Problem

Matrix multiplication is associative

We will use number of multiplications as an indication of running time

- Number of multiplications: $mnp$

Given a sequence of matrices $A_1, A_2, \ldots, A_n$

Want to multiply them together

$A_1 A_2 \cdots A_n$

Overlap Subproblems

Optimal Substructure

Elements of Dynamic Programming

Matrix-Chain Multiplication

Overview
How many ways are there to parenthesize?

Let \( P(n) \) denote the number of parenthesizations of a sequence of \( n \) matrices. When \( n \geq 2 \), there is an outermost parenthesis bringing together two subsequences that will split \( n \) into \( k \) and \( n-k \) subsequences

How many outer splits?

The total product is the number of ways to parenthesize the \( k \) matrices times the number of ways to parenthesize the \( n-k \) matrices

So \( P(2) = 1 \)

So \( P(3) = 2 \)

So \( P(4) = 5 \)

Let \( f(u) \) denote the number of parenthesizations of a sequence of matrices.

Does order matter?

A\( _1 \) A\( _2 \) A\( _3 \)

Where A\( _1 \) is 10 \( \times \) 100, A\( _2 \) is 100 \( \times \) 5, A\( _3 \) is 5 \( \times \) 50

Matrix-chain multiplication problem

Given a chain of matrices A\( _1 \), A\( _2 \), ..., A\( _n \) where A\( _i \) has dimensions \( p_{i-1} \times p_i \), determine the parenthesization that minimizes the number of scalar multiplications.

How long will it take?

\[ f(1000) = f(10 \times 100) + f(100 \times 5) + f(5 \times 50) \]

Where \( f(A) \) is 10 \( \times \) 100, 100 \( \times \) 5, and 5 \( \times \) 50, respectively.
Applying Dynamic Programming

• Construct an optimal solution from computed information
• Compute the value of an optimal solution
• Recursively define the value of an optimal solution
• Characterize the structure of an optimal solution

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
4. Construct an optimal solution from computed information

Does this equation help us solve the order problem?

$P(n) = \sum_{k=1}^{n-1} P(k) P(n-k)$

Applying Dynamic Programming

Brute Force

• Number of parenthesizations can be shown to be exponential in time.
• So to figure out the optimal way of multiplying a sequence of matrices can take exponential time!
• Let \( m[i, j] \) be the minimum number of scalar multiplications needed to compute \( A_{i,j} \). Thus \( m[1, n] \) should be minimum for the overall problem \( A_{1..n} \).

- **Trivial case:** \( m[i, i] = ? \).

  • How do we define the general case \( m[i, j] \)? Recursively!
  
  - Assume that the optimal split is at \( A_k \) and \( A_{k+1} \).
  
  - Without cheating by assuming we know \( k \),
  
    \[
    m[i, j] = \min_{i \leq k < j} \left( m[i, k] + m[k+1, j] + p_{i-1}p_k p_j \right)
    \]

  • Keep track of the splits as well (so we know how to multiply).

- **Optimal substructure:**
  
  - The optimal parenthesization of \( A_{i..j} \) will be used in \( A_{i..k} \) and \( A_{k+1..j} \).
  
  - Suppose we optimally parenthesize \( A_{i..j} \), we split it into \( A_{i..k} \) and \( A_{k+1..j} \).

- **Optimal substructure:**
  
  - 1. Cost of computing \( A_{i..k} \) and \( A_{k+1..j} \) in this way.
  
  - 2. Cost of computing \( A_{i..k} \) and then multiply them together.
  
  - 3. Cost of computing \( A_{k+1..j} \) and then multiply them together.
  
  - Let \( \delta_f \) be the minimum number of scalar multiplications needed to compute \( A_{i..j} \). If \( \delta_f < \delta_f \), then this is the optimal way.

- **Characterize the structure of an optimal solution:**

  - List of matrices into two parts.
  
  - Our characterization makes use of finding optimal place to split.

- **Recursively define the value of an optimal solution:**

  - Let \( m[i, j] \) be the minimum number of scalar multiplications needed to compute \( A_{i,j} \). Thus \( m[i, j] \) should be minimum for the overall problem \( A_{i,j} \).

  - **Trivial case:** \( m[i, i] = ? \).

  - How do we define the general case \( m[i, j] \)? Recursively!
  
  - Assume that the optimal split is at \( A_k \) and \( A_{k+1} \).
  
  - Without cheating by assuming we know \( k \),
  
    \[
    m[i, j] = \min_{i \leq k < j} \left( m[i, k] + m[k+1, j] + p_{i-1}p_k p_j \right)
    \]

  • Keep track of the splits as well (so we know how to multiply).
Function
\[
\begin{align*}
\text{compute} & \quad \gamma = [f \cdot 1]_m \\
b & \quad [f \cdot 1]_m \geq b \implies \text{for } 1 \leq l \leq f \text{ the chain length} \\
& \quad [f \cdot 1 + \gamma]_m + [\gamma]_m = b \\
& \quad 1 - l + i = f \\
& \quad 1 + f - i = l \\
& \quad u \text{ and } j \text{ are new labels} \\
& \quad \text{Matrix-Chain-Order}(d)
\end{align*}
\]

Code

- Computing the value of an optimal solution
  - Straight-forward solution to equation for \( m \) will lead to exponential algorithm
  - No better than brute force approach
  - There is an ordering of subproblems such that smaller subproblems are computed before larger subproblems
  - Can use a bottom-up approach
  - Relatively few distinct subproblems: one for each choice of \( i \) and \( j \) where \( i \leq j \)
  - No better than brute force approach
  - Using matrix multiplication

3: Computing the value of an optimal solution
4. Construct optimal solution from computed info

In the example of Figure 15.5, the call PRINT-OPTIMAL-PARENS(5, 6) prints

\[
(f \left[ f \right] \left[ f \right] \left[ f \right]) \left[ f \right] \left[ f \right]
\]

Then find recurse on \(A[1..k] \) and \(A[k+1..n]\).

