Announcements

• Reading for this week
  - 6.1-6.8

Midterm Return – Mean ~ 37
Review from last week
Weighted Interval Scheduling

$E_1, \ldots, E_n$

$v_j$

Opt $\sum_i E_i$ - Max value solution from $I_i, \ldots, I_j$

Opt $\sum_j C_i = \min(\text{Opt}\sum_{j-1} E_i), v_j + \text{Opt}\sum_{j-1} E_i \\ (\text{Case}\ \text{analysis}\ \text{is}\ \text{used})$

$E_i$ is used.
Optimal linear interpolation

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

\[
S[j, K] - \text{subset of } w_j, \ldots, w_j \text{ sums to exactly } K
\]

\[
S[j, K] = S[j-1, K] \text{ or } S[j-1, K-w_j]
\]
Counting electoral votes

\[ c[i, K] = c[i-1, K] + c[i-1, K - v_i] \]

\[ c[51, 269] \]

\[ c[0, x] = 0 \]
\[ c[0, 0] = 1 \]
\[ c[x, 0.3] = 1 \]
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
  – $b_i = (p_i, v_i)$, $v_i$: 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 $i$
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 at 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
Longest Common Subsequence
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)
- LCS\((A, B)\): A maximum length sequence that is a subsequence of both \( A \) and \( B \)

```
occurranec
occurrence

attacggct

ocurrnc
tacgacca
```
Determine the LCS of the following strings

BARTHOLOMEW SIMPSON
KRUSTY THE CLOWN
R T H O W N
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

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

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

$\text{Opt}[n,m]$
Dynamic Programming Computation

for $i = 1$ to $n$
  
  for $j = 1$ to $m$
    
    $\text{Opt}(i, j) = \text{Opt}(i-1, j-1) \oplus \text{Opt}(i-1, j) \oplus \text{Opt}(i, j-1)$
Storing the path information

\[ A[1..m], \ B[1..n] \]
\[
\text{for } i := 1 \text{ to } m \quad \text{opt}[i, 0] := 0; \\
\text{for } j := 1 \text{ to } n \quad \text{opt}[0,j] := 0; \\
\text{opt}[0,0] := 0; \\
\text{for } i := 1 \text{ to } m \\
\quad \text{for } j := 1 \text{ to } n \\
\quad \quad \text{if } A[i] = B[j] \{ \text{opt}[i,j] := 1 + \text{opt}[i-1,j-1]; \text{best}[i,j] := \text{Diag}; \} \\
\quad \quad \text{else if } \text{opt}[i-1,j] \geq \text{opt}[i,j-1] \\
\quad \quad \quad \{ \text{opt}[i,j] := \text{opt}[i-1,j], \text{best}[i,j] := \text{Left}; \} \\
\quad \quad \text{else} \quad \{ \text{opt}[i,j] := \text{opt}[i,j-1], \text{best}[i,j] := \text{Down}; \} \\
\]

Diagram: a1...am, b1...bn
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

- $LCS_{i,j}(A,B)$: 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 $b_{j+1},\ldots,b_n$

- $LCS_{4,3}(abbacbb, cbbaa)$

\[
\begin{array}{cc}
abba & cbb \\
\hline
cbb & a\n\end{array}
\]
A = \textcolor{red}{R}R\textcolor{red}{S}S\textcolor{red}{R}R\textcolor{red}{T}T\textcolor{red}{R}R\textcolor{red}{T}TS
B=\textcolor{red}{R}T\textcolor{red}{S}\textcolor{red}{R}\textcolor{red}{R}\textcolor{red}{R}\textcolor{red}{R}\textcolor{red}{R}\textcolor{red}{R}\textcolor{red}{R}\textcolor{red}{S}\textcolor{red}{S}\textcolor{red}{T}\textcolor{red}{S}\textcolor{red}{T}\textcolor{red}{S}\textcolor{red}{T}

Compute LCS_{5,0}(A,B), LCS_{5,1}(A,B), \ldots, LCS_{5,9}(A,B)

\underline{304} \quad 1+4
A = RRSSRRTTRTS
B = RTSRRSTST

Compute \( \text{LCS}_{5,0}(A, B) \), \( \text{LCS}_{5,1}(A, B) \),..., \( \text{LCS}_{5,9}(A, B) \)
Computing the middle column

- From the left, compute LCS($a_1 \ldots a_{m/2}, b_1 \ldots b_j$)
- From the right, compute LCS($a_{m/2+1} \ldots a_m, b_{j+1} \ldots b_n$)
- Add values for corresponding j’s

- Note – this is space efficient
Divide and Conquer

- \( A = a_1, \ldots, a_m \) \hspace{5mm} \( B = b_1, \ldots, b_n \)
- Find \( j \) such that
  - \( \text{LCS}(a_1 \ldots a_{m/2}, b_1 \ldots b_j) \) and
  - \( \text{LCS}(a_{m/2+1} \ldots a_m, b_{j+1} \ldots b_n) \) yield optimal solution
- Recurse
Algorithm Analysis

- $T(m,n) = T(m/2, j) + T(m/2, n-j) + cnm$
Prove by induction that
\[ T(m,n) \leq 2cmn \]

**Induction – on \( m \)**

*Base case – \( m = 1 \)*

Assume \( T(k,n) \leq 2ckn \) for \( k < m \)

\[ T(m,n) = T\left( \frac{m}{2}, j \right) + T\left( \frac{m}{2}, n-j \right) + cmn \]

\[ \leq 2c \frac{m}{2} j + 2c \frac{m}{2} (n-j) + cmn \]

\[ = 2c \frac{m}{2} (j + (n-j)) + cmn \]

\[ = cmn + cmn = 2cmn \]
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
Lemma

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

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

Find shortest path distance from $v$ to $w$. 
Algorithm, Version 1

foreach w
    \[ M[0, w] = \text{infinity}; \]
M[0, v] = 0;
for i = 1 to n-1
    foreach w
        \[ M[i, w] = \min_x (M[i-1, x] + \text{cost}[x, w]); \]
Algorithm, Version 2

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

Algorithm, Version 3

foreach w
    \( M[w] = \text{infinity}; \)

\( M[v] = 0; \)

for \( i = 1 \) to \( n-1 \)
    foreach w
        \[ M[w] = \min(M[w], \min_{x} (M[x] + \text{cost}[x,w])) \]
Correctness Proof for Algorithm 3

- Key lemma – at the end of iteration i, for all \( w \), \( M[w] \leq M[i, w] \);

- 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 $M[w] >= M[x] + \text{cost}(x,w)$
  - Equal when $w$ is updated
  - $M[x]$ could be reduced after update
- Let $v_1, v_2, ..., v_k$ be a cycle in the pointer graph with $(v_k, v_1)$ the last edge added
  - Just before the update
    - $M[v_j] >= M[v_{j+1}] + \text{cost}(v_{j+1}, v_j)$ for $j < k$
    - $M[v_k] > M[v_1] + \text{cost}(v_1, v_k)$
  - Adding everything up
    - $0 > \text{cost}(v_1, v_2) + \text{cost}(v_2, v_3) + ... + \text{cost}(v_k, v_1)$
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

\[ 1.2 \times 0.6 \times 1.024 = 0.864 \]

<table>
<thead>
<tr>
<th></th>
<th>USD</th>
<th>EUR</th>
<th>CAD</th>
</tr>
</thead>
<tbody>
<tr>
<td>USD</td>
<td>----</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>