CSE 421 Algorithms

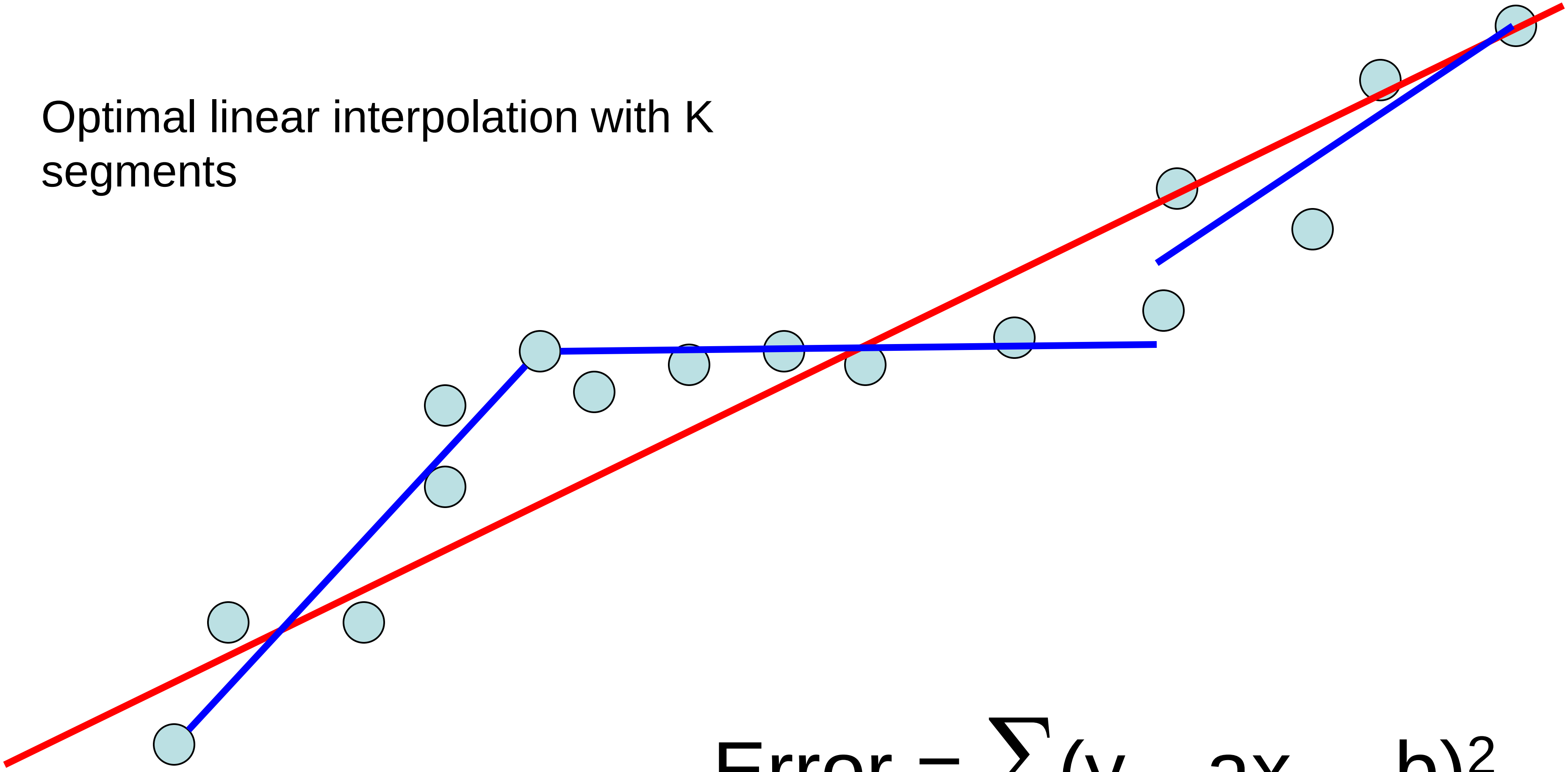
Richard Anderson

Lecture 17

Dynamic Programming

Optimal linear interpolation

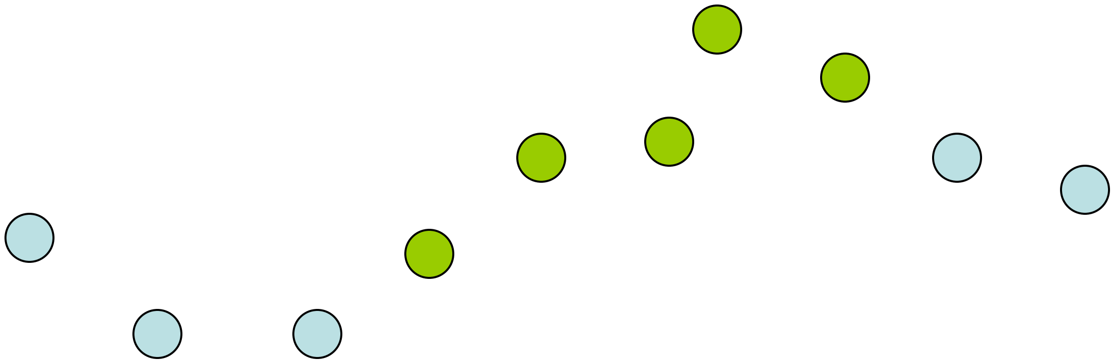
Optimal linear interpolation with K segments



$$\text{Error} = \sum (y_i - ax_i - b)^2$$

Notation

- Points $p_1, p_2, \ldots, p_n$ ordered by x-coordinate ($p_i = (x_i, y_i)$)
- $E_{i,j}$ is the least squares error for the optimal line interpolating $p_i, \ldots p_j$



Optimal interpolation with k segments

- Optimal segmentation with three segments
  - $\text{Min}_{i,j}\{E_{1,i} + E_{i,j} + E_{j,n}\}$
  - $O(n^2)$ combinations considered
- Generalization to k segments leads to considering $O(n^{k-1})$ combinations

