One dimensional dynamic programming: Interval scheduling

Opt[j] = max (Opt[j - 1], w_j + 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 Opt[1], Opt[2], ..., Opt[n]
- What is Opt[k]?

Input b_1, ..., 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]);

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

K-segment linear approximation

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

<table>
<thead>
<tr>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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 structural algorithms. Borodin et al. [13] gave a time-space tradeoff lower bound for computing ED on comparison branching programs of \(T \leq \Omega(n^{2.5}/\log n)\), and since \(S \geq \log n\), \(T \in \Omega(n^{2.5}/\log n)\). Yao [32] improved this to a near-optimal \(T \in \Omega(n^{2.5}/\log n)\), where \(v(n) = 5((\ln n)^{3/2})\) Since these lower bounds apply to the average case for randomly ordered inputs by Yao's lemmata, 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
• \(\text{Pen}(i, j)\): Penalty for putting \(w_i, w_{i+1}, \ldots, w_j\) on the same line
• \(\text{Opt}[k, m]\): minimum penalty for ending line \(k\) with \(w_m\)

<table>
<thead>
<tr>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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)
• \(\text{LCS}(A, B)\): A maximum length sequence that is a subsequence of both \(A\) and \(B\)

occurranec  attacggct
occurrence  tacgacca
Determine the LCS of the following strings

BARTHOLEMEWSIMPSON

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 $\gamma_{xx} = 0$ and $\delta_x > 0$

LCS Optimization

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

- $\text{Opt}[j, k]$ is the length of $\text{LCS}(a_1a_2...a_j, b_1b_2...b_k)$

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 $y$

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

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

String edit with Typo Distance

- Find closest dictionary word to typed word
- $\text{Dist}(a', 's') = 1$
- $\text{Dist}(a', 'u') = 6$
- Capture the likelihood of mistyping characters
- Different distance model for T9 on basic mobile phone
Dynamic Programming Computation

Code to compute Opt[ n, m]

```java
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

```java
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
        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;
```

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.