CSE 421
Algorithms
Richard Anderson
Lecture 19
Dynamic Programming

Announcements

• Nov 11, No class (holiday)

One dimensional dynamic programming: Interval scheduling
Opt[ j ] = max (Opt[ j – 1 ], wj + Opt[ p[ j ] ])  

Two dimensional dynamic programming
K-segment linear approximation
Optk[ j ] = min i { Optk-1[ i ] + Eij } for 0 < i < j

Two dimensional dynamic programming
Subset sum and knapsack
Opt[ j, K ] = max (Opt[ j – 1, K ], Opt[ j – 1, K – wj ] + wj)

Alternate approach for Subset Sum

• Alternate formulation of Subset Sum dynamic programming algorithm
  – Sum[i, K] = true if there is a subset of [w1,…,wj] that sums to exactly K, false otherwise
  – Sum[i, K] = Sum[j-1, K] OR Sum[i-1, K - wj]
  – Sum[0, 0] = true; Sum[i, 0] = false for i != 0

• To allow for negative numbers, we need to fill in the array between Kmin and Kmax
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)

Optimal line breaking

Element distinctness has been a particular focus of lower bound analysis. The first time-space tradeoff lower bound for the problem is studied through the algorithms of Borodin et al. [1] and, since S ≥ log2 n, T ∈ Ω(n log n/S), Yao [32] improved this to a near-optimal T ∈ Ω(n log(n)/S), where c(n) = 5/(ln n)^1/2. These bounds also trivially apply to all frequency moments since, for k ≠ 1, E(D) = n if F_k(s) = n. This near-quadratic lower bound seemed to suggest that the complexity of ED and F_k should closely track that of sorting.

Optimal Line Breaking

- Words have length w_i, line length L
- Penalty related to white space or overflow of the line
  - Quadratic measure often used
- Pen(i, j): Penalty for putting w_i, w_{i+1}, ..., w_j on the same line
- Opt[k, m]: minimum penalty for ending line k with w_m

String approximation

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

\[ B = \{abab, bbbaaa, ccb, ccaacc\} \]

\[ S = \text{abaccbbbaabbcccbcaabab} \]

Formal Model

- Strings from B assigned to non-overlapping positions of S
- Strings from B may be used multiple times
- Cost of $\delta$ for unmatched character in S
- Cost of $\gamma$ 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]?
Opt[k] = fun(Opt[0],…,Opt[k-1])

- How is the solution determined from subproblems?

Target string $S = s_1s_2...s_n$
Library of strings $B = \{b_1,...,b_m\}$
MisMatch(i,j) = number of mismatched characters with $b_j$ when aligned starting at position $i$ of $S$.

Solution

for $i := 1$ to $n$
    $Opt[k] = Opt[k-1] + \delta$
for $j := 1$ to $|B|$
    $p = i - \text{len}(b_j)$;
    $Opt[k] = \min(Opt[k], \text{Opt}[p-1] + \gamma \text{MisMatch}(p, j))$;

Longest Common Subsequence

- $C=c_1...c_g$ is a subsequence of $A=a_1...a_m$ 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 $B$
  
  \begin{align*}
  \text{occuranec} & \quad \text{attacggt} \\
  \text{occurrence} & \quad \text{tacgacca}
  \end{align*}

Determine the LCS of the following strings

BARTHOLEMEOYSIMPOW
KRUSTYTHECLOWN

String Alignment Problem

- Align sequences with gaps
  
  \begin{align*}
  \text{CAT} & \quad \text{TGA} & \quad \text{AT} \\
  \text{CAGAT} & \quad \text{AGGA}
  \end{align*}

- Charge $\delta_x$ if character $x$ is unmatched
- Charge $\gamma_{xy}$ if character $x$ is matched to character $y$

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

LCS Optimization

- $A = a_1a_2...a_m$
- $B = b_1b_2...b_n$

- $Opt[j, k]$ is the length of LCS($a_1a_2...a_j, b_1b_2...b_k$)
Optimization recurrence

If $a_j = b_k$, $\text{Opt}[j,k] = 1 + \text{Opt}[j-1,k-1]$

If $a_j \neq b_k$, $\text{Opt}[j,k] = \max(\text{Opt}[j-1,k], \text{Opt}[j,k-1])$

Give the Optimization Recurrence for the String Alignment Problem

- Charge $\delta_x$ if character $x$ is unmatched
- Charge $\gamma_{xy}$ if character $x$ is matched to character $y$

$\text{Opt}[j,k] = \max(\text{Opt}[j-1,k], \text{Opt}[j,k-1])$

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

Dynamic Programming Computation

Code to compute $\text{Opt}[j,k]$

Storing the path information

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.