Dynamic programming

Dynamic programming is a technique for efficiently computing recurrences by storing partial results and re-using them when needed.

We trade space for time, avoiding to repeat the computation of a subproblem.

Dynamic programming is best underestood by looking at a bunch of different examples.

Fibonacci numbers

Fibonacci recurrence: $F_n = F_{n-1} + F_{n-2}$ with $F_0 = 0, F_1 = 1$

```
function Fibonacci(n)
    if n = 0
      then
         return (0)
      elseif
         n = 1
         then return (1)
         else  return (Fibonacci(n − 1) + Fibonacci(n − 2))
      end if

    end
```

As $F_{n+1}/F_n \sim (1+\sqrt{5})/2 \sim 1.61803$ then $F_n > 1.6^n$, and to compute F_n we need 1.6^n recursive calls.

Fibonacci con tabla

function PD-Fibonacci(n)
    var
        F : **array** $[0 \,..\, n]$ **of** integer
        i : integer
    end var
    $F[0] := 0; F[1] := 1$
    for $i := 2$ **to** n **do**
        $F[i] := F[i-1] + F[i-2]$
    end for
end

To compute F_6:

$$0 \quad 1 \quad 1 \quad 2 \quad 3 \quad 5 \quad 7 \quad 9 \quad 16$$

To get F_n need $O(n)$ iterations.

Guideline to implement Dynamic Programming

1. Characterize the recursive structure of an optimal solution,
2. define recursively the value of an optimal solution,
3. compute, bottom-up, the cost of a solution,
4. construct an optimal solution.

Multiplying a Sequence of Matrices

We wish to multiply a long sequence of matrices

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

with the minimum number of operations.

Give 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 $p_0 \times p_1 \times p_2$:

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

Recall that matrix multiplication is NOT commutative, so we can not permute the order of the matrices without changing the result,

but it is associative, so we can parenthesise as we wish.

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

$(A_1 A_2)A_3$ needs $(10 \times 100 \times 5) + (10 \times 5 \times 50) = 7500$ operations,

$A_1(A_2 A_3)$ needs $(100 \times 5 \times 50) + (10 \times 100 \times 50) = 75000$ operations.

The order makes a big difference in real computation's time

The problem of given $A_1, \dots A_n$ with $\dim(A_i) = p_{i-1} \times p_i$,

decide how to multiply them to minimize the number of operations

is equivalent to

the problem of deciding how to put a correct set of parenthesis the sequence $A_1 1, \dots A_n$.

How many ways to put parenthesis $A_1, \dots A_n$?

$A_1 \times A_2 \times A_3 \times A_4$:

$(A_1(A_2(A_3A_4))), ((A_1A_2)(A_3A_4)), (((A_1(A_2A_3))A_4), (A_1((A_2A_3)A_4))), (((A_1A_2)A_3)A_4))$

Let $P(m)$ be the number of ways to put parenthesis correctly in $A_1, \dots A_n$. 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!

Therefore, brute force will take too long!

But we got a recursive definition. Let's try a recursive solution.

Characterize the structure of an optimal solution

Notation: $A_{i-j} = (A_i \times A_{i+1} \times \cdots \times A_j)$

Optimal substructure: The optimal way to put parenthesis on the subchain $(A_1 \cdots A_k)$ with the optimal way to put parenthesis on $A_{k+1} \cdots A_n$ must be an optimal way to put paranthesis on $A_1 \cdots A_n$ for some k.

Notice, that

$$\forall k, 1 \leq k \leq n, \text{ cost } (A_{1-k}) + \text{ cost } (A_{k+1-n}) + p_0 p_k p_n.$$

gives the cost associated to this decomposition.

We only have to take the minimum over all k to get a recursive solution.

Recursive solution

Let $m(i,j)$ be the minimum nomber of operations needed to compute $A_{i-j} = A_i \times \ldots \times A_j$.

$m(i,j)$ is given by choosing the value k, $i \leq k \leq j$ that minimizes

$$m(i,k) + m(k+1,j) + \text{ cost } (A_{1-k} \times A_{k+1-n}).$$

That is,

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

Computing the optimal costs

Straightforward recursive implementation of the previous recurrence:

As $\dim(A_i) = p_{i-1}p_i$, the input is given by $P =< p_0, p_1, \ldots, p_n >$,

> **function** $\mathsf{MSMR}(P, i, j)$: integer
>   **if** $i = j$ **then return** (0) **end if**;
> $m := \infty$;
>   **for** $k := i$ **to** $j - 1$ **do**
> $q := \mathsf{MSMR}(P, i, k) + \mathsf{MSMR}(P, k + 1, j) + p[i - 1]p[k]p[j]$
>   **if** $q < m$ **then** $m := q$ **end if**
>   **end for**
>   **return** (m)
> **end**

The time complexity of the previous recursive algorithm is given by

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

An exponential function.

