Announcements

- Reading for this week
  - 6.1-6.8

Review from last week

Weighted Interval Scheduling

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

Optimal linear interpolation

\[ \text{Error} = \sum (y_i - ax_i - b)^2 \]
Counting electoral votes

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

Billboard Placement

- Maximize income in placing billboards
  - \( b_i = (p_i, v_i) \); 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], \ldots, Opt[n]
- What is Opt[k]?

Opt\[k\] = fun(\text{Opt}[0], \ldots, \text{Opt}[k-1])

- How is the solution determined from subproblems?

Input \( b_1, \ldots, b_n \), where \( b_i = (p_i, v_i) \), position and value of billboard

Woodcut Placement

Input \( b_1, \ldots, b_n \), where \( b_i = (p_i, v_i) \), position and value of billboard

Input \( b_1, \ldots, b_n \), where \( b_i = (p_i, v_i) \), position and value of billboard

\begin{verbatim}
for k := 1 to n
    j := 0; // j is five miles behind the current position
    // the last valid location for a billboard, if one placed at P[k]
    for j := 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]);
\end{verbatim}

Solution
Optimal line breaking and hyphenation

• Problem: break lines and insert hyphens to make lines as balanced as possible
• Typographical considerations:
  – Avoid excessive white space
  – Limit number of hyphens
  – Avoid widows and orphans
  – Etc.

Penalty Function

• Pen(i, j) – penalty of starting a line a position i, and ending at position j

Optimal line breaking and hyphenation is computed with dynamic programming

• Key technical idea
  – Number the breaks between words/syllables

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 = abacbbbbaabbccbbccaabab

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], \ldots, Opt[n]
• What is Opt[k]?

Opt[k] = fun(Opt[0], \ldots, Opt[k-1])

• How is the solution determined from sub problems?
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] + g(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

occurance
occurrence
attachgct
tacgacca

Determine the LCS of the following strings

BARTHOHEMEWSIMPSON

KRUSTYTHECLOWN

String Alignment Problem

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

- 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 γ_{xx} = 0 and δ_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 \( y \)
\( \text{Opt}[j,k] = \)
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 100,000 on a standard desktop PC? Why or why not.
Observations about the Algorithm

- The computation can be done in $O(m+n)$ space if we only need one column of the Opt values or Best Values.
- The algorithm can be run from either end of the strings.

Computing LCS in $O(nm)$ time and $O(n+m)$ space

- Divide and conquer algorithm
- Recomputing values used to save space

Divide and Conquer Algorithm

- Where does the best path cross the middle column?
- For a fixed $i$, and for each $j$, compute the LCS that has $a_i$ matched with $b_j$.

Constrained LCS

- $\text{LCS}_{i,j}(A, B)$: The LCS such that
  - $a_1, \ldots, a_i$ paired with elements of $b_1, \ldots, b_j$
  - $a_{i+1}, \ldots, a_m$ paired with elements of $b_{j+1}, \ldots, b_n$
- $\text{LCS}_{4,3}(\text{abbacbb}, \text{cbbaa})$

$A = \text{RRSSRTTRTS}$
$B = \text{RTSRRSTST}$

Compute $\text{LCS}_{5,0}(A, B)$, $\text{LCS}_{5,1}(A, B)$, $\ldots, \text{LCS}_{5,9}(A, B)$

$A = \text{RRSSRTTRTS}$
$B = \text{RTSRRSTST}$

Compute $\text{LCS}_{5,0}(A, B)$, $\text{LCS}_{5,1}(A, B)$, $\ldots, \text{LCS}_{5,9}(A, B)$

<table>
<thead>
<tr>
<th>i</th>
<th>left</th>
<th>right</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>
Computing the middle column

- From the left, compute LCS(a₁…aₘ/2, b₁…bₗ)
- From the right, compute LCS(aₘ/2+1…aₘ, bₗ+1…bₙ)
- Add values for corresponding j's
- Note – this is space efficient

