CSE 417 Algorithms and Complexity

Richard Anderson
Lecture 20
Dynamic Programming
One dimensional dynamic programming: Interval scheduling

\[ \text{Opt}[j] = \max(\text{Opt}[j-1], w_j + \text{Opt}[p[j]]) \]
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 $\text{Opt}[1], \text{Opt}[2], \ldots, \text{Opt}[n]$
• What is $\text{Opt}[k]$?

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 $b_1, ..., b_n$, where $b_i = (p_i, v_i)$, position and value of billboard $i$
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]);
Two dimensional dynamic programming

K-segment linear approximation

$$\text{Opt}_k[j] = \min_i \{ \text{Opt}_{k-1}[i] + E_{i,j} \} \text{ for } 0 < i < j$$
Two dimensional dynamic programming

Subset sum and knapsack

\[ \text{Opt}[j, K] = \max(\text{Opt}[j - 1, K], \text{Opt}[j - 1, K - w_j] + w_j) \]

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

\[ \begin{array}{cccccccccccccccc}
4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array} \]
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 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 $w_i, w_{i+1}, \ldots, w_j$ on the same line
• $Opt[k, m]$: minimum penalty for ending line $k$ with $w_m$
Optimal Line Breaking

Optimal score for ending line $k$ on $w_m$

$$\text{Opt}_k[m] = \min_i \{ \text{Opt}_{k-1}[i] + \text{Pen}(i+1,m) \} \text{ for } 0 < i < m$$
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 $B$

ocurranc e  attacggct
occurrence  tacgacca
Determine the LCS of the following strings

BARTHOLEMES SIMPSON

KRUSTY THE CLOWN
String Alignment Problem

- Align sequences with gaps
  
  CAT TGA AT
  
  CAGAT AGGA

- 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 $\gamma_{xx} = 0$ 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 $y$

$\text{Opt}[j, k] = \ldots$

Let $a_j = x$ and $b_k = y$
Express as minimization
String edit with Typo Distance

- Find closest dictionary word to typed word
  - Dist(‘a’, ‘s’) = 1
  - Dist(‘a’, ‘u’) = 6
- Capture the likelihood of mistyping characters
- Different distance model for T9 on basic mobile phone
Dynamic Programming
Computation
Code to compute Opt[ n, m]

for (int i = 0; i < n; i++)
  for (int j = 0; j < m; j++)
    if (A[i] == B[j])
      Opt[i, j] = Opt[i-1, j-1] + 1;
    else if (Opt[i-1, j] >= Opt[i, j-1])
      Opt[i, j] := Opt[i-1, j];
    else
      Opt[i, j] := Opt[i, j-1];
Storing the path information

\[ A[1..m], \ B[1..n] \]

\[
\begin{align*}
&\text{for } i := 1 \text{ to } m \quad \text{Opt}[i, 0] := 0; \\
&\text{for } j := 1 \text{ to } n \quad \text{Opt}[0,j] := 0; \\
&\text{Opt}[0,0] := 0; \\
&\text{for } i := 1 \text{ to } m \\
&\quad \text{for } j := 1 \text{ to } n \\
&\quad \quad \text{if } A[i] = B[j] \quad \{ \text{Opt}[i,j] := 1 + \text{Opt}[i-1,j-1]; \quad \text{Best}[i,j] := \text{Diag}; \} \\
&\quad \quad \text{else if } \text{Opt}[i-1, j] \geq \text{Opt}[i, j-1] \\
&\quad \quad \quad \{ \text{Opt}[i, j] := \text{Opt}[i-1, j], \quad \text{Best}[i,j] := \text{Left}; \} \\
&\quad \quad \text{else} \quad \{ \text{Opt}[i, j] := \text{Opt}[i, j-1], \quad \text{Best}[i,j] := \text{Down}; \}
\end{align*}
\]
Reconstructing Path from Distances
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.