

Algorithmique et Complexité

5. Stratégie II: Programmation Dynamique

Nicole Bidoit

Université Paris XI, Orsay

Année Universitaire 2008–2009

Programmation dynamique

Problèmes d'optimisation

Recherche d'un élément (d'un sous-ensemble d'éléments) qui **maximise** ou **minimize** une fonction.

Exemples

| | | |
|---|-----------------------|--------------------------------------|
| Plus court chemin | Problème du sac à dos | Morphing d'images |
| Multiplication d'une chaîne de matrices | Distance d'édition | Compression de données (codes barre) |
| Triangularisation de polynômes | | ... |

Principe général

1. Résoudre le problème **P** avec un algorithme **récuratif A** (ou une relation de récurrence)
2. Etant donné une instance **I** quelconque de taille n ,
montrer que le nombre de sous-instances générées par **A** à partir de **I** est borné (par un polynôme)
 - ↪ l'exécution de **A** sur **I** résout donc "souvent" les mêmes sous-instances
 - ↪ ces **solutions "intermédiaires" sont stockées pour être réutilisées**
3. Orchestration : trouver un ordre d'évaluation des sous-instances pour que les solutions "intermédiaires" soient disponibles à temps.

!!!!!!!!! Compromis espace – temps !!!!!!!!

Programmation dynamique

Multiplication d'une chaîne de matrices et parenthésage

entrée : n matrices A_i de dimension $d_{i-1} \times d_i$

sortie : $A_1.A_2. \dots .A_n$

Coût d'une multiplication de matrices (algo simple) :

matrice $d_{i-1} \times d_i$ "par" matrice $d_i \times d_{i+1}$ \longrightarrow $d_{i-1} \times d_i \times d_{i+1}$ multiplications.

matrice $d_{i-1} \times d_i$ "par" matrice $d_i \times d_{i+1}$ \longrightarrow matrice $d_{i-1} \times d_{i+1}$

Chercher un gain de temps

\hookrightarrow non commutativité : impossible de permuter l'ordre des matrices

\hookrightarrow associativité : éviter le calcul de "grandes" matrices intermédiaires \longleftarrow **parenthésage**

Exemple : Supposons $n=3$ $d_0=10, d_1=100, d_2=5$ et $d_3=50$

| parenthésage | nb multiplications | détails |
|-----------------|--------------------|--|
| $(A_1.A_2).A_3$ | 7500 | $10 \times 100 \times 5 + 10 \times 5 \times 50$ |
| $A_1.(A_2.A_3)$ | 75000 | $100 \times 5 \times 50 + 10 \times 100 \times 50$ |

\longleftarrow **impact du parenthésage**

Énoncé du problème (d'optimisation): Étant donné un produit de n matrices $A_1.A_2 \dots A_n$,
parenthésier le produit des matrices pour **minimiser le nombre de multiplications scalaires**.

Programmation dynamique – Parenthésage d'un produit de matrices

Recherche exhaustive : Examiner chaque parenthésage, un par un et garder le meilleur (au sens du produit des matrices).

combien de parenthésages à examiner ?

$$P(n) = \sum_{k=1..n-1} P(k).P(n-k) \quad P(n) \text{ est dans } \Omega\left(\frac{4^n}{n^{\frac{3}{2}}}\right)$$

nombre de parenthésages est exponentiel \hookrightarrow pas la bonne stratégie

Programmation dynamique :

Posons $Mult_{ij}$ le nombre minimal de multiplications scalaires nécessaires pour calculer le produit $A_i \dots A_j$

\hookrightarrow le meilleur parenthésage de $A_i \dots A_j$ "coupe" la suite en k ($i \leq k < j$)

\hookrightarrow il est construit à partir d'un meilleur parenthésage de $A_i \dots A_k$ et de $A_{k+1} \dots A_j$

Récurrence : $Mult_{ii} = 0$

$$Mult_{ij} = \min_{i \leq k < j} \{Mult_{ik} + Mult_{(k+1)j} + d_{i-1} \cdot d_k \cdot d_j\} \quad \text{pour } i < j$$

Attention, ce n'est pas fini ...

