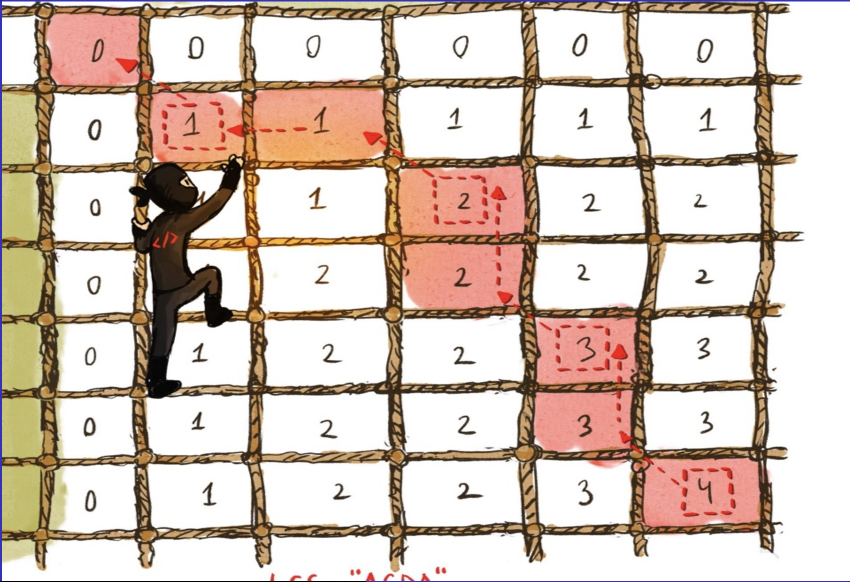
Dynamic Programming II



Multiplying a Sequence of Matrices

(This example is from Section 15.2 in CormenLRS' book.)
MULTIPLICATION OF n MATRICES Given as input a sequence of n matrices $(A_1 \times A_2 \times \cdots \times A_n)$. Minimize the number of operation in the computation $A_1 \times A_2 \times \cdots \times A_n$
Recall that Given matrices A_1, A_2 with $\dim(A_1) = p_0 \times p_1$ and $\dim(A_2) = p_1 \times p_2$, the basic algorithm to $A_1 \times A_2$ takes time at most $p_0 p_1 p_2$.

Example:

$$\begin{bmatrix} 2 & 3 \\ 3 & 4 \\ 4 & 5 \end{bmatrix} \times \begin{bmatrix} 2 & 3 & 4 \\ 3 & 4 & 5 \end{bmatrix} = \begin{bmatrix} 13 & 18 & 23 \\ 18 & 25 & 32 \\ 23 & 32 & 41 \end{bmatrix}$$

Multiplying a Sequence of Matrices

- Matrix multiplication is NOT commutative, so we can not permute the order of the matrices without changing the result.
- It is associative, so we can put parenthesis as we wish.
- How to multiply is equivalent to the problem of how to parenthesize.
- We want to find the way to put parenthesis so that the product requires the minimum total number of operations. And use it to compute the product.

Example Consider $A_1 \times A_2 \times A_3$, where dim $(A_1) = 10 \times 100$ dim $(A_2) = 100 \times 5$ and dim $(A_3) = 5 \times 50$.

- $((A_1 A_2) A_3)$ takes $(10 \times 100 \times 5) + (10 \times 5 \times 50) =$ *7500* operations,
- $(A_1 (A_2 A_3))$ takes $(100 \times 5 \times 50) + (10 \times 100 \times 50) =$ 75000 operations.

The order in which we make the computation of products of two matrices makes a big difference in the total computation's time.

How to parenthesize $(A_1 \times \ldots \times A_n)$?

- If $n = 1$ we do not need parenthesis.
- Otherwise, decide where to break the sequence $((A_1 \times \cdots \times A_k)(A_{k+1} \times \cdots \times A_n))$ for some k, $1 \leq k < n$.
- Then, combine any way to parenthesize $(A_1 \times \cdots \times A_k)$ with any way to parenthesize $(A_{k+1} \times \cdots \times A_n)$.

Using this structure, we can count the number of ways to parenthesize $(A_1 \times \cdots \times A_n)$ as well as to define a backtracking algorithm that goes over all those ways to parenthesize and eventually to a brute force recursive algorithm to solve the problem of computing efficiently the product.

How many ways to parenthesize $(A_1 \times \cdots \times A_n)$?

Let $P(n)$ be the number of ways to paranthesize $(A_1 \times \cdots \times A_n)$. Then,

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

with solution $P(n) = \frac{1}{n+1}\binom{2n}{n} = \Omega(4^n/n^{3/2})$

The Catalan numbers.

Brute force will take too long!

