Analysis of Algorithms, I

CSOR W4231.002

Eleni Drinea
Computer Science Department

Columbia University

Tuesday, February 16, 2016

Outline

Today

1. Recap: matrix chain multiplication
   - Organizing DP computations
2. Segmented least squares
   - An exponential recursive algorithm
3. A Dynamic Programming (DP) solution
   - A quadratic iterative algorithm
   - Applying the DP principle

Matrix chain multiplication

Input

- n matrices $A_1, A_2, \ldots, A_n$;
- matrix A_i has dimensions $p_{i-1} \times p_i$.

Output

- a way to compute the product $A_1 \cdots A_n$ so that the number of scalar multiplications performed is minimized;
- the minimum number of scalar multiplications.

Remark 1.

- *We do not want to compute the actual product.*
- *We want an optimal solution and its cost.*
- *There may be many optimal solutions (with the same cost).*

Parenthesized products of matrices

Definition 1.

A product of matrices is fully parenthesized if it is

1. a single matrix; or
2. the product of two parenthesized matrices, surrounded by parentheses.

Examples: $A_1, (A_1 A_2), ((A_1 A_2) A_3)$ are fully parenthesized.

Remark: a parenthesization defines a way to compute the product of the input matrices. Thus the cost of the optimal parenthesization is the minimum cost of computing $A_1 \cdots A_n$.

A divide and conquer attempt

Let $A_{i,j}$ = optimal parenthesization of the product $A_i \cdots A_j$.

By Definition 1, there exists $1 \leq k^* \leq n-1$ such that

$$A_{1,n} = ((A_1 \cdots A_{k^*})(A_{k^*+1} \cdots A_n)).$$

In fact, we showed something stronger:

$$A_{1,n} = (A_{1,k^*} \; A_{k^*+1,n})$$

Hence, the overall optimal solution contains optimal solutions to subproblems (optimal substructure).

A recurrence for the optimal cost

Notation: $OPT(i, j) =$ optimal cost for computing $A_i \cdots A_j$.

- $OPT(1, n) = OPT(1, k^*) + OPT(k^* + 1, n) + p_0 p_{k^*} p_n$

$\Rightarrow$ *If* we knew k^*, we could compute $OPT(1, n)$ recursively!

- *Solution:* consider **every possible value** for k; set k^* to the one that achieves the minimum cost. Then

$$\begin{aligned} OPT(1, n) &= \min_{1 \leq k < n} \Big\{ OPT(1, k) + OPT(k + 1, n) + p_0 p_k p_n \Big\} \quad (1) \\ k^* &= \arg \min_{1 \leq k < n} \Big\{ OPT(1, k) + OPT(k + 1, n) + p_0 p_k p_n \Big\} \end{aligned}$$

△ Recurrence (1) yields an **exponential** recursive algorithm (overlapping subproblems).

Elements of DP in matrix chain multplication

1. Overlapping subproblems
2. An "easy-to-compute" recurrence for the cost of the optimal solution in terms of the costs of optimal solutions to appropriate subproblems.
3. A **natural ordering** of the subproblems from smallest to largest that will allow us to solve them iteratively, in a bottom-up fashion.
4. A polynomial number of subproblems.

From recursion to dynamic programming

Recurrence (1) offers a natural ordering of the subproblems $OPT(i, j)$ from smaller to larger: for $1 \leq i \leq j \leq n$, let

$$OPT(i, j) = \begin{cases} 0 & \text{, if } i = j \\ \min\limits_{i \leq k < j} \left\{ OPT(i, k) + OPT(k + 1, j) + p_{i-1} p_k p_j \right\} & \text{, if } i < j \end{cases}$$

- There are $\Theta(n^2)$ subproblems that can be computed iteratively, from smaller to larger, by increasing the difference $j - i$.
- We want $OPT(1, n)$.
- $OPT(i, j)$ requires $\Theta(j - i) = O(n)$ work **if**, when solving $OPT(i, j)$, we have already solved all subproblems it uses.

