Programmation dynamique 1

Fabrice Lembrez - PSI*

Lycée Pierre de Fermat

Il s'agit de décomposer efficacement un problème d'optimisation en sous-problèmes de même nature. Cela va nous conduire à des formules de récurrence, les **équations de Bellmann**.

Plan

Dresser la tente

- on veut installer une tente carrée sur un terrain de camping rectangulaire.
- on cherche le plus gros espace carré disponible
- terrain est mal entretenu, les pyracanthas affleurent.

Quel est le plus gros espace carré disponible ?

1	0	0	1	0	0	1
0	0	0	0	0	0	0
1	0	0	0	0	0	0
0	0	0	0	0	0	0
0	1	0	0	0	0	1
1	0	0	0	1	0	1

$\Rightarrow$

1	0	0	1	0	0	1
0	0	$*$	$*$	$*$	$*$	0
1	0	$*$	$*$	$*$	$*$	0
0	0	$*$	$*$	$*$	$*$	0
0	1	$*$	$*$	$*$	$*$	1
1	0	0	0	1	0	1

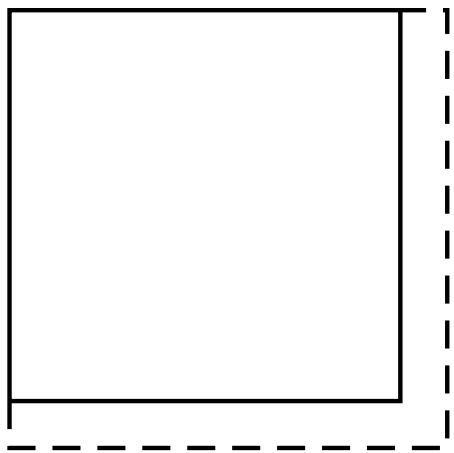
Sur cet exemple : un carré de taille 4.

Plan

Principe : pour chaque i et chaque j, on calcule le plus grand carré libre possible d'"origine" i,j. Et on garde en mémoire le carré maximal au fur et à mesure.

Principe : pour chaque i et chaque j, on calcule le plus grand carré libre possible d'"origine" i,j. Et on garde en mémoire le carré maximal au fur et à mesure.

Cela fait **quatre boucles imbriquées** :
- une boucle pour i, une pour j (origine),
- une pour le côté c du carré
    (qu'on fait grossir de plus en plus tant que c'est possible)
- et pour valider le passage de c à c+1, il faut une boucle pour vérifier que le bord est libre



Code naïf

Je préfère utiliser une sous fonction `bordureLibre` pour tester si on peut agrandir le carré en passant de c à c+1. Cette boucle est quand même imbriquée avec les autres à cause de l'appel de fonction :

```
def bordureLibre(A,i,j,c):
    for k in range(c+1):
        if A[i+k][j+c]==1 or A[i+c][j+k]==1:
            return False
    return A[i+c][j+c]==0
```

Code naïf

```
def carreMaxNaif(A):
    nbLgn,nbCol=len(A),len(A[0])
    cMax=0
    for i in range(nbLgn):
        for j in range(nbCol):
            c=0
            while i+c<nbLgn and j+c<nbCol ...
                                ...and bordureLibre(A,i,j,c)
                c+=1
            if cMax>c:
                cMax=c
    return cMax
```

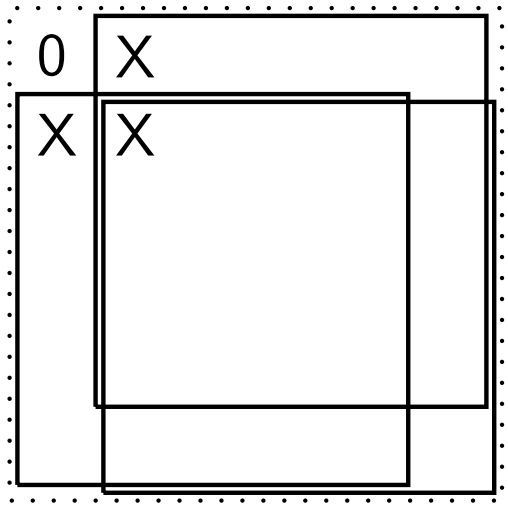
En tenant compte de la fonction appelée cela fait bien 4 boucles imbriquées.

