CSE 421
Algorithms

Richard Anderson
Lecture 18
Dynamic Programming

Dynamic Programming

- The most important algorithmic technique covered in CSE 421
- Key ideas
  - Express solution in terms of a polynomial number of sub problems
  - Order sub problems to avoid recomputation

Today - 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

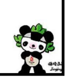
Billboard Placement

- 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]?

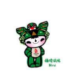
Input $b_1, \ldots, b_n$, where bi = (p_i, v_i), position and value of billboard i



Opt[k] = fun(Opt[0],…,Opt[k-1])

- How is the solution determined from sub problems?

Input $b_1, \ldots, b_n$, where bi = (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

Design a Dynamic Programming Algorithm for Optimal Line Breaking

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

Opt[k] = fun(Opt[0],…,Opt[k-1])

- How is the solution determined from sub problems?

Solution

```
for k := 1 to n
      Opt[ k ] := infinity;
      for j := 0 to k-1
            Opt[ k ] := Min(Opt[k], Opt[ j ] + Pen(j, k));
```

But what if you want to layout the text?

- And not just know the minimum penalty?

Solution

```
for k := 1 to n
    Opt[ k ] := infinity;
    for j := 0 to k-1
        temp := Opt[ j ] + Pen(j, k);
        if (temp < Opt[ k ])
            Opt[ k]  = temp;
            Best[ k ] := j;
```

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 = $s_1s_2\ldots s_n$
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.

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));
```