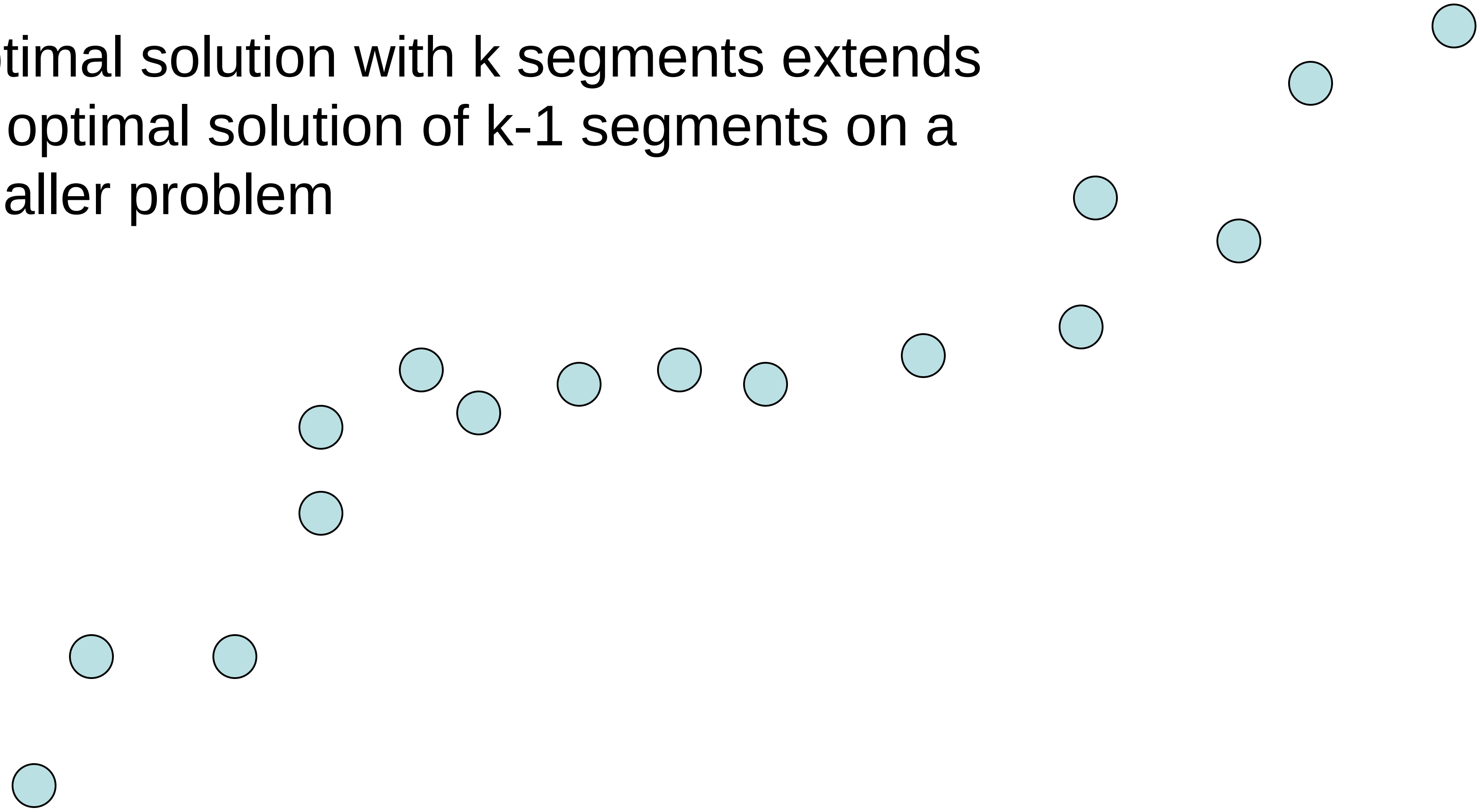
$Opt_k[\,j\,]$: Minimum error approximating $p_1 \ldots p_j$ with k segments

Express $Opt_k[\,j\,]$ in terms of $Opt_{k-1}[1], \ldots, Opt_{k-1}[\,j\,]$

$$Opt_k[\,j\,] = \min_i \{\, Opt_{k-1}[\,i\,] + E_{i,j} \,\} \text{ for } 0 < i < j$$

Optimal sub-solution property

Optimal solution with k segments extends an optimal solution of k-1 segments on a smaller problem



