906 Programmation dynamique : exemples et applications

2703 2015

La programmation dynamique, comme la méthode "diviser pour régner", résout des problèmes en combinant des solutions optimales de sous-problèmes. La méthode "diviser pour régner" partitionne les problèmes en sous-problèmes indépendants qui elle résout récursivement, puis combine les solutions pour résoudre le problème initial. La programmation dynamique peut s'appliquer quand ces sous-problèmes ne sont pas indépendants. En effet dans un tel cas, les algorithmes "diviser pour régner" effectuent plusieurs fois les mêmes opérations. En programmation dynamique, chaque sous-problème est résolu une seule fois et sa réponse est mémorisée.

[Cormen]

I Présentation du Paradigme

1. Exemple de limite des algorithmes récursifs

Problème 1 : Calculer la valeur de C_n^k.

Proposition 1 : $\forall (k,n) \in \mathbb{N}^2,\ 0 < k < n \Rightarrow C_n^k = C_{n-1}^{k-1} + C_{n-1}^{k}$

Approche diviser pour régner :

```
Fonction C(n, k) :
    si k=0 ou k=n retourner 1
    sinon retourner C(n-1, k-1) + C(n-1, k)
```

Cet algorithme est exponentiel et des calculs sont effectués plusieurs fois.

Meilleure idée : Le Triangle de Pascal.

Principe : → Remplir les k premières cases de chaque ligne de haut en bas.
→ Arrêter à la ligne n.

L'algorithme s'effectue alors en $\Theta(nk)$.

2. Mémoïsation & Programmation dynamique. [Cormen]

La programmation dynamique, principalement utilisée pour résoudre des problèmes d'optimisation, s'appuie sur le principe de Bellman : toute solution optimale s'appuie sur des sous-problèmes résolus localement de façon optimale. Les solutions des problèmes sont calculés par une approche bottom-up : on débute par les solutions des sous-problèmes les plus petits pour ensuite décrire progressivement les solutions de l'ensemble.

Définition 01 : Programmation dynamique

Le principe de la programmation dynamique est le suivant :

```
Fonction Résoudre le Problème P
    Résoudre les sous-problèmes triviaux de P
    Parcourir les sous problèmes de P dans l'ordre croissant
        Résoudre le sous-problème
        à partir des solutions des problèmes plus petit.
    Retourner une solution du problème P.
```

Exemple 1 : calcul dynamique de la suite de Fibonacci

```
F(n) =
    T(0) ← 1;
    T(1) ← 1;
    for i=2 to n
        T·(i) ← T·(i-1) + T·(i-2)
    Retourner T·(n)
```

Il existe une variante de la programmation dynamique qui offre souvent la même efficacité que l'approche usuelle mais en maintenant une approche descendante : la mémoïsation :

Définition 02 : Mémoïsation (ou recensement)

Le principe de la Mémoïsation est le suivant

```
Fonction f(n)
    Si f(n) est déjà calculé
        Retourner la valeur
    Sinon
        Calculer f(n)
        Stocker la valeur
        Retourner la valeur
```

Exemple : calcul de la suite de Fibonacci

```
F(n) =
    Si M(n) existe retourner M(n)
    Sinon M(n) := F(n-1) + F(n-2);
          Retourner M(n);
```

Remarque: Dans le cadre de problèmes d'optimisation, on recherche une sous-structure optimale (les sous-problèmes à résoudre). Mais une telle sous-structure peut ne pas exister.

exemple: Recherche du plus long chemin élémentaire dans un graphe non valué.

indydi →

II Application de la programmation dynamique à l'algo. du texte

1. Distances entre les mots [PAP]

Quand un correcteur orthographique trouve une erreur dans un texte, il cherche dans son dictionnaire pour trouver un mot le plus proche. Pour mesurer la distance entre les mots, on cherche à les aligner, c-est-à-dire, à les écrire l'un au dessus de l'autre. Voici deux alignements possible des mots SUNNY et SNOWY (le symbole _ indique un gap):

