Algorithm Design & Analysis (CS3383)[1]

Unit 4 : Dynamic Programming-More Examples

Rasoul Shahsavarifar

March 18, 2019

[1] Thanks to Dr. Ptricia Evans and Dr. David Bremner at UNB, Dr. Erik Demaine at MIT for sharing the teaching stuffs

Knapsack Problem

There is a set of n items, each with its own value v_i and weight w_i. We want to be able to take items with as much value as possible.

However, we also have a weight limit W – we cannot carry items that in total weigh more than W.

Additionally, items are indivisible, in that we must take either an entire item or leave it behind. In the 0/1 model, an item can be taken only once.

Knapsack Problem

Formally, given the values v_i, weights w_i, and weight limit W, we want to determine subset S for:

$$Knapsack(n) = \max_{\text{subset } S} \sum_{i \in S} v_i$$

subject to

$$\sum_{i \in S} w_i \leq W$$

Knapsack Problem

If we were allowed to take fractions of items, we could solve this problem using a greedy approach: take the item with the highest remaining $\frac{v_i}{w_i}$ ratio (or as much of it as we can).

Not being able to take fractions breaks this strategy, since taking a heavy item – albeit of high value – can block us from using the remaining weight allowance well.

Knapsack Problem

If we were allowed to take fractions of items, we could solve this problem using a greedy approach: take the item with the highest remaining $\frac{v_i}{w_i}$ ratio (or as much of it as we can).

Not being able to take fractions breaks this strategy, since taking a heavy item – albeit of high value – can block us from using the remaining weight allowance well.

Counterexample:

i	v_i	w_i
1	10	5
2	7	4
3	5	3

with the weight limit $W = 7$.

Recurrence for Knapsack

As for other DP approaches, we need to build a recurrence for the optimization problem (maximizing the value taken).

We consider the last item. What are the options that need to be evaluated for it?

Recurrence for Knapsack

As for other DP approaches, we need to build a recurrence for the optimization problem (maximizing the value taken).

We consider the last item. What are the options that need to be evaluated for it?

As with many other subset-selecting problems, we either take the last item or we don't.

Recurrence for Knapsack

As for other DP approaches, we need to build a recurrence for the optimization problem (maximizing the value taken).

We consider the last item. What are the options that need to be evaluated for it?

As with many other subset-selecting problems, we either take the last item or we don't.

We also need to consider what information constrains how the result can be used, and ensure that this is part of the parameters of the recurrence.

Recurrence for Knapsack

As for other DP approaches, we need to build a recurrence for the optimization problem (maximizing the value taken).

We consider the last item. What are the options that need to be evaluated for it?

As with many other subset-selecting problems, we either take the last item or we don't.

We also need to consider what information constrains how the result can be used, and ensure that this is part of the parameters of the recurrence.

So, our recurrence needs to have both the number of items as an argument, and also the weight limit.

Recurrence for Knapsack

So, let $OPT(i, k)$ $=$ maximum value of a subset of items $1 \ldots i$ whose total weight is $\leq k$.

Recurrence for Knapsack

So, let $OPT(i, k)$ = maximum value of a subset of items $1 \ldots i$ whose total weight is $\leq k$.

Then:

$$OPT(i,k) = \max \begin{cases} OPT(i-1,k) & \\ v_i + OPT(i-1, k-w_i) & \text{if } k \geq w_i \end{cases}$$

Note that the second case is conditional, so we choose the maximum of the two cases if the condition is met, and take the first case if it is not.

Base Cases:

Recurrence for Knapsack

So, let $OPT(i, k)$ = maximum value of a subset of items $1 \ldots i$ whose total weight is $\leq k$.

Then:

$$OPT(i,k) = \max \begin{cases} OPT(i-1,k) \\ v_i + OPT(i-1, k-w_i) & \text{if } k \geq w_i \end{cases}$$

Note that the second case is conditional, so we choose the maximum of the two cases if the condition is met, and take the first case if it is not.

