Dynamic Programming

Algorithm Design 6.1, 6.2, 6.3

Applications

• In class (today and next time)
  • Weighted interval scheduling
    • Set of weighted intervals with start and finishing times
    • Goal: find maximum weight subset of non-overlapping intervals

Applications

• In class (today and next time)
  • Weighted interval scheduling
  • Segmented least squares
    • Given n points in the plane find a small sequence of lines that minimizes the squared error.
Applications

- In class (today and next time)
  - Weighted interval scheduling
  - Segmented least squares
  - Sequence alignment
    - Given two strings A and B, how many edits (insertions, deletions, relabelings) is needed to turn A into B?

A C A A G T C
- C A T G T -
1 mismatch, 2 gaps

A C A A - G T C
- C A - T G T -
0 mismatches, 4 gaps

Applications

- In class (today and next time)
  - Weighted interval scheduling
  - Segmented least squares
  - Sequence alignment
  - Shortest paths with negative weights
    - Given a weighted graph, where edge weights can be negative, find the shortest path between two given vertices.

Applications

- In class (today and next time)
  - Weighted interval scheduling
  - Segmented least squares
  - Sequence alignment
  - Shortest paths with negative weights
    - Given a weighted graph, where edge weights can be negative, find the shortest path between two given vertices.

Dynamic Programming

- Greedy. Build solution incrementally, optimizing some local criterion.
- Divide-and-conquer. Break up problem into independent subproblems, solve each subproblem, and combine to get solution to original problem.
- Dynamic programming. Break up problem into overlapping subproblems, and build up solutions to larger and larger subproblems.
  - Can be used when the problem have "optimal substructure":
    - Solution can be constructed from optimal solutions to subproblems
    - Use dynamic programming when subproblems overlap.
Weighted Interval Scheduling

- **Weighted interval scheduling problem**
  - n jobs (intervals)
  - Job j starts at s_j, finishes at f_j, and has weight/value v_j.
  - Goal: Find maximum weight subset of non-overlapping (compatible) jobs.

---

**Weighted interval scheduling**

- Weighted interval scheduling problem
  - n jobs (intervals)
  - Job j starts at s_j, finishes at f_j, and has weight/value v_j.
  - Goal: Find maximum weight subset of non-overlapping (compatible) jobs.

---

**Weighted interval scheduling**

- Label/sort jobs by finishing time: f_1 ≤ f_2 ≤ ... ≤ f_n
Weighted interval scheduling

- Label/sort jobs by finishing time: \( f_1 \leq f_2 \leq \ldots \leq f_n \).
- \( p(j) \) = largest index \( i < j \) such that job \( i \) is compatible with job \( j \).
- Optimal solution \( \text{OPT} \):
  - **Case 1.** \( \text{OPT} \) selects last job
    \( \text{OPT} = v_n + \text{optimal solution to subproblem on 1,...,p(n)} \)
  - **Case 2.** \( \text{OPT} \) does not select last job
    \( \text{OPT} = \text{optimal solution to subproblem 1,...j-1} \)

 Weighted interval scheduling: brute force

\[
\text{OPT}(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\max(v_j + \text{OPT}(p(j)), \text{OPT}(j-1)) & \text{otherwise} 
\end{cases}
\]

 Weighted interval scheduling: memoization

\[
\text{OPT}(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\max(v_j + \text{OPT}(p(j)), \text{OPT}(j-1)) & \text{otherwise} 
\end{cases}
\]
Weighted interval scheduling: memoization

Input: n, s[1..n], f[1..n], v[1..n]
Sort jobs by finish time so that f[1] ≤ f[2] ≤ … ≤ f[n]
Compute p[1], p[2], ..., p[n]
for j = 1 to n
  if M[j] is empty
return M[n]

• Running time O(n log n):
  • Sorting takes O(n log n) time.
  • Computing p[j]: O(n log n)
  • For loop: O(n) time
    • Each iteration takes constant time.
  • Space O(n)

Weighted interval scheduling: bottom-up

Compute-Bottom-Up—Opt(n, s[1..n], f[1..n], v[1..n])
Sort jobs by finish time so that f[1] ≤ f[2] ≤ … ≤ f[n]
Compute p[1], p[2], ..., p[n]
M[0] = 0.
for j = 1 to n
  M[j] = max(v[j] + M[p[j]], M[j-1])
return M[n]
Weighted interval scheduling: bottom-up

**Compute-Bottom-Up-Opt(n, w[1..n], f[1..n], v[1..n])**

Sort jobs by finish time so that f[1] ≤ f[2] ≤ … ≤ f[n]

Compute p[1], p[2], …, p[n]

M[0] = 0.

for j = 1 to n
  M[j] = max(v[j] + M(p[j]), M(j-1))

return M[n]

---

**Find-Solution(j)**

if j = 0
  return emptyset
else if M[j] > M[j-1]
  return {j} ∪ Find-Solution(p[j])
else
  return Find-Solution(j-1)

---

Solution = 8, 4, 1

---

Segmented Least Squares
Least squares

• Least squares.
  • Given \( n \) points in the plane: \((x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\).
  • Find a line \( y = ax + b \) that minimizes the sum of the squared error:

  \[
  \text{SSE} = \sum_{i=1}^{n} (y_i - ax_i - b)^2
  \]

• Solution. Calculus \( \Rightarrow \) minimum error is achieved when

  \[
  a = \frac{n \sum x_i y_i - (\sum x_i)(\sum y_i)}{n \sum x_i^2 - (\sum x_i)^2}, \quad b = \frac{\sum y_i - a \sum x_i}{n}
  \]

Segmented least squares

• Segmented least squares.
  • Points lie roughly on a sequence of line segments.
  • Given \( n \) points in the plane \((x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\).
  • Find a sequence of lines that minimizes some function \( f(x) \).
  • What is a good choice for \( f(x) \) that balance accuracy and number of lines?

Dynamic programming: multiway choice

• \( \text{OPT}(j) = \) minimum cost for points \( p_1, p_2, \ldots, p_j \).
• \( e(i, j) = \) minimum sum of squares for points \( p_i, p_{i+1}, \ldots, p_j \).
• To compute \( \text{OPT}(j) \):
  • Last segment uses points \( p_{i-1}, p_{i+1}, \ldots, p_j \) for some \( i \).
  • Cost = \( e(i, j) + c + \text{OPT}(i-1) \).
Segmented least squares algorithm

\[ \text{OPT}(j) = \begin{cases} 0 & \text{if } j = 0 \\ \min_{1 \leq i \leq j} [e(i,j) + c + \text{OPT}(i-1)] & \text{otherwise} \end{cases} \]

**Segmented-least-squares**\(n, p_1, p_2, ..., p_n, c)\)

for \(j=1\) to \(n\)
   for \(i=1\) to \(j\)
      Compute the least squares \(e(i,j)\) for the segment \(p_i, p_{i+1}, ..., p_j\).

\(M[0] = 0.\)

for \(j=1\) to \(n\)
   \(M[j] = \infty\)
   for \(i=1\) to \(j\)
      \(M[j] = \min(M[j], e(i,j) + c + M[i-1])\)

Return \(M[n]\)

**Time.**
- \(O(n^3)\) for computing \(e(i,j)\) for \(O(n^2)\) pairs \(O(n)\) per pair.
- \(O(n^2)\) for computing \(M\).
- Total \(O(n^3)\)

**Space.**
- \(O(n^2)\).

---

Subproblem dag