S_NOWY	_SNOW_Y
SUNN_Y	SUN__NY
coût: 3	coût: 5

Le coût d'un alignement est défini comme le nombre de colonnes avec une différence.

définition 03: Distance d'édition
On appelle distance d'édition entre deux chaînes de caractères le coût de leur alignement de coût minimum

Le problème est de calculer la distance d'édition entre deux chaînes de caractères.
En général, il y a beaucoup d'alignements possibles entre deux chaînes de caractères. Il est par conséquent terriblement inefficace de mener une recherche naïve sur tout les alignements. Nous cherchons une solution à l'aide de la programmation dynamique.
Lors de la résolution d'un problème via la programmation dynamique, la question est d'identifier les sous-problèmes.
Ici, les sous-problèmes sont l'alignement des préfixes.

définition 04: Sous-problèmes
Soient x et y, deux chaînes de caractères de tailles respectives n et m.
$\forall (i,j) \in [\![1,n]\!] \times [\![1,m]\!]$, on appelle $E(i,j)$ la distance d'édition entre le préfixe de x de taille i et celui de y de taille j.

L'objectif final est donc de calculer $E(n,m)$.
Pour ce faire, nous exprimons $E(i,j)$ en fonction de ses sous-problèmes:

proposition 01:
$\forall (i,j) \in [\![1,n]\!] \times [\![1,m]\!]$,

$$E(i,j) = \min\left\{1 + E(i-1,j);\ 1 + E(i,j-1);\ d(i,j) + E(i-1,j-1)\right\}$$

où $d(i,j) = 0$ si $x[i] = y[j]$ et 1 sinon.

De cette relation on déduit l'algorithme

```
dist(x, y) =
    n = taille(x)
    m = taille(y)
    E = Matrice(n+1, m+1)
    for i = 0 to n
        E(i,0) ← i
    for j = 0 to m
        E(j,0) ← j
    for i = 1 to n
        for j = 1 to m
            E(i,j) = Min(1+E(i-1,j); 1+E(i,j-1), d(i,j)+E(i-1,j-1))
    Retourner E(n,m);
```

Autre exemple: Recherche de la plus longue sous-séquence commune à deux chaînes de caractères. [DVP] [Cormen p 341 et s]

2. Langages Algébriques : L'Algorithme de CYK [Canton]

Soit G, une grammaire sous forme normale de Chomsky. On s'intéresse au problème de décision suivant :

entrée : x un mot
sortie : x appartient au langage engendré par G.

Pour répondre à ce problème, il est inefficace de tester toutes les dérivations qui génèrent des mots de la taille de x.

L'algorithme de Cocke-Younger-Kasami applique le principe de la programmation dynamique. Les sous-problèmes consistent à identifier les variables non-terminales de G qui engendrent les sous-mots de x et la combinaison se fait vite :

Proposition 05 :

Avec $x = x_1 \dots x_n$, on note $x_{i,j} = x_i \dots x_j$ et $E_{i,j} = \{S \in T,\ S \xrightarrow{*} x_{i,j}\}$, il vient,

$$S \in E_{i,j} \iff \exists k \in [\![i, j-1]\!],\ \begin{cases} S \to S_1 S_2 \in P \\ S_1 \in E_{i,k} \\ S_2 \in E_{k+1,j} \end{cases}$$

où $G = (\Sigma, T, P)$

3. Programmation dynamique pour les dictionnaires. [Cormen]

Lorsque l'on veut écrire un algorithme de traduction, on peut effectuer une traduction mot à mot. Une méthode d'implantation peut être de stocker les mots d'une langue dans un arbre indicé par les mots d'une autre langue. Les mots ayant des fréquences d'apparition différentes, d'utilisation d'arbre binaire de recherche équilibrés peut s'avérer peu efficace. On utilise alors la programmation dynamique pour générer des Arbres Binaires de Recherche optimaux dont le rôle est de garantir un coût moyen minimal. [DVP]

VI Applications aux Graphes

La programmation dynamique permet de résoudre des problèmes de plus court chemins.

Algorithme de Floyd-Warshall

On s'intéresse au plus court chemin reliant deux sommet dans un graphe orienté G.

