Advanced Design and Analysis Techniques
Dynamic Programming

Yu-Tai Ching

Department of Computer Science
National Chiao Tung University

Previous Mentioned Algorithm Design Techniques

- Incremental
- Divide and Conquer
- Randomized
- Prune and Search

Advanced Design and Analysis Techniques

- Dynamic Programming
- Greedy Approach
- Amortized Analysis

Dynamic Programming

- Apply to optimization problem
- Similar to Divide and Conquer, optimal solution is obtained from subproblems of the same form. But the subproblems are not independent.
- The key technique is to store, or "memorize," the solution to each subproblem in case it should appear.

Greedy Algorithm

- Also apply to optimization problem.
- Optimal solution is obtained from making decision of a sequences of choices.
- To make each choice in a locally optimally solution.

Amortized Analysis

- A tool for analyzing algorithms that perform a sequence of similar operations.
- Some of them (those similar operations) take long time, $O(n)$, for example, but most of them take much less time (usually $O(1)$). (A 2-3-4 Tree example)
- Instead of a rough worst case time bound, we calculate the accurate time bound. The average cost is generally $O(1)$ time.
- Still the worst case, not the average case analysis.

Dynamic Programming

Development of the dynamic programming algorithm is broken into a sequence of 4 steps.

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution in the bottom-up fashion.
4. Construct an optimal solution from computed information.

Assembly-Line Scheduling

- Clonel Motors Corporation produces automobiles.
- There are two assembly lines.
- Automobile chasis enters each assembly line, has parts added to it at numbers of stations,
- finished auto exists at the end of the line.

- Each assembly line has n stations, named $j = 1, 2, \ldots, n$.
- jth station on line i, $(i = 1, 2)$ denoted by $S_{i,j}$.
- Time required for $S_{i,j}$ is $a_{i,j}$. ($a_{1,j} \neq a_{2,j}$ since $S_{1,j}$ and $S_{2,j}$ are built at different time using different technology.)
- Entry time e_i for a chasis to enter assembly line i, and an exist time x_i for the complete auto to exist assembly line i.

- Normal operation: A chasis enters an assembly line, passes through that line only. Time to go from one station to the next within the same assembly line is negligible.
- Rush Order: Chasis passes through the n stations in order, but can switch from one assembly line to the other after any station. There is cost, time to switch from line i to the other after having gone through $S_{i,j}$ is $t_{i,j}$, $i = 1, 2$, $j = 1, \ldots, n-1$.
- Problem: schedule the production line to manufacture one auto so that the total time is minimized.
- Brute force approach, All of the possibilities $= 2^n$.

Characterize the Structure of the Optimal Solution

Consider the "fastest possible way" for a chasis to get from the starting point through station $S_{1,j}$,

- if $j = 1$, there is only one way, it is the fastest.
- For $j = 2, \ldots, n$, there are two choices,
  - The fastest way is from $S_{1,j-1}$ directly to $S_{1,j}$, takes no switching time.
  - The fastest way is from $S_{2,j-1}$, takes time $t_{2,j-1}$ switch to $S_{1,j}$.

- In either case, previous stations $S_{1,j-1}$ or $S_{2,j-1}$ must be finished through the best possible (fatest) way. If not, we substitute the way by a faster one to yield a faster way throught station $S_{i,j}$, controdict to the "fastest possible way through $S_{1,j}$.
- The optimal substructure: An optimal solution containing within it, an optimal solution to subproblem.

Recursive Solution

Define the value of an optimal solution recursively in terms of the optimal solution to subproblem.

- $f_i[j]$: fastest possible time to get a chasis from the starting point through station $S_{i,j}$.
- f^*: The fastest time to get a chasis all the way through the factory,

$$f^* = \min(f_1[n] + x_1, f_2[n] + x_2).$$

$$f_1[1] = e_1 + a_{1,1}, \tag{1}$$

$$f_2[1] = e_2 + a_{2,1}. \tag{2}$$

For $f_1[j]$, $j = 2, 3, \ldots, n$,

- If it was $S_{1,j-1}$, $f_1[j] = f_1[j-1] + a_{1,j}$,
- if it was $s_{2,j-1}$, $f_1[j] = f_2[j-1] + t_{2,j-1} + a_{1,j}$.