Dynamic programming table M

Define matrices $M[1:n, 1:n]$, $S[1:n-1, 2:n]$.

- For $i \leq j$, $M[i,j]$ stores $OPT(i,j)$.

$$M[i,j] = \begin{cases} 0 & \text{, if } i = j \\ \min\limits_{i \leq k < j} \left\{ M[i,k] + M[k+1,j] + p_{i-1}p_kp_j \right\} & \text{, if } i < j \end{cases} \tag{2}$$

- For $i < j$, $S[i,j]$ stores optimal division point for $A_i \cdots A_j$.

$$S[i,j] = \ell, \quad \text{if } A_{i,j} = A_{i,\ell}A_{\ell+1,j} \tag{3}$$

S allows for fast reconstruction of the optimal parenthesizaton *(coming up)*.

Filling in M

- Only need fill in the half of M above (and including) the main diagonal.
- Starting from the main diagonal, fill in M diagonal by diagonal.
- Last entry to fill in: $M[1, n]$, corresponding to the optimal cost for computing $A_1 \cdots A_n$.
- Time to fill in the entries of M, S: $O(n^3)$
  - $\Theta(n^2)$ entries to fill in
  - each entry requires $\Theta(j - i) = O(n)$ work
- Space: $\Theta(n^2)$

Example: $n = 4, p_0 = 6, p_1 = 1, p_2 = 5, p_3 = 2, p_4 = 3$

Use recurrences 2, 3 to fill in tables M, S for the following instance:

- 6×1 matrix A_1
- 1×5 matrix A_2
- 5×2 matrix A_3
- 2×3 matrix A_4

1. Entry $M[i, j]$ is the cost $OPT(i, j)$ of subproblem $A_i \cdots A_j$.
2. $S[i, j]$ gives the optimal division point for subproblem $A_i \cdots A_j$.
3. Subproblems are defined for $i \leq j$; only subproblems with $i < j$ have division points (thus the rows of S correspond to $i = 1, 2, 3$ while its columns to $j = 2, 3, 4$). Hence M, S are empty below the main diagonal.

$M =$

0	30	22	34
-	0	10	16
-	-	0	30
-	-	-	0

$S =$

1	1	1
-	2	3
-	-	3

Pseudocode for filling in M, S in $O(n^3)$ (from CLRS)

```
MATRIX-CHAIN-ORDER(p)
n = p.length − 1
let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
for i = 1 to n
    m[i, i] = 0
for l = 2 to n                    // l is the chain length
    for i = 1 to n − l + 1
        j = i + l − 1
        m[i, j] = ∞
        for k = i to j − 1
            q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
            if q < m[i, j]
                m[i, j] = q
                s[i, j] = k
return m and s
```

Reconstructing the optimal parenthesization (from CLRS)

Recall that a fully parenthesized product of matrices is

1. a single matrix; or
2. the product of two parenthesized matrices, surrounded by parentheses.

```
PRINT-OPTIMAL-PARENS(s, i, j)
if i == j
    print "A"_i
else print "("
    PRINT-OPTIMAL-PARENS(s, i, s[i, j])
    PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
    print ")"
```

Memoized recursion

Use the original recursive algorithm together with M:

- initialize M to ∞ above the main diagonal and to 0 on the main diagonal.
- to solve a subproblem, look up its value in M
  - if it is ∞, solve the subproblem **and** store its cost in M;
  - else, directly use its value from M.

Remark 2.

- *The memoized recursive algorithm solves every subproblem once, thus overcoming the main source of inefficiency of the original recursive algorithm.*
- *Running time:* $O(n^3)$.