Programmation dynamique

Algo Parenthèse-Rec(i, j)

entrée : $P[0..n]$ integer % dimension des matrices

sortie : min integer ; auxiliaire : m, min integer;

if $i = j$ **then** return(0) **else**

$min \leftarrow +\infty$;

for $k = 1$ **to** $j - 1$ **do**

$m \leftarrow$ Parenthèse-Rec(i, k) + Parenthèse-Rec($k+1, j$) + $P[i - 1].P[k].P[j]$

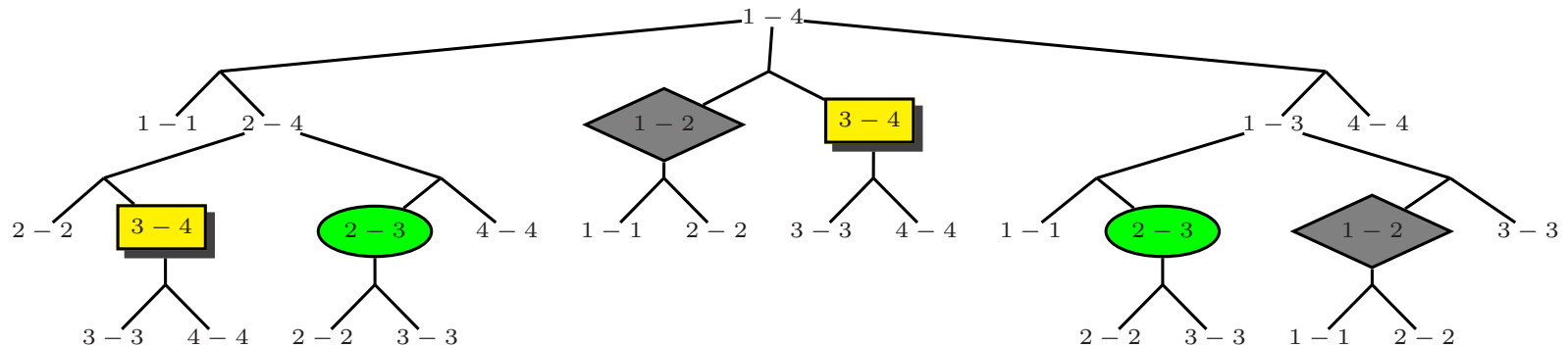
if $m \leq min$ **then** $min \leftarrow m$

endfor;

 return(min);

endelse

Algorithme récursif "direct"



Arbre des appels récursifs pour Parenthèse-Rec(1,4)

Programmation dynamique

Observation : Calcul répété des mêmes sous-instances

Pour notre exemple, Parenthèse-Rec est calculé 2 fois pour (3,4), (2,3) et (1,2)!

↪ éviter ces calculs redondants en essayant de stocker les résultats intermédiaires et en essayant de les ordonner correctement!

Vérification : Quel est le nombre de sous-instances pour n matrices ?

c'est le nombre de sous-chaîne de la chaîne $1..n$ ou encore

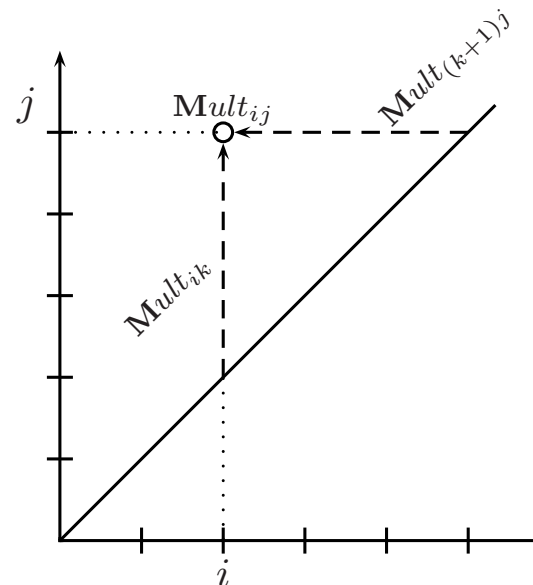
le nombre de choix de i et j dans $[1..n]$ soit $\binom{2}{n} = \Theta(n^2)$

Analyse : Que faut-il pour calculer $Mult_{ij}$?

tous les $Mult_{ik}$ et $Mult_{(k+1)j}$

Pour calculer $Mult_{2,5}$, il faut disposer de :

| k | $Mult_{ik}$ | $Mult_{(k+1)j}$ |
|---|--------------|-----------------|
| 2 | | $Mult_{3,5}$ |
| 3 | $Mult_{2,3}$ | $Mult_{4,5}$ |
| 4 | $Mult_{2,4}$ | |



