COMPSCI 311: Introduction to Algorithms

Lecture 13: Dynamic Programming

Marius Minea

University of Massachusetts Amherst

slides credit: Dan Sheldon

Dynamic Programming Recipe

- Step 1: Devise simple recursive algorithm
  - Flavor: make "first choice", then recursively solve remaining part of the problem
- Step 2: Write recurrence for optimal value
- Step 3: Design bottom-up iterative algorithm

- Weighted interval scheduling: first-choice is binary
- Rod-cutting: first choice has n options
- Subset Sum: need to "add a variable" (one more dimension)

Subset Sum: Problem Formulation

- **Input**
  - Items $1, 2, \ldots, n$
  - Weights w_i for all items (integers)
  - Capacity W
- **Goal**: select a subset S whose total weight is as large as possible without exceeding W.
- Subset Sum: need to "add a variable" to recurrence

Step 1: Recursive Algorithm, First Try

- Let O be optimal solution on items 1 through j. Is $j \in O$ or not?
- SubsetSum(j)

```
if j = 0 then return 0
▷ Case 1: j ∉ O
vmax = SubsetSum(j-1)
▷ Case 2: j ∈ O
if w_j ≤ W then                          ▷ else skip, can't fit w_j
    vmax = max(vmax, ?? w_j + SubsetSum(j-1) ??)
end if
return vmax
```

- What doesn't work?
  Second call to SubsetSum(j-1) no longer has capacity W.
  Solution: must add extra parameter (problem dimension)

Step 1: Recursive Algorithm, Add a Variable

- Find value of optimal solution O on items $\{1, 2, \ldots, j\}$ when the remaining capacity is w
- SubsetSum(j,w)

```
if j = 0 then return 0
▷ Case 1: j ∉ O
vmax = SubsetSum(j − 1, w)
▷ Case 2: j ∈ O
if w_j ≤ w then
    vmax = max(vmax, w_j + SubsetSum(j − 1, w − w_j))
end if
return vmax
```

Recurrence

- Let $\mathrm{OPT}(j, w)$ be the maximum weight of any subset of items $\{1, \ldots, j\}$ that does not exceed w

$$\mathrm{OPT}(j, w) = \begin{cases} \mathrm{OPT}(j-1, w) & w_j > w \\ \max \left\{ \begin{array}{c} \mathrm{OPT}(j-1, w) \\ w_j + \mathrm{OPT}(j-1, w - w_j) \end{array} \right\} & w_j \le w \end{cases}$$

- Base case: $\mathrm{OPT}(0, w) = 0$ for all $w = 0, 1, \ldots, W$.
- Questions
  - Do we need a base case for $\mathrm{OPT}(j, 0)$?
  - What is overall optimum to original problem? $\mathrm{OPT}(n, W)$

Step 3: Iterative Algorithm

```
SubsetSum(n,W)
  Initialize array M[0..n, 0..W]
  Set M[0,w] = 0 for w = 0,...,W
  for j = 1 to n do
     for w = 1 to W do
        if w_j > w then M[j,w] = M[j-1,w]
        else M[j,w] = max(M[j-1,w], w_j + M[j-1, w-w_j])
     end for
  end for
  return M[n,W]
```

- Running Time? $\Theta(nW)$. Note: this is "pseudopolynomial". Not strictly polynomial, because it can be exponential in the number of *bits* used to represent the values.

Polynomial vs. pseudo-polynomial

- So far, we've expressed complexity depending on problem size (n, $|V|$, $|E|$)
- For numbers involved (sorted array elements, edge weights, etc.), we assumed comparison, addition, etc., take constant time
- Actually, these operations depend on *bit width*: addition is linear, multiplication is quadratic (or better: fast multiply), etc.
- For subset sum, W appears as factor in complexity, $O(nW)$ But W is *exponential* in the number of bits used to represent W, thus the difference!

Clicker Question 1

```
for j = 1 to n do
   for w = 1 to W do
      if w_j > w then M[j,w] = M[j-1,w]
      else M[j,w] = max(M[j-1,w], w_j + M[j-1, w-w_j])
   end for
end for
```

In the computation above, can I switch outer and inner loops (outer j, inner w $\rightarrow$ outer w, inner j)

A: Yes

B: No

Knapsack Problem

Introduce an additional parameter, **value**

- **Input**
  - Items $1, 2, \ldots, n$
  - Weights w_i for all items (integers)
  - Values v_i for all items (integers)
  - Capacity W
- **Goal**: select a subset S whose total **value** is as large as possible without exceeding W.
- Does the solution change ?