Memoized recursion pseudocode (from CLRS)

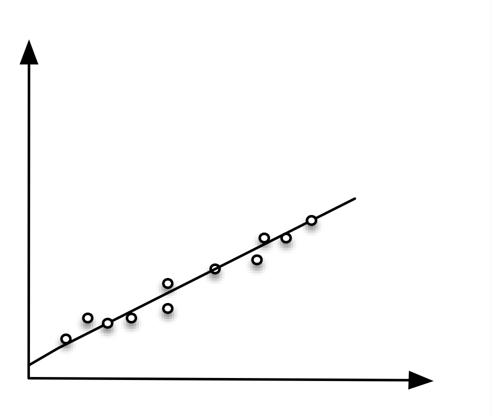
```
MEMOIZED-MATRIX-CHAIN(p)
n = p.length - 1
let m[1..n, 1..n] be a new table
for i = 1 to n
    for j = i to n
        m[i, j] = ∞
return LOOKUP-CHAIN(m, p, 1, n)

LOOKUP-CHAIN(m, p, i, j)
if m[i, j] < ∞
    return m[i, j]
if i == j
    m[i, j] = 0
else for k = i to j - 1
        q = LOOKUP-CHAIN(m, p, i, k)
              + LOOKUP-CHAIN(m, p, k + 1, j) + p_{i-1} p_k p_j
        if q < m[i, j]
            m[i, j] = q
return m[i, j]
```

Today

1. Recap: matrix chain multiplication
   - Organizing DP computations
2. Segmented least squares
   - An exponential recursive algorithm
3. A Dynamic Programming (DP) solution
   - A quadratic iterative algorithm
   - Applying the DP principle

Linear least squares fitting

A foundational problem in statistics: find a line of *best fit* through some data points.



Linear least squares fitting

Input: a set P of n data points (x_1, y_1), (x_2, y_2), ..., (x_n, y_n); we assume $x_1 < x_2 < \ldots < x_n$.

Output: the line L defined as $y = ax + b$ that minimizes the error

$$err(L, P) = \sum_{i=1}^{n} (y_i - ax_i - b)^2 \tag{4}$$

Linear least squares fitting: solution

Given a set P of data points, we can use calculus to show that the line L given by $y = ax + b$ that minimizes

$$err(L, P) = \sum_{i=1}^{n} (y_i - ax_i - b)^2 \tag{5}$$

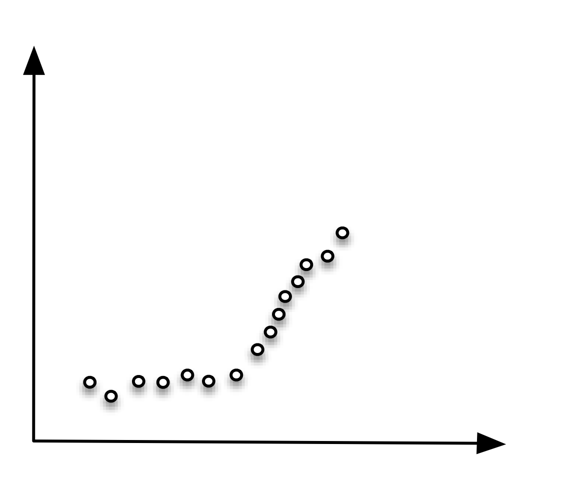
satisfies

$$a = \frac{n \sum_i x_i y_i - (\sum_i x_i)(\sum_i y_i)}{n \sum_i x_i^2 - (\sum_i x_i)^2} \tag{6}$$

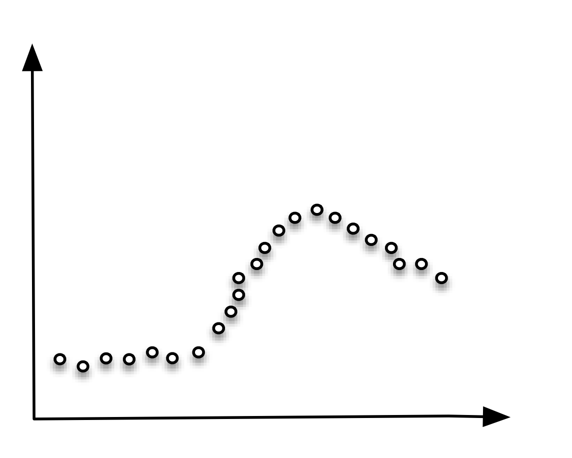
$$b = \frac{\sum_i y_i - a \sum_i x_i}{n} \tag{7}$$

How fast can we compute a, b?

