The Design and Analysis of Algorithms

Lecture 12 Dynamic Programming I

Zhenbo Wang

Department of Mathematical Sciences, Tsinghua University

Content

Algorithmic Paradigms

- *Greedy*. Build up a solution incrementally, myopically optimizing some local criterion.

- *Divide-and-conquer*. Break up a problem into *independent* subproblems,solve each subproblem, and combine solution to subproblems to form solution to original problem.

- *Dynamic programming*. Break up a problem into a series of overlapping subproblems, and build up solutions to larger and larger subproblems.

Dynamic Programming Applications

- Areas:

  Bioinformatics; Control theory; Information theory; Operations research; Computer science $\cdots$

- Some famous dynamic programming algorithms:

  Unix diff for comparing two files.

  Viterbi for hidden Markov models.

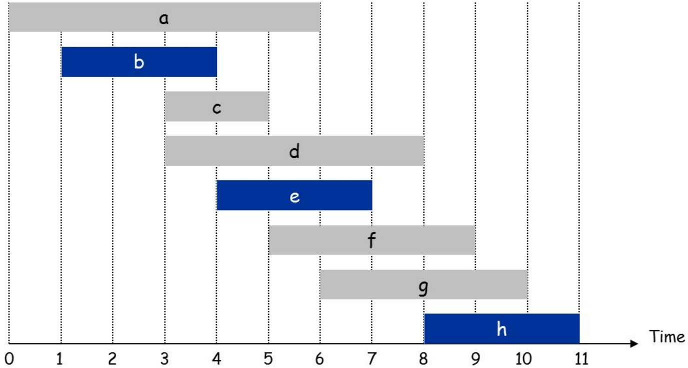
  Smith-Waterman for genetic sequence alignment.

  Bellman-Ford for shortest path routing in networks. $\cdots$

Weighted Interval Scheduling

- Job j starts at s_j, finishes at f_j and has weight v_j.
- Two jobs compatible if they don't overlap.

Goal. Find maximum weight subset of mutually compatible jobs.



Earliest-Finish-Time First Algorithm

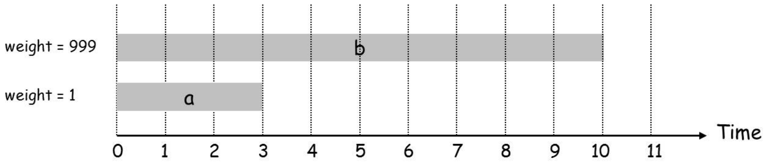
- *Earliest finish time first*.

  Consider jobs in ascending order of f_j.

  Add job to subset if it is compatible with previously jobs.

- *Recall*. Greedy algorithm is correct if all weights are 1.

- *Observation*. Greedy algorithm fails spectacularly for weighted version.

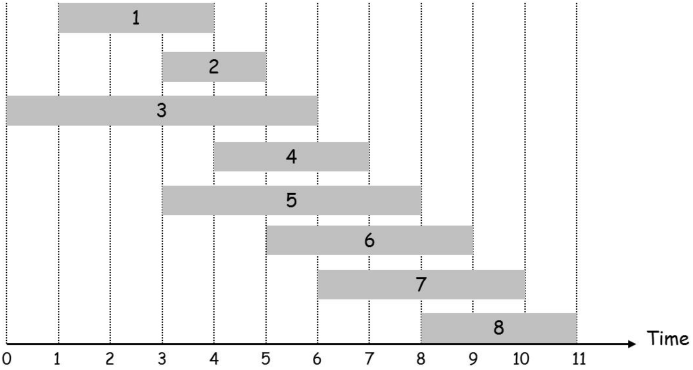


Weighted interval scheduling

- *Notation*. Label jobs by finishing time: $f_1 \leq f_2 \leq \cdots \leq f_n$.

Def. $p(j)$ = largest index $i < j$ such that job i is compatible with j.