Plan

Solution dynamique : le principe

Principe : les calculs qui ont été faits pour chaque i et chaque j ne sont pas indépendants. **On décide de les calculer dans un ordre adapté et de tous les mémoriser (dans un tableau `Kmax`).**

Si on s'intéresse à la case i,j et si on suppose que les valeurs de `Kmax` pour des indices supérieurs sont connues, il y a deux cas
- case `A[i][j]` indisponible (valeur 1) : alors `Kmax[i][j]` vaut 0
- case libre (valeur 0) : on peut utiliser la connnaissance des autres



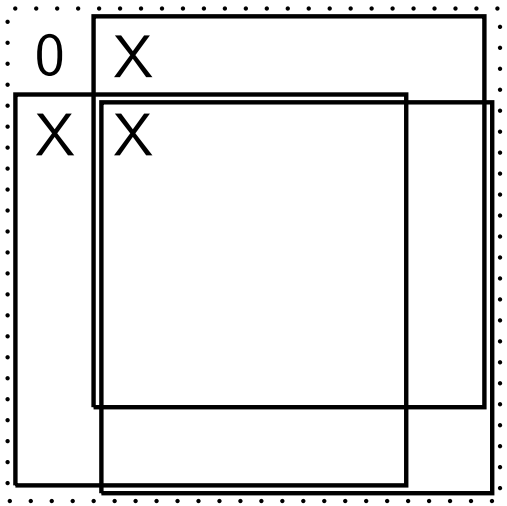
L'équation du problème

On suppose que les valeurs de `Kmax` pour des indices supérieurs à i et j sont connues, il y a deux cas

- case `A[i][j]` indisponible (valeur 1) : alors `Kmax[i][j]` vaut 0
- case libre (valeur 0) : on peut utiliser la connaissance des autres

```
Kmax[i][j]=1+min(Kmax[i+1][j],Kmax[i][j+1],Kmax[i+1][j+1])
```

Car pour regarder si un carré de taille c+1 est libre il suffit de voir si 3 carrés de taille c sont libres :



Code dynamique

On implémente alors le calcul du tableau `Kmax`, en remontant les indices i et j.
Pour initialiser, on a besoin des valeurs pour i maximum ou j maximum.
On peut utiliser une "astuce" en augmentant la taille du tableau `Kmax` de 1 et en initialisant ces valeurs supplémentaires à 0 (pas de carré libre dans ces directions).

Code dynamique

```
def carreMax(A,lgn,col):
    Kmax=[[0 for j in range(col+1)] ...
                ...for i in range(lgn+1)]
    for j in range(col-1,-1,-1):
        for i in range(lgn-1,-1,-1):
            if A[i][j]==0:
                Kmax[i][j]=1+min(Kmax[i+1][j],...
                                 ...Kmax[i][j+1],Kmax[i+1][j+
            else:
                Kmax[i][j]=0
    return Kmax
```

Si on le souhaite on peut aussi récupérer la valeur maximale de `Kmax`. Mais on a en fait ramené plus d'information que dans la solution naïve.

Comparaison des deux solutions

Pour un terrain carré de côté n, pour simplifier.

Complexité temporelle $O(n^4)$ pour la méthode naïve, $O(n^2)$ pour la méthode dynamique. Cette dernière utilise aussi de la complexité spatiale en $O(n^2)$ mais **l'amélioration de la complexité temporelle est prioritaire.**

Plan

Un exemple : le rendu de monnaie

Le problème : rendre la monnaie sur une somme S avec le moins de pièces et billets possibles. On suppose qu'on a autant de pièces et billets que nécessaire.

Version "standard" : avec le système classique (1,2,5,10...)

Version "générale" : avec un système tordu (ex : 1,7,23)
Cette version générale rend le problème plus compliqué !

Plan

Solution classique du problème (= gloutonne)

- donner la plus grosse pièce ou billet inférieure à S
- mettre à jour S
- si $S = 0$ STOP, sinon recommencer

Déf - Algorithme glouton

On regarde les étapes séparément.
A chaque étape on fait le meilleur choix disponible.
C'est une démarche très rapide.
Mais aucune garantie de tomber sur une solution globalement optimale.