Thus the optimal solution,

$$f_1[j] = \min(f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2.j-1} + a_{1,j}). \tag{3}$$

Symmetrically, we have

$$f_2[j] = \min(f_2[j-1] + a_{2,j}, f_1[j-1] + t_{1,j-1} + a_{2,j}). \tag{4}$$

Combining equations (1), (2), (3), and (4),

$$f_1[j] = \begin{cases} e_1 + a_{1,1} & \text{if } j = 1 \\ \min(f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j}) & \text{if } j \geq 2 \end{cases}$$

$$f_2[j] = \begin{cases} e_2 + a_{2,1} & \text{if } j = 1 \\ \min(f_2[j-1] + a_{2,j}, f_1[j-1] + t_{1,j-1} + a_{2,j}) & \text{if } j \geq 2 \end{cases}$$

- Computing the fastest times, cannot be implemented by using "direct recursion".
- Compute the solution (the sechdule).

Define $l_i[j]$ to be the line number (1 or 2) whose station $j-1$ is used in a fastest way through station $S_{i,j}$

Rod Cutting Problem

- Decide the best way to cut a rod of length n.
- For $i = 1, 2, \ldots,$ the price p_i for a i inches rod.
- The Rod cutting problem: Given a rod of length n inches, and a table of price p_i for $i = 1, \ldots, n$, determine the maximum revenue r_n obtained by cutting up the rod and selling the pieces.

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

What if $n = 4$?

Example, n=10

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

r_1	1	from solution	1	$=$	1
r_2	5	from solution	2	$=$	2
r_3	8	from solution	3	$=$	3
r_4	10	from solution	4	$=$	2+2
r_5	13	from solution	5	$=$	2+3
r_6	17	from solution	6	$=$	6
r_7	18	from solution	7	$=$	$1 + 6$ or $7 = 2 + 2 + 3$
r_8	22	from solution	8	$=$	$2 + 6$
r_9	25	from solution	9	$=$	$3 + 6$
r_{10}	30	from solution	10	$=$	10

General Expression for r_n

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, + \ldots, r_{n-2} + r_2, r_{n-1} + r_1) \quad (5)$$

- r_n : the optimal revenue of cutting a rod of length n
- equal to $r_k + r_{n-k}$: Length n rod cut into rods of length k and $n-k$,
- and sum of the optimal revenues of the two gives the optimal solution of the all (r_n).
- An optimal solution consists of two optimal solutions, optimal substructures.

- Eq. (5) Can be rewritten as

$$r_n = \max_{1 \le i \le n} (p_i + r_{n-i}) \tag{6}$$

- Left (p_i part): 1 piece, i inches, revenue p_i,
- Right (r_{n-i}): Find the optimal cut.
- Relate optimal solution to one optimal solution

Solve Equation 6

- Convert Equation 6 to C-code $\textsc{Cut\_Rod}(p,n)$,
- Compile and run, takes a long time even n is small, say n=40.
- Increasing n by 1, run time increases a lot, why?
- Recursion calls itself, and solve same problem repeatedly.
- Fig. 15.3.
- Total number of calls made to $\textsc{Cut\_Rod}$ of length n,
- $T(0) = 1$
- $T(n) = 1 + \sum_{j=0}^{n-1} T(j)$,
- $T(n) = 2^n$.

Using Dynamic Programming

- Prevent solving same problem repeatedly,
- Arrange for each subproblem to be solved only once, and save (remember) the solution- solution obtained by table look up.
- Use memory to save computing time, time-memory trade-off.
- Top-Down and Bottom-up.

EXTENDED-BOTTOM-UP-CUT-ROD(p, n)

```
let r[0..n] and s[0..n] be new arrays
r[0] = 0
for j = 1 to n
  q = −∞
  for i = 1 to j
    if q < p[i] + r[j − i]
      q = p[i] + r[j − i]
      s[j] = i
  r[j] = q
return r and s
```

pp. 369, $n = 4$

Matrix-chain multiplication

Given a sequence of $< A_1, A_2, ..., A_n >$ of n matrices to be multiplied, and we are looking for the product.
If the chain of the matrices is $< A_1, A_2, A_3, A_4 >$, five ways to parenthesize, all get same product
$(A_1, (A_2, (A_3, A_4)))$
$(A_1, ((A_2, A_3), A_4))$
$((A_1, A_2), (A_3, A_4))$
$((A_1, (A_2, A_3)), A_4)$
$(((A_1, A_2), A_3), A_4)$

Cost for Matrix Multiplication

- A is $p \times q$, B is $q \times r$, $C = A \times B$ then C is $p \times r$.
- Time for computing C is dominated by scalar multiplications which is $p \times q \times r$.