How many subproblems?

there are only $O(n^2)$ A_{i-j}!

We are repeating the computation of too many identical subproblems

Use dynamic programming to compute the optimal cost by a bottom-up approach.

We wil use an auxiliary table $m[1 \ldots m, 1 \ldots m]$ for storing $m[i, j]$,

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

We can fill the array starting from the diagonal.

function algorithm MCP(P)

var

m : **array** $[1 \,..\, n] of$ integer

i, j, l : integer

end var

for $i := 1$ **to** n **do**

$m[i,i] := 0;$

end for

for $l := 2$ **to** n **do**

for $i := 1$ **to** $n - l + 1$ **do**

$j := i + l - 1; m[i,j] := \infty;$

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** $m[i,j] := q$ **end if**

end for

end for

end for

return (m)

end

This algorithm has time complexity $T(n) = \Theta(n^3)$, and uses space $\Theta(n^2)$.

Constructing an optimal solution

We have the optimal number of scalar multiplications to multiply te n matrices. Now we want to construct an optimal solution.

We record which k achieved the optimal cost in computing $m[i,j]$ in an auxiliary table $s[1 \dots m, 1 \dots m]$.

From the information in s we can recover the optimal way to multiply:

$$A_i \times \cdots \times A_j = (A_i \times \cdots \times A_k)(A_{k+1} \times \cdots \times A_j).$$

The value $s[i, s[i,j]]$ determines the k to get $A_{i-s[i,j]}$ and $s[s[i,j]+1, j]$ determines the k to get $A_{s[i,j]+1-j}$.

The dynamic programming algorithm can be adapted easily to compute also s

```
function algorithm MCP(P)
    var
        m : array [1 .. n] of integer; s : array [1 .. n, 1 .. n] of integer
        i, j, l : integer
    end var
    for i := 1 to n do
        m[i, i] := 0;
    end for
    for l := 2 to n do
        for i := 1 to n − l + 1 do
            j := i + l − 1;
            m[i, j] := ∞;
            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 end if
            end for
        end for
    end for
    return (m)
end
```

Therefore after computing table s we can multiply the matrices in an optimal way:

$A_{1-n} = A_{1-s[1,n]} A_{s[1,n]+1-n}$.

function algorithm $Multiplication(A, s, i, j)$

if $j > 1$

then

$X :=$ **algorithm** $Multiplication(A, s, i, s[i, j]);$

$Y :=$ **algorithm** $Multiplication(A, s, s[i, j] + 1, j);$

return $(X \times Y)$

else

return (A_i)

end if

end

0-1 Knapsack

We have a set I of n items, item i has weight w_i and worth v_i. We can carry at most weight W in our knapsack. Considering that we can NOT take fractions of items, what items should we carry to maximize the profit?

Let $v[i, j]$ be the maximum value we can get from objects $\{1, 2, \ldots, i\}$ and taking a maximum weight of $0 \leq j \leq W$.

We wish to compute $v[n, W]$.

To compute $v[i, j]$ we have two possibilities: The i-th element is or is not part of the solution.

This gives the recurrence,

$$v[i, j] = \begin{cases} v[i-1, j-w_i] + v_i & \text{if the } i\text{-th element is part of the solution} \\ v[i-1, j] & \text{otherwise} \end{cases}$$

Define a table $v[1 \dots n, 0 \dots W]$,

Initial condition: $\forall j, v[0, j] = 0$

To compute $v[i, j]$ must look to $v[i-1, j]$ and to $v[i-1, j-w_i]$.

$v[n, W]$ will indicate the profit.

Example.

Let $I = \{1, 2, 3, 4, 5\}$ with $v(1) = 1; v(2) = 6; v(3) = 18; v(4) = 22; v(5) = 28$, $w(1) = 1; w(2) = 2; w(3) = 5; w(4) = 6; w(5) = 7$ and $W = 11$.

	0	1	2	3	4	5	6	7	8	9	10	11
1	0	1	1	1	1	1	1	1	1	1	1	1
2	0	1	6	7	7	7	7	7	7	7	7	7
3	0	1	6	7	7	18	19	24	25	25	25	25
4	0	1	6	7	7	18	22	24	28	29	29	40
5	0	1	6	7	7	18	22	28	29	34	35	40

$$v[3,5] = \max\{v[2,5], v[2,0] + v(3)\} = \max\{7, 0 + 18\} = 18$$

The time complexity is $O(nW)$.

