Chapître V.
Programmation dynamique

1. Méthodes de conception des algorithmes (efficaces)

-) Diviser pour régner

-) <u>Programmation dynamique</u>

-) Algorithme glouton

2. Problème d’optimisation

La programmation dynamique est en général appliquée aux <u>problèmes d’optimisation</u>.

-) Dans un problème d’optimisation il y a de nombreuses solutions possibles.

-) Chaque solution possible est affectée d’une valeur.

-) On souhaite trouver une solution ayant la valeur optimale (minimum ou maximum).

Exemple:

MAXIMUM INDEPENDENT SET

-) donnée: graphe non-orienté $G = (V, E)$

-) solution: $I \subseteq V$ est une solution si I est un stable de G, c'est-à-dire quelque soit deux sommets $x \in I$ et $y \in I$ alors x et y ne sont pas adjacents.

-) valeur: $val(I) = |I|$

3. Plan général de la programmation dynamique

La conception d'un algorithme de programmation dynamique peut être planifiée dans une séquence de quatre étapes.

1. Caractériser la structure d'une solution optimale.
2. Définir récursivement la valeur d'une solution optimale.
3. Calculer la valeur d'une solution optimale en remontant progressivement jusqu'à l'énoncé du problème initial.
4. Construire une solution optimale pour les informations calculées.

4. Multiplication d'une suite de matrices

Donnée: Séquence $A_1, A_2, \ldots, A_n$ de n matrices

Problème: Calculer le produit $A_1 A_2 \cdots A_n$ de façon à minimiser le nombre de multiplications scalaires.

On suppose que pour $i = 1, 2, \ldots, n$ la matrice A_i a une dimension $p_{i-1} \times p_i$.

Observations fondamentales

1. On ne peut multiplier deux matrices A et B que si le nombre de colonnes de A est égal au nombre de lignes de B.

2. Si A est une matrice $p \times q$ est B est une matrice $q \times r$, alors la matrice résultante $C = AB$ est une matrice $p \times r$.

3. Le temps de calcul C est dominé par le nombre de multiplications scalaires qui est $p \cdot q \cdot r$.

4. Dans ce qui suit, nous exprimerons les temps d'exécution en fonction du nombre de multplications scalaires.

Parenthésages

La multiplication des matrices est associative donc tous les parenthésages (complète) aboutissent à une même valeur du produit $A_1 A_2 \cdots A_n$.

Exemple: Le produit $A_1 A_2 A_3 A_4$ peut être complètement parenthésé de cinq façons distinctes:

$$
\begin{gathered}
(A_1(A_2(A_3A_4))) \\
(A_1((A_2A_3)A_4)) \\
((A_1A_2)(A_3A_4)) \\
((A_1(A_2A_3))A_4) \\
(((A_1A_2)A_3)A_4)
\end{gathered}
$$

Le nombre de multiplications scalaires dépend essentiellement du parenthésage.

Exemple:

On considère le produit $A_1A_2A_3$, où les dimensions des matrices sont respectivement
10×100, 100×5, et 5×50.

a) $((A_1A_2)A_3)$

nombre de multiplications scalaires
$m_a = 10 \cdot 100 \cdot 5 + 10 \cdot 5 \cdot 50 = 5000 + 2500 = 7500$

b) $(A_1(A_2A_3))$

nombre de multiplications scalaires
$m_b = 100 \cdot 5 \cdot 50 + 10 \cdot 100 \cdot 50 = 25000 + 50000 = 75000$

$\Rightarrow$ Le calcul du produit en suivant le premier parenthésage est donc 10 fois plus rapide.

Conséquences

Il est profitable de minimiser le nombre de multiplications scalaires.

Considérons une suite $A_1, A_2, \ldots A_n$ où la matrice A_i a une dimension $p_{i-1} \times p_i$ pour $i = 1, 2, \ldots, n$. Nous constatons que le nombre de multiplications scalaires dépend du parenthésage et de la suite $p_0, p_1, p_2, \ldots, p_n$.

Recherche exhaustive $\rightarrow$ temps exponentiel

Malheureusement on ne peut pas effectuer une recherche exhaustive pour déterminer le parenthésage optimal.