Cost for Matrix Multiplication

- Consider a chain $< A_1, A_2, A_3 >$
- Dimensions are 10×100, 100×5, 5×50.
- if $((A_1 A_2) A_3)$, we have $10 \cdot 100 \cdot 5 + 10 \cdot 5 \cdot 50 = 7500$ scalar multiplications.
- if $(A_1(A_2 A_3))$, we have $100 \cdot 5 \cdot 50 + 10 \cdot 100 \cdot 50 = 75{,}000$ scalar multiplications.

Matrix-Chain Multiplication Problem:
Given a chain of $< A_1, A_2, \ldots, A_n >$ of n matrices, where $i = 1, 2, \ldots, n$, matrix A_i has dimension $p_{i-1} \times p_i$, fully parenthesize the product $A_1, A_2, \ldots, A_n$ in a way that minimizes the number of scalar multiplications.

Matrix Chain Multiplication

Can we test all possible parenthesizes and look for the smallest?
Let $P(n)$ denote the total number of ways to parenthesize $A_1 A_2 \dots A_n$.
We can split a sequence between k and $k+1$ and then parenthesize the left and the right.
k can be anyone of 1, 2, ..., $n-1$.

$$P(n) = \begin{cases} 1 & \text{if} \quad n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if} \quad n \geq 2. \end{cases}$$

$P(n) = C(n-1)$ where $C(n) = \frac{1}{n+1} \left(\begin{array}{c} 2n \\ n \end{array} \right) = \Omega(\frac{4^n}{n^{\frac{3}{2}}})$.

Matrix Chain Multiplication

Structure of an optimal parenthesization

- Let $A_{i..j}$ denote the matrix obtained by evaluating the product of $A_i A_{i+1} \ldots A_j$.
- An optimal parenthesization of the product $A_1 A_2 \ldots A_n$ splits the product between A_k and A_{k+1}, $1 \leq k < n$.
- The final result is the product of the left $(A_1 A_2 \ldots A_k)$ and right $(A_{k+1} A_{k+2} \ldots A_n)$.
- The cost for this optimal parenthesization, cost for computing the left $(A_1 A_2 \ldots A_k)$ + cost for computing the right $(A_{k+1} A_{k+2} \ldots A_n)$ + cost for the final product.
- Key observation: Parenthesizations of $(A_1 A_2 \ldots A_k)$ and $(A_{k+1} A_{k+2} \ldots A_n)$ must be optimal. Why?

Matrix Chain Multiplication

A Recursive Solution (1)

- Let the optimal cost for computing $A_i A_{i+1} \dots A_j$ be $m[i,j]$, thus $m[1,n]$ is what we are looking for.
- $m[i,i] = 0$ since there are no multiplications.
- Form the structure of the optimal solution, there is a split between k and $k+1$ on $A_i A_{i+1} \dots A_j$. And the costs for computing $A_{i..k}$ and $A_{k+1,j}$ are optimal.

Matrix Chain Multiplication

A Recursive Solution (2)

- Computing$A_{i..k}A_{(k+1)..j}$ takes $p_{i-1}p_kp_j$ scalar multiplications, thus $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j$.
- We don't know where is the split k.

Matrix Chain Multiplication

Recursive definition of the minimum cost $m[i,j]$.

$$m[i,j] = \begin{cases} 0 & \text{if} \quad i = j, \\ \min_{i \le k < j}\{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & \text{if} \quad i < j. \end{cases}$$

Matrix Chain Multiplication

Computing the optimal solution

$\underline{A_1}$ $\underline{A_2}$ $\underline{A_3}$ $\underline{A_4}$ $\underline{A_5}$ $i = j$, length of chain $= 1$, boundary condition,

$\underline{A_1 A_2}$ $\underline{A_2 A_3}$ $\underline{A_3 A_4}$ $\underline{A_4 A_5}$ length of chain $= 2$,