Programmation dynamique

Algo Parenthèse-Dyn
entrée : n : integer; $P[0..n]$ of integer
sortie : $m[1..n][1..n]$ of integer
auxiliaires : $i, j, k, diag, m$: integer; ...

```
for  $i \leftarrow 1$  to  $n$  do  $m[i, i] = 0$ ;  
for  $diag \leftarrow 1$  to  $n - 1$  do  
  for  $i \leftarrow 1$  to  $n - diag$  do  
     $j \leftarrow i + diag$ ;  $m[i, j] \leftarrow \infty$ ;  
    for  $k \leftarrow i$  to  $j - 1$  do  
       $m \leftarrow m[i, k] + m[k + 1, j] + P[i - 1].P[k].P[j]$   
      if  $m \leq m[i, j]$  then  $m[i, j] \leftarrow m$   
    endfor;  
  endfor;  
endfor;
```

Programmation dynamique
% dimensions des matrices
% coûts meilleurs parenthésages

% initialisation $i = j$

% initialisation $i \neq j$

ligne L

Exercice : Modifier l'algorithme pour obtenir les places des parenthèses

Exercice : Ecrire un algorithme récursif équivalent

Programmation : Ecrire un programme qui visualise la chaîne des matrices avec les parenthèses

Multiplication d'une chaîne de matrices et parenthésage

Complexité :

qu'est ce qu'on compte ? (les opérations de) la ligne **L**

au pif ... en regardant l'algorithme : 3 boucles imbriquées : sur *diag* , sur *i* et sur *k*

↔ ca devrait donc être en $\Theta(n^3)$

Vérification :

$$\begin{aligned} L(n) &= \sum_{diag=1..n-1} \sum_{i=1..n-diag} \sum_{k=i..i+diag-1} 1 \\ &= \\ &= \\ &= \\ &= \end{aligned}$$

Distance d'édition

un autre exemple rapidement

La **reconnaissance / comparaison approximative de mots** est un problème fréquent

fautes de frappe fréquentes

la bonne orthographe est moins fréquente

mauvaise réponse **ou** faute dans la requête ?

pas de réponse **ou** réponse avec une autre orthographe ?

↪ **recherche sur le web**

La **notion de mots** est large :

traitement de la langue

bioinformatique - génétique

et autres langages + ou -formels (automates)

Des mots aux **arbres** :

documents XML

Distance d'édition

un autre exemple rapidement

Qu'est-ce que **deux mots proches** ?

↪ la distance entre M et N est le nombre **minimal** de transformations à effectuer pour obtenir M à partir de N .

Quelles transformations ?

Substitution : nation → ration abcdgt → abxdgt

Insertion : rat → rapt abxdgt → avbxdt

Suppression : rapt → rat avbxdt → avbxdt

et d'autres opérations plus sophistiquées :

comme **le déplacement** d'un sous-mot (sous-arbre) à une position.

↪ **calcul de la distance d'édition**

Idée de l'algorithme : (attention hypothèse = dernière étape)

Comment obtenir M à partir de $N = N'c_2$?

Le dernier caractère c_2 de $N'c_2$ doit subir l'une des transformations suivantes :

| transformation | cas |
|----------------|----------------|
| substitution | $M = N'c_1$ |
| insertion | $M = N'c_2c_1$ |
| suppression | $M = N'$ |
| -- | $M = N'c_2$ |

↔ idée à généraliser

Les fonctions utilisées dans l'algo :

1. coût de substitution de c_2 par c_1 :
Sub-fun(c_1, c_2) retourne 0 si $c_1 = c_2$ et sinon 1
2. coût de l'insertion/suppression de c :
InsDel-fun(c) retourne 1
3. Min-fun(i, j) évident !

Algorithme récursif "direct" de la distance d'édition entre 2 mots

```
Algo Dist-Rec
entrée :  $M, N$  : mots;  $i, j$  : integer                                % les 2 mots et un indice sur chaque
sortie :  $d_{ed}$  : integer                                             % distance d'édition entre  $M[1..i]$  et  $N[1..j]$ 
auxiliaire :  $Sub, Ins, Del$  : integer
if  $i = 0$  then  $d_{ed} \leftarrow j \cdot \text{InsDel-fun}(" ")$  else                                % fin du mot M
if  $j = 0$  then  $d_{ed} \leftarrow i \cdot \text{InsDel-fun}(" ")$  else                                % fin du mot N
     $Sub \leftarrow \text{Dist-Rec}(M, N, i - 1, j - 1) + \text{Sub-fun}(M[i], N[j]);$                 % Sub-fun fonction
     $Del \leftarrow \text{Dist-Rec}(M, N, i, j - 1) + \text{InsDel-fun}(N[j]);$                 % InsDel-fun fonction
     $Ins \leftarrow \text{Dist-Rec}(M, N, i - 1, j) + \text{InsDel-fun}(M[i]);$                 %
     $d_{ed} \leftarrow \text{Min-fun}(\text{Min-fun}(Ins, Del), Sub);$                 % Min-fun fonction
endelse;
return( $d_{ed}$ ).
```