Notice, that at each computation of $v[i, j]$ we just need to store two rows of the table, therefore the space complexity is $2W$

Question

As you already know 0-1 Knapsack is NP-hard.

But the previous algorithm has time complexity $O(nW)$. Therefore ¶=NP!

Is something wrong?

Exercise

Modify the 0-1 Knapsack algorithm so that in addition to computing the optimal cost it computes an optimal solution.

Travelling Sales Person

Given n cities and the distances d_{ij} between any two of them, we wish to find the shortest tour going through all cities and back to the starting sity. Usually the TSP is given as a $G = (V, D)$ where $V = \{1, 2, \ldots, n\}$ is the set of cities, and D is the adjacency distance matrix, with $\forall i, j \in V, i \neq j, d_{i,j} > 0$, the probem is to find the tour with minimal distance weight, that starting in 1 goes through all n cities and returns to 1.

The TSP is a well known and difficult problem, that can be solved in $O(n!) \sim O(n^n e^{-n})$ steps.

Characterization of the optimal solution

Given $S \subseteq V$ with $1 \in S$ and given $j \neq 1, j \in S$, let $C(S, j)$ be the shortest path that starting at 1, visits all nodes in S and ends at j.

Notice:

- If $|S| = 2$, then $C(S, k) = d_{1,k}$ for $k = 2, 3, \ldots, n$
- If $|S| > 2$, then $C(S, k) =$ the optimal tour from 1 to m, $+d_{m,k}$,

  $\exists m \in S - \{k\}$

Recursive definition of the optimal solution

$$C(S, k) = \begin{cases} d_{1,m} & \text{if } S = \{1, k\} \\ \min_{m \neq k, m \in S}[C(S - \{k\}, m) + d(m, k)] & \text{otherwise} \end{cases}$$

The optimal solution

function algorithm $TSP(G, n)$

  for $k := 2$ **to** n **do**

    $C(\{i, k\}, k) := d_{1,k}$

  end for

  for $s = 3$ **to** n **do**

    for all $S \subseteq \{1, 2, \ldots, n\} || S|| = s$ **do**

      for all $k \in S$ **do**

        $\{C(S, k) = \min_{m \neq k, m \in S}[C(S - \{k\}, m) + d_{m,k}]\}$

        $opt := \min_{k \neq 1}[C(\{1, 2, 3, \ldots, n\}, k) + d_{1,k}$

      end for

    end for

  end for;

  return (opt)

end

Complexity:

Time: $(n-1)\sum_{k=1}^{n-3}\binom{n-2}{k} + 2(n-1) \sim O(n^2 2^n) << O(n!)$

Space: $\sum_{k=1}^{n-1} k\binom{n-1}{k} = (n-1)2^{n-2} \sim O(n2^n)$

Dynamic Programming in Trees

Trees are nice structures to bound the number of subproblems.

Given $T = (N, A)$ with $|N| = n$, recall that there are n subtrees in T.

Therefore, when considering problems defined on trees, it is easy to bound the number of subproblems

Example: The Maximum Independent Set (MIS)

Given $G = (V, E)$ the Maximum Independent Set is a set $I \subseteq V$ such that no two vertices in I are connected in G, and I is as large as possible.

Difficult problem for general graphs

Characterization of the optimal solution

Given a tree $T = (N, A)$ as instance for the MIS, assume T is rooted. Then each node defines a subtree.

For $j \in N$, the MIS (j):

1. it is j plus the union of the MIS of its grandsons,
2. It does not include j, and it is the union of the MIS of its sons.

Recursive definition of the optimal solution

For any $j \in N$, let $I(j)$ be the size of the MIS in the subset rooted at j, then

$$I(j) = \max\{\sum_{k \text{ child } j} I(k), 1 + \max\{\sum_{k \text{ grandchild } j} I(k)\}$$

The optimal solution will be given by the following bottom-up procedure:

1. Root the tree,
2. for every leaf $j \in T$, $I(j) := 1$,
3. In a bottom-up fashion, for every node j, compute $I(j)$ according to the previous equation.

Complexity:

Obvious time and space complexity: $O(n^2)$.

But, at each vertex, the algorithm only looks at its children and granchildren, therefore each $j \in N$ is looked only 3 times:

1.- when the algorithm computes $I(j)$,

2.- when the algorithm computes the MIS for the father of j,

3.- when the algorithm computes the MIS for the grandfather of j.

Since each j is used a constant number of times, the total number of steps is O(n)