$\underline{A_1 A_2 A_3}$ $\underline{A_2 A_3 A_4}$ $\underline{A_3 A_4 A_5}$

$\underline{A_1 A_2 A_3 A_4}$ $\underline{A_2 A_3 A_4 A_5}$

$\underline{A_1 A_2 A_3 A_4 A_5}$

$(A_1, (A_2, (A_3, A_4)))$

$(A_1, ((A_2, A_3), A_4))$

$((A_1, A_2), (A_3, A_4))$

$((A_1, (A_2, A_3)), A_4)$

$(((A_1, A_2), A_3), A_4)$

Matrix Chain Multiplication

An example
A_1 30×35, A_2 35×15, A_3 15×5, A_4 5×10, A_5 10×20, A_6 20×25
Need an array $m[1..n][1..n]$ to memorize the optimal cost, and another array $s[1..n][1..n]$ to keep where the split is. run through the example.

Longest Common Subsequence

Subsequence

- Given a sequence $X =< x_1, x_2, \ldots, x_m >$, another sequence $Z =< z_1, z_2, \ldots, z_k >$ is a subsequence of X if there exists a strictly increasing sequence $< i_1, i_2, \ldots, i_k >$ of indices of X s.t. for all $j = 1, 2, \ldots, k$, we have $x_{i_j} = z_j$.
- Example, $Z =< B, C, D, B >$ is a subsequence of $X =< A, B, C, B, D, A, B >$ with corresponding index sequence $< 2, 3, 5, 7 >$.

Longest Common Subsequence

Common Subsequence
Given two sequences X and Y, we say that a sequence Z is a common subsequence of X and Y if Z is a subsequence of both X and Y.
$X =< A, B, C, B, D, A, B >$ and $Y =< B, D, C, A, B, A >$,
$< B, C, A >$ a subsequence of X and Y.
Longest Common Subsequence (LCS)
We are looking for the subsequence that has the longest length.
$< B, D, A, B >$ is the longest common subsequence since there isn't longer subsequence.

Longest Common Subsequence Brute-force approach?

Given sequences X and Y, check each of the subsequences of X whether it is also a subsequence of Y.
If length of X is n, there are 2^n subsequences. Thus it is not appropriate to use the brute-force approach.

Longest Common Subsequence

Dynamic Programming approach

- Define prefix: Given a sequence $X =< x_1, x_2, \ldots, x_m >$, the ith prefix of X, for $i = 0, 1, \ldots, m$ as $X_i =< x_1, x_2, \ldots, x_i >$.
- if $X =< A, B, C, B, D, A, B >$, then $X_4 =< A, B, C, B >$.
- X_0 is the empty sequence.

Longest Common Subsequence

Theorem: Optimal substructure of an LCS: Let $X =< x_1, x_2, \ldots, x_m >$ and $Y =< y_1, y_2, \ldots, y_n >$ be sequences, and let $Z =< z_1.z_2, \ldots, z_k >$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

Proof of the theorem

1.

(a) if $z_k \neq x_m$ (Note that z_k is the last char of the LCS), then we could append $x_m = y_n$ to Z to obtain a Common Subsequence (CS) of X and Y of length $k+1$, a contradiction.

(b) to show Z_{k-1} is LCS of X_{m-1} and Y_{n-1}. Z_{k-1} is a length-$(k-1)$ CS of X_{m-1} and Y_{n-1}. If Z_{k-1} is a CS but not a LCS, there is a CS W of X_{m-1} and Y_{n-1} with length greater than $k-1$. Then appending $x_m = y_n$ to W produces a CS of X and Y whose length is greater than k, a contradiction.

2. $z_k \neq x_m$ then Z is CS of X_{m-1} and Y. If there were a common sequence W of X_{m-1} and Y with length greater that k, then W would also be a CS of X_m and Y, contradicting to the assumption that Z is an LCS of X and Y.

3. Symmetric to 2.

The Theorem says

1. An LCS of two sequences contains within it an LCS of prefixes of two sequences.
2. if $x_m = y_n$, the LCS of X and Y is obtained from the LCS of X_{m-1} and Y_{n-1} by appending $x_m = y_n$ to the end of the LCS.
3. if $x_m \neq y_n$ then the LCS of X and Y is the larger one of the two
   - LCS of X_{m-1} and Y or
   - LCS of X and Y_{n-1}