Optimal multi-segment interpolation

Compute Opt[k, j] for $0 < k < j < n$

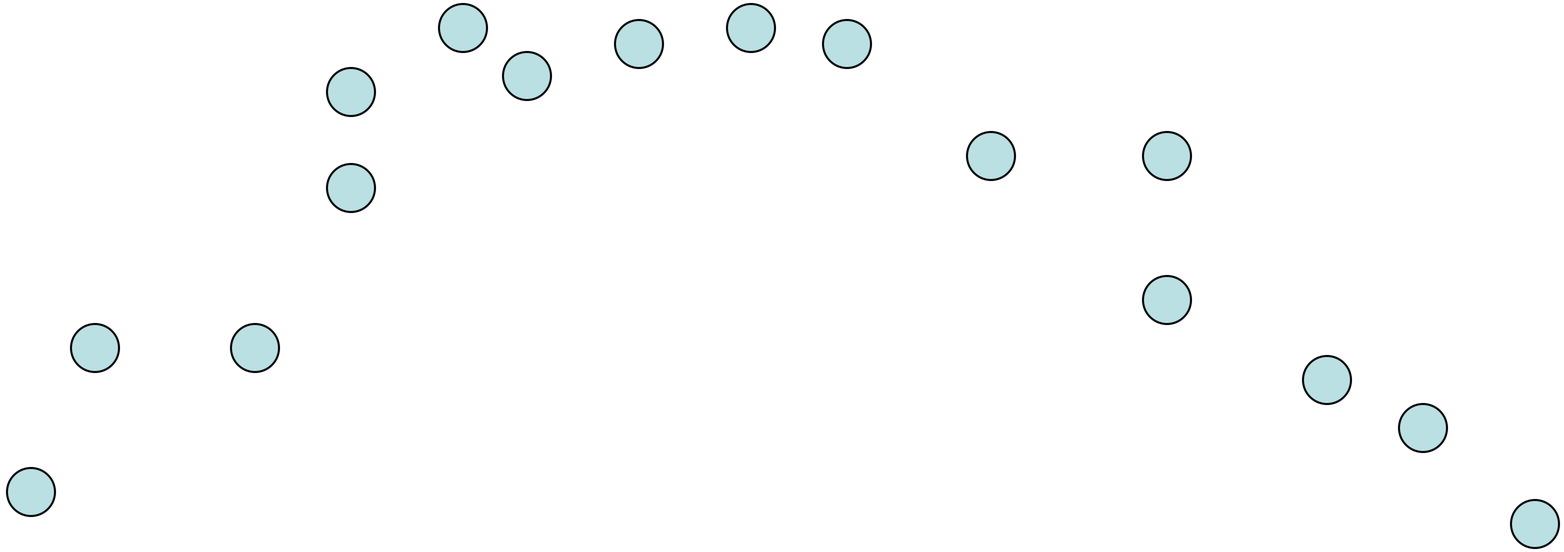
```
for j := 1 to n
    Opt[ 1, j] = E_{1,j};
for k := 2 to n-1
    for j := 2 to n
        t := E_{1,j}
        for i := 1 to j -1
            t = min (t, Opt[k-1, i ] + E_{i,j})
        Opt[k, j] = t
```

Determining the solution

- When Opt[k,j] is computed, record the value of i that minimized the sum
- Store this value in a auxiliary array
- Use to reconstruct solution

Variable number of segments

- Segments not specified in advance
- Penalty function associated with segments
- Cost = Interpolation error + C x #Segments



Penalty cost measure

- $Opt[\ j\ ] = \min(E_{1,j}, \min_i(Opt[\ i\ ] + E_{i,j} + P))$

Subset Sum Problem

- Let $w_1, \ldots, w_n$ = {6, 8, 9, 11, 13, 16, 18, 24}
- Find a subset that has as large a sum as possible, without exceeding 50

Adding a variable for Weight

- Opt[j, K] the largest subset of $\{w_1, \ldots, w_j\}$ that sums to at most K
- {2, 4, 7, 10}
  - Opt[2, 7] =
  - Opt[3, 7] =
  - Opt[3,12] =
  - Opt[4,12] =

Subset Sum Recurrence

- Opt[j, K] the largest subset of $\{w_1, \ldots, w_j\}$ that sums to at most K

Subset Sum Grid

Opt[j, K] = max(Opt[j – 1, K], Opt[j – 1, K – w_j] + w_j)

4																	
3																	
2																	
1																	
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

{2, 4, 7, 10}

Subset Sum Code

```
for j = 1 to n
    for k = 1 to W
        Opt[j, k] = max(Opt[j-1, k], Opt[j-1, k-w_j] + w_j)
```

Knapsack Problem

- Items have weights and values
- The problem is to maximize total value subject to a bound on weght
- Items $\{I_1, I_2, \ldots I_n\}$
  - Weights $\{w_1, w_2, \ldots, w_n\}$
  - Values $\{v_1, v_2, \ldots, v_n\}$
  - Bound K
- Find set S of indices to:
  - Maximize $\sum_{i \varepsilon S} v_i$ such that $\sum_{i \varepsilon S} w_i <= K$