What if the data changes direction?



What if the data changes direction more than once?



How to detect change in the data

- Any single line would have large error.
- **Idea 1:** hardcode number of lines to 2 (or some *fixed* m).
  - Fails for the dataset on the previous slide.
- **Idea 2:** pass an *arbitrary set* of lines through the points and seek the set of lines that minimizes the error.
  - Trivial solution: have a different line pass through each pair of consecutive points in P.
- **Idea 3:** fit the points well, using as few lines as possible.
  - Trade-off between complexity and error of the model

Formalizing the problem

Input: data set $P = \{p_1, \ldots, p_n\}$ of points on the plane.

- A segment $S = \{p_i, p_{i+1}, \ldots, p_j\}$ is a contiguous subset of the input.
- Let $\mathcal{A}$ be a partition of P into $m_{\mathcal{A}}$ segments $S_1, S_2, \ldots, S_{m_{\mathcal{A}}}$.
  For every segment S_k, use (5), (6), (7) to compute a line L_k that minimizes $err(L_k, S_k)$.
- Let $C > 0$ be a fixed multiplier. The cost of the partition is

$$\sum_{S_k \in \mathcal{A}} err(L_k, S_k) + m_{\mathcal{A}} \cdot C$$

Segmented least squares

This problem is an instance of change detection in data mining and statistics.

Input: A set P of n data points $p_i = (x_i, y_i)$ as before.

Output: A segmentation $\mathcal{A}^* = \{S_1, S_2, \ldots, S_{m_{\mathcal{A}^*}}\}$ of P whose cost

$$\sum_{S_k \in \mathcal{A}^*} err(L_k, S_k) + m_{\mathcal{A}^*} C$$

is minimum.

A brute force approach

We can find the optimal partition (that is, the one incurring the minimum cost) by exhaustive search.

- Enumerate every possible partition (segmentation) and compute its cost.
- Output the one that incurs the minimum cost.

△ $O(2^n)$ partitions

A crucial observation regarding the last data point

Consider the last point p_n in the data set.

- p_n belongs to a single segment in the optimal partition.
- That segment starts at an earlier point p_i, for some $1 \leq i \leq n$.

This suggests a recursive solution: **if** we knew where the last segment starts, then we could remove it and recursively solve the problem on the remaining points $\{p_1, \ldots, p_{i-1}\}$.

A recursive approach

- Let $OPT(j)$ = cost of optimal partition for points $p_1, \ldots, p_j$.
- Then, if the last segment of the optimal partition is $\{p_i, \ldots, p_n\}$, the cost of the optimal solution is

$$OPT(n) = err(L, \{p_i, \ldots, p_n\}) + C + OPT(i-1).$$

- But we don't know where the last segment starts! *How do we find the point p_i?*
- Set

$$OPT(n) = \min_{1 \leq i \leq n} \Big\{ err(L, \{p_i, \ldots, p_n\}) + C + OPT(i-1) \Big\}.$$

A recurrence for the optimal solution

Notation: let $e_{i,j} = err(L, \{p_i, \ldots, p_j\})$, for $1 \leq i \leq j \leq n$. Then

$$OPT(n) = \min_{1 \leq i \leq n} \left\{ e_{i,n} + C + OPT(i-1) \right\}.$$

If we apply the above expression recursively to remove the last segment, we obtain the recurrence

$$OPT(j) = \min_{1 \leq i \leq j} \left\{ e_{i,j} + C + OPT(i-1) \right\} \tag{8}$$

Remark 3.

1. *We can precompute and store all* $e_{i,j}$ *using equations* (5), (6), (7) *in* $O(n^3)$ *time. Can be improved to* $O(n^2)$*.*
2. *The natural recursive algorithm arising from recurrence* (8) *is* **not** *efficient (think about its recursion tree!).*

Exponential-time recursion

Notation: $T(n)$ = time to compute optimal partition for n points.