Base Cases:

$$OPT(i,k) = 0 \quad \text{if } i = 0$$

$$OPT(i,k) = 0 \quad \text{if } k = 0$$

Algorithm for Knapsack

We turn this recurrence into a Dynamic Programming Algorithm by computing subproblem results from small to large, and storing them in a table.

```
Input  : values $v[1..n]$, weights $w[1..n]$, limit $W$
Output: maximum knapsack value

Let $M[0..n][0..W]$ be an empty table
for $k = 0$ to $W$ do
    $M[0][k] \leftarrow 0$
for $i = 1$ to $n$ do
    $M[i][0] \leftarrow 0$
    for $k = 1$ to $W$ do
        $M[i][k] \leftarrow M[i-1][k]$
        if $k \geq w[i]$ and $M[i][k] < M[i-1][k-w[i]] + v[i]$ then
            $M[i][k] \leftarrow M[i-1][k-w[i]] + v[i]$
return $M[n][W]$
```

Algorithm for Knapsack

Running Time:
The initial loop iterates $W + 1$ times, by itself, and is followed by a nested pair of loops.
The outer loop of the pair iterates n times. For each outer iteration, the inner loop iterates W times. The rest of the algorithm takes constant time, so the algorithm as a whole runs in $\Theta(nW)$ time.

Algorithm for Knapsack

Running Time:
The initial loop iterates $W + 1$ times, by itself, and is followed by a nested pair of loops.
The outer loop of the pair iterates n times. For each outer iteration, the inner loop iterates W times. The rest of the algorithm takes constant time, so the algorithm as a whole runs in $\Theta(nW)$ time.

Note: this is not a polynomial running time, since W is an input value, not an input *size*.

Since it runs in time polynomial in the input values, this is a *pseudopolynomial* time algorithm.

This distinction is important, since 0/1 KNAPSACK is a well-known NP-hard problem, and no strictly polynomial time algorithm for it is known.

Algorithm for Knapsack

To extract the subset S, we need to trace back from the end result.

Algorithm 1: Trace(i,k)

```
Input  : row $i$, column $k$
Output: subset $S$

if $i = 0$ or $k = 0$ then
    $S \leftarrow \emptyset$
else
    if $k \geq w[i]$ and $M[i][k] < M[i-1][k-w[i]] + v[i]$ then
        $S \leftarrow Trace(i-1, k-w[i]) \cup \{i\}$
    else
        $S \leftarrow Trace(i-1, k)$
return $S$
```

Multiplying Many Matrices

Multiplying two matrices X and Y, where X is $n \times p$ and Y is $p \times m$, takes $n \cdot p \cdot m$ calculations.

If we have three matrices to multiply, now with a $m \times q$ matrix Z, then we can use that matrix multiplication is associative:

$$(X \times Y) \times Z \;=\; X \times (Y \times Z)$$

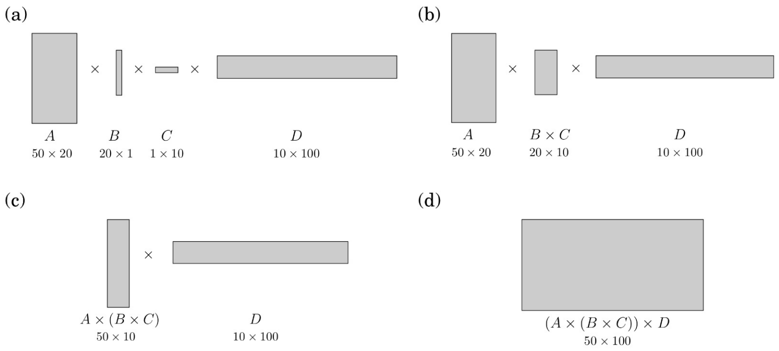
The results are equal, but the number of operations needed can differ.

- $(X \times Y) \times Z$ takes $n \cdot p \cdot m \;+\; n \cdot m \cdot q$ operations
- $X \times (Y \times Z)$ takes $p \cdot m \cdot q \;+\; n \cdot p \cdot q$ operations

Which one is faster depends on the relative values of the sizes.