Théorème. *Le nombre de parenthésage d'un produit de n matrices est le nombre de Catalan:*

$$C(n) = \frac{1}{n+1} C_{2n}^{n} = \frac{1}{n+1}\binom{2n}{n} = \Omega(\frac{4^n}{n^{3/2}}).$$

Heureusement, même si le nombre de solutions possibles est exponentiel, il y a peut-être un algorithme polynomial et efficace.

Structure d'un parenthésage optimal

La première étape du paradigme de la programmation dynamique consiste à caractériser la structure d'une solution optimale.

Considérons un parenthésage optimal quelconque du produit $A_1 A_2 \cdots A_n$. Il sépare le produit entre A_k et A_{k+1} pour un certain entier k dans l'intervalle $1 \leq k \leq n$. C'est-à-dire que le parenthésage optimal a la forme

$$((A_1 A_2 \cdots A_k)(A_{k+1} A_{k+2} \cdots A_n)).$$

Observation: Le coût de ce parenthésage optimal est donc le coût du calcul de la matrice $A_1 A_2 \cdots A_k$, plus celui du calcul de la matrice $A_{k+1} A_{k+2} \cdots A_n$, plus celui de leur multiplication
(pour une certain valeur $k \in \{1, 2, \ldots, n\}$).

$\Rightarrow$ Le parenthésage de $A_1A_2\cdots A_k$ à l'intérieur de notre parenthésage optimal doit être un parenthésage optimal de $A_1A_2\cdots A_k$.

(Sinon on peut le remplacer par un parenthésage plus économique et cela produirait un autre parenthésage de $A_1A_2\cdots A_n$ dont le coût serait inférieur au coût optimum: une contradiction.)

$\Rightarrow$ Le parenthésage de $A_{k+1}A_{k+2}\cdots A_n$ à l'intérieur du parenthésage optimal doit être un parenthésage optimal de $A_{k+1}A_{k+2}\cdots A_n$.

(Sinon on peut le remplacer par un parenthésage plus économique et on aboutirait aussi une contradiction.)

Sous-structure optimale

> **Une solution optimale du problème** (de multiplication d'une suite de matrices) **comporte des solutions optimales aux instances de sous-problème.**

Chaque fois qu'un problème fait apparaître une sous-structure optimale, c'est un bon indice pour utilisation de la programmation dynamique.

On appèle cette propriété
<u>le principe d'optimalité</u>
<u>de la programmation dynamique.</u>

5. La conception d'un algorithme de programmation dynamique

Multiplication d'une suite de matrices

-) donnée: Séquence $A_1, A_2, \ldots, A_n$, où pour $i = 1, 2, \ldots n$ la matrice A_i a une dimension $p_{i-1} \times p_i$

-) solution: Parenthésage du produit $A_1 A_2 \cdots A_n$

-) valeur: Nombre de multiplications scalaires si le calcul du produit est exécuté en suivant ce parenthésage

On a déjà observé quel type de sous-problème est important.

Solution récursive

Convention: Nous appèlons $A_{i..j}$ le sous-problème de minimiser le nombre de multiplications scalaires pour le calcul du produit $A_i A_{i+1} \cdots A_j$ où $1 \leq i \leq j \leq n$.

Soit $m[i, j]$ la valeur d'une solution optimale du sous-problème $A_{i..j}$, où $1 \leq i \leq j \leq n$. C'est-à-dire que $m[i, j]$ est le nombre minimum de multiplications scalaires nécessaires pour le calcul du produit $A_i A_{i+1} \cdots A_j$.

La manière la plus économique de calculer $A_1 A_2 \cdots A_n$ sera donc $m[1, n]$. C'est-à-dire, $m[1, n]$ est la valeur d'une solution optimale du problème.

Récurrence

On obtient la récurrence suivante qu'on utilise comme définition récursive du coût minimum pour le parenthésage du produit $A_i A_{i+1} \cdots A_j$:

$$m[i,j] = \begin{cases} 0 & \text{si } i = j, \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & \text{si } i < j. \end{cases}$$

Explication:

1) Si $i = j$, la suite est consituée d'une seule matrice A_i, et aucune multiplication n'est nécessaire pour calculer le produit. Donc, $m[i,i] = 0$ pour $i = 1, 2, \ldots, n$.