Divide and Conquer

- A = a₁,…,aₘ
- B = b₁,…,bₙ
- Find j such that
  - LCS(a₁,…,aₘ/2, b₁,…,bₗ) and
  - LCS(aₘ/2+1…aₘ, bₗ+1…bₙ) yield optimal solution
- Recurse

Algorithm Analysis

- T(m,n) = T(m/2, j) + T(m/2, n-j) + cmn

Prove by induction that T(m,n) <= 2cmn

Memory Efficient LCS Summary

- We can afford O(nm) time, but we can’t afford O(nm) space
- If we only want to compute the length of the LCS, we can easily reduce space to O(n+m)
- Avoid storing the value by recomputing values
  - Divide and conquer used to reduce problem sizes

Shortest Paths with Dynamic Programming
Shortest Path Problem

- Dijkstra's Single Source Shortest Paths Algorithm
  - $O(m \log n)$ time, positive cost edges
- General case – handling negative edges
- If there exists a negative cost cycle, the shortest path is not defined
- Bellman-Ford Algorithm
  - $O(mn)$ time for graphs with negative cost edges

Shortest paths with a fixed number of edges

- Find the shortest path from $v$ to $w$ with exactly $k$ edges

Lemma

- If a graph has no negative cost cycles, then the shortest paths are simple paths
- Shortest paths have at most $n-1$ edges

Express as a recurrence

- $\text{Opt}_k(w) = \min_x [\text{Opt}_{k-1}(x) + c_{xw}]$
- $\text{Opt}_0(w) = 0$ if $v=w$ and infinity otherwise

Algorithm, Version 1

```plaintext
foreach w
  M[0, w] = infinity;
  M[0, v] = 0;
for i = 1 to n-1
  foreach w
    M[i, w] = min(M[i-1, x] + cost[x,w]);
```

Algorithm, Version 2

```plaintext
foreach w
  M[0, w] = infinity;
  M[0, v] = 0;
for i = 1 to n-1
  foreach w
    M[i, w] = min(M[i-1, w], min_x(M[i-1, x] + cost[x,w]));
```
Algorithm, Version 3

foreach w
    M[w] = infinity;
    M[v] = 0;
for i = 1 to n - 1
    foreach w
        M[w] = min(M[w], min_x(M[x] + cost[x,w]))

Correctness Proof for Algorithm 3

• Key lemma – at the end of iteration i, for all w, M[w] <= M[i, w];

• Reconstructing the path:
  – Set P[w] = x, whenever M[w] is updated from vertex x

If the pointer graph has a cycle, then the graph has a negative cost cycle

• If P[w] = x then M[w] >= M[x] + cost(x,w)
  – Equal when w is updated
  – M[x] could be reduced after update
• Let v₁, v₂, … vₖ be a cycle in the pointer graph with (vₖ, v₁) the last edge added
  – Just before the update
    • M[vₖ] >= M[v₅₆] + cost(v₅₆, v₅₅) for j < k
    • M[v₅₆] > M[v₅₆] + cost(v₅₆, v₅₅)
  – Adding everything up
    • 0 > cost(v₁, v₂) + cost(v₂, v₃) + … + cost(vₖ, v₁)

Negative Cycles

• If the pointer graph has a cycle, then the graph has a negative cycle
• Therefore: if the graph has no negative cycles, then the pointer graph has no negative cycles

Finding negative cost cycles

• What if you want to find negative cost cycles?

Foreign Exchange Arbitrage

<table>
<thead>
<tr>
<th></th>
<th>USD</th>
<th>EUR</th>
<th>CAD</th>
</tr>
</thead>
<tbody>
<tr>
<td>USD</td>
<td>1.2</td>
<td>0.8</td>
<td>1.2</td>
</tr>
<tr>
<td>EUR</td>
<td>1.2</td>
<td></td>
<td>1.6</td>
</tr>
<tr>
<td>CAD</td>
<td>0.8</td>
<td>0.6</td>
<td></td>
</tr>
</tbody>
</table>