Démarche gloutonne

Algorithmes gloutons

Aucune garantie de tomber sur une solution globalement optimale.
Mais donne très rapidement une valeur, qui est souvent très performante même si pas optimale.
Et parfois il arrive qu'un algorithme glouton donne effectivement un optimum.

Exemple : pour le rendu de monnaie standard glouton $\rightarrow$ optimal
Pour de nombreux systèmes monétaires glouton $\rightarrow$ optimal

Mais pour le système (1,7,23) non : cf découpage de 28

Chercher une solution optimale

On va dorénavant chercher une solution pour laquelle l'optimalité est garantie, pour le système (1,7,23) (ou un système quelconque).

Pour ne pas surcharger les programmes on répond juste à la question suivante dans un premier temps : **quel est le nombre minimal de pièces pour rendre la monnaie sur une certaine somme s ?** On verra ensuite comment la rendre concrètement.

Plan

Examen exhaustif

On peut représenter un examen exhaustif sous forme arborescente : tous les découpages possibles de la somme à l'aide de (1,7,23). Cela revient à explorer l'arbre des possibles

```
def nbPieces(somme,systeme):
    if somme==0:
        return 0
    meilleurRendu=somme
    for piece in (1,7,23):
        if piece<=somme:
            rendu=1+nbPieces(somme-piece,systeme)
            if rendu<meilleurRendu:
                meilleurRendu=rendu
    return meilleurRendu
```

Examen exhaustif

L'arbre complet est très imposant, même si le problème est suffisamment simple ici pour trouver quelques stratégies de limitation des visites.
En outre il y a des redondances de plus en plus importantes.

Une telle méthode d'examen exhaustif est de complexité exponentielle. On va chercher une solution qui garantisse l'optimalité mais qui limite l'exploration à la zone "utile".

Plan

Equation associée

On voit que le problème (découper 28 de façon optimale) est ramené à un des trois problèmes analogues
- découper 27 de façon optimale
- découper 21 de façon optimale
- découper 5 de façon optimale

On peut donc généraliser le problème : comment découper une somme S de façon optimale ? Et utiliser une formule de récurrence

$$N_S = \begin{cases} \min(N_{S-1}, N_{S-7}, N_{S-23}) + 1 & \text{si } S \geq 23 \\ \min(N_{S-1}, N_{S-7}) + 1 & \text{si } S \geq 7 \\ N_{S-1} + 1 & \text{sinon} \end{cases}$$

Calcul

C'est un calcul "de bas en haut" (on verra une autre stratégie) : on remplit progressivement le tableau

```
def nbPieces(somme):
    N=[0]
    for s in range(1,somme+1):
        N1=N[s-1]
        if s>=7 and N[s-7]<N1:
            N1=N[s-7]
        if s>=23 and N[s-23]<N1:
            N1=N[s-23]
        ListeNb.append(N1+1)
    return N[len(N)-1]
```

Plan

Equations de Bellman

Déf - Equations de Bellman

Le principe général de la programmation dynamique est d'obtenir des relations de récurrence pour ramener un calcul d'optimisation à un calcul sur des sous problèmes (on parle d'équation de Bellman).