Knapsack Recurrence

Subset Sum Recurrence:

$$Opt[\, j, K] = \max(Opt[\, j - 1, K], Opt[\, j - 1, K - w_j] + w_j)$$

Knapsack Recurrence:

Knapsack Grid

$Opt[\,j, K] = \max(Opt[\,j - 1, K], Opt[\,j - 1, K - w_j] + v_j)$

4																	
3																	
2																	
1																	
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Weights {2, 4, 7, 10} Values: {3, 5, 9, 16}

Dynamic Programming Examples

- Examples
  - Optimal Billboard Placement
    - Text, Solved Exercise, Pg 307
  - Linebreaking with hyphenation
    - Compare with HW problem 6, Pg 317
  - String approximation
    - Text, Solved Exercise, Page 309

Billboard Placement

- Maximize income in placing billboards
  - $b_i = (p_i, v_i)$, v_i: value of placing billboard at position p_i
- Constraint:
  - At most one billboard every five miles
- Example
  - {(6,5), (8,6), (12, 5), (14, 1)}

Design a Dynamic Programming Algorithm for Billboard Placement

- Compute Opt[1], Opt[2], . . ., Opt[n]
- What is Opt[k]?

Input $b_1, \ldots, b_n$, where $b_i = (p_i, v_i)$, position and value of billboard i

Opt[k] = fun(Opt[0],…,Opt[k-1])

- How is the solution determined from sub problems?

Input $b_1, \ldots, b_n$, where $b_i = (p_i, v_i)$, position and value of billboard i

Solution

```
j = 0;              // j is five miles behind the current position
                    // the last valid location for a billboard, if one placed at P[k]
for k := 1 to n
        while (P[ j ] < P[ k ] – 5)
                j := j + 1;
        j := j – 1;
        Opt[ k]  = Max(Opt[ k-1] , V[ k ] + Opt[ j ]);
```

Optimal line breaking and hyphen-ation

- Problem: break lines and insert hyphens to make lines as balanced as possible
- Typographical considerations:
  - Avoid excessive white space
  - Limit number of hyphens
  - Avoid widows and orphans
  - Etc.

Penalty Function

- Pen(i, j) – penalty of starting a line a position i, and ending at position j

Opt-i-mal line break-ing and hyph-en-a-tion is com-put-ed with dy-nam-ic pro-gram-ming

- Key technical idea
  - Number the breaks between words/syllables

String approximation

- Given a string S, and a library of strings B = $\{b_1, \ldots b_m\}$, construct an approximation of the string S by using copies of strings in B.

B = {abab, bbbaaa, ccbb, ccaacc}

S = abaccbbbaabbccbbccaabab

Formal Model

- Strings from B assigned to non-overlapping positions of S
- Strings from B may be used multiple times
- Cost of δ for unmatched character in S
- Cost of γ for mismatched character in S
  - MisMatch(i, j) – number of mismatched characters of b_j, when aligned starting with position i in s.

Design a Dynamic Programming Algorithm for String Approximation

- Compute Opt[1], Opt[2], . . ., Opt[n]
- What is Opt[k]?

Target string S = $s_1 s_2 \ldots s_n$
Library of strings B = {$b_{1,} \ldots, b_m$}
MisMatch(i,j) = number of mismatched characters with b_j when aligned starting at position i of S.

Opt[k] = fun(Opt[0],…,Opt[k-1])

- How is the solution determined from sub problems?

Target string S = $s_1s_2\ldots s_n$
Library of strings B = {$b_{1,}\ldots,b_m$}
MisMatch(i,j) = number of mismatched characters with b_j when aligned starting at position i of S.

Solution

```
for i := 1 to n
        Opt[k] = Opt[k-1] + δ;
        for j := 1 to |B|
                p = i – len(b_j);
                Opt[k] = min(Opt[k],  Opt[p-1] + γ MisMatch(p, j));
```