Programmation dynamique

Exercice : Dessiner l'arbre des appels récursifs pour une ("petite") instance de votre choix.

Exercice : Quelle est la complexité de cet algorithme ?

Exercice : Que faut-il modifier dans l'algorithme pour prendre en compte des coûts différenciés pour chaque type de transformation et pour chaque caractère.

(On veut, par exemple, que le coût de la substitution de a par $!$ soit différent de celui de $\{$ par $[$ et on veut que le coût de l'insertion de a ne soit pas forcément celui de sa suppression.)

Observation : Calcul répété des mêmes sous-instances

↔ éviter ces calculs redondants

stocker $\text{Dist-Rec}(M, N, i, j)$ dans une matrice $D[1..n][1..m]$.

Analyse : Il n'y a que $n.m$ paires (i, j) .

Que faut-il pour calculer $\text{Dist-Rec}(M, N, i, j)$ i.e $D[i, j]$?

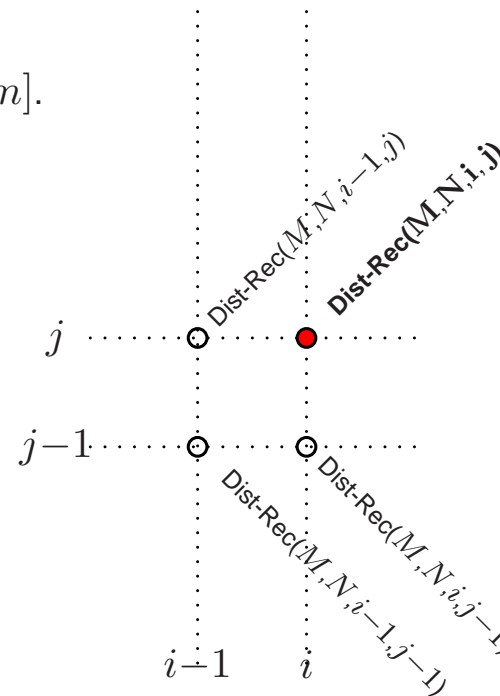
$\text{Dist-Rec}(M, N, i-1, j-1)$

$\text{Dist-Rec}(M, N, i, j-1)$

$\text{Dist-Rec}(M, N, i-1, j)$.

Ordre des calculs de la matrice $D[1..n][1..m]$

ligne par ligne !



Programmation dynamique

Exercice : Ecrire l'algorithme `Dist-Dyn` de programmation dynamique qui calcule la distance d'édition en se basant sur le principe décrit précédemment.

Complexité :

Qu'est-ce qu'on compte ? les sommes

Au pif sans même avoir écrit l'algorithme $O(n.m)$

Exercice : Ecrire un algorithme `Dist-Seq-Dyn` qui enrichit le précédent en produisant non seulement la distance d'édition entre 2 mots mais également la suite des transformations à opérer, appelé **séquence d'édition**.

Exercice de programmation: Programmer en traitant la variante "coûts différenciés" (vérifier que cela ne "bouscule" pas l'algorithme `Dist-Seq-Dyn`). Le programme doit permettre de choisir les coûts, il doit visualiser la transformation de N en M .

Quand peut-on utiliser la programmation dynamique ?

c'est efficace pour calculer des récurrences en stockant des résultats intermédiaires

↪ **darkred attention : pas trop de résultats intermédiaires !**

darkgreen Contre-exemples :

- Il y a $n!$ permutations d'une suite de n éléments
- Un ensemble de n éléments a 2^n sous-ensembles

↪ on ne peut pas stocker la "meilleure" solution pour chaque sous-permutation / sous-ensemble.

darkgreen Exemples :

- Il y a "seulement" $\frac{n(n-1)}{2}$ sous-mots d'un mot de longueur n
- Il y a "seulement" $\frac{n(n-1)}{2}$ sous-arbres possibles d'un arbre binaire de recherche

↪ à exploiter pour tout problème sur les mots, séquences arbres de recherche.