Clicker Question 2

Recall recurrence for subset sum: $\text{OPT}(j, w)$

$$= \begin{cases} \text{OPT}(j-1, w) & w_j > w \\ \max(\text{OPT}(j-1, w), w_j + \text{OPT}(j-1, w-w_j)) & w_j \leq w \end{cases}$$

The solution for the knapsack problem

A: Requires an additional dimension for values

B: Is still two-dimensional, but its complexity increases

C: Is still two-dimensional, with same complexity

Same solution, just add values $v_j + \text{OPT} \ldots$ instead of weights

Clicker Question 3

How does the solution to the knapsack problem change if we consider reals instead of integers?

A: Same for real weights and values

B: Same for real values, different for real weights

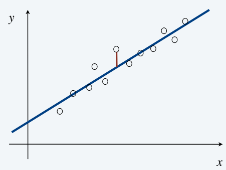
C: Same for real weights, different for real values

Least squares

Least squares. Foundational problem in statistics.

- 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:

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



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

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

Segmented least squares

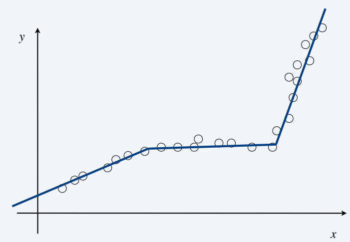
Segmented least squares.

- Points lie roughly on a sequence of several line segments.
- Given n points in the plane: $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ with $x_1 < x_2 < \ldots < x_n$, find a sequence of lines that minimizes $f(x)$.

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

goodness of fit

number of lines



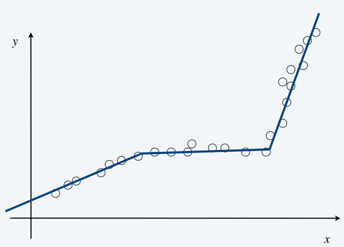
Segmented least squares

Segmented least squares.

- Points lie roughly on a sequence of several line segments.
- Given n points in the plane: $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ with $x_1 < x_2 < \ldots < x_n$, find a sequence of lines that minimizes $f(x)$.

Goal. Minimize $f(x) = E + c\,L$ for some constant $c > 0$, where

- E = sum of the sums of the squared errors in each segment.
- L = number of lines.



Dynamic programming: multiway choice

Notation.

- $OPT(j)$ = minimum cost for points $p_1, p_2, \ldots, p_j$.
- e_{ij} = SSE for for points $p_i, p_{i+1}, \ldots, p_j$.

To compute $OPT(j)$:

- Last segment uses points $p_i, p_{i+1}, \ldots, p_j$ for some $i \le j$.
- Cost = $e_{ij} + c + OPT(i-1)$. ← optimal substructure property (proof via exchange argument)

Bellman equation.

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \min\limits_{1 \le i \le j} \{ e_{ij} + c + OPT(i-1) \} & \text{if } j > 0 \end{cases}$$

Segmented least squares algorithm

```
SEGMENTED-LEAST-SQUARES(n, p1, …, pn, c)

FOR j = 1 TO n
   FOR i = 1 TO j
      Compute the SSE eij for the points pi, pi+1, …, pj.

M[0] ← 0.
FOR j = 1 TO n
   M[j] ← min 1 ≤ i ≤ j { eij + c + M[i – 1] }.    ← previously computed value

RETURN M[n].
```


Segmented least squares analysis

Theorem. [Bellman 1961] DP algorithm solves the segmented least squares problem in $O(n^3)$ time and $O(n^2)$ space.

Pf.

- Bottleneck = computing SSE e_{ij} for each i and j.

$$a_{ij} = \frac{n \sum_k x_k y_k - (\sum_k x_k)(\sum_k y_k)}{n \sum_k x_k^2 - (\sum_k x_k)^2}, \quad b_{ij} = \frac{\sum_k y_k - a_{ij} \sum_k x_k}{n}$$

- $O(n)$ to compute e_{ij}. ▪

Remark. Can be improved to $O(n^2)$ time.

- For each i: precompute cumulative sums $\sum_{k=1}^{i} x_k, \ \sum_{k=1}^{i} y_k, \ \sum_{k=1}^{i} x_k^2, \ \sum_{k=1}^{i} x_k y_k$.
- Using cumulative sums, can compute e_{ij} in $O(1)$ time.