Dynamic Programming

Reading

• Dynamic Programming Examples:
  – 6.1-6.2. Weighted Interval Scheduling
  – 6.3 Segmented Least Squares
  – 6.4 Knapsack and Subset Sum
  – 6.6 String Alignment
    • 6.7* String Alignment in linear space
  – 6.8 Shortest Paths (again)
  – 6.9 Negative cost cycles
    • How to make an infinite amount of money

Weighted Interval Scheduling

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

Record which case is used in Opt computation

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>4</th>
<th>9</th>
<th>9</th>
<th>16</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>B</td>
<td>B</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td>A</td>
</tr>
</tbody>
</table>

Optimal linear interpolation

$$\text{Error} = \sum (y_i - ax_i - b)^2$$
What is the optimal linear interpolation with three line segments

What is the optimal linear interpolation with two line segments

What is the optimal linear interpolation with \( n \) line segments

**Notation**

- Points \( p_1, p_2, \ldots, p_n \) ordered by x-coordinate \( (p_i = (x_i, y_i)) \)
- \( E_{ij} \) is the least squares error for the optimal line interpolating \( p_i, \ldots, p_j \)

**Optimal interpolation with two segments**

- Give an equation for the optimal interpolation of \( p_1, \ldots, p_n \) with two line segments

- \( E_{ij} \) is the least squares error for the optimal line interpolating \( p_i, \ldots, p_j \)

**Optimal interpolation with \( k \) segments**

- Optimal segmentation with three segments
  - \( \text{Min}_{ij} \{ E_{ij} + E_{ij} + E_{ij} \} \)
  - \( O(n^2) \) combinations considered
- Generalization to \( k \) segments leads to considering \( O(n^{k-1}) \) combinations
Opt\(_k[j]\) : Minimum error approximating \(p_1...p_j\) with \(k\) segments

How do you express Opt\(_k[j]\) in terms of Opt\(_{k-1}[1],...,\text{Opt}_{k-1}[j]\)?

Optimal sub-solution property

Optimal solution with \(k\) segments extends an optimal solution of \(k-1\) segments on a smaller problem

Optimal multi-segment interpolation

Compute Opt\([k, j]\) for \(0 < k < j < n\)

\[
\text{for } j = 1 \text{ to } n \\
\quad \text{Opt}[1, j] = E_{1,j};
\]

\[
\text{for } k = 2 \text{ to } n-1 \\
\quad \text{for } j = 2 \text{ to } n \\
\quad \quad t = E_{1,j} \\
\quad \quad \text{for } i = 1 \text{ to } j-1 \\
\quad \quad \quad t = \min(t, \text{Opt}[k-1, i] + E_{i,j}) \\
\quad \quad \text{Opt}[k, j] = t
\]

Determining the solution

• When Opt\([k,j]\) is computed, record the value of \(i\) that minimized the sum
• Store this value in an auxiliary array
• Use to reconstruct solution

Variable number of segments

• Segments not specified in advance
• Penalty function associated with segments
• Cost = Interpolation error + \(C\) x #Segments

Penalty cost measure

\[
\text{Opt}[j] = \min(E_{1,j}, \min_i(\text{Opt}[i] + E_{i,j} + P))
\]
Summary for Segmented Interpolation

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

Optimal solution with \( k \) segments extends an optimal solution of \( k-1 \) segments on a smaller problem.

Subset Sum Problem

- Let \( w_1, \ldots, w_n = \{6, 8, 9, 11, 13, 16, 18, 24\} \)
- Find a subset that has as large a sum as possible, without exceeding 50

Adding a variable for Weight

- \( \text{Opt}[j, K] \) the largest subset of \( \{w_1, \ldots, w_j\} \) that sums to at most \( K \)
- \( \{2, 4, 7, 10\} \)
  - \( \text{Opt}[2, 7] = \)
  - \( \text{Opt}[3, 7] = \)
  - \( \text{Opt}[3, 12] = \)
  - \( \text{Opt}[4, 12] = \)

Subset Sum Recurrence

- \( \text{Opt}[j, K] \) the largest subset of \( \{w_1, \ldots, w_j\} \) that sums to at most \( K \)

Subset Sum Grid

\[
\begin{array}{cccccccc}
4 & 3 & 2 & 1 & 0 & 0 & 0 & 0 \\
3 & 2 & 1 & 0 & 0 & 0 & 0 & 0 \\
2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

\( \{2, 4, 7, 10\} \)

Subset Sum Code

\[
\text{for } j = 1 \text{ to } n \\
\text{for } k = 1 \text{ to } W \\
\text{Opt}[j, k] = \max(\text{Opt}[j-1, k], \text{Opt}[j-1, k-w_j] + w_j)
\]
Knapsack Problem

- Items have weights and values
- The problem is to maximize total value subject to a bound on weight
- Items \( \{I_1, I_2, \ldots, I_n\} \)
  - Weights \( \{w_1, w_2, \ldots, w_n\} \)
  - Values \( \{v_1, v_2, \ldots, v_n\} \)
  - Bound \( K \)
- Find set \( S \) of indices to:
  - Maximize \( \sum_{i \in S} v_i \) such that \( \sum_{i \in S} w_i \leq K \)

Knapsack Recurrence

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

Knapsack Recurrence:

Alternate approach for Subset Sum

- Alternate formulation of Subset Sum dynamic programming algorithm
  - \( \text{Sum}[i, K] = \text{true} \) if there is a subset of \( \{w_1, \ldots, w_i\} \) that sums to exactly \( K \), \( \text{false} \) otherwise
  - \( \text{Sum}[i, K] = \text{Sum}[i - 1, K] \text{ OR } \text{Sum}[i - 1, K - w_i] \)
  - \( \text{Sum}[0, 0] = \text{true}; \text{Sum}[i, 0] = \text{false} \) for \( i \neq 0 \)

- To allow for negative numbers, we need to fill in the array between \( K_{\min} \) and \( K_{\max} \)

Run time for Subset Sum

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