Ex. $p(8) = 5$, $p(7) = 3$, $p(2) = 0$.



Dynamic programming: Binary Choice

- *Notation*. $OPT(j)$ = value of optimal solution to the problem consisting of job requests $1, 2, \cdots, j$.

Case 1. *OPT* selects job j.

Collect profit v_j.

Can't use incompatible jobs $\{p(j)+1, p(j)+2, \cdots, j-1\}$.

Must include optimal solution to problem consisting of remaining compatible jobs $1, 2, \cdots, p(j)$.

Case 2. *OPT* does not select job j.

Must include optimal solution to problem consisting of remaining compatible jobs $1, 2, \cdots, j-1$.

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

Weighted Interval Scheduling: Brute Force

INPUT: n, $s[1..n]$, $f[1..n]$, $v[1..n]$.

Sort jobs by finish time so that $f[1] \leq f[2] \leq \cdots \leq f[n]$.

Compute $p[1], p[2], \cdots, p[n]$.

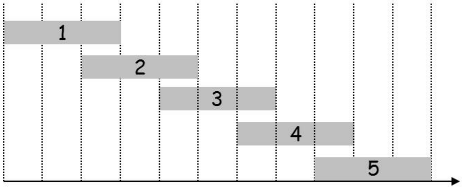
Compute – Opt(j)

if $j = 0$ **then**
    return 0.
else
    return $\max\{v[j] + Compute-Opt(p[j]), Compute-Opt(j-1)\}$.
end if

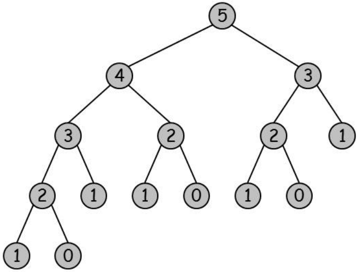
Weighted Interval Scheduling: Brute Force

- *Observation*. Recursive algorithm fails spectacularly because of redundant subproblems ⇒ exponential algorithms.

Ex. Number of recursive calls for family of “layered” instances grows like Fibonacci sequence.



p(1) = 0, p(j) = j-2



Weighted Interval Scheduling: Memoization

- *Memoization*. Store results of each sub-problem in a cache; lookup as needed.

INPUT: n, $s[1..n]$, $f[1..n]$, $v[1..n]$.

Sort jobs by finish time so that $f[1] \leq f[2] \leq \cdots \leq f[n]$.

Compute $p[1], p[2], \cdots, p[n]$.

```
for j = 1 to n do
    M[j] ← empty.
end for
return M[0] ← 0.
```

$M - Compute - Opt(j)$

```
if M[j] is empty then
    M[j] ← max{v[j]+M−Compute−Opt(p[j]), M−Compute−Opt(j − 1)}.
end if
return M[j].
```

Weighted Interval Scheduling: Running Time

Claim. Memoized version of algorithm takes $O(n \log n)$ time.

Pf. Sort by finish time: $O(n \log n)$.

Computing $p(\cdot)$: $O(n \log n)$ via binary search.

$M-Compute-Opt(j)$: each invocation takes $O(1)$ time:

(i) returns an existing value $M[j]$;

(ii) or fills in one new entry $M[j]$ and makes two recursive calls.

Progress measure $\Phi = \sharp$ nonempty entries of $M[]$.

initially $\Phi = 0$, throughout $\Phi \leq n$.

(ii) increases Φ by 1 $\Rightarrow$ at most $2n$ recursive calls.

Overall running time of $M-COMPUTE-OPT(n)$ is $O(n)$. $\square$

Weighted Interval Scheduling: Finding a Solution

Q. DP algorithm computes optimal value. How to find solution itself?

A. Make a second pass.

Find – Solution(j)

```
if j = 0 then
    return ∅.
else if v[j] + M[p[j]] > M[j – 1] then
    return {j} ∪ Find – Solution(p[j]).
else
    return Find – Solution(j – 1).
end if
```

- $\sharp$ of recursive calls $\leq n \Rightarrow O(n)$.