Le principe consiste à étudier les sommets intermédiaires. On numérote les sommets de G de 1 à n et on note $\forall (i,j) \in [\![1, n]\!]$, $d_{i,j}^{(k)}$ le poids du plus court chemin du sommet i vers le sommet j dont tout les sommets intermédiaires sont numérotés de 1 à k.

Il vient dont

Proposition :

$$\forall k \geq 1,\ \forall (i,j) \in [\![1, n]\!]^2,\quad d_{i,j}^{(k)} = \min\left(d_{i,j}^{(k-1)},\ d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}\right)$$

Basé sur cette formule de récurrence, on peut donc calculer de manière ascendante les valeurs des $d_{i,j}^{(k)}$

= Bellman-Ford
- pb du sac-à-dos [PAP]

[PAP] : Papadimitriou
[COR] : Cormen
[DVP] : Beauquier, Froidevaux, Roguiez.

CYK ← [CAR] : Canton

2.10 Plus longue sous-séquence commune

Cormen *et al.* (n.d.)

Étant donnés deux mots $X = x_1 \cdots x_m$ et $Y = y_1 \cdots y_n$ sur un alphabet Σ, on cherche à savoir quel est le plus long sous-mot commun à X et Y.

La solution naïve est d'énumérer tous les sous-mots de X, et de regarder ensuite s'ils sont aussi sous-mots de Y, mais cette solution est inefficace : il y a 2^m sous-mots de X.

NOTATION – Si U est un mot de longueur p, on note U_{p-j} le préfixe de U de longueur $p - j$.

Un algorithme plus efficace pourra être déduit du théorème de structure suivant :

Théorème 2.10.1 : de structure des plus longues sous-séquences communes
Supposons que $Z = z_1 \cdots z_k$ est une plus longue sous-séquence commune à X et Y. Alors :

(i) Si $x_m = y_n$, alors $z_k = x_m (= y_n)$ et Z_{k-1} est une plus longue sous-séquence commune à X_{m-1} et Y_{n-1}.

(ii) Si $x_m \neq y_n$ alors

$$(z_k \neq x_m) \Rightarrow Z \text{ est une plus longue sous-séquence commune à } X_{m-1} \text{ et } Y.$$

(iii) Si $x_m \neq y_n$ alors

$$(z_k \neq y_n) \Rightarrow Z \text{ est une plus longue sous-séquence commune à } X \text{ et } Y_{n-1}.$$

Démonstration. C'est plutôt facile :

(i) $x_m = y_n$. Si $z_k \neq x_m$, alors Zx_m est un sous-mot commun à X et Y, ce qui contredit l'hypothèse de maximalité. D'où $z_k = x_m = y_n$.
De plus, Z_{k-1} est un sous-mot commun à X_{n-1} et Y_{m-1}. Supposons qu'il existe un sous-mot commun W de longueur plus grande que $k - 1$.
Alors, Wx_m est de longueur plus grande que k, et Wx_m est un sous-mot commun à X et Y, d'où une contradiction.
Donc Z_{k-1} est une plus longue sous-séquence commune à X_{m-1} et Y_{n-1}.

(ii) $x_m \neq y_n$. Supposons $z_k \neq x_m$. Alors Z est un sous-mot de X_{m-1}, et donc *a fortiori un sous-mot commun à X_{m-1} et Y.*
Supposons qu'il existe un sous-mot W commun à X_{m-1} et Y, de longueur plus grande. Alors W est a fortiori un sous-mot commun à X et Y, ce qui contredit l'hypothèse de maximalité de Z.

