Dynamic Programming III: Knapsack and Matrix Chain Multiplication

Admin
→ PS 6 due Thursday (max 1 grace day)
→ Cookie challenge: 30% of code submissions complete by 11:59 pm tonight

🌟 → Final Exam Problem Solving Session—led by instructors
  • Thursday, 12 May, 4-7:00-9:00 pm, 26-100
  • Dinner food available

Review

• Rod cutting
  \[ r_n = \max_{1 \leq i \leq n} \{ p_i + r_{n-i} \} \]
  \[ \begin{aligned}
  r_0 &= 0 \\
  r_i &= 0, i < 0 \\
  r_i &= p_i, i \geq 0
  \end{aligned} \]

• Fibonacci
  \[ F_n = F_{n-1} + F_{n-2} \]
  \[ F_1 = F_2 = 1 \]

• Longest DAG path
  \[ \delta(s,v) = \max_{(u,v) \in E} \{ \delta(s,u) + w(u,v) \} \]
  \[ \delta(s,s) = 0 \]

• Shortest Path
  \[ \delta_k(s,v) = \min_{(u,v) \in E} \{ \delta_{k-1}(s,u) + w(u,v) \} \]
  \[ \delta_0(s,v) = \infty \]
  \[ \delta_k(s,s) = 0 \]

Today
Matrix Chain Multiplication & the Knapsack Problem
**Matrix Chain Multiplication**

**Given:** A matrix product to compute, \( A_1 \cdot A_2 \cdot A_3 \cdots \cdot A_n \) with matrices \( A_i \) of dimensions \( r_i \times c_i \).

**Find:** The set of associative parentheses leading to the lowest-cost computation of the matrix product.

**Notes:**
1. For compatibility of multiplication, \( c_i = r_{i+1} \).
2. Number of individual scalar multiplies = \( r_i \cdot c_i \cdot c_{i+1} \).  

\[
\begin{pmatrix}
A_{11} & A_{12} & \cdots & A_{1k} \\
A_{21} & \vdots & \ddots & \vdots \\
\vdots & \ddots & \ddots & \vdots \\
A_{r_1} & \vdots & \ddots & A_{r_1 k}
\end{pmatrix} \cdot 
\begin{pmatrix}
b_{11} \\
b_{21} \\
\vdots \\
b_{r_2 1}
\end{pmatrix} = 
\begin{pmatrix}
\sum_{k=1}^{c_1} A_{1k} b_{k1} \\
\vdots \\
\vdots \\
\sum_{k=1}^{c_1} A_{r_1 k} b_{r_2 k}
\end{pmatrix}
\]

**Example:**

\( (A_1 \cdot A_2) \cdot A_3 \) vs. \( A_1 \cdot (A_2 \cdot A_3) \)

- \( 1 \times 1000 \times 1000 \times 1 \times 1000 \)
- \( 1 \times 1000 \times 1000 \times 1 \times 1000 \)

Cost: \( 1 \times 1000 \times 1 + 1 \times 1000 = 2000 \) vs. \( 1000 \times 1 \times 1000 + 1 \times 1000 \times 1000 = 2,000,000 \)

1,000x difference in cost!!
Another example: Write out different parenthesizations

\[ A_1 \cdot A_2 \cdot A_3 \cdot A_4 \] becomes

\[
\begin{align*}
A_1 \cdot (A_2 \cdot (A_3 \cdot A_4)) \\
A_1 \cdot ((A_2 \cdot A_3) \cdot A_4) \\
(A_1 \cdot A_2) \cdot (A_3 \cdot A_4) \\
(A_1 \cdot (A_2 \cdot A_3)) \cdot A_4 \\
((A_1 \cdot A_2) \cdot A_3) \cdot A_4
\end{align*}
\]

Step 0: Enumeration over different parenthesizations while minimizing cost as evaluated by total number of scalar multiplies.

Step 1: Subproblem structure / recursion / base case / optimal substructure

\[ m_{1n} = \text{cost of optimal parenth. for } A_1 \cdots A_n \]

\[ m_{ij} = \min_{i \leq k \leq j} (m_{ik} + m_{kj} + \sum_{l=k}^{j} c_k c_l) \]

\[ m_{ii} = 0 \quad \text{[base case]} \]

An optimal parenthesization is constructed from optimal subparenthesizations. Assume not,

\[ A_1 \cdots A_n = (A_1 \cdots A_k) \cdot (A_{k+1} \cdots A_n) \] is optimal but \( A_1 \cdots A_k \) is not. Then \( A_1 \cdots A_n \) could be improved by improving \( A_1 \cdots A_k \). Contradiction.
Step 2: Subproblem dependence graph a DAG?
Yes, because each subproblem breaks into two subsubproblems that are smaller than (and non-overlapping subsets of) the subproblem.