Weighted Interval Scheduling: Bottom-up

- *Bottom-up dynamic programming*. Unwind recursion.

$BOTTOM - UP(n, s_1, \cdots, s_n, f_1, \cdots, f_n, v_1, \cdots, v_n)$

```
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 do
    M[j] ← max{v_j + M[p(j)], M[j − 1]}.
end for
```

Least Squares

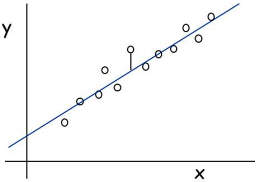
- *Least squares*.

  Foundational problem in statistic and numerical analysis.

  Given n points in the plane: $(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)$.

  Find a line $y = ax + b$ that minimizes:

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



- *Solution*. Calculus $\Rightarrow$ min error is achieved when

$$a = \frac{n \sum_i x_i y_i - (\sum_i x_i)(\sum_i y_i)}{n \sum_i x_i^2 - (\sum_i y_i)^2}, \quad b = \frac{\sum_i y_i - a \sum_i x_i}{n}$$

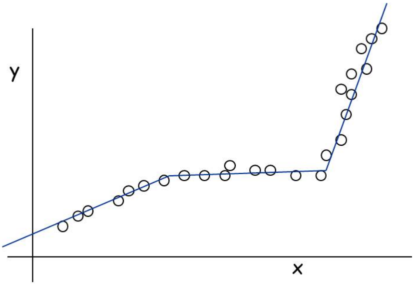
Segmented Least Squares

- *Segmented least squares*.

  Points lie roughly on a sequence of several line segments.

  Given n points in the plane: $(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)$ with $x_1 < x_2 < \cdots < x_n$, find a sequence of lines that minimizes $f(x)$.

Q. What is a reasonable choice for $f(x)$ to balance accuracy and parsimony?



Segmented Least Squares

- Given n points in the plane: $(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)$ with $x_1 < x_2 < \cdots < x_n$, find a sequence of lines that minimizes $f(x) = E + cL$.
- $E =$ the total squared errors in each segment.
- $L =$ the number of lines.
- *Notation.*

  $OPT(j) =$ minimum cost for points $p_1, p_2, \cdots, p_j$.

  $e(i, j) =$ minimum sum of squares for points $p_i, p_{i+1}, \cdots, p_j$.
- To compute $OPT(j)$:
- Last segment uses points $p_i, p_{i+1}, \cdots, p_j$ for some i.

Segmented Least Squares Algorithm

- $Cost = e(i,j) + c + OPT(i-1)$.

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \min\{e(i,j) + c + OPT(i-1)\} & \text{otherwise} \end{cases}$$

$SEGMENTED-LEAST-SQUARES(n, p_1, \cdots, p_n, c)$

```
for j = 1 to n do
    for i = 1 to n do
        Compute the least squares e(i, j) for the segment p_i, p_{i+1}, ..., p_j.
    end for
end for
M[0] ← 0.
for j = 1 to n do
    M[j] ← min_{1≤i≤j}{e_ij + c + M[i − 1]}.
end for
return M[n].
```

Segmented Least Squares Analysis

Theorem 1 (Bellman 1961)

The dynamic programming algorithm solves the segmented least squares problem in $O(n^3)$ time and $O(n^2)$ space.

Pf. Bottleneck = computing $e(i, j)$ for $O(n^2)$ pairs.

- $O(n)$ per pair using formula.

$$a = \frac{n \sum_i x_i y_i - (\sum_i x_i)(\sum_i y_i)}{n \sum_i x_i^2 - (\sum_i y_i)^2}, \quad b = \frac{\sum_i y_i - a \sum_i x_i}{n}. \square$$

Homework

- Read Chapter 6 of the textbook.
- Exercises 6 & 8 in Chapter 6.