(iii) *cf* (ii)

□

Connaissant la plus longue sous-séquence commune au mot vide et n'importe quel mot, on peut en déduire une formule de récurrence sur la longueur de la plus longue sous-séquence commune de X et Y. Si on note $c[i, j]$ la longueur de la plus longue sous-séquence commune à X_i et Y_j, on a :

$$c[i,j] = \begin{cases} 0 & \text{si } i = 0 \text{ ou } j = 0 \\ c[i-1, j-1] + 1 & \text{si } i, j > 0 \text{ et } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{si } i, j > 0 \text{ et } x_i \neq y_j \end{cases} \tag{2}$$

Cette formule pourrait nous donner un algorithme récursif pour calculer $c[m, n]$, mais sa complexité serait à nouveau exponentielle. On va plutôt utiliser un algorithme de *programmation dynamique.*

Algorithm 6 Longueur-PLSC

Préconditions: X, Y mots sur Σ

```
m ← Longueur(X)
n ← Longueur(Y)
pour i = 1 to m faire
        c[i, 0] ← 0
ruop
pour j = 0 to n faire
        c[0, j] ← 0
ruop
pour i = 1 to m faire
        pour j = 1 to n faire
                si x_i = y_j alors
                        c[i, j] ← c[i − 1, j − 1] + 1
                        b[i, j] ← " ↖ "
                sinon si c[i − 1, j] ⩾ c[i, j − 1] alors
                        c[i, j] ← c[i − 1, j]
                        b[i, j] ← " ↑ "
                sinon
                        c[i, j] ← c[i, j − 1]
                        b[i, j] ← " ← "
                is
        ruop
ruop

retourner  c et b
```

Cet algorithme va calculer les uns après les autres les valeurs des $c[i, j]$, en utilisant la formule de récurrence, en remplissant le tableau par lignes.

La matrice b sert à noter "d'où on vient" pour calculer la valeur $c[i, j]$ d'une case. Grâce à ce tableau, on peut donc retrouver la plus longue sous-séquence commune à X et Y par l'algorithme :

Algorithm 7 PLSC

Préconditions: X mot sur Σ, b matrice calculée par Longueur-PLSC, i, j indices.

```
si i = 0 ou j = 0 alors

        afficher ()
is
si b[i, j] = " ↖ " alors
        PLSC(b, X, i − 1, j − 1)

        afficher x_i
sinon si b[i, j] = " ↑ " alors
        PLSC(b, X, i − 1, j)
sinon
        PLSC(b, X, i, j − 1)
is
```

Théorème 2.10.2

La procédure Longueur-PLSC a une complexité en $\mathcal{O}(mn)$, et la procédure PLSC a une complexité en $\mathcal{O}(m+n)$.

Démonstration. Dans Longueur-PLSC, on a deux boucles "pour" imbriquées, et dans ces boucles des opérations en $\mathcal{O}(1)$. D'où la complexité.

Dans PLSC, on réalise un "chemin direct" dans notre matrice $(i+1) \times (j+1)$, et donc ce chemin ne peut avoir plus de $i+j$ étapes. □

EXEMPLE – On cherche la plus longue sous-séquence commune aux chaînes `BDCABA` et `ABCBDAB`. Le tableau construit par Longueur-PLSC est :

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0↑	0↑	0↑	1↖	1←	1↖
2	B	0	1↖	1←	1←	1↑	2↖	2←
3	C	0	1↑	1↑	2↖	2←	2↑	2↑
4	B	0	1↖	1↑	2↑	2↑	3↖	3←
5	D	0	1↑	2↖	2↑	2↑	3↑	3↑
6	A	0	1↑	2↑	2↑	3↖	3↑	4↖
7	B	0	1↖	2↑	2↑	3↑	4↖	4↑

On suit les flèches (chemin grisé) et on note les lettres correspondant aux flèches obliques : `BCBA`.

PLSC : on peut utiliser un tableau de taille $2 \times n$, en écrasant une ligne.

2 Informatique

2.1 Arbres binaires de recherche optimaux

Cormen *et al.* (n.d.)

On s'intéresse ici à optimiser les arbres binaires de recherche quand on connait la probabilité pour chaque clef d'être recherchée.

On se donne n clefs distinctes triées dans l'ordre croissant $k_1 < \cdots < k_n$, où pour tout i, k_i a une probabilité p_i d'être recherchée.

On rajoute aussi $n+1$ clefs factices $d_0, \ldots, d_n$ représentant les recherches qui ne sont aucun des k_i.

d_0 représente toutes les valeurs inférieures à k_1, d_n celle supérieures à k_n, et d_i celles comprises entre k_i et k_{i+1}. On se donne de plus une probabilité q_i que la recherche soit dans d_i.