Print at \(A[i..j]\):

1. If \(i = j\), print \(A[i]\).
2. Print \(A[i..j]\).

Example

Matrices

Matrices rotated to better reflect order of computations

\[
\begin{array}{cccccc}
0 & 0 & 0 & 0 & 0 & 0 \\
15,750 & 2,625 & 750 & 1,000 & 5,000 \\
7,875 & 4,375 & 2,500 & 3,500 \\
9,375 & 7,125 & 5,375 \\
11,875 & 10,500 \\
15,125
\end{array}
\]

\[
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 \\
1 & 3 & 3 & 5 \\
3 & 3 & 3 \\
3 & 3 \\
3 \\
1 & 2 & 3 & 4 & 5 & 6
\end{array}
\]

Figure 15.5

The \(m\) and \(s\) tables computed by MATRIX-CHAIN-ORDER for \(n = 6\) and the following matrix dimensions:

<table>
<thead>
<tr>
<th>Matrix Dimension</th>
<th>Dimension</th>
</tr>
</thead>
<tbody>
<tr>
<td>30 x 30</td>
<td>10 x 10</td>
</tr>
<tr>
<td>10 x 20</td>
<td>5 x 5</td>
</tr>
<tr>
<td>20 x 20</td>
<td>10 x 10</td>
</tr>
<tr>
<td>25 x 25</td>
<td>15 x 15</td>
</tr>
<tr>
<td>15 x 15</td>
<td>10 x 10</td>
</tr>
<tr>
<td>35 x 35</td>
<td>20 x 20</td>
</tr>
</tbody>
</table>

Matrix \(B\) is:

\[
\begin{array}{cccccc}
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 2 & 0 & 0 & 0 \\
0 & 0 & 0 & 3 & 0 & 0 \\
0 & 0 & 0 & 0 & 4 & 0 \\
0 & 0 & 0 & 0 & 0 & 5
\end{array}
\]

Matrices rotated to better reflect order of computations

Example \(m\) and \(s\) Matrices
Elements of Dynamic Programming

• Optimal Substructure:
  - The division into subproblems will give you the optimal solution.
  - There is an obvious way to break the problem into subproblems.

• Non-overlapping Subproblems:
  - Good clue that dynamic programming might apply.
  - Combination of optimal solutions to subproblems to obtain the solution to a given optimization problem can be obtained by the

Matrix-Chain Multiplication

Elements of Dynamic Programming

Overview

Elements of Dynamic Programming

Non-overlapping Subproblems

Optimal Substructure

Matrix-Chain Multiplication
Common Pattern to Discovering Optimal Substructure

- Show that a solution to the problem consists in making a choice.
- You suppose that for a given problem, you are given the choice.
- Given this choice, you determine which subproblems ensue.
- You show that the solutions to the subproblems used within an optimal solution to the problem must themselves be optimal.
- If a solution to a problem did not use optimal solution to subproblem, this lead to a contraction that solution to subproblem is optimal.

Overview

- Matrix-Chain Multiplication
- Elements of Dynamic Programming
- Overlapping Subproblems
- Optimal Substructure

© P. Heeman 2017
Two Examples

Unweighted Shortest Path:
- Find a simple path from u to v consisting of the fewest edges
- Such paths will be simple, since removing a cycle from a path produces a path with fewer edges
- Can be solved by dynamic programming (homework)

Unweighted Longest Simple Path:
- Find a simple path from u to v consisting of the most edges
- Such paths will be simple, since removing a cycle from a path produces a path with fewer edges
- Solvable by dynamic programming?

How to Characterize the Subproblems

- Depend on the two factors:
  - Running time
    - How many choices we have in determining which subproblems to use in an optimal solution
    - How many subproblems an optimal solution to the original problem uses
  - Optimal substructure variables
    - Matrix multiplication: need to consider any split
    - Shortest path: sum to cut of last assignment
    - Rod-cutting: simpler to cut off initial piece
    - Necessary
- Keep subproblems as simple as possible, and expand if necessary

Two Examples

Unweighted Shortest Path:
- Find a simple path from u to v consisting of the fewest edges
- Such paths will be simple, since removing a cycle from a path produces a path with fewer edges
- Can be solved by dynamic programming (homework)

Unweighted Longest Simple Path:
- Find a simple path from u to v consisting of the most edges
- Such paths will be simple, since removing a cycle from a path produces a path with fewer edges
- Solvable by dynamic programming?
Difference between Problems

- **Shortest path**
  - Subproblems are independent: do not share resources
  - There will be no vertices in common in shortest path from $u$ to $v$ and $v$ to $w$

- **Longest path**
  - Subproblems compete for resources
  - Subproblems both might want to use a resource that can only be used once

Longest Path

- Does it have optimal substructure?
  - Let $d$ be a longest simple path from $a$ to $n$.
  - Let $d_d$ be a longest simple path from $a$ to $d$.
  - Let $d_n$ be a longest simple path from $d$ to $n$.
  - Are we guaranteed that $d$ is the longest simple path from $a$ to $n$?

\[ a \xrightarrow{d} d \xrightarrow{n} n \]
Overlapping Subproblems

- Total number of subproblems is typically polynomial in input size.
- Rather than always generating new subproblems, recursive algorithm for the problem solves same subproblems over and over.
- Must be a small set of subproblems.

Overview

- Elements of Dynamic Programming
- Matrix-Chain Multiplication
- Optimal Substructure
- Overlapping Subproblems