Multiplying Many Matrices

Figure 6.6 $A \times B \times C \times D = (A \times (B \times C)) \times D$.



Multiplying Many Matrices

Parenthesization	Cost computation	Cost
$A \times ((B \times C) \times D)$	$20 \cdot 1 \cdot 10 + 20 \cdot 10 \cdot 100 + 50 \cdot 20 \cdot 100$	$120,200$
$(A \times (B \times C)) \times D$	$20 \cdot 1 \cdot 10 + 50 \cdot 20 \cdot 10 + 50 \cdot 10 \cdot 100$	$60,200$
$(A \times B) \times (C \times D)$	$50 \cdot 20 \cdot 1 + 1 \cdot 10 \cdot 100 + 50 \cdot 1 \cdot 100$	$7,000$

As a greedy heuristic, we could try eliminating the dimensions in nonincreasing order of size (eliminating the largest dimensions first). This approach however does not work in all situations.

Instead, we will try to build a recurrence for the minimum number of operations needed, and use dynamic programming.

Matrix Chain Multiplication

The MATRIX CHAIN MULTIPLICATION problem is an example of a parenthesization problem, which can be solved through dynamic programming.

We want to multiply

$$A_1 \times A_2 \times \cdots \times A_n$$

where the matrix dimensions are

$$(m_0 \times m_1), (m_1 \times m_2), \ldots, (m_{n-1} \times m_n)$$

We want to find the best way of decomposing the chain of matrix multiplications into pairs of multiplications.

Matrix Chain Multiplication

Where $1 \leq i \leq j \leq n$, define

$$C(i,j) \;=\; \text{minimum cost of multiplying } A_i \times \cdots \times A_j$$

Note that we do not fix one end, as we have for some of the sequence problems; any segment of the matrices could be a level of the multiplication.

Matrix Chain Multiplication

Where $1 \leq i \leq j \leq n$, define

$$C(i,j) \;=\; \text{minimum cost of multiplying } A_i \times \cdots \times A_j$$

Note that we do not fix one end, as we have for some of the sequence problems; any segment of the matrices could be a level of the multiplication.

We then consider what the options are for breaking this multiplication down. Any position in the sequence of multiplications could be the “top-level” multiplication.

Matrix Chain Multiplication

Where $1 \leq i \leq j \leq n$, define

$$C(i, j) \;=\; \text{minimum cost of multiplying } A_i \times \cdots \times A_j$$

Note that we do not fix one end, as we have for some of the sequence problems; any segment of the matrices could be a level of the multiplication.

We then consider what the options are for breaking this multiplication down. Any position in the sequence of multiplications could be the “top-level” multiplication.

$$C(i, j) \;=\; \min_{i \leq k < j} \{C(i, k) + C(k + 1, j) + m_{i-1} \cdot m_k \cdot m_j\}$$

Matrix Chain Multiplication

So, we have the recurrence

$$C(i,j) = \min_{i \le k < j} \{C(i,k) + C(k+1,j) + m_{i-1} \cdot m_k \cdot m_j\}$$

What computation order should we use, and what are the base cases?

Matrix Chain Multiplication

So, we have the recurrence

$$C(i,j) = \min_{i \le k < j} \{C(i,k) + C(k+1,j) + m_{i-1} \cdot m_k \cdot m_j\}$$

What computation order should we use, and what are the base cases?

Computation Order: small to large. For this recurrence, the size would depend on the chain length $j - i$.

Base cases: length 0 (single matrices), requiring no operations. $C(i,i) = 0$.