Define $C[i,j]$ to be the length of the LCS of the sequences X_i and Y_j

$$C[i,j] = \begin{cases} 0 & \text{if} \quad i = 0 \quad \text{or} \quad j = 0 \\ c[i-1, j-1] + 1 & \text{if} \quad i,j > 0 \quad \text{and} \quad x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{if} \quad i,j > 0 \quad \text{and} \quad x_i \neq y_j \end{cases}$$

trace the example $X = < A, B, C, B, D, A, B >$ and $Y = < B, D, C, A, B, A >$

Optimal Polygon Triangulation

- **Polygon**: A piecewise-linear, closed curve in the plane.
- **Side**: The sequence of the straight lines segments.
- **Vertex**: The point jointing the two consecutive sides.
- **Simple**: A polygon, its side does not cross itself.
- **Interior, boundary, exterior**: Well defined if the polygon is simple.
- **Convex**: The interior of the polygon is convex.

Optimal Polygon Triangulation

- Representation of a polygon: $P = < v_0, v_1, \ldots, v_{n-1} >$
- We consider convex polygon.
  - Given two nonadjacent vertices v_i and v_j, $\overline{v_i v_j}$ is a chord of the polygon.
  - A triangulation of a polygon is a set T of chords of the polygon that divide the polygon into disjoint triangles.
  - No chords intersect. If there are n vertices, there are $n-3$ chord and $n-2$ triangles.

Optimal Polygon Triangulation Problem

- We are given a convex polygon $P = < v_0, v_1, \ldots, v_{n-1} >$ and a weight function w defined on triangles formed by sides and chords of P.
- The problem is to find a triangulation that minimizes the sum of the weights of the triangles in the triangulation.
- For example, the weight could be

$$w(\triangle v_i v_j v_k) = |\overline{v_i v_j}| + |\overline{v_j v_k}| + |\overline{v_k v_i}|$$

where $|\overline{v_i v_j}|$ is the Euclidean distance from v_i to v_j.

Parsing Tree, Matrix Chain Multiplication, Polygon Triangulation

- An expression can be converted to a parsing tree
- $a + b * c == (a + (b * c))$
- $A_1A_2A_3A_4A_5A_6$ $((A_1(A_2A_3))(A_4(A_5A_6)))$
- A parsing tree can be obtained from polygon triangulation.

- To triangulate the polygon $< v_0, v_1, \ldots, v_{n-1} >$, root of the parsing tree, the edge $\overline{v_0 v_{n-1}}$.
- roots of two subtrees, two chords, $\overline{v_0 v_k}$, $\overline{v_{n-1} v_k}$
- We have two polygons to triangulate $< v_0, v_1, \ldots, v_k >$ and $< v_k, v_{k+1}, \ldots, v_{n-1} >$.
- Parsing tree says that, compute $A_2 \times A_3$ first. The resulted matrix times A_1 to get the result in the left subtree. Right subtree is calculated in the similar way. Left subtree times right subtree get the final result.

Why triangulation solves matrix chain multiplication

1. Each chain multiplication problem can be casted to Triangulation problem.
2. Given a matrix chain multiplication $A_1 A_2 \ldots A_n$ we determine a polygon $P = < v_0, v_1, v_2 \ldots, v_n >$
3. A_i has dimension $p_{i-1} \times p_i$, A_i the edge $\overline{v_{i-1} v_i}$.
4. each vertex v_i is associated with a weight p_i.
5. weight of a triangle $w(\triangle v_i v_j v_k) = p_i p_j p_k$

Introduction

- Your introduction goes here!
- Use `itemize` to organize your main points.

Examples

Some examples of commonly used commands and features are included, to help you get started.

Tables and Figures

- Use `tabular` for basic tables — see Table 1, for example.
- You can upload a figure (JPEG, PNG or PDF) using the files menu.
- To include it in your document, use the `includegraphics` command (see the comment below in the source code).

Item	Quantity
Widgets	42
Gadgets	13

Table 1: An example table.

Readable Mathematics

Let $X_1, X_2, \ldots, X_n$ be a sequence of independent and identically distributed random variables with $\mathrm{E}[X_i] = \mu$ and $\mathrm{Var}[X_i] = \sigma^2 < \infty$, and let

$$S_n = \frac{X_1 + X_2 + \cdots + X_n}{n} = \frac{1}{n} \sum_{i}^{n} X_i$$

denote their mean. Then as n approaches infinity, the random variables $\sqrt{n}(S_n - \mu)$ converge in distribution to a normal $\mathcal{N}(0, \sigma^2)$.