Structure of an optimal solution

- We want to compute $(A_1 \times \cdots \times A_n)$ efficiently.
- In an optimal solution the last matrix product must correspond to a break at some position k, $((A_1 \times \cdots \times A_k)(A_{k+1} \times \cdots \times A_n))$ Let $A_{i-j} = (A_i A_{i+1} \cdots A_j)$.
- The parenthesization of the subchains $(A_1 \times \cdots \times A_k)$ and $(A_{k+1} \times \cdots \times A_n)$ within the optimal parenthesization must be an optimal paranthesization of $(A_1 \times \cdots \times A_k)$, $(A_{k+1} \times \cdots \times A_n)$. So,

$$\begin{aligned}\text{cost}(A_1 \ldots A_n) = &\text{cost}(A_1 \ldots A_k) \\ &+ \text{cost}(A_{k+1} \ldots A_n) + p_0 p_k p_n.\end{aligned}$$

Structure of an optimal solution

- An optimal solution decomposes in optimal solutions of the same problem on subchains.
- Subproblems: compute the product $A_i \times A_{i+1} \times \cdots \times A_j$, for $1 \leq i \leq j \leq n$
- Let us call $B_i^j = A_i \times A_{i+1} \times \cdots \times A_j$.

Cost Recurrence

- Let $m[i,j]$ be the minimum cost of computing $B_i^j = (A_i \times \ldots \times A_j)$, for $1 \leq i \leq j \leq n$.
- $m[i,j]$ is defined by the value k, $i \leq k \leq j$ that minimizes

$$m[i,k] + m[k+1,j] + \text{ cost } (B_i^k, B_{k+1}^j).$$

- That is,

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \leq k < j}\{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\} & \text{otherwise} \end{cases}$$

Computing the cost of an optimal solution: Rec

Assume that vector P holds the values $(p_0, p_1, \ldots, p_n)$.

MCR(i, j)
if $i = j$ **then**
  return 0
$m[i, j] = \infty$
for $k = i$ **to** $j - 1$ **do**
  $q = \text{MCR}(i, k) + \text{MCR}(k + 1, j) + P[i - 1] * P[k] * P[j]$
  if $q < m[i, j]$ **then**
    $m[i, j] = q$
return $(m[i, j])$

Cost: $T(n) \geq 2\sum_{i=1}^{n-1} T(i) + n \sim \Omega(2^n)$.

Can we apply dynamic programming?

- We have an optimal recursive algorithm which takes exponential time.
- Subproblems?
  The subproblems are identified by the two inputs in the recursive call, the pair (i, j).
- How many subproblems?
  As $1 \leq i < j \leq n$, we have only $O(n^2)$ subproblems.
- We can use DP!

Dynamic programming: Memoization

MCP(P)
for all $1 \leq i < j \leq n$ **do**
  $m[i,j] = -1$
for $i = 1$ **to** n **do**
  $m[i,i] = 0$
MCR$(1,n)$
return $(m[1,n])$

MCR(i,j)
if $m[i,j]! = -1$ **then**
  return $(m[i,j])$
$m[i,j] = \infty$
for $k = i$ **to** $j-1$ **do**
  $q = \text{MCR}(i,k) + \text{MCR}(k+1,j) + P[i-1] * P[k] * P[j]$
  if $q < m[i,j]$ **then**
    $m[i,j] = q$
return $(m[i,j])$

$T(n) = \Theta(n^3)$ additional space $\Theta(n^2)$.

Dynamic programming: Tabulating

To compute the element $m[i,j]$ the base case is when $i = j$, we need to access $m[i,k]$ and $m[k+1,j]$. We can achieve that by filling the (half) table by diagonals.

MCP(P)
for $i = 1$ **to** n **do**
  $m[i,i] = 0$
for $d = 2$ **to** n **do**
  for $i = 1$ **to** $n - d + 1$ **do**
    $j = i + d - 1$
    $m[i,j] = \infty$
    for $k = i$ **to** $j - 1$ **do**
      $q =$
      $m[i,k] + m[k+1,j] + P[i-1] * P[k] * P[j]$
      if $q < m[i,j]$ **then**
        $m[i,j] = q$
return $(m[1,n])$

$T(n) = \Theta(n^3)$,
space $= \Theta(n^2)$.

Example.

We wish to compute $A_1 \times A_2 \times A_3 \times A_4$ with $P =< 3, 5, 3, 2, 4 >$

$i \setminus j$	1	2	3	4
1				
2				
3				
4				

Example.

We wish to compute $A_1 \times A_2 \times A_3 \times A_4$ with $P =< 3, 5, 3, 2, 4 >$

$i \setminus j$	1	2	3	4
1	0			
2		0		
3			0	
4				0

Example.