2) Si $i < j$, on tire avantage de la structure d'une solution optimale (définie à l'étape 1). Supposons qu'un parenthésage optimal sépare le produit $A_iA_{i+1}\cdots A_j$ entre A_k et A_{k+1}. Donc $i \leq k < j$.

Alors, $m[i,j]$ est égal au coût minimum pour le sous-problèmes $A_{i..k}$ et $A_{k+1..j}$, plus le coût de la multiplication de matrice résultant du produit $A_iA_{i+1}\cdots A_k$ (qui a une dimension $p_{i-1} \times p_k$) par la matrice resultant du produit $A_kA_{k+1}\cdots A_j$ (qui a une dimension $p_k \times p_j$).

On obtient $m[i,j] = m[i,k]+m[k+1,j]+p_{i-1}p_kp_j$, en supposant qu'on connaisse la valeur k, ce qui n'est pas le cas.

Heureusement il suffit de vérifier toutes les valeurs possibles pour k, c'est-à-dire $k = i, i+1, \ldots, j$, pour trouver $m[i,j]$.

Calcul des coûts optimaux

Pratiquement nous avons résoulu notre problème. Il n'y a plus qu'à écrire un algorithme récursif basé sur la récurrence pour calculer le coût minimum $m[1, n]$.

Attention: La méthode directe pour obtenir un algorithme recursif fournit un algorithme en temps exponentiel. Donc cette méthode n'est pas meilleure que la recherche exhaustive.

Bottom-up

Il faut remplacer l'algorithme récursif, c'est-à-dire une approche descendante (**top-down**), par une approche ascendante (**bottom-up**).

Remarque. Comme la sous-structure optimale, l'approache ascendante est une propriété caractéristique de la programmation dynamique.

Nombre de sous-problèmes

Observation: Le nombre de sous-problèmes à résoudre est $C_n^2 + n = \Theta(n^2)$ parce qu'il y a un sous-problème pour chaque choix de i et j satisfaisant $1 \leq i \leq j \leq n$.

Attention: Un algorithme récursif peut rencontrer chaque sous-problème de nombreuses fois dans différentes branches de l'arbre récursif.

Maintenant il faut écrire un algorithme qui rencontre chaque sous-problème une seule fois (et utilise la récurrence).

Troisième étape: Conception d'algorithme

Entrée: Séquence $p_0, p_1, \dots, p_n$.

La procédure utilise un tableau auxiliaire $m[1..n, 1..n]$ pour conserver les coûts $m[i,j]$ et un tableau auxiliaire $s[1..n, 1..n]$ qui enregistre l'indice de k quand le coût optimal du calcul $m[i,j]$ est atteint.

$ORDONNER\text{–}CHAINE\text{–}DE\text{–}MATRICES(p)$

pour $i \leftarrow 1$ **à** n
  faire $m[i,i] \leftarrow 0$
pour $l \leftarrow 2$ **à** n
  faire pour $i \leftarrow 1$ **à** $n - l + 1$
      faire $j \leftarrow i + l - 1$
        $m[i,j] \leftarrow \infty$
        pour $k \leftarrow i$ **à** $j - 1$
          faire $q \leftarrow m[i,k] + m[k+1,j] + p_{i-1}p_kp_j$
            si $q < m[i,j]$
              alors $m[i,j] \leftarrow q$
                  $s[i,j] \leftarrow k$
retourner m et s

Exemple:

Entrée: $30, 35, 15, 5, 10, 20, 25$

Analyse de temps

Un simple coup d'œil à la **pour**-boucles imbriquées de $ORDONNER-CHAINE-DE-MATRICES(p)$ donne un temps d'exécution en $O(n^3)$ pour l'algorithme. Les boucles sont imbriquées sur trois niveaux, et chaque indice de boucle (l, i et k) traite au plus n valeurs.

Théorème. *L'algorithme $ORDONNER-CHAINE-DE-MATRICES(p)$ determine le nombre minimum de multiplications scalaires pour calcul le produit $A_1A_2\cdots A_n$ où pour $i = 1, 2, \ldots n$ la matrice A_i a une dimension $p_{i-1} \times p_i$. L'algorithme s'exécute un temps en $O(n^3)$.*

Remarque. L'algorithme nécessite un espace de stockage en $\Theta(n^2)$ pour les tableaux m et s.