Les nœuds internes de l'arbre seront les k_i, et les feuilles les d_i.

On a donc

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1.$$

Le coût moyen d'une recherche dans notre arbre sera

$$1 + \sum_{i=1}^{n} \mathrm{Prof}(k_i) p_i + \sum_{i=0}^{n} \mathrm{Prof}(d_i) q_i.$$

Un arbre binaire de recherche sera dit *optimal* si ce coût moyen est minimal. On abrège en "ABRO".

La recherche d'un ABRO en listant tous les arbres possibles a une complexité bien trop grande, on va donc essayer un algorithme de programmation dynamique.

On remarque pour cela que tout sous-arbre de notre arbre *doit* être optimal : sinon, on le remplace par un optimal, faisant ainsi baisser le coût total.

Le sous-problème à résoudre est donc, étant donnés $i \geqslant 1$ et $i - 1 \leqslant j \leqslant n$ un ABRO contenant les clefs $k_i, \ldots, k_j$.

Soit $e[i, j]$ le coût d'un arbre contenant $k_i, \ldots, k_j$.

Si $j = i - 1$, alors il n'y a que la clef factice d_{i-1} dans l'arbre : le coût moyen est q_{i-1}.

Sinon, $j \geqslant i$. On choisit un nœud k_r, $i \leqslant r \leqslant j$, et on construit l'arbre de racine k_r. Le sous-arbre gauche de k_r contiendra les clefs $k_i, \ldots, k_{r-1}$, et le sous arbre droit les clefs $k_{r+1}, \ldots, k_j$.

Quand on fait "descendre" un sous-arbre, on augmente la profondeur de ce sous-arbre de 1, on donc on augmente le coût de

$$w(i, j) = \sum_{k=i}^{j} p_k + \sum_{k=i-1}^{j} q_k.$$

On obtient donc la formule :

$$e[i, j] = p_r + e[i, r-1] + w(i, r-1) + e[r+1, j] + w(r+1, j).$$

On remarque que

$$w(i, j) = w(i, r-1) + p_r + w(r+1, j),$$

et donc

$$e[i,j] = e[i,r-1] + e[r+1,j] + w(i,j).$$

Cette formule est valide si on a choisi la bonne racine pour notre sous-arbre. On optimise donc :

$$e[i,j] = \begin{cases} q_{i-1} & \text{si } j = i-1 \\ \min_{i \leqslant r \leqslant j} e[i,r-1] + e[r+1,j] + w(i,j) & \text{si } i \leqslant j \end{cases}$$

Pour garder une trace de notre construction, on définit $racine[i,j]$ comme l'indice de la racine optimale pour le sous-arbre contenant $k_i, \ldots, k_j$.

On va maintenant utiliser la programmation dynamique pour calculer $e[1,n]$.

REMARQUE – Plutôt que de calculer tous les $w(i,j)$ à chaque fois, on crée un tableau défini par

$$\begin{cases} w(i,i-1) = q_{i-1} \\ w(i,j) = w(i,j-1) + p_j + q_j \end{cases}$$

On a donc l'algorithme suivant :

Algorithm 1 ABRO

Préconditions: p, q probabilités, n nombre de clefs

```
pour i = 1 à n + 1 faire
        e[i, i − 1] ← q_{i−1}
        w[i, i − 1] ← q_{i−1}
ruop
pour ℓ = 1 à n faire
        pour i = 1 à n − ℓ + 1 faire
                j ← i + ℓ − 1
                e[i, j] ← ∞
                w[i, j] ← w[i, j − 1] + p_j + q_j
                pour r = i à j faire
                        t ← e[i, r − 1] + e[r + 1, j] + w[i, j]
                        si t < e[i, j] alors
                                e[i, j] ← t
                                racine[i, j] ← r
                        is
                ruop
        ruop
ruop

retourner e et racine
```

Les trois boucles imbriquées nous donnent une complexité en $\mathcal{O}(n^3)$.