

PROGRAMMATION DYNAMIQUE

INTRODUCTION

C'est une méthode introduite dans le cadre des algorithmes d'optimisation.

OPTIMISATION. — Répond à la problématique de maximisation d'une fonction sous un ensemble de contraintes.

EXEMPLES. — En autres :

- la planification ;
- satisfaction de contraintes ;
- mathématiques financières.

Il y a deux grandes familles de méthodes en optimisation :

- la programmation dynamique ;
- la programmation linéaire.

1. CALCULER UNE SUITE DÉFINIE PAR UNE RÉCUR- RENCE SUR DEUX INDICES

Soit $f : \mathbf{N}^3 \rightarrow \mathbf{N}$. On suppose que l'on dispose d'un algorithme F pour calculer f .
On s'intéresse à la suite définie par :

- $(a_{0,j})_{j \in \mathbf{N}}, (a_{i,0})_{i \in \mathbf{N}}$ donnés ;
- $a_{i+1,j+1} = f(a_{i,j+1}, a_{i+1,j}, a_{i,j})$.

On vérifie qu'il s'agit bien d'une suite :

LEMME 1.1

Soient $(a_{i,j})_{i,j \in \mathbf{N}}$ et $(b_{i,j})_{i,j \in \mathbf{N}}$ telles que :

$$a_{i+1,j+1} = f(a_{i,j+1}, a_{i+1,j}, a_{i,j})$$

$$b_{i+1,j+1} = f(b_{i,j+1}, b_{i+1,j}, b_{i,j})$$

et :

$$\forall i \in \mathbf{N}, a_{i,0} = b_{i,0} \text{ et } a_{0,i} = b_{0,i}.$$

Alors $a_{i,j} = b_{i,j}$ pour tout couple $(i,j) \in \mathbf{N}^2$.

DÉMONSTRATION 1.1

Par double récurrence.

On imbrique une récurrence (externe) sur i avec une récurrence (interne) sur j . Sur i on veut montrer :

$$P_i \equiv \forall j \in \mathbf{N}, a_{i,j} = b_{i,j}.$$

Par définition, P_0 est vraie. On suppose P_i vérifiée pour tous $i \leq i_0$. Il s'agit de montrer P_{i_0+1} par récurrence sur j .

On définit la proposition :

$$Q_j^{i_0+1} \equiv a_{i_0+1,j} = b_{i_0+1,j}.$$

On remarque que P_{i_0+1} est vraie si, et seulement si, $Q_0^{i_0+1}, Q_1^{i_0+1}, \dots$ sont vraies. C'est-à-dire que P_{i_0+1} est vraie si, et seulement si :

$$\bigwedge_{j \in \mathbf{N}} Q_j^{i_0+1}.$$

Par définition, $Q_0^{i_0+1}$ est vraie. On suppose $Q_j^{i_0+1}$ vraie pour tous $j \leq j_0$. On a :

$$a_{i_0+1,j_0+1} = f(a_{i_0,j_0+1}, a_{i_0+1,j_0}, a_{i_0,j_0})$$

$$a_{i_0+1,j_0+1} = f(b_{i_0,j_0+1}, b_{i_0+1,j_0}, b_{i_0,j_0})$$

$$a_{i_0+1,j_0+1} = b_{i_0+1,j_0+1}$$

et donc $Q_{j_0+1}^{i_0+1}$ est vraie.

On en conclut que pour tout i , P_i est vraie et donc le lemme est démontré.

Soit T un tableau tel que $T[i,j] = a_{i,j}$.

ALGORITHME DE CALCUL DE T . — On se donne un algorithme A qui calcule $(a_{i,0})$ et un autre algorithme B qui calcule $(a_{0,i})$.

```

1  function tabIter(n,m)
2      for i from 0 to n do
3          T[i,0] = algoA(i)
4      for j from 0 to m do
5          T[0,j] = algoB(j)
6      for i from 1 to n do
7          for j from 1 to m do
8              T[i,j]=F(T[i-1,j],T[i,j-1],T[i-1,j-1])
9      return T

```

```

1 function tabRec(i, j)
2     if i = 0 then
3         return algoA(i)
4     if j = 0 then
5         return algoB(j)
6     return F(tabRec(i-1, j), tabRec(i, j-1), tabRec(i-1, j-1))

```

2. LCS (LEAST COMMON SUBSEQUENCE) : PROBLÈME D'UNE PLUS LONGUE SOUS-SUITE COMMUNE

EXEMPLE. — Supposons que l'on dispose de deux brins d'ADN :

$$\begin{aligned}
 u &= \text{t t a t a t g c g t} \\
 v &= \text{t a t c c c c t t a},
 \end{aligned}$$

le mot « t a t t t » apparaît dans les deux brins et est le plus grand.