We wish to compute $A_1 \times A_2 \times A_3 \times A_4$ with $P =< 3, 5, 3, 2, 4 >$

$i \setminus j$	1	2	3	4
1	0	45		
2		0	30	
3			0	24
4				0

Example.

We wish to compute $A_1 \times A_2 \times A_3 \times A_4$ with $P =< 3, 5, 3, 2, 4 >$

$i \setminus j$	1	2	3	4
1	0	45	60	
2		0	30	70
3			0	24
4				0

Example.

We wish to compute $A_1 \times A_2 \times A_3 \times A_4$ with $P =< 3, 5, 3, 2, 4 >$

$i \setminus j$	1	2	3	4
1	0	45	60	84
2		0	30	70
3			0	24
4				0

Recording more information about the optimal solution

We have been working with the recurrence

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

To keep information about the optimal solution the algorithm keep additional information about the value of k that provides the optimal cost as

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

Dynamic programming: Memoization

MCP(P)
for all $1 \leq i < j \leq n$ **do**
  $m[i,j] = -1$
for $i = 1$ **to** n **do**
  $m[i,i] = 0$; $s[i,i] = i$;
MCR$(1,n)$
return m, s

MCR(i,j)
if $m[i,j]! = -1$ **then**
  return $(m[i,j])$
$m[i,j] = \infty$
for $k = i$ **to** $j - 1$ **do**
  $q = \text{MCR}(i,k) + \text{MCR}(k+1,j) + P[i-1] * P[k] * P[j]$
  if $q < m[i,j]$ **then**
    $m[i,j] = q$; $s[i,j] = k$;
return $(m[i,j])$

Dynamic programming: Tabulating

MCP(P)
for $i = 1$ **to** n **do**
  $m[i,i] = 0$; $s[i,i] = 0$;
for $d = 2$ **to** n **do**
  for $i = 1$ **to** $n - d + 1$ **do**
    $j = i + d - 1$
    $m[i,j] = \infty$
    for $k = i$ **to** $j - 1$ **do**
      $q =$
      $m[i,k] + m[k+1,j] + P[i-1] * P[k] * P[j]$
      if $q < m[i,j]$ **then**
        $m[i,j] = q$; $s[i,j] = k$;
return m, s.

Example.

We wish to compute $A_1 \times A_2 \times A_3 \times A_4$ with $P = (3, 5, 3, 2, 4)$

$i \setminus j$	1	2	3	4
1				
2				
3				
4				

Example.

We wish to compute $A_1 \times A_2 \times A_3 \times A_4$ with $P = (3, 5, 3, 2, 4)$

$i \setminus j$	1	2	3	4
1	0 1			
2		0 2		
3			0 3	
4				0 4

Example.

We wish to compute $A_1 \times A_2 \times A_3 \times A_4$ with $P = (3, 5, 3, 2, 4)$

$i \setminus j$	1	2	3	4
1	0 1	45 1		
2		0 2	30 2	
3			0 3	24 3
4				0 4

Example.

We wish to compute $A_1 \times A_2 \times A_3 \times A_4$ with $P = (3, 5, 3, 2, 4)$

$i \setminus j$	1	2	3	4
1	0 1	45 1	60 1	
2		0 2	30 2	70 3
3			0 3	24 3
4				0 4

Example.

We wish to compute $A_1 \times A_2 \times A_3 \times A_4$ with $P = (3, 5, 3, 2, 4)$

$i \setminus j$	1	2	3	4
1	0 1	45 1	60 1	84 3
2		0 2	30 2	70 3
3			0 3	24 3
4				0 4

Computing optimally the product

- $s[i, j]$ contains the value of k that decomposes optimally the product as product of two submatrices, i.e.,

$$A_i \times \cdots \times A_j = (A_i \times \cdots \times A_{s[i,j]})(A_{s[i,j]+1} \times \cdots \times A_j).$$

- Therefore,

$$A_1 \times \cdots \times A_n = (A_1 \times \cdots \times A_{s[1,n]})(A_{s[1,n]+1} \times \cdots \times A_n).$$

- We can design a recursive algorithm to perform the product in an optimal way.

The product algorithm

The input is the sequence of matrices $A = A_1, \ldots, A_n$ and the table s computed before.

Product(A, s, i, j)
if $i = j$ **then**
  return (A_i)
$X =$**Product**$(A, s, i, s[i, j])$
$Y =$**Product**$(A, s, s[i, j] + 1, j)$
return $(X \times Y)$

The total number operations required to compute the product is $m[1, n]$ and the cost of the complete algorithm is $T(n) = O(n^3 + m[1, n])$

Example.