Matrix Chain Multiplication Algorithm

We turn the recurrence into an algorithm by filling in the upper right of a table, diagonal by diagonal.

```
Input   : dimensions $d[0..n]$
Output: minimum number of multiplications

Let $M[1..n][1..n]$ be an empty table
for $i = 1$ to $n$ do
    $M[i][i] \leftarrow 0$
for $s = 1$ to $n - 1$ do
    for $i = 1$ to $n - s$ do
        $j \leftarrow i + s$
        $M[i][j] \leftarrow M[i][i] + M[i + 1][j] + d[i - 1] \cdot d[i] \cdot d[j]$
        for $k = i + 1$ to $j - 1$ do
            if $M[i][j] > M[i][k] + M[k + 1][j] + d[i - 1] \cdot d[k] \cdot d[j]$ then
                $M[i][j] \leftarrow M[i][k] + M[k + 1][j] + d[i - 1] \cdot d[k] \cdot d[j]$
return $M[1][n]$
```

Matrix Chain Multiplication Algorithm

Running Time:

Upper Bound:
The initial loop iterates n times. Of the nested loops, the outer loop iterates $n-1$ times. For each outer iteration, the middle loop iterates at most $n-1$ times. For each middle iteration, the innermost loop iterates at most $n-2$ times. The rest of the algorithm takes constant time, so the algorithm runs in $O((n-1)^2(n-2)) = O(n^3)$ time.

Lower Bound:
Consider the algorithm's iterations such that $\frac{n}{3} \leq s \leq \frac{2n}{3}$. Under these restrictions, there are at least $\lfloor \frac{n}{3} \rfloor$ iterations of the outer loop. For each of the outer iterations, there are at least $\lfloor \frac{n}{3} \rfloor$ iterations of the middle loop, since it iterates from 1 to $n-s$. For each middle iteration, there are at least $s-1 \geq \lfloor \frac{n}{3} \rfloor - 1$ iterations of the innermost loop. Thus the algorithm runs in $\Omega((\lfloor \frac{n}{3} \rfloor)^2 \cdot (\lfloor \frac{n}{3} \rfloor - 1)) \; = \; \Omega(n^3)$ time.

Since the algorithm runs in $O(n^3)$ and $\Omega(n^3)$ time, it runs in $\Theta(n^3)$ time.

Finding the Parenthesization

The preceding algorithm finds the minimum number of multiplications needed. If you want to carry out this multiplication, you need to trace back to find how the chain of multiplications needs to be broken down.

The structure of the traceback is the same as the recurrence. Note, though, that a case of this recurrence is based on two previous results, not one, so the traceback needs to recurse twice each time.

Additionally, because the optimal case for each subproblem needed a loop to calculate, it’s better to save the information about what case gives the optimal result, rather than retest the cases for the minimum.

Finding the Parenthesization

```
Input   : dimensions $d[0..n]$
Output: optimal parenthesization

Let $M[1..n][1..n]$ and $T[1..n][1..n]$ be empty tables
for $i = 1$ to $n$ do
    $M[i][i] \leftarrow 0$
for $s = 1$ to $n - 1$ do
    for $i = 1$ to $n - s$ do
        $j \leftarrow i + s$
        $M[i][j] \leftarrow M[i][i] + M[i+1][j] + d[i-1] \cdot d[i] \cdot d[j]$
        $T[i][j] \leftarrow i$
        for $k = i + 1$ to $j - 1$ do
            if $M[i][j] > M[i][k] + M[k+1][j] + d[i-1] \cdot d[k] \cdot d[j]$ then
                $M[i][j] \leftarrow M[i][k] + M[k+1][j] + d[i-1] \cdot d[k] \cdot d[j]$
                $T[i][j] \leftarrow k$
return Trace($T$, 1, $n$)
```

Finding the Parenthesization

Note that the cases for the traceback mirror the cases considered for the recurrence and table calculation (they have the same indices).

Algorithm 2: Trace(T,i,j)

Input : table T, row i, column j
Output: parenthesization sequence S

```
if $i = j$ then
    $S \leftarrow$ “$A_i$“
else
    $k \leftarrow T[i][j]$
    $S \leftarrow$ “(“ $+$ $Trace(i, k)$ $+$ “ $\times$ “ $+$ $Trace(k + 1, j)$ $+$ “)“
return $S$
```