2.1. Propriétés

Soit Σ un alphabet fini. Soient $u, v \in \Sigma^*$. Une sous-suite commune à u et v est un mot $w \in \Sigma^*$ tel que :

$$\exists 1 \leq i_0 < i_1 < \dots < i_{|w|}, \exists 1 \leq j_0 < j_1 < \dots < j_{|w|}, \forall k \leq |w|, w_k = u_{i_k} = v_{j_k}.$$

NOTATIONS. — On suppose que les mots $w \in \Sigma^*$ sont indexés de 1 à $|w|$. On note également

$$w[1 \dots i] = w[1]w[2] \dots w[i] = w_1 w_2 \dots w_i.$$

Dans la suite on considère le tableau $A : \{0, \dots, n\} \times \{0, \dots, m\}$ tel que $A[i, j]$ est la longueur de la plus longue sous-suite commune à $u[1 \dots i]$ et $v[1 \dots j]$.

PROPOSITION 2.1 (Optimalité des sous-structures)

Si w est une sous-suite commune à u et v de longueur maximale k , alors pour tout $h \leq k$, $w[1 \dots h]$ est une sous-suite commune maximale à $u[1 \dots i_h]$ et $v[1 \dots j_h]$.

DÉMONSTRATION 2.1

$w[1 \dots h]$ est bien une sous-suite commune. Si b était une sous-suite, strictement plus grande, à $u[1 \dots i_h]$ et $v[1 \dots j_h]$ alors $bw[h+1, \dots k]$ serait une sous-suite qui contredirait la maximalité de w .

NOTATION. — $\wedge_{x=y}$ est la fonction constante égale à 1 si $x = y$ et 0 sinon.

PROPOSITION 2.2

Si $A[0, j] = A[i, 0] = 0$. Alors

$$A[i+1, j+1] = \max \begin{cases} A[i+1, j] \\ A[i, j+1] \\ \wedge_{u[i+1]=v[j+1]} + A[i, j] \end{cases}.$$

DÉMONSTRATION 2.2

Si une plus longue sous-suite commune à $u[1 \dots i+1]$ et $v[1 \dots j+1]$ à pour dernier élément $u[i+1] = v[j+1]$ alors c'est vérifié.

Si ce n'est pas le cas, le dernier élément de w est différent, soit de $u[i+1]$ soit de $v[j+1]$

- dans le premier cas, w est une plus longue sous-suite commune à $u[1 \dots i]$ et $v[1 \dots j+1]$ et c'est donc vérifié;
- dans le second cas, w est une plus longue sous-suite commune à $u[1 \dots i+1]$ et $v[1 \dots j]$ et c'est vérifié.

2.2. Algorithme

```

1  function tabDynLCS(u, v)
2      n, m = len(u), len(v)
3      for i from 0 to n do A[i, 0] = 0
4      for j from 0 to m do A[0, j] = 0
5      for i from 1 to n do
6          for j from 1 to m do
7              if A[i-1, j] > A[i, j-1] then
8                  L[i, j] = nord
9                  A[i, j] = A[i-1, j]
10             else
11                 L[i, j] = ouest
12                 A[i, j] = A[i, j-1]
13             if u[i] = v[j] and A[i, j] < 1 + A[i-1, j-1] then
14                 L[i, j] = nord-ouest
15                 A[i, j] = 1 + A[i-1, j-1]
16         return A, L
17  function LCS(L, u, v)
18      k, i, j = 0, len(u), len(v)
19      while min(i, j) > 0 do
20          if L[i, j] = nord-ouest then

```

```

21         k = k + 1
22         z[k] = u[i]
23         i = i - 1
24         j = j - 1
25     else if L[i, j] = ouest then j = j - 1
26     else i = i - 1
27 return reverse(z)

```

2.3. Coût

Le coût de l'algorithme sera le coût des fonctions tabDynLCS et LCS.

— tabDynLCS demande $O(n \times m)$.

— LCS demande $n + m$ itérations, c'est-à-dire $O(n + m)$.

Finalement on peut dire que c'est l'algorithme est en $O(n \times m)$ avec n la longueur de la suite u et m celle de v .

