CSE 421
Algorithms

Richard Anderson

Lecture 18

Dynamic Programming

Announcements

One dimensional dynamic programming: Interval scheduling

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

Two dimensional dynamic programming

K-segment linear approximation

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

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

Two dimensional dynamic programming

Subset sum and knapsack

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

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

Alternate approach for Strict Subset Sum

Alternate solution of Strict Subset Sum with a dynamic programming algorithm:

- Sum[i, K] = true if there is a subset of $\{w_1,\ldots w_i\}$ that sums to exactly K, false otherwise
- Sum [i, K] = Sum [i -1, K] **OR** Sum[i - 1, K - w_i]
- Sum [0, 0] = true; Sum[0, K] = false for K != 0

Saving space: Can you replace the two dimensional array with a one dimensional array?

Run time for Subset Sum

- With n items and target sum K, the run time is O(nK)
- If K is 1,000,000,000,000,000,000,000,000 this is very slow
- Alternate brute force algorithm: examine all subsets: $O(n2^n)$

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
  - Longest Common Subsequence
    - Text, Section 6.6

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

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

Element distinctness has been a particular focus of lower bound analysis. The first time-space tradeoff lower bounds for the problem apply to structured algorithms. Borodin et al. [13] gave a time-space tradeoff lower bound for computing ED on *comparison* branching programs of $T \in \Omega(n^{3/2}/S^{1/2})$ and, since $S \geq \log_2 n$, $T \in \Omega(n^{3/2}\sqrt{\log n}/S)$. Yao [32] improved this to a near-optimal $T \in \Omega(n^{2-\epsilon(n)}/S)$, where $\epsilon(n) = 5/(\ln n)^{1/2}$. Since these lower bounds apply to the average case for randomly ordered inputs, by Yao's lemma, they also apply to randomized comparison branching programs. These bounds also trivially apply to all frequency moments since, for $k \neq 1$, $ED(x) = n$ iff $F_k(x) = n$. This near-quadratic lower bound seemed to suggest that the complexity of ED and F_k should closely track that of sorting.

Optimal Paragraphing

- Optimal line breaking:
  - Assign words to paragraphs to minimize the sum of line penalties (square of excess white space)
  - Words $w_1, w_2, . . . W_n$
  - Pen(i, j): penalty of filling a line with $w_i,\ldots,w_j$
  - Opt[j, k]: minimum penalty of ending line k with w_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));
```

Longest Common Subsequence

- $C=c_1\ldots c_g$ is a subsequence of $A=a_1\ldots a_m$ if C can be obtained by removing elements from A (but retaining order)
- LCS(A, B): A maximum length sequence that is a subsequence of both A and B

```
ocurranec          attacggct
occurrence         tacgacca
```

Determine the LCS of the following strings

BARTHOLEMEWSIMPSON

KRUSTYTHECLOWN

String Alignment Problem

- Align sequences with gaps

```
CAT TGA  AT
CAGAT AGGA
```

- Charge δ_x if character x is unmatched
- Charge γ_{xy} if character x is matched to character y

Note: the problem is often expressed as a minimization problem, with $\gamma_{xx} = 0$ and $\delta_x > 0$

LCS Optimization

- $A = a_1a_2\ldots a_m$
- $B = b_1b_2\ldots b_n$
- Opt[j, k] is the length of $LCS(a_1a_2\ldots a_j, b_1b_2\ldots b_k)$

Optimization recurrence

If $a_j = b_k$, Opt[j,k] = 1 + Opt[j-1, k-1]

If a_j != b_k, Opt[j,k] = max(Opt[j-1,k], Opt[j,k-1])

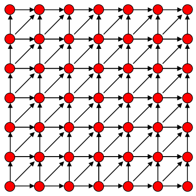
Give the Optimization Recurrence for the String Alignment Problem

- Charge δ_x if character x is unmatched
- Charge γ_{xy} if character x is matched to character y

Opt[j, k] =

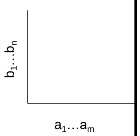
Let $a_j = x$ and $b_k = y$
Express as minimization

Dynamic Programming Computation



Code to compute Opt[j,k]

Storing the path information




```
A[1..m],  B[1..n]
for i := 1 to m     Opt[i, 0] := 0;
for j := 1 to n     Opt[0,j] := 0;
Opt[0,0] := 0;
for i := 1 to m
        for j := 1 to n
                if A[i] = B[j]  {  Opt[i,j] := 1 + Opt[i-1,j-1];  Best[i,j] := Diag; }
                else if Opt[i-1, j] >= Opt[i, j-1]
                        {  Opt[i, j] := Opt[i-1, j], Best[i,j] := Left; }
                else        {  Opt[i, j] := Opt[i, j-1], Best[i,j] := Down; }
```

How good is this algorithm?

- Is it feasible to compute the LCS of two strings of length 300,000 on a standard desktop PC? Why or why not.