La programmation dynamique est une stratégie qui donne de bons résultats en général, lorsque les objets considérés sont ordonnés: les caractères d'un mot, les matrices d'une chaîne produit, les points à la frontière d'un polygone, les feuilles d'un arbre binaire de recherche.

**Steven Skiena : "Whenever you objects are ordered in a left-to-right way
you should smell dynamic programming"**

Problèmes d'optimisation

quand la programmation dynamique est trop "lourde"

rappel : les objets ne sont pas triés, trop de sous-solutions à stocker

Principe : meilleur choix localement

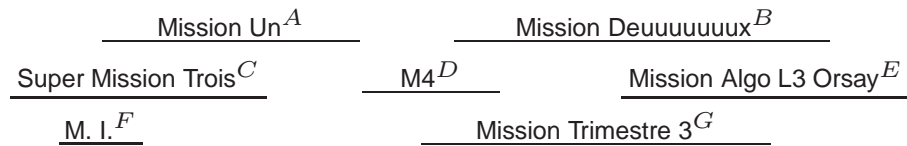
1. Etant donné une instance **I** quelconque de taille n , ordonner les éléments de **I**.
Cet ordonnancement est une sorte d'heuristique !
2. Résoudre le problème **P** avec un algorithme **A** qui à chaque étape choisit la meilleure solution relativement à l'heuristique précédente.

Difficultés majeures : Choisir une heuristique

Montrer la correction

Algorithmes Gloutons

Problème 2-a : Un intérimaire souhaite organiser son emploi du temps de sorte à effectuer le plus grand nombre possible de missions, sans que deux missions se recouvrent dans le temps.



Une instance du problème

Énoncé formalisé du problème 2-a :

Entrée = un ensemble I de n intervalles

Sortie = un plus grand[†] sous ensemble d'intervalles de I deux à deux disjoints ?

[†] plus grand au sens de la cardinalité des ensembles.

- la stratégie "choisir la première mission qui se présente" est incorrecte
Pour l'instance ci-dessus c'est correct: C, D, E
- la stratégie "choisir la mission la plus courte" est incorrecte
Pour l'instance ci-dessus c'est correct : F, D, E
- "choisir la mission dont la fin est le plus tôt" est correcte
Pour l'instance ci-dessus : F, D, E

Algorithmes Gloutons

Algorithme :

Considérons la suite I_1, \dots, I_n des n intervalles de I ordonnée par ordre croissant des bornes supérieures

$$i < j \Rightarrow f(I_i) < f(I_j) \quad \text{où } f(I_i) \text{ est la borne sup de } I_i$$

Algo Interim-Glou

entrée : $I_1 \dots I_n$ % suite ordonnée des intervalles

sortie : $J_1 \dots J_m$ % missions optimales ordonnées

auxiliaire : i, j integer

$J_1 \leftarrow I_1; \quad j \leftarrow 1$

for $i \leftarrow 2$ **to** n **do**

if $d(I_i) \geq f(J_j)$ **then** $j \leftarrow 1; J_j \leftarrow I_i;$ % ligne C

Complexité : Que compte-t-on ? comparaisons ligne C

$C(n)$ est dans $\Theta(n)$ (linéaire!)

Preuve de la correction de la stratégie "fin proche":

Supposons que $J_1 \dots J_m$ est la "solution" produite par Interim-Glou pour l'instance $I_1 \dots I_n$.

Supposons que cette solution soit contestable c'est à dire qu'il existe $K_1 \dots K_p$ satisfaisant les contraintes et telle que $m < p$.

On montre que " J " est inclus dans " K " et donc que " J " est " K ".

Algorithmes Gloutons

Coloriage d'un graphe :

On veut colorier les sommets d'un graphe $G=(S,A)$ sans que 2 sommets reliés par une arête ne soient de la même couleur.

La coloration col est une fonction de S dans $[1..k]$ telle que $s, r \in S \Rightarrow col(s) \neq col(r)$.

Objectif : minimiser le nombre de couleur utilisées !

C'est un problème très difficile : le problème général est NP-complet

1. algorithme glouton pour les graphes d'intervalles
2. algorithme glouton pour les graphes bipartis