Then
$$T(n) \geq T(n-1) + T(n-2).$$

- Can show that $T(n) \geq F_n$, the n-th Fibonacci number (by strong induction on n).
- From Problem 5a in Homework 1, $F_n = \Omega(2^{n/2})$.
- Hence $T(n) = \Omega(2^{n/2})$.

$\Rightarrow$ The recursive algorithm requires $\Omega(2^{n/2})$ time.

Today

1. Recap: matrix chain multiplication
   - Organizing DP computations
2. Segmented least squares
   - An exponential recursive algorithm
3. A Dynamic Programming (DP) solution
   - A quadratic iterative algorithm
   - Applying the DP principle

Elements of DP in segmented least squares

1. Overlapping subproblems
2. An easy-to-compute recurrence (8) for combining solutions to the smaller subproblems into a solution to a larger subproblem in $O(n)$ time (once smaller subproblems have been solved).
3. Iterative, bottom-up computations: compute the subproblems from smallest (0 points) to largest (n points), iteratively.
4. Small number of subproblems: we only need to solve n subproblems.

A dynamic programming approach

$$OPT(j) = \min_{1 \leq i \leq j} \left\{ e_{i,j} + C + OPT(i-1) \right\}$$

- The optimal solution to the subproblem on $p_1, \ldots, p_j$ contains optimal solutions to smaller subproblems.
- Recurrence 8 provides an **ordering** of the subproblems from smaller to larger, with the subproblem of size 0 being the smallest and the subproblem of size n the largest.

$\Rightarrow$ There are $n+1$ subproblems in total. Solving the j-th subproblem requires $\Theta(j) = O(n)$ time.

$\Rightarrow$ The overall running time is $O(n^2)$.

- Boundary conditions: $OPT(0) = 0$.
- Segment $p_k, \ldots, p_j$ appears in the optimal solution only if the minimum in the expression above is achieved for $i = k$.

An iterative algorithm for segmented least squares

Let M be an array of n entries. $M[i]$ stores the cost of the optimal segmentation of the first i data points.

$\texttt{SegmentedLS}(n, P)$

  $M[0] = 0$
  for all pairs $i \leq j$ **do**
    Compute $e_{i,j}$ for segment $p_i, \ldots, p_j$ using (5), (6), (7)
  end for
  for $j = 1$ to n **do**
    $M[j] = \min_{1 \leq i \leq j} \{e_{i,j} + C + M[i-1]\}$
  end for
  Return $M[n]$

Running time: time required to fill in dynamic programming array M is $O(n^3) + O(n^2)$. Can be brought down to $O(n^2)$.

Reconstructing an optimal segmentation

- Suppose we want the optimal solution in addition to its value, that is, the actual segmentation that achieves the minimum cost $M[n]$.
- We can trace back through the dynamic programming array M to compute the optimal segmentation.

Initial call: `OPTSegmentation`(n)

`OPTSegmentation`(j)

if $(j == 0)$ **then** return

else

Find $1 \leq i \leq j$ such that $M[j] = e_{i,j} + C + M[i-1]$

`OPTSegmentation`$(i-1)$

Output segment $\{p_i, \ldots, p_j\}$

end if

Obtaining efficient algorithms using DP

1. Optimal substructure: the optimal solution to the problem contains optimal solutions to the subproblems.
2. A recurrence for the overall optimal solution in terms of optimal solutions to appropriate subproblems. The recurrence should provide a natural ordering of the subproblems from smaller to larger and require polynomial work for combining solutions to the subproblems.
3. Iterative, bottom-up computation of subproblems, from smaller to larger.
4. Small number of subproblems (polynomial in n).

Dynamic programming vs Divide & Conquer

- They both combine solutions to subproblems to generate the overall solution.
- However, divide and conquer starts with a large problem and divides it into small pieces.
- While dynamic programming works from the bottom up, solving the smallest subproblems first and building optimal solutions to steadily larger problems.