Step 3: Memoize/Store for reuse: M_{ij} for 1 \leq i \leq j \leq n
\rightarrow two \ dimensional

Step 4: Write algorithm

Running Time Analysis
- # vertices in subproblem graph is
  \[
  \left( \begin{array}{c} \# M_{ij} \text{ 's with } 1 \leq i \leq j \leq n \end{array} \right) = n + \binom{n}{2} = \Theta(n^2)
  \]
- total # edges in subproblem graph is
  \[
  \sum_{1 \leq i < j \leq n} (j-i) \cdot 2 = \Theta(n^3)
  \]
  # of possible "cuts" in a chain of j-i+1 matrices
  \# of subproblems from a "cut"

Total RT = \Theta(n^3 + n^2) = \Theta(n^3)

Because cost of solving each subproblem is linear in the number of recursive calls it makes (# of outgoing edges in dependency graph)

\rightarrow Don't confuse cost of computing parenthesization with cost of matrix multiplication
Knapsack Problem

Imagine we wish to fill a knapsack (backpack) with goods of various value ($v_i$) and weight ($w_i$). Our goal is to produce a package of maximum value without exceeding a ceiling limit on the weight ($W$). Assume $W$ and all weights $w_i$ are positive integers.

**Knapsack with repetition** - can choose multiple copies of each item without limit

**Subproblem structure**

- Should we allow a lower weight ceiling?
- Or a smaller catalog of items?

Let $V(w) =$ value of optimal knapsack with weight ceiling $w$

If I have the optimal solution for $V(w)$, then I can remove one item $i$ to produce an optimal solution $V(w - w_i)$. Why?

Because I don't know items $i$, I need to enumerate all possibilities

$$V(w) = \max_{i : w_i \leq w} \left\{ V(w - w_i) + v_i \right\}$$

**Base case**

$V(0) = 0$ - Max of empty set $\equiv 0$
Bottom-up

\[ V(0) \leftarrow 0 \]
\[ \text{for } w \leftarrow 1 \text{ to } W \]
\[ V(w) \leftarrow \max_{i : w_i \leq w} \{ V(w - w_i) + v_i \} \]
return \( V(W) \)

Running Time: Fills 1D table of length \( W+1 \) from left to right. Each entry may need to try removing up to \( n \) items (the number of catalog items). Thus, \( O(nW) \).

If you make the subproblem dependency graph and weight each edge by \( v_i \), this knapsack variant amounts to computing the longest path in a DAG.

Imagine all items have weight that is a multiple of 5, and \( W \) is 100. The bottom-up approach would solve 101 subproblems \((0, 1, 2, \ldots, 100)\), but a memoized, top-down approach would only solve multiples of 5 — 21 subproblems \((0, 5, 10, 15, \ldots, 100)\) at most. This illustrates an important advantage of the top-down approach, which must be balanced with its disadvantageous overhead costs for recursion.
Knapsack without repetition - unique items

Our previous subproblem structure can't be used. Constructing $V(w)$ from $V(w-w_i)$ is only useful if item $i$ is absent from $V(w-w_i)$. This represents a violation of subproblem independence. We require to keep track of the items used up in our subproblem definition.

Let $V(w,j) = \text{maximum value with weight ceiling } w \text{ and catalog of items } 1,...,j \ (j \leq n)$

We seek $V(W,n)$

How should we index our enumeration?

$\rightarrow$ Item $j$ is either included or not in opt. solut.

$V(w,j) = \max \{ V(w-w_j,j-1) + v_j, \ V(w,j-1) \}$

The max of the empty set is 0.

So the subproblems that include item $j$ are expressed in terms of smaller subproblems that don't
Bottom-up

Initialize all \( V(0, j) \leftarrow 0 \) and all \( V(w, 0) \leftarrow 0 \)
for \( j \leftarrow 1 \) to \( n \)
    for \( w \leftarrow 1 \) to \( W \)
        if \( w_j > w \) : \( V(w, j) \leftarrow V(w, j-1) \)
        else : \( V(w, j) \leftarrow \max(V(w, j-1), V(w-w_j, j-1) + y_j) \)
    return \( V(W, n) \)

This time the approach fills out a 2D table, with each table entry taking constant time. So the running time is still \( O(nW) \).

Compare to enumerating \( 2^n \) packages and checking each for weight ceiling and maximum value.

\( O(2^n) \) is exponential, and so very poor.

\( O(nW) \) is better, but not great and here's why.
We judge asymptotic running time by size of input (in terms of bits or bytes of storage),
\[
nW = n2 \log_2 W = n2^{\log_2 W} \Rightarrow O(n2^b)
\]
number of bits required to store \( W \)

This is called "pseudo-polynomial" running time (polynomial in \# of items but exponential in storage of weights [and values]). It is better than exponential but not as good as polynomial.