We wish to compute $A_1 \times A_2 \times A_3 \times A_4$ with $P = (3, 5, 3, 2, 4)$

$i \setminus j$	1	2	3	4
1	0 1	45 1	60 1	84 3
2		0 2	30 2	70 3
3			0 3	24 3
4				0 4

The optimal way to minimize the number of operations is

$$(((A_1) \times (A_2 \times A_3)) \times (A_4))$$

Multiplying matrices

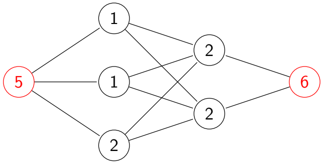
- In order to compute s, we only need the dimensions of the matrices.
- What if we use Strassen algorithm to compute a two matrices product instead of the naive algorithm?

Dynamic Programming in Trees

- Trees are nice graphs easily adapted to recursion.
- Once you root the tree each node can be seen as the root of a subtree .
- We can use Dynamic Programming to give polynomial solutions to "difficult" graph problems when the input is restricted to be a tree, or to have a treee-like structure (small treewidth).
- In this case instead of having a global table, each node in the tree keeps additional information about the associated subproblem.

The Maximum Weight Independent Set (MWIS)

Given as input $G = (V, E)$, together with a weight $w : V \rightarrow \mathbb{R}$. Find the heaviest $S \subseteq V$ such that no two vertices in S are connected in G.

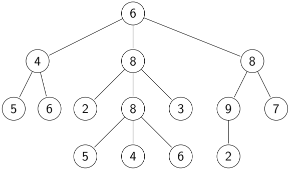


For general graphs, the problem is hard, even for the case in which all vertex have weight 1, i.e. Maximum Independent Set is NP-complete.

Maximum Weight Independent Set on Trees

Given a tree $T = (V, E)$ choose a $r \in V$ and root it from r

i.e. Given a rooted tree $T = (V, E, r)$ and weights $w : V \to \mathbb{R}$, find the independent set with maximum weight.



Notation:

- For $v \in V$, let T_v be the subtree rooted at v. $T = T_r$.
- Given $v \in V$ let $C(v)$ be the set of children of v, and $G(v)$ be the set of grandchildren of v.

Characterization of the optimal solution

Key observation: An IS can't contain vertices which are father-son.

Let S be an optimal solution.

- If $r \in S$: then $C(r) \not\subseteq S_r$. So $S - \{r\}$ contains an optimum solution for each T_v, with $v \in G(r)$.
- If $r \notin S$: S contains an optimum solution for each T_u, with $u \in C(r)$.

Recursive definition of the optimal solution

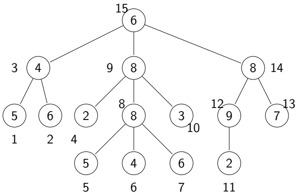
- To implement DP, tor every node v, we add one value, $v.M$: the value of the optimal solution for T_v
  Following the recursive structure of the solution we have the following recurrence

$$v.M = \begin{cases} w(v) & v \text{ a leaf,} \\ \max\{\sum_{u \in C(v)} u.M, w(v) + \sum_{u \in G(v)} u.M\} & \text{otherwise.} \end{cases}$$

- Notice that for any $v \in T$: we have to compute $\sum_{u \in C(v)} u.M$ and for this we must access to the children of its children
- To avoid this we add another value to the node $v.M'$: the sum of the values of the optimal solutions of their children, i.e., $\sum_{u \in C(v)} u.M$.

Post-order traversal of a rooted tree

To perform the computation, we can follow a DFS, post-order, traversal of the nodes in the tree, computing the additional values at each node.



DP Algorithm to compute the optimal weight

Let $v_1, \ldots, v_n = r$ be the post-order traversal of T_r

WIS T_r
Let $v_1, \ldots, v_n = r$ the post-order traversal of T_r
for $i = 1$ **to** n **do**
  if v_i is a leaf **then**
    $v_i.M = w[v_i], v_i.M' = 0$
  else
    $v_i.M' = \sum_{u \in C(v)} u.M$
    $aux = \sum_{u \in C(v)} u.M'$
    $v_i.M = \max\{aux + w[v_i], v_i.M'\}$
return r.M

Complexity: space $= O(n)$, time $= O(n)$

Top-down traversal to obtain an optimal IS

RWIS(v)
if v is a leaf **then**
  return $(\{v\})$
if $v_i.M = v_i.M' + w[v_i]$ **then**
  $S = S \cup \{v_i\}$
  for $w \in G(v)$ **do**
    $S = S \cup$ **RWIS**(w)
else
  for $w \in N(v)$ **do**
    $S = S \cup$ **RWIS**(w)
return S

RWIS(r)
provides an optimal solution in time $O(n)$

Total cost $O(n)$ and additional space $O(n)$