3. PROBLÈME DU SAC À DOS

3.1. Présentation du problème

Soient des objets numérotés de 1 à n tel que pour tout $i \leq n$, w_i est le poids de l'objet et p_i sa valeur. La capacité du sac est notée W . Le problème étant de maximiser la somme des valeurs des objets mis dans le sac sous la contrainte du poids.

Notons (x_1, \dots, x_n) une suite de 1 et 0 tel que $x_i = 1$ si l'objet i est dans le sac et vaut 0 sinon. Il s'agit de résoudre le problème

$$\begin{cases} \max_{x_i} \sum_{i=1}^n x_i p_i \\ \sum_{i=1}^n x_i w_i \leq W \end{cases}.$$

3.2. Propriétés

On définit un tableau $KP[i, c]$, un tableau à deux dimensions avec $0 \leq i \leq n$ et $0 \leq c \leq W$. $KP[i, c]$ est la valeur optimale d'un sac à dos ne contenant que des objets d'indices inférieurs à i et dont le poids total ne peut dépasser c . Plus formellement :

$$KP[i, c] = \max \left\{ \sum_{j \in \mathcal{L}} p_j \mid \mathcal{L} \subset \{1, \dots, i\} \text{ et } \sum_{j \in \mathcal{L}} w_j \leq c \right\}.$$

De plus :

$$KP[0, c] = KP[i, 0] = 0.$$

PROPOSITION 3.1

Si $c \geq w_{i+1}$ alors :

$$KP[i+1, c] = \max \begin{cases} KP[i, c] \\ p_{i+1} + KP[i, c - w_{i+1}] \end{cases}.$$

DÉMONSTRATION 3.1

Si $\mathcal{L} \subset \{1, \dots, i\} \subset \{1, \dots, i, i+1\}$ et donc $KP[i, c] \leq KP[i+1, c]$.

Si $\mathcal{L} \subset \{1, \dots, i+1\}$ et $i+1 \in \mathcal{L}$ alors

$$\sum_{j \in \mathcal{L}} p_j = p_{i+1} + \sum_{j \in \mathcal{L} \setminus \{i+1\}} p_j.$$

On a

$$KP[i+1, c] \geq p_{i+1} + KP[i, c].$$

Et donc

$$KP[i+1, c] \geq \max \{KP[i, c], p_{i+1} + KP[i, c - w_{i+1}]\}.$$

Supposons qu'il existe une solution \mathcal{L} au problème $(i+1, c)$.

— Si $i+1 \in \mathcal{L}$ alors $KP[i+1, c] = p_{i+1} + \sum_{j \in \mathcal{L} \setminus \{i+1\}} p_j$. Or

$$\sum_{j \in \mathcal{L}} w_j \leq c \iff \sum_{j \in \mathcal{L} \setminus \{i+1\}} w_j \leq c - w_{i+1}.$$

$\mathcal{L} \setminus \{i+1\}$ est nécessairement optimale pour $(i, c - w_{i+1})$. Dans le cas contraire $\mathcal{L}' \cup \{i+1\}$ nierait l'optimalité de \mathcal{L} . En conclusion, on a

$$KP[i+1, c] = p_{i+1} + KP[i, c - w_{i+1}].$$

— Si $i+1 \notin \mathcal{L}$ alors $\mathcal{L} \subset \{1, \dots, i\}$ et donc

$$KP[i+1, c] = KP[i, c].$$

On a bien

$$KP[i+1, c] = KP[i, c] \bigwedge_{i+1 \notin \mathcal{L}} + (p_{i+1} + KP[i, c - w_{i+1}]) \bigwedge_{i+1 \in \mathcal{L}} = \max \{KP[i, c], p_{i+1} + KP[i, c - w_{i+1}]\}.$$

3.3. L'algorithme

```
1 function knapSack(w,n,p,W)
2   for i from 0 to n do
3     kp[i,0] = 0
4   for c from 0 to W do
5     kp[0,c] = 0
6   for i from 1 to n do
7     for c from 1 to W do
8       if c > w[i] then
```

```

9                                      $kp[i, c] = \max(kp[i-1, c], p[i] + kp[i-1, c-w[i]])$ 
10                                else
11                                      $kp[i, c] = kp[i-1, c]$ 
12    for i from 1 to n do
13         $w[i] = 0$ 
14     $c = W$ 
15    for i from n to 1 do
16        if  $kp[i, c] \neq kp[i-1, c]$  then
17             $x[i] = 1$ 
18             $c = c - w[i]$ 
19    return x, kp

```