialized to 0
- Entry (i,j) is computed from (i-1, j-1), (i-1, j) and (i, j-1)

A valid order is:

```
For i:=1 to m do
     For j:=1 to n do
         … compute lcs[i,j]
```

Time complexity: O(n*m)

Memory complexity: n*m

LCS – The Complete Version

- The *Complete* version of the problem: we are also interested in finding the characters of the longest common subset

Result is empty string

Add common character to result

$$LCS(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ LCS(i-1, j-1) + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max(LCS(i-1, j), LCS(i, j-1)) & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

Just return result of a subproblem

```
LCS-LENGTH(X, Y)
m = X.length
n = Y.length
let b[1..m, 1..n] and c[0..m, 0..n] be new tables
for i = 1 to m
    c[i, 0] = 0
for j = 0 to n
    c[0, j] = 0
for i = 1 to m
    for j = 1 to n
        if x_i == y_j
            c[i, j] = c[i - 1, j - 1] + 1
            b[i, j] = "↖"
        elseif c[i - 1, j] ≥ c[i, j - 1]
            c[i, j] = c[i - 1, j]
            b[i, j] = "↑"
        else c[i, j] = c[i, j - 1]
            b[i, j] = "←"
return c and b
```

[CLRS – chap 15.4, page 394]

LCS – Restoring the common sequence

```
PRINT-LCS(b, X, i, j)
if i == 0 or j == 0
    return
if b[i, j] == "↖"
    PRINT-LCS(b, X, i − 1, j − 1)
    print x_i
elseif b[i, j] == "↑"
    PRINT-LCS(b, X, i − 1, j)
else PRINT-LCS(b, X, i, j − 1)
```

[CLRS – chap 15.4, page 395]

LCS - Example

i	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑0	↑0	↑0	↖1	←1	↖1
2	B	0	↖1	←1	←1	↑1	↖2	←2
3	C	0	↑1	↑1	↖2	←2	↑2	↑2
4	B	0	↖1	↑1	↑2	↑2	↖3	←3
5	D	0	↑1	↖2	↑2	↑2	↑3	↑3
6	A	0	↑1	↑2	↑2	↖3	↑3	↖4
7	B	0	↖1	↑2	↑2	↑3	↖4	↑4

[CLRS – Fig. 15.8]

Dynamic Programming Results