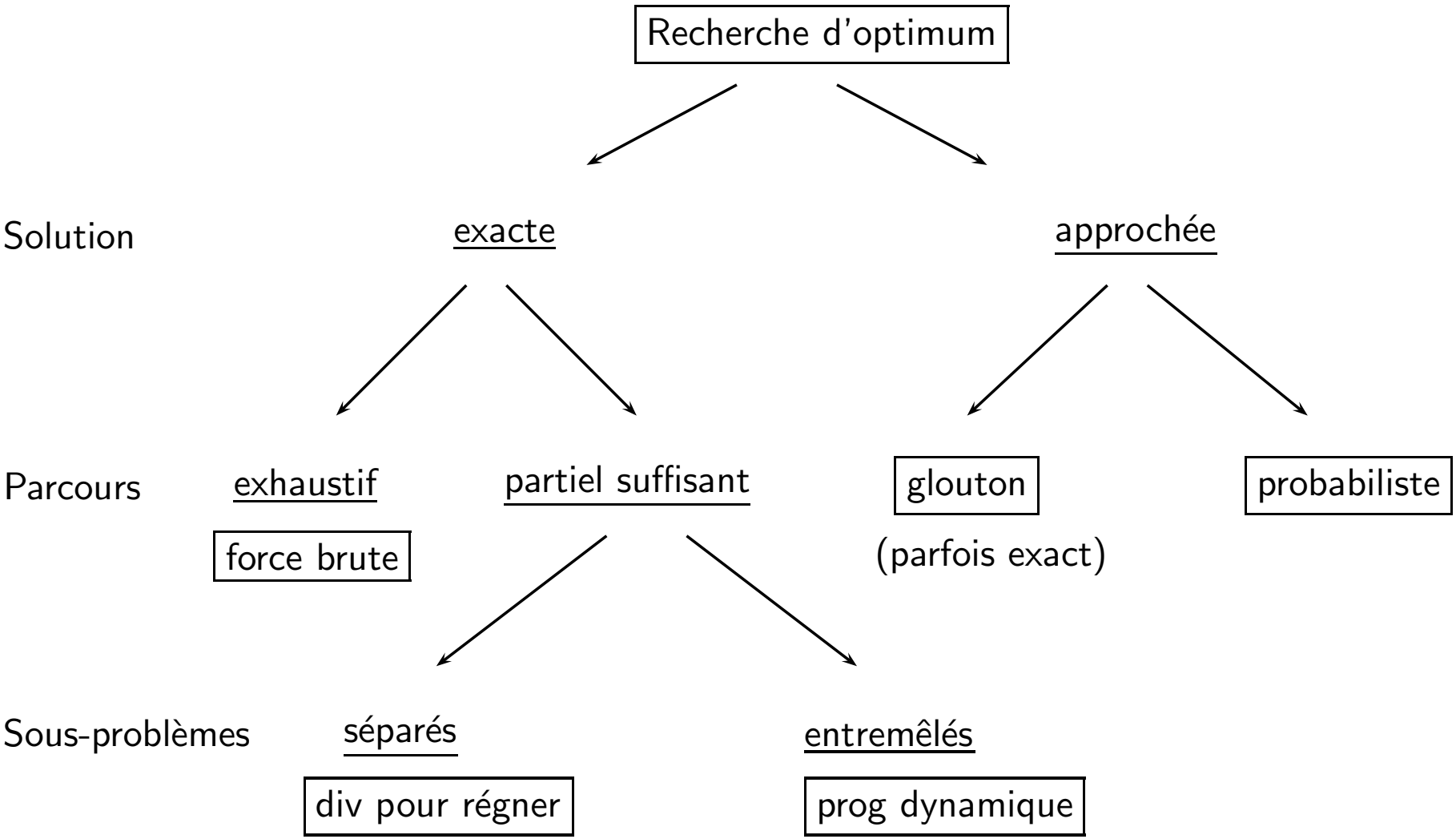
Le point clef de la preuve

Pour que cela fonctionne il faut pouvoir dire qu'un tel sous-problème est lui-même optimal.

Exemple : redétailler la formule de récurrence pour N_S.

Plan

Typographie rapide



Visualisation dans le cas du rendu de monnaie

- Parcours exhaustif : 3 branches à chaque fois
- Parcours glouton : une branche choisie a priori
- Parcours dynamique : 3 branches considérées, une seule mémorisée

Dynamique vs divide and conquer

Ce sont deux idées proches : trouver la solution exacte par un parcours partiel en reliant le problème à des sous problèmes plus simples.

Diviser pour régner (Divide and conquer)

Formellement $A = A_1 \cup A_2$ disjoint, relier M à M_1 et M_2.
On s'est totalement **ramené à deux sous problèmes optimaux indépendants.**
Exemples : recherche dichotomique, tri fusion ou tri rapide

Programmation dynamique

Ici on **retrouve avec des sous problèmes optimaux mais qui se chevauchent.**
Ex : le problème des tentes, celui du rendu de monnaie